UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

Electronics and Computer Science

From Transient Computing to Transient Systems: Overcoming Application Challenges in Energy Harvesting Sensor Systems

by

Alberto Rodriguez Arreola

Thesis for the degree of Doctor of Philosophy

June 2019

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
Electronics and Computer Science

Doctor of Philosophy

FROM TRANSIENT COMPUTING TO TRANSIENT SYSTEMS: OVERCOMING APPLICATION CHALLENGES IN ENERGY HARVESTING SENSOR SYSTEMS

by Alberto Rodriguez Arreola

Energy harvesting is a potential solution to power sensor systems, avoiding periodic battery replacements. Nevertheless, these sources usually incorporate supercapacitors to cope with energy intermittency caused by temporal variation in environmental conditions. Therefore, they do not solve the problem of dealing with the size and cost of energy storage. A relatively new concept termed transient computing, aims to remove big storage and enable systems to operate safely when powered from highly-variable energy harvesting sources. However, certain elementary functions of typical sensor systems, such as working with external peripherals or keeping track of time, represent important challenges in systems that operate transiently.

This thesis highlights the challenges that are fundamental to enable transiently-powered system applications and the proposed approaches to address them. This involves a quantitative evaluation of four state-of-the-art approaches to transient computing. The comparison was performed in a system powered by synthesized signals of different harvester sources. Their performances were used to identify the scenarios where one approach outperforms others, and thus, aid designers to choose the most suitable.

In order to retain the peripheral state in transiently-powered sensor systems, a generic middleware was proposed. This approach allows an application to keep the coherency between the processing unit and the state of external peripherals from one power cycle to another. The proposed middleware was tested in a transient sensor system with multiple peripherals and was able to successfully operate with I^2C and SPI protocols, causing a time overhead of only 0.82% during the complete execution of the sensing application.

A novel framework to design transient systems is also presented, including a strategy for keeping track of time. The viability of the framework was proven by designing and implementing a step counter. The experimental validation demonstrated that the step counter is able to calculate step rate, metabolic equivalent and activity time, as well as encrypt and wirelessly transmit data, reducing the required capacitance by up to 60%.

Contents

List of Figures

List of Tables

Abbreviations

ADC	Analog to Digital Converter
CPU	Central Processing Unit
CRC	Cyclical Redundancy Checking
DAC	Digital to Analog Converter
DC	Direct Current
DEBS	Dynamic Energy Burst Scaling
DMA	Direct Memory Access
EEPROM	Electrically Erasable Programmable Read Only Memory
EH	Energy Harvesting
EMU	Energy Management Unit
EPROM	Erasable Programmable Read Only Memory
FFT	Fast Fourier Transform
FRAM	Ferroelectric Random Access Memory
GPIO	General Purpose Input-Output
GPR	General Purpose Register
HB	Hibernus
IC	Integrated Circuit
IoT	Internet of Things
ISR	Interruption Service Routine
I2C	Inter-Integrated Circuit
LDO	Low Drop Out
LED	Light Emitting Diode
LPM	Low Power Mode
Mbps	Megabits per second
MCU	Microcontroller
MET	Metabolic Equivalent
MPPT	Maximum Power Point Tracking
NFC	Near Field Communication
NVM	Non-volatile Memory
ODR	Output Data Rate
PMM	Power Management Module
PP	Polypropylene
PV	Photovoltaic
PVDF	Polyvinylidene Fluoride
PZT	Lead Zirconate Titanate
RAM	Random Access Memory
RESTOP	Retaining the State of Peripherals
RF	Radio Frequency
RFID	Radio Frequency Identification
RMS	Root Mean Square

RTC	Real Time Clock
SoC	System on Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TC	Transient Computing
TEG	Thermoelectric Generator
UART	Universal Asynchronous Receiver-Transmitter
UWB	Ultra Wide Band
WiFi	Wireless Fidelity
WISP	Wireless Identification and Sensing Platform
WLAN	Wireless Local Area Network
WSN	Wireless Sensor Network

Nomenclature

E_{α}	Energy required to copy SRAM contents
E_{β}	Energy required to copy registers
E_{σ}	Energy required to save a checkpoint
$E_{capacitor}$	Energy stored in a capacitor
$E_{HB,QR}$	Total energy consumed by each approach to execute a task
$E_{H,Q}$	Energy required by each approach, to save and restore a checkpoint
E_{system}	Energy required by a system to operate
f_{source}	Frequency of the input voltage
I_{LP}	Current consumed by a system in Low Power mode
I_{o}	Current consumed by a system to perform a task
n_{α}	Number of bytes of SRAM
n_{β}	Number of bytes used by registers
n_{ι}	Number of interruptions
n_{m}	Number of trigger points
P_{A}	Power consumed in Active mode
P_{H}	Power required to checkpoint
P_{LP}	Power consumed in Low Power mode
P_{R}	Power required to restore
ρ_{s}	Proportion of trigger points resulting in a checkpoint
t_{A}	Time in Active mode
t_{a}	Time to execute an algorithm
$t_{discharge}$	Time to discharge a capacitor
t_{H}	Time to save a checkpoint
$T_{Hibernus\_QuickRecall}$	Time taken by Hibernus and QuickRecall to complete a task
$\overline{t_{\lambda}}$	Time spent sleeping
t_{LP}	Time spent in Low Power mode
t_{m}	Time to read the voltage value
$T_{Mementos}$	Time taken by Mementos to complete a task
t_{R}	Time to restore
V_{cal}	Voltage threshold to start calibration
V_{cc}	Voltage of Microcontroller
V_{check}	Voltage measured after executing a short code

V_{class}	Voltage threshold to start source classification
V_H	Voltage threshold to hibernate
V_{in-H}	High threshold
V_{in-L}	Low threshold
V_{max}	Maximum voltage allowed by the microcontroller to operate
V_{meas}	Voltage measured after saving a checkpoint
V_{min}	Minimum voltage needed by the microcontroller to operate
V_R	Voltage threshold to restore
V_{TH}	Voltage threshold to operate

Declaration of Authorship

I, Alberto Rodriguez Arreola , declare that the thesis entitled *From Transient Computing to Transient Systems: Overcoming Application Challenges in Energy Harvesting Sensor Systems* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as listed in Chapter 1, Section 1.3

Signed:..

Date: 18/June/2019

Acknowledgements

I want to thank my supervisors Dr. Alex S. Weddell and Dr. Geoff V. Merrett for giving me the opportunity to pursue a PhD and for sharing their vision and experience to route my studies. I also thank my parents Alicia and Abundio, my wife and my siblings for always supporting me and being my source of motivation, I would not be here without them.

Special thanks to Dr. Domenico Balsamo, for the long discussions we had, from which came great ideas to expand or improve my research work, and for always encouraging me to give my best effort. My gratitude to all my friends that have been with me and helped to make the PhD a wonderful experience. Last but not least, I would like to thank CONACYT from Mexico for providing me with the resources to fund my post-graduate degree. Without them, it would have been impossible for me to study here.

To my wife Elizabeth,
who left everything behind to follow this dream...

Chapter 1

Introduction

A *wireless sensor system* consists of one or various sensors to monitor physical parameters, a microcontroller to process the collected data, a power unit and a wireless communication component (such as a transceiver) to transmit or receive information [16, 17]. Wireless sensor systems have been used in smart grid applications, for traffic or air pollution monitoring, as well as tsunami detection, structural monitoring, body area networks, etc., [18, 19, 20].

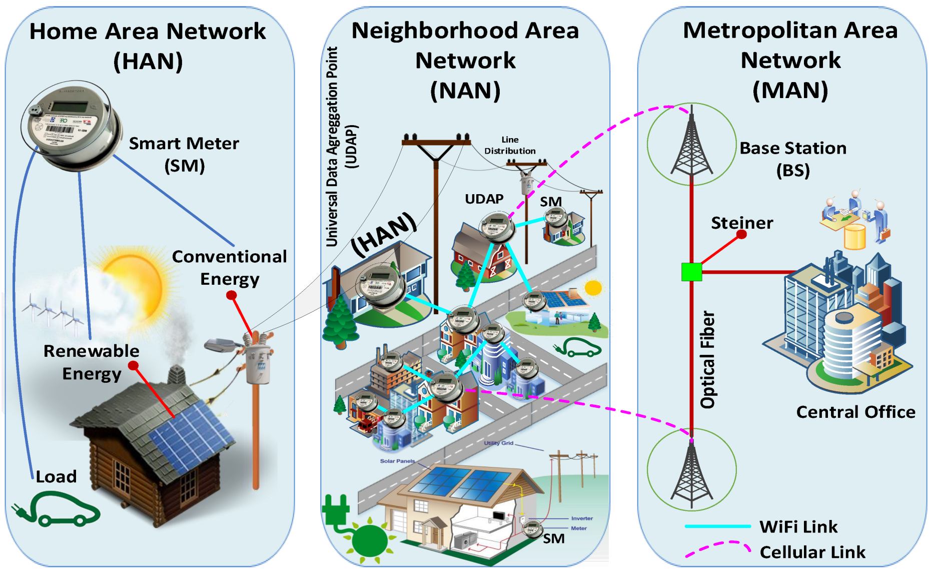


Figure 1.1: Wireless sensor network with Internet access. Extracted from [1].

When multiple wireless sensor systems are interconnected in order to receive or transmit data between them or between a sensor system and a base station, they build a *wireless sensor network* (WSN) [21, 22]. If the base station is a gateway that is able to transmit the information from sensor systems to Internet, the WSN becomes part of the Internet of Things (IoT) [16, 23]. Figure 1.1 shows a WSN with internet access for smart meter monitoring.

The aim of IoT is to interconnect "things", such as any object, animal or human organ, by providing them with the ability to transfer data over the internet network, keeping in mind these devices are ultra low-power and resource-constrained [24, 25, 26]. One of the main attributes of IoT devices is their capacity to be placed in remote locations. However, this quality provokes the challenge of powering them efficiently in order to reduce, or eliminate, the necessity of replacing batteries due to the high cost that represents [27, 28].

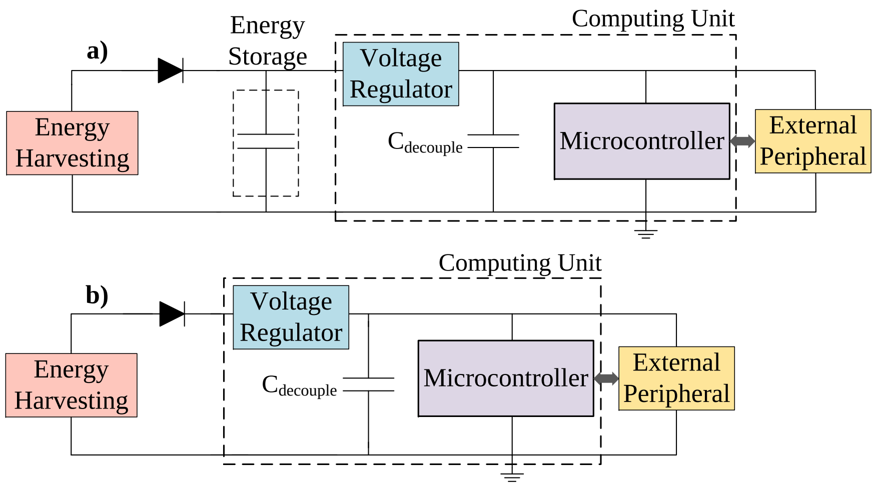


Figure 1.2: Schematic of: (**a**) a typical and (**b**) a transient EH sensor system.

Energy Harvesting (EH) is a potential solution to power autonomous sensor networks for which a power source that needs to be constantly replaced or recharged, is infeasible [29]. These sources harvest the energy from environment and convert it into electricity to supply the nodes [30, 10]. However, these sources are intermittent because they depend on the availability of the energy to be harvested. A common system executed under these conditions would be frequently interrupted every time a power failure occurs[1]. To avoid that, some energy storage units, such as rechargeable batteries or supercapacitors (Figure 1.2a), are added to sustain the operation in case a power failure occurs.

The presence of large energy storage increases the size, cost and weight of sensor systems, joined with the need of control units for power management. Several researchers have considered the necessity of reducing or removing energy storage units from sensor systems that are powered by energy harvesting sources. This has led to a new concept called *intermittent computing* or *transient computing* (TC) [31, 32]. These systems have the characteristic that their computations are governed by the intermittent energy supply (Figure 1.2b). Based on the way the system executes its functions, transient computing is grouped in two main schemes: *task-based* and *system state retention.*

Figure 1.3 shows the comparative operation of both schemes when powered from an intermittent source. In the *task-based* scheme, the system starts working when the

[1]A power failure is defined as a drop in the input voltage, below the operating threshold.

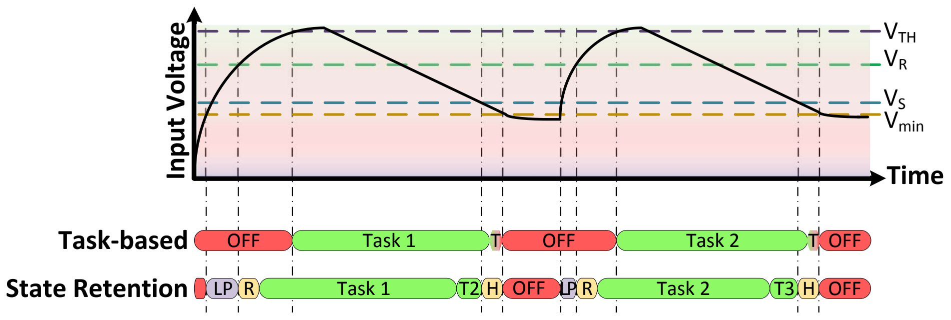


Figure 1.3: Comparative operation of a task-based and a state retention schemes, when powered by an intermittent source.

input voltage reaches the predefined threshold (V$_{TH}$) that guarantees the capacitor has stored enough energy to complete their tasks before the transition between active and off states (T). Thus, an *atomic* task[2] (the same or a different one) is executed at each power cycle. However, those existing systems are completely oriented to solve specific tasks, i.e. they do not offer a general solution applicable to different scenarios.

In *state retention* scheme, as also shown in Figure 1.3, application tasks are progressively executed through various power cycles by saving the system state (snapshot) into a non-volatile memory (NVM) when the system voltage drops below the snapshot threshold (V$_S$), and restoring it when the voltage rises over V$_R$. The system remains in low power mode (LP) while the voltage is between V$_R$ and the minimum level required by the microcontroller (MCU) to work (V$_{min}$). These schemes are application agnostic and aim to operate directly from the energy harvester, taking advantage in some cases of the capacitance required by microcontrollers to decouple the electrical noise of the source from the system. Some *state retention* approaches [33, 34] are based on trigger points inserted in the system code, during application development or at compile time, which check the input voltage to decide whether a snapshot has to be saved. To prevent saving periodic snapshots, some researchers have proposed approaches [35, 36] that dynamically adapt trigger points in the code, based on the energy conditions. Others [3, 8] monitor the input voltage and save a snapshot when level drops below a defined level.

1.1 Research Justification

Existing *task-based* transient sensor systems have the important limitation that the size of the small capacitor depends on the amount of energy needed to complete their functions without power failures. The longer the task, the larger the capacitor. This means that if the incoming power is not enough to charge the energy storage until a predefined voltage threshold, the system would never work or its operation would be

[2]A task is defined as *atomic* when it is not splittable, i.e., it is completed before a power failure occurs.

interrupted before completing the task, restarting the execution from the beginning when energy is again available [37, 38].

In the case of *system state retention* approaches, they aim to maximise the active time (without adding any extra energy storage) and protect the system from volatility by saving a snapshot before a power failure. Thus, a relatively long-running computation (also called long-term computation, which is defined as a task that the processing unit takes longer to complete than the time the energy source can power the system), can be split and progressively completed through various power cycles. However, these approaches may fail to execute tasks that need to be executed without interruptions such as wireless transmissions. Moreover, each approach attempts to retain system state through distinct methods, which causes their performance to vary when powered by different types of energy harvesting sources. These approaches are also only effective in retaining the state of on-chip peripherals that are controlled by the special function registers of the microcontroller unit (MCU), e.g. internal ADCs. However, the vast majority of sensing systems also include external sensors, actuators, radio transceivers, etc., [39, 40, 41].

In addition to these limitations, there is an important challenge that *task-based* and *system state retention* approaches share, which is keeping track of time. Several IoT applications need to have a sense of time as part of their functionality, e.g. to know whether a sampled value is valid or has already expired. However, this function is not trivial in a system that suffers from frequent power losses. Although different solutions to keep track of time in transient systems have been proposed [42, 9], the existing approaches are only able to keep it for short periods (in the order of few seconds).

In summary, the perspective has to move from transient computing, which is only aware of the processing unit, to transient systems, which implies to address application requirements, such as operating with external peripherals, keeping a sense of time and performing long-term computation, working from highly variable and scarce EH sources but still minimizing the need of extra energy storage.

1.2 Research Questions

Based on the critical analysis of existing autonomous wireless sensor nodes and the above justification, this thesis seeks to answer the following research questions:

1. **How does the performance of prominent *system state retention* approaches compare, under distinct conditions when powered by different energy harvesting sources?** Each proposed approach attempts to retain system state with different techniques. Therefore, under certain conditions, one approach may outperform the others, and be more suitable for certain applications.

2. **How can the state of external peripherals be retained in transient sensor systems?** Efficient and low-power methods to retain the state of peripherals have to be proposed, which can work with off-the-shelf peripherals. The research should also be oriented to define how the existing *system state retention* approaches need to be adapted to operate with a method to retain peripheral state, keeping the coherence with the main application, between power failures.

3. **How can application execution be performed under highly variable energy harvesting conditions, whilst minimizing the need for extra energy storage?** In order to increase the system efficiency, it is necessary to enable systems whose operation is linked to the energy availability, and therefore, be able to work even in cases of severe energy shortages, without increasing the size of capacitors.

1.3 Research Contributions

Addressing the research questions has led to the following contributions:

1. A **mathematical and quantitative evaluation of four state-of-the-art *system state retention* approaches (Mementos, Hibernus, Hibernus++ and QuickRecall)**, in order to aid designers to select the most suitable approach, depending on the energy harvester conditions. This contribution addresses the Research Question 1, and the obtained results were reported in the following publications:

   A. Rodriguez Arreola, D. Balsamo, A. K. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, and G. V. Merrett, "Approaches to Transient Computing for Energy Harvesting Systems: A Quantitative Evaluation," in *Proceedings of the Third ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems - ENSsys'15*, (Seoul, South Korea), pp. 3-8, 2015. DOI: 10.1145/2820645.2820652 [43].

   D. Balsamo, A. S. Weddell, A. Das, **A. Rodriguez Arreola**, D. Brunelli, B. Al-Hashimi, G. V. Merrett and L. Benini, "Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.35, no.12 pp. 1968-1980, 2016. DOI: 10.1109/TCAD.2016.2547919 [8]. My **contribution** to this paper was the experimental comparative performance between Hibernus, QuickRecall and the proposed scheme Hibernus++.

2. **Design and implementation of a case study for transient computing**. As a motivation for existing challenges of transient computing systems and as a comprehensive validation to demonstrate the viability of the proposed design

framework to address application requirements (sense, process and transmit), a transient computing step counter was implemented and experimentally validated. The obtained results were reported in the following publication:

A. Rodriguez Arreola, D. Balsamo, Z. Luo, S. P. Beeby, G. V. Merrett, and A. S. Weddell, "*Intermittently-powered Energy Harvesting Step Counter for Fitness Tracking*," in 2017 IEEE Sensors Applications Symposium (SAS), pp. 1-6. DOI: 10.1109/SAS.2017.7894114 [44].

3. A **generic middleware to retain the state of digitally-interfaced peripherals in transiently-powered sensor systems.** For this purpose, **RESTOP** (**R**etaining the **ST**ate **O**f **P**eripherals), was proposed, capable of saving and restoring the state of external peripherals connected through I^{2}C or SPI protocols. This contribution addresses the Research Question 2, and the obtained results were reported in the following publications:

   A. Rodriguez Arreola, D. Balsamo, G. V. Merrett, and A. S. Weddell, "RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems," *MDPI Sensors Journal*, vol. 18, no. 1, p. 172, Jan. 2018. DOI: 10.1145/2820645.2820652 [45].

   A. Rodriguez Arreola, D. Balsamo, G. V Merrett, and A. S. Weddell, "A Generic Middleware for External Peripheral State Retention in Transiently-Powered Sensor Systems," *in Proceedings of the Fifth ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems - ENSsys'17*, (Delft, The Netherlands), pp. 37-39, ACM Press, 2017. DOI: 10.1145/3142992.3143000 [46].

4. A **design framework that intelligently combines the strengths of *task-based* and *system state retention* schemes in an energy-aware manner, which also proposes a strategy for keeping track of time.** The design flow incorporates the energy harvesting process in the application design and enable both schemes to coexist as a function of the harvested energy, in order to reduce the required capacitance. This contribution addresses the Research Question 3, and the proposed framework is intended to be reported in the following publication:

   A. Rodriguez Arreola, D. Balsamo, O. Cetinkaya, S. Wong, G. V. Merrett, and A. S. Weddell, "Meeting Application Requirements in Transiently-powered Energy Harvesting Systems," *IEEE Sensors Journal* (Under preparation).

1.4 Thesis Outline

The reminder of this thesis is highlighted in the diagram of Figure 1.4. **Chapter 2** presents a survey of state-of-the-art articles in transient computing systems, their drawbacks and limitations. The chapter concludes with a quantitative evaluation of four *system state retention* approaches, when powered by different intermittent sources and existing challenges in transient computing systems. **Chapter** 3 describes a methodology to define whether an application is suitable to operate transiently. Then, the design and implementation of a case study (a basic step counter) is presented, to demonstrate the viability of transient systems. **Chapter** 4 proposes a generic middleware to retain the state of external peripherals when working on transient systems. This chapter also presents the characteristics and advantages of the proposed system as well as a thorough practical evaluation to validate the operation of the middleware and the time overhead it causes in an intermittently-powered sensor system. **Chapter** 5 describes the novel framework to design transient systems able to perform *atomic* tasks and long-running computation without adding any extra capacitance. This framework also proposes a strategy to solve time issues in transient systems. The chapter ends with a case study (a transient computing step counter), able to keep track of time, perform long-running computation and transmit data wirelessly, in order to probe the viability of the proposed framework. **Chapter** 6 concludes this thesis and outlines different areas of future research work.

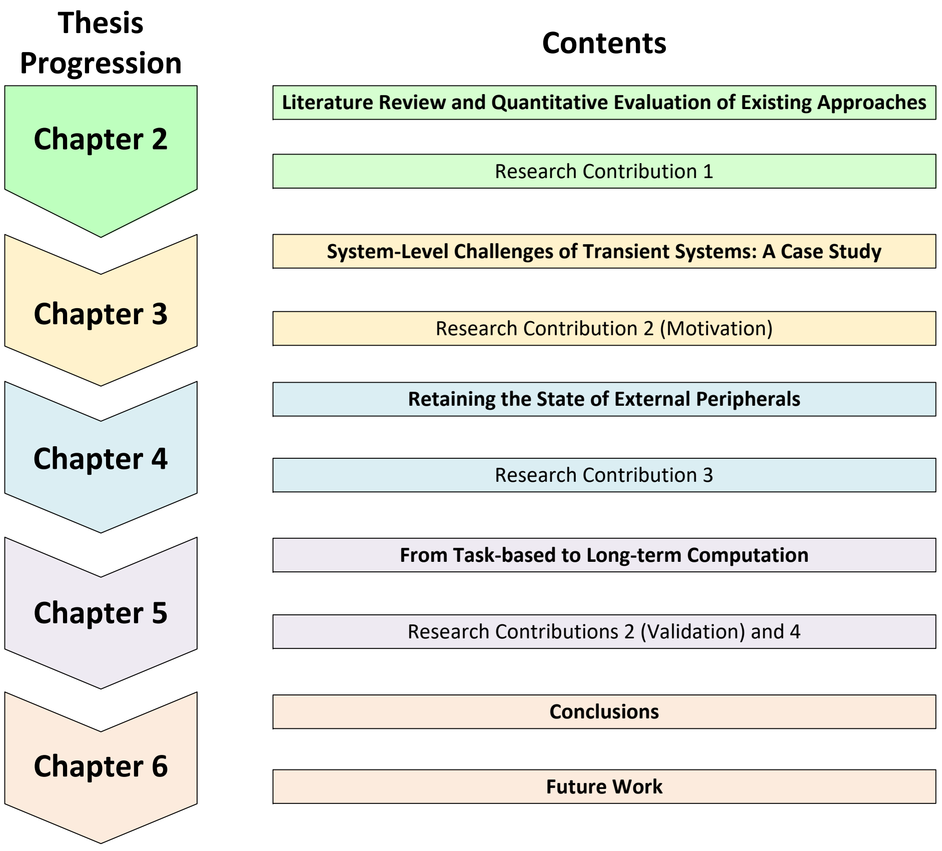

Thesis Progression
Contents
Chapter 2
Literature Review and Quantitative Evaluation of Existing Approaches
Research Contribution 1
Chapter 3
System-Level Challenges of Transient Systems: A Case Study
Research Contribution 2 (Motivation)
Chapter 4
Retaining the State of External Peripherals
Research Contribution 3
Chapter 5
From Task-based to Long-term Computation
Research Contributions 2 (Validation) and 4
Chapter 6
Conclusions
Future Work

Figure 1.4: Chapters content of this research work.

Chapter 2

Transiently-Powered Energy Harvesting Sensor Systems

Sensor systems are becoming pervasive. Their characteristic of being placed in any location, including nature, buildings, objects and even human organs, is determined not only by their capacity of processing data but also by their dimensions and life time. These parameters offer important challenges that have been solved in different manners by researchers. In the context of this research work, it is presented a survey of the state-of-the-art in transiently-powered energy harvesting sensor systems and a quantitative evaluation of prominent software approaches to transient computing. This includes the description of sensor systems and their structural components, as well as the dynamics of energy harvesting sources.

2.1 Autonomous Wireless Sensor Systems

A wireless sensor system is autonomous when it is capable of sensing, processing and communicating wirelessly, without the need for maintenance or supervision [47, 48]. As part of the IoT, autonomous sensor systems have had a rapid growth in applications such as smart homes, intelligent transportation, security, etc., [49, 50, 51]. A new model, called *Smart City* has been developed based on IoT and cloud computing (defined as a group of accessible and virtual compute resources that can be reconfigured in a pay-per-use model via the Internet [21, 52]), in order to share information and coordinate city systems [53, 54, 55]. This method includes the installation of thousands of wireless sensors to build a network connected to internet. An example of this is to monitor the public street lighting by sensing the light intensity at each lamp or collecting environmental data to measure noise, temperature, pollution levels, etc.

The IoT growth offers many research opportunities to solve the challenges that appear in each new application. One of the principal concerns is powering IoT devices in an efficient manner in order to increase their lifetime, autonomy and ability to be placed in remote locations [56, 57]. This provokes the necessity of developing new models to power them, reduce their size and cost, and increase their capabilities, which causes a change in the way their components interact each other [58]. Figure 2.1 shows a simplified diagram of the structure of a typical IoT device, which includes the energy management and processing units, the non-volatile memory and peripherals. In the following sections, each element linked to the processing unit is described and their existing challenges, when looking for a cheaper, smaller and more energy-efficient design, are analysed.

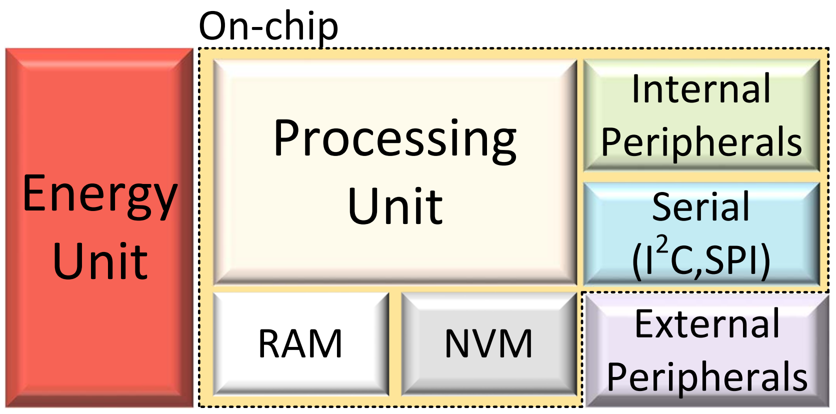


Figure 2.1: A simplified block diagram of the structure of a typical sensor system designed for IoT.

2.1.1 Energy Sources

Energy is one of the main constraints in wireless sensor networks. The ability to place sensor systems in remote and inaccessible locations can be limited by the necessity of replacing their batteries [30, 59]. Therefore, *self-powered devices* have to be designed, which do not need a separate connection to an external power source or their batteries do not have to be replaced or externally charged [60, 61]. This means the system operates autonomously until fails for the natural degradation of components [62].

In last decades, technology has advanced in the miniaturization of solutions. However, the batteries to power them still cover much of the hardware weight and space (and their lifetime is reduced). The cost of batteries is high and in consequence, the complete solution as well. In Figure 2.2 it is possible to observe the size of the battery compared with the digital circuit. Therefore, in the vision of IoT expansion and miniaturization, electrochemical batteries represent an important drawback.

Energy Harvesting (EH) is an efficient solution to power sensor systems. EH sources harvest energy from ambient sources such as light, vibration, pressure, wind, radio frequency signals or temperature differences [63, 64], and convert it into electric power.

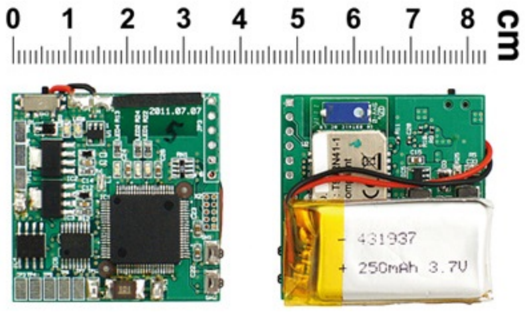


Figure 2.2: Comparative between battery size and sensor circuit. Extracted from [2].

Each harvesting method has a different power generation capability and in consequence, the power requirements of the system and the environmental conditions where the sensor will operate have to be analysed, in order to choose the method that best suits [65]. Table 2.1 shows different harvesting methods with their power capabilities.

Table 2.1: Power density of energy harvesting sources, extracted from [15].

Energy Source	**Harvested Power**
Vibration/Motion	
Human	$4\mu W/cm^2$
Industry	$100\mu W/cm^2$
Temperature Difference	
Human	$25\mu W/cm^2$
Industry	$1\text{-}10mW/cm^2$
Light	
Indoor	$10\mu W/cm^2$
Outdoor	$10mW/cm^2$
RF	
GSM	$0.1\mu W/cm^2$
WiFi	$1\mu W/cm^2$

However, EH sources are typically intermittent due to temporal variation in the environmental parameter (e.g., time of day, weather conditions and available light) [66, 67]. Figure 2.3 shows an example of the harvested energy from a micro-wind turbine and a photovoltaic module. There, the voltage and frequency of the micro-wind turbine are dependent on the wind velocity. Similarly, the current level from the photovoltaic cell, used in this example, changes depending on the intensity of the light source.

There are other energy harvesting sources based on piezoelectric effect that converts mechanical energy into electrical energy [68, 69, 70]. The most common piezoelectric materials used in kinetic energy harvester are lead zirconate titanate (PZT) and polyvinylidenefluride (PVDF) [71, 72]. Some systems, powered by insoles based on

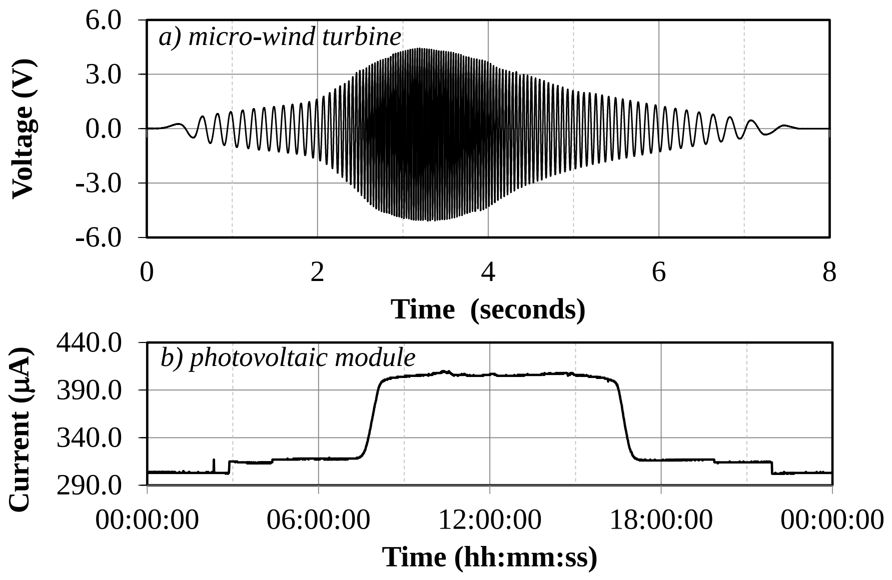


Figure 2.3: Output signal of (a) micro-wind turbine and (b) photovoltaic module. Adapted from [3].

piezoelectric materials are presented in [73, 72]. However, these insoles are rigid and reduce the comfort of footwear.

Recently, new materials, called ferroelectret, have been developed, being the cellular polypropylene (PP) one of the first commercialized ferroelectrets [74]. Ferroelectret materials are flexible cellular polymer foams, that can store charges in their internal gaps, generating electric pulses when the lamina is bended or compressed [75]. PP foams are light, soft, thin and with relatively high piezoelectric charge constant compared with PZT and PVDF piezoelectric materials [76]. These properties make PP materials more suitable for wearable sources that harvest energy from human motion [77]. Figure 2.4 shows the rectified power signal of a ferroelectret insole [77], with a high-impedance workload after three steps. Here, it is possible to see these harvesting sources generate high-power bursts for few ms.

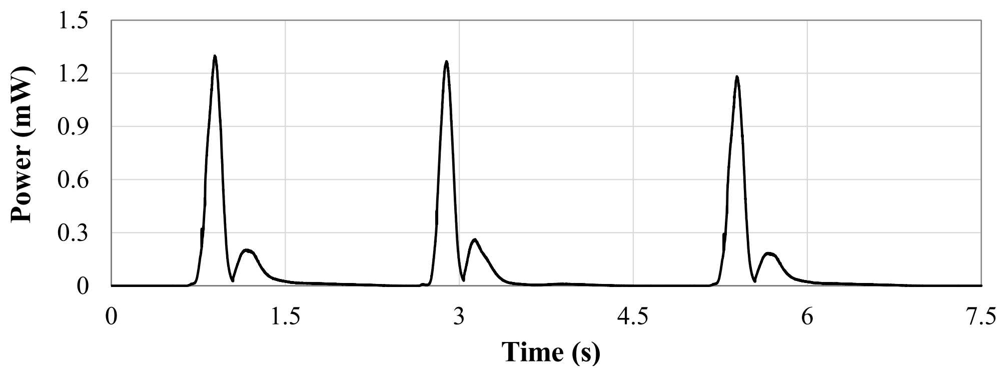


Figure 2.4: Rectified power signal generated by a PP ferroelectret insole.

From the analysis of different EH sources, it is possible to conclude that an application executed on the sensor system powered by them, can potentially be interrupted depending on the harvested power availability and the minimum voltage level required by the platform to operate. To overcome this limitation, sensor systems typically integrate energy storage units in the form of rechargeable batteries or supercapacitors (that are also affected by the environmental conditions, increasing the leakage current, causing a faster discharging [78]) to buffer energy in order to sustain computation at times of power unavailability [79]. Therefore, EH does not solve the problem of big storage. In recent years, designers have looked for alternatives to reduce the size of the energy storage elements (or remove them if possible) whilst ensuring their functionality.

2.1.2 Non-volatile Memory

Non-Volatile Memory is a type of computer memory that can retain the saved data regardless whether the power is interrupted [80]. Unlike SRAM or other volatile memory, NVM does not need to refresh its memory data periodically. These kind of memories are widely used in different digital devices because they eliminate the necessity of secondary storage systems such as hard disks. There are different NVM families, which can be qualitatively compared in terms of the possibility to be programmed and erased many times and the minimum cell size that can be written a time. One of the first NVMs is the erasable programmable read only memory (EPROM) which is programmed by channel hot electron (CHE) and erased by ultraviolet (UV)[81, 82]. Shortly after, the Flash EEPROM was proposed, which is an EPROM cell that can be electrically erased [83]. This memory was named *Flash* because it has the capability of erasing a whole memory cell in the same (fast) time. The capacity of initial Flash memories was in the order of 16Mb, but nowadays they reach up to 64Gb or above [80].

Nevertheless, the characteristics of these memories may cause a significant time and energy overhead (up to 200x [84]). For example, flash memory bit-cell can only be written from logic 1 to logic 0. That means that if it is necessary to write a logic 1 in a cell that was previously written with a logic 0, the bit-cell has to be first erased. Other disadvantage is memory Flash can only be written segments whose size (number of bytes) depends on the designer, e.g. a segment of the Flash memory included in the board MSP430F2142 contains 512 bytes and requires about 14ms to erase it [85]. It has to be considered that an erase operation consumes more energy than writing, which is an important concern in systems that need to write data constantly [86].

Recently, other NVM designs have emerged such as Ferroelectric RAM (FRAM) and Magnetoresistive RAM (MRAM), being the FRAM more widely adopted in low power microcontrollers. MRAM has not been extensively incorporated in the market because they are still over 180nm region (some in the simulation phase [87]), which makes them obsolete for new technologies [88]. FRAM memory technology combines the best of

Flash and SRAM. It is non-volatile as Flash but consumes less power and is faster to be written. These properties allow it to work with dual purposes in some applications: as volatile and non-volatile memory [7].

2.1.3 Peripherals

IoT devices need both inputs and outputs to be profitable. These functions are performed by devices called *Peripherals* [89]. These devices are not part of the essential or core computer but are connected to the central processing unit, allowing sensing, transmitting or receiving information. In Figure 2.5, a basic block diagram of a microprocessor is presented, including the on-board peripherals.

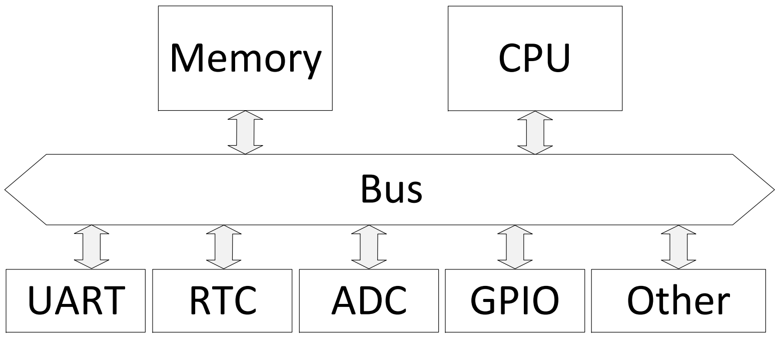


Figure 2.5: Interaction of the CPU with on-board peripherals. Adapted from [4].

The communication between the processor and the on-board peripherals is done through channels that are called *buses*. The information that is transmitted between two devices in different lines of the bus could be of address, control or data, whilst the communication between the microprocessor and devices can be synchronous or asynchronous. In the first case, the communication is synchronized by the clock according to a protocol. In asynchronous communication, it is done by either a handshaking protocol or interrupts, without the clock signal [90]. The interrupts allow to stop the process and receive an instruction from devices, e.g. the internal comparator of an MSP430 microcontroller [91] stops the program execution and performs a predefined instruction when the pre-configured condition is met.

In order to facilitate the communications between the central processing unit (CPU) and the on-board peripherals (because that communication is slower than operations inside the CPU), the following methods are implemented [90]:

- Special input/output (I/O) instructions: to specify the device number and the command.
- Memory-mapped I/O: uses the same bus to address the memory and I/O devices.

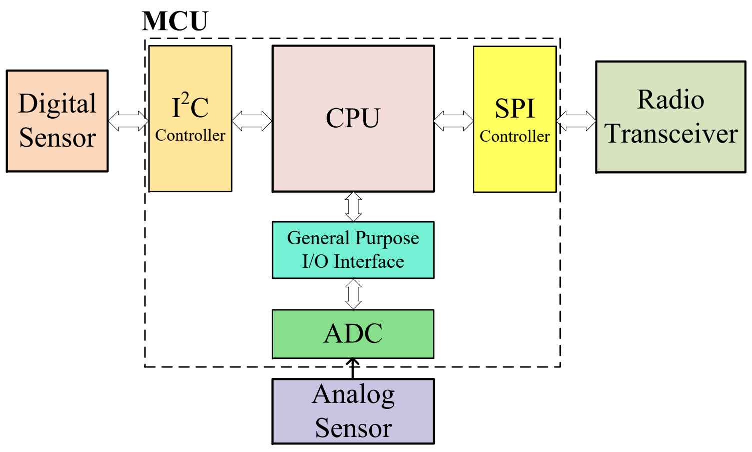


Figure 2.6: Block diagram of an MCU interacting with different peripherals.

- Interrupt: as explained before, this is from the peripheral to the CPU.

For data transfer between the memory and the I/O, there are two important methods to control this procedure:

- Direct Memory Access (DMA): the main memory (which is usually a volatile memory such as RAM) is accessed directly by the peripheral without requesting permission to the CPU.
- CPU control: the processor controls the transfer between memory and I/O.

Some microcontrollers manage the peripherals with general purpose-registers, e.g. the MSP430 microcontroller [92] has 230 registers (which are dedicated memory allocations) for the different peripherals, such as ADC, internal comparator, general-purpose input-output pins (GPIO), etc. However, the vast majority of sensing systems also include external sensors, actuators, radio transceivers, etc., that are attached to the MCU through serial protocols such as SPI and I^2C. Figure 2.6 shows a block diagram of an MCU interacting with three peripherals: an analog sensor connected via an on-chip ADC, a transceiver connected through SPI and a digital sensor attached via I^2C.

Sensor

A sensor is a device that converts physical parameters into a signal that can be electrically measured [93, 94]. The physical parameters could be temperature, light, speed, pressure, etc. and the output is a signal that can be transmitted electronically over a network for reading or processing. The sensors can be classified according with their functionality as [95]:

- Analog: These sensors transmit signals comprised of a field of instantaneous values that vary over time, and are proportional to the effect being measured, e.g. a thermometer. In microcontrollers, those sensors are commonly connected to the analog-to-digital converter (ADC).
- Digital: These sensors digitally convert the sampled data and typically transmit it through serial interfaces, such as serial peripheral interface (SPI) or inter-integrated circuit (I^2C).

Transceiver

In wireless sensor networks, the communication between systems or between a system and a base station, is mainly performed through radio frequency (RF) electromagnetic signals [96]. Nowadays, the wireless communication is usually developed by a single peripheral (that transmits and receives) which is called *transceiver* [97]. The performance of these peripherals is important because they are generally the most energy consuming elements in a wireless sensor system.

Transceivers usually operate in three main modes, which are: *active*, *transmitting* and *receiving* [98]. In *active* mode, the energy consumption is relatively low, because the peripheral is exclusively waiting to start the transmission process, to send periodic beacon messages or to decode an incoming packet. In *transmitting* mode, the power consumption is higher, caused by the radiated energy and the losses within the antenna during the packet transmission. Finally, in *receiving* mode, the energy is mainly consumed by the amplifier needed to boost the received signal in order to be properly decoded.

2.2 Making Forward Progress in Transient Systems

In order to reduce the inconveniences caused by the presence of big energy storage, some designers have focused their research activities on self-powered sensor systems that reduce or remove the energy storage units. The operation of these new designs, named *transient* computing (TC) or *intermittent* computing [3, 99], is governed by intermittent energy supplies. Nevertheless, operating transiently makes the computation difficult. Common programs could not properly operate because they would be restarted every time a power failure occurs, which is not an expected behaviour. It is necessary to design new software and/or hardware units that can operate under intermittent supply conditions. In the following sections, a survey of existing systems is presented, grouped in two main orientations: *task-based* and *system state retention.*

2.2.1 Task-based Transient Schemes

Task-based designs attempt to split the application into *atomic* tasks, which are executed without power interruptions. For this purpose, they typically incorporate capacitors, whose sizes depend on the amount of energy needed to complete the largest *atomic* task (e.g. process or transmit data) before a power failure. In this section, a critical review of existing *task-based* approaches, is presented.

One of the prominent designs is the wireless identification and sensing platform (WISP) [5], capable of operating by harvesting 915-MHz RF signals and in a maximum distance from the RF source of 4.3m. The platform is able to wirelessly transmit through backscatter modulation. WISP includes a 10μF capacitor that is charged up to 1.9V in order to be capable of sensing, processing and transmitting data before a power failure. Therefore, this design is only able to perform the same task at each power cycle. If the EH source performance or the processing application changes, the design has to be updated.

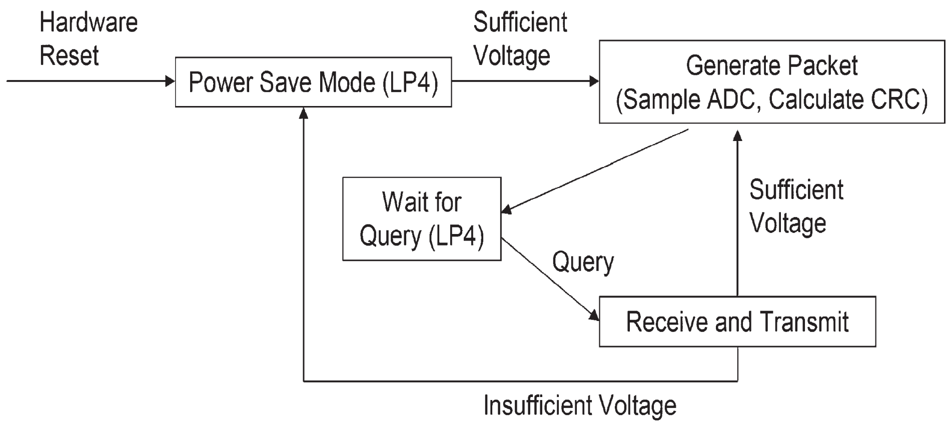


Figure 2.7: Operational power cycle of WISP platform. Extracted from [5].

Figure 2.7 shows the block diagram of the operational power cycle of the WISP platform. WISP has a circuitry that checks the voltage supply level in order to define whether there is enough energy. If so, the unit sends an interruption and enables the unit called *Generate Packet*. This unit is in charge of powering the attached sensor, checking the data sent by the sensor through ADC and calculating the cyclical redundancy checking (CRC) that will be added in the packet. Then the system enters in low power mode until a RFID query is received. This signal enables the unit called *Receive and Transmit*. If there is sufficient voltage, the MCU receives the reader command and sends the packet generated (after recognizing the query). This procedure is done twice (if the stored energy is sufficient) to improve communication reliability. If the voltage level is below the threshold, the system goes to sleep instead of sending the packet. In the case a power failure occurs while processing the information, all that work is lost and the system has to be restarted the next time the power is available.

S. Naderiparizi *et al.* [100] proposed an application of the WISP platform previously described. It is a system named WISPCam which takes a picture every time the energy stored in a capacitor is over a predefined threshold. This system uses a BQ35570 boost to amplify the voltage level in order to accelerate the charging process of a capacitor of 6.08mF. If the RFID source is placed at 20cm away from the WISPCam, it is able to take a picture every 10s. However, the maximum distance the WISPCam could operate is 5m from the source, taking an image every 15 minutes. The MSP430FR5969 microcontroller was used, taking advantage of its 64KB FRAM, which allows a fast-transfer speed of each taken picture from the camera to the NVM before a power failure occurs. Then, that image is transmitted via backscatter communication, which allows to use the same RFID signal by modulating the impedance presented to the antenna.

Another *task-based* device was presented by V. Talla *et al.* [27]. They proposed a Wi-Fi Radio Frequency EH source for a wearable temperature sensor. This system is capable of harvesting energy from Wi-Fi transmissions in a frequency range from 2GHz to 5GHz. The design has two antennas, one to harvest the RF energy, and the second to transmit data. The system operates in three modes: Off, Sleep and Active. In Off mode, the system consumes about 350nA of leakage current. In Active mode, the sensor takes a sample, encapsulates it in a packet and transmits it using the asynchronous ANT transmission to the access point. When the transmission is completed, the system enters in Sleep mode. If the voltage is not enough to power the system, it enters in Off state. This solution uses a 160μF capacitor to store the energy harvested by the EH source. The maximum distance between the Wi-Fi source and the RF harvesting, to provide enough energy to operate is 11.5cm. This implies that the advantages of having a batteryless system, are lost by the need of having a battery-powered device (smartphone) at a short distance. Another RFID-based system was proposed by Sabina Manzari *et al.* [101], who presented a chemical sensor array for batteryless ambient sensing. However, these *task-based* systems powered by RF signals, may not be active at the time the parameters have to be sensed. For example, the WISPCam was used to monitor a pressure gauge. Nevertheless, the system could not be working (due to the absence of RF signals to be harvested) at the time a pressure change occurs.

H. Lee *et al.* [6] proposed a converter-less system powered from a photovoltaic cell (an extension of this work was presented in [102]). This design removes the long-term energy storage and voltage converters in order to reduce the cost of storing and converting energy. It just adds a bulk capacitor to buffer energy and a maximum power point tracking (MPPT) to scavenge the maximum possible energy from the EH source. A duty-controlled power management unit (PMU) was designed, which demonstrated an enhancement of almost 60%. This PMU has two threshold voltages, V_U=2.70V and V_L=2.65V. When the voltage, supplied by the photovoltaic cell (PV), reaches the value of V_U, the PMU turns on the power switch and enables the circuitry to sense. If the input voltage drops below V_L, the PMU turns off the switch and the digital circuit stops working. The measurement process was controlled by a complex programmable logic device (CPLD).

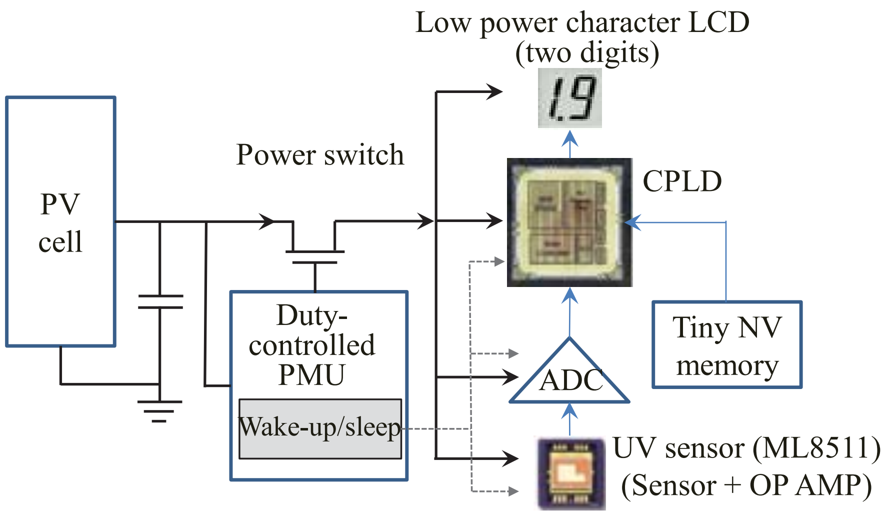


Figure 2.8: Block diagram of the SmartPatch. Extracted from [6].

The hardware solution was applied to an ultraviolet (UV) sensor (Figure 2.8), which could operate about 9 hours per day (the complete design is named *SmartPatch*). The aim of the sensor was to accumulate UV exposure over time and notifies (through a 2-digit LCD) when to apply UV protection lotion again. The tiny memory is used to retain the data that is being presented on the LCD. However, authors claimed a solution for IoT devices but their case study does not have any radio transceiver. Moreover, it is not specified whether they reached any improvement in the PV size by including their proposal.

Z. Luo *et al.* [77] proposed an application for 80-layer ferroelectret insole. The system consists on connecting two capacitors of 4.7μF each, and using a voltage detector which activates a Zigbee transmitter when the voltage reaches 5.7V. The transceiver sends a message every 3 or 4 steps and a LED turns on when the message is sent. This application is just to demonstrate that the insole is able to power a system. However, it does not sense or process any useful information and the initialization of the transceiver was performed by powering the system with a battery.

An Energy-Harvesting Energy Meter named Monjolo was proposed by S. DeBruin *et al.* [103]. This design calculates the energy consumed in domestic electrical installations by using the energy-harvesting power supply such as sensor. This allows the system to operate without high-voltage AC-DC power supply and AC metering circuitry. The design consists of an LTC3588-based energy harvester, an Epic mote, an RC time keeping circuit, a 1:3000 turns-ratio current transformer configuration, switches to select between input capacitances of 200μF, 300μF, and 500μF, and output capacitances of 47μF, 200μF, and 247μF. It also includes a half/full-wave rectification, a regulator output voltage, a microcontroller (TI MSP430F1611), an IEEE 802.15.4 compatible radio

(TI CC2420) and an FRAM (Ramtron LM25L04B). When the sensor starts harvesting energy, the voltage in the capacitor (V_{store}) is increased until it reaches 5.1 V. Then, a regulator is enabled to reduce the voltage to 3.3 V and supply it on the V_{out} rail in order to enable the microcontroller and the transceiver. The MCU reads and increment the wakeup counter, which is stored in FRAM and samples the V_{timer} line to determine whether sufficient time has passed since the previous transmission. Therefore, the load power is estimated by measuring the interval among activations, considering the system consumes the same amount of energy at each activation.

Another *task-based* monitoring system was proposed by P. Martin *et al.* [104]. This approach, named Doubledip, aims to monitor water flow, powered by a thermoelectric generator (TEG). This TEG is also used to wake up the system by detecting the small voltage change caused when a water flow begins. However, they incorporate a rechargeable battery in order to power the system during long zero-power intervals. A similar event-detector approach, named Trinity, was proposed by T. Xiang *et al.* [105]. This design uses an energy harvesting piezoelectric source, which not only powers the system but also acts as a sensor to measures the airflow. However, similarly to Doubledip, this approach incorporates DC-DC converters and batteries to sustain their operation, which increase the size, cost and weight of solutions.

A. Gomez *et al.* [37] proposed a design that dynamically adapts the voltage level to the load's requirements. It uses a booster *TI bq2505*, to build the named Energy Management Unit (EMU). This chip is used to decouple the load from the source and convert the input voltage into a desired level, depending on the necessities of each component. Defining the minimum voltage required by each unit is done by the Dynamic Energy Burst Scaling (DEBS) unit. In this way, each component will be enabled with its minimum voltage required, reducing the energy consumption, instead of using the same voltage level for the whole system.

The proposed design was tested with a low power image acquisition application. The chosen microcontroller was a TIMSP430FR5969 and a Centeye Stonyman image sensor. The camera is enabled with 3V, consuming 3.77mW. The taken image is stored in the FRAM. Then, the microcontroller is enabled with 2V, consuming 2.47mW, executing a basic image processing task, which was not defined. The system was powered by a flexible solar panel which charges a capacitor of 80μF, but the maximum voltage at which it is charged, was not specified.

The DEBS unit was also included in an energy-driven wearable vision sensing powered by a solar panel and a 150μF capacitor [106]. This *task-based* approach estimates the walking speed of users through a vision sensor attached to glasses and an algorithm that interprets the measurements. The system adapts the sampling rate as a function of the amount of energy available. Nevertheless, the system can accurately calculate the walking speed exclusively when the user keeps their eyesight constantly at a certain angle, which is not a typical posture when walking. Increasing the energy storage, however, may cause delays to enable system for the time needed to charge the capacitor until an operating level, when operating from low-power EH sources (sources that generates less electrical energy than the system consumes).

In order to reduce the start-up time, Josiah Hester *et al.* [9] presented an approach that includes a specific capacitor for each element of the design (i.e. sensor, microcontroller, transceiver) in order to reduce the cold-start time caused by a single big storage. A similar approach was presented by B. Munir *et al.* [107], who proposed an RF EH sensor system with two capacitors of different sizes. The "small" capacitor is used for a fast boot and to execute small tasks such as sensing, while the "big" one is used to establish the communication with a base station.

Nevertheless, including various capacitors contrasts with the idea of reducing the energy storage elements. Moreover, these solutions typically incorporate extra circuitry such as maximum power point tracking (MPPT) or boost converter circuits, which increase the cost of systems. Besides that, the size of the capacitor (or capacitors), would be increased with the energy requirements of the longest atomic task. The greater the task to be executed, the larger the size of the capacitor as well as the required voltage. In addition to these drawbacks, if a power failure occurs before the system finishes its task, the execution would be restarted from the beginning in the next power cycle.

2.2.2 System State Retention Schemes

In order to protect the system from volatility, various researchers have proposed *system state retention* schemes, which are application agnostic and that aim to remove the energy storage and operate directly from the energy harvester. The computation is progressively completed, through various power cycles by saving the system state (checkpoint) into a non-volatile memory before a power failure occurs, and restoring the state, when the energy is again available.

The concept of checkpoint has been used in large-scale computing for decades to provide robustness against errors or hardware failure. However, there is not a unified definition used by all authors. Bernstein et al. [108] defines *checkpointing* as "*an activity that writes information to stable storage during normal operation in order to reduce the amount of work that Restart has to do after a failure*". The processing transactions are periodically interrupted to allow the checkpointing function took place. Authors call *checkpoints* as the action of checkpointing and *Checkpoint* as the routine that performs the checkpointing activity.

Plank [109] defines *checkpointing* as "*the act of saving the state of a running program so that, it may be reconstructed later in time*". This author implemented checkpointing in uniprocessor and parallel processing systems but in different layers, and according with the layer, the checkpointing activity is performed by a unit, e.g. *User-level checkpointing* is performed by the program itself. The aim of checkpointing in this work is for fault-tolerance. The system saves periodically checkpoints (which is defined as a program state saved on a stable storage, such as magnetic disk). When a failure occurs, the application restores the state from the most recent checkpoint, losing at most an interval of computation. This definition is very close to that used some years later in Mementos, a transient computing approach [33]. Here, a checkpoint is a copy of the current state of the program under execution, saved into a non-volatile memory, and checkpointing is the act of saving a checkpoint.

A different concept of *checkpoint* and *checkpointing* was introduced by Domenico Balsamo in another transient computing method called Hibernus [3]. *Checkpoint* is defined as a routine that checks the voltage supply level in order to decide whether a snapshot should be taken or not. A *snapshot* is described as the activity of saving the program state (which includes RAM, on-board peripherals and processor registers) into NVM memory. Finally, *checkpointing* is the act of executing the checkpoint routine.

Below it is described a unified definition of Checkpoint that will be applied along this research work, avoiding possible confusions or misunderstandings.

- `Checkpoint`. Current program state saved into NVM before a power loss occurs.
- `Checkpointing`. The act of saving a checkpoint into NVM.
- `Hibernate`. The act of entering in a low-power mode.

Some checkpoint-based approaches attempt to protect computation from volatility by inserting trigger points in the system code, during application development or at compile time. These trigger points check the input voltage to decide whether a checkpoint has to be saved or not. To avoid saving periodic checkpoints, some researchers have proposed approaches that dynamically adapt trigger points in the code, based on the energy conditions. Others monitor the input voltage and save a checkpoint when level drops below a defined level. Next, a survey of existing *system state retention* approaches is presented.

Based on Static Trigger Points

Mementos [33] is one of the first approaches to retain system state in RFID-powered sensors. This design inserts trigger points in the main program at a compile time. A *trigger point* is a function call in charge of comparing the voltage supply against a predefined threshold to predict a power failure and save a checkpoint into NVM if necessary. However, these routines consume power resources, therefore, it is important not to insert too many trigger points whose cost exceeds over the program execution.

Mementos uses three different heuristics to insert trigger points and verify the input voltage level.

1. **Loop-latch mode**. A trigger point is inserted for each loop of the program in order to check the input voltage level at each iteration.
2. **Function-return mode**. The trigger points are inserted after each function in order to check the input voltage level when the program returns from a function call.
3. **Timer-aided mode**. This heuristic works in conjunction with the two previous heuristics. Here, Mementos inserts a timer interrupt that sets a flag at predefined execution intervals. At the trigger points, the voltage level is checked only if the flag is set. Thus, the system reduces the number of checkpoints, which allows to save energy.

In order to predict a possible power failure, Mementos compares the input voltage against a threshold by using an ADC. Above this threshold (V_{thresh}), it is assumed that the system can continue working without saving a checkpoint. On the other side, if the input voltage level is below the threshold, it is interpreted as a power failure is imminent and in consequence, Mementos starts saving a checkpoint into Flash memory. When Mementos saves a checkpoint, it places a number at the end of the block indicating that a complete checkpoint was stored. After a power failure, Mementos looks for that number in the memory block that was last saved. If it is not detected, the memory segment is marked to be deleted immediately after booting, when the input voltage level is over the threshold. This action has to be done because to write in a Flash segment that was previously used (Flash memory can only be written in segments), it is needed first to delete its content. Then, Mementos looks for a valid checkpoint in the other memory block. If valid, it is restored (copies its content into RAM) and continues the program execution. If not, the program is restarted from scratch.

