Delft University of Technology
Master of Science Thesis in Embedded Systems

# Debugging Intermittently-Powered Embedded Systems Like Any Other Embedded System

**Tom Sebastiaan Hoefnagel**

Embedded Networked Systems

TU Delft Delft University of Technology

# Debugging Intermittently-Powered Embedded Systems Like Any Other Embedded System

Master of Science Thesis in Embedded Systems

Embedded Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Tom Sebastiaan Hoefnagel

31st October 2022

**Author**
Tom Sebastiaan Hoefnagel
**Title**
Debugging Intermittently-Powered Embedded Systems Like Any Other Embedded System
**MSc Presentation Date**

31 October 2022

**Graduation Committee**

| | |
|---|---|
| Przemysław Pawełczak | Delft University of Technology |
| Annibale Panichella | Delft University of Technology |
| Jasper de Winkel | Delft University of Technology |

The work presented in this thesis has led to a paper that has been submitted and accepted to the ACM SenSys 2022 conference for publication (`https://sensys.acm.org/2022/`)

## 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. Users evaluate our debugger and we have found unseen errors in a state-of-the-art software framework for intermittently powered systems.

# Preface

This work you face is part of research imposed by the Embedded and Networked Systems group at the Delft University of Technology. It is supervised by Jasper de Winkel and Przemysław Pawełczak. Batteries have had an environmental impact since the beginning, and demand for smaller embedded systems is only growing. With this work in battery-free systems, I hope to make battery-free embedded programming more alluring and fun. Creating an easier way to produce intermittently-powered systems should lead to more embedded engineers choosing battery-free solutions instead of continuously-powered options like nowadays.

Now, when reaching the end of my masters, I look back gratefully. First and foremost, I am extremely grateful to my supervisors Jasper de Winkel and Przemysław Pawełczak, for their innumerous support and priceless advice during my study. During a difficult pandemic period in 2021, it was hard to find an interesting thesis subject. During on of the lectures given by Przemysław Pawełczak, I got introduced to intermittently-powered devices, which eventually led to this thesis. During my research period, Jasper de Winkel and I had countless brainstorm sessions, I would thank him for these sessions, which eventually lead to a better end result. I also would like to thank Annibale Panichella for reviewing my work as the graduation committee with my daily supervisors.

Tom Hoefnagel

Delft, The Netherlands
31st October 2022

# Contents

# Chapter 1

# Introduction

There are currently over 12.2 billion [33] connected Internet Of Things (IoT) devices, most of which perform just a single, simple task, providing new solutions to societal-scale problems such as home automation, security, and wearable computing. A significant challenge lies in powering all these IoT devices. While batteries are still the common energy source, they are not sustainable and come with environmental and social-cultural impacts [43]. Energy harvesting technology is a more promising way of powering these IoT devices. By applying energy harvesting, the device is directly powered by the ambient source without using chemical batteries [19]. Nevertheless, removing the primary energy source is challenging as standard embedded systems are not built for intermittent operations. This is where a new category of low-powered embedded systems arises, Intermittently-powered devices.

Intermittently-powered devices 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 small energy storage, i.e. (super-)capacitor, that cannot buffer incoming energy as much as the battery,

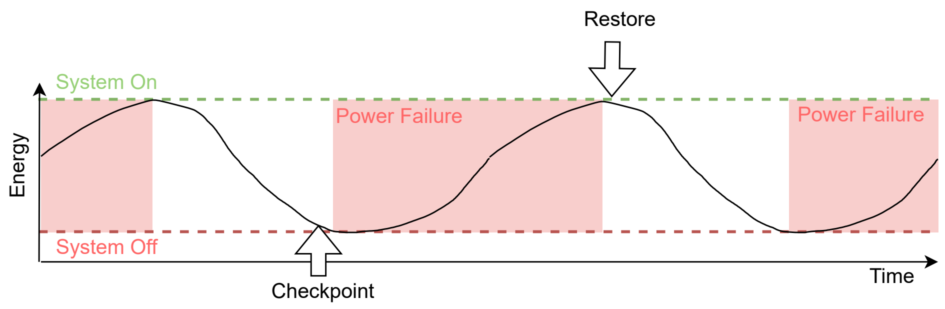


Figure 1.1: **Battery-free embedded systems operate intermittently; power failures frequently happen for an indefinite duration. The figure includes checkpointing capability, checkpointing crucial memory to non-volatile memory before a power failure and restoring after a recovery.**

elevating the intermittency rate further. Battery-free operation promises perpetual operation: as long as there is an ambient energy source, battery-free devices will continue operating [39]. Examples of these intermittently-powered embedded applications include a battery-free handheld gaming console [14], battery-free computational Radio Frequency Identification (RFID) tags [42], battery-free sensors [11] and sensor networks [1], battery-free eye tracker [27], as well as battery-free edge computing platform [31].

Unfortunately, despite the increasing number of battery-free intermittently-powered platforms, they are still hard to develop [24, Section 7], the main reason ensuring the continuation of the executed program after a power interruption. The developer of the intermittent program must programmatically account for twofold events. That is, whenever a power interrupt happens, at any moment in time, the device (i) must resume operation from the moment that a power interrupt happens, and (ii) the state of the device's memory and its peripherals must be correctly restored. To assure both events, the developer can instrument the code utilising two approaches. The first one is the inclusion of checkpoints and restore, as visualised in Figure 1.1, which forces saving the program's state from volatile to non-volatile memory (examples of such approaches include [53, 25]). The second one is code transformations, where code is divided into atomic blocks (in the context of intermittently-powered devices called tasks [28] or threads [55]) whose execution time matches the specific energy budget of the intermittently-powered device. In either method, the intermittently-powered device's code needs to be debugged and tested during the development phase.

## 1.1 Problem Statement

Sadly, debugging intermittently-powered software is far from easy [7, Section 2.2]. This is because, above the existence of regular bugs (not related to intermittently-powered operation), the developer also has to cope with bugs related to these power failures. Debuggers designed for continuously-powered embedded systems, such as [44] and [51], assume the target is continuously available and relinquish the connection on every power failure, so the developer has to reestablish the connection between the chip and the debugger manually, every time, leading to a time-consuming or even unworkable debug process. The notion to solve this debugging problem for intermittently-powered embedded systems could be defined as follows:

> *How could we modify intermittently-powered embedded systems' debugging process so developers can debug these systems like any continuously-powered embedded system?*

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. (i) A complete hardware debugger based on the GNU Debugger (GDB) Project [38, 48]—to make sure the steps of debugging are the same as already existing (non-intermittently powered) embedded debugging platforms on the market, and (ii) an energy emulator capable of replaying energy traces, powering the battery-free embedded systems following either a pre-recorded or synthetically generated energy profile—for emulating specific intermittency patterns such as in [16].

## 1.2 Contributions

This work is mainly focused on software, although the complete project presented in the paper [12], submitted and accepted to ACM SenSys 2022, consists of both hardware and software, the hardware is primarily developed by Jasper de Winkel, and will only be briefly described in this thesis report. The software component can be divided into multiple sections:

1. *Embedded Debugger Firmware.* We present custom debugging firmware for the DIPS prototype PCB board. This software enables regular debugging for intermittently-powered embedded ARM systems based on the GDB principles. The firmware allows debugging without interruption due to intermittent power failures, supporting battery-less debugging operations.

2. *Embedded Energy Emulator Firmware.* We created a custom energy emulator firmware for the second component of the DIPS prototype PCB board, allowing either (i) replaying standardised pre-recorded energy profiles and (ii) playing generated synthetic energy profiles. The embedded energy emulator has a tight interconnection with the embedded debugger to allow real-time communication between the emulator and the hardware debugger.

3. *Emulator Graphical User Interface (GUI).* For easy parameter adjustment of the energy emulator and visualising real-time energy traces, we create a GUI. The interface (shown in Figure 5.4) allows embedded developers to upload pre-recorded energy traces to the emulator or modify synthetic wave generation and-or buck-boost simulation parameters.

4. *GDB Debugger Command-line interface (CLI) wrapper.* With the addition of intermittently-powered devices' automated software testing and specific intermittently-powered debugging features, the GDB client software had to be modified. This allows tight integration with existing Integrated Development Environment (IDE) and using standardised GDB functionalities in an intermittently-powered scope without completely recreating the debugger experience.

Other non-software-based contributions mainly related to the evaluation are the user testing experience and case studies.

# Chapter 2

# Related Work

Since the beginning of research in intermittently-powered embedded systems, numerous institutions have searched for new future-proof intermittently-powered frameworks and development platforms. DIPS can be placed within the following context related:

## 2.1 Intermittently-Powered Debugging

To the best of our knowledge, there is only one dedicated intermittent-aware debugger, i.e. Energy-interference-free Debugger (EDB) [7] which addresses some of the core limitations of existing continuously-powered debuggers used for intermittently-powered embedded systems. Nevertheless, debugging with EDB requires manual code adjustment, using the EDB-specific API. One has to recompile and flash the complete code base for each new assertion or breakpoint, resulting in a time-consuming debug process. A more detailed table of limitations of EDB is provided in Table 2.1. The state-of-the-art Segger J-Link Debugger [44] is the other system used, mainly for continuously-powered embedded systems. This debug probe has no support for intermittently-powered devices, so will break the connection after every power interrupt, but does have rich debugging features like GDB and IDE support and hardware assist debugging.

## 2.2 Intermittently-Powered Energy Emulation

Some Energy Emulation solutions have been proposed before this work. Starting with [16], who promised a complete converter-based portable testbed for intermittently-powered devices capable of recording and replaying harvesting current and voltage traces. DIPS is directly capable of replaying these Shepherd traces, including running these traces in a buck-boost converter as described by the original paper [12]. The buck-boost converter calculation and implementation originated from [5], which is also related work. The last bit is [18], another emulator capable of recording and replaying energy harvesting conditions, mimicking the physical characteristics of surrounding energy harvesting environments.

