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

A Programming Language-Based Approach

Francisco Sant'Anna
francisco@ime.uerj.br
Rio de Janeiro State University, Brazil

Alexandre Sztajnberg
alexszt@ime.uerj.br
Rio de Janeiro State University, Brazil

Ana Lúcia de Moura
amoura@inf.puc-rio.br
PUC-Rio, Brazil

Noemi Rodrigues
noemi@inf.puc-rio.br
PUC-Rio, Brazil

Abstract

Standby efficiency for connected devices is one of the priorities of the *G20's Energy Efficiency Action Plan*. We propose transparent programming language mechanisms to enforce that applications remain in the deepest standby modes for the longest periods of time. We extend the programming language Céu with support for interrupt service routines and with a simple power management runtime. Based on these primitives, we also provide device drivers that allow applications to take advantage of standby automatically. Our approach relies on the synchronous semantics of the language which guarantees that reactions to the environment always reach an idle state amenable to standby. In addition, to lower the programming barrier of adoption, we show that programs in Céu can keep a sequential syntactic structure, even when applications require non-trivial concurrent behavior.

CCS Concepts • **Computer systems organization → Embedded software**; • **Software and its engineering →** *Runtime environments*;

Keywords Arduino, Concurrency, Embedded Systems, Esterel, IoT, Standby

ACM Reference Format:
Francisco Sant'Anna, Alexandre Sztajnberg, Ana Lúcia de Moura, and Noemi Rodrigues. 2018. WIP: Transparent Standby for Low-Power, Resource-Constrained Embedded Systems: A Programming Language-Based Approach. In *Proceedings of ACM SIGPLAN/SIGBED (LCTES'18).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.475/123_4

LCTES'18, June 2018, Philadelphia, USA

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

According to the International Energy Agency (IEA), the number of network-connected devices is expected to reach 50 billion by 2020 with the expansion of the Internet of Things (IoT) [6]. 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 approximately 10–15% of residential electricity consumption, and CO_2 emissions related to standby are equivalent to those of 1 million cars [6, 7]. The projected growth of IoT devices, together with the surprising effects of standby consumption, made network standby efficiency one of the six pillars of the *G20's Energy Efficiency Action Plan*[1]. However, making effective use of standby requires software-related efforts in order to detect idle periods of activity in a device, identify peripherals that must remain functional, and apply appropriate sleep mode levels in its microcontroller.

Given the projected scale of the IoT, the role of low-power standby towards energy efficiency, and the posed software-related challenges, our research has the following goals: (i) address energy efficiency through rigorous use of standby; (ii) target low-power, resource-constrained embedded architectures that form the IoT; (iii) provide standby mechanisms at the programming language level that scale to all applications; and (iv) support transparent/non-intrusive standby mechanisms that reduce barriers of adoption.

Our proposal lies at the bottom of the software development layers—programming language mechanisms—meaning that *all* applications should take advantage of low-power standby modes automatically, without extra programming efforts. We extend the programming language Céu [9, 10] with support for interrupt service routines (ISRs) and with a simple power management runtime (PMR). In contrast with other concurrency models (e.g., thread and actor based), the synchronous semantics of Céu guarantees that reactions to the environment always reach an idle state amenable to standby. Previous work [10] evaluates the expressiveness

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

of Céu in the context of Wireless Sensor Networks and discusses the development of drivers, network protocols, and full applications in the language. In addition, Céu incurs a small overhead of memory in comparison to C (around 5-10%), thus being a suitable alternative for constrained devices. The language runtime is in the order of a few kilobytes only (less than 5Kb).

In our approach, each supported microcontroller requires hooks in C for the ISRs and PMR, and each peripheral requires a driver in Céu. These are write-once code that is typically packaged and distributed in a software development kit (SDK). Then, all new applications built on top of these drivers take advantage of standby automatically. As a proof of concept, we provide an open source SDK with support for 8-bit *AVR/ATmega* and 32-bit *ARM/Cortex-M0* microcontrollers, and a variety of peripherals, such as for GPIO, A/D converter, USART, SPI, and the nRF24L01 transceiver. We developed a number of simple applications using these peripherals concurrently and could verify that the applications remain in the deepest standby modes for the longest periods of time.