In a resource-constrained system, it is important to consider the time that the approach needs to complete a task when powered by an intermittent source, i.e. including not only the time spent by the CPU to execute a function but also the time required to check the input voltage, save a checkpoint, etc. The total time spent by Mementos (T_{Mementos}) is given by the following equation [3]:

$$\underbrace{T_{\text{Mementos}}}_{\text{Total execution}} = \overbrace{t_a}^{\text{Algorithm}} + \underbrace{n_\iota}_{\text{No. interruptions}}(\overbrace{t_R}^{\text{Restore checkpoints}} + \underbrace{\frac{t_a}{2n_m}}_{\text{Backtrack}}) + \overbrace{n_m(t_m + \rho_s t_H)}^{\text{Monitoring and save checkpoint}} \tag{2.1}$$

where t_a is the time required by the CPU to execute the application program, n_ι is the number of power outages (where $V_{\text{cc}} < V_{\text{min}}$), t_R is the time required to restore the system state, n_m is the number of trigger points per complete execution of the algorithm, t_m is the time taken for an ADC reading V_{cc}, ρ_s is the proportion of trigger points resulting in a checkpoint, and t_H is the time taken to save a checkpoint into the NVM. The *backtrack* is the program portion that was executed but not saved into NVM, and therefore, it is executed again after restoring. It is not possible to know the exact time because it varies from one case to another. For that reason, an estimation was established, which is the total time to execute the algorithm divided by two times the number of trigger points per complete execution of the algorithm, e.g. if 3 trigger points were run and the CPU needed 120ms to complete an algorithm, the backtrack estimated would be of 20ms. Figure 2.9 shows the graphic representation of each parameter of Equation 2.1. The *wasted execution* is the part of the algorithm that was executed but not saved before a power failure which provokes the *backtrack* previously explained.

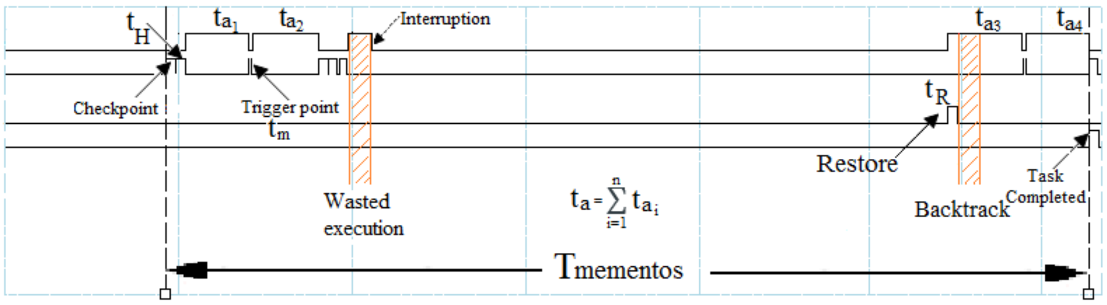


Figure 2.9: Total time taken by Mementos to execute a task.

An important drawback of Mementos is the redundant checkpoints saved when the input voltage is below V_{thresh}, as well as the Flash blocks exclusively reserved for two complete checkpoints in case a power interruption occurs whilst one is being taken. Besides that, this approach may incur in state inconsistency when using volatile and non-volatile variables.

Based on Programming Support

In order to avoid data inconsistency, B. Lucia *et al.* [34] proposed *DINO*, a programming and execution model for transient systems that avoids volatile and non-volatile data inconsistency. However, *DINO* still needs that designers to divide programs into a series of trigger points that connect tasks. A similar programming language for transient systems, which also copes with inconsistency data issues is *Chain* [110]. This approach is based on a control-flow and a channel-based abstraction for NVM, which keeps data consistency by diving the application program into atomic tasks that are completed without interruptions, and then the obtained data is passed to other tasks through channels.

A more recent programming interface, to write software that operates under intermittent conditions, is *Alpaca* [111]. In this case, the designer has to divide the application program into tasks (similarly to *Chain*). A single task is executed at a time (it does not allow parallel execution) and when one task is executed, it transfers the control to the next, in an order defined by the programmer. This system privatizes the data shared between tasks. This data is placed in non-volatile variables, which allows to avoid saving checkpoints before a power failure. Only local variables of each task are saved into volatile memory, which are clear after a power failure.

Based on Dynamic Trigger Points or Online Checkpointing

Different designs that dynamically adapt checkpoints (routine calls to save system state) in the program code are based on Low-Level Virtual Machine (LLVM) compiler [112, 113, 114], which allows to analyse application code in order to develop solutions that are transparent for final user. A recent approach based on LLVM is *Ratchet* [35], which adds trigger points between every Write-After-Read (WAR) dependency (also known as *anti-dependency*). The program is divided into *idempotent* sections connected by trigger points stored in non-volatile memory and targeting EH platforms that use the ARM architecture. The idea is to track the data that is being modified after each power failure, in order to update it in the NVM. However, *Ratchet* causes a high overhead if the EH provides enough energy to complete various iterations, because this approach needs to checkpoint in order to keep the data consistency. Another LLVM-based approach is *Chinchilla* [115]. This proposal is similar to *Ratchet* but reduces the overhead by dynamically disabling checkpoints according to energy conditions. However, this approach only presents the improvements in terms of reducing the number of saved checkpoints compared with *Ratchet*, but it does not present a performance comparison with other *system state retention* approaches that are not based on LLVM.

Z. Ghodsi *et al.* [116] proposed an online checkpointing policy based on *Progress* (PrC) and *Checkpoint* (CC) counters. PrC tracks the forward progress of the program by counting the number of elapsed intervals. The CC indicates when the last checkpoint was saved. The online checkpointing is implemented based on the *Markov decision process* (MDP), whose state is given by PrC, CC and battery level values. In order to obtain the *optimal* policy at which the overhead caused by checkpointing is minimum, an off-line Q-learning algorithm is executed. Therefore, the approach needs to be trained considering traces of energy harvesting sources, which could take too long. However, if the environmental conditions change in relation with the obtained traces, the system could malfunction because it would checkpoint more frequently than needed or not at the right point before a power failure. Moreover, this approach also wastes energy by executing portions of code that are not save into NVM (*backtrack*).

Based on Input Voltage Level

Hibernus [3] is a refinement to Mementos technique. Unlike Mementos, Hibernus is an interrupt approach, which does not insert trigger points in the main program. Instead, it uses a comparator to monitor the voltage supply level and sends an interruption when appropriate. Hibernus has two threshold references. The first threshold is to indicate that a power failure is imminent (V_H). When the level drops below V_H, the comparator interrupts the process, the system starts saving a checkpoint and hibernates. The second threshold (V_R)is to indicate that there is enough energy available to be in active mode. Therefore, if the voltage supply is again recovered and exceeds V_R, another interruption is sent by the comparator and Hibernus restores the state and continues the program execution. In Figure 2.10 it is presented the flow chart of Hibernus approach.

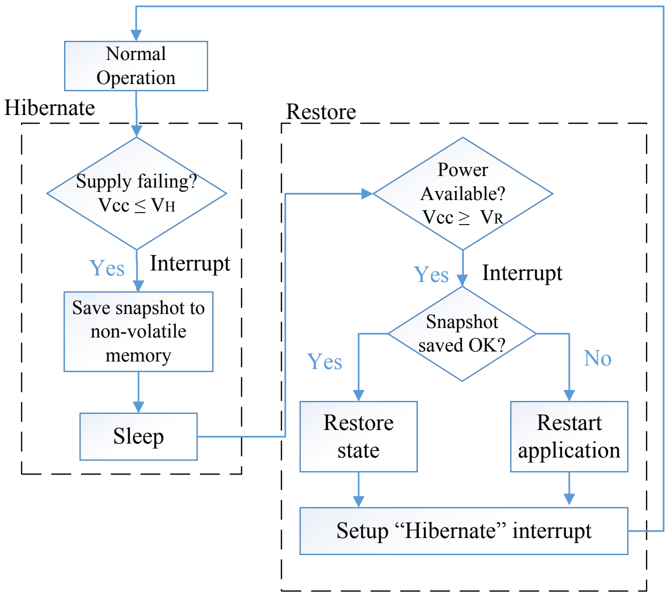


Figure 2.10: Hibernus software flow-chart. Adapted from [3].

The “hibernate” interrupt unit refers to the configuration of the comparator, setting the threshold values and the interruption. When Hibernus saves a complete checkpoint, a flag is set to indicate that a valid checkpoint can be restored when the energy is again available. If this flag is not set, Hibernus interprets that an error occurs and the application is restarted. To get the values for both thresholds, the parameters of the evaluation board were considered. Hibernus was implemented on a TI MSP430FR5739 microcontroller, which has an internal comparator that is set to monitor the voltage supply level. This method uses the FRAM as a non-volatile memory, which is faster and consumes less power than the Flash memory used in Mementos. In order to save a checkpoint, Hibernus uses the energy stored in the decoupling capacitance of the microcontroller. This allows to have a low V_H value, which increases the active period of the main program. V_H is determined considering the time required to charge the decoupling capacitor to ensure there is enough energy for saving a checkpoint before a power failure. The energy required to save a checkpoint is obtained as follows [3]:

$$E_\sigma = n_\alpha E_\alpha + n_\beta E_\beta \tag{2.2}$$

where n_α is the number of bytes of the RAM, n_β the number of bytes used by registers, E_α and E_β are the energy required to copy RAM and registers into NVM, respectively. The MCU works in a range of voltage between V_{min} and V_{max}. Given the total capacitance (C), the difference of the energy stored in the capacitor when the voltage supply is maximum (V) and minimum (V_{min}) is given by:

$$E_\delta = C\frac{V^2 - V_{min}^2}{2} \tag{2.3}$$

The total capacitance of the MCU [91] is 16μF, and the size in bytes of the RAM and registers is 1024 and 512 bytes, respectively. Copying RAM content (E_α) has an energy cost of 4.2nJ per byte, and 2.7nJ/byte in the case of registers (E_β). Substituting these values in (2.2) it is obtained that saving a checkpoint consumes 5.7 μJ (E_σ). To save a complete checkpoint requires that $E_\sigma \leq E_\delta$, therefore it is possible to substitute E_δ by E_σ in 2.3. The minimum voltage required by the microcontroller to operate is V_{min}=1.9V. Therefore, V will be the threshold (V_H) at which the capacitor has stored enough energy to save a checkpoint.

$$V_H = \sqrt{\frac{2 \cdot E_\delta}{C} + V_{min}^2} \tag{2.4}$$

Substituting the values in 2.4, the threshold value obtained is *2.17V*. In order to add hysteresis, V_R was set higher (*2.27V*) to allow V_{cc} to be over V_H. Hibernus saves one checkpoint per power interruption, therefore it is needed to reserve only one memory block to save the checkpoint, unlike Mementos, which reserves two memory blocks and may save repeated checkpoints, when input voltage is below V$_{thresh}$. The time and energy overheads caused by Hibernus are smaller compared with Mementos approach, because it is based on interrupts (not a polling approach). In Section 2.3, this approach is quantitatively evaluated in order to determine the magnitude of its performance. The total execution time for Hibernus (T_{Hibernus}) can be calculated as follows [3]:

$$\underbrace{T_{\text{Hibernus}}}_{\text{Total execution}} = \overbrace{t_a}^{\text{Algorithm}} + \underbrace{n_\iota}_{\text{No. interruptions}} (\overbrace{t_H}^{\text{Save checkpoint}} + \underbrace{t_R}_{\text{Restore checkpoint}} + \overbrace{t_\lambda}^{\text{Sleep}}) \tag{2.5}$$

where t_a is the CPU time required to execute the algorithm, n_ι is the number of system interruptions per algorithm execution (where $V_{cc} < V_{\text{H}}$). V_{H} is the threshold voltage defined to start checkpointing. t_H is the time required to save a checkpoint to NVM,

t_R is the time required to restore from NVM memory, and $\overline{t_\lambda}$ is the average time spent sleeping (after a checkpoint has been saved but before $V_{\mathrm{cc}} < V_{\mathrm{min}}$), and on power-up (when $V_{\mathrm{min}} < V_{\mathrm{cc}} < V_{\mathrm{R}}$). The absolute limit of supply interruption frequency, f_ι, where both methods can properly operate is $1/(t_H + t_R)$. This means the active time (when $V_{\mathrm{cc}} > V_{\mathrm{min}}$) should be enough to save and restore a checkpoint, on the contrary, the system is indefinitely restarted from the beginning.

Another checkpoint-based approach is QuickRecall [7], which is similar to Hibernus but designed to utilize the FRAM as a NVM and as a RAM, enabling the design to work as a “unified memory system”. Figure 2.11 shows the flow chart and the Interruption Service Routine of QuickRecall. It uses an external comparator which is connected to the GPIO pins of the MCU, configured with a trigger voltage (V_{trig}) and sends a signal output when the input voltage level (V_{cc}) drops below V_{trig}. This approach has just one threshold voltage, because it starts working as soon as the voltage supply reaches the minimum level required by the board to operate. V_{trig} is not required to be relative high, unlike Mementos, because QuickRecall only needs to back up the program counter, stack pointer, status register and general-purpose registers (GPR) before a power failure occurs. Thus, author determined a V_{trig} of 2.0003V for a correct operation (the minimum voltage required by the MSP430 microcontroller is 2V).

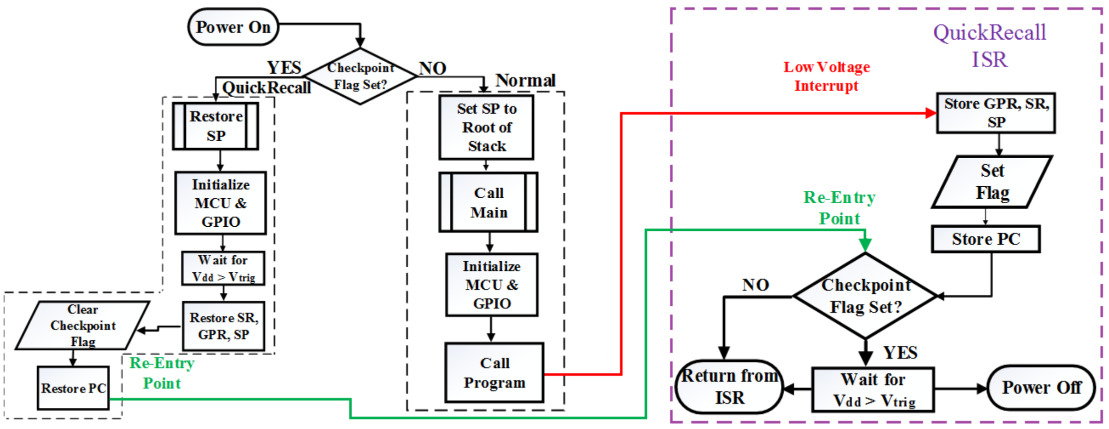


Figure 2.11: QuickRecall flow-chart, including the Interruption Service Routine. Adapted from [7].

QuickRecall does not need to restore the program state because, as mentioned before, it is using a FRAM instead of RAM, therefore the current state is always into the NVM. However, using the FRAM as unified memory could have some disadvantages such as the approach cannot operate in a processor without a unified memory, a FRAM consumes more energy and is slower than a RAM. Therefore, in situation where the system has a stable power supply, the energy and time overhead can be considerably high.

The total execution time for QuickRecall ($T_{\mathrm{QuickRecall}}$) can be calculated with the same equation (2.5) than that used for Hibernus. This is because both approaches save a single checkpoint per power failure. The difference is in the values of the parameters, considering that QuickRecall operates with the FRAM as unified memory.

Another checkpointing-based approach is Hibernus++ [8], which is an adaptive version of Hibernus technique. The objective of this approach is to maximize the active time of a transiently-powered system in order to perform as many operations as possible without the necessity of adding energy storage. Hibernus++ dynamically adjusts the thresholds for Hibernate and Restore routines, in response to the system power consumption, the on-board decoupling capacitance and energy provided by the EH source, allowing to save, similarly to Hibernus, a single checkpoint per power failure.

Figure 2.12 shows the flow chart of Hibernus++ algorithm. When the input voltage rises above the minimum operating level required by the MCU to operate (V_{min}), the algorithm checks whether the system is calibrated; if not, Hibernus++ executes the calibration routine in order to get the threshold value for hibernation (V_H). The algorithm obtains this value by evaluating the rate of voltage drop in the case of an abrupt loss of energy. After finishing this routine, the algorithm verifies the voltage supply in order to define whether there is enough energy to continue its operation. If not, the system hibernates until the input voltage reaches a sustainable value. If a valid checkpoint was previously stored into NVM, the system state is restored. If not, the system is restarted from the beginning. In the case that the voltage supply recovers without dropping below the minimum voltage, the system resumes operation without restoring its state.

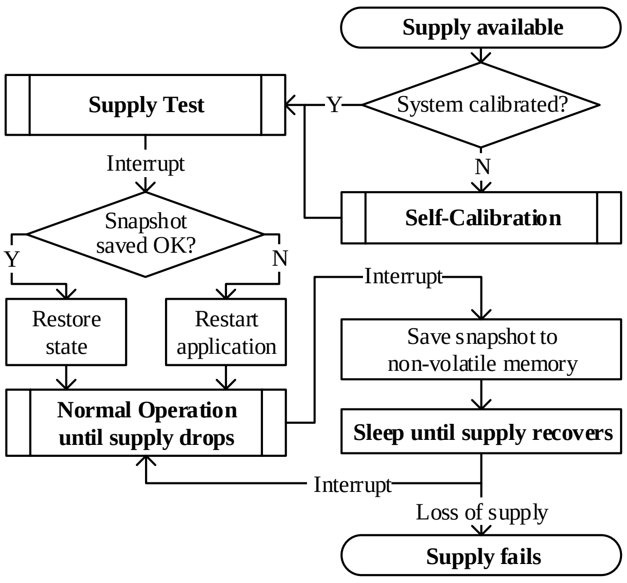


Figure 2.12: Hibernus++ software flow-chart. Extracted from [8].

The calibration routine waits for the input voltage to reach the calibration start level (V_{cal}). Once this level is reached, the EH source is short-circuited and a complete checkpoint is saved to NVM. The drop in input voltage due to chekpointing is given by the difference: $V_{cal} - V_{meas}$, where V_{meas} is the voltage measured at the end of the checkpointing process. To ensure the MCU has enough time for checkpointing before the input voltage drops below V_{min}, the hibernation threshold is set as follows [8]:

$$V_H = V_{min} + (V_{cal} - V_{meas}) \tag{2.6}$$

V_{cal} is initially set to the minimum voltage required by the MCU to operate. If the calibration routine fails, V_{cal} is increased 10%, trying to get the lowest value of V_H. After calculating the value of V_H, the next step is to calculate the value for restoring the system. First, Hibernus++ classifies the voltage supply as either *High-power* or *low-power*. In the first case, the source is able to sustain the operation of the microcontroller in active mode. In the second case, the EH source cannot supply enough power to enable the microcontroller.

If the EH source is classified as "high-power", the system is restored immediately, if not, it is executed a stable voltage detection routine, where the system hibernates until the input voltage reaches a stable level. This routine has two separated interrupts, one to detect increasing voltage and the other to act as a time-out. If the input voltage is increasing, the algorithm resets the timer. In the case that the input voltage stops increasing, the timer interrupts the process indicating that the system has to be restored due to the capacitor is not being charged anymore. Hibernus++ saves only one checkpoint per power failure (when $V_{cc} < V_H$).

Once these approaches were described, it is important to compare their performance under different conditions. In the following section, a mathematical analysis of Hibernus and QuickRecall is presented. This section also describes the experimental set-up and the different executed tests, as well as the obtained results are critically compared.

2.3 A Quantitative Evaluation of System State Retention Approaches

In order to compare the performance of prominent *system state retention* approaches, a mathematical analysis was developed and a practical evaluation was performed in a common platform, powered by different sources.

2.3.1 Mathematical Description of Hibernus and QuickRecall

Although Hibernus (HB) and QuickRecall (QR) are checkpointing-based approaches, which use the FRAM as NVM, they are affected in different manner by multiple energy interruptions. In situations where the input power is constantly interrupted, the energy consumed while checkpointing could become more significant than that consumed in active mode. This scenario is where Hibernus could consume more energy than QuickRecall. Therefore, both methods were mathematically analysed in terms of energy interruption frequency (f), which means that the system is interrupted f times per second.

The total energy consumed by Hibernus and QuickRecall, to execute a task is given by the following:

$$E_{total} = t_{LP}P_{LP} + t_R P_R + t_A P_A + t_H P_H \tag{2.7}$$

where t_{LP} and P_{LP} are the time and power spent in low power mode, t_R and P_R are the time and power needed to restore the state, t_A and P_A are the time and power spent performing an algorithm, t_H and P_H are the time and power spent by the MCU to save a checkpoint, respectively. If Formula 2.7 is rephrased to be applied separately to Hibernus and QuickRecall, it is obtained:

$$E_{HB} = t_{HLP}P_{HLP} + E_H + t_{HA}P_{HA} \tag{2.8}$$

$$E_{QR} = t_{QLP}P_{QLP} + E_Q + t_{QA}P_{QA} \tag{2.9}$$

where E_{HB} is the total energy consumed by Hibernus, E_{QR} is the energy consumed by QuickRecall, E_H and E_Q are the energy consumed to save and restore a checkpoint by Hibernus and QuickRecall, respectively. The other elements of the formulas were already explained in Formula 2.7. The subscript H was added for Hibernus parameters and Q for parameters of QuickRecall.

As shown in Figure 2.13, Hibernus and QuickRecall manage two thresholds, one to restore the state (V_R) and the other to checkpoint (V_H). V_{min} is minimum voltage required by the microcontroller to operate. The MCU will be in low power mode (LPM) when the input voltage is higher than V_{min} but lower than V_R (t_{LP1}), or when the voltage level drops below V_H but it is still higher than V_{min} (t_{LP2}). From Figure 2.13, t_{LP1} and t_{LP2} can be described as:

$$t_{LP1} = \frac{sin^{-1}\left(\frac{V_R}{V_{max}}\right) - sin^{-1}\left(\frac{V_{min}}{V_{max}}\right)}{2\pi f_{source}} \tag{2.10}$$

$$t_{LP2} = \frac{C(V_H - V_{min})}{I_{LP}} \tag{2.11}$$

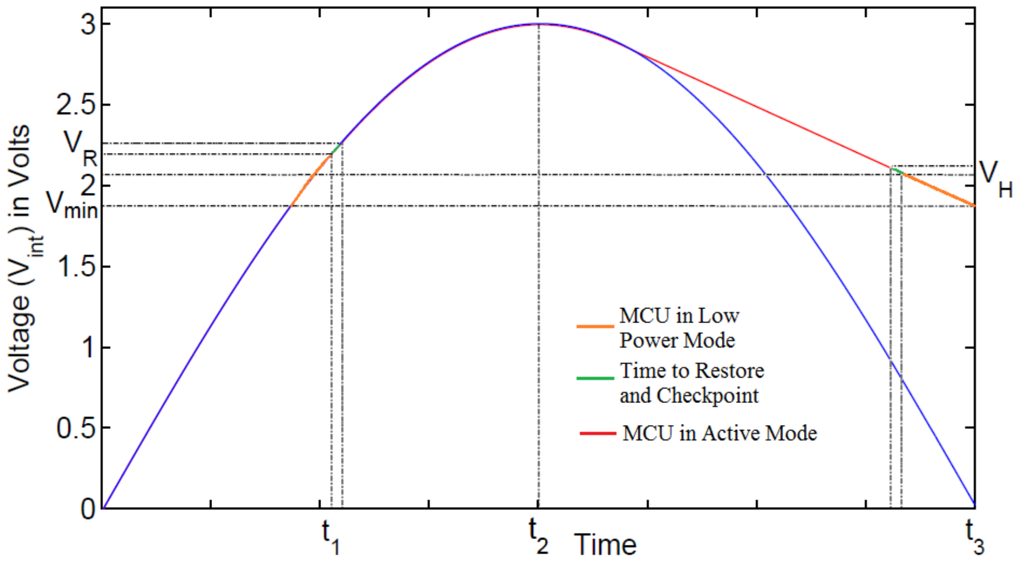


Figure 2.13: Behaviour of Hibernus and QuickRecall when powered by a sinusoidal signal.

where f_{source} is the frequency of the input voltage, C is the decoupling capacitance of the MCU and I_{LP} is the current consumed by the MCU in LPM. The time between the Restore threshold and the maximum voltage level (from t_1 to t_2 in Figure 2.13) is given by the following formula:

$$t_{rise} = \frac{1}{4f_{source}} - \frac{sin^{-1}\left(\frac{V_R}{V_{max}}\right)}{2\pi f_{source}} \tag{2.12}$$

In the case of the discharge time of the decoupling capacitance, it can be calculated as follows:

$$t_{discharge} = \frac{C(V_{max} - V_H)}{I_o} \tag{2.13}$$

where I_o is the current consumed by the system when executing the algorithm. The active time (t_A) in a single period of an input voltage signal is given by:

$$t_A = \frac{1}{4f_{source}} - \frac{sin^{-1}\left(\frac{V_R}{V_{max}}\right)}{2\pi f_{source}} + \frac{C(V_{max} - V_H)}{I_o} - t_R \tag{2.14}$$

Formula 2.14 is similar for both approaches, only the values of the parameters vary from one method to the other. Substituting t_A in Formula 2.8 for Hibernus (t_{HA}) and 2.9 for QuickRecall (t_{QA}), the following equations are obtained:

$$E_{HB} = t_{HLP}P_{HLP} + E_H + P_{HA}\left(\frac{1}{4f_{source}} - \frac{sin^{-1}\left(\frac{V_{HR}}{V_{max}}\right)}{2\pi f_{source}} + \frac{C(V_{max} - V_{HH})}{I_{HO}} - t_{HR}\right) \tag{2.15}$$

$$E_{QR} = t_{QLP}P_{QLP} + E_Q + P_{QA}\left(\frac{1}{4f_{source}} - \frac{sin^{-1}\left(\frac{V_{QR}}{V_{max}}\right)}{2\pi f_{source}} + \frac{C(V_{max} - V_{QH})}{I_{QO}} - t_{QR}\right) \tag{2.16}$$

The elements of both formulas were previously described. The difference is the addition of the subscript H for parameters of Hibernus and Q for parameters of QuickRecall. The sum of t_{LP1} and t_{LP2} represents the total time the system stays in low power mode, which is t_{HLP} for Hibernus and t_{QLP} for QuickRecall. Substituting those values in Formula 2.15 for Hibernus and 2.16 for QuickRecall, the formulas for the total energy consumption can be defined as follows:

$$\begin{split} E_{HB} = E_H + P_{HLP}\left(\frac{sin^{-1}\left(\frac{V_{HR}}{V_{max}}\right) - sin^{-1}\left(\frac{V_{min}}{V_{max}}\right)}{2\pi f_{source}} + \frac{C(V_{HH} - V_{min})}{I_{HLP}}\right) \\ +P_{HA}\left(\frac{\pi - 2sin^{-1}\left(\frac{V_{HR}}{V_{max}}\right)}{4\pi f_{source}} + \frac{C(V_{max} - V_{HH})}{I_{HO}} - t_{HR}\right) \end{split} \tag{2.17}$$

$$\begin{split} E_{QR} = E_Q + P_{QLP}\left(\frac{sin^{-1}\left(\frac{V_{QR}}{V_{max}}\right) - sin^{-1}\left(\frac{V_{min}}{V_{max}}\right)}{2\pi f_{source}} + \frac{C(V_{QH} - V_{min})}{I_{QLP}}\right) \\ +P_{QA}\left(\frac{\pi - 2sin^{-1}\left(\frac{V_{QR}}{V_{max}}\right)}{4\pi f_{source}} + \frac{C(V_{max} - V_{QH})}{I_{QO}} - t_{QR}\right) \end{split} \tag{2.18}$$

As mentioned before, the aim of this research is to verify whether at different energy interruption frequencies, one technique outperforms the other. Therefore, it is looked for a frequency where the total energy consumed by Hibernus is bigger than that consumed by QuickRecall, i.e. $E_{HB} > E_{QR}$. Applying this criteria to formulas 2.17 and 2.18, and solving for f_{source}, it is possible to get the interruption frequency at which Hibernus consumes more energy than QuickRecall. Due to the size of formula, it was separated in two parts, *Power* and *Energy*. Then, these two elements are evaluated to get the interruption frequency in Formula 2.21.

$$\begin{aligned}Power = \pi(P_{QA} - P_{HA}) + 2\Bigg(& P_{HLP} sin^{-1}\Big(\frac{V_{min}}{V_{max}}\Big) - P_{HLP} sin^{-1}\Big(\frac{V_{HR}}{V_{max}}\Big) \\ & + P_{HA} sin^{-1}\Big(\frac{V_{HR}}{V_{max}}\Big) - P_{QLP} sin^{-1}\Big(\frac{V_{min}}{V_{max}}\Big) + P_{QLP} sin^{-1}\Big(\frac{V_{QR}}{V_{max}}\Big) \\ & - P_{QA} sin^{-1}\Big(\frac{V_{QR}}{V_{max}}\Big)\Bigg)\end{aligned} \tag{2.19}$$

$$\begin{aligned}Energy = E_H - E_Q + P_{QA}t_{QR} - P_{HA}t_{HR} - \frac{I_{QO}P_{QLP}C(V_{QH} - V_{min})}{I_{QLP}I_{QO}} \\ + \frac{I_{QLP}P_{QA}C(V_{max} - V_{QH})}{I_{QLP}I_{QO}} + \frac{I_{HO}P_{HLP}C(V_{HH} - V_{min})}{I_{HLP}I_{HO}} \\ + \frac{I_{HLP}P_{HA}C(V_{max} - V_{HH})}{I_{HLP}I_{HO}}\end{aligned} \tag{2.20}$$

$$f_{source} > \frac{Power}{Energy} \tag{2.21}$$

Table 2.2: Electrical values of each parameter.

Parameter	**Value**	**Description**
C	20μF	Decoupling Capacitance
E_H	6.35μJ	Energy consumed by HB when saving and restoring a Checkpoint
E_Q	2.75μJ	Energy consumed by QR when saving and restoring a Checkpoint
I_{HO}	1mA	Current consumed by HB in Active Mode
I_{QO}	1.2mA	Current consumed by QR in Active Mode
$I_{HLP,QLP}$	0.0122mA	Current consumed by HB and QR in Low Power Mode
P_{HLP}	25.9μW	Power required by HB in Low Power Mode
P_{QLP}	24.8μW	Power required by QR in Low Power Mode
P_{HA}	2.8mW	Power required by HB in Active Mode
P_{QA}	3.36mW	Power required by QR in Low Power Mode
t_{HR}	1.40ms	Time taken by HB when Restoring a Checkpoint
t_{QR}	0.71ms	Time taken by QR when Restoring a Checkpoint
V_{max}	3V	Maximum Voltage of the input signal
V_{min}	1.98V	Minimum voltage required by MCU to operate
V_{HH}	2.17V	Voltage threshold of HB to start Checkpointing
V_{HR}	2.27V	Voltage threshold of HB to start Restoring
V_{QH}	2.003V	Voltage threshold of QR to start Checkpointing
V_{QR}	2.103V	Voltage threshold of HB to start Restoring

Table 2.2 shows the values for the parameters of power and energy were obtained by inspecting the data sheet [91] of the microcontroller used in these approaches (MSP430FR5739)

and also by experimental measurements. Substituting the values from Table 2.2 in Formulas 2.19, 2.20 and finally in 2.21, the interruption frequency can be estimated. Formula 2.22 shows that the calculated value for the Energy is **Negative**, which is **Not Valid**. Therefore, there is not an interruption frequency at which Hibernus consumes more energy than QuickRecall. In consequence, using FRAM as unified memory always consumes more energy than using the SRAM in active mode and the FRAM to save checkpoints before a power failure.

$$f_{source} > \frac{106.7\mu W}{-0.082\mu J} \tag{2.22}$$

2.3.2 Comparative Simulation of Hibernus and QuickRecall

In order to demonstrate the conclusions from Section 2.3.1, a script in Matlab was implemented to simulate the performance of Hibernus and QuickRecall approaches. The simulation was performed using a sine wave as input voltage with two ranges: Low Interruption Frequencies from 2 to 40Hz and High Interruption Frequencies from 60 to 340Hz.

Table 2.3: Performance of Hibernus in a single power cycle at low frequencies.

HIBERNUS: Low Frequencies (2-40Hz)				
Frequency (Hz)	**Active Time (ms)**	**Restore (ms)**	**Hibernate (ms)**	**Total Energy (μJ)**
2	72.12	1.35	1.40	220
4	43.82	1.35	1.40	140
6	34.32	1.35	1.40	110
8	29.62	1.35	1.40	100
10	26.82	1.35	1.40	90
12	24.92	1.35	1.40	80
14	23.52	1.35	1.40	80
16	22.52	1.35	1.40	80
18	21.72	1.35	1.40	70
20	21.12	1.35	1.40	70
40	18.22	1.35	1.40	70

In Table 2.3 and 2.4, the obtained results at low frequencies are listed for Hibernus and QuickRecall, respectively. The results do not include the *start-up* time (i.e., the elapsed time since the system is powered until it starts working), because it may vary from one platform to another. Therefore, only the results that do not depend on the experimental board, are listed. Here, it is possible to see that the energy consumed at 2Hz is almost twice that consumed at 4Hz. This is caused by the fact that at 2Hz, the microcontroller remains in Active mode 60% more time than at 4Hz. It is important to highlight that these values are only for a single power cycle. It is important to mention that the scenario where the power signal is not periodic, i.e., the time from one power

Table 2.4: Performance of QuickRecall in a single power cycle at low frequencies.

QUICKRECALL: Low Frequencies (2-40Hz)				
Frequency (Hz)	**Active Time (ms)**	**Restore (ms)**	**Hibernate (ms)**	**Total Energy (μJ)**
2	78.62	0.71	0.47	270
4	47.02	0.71	0.47	160
6	36.52	0.71	0.47	130
8	31.22	0.71	0.47	110
10	28.12	0.71	0.47	100
12	25.92	0.71	0.47	90
14	24.42	0.71	0.47	80
16	23.32	0.71	0.47	80
18	24.42	0.71	0.47	80
20	21.72	0.71	0.47	80
40	18.62	0.71	0.47	70

cycle to the other varies, would not affect the performance of both approaches, because they save a single checkpoint per power failure and the energy consumed by QuickRecall in active mode is always higher than that consumed by Hibernus.

QuickRecall shows to be in Active mode for longer than Hibernus because, as mentioned before, QuickRecall only checkpoints the general purpose registers, unlike Hibernus, which has to copy all registers and SRAM content into FRAM (and vice versa when restoring), increasing the time for checkpointing. In order to get a better comparative visualization of results, the energy consumed by each approach is plotted in Figure 2.14. Here, it is possible to see that as the interruption frequency increases, the difference of the energy consumption between the two approaches is smaller but Hibernus always consumes less energy than QuickRecall. For this reason, it is important to consider higher frequencies to verify whether at some frequency, there is a crossover of the plots. In Table 2.5 and 2.6, the results at higher frequencies are listed.

Unlike low frequencies, at higher frequencies the difference of time spent in LPM4 between 60Hz and 340Hz is less than 1ms. In the case of the energy consumption, the difference, at these supply frequencies, is of approximately 2μJ in Hibernus and 6μJ in QuickRecall. In Figure 2.15, the simulation results are plotted. It is possible to see that the amount of energy consumed by QuickRecall is always higher than that of Hibernus. Therefore, the conclusions of mathematical analysis are demonstrated.

2.3.3 Experimental Setup

Mementos, Hibernus, Hibernus++ and QuickRecall were implemented on the experimental board TI MSP430FR5739 microcontroller. This platform has 1KB of RAM and 16KB of FRAM. To perform the required experiments, a Signal Generator was used to provide a range of supply frequencies (2-10 Hz, and DC) to represent the intermittent

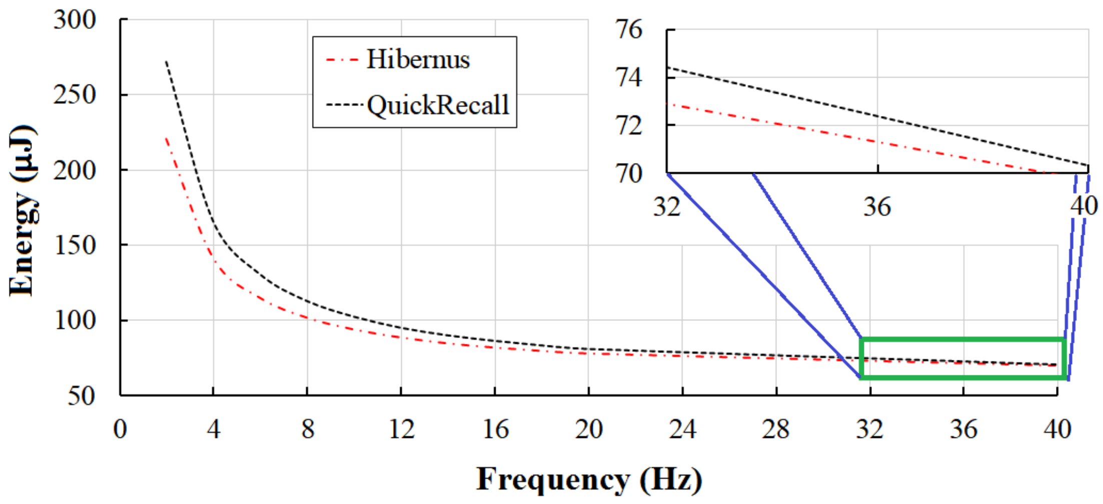


Figure 2.14: Simulation results of Hibernus and QuickRecall at low interruption frequencies (2 to 40Hz).

Table 2.5: Performance of Hibernus in a single power cycle at high frequencies.

HIBERNUS: High Frequencies (60-350Hz)				
Frequency (Hz)	**Active Time (ms)**	**Restore (ms)**	**Hibernate (ms)**	**Total Energy (μJ)**
60	17.10	1.35	1.40	66
80	16.61	1.35	1.40	65
100	16.33	1.35	1.40	64
120	29.62	1.35	1.40	63
140	16.01	1.35	1.40	63
160	15.90	1.35	1.40	63
180	15.83	1.35	1.40	63
200	15.76	1.35	1.40	62
220	15.71	1.35	1.40	62
240	15.67	1.35	1.40	62
260	15.63	1.35	1.40	62
340	15.52	1.35	1.40	62

power output that might be expected from a high-power EH. A Power Analyser was include to measure the current and power consumption, as well as a Digital-Analog signal analyser to get the time performance. Figure 2.16 shows the experimental set-up used for Hibernus, QuickRecall and Mementos. $C_{decouple}$ represents the total on-board decoupling capacitance. The internal voltage comparator of the MSP430FR platform is used for voltage comparison for Hibernus and QuickRecall approaches. In the case of Mementos, the internal ADC was used.

The evaluation test case chosen was an algorithm to calculate the Fast Fourier Transform (FFT), which analyses three arrays, each holding 128 8-bit samples of triaxial accelerometer data. The MCU clock was configured to run at 8MHz, which spends 100 ms to execute the FFT application. The system is powered with three different sources

Table 2.6: Performance of QuickRecall in a single power cycle at high frequencies.

QUICKRECALL: High Frequencies (60-350Hz)				
Frequency (Hz)	**Active Time (ms)**	**Restore (ms)**	**Hibernate (ms)**	**Total Energy (μJ)**
60	18.05	0.71	0.47	68
80	17.53	0.71	0.47	66
100	17.19	0.71	0.47	65
120	16.99	0.71	0.47	64
140	16.79	0.71	0.47	64
160	16.69	0.71	0.47	63
180	16.60	0.71	0.47	63
200	16.49	0.71	0.47	63
220	16.47	0.71	0.47	63
240	16.43	0.71	0.47	62
260	16.39	0.71	0.47	62
340	16.26	0.71	0.47	62

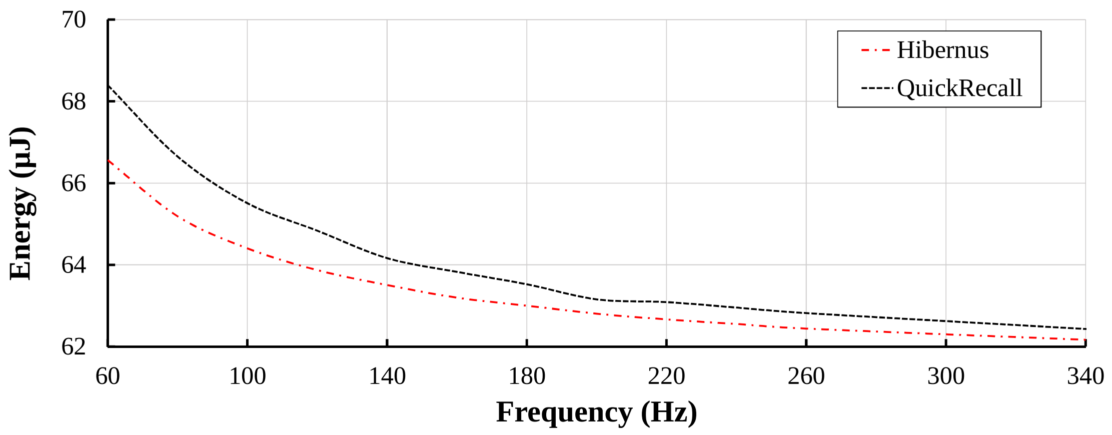


Figure 2.15: Simulation results of Hibernus and QuickRecall at high interruption frequencies (60 to 340Hz).

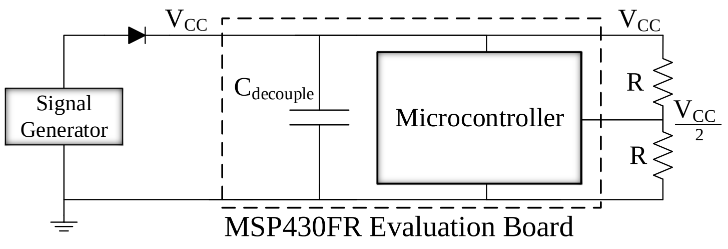


Figure 2.16: Experimental Set-up.

– a 3.4V DC, a Square and Sinusoidal sources with ±3.4V amplitude operating at frequencies ranging from 2Hz to 10Hz. In Figure 2.17, it is shown the whole set-up with the microcontroller and the required external devices.

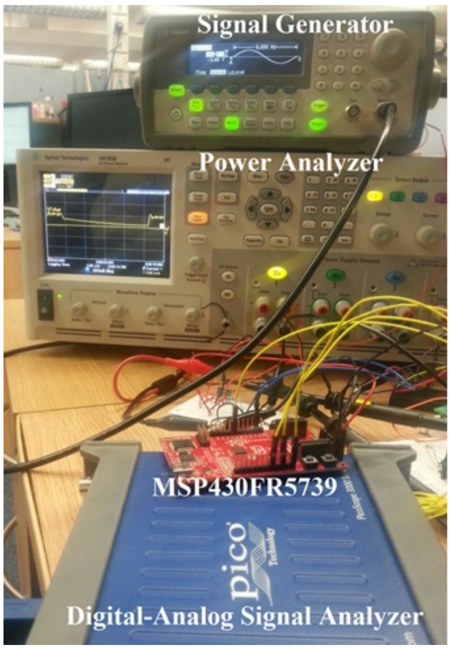


Figure 2.17: Experimental Set-up with External Components.

2.3.4 Experimental Results

The results obtained experimentally are presented including a performance comparative of the three approaches. All values were taken during the execution of one FFT application. As mentioned before, specific parameters were chosen that could have an impact on future design, because they may represent the most important factors of energy consumption and time overhead.

Checkpoint, Restore, Time Overhead and Current Consumption Comparison

In this section, it is presented the comparative results of Mementos, Hibernus and Quick-Recall, in terms of the number of checkpoints, restores routines and time overhead when performing a complete Fast Fourier Transform (FFT).

Figure 2.18 shows the number of checkpoints saved by the three approaches. Hibernus and QuickRecall saves a checkpoint every time the hibernate routine is executed, while Mementos saves a checkpoint only when $V_{cc} < V_H$. The number of checkpoints with Mementos is therefore correlated to each trigger point placement, the value of V_{min} and the supply interruption frequency. For Hibernus and QuickRecall, the amount only depends on the supply interruption frequency.

Hibernus and QuickRecall takes the same number of checkpoints (between 0 and 4). Mementos in *loop-latch* mode, at 10 Hz, takes 21 checkpoints and 28 in *function* mode. This is about 60% more than the two first approaches. Mementos in *loop-latch* mode operates unstably at frequencies of 6 and 8 Hz due to the supply is interrupted in the

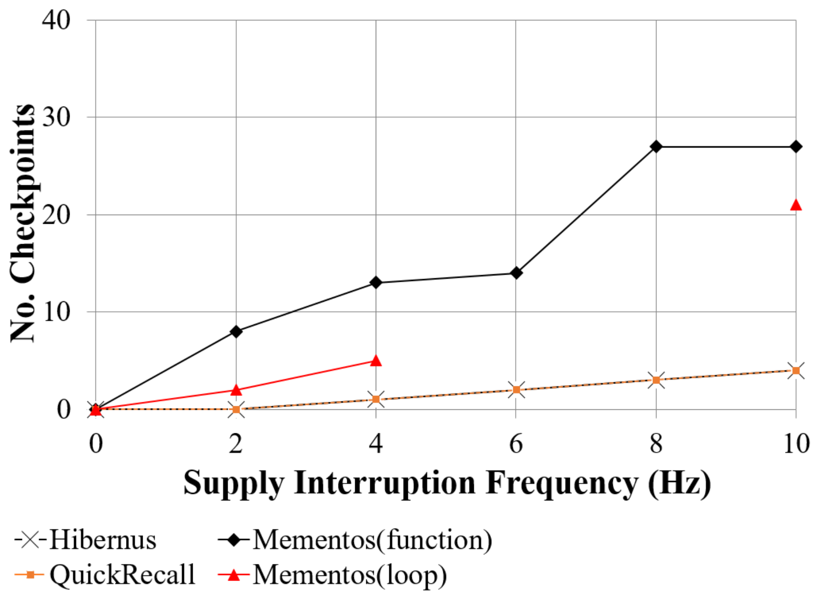


Figure 2.18: Number of checkpoints taken by each approach.

period between a restore and the next checkpoint being saved. It causes the system is not able to save a valid checkpoint before a power failure. Therefore, the approach is "stuck" restoring an old checkpoint instead of moving forward in the program.

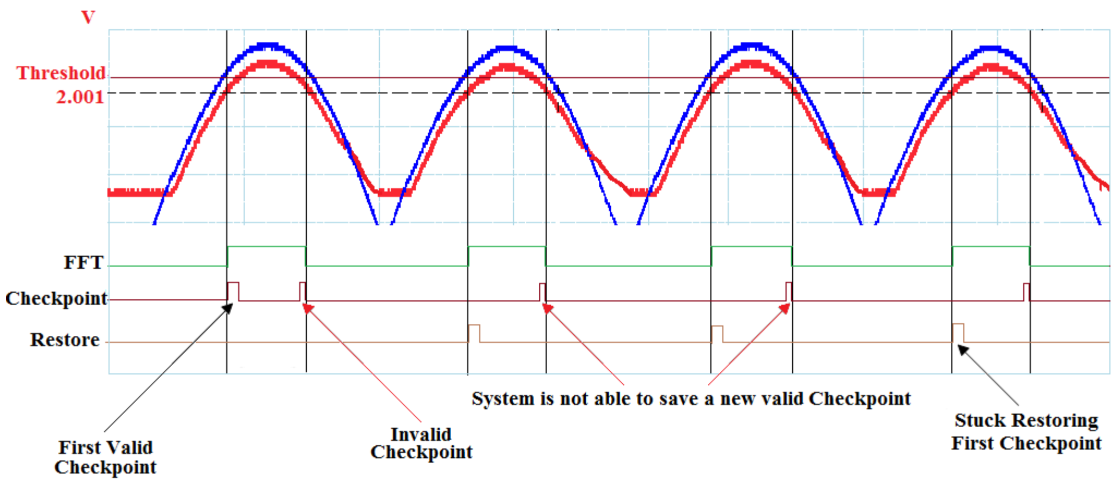


Figure 2.19: Unstable operation of Mementos in Loop-latch mode.

Figure 2.19 shows a case where Mementos is stuck at certain interruption frequencies. When the approach starts working, it saves a checkpoint in one of the two memory blocks reserved. Then, it starts saving a new checkpoint but a power failure occurs before finishing. Therefore, when the power is again available, Mementos restores the state from a valid checkpoint, which is stored in the first memory block. After restoring, the approach tries to save a new checkpoint but it is interrupted again. In consequence, Mementos is stuck restoring an "old" checkpoint because it is not able to save a new one in the second memory block with the progress of the algorithm under execution.

In the case of Restore routines, it is expected that both Hibernus and QuickRecall execute the same number of restore routines than checkpoints. Figure 2.20 shows that Hibernus and QuickRecall complete the execution of the FFT application over the same number of power interruptions, which means both approaches require restore the system the same number of times (between 0 and 4) while Mementos needs up to 7 and 8 restore routines to complete the application in *function* and *loop-latch* modes, respectively.

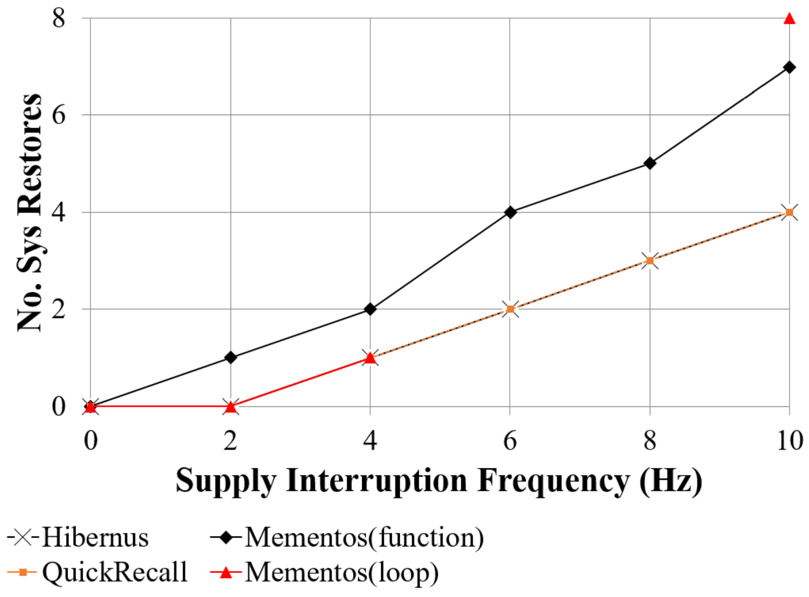


Figure 2.20: Number of restores executed by each approach.

The time overhead is the extra time spent by the approach to complete a task, e.g. if the approach spends 120 ms to complete the FFT application, the time overhead is of 20 ms. This extra time is caused by the approaches when checkpointing, restoring or sleeping. Figure 2.21 plots the time overhead of the three approaches for different interruption frequencies while executing the FFT application. The results shown in Figure 2.21 demonstrate that the time overhead of Mementos is much higher than that of Hibernus and QuickRecall. It is important to observe that as the supply interruption frequency increases, the execution time overhead of Mementos in the function mode increases rapidly, exceeding 100% overhead (2x execution time) for an interruption frequency of 10 Hz. Finally, the time overhead for QuickRecall is similar to that of Hibernus with a maximum of about 15% at 10 Hz.

The current consumption is an important factor to define which approach is more energy efficient. For this experiment, a DC signal was used in order to compare the performance of the three approaches in active mode. Figure 2.22 shows the performance comparison of the three approaches. The current peaks in the figure correspond to the time when the MCU is executing the FFT. At other times, the lower current level is when the system is not performing a task. QuickRecall consumes more current than the other two approaches. This can be a consequence of the usage of FRAM as a unified memory.

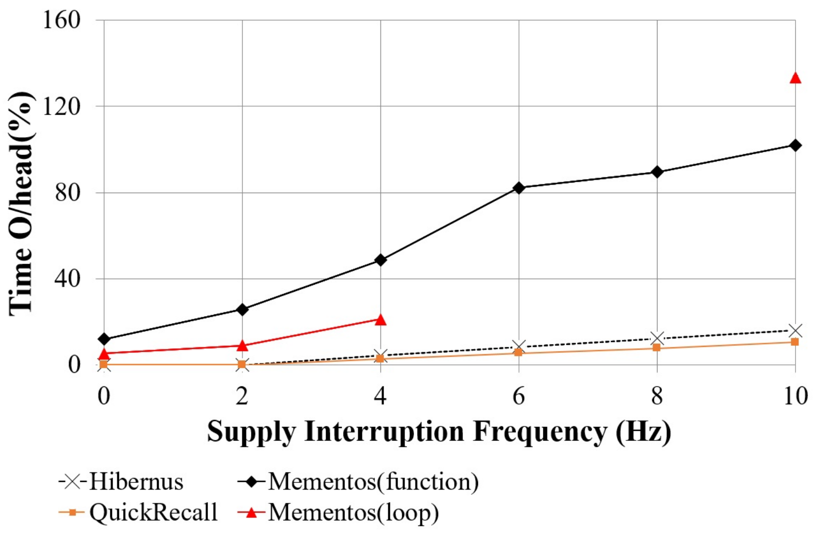


Figure 2.21: Time Overhead generated by each approach.

While working with a DC power supply, Hibernus and Mementos use the SRAM during the execution, which is faster ($< 10ms$ write speed) and consumes less current ($< 60\mu$A/MHz) than the FRAM. This is because FRAM contains ferroelectric films, which have to be polarized by an electric field from an external source, increasing the time and energy consumption compared with SRAM. However, these ferroelectric films are able to remain polarized when the electric field is removed. In the case of Mementos, the current consumption is a little smaller than in Hibernus because Mementos uses the ADC instead of using the internal comparator. The ADC is enabled and disabled in the trigger point routine meanwhile the comparator is always active during the execution, consuming more current than that consumed by using the ADC. However, it is possible to see that the high current level, while running Mementos, remains longer (caused by its time overhead) compared with the other two approaches, which has a quantitative effect in the final energy overhead.

Performance with Synthesized Sources

In this section, it is presented the measurement results of two transient computing approaches: Hibernus and Hibernus++. The aim of these experiments is to verify whether there is any improvement in the performance of Hibernus++ with respect to previous existing approach.

Four power supplies were used to power the system, creating four different scenarios. A constant current source of 200μA and three synthesized signals of three EH sources: a Photovoltaic Cell, a wearable kinetic watch (Seiko watch) and a Wind Turbine (2.23). The kinetic and wind turbine signals were classified as high-power sources whilst the micro PV cell and the constant current source were classified as low-power sources. With the first two signals, the system behaves as when powered by sinusoidal sources,

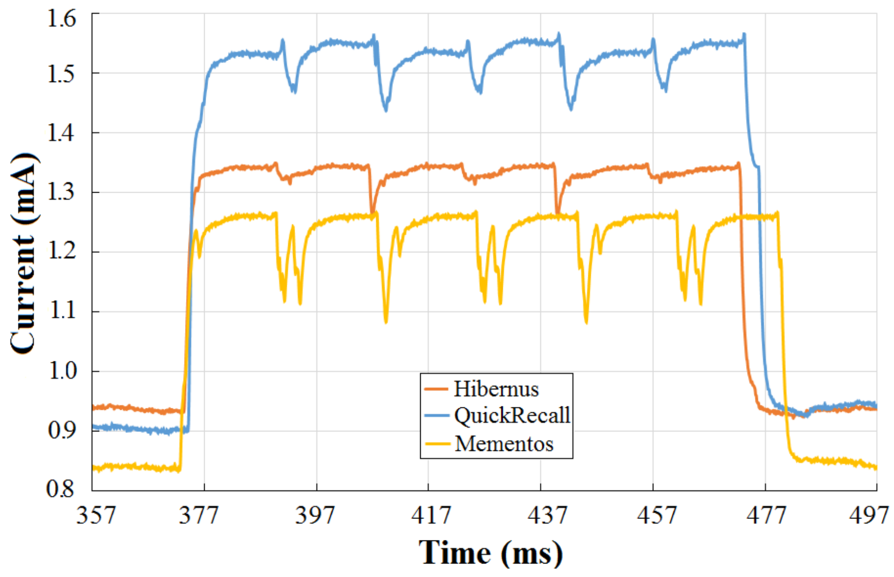


Figure 2.22: Current consumed by each approach while executing one FFT application.

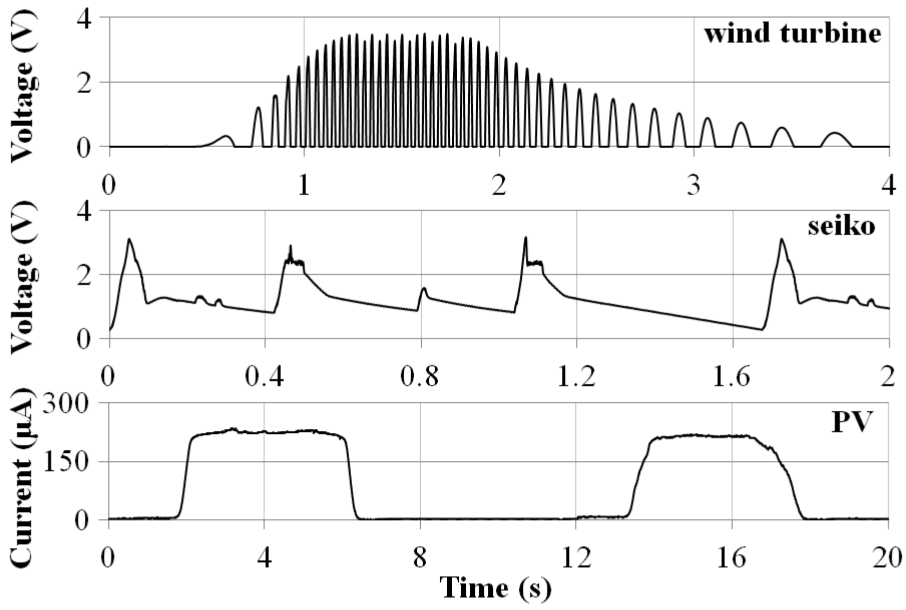


Figure 2.23: Synthesized signals of three EH sources used during the performance comparison. Extracted from [8].

while with the second ones (constant current source and micro PV) the system V_{cc} never drops below V_{min}, which means that the algorithm just classifies the source once and it never saves a checkpoint. In Table 2.7 it is shown the values obtained for the different evaluated parameters when running Hibernus++.

The evaluation test case was the same than in the previous experiments (a Fast Fourier Transform). *MCU ON* means the time the system is active and $\sum$*Test* refers to the time spent classifying the EH source. From Table 2.7, it is possible to see that the maximum time overhead is when the system is powered by the current source (448%) and in the case of energy overhead, the maximum value is presented when powered by the wind turbine signal (28%).

Table 2.7: Hibernus++ performance when powered by different synthesized sources.

Synthesized Sources	MCU ON (ms)	FFT (ms)	Low power (ms)	∑ Test (ms)	Calibration (ms)	∑ Restore (ms)	∑ Hibernate (ms)	Time O/head (%)	Energy O/head (%)
Wind Turbine	173	100	40.3	9.0	2.2	10.8	11.2	73.5	28.0
Kinetic	156	100	45.3	3.0	2.2	2.7	2.8	56.0	9.4
PV	511	100	401	1.0	2.2	0.0	7.0	411.0	13.4
Input Current 200uA	548	100	428	1.0	2.2	0.0	16.8	448.0	21.1

The next evaluation consists of comparing the performance of Hibernus and Hibernus++ in three parameters: Number of Restore routines, Number of Checkpoint routines and Total Time. The meaning of “Total Time” is the same than that used in previous experiments, i.e. the time required by the system to complete an evaluation test including time to restore and save checkpoints, to calibrate, the time spent sleeping (when $V_{min} < V_{cc} < V_H$ and $V_R > V_{cc} > V_{min}$) and the time when the input voltage is below V_{min}. Table 2.8 presents the comparison results obtained in the experiments for Hibernus and Hibernus++.

Table 2.8: Comparative Results of Hibernus and Hibernus++ powered by different synthesized sources.

Synthesized source	Hibernus			Hibernus++		
	N. Restore	N. Checkpoint	Total Time (ms)	N. Restore	N. Checkpoint	Total Time (ms)
Wind Turbine	7	7	**584.9**	8	8	**512.6**
Kinetic	6	6	**3399.0**	2	2	**2145.0**
PV	-	-	-	0	5	**582.4**
Input Current 200uA	-	-	-	0	12	**806.8**

From Table 2.8, it is possible to see that Hibernus cannot operate with neither the PV signal nor the current source. In the case of wind turbine and kinetic signals, Hibernus takes more time to execute a FFT algorithm than Hibernus++. Hibernus needs to checkpoint and restore the system 7 times to complete a FFT routine, when powered by the wind turbine and 6 times in the case of kinetic. For Hibernus++, it takes 8 Restore and Checkpoint routines when powered by the wind turbine, and only 2 for the kinetic signal. Although Hibernus++ takes more checkpoints than Hibernus when powered by the wind turbine signal, it completes the FFT algorithm faster due to Hibernus++ spends less time sleeping than Hibernus.

A third comparison scenario was made considering different values for decoupling capacitors. The aim of this is to demonstrate whether Hibernus++ is able to work with different platforms and compare its performance with Hibernus. The system was powered by a 3V sinusoidal signal at 6Hz. The results are shown in Table 2.9. Here, it is possible to see that Hibernus does not work when the decoupling capacitance is lower than 20μF, meanwhile Hibernus++ properly operates. In the case of the other three values, Hibernus++ shows a performance improvement according with the value of the capacitor (the higher the capacitor, the smaller the total time), whilst Hibernus presents a constant execution time, showing no improvements with higher capacitances.

