

WIP: 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 the deepest standby modes for the longest periods of time. We extend the programming language C  U 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, we show that programs in C  U can keep a sequential structure, even when applications require non-trivial concurrent behavior, to lower the barrier of adoption.

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

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

ACM Reference Format:

Anonymous Author(s). 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, 4 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) [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%

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

LCTES'18, June 2018, Philadelphia, USA

   2018 Copyright held by the owner/author(s).

ACM ISBN 123-4567-24-567/08/06.

https://doi.org/10.475/123_4

of residential electricity consumption, and CO₂ 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 the device, identify peripherals that must remain functional, and apply appropriate sleep mode levels in the 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 meticulous 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 approach 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 C  U [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 the synchronous semantics of C  U guarantees that reactions to the environment always reach an idle state amenable to standby.

Each supported microcontroller requires bindings in C for the ISRs and PMR, and each peripheral requires a driver in C  U. These are write-once code that 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 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.

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

<pre> 111 while (1) { 112 delay(1000); 113 int v = 114 analogRead(); 115 radioWrite(v); 116 } </pre>	<pre> 1 loop do 2 await 1s; 3 var int v = 4 await AnalogRead(); 5 await RadioWrite(v); 6 end </pre>
[a] Version in Arduino	[b] Version in CéU

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

In Section 1, we compare the structure of programs in CéU and Arduino [2], whose primary goal is to reduce the 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 3, 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, 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.

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 change the structure of the program. Figure 2.a changes 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. 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 [4], 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. 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 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

```

221 uint32_t prv = 1 par/or do
222   millis(); 2 await RadioAvail();
223 while (1) { 3 with
224   if (radioAvail()) { 4 loop do
225     break; 5 await 1s;
226   } 6 var int v =
227   uint32_t cur = 7 await AnalogRead();
228   millis(); 8 await RadioWrite(v);
229   if (cur>prv+1000) { 9 end
230     prv = cur; 10 end
231     int v = 11
232     analogRead(); 12
233     radioWrite(v); 13
234   } 14
235 } 15 .

```

[a] Version in Arduino

[b] Version in C   

Figure 2. Achieving concurrency between I/O operations.

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 developed some extensions to C   :

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

Figure 3 shows the driver for the A/D converter in C   . 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 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` (ln. 9–15) enables ADC interrupts and starts an analog-to-digital conversion asynchronously in the peripheral for the single channel `A0`. In C   , 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 C    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 is idle. This way, race conditions are only possible with `async/isr` blocks, which are typically hidden inside device drivers. C    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 C    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 is 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 mode² (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.

3 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 C    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

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

```

331 1 // Exposed driver functionality
332 2
333 3 output void ADC_REQUEST; // low-level request
334 4 input int ADC_DONE; // low-level response
335 5 code AnalogRead (void) -> int; // high-level abstraction
336 6
337 7 // Driver implementation
338 8
339 9 output void ADC_REQUEST do
340 10 {
341 11     ADMUX = 0x40 | (A0 & 0x07);
342 12     bitSet(ADCSRA, ADIE); // enables interrupt
343 13     bitSet(ADCSRA, ADSC); // starts the conversion
344 14 }
345 15 end
346 16
347 17 async/isr {ADC_vect_num} do
348 18 { bitClear(ADCSRA, ADIE); } // disables interrupt
349 19 var int value = {ADC}; // reads register with the value
350 20 emit ADC_DONE(value);
351 21 end
352 22
353 23 code AnalogRead (void) -> int do
354 24 {PM_SET(PM_ADC, 1);}
355 25 do finalize with
356 26 {PM_SET(PM_ADC, 0);}
357 27 end
358 28
359 29 emit ADC_REQUEST;
360 30 var int value = await ADC_DONE;
361 31
362 32 escape value;
363 33 end

```

Figure 3. C  U driver for the ATmega328p A/D converter.

an idle state amenable to standby. This way, application written in C  U 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 C  U to take advantage of transparent standby. In this scenario, we can evaluate the time to rewrite, the resulting program structure, and the actual energy consumption efficiency.

References

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

```

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.

- [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 Ierusalimsky, Noemi Rodriguez, Silvana Rossetto, and Adriano Branco. 2017. The Design and Implementation of the Synchronous Language C  U. *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 Ierusalimsky, Olaf Landsiedel, and Philippas Tsigas. 2013. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM.