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

A Programming Language-Based Approach

Anonymous Author(s)

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 deepest standby modes for longest periods of time. We extend the synchronous programming language Céu with support for interrupt service routines and with a simple power management runtime. We developed device drivers based on these primitives on top of which applications can be built to take advantage of standby automatically. We also show that programs in Céu can keep a sequential structure to lower the barrier of adoption, even when applications require non-trivial concurrent behaviors.

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, 3 pages. 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) [5]. 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 [5, 6]. 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].

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—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 [8, 9] with support for interrupt service routines (ISRs) and with 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. These are a one-time procedures and are 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 GPIO, A/D converter, USART, SPI, and the nRF24L01 transceiver.

We developed a number of applications using these peripherals concurrently and could verify that the applications remain in deepest standby modes for longest periods of time. We also compare the structure of programs in Céu and Arduino [2], whose primary goal is to reduce the barrier of adoption for non-technical users (e.g., designers and artists). We show that we can keep the intended sequential reasoning of Arduino even when applications require non-trivial concurrent behaviors.

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

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

```
while (1) {
  delay(1000);
  int v =
    analogRead();
  radioWrite(v);
}
```

[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.

1 The Structured Synchronous Programming Language Céu

Céu is a Esterel-based[8] reactive programming language targeting resource-constrained embedded systems [9]. 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 world are guaranteed to be computed in bounded time, 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.
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, 4, 7].

A Motivating Example

Figure 1.a shows a simple, easy-to-read program chunk 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 value read (ln. 5). Figure 1.b shows the same chunk in Céu, with a 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)`, prevents that other operations execute concurrently.

```
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

```
1  par/or do
2    await RadioAvail();
3  with
4    loop do
5      await 1s;
6      var int v =
7        await AnalogRead();
8      await RadioWrite(v);
9    end
10 end
11
12
13
14
15 .
```

[b] Version in Céu

Figure 2. Achieving concurrency between I/O operations.

Suppose we now want that, at any time, receiving a message via radio should immediately abort the loop in Figure 1.a. Since the message might arrive concurrently with any of the blocking operations, we need to change the structure of the program. Figure 2.a changes the blocking operation `delay` to the polling operation `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 the 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 has been drastically violated to accommodate concurrency. In the example, 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 [**?**], but ultimately, the program readability would still be compromised.

The program in Figure 2.b in Céu extends the one in Figure 1.b to accommodate concurrency. The original code remains unmodified (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 accommodating 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 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 Transparent Standby Mechanisms

In order to empower the example in Figure 2.b with automatic standby, we had to make some modifications and extensions to Céu:

- Modify the runtime of Céu to be interrupt driven and to put the microcontroller in standby after each reaction to the environment.
- Provide operations for the drivers to indicate which interrupts can awake the program.
- Support ISRs in Céu to generate input events to the program.

Figure 3 shows the driver for .

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

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

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

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

[8] 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

[9] 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.


```
1  // Exposed driver functionality
2
3  output void ANALOG_REQUEST;      // low-level request
4  input  int  ANALOG_DONE;         // low-level response
5  code AnalogRead (void) -> int;   // high-level abstraction
6
7  // Driver implementation
8
9  output void ANALOG_REQUEST do
10   <...>   // port manipulation to start the conversion
11 end
12
13 async/isr ADC_vect_num do
14   var int value = <...>; // register with the value read
15   emit ANALOG_DONE(value);
16 end
17
18 code AnalogRead (void) -> int do
19   {PM_SET(PM_ANALOG, 1);}
20   do finalize with
21       {PM_SET(PM_ANALOG, 0);}
22   end
23
24   emit ANALOG_REQUEST(pin);
25   var int value = await ANALOG_DONE;
26
27   escape value;
28 end
```

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

```
#define PM_GET(peripheral) \
        bitRead(pm, peripheral)
#define PM_SET(peripheral,state) \
        bitWrite(pm, peripheral, state)

static u32 pm = 0;

void pm_sleep (void) {
  if (PM_GET(PM_TIMER1)  PM_GET(PM_USART)  PM_GET(PM_SPI))
      LowPower.idle(PM_GET(PM_ADC),...)
    } else if (PM_GET(PM_ADC)) {
      LowPower.adcNoiseReduction(...);
    } else {
      LowPower.powerDown(...);
    }
  }
}
```

Figure 4. Power management module for the ATmega328p microcontroller.