Table 2.9: Hibernus and Hibernus++ with different decoupling capacitances.

Decoupling capacitance ∑C (µF)	**Hibernus**			**Hibernus++**		
	N. Restore	**N. Checkpoint**	**Total Time (ms)**	**N. Restore**	**N. Checkpoint**	**Total Time (ms)**
10	-	-	-	2	2	**395.2**
20	2	2	**376.3**	2	2	**389.4**
30	2	2	**376.1**	1	1	**243.7**
40	2	2	**376.0**	1	1	**238.9**

2.4 Transient Computing Systems: Challenges

Although *system state retention* schemes have enabled computation to be completed across multiple power cycles, there are two important challenges that these approaches have not addressed, which are retaining the state of external peripherals and keeping track of time.

2.4.1 Retaining External Peripherals State

A transiently-powered sensor system could not properly operate unless the peripheral state is restored after a power outage. Figure 2.24 shows an example of an incorrect operation that may occur in a conventional transient system based on a *system state retention* approach (e.g., HarvOS [117]). The application example first configures the serial protocol (Configure_protocol) to communicate with the external peripheral (a digital sensor in this example). Then, the MCU sends a reset instruction to the peripheral (Sensor_reset) and configures the sensor to start sampling data (Configure_sensor). However, a power failure occurs before the peripheral starts sampling. Then, when the energy is again available the system state is restored. Nevertheless, the peripheral's state is not restored, i.e., the sensor is not properly configured (the configuration was not saved into NVM). Therefore, the sensor would revert to its default configuration after the power outage, and the program would be unaware of this.

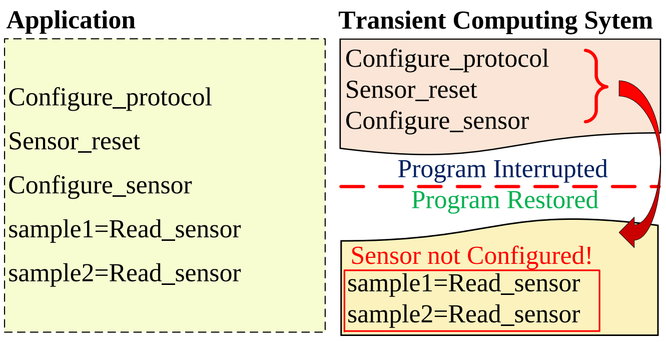


Figure 2.24: Malfunction of existing system state retention approaches when working with external peripherals after a power failure.

The *system state retention* approaches described in Section 2.2.2 are focused on the main memory, core registers (i.e., general purpose registers, link register, program counter and stack pointer) and peripheral registers (which are usually controlled by internal peripherals such as ADC, DAC, GPIO, etc) [118]. These approaches are not concerned with retaining the configuration of external peripherals because they are not included in the design. This has led researchers to engage in developing solutions that allow the state of external peripherals to be retained between power outages.

Berthou *et al.* [119] proposed Sytare, a software approach which retains not only the system state but also the configuration of external peripherals attached to the MCU through SPI. This approach includes a so-called *kernel code*, which operates between the main application and the library to access the external peripheral features (peripheral driver), and it is in charge of saving and restoring the peripheral state. However, the user not only has to write the function to configure the peripherals but also implement a structure, called *device context*, for each attached peripheral. This structure is used by Sytare to encapsulate the functions and save the data exchanged between the MCU and the external peripheral. Moreover, the user has to write a function to restore the peripheral configuration (one per connected peripheral).

If more than one peripheral is attached, the developer has to indicate in which order they have to be restored, because this solution would not work in a system where the peripherals are accessed in a different order from one cycle to another. However, it excludes essential details: it does not describe how the developer has to change each peripheral driver in order to update the structure (device context) needed to avoid peripheral volatility and how to implement the restore function for each attached peripheral. Besides that, Sytare incurs a time overhead of over 30μs per peripheral instruction because it needs to save the peripheral state each time an instruction is issued. This imposes an overhead of up to 137% when configuring a radio transceiver.

Recently, designers have oriented their research towards implementing non-volatile solutions for external peripherals. Li et al. [120] proposed a ferroelectric non-volatile flip-flop based input-output (IO) architecture that aims to reduce the initialization overhead caused by power outages. They replaced typical IO D-type flip flops with non-volatile flip-flops by adding two ferroelectric capacitors. Thus, when a power outage occurs, the peripheral configuration is retained in local ferroelectric capacitors, allowing a fast backup operation. However, this approach is focused exclusively on sensors, requires special-purpose hardware and does not offer a solution for off-the-shelf peripherals.

Hardware approaches such as Non-Volatile Processors (NVPs) [121, 122] attempt to save *in-place* snapshots by adopting non-volatile SRAM and registers. However, these hardware approaches do not offer solutions as they only retain the system state (main memory and processor registers), but not the configuration of digitally interfaced peripherals. Other solutions such as WISP [5], WISPCam [100] or federated energy storage [9] do not snapshot the configuration of the external peripherals because they operate only when the energy stored in small capacitors is enough to complete the required task. The peripherals are configured from scratch and perform the same function each time they are enabled.

In summary, there is an unmet need for a generic solution capable of retaining the state of multiple peripherals connected to the MCU through external interfaces such as I^2C and SPI. This would enable the state of complete sensor systems, e.g., incorporating an MCU, a digital luminosity sensor [123] and a transceiver [124].

2.4.2 Keeping Track of Time

Keeping track of time is an important element in sensor systems and specially in wireless sensor systems. The knowledge of time could help to program processing tasks or define whether a sampled data is valid or has expired. However, in transient systems that are constantly interrupted due to energy failures, keeping track of time is an important challenge that have to be solved. Some authors, whose designs are based on the WISP platform, which is powered by RFID signals, consider to use the RFID reader itself as the natural reference for synchronization among sensor systems [125, 126]. Nevertheless, the systems need to be synchronized after every energy interruption and this method only works on designs based on WISP platforms powered by RFID signals.

J. Hester *et al.* [127] proposed a method (TARDIS) to calculate the time that has passed after a power failure, based on the SRAM decay. Here, it is considered a section of SRAM of 256 bytes that is initialized to 1. When the power is cut off, the value of an SRAM cell is decayed if it is reset from 1 to 0. Therefore, the elapsed time from one power cycle to another is calculated based on the percentage of bytes that decayed, while the system was out of energy.

However, it is not desirable to reserve 256 bytes of SRAM, just for calculating time. That size may represent up to 25% of the memory of a typical MCU. Moreover, the process of initialization and decay verification of the SRAM consumes approximately 50μJ and takes 15ms to be completed. This is an important drawback considering that transient systems may operate during periods shorter than the time needed for this purpose. In addition to this drawback, the SRAM decay is affected by the temperature, reducing its accuracy.

In order to overcome the disadvantages of TARDIS, a new method named CusTARD [9] was proposed, which is based on a small hardware addition around a ceramic capacitor (instead of using an SRAM section), which is charged until a specific voltage, and after a power failure, the algorithm measures the voltage in the capacitor and estimates the time as a function of how much the voltage decayed. Figure 2.25 shows the CusTARD circuit.

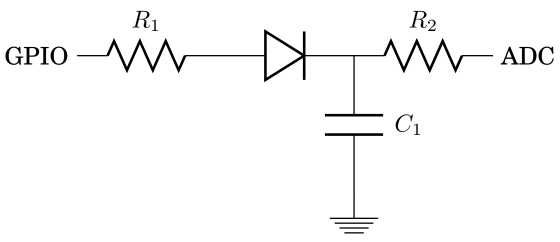


Figure 2.25: Schematic of CusTARD circuit. Extracted from [9].

The CusTARD capacitor is charged by a GPIO through the charge resistor and a diode. The measurement resistor is connected to an analog-to-digital converter (ADC) pin of the microcontroller. The elapsed time this method can measure, depends on the size of the capacitor. However, the larger the capacitor, the greater the energy expenditure and charging time, which are important concerns in systems powered from highly variable sources and that lack batteries.

Another approach was proposed by Uvis Senkans *et al.* [128], who estimates the elapsed time in a cycle trip counter, dividing the travelled distance by the cycling speed. An alternative method was proposed by S. DeBruin *et al.* [103], in which the wireless receiver is in charge of calculating the elapsed time from one received packet to the other, and based on that estimating the energy being harvested by the sensor system. However, none of these solutions has solved the problem of tracking time in a real application with energy constraints, that operates transiently and that needs a sense of time over a very wide dynamic range (milliseconds, seconds, hours, etc.)

2.5 Summary and Discussion

In recent years, researchers have proposed a new concept termed *transient computing* or *intermittent computing*, which aims to add small capacitors or totally eliminate the need of energy storage and operating directly from the harvesting sources. Some transient computing approaches are based on tasks that include a small capacitor whose size depends on the energy needed to complete their functions without interruptions. Others, proposed *system state retention* approaches that attempt to enable system to operate without extra capacitance and retaining the system state during power outages.

Task-based approaches have not proposed generic solutions that can be implemented in different applications. Besides that, if the EH source is not able to provide enough energy to complete application's tasks, the system would never operate or would be restarted from the beginning at each power failure. Moreover, these systems typically incorporate additional circuitry such as MPPT, power management units, etc., which increase the cost of solutions.

In the case of existing *system state retention* approaches, they offer different methods to retain system state, which implies that one approach can outperform the others depending on the energy operating conditions. However, they are focused on the processing unit, i.e. they do not consider the interaction of the MCU with external peripherals. Moreover, they may fail to execute *atomic* tasks than need to be completed without power interruptions, such as wireless transmissions.

An important challenge that transient computing systems have, is keeping track of time with diverse granularity. Existing solutions have solved time issues for short intervals (in the range of seconds), while typical sensor system applications may need time for longer intervals (minutes, hours, days).

In conclusion, existing transient computing designs have not proposed solutions to enable system to operate with external peripherals, capable of performing *atomic* and long-term computation, minimizing the need for extra energy storage and keeping track of time with diverse granularity. Additionally, not all sensor system applications are suitable to operate transiently. In order to drive some of these challenges, the next chapter presents an evaluation of application suitability for transient computing as well as a case study to motivate some of these issues.

Chapter 3

System-level Challenges of Transient Computing: A Case Study

In the critical review of the state-of-the-art in transient computing systems presented in Chapter 2, the advantages and drawbacks of existing approaches were analysed. However, despite the fact that transient computing devices offer important solutions to reduce the size and cost of sensor systems, they are not targeted for all possible scenarios where common sensors could operate. This chapter presents suitable scenarios where transient systems are able to operate, including the design, implementation and experimental validation of a case study: a *task-based* transient step counter.

3.1 An Evaluation of Application Suitability to Transient Computing

A transient sensor node can only work when the amount of energy, that is being harvested, is enough to maintain the system operation [129]. If not, the device will stay either in low power mode or totally off due to transient computing devices lack batteries to sustain the operation during periods of power shortage. Because transient systems depend on harvested energy, they are suitable for applications that operate under two main scenarios, which are:

1. The parameter to be sampled is **the same than that which is being harvested.** These systems are commonly called Energy Harvesting Sensors [130] because any change in the source, provokes a change in the measured results.

2. The operations of sensing, processing and transmission are **executed opportunistically when there is harvestable energy.** The applications that can operate under this scenario are those where the time is not the main concern, because transient systems may need to checkpoint, sleep and restore several times before completing a task.

Some designs that work under the first condition were described in Chapter 2 [5, 103, 130]. In the second scenario, systems operate in a *task-based* scheme (adding small capacitors to save enough energy to complete their tasks). Several proposed designs are powered by photovoltaic (PV) cells or RF energy harvesters [131, 132, 101].

In order to make a scenario suitable for working transiently, it could be necessary to design new algorithms to solve the challenges without complicating excessively the operation of devices. Some of the existing challenges may require research to find a solution. However, certain scenarios have characteristics that make them infeasible for transient computing, or challenges that could not be resolved. In consequence, transient computing approaches can be classified according to the stage at which they are targeted, others where they could potentially work if some challenges are addressed, and finally those for which they are not targeted. In Table 3.1, some possible scenarios are classified.

Table 3.1: Classification of Possible Scenarios.

Targeted for	**May Work**	**Not Targeted for**
Event Detection	Periodic Sampling	Real-time Operation
EH Proportional	Sequential Sampling	Continuous Operation

As shown in Table 3.1, *event detection* and *EH proportional* are scenarios where transient computing devices can properly operate. In the *event detection* sensors, the harvestable energy is a consequence of the event to be sampled. For example, a piezo-film vibration sensor attached to a door to monitor the entry and exit of personnel [130, 133]. A vibration is produced when the door is opened and closed, which generates the current to activate the sensor. The force with which the door is closed, could increase or decrease the harvestable energy. Nevertheless, the system only detects the event of opening and closing the door. The increment or decrement of energy does not alter the result.

The *EH proportional* sensors are similar to *event detection* ones, but they sample the same parameter that is being harvested. The activity of the sensor node is increased proportionally to the rate that energy is being harvested, e.g., a sensor powered by a wind turbine to measure the wind's speed. Depending on the speed of the wind, the harvestable energy will be reduced or increased.

In the scenarios where transient computing approaches could potentially work, it is necessary to look for different solutions or define under which conditions, transient computing systems could operate. Below, some existing challenges of possible scenarios, are described:

- **Data Coherency:** It is necessary to verify whether after each restoring routine, the function that was previously interrupted should continue or the system has to be restarted from the beginning.
- **Periodic Sampling:** This may be suitable for predictable energy harvesting sources, and systems that may have some timebase to be able to sample regularly. Synchronizing the local clock of a sensor system with the gateway or host may be needed, at each time the system exits from low power mode or it is restored [134].
- **Possible substantial time/energy overhead:** This may be caused by checkpointing, low power mode, switching tasks, etc. It is important to minimize these factors in order to make feasible transient computing devices.
- **Interfacing with External Peripherals:** Peripherals are managed by registers that are located into a volatile memory. In order to preserve peripheral state and keep the application coherency, these registers have to be retained between power failures.
- **Peripherals Latency:** Analysing the effect of the peripherals latency during checkpointing and restoring routines, is needed. This latency has to be bounded because the energy could be available during a short time.
- **Perform *atomic* tasks and long-term computation:** Typical *task-based* approaches usually perform exclusively *atomic* tasks that are completed without interruptions. In the case of *system state retention*, they are oriented to perform long-term computation without adding any energy storage. However, they may fail to execute *atomic* tasks such as wireless transmissions.
- **Keep track of time:** As mentioned before, some application may need a time base to process the information or to indicate if the sampled data is valid or not. Therefore, it is necessary to look for existing designs that could be implemented in transient computing systems, or propose new methods that could solve it.
- **Transmitting data wirelessly:** Transmitting data is an important challenge because transient devices lack supercapacitors to buffer energy, and the power consumption of transceivers is commonly higher than that of the microcontroller and sensors.

Table 3.2a: Suitable Applications for Transient Computing Systems (Part 1).

APPLICATION	DETAILS	CHALLENGES TO OPERATE TRANSIENTLY
Measuring Vibration/Acoustic Vibration Meter	It is possible to power these sensors not only by kinetic but also with other EH sources such as PV cells.	Configure and calibrate the sensor after a power failure (e.g. when using an accelerometer).
		Self-validation and self-compensation to achieve high accuracy.
		To have enough energy to transmit data when needed and/or when channel is free.
		Failure to communicate with other nodes or base station to send data due to noise, collision or power failure.
		Performing the operations such as vibration acceleration, RMS vibration velocity, vibration displacement and Vibration spectrum analysis in FFT, may need longer intervals than those the EH source can provide without interruptions.
		Powering the sensor when placed on remote locations.
Green Houses: Control micro-climate conditions to maximize the production.	Transient Computing sensors (TC) can be in charge of monitoring soil moisture, control of fertilizing, etc. that can be done sporadically.	Synchronize with coordinator.
		To have enough energy to transmit data when needed and/or when channel is free.
Sportsmen Care: Vital Signs monitoring.	TC can operate powering the sensor with PV cell or kinetic energy and should back up the information before a power failure.	Efficient and practical Energy Harvesting source to power the device.
		Checkpointing the state of sensor and collected information in case of a power failure.
		Fast recovery to reduce the time overhead and in consequence, the data loss.

Transient computing devices are not targeted for systems that have to operate continuously, because they do not have batteries or supercapacitors that could sustain the operation in situations of scarcity or lack of harvestable energy. This property is also a limiting factor for real-time operations (applications that require bounded delay latency [135, 60, 136]). Transient computing devices cannot guarantee the execution of a certain task within a specified time constraint [137], for different circumstances such as:

- Latency to restore the system.
- The system might not be able to complete a task in one fell swoop.
- Availability and amount of energy is not accurately predictable.

With the previous description, it is possible to present some applications that are feasible for transient computing systems. In Table 3.2a and 3.2b, some sensor applications are listed, including the challenges that have to be solved in order to implement a transient system. As shown in tables, there are important scenarios were transient systems can operate. For this research work, a *task-based* step counter will be designed and implemented, attempting to operate without adding any extra energy storage.

Table 3.2b: Suitable Applications for Transient Computing Systems (Part 2).

APPLICATION	DETAILS	CHALLENGES TO OPERATE TRANSIENTLY
Smart Lens to monitor body parameters	Due to the limited space these system have to operate, they cannot have big batteries or supercapacitors to power the system continuously, therefore Transient computing devices may be a potential solution.	Energy available for short periods
		The system would be constantly interrupted due to power failures.
		Energy limited due to the EH must be too small and there is not space for capacitors.
Ultraviolet Radiation to monitor People exposure	Due to UV radiation is linked to solar exposure, it is possible to have a TC that can monitor the UV radiation and save the state in case of a power interruption caused by the movement of the person under test.	Checkpointing the state of sensor and collected information in case of a power failure.
		Reduce the time overhead of the system between the energy is available and the sensor starts working.
Smart Grid: Energy consumption monitoring and management	The sensor is powered by the same parameter that is being sampled, such as current flow in a domestic installation.	Self-validation and self-compensation to achieve high accuracy.
		Reduce the time overhead of the system between the energy is available and the sensor starts working.
Smart lighting	If the lighting starts from the fact that all lamps are ON, as initial state and then, they are dimmed depending on the necessities.	Communicate with other nodes to manage the lamps.
		Failure to communicate with other nodes due to noise, collision or power failure.
		Acknowledge messages to ensure the instructions were properly received.
Step Counter	Counting steps, steps frequency, transmitting collected information.	Suitable EH source.
		Keep track of time.
		Save state of peripherals.
		Manage voltage thresholds to be able of powering a transceiver.

3.2 Transient System Application: Step Counter

In order to determine whether a basic step counter (i.e., that only counts steps) is a feasible application for transient computing, it is necessary to analyse the key aspects of its functionality. First, the parameter to be sample are the taken steps. Therefore, it is important to define if an EH source can harvest energy from each step. In Section 2.1.1, different sources that harvest energy from human motion, were described. Thus, the operation of the system can be synchronized with the presence of energy. Second, the application to be designed only needs to operate when a step is taken, i.e., it does not have to be permanently active, which would be a limitation in a transiently-powered system. Finally, the step counter can work under an *event detection* scenario if the EH source provides the energy required to enable the system at each step, or as an *EH proportional*, in which the system works opportunistically triggered when there is enough energy. The operating strategy will be further defined during the design process. Therefore, a step counter is a feasible application for transient computing, if the challenges of finding a suitable EH source, matching the system energy requirements with the energy budget and providing an acceptable accuracy, are properly addressed. In this Section, the implementation of a *task-based* step counter is described, including the performance evaluation of the harvesting source, the energy requirements and feasible components for the system, and the proposed design to solve the existing challenges.

3.2.1 Energy Harvester Performance Evaluation

It is important to look for energy harvesting sources that could enable the system without adding extra capacitance and keeping its portability and the possibility of operating under different environmental conditions. Besides that, the source has to harvest energy while the person is using the system in order to synchronize its operation with the energy availability. The EH source attempted to be used to power the step counter is a ferroelectret insole [77, 76]. Figure 3.1 shows the model of a 3-layer insole of polypropylene ferroelectret (PP) connected in parallel with bonding films between layers. A similar insole, but with 30 layers, is going to be used to power the step counter.

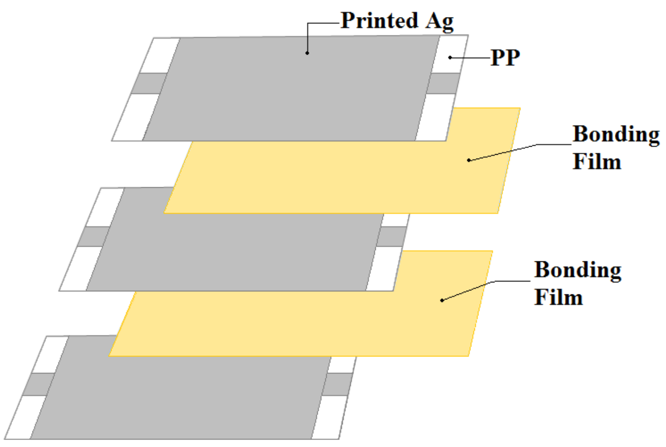


Figure 3.1: Structure of a multilayer ferroelectret insole.

The designers characterized the multilayer insoles, considering a capacitor of 2.2μF as a load, and calculating the energy stored by measuring the voltage in the capacitor after every step. However, that characterization does not solve the questions related with the 30-layer insole with different capacitors. Besides that, the instrument used to simulate a step, applies a uniform force over the insole, unlike the deformation caused by a real step. Therefore, the performance of the insole has to be evaluated with real steps and the proper loads, in order to aid the design process.

The designed insole is a prototype, which is still in the improvement phase and cannot be properly worn inside the shoe. For this reason, three different methods to evaluate the insole were proposed and validated. The idea was to implement a setup that made the experiments repeatable. The first method, named *Sole*, consists of placing the insole in the sole of the shoe, as shown in Figure 3.2. This methods could be consider more

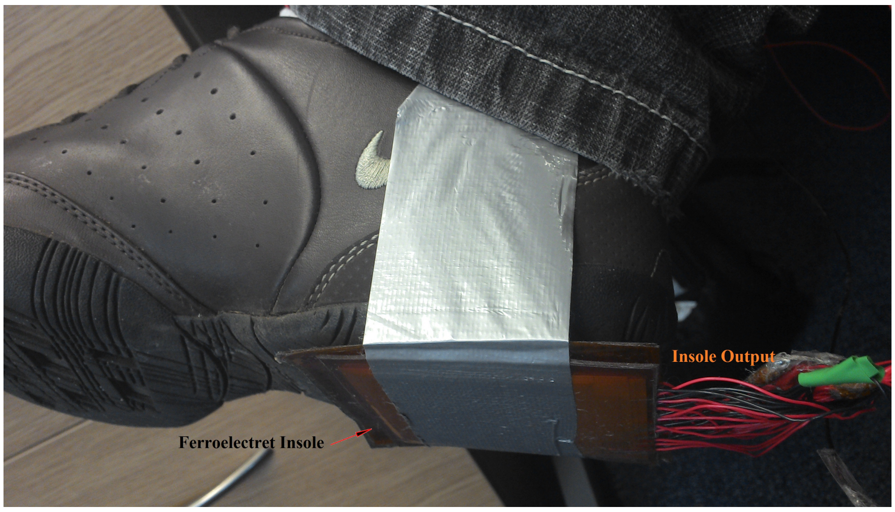


Figure 3.2: Sole Method. The insole is placed in the sole of the shoe.

realistic but the insole may be damaged and it is not possible to walk with it due to the insole is connected to the load.

The second method, called *Brink*, consists of placing the insole in a plastic block where the foot strikes in a specific area. The plastic has a border of about 1.5cm and it is placed on the floor. In order to test it, it is possible to walk around several times while the parameters are measured. Figure 3.3 shows the frame of the method.

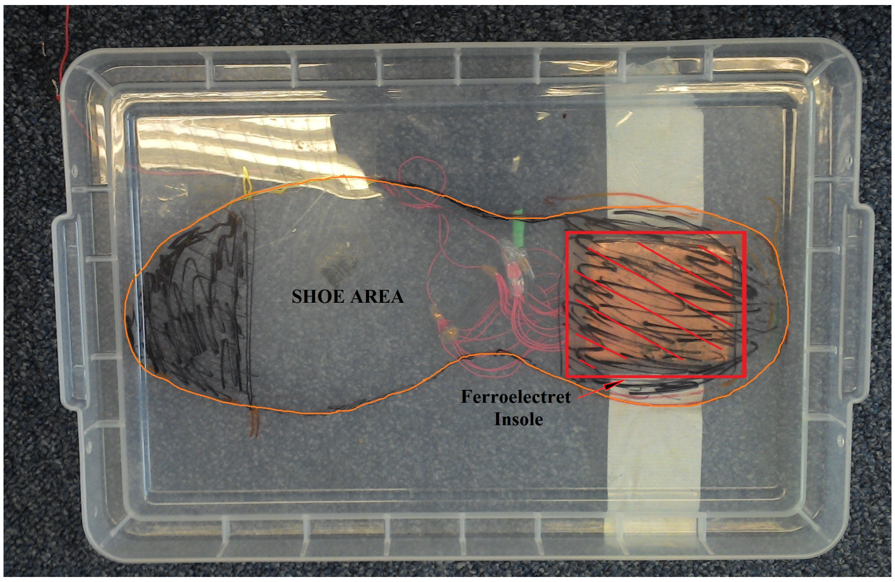


Figure 3.3: Brink Method. The insole is placed in the bottom of the plastic.

In the third method, called *Plastic*, the insole is also placed in a plastic sheet (shown in Figure 3.4), a little more rigid than the previous method, but without brink, which allows a normal step, i.e. it is not needed to rise the foot such in previous method to avoid the brink of the plastic.

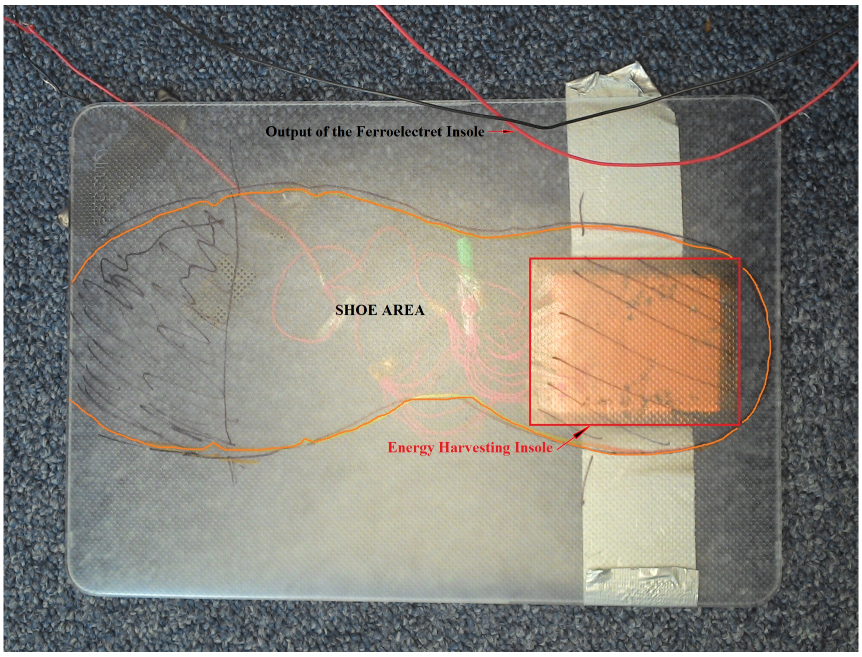


Figure 3.4: Plastic method. The material is softer than that used in brink method.

A resistor of 100kΩ was used as a passive load. A power analyser (Agilent N6705B) was used to measure the voltage, current and power generated by the insole. This system provides a log file with 16 thousand samples (one sample every 100μs) per each parameter. Therefore, to calculate the energy generated by the insole, the power signal was integrated by means of a Matlab script, following:

$$E_{insole} = \int_0^T P_{insole}(t)dt \tag{3.1}$$

where E_{insole} is the electrical energy generated by the insole, T is the duration time of the sampled signal, which in this experiment is of 1.6s, and P is the power signal generated by the insole.

The insole generates two peaks of voltage, one positive and one negative at each step. In order to rectify the negative peak, a schottky-diode bridge (BAT754S) was included between the insole and the loads. In Figure 3.5, the diagram of the evaluation circuit is presented.

The setup also considered the distance between steps (for *Brink* and *Plastic* methods), and the area where the right foot has to strike the insole, in order to have a constant applied force in each attempt, as shown in Figure 3.6.

For this experiment, 15 consecutive steps were taken for each method and the obtained energy was calculated with the following formula:

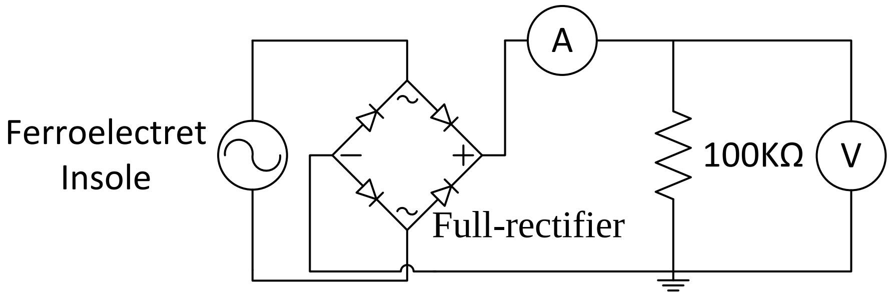


Figure 3.5: Circuit used to evaluate the proposed methods.

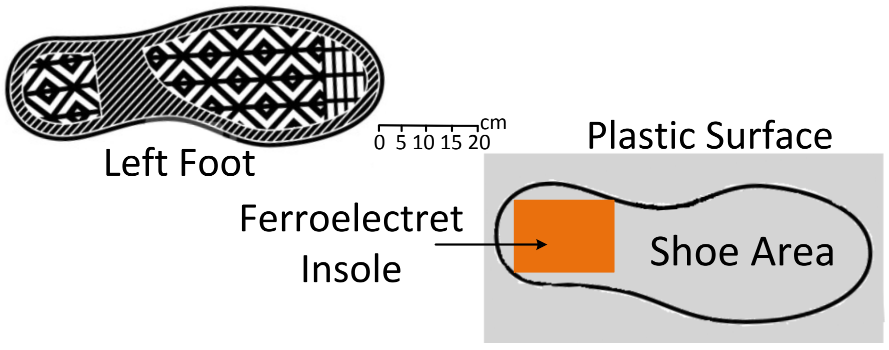


Figure 3.6: A constant distance between steps was defined.

$$E_{insole} = \int_0^T P_{insole}(t)dt \tag{3.2}$$

The obtained results were analysed and compared. Table 3.3 shows the average electric energy generated by each method as well as the calculated root mean square power (RMS) and the standard deviation (SD) of the obtained values of each step, in order to determine the variation or dispersion of the energy obtained at each step.

In Table 3.3, it is possible to see that the *Plastic* method generates less electrical energy than the other two methods. However, the main factor to consider the proper method is how similar the results are after several tests. Thus, *Plastic* method presents the smallest standard deviation among the 15 steps, which means that this method generates more repeatable results. This can be easily observed by using box plots of the 15 different values of energy obtained for each method (Figure 3.7). The low whiskers represent the distance between the minimum value and the first quartile, meanwhile the higher whiskers represent the value from the third quartile to the maximum.

As shown in Figure 3.7, the method that provides more energy was *Sole*. However, the values of energy for each step are more dispersed than the other methods (the point in yellow represents the mean value). Thus, the *Plastic* method will be used to evaluate the insole because the values of each step are closer to each other. Additionally, this method allows to walk around several times while the parameters are measured. On the contrary, if the insole was placed inside the shoe, it would not be possible to walk and

Table 3.3: Results comparison of the three methods.

Method	**Energy Average (μJ)**	**Power RMS (μW)**	**SD of data (μJ)**
Sole	3.517	11.535	2.008
Brink	3.187	24.783	1.686
Plastic	2.991	22.943	1.094

measure its parameters at the same time (the insole has to be connected to the load and the power analyser).

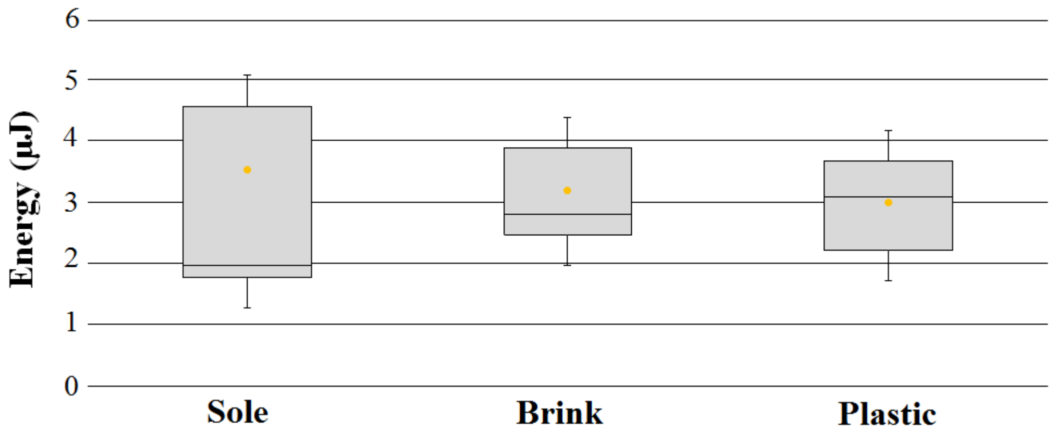


Figure 3.7: Box plot of the energy provided by the ferroelectret insole after 15 steps for each method.

In order to evaluate the performance of the ferroelectret insole when connected to an MCU, a range of capacitance values are used as loads. Although this work aims to negate the need for large energy buffers, and thus demonstrate the viability of transient systems in real applications, some capacitance (usually called decoupling capacitance) is inherently required by MCU's to decouple the energy source (and possible electrical noise) from the system. In *task-based* transient systems, this capacitance may also be exploited as a small energy storage.

The chosen values ranged between the minimum decoupling capacitance recommended by manufacturer for a typical MCU (i.e. 4.7μF), and the value incorporated in a reference design for that device (i.e. 16μF) [91]. These capacitors were tested using the characterization circuit showed in Figure 3.8. The circuit includes a bridge-rectifier connected at the insole output.

The ferroelectret insole is represented by a basic equivalent circuit for piezoelectric transducers [138]. The capacitance C_E is the inverse of the mechanical elasticity of the insole. L_M represents the seismic mass of the transducer. The capacitor C_0 is the static capacitance of the harvesting source and R_0 is the resistance of the dielectric material that

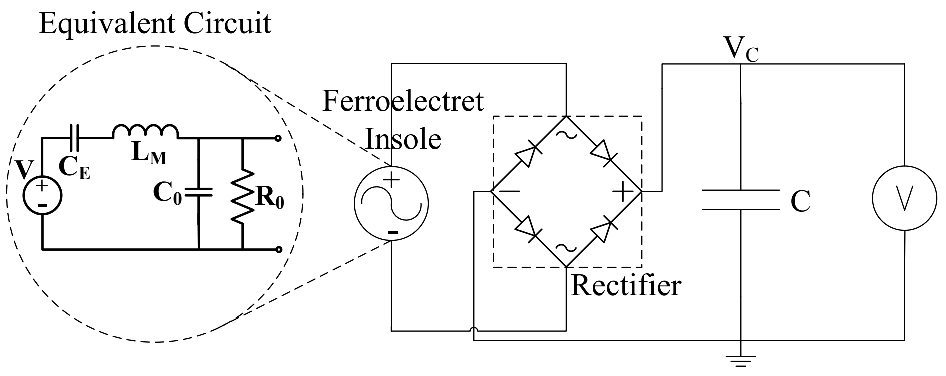


Figure 3.8: Characterization circuit including the full-wave rectifier.

forms the static capacitance (insulation resistance), whose ideal value should be over $10^{12}\Omega$. The voltage (V_C) is measured across the capacitor, and the electrical energy generated at each step is calculated.

In Figure 3.9, the sequence of one step over the insole is presented. The experiment was performed by a 70-kg person moving at two different step rates (referred to herein as operating modes): *walking* (approximately 70 steps per minute) and *running* (approximately 120 steps per minute). This test was executed five times for each capacitor in each mode and the results were averaged. The difference between *Walking* and *Running* is the speed and in consequence, the force at which the foot strikes the insole, which may provoke more or less energy at the output of the source.

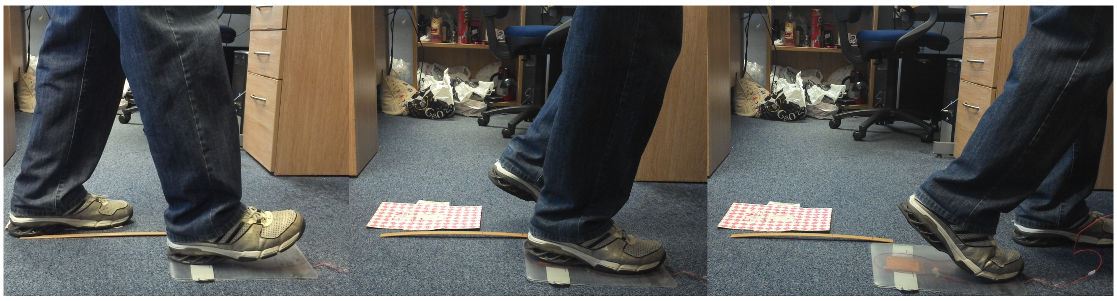


Figure 3.9: Sequence of a step. This procedure was repeated 5 times per each load.

As mentioned in Chapter 2.1.1, the ferroelectret insole generates two pulses (Figure 3.10), one positive (heel-strike) and one negative (heel-off), at each step. When the signal is rectified, it causes two increases in the capacitor voltage as shown in Figure 3.11. In *running* mode, the force applied to the insole is higher than the one in *walking* mode, whereby a higher voltage increment occurs at each step. This yields to uncertainty, which is higher for bigger decoupling capacitors, in the number of steps taken before the system becomes active. It is important to mention that a variation in the user's weight would only affect the magnitude of the power bursts, but not the pattern, i.e., the insole would always generate two pulses per step.

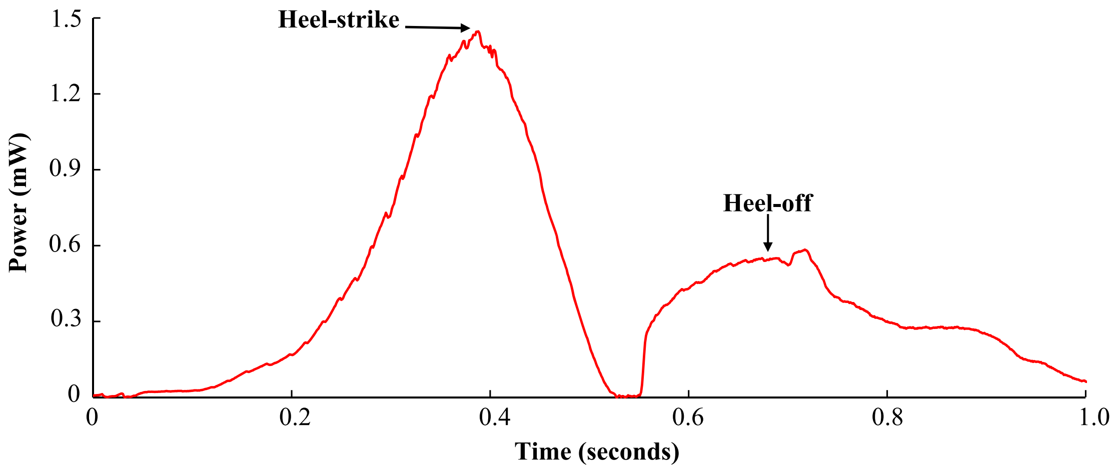


Figure 3.10: Rectified power bursts generated per step by the ferroelectret insole.

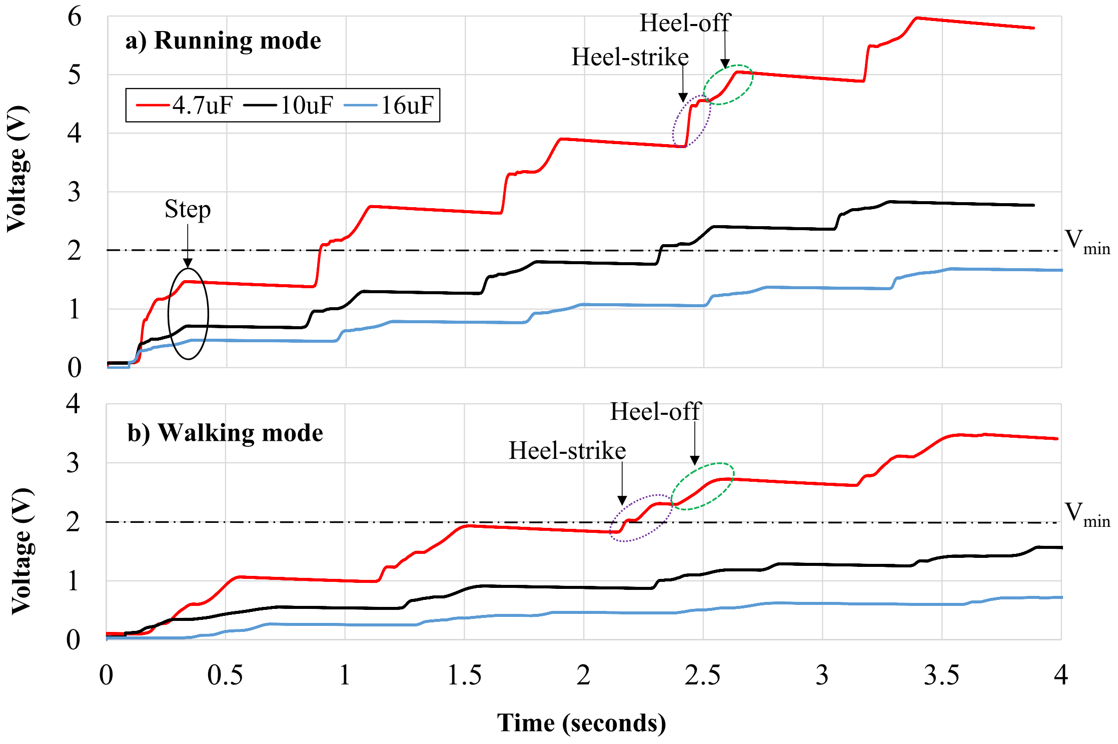


Figure 3.11: Voltage across each capacitor in **(a)** running mode and **(b)**walking mode, after multiple steps.

Figure 3.11 presents the plots of voltage level reached at each step for each capacitor. This figure shows that with a capacitance of 16μF, the voltage level is scarcely increased, whilst with a capacitance of 4.7μF, after two or three steps, the voltage level reaches a value more suitable for microcontrollers. The energy generated per footstep can also be calculated from this test by using the following equation:

$$E_{capacitor} = C\frac{V_2^2 - V_1^2}{2} \tag{3.3}$$

where C is the decoupling capacitance, V_1 is the voltage in the capacitor before the step and V_2 is the voltage reached after the footstep is taken. Table 3.4 shows the average energy that the ferroelectret insole charges into each capacitor in each mode. The maximum amount of energy is obtained with the lowest capacitance, being up to fourteen times higher.

Table 3.4: Mean energy obtained per single step.

Capacitor	Walking mode (μJ)				Running mode (μJ)			
(μF)	Step 1	Step 2	Step 3	Total	Step 1	Step 2	Step 3	Total
4.7	2.3	7.05	9.21	18.61	5.30	13.14	18.89	37.33
10	0.85	1.27	1.41	3.53	3.21	2.47	4.66	10.34
16	0.21	0.37	0.73	1.31	0.59	0.92	1.33	2.84

In next section, the energy requirements of the system to be designed, are analysed. With that information, it is possible to compare the energy provided by the insole with the necessities of the step counter in order to define the strategy to address any existing mismatch.

3.2.2 Overall System Design

The performance evaluation of the insole in Section 3.2.1, did not consider the effect of an active load, which causes a fast discharge of the decoupling capacitance and a consequent voltage drop across the MCU. Describing this requires the knowledge of the overall system, the energy requirements when operating in different modes (i.e. start-up, active and low-power) and the usage of the energy buffered during these modes to efficiently complete a task (i.e. count a step).

Architecture

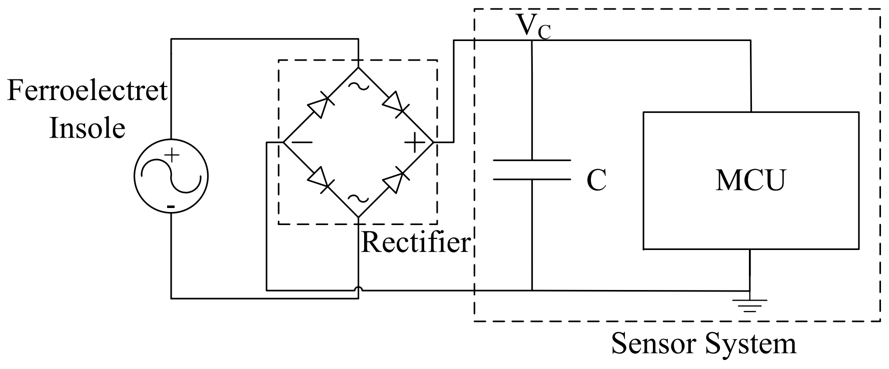


Figure 3.12: Simplified system architecture of the step counter with ferroelectret insole.

Figure 3.12 represents a simplified example of the system architecture, where the ferro-electret insole output is rectified and used to power the sensor system, which is composed of an evaluation MSP-EXP430FR5739 test board [91]. This board contains an MCU with a non-volatile Ferroelectric RAM (FRAM) memory and a minimum decoupling capacitance (C) of 4.7μF. This board also integrates a 3-axis accelerometer (acc) that could potentially be used to detect a footstep (e.g. by detecting movements which are then classified as a step). However, the energy overhead due to active sensors can limit their use in the case of resource-constrained systems. In the next subsection, the energy requirements of the overall system is analysed, by first considering the accelerometer as an option for step counting, along with a sensor-less solution that is more power efficient.

System Energy Requirements with Active Sensor

Figure 3.13 shows the process of counting a step that is divided into four stages:

- **Stage 1**. MCU off due to a power outage;
- **Stage 2**. MCU Start-up when power is available (i.e. when a pre-defined threshold V_{th} is reached);
- **Stage 3**. Counting a step (i.e. MCU configuration, accelerometer initialization, step detection and data retention in the FRAM);
- **Stage 4**. MCU in low power mode (LPM), after counting a step.

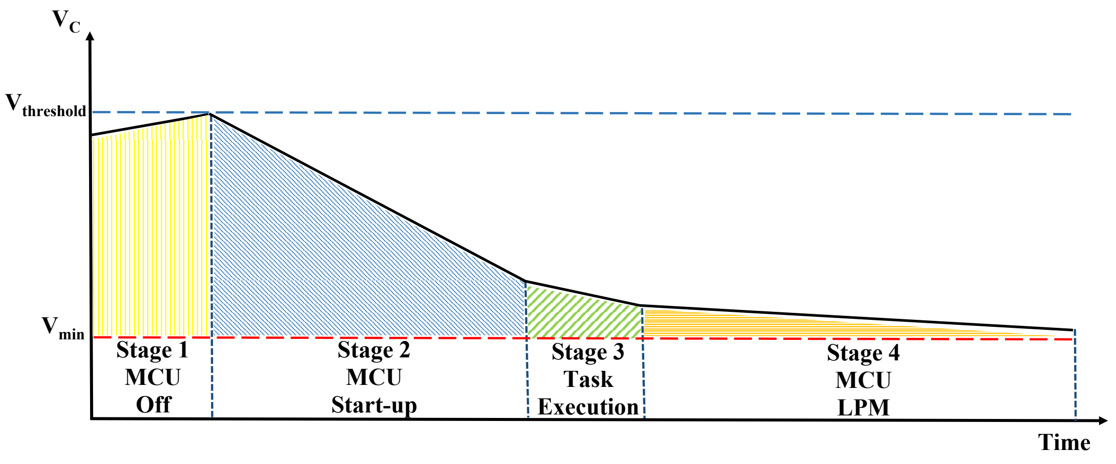


Figure 3.13: Stages in which the process of counting a step is divided.

In order to quantify the energy required during these stages and set the right value of V_{th}, the MCU has been characterized using a constant voltage (i.e. 2.4V) and by using the following algorithm:

1. The system is initially in off mode (stage 1).

2. The system is powered and, after the start-up, the MCU is configured (i.e. frequency and internal peripheral setting) and the on-board accelerometer is initialized.

3. A step is validated (accelerometer sampling plus data processing).

4. The state is retained in FRAM and enters in LPM.

Table 3.5 shows the current and time needed to successfully complete these stages. The current values include the possible noise of the energy source as well as the static current of devices. In particular, the MCU start-up current is much higher than the other stages, while the accelerometer requires a very long time to be set up.

Table 3.5: Current and time consumption of the system at each stage

Stage	**Tasks**	**Current (μA)**	**Time (ms)**
2	MCU Start-up	1000	0.9
3	MCU + acc	500	13.5
3	Validate a step	300	3.6

The values in Table 3.5 will be used in the following analysis to evaluate the optimum value for V_{th} which can be defined as:

$$V_{th} = V_{min} + \frac{1}{C}\int_0^t I(\tau)d(\tau) \tag{3.4}$$

where V_{min} is the minimum operating voltage of the MCU (1.8V), C is the decoupling capacitance and I is the current consumption. From Table 3.5, Stage 3 has two different sub-tasks that require different current values. The highest current value refers to MCU configuration and accelerometer initialization, while the second value refers to the current needed to validate a step. Thus, the Equation 3.4 can be presented as follows:

$$V_{th} = V_{min} + \frac{1}{C}(I_{su}t_{su} + I_{acc}t_{acc} + I_{vs}t_{vs}) \tag{3.5}$$

where I_{su} and t_{su} are the current and the time needed for the start-up (Stage 2), I_{acc} and t_{acc} are the current an time for setting the accelerometer (Stage 3), I_{vs} and t_{vs} are the current and time taken by the MCU to validate a step. Combining the values in Table 3.5 and the Equation 3.5, a value of V_{th} equal to 4.01V is obtained, that is higher than the maximum operating voltage for the MCU (3.6V).

For the reason underlined above (i.e. high energy overhead), using the accelerometer is not feasible in an intermittently-powered and resource-constrained system. In the next subsection, a solution that allows to build a more energy efficient step counter is presented, without using an active sensor but which delivers a higher accuracy.

Sensor-less System

An alternative way to implement a step counter without active sensors is to use the energy harvesting source as an *event detection* sensor, where the availability of energy is an indicator for a footstep. This approach would reduce the energy required to validate a step, removing the overhead for the accelerometer setting and data processing. Moreover, this will possibly increase the accuracy of the system discarding 'false' steps.

Table 3.6: Parameter values of the system in each stage without accelerometer

Stage	**Tasks**	**Current (μA)**	**Time (ms)**
2	MCU Start-up	1000	0.9
3	Validate a step	300	0.14

Thus, the MCU needs to be re-characterized by using the following updated algorithm: the system is powered and, after start-up, the MCU is configured. It then validates a step, retains the state in FRAM and enters in LPM. Table 3.6 shows the values for current and time with this algorithm. The time for counting a step is much smaller than the time needed with the accelerometer. Thus, Equation 3.5 can be now updated by removing the current and time overhead of the accelerometer:

$$V_{th} = V_{min} + \frac{1}{C}(I_{su}t_{su} + I_{vs}t^{*}_{vs}) \tag{3.6}$$

where t^{*}_{vs} is the actual time needed for counting a step. Combining the values from Table 3.6 and Equation 3.6, a V_{th} equal to 2.2V is obtained. From Equation 3.3, the energy required to charge the decoupling capacitance to that voltage is 11.37μJ. As shown in Table 3.4, the insole is able to charge up to 18.61μJ in walking mode, after 3 steps. Therefore, it is possible to implement an event detection sensor based on the energy harvesting source and using the 4.7μF decoupling capacitance.

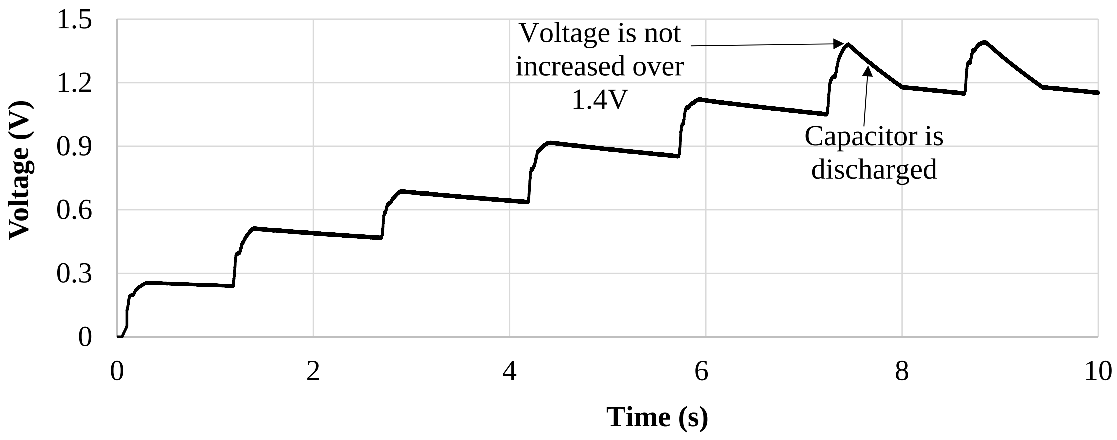


Figure 3.14: Effect of the quiescent current of the MCU, when charging the decoupling capacitance.

An important challenge during the step counter implementation, was to cope with the quiescent current of the MCU (in the order of 100μA), when the voltage is between 1.4V and 1.8V. Figure 3.14 shows the effect of the quiescent current in the charge of the decoupling capacitance. When the voltage reaches 1.4V, the capacitor is quickly discharged before reaching the minimum operating voltage, which causes the MCU is never active. Therefore, an additional start-up circuit was included to minimize the quiescent current. This circuit guarantees a reliable start by detecting the input voltage and only turning on the supply to the MCU when its input is above V_{th} and switching it off when the voltage drops below V_{min}. This is enabled by two voltage monitors, which are configured in a MOSFET latch arrangement. Figure 3.15 shows the final design including the ferroelectret insole, the rectifier, the sensor system and the additional start-up circuit.

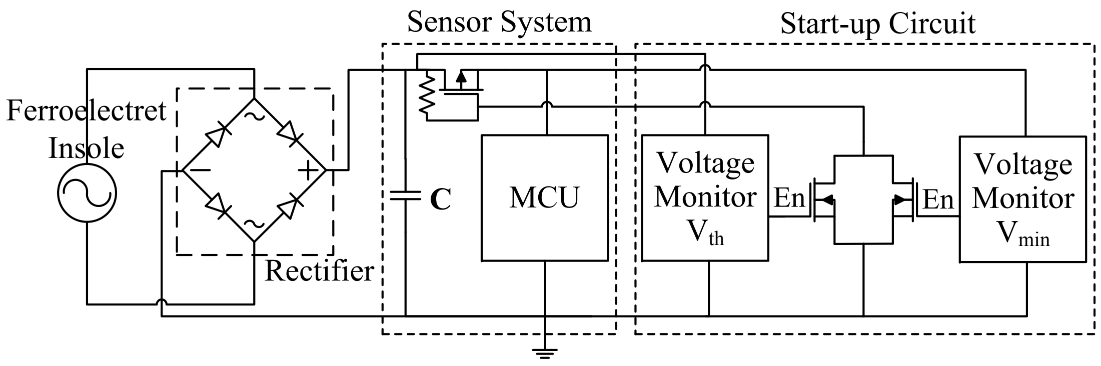


Figure 3.15: Task-based Transient Step Counter circuit diagram.

3.2.3 Functional Validation and Comparative Evaluation

The presented step counter has been implemented and experimentally validated. The accuracy of the solution was compared against two smartphone applications. To verify system operation, two GPIO pins of the test board were configured to indicate when the system counts and enters the LPM, respectively. Both signals were monitored by a PicoScope at the same time as the voltage across the capacitor and the MCU. That information was logged and later plotted. Thus, it was possible to visualize how many steps were taken before the system started working, to detect possible errors. The number of steps is saved in a FRAM variable in order to retain the value between power failures.

As shown in Figure 3.16a, the system is active when the voltage reaches V_{th}. The sensor node takes into account the initial three steps needed to reach this voltage. Then, the MCU is configured and the step is counted. Once the task is executed, the system enters low power mode before having a power outage. Figure 3.16b shows a detailed snapshot of the process of counting a step from Figure 3.16a, including the time needed by the system to start up, configure the MCU and increment the counter. As mentioned in

Table 3.6, configuring the MCU and counting a step takes approximately 140μs. The MCU remains in LPM until the voltage drops below V_{min}.

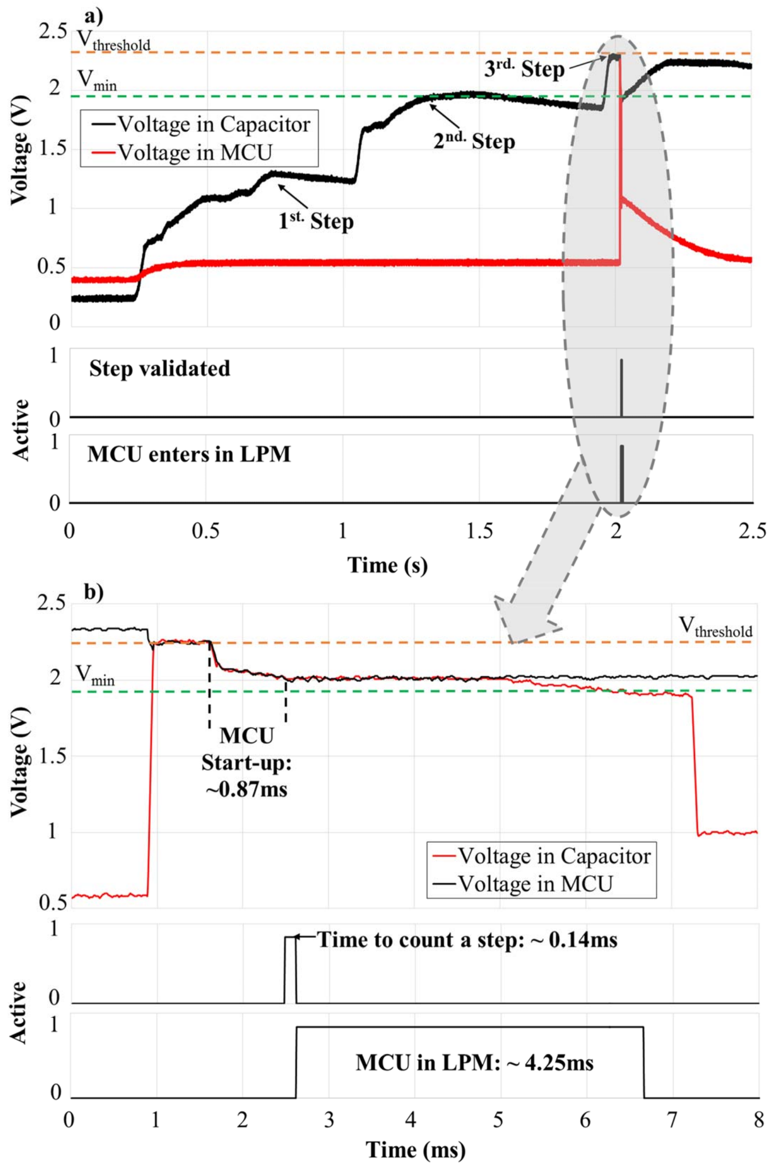


Figure 3.16: Operation of the task-based transient step counter. **(a)** After three steps the system starts working. **(b)** Detailed description of the process of counting a step.

Figure 3.17 shows the system counting three consecutive steps. The sharp voltage drop occurs because as soon as the input voltage reaches V_{th}, the MCU is enabled which causes the tiny amount of energy stored in the decoupling capacitor to be immediately consumed. In walking mode, the average time between two steps is approximately 0.9s (worst case). During this interval, the ferroelectret insole does not provide any further energy and the voltage drop, due to the start-up circuit and leakage current, is approximately 0.1V. This means that the voltage across the decoupling capacitance is about 1.7V, before the next step is taken. Considering that each step in walking mode increases the voltage by 0.8V, the sensor system is able, after the first three steps, to count each subsequent step. Therefore, the counter of taken steps is increased by three the first time the system starts working in order to compensate the number of steps needed to reach the operating voltage, and then, the step is validated by solely reaching V_{th}.

