Transparent Standby for Low-Power, Resource-Constrained Embedded Systems

A Programming Language-Based Approach

Anonymous Author(s)

Abstract

This paper provides a sample of a LaTeX document which conforms, somewhat loosely, to the formatting guidelines for ACM SIG Proceedings.

CCS Concepts • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

Keywords ACM proceedings, LaTeX, text tagging

ACM Reference Format:
Anonymous Author(s). 1997. Transparent Standby for Low-Power, Resource-Constrained Embedded Systems: A Programming Language-Based Approach. In *Proceedings of ACM Woodstock conference (WOODSTOCK'97).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.475/123_4

1 Introduction

According to the International Energy Agency (IEA), the number of network-connected devices is expected to reach to 50 billion by 2020 with the expansion of the Internet of Things (IoT). [2] However, most of the energy to power these devices will be consumed in *standby mode*, i.e., when they are neither transmitting or processing data. For instance, standby power currently accounts for 10–15% of residential electricity consumption, and CO_2 emissions related to standby are equivalent to those of 1 million cars [2, 3]. The projected growth of IoT devices, together with the surprising effects of standby consumption, made network standby efficiency one of the six pillars of G20's *Energy Efficiency Action Plan*[1].

Given the projected scale of the IoT and the role of low-power standby towards energy efficiency, this paper has the following goals:

1. Address energy efficiency through extensive use of standby.
2. Target low-power, resource-constrained embedded architectures that form the IoT.
3. Provide standby mechanisms at the programming language level that scale to all applications.
4. Support transparent/non-intrusive standby mechanisms that reduce barriers of adoption.

Our approach lies at the bottom of the software development layers—transparent programming language mechanisms—meaning that *all* applications take advantage of low-power standby modes automatically, without extra programming efforts. We extend the synchronous programming language Céu [4, 5] with interrupt service routines (ISRs) and a simple power management runtime (PMR). Each supported microcontroller requires bindings in C for the ISRs and PMR, and each peripheral requires a driver in Céu. This is a one-time process and is typically packaged and distributed in a SDK. All new applications built on top of these drivers will take advantage of standby automatically. As a proof of concept, we support the 8-bit *AVR/ATmega* and 32-bit *ARM/Cortex-M0* microcontrollers, and a variety of peripherals (e.g., ADC, SPI, USART, nRF24L01 transceiver). The project is open-source.[2]

- automatic power management: standby as well as enabled/disabled, powered, switched
- application on IoT
- results - atmega 328p/2560 - arm cortex-m0 - how much efficiency?
- limitations
- paper structure

In Section 3, we introduce the structured synchronous model and the programming language Céu. In Section **??**, we present the language extensions that support transparent standby. In Section **??**, we evaluate .

2 The Structured Synchronous Programming Language Céu

Céu is a Esterel-based[4] reactive programming language targeting resource-constrained embedded systems [5]. It is grounded on the synchronous concurrency model, which has been successfully adopted in the context of hard real-time systems such as avionics and automobiles industry since the 80's [1]. The synchronous model trades power for

[1] G20's Energy Efficiency Action Plan: https://www.iea-4e.org/projects/g20

[2] https://github.com/fsantanna/ceu-arduino/

WOODSTOCK'97, July 1997, El Paso, Texas USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

```
digitalWrite(13, 0);
int old = 0;
while (1) {
    int cur = digitalRead(2);
    if (cur != old) {
        digitalWrite(13, cur);
        old = cur;
    }
}
```

[a] Active polling in C

```
input  int PIN_02;
output int PIN_13;

emit PIN_13(0);
loop do
    var int v = await PIN_02;
    emit PIN_13(v);
end
.
```

[b] Reactive execution in Céu

Figure 1. Controlling a LED from button state changes.

reliability and has a simpler model of time that suits most requirements of IoT applications. On the one hand, this model cannot directly express time-consuming computations, such as compression and cryptography algorithms, which are typically either absent or delegated to auxiliary chips in the context of the IoT. On the other hand, all reactions to the external world are guaranteed to be computed in bounded time, ensuring that applications always reach an idle state amenable to standby mode.

2.1 A Motivating Example

Figure 1 shows a basic program that makes a button connected to the microcontroller's pin 2 to control a LED connected to pin 13. Whenever the state of the button changes (i.e., pressed or unpressed), the state of the LED also changes (i.e., on or off).

The implementation in C in Figure 1.a first turns off the LED (ln. 1), then sets an initial state for the button (ln. 2), and then reads in a loop (ln. 3-9) its current state (ln. 4) as fast as possible. If the current state differs from the old state (ln. 5-8), the new state is copied to the LED (ln. 6), and is also set as the old state (ln. 7). This approach which actively samples the state of an external device (e.g., a button) is known as *polling* or *software-driven I/O*. On the one hand, this style does not involve dealing with the complexity of callbacks (as discussed in Section **??**) and is widely adopted in Arduino, the most popular embedded platform (to be discussed in Section **??**). On the other hand, polling wastes CPU cycles and prevents the device to enter in standby.

The implementation in Céu in Figure 1.b uses dedicated vocabulary to handle events from I/O devices without resorting to callbacks. The pins are first declared to determine their purpose in the program, i.e., `input` for the pin connected to the button and `output` for the pin connected to the LED (ln 1-2). Then, the LED is turned off with an `emit` (ln. 4), which indicates an output operation. Then, the program enters in a loop to react to changes in the button (ln. 5-8). The `await` primitive accounts for the reactive nature of Céu. The program literally waits for a change in the button (ln. 6) to be able to proceed and set the new state of the LED (ln 7). This way, when an application reaches an idle state, the language has precise information about which events can awake that application.

By design, all lines of execution in Céu always reach an `await` at the end of a reaction to an event (otherwise, the application does not compile [5]). This not only allows the application to enter standby mode, but effectively use the deepest sleeping level considering all possible awaking events. While in standby, only a hardware interrupt associated with the events can awake the application, making it sleep for longest possible periods of time.

3 Transparent Standby Mechanisms

References

[1] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages twelve years later. In *Proceedings of the IEEE*, Vol. 91. 64–83.

[2] OECD/IEA. 2014. *More Data Less Energy—Making Network Standby More Efficient in Billions of Connected Devices.* Technical Report. International Energy Agency.

[3] Australian Greenhouse Office. 2002. *Money isn't all you're saving: Australia's standby power strategy 2002–2012.* Australian Greenhouse Office. 23 pages.

[4] Francisco Sant'anna, Roberto Ierusalimschy, Noemi Rodriguez, Silvana Rossetto, and Adriano Branco. 2017. The Design and Implementation of the Synchronous Language CÉU. *ACM Trans. Embed. Comput. Syst.* 16, 4, Article 98 (July 2017), 26 pages. https://doi.org/10.1145/3035544

[5] Francisco Sant'Anna, Noemi Rodriguez, Roberto Ierusalimschy, Olaf Landsiedel, and Philippas Tsigas. 2013. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13.* ACM.