Poster: A Generic Middleware for External Peripheral State Retention in Transiently-Powered Sensor Systems

Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V. Merrett, Alex S. Weddell
Department of Electronics and Computer Sciences
University of Southampton
{ara1g13, db2a12, gvm, asw}@ecs.soton.ac.uk

ABSTRACT

Sensor systems powered by energy harvesting usually include batteries or supercapacitors which impact the system cost and size, need time to be charged and are not environmentally friendly. In recent years, designers have proposed a new concept called *transient computing* that aims to remove these energy storage units and retain the system's state between power outages, in order to cope with an unreliable energy source. However, existing approaches cannot retain the state of external peripherals or are specific to certain peripherals, i.e. they are not generic. This poster proposes a generic middleware, capable to retain the state of external peripherals that are connected to a microcontroller through SPI. The validation shows the proposed approach retains the peripheral configuration between power failures with a maximum time overhead of 15% when configuring the peripheral. However, this represents a 0.77% overhead for a complete example application, which is lower than that caused by existing approaches.

CCS CONCEPTS

• **Computer systems organization** → *Sensors and actuators*;

KEYWORDS

Energy Harvesting;External Peripheral;Transient Computing

1 INTRODUCTION AND BACKGROUND

The Internet of Things (IoT) is the interconnection of trillions of ultra-low power and resource-constrained sensor systems. Efficiently powering IoT systems is an important challenge as they have to operate autonomously for years without charging or replacing batteries. Energy harvesting (EH) can potentially power these devices by generating electrical power from the environment. However, as these sources are unpredictable, EH powered systems incorporate supercapacitors or rechargeable batteries to sustain computation when EH is insufficient.

ENSsys'17, November 5, 2017, Delft, Netherlands

ACM ISBN 978-1-4503-5477-X/XX/XX.
https://doi.org/10.1145/XXXXXXX.XXXXXXX

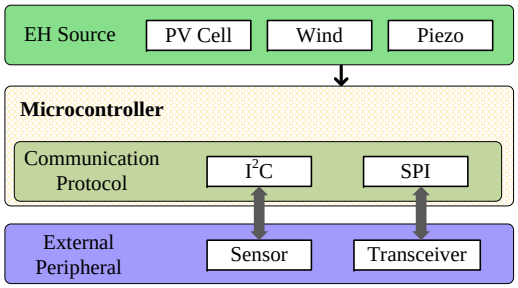

Figure 1: Block diagram of a typical EH sensor system including two external peripherals.

Nevertheless, these energy storage units have a limited life time and increase system cost and physical dimensions.

Transient computing aims to develop systems that operate directly from the EH source, removing the need for energy storage. Prominent approaches to transient computing (e.g. *Hibernus* [1] and *HarvOS* [3]) have solved the challenge of frequent power outages by saving the system's state (main memory, registers, etc.) into non-volatile memory (NVM) before a power failure. Nevertheless, these approaches do not retain the state of peripherals that are attached to the MCU through serial protocols such as I^2C or SPI (Fig. 1).

Another approach called *Sytare* [2] is able to retain the system state and the configuration of SPI peripherals. However, this approach requires the user to write not only the structure where the functions to configure the peripheral are encapsulated and saved, but also the function to restore the peripheral state after a supply interruption. Moreover, this approach saves a snapshot each time a peripheral instruction is issued, which introduces a time overhead of up to 30μs per instruction. This represents an overhead over 137% when configuring a radio transceiver.

This poster proposes a generic middleware, which is able to retain the state of external peripherals that interact with the MCU through SPI. The proposed approach was experimentally validated with an SPI radio transceiver in a transiently-powered system. Results demonstrate the proposed approach properly retains the peripheral state causing a maximum time overhead on the example application of 0.77%, which is lower than that caused by existing approaches.

2 PROPOSED METHODOLOGY

Fig. 2 shows a block diagram of the proposed middleware, which acts as an interface between the main application and the external peripheral. Each instruction, issued by the main application, is saved in a table and then executed on the

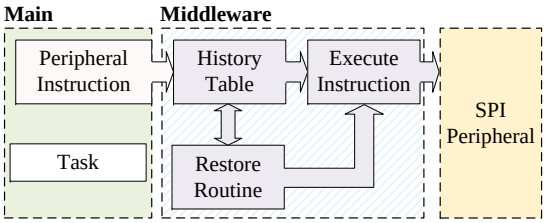

Figure 2: The middleware acts as an interface between the main application and peripherals.

external peripheral. The table (so-called Instruction History Table) can be located in the main memory and protected from supply interruptions through one of the existing approaches to transient computing [1, 3]. The criteria to save an instruction in the history table is explained in Section 2.1, and the process to restore the peripheral state is detailed in Section 2.2.