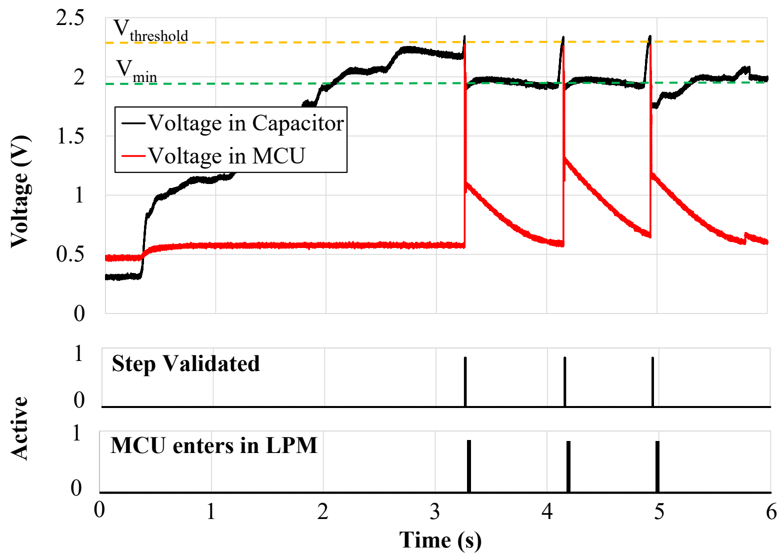


Figure 3.17: Transient system counting three consecutive steps.

In order to evaluate the performance of the design, its accuracy is compared against two existing Android® smartphone applications: Pacer [139] and WalkMate [140]. The applications use the 3-axis accelerometer built in the smartphone in order to sense the movement and, based on an algorithm, validate when the movement was caused by a taken step. A total of 400 steps were taken per step counter solution (1200 steps in total): 20 steps for 10 attempts in each operating mode. These experiments were performed by a 70-kg person with the ferroelectret insole attached to the shoe and the smartphone placed at their waist. Figure 3.18 shows the experimental results of the proposed step counter and the two smartphone applications, at each attempt in walking and running mode. The dotted line (in green) represents the real number of steps.

The proposed step counter has an average error of 3.5% in walking mode, while in running mode the average error is 1%. The maximum error in a single attempt is 2 steps in walking mode (attempts 4 and 9) and 1 step in running mode (attempts 4 and 7). In the case of the first smartphone application (Pacer), it has an average error of 25.5% and 12.5% in walking and running modes, respectively. The maximum error in a single attempt is 8 steps (attempt 5) when walking and 4 steps (attempt 4) when running. The second application (WalkMate) shows the worst performance in walking mode, having an error rate of 41.5% with a maximum error of 10 steps in attempts 5, 6 and 10. However, in running mode, this application has a better performance, with an error rate of 4% and a maximum error of 2 steps in attempts 2 and 5.

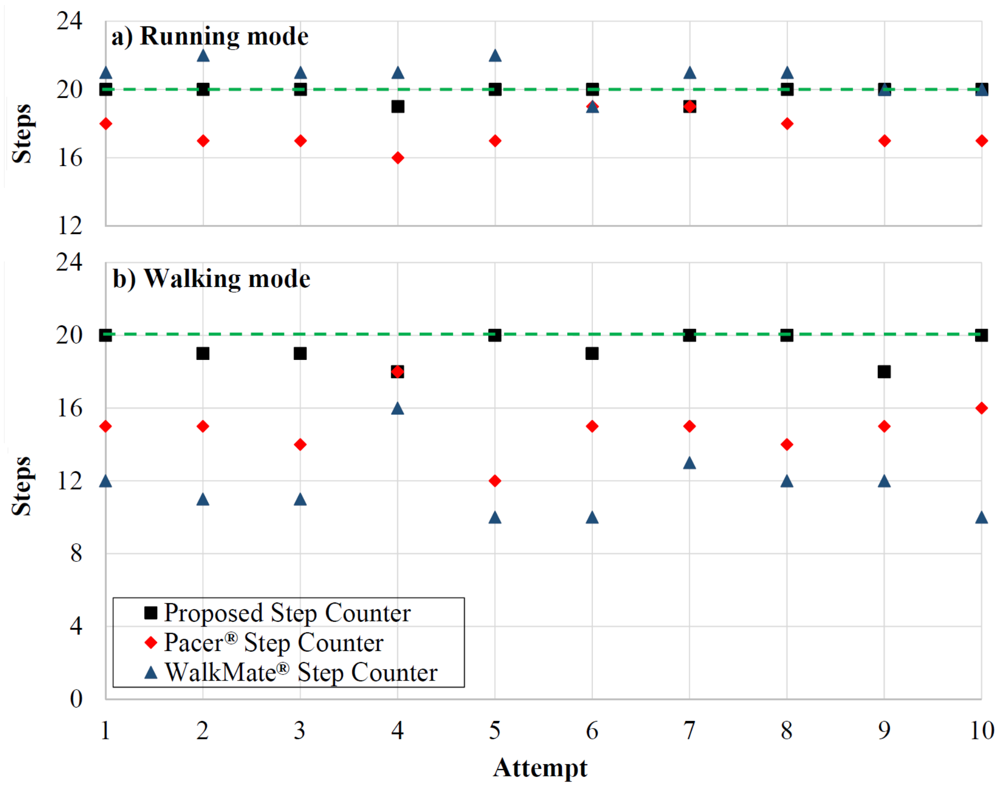


Figure 3.18: Performance comparison between the intermittently-powered step counter and two smartphone applications in a) running mode and b) walking mode.

Another test was executed, incrementing the number of steps to 50. Table 3.7 shows the results obtained for each step counter in walking and running mode. Also in this case, the designed step counter has the lowest error rate in walking mode and it does not present errors in running mode. The WalkMate application has the highest error (44%) in walking mode, while Pacer had maximum error in running mode (4%).

Table 3.7: Performance comparison of the three devices tested with 50 steps.

System	**N$^{o.}$ Steps (Walking)**	**Error Rate (%)**	**N$^{o.}$ Steps (Running)**	**Error Rate (%)**
Proposed Step Counter	49	2	50	0
Pacer Step Counter	40	20	48	4
WalkMate Step Counter	22	44	51	2

The marginal error in the proposed step counter is because of the adopted solution for compensating the initial number of steps (i.e. when the system is still not active). In a small number of cases, after three steps the voltage across the decoupling capacitance does not reach V_{th}. This mainly happens in *walking* mode because the amount of energy generated per step is more affected by the way the heel strikes the ground, i.e., the insole is differently bended from one step to another. In these cases, the required threshold can be reached after 4 or 5 steps, but the counter of taken steps is not able to detect that more than three steps were taken, and this is the case when the error occurs. Despite this, the error rate is less than 4% in the worst case, which is lower than that of the battery-powered systems.

3.3 Discussion

A sensor system operates transiently when it is governed by the intermittent energy supply. Because transient systems are not able to operate continuously, it is necessary to analyse the characteristics and requirements of an application, in order to decide whether a transient device can properly operate. In this Chapter, some scenarios were listed, where transient devices work, may work or are not targeted to operate. A list of feasible application was proposed, including some challenges that have to be solved. From that list, an application was chosen to be implemented, which was a *Task-based* Transient Step Counter.

The Step Counter was implemented to operate directly powered by a ferroelectret insole, i.e. without adding any battery or extra capacitance. Although the insole was characterized with different resistances and capacitances, the design process concluded that no extra circuitry (i.e. MPPT or impedance matching) or capacitance was needed to enable the system. The proposed step counter was experimentally validated and its performance compared with two smartphone applications, presenting a maximum error rate of 4%, which is lower than that of the applications. The microcontroller was not affected by the cold start circuit placed after the decoupling capacitance and the energy stored in that capacitor was enough to power the MCU during the time needed to be initialize, validate a step and increment the counter variable, keeping a better accuracy compared with existing smartphone applications.

However, expanding the capabilities not only for the step counter but also for other feasible applications for transient systems, is required. As discussed in Section 2.4.1, existing transient systems are not able to retain external peripheral state between power outages. Moreover, they cannot perform *atomic* tasks (e.g. wireless transmission) and long-term computation (e.g. filtering or encrypting) without adding extra capacitance. Besides these drawbacks, keeping track of time in transient systems is not an issue that has been solved yet. Sensor systems typically need time to properly perform their functions, e.g. the step counter needs time to calculate the step rate or to distinguish when the user begins or ends their physical routine. In the following chapters, the research contributions to overcome these challenges are detailed.

Chapter 4

Retaining External Peripheral State in Transient Sensor Systems

This chapter proposes **RESTOP** (**RE**taining the **ST**ate **O**f **P**eripherals), a novel middleware to retain the state of digitally interfaced peripherals in transiently-powered systems. RESTOP provides functions to read data from the external peripherals or write to them, keeps track of and saves the transmitted configuration data, and hence retains the peripheral state. Thus, after a power failure, the peripheral state can be restored, without requiring for the user to implement the save and restore functions for each attached peripheral and indicate the order in which they have to be restored. This chapter also presents a thorough practical evaluation of RESTOP in order to validate the operation of the middleware and the time and energy overhead it causes in an intermittently-powered sensor system.

4.1 Inefficiencies in Existing Approaches

External digital peripherals are typically connected to the MCU via serial protocols, unlike the on-chip peripherals that are controlled by special function registers. In Chapter 2.2, the drawbacks and existing challenges of transiently-powered sensor systems when working with external peripherals were described. Most of proposed approaches are focused on the processing unit, ignoring possible coherence errors that may occur between the system state and the configuration of peripherals, from one power cycle to another.

A recent approach called *Sytare*, is able to retain the state of external peripherals but only those that work with SPI protocol. Moreover, the user has to create the structure to retain the state of each attached peripheral and the function to restore their configuration after a power failure, joined with the excessive time overhead (up to 137%) caused by the need of checkpointing after each peripheral instruction is issued. Besides that, *Sytare* cannot be included in any of the off-the-shelf transient computing approaches, because it was designed specifically to work with their own solution. Therefore, this approach has not efficiently solved the problem of transiently-powered sensor systems when working with peripherals that communicate through SPI protocol, and has not proposed any solution for I^2C protocol.

An approach to retain the state of external peripherals, has to be as transparent as possible to designers, because it is not desirable to increase the complexity in the way the user accesses to these devices. It needs to act as a middleware between the main application and external peripherals, keeping the order in which each instruction are issued. Moreover, it has to be able to work with any of the existing *system state retention* approaches, in order to take advantage of their capabilities of protecting systems from volatility. Additionally, designers cannot have different procedures and functions to access an SPI peripheral, than to access an I^2C one, because that is impractical and confusing.

Therefore, retaining the state of external peripherals implies not only to implement a middleware but also a research process in order to define how to address each requirement without losing generality and causing minimal overhead. In the following sections, each element of the proposed middleware is presented, including the design process and a detailed explanation of its functionality.

4.2 RESTOP: A Middleware for External Peripheral State Retention

In order to address the inefficiencies and limitations of existing approaches when working with digitally interfaced peripherals, a middleware, so called RESTOP, is proposed, which is able to operate with different peripherals and serial communication protocols (e.g., SPI or I^2C), capable of retaining (saving and restoring) peripheral configurations between power failures. The following terms are introduced here to aid understanding of the operating principles of RESTOP:

- **Peripheral operation**: The action to be performed on the peripheral, i.e., write or read.
- **Peripheral instruction:** The information required by the system to issue the operation on the peripheral (e.g., peripheral address, register to be read, value to be written on the register, etc.).
- **Parameters:** Elements that constitute each function that executes the peripheral instructions.

Figure 4.1 shows the parts that make up RESTOP and how this middleware interacts in a sensor system when saving and executing a peripheral instruction (the Restore module is later described in Figure 4.3). Each peripheral instruction is issued through the generic functions provided by RESTOP and saved in a history table. In order to execute the instruction on the peripheral, RESTOP complements the information entered through the generic functions with that defined in a configuration file (these modules are detailed in Section 4.3). The history table can either be: (1) placed in main memory; or (2) directly located in NVM.

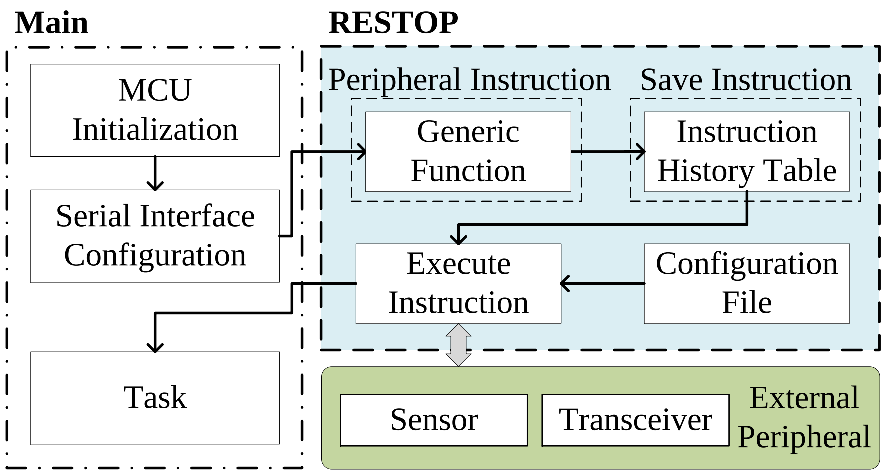


Figure 4.1: Diagram of RESTOP interacting with the application and external peripherals.

In the first case, the developer can utilize any of the existing approaches for transient computing [129, 3, 8, 117] that can save the system state (including main memory) to NVM at the right time before a power failure. Thus, after a power outage, the system state (including the instruction table) is restored and then RESTOP restores the peripheral configuration by re-issuing the instructions from the table. In the second case, RESTOP has to be included with interrupt-based approaches such as Hibernus [3] and QuickRecall [7] in order to ensure that the system and peripheral states are restored in the same point where they were interrupted, i.e., there is no more code executed after the snapshot.

Therefore, it is possible to maintain coherence, avoiding the table being modified after the last snapshot was saved. This is important because in a transient system with external peripherals, repeated peripheral instructions (or system code) may result in functionality issues [34]. In Section 4.2.1, the different factors that RESTOP considers before saving and executing a peripheral instruction are described. Later, in Section 4.2.2, the process of restoring the peripheral configuration is detailed.

4.2.1 Saving and Executing Peripheral Instructions

Figure 4.2 details the process of saving and executing a peripheral instruction issued over a serial protocol. The decision about whether RESTOP should save an instruction in the table is made by the programmer at design time for each peripheral instruction, considering four choices:

1. **Not-save**: The user might consider that a certain instruction should not be saved because it is not a peripheral configuration instruction (e.g., reading a status register).
2. **Save**: The issued instruction must be saved in the history table without checking whether a similar instruction (i.e., with same peripheral address and register value) was previously saved.
3. **Save-but-replace**: The issued instruction would replace any other similar instruction (i.e., same peripheral address and register value) that was previously saved in the history table.
4. **Preserve**: An instruction has to be kept in the history table regardless of whether a similar instruction is later issued.

As shown in Figure 4.2, RESTOP first checks whether the issued instruction is applying a *Reset* on the peripheral. *Reset* is a *write* peripheral instruction that, when issued, causes RESTOP to delete all instructions saved in the table for that peripheral. This condition is important for efficient memory usage because it is unnecessary to keep peripheral instructions prior to a *Reset*.

Next, RESTOP checks whether the issued instruction has to be saved. If not (*Not-save*), RESTOP executes the instruction on the peripheral and the application continues to the next task. If the peripheral instruction must be saved, RESTOP considers two choices: *Save* and *Save-but-replace*. If the first option is asserted, the issued instruction is saved in the table and then executed on the peripheral. In case that *Save-but-replace* is selected, the middleware looks in the table for a similar instruction previously saved.

If a similar instruction is found in the table, RESTOP checks whether the instruction is marked to be preserved (*Preserve*). If so, the instruction is saved in a new element in the table and then executed. Preserving an instruction, instead of replacing it, is particularly useful for certain peripherals that need an *unlock* instruction, which enables the peripheral to be accessed or configured. Thus, each unlock instruction is saved in the table no matter how many times it is repeated. If *Preserve* is not asserted, the previous instruction is deleted and the issued one is saved instead, but keeping the chronological order in which the instructions are sent. Keeping track of the instruction sequence is important because peripherals often need the registers to be accessed in a certain order to operate properly (e.g., in a transceiver, it has to set the channel before transmitting the data, not the other way around).

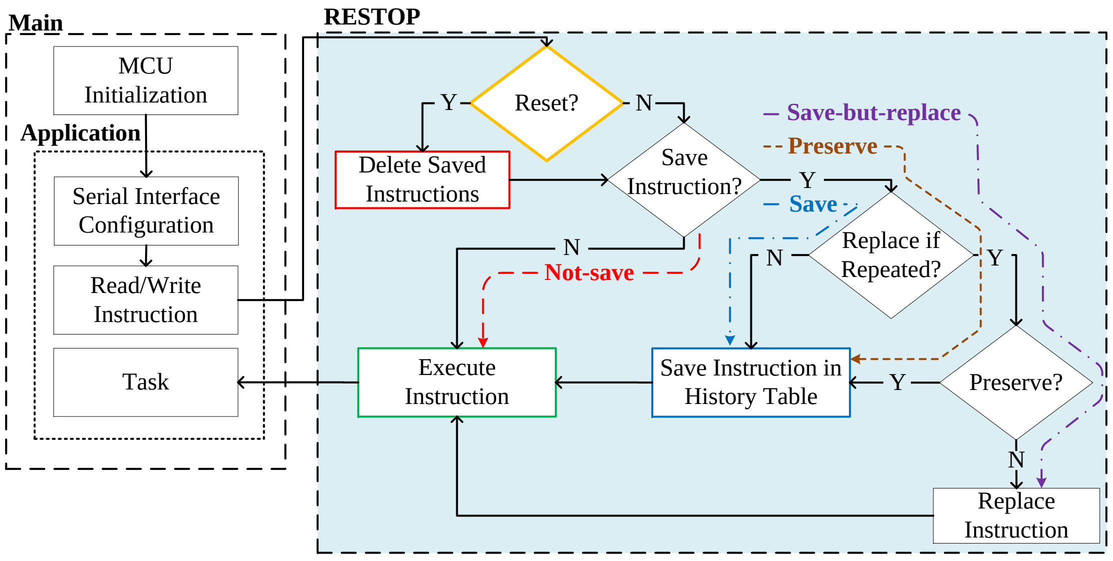


Figure 4.2: Path followed to save and execute an instruction, based on the selected criteria.

It is important to mention that each instruction must be saved before executing it in order to cope with power failures occurring before peripheral access is completed, avoiding consistency issues [141]. Issuing an instruction on a serial interface involves, among other things, enabling the peripheral, sending the register to be read or written, waiting for the transmission to be completed, and disabling the peripheral. This sequence has to be completed without interruptions, i.e., if a supply failure occurs while an instruction is issued, the sequence has to be restarted from scratch when power recovers (e.g., it is not possible to send a partial packet).

Therefore, if the instruction was not saved into the history table before the power failure, it would neither be properly executed because the sequence was interrupted, nor restored because it was not saved. A possible concern is that, when the peripheral is a radio transceiver, the user may send the instruction with the packet to be transmitted, but there would be no certainty that it was sent (i.e., a power outage may occur before packet transmission has completed). When restoring the transceiver state, the packet would be resent, leading to a duplicate packet being received. However, this can occur normally in a noisy wireless network, and communication protocols are typically already present to ignore duplicate packets and request those missed.

4.2.2 Restoring Peripheral State

The restore routine is shown in Figure 4.3. This is executed after the system state has been restored by the transient computing approach used to protect the system from volatility. Here, RESTOP fetches each instruction from the history table and issues it over the digital interface in the correct order, i.e., in the same order in which the user issued each peripheral instruction. This process is repeated until all saved instructions are executed, and, therefore, the state of the peripheral is restored. It is important to mention that some peripherals may need some delays from one instruction to another, in order to be properly configured. This delays have to be kept when restoring the peripheral configuration after a power failure. If this is the case, the delays can be easily added after the required instruction of the proper peripheral in the restore routine of RESTOP. Once the routine is completed, the main application continues its execution from the point where it was interrupted by the power outage.

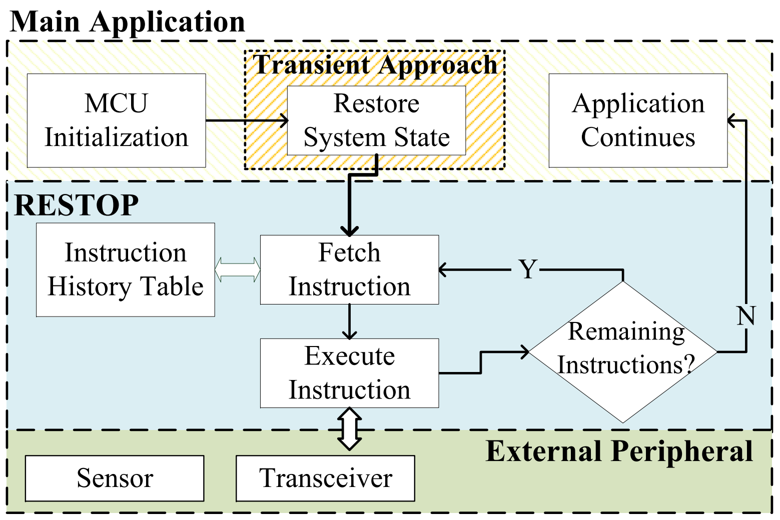


Figure 4.3: RESTOP has to re-issue each saved instruction to restore the peripheral state, after a power outage.

4.3 Software Algorithm Design

The requirement for a generic interface implies that it can work across different protocols and handle different types of instructions, and that it fits in not only with programming structures but also with the use cases of transient technologies. In this section, it is detailed the RESTOP functions that will execute the peripheral instructions (Section 4.3.1). Then, the parameters that have to be saved to properly describe each peripheral instruction without losing generality are described, as well as how the users will introduce the required information for each instruction (Section 4.3.2). Lastly, it is explained how the instruction history table will be efficiently built in terms of time and memory usage (Section 4.3.3).

4.3.1 Function Implementation

Defining the functions to execute each peripheral configuration instruction, and the information to be saved from them, required the analysis of various peripherals with digital interfaces, identifying patterns that help to implement the RESTOP functions that are generic for different peripherals and serial protocols. This analysis concluded that peripheral instructions perform two main operations: read and write. However, the number of parameters required to perform these operations varies from one peripheral to another.

For example, some peripherals [142] need a 1-byte parameter called *command* to indicate the type of operation the issued instruction will perform (i.e., *read* or *write*) on a register address (i.e., the sequence would be <command byte><register address><data byte>). Others allow certain *single byte instructions* (no data is transferred), usually so-called *command strobes* that cause internal sequences to start in the peripheral, e.g., some peripherals have a single header byte that, when addressed, starts a self-calibration routine to define the sampling frequency [142]. Most peripherals support multiple byte transfers also known as *data burst transmissions* which send first the register address and then a sequence of different values to write to this address (this can also be applied for read operations).

Considering this analysis, it is possible to define the functions that will be used by the middleware to save, execute and restore each instruction:

1. RESTOP_read(): Returns the read value from the desired register.
2. RESTOP_write(): Writes a value into a peripheral register.
3. RESTOP_strobe(): Performs write operations that, unlike RESTOP_write(), executes single byte instructions.
4. RESTOP_restore(): Restores the peripheral state by executing all the instructions saved in the history table after a power failure. It has to be incorporated into the restore routine of the main application.

These functions have to be used by the developer to configure the peripherals and obtain data from them. The parameters of each function are described in Section 4.3.2.

4.3.2 Parameters to be Saved and Configuration File

Following the definition of the generic functions, the parameters that will constitute each peripheral instruction need to be defined. For this purpose, the parameters are classified as *dynamic*, if they vary from one peripheral instruction to another, and *static*, if they are constant for each peripheral attached to the system. Table 4.1 shows the *dynamic* parameters that will be saved in the history table. The size (number of bits) of each parameter varies depending on the information that they contain.

The first parameter (*Protocol*) is a 1-bit flag to indicate the serial protocol type of each peripheral (0→SPI; 1→I^2C). Parameter *Burst* is also a 1-bit flag that has to be set to 1 when the function will execute a *burst read/write* instruction. *Read* is a flag used by RESTOP to distinguish when the instruction is for a *read* (R=1) or *write* (R=0) operation. *Prv* is a 3-bit flag that can have five different values as shown in Table 4.2.

The values listed in Table 4.2 are defined following the criteria described in Section 4.2.1. Thus, the first three options indicate whether the instruction will not be saved in the table (*Not-save*), will be saved in a new element (*Save*) or will replace a similar one if it was previously saved in the table (*Save-but-replace*). The last two criteria, shown in Table 4.2, indicate the instruction will be not only saved but also preserved in the table regardless of whether a similar instruction is later issued (*Preserve*).

Table 4.1: Dynamic parameters needed to describe a peripheral instruction.

Parameter	**Size (Bits)**	**Definition**
Protocol	1	Serial Protocol of Peripheral
Burst	1	Burst instruction
Read	1	Read or write instruction
Prv	3	Preserve flag
ID	3	Peripheral identification
Register	8	Address to be accessed
Value	8	Value to be written in the register
Next	8	Pointer to the next instruction
Previous	8	Pointer to the previous instruction

The parameter *ID* is used to indicate the peripheral to which the saved instruction corresponds. A system may have more than one peripheral attached to the MCU, hence, it is necessary to identify which instruction corresponds to each peripheral. The parameters *Register* and *Value* are each one byte, corresponding to the register width of typical digital interface peripherals. *Next* and *Prev* are used to keep track of the order in which the instructions are issued.

Thus, when a new instruction replaces another previously saved or is added in a new element in the table, RESTOP can keep the chronological sequence in order to properly restore the peripheral state after a power outage. The size of these parameters is one byte each too, allowing the system to map up to 256 peripheral instructions. This is considered sufficient for most peripherals (e.g., a typical transceiver [124] is configured with less than 50 instructions), but their size could be expanded for particular scenarios, where multiple peripherals are attached and the number of saved instructions exceeds 256.

Table 4.2: Values that *Prv* flag can have.

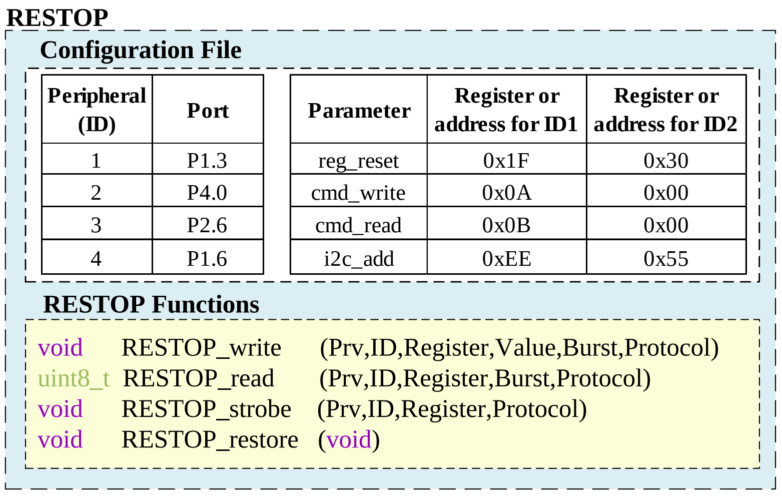
Bit 2	**Bit 1**	**Bit 0**	**Criteria**
0	0	0	Not-save
0	0	1	Save
0	1	0	Save-but-replace
1	0	1	Save and Preserve
1	1	0	Save-but-replace and Preserve

In the case of the *static* parameters, four are defined:

- reg_reset: To declare the register address that represents a reset instruction in each peripheral.
- cmd_write: This parameter is used to introduce the *write command* value for peripherals that need it as explained in Section 4.3.1.
- cmd_read: This is similar to the previous one, but this is the command for reading operations. If no *command* is needed in a peripheral, it has to be filled with zeros.
- i2c_add: This parameter is used to define the address of the peripherals that are attached to the MCU through I^{2}C protocol.

The *static* parameters are declared in a configuration file unlike the *dynamic* ones, which are saved in the history table and entered by the user through the generic functions (except *R*, *Next* and *Previous*, which are defined by RESTOP). To reset a peripheral, the user not only has to use the *RESTOP_write()* function but also declare the register address in *reg_reset*.

As mentioned in Section 4.2.1, *Reset* is a *write* instruction that when issued causes RESTOP to delete the saved instructions that correspond to the reset peripheral. *cmd_write* and *cmd_read* have to be filled with zeros for those peripherals that do not need a *command* parameter (described in Section 4.2.1). If an I^{2}C peripheral is attached to the system, its address has to be written in *i2c_add*, if not, this parameter has to be filled with zeros as well. Figure 4.4 shows the *configuration file* with example values and the description of the *dynamic* parameters that each generic function requires. The *configuration file* also includes the microcontroller ports where the peripherals are attached. For example, if a user connects a peripheral to port P1.3, it will be marked with the peripheral identification ID1.



Peripheral (ID)	Port
1	P1.3
2	P4.0
3	P2.6
4	P1.6

Parameter	Register or address for ID1	Register or address for ID2
reg_reset	0x1F	0x30
cmd_write	0x0A	0x00
cmd_read	0x0B	0x00
i2c_add	0xEE	0x55

```
void     RESTOP_write    (Prv,ID,Register,Value,Burst,Protocol)
uint8_t  RESTOP_read     (Prv,ID,Register,Burst,Protocol)
void     RESTOP_strobe   (Prv,ID,Register,Protocol)
void     RESTOP_restore  (void)
```

Figure 4.4: Configuration file with example values and the functions description.

4.3.3 Instruction History Table

As already mentioned in Section 4.3.2, RESTOP requires an *instruction history table* to which it can save the data exchanged between the MCU and the peripherals. The important factors to be considered when implementing software approaches for transiently-powered systems, are the time, energy and memory overheads proposed solutions may cause. In a resource-constrained EH system, it is not desirable for the MCU to spend more time executing RESTOP instead of running the task for which the sensor system was designed. Therefore, an efficient method has to be found that allows to implement the table where the instructions will be saved, with a minimum overhead.

The required instruction history table is based on a linked list. This list can be implemented by allocating memory dynamically (using the function *malloc* to allocate memory from "heap" [143]) or statically. Allocating memory dynamically to implement the instruction table, may incur an unnecessary waste of time and memory [144, 145]. A one-byte variable can point up to 256 saved configuration instructions. The transceivers are the peripherals that need a greater number of configuration instructions (more than 30), while digital sensors can be configured with less than 10. Thus, one byte can address more than five times the number of instructions required by a typical transceiver. Therefore, implementing the instruction history table with a static array can simplify the problem.

These two methods were implemented and executed in an MCU [91], to look for the most efficient to build the instruction history table. Figure 4.5 shows the comparison in terms of time taken to save different number of instructions in the list. It is possible to see that the main difference between these two methods occurs when saving only one instruction, with the static allocation method being faster. As the number of instructions to be saved increases, the difference in time reduces, although, allocating memory dynamically is still slower than statically.

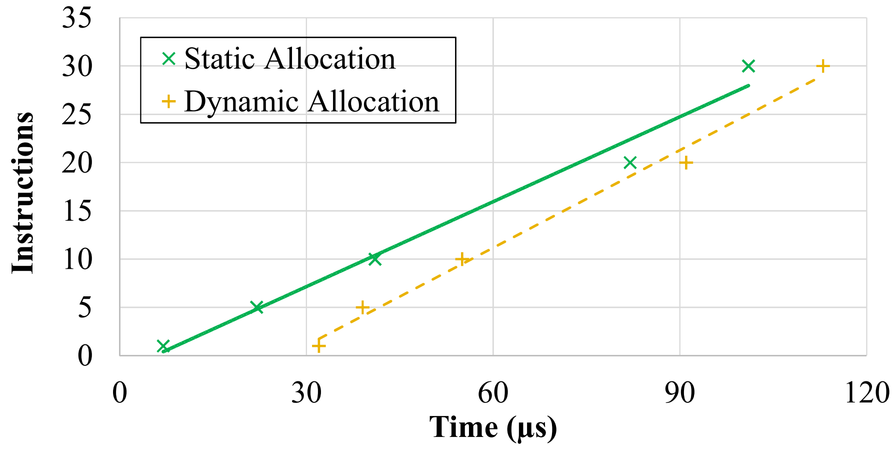


Figure 4.5: Time overhead comparison between the static and the dynamic memory allocation.

Another experiment executed, to compare the performance of both methods, consisted of measuring the time needed to save a new instruction and to search and replace a repeated instructions when the table contains 5, 10 and 30 blocks. Table 4.3 shows the obtained results. Here, it is possible to see that the difference when searching and replacing a repeated instruction decreases as the number of blocks in the table increases. However, the highest difference occurs when saving an instruction for the first time (i.e. when the list is first created in the dynamic method), in which the dynamic list is up to four times slower than the static list.

Table 4.3: Time spent by each method when saving for the first time, and when searching and replacing a repeated peripheral instruction.

Method	**First Instruction (μs)**	**Search in a List of:**		
		Five Blocks (μs)	**Ten Blocks (μs)**	**Thirty Blocks (μs)**
Dynamic List	31	44	63	126
Static List	8	23	46	115

Another concern with dynamic memory allocation is when the system has enough free memory for a required heap but is not available in a contiguous block [144, 145]. Figure 4.6 shows an example of this memory fragmentation. There is a heap of 160 bytes and first it is allocated an area of 60 bytes. Then, another 40 bytes are allocated, remaining 60 bytes free. If the first 60 bytes were released, 120 bytes would be available in two chunks of 60 bytes each. However, if a new request for 80 bytes is issued, the function call would result in an error. Therefore, static memory allocation is used to implement the table for saving instructions in RESTOP.

The maximum size of the table (i.e., the number of available locations where the instructions are saved) is defined by the user in the *configuration file*. Figure 4.7 shows

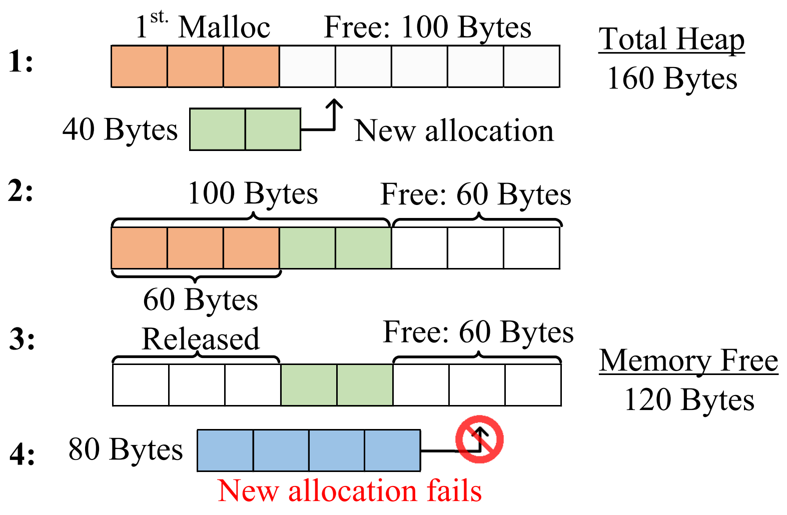
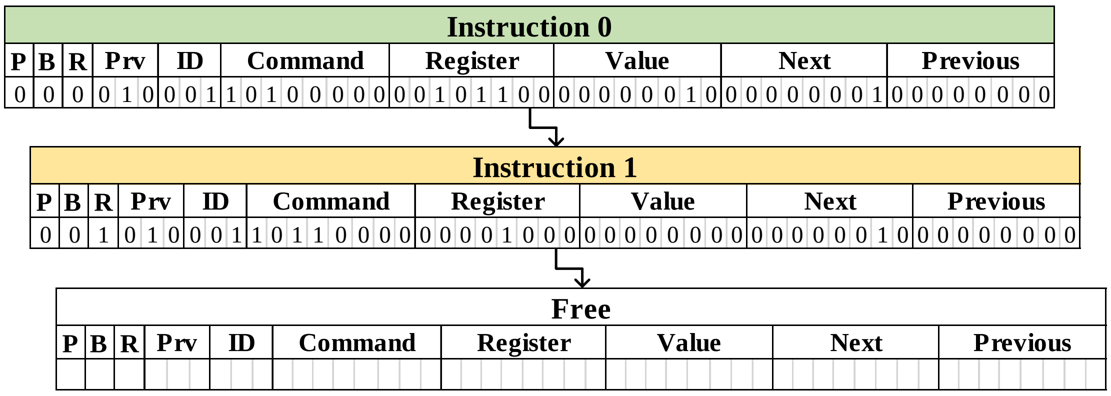


Figure 4.6: Memory fragmentation when using dynamic memory allocation.

an example of the history table with two saved instructions. From the values showed in the figure, the *static* parameters and the generic functions for the two saved instructions would be as follows:

- reg_reset [] = {0x1F}
- cmd_write [] = {0x0A}
- cmd_read [] = {0x0B}
- i2c_add [] = {0x00}
- RESTOP_write(2,1,0x2C,0x02,0,0)
- RESTOP_read(2,1,0x08,0,0)

In this example, the value of the *Protocol* flag ($P = 0$) indicates both instructions correspond to the same SPI peripheral, which is connected to the P1.3 port ($ID = 1$). Therefore, the parameter *i2c_add* is filled with zeros. The *Burst* flag (B) is zero which means these are not *burst* instructions. According with the *Read* flag (R), the first instruction is to *write* on the peripheral ($R = 0$) and the second is to *read* from it ($R = 1$). The *Prv* flag value is 2 in both saved instructions, which means that they would replace any similar instruction (same peripheral, command and register) previously saved, but they can also be replaced if a similar instruction is later issued (*Preserve* bit = 0). The attached peripheral needs a *command* to indicate when the instruction is to write (0x0A) and when to read (0x0B). If an instruction was issued with a register value of 0x1F, RESTOP would apply a reset in the peripheral and delete the two instructions from the table.



Instruction 0									
P	B	R	Prv	ID	Command	Register	Value	Next	Previous
0	0	0	0 1 0	0 0 1	1 0 1 0 0 0 0 0	0 0 1 0 1 1 0 0	0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0

Instruction 1									
P	B	R	Prv	ID	Command	Register	Value	Next	Previous
0	0	1	0 1 0	0 0 1	1 0 1 1 0 0 0 0	0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0

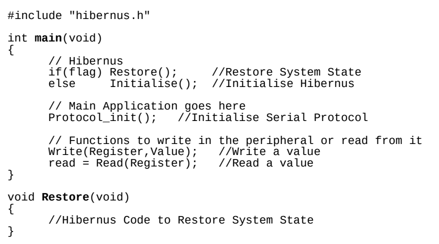
Free									
P	B	R	Prv	ID	Command	Register	Value	Next	Previous

Figure 4.7: Instruction history table of two saved instructions.

4.4 Experimental Validation

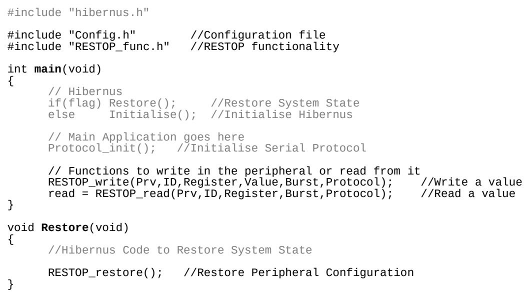
RESTOP has been practically implemented and experimentally validated. To allow computation to span across power cycles, RESTOP is combined with Hibernus [3]. This solution was chosen because it is platform and application agnostic, and has excellent performance in terms of energy and time overhead [43]. However, RESTOP can be integrated with any other software approach for transient computing.

Figure 4.8 shows an example application before (Figure 4.8(a)) and after (Figure 4.8(b)) incorporating the proposed middleware. The example code includes Hibernus to retain the system state between power outages. The inclusion of RESTOP in an application is simple. The developer only has to import the configuration file (*Config.h*) and the library that contains RESTOP functionality (*RESTOP_func.h*), and use the RESTOP functions (described in Section 4.3.1) to configure the peripherals and read data from them. In order to restore the peripheral state after a power outage, the RESTOP restore function has to be included in the restore routine of the transient approach as shown in Figure 4.8b.



```
#include "hibernus.h"

int main(void)
{
      // Hibernus
      if(flag) Restore();     //Restore System State
      else     Initialise();  //Initialise Hibernus

      // Main Application goes here
      Protocol_init();   //Initialise Serial Protocol

      // Functions to write in the peripheral or read from it
      Write(Register,Value);   //Write a value
      read = Read(Register);   //Read a value
}

void Restore(void)
{
      //Hibernus Code to Restore System State
}
```

(a) Application without RESTOP.



```
#include "hibernus.h"

#include "Config.h"        //Configuration file
#include "RESTOP_func.h"   //RESTOP functionality

int main(void)
{
      // Hibernus
      if(flag) Restore();     //Restore System State
      else     Initialise();  //Initialise Hibernus

      // Main Application goes here
      Protocol_init();   //Initialise Serial Protocol

      // Functions to write in the peripheral or read from it
      RESTOP_write(Prv,ID,Register,Value,Burst,Protocol);    //Write a value
      read = RESTOP_read(Prv,ID,Register,Burst,Protocol);    //Read a value
}

void Restore(void)
{
      //Hibernus Code to Restore System State

      RESTOP_restore();   //Restore Peripheral Configuration
}
```

(b) Application including RESTOP.

Figure 4.8: Example code of how to use RESTOP in an application, including Hibernus, showing: (**a**) code without RESTOP; and (**b**) including RESTOP.

The voltage threshold at which Hibernus restores the system state (V_R) has to be adjusted because now the system incorporates external peripherals whose state is restored as well. Therefore, it is described how V_R is modified for Hibernus, which is used in the validation (a similar modification would need to be made for other approaches). In this sense, it is necessary to first calculate the amount of energy required to restore the state of attached peripherals (E_{r_ps}), which is given by:

$$E_{r\_ps} = \sum_{i=1}^{n} \left(P_{p_i} \sum_{j=1}^{m_i} T_{p_i inst_j} \right) \tag{4.1}$$

where n is the number of attached peripherals, P_{p_i} is the power consumed by the system while undertaking serial communications with each peripheral, m_i is the number of saved instructions for each attached peripheral and $T_{p_i inst_j}$ is the time taken by the system to issue each instruction to the peripheral. These parameters (power and time) may be obtained from datasheets, or measured experimentally.

The time varies from one instruction to another depending on the data rate of each peripheral and the number of bytes that are transmitted for each instruction. In Section 4.3.1, it is detailed how the number of parameters that are transmitted for each instruction (i.e., one parameter is equal to one byte) varies by peripheral. It is important to mention that Equation (4.1) only accounts for the power consumption of the MCU. The effect of issuing the instructions may cause additional energy to be consumed by the external peripherals, e.g., a restoration of state causing a wireless transceiver to make an energy-intensive radio transmission. This is not currently modeled, but is a potential area of future investigation.

Once E_{r_ps} is calculated, and considering the energy required to restore the system state (E_{r_sys} [3]), V_R can be calculated as follows:

$$V_R = \sqrt{\frac{2(E_{r\_sys} + E_{r\_ps})}{C} + V_{min}^2} \tag{4.2}$$

where V_{min} is the minimum voltage required by the system to operate and C is the total capacitance on the supply lines, which can be used as an energy buffer. The process of calculating E_{r_ps} and adjusting V_R is performed at the beginning of the snapshotting routine, in order to guarantee that the restore threshold is properly set before a power failure occurs. Although Equation (4.1) is performed once per supply interruption, a running total of $T_{p_i inst_j}$ is updated each time a peripheral instruction is saved. This reduces the complexity of the calculation that needs to be performed at the start of the snapshotting procedure. Thus, V_R can be dynamically adjusted considering the number of saved instructions (provided by RESTOP) for each attached peripheral.

An important concern is the size of C. Transient computing schemes commonly use only the system decoupling capacitance, $C_{decouple}$ (Figure 4.9), but this could be insufficient in systems interfacing with external peripherals. It may be necessary to introduce additional capacitance to deliver reliable operation. To do this, and for design purposes only, the worst case of energy used for restoring the peripheral state (E_{r_max}) has to be calculated. In this sense, Equation (4.1) is simplified as follows:

$$E_{r\_max} = P_{p\_max} \cdot n_{inst} \cdot T_{p\_max} \tag{4.3}$$

where P_{p_max} corresponds to the maximum power consumed by the system when an instruction is issued, n_{inst} is the maximum number of instructions than can be saved in the *instruction history table* and T_{p_max} is the *longest* time taken to issue a single instruction. Once E_{r_max} is obtained, and with knowledge of the V_{min} and V_{max} (the system's maximum operating voltage), the required C can be calculated as:

$$C \geq \frac{2(E_{r\_sys} + E_{r\_max})}{V_{max}^2 - V_{min}^2} \tag{4.4}$$

If $C > C_{decouple}$, RESTOP will require additional capacitance to supplement the decoupling capacitance. However, no other hardware changes are required.

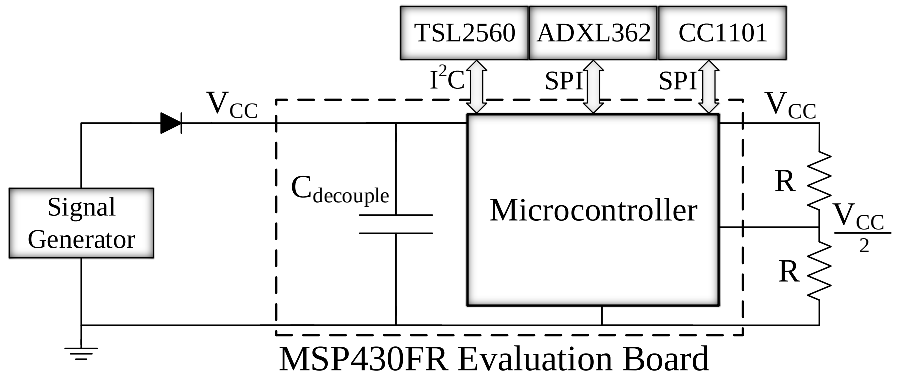


Figure 4.9: Schematic of the test platform, including the external peripherals.

Figure 4.9 shows the experimental circuit which consists of a test board and three peripherals. The chosen board was the MSP-EXP430FR5739 [91], which contains an MCU with FRAM, an on-chip comparator and supports SPI and I^{2}C communication protocols. The comparator is used by Hibernus to monitor the input voltage, and it was configured with an on-chip variable reference voltage generator and an external voltage divider (R = 1 MΩ) giving $V_{CC}/2$ as input. Three different external peripherals were considered: an ADXL362 digital accelerometer [142], a TSL2560 digital luminosity sensor [123] and a CC1101 radio transceiver [124]. Figure 4.10 shows the complete set-up implemented to validate the proper functionality of RESTOP.

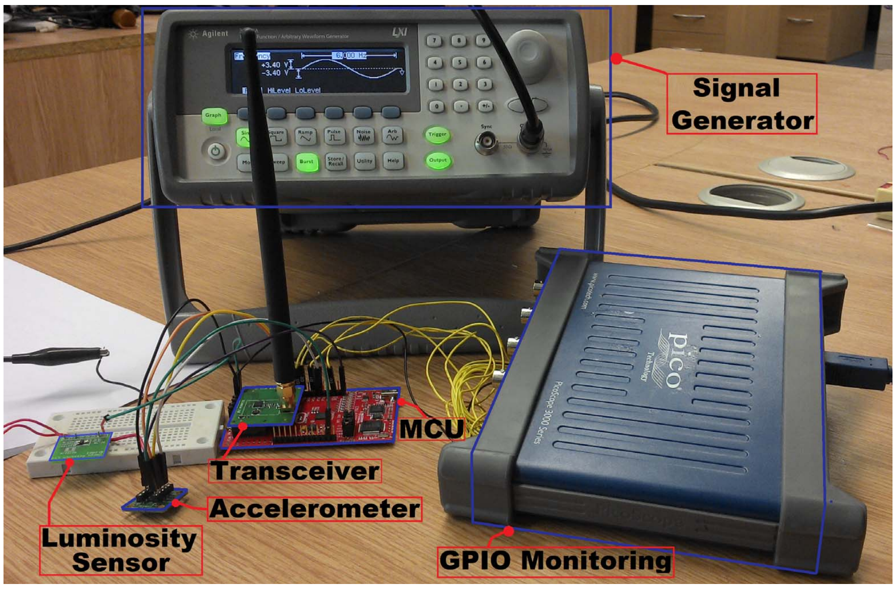


Figure 4.10: Experimental set-up to validate RESTOP, including the MCU, external peripherals, signal generator and the PicoScope used to monitor the program status.

The accelerometer and the transceiver are attached to the MCU through SPI, while the luminosity sensor is accessed via I^2C. Each peripheral was tested separately (i.e., only one peripheral is used for each of the tests), giving three different scenarios in total. $C_{decouple}$ represents the total decoupling capacitance of the board, which is 20 μF. To verify whether additional capacitance was needed, the worst case energy use was evaluated for each of the attached peripherals and the minimum capacitance needed for each scenario; the results are listed in Table 4.4.

From the values presented in Table 4.4, $C_{decouple}$ is sufficient for all cases, and hence no additional capacitance is needed. The whole system was powered by two different signals:

1. A half-wave rectified sinusoidal signal with ±3.4 V amplitude operating at a frequency of 6 Hz, to emulate an intermittent source, in order to validate whether RESTOP is able to retain the peripheral's state between power failures.

2. A square wave signal with 3.4 V amplitude and variable duty cycle, sweeping the active time from 10 ms to 100 ms, to measure the time and energy overhead caused by RESTOP with respect to the total application execution time.

The aim of these variable signals is to emulate intermittent sources. Behaviour with a real EH source was not evaluated, as this has already been demonstrated for Hibernus-based systems in [3, 43, 8].

Table 4.4: Worst case of energy consumed by each external peripheral, and the minimum capacitance needed.

Peripheral	**E_{r_max} (μJ)**	**C (μF)**
Accelerometer	1.40	1.58
Luminosity	2.87	2.76
Transceiver	9.37	3.36

4.4.1 Accelerometer

To validate the proper operation of RESTOP with the accelerometer, an application was implemented, which changes the output data rate (ODR) to reduce the current consumption of the sensor. As shown in Figure 4.11, after configuring the SPI protocol, the accelerometer is reset and the ODR is set to 50 Hz (ODR = 0x02). Then, the accelerometer is configured in measurement mode and the application enters in a loop where the three axes are read to detect movement at each iteration.

During the time the program is running inside the loop, a voltage drop occurs and the snapshotting routing of Hibernus is called. There, E_{r_ps} is calculated using Equation (4.1) and the obtained value is 0.6 μJ. Substituting it in Equation (4.2) and considering $E_{r\_sys} = 5.7$ μJ [3], the new restore threshold is set to 2.15 V. After V_R is adjusted, the system state is saved in NVM.

Later, when the power is restored, an ODR reading is taken before and after restoring the accelerometer state. This step was purely for testing purposes in order to check RESTOP operation: the ODR reads would not be needed in a real application. Thus, the ODR value read before restoring has to be the default (ODR = 0x03), whilst the value after restoring has to be the same as before the power failure (ODR = 0x02). As shown in Figure 4.12, the value read before restoring the peripheral state is the default (ODR = 0x03), but once it is restored, the ODR value is the same as before the interrupt. This demonstrates that RESTOP is able to properly restore the accelerometer state.

4.4.2 Luminosity Sensor

RESTOP was tested with a luminosity sensor to validate the proposed middleware with an I^{2}C peripheral. Figure 4.13 shows the test algorithm, which consists of initializing the MCU, configuring the I^{2}C protocol, and then changing the integration time (T_{int}) from the default value (400 ms) to 13.7 ms. T_{int} defines the time after which the ADC channels begin a conversion. Once the integration time was changed, an end-of-conversion signal is configured in order to generate an interrupt when an ADC conversion is completed.

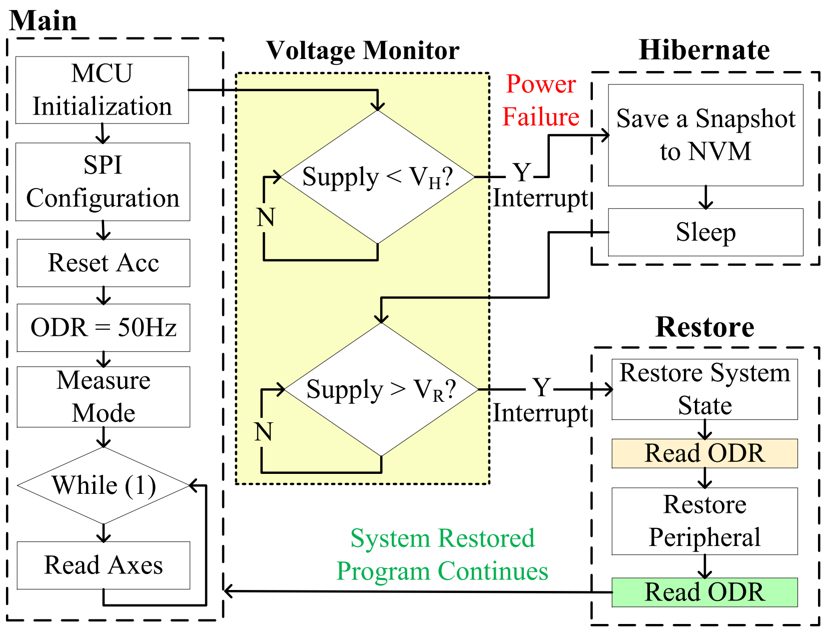


Figure 4.11: Testing routine to validate RESTOP with the accelerometer.

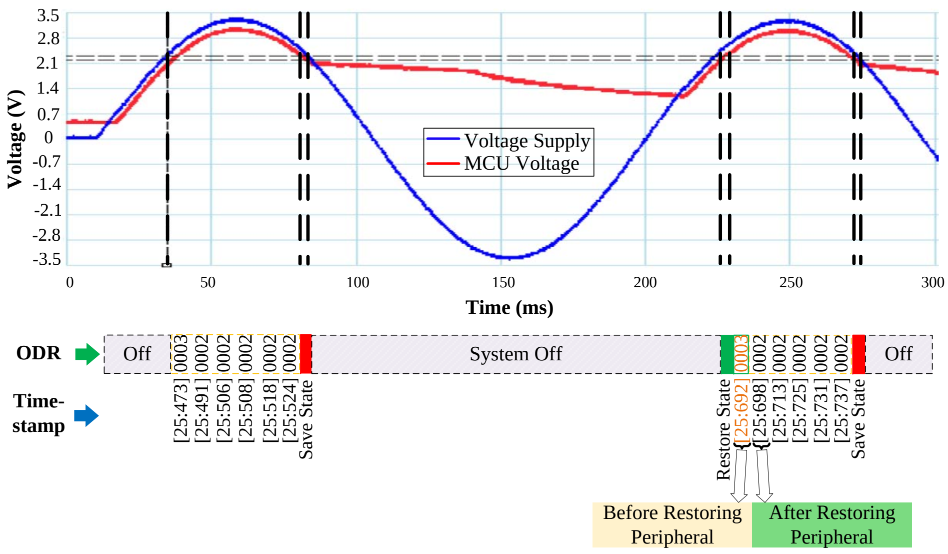


Figure 4.12: Operation of the accelerometer testing routine. After the power failure, ODR is read before and after RESTOP restores the accelerometer state.

Thus, the light intensity value is available in the data registers after 13.7 ms. Figure 4.14 shows the experimental results. After configuring the sensor, the light intensity is continuously read until the input voltage drops and the snapshotting routine is executed. In the same way as with the accelerometer, E_{r_ps} and V_R are calculated. However, for the luminosity sensor, the minimum operating voltage is 2.6 V, which is then defined as V_{min} in Equation (4.2) (unlike the accelerometer's, which is 2 V); therefore, the obtained values for E_{r_ps} and V_R are 1.72 μJ and 2.74 V, respectively.

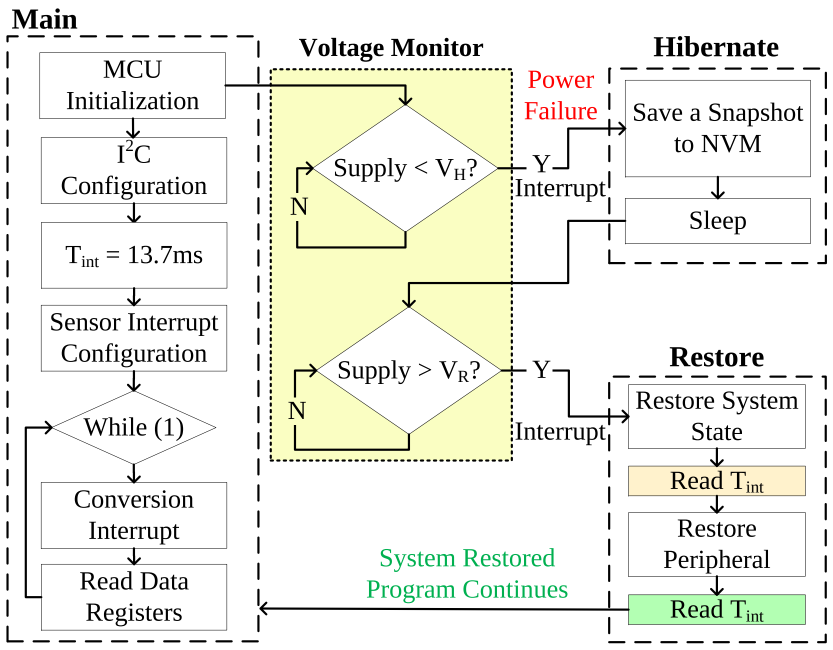


Figure 4.13: Testing routine to validate RESTOP with the luminosity sensor.

To validate the proper operation of RESTOP, the integration time register is read before and after restoring the peripheral state. As shown in Figure 4.14, the value read before restoring the peripheral configuration is T_{int} = 0x02 corresponding to an integration time of 400 ms. Then, when the peripheral state is restored, the value read is T_{int} = 0x00, which corresponds to 13.7 ms. This can also be proved because the end-of-conversion interrupt signal of the sensor is enabled every 13.7 ms, which means the sensor's configuration was successfully restored by RESTOP.

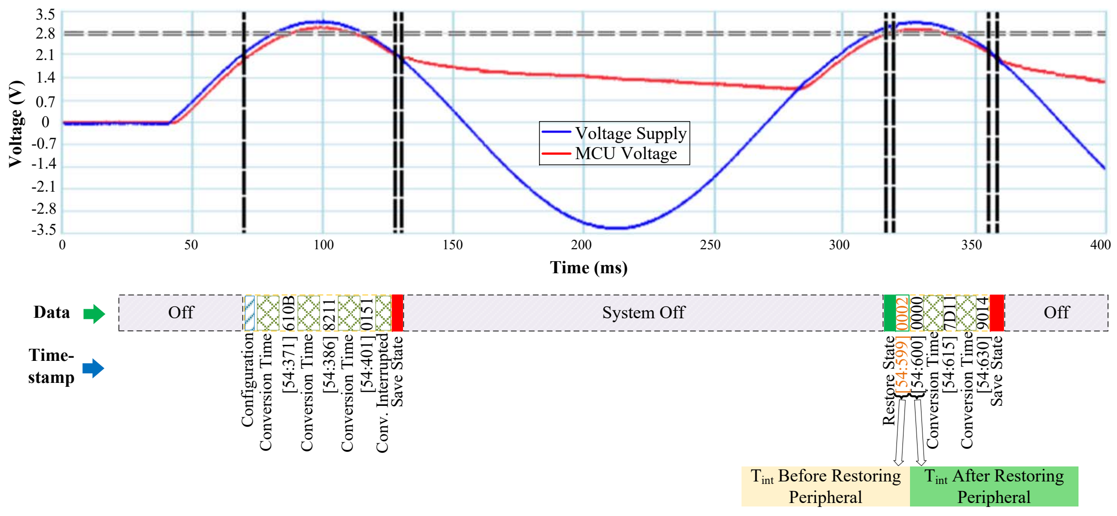


Figure 4.14: Operation of the luminosity sensor in an intermittently-powered system. After the power failure, the timing registers are configured as before the interruption.

4.4.3 Transceiver

The operation of RESTOP was also validated with a CC1101 radio transceiver. Exclusively for debugging purposes, a routine was implemented (Figure 4.15) that initializes the MCU, configures the SPI protocol, resets and configures the peripheral and then, inside an infinite loop, the program changes the transmission channel (from 0 to 20) and sends a packet at each cycle. The idea is that the channel number can be checked before the power failure, and before and after restoring the peripheral state. This is to verify whether the transceiver configuration is restored after a power outage and in consequence the channel number is retained.

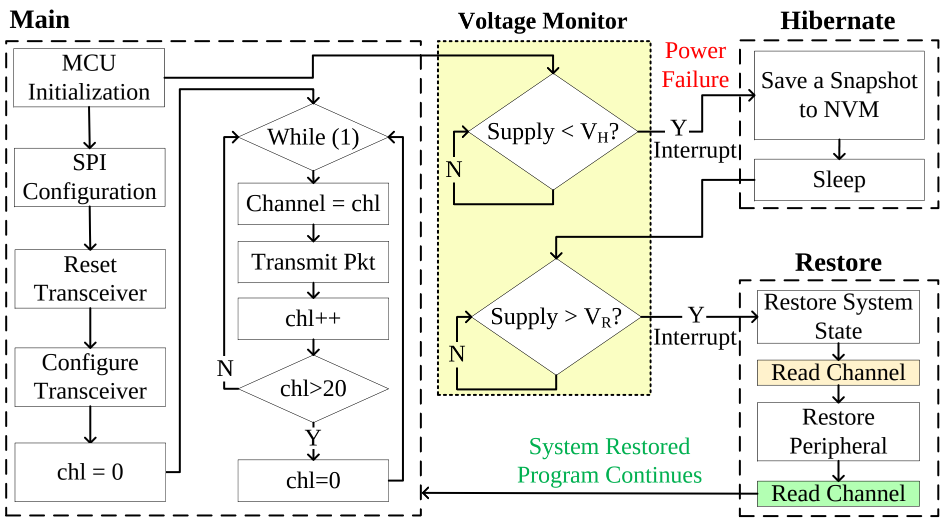


Figure 4.15: Testing routine to validate RESTOP with the transceiver.