Table 2.1: **A feature comparison of debuggers used for intermittently-powered embedded systems. The systems used are the EDB [7], J-Link [44], and our work DIPS. Where both EDB and J-Link are state-of-the-art debugging probes**

| Feature | **EDB** | **J-Link** | **DIPS** |
|---|---|---|---|
| Intermittent support | Yes ✓ | No ✗ | Yes ✓ |
| Energy breakpoints | Yes ✓ | No ✗ | Yes ✓ |
| Software breakpoints | Yes ✓ | Yes ✓ | Yes ✓ |
| Hardware breakpoints | No ✗ | Yes ✓ | Yes ✓ |
| Single step | No ✗ | Yes ✓ | Yes ✓ |
| GDB support | No ✗ | Yes ✓ | Yes ✓ |
| IDE support | No ✗ | Yes ✓ | Yes ✓ |

## 2.3 Testing Frameworks For Intermittently-Powered Devices

The second significant work in this paper lies within the testing frameworks for intermittently-powered devices. In [29], the authors introduce the Write After Read (WAR) hazards and a framework for detecting these using code analysis. Another example of an intermittently-powered code analysis tool based on a statistical model is CleanCut [8]. CleanCut checks and report non-terminating tasks in existing code. The last testing framework related is Siren [15], Siren introduces NVM and energy simulation capabilities using the MSPSim [9] simulator.

# Chapter 3

# Motivation

Developing of Intermittently-Powered IoT devices is still a complicated task that copes with many specific challenges and adds new kinds of bugs to embedded programming. Even though battery-free devices create possibilities for a more sustainable environment in the future, further research is needed before battery-free devices can be the standard, starting with easing up the development and testing process.

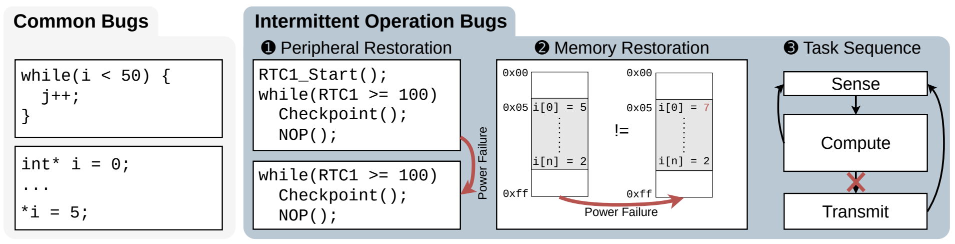
## 3.1 Debugging Intermittently-Powered Embedded Systems

Debugging of embedded systems code is different from PC-based code debugging [4, Chapter 8], as PC-based programs execute on the same device. Unlike embedded systems where code is executed on an external embedded system. This makes the developer rely on external hardware–such as [44]–acting as an interface between the development computer and the Microcontroller (MCU) on-board debug hardware.

We can extend this requirement even further when looking at debugging an intermittently-powered embedded system. As intermittently-powered devices have frequent power outages, the interface between the development PC and MCU on-board debug hardware should cope with this unknown duration of unconnectivity.

## 3.2 Intermittent Bugs Classification

At first, it is important to understand the type of bugs conceived by programming intermittently-powered embedded systems. We can categorise bugs into two classes presented in Figure 3.1. The first class is the common programming language and embedded system-related bugs. Examples of such bugs are algorithm implementation bugs like wrong implementations or typos and code errors in embedded system-related functionalities such as inaccurate peripheral initialisation. These bugs have been extensively analysed since the dawn of programming languages and will not be explained here. The second, more related class of bugs are intermittent operation-related bugs. These bugs are the



```
while(i < 50) {
  j++;
}
```

```
int* i = 0;
...
*i = 5;
```

```
RTC1_Start();
while(RTC1 >= 100)
  Checkpoint();
  NOP();
```

```
while(RTC1 >= 100)
  Checkpoint();
  NOP();
```

Figure 3.1: **The major classes of bugs which are present in intermittently-powered systems. First common programming and embedded bugs and second the specifically intermittent bugs introduced in this work, categorised as peripheral restoration, memory restoration and task sequence bugs.**

ones which the DIPS specifically targets. We can even further categorise these intermittent operation bugs into (i) peripheral restoration bugs, (ii) memory restoration bugs, and (iii) task sequence bugs.

(i) **Peripheral restoration bugs** occur due to inaccurate re-initialisation of the peripheral configuration after checkpoint restoration. Figure 3.1 defines an example of these kinds of bugs. In the example, one can see that after the power failure, the program restarts directly from within the while loop without re-initializing the correct peripheral, causing the device to reach a state where undefined behaviour such as an infinite loop occurs from the not re-initiated Real-Time Clock (RTC). Peripheral restoration bugs also occur when the state of any external peripheral such as displays, sensors and radios is not carefully considered within the application's context, especially when these peripherals have a persistent state and were originally designed for continuously-powered applications. 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 issue where the external peripherals state is not aligned with the expected state.

(ii) **Memory restoration bugs** is a category of bugs related to the checkpoint and restore point architecture. In the first place, they can relate to the wrong placement of check and/or restore points (functions *Checkpoint*() and *Restore*() as illustrated in Listing 3.2 (a). In this example, one can find a classic write-after-read error, which could occur due to the lack of a checkpoint in-between reading from and writing to non-volatile memory. Memory restoration bugs can also occur whenever the checkpoint or restore is not implemented correctly. For example, checkpointing is often based on double buffering (as [24], where the whole memory is copied to the non-volatile memory on a predefined clock interval), here a binary flag specifies from to which memory region checkpoint needs to be stored and restored. If the checkpoint happens at the moment of a flag update, the checkpoint could be corrupted and should

Listing 3.2: **Masked Write After Read (WAR) error due to breakpoint insertion (listing (b)). When function calls are instrumented as checkpoints such as in [25], the addition of breakpoints, required by 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-related error**

```
Checkpoint()
y = nv_x // wrong
// after restart
z = y + 1

nv_x=z
# Power failure
...
Checkpoint()
```

(b) **Masked WAR-related error**

```
Checkpoint()
y = nv_x // correct
// after restart
z = y + 1
breakpoint(0)
nv_x = z
# Power failure
...
Checkpoint()
```

be repaired. The probability of an error in the checkpoint implementation could be even higher with more complicated checkpoint routines, like differential checkpointing of [14] or undo logging-based checkpointing in [26].

(iii) **Task sequence bugs** are the last special types of bugs, these bugs are specific to run-time systems for intermittently-powered devices where input code is transformed into special tasks (such as 'Calculate', 'Read' and 'Commit') and checkpointing is performed on every task transition. Examples of such implementation include InK [54] and Alpaca [28]. Bugs can occur not only by incorrect implementation of the task state machine but also when defining the volatile memory associated with each task–if not accurately defined, it could result in writing and reading from unrestored memory.

# Chapter 4

# Architecture

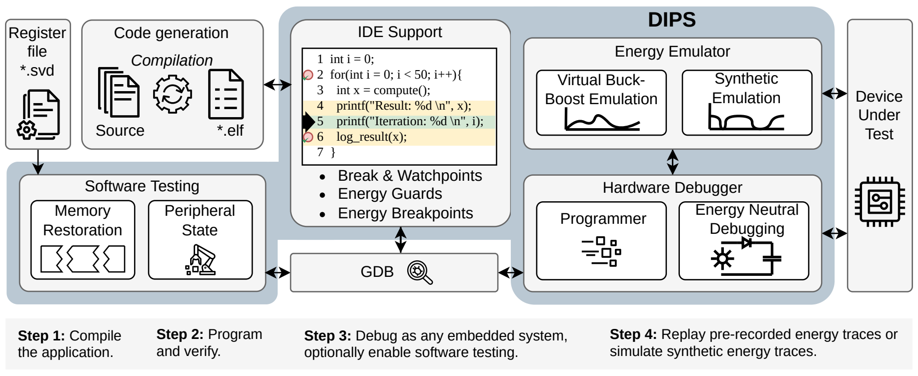
Driven by requirements listed in Table 2.1 we propose a new method of developing and testing battery-free intermittently-powered systems. These development and testing methods are designed following the principles discussed in this chapter. This design combines a hardware debugger and an energy emulator, enabling seamless debugging and automated testing of intermittently-powered embedded systems. The entire design and workflow of DIPS are visualised in Figure 4.1.

## 4.1 Hardware Intermittence-free Debugger

A core part of debugging any system is the hardware debugger. It interfaces with the Device Under Test (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.

### 4.1.1 Intermittently-Power Supporting Debugger

The first requirement listed in Table 2.1, is intermittent support. To achieve intermittent support for DIPS we have to find a way for the debugger interface to either (i) stay connected to the debugger module on the MCU or (ii) reconnect directly after a power failure and restore the current debug parameters, i.e. break- and watch-points, without the end user noticing this at the performance of the debugger. While the first solution sounds the most practical, keeping the connection intact, without the hassle of restoring the connection and last debug information. It is so far not possible to keep the debug module powered without powering the other MCU core functionalities, by looking at the common ARM System on a Chip (SoC) system designs [47, 50]. Adding this support must include changes to the SoC architectures, allowing powering the debugger module separate from other MCU components. As DIPS should form a debug interface for existing intermittently-powered battery-free systems, changing common architecture is not an applicable option. The second approach sounds more attainable, without eliminating already existing intermittently-powered systems and changing common chip architectures. With this solution, any state not specifically stored before a power failure is lost and not recoverable. This



```
int i = 0;
for(int i = 0; i < 50; i++){
  int x = compute();
  printf("Result: %d \n", x);
  printf("Iterration: %d \n", i);
  log_result(x);
}
```

Figure 4.1: **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.**

includes the configuration of the MCU's debugging registers. Even worse, these debugging registers are usually not configurable within the MCU due to the associated security risks. Hence the hardware debugger must be able to quickly reconnect and keep track of all debugging attributes, such as break- and watchpoints.