In Section 1, we compare the structure of programs in Céu and Arduino [2], whose primary goal is to reduce the programming barrier of adoption for a non-technical audience (e.g., designers and artists). We show that we can keep the intended sequential reasoning of Arduino even when applications require non-trivial concurrent behavior. In Section 2, we discuss the software infrastructure that allows for unmodified programs in Céu to take advantage of standby automatically. In Section 4, we discuss future work and conclude the paper.

1 The Structured Synchronous Programming Language Céu

Céu is a Esterel-based [9] reactive programming language targeting resource-constrained embedded systems [10]. 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 [3]. The synchronous model trades power for 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 environment are guaranteed to be computed in bounded time [10], ensuring that applications always reach an idle state amenable to standby mode. Overall, Céu aims to offer a concurrent, safe, and expressive alternative to C with the characteristics that follow:

Reactive: code only executes in reactions to events and is idle most of the time.

```
1 while (1) {
2   delay(1000);
3   int v =
4     analogRead();
5   radioWrite(v);
6 }
```

[a] Version in Arduino

```
1 loop do
2   await 1s;
3   var int v =
4     await AnalogRead();
5   await RadioWrite(v);
6 end
```

[b] Version in Céu

Figure 1. Sequence of I/O operations running in a loop.

```
uint32_t prv =
  millis();
while (1) {
  if (radioAvail()) {
    break;
  }
  uint32_t cur =
    millis();
  if (cur>prv+1000) {
    prv = cur;
    int v =
      analogRead();
    radioWrite(v);
  }
}
```

[a] Version in Arduino

```
par/or do
  await RadioAvail();
with
  loop do
    await 1s;
    var int v =
      await AnalogRead();
    await RadioWrite(v);
  end
end
.
```

[b] Version in Céu

Figure 2. Achieving concurrency between I/O operations.

Structured: programs use structured control mechanisms, such as `await` (to suspend a line of execution), and `par` (to combine multiple lines of execution).

Synchronous: reactions run atomically and to completion on each line of execution, i.e., there's no implicit preemption or real parallelism.

Structured reactive programming lets developers write code in direct style, recovering from the inversion of control imposed by event-driven execution [1, 5, 8].

A Motivating Example

Figure 1.a shows a straightforward, easy-to-read code snippet in Arduino that executes forever in a loop a sequence of operations as follows: waits for 1 second (ln. 2), performs an A/D conversion (ln. 3–4), and broadcasts the read value (ln. 5). Figure 1.b shows the same code in Céu, with the noteworthy difference that operations that interact with the environment and take time use the `await` keyword. The traditional structured paradigm encouraged in Arduino (with blocks, loops, and sequences) allows for simple and readable code, avoiding the complexity of dealing with ISRs. However, the use of blocking operations, such as `delay(1000)` (ln. 2), prevents that other operations execute concurrently.

Suppose that we now want to immediately abort the loop in Figure 1.a at any time, as soon as a radio message arrives. Since the message might arrive concurrently with any of the blocking operations, we need to modify the structure of the program in Arduino. Figure 2.a replaces the blocking `delay` to the polling `millis`, which immediately returns the number of milliseconds since the reset. Now, we start by registering the current time (ln. 1–2) and, on each loop iteration, we recheck the time to see if one second has elapsed (ln. 7–9). Since these operations are non-blocking, we can intercalate their execution with checks for message arrivals (ln. 4–6). If the time is up, we start counting it again (ln. 10) before proceeding to the original operations in sequence (ln. 11–13). The original structured style in Figure 1.a has been drastically violated to accommodate concurrency in Figure 2.a. Furthermore, we only adapted the `delay` operation, but the other blocking operations (`analogRead` and `radioWrite`) would also need to be changed to achieve maximum concurrency. Alternatively, we could resort to ISRs or implement an event-driven scheduler to handle the operations [4], but ultimately, the program readability would still be compromised in the same way.

The program in Figure 2.b in Céu extends the one in Figure 1.b to accommodate concurrency. In contrast with the Arduino version, the original code in Céu remains unmodified (Figure 2.b, ln. 4–9) and concurrency is achieved through the `par/or` construct, which creates two lines of execution and terminates when either of them terminates, aborting the other automatically. This approach preserves the sequential, easy-to-read style while introducing concurrency seamlessly.