Figure 4.16 shows the experimental results of RESTOP with the transceiver. When the input voltage drops, the channel number read is 4. Then, inside the snapshotting routine, the energy required to restore the transceiver state is calculated. The obtained value is 8.43 μJ, which is used to calculate $V_R = 2.33$ V. When the supply voltage rises above V_R, and before the peripheral state is restored, the channel number read is 0, which is the default value. Then, RESTOP restores the peripheral state and the channel number read is 4, which is the same channel as before the power failure.

4.4.4 Time Overhead

The time overhead caused by RESTOP, when executing and saving different amount of peripheral instructions, was measured. Figure 4.17 shows the comparative in terms of time when executing different amount of instruction with and without including RESTOP. In direct access, the instructions are immediately sent through the serial port to be executed in the external peripheral. The time overhead caused by RESTOP is directly related with the number of peripheral instructions to be saved, causing a maximum overhead of about 15%.

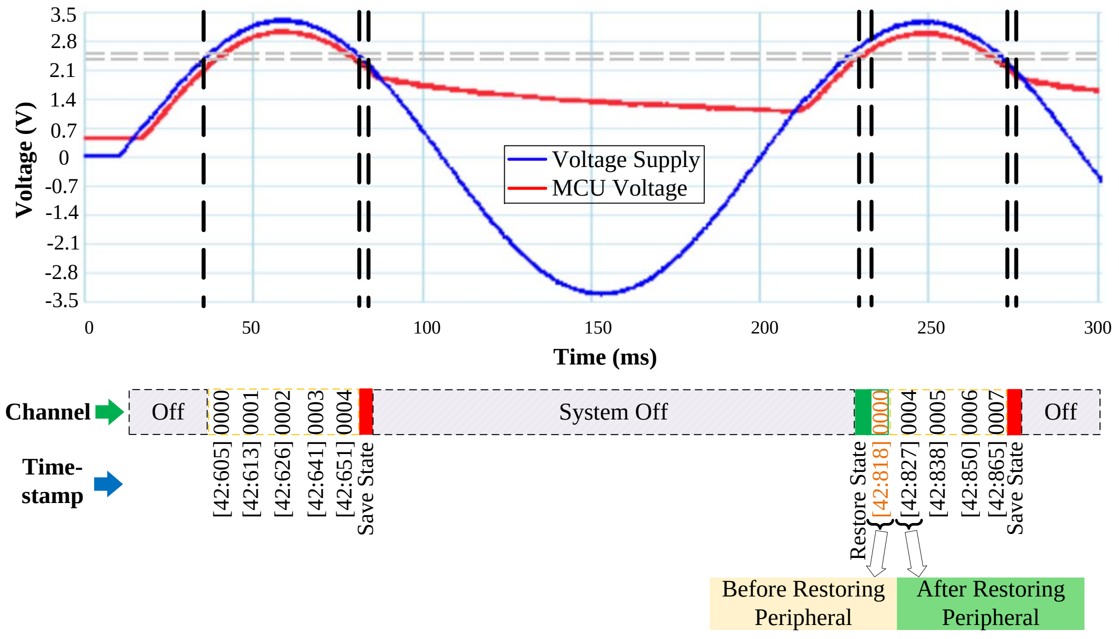


Figure 4.16: Operation of the transceiver testing routine. After the power failure, the transmission channel is read before and after RESTOP restores the transceiver state.

RESTOP was also evaluated in an intermittently-powered system with multiple external peripherals attached. Three applications were implemented, which run under two different scenarios powered by a square wave signal with 3.4 V amplitude and variable duty cycle (from 10 ms to 100 ms). In the first scenario, the peripherals are accessed without using RESTOP (restarting the peripheral's state from scratch after each power failure), while, in the second scenario, the proposed middleware is included.

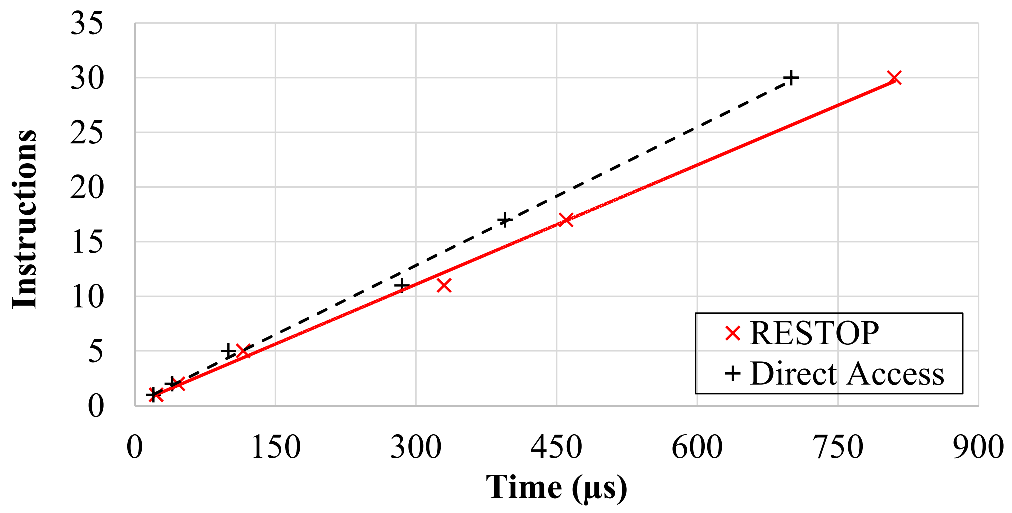


Figure 4.17: Time comparison when executing peripheral instruction with and without using RESTOP.

Each application consists of reading data sampled by the luminosity sensor and by the accelerometer (ACC) in order to process it with a Fast Fourier Transform (FFT).

Then, the sampled and processed data is transmitted through the radio transceiver. The difference in the applications is the number of samples (32, 64 and 128) that are obtained from the accelerometer and processed by the FFT.

The experimental results when accessing the peripherals with and without RESTOP are shown in Table 4.5. The proposed middleware saves and executes all the *write* instructions to configure the three peripherals and the *read* instructions to get the data from the sensors. The time taken by the FFT is the same in both scenarios because RESTOP is transparent for this task. In the case of the luminosity sensor and the transceiver, they spend the same time in all the applications because they operate only once per case, unlike the accelerometer which takes different amounts of samples.

Table 4.5: RESTOP's time overhead in a system with three external peripherals.

Active time (ms)	N° Samples	N° Executed Instructions	N° snapshot	N° restore	Time (ms)										RESTOP Overhead (%)
					FFT	Without RESTOP				With RESTOP					
						Luminosity	ACC	Transceiver	Total	Luminosity	ACC	Transceiver	Total		
10	32	80	2	2	19.71	-	0.72	1.05	26.96	-	0.78	1.21	27.18		0.82
10	64	112	6	6	47.27	-	1.43	1.05	66.19	-	1.49	1.21	66.41		0.33
10	128	176	14	14	100	-	2.97	1.05	142.38	-	3.03	1.21	142.6		0.15
40	32	85	0	0	19.71	14.1	0.72	1.05	35.58	14.16	0.78	1.21	35.86		0.79
40	64	117	1	1	47.27	14.1	1.43	1.05	66.59	14.16	1.49	1.21	66.87		0.42
40	128	181	3	3	100	14.1	2.97	1.05	126.34	14.16	3.03	1.21	126.62		0.22
70	32	85	0	0	19.71	14.1	0.72	1.05	35.58	14.16	0.78	1.21	35.86		0.79
70	64	117	0	0	47.27	14.1	1.43	1.05	63.85	14.16	1.49	1.21	64.13		0.44
70	128	181	1	1	100	14.1	2.97	1.05	120.86	14.16	3.03	1.21	121.14		0.23
100	32	85	0	0	19.71	14.1	0.72	1.05	35.58	14.16	0.78	1.21	35.86		0.79
100	64	117	0	0	47.27	14.1	1.43	1.05	63.85	14.16	1.49	1.21	64.13		0.44
100	128	181	1	1	100	14.1	2.97	1.05	120.86	14.16	3.03	1.21	121.14		0.23

Table 4.5 also presents the total time spent to complete the FFT, including the time to snapshot and restore both the system state and the peripheral configuration. The last column (at the right side) indicates the time overhead caused by RESTOP on the whole application at different duty cycles of the power signal.

As mentioned before, RESTOP causes a time overhead of about 15% when configuring a peripheral. However, this represents a maximum overhead of 0.82% during the complete execution of the proposed sensing application and it is substantially lower than the existing approach *Sytare* [119], which causes a time overhead of up to 137% (30 μs per peripheral instruction) when configuring a radio transceiver. Moreover, the time overhead caused by RESTOP will decrease further as the ratio of the peripheral instructions:normal operation, decreases.

4.4.5 Energy Overhead

The energy consumed by the system, when sending the instructions through the serial terminal (i.e. without considering the energy consumed by the peripherals) to configure two different transceivers, was also calculated. The reason to choose transceivers, is because they need more instructions to be configured than a digital sensor.

The first transceiver (CC1101) requires 45 instructions to be configured. Sending these instructions by direct access (i.e. without including RESTOP) takes about 950μs and 3.8μJ. Accessing the transceiver through RESTOP, takes about 1.1ms and consumes about 4.5μJ, i.e. 18% of overhead (0.7μJ). However, this peripheral consumes about 9.37μJ only to transmit, but the energy consumed by the MCU and the peripheral when sending and executing the instructions is up to 64.82μJ. Therefore, the energy overhead caused by RESTOP only represents 7% of the energy consumed by the peripheral when transmitting and 1% for the complete transmission process.

The second transceiver (nRF24L01) requires 10 instructions to be configured. Sending the instructions without RESTOP takes 290μs and 1.1μJ, while RESTOP spends about 310μs and 1.3μJ, which represents 14% of energy overhead (0.2μJ). The transceiver consumes about 1.9μJ when transmitting, and the whole transmission process needs about 38.24μJ. Therefore, the energy overhead caused by the proposed middleware only represents 0.5% of the total energy needed for a complete transmission.

In summary, the energy overhead caused by RESTOP is minimal even in the worst case. This difference mainly affects when searching and replacing an instruction in the history table. This means that the greater the number of saved instructions, the greater the cost caused by the proposed approach. However, RESTOP does not cause any energy overhead when restoring the peripheral configurations after a power failure.

4.5 Discussion

This chapter proposed RESTOP, a new approach to retain the state of peripherals that communicate with an MCU through a digital interface, in transient computing systems. The presented middleware provides generic functions to read data from the external peripherals or write to them, and keeps track of and saves the transmitted configuration data into the instruction history table from where the peripheral state is restored after a power failure. With these characteristics, RESTOP can be integrated into any of the existing *system state retention* approaches and, unlike existing designs, it is able to operate generically with multiple devices that communicate with the MCU through different protocols such as SPI or I^{2}C.

RESTOP has been validated with a digital accelerometer (SPI), a luminosity sensor (I^{2}C) and a radio transceiver (SPI) in a transiently-powered system. Results demonstrate that RESTOP is capable of restoring the peripheral state after power outages causing a time overhead to the application of up to 0.82% during complete execution of the sensing application, which is considerably lower than that caused by existing approaches. This can be considered as a corner case, because the experimental validation included three peripherals that are configured and accessed multiple times (including the *read* instructions), with several power failures. In a real scenario, designers may save only the *write* peripheral instructions, reducing the time overhead caused by RESTOP when storing the instructions into the *instruction history table.*

However, retaining the external peripheral state is not the only challenge that transient systems have to address. Although the proposed approach was validated in a system that performed *atomic* tasks (i.e. sense and transmit data) as well as long-term computation (FFT), designing a real application capable of performing *atomic* and long-term computation requires a proper methodology in order to lead with energy shortage. Moreover, typical applications need to have a sense of time to properly perform their functions, e.g. to determine whether the sample data is still valid or has expired. In the following chapter, the contributions made to address these drawbacks, are presented.

Chapter 5

From Task-based to Long-term Computation

Typical sensor systems have to perform *atomic* tasks (e.g. sensing and transmitting) and long-term computation (e.g. filtering and encrypting) as part of their functionality. However, attempting to complete these tasks before a power failure occurs, may imply to add a capacitor, whose size may cause a delay to be charged, when powered from low-power EH sources. For example, the ferroelectret insole used in Chapter 3.2, needs three steps to charge a 4.7μF capacitor to an operating voltage level. However, the insole may need up to ten steps to charge a capacitor of 10μF, which would increase the uncertainty. Moreover, applications usually need to have a sense of time to properly perform their tasks, but in a system that suffers of frequent power losses and lacks of energy storage, keeping track of time is not so trivial.

In this chapter, a novel design framework is detailed, which includes the strengths of *task-based* and *system state retention* schemes in an energy-aware manner, in order to address application requirements, minimizing the energy storage size. The framework also proposes a strategy for keeping track of time, which intelligently combines existing approaches in order to solve time issues with diverse granularity. The viability of the proposed approach was validated through a case study, a transient computing step counter, able to calculate step rate and METs, as well as encrypt and wirelessly transmit the collected data.

5.1 EH-based Design Framework for Transient Computing Systems

The proposed design framework restructures the application, based on how the tasks can be executed (Figure 5.1) as a function of the harvested energy only, and therefore, reduce or totally remove the obtrusive and expensive energy storage, unlike existing proposals [146, 147, 148], which attempt to match the input power with the voltage-current requirements and are focused in the computing unit. The framework is made of six main stages, which are: energy source profiling, a strategy to keep track of time, task classification, energy requirements, task control flow and the optimum operating voltage.

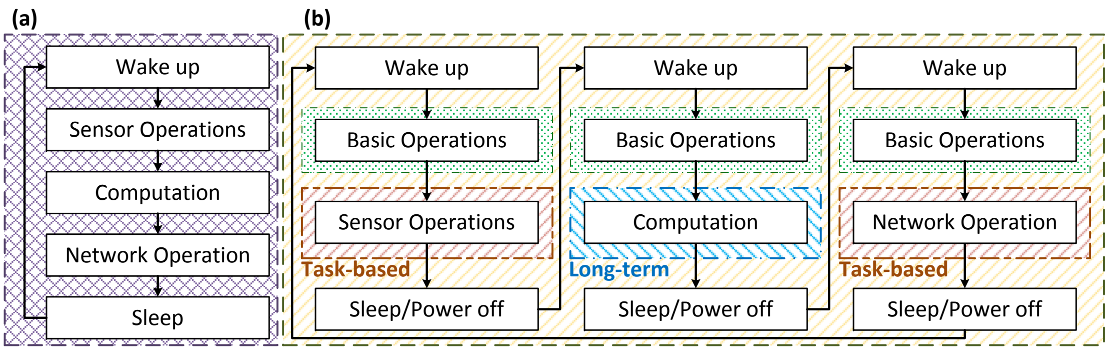


Figure 5.1: Comparative control flow of **(a)** a typical IoT system (adapted from [10]) and **(b)** a restructured transient system under the proposed framework.

5.1.1 Energy Source Profiling

The first stage is to profile the EH source, keeping in mind the application requirements. Transient systems are oriented for schemes powered by highly variable energy harvesting sources. Therefore, profiling is important not only to obtain the amount of energy supplied, but also to define whether the energy source can also provide information about the parameter to be sensed. If so, the system can be designed as an *event-based* application, where the harvester coincides with the parameter that is being sensed. This means the energy availability is an indicative that the event to be measured, has occurred.

In the case that the harvester does not coincide with the sensing parameter, the system has to be designed to operate opportunistically-triggered when the amount of energy is enough to complete their tasks. The energy profiling is useful as well to determine whether it is possible to obtain information related with time, which is needed by systems for different purposes (e.g. to define whether a sampled value has expired). In some applications, the elapsed time can be estimated, based on the shape of the power signal [128]. If not, the strategy for keeping track of time has to be followed, which is described in the next subsection.

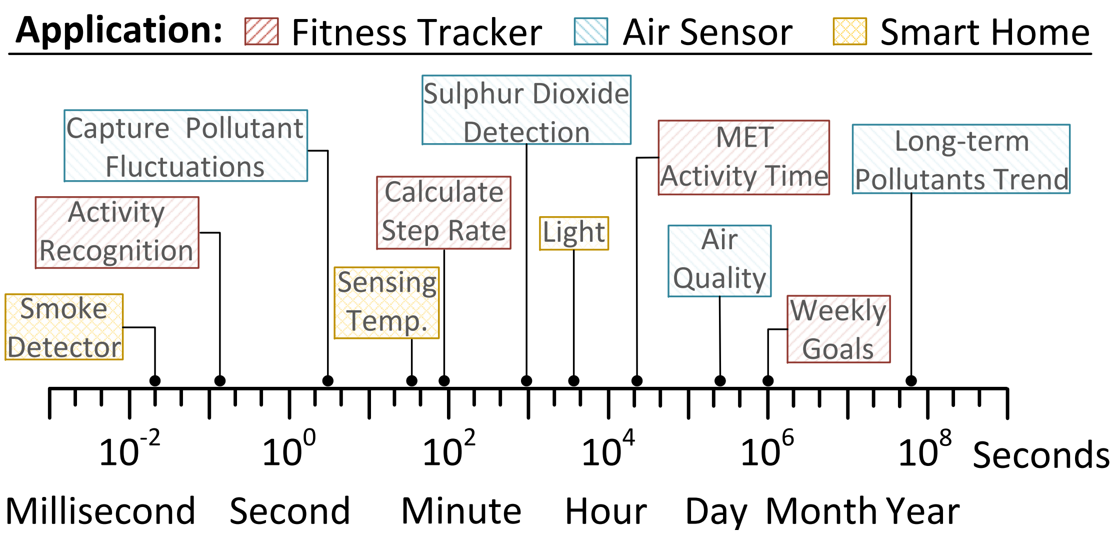


Figure 5.2: Time granularity needed for typical sensor system applications [11, 12, 13, 14].

5.1.2 Keeping Track of Time

Fig. 5.2 shows the time granularity needed for typical sensor applications. The precision required when measuring time depends on the task a sensor system is performing. For example, a typical fitness tracker needs accurate time in the order of milliseconds (ms) for activity recognition, seconds and minutes for step rate, hours for activity time or caloric expenditure and days to check whether the user meets their weekly goals. Other applications such as air pollution sensor systems, need to know at what time each sample is taken for future statistical analysis or ecological policies.

In typical EH sensor systems, the time is easily measured (e.g. through an RTC) because they are permanently active. However, in a transiently-powered application, the active periods are usually smaller than the time the system remains off (due to power shortages). Nevertheless, although the active periods of transient systems may be as short as few ms, they are fundamental for tracking longer intervals. During these periods, it is possible to accurately measure the shortest periods needed by system to properly perform their functions.

However, the main challenge in transient systems is to track time during off intervals, when the availability of energy is low or zero. Some transient EH systems operate periodically, where the off periods are in the order of seconds. In this case, the elapsed time from one power cycle to the other, can be estimated with some of the *hourglass-like* methods described in Section 2.4.2. Adding together these methods to calculate time during active (ms) and off (seconds) intervals, it is possible to continuously track the time (which can go from several minutes to few hours), until the zero-power interval is longer than what can be measured with the *hourglass-like* method.

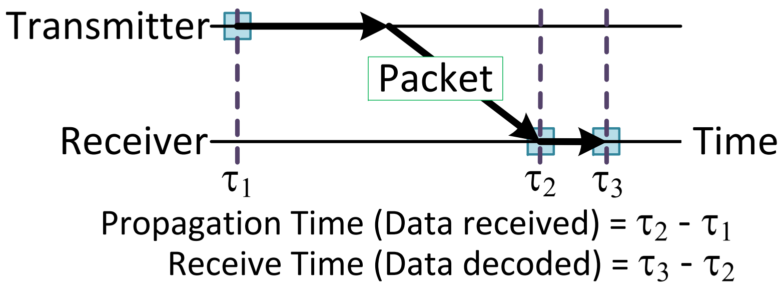

Transmitter
Packet
Receiver
Time
τ_1
τ_2
τ_3
Propagation Time (Data received) = $\tau_2 - \tau_1$
Receive Time (Data decoded) = $\tau_3 - \tau_2$

Figure 5.3: Propagation and receive time between transmitter and receiver.

In the applications where the off intervals are in the order of several minutes or hours, it is necessary to take advantage of time synchronization through wireless communication. In these cases, each received packet has to be timestamped. Then, comparing the stamps of the last and more recent packets, it is possible to calculate how long has passed since the last time the sensor system was active.

However, the calculated time may need to be compensated considering the time spent between the data was sent and it was received (*propagation* time) and decoded (*receive* time) [134]. As shown in Figure 5.3, *propagation* time may be the main factor of error because both the sensor system and the receiver cannot measure that time, unless it is a fixed value that does not vary among transmissions.

5.1.3 Task Classification

A significant drawback of task-based approaches is the longer the tasks the larger the capacitor. Although there are tasks that have to be atomically executed, i.e. without interruptions, others can be progressively completed through various power cycles, and thus, avoid increasing the energy storage. Therefore, typical tasks performed by sensor systems can be classified depending on whether they are *atomic* or *interruptible*.

Sensing and transmitting tasks have to be completed without interruptions because in the first case, it is not desirable to get a partial sample in a power cycle and complete it in the next. In the case of wireless transmission, the communication can only be established with complete packets that contain not only data but also the synchronization information. However, longer tasks such as filtering and encrypting, which do not have *atomic* constraints, can be split into various power cycles by saving the system state before a power failure and restoring it in the point where it was interrupted, when energy is available again. It is therefore not necessary to add capacitance to complete them.

5.1.4 Estimation of the Required Energy Storage Size

As part of the design process, the largest *atomic* task (in terms of energy consumption, which may involve peripherals) has to be defined, as well as the minimum capacitance required by the system to properly execute that task without power interruptions. The minimum capacitance can be calculated with the following equation:

$$C_{min,i} = \frac{2E_{task,i}}{V_{max}^2 - V_{min}^2} \tag{5.1}$$

where $C_{min,i}$ and $E_{task,i}$ are the minimum capacitance and the energy consumption required to complete the task i, respectively. V_{max} and V_{min} are the maximum and minimum operating voltages of the whole system.

5.1.5 Task Control Flow

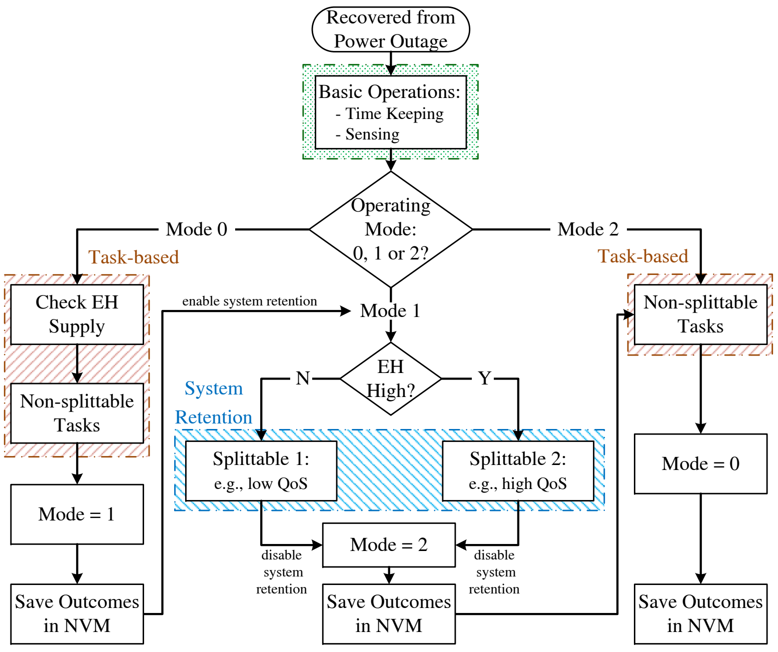


Figure 5.4: Proposed design flow to combine task-based and system state retention schemes.

Fig. 5.4 shows the proposed algorithm to combine *task-based* with *system state retention* schemes, in order to execute *atomic* tasks and long-term computation, in an energy-aware manner[1]. When the system starts working for the first time or after a power failure, basic operations are executed. Then, the latest operating mode is verified. If it is working for the first time (Mode 0), the system checks for the parameter that will define the sensing accuracy or quality of service (QoS) in the processing task. When

[1]This flowchart was enhanced using the feedback received from Dr. D. Balsamo and Dr. O. Cetinkaya, during the elaboration of the paper.

tasks of Mode 0 are completed, the system saves the outcomes into a NVM and moves to Mode 1.

In Mode 1, the system executes the processing task, whose accuracy or quality of service (QoS) is based on the collected information in Mode 0, under two main criteria:

1. **Based on the collected data.** Systems may need to process (e.g. encrypt) the sensed information before transmitting it. This criterion is usually applied for designs with a fixed operating threshold (V_{TH}).
2. **Based on the energy available.** These approaches usually have a dynamic V_{TH} (high power EH source) and perform more complex tasks (e.g. sensor fusion and filtering) depending on the amount of energy that the EH source is providing.

The processing task is interruptible and can be completed through various power cycles. For this purpose, a *system state retention* approach has to be included in order to protect system from volatility. Once Mode 1 is completed, the system moves to Mode 2, where *atomic* tasks (e.g. transmitting) are executed, and Mode 0 is again set. Therefore, following this design flow, the size of the capacitor is determined by the largest atomic task (not by the interruptible computation), which in many cases can be completed with the energy stored in the decoupling capacitance of typical MCUs.

5.1.6 Optimum Operating Voltage

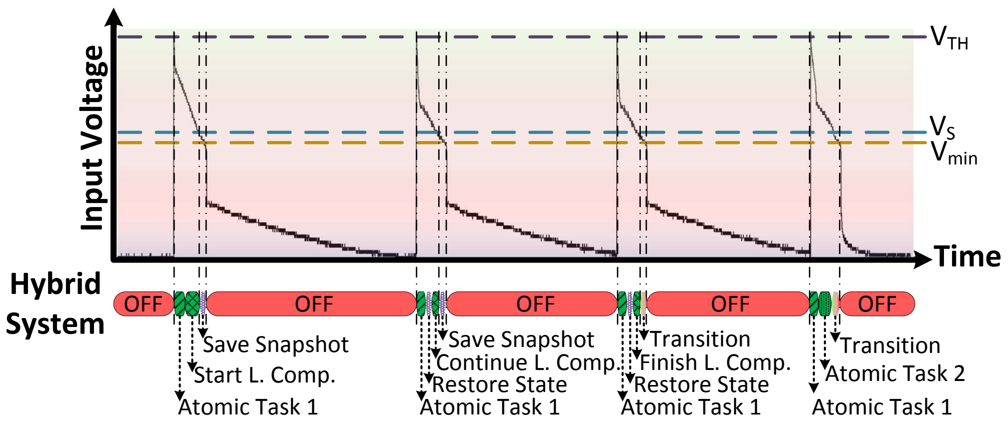


Figure 5.5: Operation of the new hybrid system that performs atomic tasks and long-term computation.

Once the algorithm, which controls when to execute each task of the sensor application, is defined, the final stage is to calculate the optimum voltage value (V_{TH}), that guarantees there is enough energy to complete each atomic task. V_{TH} can be calculated with the following equation:

$$V_{TH,i} = V_{min} + \frac{1}{C_{min,i}} \int_0^t I_{task,i}(\tau)d(\tau) \tag{5.2}$$

where $I_{task,i}$ is the current consumption of the task i. Depending on the application design, the sensor system may execute the same atomic task (keeping a fixed V_{TH}) or different (dynamically adjusting V_{TH}), at each power cycle. Thus, as shown in Figure 5.5, when the input voltage reaches V_{TH}, the transiently-powered sensor system starts working and executes an *atomic* task (the same or a different one) at each power cycle and performs portions of long-term computation with the remaining energy, before input voltage drops to the snapshot threshold (V_S). In the next section, the proposed framework is proved through the design, implementation and experimental validation of a case study.

5.2 Case Study: A Transient Computing Step Counter

In these sections, the proposed framework is validated, through the implementation and experimental validation of a case study: a transient computing step counter. The transient system to be designed attempts not only to count steps, but also to calculate step rate, metabolic equivalent (MET) and run/walk time as well as encrypt and wirelessly transmit the collected data, without active sensors in a storage-less design (Figure 5.6).

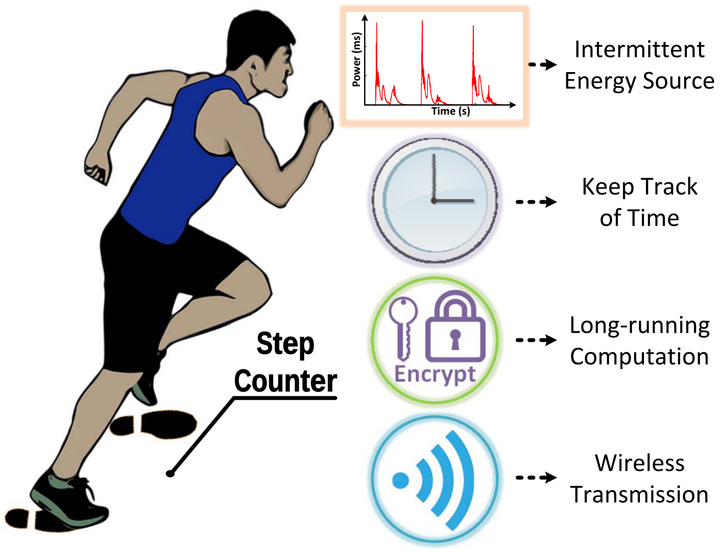


Figure 5.6: Challenges to be addressed in the proposed case study.

5.2.1 Ferroelectret EH Insole: Profiling

Following the proposed framework, the first stage is to profile the energy source, which is a 50-layer ferroelectret insole [149]. Unlike the insole described in Section 3.2.1, this one includes thin nickel films symmetrically placed in relation to the stack center. This is to better propagate the mechanical stress suffered by the inner layers of the insole and thus increase the energy output.

Figure 5.7 shows the model of a two-layer ferroelectret insole with (Figure 5.7b) and without (Figure 5.7a) a nickel film. In both structures, a silver layer was printed at each side of the PP films in order to build the electrodes, and bonding films were added between PP layers to prevent them from self-discharging. Figure 5.7b shows how a nickel film is placed between two bonding layers to isolate it from the PP films. The experiments demonstrated the insole with nickel films provides up to 17% more energy than the insole without these metal layers.

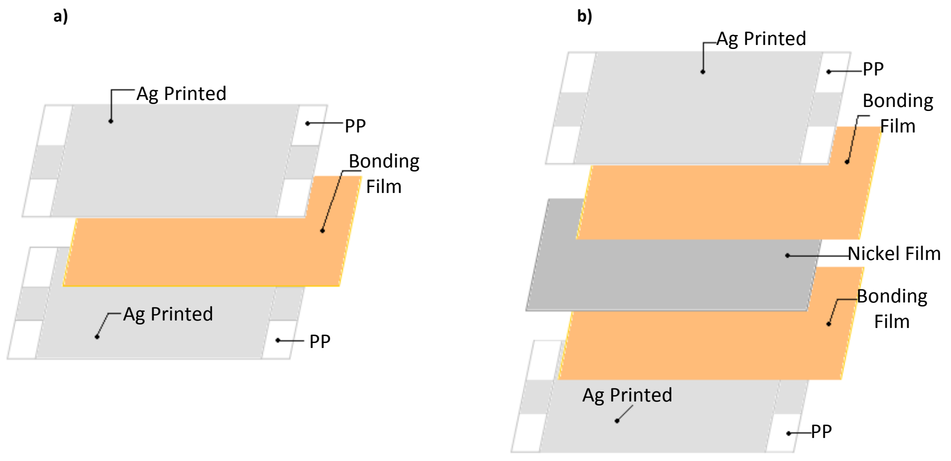


Figure 5.7: Multilayer ferroelectret insole structure, showing: (**a**) an insole without nickel films; and (**b**) with a nickel film.

The 50-layer ferroelectret energy harvester is mounted in a shoe insole as shown in Figure 5.8. Because this material converts the mechanical energy caused by the heel striking against the floor into electricity, it is possible to design an event detection sensor, where the presence of energy is an indicative that a step was taken. This source generates two main high-power bursts, one positive when compressed and one negative when released, whose magnitudes depend on the applied mechanical stress, as well as on the impedance of the connected load. The greater the applied force, the greater the amplitudes but the shorter the length of the power signal. Figure 5.9 shows the rectified power output of the 50-layer insole after three steps, when a 50MΩ load is connected. The power peaks reach up to 1.4mW but they last for few microseconds.

Typical microcontrollers require a small capacitance, whose size usually ranges from 4.7μF to 20μF [91], to shunt the possible noise caused by the energy source. Therefore, the performance of a 50-layer ferroelectret insole [149] is evaluated with three different capacitors (4.7μF, 10μF and 20μF), to emulate the decoupling capacitance of a typical MCU [91] (the plan is not to include any extra capacitance).

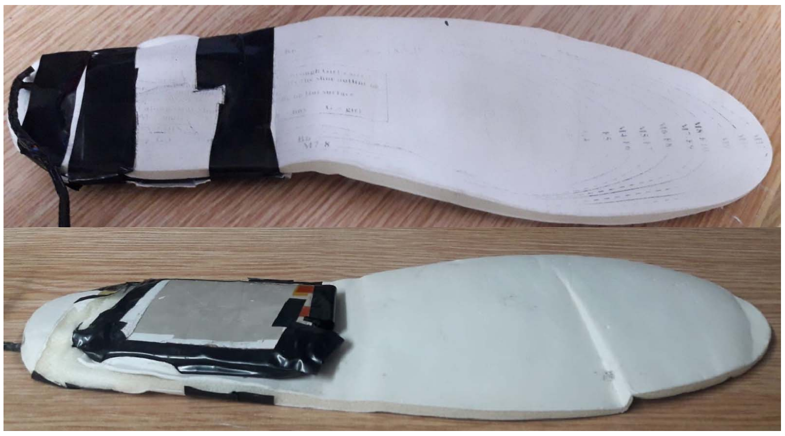


Figure 5.8: 50-layer polypropylene ferroelectret energy harvester mounted in a shoe insole.

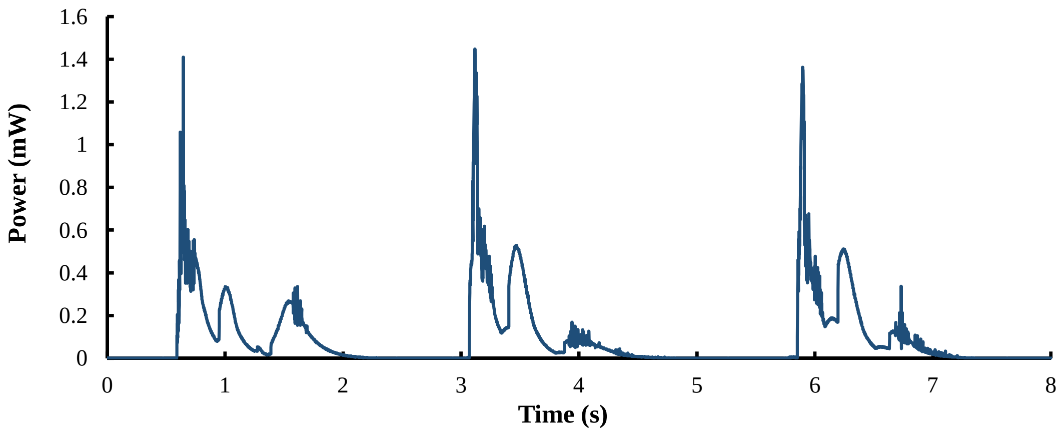


Figure 5.9: Rectified output power signal of the 50-layer insole after three steps.

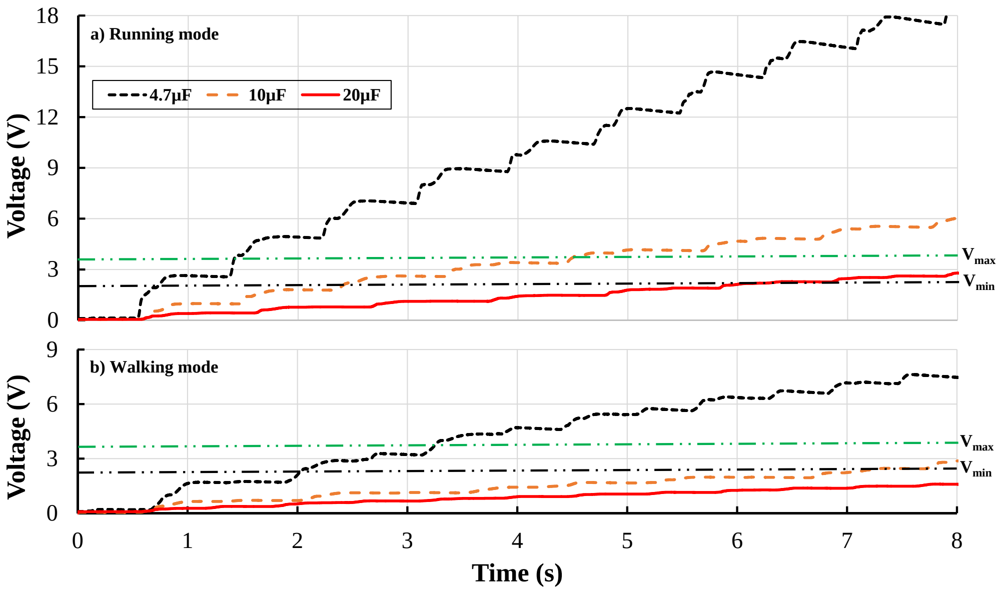


Figure 5.10: Output voltage signal for each capacitor, showing: **(a)** voltage in *running* mode; and **(b)** voltage in *walking* mode.

The evaluation circuit includes a full-wave rectifier, which is built with a diode bridge, and a capacitor as a unique load. The voltage (V_C) was continuously measured across the capacitor. This experiment was performed by a 70-kg person, with the insole worn inside the shoe and in two operating modes: *walking* (about 70 steps per minute) and *running* (approximately 120 steps per minute). The test was performed five times per capacitor in each mode and the averaged values are shown in Figure 5.10.

Table 5.1: Energy charged in each capacitor per footstep

Step	**Walking mode (μJ)**			**Running mode (μJ)**		
	20μF	**10μF**	**4.7μF**	**20μF**	**10μF**	**4.7μF**
1	1.21	2.06	6.66	2.31	4.79	16.01
2	2.93	3.61	17.85	3.97	11.44	40.64
3	3.70	7.55	27.85	5.88	18.53	60.20
4	4.26	8.35	28.64	7.52	21.35	70.41
5	5.31	11.05	31.95	13.87	28.97	81.82
Total	17.41	32.62	112.95	33.55	86.70	269.08

The energy generated per step was also calculated (using Formula 3.3) and presented in Table 5.1, which shows the amount of energy charged in each capacitor in both operating modes. The maximum amount of energy is obtained with the 4.7μF capacitance. However, the voltage is over the maximum operating value of a typical MCU (Figure 5.10). These factors will be taken into account for the design process in the following sections.

5.2.2 Keeping Track of Time in the Step Counter

For the characteristics of the system to be designed, it is not possible to obtain information related with time, because the output voltage is affected by different factors such as walking speed, user's height and weight, etc. Therefore, *CusTARD* [9] is included to keep a sense of time in the proposed design during the zero-power intervals from one step to the other. Figure 5.11 shows the timekeeper circuit and how it interacts with the MCU.

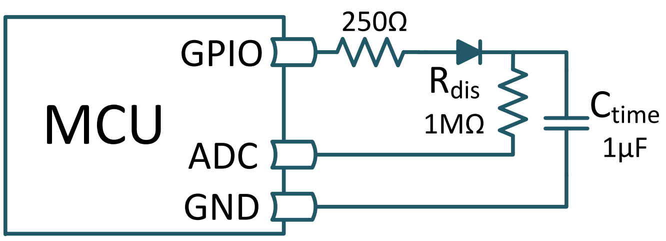


Figure 5.11: Timekeeper circuit.

The size of the capacitor and the value of the discharge resistor were determined considering the longest zero-power interval to be measured when walking at the slowest speed, based on the *optimal walking* study performed by Noboru Sekiya *et al.* [150]. The capacitor (C_{time}) is charged by a GPIO pin of the MCU, through a resistor and a schottky diode to avoid inverse current. Thus, C_{time} is discharged through R_{dis} and the ADC measures the voltage value before and after a power failure occurs. Implementing this scheme implies a thorough tailored calibration and a characterization to generate a look-up table in order to map the capacitor voltage decay rate to corresponding elapsed time.

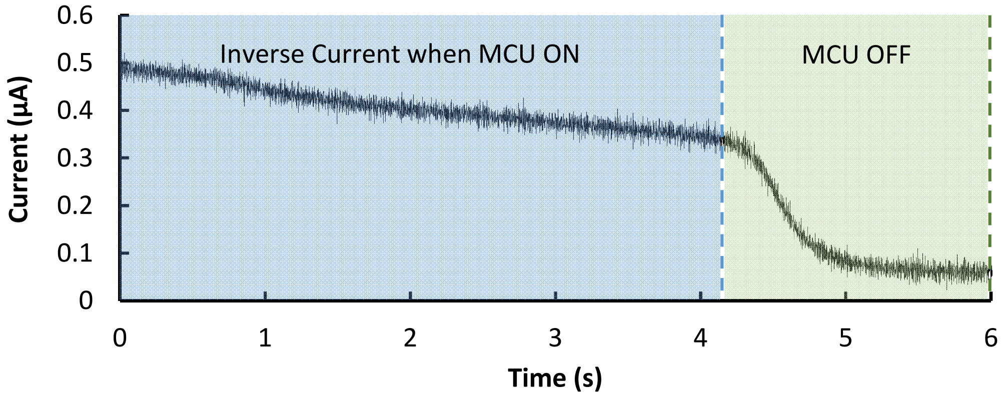


Figure 5.12: Inverse current of the diode measured when MCU was active and off.

The calibration of the timekeeper circuit includes to measure the inverse current through the diode and the leakage current of the capacitor. For the first case, an ammeter was placed between the charge resistor and the diode. Then, the current was logged since the point that C_{time} is charged until the MCU is off and the obtained results are plotted in Figure 5.12. Here, it is possible to observe that the inverse current is higher when the MCU is active than when the system is off. This is because when the MCU is not active, the GPIO pins are set as inputs. Therefore, the charge of the capacitor is drawn through the ADC pin, reducing the inverse current passing through the diode. The RMS of the inverse current was also calculated, giving $0.41\mu A$ when the system was powered and $0.2\mu A$ when it was off.

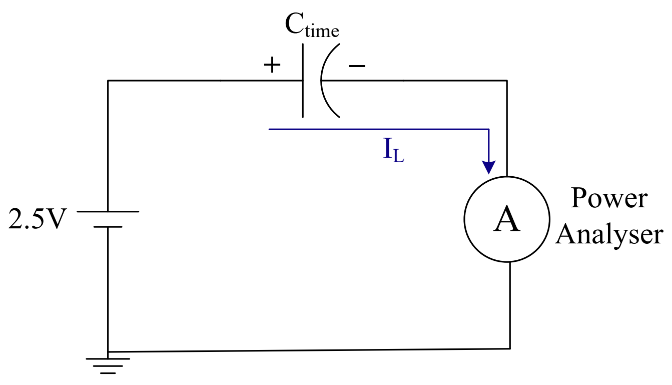


Figure 5.13: Implemented circuit to measure the leakage current of the timekeeper capacitor.

In order to measure the leakage current (I_L) of the timekeeper capacitor, the circuit of Figure 5.13 was implemented. C$_{time}$ was charged up to 2.5V, while the ammeter was registering the current. Then, the logged data was plotted in Figure 5.14. The RMS leakage current of this signal is 0.07μA. The leakage current of the capacitor and the inverse current of the diode will be considered when calculating the elapsed time in order to estimate accurate values.

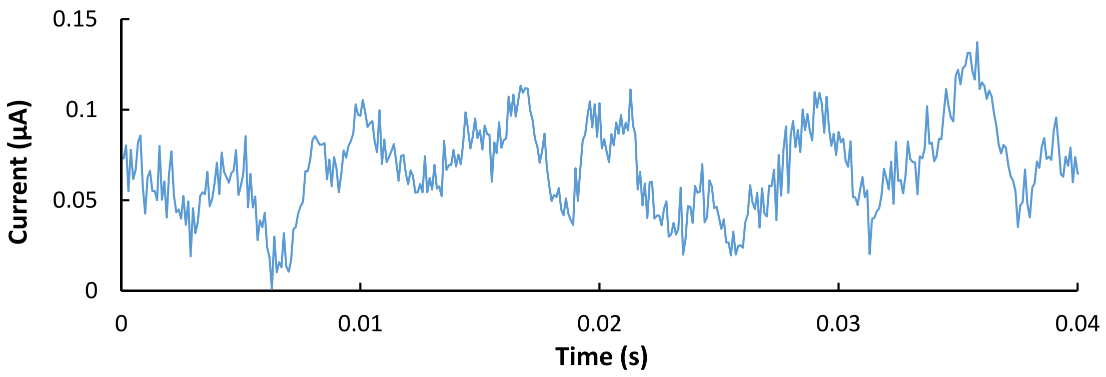


Figure 5.14: Leakage current of the timekeeper capacitor.

In order to calculate the elapsed time, based on the discharge rate of C$_{time}$, the following equation has to be solved:

$$t = [ln(V_O) - ln(V_C)]RC_{time} \tag{5.3}$$

where V_O is the voltage across C$_{time}$ measured before a power failure occurs, V_C is the voltage after the power outage and R is the discharge resistor. However, calculating the natural logarithm takes about 200μs in a low power MCU [91], and estimating the elapsed time would take up to 500μs. This is an important drawback in a system powered by a EH source, which may provide energy only for few ms. An alternative method is to use the Tylor series for natural logarithm, described below:

$$ln(x) = 2 * \sum_{n=0}^{\infty} \frac{1}{2n+1} \left(\frac{x-1}{x+1} \right)^{2n+1} \tag{5.4}$$

Taylor series attempts to approximate the result of a function, through the sum of an infinite number of iterations (n). However, the series includes an exponential operation to solve the natural logarithm, which also consumes important resources (time and memory usage) in order to be solved. Therefore, the series was evaluated with different number of iterations to evaluate its accuracy, and the results are listed in Table 5.2.

Table 5.2 shows that with a single iteration (n=0), the accuracy is over 93% when calculating the natural logarithm and almost 90% to estimate the elapsed time, with the advantage that no exponential operation would be needed. This accuracy level may

Table 5.2: Accuracy of Taylor Series when calculating Ln and elapsed time with different number of coefficients.

Iterations	**Accuracy for Ln (%)**	**Accuracy for Time (%)**
n=0	93.53	89.46
n=1	99.26	91.55
n=2	99.89	91.58

be good enough for the time requirements of the step counter. For example, an error of 10% when measuring activity time, represents an error of 9% when calculating step rate, which is inside the error rate of typical fitness trackers [151]. However, this implies an error of much less than 1% when estimating the metabolic equivalent (MET). Moreover, the accuracy could be improved (i.e. increasing the number of iterations) when the step rate is increased and therefore, the available energy.

Therefore, for calculating the elapsed time from one power cycle to another, the voltage in C$_{time}$ is measured before (V_O) and after (V_C) a power failure. Then, the Taylor series is solved for obtaining the natural logarithm and the obtained values are substituted in Formula 5.3. Finally, the effects of the inverse current of the diode, the leakage current of C$_{time}$ and the current drawn through the ADC when the MCU is off (in the order of 6μA), are compensated.

Following the proposed strategy for keeping track of time from Section 5.1.2, the time measured during active periods is added to the estimated time during off intervals. Thus, it is possible to track for how long the user has been walking or running, calculate the step rate or distinguish when the user starts and stops their physical activity. Then, when the computed data is transmitted, in the receiver unit it is possible to analyse the daily/weekly activity goals by comparing the timestamped data.

5.2.3 Step Counter Tasks Classification

The basic function of the system is counting steps, which is an *atomic* task that has to be executed at each power cycle, as well as measuring the elapsed time between power cycles, based on the strategy defined in Section 5.2.2. Calculating the step rate, meanwhile, implies not only the number of steps but also the time spent to take them. Because the step rate is usually presented as steps per minute (spm), it is necessary to keep track of time for longer intervals than just the active periods, which is usually of few milliseconds per power cycle. Therefore, this *atomic* task is not executed at each power cycle.

Another *atomic* task is obtaining the METs. D. Harrington *et al.* [152] proposed the following equation to calculate METs based on the step rate and the activity duration:

$$MET \cdot h^{-1} = 1.4 * d + 2.6 * \frac{sr}{120} * d \tag{5.5}$$

where sr is the step rate and d is the activity duration (in hours). The activity duration implies to track time for longer periods than just few seconds. To do this, the time measured during active and off periods is added together, along the run/walk routine.

The collected and calculated data (number of steps, step rate, activity duration and METs) are meant to be encrypted before transmitting. A cryptographic algorithm suitable for embedded systems is the advanced encryption standard (AES)[153], which supersedes the older data encryption standard (DES). The encryption takes about 12ms to be completed, but it does not have time constraints. Therefore, it can be classified as an *interruptible* task to be completed through various power cycles.

In order to progressively complete the encryption, Hibernus [3] is incorporated with a selective policy for efficient state retention proposed in [84]. This solution was chosen because it is application agnostic, and has a good performance in terms of energy and time overhead [43]. Finally, the last function is wirelessly transmit the encrypted data, which is an *atomic* task that does not have to be executed at each power cycle.

5.2.4 System Energy Requirements

The system is attempted to be implemented in an MSP-EXP430FR5739 test board [91], that includes an MCU with a low power Ferroelectric RAM (FRAM) and a decoupling capacitance (C_S) of 10μF (when removing the unused on-board sensors). An nRF24L01 [154] transceiver is also included for wireless communication, which presents one of the best performance in terms of the relation between the power consumption and its high data rate output of up to 2Mbps [155]. Based on this, the current consumption of each task is measured with a power analyser (Agilent N6705B). The measurements include the static current of devices in order to consider the worst case scenario. To measure the time, a PicoScope 3000 Series is used to monitor the start and end of each task. Then, the energy consumed by each task is calculated and the obtained results are presented in Table 5.3 (including the intrinsic MCU start-up).

Based on the values presented in Table 5.3, the *atomic* task that consumes more energy is the wireless transmission with about 38.24μJ. Substituting this value in equation 5.1 and considering the MCU operates between 3.6V (V$_{max}$) and 2V (V$_{min}$), the minimum capacitance needed is of 8.53μF. Therefore, no additional capacitance will be needed, fulfilling the storage-less requirement. On the contrary, if the longest task (encrypt) was attempted to be completed without interruptions, the minimum capacitance needed would be over 66% larger (just to complete that task).

Table 5.3: Parameter values of each system task

Task	**Current (mA)**	**Time (ms)**	**Energy (μJ)**
Encrypt	1.5	12.153	56.11
CusTARD	2.6	0.215	1.72
Count steps	0.9	0.092	0.26
MCU Start-up	1.5	2.232	10.32
TX transmission	8.1	0.01	0.25
Snapshot/Restore	0.9	1.107	3.07
TX configuration	3.66	2.436	27.41
Calculate elapsed time	1.15	0.277	0.98
Calculate step rate & MET	1.05	0.235	0.76

5.2.5 Task Control Algorithm

Before calculating the optimum operating voltage, the algorithm to enable each task at each power cycle has to be defined. Figure 5.15 shows the control flow of the step counter algorithm, which operates in three modes. There are two tasks (so-called *primitives*) that are executed in the three modes: counting steps and calculating the elapsed time. These *primitive* tasks are directly saved into NVM, in order to retain their values between power outages. In mode 0, when the activity time is longer than a certain value (it depends on how frequent the user wants to transmit), the system calculates the step rate and the METs consumed. Then, it moves to Mode 1, which implies to enable Hibernus and start the long-term computation (based on the collected data), which is encrypting.

The size of the encryption key is determined by the step rate, which is directly related with the available energy (the faster the step rate, the higher the energy provided by the insole). Thus, in addition to the *primitive* tasks, the program moves forward in the encryption process at each cycle. Once the encryption is completed, Hibernus is disabled and the system moves to Mode 2 and checks whether there is enough energy to transmit. If so, data is immediately transmitted. On the contrary, it waits for the next power cycle to transmit the packet, and moves back to Mode 0.

5.2.6 Optimum Operating Threshold and System Architecture

Considering the tasks that have to be executed at each power cycle and substituting their parameters from Table 5.3 in the equation 5.2, the optimum operating voltage obtained is 3.4V. However, in Mode 2, the tiny capacitor used by *CusTARD* to measure elapsed time is not charged, because it would need a higher voltage than V_{max}. Therefore, it is preferable to loss a bit of accuracy when calculating time in Mode 2, than increasing the decoupling capacitance.

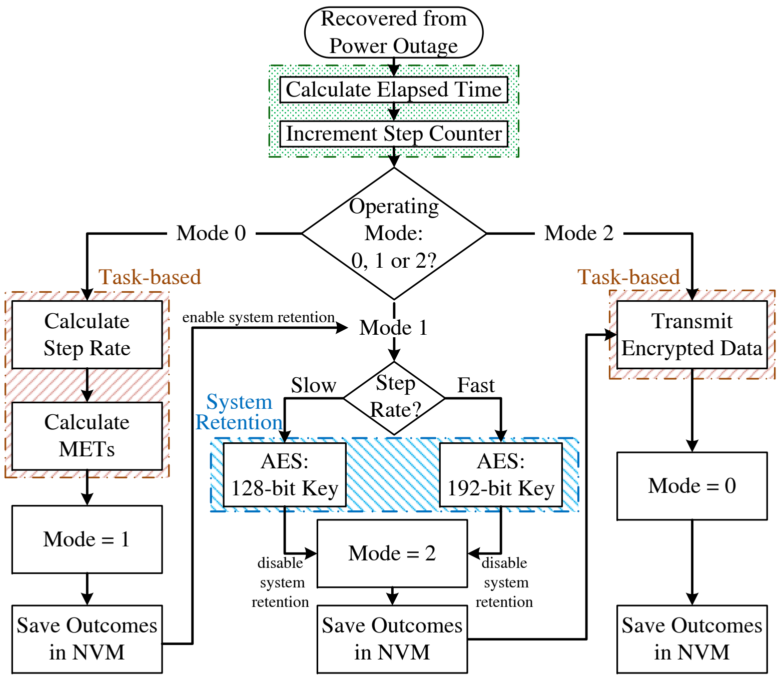


Figure 5.15: Flow chart of the step counter algorithm.

Figure 5.16 shows the final design including the ferroelectret insole, the rectifier, the timekeeper circuitry and the 10μF decoupling capacitance (C). An additional cold-start circuit is included, to reduce the quiescent current of the MCU when its supply voltage is lower than V$_{min}$. This circuit turns on the supply to the MCU when the input voltage reaches V$_{TH}$ and switches it off when voltage drops below V$_{min}$.

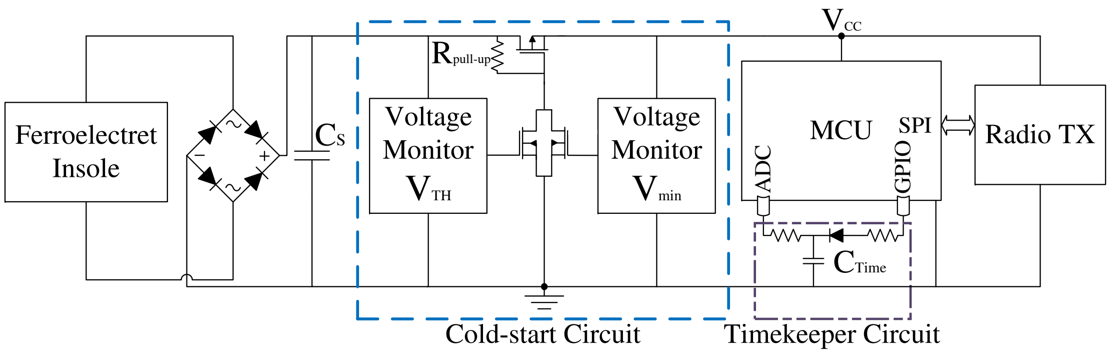
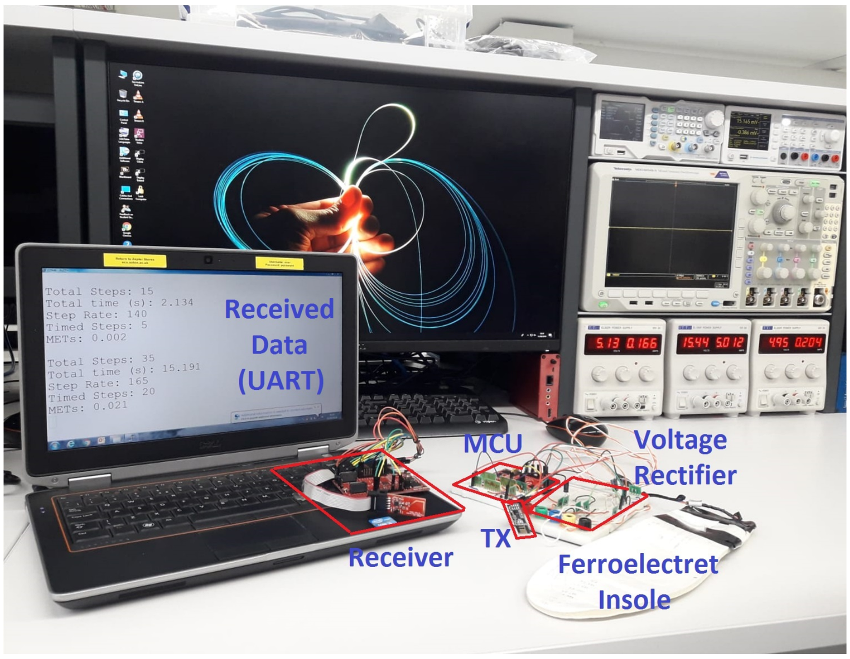


Figure 5.16: Transiently-powered Step Counter circuit diagram.

5.2.7 Functional Validation and Comparative Evaluation

The transiently-powered step counter has been implemented and experimentally validated. Figure 5.17 shows the experimental set up to validate the step counter functionality, including the lap top where the received data is presented for analysis. The proper functionality of the whole system is first verified, i.e. whether the step counter is able to execute all the tasks. Then, its accuracy to measure time at different step rates was evaluated. Finally, the performance of the system when counting steps, was compared against a fitness band and a smartphone application. The collected and encrypted data was wirelessly transmitted and presented by the receiver through a serial terminal, in order to analyse the obtained results.



```
Total Steps: 15
Total time (s): 2.134
Step Rate: 140
Timed Steps: 5
METs: 0.002

Total Steps: 35
Total time (s): 15.191
Step Rate: 165
Timed Steps: 20
METs: 0.021
```

Figure 5.17: Experimental step counter set-up including the lap top to show the transmitted data.

Figure 5.18 shows the complete process since the step counter works for the first time, until it transmits the encrypted data wirelessly. To verify the proper operation of the proposed design, a PicoScope was used to monitor the voltage signals of the decoupling capacitance, MCU and timekeeper capacitor. Additionally, five GPIO pins of the evaluation board were configured in order to indicate when the system is measuring the voltage in the timekeeper capacitor before and after a power failure, as well as calculating the elapsed time (Time). The signals to indicate when the application is saving (HB) and restoring (Rst) the system state, encrypting (AES) and wirelessly transmitting (TX), are also monitored.

In this example, the system needs about 8 steps to reach the operating voltage (V_{TH}) the first time, and up to 5 steps in the following cycles. The voltage in the timekeeper capacitor is measured just before the supply drops below the minimum operating (V_{min}) and immediately after V_{TH} is reached, in order to calculate the elapsed time. The process of encrypting takes three power cycles. Therefore, Hibernus is enabled to retain the system state, and thus, progressively complete this task through various power cycles. When the data is encrypted, Hibernus is disabled and the system checks whether there is enough energy to transmit. If so, the data is transmitted; if not, it is sent during the next power cycle.

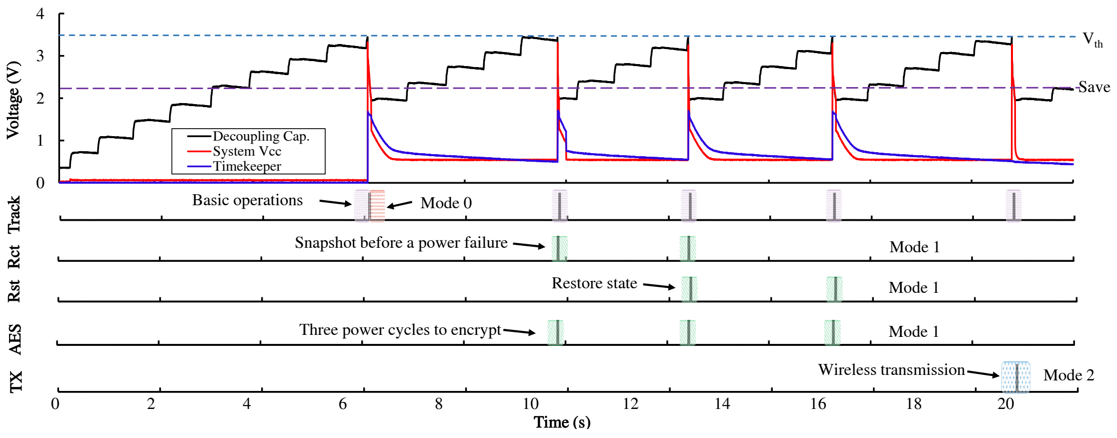


Figure 5.18: Operation of the transiently-powered EH step counter, from the first step until the encrypted data is wirelessly transmitted.