### 4.1.2 Intermittently Energy-Neutral Debugging

One of the core features of DIPS is the energy-isolated interface between DIPS and the DUT. This allows monitoring the DUT whilst not interfering with the power consumption of the DUT. If the DUT is halted by any of the debugging actions e.g., a breakpoint, the hardware debugger automatically pauses the energy emulator, making sure the DUT remains powered whilst not affecting the energy levels, so when execution resumes, the energy states stay as prior.

We introduce two debugging modes of the DIPS, (i) attached and (ii) detached debugging, each of them is described below; the hardware implementation is further described in chapter 5.

**Attached Debugging** In the attached debugging mode, the debugger is constantly connected to the DUT. After every power failure, the debugger automatically reconnects and restores debugging attributes. The debugger behaves like a normal embedded system to the developer, masking intermittency effects.

**Detached Debugging** Detached debugging is intended for operation when using an external energy source, e.g. harvesting source. In this mode, the debugger only initiates a connection when it receives a hardware interrupt from the DUT. This mode allows minimum interaction between DIPS and DUT. Whenever a hardware interrupt is generated by the DUT, the debugging emulator takes over the power, mitigating the energy required by the debugging.

After the execution of the DUT is resumed, the debugger detaches and the emulator will finish mitigating, restoring the energy levels to prior.

### 4.1.3 Energy-Aware Debugging Features

Besides regular embedded debugging functions described earlier, mostly utilized by the built-in MCU's debugging hardware, DIPS also introduces some custom features related to energy-aware debugging. At first, DIPS does not require the usage of a specific API for debugging. We extend GDB with extra CLI commands to implement two key intermittent specific debugging functions: (i) energy breakpoints and (ii) energy guards.

(i) Energy breakpoints extend traditional breakpoints by only triggering when the DUT's capacitor value is lower than the provided threshold. This allows debugging in specific situations with low energy levels

(ii) Energy guards or energy-neutral sections allow users to execute debug code whilst emulation is paused and continuous power is provided. This allows for eliminating certain code from testing or compensating energy levels for debug functions

Apart from these extensions, DIPS is also capable of programming/flashing of supported MCUs. This completes the all-in-one suite of features served by DIPS for the developer of intermittently-powered embedded systems.

## 4.2 DIPS Energy Emulator

The second core component of the DIPS is the energy trace emulator. State-of-the-art intermittently-powered systems harvest energy and store this into a (super-)capacitor. The voltage of this capacitor is often used as a threshold to determine when the voltage regulator powering the MCU is turned on or off. In this case, the MCU is only provided with a regulated supply that is switched accordingly to the voltage of the (super-)capacitor. Often this regulator is implemented using a buck-boost converter to generate the MCU supply. We emulate the buck-boost converter and the storage capacitor and directly provide the resulting on/off output to the DUT. As this behaviour is entirely analyzed by another master thesis [5], we will not discuss this matter. Please refer to [5] or [12] for an in-depth explanation of the virtual buck-boost emulation. The emulator is implemented as a configurable power supply allowing fast on/off switching the supply to the DUT. It can also accurately measure the power consumption of the DUT.

**Synthetic Trace Emulation** Apart from operating as a virtual buck-boost converter, our emulator can also generate arbitrary signals. These signals include square, triangle and sawtooth waves with adjustable frequency, duty cycle and amplitude, and a constant supply mode. Synthetic emulation can be used for stress testing and general testing of any intermittently-powered device.

## 4.3 DIPS Automated Software Testing

DIPS provides full access to the memory space of the DUT. By levering debug symbols of the compiled code via GDB, DIPS can provide a full debugging context to the developer, including rendering the call stack, variable values and all other debugging features of normal batter-free systems. Leveraging the emulator we can detect issues that are traditionally present in intermittently-powered devices, as introduced in section 3.2. Through the use of GDB and the utilisation of GDB's CLI, we are able to run custom scripts to automate specific testing for intermittent systems. We developed two software testing scripts:

**Checkpoint Correctness Test** To prove checkpoint correctness, one has to check for memory restoration correctness through power failures. The developer must specify the checkpoint function and the first function called directly after a restoration when initialising the script. Additionally, the volatile memory regions that should be compared need to be specified. After initialisation, the test compares the specified memory regions before and after a power failure and resolves the symbolic name and values when a mismatch occurs. Running this checkpoint correctness test for an extended period without mismatches in memory is often a good indicator of a correct check-point restore architecture.

**Peripheral State Restoration Test** The second automated testing procedure is for peripheral initialisation state correctness. As explained in section 3.2, this test sequence focuses on the peripheral restoration bugs. Since DIPS has access to the entire address space, including the peripheral address space, we are able to monitor peripheral states and configurations during checkpoints and check if they are properly restored. To detect the configuration registers our script parses standardised DUT MCU's .svd file–a .svd file that is commonly provided as part of a software development kit and can be requested by the manufacturer of the MCU. This .svd or register description file can be used for retrieving information on the different configuration register addresses.

# Chapter 5

# Implementation

After thoughtfully explaining the design, this chapter will list and discuss the parameters and components we used to implement DIPS. This section is again divided into the different subsystems: (i) debugger firmware, (ii) emulator firmware, (iii) emulator GUI, and (iv) DIPS CLI wrapper.

## 5.1 Debugger Firmware Implementation

The hardware debugger features are implemented by taking a popular open-source hardware debugger–the Black Magic Debug Probe [37]–as a base and building upon its functionality addressing the required features to debug and test intermittently-powered systems. The Black Magic Debug Probe is built around GDB and has support for many popular MCUs out of the box–For a complete list, please refer again to [37]. In addition, the popular power-efficient Ambiq Apollo3 [47] is supported by adding custom-created drivers.

**Intermittent Device support** The Black Magic Debug Probe [37] is modified to support intermittently-powered embedded systems. At the base, any intermittently-powered activity gives a fatal error that leads to freeing the device from memory and requiring a new connection initialisation from scratch. With

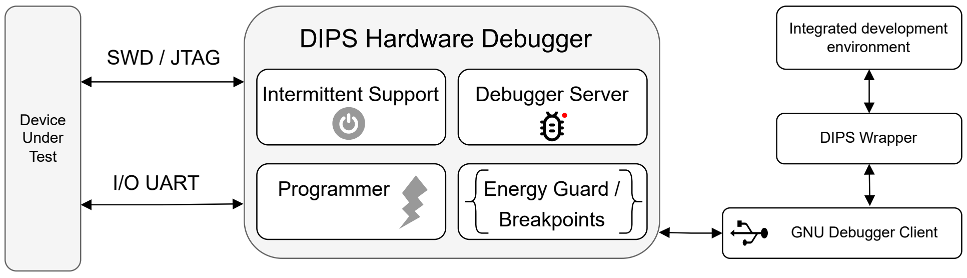


Figure 5.1: **Diagram of DIPS' debugger implementation and its relation to the energy emulator, DUT, and GDB_client. DIPS has four main functions, intermittently-powered device support, energy guards and energy breakpoints, programmer and built-in GDB server.**

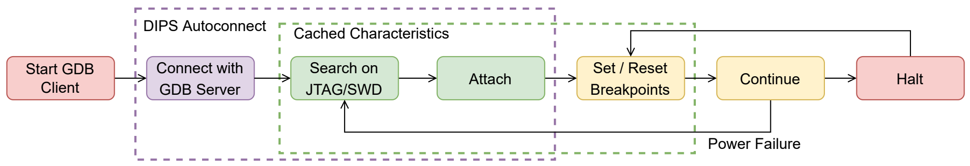


Figure 5.2: **A simplified intermittently-powered debugging process with DIPS.** **`DIPS_AUTOCONNECT` automatically searches for the correct serial port, connects to the gdb server, and attaches to the target without any user input. After the first connection, the connection characteristics are kept in the cache to allow quick reconnection. Set/Reset breakpoints are a simplified combination of multiple GDB-specific actions, either done automatically from the cache or manually from the GDB client.**

Table 5.1: **DIPS API calls for in both the GDB CLI and the extended C API, explained in section 5.1**

| 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) |

our modified firmware, instead of this error, it starts a reconnecting sequence, keeping all the DUT characteristics in place. After a successful reconnection, the DIPS retransmits the DUT's debugging parameters, i.e. breakpoint and watchpoints, allowing a quick reconnection process without notice. Intermittent support can be activated/deactivated via the DIPS CLI, by using the `intermittent_mode enable/disable` function.

**Attached and Detached Debugging** The hardware debugger and emulator are tightly interconnected by a Serial Peripheral Interface (SPI) connection onboard. Switching between the attached and detached mode becomes intuitive due to this connection, explained in section 5.2.4. Detached debugging, intended for operation when using an external harvested energy source, listens to a hardware interrupt from the DUT. This interrupt is generated by any call to the C API listed in Table 5.1. When an interrupt is generated, the emulator takes over, powering DUT. Then 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 not affected.

**DIPS Energy-Aware Debugging Features** In section 4.1.3 we introduced the concept of (i) *energy_breakpoints* and (ii) *energy_guards*.

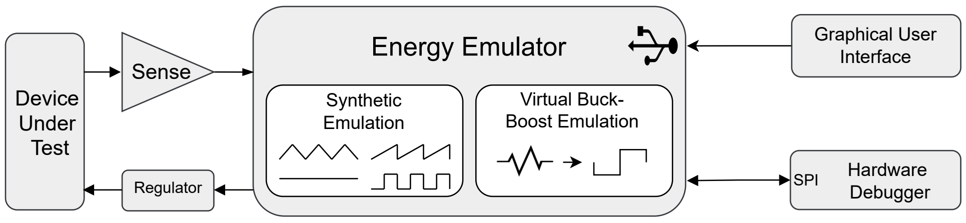


Figure 5.3: **Diagram of DIPS' energy emulator implementation and its relations to the hardware debugger, graphical interface and the DUT. Sense is implemented by the ADC of the energy emulator, regulator by the DAC regulating the linear voltage regulator on the DIPS PCB.**

For `energy_breakpoints`, we make use of the regular breakpoint implementation and a supply voltage check. Every time an energy_breakpoint is toggled, the CLI checks if the breakpoint is any form of `energy_breakpoint`, triggering the `check_eb`. This sequence either halts or continues the execution based on the threshold and measured voltage.

