Where Do Events Come From?

Reactive and Energy-Efficient Programming From The Ground Up

Anonymous Author(s)

Abstract

In reactive and event-based systems, execution is guided by an external environment that generates inputs to the application and consumes outputs from it. Reactive languages provide dedicated syntax and semantics to deal with events and greatly simplify the programming experience in this domain. Nevertheless, the environment is typically prefabricated in a host language and the very central concept of events is implemented externally to the reactive language. In this work, we propose an interrupt handler primitive for a reactive language targeting embedded systems in order to take control of the whole event loop: from a sensor input source and back to an actuator output. We propose the new asynchronous primitive in the context of the synchronous language CÉU and discuss how they synergize to avoid race conditions at compile time, support lexically-scoped drivers, and provide automatic standby for applications.

Keywords interrupt service routines, reactive programming, sleep modes, standby, synchronous/asynchronous execution

1 Introduction

Reactive applications interact continuously and in real time with the external world through hardware peripherals such as sensors and actuators (e.g., buttons, displays, timers, etc.). These interactions are typically represented in software as input events flowing from the peripherals to the application and as output events flowing from the application to the peripherals. As illustrated in Figure 1, peripherals can be encapsulated as a single component, *the environment*, which connects with the application in an event loop: the application sits idle until the environment awakes it on the occurrence of an input; the application reacts to the input and generates back one or more outputs which affect the environment; the application becomes idle and the loop restarts.

The environment is typically implemented in a host language (e.g., C) and controls the main event loop, invoking entry points into the reactive language runtime on the occurrence of inputs, and also receiving output calls from it, both through a documented API. As examples, Esterel [3] relies on C for passing events between the environment and the running program [10], while Elm [5] uses the concept of *ports*, which allows sending out values to JavaScript as commands and listening for values as subscriptions [4].

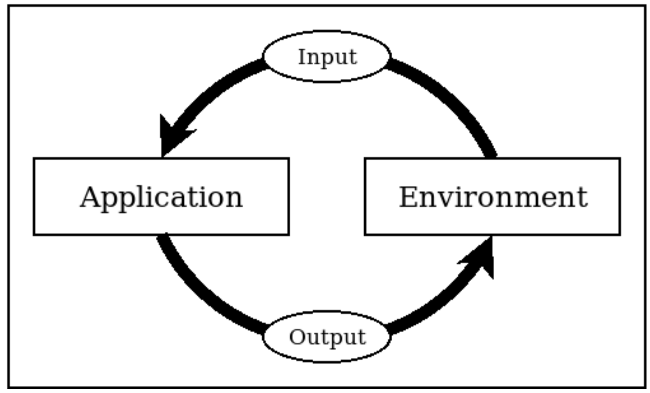

Figure 1. Event loop in reactive systems. The environment controls the application through input & output events.

The event-based interface between the application and the environment is arguably inevitable and also happens at the appropriate layer, since it connects the application with the operating system resources through a non-invasive API. However, this separation reveals the environment as a rigid system component that evolves in separate from the application. It also requires two languages and the programmer has to deal with multiple syntaxes, incompatible type systems, and different address spaces. Furthermore, in the context of embedded systems, a proper host OS may even be absent or lacking enough device drivers, which requires more low-level intervention from the application.

In this work, we propose interrupt service routines (ISRs) as an asynchronous primitive for the synchronous language CÉU [12]. ISRs empower reactive applications to also implement their own device drivers and self-generate inputs, bypassing the need for a host environment. CÉU targets resource-constrained architectures, such as Arduino-compatible microcontrollers, in which applications run in the bare metal, without operating system support. In this context, device drivers are typically libraries compiled together with the main program.

The lack of an OS opens an opportunity to explore a tighter integration between the application and its device drivers at the language level, resulting in an overall simplicity and tractability of the system. In particular,

REBLS'18, November 04, 2018, Boston, MA, USA
2018.

Céu is already amenable to a simple static analysis to detect race conditions which we extended in this work to also encompass ISRs. Applications can share data buffers with drivers to avoid unnecessary copying, with some static guarantee that no data races will occur. Céu also provides a lexical finalization mechanism that we adopt in drivers to properly disable interrupts and turn off peripherals completely. For instance, we can delimit the scope of a driver similarly to a lexically-scoped variables. Finally, the synchronous semantics of Céu enforces that applications react to inputs in bounded time and remain in idle states susceptible to standby. With the help of drivers and a power manager, we can put the microcontroller to sleep automatically at optimal sleeping modes after each input reaction.