Table 5.4: System accuracy when measuring time.

Step Rate (spm)	Time between steps (ms)			Activity Time (min)	
	Expected	Measured	Error (%)	Measured	Error (%)
64	1875	1916	2.15	10.188	1.88
84	1428	1454	1.84	10.110	1.10
108	1111	1127	1.44	10.109	1.09
127	945	956	1.18	10.080	0.80
148	811	817	0.74	10.064	0.64

Once the proper operation of the design is proved, the next experiment is to evaluate its accuracy when measuring time with diverse granularity (time between steps at different speeds, activity time, etc.). Five different step rates were considered, following the range presented by Noboru Sekiya *et al.* [150]. Table 5.4 shows the averaged results obtained when measuring different time intervals. The experiment was performed five times per step rate, with the insole worn in the shoe.

In the case of the activity time, only 10 minutes were considered for practical reasons, because it is not feasible to walk/run for 1 hour or more for each scenario, with the prototype attached. As shown in Table 5.4, the biggest error occurs at the lowest step rate. This is because the lower the step rate, the longer the time the system is off. Accordingly, the timekeeper capacitor suffers a greater discharge and is more affected by temperature, leakage current and parasitic capacitances, reducing the accuracy when estimating the elapsed time.

For step rate evaluation, the same range of step rates, used for time accuracy, was considered. Figure 5.19 shows the dispersion of the obtained results when calculating the step rate. Although the biggest dispersion occurs at a step rate of 84 steps per minute (spm), the maximum errors occurs at the lowest step rate. This is because the lower the step rate, the greater the variability in the energy generated due to the

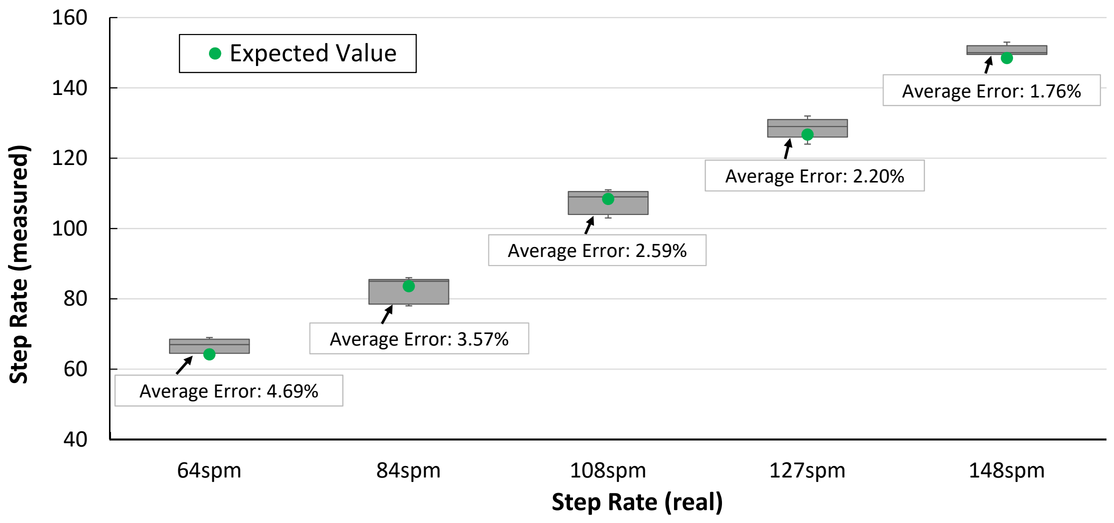


Figure 5.19: Transient step counter accuracy when calculating the step rate.

difference in which the heel strikes the ground from one step to the other. The source of error in the proposed design is the measured time, unlike battery-powered devices in which, the main source of error is when counting steps. However, the accuracy shown by the transient step counter is inside the range demonstrated by existing fitness trackers [156].

The error of the step counter when calculating the step rate, has to be considered when estimating METs. Therefore, the effect of step rate error, from the previous experiment shown in Figure 5.19, was also validated and the results listed in Table 5.5. Here, the real and the calculated step rate are listed for each attempt (five attempts per step rate) as well as the METs consumed during 1 minute of activity and the percentage of error between the calculated and the expected values. Table 5.5 shows that the maximum error when calculating METs was of 4.76%, which is within the error rate presented during the validation of the proposed formula to estimate METs based on the step rate [152].

The accuracy of the step counter (when counting steps) was also compared with a Xiaomi Mi Fitness Band [157] and a Fitness Tracker smartphone application [13]. These systems are based on a 3-axis accelerometer in order to detect the human motion. Then, the acceleration data is analysed by a tuned algorithm which looks for motion patterns in order to determine whether the movement was caused by a step or not. The three devices were simultaneously used during the experimentation. In Appendix C, the experimental set-up is graphically described for a better understanding. The ferroelectret energy harvester was worn inside the shoe, and the data was wirelessly transmitted and presented in a laptop, from where it is possible to access the information via internet. The experiment consisted in three modes: *walk*, *run* and *walk-run*. In each mode, 50 steps were taken for 5 attempts (in *walk-run* mode, 25 steps were taken walking and then 25 steps running, without stopping), giving a total of 750 steps.

Table 5.5: Effect of the step rate error when calculating METs.

Step Rate (spm)		**MET**		
Real	**Calculated**	**Expected**	**Estimated**	**Error(%)**
64	69	0.0429	0.0449	4.66
	68		0.0445	3.75
	67		0.0441	2.85
	63		0.0426	0.78
	66		0.0438	1.94
84	85	0.0508	0.0511	0.60
	78		0.0484	4.76
	85		0.0511	0.60
	79		0.0488	3.99
	86		0.0515	1.36
108	109	0.0594	0.0605	1.86
	103		0.0581	2.07
	105		0.0589	0.76
	111		0.0613	3.17
	110		0.0609	2.51
127	130	0.0675	0.0686	1.66
	129		0.0683	1.09
	128		0.0679	0.51
	124		0.0663	1.79
	132		0.0694	2.81
148	153	0.0756	0.0776	2.66
	149		0.0760	0.60
	150		0.0764	1.11
	151		0.0768	1.63
	150		0.0764	1.11

Figure 5.20 shows the comparison results of the proposed step counter and the other two devices at each experimental mode. The dotted line in green indicates the real number of taken steps. The transient step counter has a maximum average error of 13.2% in *walk-run* mode, while in *run* mode the average error is of 4.8%. In the case of the Mi Fitness Band, it presented a maximum average error of 13.6% in *run* mode and a minimum of 4.4% in *walk* mode. The smartphone shows the best performance, having a maximum error of 7.6% when running and 5.2% in *walk-run* mode. The maximum error in a single attempt for the proposed design, was of 11 steps in the first attempt of *walk-run* mode, while for the Mi Fitness Band, it was of 13 steps (first attempt of *walk-run* mode). For the smartphone application, the maximum error was of 7 steps in the third attempt of *run* mode.

The error rates when counting steps in these experiments are different to the obtained values from the *task-based* step counter presented in Chapter 3.2, for three main reasons. First, the transient step counter needs more steps in order to reach an operating voltage, because it performs more complex tasks. This causes a greater uncertainty to determine the exact number of taken steps before the system starts working. Second, a different ferroelectret insole was included in this design, which was worn inside the shoe and the step counter circuitry was carried in the hand, during the experimental routine, unlike the *task-based* step counter, which was not completely portable. Finally a newer smartphone was used to run the step counter application, and a Mi Fitness Band was worn in the wrist. However, the obtained results demonstrated that the transiently-powered step counter presented a similar accuracy when counting steps, than the battery-powered devices, without adding any extra energy storage.

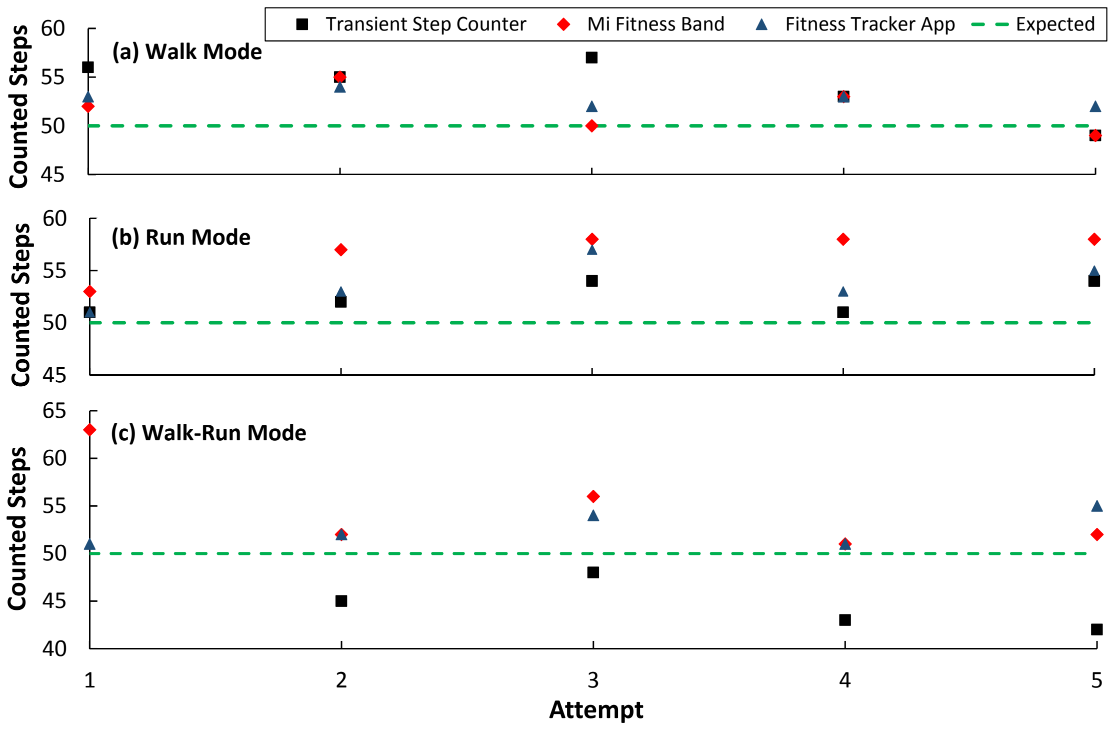


Figure 5.20: Performance comparison of the three devices in **(a)** walk mode, **(b)** run mode and **(c)** walk-run mode.

5.3 Discussion

Transient systems attempt to reduce or totally remove the need of capacitance. Researchers have proposed *task-based* and *system state retention* approaches. In the first case, the application is divided into small and indivisible tasks (so called *atomic*), which have to be completed without power interruptions. The obtained results of each task, are saved into NVM in order to retain data between power outages.

In the case of *system state retention* approaches, the system state is retained between power failures, which allows to progressively complete relatively long-term computation through various power cycles, without adding extra capacitance. However, *task-based* approaches include capacitors whose sizes depend on the amount of energy needed to complete the most energy-consuming task (typically the computation), without power interruptions. On the other hand, *system state retention* approaches may fail to execute *atomic* tasks such as wireless transmissions. In addition to these drawbacks, both schemes share an important challenge, which is keeping track of time.

In order to cope these limitations, a design framework was proposed, which incorporates the dynamics of the EH into the application design, in order to enable *task-based* and *system state retention* approaches to coexist, as well as providing a strategy to keep a sense of time from the point the system senses, until the data is timestamped by the wireless receiver.

The viability of the proposed scheme was validated through a case study: a step counter. This system was able to operate powered from a ferroelectret insole, reducing up to 60% the energy storage compared with the capacitor needed if the step counter worked under the *task-based* scheme, treating each task as *atomic*, which allows to use the decoupling capacitance as a tiny storage. From the experimental evaluation, the step counter proved to be able to perform its tasks (counting taken steps, measuring activity time, calculating step rate and METs, encrypting and wirelessly transmitting the collected data) with a similar accuracy than battery-powered devices.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Transient computing systems attempt to operate directly from the intermittent supply, which helps to simplify the circuitry and reduce both the cost and size of sensor systems. In recent years, different researchers have proposed *task-based* applications that incorporate small capacitors, whose size depends on the amount of energy needed to complete their tasks without power interruptions, and save the outcomes into a NVM. However, the size of the capacitor is defined by the biggest task (e.g. processing). Therefore, more elements such as boosters with MPPT may be needed, in order to reduce the delay caused by charging the capacitors until an operating voltage. This causes an increase in the size and cost of sensor system applications.

Other scheme, namely *system state retention*, enables relatively long-running computation in a storage-less energy harvesting sensor system, by retaining the system state into a NVM during power outages. Different *system state retention* approaches have been proposed. Their performance is affected by the variations of the energy harvesting sources. From the quantitative evaluation presented in Chapter 2, Hibernus++ showed the best performance, slightly surpassing Hibernus, but with the characteristic that Hibernus++ has a more complex algorithm with a larger number of hardware components than the second approach.

However, these approaches do not consider the interaction of the processing unit with external peripherals and may fail to execute tasks that need to be completed without power failures. Moreover, none of these approaches has proposed a strategy for keeping track of time over a diverse granularity. Besides that, not all sensor system applications are suitable to operate transiently. This thesis has addressed these challenges that are fundamental to enable transient computing applications. First, in Chapter 3, a methodology to assess whether an application is suitable to operate transiently, was presented. Then, a case study (a basic wearable step counter able only to count steps), was proposed as a motivation to probe the concept of transient computing in a real application. This case study demonstrated that the challenges previously described had to be addressed in order to enable a system to perform more complex tasks (e.g. process or wirelessly transmit data), without adding extra energy storage. It is important to emphasize that this case study is not attempting to propose a better step counter to compete with existing fitness trackers. If that was the case, a special characterization would be needed to define the energy profiles generated by the source, when different people of different heights and weights, wear the ferroelectret insole. This is because the current version works based on the amount of steps the specific user needs to take in order to charge the capacitor until the defined threshold. However, the error rate would be affected if a different user needs more or less steps to enable the system.

For transient systems to be effective, methods to retain the external peripherals state, are needed. Therefore, Chapter 4 proposed RESTOP (Retaining the State of Peripherals), a generic middleware capable of retaining the state of multiple external peripherals that are connected to an MCU through serial protocols such as SPI or I^{2}C. The reason for proposing a software approach instead of designing a non-volatile peripheral, was to retrofit to existing *system state retention* approaches, in order to enable systems to operate with off-the-shelf peripherals. On the contrary, designing a special-purpose hardware would exclude the commercially available digital peripherals to be used in transient systems. The generic middleware was validated in a system with multiple external peripherals and including Hibernus to protect system state from volatility among power outages. The obtained results demonstrated the proposed approach properly retains the peripherals state, causing a time overhead of up to 0.82% to the complete sensing application. However, a drawback of this work is that although the energy cost of restoring the state of peripherals was modeled, any additional energy used by the peripherals (e.g., a restored instruction that then triggers a radio transmission) was not taken into account. Hence, in the future, the energy requirements of the complete system have to be considered, i.e., not just the MCU. A potential solution to this is to implement a calibration routine for measuring the energy consumed when executing each peripheral instruction. Additionally, some extra functionalities (e.g., a delay function) have to be included in RESTOP in order to facilitate its usage and make it more generic.

In order to enable transient system applications to operate under sever conditions of energy availability, a novel design framework was presented in Chapter 5. The proposed scheme combines the strengths of schemes that divide the applications into small and non-interruptible tasks (*task-based*), with those that allow to split task computation among multiple power cycles (*system state retention*), as well as a strategy for time keeping. This enables to reduce the energy storage size but still meeting the application requirements. To enable timekeeping, the framework intelligently combines different solutions in order to track time during different intervals (e.g. seconds, minutes, hours).

The design scheme was validated through a case study: a transient computing step counter. The proposed application reduced the needed capacitance up to 60% and was able not only to count steps but also calculate the step rate, the activity duration and the METs consumed, as well as encrypt and wirelessly transmit the collected data. The experimental results demonstrated the system performs their functions with a similar accuracy compared with commercial fitness bands and smartphone solutions. The proposed case study may be subject to improvements. Although the aim of this work is to prove the viability of the proposed design framework, not to sell a better step counter, the system can include a more efficient power unit, which maximizes the energy obtained from the insole, and reduces the power dissipation in the diode-bridge rectifier.

In summary, this thesis has contributed with novel, relevant and power-efficient solutions to enable sensor systems, powered by highly variable and scarce EH sources, to operate transiently, without adding any extra energy storage. This has led the transient systems to perform the same tasks with a similar accuracy than typical EH systems with big energy storage units. The conclusions drawn in this research work are supported by a thorough experimental validation of the contributions made, which includes the design and implementation of a transient computing step counter capable of sensing, processing and wirelessly transmitting data.

6.2 Future Work

The research developed in this thesis has successfully answered the questions of Section 1.2. However, there are still additional investigations that could be performed due to this is an exhaustive application area.

- Operating systems have components and abstractions for peripheral interface. Therefore, in order to increase the accessibility of transient computing systems and promote standardization, a package of future work is to integrate RESTOP with a generic operating system such as TinyOS [158] or the ARM mbed OS [159].
- RESTOP scheme is not completely transparent because designers need to use the functions provided by the middleware to access the peripherals. Besides that, the approach has a time cost caused by saving each peripheral instruction before executing it. Therefore, further work may be guided to research and develop a non-intrusive approach to retain the state of external peripherals in transiently-powered EH sensor systems. A possible solution is to propose a hardware approach capable of “sniffing” in the serial transmission bus in order to directly capture the data that is being exchanged.

- The wireless communication between two systems is only successful if both the transmitter and the receiver are active for the duration of the transmission. However, in transient systems, it is not possible to accurately know when two nodes are operating in order to establish the communication. Although this research work has proved the viability of transient systems by implementing a real application (a transient computing EH step counter), the operation and synchronization of a group of transient sensor systems represents an important challenge that has not yet been solved.

- Potential applications for transient sensor systems are for operating in hard-to-reach places. Upgrading the software of a sensor network deployed in a remote location, may be impractical and with a high cost. A package of future work is to propose dynamically reconfigurable software architectures for transiently-powered energy harvesting sensor systems, in order to easily upgrade and reuse the software of applications. To reconfigure software is important for adapting applications to changes in the environments, which are directly linked with the operation of transient systems.

Appendix A

Publications

In this appendix, the following published papers are annexed:

- A. Rodriguez Arreola, D. Balsamo, G. V. Merrett, and A. S. Weddell, "RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems." MDPI Sensors Journal 2018, 18, 172, DOI: 10.3390s18010172.
- A. Rodriguez Arreola, D. Balsamo, Z. Luo, S. P. Beeby, G. V. Merrett, and A. S. Weddell, "Intermittently-powered Energy Harvesting Step Counter for Fitness Tracking." In 2017 IEEE Sensors Applications Symposium (SAS), Glassboro, NJ, USA. IEEE, 2017; pp. 1-6, DOI: 10.1109SAS.2017.7894114
- A. Rodriguez Arreola, D. Balsamo, A. K. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, and G. V. Merrett, "Approaches to Transient Computing for Energy Harvesting Systems." In Proceedings of the Third ACM International Workshop on Energy Harvesting & Energy Neutral Sensing Systems - ENSsys '15, (Seoul, South Korea), pp. 3-8, ACM Press, 2015. DOI: 10.11452820645.2820652

sensors

Article

RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems

Alberto Rodriguez Arreola *, Domenico Balsamo, Geoff V. Merrett and Alex S. Weddell

Department of Electronics and Computer Science, University of Southampton, Southampton SO 17 1BJ, UK; d.balsamo@soton.ac.uk (D.B.); gvm@ecs.soton.ac.uk (G.V.M.); asw@ecs.soton.ac.uk (A.S.W.)
* Correspondence: ara1g13@soton.ac.uk; Tel.: +44-(0)23-8059-3119

Received: 10 November 2017 ; Accepted: 5 January 2018; Published: 10 January 2018

Abstract: Energy harvesting sensor systems typically incorporate energy buffers (e.g., rechargeable batteries and supercapacitors) to accommodate fluctuations in supply. However, the presence of these elements limits the miniaturization of devices. In recent years, researchers have proposed a new paradigm, transient computing, where systems operate directly from the energy harvesting source and allow computation to span across power cycles, without adding energy buffers. Various transient computing approaches have addressed the challenge of power intermittency by retaining the processor's state using non-volatile memory. However, no generic approach has yet been proposed to retain the state of peripherals external to the processing element. This paper proposes RESTOP, flexible middleware which retains the state of multiple external peripherals that are connected to a computing element (i.e., a microcontroller) through protocols such as SPI or I^{2}C. RESTOP acts as an interface between the main application and the peripheral, which keeps a record, at run-time, of the transmitted data in order to restore peripheral configuration after a power interruption. RESTOP is practically implemented and validated using three digitally interfaced peripherals, successfully restoring their configuration after power interruptions, imposing a maximum time overhead of 15% when configuring a peripheral. However, this represents an overhead of only 0.82% during complete execution of our typical sensing application, which is substantially lower than existing approaches.

Keywords: energy harvesting; external peripheral; sensor system; transient computing

1. Introduction

Energy harvesting (EH) potentially enables the long-term deployment of low-power sensor systems without the need to replace batteries. However, EH sources are usually intermittent and unpredictable because they depend on external conditions (i.e., availability of energy to be harvested) [1]. To overcome this limitation, systems typically integrate energy storage devices (e.g., supercapacitors or rechargeable batteries) to smooth out supply variations. This approach is known as energy-neutral operation, where energy storage is used to balance the stored energy with the long-term energy consumed and, thus, sustain operation during power shortages [2]. This is shown in Figure 1a, where energy storage is used to sustain computation when there is insufficient energy being harvested. However, energy storage increases the system's cost, size and mass. Transient computing (Figure 1b) aims to power systems directly from the EH source, operating when energy is available and retaining system state during supply interruptions.

Various software solutions for transient computing have coped with power intermittency by saving the system state (contents of main memory, core and general purpose registers) into a Non-Volatile Memory (NVM) [3–7]. Thus, after a supply interruption, the system state is restored and the program continues from the point where it was interrupted, instead of restarting from the beginning. Recent hardware approaches overcome the need to save and restore the system's state by using non-volatile processors [8]. These approaches are only effective in retaining the state of on-chip peripherals that are controlled by the special function registers of the microcontroller unit (MCU), e.g., internal ADCs. However, the vast majority of sensing systems also include external sensors, actuators, radio transceivers, etc. A recent approach [9] attempted to save and restore the state of external peripherals; however, it only operates with peripherals that interact with the MCU through SPI. Moreover, this approach requires the user to make several adjustments depending on the type and number of peripherals connected (it is not generic) and causes a high time overhead.

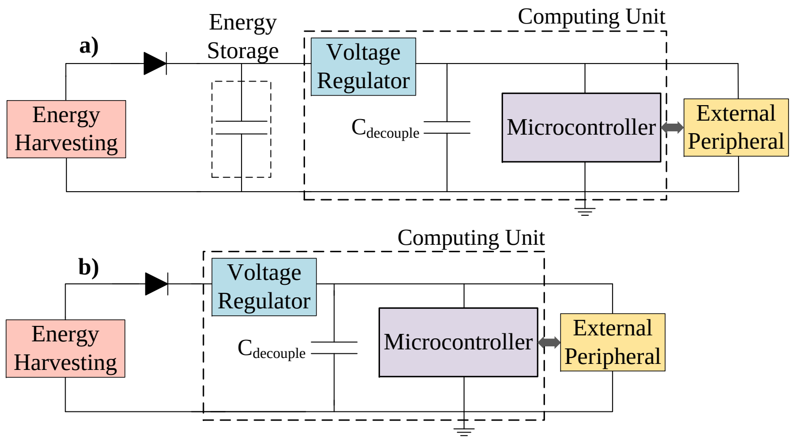


Figure 1. Schematic of: (**a**) an energy-neutral; and (**b**) a transient, EH sensor system.

In this paper, we present RESTOP (REtaining the STate Of Peripherals), a novel middleware to retain the state of digitally interfaced peripherals in transiently-powered systems. RESTOP provides generic functions to read data from the external peripherals or write to them, keeps track of and saves the transmitted configuration data, and hence retains the peripheral state. Thus, after a power failure, the peripheral state can be restored, without requiring for the user to implement the save and restore functions for each attached peripheral and indicate the order in which they have to be restored. The key contributions of this work are:

- A novel and generic approach for transient computing systems, which retains the state of multiple digitally interfaced peripherals between power outages (Section 3).
- Implementation of the approach into a middleware (available open-source to download from http://www.transient.ecs.soton.ac.uk) that works with both SPI and I^2C protocols (Section 4).
- A thorough practical evaluation of RESTOP in order to validate the operation of the middleware and the time overhead it causes in an intermittently-powered sensor system (Section 5).

RESTOP can be integrated into any of the existing approaches to transient computing [3,4,6,7,10]. Experiments demonstrate that RESTOP is able to retain and restore peripheral state with a peripheral configuration time overhead of up to 15%. However, this represents an overhead of only 0.82% during complete execution of our typical sensing application.

2. Problem Statement and Motivation

A transiently-powered sensor system could not operate properly unless the peripheral state was restored after a power outage. Figure 2 shows an example of an incorrect operation that may occur in a conventional transient system (e.g., HarvOS [10]). The application first configures the serial protocol (Configure_protocol) to communicate with the external peripheral (in this example, a digital sensor). Then, the MCU sends a reset instruction to the peripheral (Sensor_reset) and configures the sensor to start sampling data (Configure_sensor). However, before the peripheral starts sampling, a power failure occurs. Then, when the energy is again available the system state is restored. Nevertheless, the peripheral's state is not restored, i.e., the sensor is not properly configured (the configuration was not saved into NVM). Therefore, the sensor would revert to its default configuration after the power outage, and the program would be unaware of this.

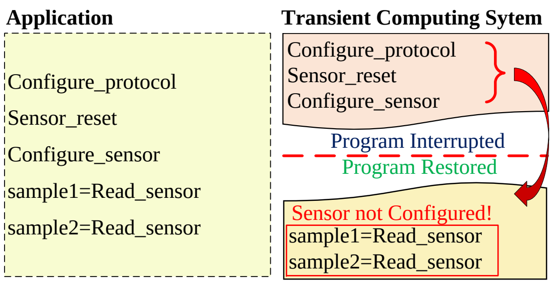


Figure 2. Operation of existing transient computing approaches when working with external peripherals after a power failure.

There are various software approaches to transient computing that save a copy of system state into NVM (snapshot), before a power failure [3,4,6,7,10] and restore it when the energy is again available instead of starting from the beginning. However, these solutions are focused on retaining the state of the main memory, core registers (i.e., program counter, stack pointer, link register and general purpose registers) and peripheral registers (which are used to control internal peripherals such as ADC, DAC, GPIO, etc). These approaches are not concerned with retaining the configuration of external peripherals because they are not included in the design. This has led researchers to engage in developing solutions that allow the state of external peripherals to be retained between power outages.

Berthou et al. [9] proposed Sytare, a software approach which retains not only the system state but also the configuration of external peripherals attached to the MCU through SPI. This approach includes so-called kernel code, which operates between the main application and the library to access the external peripheral features (peripheral driver), and it is in charge of saving and restoring the peripheral state. However, the user not only has to write the function to configure the peripherals but also implement a structure called device context for each attached peripheral. This structure is used by Sytare to encapsulate the functions and save the data exchanged between the MCU and the external peripheral. Moreover, the user has to write a function to restore the peripheral configuration (one per connected peripheral). If more than one peripheral is attached, the developer has to indicate in which order they have to be restored, because this solution would not work in a system where the peripherals are accessed in a different order from one cycle to another. However, it excludes essential details: it does not describe how the developer has to change each peripheral driver in order to update the structure (device context) needed to avoid peripheral volatility and how to implement the restore function for each attached peripheral. Besides that, Sytare incurs a time overhead of over 30 µs per peripheral instruction because it needs to save the peripheral state each time an instruction is issued. This imposes an overhead of up to 137% when configuring a radio transceiver.

Recently, designers have oriented their research towards implementing non-volatile solutions for external peripherals. Li et al. [11] proposed a ferroelectric non-volatile flip-flop based input-output (IO) architecture that aims to reduce the initialization overhead caused by power outages. They replaced typical IO D-type flip flops with non-volatile flip-flops by adding two ferroelectric capacitors. Thus, when a power outage occurs, the peripheral configuration is retained in local ferroelectric capacitors, allowing a fast backup operation. However, this approach is focused exclusively on sensors, requires special-purpose hardware and does not offer a solution for off-the-shelf peripherals. Hardware approaches such as Non-Volatile Processors (NVPs) [8] attempt to save *in-place* snapshots by adopting non-volatile SRAM and registers. However, these hardware approaches do not offer solutions as they only retain the system state (main memory and processor registers), but not the configuration of digitally interfaced peripherals. Other solutions such as WISP [12], WISPCam [13] or federated energy storage [14] do not snapshot the configuration of the external peripherals because they operate only when the energy stored in small capacitors is enough to complete the required task. The peripherals are configured from scratch and perform the same function each time they are enabled.

In summary, there is an unmet need for a generic solution capable of retaining the state of multiple peripherals connected to the MCU through external interfaces such as I^2C and SPI. This would enable the state of complete sensor systems, e.g., incorporating an MCU, a digital luminosity sensor [15] and a transceiver [16].

3. RESTOP: A Middleware for Peripheral State Retention

External digital peripherals are typically connected to the MCU via serial protocols, unlike the on-chip peripherals that are controlled by special function registers. Figure 3 shows a block diagram of an MCU interacting with three peripherals: an analog sensor connected via an on-chip ADC, a transceiver connected through SPI and a digital sensor attached via I^2C. In Section 2, the limitations of existing transient computing systems when working with digitally interfaced peripherals were described. To address them, we propose RESTOP, a middleware which is generic for different peripherals and serial communication protocols (e.g., SPI or I^2C), capable of retaining (saving and restoring) peripheral configurations between power failures. The following terms are introduced here to aid understanding of the operating principles of RESTOP:

- Peripheral operation: The action to be performed on the peripheral, i.e., write or read.
- Peripheral instruction: The information required by the system to issue the operation on the peripheral (e.g., peripheral address, register to be read, value to be written on the register, etc.).
- Parameters: Elements that constitute each function that executes the peripheral instructions.

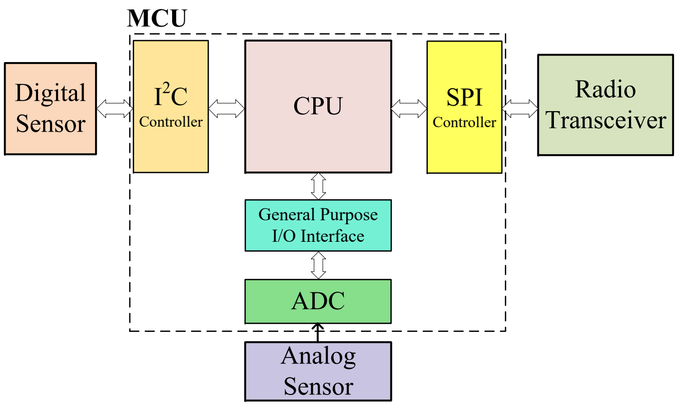


Figure 3. Block diagram of an MCU interacting with different peripherals.

Figure 4 shows the parts that make up RESTOP and how this middleware interacts in a sensor system when saving and executing a peripheral instruction (the Restore module is later described in Figure 6). Each peripheral instruction is issued through the generic functions provided by RESTOP and saved in a history table. In order to execute the instruction on the peripheral, RESTOP complements the information entered through the generic functions with that defined in a configuration file (these modules are detailed in Section 4). The history table can either be: (1) placed in main memory; or (2) directly located in NVM. In the first case, the developer can utilize any of the existing approaches for transient computing [3,6,7,10] that can save the system state (including main memory) to NVM at the right time before a power failure. Thus, after a power outage, the system state (including the instruction table) is restored and then RESTOP restores the peripheral configuration by re-issuing the instructions from the table. In the second case, RESTOP has to be included with interrupt-based approaches such as Hibernus [6] and QuickRecall [4] in order to ensure that the system and peripheral states are restored in the same point where they were interrupted, i.e., there is no more code executed after the snapshot. Thus, it is possible to maintain coherence, avoiding the table being modified after the last snapshot was saved. This is important because in a transient system with external peripherals, repeated peripheral instructions (or system code) may result in functionality issues [5]. In Section 3.1, the different factors that RESTOP considers before saving and executing a peripheral instruction are described. Later, in Section 3.2, the process of restoring the peripheral configuration is detailed.

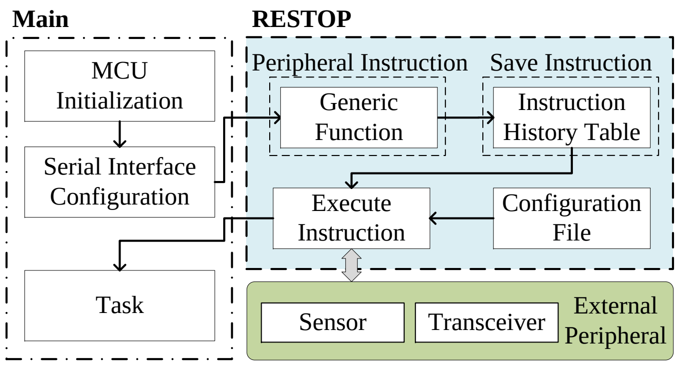


Figure 4. Diagram of RESTOP interacting with the application and peripherals.

3.1. Saving and Executing Peripheral Instructions

Figure 5 details the process of saving and executing a peripheral instruction issued over a serial protocol. The decision about whether RESTOP should save an instruction in the table is made by the programmer at design time for each peripheral instruction, considering four choices:

1. Not-save: The user might consider that a certain instruction should not be saved because it is not a peripheral configuration instruction (e.g., reading a status register).
2. Save: The issued instruction must be saved in the history table without checking whether a similar instruction (i.e., with same peripheral address and register value) was previously saved.
3. Save-but-replace: The issued instruction would replace any other similar instruction (i.e., same peripheral address and register value) that was previously saved in the history table.
4. Preserve: An instruction has to be kept in the history table regardless of whether a similar instruction is later issued.

As shown in Figure 5, RESTOP first checks whether the issued instruction is applying a *Reset* on the peripheral. *Reset* is a *write* peripheral instruction that, when issued, causes RESTOP to delete all instructions saved in the table for that peripheral. This condition is important for efficient memory usage because it is unnecessary to keep peripheral instructions prior to a *Reset*. Next, RESTOP checks whether the issued instruction has to be saved. If not (*Not-save*), RESTOP executes the instruction on the peripheral and the application continues to the next task. If the peripheral instruction must be saved, RESTOP considers two choices: *Save* and *Save-but-replace*. If the first option is asserted, the issued instruction is saved in the table and then executed on the peripheral. In case that *Save-but-replace* is selected, the middleware looks in the table for a similar instruction previously saved. If a similar instruction is found in the table, RESTOP checks whether the instruction is marked to be preserved (Preserve). If so, the instruction is saved in a new element in the table and then executed. Preserving an instruction, instead of replacing it, is particularly useful for certain peripherals that need an *unlock* instruction, which enables the peripheral to be accessed or configured. Thus, each unlock instruction is saved in the table no matter how many times it is repeated. If *Preserve* is not asserted, the previous instruction is deleted and the issued one is saved instead, but keeping the chronological order in which the instructions are sent. Keeping track of the instruction sequence is important because peripherals often need the registers to be accessed in a certain order to operate properly (e.g., in a transceiver, it has to set the channel before transmitting the data, not the other way around).

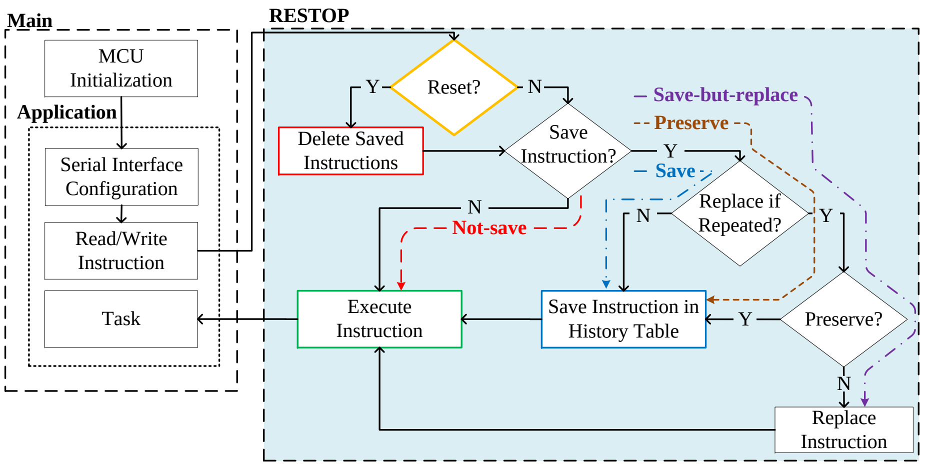


Figure 5. Path followed to save and execute an instruction depending on the selected criteria.

It is important to mention that each instruction must be saved before executing it in order to cope with power failures occurring before peripheral access is completed, avoiding consistency issues. Issuing an instruction on a serial interface involves, among other things, enabling the peripheral, sending the register to be read or written, waiting for the transmission to be completed, and disabling the peripheral. This sequence has to be completed without interruptions, i.e., if a supply failure occurs while an instruction is issued, the sequence has to be restarted from scratch when power recovers (e.g., it is not possible to send a partial packet). Therefore, if the instruction was not saved into the history table before the power failure, it would neither be properly executed because the sequence was interrupted, nor restored because it was not saved. A possible concern is that, when the peripheral is a radio transceiver, the user may send the instruction with the packet to be transmitted, but there would be no certainty that it was sent (i.e., a power outage may occur before packet transmission has completed). When restoring the transceiver state, the packet would be resent, leading to a duplicate packet being received. However, this can occur normally in a noisy wireless network, and communication protocols are typically already present to ignore duplicate packets and request those missed.

3.2. Restoring Peripheral State

The restore routine is shown in Figure 6. This is executed after the system state has been restored by the transient computing approach used to protect the system from volatility. Here, RESTOP fetches each instruction from the history table and issues it over the digital interface in the correct order. This process is repeated until all saved instructions are executed, and, therefore, the state of the peripheral is restored. Once the routine is completed, the main application continues its execution from the point where it was interrupted by the power outage.

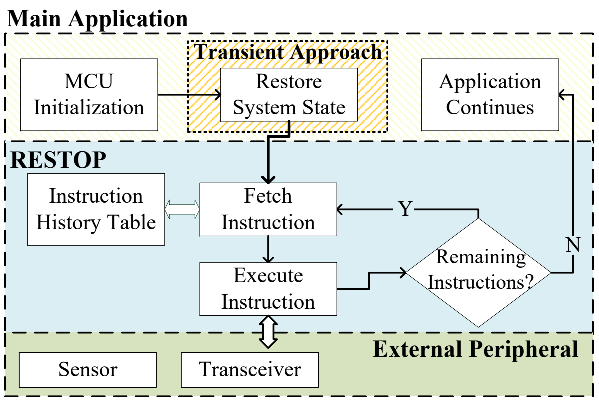


Figure 6. RESTOP has to re-issue each saved instruction to restore the peripheral state, after a power outage.

4. Software Algorithm Design

The requirement for a generic interface implies that it can work across different protocols and handle different types of instructions, and that it fits in not only with programming structures but also with the use cases of transient technologies. In this section, we describe the implementation of the three main elements that compose RESTOP. First, we detail the RESTOP functions that will execute the peripheral instructions (Section 4.1). Then, we list the parameters that have to be saved to properly describe each peripheral instruction without losing generality, and how the users will introduce the required information for each instruction (Section 4.2). Lastly, we explain how the instruction history table will be efficiently built in terms of time and memory usage (Section 4.3).

4.1. Function Implementation

Defining the functions to execute each peripheral configuration instruction, and the information to be saved from them, required the analysis of various peripherals with digital interfaces, identifying patterns that help to implement the RESTOP functions that are generic for different peripherals and serial protocols. From this analysis, we found that peripheral instructions perform two main operations: read and write. However, the number of parameters required to perform these operations varies from one peripheral to another. For example, some peripherals [17] need a 1-byte parameter called *command* to indicate the type of operation the issued instruction will perform (i.e., *read* or *write*) on a register address (i.e., the sequence would be <*command byte*><register address><data byte>). Others allow certain *single byte instructions* (no data is transferred), usually so-called *command strobes* that cause internal sequences to start in the peripheral, e.g., some peripherals have a single header byte that, when addressed, starts a self-calibration routine to define the sampling frequency [17]. Most peripherals support multiple byte transfers also known as *data burst transmissions* which send first the register address and then a sequence of different values to write to this address (this can also be applied for read operations).

Considering this analysis, we have defined the functions that will be used by the middleware to save, execute and restore each instruction:

1. RESTOP_read(): Returns the read value from the desired register.
2. RESTOP_write(): Writes a value into a peripheral register.
3. RESTOP_strobe(): Performs write operations that, unlike RESTOP_write(), executes single byte instructions.
4. RESTOP_restore(): Restores the peripheral state by executing all the instructions saved in the history table after a power failure. It has to be incorporated into the restore routine of the main application.

These functions have to be used by the developer to configure the peripherals and obtain data from them. The parameters of each function are described in Section 4.2.

4.2. Parameters to be Saved and Configuration File

Following the definition of the generic functions, the parameters that will constitute each peripheral instruction need to be defined. For this purpose, we separate the *dynamic* parameters that vary from one peripheral instruction to another, and those that are *static* for each peripheral attached to the system. Table 1 shows the *dynamic* parameters that will be saved in the history table. The size (number of bits) of each parameter varies depending on the information that they contain. The first parameter (*Protocol*) is a 1-bit flag to indicate the serial protocol type of each peripheral (0$\rightarrow$SPI; 1$\rightarrow$I^{2}C). Parameter *Burst* is also a 1-bit flag that has to be set to 1 when the function will execute a *burst read/write* instruction. *Read* is a flag used by RESTOP to distinguish when the instruction is for a *read* (*R*=1) or *write* (*R*=0) operation. *Prv* is a 3-bit flag that can have five different values as shown in Table 2. These values are defined following the criteria described in Section 3.1. Thus, the first three options indicate whether the instruction will not be saved in the table (*Not-save*), will be saved in a new element (*Save*) or will replace a similar one if it was previously saved in the table (*Save-but-replace*). The last two criteria, shown in Table 2, indicate the instruction will be not only saved but also preserved in the table regardless of whether a similar instruction is later issued (*Preserve*).

Table 1. Dynamic parameters to be considered for describing a peripheral instruction.

Parameter	Size (Bits)	Definition
Protocol	1	Serial Protocol of Peripheral
Burst	1	Burst instruction
Read	1	Read or write instruction
Prv	3	Preserve flag
ID	3	Peripheral identification
Register	8	Address to be accessed
Value	8	Value to be written in the register
Next	8	Pointer to the next instruction
Previous	8	Pointer to the previous instruction

The parameter *ID* is used to indicate the peripheral to which the saved instruction corresponds. A system may have more than one peripheral attached to the MCU, hence, we would have to identify which instruction corresponds to each peripheral. The parameters *Register* and *Value* are each one byte, corresponding to the register width of typical digital interface peripherals. *Next* and *Prev* are used to keep track of the order in which the instructions are issued. Thus, when a new instruction replaces another previously saved or is added in a new element in the table, RESTOP can keep the chronological sequence in order to properly restore the peripheral state after a power outage. The size of these parameters is one byte each too, allowing the system to map up to 256 peripheral instructions. This is considered sufficient for most peripherals (e.g., a typical transceiver [16] is configured with less than 50 instructions), but their size could be expanded for particular scenarios.

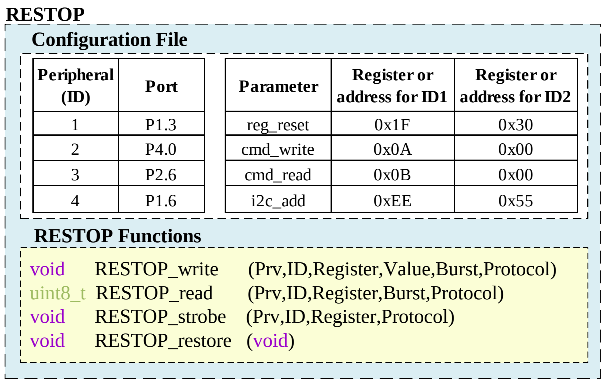
Table 2. Values that Prv flag can have.

Bit 2	Bit 1	Bit 0	Criteria
0	0	0	Not-save
0	0	1	Save
0	1	0	Save-but-replace
1	0	1	Save and Preserve
1	1	0	Save-but-replace and Preserve

In the case of the *static* parameters, we define four:

- reg_reset: To declare the register address that represents a reset instruction in each peripheral.
- cmd_write: This parameter is used to introduce the *write command* value for peripherals that need it as explained in Section 4.1.
- cmd_read: This is similar to the previous one, but this is the command for reading operations. If no *command* is needed in a peripheral, it has to be filled with zeros.
- i2c_add: This parameter is used to define the address of the peripherals that are attached to the MCU through I^2C protocol.

The *static* parameters are declared in a configuration file unlike the *dynamic* ones, which are saved in the history table and entered by the user through the generic functions (except *R*, *Next* and *Previous*, which are defined by RESTOP). To reset a peripheral, the user not only has to use the *RESTOP_write()* function but also declare the register address in *reg_reset*. As mentioned in Section 3.1, *Reset* is a *write* instruction that when issued causes RESTOP to delete the saved instructions that correspond to the reset peripheral. *cmd_write* and *cmd_read* have to be filled with zeros for those peripherals that do not need a *command* parameter (described in Section 3.1). If an I^2C peripheral is attached to the system, its address has to be written in *i2c_add*, if not, this parameter has to be filled with zeros as well. Figure 7 shows the *configuration file* with example values and the description of the *dynamic* parameters that each generic function requires. The *configuration file* also includes the microcontroller ports where the peripherals are attached. For example, if a user connects a peripheral to port P1.3, it will be marked with the peripheral identification ID1.


RESTOP

Configuration File

Peripheral (ID)	Port
1	P1.3
2	P4.0
3	P2.6
4	P1.6

Parameter	Register or address for ID1	Register or address for ID2
reg_reset	0x1F	0x30
cmd_write	0x0A	0x00
cmd_read	0x0B	0x00
i2c_add	0xEE	0x55

RESTOP Functions

```
void     RESTOP_write     (Prv,ID,Register,Value,Burst,Protocol)
uint8_t  RESTOP_read      (Prv,ID,Register,Burst,Protocol)
void     RESTOP_strobe    (Prv,ID,Register,Protocol)
void     RESTOP_restore   (void)
```

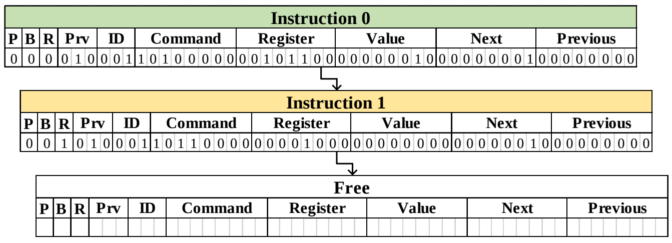
Figure 7. Configuration file with example values and the functions description.

4.3. Instruction History Table

As already mentioned in Section 4.2, RESTOP requires an *instruction history table* to which it can save the data exchanged between the MCU and the peripherals. It is based on a linked list in which each element corresponds to a peripheral instruction. A static array of structures can simplify the implementation of this table, as allocating memory dynamically may substantially increase time and memory overheads [18,19]. The maximum size of the table (i.e., the number of available locations where the instructions are saved) is defined by the user in the *configuration file*. Figure 8 shows an example of the history table with two saved instructions. From the values showed in the figure, the *static* parameters and the generic functions for the two saved instructions would be as follows:

- reg_reset [] = {0x1F}
- cmd_write [] = {0x0A}
- cmd_read [] = {0x0B}
- i2c_add [] = {0x00}
- RESTOP_write(2,1,0x2C,0x02,0,0)
- RESTOP_read(2,1,0x08,0,0)

In this example, the value of the *Protocol* flag (P = 0) indicates both instructions correspond to the same SPI peripheral, which is connected to the P1.3 port (ID = 1). Therefore, the parameter *i2c_add* is filled with zeros. The *Burst* flag (B) is zero which means these are not *burst* instructions. According with the *Read* flag (R), the first instruction is to *write* on the peripheral (R = 0) and the second is to *read* from it (R = 1). The *Prv* flag value is 2 in both saved instructions, which means that they would replace any similar instruction (same peripheral, command and register) previously saved, but they can also be replaced if a similar instruction is later issued (*Preserve* bit = 0). The attached peripheral needs a *command* to indicate when the instruction is to write (0x0A) and when to read (0x0B). If an instruction was issued with a register value of 0x1F, RESTOP would apply a reset in the peripheral and delete the two instructions from the table.



Instruction 0									
P	B	R	Prv	ID	Command	Register	Value	Next	Previous
0	0	0	0 1 0	0 0 1	1 0 1 0 0 0 0 0	0 0 1 0 1 1 0 0	0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0

Instruction 1									
P	B	R	Prv	ID	Command	Register	Value	Next	Previous
0	0	1	0 1 0	0 0 1	1 0 1 1 0 0 0 0	0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0

Free									
P	B	R	Prv	ID	Command	Register	Value	Next	Previous

Figure 8. Instruction history table of two saved instructions.

5. Experimental Validation

RESTOP has been practically implemented and experimentally validated. To allow computation to span across power cycles, we combined RESTOP with Hibernus [6]. This solution was chosen because it is platform and application agnostic, and has excellent performance in terms of energy and time overhead [20]. However, we believe that RESTOP can be integrated with any other software approach for transient computing. Figure 9 shows an example application before (Figure 9a) and after (Figure 9b) incorporating the proposed middleware. The example code includes Hibernus to retain the system state between power outages. The inclusion of RESTOP in an application is simple. The developer only has to import the configuration file (*Config.h*) and the library that contains RESTOP functionality ($RESTOP\_func.h$), and use the RESTOP functions (described in Section 4.1) to configure the peripherals and read data from them. In order to restore the peripheral state after a power outage, the RESTOP restore function has to be included in the restore routine of the transient approach as shown in Figure 9b.

The voltage threshold at which Hibernus restores the system state (V_R) has to be adjusted because now the system incorporates external peripherals whose state is restored as well. Therefore, we describe how V_R is modified for Hibernus, which is used in our validation (a similar modification would need to be made for other approaches). In this sense, it is necessary to first calculate the amount of energy required to restore the state of attached peripherals (E_{r_ps}), which is given by:

$$E_{r\_ps} = \sum_{i=1}^{n} \left(P_{p_i} \sum_{j=1}^{m_i} T_{p_i inst_j} \right) \tag{1}$$

where n is the number of attached peripherals, P_{p_i} is the power consumed by the system while undertaking serial communications with each peripheral, m_i is the number of saved instructions for each attached peripheral and $T_{p_i inst_j}$ is the time taken by the system to issue each instruction to the peripheral. These parameters (power and time) may be obtained from datasheets, or measured experimentally. The time varies from one instruction to another depending on the data rate of each peripheral and the number of bytes that are transmitted for each instruction. In Section 4.1, we detailed how the number of parameters that are transmitted for each instruction (i.e., one parameter is equal to one byte) varies by peripheral. It is important to mention that Equation (1) only accounts for the power consumption of the MCU. The effect of issuing the instructions may cause additional energy to be consumed by the external peripherals, e.g., a restoration of state causing a wireless transceiver to make an energy-intensive radio transmission. This is not currently modeled, but is a potential area of future investigation.

```
#include "hibernus.h"

int main(void)
{
      // Hibernus
      if(flag) Restore();     //Restore System State
      else     Initialise();  //Initialise Hibernus

      // Main Application goes here
      Protocol_init();   //Initialise Serial Protocol

      // Functions to write in the peripheral or read from it
      Write(Register,Value);   //Write a value
      read = Read(Register);   //Read a value
}

void Restore(void)
{
      //Hibernus Code to Restore System State
}
```

(**a**) Application without RESTOP.

```
#include "hibernus.h"

#include "Config.h"        //Configuration file
#include "RESTOP_func.h"   //RESTOP functionality

int main(void)
{
      // Hibernus
      if(flag) Restore();     //Restore System State
      else     Initialise();  //Initialise Hibernus

      // Main Application goes here
      Protocol_init();   //Initialise Serial Protocol

      // Functions to write in the peripheral or read from it
      RESTOP_write(Prv,ID,Register,Value,Burst,Protocol);    //Write a value
      read = RESTOP_read(Prv,ID,Register,Burst,Protocol);    //Read a value
}

void Restore(void)
{
      //Hibernus Code to Restore System State

      RESTOP_restore();   //Restore Peripheral Configuration
}
```

(**b**) Application including RESTOP.

Figure 9. Example code of how to use RESTOP in an application, including Hibernus, showing: (**a**) code without RESTOP; and (**b**) including RESTOP.

Once E_{r_ps} is calculated, and considering the energy required to restore the system state (E_{r_sys} [6]), V_R can be calculated as follows:

$$V_R = \sqrt{\frac{2(E_{r\_sys} + E_{r\_ps})}{C} + V_{min}^2} \tag{2}$$

where V_{min} is the minimum voltage required by the system to operate and C is the total capacitance on the supply lines, which can be used as an energy buffer. The process of calculating E_{r_ps} and adjusting V_R is performed at the beginning of the snapshotting routine, in order to guarantee that the restore threshold is properly set before a power failure occurs. Although Equation (1) is performed once per supply interruption, a running total of $T_{p_i inst_j}$ is updated each time a peripheral instruction is saved. This reduces the complexity of the calculation that needs to be performed at the start of the snapshotting procedure. Thus, V_R can be dynamically adjusted considering the number of saved instructions (provided by RESTOP) for each attached peripheral.

An important concern is the size of C. Transient computing schemes commonly use only the system decoupling capacitance, $C_{decouple}$ (Figure 10), but this could be insufficient in systems interfacing with external peripherals. It may be necessary to introduce additional capacitance to deliver reliable operation. To do this, and for design purposes only, the worst case of energy used for restoring the peripheral state (E_{r_max}) has to be calculated. In this sense, Equation (1) is simplified as follows:

$$E_{r\_max} = P_{p\_max} \cdot n_{inst} \cdot T_{p\_max} \tag{3}$$

where P_{p_max} corresponds to the maximum power consumed by the system when an instruction is issued, n_{inst} is the maximum number of instructions than can be saved in the *instruction history table* and T_{p_max} is the *longest* time taken to issue a single instruction. Once E_{r_max} is obtained, and with knowledge of the V_{min} and V_{max} (the system's maximum operating voltage), the required C can be calculated as:

$$C \geq \frac{2(E_{r\_sys} + E_{r\_max})}{V_{max}^2 - V_{min}^2} \tag{4}$$

If $C > C_{decouple}$, RESTOP will require additional capacitance to supplement the decoupling capacitance. However, no other hardware changes are required.

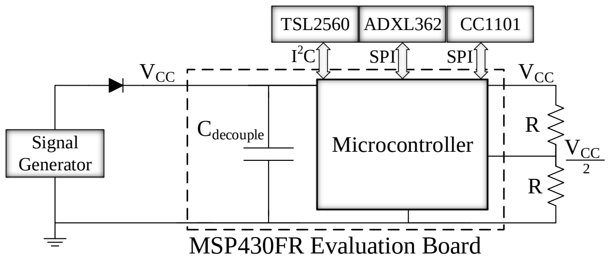


Figure 10. Schematic of the test platform, including the external peripherals.

Figure 10 shows the experimental set-up which consists of a test board and three peripherals. The chosen board was the MSP-EXP430FR5739 [21], which contains an MCU with FRAM, an on-chip comparator and supports SPI and I^2C communication protocols. The comparator is used by Hibernus to monitor the input voltage. It was configured with an on-chip variable reference voltage generator and an external voltage divider (R = 1 MΩ) giving $V_{CC}/2$ as input. We considered three different external peripherals: an ADXL362 digital accelerometer [17], a TSL2560 digital luminosity sensor [15] and a CC1101 radio transceiver [16]. The accelerometer and the transceiver are attached to the MCU through SPI, while the luminosity sensor is accessed via I^2C. Each peripheral was tested separately (i.e., only one peripheral is used for each of the tests), giving three different scenarios in total. $C_{decouple}$ represents the total decoupling capacitance of the board, which is 20 µF. To verify whether additional capacitance was needed, we evaluated the worst case energy use for each of the attached peripherals and the minimum capacitance needed for each scenario; the results are listed in Table 3.

Table 3. Worst case energy use for each peripheral, and the minimum capacitance needed.

Peripheral	E_{r_max} (µJ)	C (µF)
Accelerometer	1.40	1.58
Luminosity	2.87	2.76
Transceiver	9.37	3.36

It was therefore concluded that $C_{decouple}$ was sufficient for all cases, and hence no additional capacitance was needed. The whole system was powered by two different signals:

1. A half-wave rectified sinusoidal signal with ±3.4 V amplitude operating at a frequency of 6 Hz, to emulate an intermittent source, in order to validate whether RESTOP is able to retain the peripheral's state between power failures.
2. A square wave signal with 3.4 V amplitude and variable duty cycle, sweeping the active time from 10 ms to 100 ms, to measure the time overhead caused by RESTOP with respect to the total application execution time.

The aim of these variable signals is to emulate intermittent sources. Behaviour with a real EH source was not evaluated, as this has already been demonstrated for Hibernus-based systems in [6,7,20].

5.1. Accelerometer

To validate the proper operation of RESTOP with the accelerometer, we implemented an application that changes the output data rate (ODR) to reduce the current consumption of the sensor. As shown in Figure 11, after configuring the SPI protocol, the accelerometer is reset and the ODR is set to 50 Hz (ODR = 0x02). Then, the accelerometer is configured in measurement mode and the application enters in a loop where the three axes are read to detect movement at each iteration. During the time the program is running inside the loop, a voltage drop occurs and the snapshotting routing of Hibernus is called. There, E_{r_ps} is calculated using Equation (1) and the obtained value is 0.6 µJ. Substituting it in Equation (2) and considering E_{r_sys} = 5.7 µJ [6], the new restore threshold is set to 2.15 V. After V_R is adjusted, the system state is saved in NVM. Later, when the power is restored, an ODR reading is taken before and after restoring the accelerometer state. This step was purely for testing purposes in order to check RESTOP operation: the ODR reads would not be needed in a real application. Thus, the ODR value read before restoring has to be the default (ODR = 0x03), whilst the value after restoring has to be the same as before the power failure (ODR = 0x02). As we can see in Figure 12, the value read before restoring the peripheral state is the default (ODR = 0x03), but once it is restored, the ODR value is the same as before the interrupt. This shows that RESTOP is able to restore the accelerometer state.

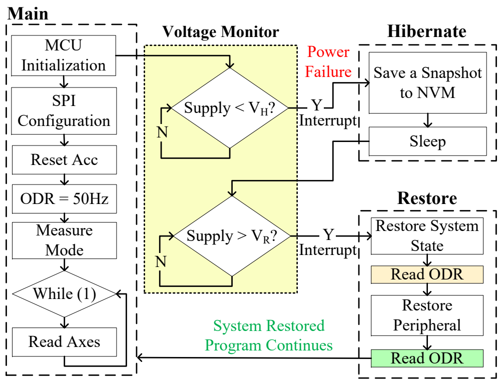


Figure 11. Testing routine to validate RESTOP with the accelerometer.

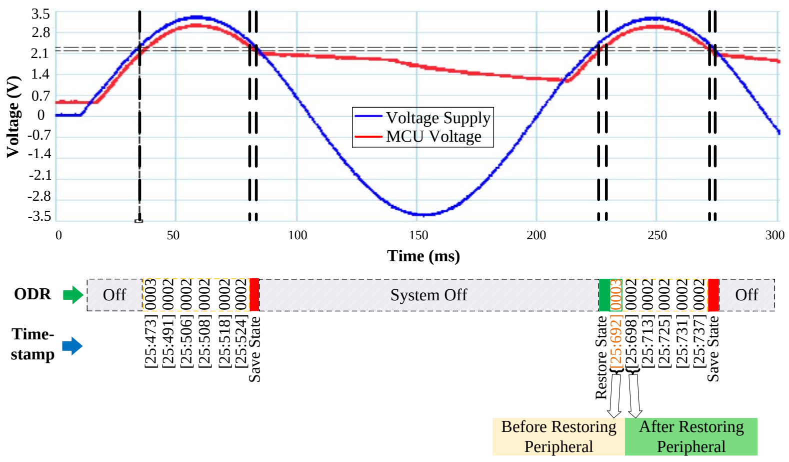


Figure 12. Operation of the accelerometer testing routine. After the power failure, ODR is read before and after RESTOP restores the accelerometer state.