The `energy_guard` is implemented primarily in the energy emulator. An `energy_guard` is defined between two regular breakpoints, whenever the first of these breakpoints is triggered, the debugger enables the energy_guard threshold on the emulator, commuted over the SPI connection. The same energy_guard is again disabled after the second breakpoint is triggered, allowing a seamless transition to and from an energy_guard.

The DIPS library includes C API functions, as shown in table 5.1. We introduce three functions: (i) `DIPS_PRINTF`,a simple `printf` console TX implementation based on the serial output. Developers can trace the output directly from an external serial port, as supported by the base version of Black Magic Probe [37]; (ii) `DIPS_ASSERT`, a simple assertion, when parameters assert to false, code execution is halted and the `DIPS_ATTACH` sequence executes; (iii) `DIPS_ATTACH`, starts the hardware interrupt procedure connecting the DUT to the DIPS in detached mode. During this connection, the emulator is notified and continuous supply is provided, so no energy of the energy-harvesting device's (super-)capacitor is yielded on debug functions.

## 5.2 Energy Emulator Implementation

The energy trace emulator, based on an STM32F373 MCU [49] on the other end of the SPI-bus, is integrated with the hardware debugger. The emulator is implemented as a configurable power supply and is able to quickly switch off/on its supply to DUT. It can also accurately measure the power consumption of the DUT, as depicted in Figure 5.3.

### 5.2.1 Synthetic Wave Emulation

The emulator has built-in synthetic wave generators. Developers can adjust parameters in the GUI. The synthetic waves are generated using the built-in interrupt-based timer to allow accurate timing emulation. Currently, DIPS can generate four different artificial waves: (i) constant supply–allowing a continuous voltage for testing non-intermittent modes and required for flashing; (ii) Sawtooth wave–generating an increasing voltage to $V_{\text{in}}$, then have a sharp drop off in another straight line to zero; (iii) Square wave–a block signal between $v_{\text{in}}$ and zero over a period $p$ and with a duty-cycle of $dc$; (iv) Triangle wave–generating a symmetrical pattern over half of the period $p$ with a triangular shape. Besides these four modes, the emulator allows switching off/on by directly interfering with the linear regulator and replaying energy traces. All synthetic traces are defined using an initial or construct function—called for initialisation when switched to the related supply mode; a periodic update function—called every period, generating time-based updates in the signal; and a destruct function–called after selecting a different supply mode, for deconstruction and a clean exit (i.e. stopping hardware timers). Next, we explain how the combination of these different functions together generates synthetic waves:

i Constant supply is initialised within the construct function, and no periodic updates are required except for changes in voltage level from the end user.

ii Sawtooth wave is generated by a period function $V_{\text{out}} = (V_{\text{in}}/T_{\text{period}}) * \text{Timer}$ where Timer is reset on every complete $T_{\text{period}}$. The construct and destruct functions take care of the timer logic, including constructing and destructing the timer, respectively.

iii Square wave is produced by looking at the relative supply duty cycle provided by the end user. Again using a hardware timer, we can switch between $V_{\text{in}}$ and zero, respectively. The hardware timer either checks if its current timing is over the period threshold, calling a timer reset and making the output high or whenever the timer value is over $T_{\text{Period}} * \frac{\text{DutyCycle}}{100}$ disabling the output all together to zero.

iv Triangle wave is generated by keeping track of the up/down phase (halfway phase). Each value of the $v_{\text{in}}$ is divided by half-a-period ($\frac{2v_{\text{in}}}{T_{\text{period}}}$), multiplied by the current timer value $t$ and subtracted from $v_{\text{in}}$. The timer is reset every half-completed period, creating a real-time triangle wave emulation.

All parameters, i.e. duty-cycle, $v_{\text{in}}$ and $t_{\text{period}}$, can be changed respectively to the application within the GUI. Communication of these parameters is described in section 5.2.4.

### 5.2.2 Replay of Energy Traces

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 commonly used in state-of-the-art intermittently powered systems: the buck-boost converter and the storage (super-)capacitor, including an accurate replay of industrial standard energy trace profiles. We use the Hierarchical Data Format (HDF) file format introduced in [16] for bi-directional compatibility. The traces can be imported from within the GUI, as explained in Section 5.4. After receiving the data within our MCU, the information is added to a queue in memory, taking care of the order of received profiles and making the implementation thread-safe. The hardware-interrupt timer of the MCU interacts with both the supply and this queue, enabling a precise timing of changes within our data profile. After polling an entry from the queue, the supply is updated. Whenever only five or fewer entries are left in the queue, the queue is again replenished by requesting new data from the host.

### 5.2.3 Hardware Debugger Integration

If the energy emulator actively powers 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 pause emulation until completed. The energy emulator also implements a passive mode compatible with the debugger's detached mode, intended for a case when debugging an intermittently-powered system operating using its energy harvesting supply. When a DIPS API call occurs, the DUT voltage is first sampled in this mode. Then the emulator supplies a safe, higher voltage than the system voltage to the DUT—taking over and powering the DUT until the debugging action completes.

### 5.2.4 GUI Emulator Serial Communication

The GUI, further explained in section 5.4, has bi-directional serial communication over USB protocol to the emulator MCU. For easy implementation and future expansion possibilities, we use Protocol Buffers (Protobuf) [20], designed by Google. The Protobuf implementation takes care of the complete serial conversion, considering the challenges like false start-bit detection and differences in transmitter and receiver module, as explained in [3]. Protobuf includes an interface description language and serialises all data to binary structures, which are cross-language compatible. We use `Nanopb` [2] as plain-C implementation of Protobuf as this is not supported by Protobuf out-of-the-box.

The communication protocol contains seven different upstream and seven downstream messages, described as follows: [1]

- `CalibrationResponse` and `CalibrationRequest` are the request-response pair for the calibration options. Current and voltage calibration can be requested using the `CalibrationCommand`. The `CalibrationResult` is shared back over the `CalibrationResponse`, containing either success or failure information.
- `DataStream`, `DataStreamResponse` and `DataStreamRequest` are used for the communication of Analog-to-digital converter (ADC) data. The GUI can request a `DataStream` using the `DataStreamRequest`. The related `DataStreamResponse` contains the latest voltage, current, $\text{current}_{\text{high}}$ and $\text{current}_{\text{low}}$ data from the ADC.
- `BreakpointStream`, used for informing the GUI about the current breakpoint status, the message is broadcast whenever the system is halted or resumed.
- `StatusStream` is used for informing the GUI about the current state of the emulator. The packet is broadcast together with the `DataStream`, after a `DataStreamRequest`.

[1]The complete definitions are attached in the artefacts [13].

- `ProfileDataRequest` and `ProfileDataRespond` are the request-response pair for profile data in replay (or buck-boost) mode. When the emulator transmits a `ProfileDataRequest`, multiple `ProfileDataRespond` packages are transmitted from the GUI, containing the profile information for the next period.

- `InitialParameterResponse` and `InitialParameterRequest` are again pair-wise communication for sharing the initial configuration parameters of the emulator. Whenever the GUI is started or restarted, an `InitialParameterResponse` is requested by the `InitialParameterRequest`. The `InitialParameterResponse` contains multiple messages containing previously set values of all configurable parameters to keep the GUI and emulator consistent.

- `ModeRequest` is sent whenever a mode change is requested in the GUI. After a `ModeRequest`, the `StatusStream` enables the change.

- `PauseRequest` is a request from the GUI, requesting to pause or resume the emulation. On a `PauseRequest`, the supply is either paused or resumed, depending on the included pause or continue bit.

- `SupplyValueChangeRequest` is a particular request to change a certain parameter. Within a `SupplyValueChangeRequest` packet, one can include either a boolean, integer or float value, depending on the to-be-changed configuration parameter.

## 5.3 Automated Software Testing

The following section defines the implementation of DIPS' automated software testing protocol. We currently implemented two software tests, introduced in section 4.3. Algorithm 1 contains the pseudo-code of the common grounds of both test scenarios.

### 5.3.1 Checkpoint-Restore Memory Test

The first automated software test, based on comparing the memory after a checkpoint and after a restore point, is used to find intermittently-powered related bugs within the check/restore implementation. Before running the test, intermittently-powered embedded testers should configure the critical memory regions in the `configure.json` configuration file, allowing multiple regions. Both definitions of the checkpointing and restore functions can also be enabled in this file. As shown in Algorithm 1. After initialising the memory regions and the function definitions, the breakpoints are set on the configured lines, and the execution is resumed, waiting for any interruption. On an interrupt halt, first, we check if the halt reason is related to the checkpoint or restore breakpoint. If not, we will continue the execution. If so, we copy the memory within the supplied boundaries. If the request is part of the restore breakpoint, the latest dumped memory is pair-wise compared with the current memory dump. The memory address and the contents of this memory address are presented to the user for further investigation when a mismatch occurs. If no difference exists, the execution is resumed. When the entire utilised volatile memory is saved

```
$t_0 \leftarrow t_{\text{current}}$;
while 1 do
    $t \leftarrow t_{\text{current}}$;
    if $b$ is $B_1$ then
        /* Current breakpoint checkpoint                    */
        $t_0 \leftarrow t$;
        $c \leftarrow$dump_memory();
    else if $b$ is $B_2$ then
        /* Current breakpoint restore                       */
        $t_0 \leftarrow t$;
        $r \leftarrow$ dump_memory();
        if $r$ is not $c$ then return;     /* Pair-wise comparison */
        ;
    if $t - t_0 > Th$ then return;                     /* Timeout */
    ;
end
```

**Algorithm 1:** DIPS' simplified automated testing procedure for memory comparison. The `dump_memory()` function returns values of the assigned memory locations. $t_{\text{current}}$ is the current timestamp, required for the timeout threshold $r$. $B_1$ and $B_2$ are the relative breakpoint numbers of the checkpoint and restore functions, created during the initialisation of the script. $Th$ is the timeout threshold between checkpointing actions, defined in the settings.

at each checkpoint, then this memory state can also be restored, allowing to recreate the scenario causing the failure after the DUT experiences memory restoration failure.