Standby Considerations

The structure of the program in Figure 2.b also indicates which peripherals are active at a given time. For instance, when the program is awaiting concurrently in lines 2 and 7, only the radio transceiver and A/D converter can awake the program. Hence, the language runtime can choose the most energy-efficient sleep mode that allows these two peripherals to awake the microcontroller from associated interrupts. Since the semantics of Céu enforces the program to always reach `await` statements in all active lines of execution, it is always possible to put the microcontroller into the optimal sleep mode after each reaction to the environment.

2 Standby Infrastructure

In order to empower the example in Figure 2.b with automatic standby, we have developed some extensions to Céu as follows:

- We made the runtime of Céu interrupt driven and put the microcontroller in standby after each reaction to the environment.
- We provided operations for the drivers to indicate which interrupts might awake the program.
- We included support for ISRs in Céu to generate input events to the program and awake the microcontroller.

```
1  // Exposed driver functionality
2
3  output void ADC_REQUEST; // low-level request
4  input int ADC_DONE;      // low-level response
5  code AnalogRead (void) -> int; // high-level abstraction
6
7  // Driver implementation
8
9  output void ADC_REQUEST do
10   {
11     ADMUX = 0x40 | (A0 & 0x07); // selects channel A0
12     bitSet(ADCSRA, ADIE);       // enables interrupt
13     bitSet(ADCSRA, ADSC);       // starts the conversion
14   }
15 end
16
17 async/isr {ADC_vect_num} do
18   { bitClear(ADCSRA, ADIE); } // disables interrupt
19   var int value = {ADC}; // reads register with the value
20   emit ADC_DONE(value);
21 end
22
23 code AnalogRead (void) -> int do
24   {PM_SET(PM_ADC, 1);}
25   do finalize with
26       {PM_SET(PM_ADC, 0);}
27   end
28
29   emit ADC_REQUEST;
30   var int value = await ADC_DONE;
31
32   escape value;
33 end
```

Figure 3. Céu driver for the *ATmega328p* A/D converter.

Figure 3 shows the driver for the A/D converter in Céu. This code is specific to the *ATmega328p* microcontroller and must be adapted to work in other platforms. For simplicity, we assume in the paper that the converter has a single channel to avoid having to deal with multiplexing.

The driver exposes raw I/O events (ln. 3–4) that will only deal with low-level port manipulation in the microcontroller. Output events are triggered with the `emit` keyword (ln. 29), while input events are captured with the `await` keyword (ln. 30). The output event `ADC_REQUEST` actual implementation (ln. 9–15) enables ADC interrupts and starts an analog-to-digital conversion asynchronously in the peripheral for the single channel `A0`. In Céu, any code in between `{` and `}` is treated as an inline *C* chunk, allowing for easy integration with *C* for low-level operations.

The `async/isr` construct of Céu defines an ISR which executes asynchronously with the program when the specified interrupt occurs. Only ISRs can emit input events to the program. In the example, we define an ISR to handle `ADC` interrupts which fire whenever a conversion is complete (ln. 17–21). Although the ISR body executes asynchronously on interrupts, the input emission (ln. 20) only takes effect on a subsequent reaction, when the synchronous part of the program becomes idle. This way, race conditions are only possible with `async/isr` blocks, which are typically hidden inside device drivers. Céu also provides an `atomic` primitive to protect critical sections of code.

The low-level events are the pieces that vary among platforms. A driver can also expose a higher-level portable abstraction to client code. In the example, the `AnalogRead` abstraction (ln. 23–33) takes care of starting and awaiting the conversion (ln. 29–30), as well as dealing with the power management runtime (PMR). The `PM_SET(PM_ADC,1)` (ln. 24) tells the system that, when entering in sleep mode, the `ADC` must be kept running. The `PM_SET(PM_ADC,0)` inside the `finalize` clause (ln. 25–27) releases the `ADC` subsystem from the PMR.

The `finalize` construct of Céu executes the nested code whenever its enclosing block terminates or is aborted externally. The example of Figure 2.b invokes the `AnalogRead` abstraction (ln. 7) concurrently with `RadioAvail` (ln. 2). The `AnalogRead` may terminate normally or a radio message may arrive during the A/D conversion, causing the `AnalogRead` to abort abruptly. In either case, the `finalize` clause executes and puts the PMR in a consistent state.