5.2. Luminosity Sensor

RESTOP was tested with a luminosity sensor to validate the proposed middleware with an I^2C peripheral. Figure 13 shows the test algorithm, which consists of initializing the MCU, configuring the I^2C protocol, and then changing the integration time (T_{int}) from the default value (400 ms) to 13.7 ms. T_{int} defines the time after which the ADC channels begin a conversion. Once the integration time was changed, an end-of-conversion signal is configured in order to generate an interrupt when an ADC conversion is completed. Thus, the light intensity value is available in the data registers after

13.7 ms. Figure 14 shows the experimental results. After configuring the sensor, the light intensity is continuously read until the input voltage drops and the snapshotting routine is executed. In the same way as with the accelerometer, E_{r_ps} and V_R are calculated. However, for the luminosity sensor, the minimum operating voltage is 2.6 V, which is then defined as V_{min} in Equation (2) (unlike the accelerometer's, which is 2 V); therefore, the obtained values for E_{r_ps} and V_R are 1.72 µJ and 2.74 V, respectively. To validate the proper operation of RESTOP, the integration time register is read before and after restoring the peripheral state. As we can see in Figure 14, the value read before restoring the peripheral configuration is T_{int} = 0x02 corresponding to an integration time of 400 ms. Then, when the peripheral state is restored, the value read is T_{int} = 0x00, which corresponds to 13.7 ms. This can also be proved because the end-of-conversion interrupt signal of the sensor is enabled every 13.7 ms, which means the sensor's configuration was successfully restored by RESTOP.

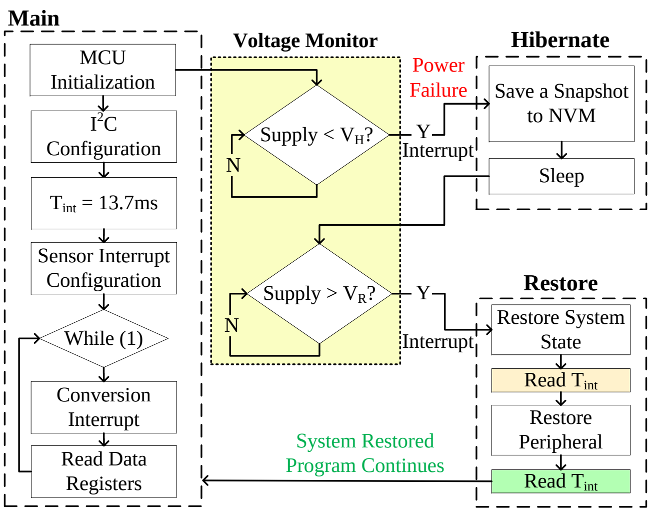


Figure 13. Testing routine to validate RESTOP with the luminosity sensor.

5.3. Transceiver

The operation of RESTOP was also validated with a CC1101 radio transceiver. Exclusively for debugging purposes, we implemented a routine (Figure 15) that initializes the MCU, configures the SPI protocol, resets and configures the peripheral and then, inside an infinite loop, the program changes the transmission channel (from 0 to 20) and sends a packet at each cycle. The idea is that we can check the channel number before the power failure, and before and after restoring the peripheral state. This is to verify whether the transceiver configuration is restored after a power outage and in consequence the channel number is retained. Figure 16 shows the experimental results of RESTOP with the transceiver. When the input voltage drops, the channel number read is 4. Then, inside the snapshotting routine, the energy required to restore the transceiver state is calculated. The obtained value is 8.43 µJ, which is used to calculate V_R = 2.33 V. When the supply voltage rises above V_R, and before the peripheral state is restored, the channel number read is 0, which is the default value. Then, RESTOP restores the peripheral state and the channel number read is 4, which is the same channel as before the power failure.

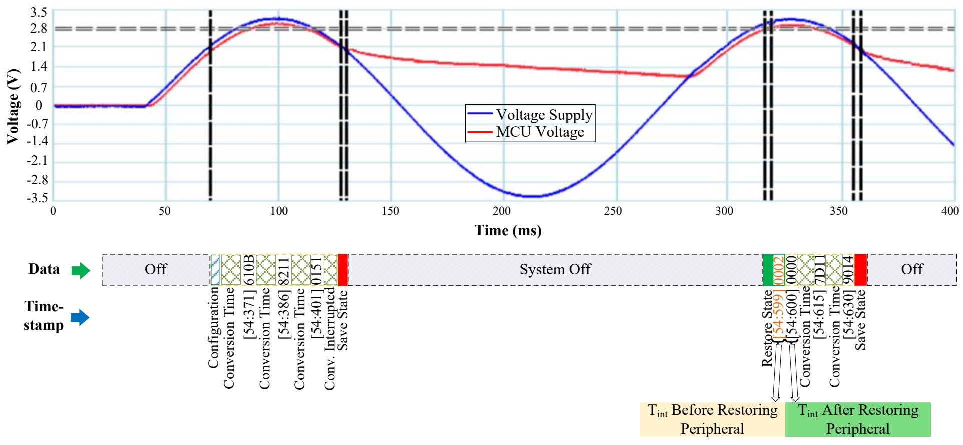


Figure 14. Operation of the luminosity sensor in an intermittently-powered system. After the power failure, the timing registers are configured as before the interruption.

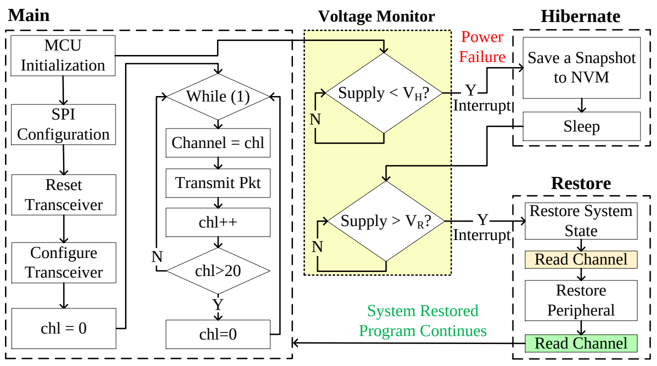


Figure 15. Testing routine to validate RESTOP with the transceiver.

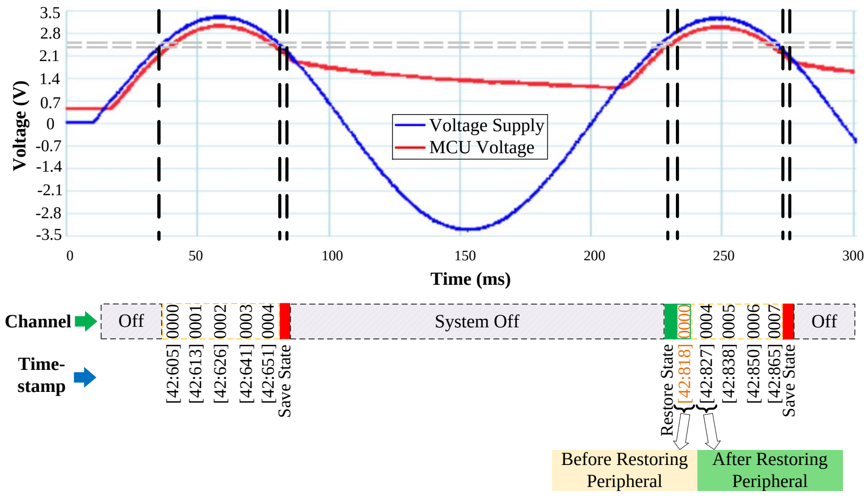


Figure 16. Operation of the transceiver testing routine. After the power failure, the transmission channel number is read before and after RESTOP restores the transceiver state.

5.4. Time Overhead

To analyse the time overhead caused by RESTOP in an intermittently-powered system, we implemented three applications that run under two different scenarios powered by a square wave signal with 3.4 V amplitude and variable duty cycle (from 10 ms to 100 ms). In the first scenario, the peripherals are accessed without using RESTOP (restarting the peripheral's state from scratch after each power failure), while, in the second scenario, our middleware is included. Each application consists of reading data sampled by the luminosity sensor, and reading data from the accelerometer (ACC) and processing it with a Fast Fourier Transform (FFT). Then, the sampled and processed data is transmitted through the radio transceiver. The difference in the applications is the number of samples (32, 64 and 128) that are obtained from the accelerometer and processed by the FFT.

Table 4 shows the time needed to access the peripherals with and without RESTOP. The proposed middleware saves and executes all the *write* instructions to configure the three peripherals and the *read* instructions to get the data from the sensors. The time taken by the FFT is the same in both scenarios because RESTOP is transparent for this task. In the case of the luminosity sensor and the transceiver, they spend the same time in all the applications because they operate only once per case, unlike the accelerometer which takes different amounts of samples. Table 4 also presents the total time spent to complete the FFT, including the time to snapshot and restore both the system state and the peripheral configuration. The last column (at the right side) indicates the time overhead caused by RESTOP on the whole application with different active times. RESTOP causes a time overhead of about 15% when configuring a peripheral. However, this represents a maximum overhead of 0.82% during complete execution of our typical sensing application and is substantially lower than the existing approach *Sytare* [9], which causes a time overhead of up to 137% (30 µs per peripheral instruction) when configuring a radio transceiver. Moreover, the time overhead caused by RESTOP will decrease further as the ratio of the peripheral instructions: normal operation decreases.

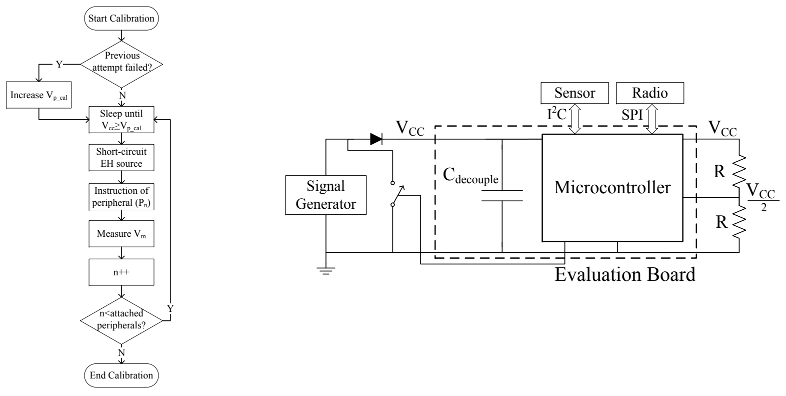
Table 4. Time overhead caused by RESTOP in a system with three external peripherals.

Active time (ms)	No. Sampl.	No. Exec. Inst's	No. S'shot	No. Rest.	Time (ms)									O'head (%)
						Without RESTOP				With RESTOP				
					FFT	Lum.	Acc.	Xcvr.	Total	Lum.	Acc.	Xcvr.	Total	
10	32	80	2	2	19.7	-	0.72	1.05	26.96	-	0.78	1.21	27.18	0.82
10	64	112	6	6	47.3	-	1.43	1.05	66.19	-	1.49	1.21	66.41	0.33
10	128	176	14	14	100	-	2.97	1.05	142.4	-	3.03	1.21	142.6	0.15
40	32	85	0	0	19.7	14.1	0.72	1.05	35.58	14.2	0.78	1.21	35.86	0.79
40	64	117	1	1	47.3	14.1	1.43	1.05	66.59	14.2	1.49	1.21	66.87	0.42
40	128	181	3	3	100	14.1	2.97	1.05	126.3	14.2	3.03	1.21	126.6	0.22
70	32	85	0	0	19.7	14.1	0.72	1.05	35.58	14.2	0.78	1.21	35.86	0.79
70	64	117	0	0	47.3	14.1	1.43	1.05	63.85	14.2	1.49	1.21	64.13	0.44
70	128	181	1	1	100	14.1	2.97	1.05	120.9	14.2	3.03	1.21	121.1	0.23
100	32	85	0	0	19.7	14.1	0.72	1.05	35.58	14.2	0.78	1.21	35.86	0.79
100	64	117	0	0	47.3	14.1	1.43	1.05	63.85	14.2	1.49	1.21	64.13	0.44
100	128	181	1	1	100	14.1	2.97	1.05	120.9	14.2	3.03	1.21	121.1	0.23

6. Conclusions and Future Work

We have proposed RESTOP, a new approach to retain the state of peripherals that communicate with an MCU through a digital interface, in transient computing systems. The presented middleware provides generic functions to read data from the external peripherals or write to them, and keeps track of and saves the transmitted configuration data into the instruction history table from where the peripheral state is restored after a power failure. With these characteristics, RESTOP can be integrated into any of the existing approaches for transient computing and, unlike existing approaches, it is able to operate generically with multiple devices that communicate with the MCU through different protocols such as SPI or I^2C. RESTOP has been validated with a digital accelerometer (SPI), a luminosity sensor (I^2C) and a radio transceiver (SPI) in an intermittently-powered system. Results demonstrate that RESTOP is capable of restoring the peripheral state after power outages causing a time overhead to the application of up to 0.82% during complete execution of our typical sensing application, which is considerably lower than that caused by existing approaches.

In this work, the energy cost of restoring the state of peripherals was modeled, but any additional energy used by the peripherals (e.g., a restored instruction that then triggers a radio transmission) was not taken into account. Hence, in the future we are looking to account for the energy requirements of the complete system, i.e., not just the MCU. A potential solution to this is to implement a calibration routine similar to that used in Hibernus++ [7], but in this case for measuring the energy consumed when executing each peripheral instruction. As shown in Figure 17a, the calibration routine would wait for the supply voltage to reach the calibration voltage (V_{p_cal}). When this voltage is reached, the EH source would be short-circuited by closing the switch in Figure 17b, and an instruction is issued to the peripheral. The drop in supply voltage caused by issuing and executing the instruction is given by V_{p_cal} - V_m, where V_m is the voltage measured after the instruction has been completed. This process would be executed once per each attached peripheral.



(**a**) Calibration Algorithm. (**b**) Calibration Circuit.

Figure 17. Calibration routine to measure the energy consumed by external peripherals when executing an instruction, showing: (**a**) the algorithm; and (**b**) the circuit schematic.

Another package of work is to integrate RESTOP with a generic operating system, e.g., the ARM mbed OS which has already been demonstrated with Hibernus [22]. Operating systems have components and abstractions for peripheral interface, which could be combined with RESTOP. This would further increase the accessibility of transient computing systems and promote standardization.

Acknowledgments: This work was supported in part by the Mexican CONACYT. It was also supported by the UK EPSRC under EP/P010164/1. RESTOP source code can be downloaded from http://www.transient.ecs.soton.ac.uk. Experimental data used in this paper can be found at DOI:10.5258/SOTON/D0373 (http://doi.org/10.5258/SOTON/D0373).

Author Contributions: All authors collaborated in the design of the proposed approach and conceived the experiments. A.R.A. implemented and experimentally validated the approach and wrote the manuscript. D.B., G.V.M. and A.S.W. proofread and edited the manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Beeby, S.; White, N. *Energy Harvesting for Autonomous Systems*; Artech House: Norwood, MA, USA, 2014.
2. Escolar, S.; Chessa, S.; Carretero, J. Energy-neutral networked wireless sensors. *Simul. Model. Pract. Theory* **2014**, *43*, 1–15.

3. Ransford, B.; Sorber, J.; Fu, K. Mementos: System support for long-running computation on RFID-scale devices. *ACM SIGPLAN Notices* **2012**, *47*, 159–170.
4. Jayakumar, H.; Raha, A.; Raghunathan, V. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In Proceedings of the 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, Mumbai, India, 5–9 January 2014.
5. Lucia, B.; Ransford, B. A simpler, safer programming and execution model for intermittent systems. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015; ACM Press: Portland, OR, USA, 2015; pp. 575–585.
6. Balsamo, D.; Weddell, A.S.; Merrett, G.V.; Al-Hashimi, B.M.; Brunelli, D.; Benini, L. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Syst. Lett.* **2015**, *7*, 15–18.
7. Balsamo, D.; Weddell, A.S.; Das, A.; Arreola, A.R.; Brunelli, D.; Al-Hashimi, B.M.; Merrett, G.V.; Benini, L. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Trans. Comput. Des. Integr. Circuits Syst.* **2016**, *35*, 1968–1980.
8. Liu, Y.; Xie, Y.; Shu, J.; Yang, H.; Li, Z.; Li, H.; Wang, Y.; Li, X.; Ma, K.; Li, S.; Chang, M.F.; John, S. Ambient energy harvesting nonvolatile processors. In Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, 7–11 June 2015; ACM Press: New York, NY, USA, 2015; pp. 1–6.
9. Berthou, G.; Delizy, T.; Marquet, K.; Risset, T.; Salagnac, G. Peripheral state persistence for transiently-powered systems. In Proceedings of the 2017 Global Internet of Things Summit (GIoTS), Geneva, Switzerland, 6–9 June 2017; pp. 1–6.
10. Bhatti, N.A.; Mottola, L. HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '17), Pittsburgh, PA, USA, 18–21 April 2017; ACM Press: New York, NY, USA, 2017.
11. Li, Z.; Liu, Y.; Zhang, D.; Xue, C.J.; Wang, Z.; Shi, X.; Sun, W.; Shu, J.; Yang, H. HW/SW co-design of nonvolatile IO system in energy harvesting sensor nodes for optimal data acquisition. In Proceedings of the 53rd Annual Design Automation Conference (DAC '16), Austin, TX, USA, 5–9 June 2016; ACM Press: Austin, TX, USA, 2016; pp. 1–6.
12. Sample, A.P.; Yeager, D.J.; Powledge, P.S.; Mamishev, A.V.; Smith, J.R. Design of an RFID-based battery-free programmable sensing platform. *IEEE Trans. Instrum. Meas.* **2008**, *57*, 2608–2615.
13. Naderiparizi, S.; Parks, A.N.; Kapetanovic, Z.; Ransford, B.; Smith, J.R. WISPCam: A Battery-Free RFID Camera. In Proceedings of the IEEE International Conference on RFID, Tokyo, Japan, 16–18 September 2015; pp. 166–173.
14. Hester, J.; Sitanayah, L.; Sorber, J. Tragedy of the Coulombs. In Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, Seoul, Korea, 1–4 November 2015; ACM Press: Seoul, Korea, 2015; pp. 5–16.
15. Texas Advanced Optoelectronic Solutions. TSL2560 Datasheet, 2005. Available online: http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/LightImaging/TSL2561.pdf (accessed on 11 July 2017).
16. Texas Instruments. RF CC1101 Datasheet, 2013. Available online: http://www.ti.com/lit/ds/symlink/cc1101.pdf (accessed on 15 January 2017).
17. Analog Devices. ADXL362 Datasheet, 2012. Available online: http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf (accessed on 4 January 2017).
18. Johnstone, M.S.; Wilson, P.R. The Memory Fragmentation Problem: Solved? In Proceedings of the 1st International Symposium on Memory Management (ISMM '98), Vancouver, BC, Canada, 17–19 October 1998; ACM: New York, NY, USA, 1998.
19. Schwalb, D.; Berning, T.; Faust, M.; Dreseler, M.; Plattner, H. nvm malloc: Memory Allocation for NVRAM. Available online: https://www.semanticscholar.org/paper/nvm-malloc-Memory-Allocation-for-NVRAM-Schwalb-Berning/1d00b3a1030a653b22cd40659ca1ad59cba5aa5f (accesed on 10 January 2018).
20. Rodriguez Arreola, A.; Balsamo, D.; Das, A.K.; Weddell, A.S.; Brunelli, D.; Al-Hashimi, B.M.; Merrett, G.V. Approaches to Transient Computing for Energy Harvesting Systems. In Proceedings of the 3rd International Workshop on Energy Harvesting & Energy Neutral Sensing Systems, Seoul, Korea, 1–4 November 2015; ACM Press: Seoul, Korea, 2015.

21. Texas Instruments. MSP430FR5739 Datasheet, 2011. Available online: http://www.ti.com/lit/ds/symlink/msp430fr5739.pdf (accessed on 4 December 2016).
22. Balsamo, D.; Elboreini, A.; Al-Hashimi, B.M.; Merrett, G.V. Exploring ARM mbed support for transient computing in energy harvesting IoT systems. In Proceedings of the 2017 7th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI), Vieste, Italy, 15–16 June 2017; pp. 115–120.

Intermittently-Powered Energy Harvesting Step Counter for Fitness Tracking

Alberto Rodriguez[†], Domenico Balsamo[†], Zhenhua Luo[‡], Steve P. Beeby[†], Geoff V. Merrett[†], Alex S. Weddell[†]
[†]Department of Electronics and Computer Science, University of Southampton
[‡]School of Water, Energy and Environment, Cranfield University
Email: [†]{ara1g13,db2a12,spb,gvm,asw}@ecs.soton.ac.uk, [‡]z.luo@cranfield.ac.uk

***Abstract*—Over the past decade, there has been a rapid increase in the popularity of wearable and portable devices, such as step counters, to monitor fitness performance. However, these devices are battery-powered, meaning that their lifetimes are restricted by battery capacity. Ideally, wearable devices could be powered by energy harvested from human motion. Energy harvesting systems traditionally incorporate energy storage to cope with source variability. However, energy storage takes time to charge and increases the size and cost of systems. This paper proposes an intermittently-powered energy harvesting step counter for integrated wearable applications, which aims to remove the energy storage element. The proposed step counter sustains its operation by harvesting energy from footsteps using a ferroelectret insole, which also works as an event detection sensor, i.e. the system is powered by the parameter that is being sensed. Designing this required the characterization of the insole to evaluate the amount of energy provided, and analysis of the energy needed by the overall system. Finally, the system was implemented and experimentally validated. The proposed step counter has an error of less than 4% when walking, which is lower than the error in conventional smartphone applications.**

***Index Terms*—Step Counter, Energy Harvesting, Event Sensor, Ferroelectret Insole, Intermittent Source.**

I. Introduction

Typical systems to monitor fitness metrics such as number of steps, distance walked, calorie consumption, etc. are battery-powered, meaning that their lifetime becomes restricted by battery capacity and discharge rate [1], [2]. For example, smart watches for fitness performance tracking typically last for 18-20 hours, depending on the model and activity [3], while fitness trackers can last for 3-4 days [4], having an average charge time of 2 hours.

Motivated by the limited lifespan achievable with battery-powered systems, research has recently looked at replacing batteries with energy harvesting solutions. For wearable systems, the energy harvested from human motion (e.g. footsteps) is attractive, potentially avoiding periodic battery replacement or charging [5]. However, the power obtained from human motion is typically in the range of mW, intermittent and of short duration [6], [7]. To sustain computation and mitigate variability, energy harvesting systems normally integrate energy storage units, which require time to charge and increase their volume, mass and cost.

This paper proposes an intermittently-powered energy harvesting step counter for integrated wearable applications. In contrast to conventional energy harvesting systems, the proposed step counter removes the energy storage element and operates directly from the harvesting source (Figure 1). It sustains its operation by harvesting energy from footsteps, using a ferroelectret insole which generates electricity under mechanical stress [8]. The system uses a microcontroller (MCU), with a low-power non-volatile memory, which wakes up at each step and retains data during the power outage between steps. The ferroelectret insole also acts as an event detection sensor: the system is only powered when the event (i.e. a footstep) is detected.

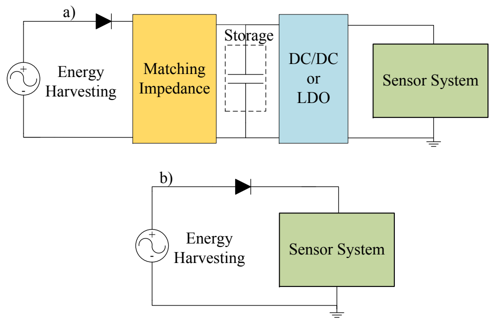


Fig. 1. Energy harvesting systems. a) Conventional system with energy storage. b) Storage-less energy harvesting system.

The novel contributions of this work are:

- Characterization of the dynamics of human powered energy harvesting, with a ferroelectret insole (Section II).
- Analysis of the energy requirements and a methodology for designing the step counter (Section III).
- Design and practical testing of a self-powered step counter. (Section IV).

Experiments demonstrate that the proposed step counter has an error in counting steps of less than 4% when walking, significantly better than existing smartphone applications.

II. Ferroelectret Insole Characterization

In this section, we look at the performance of the ferroelectret insole when mounted in a shoe. We also calculate the energy generated per step, which aids the design of the system in Section III.

Ferroelectret materials are flexible cellular polymer foams that, similarly to a piezoelectric source, convert mechanical

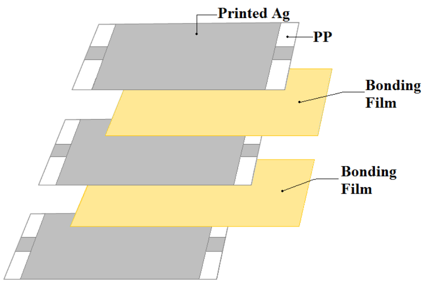


Fig. 2. Structure of a multilayer ferroelectret insole.

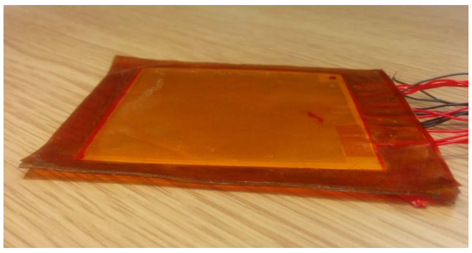


Fig. 3. 30-layer ferroelectret insole used in our design.

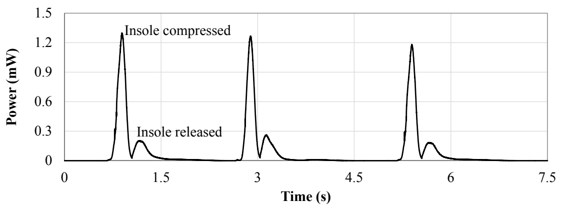


Fig. 4. Rectified power signal generated by the ferroelectret insole with a high-impedance workload after three steps.

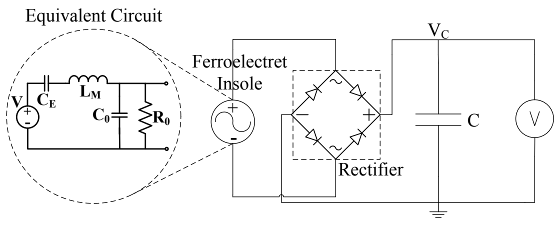


Fig. 5. Characterization circuit including the full-wave rectifier.

energy into electricity when they are bent or compressed [9]. In an energy harvesting insole made with ferroelectret materials, the mechanical stress is generated by the heel striking against the floor when walking or running. Figure 2 shows the model of a three-layer insole of polypropylene ferroelectret (PP) connected in parallel with bonding films between layers [10]. A similar insole with 30 layers has been used in our design (Figure 3).

As shown in Figure 4, these harvesting sources generate high-power energy bursts for a short period of time (in the order of *ms*). The magnitude of these power bursts depends on the momentum of applied compression and the impedance of the load that is connected to the source. The greater the momentum, the greater the voltage but the shorter the duration of the power burst. The ferroelectret insole generates two voltage peaks at each step: positive when compressed and negative when released.

Although this work aims to negate the need for large energy buffers, some capacitance is inherently required by microcontrollers to decouple the energy source (and possible electrical noise) from the system. This is known as decoupling capacitance. In intermittently-powered systems [11], [12], this capacitance may also be exploited as a small energy store. In order to evaluate the performance of the ferroelectret insole when connected to a microcontroller, we use a range of capacitance values. The values chosen ranged between the minimum decoupling capacitance recommended by manufacturer for a typical MCU (i.e. 4.7μF), and the value incorporated in a reference design for that device (i.e. 16μF) [13].

Figure 5 shows the characterization circuit with a purely capacitive load. The circuit includes a bridge-rectifier connected at the insole's output. The ferroelectret insole is represented by a basic equivalent circuit for piezoelectric transducers [14]. The capacitance C_E is the inverse of the mechanical elasticity of the insole. L_M represents the seismic mass of the transducer. The capacitor C_0 is the static capacitance of the harvesting source and R_0 is the resistance of the dielectric material that forms the static capacitance (insulation resistance), whose ideal value should be over $10^{12}\Omega$.

The voltage (V_C) is measured across the capacitor, and the electrical energy generated at each step is calculated. The experiment was performed by a 70-kg person moving at two different step rates (referred to herein as operating modes): walking (approximately 70 steps per minute) and running (approximately 120 steps per minute). This test was executed five times for each capacitor in each mode and the results were averaged.

Figure 6 shows the voltage measured with each capacitor after each step in running and walking modes. Here, V_{min} is the minimum operating voltage of the typical low-power MCU. This allows us to observe when an MCU would became active. In running mode, the force applied to the insole is higher than in walking mode, and produces a higher voltage increment at each step. This introduces uncertainty related to the number of steps walked before the system becomes active. This uncertainty increases with larger values of decoupling capacitance. For example, with the lowest capacitance, V_{min} is reached after 2 or 3 steps in both walking modes, while with higher capacitances the voltage threshold is not reached even after 5 steps.

The energy generated per footstep can also be calculated from this test by using the following equation:

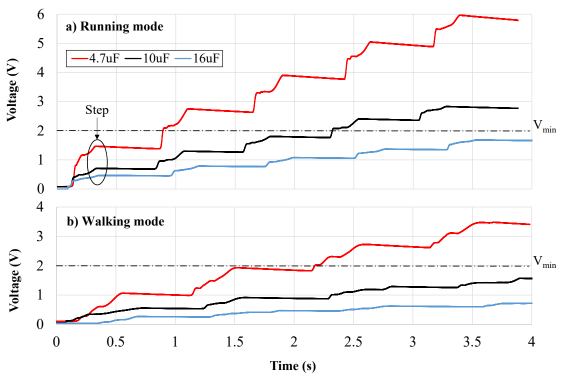


Fig. 6. Ferroelectret insole characterization with three different values of capacitances in walking and running mode.

TABLE I
MEAN ENERGY OBTAINED PER SINGLE STEP

Capacitance (μF)	Walking mode (μJ)				Running mode (μJ)			
	1st. Step	2nd. Step	3rd. Step	Total	1st. Step	2nd. Step	3rd. Step	Total
4.7	2.35	7.05	9.21	18.61	5.30	13.14	18.89	37.33
10	0.85	1.27	1.41	3.53	3.21	2.47	4.66	10.34
16	0.21	0.37	0.73	1.31	0.59	0.92	1.33	2.84

$$E_{capacitor} = C\frac{V_2^2 - V_1^2}{2} \tag{1}$$

where C is the decoupling capacitance, V_1 is the voltage in the capacitor before the step and V_2 is the voltage reached after the footstep is taken.

Table I shows the average energy that the ferroelectret insole charges into each capacitor in each mode. The maximum amount of energy is obtained with the lowest capacitance, being up to fourteen times higher. Therefore, with the capacitance of 4.7μF the insole gives more energy per footstep and allows the MCU to become active more quickly.

III. Overall System Design

The insole characterization performed in Section II did not consider the effect of an active load, which causes a fast discharge of the decoupling capacitance and a consequent voltage drop across the MCU. Describing this requires the knowledge of the overall system, the energy requirements when operating in different modes (i.e. start-up, active and low-power) and the usage of the energy buffered during these modes to efficiently complete a task (i.e. count a step).

A. System Architecture

Figure 7 represents a simplified example of the system architecture, where the ferroelectret insole output is rectified and used to power the sensor system, which is composed of an evaluation MSP-EXP430FR5739 test board [13]. This board contains an MCU with a non-volatile Ferroelectric RAM (FRAM) memory and a 4.7μF decoupling capacitance C from

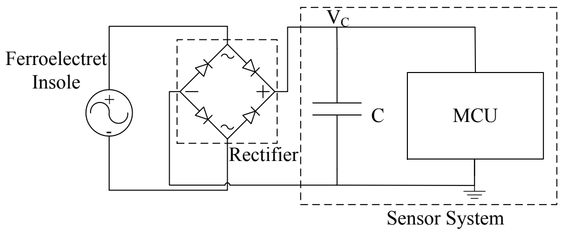


Fig. 7. Simplified system architecture of the step counter with ferroelectret insole.

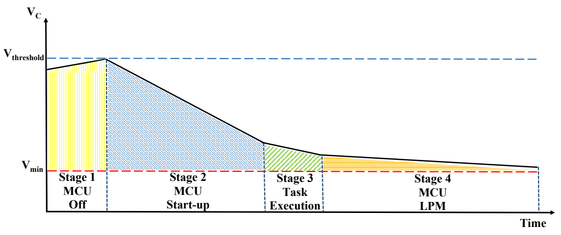


Fig. 8. Stages in which is divided the process of counting a step.

the insole characterization (see Section II). This board also integrates a 3-axis accelerometer (acc) that could potentially be used to detect a footstep (e.g. by detecting movements which are then classified as a step). However, the energy overhead due to active sensors can limit their use in the case of resource-constrained systems. In the next subsection, we analyse the energy requirements of the overall system, by first considering the accelerometer as an option for step counting, along with a sensor-less solution that is more power efficient.

B. System Energy Requirements with Sensor

Figure 8 shows the process of counting a step that is divided into four stages:

- **Stage 1**. MCU off due to a power outage;
- **Stage 2**. MCU Start-up when power is available (i.e. when a pre-defined threshold V_{th} is reached);
- **Stage 3**. Counting a step (i.e. MCU configuration, accelerometer initialization, step detection and data retention in the FRAM);
- **Stage 4**. MCU in low power mode (LPM), after counting a step.

In order to quantify the energy required during these stages and set the right value of V_{th}, the MCU has been characterized using a constant voltage (i.e. 2.4V) and by using the following algorithm:

1) The system is initially in off mode (stage 1).
2) The system is powered and, after the start-up, the MCU is configured (i.e. frequency and internal peripheral setting) and the on-board accelerometer is initialized.
3) A step is validated (accelerometer sampling plus data processing).

TABLE II
PARAMETER VALUES OF THE SYSTEM IN EACH STAGE

Stage	Tasks	Current (µA)	Time (ms)
2	MCU Start-up	1000	0.9
3	MCU+acc	500	13.5
3	Validate a step	300	3.6

4) It is retained the state in FRAM and enters in LPM.

Table II shows the current and time needed to successfully complete these stages. In particular, the MCU start-up current is much higher than the other stages, while the accelerometer requires a very long time to be set up.

The values in Table II will be used in the following analysis to evaluate the optimum value for V_{th} which can be defined as:

$$V_{th} = V_{min} + \frac{1}{C}\int_0^t I(\tau)d(\tau) \tag{2}$$

where V_{min} is the minimum operating voltage of the MCU (1.8V), C is the decoupling capacitance and I is the current consumption. From Table II, Stage 3 has two different subtasks that require different current values. The highest current value refers to MCU configuration and accelerometer initialization, while the second value refers to the current needed to validate a step. Thus, the Equation 2 can be presented as follows:

$$V_{th} = V_{min} + \frac{1}{C}(I_{su}t_{su} + I_{acc}t_{acc} + I_{vs}t_{vs}) \tag{3}$$

where I_{su} and t_{su} are the current and the time needed for the start-up (Stage 2), I_{acc} and t_{acc} are the current an time for setting the accelerometer (Stage 3), I_{vs} and t_{vs} are the current and time taken by the MCU to validate a step. Combining the values in Table II and the Equation 3, we obtain a value of V_{th} equal to 4.01V, that is higher than the maximum operating voltage for the MCU (3.6V).

For the reason underlined above (i.e. high energy overhead), it is not feasible to use the accelerometer in an intermittently-powered and resource-constrained system. In the next subsection, we present a solution that allows us to build a more energy efficient step counter, without using an active sensor but which delivers a higher accuracy.

C. Sensor-less System

An alternative way to implement a step counter without active sensors is to use the energy harvesting source as an event detection sensor, where the availability of energy is an indicator for a footstep. This approach would reduce the energy required to validate a step, removing the overhead for the accelerometer setting and data processing. Moreover, this will possibly increase the accuracy of the system discarding 'false' steps.

Thus, the MCU needs to be re-characterized by using the following updated algorithm: the system is powered and, after start-up, the MCU is configured. It then validates a step, retains the state in FRAM and enters in LPM. Table III shows the values for current and time with this algorithm.

TABLE III
PARAMETER VALUES OF THE SYSTEM IN EACH STAGE WITHOUT ACCELEROMETER

Stage	Tasks	Current (µA)	Time (ms)
2	MCU Start-up	1000	0.9
3	Validate a step	300	0.14

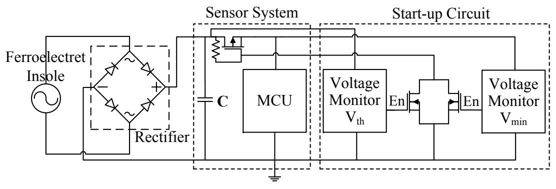


Fig. 9. Intermittently-powered Step Counter circuit diagram.

The time for counting a step is much smaller than the time needed with the accelerometer. Thus, Equation 3 can be now updated by removing the current and time overhead of the accelerometer:

$$V_{th} = V_{min} + \frac{1}{C}(I_{su}t_{su} + I_{vs}t^*_{vs}) \tag{4}$$

where t^*_{vs} is the actual time needed for counting a step. Combining the values from Table III and Equation 4, we obtain a V_{th} equal to 2.2V. From Equation 1, the energy required to charge the decoupling capacitance to that voltage is 11.37μJ. As shown in Table I, the insole is able to charge up to 18.61μJ in walking mode, after 3 steps. Therefore, it is possible to implement an event detection sensor based on the energy harvesting source and using the 4.7μF decoupling capacitance.

Figure 9 shows the final design including the ferroelectret insole, the rectifier, the sensor system and an additional start-up circuit, which minimizes the quiescent current due to the MCU when its supply voltage is below V_{min}. This circuit guarantees a reliable start by detecting the input voltage and only turning on the supply to the MCU when its input is above V_{th} and switching it off when the voltage drops below V_{min}. This is enabled by two voltage monitors, which are configured in a MOSFET latch arrangement.

IV. FUNCTIONAL VALIDATION AND COMPARATIVE EVALUATION

The presented step counter has been implemented and experimentally validated. We compared the accuracy of our solution against two smartphone applications. To verify system operation, two GPIO pins of the test board were configured to indicate when the system counts and enters the LPM, respectively. Both signals were monitored by a PicoScope at the same time as the voltage across the capacitor and the MCU.

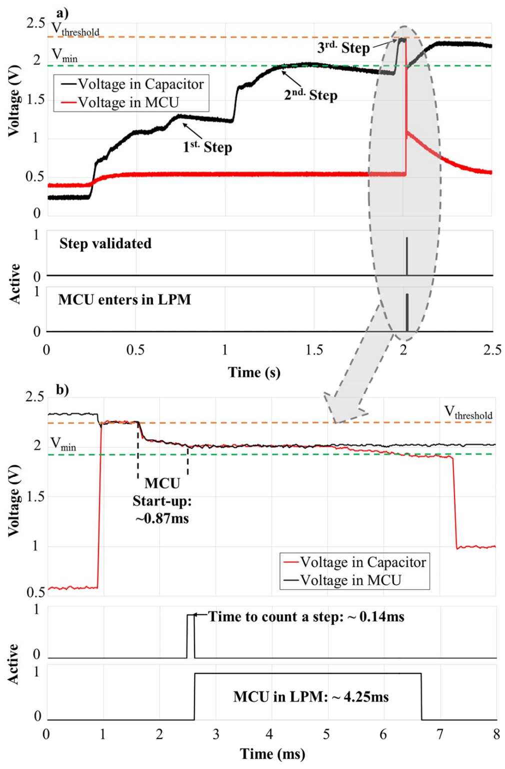


Fig. 10. Operation of the intermittently-powered step counter. a) After three steps the system starts working. b) Detailed description of the process of counting a step.

That information was logged and later plotted. Thus, it was possible to visualize how many steps were taken before the system started working, to detect possible errors. The number of steps is saved in a FRAM variable in order to retain the value between power failures.

As shown in Figure 10a, the system is active when the voltage reaches V_{th}. The sensor node takes into account the initial three steps needed to reach this voltage. Then, the MCU is configured and the step is counted. Once the task is executed, the system enters low power mode before having a power outage. Figure 10b shows a detailed snapshot of the process of counting a step from Figure 10a, including the time needed by the system to start up, configure the MCU and increment the counter. As mentioned in Table III, configuring the MCU and counting a step takes approximately 140μs. The MCU remains in LPM until the voltage drops below V_{min}.

Figure 11 shows the system counting three consecutive steps. In walking mode, the average time between two steps is approximately 0.9s (worst case). During this interval, the ferroelectret insole does not provide any further energy and the voltage drop, due to the start-up circuit and leakage current, is approximately 0.1V. This means that the voltage across the

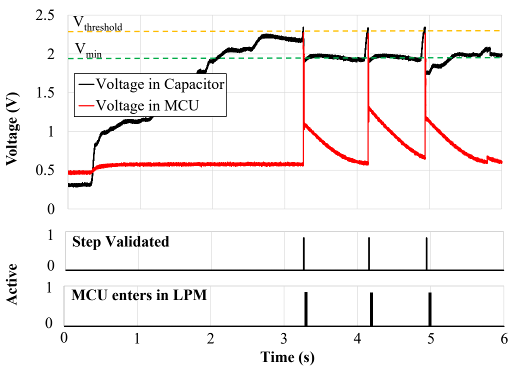


Fig. 11. System counting three consecutive steps.

decoupling capacitance is about 1.7V, before the next step is taken. Considering that each step in walking mode increases the voltage by 0.8V, the sensor system is able, after the first three steps, to count each subsequent step.

In order to evaluate the performance of our design, we compared its accuracy against two existing Android® smartphone applications: Pacer [15] and WalkMate [16]. A total of 400 steps were taken per step counter solution (1200 steps in total): 20 steps for 10 attempts in each operating mode. These experiments were performed by a 70-kg person with the ferroelectret insole attached to the shoe and the smartphone placed at his waist. Figure 12 shows the experimental results of our step counter and the two smartphone applications, at each attempt in walking and running mode. The dotted line (in green) represents the real number of steps.

The proposed step counter has an average error of 3.5% in walking mode, while in running mode the average error is 1%. The maximum error in a single attempt is 2 steps in walking mode (attempts 4 and 9) and 1 step in running mode (attempts 4 and 7). In the case of the first smartphone application (Pacer), it has an average error of 25.5% and 12.5% in walking and running modes, respectively. The maximum error in a single attempt is 8 steps (attempt 5) when walking and 4 steps (attempt 4) when running. The second application (WalkMate) shows the worst performance in walking mode, having an error rate of 41.5% with a maximum error of 10 steps in attempts 5, 6 and 10. However, in running mode, this application has a better performance, with an error rate of 4% and a maximum error of 2 steps in attempts 2 and 5.

Another test was executed, incrementing the number of steps to 50. Table IV shows the results obtained for each step counter in walking and running mode. Also in this case, our step counter has the lowest error rate in walking mode and it does not present errors in running mode. The WalkMate application has the highest error (44%) in walking mode, while Pacer had maximum error in running mode (4%).

The marginal error in the proposed step counter is due to the adopted solution for compensating the initial number of steps

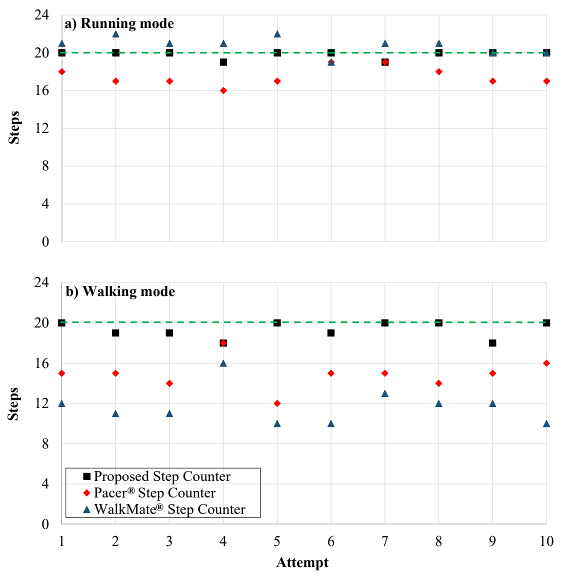


Fig. 12. Performance comparison between the intermittently-powered step counter and two smartphone applications in a) running mode and b) walking mode.

(i.e. when the system is still not active). In a small number of cases, after three steps the voltage across the decoupling capacitance does not reach V_{th}. This mainly happens in walking mode because of the lower energy generated per step. In these cases, the required threshold can be reached after 4 or 5 steps. Despite this, the error rate is less than 4% in the worst case, which is lower than that of the battery-powered systems.

V. Conclusion

In this paper, an intermittently-powered energy harvesting step counter for fitness tracking has been proposed. The presented step-counter aims to remove the energy storage element and self-sustains its operation by harvesting energy from footsteps. To achieve this, it uses a ferroelectret insole, which also acts as an event detection sensor. We have characterized the insole to evaluate the amount of energy provided, and analysed of the energy needed by the overall system. The system has been then implemented and experimentally validated. The results show an error of less than 4% when walking, which is lower than the error in conventional smartphone applications.

Acknowledgment

This work was supported in part by the Mexican CONACYT. It was also supported by the UK Engineering and Physical Sciences Research Council (EPSRC) Grant EP/L000563/1 (GRACEFUL) and Grant EP/K031910/1 (SPHERE IRC). Experimental data used in this paper can be found at DOI:10.5258/SOTON/405149 (http://doi.org/10.5258/SOTON/405149).

TABLE IV
Performance of the three step counters tested with 50 steps

System	N° Steps Walking	Error Rate (%)	N° Steps Running	Error Rate (%)
Proposed Step Counter	49	2	50	0
Pacer Step Counter	40	20	48	4
WalkMate Step Counter	22	44	51	2

References

[1] J. Williamson, Q. Liu, F. Lu, W. Mohrman, K. Li, R. Dick, and L. Shang, "Data sensing and analysis: Challenges for wearables," in *The 20th Asia and South Pacific Design Automation Conference*, Jan 2015, pp. 136–141.

[2] D. Ferreira, A. K. Dey, and V. Kostakos, *Understanding Human-Smartphone Concerns: A Study of Battery Life*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 19–33. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21726-5_2

[3] "Apple Watch. General Battery Information," [Online Document], 2016. Available: http://www.apple.com/watch/battery.html [Accessed: Nov 2016].

[4] "FitBit life expectancy?" [Online Document], 2016. Available: https://www.fitbit.com/ [Accessed: Nov 2016].

[5] J. A. Paradiso and T. Starner, "Energy scavenging for mobile and wireless electronics," *IEEE Pervasive Computing*, vol. 4, no. 1, pp. 18–27, Jan 2005.

[6] H. Huang, G. V. Merrett, and N. M. White, "Human-powered inertial energy harvesters: the effect of orientation, location and activity on obtainable power," *Procedia Engineering*, vol. 25, pp. 815 – 818, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877705811058693

[7] P. Mitcheson, E. Yeatman, G. Rao, A. Holmes, and T. Green, "Energy Harvesting From Human and Machine Motion for Wireless Electronic Devices," *Proceedings of the IEEE*, vol. 96, no. 9, pp. 1457–1486, sep 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4618735

[8] Z. Luo, D. Zhu, J. Shi, S. Beeby, C. Zhang, P. Proynov, and B. Stark, "Energy harvesting study on single and multilayer ferroelectret foams under compressive force," *IEEE Transactions on Dielectrics and Electrical Insulation*, vol. 22, no. 3, pp. 1360–1368, Jun 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7116323

[9] J. Hillenbrand and G. M. Sessler, "Piezoelectricity in cellular electret films," *IEEE Transactions on Dielectrics and Electrical Insulation*, vol. 7, no. 4, pp. 537–542, Aug 2000.

[10] Z. Luo, D. Zhu, and S. P. Beeby, "Multilayer ferroelectret-based energy harvesting insole," *Journal of Physics: Conference Series*, vol. 660, p. 6, Dec 2015. [Online]. Available: http://stacks.iop.org/1742-6596/660/i=1/a=012118?key=crossref

[11] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-hashimi, D. Brunelli, and L. Benini, "Hibernus : Sustaining Computation during Intermittent Supply for Energy-Harvesting Systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 15 – 18, mar 2015.

[12] A. Rodriguez Arreola, D. Balsamo, A. K. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, and G. V. Merrett, "Approaches to Transient Computing for Energy Harvesting Systems," in *Proceedings of the 3rd International Workshop on Energy Harvesting & Energy Neutral Sensing Systems - ENSsys '15*, Seoul, Korea, 2015, pp. 3–8.

[13] "Texas Instruments MSP430FR5739 Datasheet," [Online Document], 2011. Available: http://www.ti.com/lit/ds/symlink/msp430fr5739.pdf.

[14] A. Vazquez, "Novel Piezolectric Transducers for High Voltage Measurements," Ph.D. dissertation, Universitat Politecnica de Catalunya, 2000. [Online]. Available: https://spqr.eecs.umich.edu/papers/ransford-thesis.pdf

[15] "Pacer: Pedometer and Weight Trainer," [Smartphone Application], 2016. Available: http://www.pacer.cc/ [Accessed: Nov 2016].

[16] "Sony WalkMate," [Smartphone Application], 2016. Available: http://www.sonymobile.com/ [Accessed: Nov 2016].

Approaches to Transient Computing for Energy Harvesting Systems: A Quantitative Evaluation

Alberto Rodriguez[†], Domenico Balsamo[†], Anup Das[†],
Alex S. Weddell[†], Davide Brunelli[‡], Bashir M. Al-Hashimi[†], Geoff V. Merrett[†]
[†]Department of ECS, University of Southampton
[‡]Department of Electronics, University of Trento
[†]{ara1g13, db2a12, a.k.das, asw, bmah, gvm}@ecs.soton.ac.uk,
[‡]davide.brunelli@unitn.it

ABSTRACT

Systems operating from harvested sources typically integrate batteries or supercapacitors to smooth out rapid changes in harvester output. However, such energy storage devices require time for charging and increase the size, mass and cost of the system. A recent approach to address this is to power systems directly from the harvester output, termed transient computing. To solve the problem of having to restart computation from the start due to power-cycles, a number of techniques have been proposed to deal with transient power sources. In this paper, we quantitatively evaluate three state-of-the-art approaches on a Texas Instruments MSP430 microcontroller characterizing the application scenarios where each performs best. Finally, recommendations are provided to system designers for selecting the most suitable approach.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Transient Computing, Energy Harvesting, Wind Turbines, Photo Voltaic Cells

Keywords

Checkpoint, Hibernus, IoT, Mementos, QuickRecall

1. INTRODUCTION

The Internet-of-Things (IoT) is the interconnection of billions of things. Each IoT device could be considered as an ultra-low power and resource-constrained sensor elaboration platform. Power management of these devices is emerging as a primary challenge for system designers as they typically have to last for few years without intervention to charge or replace batteries. Energy harvesting (EH) is an efficient solution to power sensor nodes present in these connected devices. EH sources harvest electric power from ambient sources or human motion including light, vibration, or temperature differences [2, 8, 11]. This harvested energy is converted into a DC signal (voltage and current) that is used to power up sensor devices. EH power sources are typically intermittent due to temporal variation in the environmental parameter (e.g., time of day, weather condition and available light). 1 shows an example of the harvested power from a micro-wind turbine and a photovoltaic module. The voltage of the micro-wind turbine varies between -4V to 4V. The frequency of the power-cycle is dependent on the wind velocity. Similarly, the output from the photovoltaic cell, used in this example, changes depending on the intensity of the light source. An application executed on the sensor node powered by these EH sources can potentially be interrupted depending on the harvested power availability. To overcome this limitation, sensor nodes typically integrate energy storage in the form of supercapacitors to buffer energy in order to sustain computation at times of power unavailability.

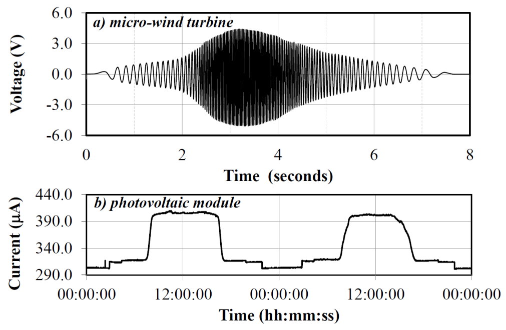


Figure 1: Harvester output from (a) micro-wind turbine and (b) photovoltaic module

Energy storage devices require time to power on and increases system size, mass and cost. An alternative solution is to power a system directly using the harvested source, eliminating the need of an energy storage device. However, to sustain computation with transient sources, the common approach is to checkpoint the system state to save it into a non-

ENSsys'15, November 1, 2015, Seoul, South Korea.
© 2015 ACM. ISBN 978-1-4503-3837-0/15/11 ...$15.00.
DOI: http://dx.doi.org/10.1145/2820645.2820652.

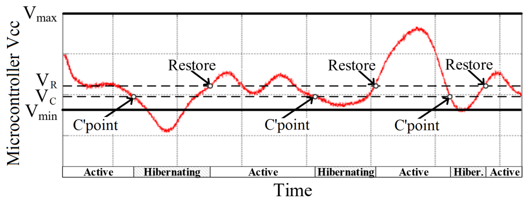


Figure 2: Checkpointing and restore to sustain operation with transient energy harvester outputs.

volatile memory. Later when the supply is restored, the last checkpoint saved before power loss is restored and the operation is continued from the point where it was halted. Three prominent techniques – *Mementos* [10], *QuickRecall* [5] and *Hibernus* [1] have been proposed based on the checkpointing concept, differing in the checkpointing approach leading to different timing/power overheads. This work studies these three approaches and evaluates them theoretically and experimentally on a common platform – the Texas Instruments MSP430FR microcontroller. The objective is to identify scenarios where one approach outperforms others.

The remainder of this paper is organized as follows. A brief background on the related works is provided in Section 2. This is followed by a description of the three approaches in Section 3. The quantitative evaluation of these approaches is provided in Section 4. Finally, the paper is concluded in Section 5 with recommendations for system designers.

2. RELATED WORKS

A new paradigm, which addresses the presented challenges, is of 'transiently-powered computing' [7] allowing systems to operate reliably from intermittent or limited sources such as energy harvesting. This borrows from the concept of checkpointing, which has been used in large-scale computing for decades to provide robustness against errors or hardware failure [3]. This technique involves systematically saving data to non-volatile memory (NVM). To recover from a failure, systems roll back to the previous valid checkpoint, before continuing operation. State-of-art embedded systems use a variety of classic and advanced NVM structures to save their state. Examples of memories used for state retention are Flash or battery-backed SRAM memories [6]. However, a drawback of checkpointing is that it is impossible to predict the exact time of failures, so computation time and energy will be wasted by (1) taking unnecessary checkpoints, and (2) rolling back by the period between the checkpoint and failure. Attempts have been made to address these problems, for example by assuming different failure distributions. Moreover, system shut-down and wake-up have significant time and energy cost, and they must be minimized.

Recently, the checkpointing concept has been applied to embedded devices with unstable power supplies, to avoid power-cycling causing the loss of data and to enable long-running computations across several power cycles. This is enabled by systems saving their state so that, when their power supply fails, they can resume operation when it recovers. Figure 2 shows the output voltage of an energy harvester, which is used to power a microcontroller. Whenever, the supply voltage crosses the checkpoint threshold V_c, a checkpoint is stored in the NVM. On the other hand, whenever the supply voltage crosses the restore threshold V_R, a checkpoint is restored. As shown in Figure 2, this allows computation to continue across several power-cycles, which would conventionally have caused a system to reset repeatedly. A few recently published papers show that the time and energy cost of distributed state-retentive logic elements can be lowered by orders of magnitude with respect to traditional Flash-based approaches using alternative non-volatile memory technology, such as FRAM [4]. Some more advanced technologies are currently under development, such as ReRAM [9], which could further reduce NVM storage energy and cost.

Prominent works in this area include (1) Mementos [10], which uses checkpoints placed at compile-time to save periodic snapshots of system state to NVM; (2) Hibernus [1], which monitors the external voltage to store RAM and register contents in NVM when a power failure is imminent and (3) QuickRecall [5], a refinement of Hibernus, which uses NVM as a a unified memory; and these three techniques are discussed in more details in the subsequent section.

3. TRANSIENT COMPUTING METHODS

In this section, we describe the three transient computing techniques in details.

3.1 Mementos

The first presented solution is Mementos [10], which uses checkpoints placed at a compile-time. It saves periodic snapshots of system state to non-volatile memory (NVM), which enables it to return to a previous checkpoint after a power failure. Mementos uses the following three different heuristics to insert checkpoints and verify the input voltage level.

- The first heuristic is the `loop-latch mode`. Here, Mementos inserts a trigger point for every loop of the program in order to check the input voltage level at each iteration.
- The second heuristic is the `function-return mode`. In this mode, Mementos inserts trigger points after every function call in order to check the input voltage level when the program returns from a function call.
- The third heuristic is the `timer-aided mode`. This heuristic works in conjunction with the two previous heuristics. Here, Mementos inserts a timer interrupt that sets a flag at predefined execution intervals. At the trigger points, the voltage level is checked only if the flag is set. This heuristic avoids frequent checkpointing, saving energy. We will not consider in the quantitative evaluation section this last technique.

Mementos also has the option of inserting trigger points manually in any position of the program or forcing a snapshot without checking the input voltage level. In order to predict a possible power failure, Mementos compares the input voltage against a threshold by using an analog-to-digital converter (ADC). For Mementos, the checkpoint threshold can be calculated considering a constant current draw I so that the time Δt between two voltage levels V and V_{min} is $\Delta t = C\ (V\text{-}V_{min})/I$. However, a factor complicates the task of checking: Mementos's ability to precisely complete a checkpoint depends on the frequency of trigger points.

This is the main reason why we decided to fix V_{min} bigger than necessary ($V_{min} = 2.4V$), assuming that no energy will be harvested between a trigger point and a power failure. When the supply voltage reaches this threshold, the system considers an imminent power failure and starts checkpointing. Mementos uses two memory blocks and alternates between saving to each of them in order to have always a state-saved start. When power is available again, Mementos looks for a valid checkpoint and copies its content into RAM and registers to continue the program execution from the point it was stopped. The main application of Mementos is on RFID-scale devices powered by a RF-harvesting source, which stores the energy in a capacitor.

Disadvantages of this approach include the fact that many checkpoints will be taken (most of which will be redundant) and that space must be reserved in non-volatile memory for two complete checkpoints in case a power interruption occurs whilst one is being taken.

3.2 Hibernus

Hibernus is the second presented solution, a refinement to Mementos technique for sustaining computation powered by intermittent sources. Hibernus stores a snapshot before a power failure without inserting trigger points in the main program [1]. This technique allows to save only one snapshot every power failure. Hibernus has two states: Active, when the input voltage level is over a restore value (V_R). Hibernating when input voltage is below a threshold (V_H).

Hibernus is implemented on a TI MSP430FR5739 microcontroller which has an internal comparator that was used to send an interruption when the input voltage level crosses either hibernate or restore thresholds. This method uses the FRAM as a non-volatile memory. In order to save a snapshot, Hibernus uses the energy stored in the decoupling capacitance of the microcontroller. This allows to have a low V_H value which increases the active period of the main program. V_H is determined considering the time required to charge the decoupling capacitor in order to have enough energy for saving a snapshot before a power failure. In order to obtain the threshold value, first, it was calculated the energy required to save a snapshot, which is obtained as follows [1]:

$$E_\sigma = n_\alpha E_\alpha + n_\beta E_\beta \tag{1}$$

Where n_α is the number of bytes of the RAM, n_β the number of bytes used by registers, E_α and E_β are the energy required to copy RAM contents and the registers respectively. The microcontroller works in a range of voltage between V_{min} and V_{max}. Given the total capacitance ($\sum C$), the energy E_δ stored in the decoupling capacitor between a given voltage V and V_{min} is calculated as follows [1]:

$$E_\delta = \frac{V^2 - V_{min}^2}{2} . \sum C \tag{2}$$

Inspecting the parameters of the microcontroller [4], it was obtained that the total capacitance is 16μF, and the size in bytes of the RAM and core registers is 1024 and 512 bytes respectively. 4.2nJ energy is needed to save a byte into RAM (E_α) and 2.7nJ in case of FRAM (E_β). Substituting these values in (1) it is obtained that to save a snapshot consumes 5.7 μJ (E_σ). To save a complete snapshot requires that $E_\sigma \leq E_\delta$. The microcontroller works in a range from V_{min}=1.9V to V_{max}=3.6V and the obtained threshold, considering $E_\sigma = E_\delta$, to save a complete snapshot is 2.17V. In order to add hysteresis, V_R was set higher to allow V_{cc} to be over V_H. An internal comparator is checking the voltage level and when it is below V_H, the comparator generates an interrupt. Inside the interrupt handler a function is called to save the snapshot into FRAM, the checkpointing is set and then, system enters in low-power mode. Whether the input voltage is never lower than 1.9V and its value rises again over V_R, the system exits from low-power mode and continues where it was stopped without restoring the whole system. In the case that the input voltage goes below 1.9V, the microcontroller is turned off. When the energy is available again, the system first checks the flag. If the flag is set means that a snapshot is saved. Therefore, Hibernus restores the RAM's contents and the registers' values. Then, it resets the flag and the program continues where it was interrupted. Hibernus is transparent to the programmer. It just need to include *hibernus.h* file that contains all the functionality and call the routines *initialise()*, *hibernate()* and *restore()*.

3.3 QuickRecall

The last proposed solution is QuickRecall [5], which is similar to Hibernus but it allows FRAM to be also utilized as RAM, enabling the system to work as an "unified memory system". In this way, only the FRAM is used as a unified memory while the system's RAM is not used. In order to check the voltage level, the system uses an external comparator, which is connected to the GPIO pins of the microcontroller. This comparator is configured with a trigger voltage (V_{trig}) and sends a signal output when the input voltage level (V_{cc}) is smaller than V_{trig}. The value of trigger voltage is not required to be relative high, unlike Mementos, because it just needs to back up peripherals, program counter, stack pointer, status register and general purpose registers (GPR) before a power failure occurs. Thus, it requires a value of 2.0003V. QuickRecall uses a flag which is set during checkpointing. This flag is used by system to know whether there is a stored checkpoint or not after a power failure. If flag is set, all peripherals are initialized; a check is performed to determine if $V_{cc} > V_{trig}$: if so, core registers are restored, the flag is cleared and the main program is executed. A possible disadvantage of QuickRecall is that it relies on the use of a processor with a unified FRAM memory.