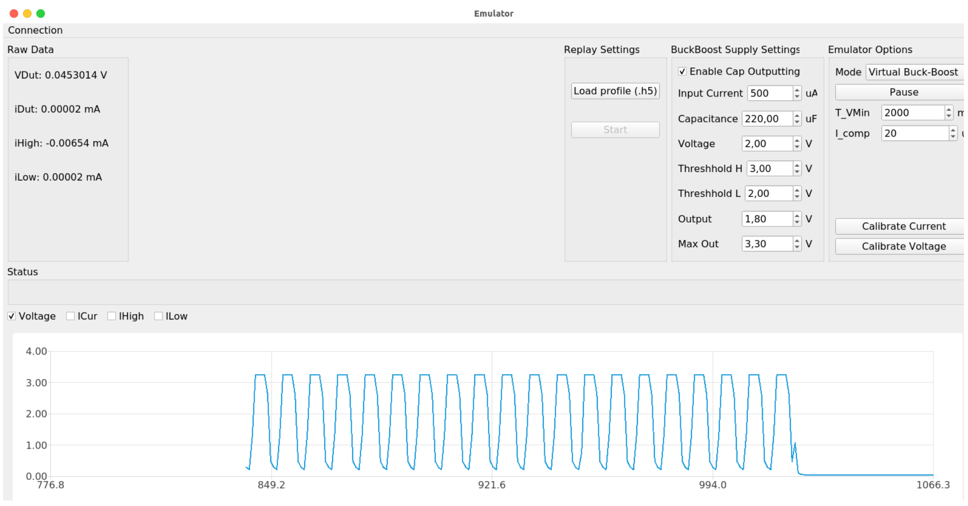
### 5.3.2 Peripheral Restoration Test

The second automated software test, like the memory restore test, is based on comparing volatile memory regions. Nevertheless, the peripheral restoration test uses register information for gathering these memory regions. Testers should configure the critical peripheral register names in the `configure.json` configuration file. The addresses of these registers can be achieved using the register description file, allowing the possibility of continuing the same testing sequence as the memory test.

## 5.4 Emulator GUI Implementation

The GUI is created using the Qt development tool [40]. The GUI has four primary tasks: (i) Serial connectivity with the emulator, (ii) visualising the ADC received signals graphically, (iii) easy adjustment of parameters and characteristics of the emulator, and (iv) writing energy traces to the buck-boost converter–replay module. The following paragraphs explain these tasks.

**Serial USB Connectivity** The physical USB connection is the first layer between the emulator and the emulator host/GUI software. The emulator and its host share a UART–Serial connection over this USB protocol, explained


Emulator

Connection

Raw Data

VDut: 0.0453014 V

iDut: 0.00002 mA

iHigh: -0.00654 mA

iLow: 0.00002 mA

Replay Settings

Load profile (.h5)

Start

BuckBoost Supply Settings

✓ Enable Cap Outputting

| | | |
|---|---|---|
| Input Current | 500 | uA |
| Capacitance | 220,00 | uF |
| Voltage | 2,00 | V |
| Threshold H | 3,00 | V |
| Threshold L | 2,00 | V |
| Output | 1,80 | V |
| Max Out | 3,30 | V |

Emulator Options

Mode: Virtual Buck-Boost

Pause

T_VMin: 2000

I_comp: 20

Calibrate Current

Calibrate Voltage

Status

✓ Voltage ☐ ICur ☐ IHigh ☐ ILow

Figure 5.4: **Screenshot of the GUI of the DIPS' emulator acting as the emulator host. The above screenshot shows the Replay/Buck-Boost emulation mode, allowing changing the Buck-Boost configuration, uploading a pre-recorded data profile and changing back to other supply modes.**

earlier in section 5.2.4. The `QSerialPort` automatically initialises a connection with the hardware emulator by searching for the related device characteristics. The Protobuf library deserialises the packets received over this connection, making it robust and easily expandable. This sequence of events ran on an independent thread to avoid distortion with other tasks.

**ADC Graphical Visualisation** The second part of the emulator host is the graphical visualisation of the received ADC data. Four values have to be visualised in graph formation. We use `QtCharts` for this matter. The four values voltage, current, current$_{\text{high}}$, and current$_{\text{low}}$ all represent as a `QLineSeries`, allowing it to be plotted within a `QtChart`. The Graph view automatically scrolls with the latest data points, creating an oscilloscope-like feeling. The `QtGraphView` allows easy pan-zoom modifications on the graph. After 1000 data entries, the first entry will be removed to avoid memory leaks on the host computer. Besides the graphical representation, the latest read values are also exhibited in decimal to allow quick lookups.

**Adjusting Emulator Parameters and Characteristics** The third, and maybe most vital task of the emulator host, is to write/read parameters and characteristics–to and from the emulator. At first we can change operating modes using the `ModeRequest` data packet, explained in Section 5.2.4. Each supply mode contains a set of unique parameters saved on the emulator and synchronised with the host on the first launch. Only parameters related to the selected supply mode can be modified by only showing these parameters and hiding unrelated properties; this creates a cleaner and more straightforward GUI design.

Table 5.2: **DIPS API calls within the GDB Client Wrapper, explained in section 5.5**

| GDB CLI | Description |
|---|---|
| `connect` | Automatically connect to the DIPS Hardware Debugger |
| `dips_memory_test` | Executes the memory testing procedure |
| `dips_peripheral_test` | Executes the peripheral memory testing procedure |
| `energy_breakpoint` | Defines a voltage-dependant breakpoint |
| `energy_guard` | Defines an energy neutral section |
| `intermittent_count` | Return number of power failures/intermittences |
| `dips_help` | Print list of custom DIPS commands |

**Writing Energy Traces to the Emulator** The last use case of the host software is the distribution of energy trace entries. As all calculations regarding the Digital-to-analog converter (DAC) and ADC are performed on the emulator MCU, we are only requested to share the voltage and current values per time unit. As we mentioned earlier, DIPS is compatible with Shepherd's [16] HDF5-format files. These files can easily be loaded within the host using the file dialog. DIPS uses the *HighFive* plain-C library [6] for decoding HDF5-formatted files. After selecting the file, an automatic process of determining the refresh rate interval is executed. To achieve enough accuracy but not overflow the serial bus, a fixed update rate of 10 Hz is chosen by effectively sending ten entries once every second. Nevertheless, a weighted average is taken over the measurements between interval updates to achieve higher resolutions within every interval. After initialising the energy traces within the host, the first measurements are written to the emulator MCU after pressing the start/resume button. The end user can always pause or stop the emulation at any time.

## 5.5 GDB Client Wrapper

As DIPS has extended the standard GDB possibilities, the standard ARM GDB-client also lacks support for DIPS' added functionalities. To confront this, we create a python-based wrapper primarily running a GDB-client subprocess. This allows us to extend standard GDB operations, adding custom ones for DIPS, based on combinations of existing GDB functions, without recompiling the complete *GDB_client* binary. Using the GDB/Machine Oriented Text Interface (MI) in combination with the `pygdbmi` package [46], we create a powerful interface allowing easy bi-directional conversion between GDB's process and python objects. Table 5.1 introduced both `energy_guard` and `energy_breakpoint`, both noteworthy functions of DIPS. Table 5.2 extends this interface with extra functions implemented within the DIPS client wrapper. The function `connect` automatically searches for the DIPS' serial port, connects, and attaches to the first debug port. `dips_memory_test` and `dips_peripheral_test` run the automated testing procedures explained in Section 5.3. `intermittent_count` returns the times the attached device has been intermittent via a power failure.

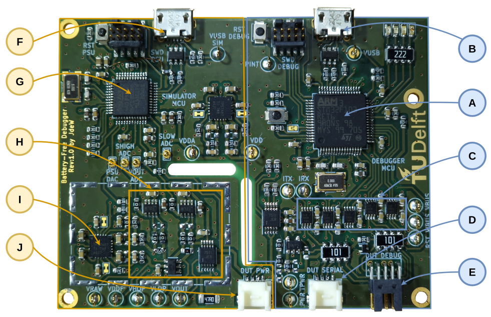


Figure 5.5: **The DIPS PCB, with the hardware debugger components (Ⓐ–Ⓔ) and energy emulator components (Ⓕ–Ⓙ).**

## 5.6 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 5.5 and marked by the blue and orange polygon, respectively. The details of each subsystem hardware implementation are briefly described as this design is beyond the scope of this thesis.

### Hardware Debugger

The hardware design of the debugger centers around a STM32F103RET [50] MCU Ⓐ and is based on the Black Magic Debug Probe [37]. It is able to communicate with the energy emulator through SPI and shares two interrupt lines. 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 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 Ⓑ.

### Energy Emulator

Central to the energy emulator is the low noise TPS7A87 [22] Ⓘ 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 [23] Ⓗ. The first amplifier 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 [21] 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 Ⓕ.

# Chapter 6

# Evaluation

We now proceed with the evaluation of DIPS. We divided the evaluation into three parts. First, we characterise DIPS in section 6.1. Second, we perform user studies to find whether DIPS is a useful (and better than state-of-the-art) tool for debugging battery-free systems in Section 6.2. Next, we show how developers can use DIPS to find bugs in differently designed case studies in Section 6.3. And finally, we use DIPS to find bugs in a state-of-the-art intermittently-powered system.

## 6.1 DIPS Characterisation

To evaluate DIPS, we conduct several measurements to evaluate the performance of our debugger.

At first, we measure the connection and reconnection rates, this shows us the average time required to establish a connection, listed in Table 6.1. The times listed in Table 6.1 indicate the overhead of the DIPS hardware debugger operating in attached mode. We might notice that early breakpoints might be missed when the device runs and the debugger is only connected after the breakpoint location. When this is unacceptable, `DIPS_ATTACH` can be placed at the program's start. The software is halted at the start, waiting for the debugger. This way, the debugger connects before any code execution at the cost of a slight delay.