The PMR also expects a platform-specific power management module to be able to put the microcontroller into the most efficient sleep mode possible. The code in Figure 4 implements the `pm_sleep` function for the *ATmega328p* microcontroller which the PMR calls when the program becomes idle. Each device has an associated index (ln. 6–10) in the `pm` bit vector (ln. 4). The driver manipulates its device's index to indicate its state (Figure 3, ln. 24,26). The `pm_sleep` queries the vector to choose the appropriate sleep mode. In the example, if the timer is active (ln. 13), the microcontroller can only use the least efficient mode[2] (ln. 14). In the best case, e.g., if only external interrupts are required, the microcontroller can use the most efficient mode (ln. 18).

With all the standby infrastructure set, the unmodified program of Figure 2.b will automatically take advantage of the deepest sleep modes for the longest periods of time possible.

```
1  #define PM_GET(dev)   bitRead(pm,dev)
2  #define PM_SET(dev,v) bitWrite(pm,dev,v)
3
4  static u32 pm = 0; // up to 32  peripherals
5
6  enum {
7    CEU_PM_ADC = 0,
8    CEU_PM_TIMER1,
9    <...>,
10 };
11
12 void pm_sleep (void) {
13   if (PM_GET(PM_TIMER1) || <...>) {
14       LowPower.idle(PM_GET(PM_ADC),<...>)
15     } else if (PM_GET(PM_ADC)) {
16       LowPower.adcNoiseReduction(<...>);
17     } else {
18       LowPower.powerDown(<...>);
19     }
20   }
21 }
```

Figure 4. Power management module for the *ATmega328p* microcontroller.

3 Discussion

The application of Figure 2.b relies solely on the driver of Figure 3 to achieve transparent standby. In Céu, the burden to deal with standby is transferred to the device drivers, which are write-once code written by specialists and distributed with an SDK. By transferring the work from the applications to the language level, novice or domain programmers never have to deal with standby explicitly. In contrast, general-purpose languages typically provide low-power libraries to deal with standby. However, programmers still have to call these libraries explicitly, characterizing a mechanism that is manual and error prone.

In Figure 2.b, when introducing concurrency, the structure of the program remains sequential and amenable to inference of the appropriate sleep mode. In comparison with Arduino, whose main goal is to lower the entry barrier for embedded development, Céu also preserves the sequential structure for concurrent applications.

The synchronous model of Céu provides logical parallelism to enable proper separation of concerns, while avoiding the hassle of explicit synchronization primitives (e.g., locks and mutexes). Yet, asynchronous interrupts provide real-time responsiveness for time-sensitive operations closer to the hardware.

[2] We use an external library for the sleep modes: http://www.rocketscream.com/blog/2011/07/04/lightweight-low-power-arduino-library/

4 Conclusion and Future Work

In this work, we address standby efficiency for embedded devices at the level of programming languages. We propose a software infrastructure for the programming language Céu that encompasses a power management runtime and support for interrupt service routines in the language. Our approach relies on the synchronous semantics of the language which guarantees that reactions to the environment always reach an idle state amenable to standby. This way, application written in Céu can take advantage of the longest periods of time and deepest sleep modes possible without extra programming efforts.

In future work, in order to evaluate the gains in energy efficiency with the proposed infrastructure, we will evaluate the consumption of realistic applications. The Arduino community has an abundance of open-source projects which can be rewritten in Céu to take advantage of transparent standby. In this scenario, we can evaluate the time to rewrite, the resulting program structure, and the actual energy efficiency.

Acknowledgments

The authors would like to thank Guilherme Simas for the initial explorations with the language extensions and development of some of the device drivers.

References

[1] A. Adya et al. 2002. Cooperative Task Management Without Manual Stack Management. In *Proceedings of ATEC'02*. USENIX Association, 289–302.

[2] Massimo Banzi and Michael Shiloh. 2014. *Getting started with Arduino: the open source electronics prototyping platform*. Maker Media, Inc.

[3] 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.

[4] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of PLDI'03*. 1–11.

[5] Ingo Maier, Tiark Rompf, and Martin Odersky. 2010. *Deprecating the observer pattern*. Technical Report.

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

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

[8] Guido Salvaneschi et al. 2014. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of Modularity'13*. ACM, 25–36.

[9] 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

[10] 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.