4. QUANTITATIVE EVALUATION

In this section we first evaluate the three techniques mathematically, establishing the scenario where one technique outperforms the others. Later we validate the same using a signal generator on a common microcontroller platform.

4.1 Mathematical Evaluation

4.1.1 Execution Time Comparison

The total time, T_{hibernus}, to execute a test algorithm with *Hibernus* is given by (3), where T_a is the CPU time required to execute the algorithm, n_ι is the number of power interruptions (where $V_{\text{cc}} < V_{\text{min}}$) per algorithm execution, T_s is the time required to save a snapshot to NVM, T_r is the time required to restore from NVM memory, and $\overline{T_\lambda}$ is the average time spent sleeping (after a snapshot has been saved but before $V_{\text{cc}} = V_{\text{min}}$, and on power-up when $V_{\text{min}} < V_{\text{cc}} < V_{\text{R}}$). The absolute limit of supply interruption frequency, f_ι, is $1/(T_s+T_r)$. The execution time of the QuickRecall is similar to that of the Hibernus and is therefore given by Equation 3.

$$\underbrace{T_{\text{Hibernus\_QuickRecall}}}_{\text{Total execution}} = \overbrace{T_a}^{\text{Algorithm}} + \underbrace{n_\iota}_{\text{No. interruptions}}(\overbrace{T_s}^{\text{Save snapshot}} + \underbrace{T_r}_{\text{Restore snapshot}} + \overbrace{\overline{T_\lambda}}^{\text{Sleep}}) \tag{3}$$

The total time, T_{mementos}, to execute an algorithm with Mementos is given by (4), where n_m is the number of checkpoints per complete execution of the algorithm, T_m is the time taken for an ADC reading of V_{cc}, and ρ_s is the proportion of checkpoints resulting in a snapshot, taking T_s.

$$\underbrace{T_{\text{mementos}}}_{\text{Total execution}} = \overbrace{T_a}^{\text{Algorithm}} + \underbrace{n_\iota}_{\text{No. interruptions}}(\overbrace{T_r}^{\text{Restore snapshot}} + \frac{T_a}{\underbrace{2n_m}_{\text{Backtrack}}}) + \overbrace{n_m(T_m + \rho_s T_s)}^{\text{Monitoring and save snapshot}} \tag{4}$$

Hence, $T_{\text{hibernus}} < T_{\text{mementos}}$ provided $n_\iota(T_a/2n_m)+n_mT_m+(n_m\rho_s - n_\iota)T_s > n_\iota\overline{T_\lambda}$; that is, *Hibernus* spends less time sleeping than Mementos spends on backtracks (re-running code that was executed between a snapshot and a power interruption), sampling V_{cc}, and redundant snapshot saves. This is evaluated experimentally in the next section.

4.1.2 *Comparison of Energy Consumption*

Let P_F and P_R denote the average power consumption for accessing the FRAM and RAM, respectively. Usually, $P_F > P_R$. In QuickRecall, both the application code and dynamic data structures are stored in FRAM, while in Hibernus, the application code resides in the FRAM while the dynamic data structures in the RAM.

When the system is powered using a time varying source, e.g., a sinusoidal signal, both Hibernus and QuickRecall approaches behave similarly by storing and restoring checkpoints. The energy overhead for checkpoints in the two approaches can be evaluated as follows.

The energy consumed by Hibernus, $E_{hibernus}$, depends on the size of the volatile memory and the energy consumption for copying each byte.

$$E_{hibernus} = n_\alpha E_\alpha + n_\beta E_\beta \tag{5}$$

Here, n_α and n_β are the sizes of the RAM and registers (in bytes) respectively. E_α and E_β are the energy required to copy each RAM and register byte to NVM (J/byte).

The energy consumed by QuickRecall is given by

$$E_{quickrecall} = n_\beta E_\beta \tag{6}$$

Clearly, $E_{quickrecall} < E_{hibernus}$. As can be seen, the energy for checkpointing and restore for QuickRecall is lower than that of Hibernus. However, for a system powered by a DC source, the energy consumption of Hibernus is lower than that of QuickRecall. We are interested in finding the crossover frequency where one technique outperforms the other. To do so, it is important to note that, in each power cycle, the system will hibernate and restore once. Assuming f is the frequency of input source, the energy overhead of checkpointing and restore for Hibernus = $E_{hibernus} - E_{quickrecall}$. The crossover frequency is given by

$$f\_cross = \frac{(P_F - P_R)}{(E_{hibernus} - E_{quickrecall})} \tag{7}$$

It is important to note that P_R and P_F depend on the application code and hence the crossover frequency is dependent on the application being executed. This crossover frequency is validated experimentally in the next section.

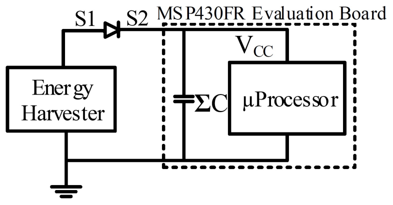


Figure 3: Experimental setup

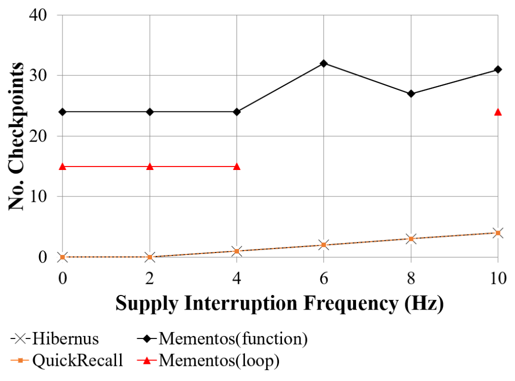


Figure 4: Number of checkpoints of the three approaches.

4.2 Experimental Validation

4.2.1 *Experimental Setup*

This section provides the experimental validation of the three approaches implemented on a TI MSP430FR5739 microcontroller. This platform has 1KB of RAM and 16KB of FRAM. To perform the required experiments, a signal generator is used to power the system, and a DC power analyzer is used to record power consumption. Figure 3 plots the experimental setup used for Hibernus, QuickRecall and Mementos. As shown in this figure, the microcontroller is powered using an energy harvester through the diode. C represents the total on-board decoupling capacitance. The internal voltage comparator of the MSP430FR platform is used for voltage comparison for all the three approaches.

4.2.2 *Application Scenarios*

The microcontroller's clock is configured to run at 8MHz executing the FFT application, which analyses three arrays, each holding 128 8-bit samples of tri-axial accelerometer data. The system is powered with two different sources – a 3.4V DC and Sinusoidal sources with ±3.4V amplitude operating at frequencies ranging from 2 Hz to 10 Hz.

4.2.3 *Number of Checkpoints Executed*

Figure 4 shows the number of checkpoints executed by the three transient computing approaches during the execution of the FFT. A range of supply frequencies (2-10 Hz, and

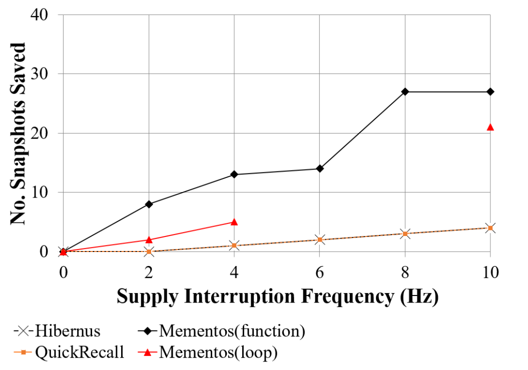


Figure 5: Number of snapshots of the three approaches.

DC) were chosen to represent the intermittent power output that may be expected from a high-power EH. As can be seen, Hibernus and QuickRecall modulate the number of times snapshots are taken as a function of the supply interruption frequency, while Mementos executes a static number of checkpoints (15 and 24 times), although some are repeated when $V_{cc} < V_{min}$ during a snapshot. Moreover, Mementos (loop approach) operates unstably with frequencies higher than 4 Hz due to the static and uneven placement of checkpoints at compile time: checkpoints are only inserted at function calls or loops. In cases where the supply is interrupted in the period between a restore and the next snapshot being saved, the system can become 'stuck', i.e. executes the same portion of code from the last saved checkpoint before $V_{cc} < V_{min}$ without reaching or being able to save a snapshot at the next checkpoint.

4.2.4 *Number of Snapshots Executed*

Figure 5 shows the number of snapshots saved by the three approaches. Hibernus and QuickRecall saves a snapshot every time the hibernate routine is executed, while Mementos saves a snapshot only when $V_{cc} < V_{min}$. The number of snapshots with Mementos is therefore correlated to each checkpoint placement, the value of V_{min} and the supply interruption frequency, while for Hibernus and QuickRecall this depends on the supply interruption frequency only.

4.2.5 *Number of Restores Executed*

Fig. 6 shows that Hibernus and QuickRecall complete execution of the FFT application over the same number of power interruptions while Mementos takes for both loop and function approaches a bigger number of cycles.

4.2.6 *Time Overhead*

Figure 7 plots the time overhead of the three approaches for different interruption frequencies while executing the FFT application. As established mathematically earlier in this section, the time overhead of Mementos is much higher than that of QuickRecall and Hibernus. This is also validated in the figure. It is important to observe that as the supply interruption frequency increases, the execution time overhead of Mementos in the function mode increases rapidly, increasing to over 100% overhead (2x execution time) for an

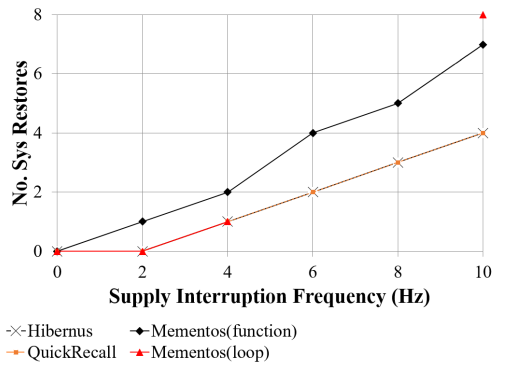


Figure 6: Number of restores of the three approaches.

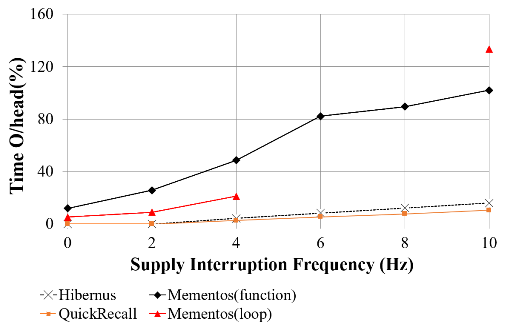


Figure 7: Time Overhead of the three approaches.

interruption frequency of 10 Hz. Finally, the time overhead for QuickRecall is similar to that of the Hibernus approach.

4.2.7 *Current Consumption*

Figure 8 reports the current consumption of QuickRecall and Hibernus using a low-frequency input source while executing the FFT application. The current peaks in the figure correspond to the time when the microcontroller is on. At other times, the current is very close to zero. This is because at these times, the microcontroller is in hibernate state and does not consume any current. It is important to note that the current consumption of Mementos is similar to that of Hibernus and is therefore not included. As can be seen from the figure, the current consumption of QuickRecall is higher than that of Hibernus.

4.2.8 *Hibernus vs QuickRecall as a function of Interruption Frequencies*

Figure 9 plots the energy results for QuickRecall and Hibernus as a function of the supply interruption frequency while executing the FFT application. The system is powered using a square wave generator to simulate the interruption behavior. An interruption frequency f signifies that the system is interrupted f times per second. In other words, the system is interrupted every $1/f$ seconds. The interruption frequency reported in the figure covers the typical scenarios

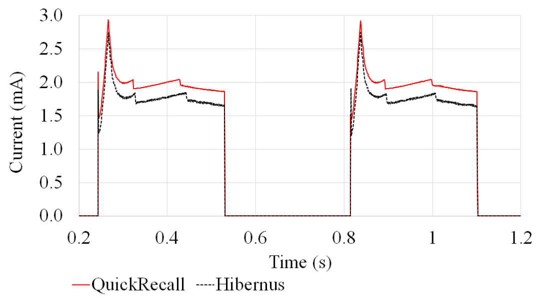


Figure 8: Current comparison

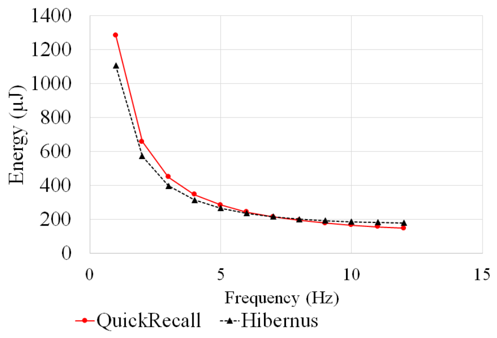


Figure 9: Energy Comparison of Hibernus and QuickRecall

encountered in real energy harvesters such as photovoltaic cell and wind turbines. As seen from the figure, the energy consumption of QuickRecall is higher than the Hibernus at lower interruption frequencies (less than 7Hz). As the frequency is increased beyond 7 Hz, the energy consumption of Hibernus increases. Thus, for the FFT application, it is energy efficient to use Hibernus for interruption frequencies lower than 7Hz, while for higher interruption frequencies, QuickRecall is more energy efficient.

5. CONCLUSIONS

In this paper, we provided a quantitative analysis of three transient computing methods. These approaches are first evaluated theoretically, and then validated with experimental measurements on the same microcontroller platform with standard FFT application. The objective is to evaluate and to compare them to identify in which conditions or scenarios one outperform the others. In particular, Mementos is useful when an application is known a priori, as it is possible to place checkpoints near critical sections (loop or function calls are just a few examples of possible strategies), enabling systems to restart execution at the beginning or after these sections. On the other hand, Hibernus and QuickRecall are completely application agnostic and they introduce a smaller time and energy overhead. However, QuickRecall can only be used with unified memory systems while Hibernus is more platform agnostic and can be used with different kind of standard systems. Apart from this, Hibernus is more energy efficient at lower interruption frequencies, while QuickRecall is more energy efficient at higher frequencies. One of the important limitations of these approaches is that they are not adaptive to the dynamics of the energy harvesting source. In future, we will investigate adaptive checkpointing approaches that takes system snapshots depending on the dynamics of the energy harvesting sources.

6. REFERENCES

[1] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-hashimi, D. Brunelli, and L. Benini. Hibernus : Sustaining Computation during Intermittent Supply for Energy-Harvesting Systems. 7(1):1–4, 2015.

[2] S. Beeby and N. White. *Energy harvesting for autonomous systems.* Artech House, 2014.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[4] Datasheet. MSP430FR5739, 2012. [Online] Available: http://www.ti.com/lit/ds/symlink/msp430fr5739.pdf.

[5] H. Jayakumar, A. Raha, and V. Raghunathan. QUICKRECALL: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. *Proc. IEEE Int. Conf. VLSI Des.*, pages 330–335, 2014.

[6] H. Kim, E. Kim, J. Choi, D. Lee, and S. Noh. Building fully functional instant on/off systems by making use of non-volatile ram. In *Consumer Electronics (ICCE), 2011 IEEE International Conference on*, pages 675–676, Jan 2011.

[7] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 526–537, Feb 2015.

[8] P. Mitcheson, E. Yeatman, G. Rao, A. Holmes, and T. Green. Energy harvesting from human and machine motion for wireless electronic devices. *Proceedings of the IEEE*, 96(9):1457–1486, Sept 2008.

[9] S. Onkaraiah, M. Reyboz, F. Clermidy, J. Portal, M. Bocquet, C. Muller, H. Hraziia, C. Anghel, and A. Amara. Bipolar reram based non-volatile flip-flops for low-power architectures. In *New Circuits and Systems Conference (NEWCAS), 2012 IEEE 10th International*, pages 417–420, June 2012.

[10] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on RFID-scale devices. *ACM SIGPLAN Not.*, pages 159–170, 2011.

[11] G. Rebel, F. Estevez, P. Gloesekoetter, and J. M. Castillo-Secilla. Energy harvesting on human bodies. In *Smart Health*, pages 125–159. Springer, 2015.

Appendix B

RESTOP User Manual

In this appendix, the user manual of RESTOP is presented. In order to incorporate the functionality of the proposed middleware, two header files have to be included in the main application:

1. #include "Config.h"
2. #include "RESTOP_func.h"

1. Config.h

In this file, the user defines the following parameters:

- **Table**. This parameter (#define Table *size*) is used to define the size of the Instruction History Table. Here, size indicates the number of locations where the peripheral instructions will be saved.
- **Cap**. This parameter (#define Cap *value*) is needed to set the capacitance value used as energy buffer. This value is used by RESTOP to calculate and update the restore threshold (V_R) in the snapshotting routine of the transient computing approach.
- **$\mathbf{V}_{min}$**. This parameter (#define V_{min} *voltage*) is the minimum operating voltage of the system. It is also needed by RESTOP to calculate and update V_R.
- **reg_reset**. This is an array where the user has to define the reset register of each attached peripheral. The format is as follows:

  reg_reset [] = {{reg_peripheral1}, {reg_peripheral2}, {reg_peripheralN}};

  The number of positions depends on the number of attached registers. If only one peripheral is needed, the user can remove the unused sections or just ignore them.

- **cmd_write**. This array is needed to define the command value for those peripherals that need this parameter before a write operation. The format is:

  cmd_write [] = {{cmd_peripheral1}, {cmd_peripheral2}, {cmd_peripheralN}};

  The number of positions depends on the number of attached registers. If no command is needed, all the array sections have to be filled with zeros, e.g. two peripherals are attached but none of them needs a command:

  cmd_write [] = {{0x0}, {0x0}};

- **cmd_read**. The function and format of this array are similar than the previous one but this is for read operations.

- **i2c_add**. In this array, the user set the address for the I^2C peripherals. The format is as follows:

  i2c_add [] = {{add_peripheral1}, {add_peripheral2}, {addr_peripheralN}};

  Depending on the number that corresponds to the I^2C peripheral is where the address has to be set and the others have to be filled with zeros, e.g. there are two attached peripherals, but the second is the I^2C one, the array would be filled as follows:

  i2c_add [] = {{0x0}, {0xEE}};

- **pwr_pi**. Here, the user has to set the power consumption of the MCU when issuing an instruction through SPI and I2C protocols. The format is:

  pwr_pi [] = {{W_SPI}, {W_I2C}};

  The values have to be introduced in Watts. The first location is for the power consumption when issuing instructions to SPI and the second for I^2C peripherals. If only one type of protocol is used, the other must be filled with zeros, e.g. a system where only an I^2C peripheral is attached, the array would be:

  pwr_pi [] = {{0}, {0.002}};

- **Time_spi**. Here, the user set the time (in seconds) spent by the MCU to issue SPI peripheral instructions of different number of bytes. The format is described below:

  Time_spi [] = {{time_3params}, {time_2params}, {time_1param}};

  From left to right, each position corresponds to the time taken to issue three, two and one parameters per instruction, respectively (1 parameter = 1 byte). If the attached peripheral does not operate with the three different number of parameters, the unused sections have to be filled with zeros, e.g. if a peripheral only operates with two parameters per instruction, the array would be filled as follows:

  Time_spi [] = {{0}, {0.000120}, {0}};

- **Time_i2c**. This array follows the same format than Time_spi but this is for I2C peripherals.

  Time_i2c [] = {{time_3params}, {time_2params}, {time_1param}};

2. RESTOP_func.h

This file contains the declaration of the functions required by RESTOP to save and issue each peripheral instruction. The user is not expected to modify this file.

GENERIC FUNCTIONS

RESTOP is composed by three generic functions to perform read or write operations on the peripheral, which are:

1. *RESTOP_write*
2. *RESTOP_read*
3. *RESTOP_strobe*
4. *RESTOP_restore*
5. *RESTOP_VR_adj*

1. RESTOP_write

This function performs *write* operations on the peripheral. It is composed by six parameters:

RESTOP_write (Prv, ID, Register, Value, Burst, Protocol)

The first parameter is to define the criteria of *Not-save* ('d0), *Save* ('d1), *Save-but-replace* ('d2), *Save and preserve* ('d5), *Save-but-replace* and *Preserve* ('d6). The second parameter (ID) is to define the peripheral to which the instruction will be issued (the value goes from 1 to N). The parameters Register and Value are each one byte, corresponding to the register width of typical digital interface peripherals. Burst is a flag to indicate if the instruction is one of a write-burst operation. From the first instruction to the N-1, this flag is set to 1. The last one from the burst operation is set to zero. Below, it is shown an example of how to use the burst flag when three burst instructions are issued:

1. **RESTOP_write** (1, 1, 0x0E, 0x2A, 1, 0);
2. **RESTOP_write** (1, 1, 0x00, 0x2B, 1, 0);
3. **RESTOP_write** (1, 1, 0x00, 0x2C, 0, 0);

In the first instruction, the Register where the values will be written is set. It is also enable the burst flag. In the second instruction, instruction, the Register parameter is set to zero, and only the value to be written and the burst flag are set. Finally, in the last burst instruction, the flag is set to zero. The last parameter of this function (Protocol) is to define the protocol of the peripheral (0=SPI; 1=I^2C).

2. RESTOP_read

This function performs read operations on the peripheral. The parameters are the same as in the previous function, except for the Value that is not needed for read operations. This function returns the read value from the peripheral.

uint8_t **RESTOP_read** (Prv, ID, Register, Burst, Protocol)

3. RESTOP_strobe

This function performs write operations that, unlike *RESTOP_write()*, executes single byte instructions. The criteria for the parameter is the same than in the previous functions.

RESTOP_strobe (Prv, ID, Register, Protocol)

4. RESTOP_restore

This function is in charge of restoring the peripheral state. It has to be placed at the end of the restoring routing of the transient computing approach. The function does not receive or return any value.

RESTOP_restore ()

5. RESTOP_VR_adj

This function dynamically calculates the value for V_R depending on the number of peripheral instructions that are saved before a power failure. It has to be placed at the beginning of the snapshotting routine of the transient computing approach and returns the new value for V_R, which has to substitute the one previously set.

float **RESTOP_VR_adj** ()

Appendix C

Step Counter Experimental Set-up

In this appendix, the experimental set-up of the step counter is graphically described. Figure C.1 shows each element of the transient step counter. The ferroelectret insole was worn inside the shoe and the step counter circuitry was placed in a box in order to carry it in the hand during the physical routines.

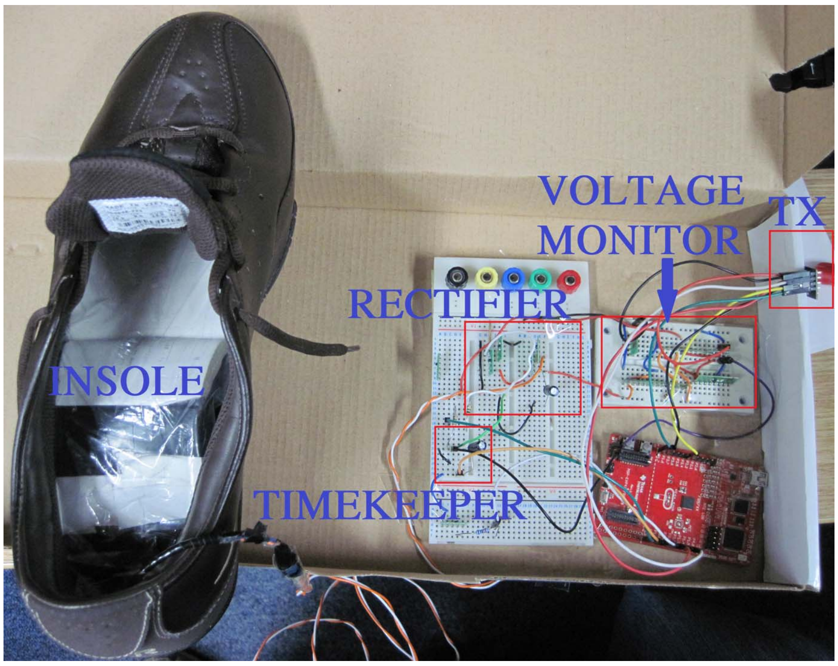


Figure C.1: Transient step counter set-up. The ferroelectret insole was worn in the shoe and the circuitry placed in a box to carry it in the hand

Figure C.2 shows the three devices used in the experimental routine: a Mi Fitness Band, a Fitness tracker smartphone application and the proposed transient step counter inside the white box.

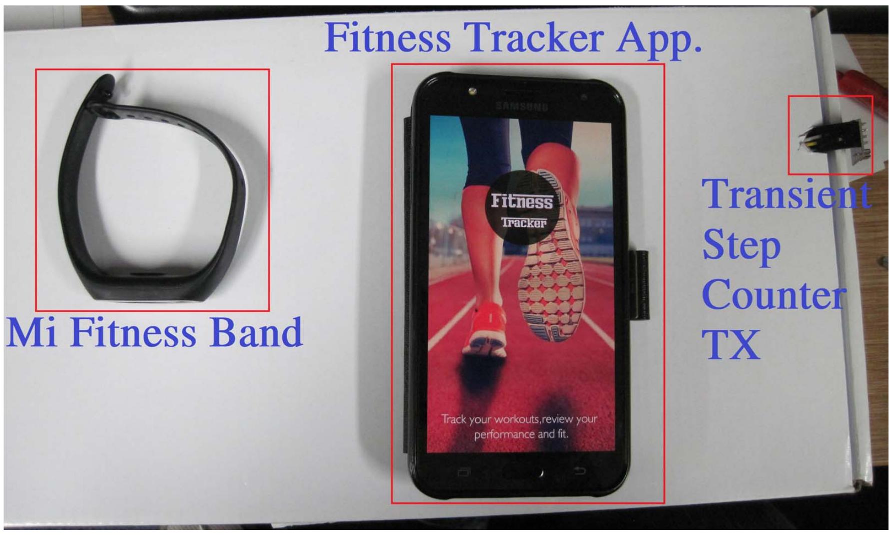

Fitness Tracker App.
Mi Fitness Band
Transient Step Counter TX
Fitness Tracker
Track your workouts,review your performance and fit.

Figure C.2: The three devices used during the experimental validation.

A laptop was configured as a receiver (Figure C.3). Thus, the collected and processed data was presented in the screen through a serial terminal and saved for future analysis.

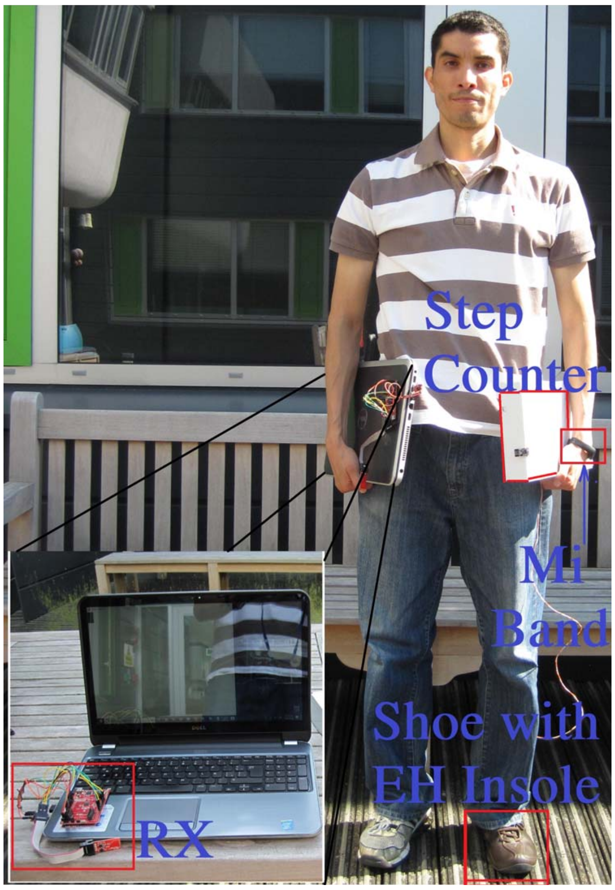


Figure C.3: A laptop was used as a receiver in order to present the data collected and processed by the transient step counter.

For a better understanding of how the transient step counter works, please visit: https://youtu.be/mKXCUrFGk4Y

References

[1] E. Inga, M. Campaña, R. Hincapié, and O. Moscoso-Zea, "Optimal Deployment of FiWi Networks Using Heuristic Method for Integration Microgrids with Smart Metering," *Sensors*, vol. 18, no. 8, pp. 10:1–10:21, 2018.

[2] "Touche: Making Real World Touch Sensitive," Technical Program Lead Google, USA [Online Document], 2012. Available: http://www.ivanpoupyrev.com/projects/touche [Accessed: Dec 2015].

[3] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-hashimi, D. Brunelli, and L. Benini, "Hibernus : Sustaining Computation during Intermittent Supply for Energy-Harvesting Systems," *IEEE Embedded Systems Letters*, vol. 7, pp. 15 – 18, mar 2015.

[4] "Renesas Electronics Corporation: MCU Programming. Learning about Peripherals and GPIO," [Online Document], 2015. Available: http://www.renesas.com/edge_ol/engineer/16/index.jsp [Accessed: Dec 2015].

[5] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith, "Design of an RFID-based battery-free programmable sensing platform," *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 11, pp. 2608–2615, 2008.

[6] H. G. Lee and N. Chang, "Powering the IoT: Storage-less and converter-less energy harvesting," *The 20th Asia and South Pacific Design Automation Conference*, pp. 124–129, 2015.

[7] H. Jayakumar, A. Raha, and V. Raghunathan, "QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pp. 330–335, Jan 2014.

[8] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: A Self-Calibrating and

Adaptive System for Transiently-Powered Embedded Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[9] J. Hester, N. Tobias, A. Rahmati, L. Sitanayah, D. Holcomb, K. Fu, W. P. Burleson, and J. Sorber, "Persistent Clocks for Batteryless Sensing Devices," *ACM Trans. Embed. Comput. Syst.*, vol. 15, pp. 77:1–77:28, Aug. 2016.

[10] C. W. Yau, T. T. O. Kwok, C. U. Lei, and Y. K. Kwok, *Energy Harvesting in Internet of Things*, pp. 35–79. Singapore: Springer Singapore, 2018.

[11] S. Moltchanov, I. Levy, Y. Etzion, U. Lerner, D. M. Broday, and B. Fishbain, "On the feasibility of measuring urban air pollution by wireless distributed sensor networks," *Science of The Total Environment*, vol. 502, pp. 537 – 547, 2015.

[12] "Smoke Detector IC," [Online Document]. Available: https://www.nxp.com/docs/en/data-sheet/MC146010.pdf [Accessed: Dec. 2018].

[13] "Fitness Tracker," [Online Document]. Available: http://rallysolutions.com/ [Accessed: Sept. 2018].

[14] "The influence of long term trends in pollutant emissions on deposition of sulphur and nitrogen and exceedance of critical loads in the united kingdom," *Environmental Science & Policy*, vol. 12, no. 7, pp. 882 – 896, 2009.

[15] Murugavel Raju, "Energy Harvesting. ULP meets energy harvesting: A game-changing combination for design engineers (Whitepaper)," tech. rep., Texas Instruments, Austin, Texas, 2008.

[16] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless Sensor Networks: A Survey," *Computer Network*, vol. 38, pp. 393–422, Mar. 2002.

[17] Y. E. Krasteva, J. Portilla, E. de la Torre, and T. Riesgo, "Embedded Runtime Reconfigurable Nodes for Wireless Sensor Networks Applications," *IEEE Sensors Journal*, vol. 11, pp. 1800–1810, Sep. 2011.

[18] B. Rashid and M. H. Rehmani, "Applications of wireless sensor networks for urban areas: A survey," *Journal of Network and Computer Applications*, vol. 60, pp. 192 – 219, 2016.

[19] S. Kurt, H. U. Yildiz, M. Yigit, B. Tavli, and V. C. Gungor, "Packet Size Optimization in Wireless Sensor Networks for Smart Grid Applications," *IEEE Transactions on Industrial Electronics*, vol. 64, pp. 2392–2401, March 2017.

[20] R. Negra, I. Jemili, and A. Belghith, "Wireless Body Area Networks: Applications and Technologies," *Procedia Computer Science*, vol. 83, pp. 1274 – 1281,

2016. The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016) / The 6th International Conference on Sustainable Energy Information Technology (SEIT-2016) / Affiliated Workshops.

[21] T. Liu and D. Lu, "The application and development of IOT," in *2012 International Symposium on Information Technologies in Medicine and Education*, vol. 2, pp. 991–994, 2012.

[22] D. B. Rawat, "Wireless Sensor Networks for Large Infrastructure Monitoring," [Online Document], 2015. Available: http://www.cwins.org/research/wireless-sensor-networks [Accessed: Oct 2016].

[23] S. Rani, R. Talwar, J. Malhotra, S. H. Ahmed, M. Sarkar, and H. Song, "A Novel Scheme for an Energy Efficient Internet of Things Based on Wireless Sensor Networks," *Sensors*, vol. 15, no. 11, pp. 28603–28626, 2015.

[24] B. Dorsemaine, J.-P. Gaulier, J.-P. Wary, N. Kheir, and P. Urien, "Internet of Things: A Definition & Taxonomy," in *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, pp. 72–77, IEEE, Sep. 2015.

[25] I. Lee and K. Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises," *Business Horizons*, vol. 58, no. 4, pp. 431 – 440, 2015.

[26] L. Xu, W. He, and S. Li, "Internet of Things in Industries: A Survey," *IEEE Transactions on Industrial Informatics*, vol. PP, no. 99, pp. 1–11, 2014.

[27] V. Talla, S. Pellerano, H. Xu, A. Ravi, and Y. Palaskas, "Wi-Fi RF energy harvesting for battery-free wearable radio platforms," in *2015 IEEE International Conference on RFID (RFID)*, pp. 47–54, IEEE, Apr. 2015.

[28] M. Naumann, R. C. Karl, C. N. Truong, A. Jossen, and H. C. Hesse, "Lithium-ion Battery Cost Analysis in PV-household Application," *Energy Procedia*, vol. 73, pp. 37 – 47, 2015. 9th International Renewable Energy Storage Conference, IRES.

[29] W. K. G. Seah, Z. A. Eu, and H.-p. Tan, "Wireless Sensor Networks Powered by Ambient Energy Harvesting (WSN-HEAP) – Survey and Challenges," in *IEEE 1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology*, (Aalborg, Denmark), pp. 1–5, 2009.

[30] D. Niyato, E. Hossain, M. Rashid, and V. Bhargava, "Wireless sensor networks with energy harvesting technologies: a game-theoretic approach to optimal energy management," *IEEE Wireless Communications*, vol. 14, no. August, pp. 90–96, 2007.

[31] B. Ransford, *Transiently Powered Computers.* PhD thesis, University of Massachusetts Amherst, 2013.

[32] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "InK: Reactive Kernel for Tiny Batteryless Sensors," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, (New York, NY, USA), pp. 41–53, ACM, 2018.

[33] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on RFID-scale devices," *ACM SIGPLAN Notices*, pp. 159–170, 2011.

[34] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, (New York, New York, USA), pp. 575–585, ACM Press, 2015.

[35] J. V. D. Woude and M. Hicks, "Intermittent Computation without Hardware Support or Programmer Intervention," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).*, pp. 16–32, USENIX Association, 2016.

[36] K. Maeng and B. Lucia, "Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 129–144, USENIX Association, 2018.

[37] A. Gomez, L. Sigrist, M. Magno, L. Benini, and L. Thiele, "Dynamic energy burst scaling for transiently powered systems," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 349–354, March 2016.

[38] G. V. Merrett, "Invited - Energy harvesting and transient computing," in *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16*, no. April, (New York, New York, USA), pp. 1–2, ACM Press, 2016.

[39] H. Navarro-Hellín, R. Torres-Sánchez, F. Soto-Valles, C. Albaladejo-Pérez, J. López-Riquelme, and R. Domingo-Miguel, "A wireless sensors architecture for efficient irrigation water managemen," *Agricultural Water Management*, vol. 151, pp. 64 – 74, 2015.

[40] L. Catarinucci, D. de Donno, L. Mainetti, L. Palano, L. Patrono, M. L. Stefanizzi, and L. Tarricone, "An IoT-Aware Architecture for Smart Healthcare Systems," *IEEE Internet of Things Journal*, vol. 2, pp. 515–526, Dec 2015.

[41] E. Cordelli, G. Pennazza, M. Sabatini, M. Santonico, and L. Vollero, "A Smart Sensor Architecture for eHealth Applications," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 02, pp. 800–804, July 2018.

[42] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. P. Burleson, and K. Fu, "TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices Without Clocks," in *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, (Berkeley, CA, USA), pp. 36–36, USENIX Association, 2012.

[43] A. Rodriguez Arreola, D. Balsamo, A. K. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, and G. V. Merrett, "Approaches to Transient Computing for Energy Harvesting Systems: A Quantitative Evaluation," in *Proceedings of the 3rd International Workshop on Energy Harvesting and Energy Neutral Sensing Systems*, ENSsys '15, (New York, NY, USA), pp. 3–8, ACM, 2015.

[44] A. R. Arreola, D. Balsamo, Z. Luo, S. P. Beeby, G. V. Merrett, and A. S. Weddell, "Intermittently-powered Energy Harvesting Step Counter for Fitness Tracking," in *2017 IEEE Sensors Applications Symposium (SAS)*, pp. 1–6, March 2017.

[45] A. Rodriguez Arreola, D. Balsamo, G. V. Merrett, and A. S. Weddell, "RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems," *Sensors*, vol. 18, no. 1, 2018.

[46] A. Rodriguez Arreola, D. Balsamo, G. V. Merrett, and A. S. Weddell, "A Generic Middleware for External Peripheral State Retention in Transiently-Powered Sensor Systems," in *Proceedings of the Fifth ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys'17, (New York, NY, USA), pp. 37–39, ACM, 2017.

[47] R. J. M. Vullers, R. v. Schaijk, H. J. Visser, J. Penders, and C. V. Hoof, "Energy Harvesting for Autonomous Wireless Sensor Networks," *IEEE Solid-State Circuits Magazine*, vol. 2, pp. 29–38, Spring 2010.

[48] B. Gyselinckx, C. Van Hoof, J. Ryckaert, R. F. Yazicioglu, P. Fiorini, and V. Leonov, "Human++: autonomous wireless sensors for body area networks," in *Proceedings of the IEEE 2005 Custom Integrated Circuits Conference, 2005.*, pp. 13–19, Sep. 2005.

[49] S. Chen, S. Member, H. Xu, D. Liu, B. Hu, and H. Wang, "A Vision of IoT : Applications , Challenges , and Opportunities With China Perspective," *IEEE Internet of Things Journal*, vol. 1, no. 4, pp. 349–359, 2014.

[50] Luigi Atzori and Antonio Iera and Giacomo Morabito, "The Internet of Things: A survey"," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.

[51] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future Internet: The Internet of Things Architecture, Possible Applications and Key Challenges," in *2012 10th International Conference on Frontiers of Information Technology*, pp. 257–260, Dec 2012.

[52] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, p. 50, 2008.

[53] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of Things for Smart Cities," *IEEE Internet of Things Journal*, vol. 1, pp. 22–32, feb 2014.

[54] A. Gaur, B. Scotney, G. Parr, and S. McClean, "Smart City Architecture and its Applications Based on IoT," *Procedia Computer Science*, vol. 52, pp. 1089 – 1094, 2015. The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015).

[55] K. Zhang, J. Ni, K. Yang, X. Liang, J. Ren, and X. S. Shen, "Security and Privacy in Smart City Applications: Challenges and Solutions," *IEEE Communications Magazine*, vol. 55, pp. 122–129, January 2017.

[56] Y. K. Tan and S. K. Panda, "Self-Autonomous Wireless Sensor Nodes With Wind Energy Harvesting for Remote Sensing of Wind-Driven Wildfire Spread," *IEEE Transactions on Instrumentation and Measurement*, vol. 60, pp. 1367–1377, April 2011.

[57] "Alternative power sources for remote sensors: A review," *Journal of Power Sources*, vol. 245, pp. 129 – 143, 2014.

[58] P. Kadionik, "Introduction to Embedded Systems," in *Communicating Embedded Systems*, pp. 1–28, John Wiley & Sons, Inc., March 2013.

[59] J. A. Khan, H. K. Qureshi, and A. Iqbal, "Energy management in Wireless Sensor Networks: A survey," *Computers & Electrical Engineering*, vol. 41, pp. 159 – 176, 2015.

[60] C. Moser, D. Brunelli, L. Thiele, and L. Benini, "Real-time scheduling for energy harvesting sensor nodes," *Real-Time Systems*, vol. 37, pp. 233–260, Dec 2007.

[61] P. Mitcheson, E. Yeatman, G. Rao, A. Holmes, and T. Green, "Energy Harvesting From Human and Machine Motion for Wireless Electronic Devices," *Proceedings of the IEEE*, vol. 96, pp. 1457–1486, sep 2008.

[62] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava, "Power management in energy harvesting sensor networks," *ACM Transactions on Embedded Computing Systems*, vol. 6, Sept. 2007.

[63] S. Beeby and N. White, *Energy Harvesting for Autonomous Systems.* Artech House series smart materials, structures, and systems, Artech House, 685 Canton Street. Norwood, MA 02062, 2010.

[64] G. Rebel, F. Estevez, P. Gloesekoetter, and J. M. Castillo-Secilla, “Energy Harvesting on Human Bodies,” in *Smart Health*, pp. 125–159, Springer, 2015.

[65] S. Chalasani and J. M. Conrad, “A survey of energy harvesting sources for embedded systems,” *Conference Proceedings - IEEE SoutheastCon*, pp. 442–447, 2008.

[66] F. Akhtar and M. H. Rehmani, “Energy Harvesting for Self-Sustainable Wireless Body Area Networks,” *IT Professional*, vol. 19, pp. 32–40, March 2017.

[67] M. Ku, W. Li, Y. Chen, and K. J. Ray Liu, “Advances in Energy Harvesting Communications: Past, Present, and Future Challenges,” *IEEE Communications Surveys Tutorials*, vol. 18, pp. 1384–1412, Secondquarter 2016.

[68] S. P. Beeby, M. J. Tudor, and N. M. White, “Energy harvesting vibration sources for microsystems applications,” *Measurement Science and Technology*, vol. 17, pp. R175–R195, Dec. 2006.

[69] A. H. Rajabi, M. Jaffe, and T. L. Arinzeh, “Piezoelectric materials for tissue regeneration: A review,” *Acta Biomaterialia*, vol. 24, pp. 12 – 23, 2015.

[70] S. Orrego, K. Shoele, A. Ruas, K. Doran, B. Caggiano, R. Mittal, and S. H. Kang, “Harvesting ambient wind energy with an inverted piezoelectric flag,” *Applied Energy*, vol. 194, pp. 212 – 222, 2017.

[71] W. Heywang, K. Lubitz, and W. Wersing, *Piezoelectricity. Evolution and Future of a Technology*, vol. 114. Springer Series, Dec 2008.

[72] N. S. Shenck and J. A. Paradiso, “Energy Scavenging with Shoe-mounted Piezoelectrics,” *IEEE Micro*, vol. 21, no. 3, pp. 30–42, 2001.

[73] D. Fourie, “Shoe-Mounted PVDF Piezoelectric Transducer for Energy Harvesting,” *MIT Undergraduate Research Journal*, vol. 19, no. 1, pp. 66–70, 2010.

[74] R. Gerhard-Multhaupt, “Less can be more. Holes in Polymers lead to a New Paradigm of Piezoelectric Materials for Electret Transducers,” *IEEE Transactions on Dielectrics and Electrical Insulation*, vol. 9, no. 5, p. 10, 2002.

[75] G. M. Sessler, “Charge distribution and transport in polymers,” *IEEE Transactions on Dielectrics and Electrical Insulation*, vol. 4, pp. 614–628, Oct 1997.

[76] Z. Luo, D. Zhu, J. Shi, S. Beeby, C. Zhang, P. Proynov, and B. Stark, “Energy harvesting study on single and multilayer ferroelectret foams under compressive force,” *IEEE Transactions on Dielectrics and Electrical Insulation*, vol. 22, pp. 1360–1368, jun 2015.

[77] Z. Luo, D. Zhu, and S. P. Beeby, “Multilayer ferroelectret-based energy harvesting insole,” *Journal of Physics: Conference Series*, vol. 660, p. 012118, dec 2015.

[78] H. A. Sodano, D. J. Inman, and G. Park, “A Review of Power Harvesting from Vibration Using Piezoelectric Materials,” *SAGE The Shock and Vibration Digest*, vol. 36, pp. 197–205, may 2004.

[79] A. C. R. Amirtharajah, “Self-powered low power signal processing,” *IEEE Symposium, VLSI Circuits, Digest of Technical Papers*, vol. 30, no. 1, pp. 25–26, 1997.

[80] Y. Shin, “Non-volatile memory technologies for beyond 2010,” *VLSI Circuits, 2005. Digest of Technical Papers. 2005 Symposium on*, vol. 2, pp. 156–159, 2005.

[81] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, “Introduction to Flash Memory,” *Proceedings of the IEEE*, vol. 91, no. 4, pp. 489–501, 2003.

[82] “Atmel ATmega640 Datasheet,” [Online Document]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf.

[83] S. Mukherjee, T. Chang, R. Pang, M. Knecht, and D. Hu, “A single transistor EEPROM cell and its implementation in a 512K CMOS EEPROM,” *1985 International Electron Devices Meeting*, vol. 31, pp. 616–619, 1985.

[84] T. D. Verykios, D. Balsamo, and G. V. Merrett, “Selective policies for efficient state retention in transiently-powered embedded systems: Exploiting properties of NVM technologies,” *Sustainable Computing: Informatics and Systems*, pp. 5:1–5:12, 2018.

[85] “Texas Instruments MSP430F21x2 Datasheet,” [Online Document], 2013. Available: http://www.ti.com/lit/ds/symlink/msp430f21x2.pdf.

[86] S. Baek, J. Choi, D. Lee, and S. H. Noh, “Energy-efficient and high-performance software architecture for storage class memory,” *ACM Transactions on Embedded Computing Systems*, vol. 12, pp. 81:1—-81:22, mar 2013.

[87] J. M. Hu, Z. Li, L. Q. Chen, and C. W. Nan, “High-density magnetoresistive random access memory operating at ultralow voltage at room temperature,” *Nature Communications*, vol. 2, no. 1, pp. 553–558, 2011.

[88] B. N. Engel, J. Akerman, B. Butcher, R. W. Dave, M. DeHerrera, M. Durlam, G. Grynkewich, J. Janesky, S. V. Pietambaram, N. D. Rizzo, J. M. Slaughter, K. Smith, J. J. Sun, and S. Tehrani, “A 4-Mb toggle MRAM based on a novel bit and switching method,” *IEEE Transactions on Magnetics*, vol. 41, pp. 132–136, Jan 2005.

[89] P. A. Laplante, “Input-Output Devices,” [Online Document], 2012. Available: http://www.bottomupcs.com/peripherals [Accessed: Dec 2015].

[90] I. Wienand, “Computer Architecture: Peripherals and buses,” [Online Document], 2015. Available: https://www.bottomupcs.com/peripherals.xhtml [Accessed: Dec 2015].

[91] “Texas Instruments MSP430FR5739 Datasheet,” [Online Document], 2011. Available: http://www.ti.com/lit/ds/symlink/msp430fr5739.pdf.

[92] “Texas Instruments MSP430FR5847 Datasheet,” [Online Document], 2012. Available: http://www.ti.com/product/MSP430FR5847/datasheet.

[93] S. Kim, S.-M. Song, and Y.-I. Yoon, “Smart Learning Services Based on Smart Cloud Computing,” *Open Access Sensors*, vol. 11, pp. 7835–7850, aug 2011.

[94] B. Latré, B. Braem, I. Moerman, C. Blondia, and P. Demeester, “A Survey on Wireless Body Area Networks,” *Wirel. Netw.*, vol. 17, pp. 1–18, Jan. 2011.

[95] G. J. Pottie and W. J. Kaiser, “Wireless Integrated Network Sensors,” *Communication ACM*, vol. 43, pp. 51–58, May 2000.

[96] Q. Gu, *RF System Design of Transceivers for Wireless Communications.* Springer Science & Business Media, 2006.

[97] A. Luzzatto and M. Haridim, *Wireless transceiver design: mastering the design of modern wireless equipment and systems.* John Wiley & Sons, 2016.

[98] A. Ephremides, “Energy concerns in wireless networks,” *IEEE Wireless Communications*, vol. 9, pp. 48–59, Aug 2002.

[99] J. Hester and J. Sorber, “The Future of Sensing is Batteryless, Intermittent, and Awesome,” in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys ’17, (New York, NY, USA), pp. 21:1–21:6, ACM, 2017.

[100] S. Naderiparizi, A. N. Parks, Z. Kapetanovic, B. Ransford, and J. R. Smith, “WISPCam : A Battery-Free RFID Camera,” in *IEEE International Conference on RFID*, (San Diego, California. USA), pp. 166–173, 2015.

[101] S. Manzari, A. Catini, G. Pomarico, C. D. Natale, and G. Marrocco, “Development of an UHF RFID Chemical Sensor Array for Battery-Less Ambient Sensing,” *IEEE Sensors Journal*, vol. 14, pp. 3616–3623, Oct 2014.

[102] Y. Wang, Y. Liu, C. Wang, Z. Li, X. Sheng, H. G. Lee, N. Chang, and H. Yang, “Storage-Less and Converter-Less Photovoltaic Energy Harvesting With Maximum Power Point Tracking for Internet of Things,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 173–186, Feb 2016.

[103] S. DeBruin, B. Campbell, and P. Dutta, “Monjolo: An Energy-harvesting Energy Meter Architecture,” in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys ’13, (New York, NY, USA), pp. 18:1–18:14, ACM, 2013.

[104] P. Martin, Z. Charbiwala, and M. Srivastava, “DoubleDip: Leveraging Thermoelectric Harvesting for Low Power Monitoring of Sporadic Water Use,” in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys ’12, (New York, NY, USA), pp. 225–238, ACM, 2012.

[105] T. Xiang, Z. Chi, F. Li, J. Luo, L. Tang, L. Zhao, and Y. Yang, “Powering Indoor Sensing with Airflows: A Trinity of Energy Harvesting, Synchronous Duty-cycling, and Sensing,” in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys ’13, (New York, NY, USA), pp. 16:1–16:14, ACM, 2013.

[106] A. Gomez, L. Sigrist, T. Schalch, L. Benini, and L. Thiele, “Wearable, energy-opportunistic vision sensing for walking speed estimation,” in *2017 IEEE Sensors Applications Symposium (SAS)*, pp. 1–6, March 2017.

[107] B. Munir and V. Dyo, “On the Impact of Mobility on Battery-Less RF Energy Harvesting System Performance,” *Sensors*, vol. 18, no. 11, 2018.

[108] N. G. Philip A. Bernstein, Vassos Hadzilacos, *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing, 1987.

[109] J. S. Plank, “An Overview of Checkpointing in Uniprocessor and Distributed Systems. Focusing on Implementation and Performance,” tech. rep., Department of Computer Science. University of Tennessee, Knoxville, Tennessee, 1997.

[110] A. Colin and B. Lucia, “Chain: tasks and channels for reliable intermittent programs,” in *Proc. 2016 ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl. - OOPSLA 2016*, pp. 514–530, ACM Press, 2016.

[111] K. Maeng, A. Colin, and B. Lucia, “Alpaca: Intermittent Execution Without Checkpoints,” *Proc. ACM Program. Lang.*, vol. 1, pp. 96:1–96:30, Oct. 2017.

[112] C. Lattner and V. Adve, “LLVM: a compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, March 2004.

[113] “The LLVM Compiler Infrastructure,” [Online] Available: https://llvm.org/ [Accessed: Nov 2018].

[114] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, “Formalizing the LLVM Intermediate Representation for Verified Program Transformations,” *SIGPLAN Not.*, vol. 47, pp. 427–440, Jan. 2012.

[115] K. Maeng and B. Lucia, “Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 129–144, USENIX Association, 2018.

[116] Z. Ghodsi, S. Garg, and R. Karri, “Optimal Checkpointing for Secure Intermittently-powered IoT Devices,” in *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD ’17, (Piscataway, NJ, USA), pp. 376–383, IEEE Press, 2017.

[117] N. A. Bhatti and L. Mottola, “HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing,” in *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN ’17, (New York, NY, USA), pp. 209–219, ACM, 2017.

[118] J. S. Miguel, K. Ganesan, M. Badr, C. Xia, R. Li, H. Hsiao, and N. E. Jerger, “The EH Model: Early Design Space Exploration of Intermittent Processor Architectures,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 600–612, Oct 2018.

[119] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, “Peripheral state persistence for transiently-powered systems,” in *2017 Global Internet of Things Summit (GIoTS)*, pp. 1–6, June 2017.

[120] Z. Li, Y. Liu, D. Zhang, C. J. Xue, Z. Wang, X. Shi, W. Sun, J. Shu, and H. Yang, “HW/SW co-design of nonvolatile IO system in energy harvesting sensor nodes for optimal data acquisition,” in *Proceedings of the 53rd Annual Design Automation Conference on - DAC ’16*, (Austin, TX, USA), pp. 1–6, ACM Press, 2016.

[121] Y. Liu, Y. Xie, J. Shu, H. Yang, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M.-F. Chang, and S. John, “Ambient energy harvesting nonvolatile processors: From Circuit to System,” *Proceedings of the 52nd Annual Design Automation Conference on - DAC ’15*, pp. 1–6, 2015.

[122] K. Ma, Y. Zheng, S. Li, K. Swaminathan, and X. Li, “Architecture Exploration for Ambient Energy Harvesting Nonvolatile Processors,” in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[123] “Texas Advanced Optoelectronic Solutions. TSL2560 Datasheet,” 2005. [Online Document] Available: https://static.sparkfun.com/datasheets/Sensors/LightImaging/TSL2561.pdf.

[124] “Texas Instruments. RF CC1101 Datasheet,” 2013. [Online Document] Available: http://www.ti.com/lit/ds/symlink/cc1101.pdf.

[125] K. S. Yildirim, H. Aantjes, A. Y. Majid, and P. Pawelczak, "On the Synchronization of Intermittently Powered Wireless Embedded Systems," in *HLPC*, (Atlanta, USA), pp. 1–6, 2016.

[126] K. S. Yildirim, H. Aantjes, P. Pawelczak, and A. Y. Majid, "On the Synchronization of Computational RFIDs," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2018.

[127] A. Rahmati, M. Salajegheh, D. Holcomb, and J. Sorber, "TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks," in *Berkeley, The USENIX Association*, USENIX, 2012.

[128] U. Senkans, D. Balsamo, T. D. Verykios, and G. V. Merrett, "Applications of Energy-Driven and Transient Computing: A Wireless Bicycle Trip Counter," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, (New York, NY, USA), pp. 76:1–76:2, ACM, 2017.

[129] B. Ransford, S. Clark, M. Salajegheh, and K. Fu, "Getting Things Done on Computational RFIDs with Energy-Aware Checkpointing and Voltage-Aware Scheduling," *Proceedings of the 2008 Workshop on Power Aware Computing and Systems (HotPower'08)*, 2008.

[130] B. Campbell and P. Dutta, "An Energy-harvesting Sensor Architecture and Toolkit for Building Monitoring and Event Detection," *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, pp. 100–109, 2014.

[131] C. K. Ho and R. Zhang, "Optimal Energy Allocation for Wireless Communications With Energy Harvesting Constraints," *IEEE Transactions on Signal Processing*, vol. 60, pp. 4808–4818, Sep. 2012.

[132] R. D. Fernandes, N. B. Carvalho, and J. N. Matos, "Design of a battery-free wireless sensor node," in *2011 IEEE EUROCON - International Conference on Computer as a Tool*, pp. 1–4, April 2011.

[133] A. Yamawaki and S. Serikawa, "Door Monitoring System Using Sensor Node with Zero Standby Power," in *Transactions on Engineering Technologies* (S.-I. Ao, H. K. Kim, X. Huang, and O. Castillo, eds.), (Singapore), pp. 73–87, Springer,, 2017.

[134] F. Sivrikaya and B. Yener, "Time Synchronization in Sensor Networks: A Survey," *IEEE Network*, vol. 18, no. 4, pp. 45–50, 2004.

[135] K. S. Prabh, *Real-Time Wireless Sensor Networks.* PhD thesis, University of Virginia, 2007.

[136] X. Lin, Y. Wang, N. Chang, and M. Pedram, "Concurrent Task Scheduling and Dynamic Voltage and Frequency Scaling in a Real-Time Embedded System With

Energy Harvesting," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 1890–1902, Nov 2016.

[137] National Instrumets, "What is a Real-Time Operating System (RTOS)? (Whitepaper)," tech. rep., National Instrument, Austin, Texas, 2013.

[138] S. Kärki and J. Lekkala, "A lumped-parameter transducer model for piezoelectric and ferroelectret polymers," *Measurement*, vol. 45, no. 3, pp. 453 – 458, 2012.

[139] "Pacer: Pedometer and Weight Trainer," [Smartphone Application], 2016. Available: http://www.pacer.cc/ [Accessed: Nov 2016].

[140] "Sony WalkMate," [Smartphone Application], 2016. Available: http://www.sonymobile.com/ [Accessed: Nov 2016].

[141] Q. Liu and C. Jung, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6, Aug 2016.

[142] "ADXL362 Datasheet," 2012. [Online Document] Available: http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf.

[143] A. Diwan, D. Tarditi, and E. Moss, "Memory System Performance of Programs with Intensive Heap Allocation," *ACM Trans. Comput. Syst.*, vol. 13, pp. 244–273, Aug. 1995.

[144] M. S. Johnstone and P. R. Wilson, "The Memory Fragmentation Problem: Solved?," in *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, (New York, NY, USA), ACM, 1998.

[145] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory Allocation for NVRAM.," in *Accelerating Analytics and Data Management Systems in conjunction with Very Large Data Bases*, 2015.

[146] R. Shafik, A. Yakovlev, and S. Das, "Real-Power Computing," *IEEE Transactions on Computers*, vol. 67, pp. 1445–1461, Oct 2018.

[147] C. Pan, M. Xie, and J. Hu, "ENZYME: An Energy-Efficient Transient Computing Paradigm for Ultralow Self-Powered IoT Edge Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 2440–2450, Nov 2018.

[148] Q. Ju and Y. Zhang, "Predictive Power Management for Internet of Battery-Less Things," *IEEE Transactions on Power Electronics*, vol. 33, pp. 299–312, Jan 2018.

[149] S. Yong, J. Shi, and S. P. Beeby, "Metal Layer reinforced multilayer ferroelectret-based energy harvester," *Journal of Physics: Conference Series*, vol. 1052, p. 012115, jul 2018.

[150] N. Sekiya, H. Nagasaki, H. Ito, and T. Furuna, "Optimal Walking in Terms of Variability in Step Length," *J. Orthop. Sport. Phys. Ther.*, vol. 26, pp. 266–272, Nov. 1997.

[151] A. Brajdic and R. Harle, "Walk Detection and Step Counting on Unconstrained Smartphones," in *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, (New York, NY, USA), pp. 225–234, ACM, 2013.

[152] D. M. Harrington, G. J. Welk, and A. E. Donnelly, "Validation of MET estimates and step measurement using the ActivPAL physical activity logger," *Journal of Sports Sciences*, vol. 29, pp. 627–633, Mar. 2011.

[153] "Advanced Encryption Standard (AES)," [Online Document], 2001. Available: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf [Accessed: July 2018].

[154] "Nordic Semiconductor nRF24L01+ Single Chip Datasheet. 2.4GHz ISM Transceiver," [Online Document], 2008. Available: https://www.sparkfun.com/datasheets/RF/nRF2401rev1_1.pdf.

[155] A. E. Kouche, L. Al-Awami, and H. Hassanein, "Dynamically Reconfigurable Energy Aware Modular Software (DREAMS) Architecture for WSNs in Industrial Environments," *Procedia Computer Science*, vol. 5, pp. 264 – 271, 2011. The 2nd International Conference on Ambient Systems, Networks and Technologies (ANT-2011) / The 8th International Conference on Mobile Web Information Systems (MobiWIS 2011).

[156] E. Burton, K. D. Hill, N. T. Lautenschlager, C. Thøgersen-Ntoumani, G. Lewin, E. Boyle, and E. Howie, "Reliability and validity of two fitness tracker devices in the laboratory and home environment for older community-dwelling people," *BMC Geriatrics*, vol. 18, pp. 103–115, Dec. 2018.

[157] "Mi Band 2," [Online Document]. Available: https://www.mi.com/en/miband2/ [Accessed: Sept. 2018].

[158] P. A. Levis, "TinyOS: An Open Operating System for Wireless Sensor Networks (Invited Seminar)," in *7th International Conference on Mobile Data Management (MDM'06)*, pp. 63–63, May 2006.

[159] D. Balsamo, A. Elboreini, B. M. Al-Hashimi, and G. V. Merrett, "Exploring ARM mbed support for transient computing in energy harvesting IoT systems," in *2017 7th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI)*, pp. 115–120, June 2017.