Moreover, we discuss the stability of DIPS by running a duration test. Therefore, we ran the system in the attached mode for over ten hours. This test is done twice, using both [35, 47]. Both systems were put to the test by intermitting the power every 300 ms, forcing a re-connection on the system, creating at least 120000 intermittent power cycles on the DUT. Afterwards, we tested the systems on responsiveness and inspected DIPS for memory leakage, by comparing the memory allocations by `malloc`, `calloc` and `realloc` with the number of deallocations of the memory by `free`. After the duration, the number of these memory allocations was still generally moderate (4 memory allocations by the debugger), and the system behaved similarly in responsiveness after this benchmarking period.

.

Table 6.1: **DIPS debugger characterization: $t_{\text{init}}$ (initial connection time) and $t_{\text{rec}}$ (re-connection time) while connected to different devices. Data points were collected using a Saleae Logic Pro 8 [41] and averaged over ten measurements.**

| Device Under Test | $t_{\text{init}}$ (ms) | $t_{\text{rec}}$ (ms) |
|---:|:---:|:---:|
| nRF52 [Arm-M4] [35] | 311.1 | 72.7 |
| SAM4L8 [Arm-M4] [30] | 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 |

## 6.2 User Experience Studies

To assess the effectiveness of bugs found in code written for intermittently powered systems, we have designed two user experience studies. In both studies, participants were asked to experiment with DIPS and EDB [7]—state-of-the-art debugger for intermittently-powered systems. In particular, we ask 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.

### User Experience Study Participants

We have invited seven participants to the first study. Participants were recruited through mailing lists and personal contacts. Special care was taken of not recruiting people that are in a current or former professional relation with the responsible persons for this study. The second study can be seen as an extended version of this user study. Here 16 people were invited to participate, and the time per assignment was doubled, giving the participants more time to introduce both systems.

In the first study, six participants were men and one woman, the median age was 26 (youngest: 23, oldest: 44). The most comfortable programming language in which participants code was C/C++ (four participants). All participants have used an IDEs before when developing their applications, and all but one participant preferred to develop their application using this IDE. Four participants self-assessed themselves as having a lot of embedded programming skills, two some experience, one little experience, and none no experience. A large majority (i.e. five) of the participants used hardware-based debuggers for their embedded project (such as Segger J-link [45]). All participants used at least one of the debugging techniques while debugging an embedded system, such as breakpoints, watchpoints, memory views, peripheral views, etc. Where `printf` was mentioned by six participants, single stepping by three, memory inspection by three and ‘measuring voltage‘ mentioned by one participant. Only two participants did not hear about battery-free intermittently-powered systems before the start of this first study. Based on the Likert scale, on the question ‘how difficult is the application development for battery-free intermittently-powered systems?', five participants answered 'somewhat difficult' and two participants' similar to battery-powered embedded systems' (for the record, nobody found them either 'very difficult', 'easy' or 'very easy' to program).

For the main study, fourteen participants were men and two women. the median age was 26.5 (youngest: 20, oldest: 36). C/C++ was also their most comfortable programming language, all but one have used an IDE before, and five out of the 16 participants preferred to develop with an IDE. Three participants elf-assessed themselves as having many embedded programming skills, six had some experience, five had little experience and two had no experience. Next, five out of 16 have used hardware-based debuggers for their embedded project. 11 out of 16 participants were exposed to the questioned debugging techniques. `printf` was again their main debugging technique (mentioned six times), followed by single-stepping (three participants). Two out of 16 had heard of battery-free intermittently-powered systems before the study, and one had experience programming intermittently-powered systems. On the question 'how difficult is the application development for battery-free intermittently-powered systems?', 13 participants answered 'somewhat difficult' and three participants 'similar to battery-powered embedded systems; Again, 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 well-skilled (considering their background and experience) and well-informed to participate in this user experience study.

**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) [42] (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 was required to power the devices under test. Both PCs had VSCode [10] 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 of intermittently-powered systems and to read an instruction on 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 determined randomly per each participant). After 15 minutes of debugging, 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

[1]The exact text given to the participants, with the code given for debugging for both systems, together with the user experience study results, which are available as part of the original open-source repository of DIPS [13].

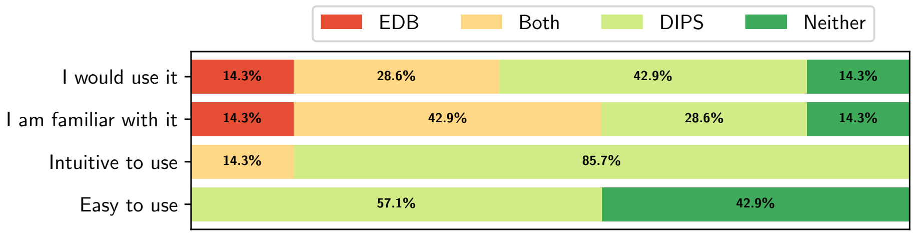


Figure 6.1: **Responses to questions in the pre-study given to user experience study participants after completing bug-finding sessions for DIPS and EDB. Note that numbers in this figure, and Figures 6.2, 6.3 and 6.4, are rounded to the nearest decimal digit.**

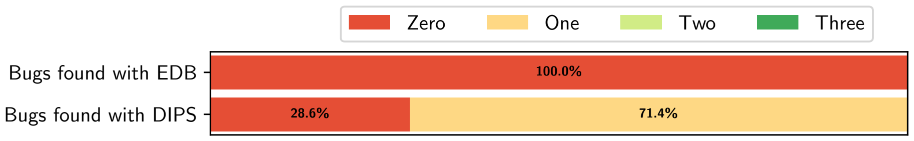


Figure 6.2: **Number of bugs found by users participated in the first (pre-) study, categorized per debugger system.**

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. In the same spirit, the study was approved by the human ethics committee of the TU Delft. The second user experience study was on all fronts the same except for the duration, which was doubled to 30 minutes per system. This study was done to improve the confidence interval by increasing the number of participants and to see whether more time would allow more people to find more bugs in either ‘System A‘ or ‘System B‘.

### User Experience Study Results First Study

The first result of the user experience pre-study is presented in Figure 6.1. We see that more users would use DIPS than EDB. Moreover, many 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 EDB debugging process works— the majority of them are familiar with the ‘regular’ way programs are debugged, what DIPS does. 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 6.2. One surprising result is that nobody was able to find any bug with EDB, while the majority of the participants were able to localise at least one bug with DIPS. We speculate that such an 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

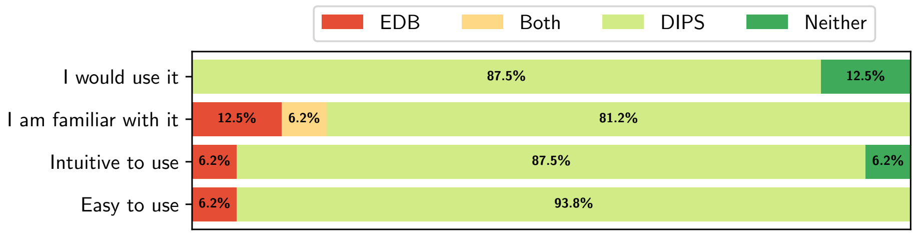


Figure 6.3: **Responses to questions in the main (and second) study given to user experience study participants after the completion of bug finding sessions for DIPS and EDB.**

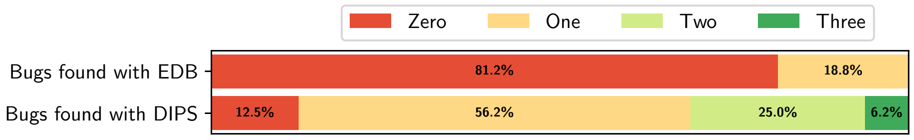


Figure 6.4: **Number of bugs found by users who participated in the main user experience study, categorized per debugger system.**

bugs in a session. On the other hand, a short time for bug finding was a stress test for both systems, suggesting that the usefulness of DIPS in code debugging (even for complex and still unexplored systems such as intermittently-powered devices) is higher than for EDB. With the main study, with more time allocated to debugging, we shall find whether this extra time would result in a significantly better perception of EDB. The results are presented in the next section.

### User Experience Study Results Second Study

The results of the main study are presented in Figure 6.3 and Figure 6.4. Comparing them with Figure 6.1 and Figure 6.2, respectively, we can conclude that increased time to find bugs in the code (from 15 minutes to 30 minutes per debugging session) did not significantly affected the perception of which system is better (Figure 6.3) nor how many bugs have been found (Figure 6.4). Actually, the main study shows that participants are more positive about DIPS than EDB, as fewer participants were pointing to EDB or to both debugging systems (Figure 6.4). Most importantly, however, more time assigned to participants of the user study resulted in more bugs to be found with EDB, but also *more* bugs with DIPS (Figure 6.4).

### Generic Observations by Study Participants

In addition to closed questions given to the participants, which results are presented in Figure 6.1, Figure 6.2, in Figure 6.3 and Figure 6.4, we have ask 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 the ability to set breakpoints and see 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 having no ability to resolve symbols. Conflicting points were listed; however, one person described EDB as 'programmer friendly' while others 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 an 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 remark stemming from all questions posed to the participants—overall, we can conclude that users found DIPS *easier to use, more intuitive and more familiar* than EDB.

## 6.3 DIPS Case Studies

Within this section, we developed two intermittent bug scenarios and ran them on an intermittently-powered embedded development environment based on the NRF52-DK [35]. The source for both scenarios is included in the artefacts [13]. The first scenario is based on the memory restoration bug explained in Section 3.2, and the second on the intermittent peripheral initialisation bug, also defined in Section 3.2. The code used for both case studies can be found in the artefacts [13].