2.1 Criteria to Save an Instruction

The decision to save an instruction in the history table is taken off-line by the developer following three options:

(1) **Not-save**: The issued instruction is not saved because it does not affect the peripheral configuration (e.g. reading a status register).
(2) **Save**: The issued instruction is saved in the table but it will be replaced if a similar instruction (i.e. with the same register value) is later issued.
(3) **Preserve**: The saved instruction has to be kept regardless of whether a similar instruction is later issued.

Fig. 3 shows the process to save and execute a peripheral instruction. The middleware first evaluates whether the issued instruction is to reset the peripheral. *Reset* is a *write* instruction that not only resets the peripheral but also removes the saved instructions from the table because it discards all previously executed configuration. Once the *Reset* condition is evaluated, the middleware checks whether the issued instruction has to be saved. If not (*Not-save*), the instruction is executed on the peripheral (through the SPI protocol) and the main application continues.

In case the developer wants to save the instruction (*Save*), the middleware checks whether a similar instruction was previously saved in the table. If not, the issued instruction is saved and then executed on the peripheral. If a similar instruction already exists in the table, it evaluates whether that instruction has to be preserved or can be replaced. In the first case (*Preserve*), the issued instruction is saved in a new location in the history table and then it is executed. If the saved instruction can be replaced, it is deleted from the table and the new one is added instead.

2.2 Restore Routine

The restore routine for the proposed middleware is much simpler than the process of saving an instruction. Restoring the peripheral state does not add any substantial overhead, other than that caused by executing the saved instructions. After a power failure, the middleware fetches each saved instruction from the history table and issues it on the SPI

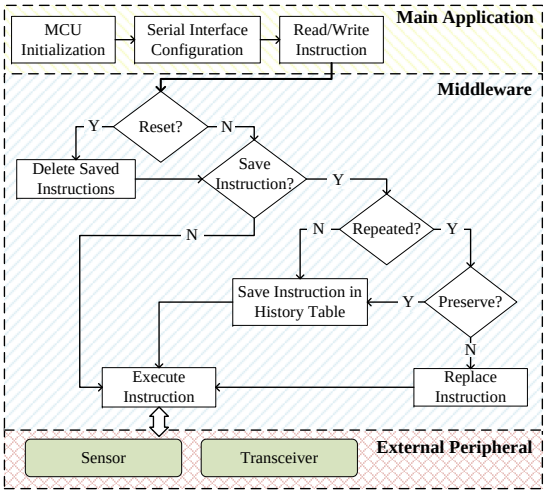

Figure 3: Process followed by the middleware to save and execute a peripheral instruction.

Middleware

Generic Functions

```
void  SPI_write    (Prv,Register,Value,B)
int   SPI_read     (Prv,Register,B)
void  SPI_strobe   (Prv,Register)
void  SPI_restore  (void)
```

Figure 4: Parameters of each generic function.

interface, following the same order the instructions were issued in before the power outage. When the peripheral state is restored, the main application continues execution.

3 SOFTWARE ALGORITHM

The proposed middleware is composed of three main blocks: generic functions to execute each peripheral instruction, parameters required to describe each peripheral instruction and an instruction history table where the issued instructions are saved.

3.1 Generic Functions

The middleware has four functions that wrap the peripheral instructions to save the exchanged data in the history table:

(1) **SPI_read():** This function returns the value read from the specified peripheral register.
(2) **SPI_write():** Performs *write* operations on the peripheral.
(3) **SPI_strobe():** Performs single byte instructions (no data) that when issued, a peripheral sequence starts.
(4) **SPI_restore():** This function is in charge of executing all the instructions stored in the history table and has to be included by the user in the restoring routine of the system.

3.2 Parameters and Configuration File

We have separated the parameters that vary from one peripheral instruction to another (which are entered through the

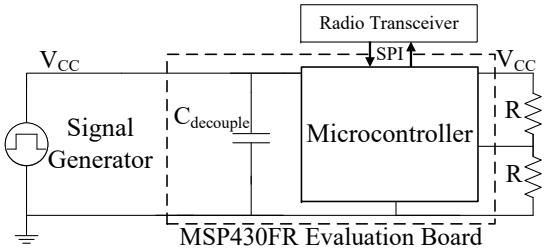

Figure 5: Schematic of the test platform.

generic functions) and those that are static for the peripheral which are declared in a configuration file. Fig. 4 shows the parameters that each function receives from the user. Parameter *Prv* is a 2-bit flag that can have three different values following the criteria detailed in Section 2.1:

- 'b00: Not-save.
- 'b01: Save.
- 'b11: Save and Preserve.