### 6.3.1 Scenario 1: Memory Restoration Failure

In this first scenario, we have to mimic a check-restore scenario. This is done by initialising the non-volatile memory. For the load, we use a program based on the algorithm designed by Xavier Gourdon [17], which estimates the n-th digit of Pi on a rather superficial embedded system. Thanks to Cristiano Monteiro [32] for porting this to a simple Arduino-based example, which we can use in our environment. Once rewriting the code for the NRF52-DK [35] development board, we make it intermittently powered by storing the latest calculated digit (`n`) in the non-volatile memory after every iteration of calculating the next number. On startup of the development board, we restore the values of the non-volatile memory to the exact location in the volatile memory we got it from in the first place. Nevertheless, a bug is added to the restore function to mime the memory restoration failure.

[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 a stand-alone debugger in the console.

```
void FlashClass::write(int idx, uint8_t val);
.
..
void store_checkpoint(uint32_t data){
    // Writing int data to a char-sized memory space
    flash.write(0, data);
}
```

Listing 6.5: **Writing memory data to a limited memory space. In this case, the allocated memory in the Non-Volatile Memory (NVM) for the function `write` is only a single byte. Nevertheless, we tried writing a 4-byte integer to this space within the `store_checkpoint` function. This creates a buffer overflow after reaching a value of 255, losing the most-significant bit.**

**Symptoms** When inducing power failures after the value of `n` is over 255, the digit is read out as 0 again. This is no issue when running in continuous mode (no power failures).

**Diagnosis** The symptoms hint at a memory restoration issue, where either the value of `n` gets corrupted or stored/restored incorrectly within the check-restore architecture. We start by looking up the memory allocation address of our volatile restoration variables by inspecting the generated map-symbol file. We only store `n` to our non-volatile memory, so we look for the allocation of `.bss.n`. `.bss.n` is allocated at `0x200000b8` (size $0x4$), so we update our settings by filling in the related address space. After initialising the test configuration, we can run the debugger's memory inspection test, running an inspection on the functions `checkpoint` and `complete_checkpoint`. Next to that, we open the serial output to print out our value of `n` and `pi`. As supply for the system, we connected DIPS' emulator on square-wave mode, generating enough power failures to mimic a power intermittent event every five cycles. As explained in Section 1, the testing sequence compares the memory after every restore with the latest checkpoint. Running the test for the first cycles, the allocated memory locations look fine; no difference is found between the memory on checkpoint and after restore. After running the test for more cycles `n` $>$ 255, an error is located, and the execution of the test stops and outputs the inconsistency found on address `0x200000b9` (part of the `n` address space). DIPS persists to a halt for further investigation of the program on this point. Looking back to the code after identifying the issue with `n`, we notice the conversion when storing an integer to only a char-sized memory space, allowing a maximum of `n` being 255 (see Listing 6.5).

### 6.3.2 Scenario 2: Peripheral Initialisation Failure

In the second scenario, we mitigate the issue of incorrect peripheral initialisation. This is done by a faulty location of the serial initialising. A simple program is used, performing 3 simple unrelated tasks. After each finished task, the latest task ID is saved to the NVM, allowing the continuation of the program flow by restoring this `taskID` after a power failure. The case study is once again done

```
char taskID = restore_checkpoint();
switch (taskID) {
    case 1:
      goto task2;
    case 2:
      goto task1;
    default:
        break;
}
INIT_SERIAL();
...
```

Listing 6.6: **Initialisation of the serial in the wrong place, allowing a state where `INIT_SERIAL` is never called. This can be solved by putting the `INIT_SERIAL` before the restoration of the checkpoint.**

on the NRF52-DK [35] development board and can be downloaded from our artefacts [13].

**Symptoms** After each task, the application's process is shared over the serial connection, in some cases, the serial is non-responsive, sometimes even ending in a hard fault or deadlock.

**Diagnosis** The symptoms hint towards a peripheral initialisation issue, where the serial is not (completely) initialized or somehow incorrectly restored. By looking at the NRF52-DK datasheet [35], we can find the `UART0` group with the registers `CONFIG`, `BAUDRATE` and `ENABLE`. We include them in our config file `dips_testing.json`, so they will be loaded in our test. Next, we start the debugger, connect to the development board and start the DIPS peripheral memory check with our emulator on square wave mode (1000 ms, 50%, 2500 mV). After a few seconds, our script stopped, calling a mismatch between the `CONFIG` address of the checkpoint and the restore point. After locating this matter in the source (see Listing 6.6), the bug shown to be a clear example of an intermittently-powered peripheral initialisation mistake (i.e. calling the `INIT_SERIAL()` after restoring and jumping to the latest checkpointed task).

## 6.4 ENGAGE's Memory Restoration Issue

In this next section, we perform a complete system test using a start-of-the-art intermittently-powered system. We chose Engage, the system used in the Battery-Free Game Boy [14]–battery-free intermittently-powered handheld gaming console, as the selected DUT. Engage uses an optimized version of checkpointing, where only the memory regions that were changed since the last checkpoint (denoted as patches) are stored in non-volatile memory at each new checkpoint. We have connected DIPS to the DUT, as shown in Figure 6.7 and powered Engage system using the energy emulator of DIPS in a square wave mode.

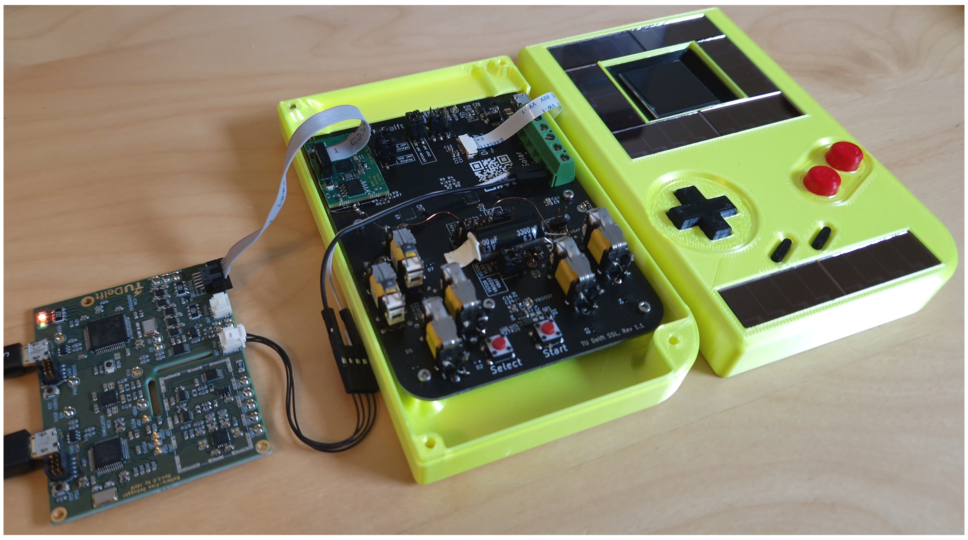


Figure 6.7: **DIPS connected to Engage [14] aiding in finding a memory restoration bug. After a certain time of continuous, intermittent execution, a checkpoint of Engage is corrupted, resulting in the handheld console failing to start.**

**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 unable to start (i.e. only a black screen is seen instead of the game content). After this failure, even when continuous power is supplied to Engage, Engage fails to start the game it intended to play.

**Diagnosis** The symptoms hint at a memory restoration issue, where either Engage's memory gets corrupted or something prevents the Engage from booting. First, we ran our software testing memory restoration script for 15 minutes, verifying that the volatile memory was 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 configurations that could prevent Engage from, for example, accessing the external non-volatile FRAM where the checkpoints are stored.

As these 15-minute long tests could not reproduce the symptoms, we have extended the testing time. After extended testing of the memory restoration, our DIPS paused code execution. This pause was triggered as no checkpoint had occurred within the predefined time window of five minutes, hinting that, most likely, no forward progress had 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. By replicating this multiple times with a square wave (800 mV, 10%, 500 ms), we were able to reproduce this issue within half an hour every time using DIPS.

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, indirectly pointing the developer to the issue (which prevents Engage from starting).

# Chapter 7

# 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 critical limitations of DIPS, with its current study below. We discuss possible directions of DIPS, DIPS limitations and potential impacts.

## 7.1 Scope of Generic Embedded Systems Applications

Even though DIPS is created for debugging intermittently-powered embedded systems, DIPS is also capable of debugging and powering generic embedded systems applications. DIPS can thereby be seen as a complete development platform also supporting intermittently-powered systems. On top of that, we challenge embedded developers to rethink their low-power embedded systems, applying energy harvesting options.

## 7.2 Support for non-ARM MCU Architectures

Beyond the list of already supported ARM devices, DIPS does not yet support debugging of non-ARM MCU architectures. One particular MCU series that requires immediate support from DIPS is Texas Instruments' MSP430 MCU's [52]–used in numerous previous projects on intermittently-powered systems. Luckily, no technical limitations would disallow support for MSP430-based systems by DIPS.

## 7.3 Increasing Reconnecting Speed

Even though DIPS increased the reconnecting rate by at least three times compared to creating a new connection, 80 ms reconnecting overhead might still be too much in some intermittently-powered cases. One could increase this even further by applying cache techniques on the different PROBE functions, which define the device connecting specifics. These PROBE functions are called recursively and count for the current overhead in the connection process.

## 7.4 Per-Line Code Inspection

What DIPS cannot do is 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.

## 7.5 Future 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 introducing new functionalities and support for new platforms, such as including MSP430 MCU [52] support mentioned above. Furthermore, since we make DIPS available to the broader community, additional features, further improvements, and evaluation can be contributed to the project by the community itself.

# Chapter 8

# Conclusions

We have presented Debugger for Intermittently-Powered Systems (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 that replicates real-life and synthesised power supply traces. Both the case studies and the user experience studies have shown that DIPS enables debugging of intermittently-powered embedded devices like any typical embedded system debugging process.

# Bibliography

[1] 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. Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus. In *Proc. SenSys*, pages 368–381, Virtual Event, 2020. ACM. `https://doi.org/10.1145/3384419.3430722`.

[2] Petteri Aimonen. Nanopb - protocol buffers with small code size. `https://jpa.kapsi.fi/nanopb/`, 2011. Last accessed: Sep 19, 2022.

[3] Liakot Ali, Roslina Sidek, Ishak Aris, Alauddin Mohd. Ali, and Bambang Sunaryo Suparjo. Design of a micro-UART for SoC application. *Computers & Electrical Engineering*, 30(4):257–268, 2004. `https://doi.org/10.1016/j.compeleceng.2003.01.002`.

[4] Brian Amos. *Hands-On RTOS with Microcontrollers: Building Real-Time Embedded Systems using FreeRTOS, STM32 MCUs, and SEGGER Debug Tools*. Packt Publishing Limited, Birmingham, United Kingdom, May 2020.

[5] Boris Blokland. Energy harvesting emulation testbed for battery-less iot. *Delft University of Technology*, 2019. Master of Science Thesis in Embedded Systems. `http://resolver.tudelft.nl/uuid:f9289d0a-bf2b-4883-83d0-f5881cc8f51f`.

[6] Bluebrain. Highfive - header-only C++ HDF5 interface GitHub repository. `https://github.com/bluebrain/HighFive`. Last accessed: Jun 16, 2022.

[7] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson Sample. An energy-interference-free hardware/software debugger for intermittent energy-harvesting systems. In *Proc. ASPLOS*, pages 577–589, Atlanta, GA, USA, 2016. ACM. `https://doi.org/10.1145/2980024.2872409`.

[8] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, page 116–127, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Contiki-NG. Official MSPSim Software Github repository. `https://github.com/contiki-ng/mspsim`, May 2022. Last accessed: Oct 2022.

[10] Microsoft Corp. Visual studio code, 2022. `https://code.visualstudio.com`.

[11] Jasper de Winkel, Carlo Delle Donne, Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proc. ASPLOS*, pages 53–67, Lausanne, Switzerland, 2020. ACM. `https://doi.org/10.1145/3373376.3378464`.

[12] Jasper de Winkel, Tom Hoefnagel, Boris Blokland, and Przemysław Pawełczak. DIPS: Debug intermittently-powered systems like any embedded system. In *SenSys'22*, Boston, MA, USA, 2022. ACM. `https://doi.org/10.1145/3560905.3568543`.

[13] Jasper de Winkel, Tom Hoefnagel, and Przemysław Pawełczak. DIPS: Debug intermittently-powered systems like any embedded system artefacts, 2022. `https://github.com/TUDSSL/DIPS`.

[14] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(3):111:1–111:34, September 2020. `https://doi.org/10.1145/3411839`.

[15] Matthew Furlong, Josiah Hester, Kevin Storer, and Jacob Sorber. Realistic simulation for tiny batteryless sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys'16, page 23–26, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. Shepherd: a portable testbed for the batteryless IoT. In *Proc. SenSys*, pages 83–95, New York, NY, USA, 2019. ACM. `https://doi.org/10.1145/3356250.3360042`.

[17] Xavier Gourdon. Computation of the n-th decimal digit of $\pi$ with low memory. In *unpublished*, 2003. `http://numbers.computation.free.fr/Constants/Algorithms/nthdecimaldigit.pdf`.

[18] Josiah Hester, Timothy Scott, and Jacob Sorber. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *Proc. SenSys*, pages 1–15, Memphis, TN, USA, 2014. ACM. `https://doi.org/10.1145/2668332.2668336`.

[19] Josiah Hester and Jacob Sorber. The future of sensing is batteryless, intermittent, and awesome. In *Proc. SenSys*, pages 21:1–21:6, Delft, The Netherlands, 2017. ACM. `https://doi.org/10.1145/3131672.3131699`.

[20] Google Inc. Protocol Buffers. `https://developers.google.com/protocol-buffers`, 2022. Last accessed: Sep 19, 2022.

[21] Texas Instruments Inc. TS5A23159 1-$\omega$ 2-channel **SPDT!** (**SPDT!**) analog switch 5-v / 3.3-v 2-channel 2:1 multiplexer / demultiplexer. `ttps://www.ti.com/lit/ds/symlink/ts5a23159.pdf`, 2005. Last accessed: May 9, 2022.

[22] Texas Instruments. Tps7a87 dual, 500-ma, low-noise, ldo voltage regulator. `https://www.ti.com/lit/ds/symlink/tps7a87.pdf`, 2016. Last accessed: May 9, 2022.

[23] Texas Instruments. Ina186, current-sense amplifier. https://www.ti.com/lit/ds/symlink/ina186.pdf, 2021. Last accessed: May 9, 2022.

[24] Vito Kortbeek, Abu Bakar, Stefany Cruz Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. BFree: Enabling battery-free sensor prototyping with python. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(4):135:1–111:39, December 2020. https://doi.org/10.1145/3432191.

[25] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. WARio: Efficient code generation for intermittent computing. In *Proc. PLDI*, pages 777–791, San Diego, CA, USA, 2022. ACM. https://doi.org/10.1145/3519939.3523454.

[26] Vito Kortbeek, Kasım Sinan Yıldırım, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. In *Proc. ASPLOS*, pages 85–99, Lausanne, Switzerland, 2020. ACM. https://doi.org/10.1145/3373376.3378476.

[27] Tianxing Li and Xia Zhou. Battery-free eye tracker on glasses. In *Proc. MobiCom*, pages 67–82, New Delhi, India, 2018. ACM. https://doi.org/10.1145/3241539.3241578.

[28] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent Execution without Checkpoints. In *Proc. OOPSLA*, pages 96:1–96:30, Vancouver, BC, Canada, 2017. ACM. https://doi.org/10.1145/3133920.

[29] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Discovering the hidden anomalies of intermittent computing. In *Proc. EWSN*, pages 1–12, Delft, The Netherlands, 2021. ACM. https://dl.acm.org/doi/10.5555/3451271.3451272.

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

[31] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. ePerceptive: energy reactive embedded intelligence for batteryless sensors. In *Proc. SenSys*, pages 382–394, Virtual Event, 2020. ACM. https://doi.org/10.1145/3384419.3430782.

[32] Cristiano Monteiro. Happy *pi* day 2022, linkedin post, 2022. https://www.linkedin.com/pulse/happy-%25CF%2580-day-2022-cristiano-monteiro/.Lastaccessed:Sep20,2022.

[33] IoT Business News. State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally. *IoT Business News*, may 2022. https://iotbusinessnews.com/2022/05/19/70343. Last accessed: Jun. 11, 2022.

[34] Nexperia. 74LVC2T45; 74LVCH2T45 dual supply translating transceiver; 3-state. https://assets.nexperia.com/documents/data-sheet/74LVC_LVCH2T45.pdf, 2021. Last accessed: May 9, 2022.

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

[36] NXP Semiconductors N.V. 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, July 2013. Last accessed: Sep. 11, 2021.

[37] Open Source Community Developers. Black magic debug repository. https://github.com/blackmagic-debug, 2022. Last accessed: May 9, 2022.

[38] Open Source Community Developers. GDB: The GNU project debugger repository. https://sourceware.org/git/binutils-gdb.git, 2022. Last accessed: May 10, 2022.

[39] Matthai Philipose, Joshua R. Smith, Bing Jiang, Alexander Mamishev, Sumit Roy, and Kishor Sundara-Rajan. Battery-free wireless identification and sensing. *IEEE Pervasive Comput.*, 4(1):37–45, Jan.–Mar. 2005. https://doi.org/10.1109/MPRV.2005.7.

[40] QT Group. QT software development framework product website, October 2022. https://www.qt.io/product/framework. Last accessed: Oct. 15, 2022.

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

[42] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an RFID-based battery-free programmable sensing platform. *IEEE Trans. Instrum. Meas.*, 57(11):2608–2615, November 2008. https://doi.org/10.1109/TIM.2008.925019.

[43] Muhammad Moid Sandhu, Sara Khalifa, Raja Jurdak, and Marius Portmann. Task scheduling for energy-harvesting-based iot: A survey and critical analysis. *IEEE Internet of Things Journal*, 8(18):13825–13848, 2021. https://doi.org/10.1109/jiot.2021.3086186.

[44] SEGGER Microcontroller GmbH. J-Link debug probe. https://www.segger.com/products/debug-probes/j-link/models/j-link, July 2021. Last accessed: Sep 14, 2022.

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

[46] Chad Smith. Python (py) GDB machine interface (mi) package. https://github.com/cs01/pygdbmi. Last accessed: Jun 16, 2022.

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

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

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

[50] STMicroelectronics. Mainstream performance line, Arm Cortex-M3 MCU with 512kbyte of flash memory, 72mhz CPU, motor control, USB and CAN. `https://www.st.com/en/microcontrollers-microprocessors/stm32f103re.html`, 2018. Last accessed: May 9, 2022.

[51] STMicroelectronics. ST-Link/v2 debug probe. `https://www.st.com/en/development-tools/st-link-v2.html`, September 2022. Last accessed: Sep 14, 2022.

[52] Texas Instruments Inc. MSP430FR59xx mixed-signal microcontrollers (Rev. F), March 2017. Last accessed: May 18, 2022, `http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf`.

[53] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *Proc. OSDI*, pages 17–32, Savannah, GA, USA, 2016. ACM. `https://www.usenix.org/system/files/conference/osdi16/osdi16-van-der-woude.pdf`.

[54] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemysław Pawełczak, and Josiah Hester. InK: Reactive kernel for tiny batteryless sensors. In *Proc. SenSys*, pages 41–53, Shenzhen, China, 2018. ACM. `https://doi.org/10.1145/3274783.3274837`.

[55] Eren Yıldız, Lijun Chen, and Kasım Sinan Yıldırım. Immortal threads: Multithreaded event-driven intermittent computing on ultra-low-power microcontrollers. In *Proc. OSDI*, pages 339–355, Carlsbad, CA, USA, 2022. USENIX. `https://www.usenix.org/system/files/osdi22-yildiz.pdf`.