Register and *Value* have a width of one byte each, which is the register width of typical SPI peripherals. Parameter *B* indicates the function will execute a *data burst transmissions* that consist of setting a register which prompts the peripheral to send back a sequence of bytes (if it is a *read* operation), or receive a sequence of values to be written. The function *SPI_restore()* does not receive any parameter.

In the case of the configuration file, the user declares the values for three different parameters:

- **reg_reset**: The user has to write the register that resets the attached peripheral.
- **cmd_read**: Refers to the peripherals that need a *command* parameter to indicate the instruction is to read from the peripheral.
- **cmd_write**: Similar to the previous one but for write operations.

3.3 Instruction History Table

The instruction history table is based on a linked list where each element corresponds to a peripheral instruction. The linked list is implemented by an array of structures, where each structure contains the parameters received through the generic functions. The size of the array can vary depending on the number of instructions that have to be saved.

4 FUNCTIONAL VALIDATION

Fig. 5 shows the experimental set-up, which consists of a signal generator (to emulate an intermittent source), a CC1101 SPI radio transceiver and a TI MSP-EXP430FR5739 evaluation board. The voltage divider is used by the on-chip comparator to monitor input voltage. $C_{decouple}$ refers to the total on-board decoupling capacitance. In order to protect the system from power failures, we included Hibernus [1] in the test application, as it has the best performance in terms of energy and time overheads [4]. However, the proposed middleware can be incorporated in any other transient computing approach. It is important to mention that the inclusion of our middleware in any system is straightforward.

Table 1: Time overhead caused by the middleware.

Active time (ms)	N°. Samples	N°. snapshot	N°. restore	Time (ms)					O/head (%)
				FFT	No Middleware		Middleware		
					Transceiver	Total	Transceiver	Total	
10	32	2	2	19.71	1.05	26.24	1.21	26.4	0.61
10	64	6	6	47.27	1.05	64.76	1.21	64.92	0.25
10	128	14	14	100	1.05	139.41	1.21	139.57	0.11
50	32	0	0	19.71	1.05	20.76	1.21	20.92	0.77
50	64	0	0	47.27	1.05	48.32	1.21	48.48	0.33
50	128	1	1	100	1.05	103.79	1.21	103.95	0.15
100	32	0	0	19.71	1.05	20.76	1.21	20.92	0.77
100	64	0	0	47.27	1.05	48.32	1.21	48.48	0.33
100	128	1	1	100	1.05	103.79	1.21	103.95	0.15

The developer only needs to import the library and use the functions provided (described in Section 3.1) to configure the peripheral, obtain data from it or restore its configuration after a supply interruption.

The main application consists of a Fast Fourier Transform (FFT) that processes 32, 64 and 128 samples previously obtained from a tri-axial accelerometer. In order to measure the time overhead caused by the middleware to the main application, we ran the test with and without including the proposed solution. As shown in Table 1, the middleware causes a maximum time overhead of about 0.16ms (15%) when configuring the transceiver. However, this only represents an overhead of 0.77% for the complete application.

5 CONCLUSION

In this poster, a new middleware for external peripheral state retention in transiently-powered systems has been proposed. The presented solution retains the configuration of SPI peripherals between power outages. Each peripheral instruction is saved in an instruction history table from where the approach restores the peripheral state. The proposed middleware was implemented and the experiments demonstrate that our middleware properly saves and restores the peripheral configuration causing a time overhead to the main application of up to 0.77%, which is substantially lower than that caused by existing approaches.

6 ACKNOWLEDGEMENTS

This work was supported by the Mexican CONACYT and the UK EPSRC under EP/P010164/1. Experimental data used in this work can be found at DOI:10.5258/SOTON/D0264 (http://doi.org/10.5258/SOTON/D0264).

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. *Embedded Systems Letters, IEEE*, 2015.

[2] G. Berthou, T. Delizy, K. Marquet, and T. Risset. Peripheral State Persistence For Transiently Powered Systems. Technical report, 2017.

[3] 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*. ACM Press, 2017.

[4] 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 *ENSsys '15*, pages 3–8, Seoul, Korea, 2015. ACM Press.