We implemented an extensible runtime that exposes hooks for the interrupts and power manager, which we validated in two microcontrollers: an *8-bit AVR/ATmega* and a *32-bit ARM/Cortex-M0*. We also implemented drivers for a variety of peripherals, such as GPIO, A/D converter, USART, SPI, and nRF24L01 transceiver. The applications built on top of these drivers show significant energy savings due to automatic standby.

Our work is largely inspired by TinyOS [7], a low-power OS for wireless devices, which also provides asynchronous events with a compile-time data race detector [6], and automatic energy management triggered from idle CPU states [8]. We adapted these ideas to the structured reactive programming model of Céu, and refined them with stronger guarantees due to lexical scope and automatic finalization.

2 Céu: Structured Synchronous Reactive Programming

Céu [12] is a Esterel-based [13] reactive programming language targeting resource-constrained embedded systems, with the characteristics that follow:

- **Reactive:** code only executes in reactions to events and is idle most of the time.
- **Structured:** programs use lexically-scoped structured control mechanisms such as `spawn` and `await` (to create and suspend 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, 9, 11].

Listing 1 illustrates the main characteristics of Céu, namely event-driven I/O, lexically-scoped compositions, and synchronous execution. The program toggles the state of the LED whenever a radio packet is received, terminating on a button press, always with the LED off.

```
 1 output high/low LED;
 2 input  high/low BUTTON;
 3 input  Packet   RADIO_RECV;
 4 par/or do
 5     var high/low v = await BUTTON until (v==low);
 6 with
 7     finalize with
 8         emit LED(low);
 9     end
10     loop do
11         emit LED(high);
12         await RADIO_RECV;
13         emit LED(low);
14         await RADIO_RECV;
15     end
16 end
```

Listing 1. A Céu program that toggles the state of the LED whenever a radio packet is received, terminating on a button press, always with the LED off.

The program first declares the `LED` output, and the `BUTTON` and `RADIO_RECV` input events (ln 1–3). The declarations include a payload type, i.e., each event occurrence carries a value of that type (`high/low` is a boolean type). Then, the program uses a `par/or` composition (ln 4–16) to run two lines of execution, aka *trails*, in parallel: a single-statement trail that waits for a button press before terminating (ln 5), and an endless loop that toggles the LED on and off whenever a radio packet is received (ln 10–15). The `finalize` clause (ln 7–9) ensures that, no matter how its enclosing trail terminates (e.g., from a button press), the LED will be unconditionally turned off (ln 8).

All communication between the application and the environment is done through the `await` and `emit` primitives, which awaits an input event and generates an output event, respectively.

The `par/or`, which stands for *parallel-or*, is a lexical composition that terminates as soon as one of the trails terminates, but which also automatically aborts and finalizes the other trail(s). In the example, when the button is pressed (ln 5), not only the toggling loop will be aborted (ln 10–15), but the `finalize` clause will turn the LED off unconditionally (ln 8), since its enclosing block goes out of scope. The `par/or` is regarded as an *orthogonal preemption primitive* [2] because the trails need not to be tweaked in order to affect each other.

The synchronous execution model of Céu dictates that reactions to input events are atomic and that incoming events are never lost. The event loop in Figure 1 suggests this behavior since the environment can only generate a new input after the application yields control

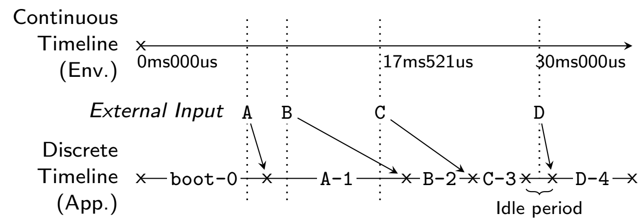

Figure 2. The discrete notion of time in Céu.

and closes the loop. In Listing 1, even if two packets arrive simultaneously in the environment, the synchronous model ensures atomicity and responsiveness in the application: the first `await` awakes and turns the LED off atomically (ln 12–13); and the second `await` will awake in sequence (ln 14).

Figure 2 illustrates the synchronous execution model of Céu. The continuous timeline (representing the environment) shows "physical timestamps" for the event occurrences (e.g., event `C` occurs exactly at 17ms521us). The discrete timeline in the application shows how the same occurring events fit in the logical notion of time of Céu. The boot reaction `boot-0` happens before any input and starts the program. Event `A` "physically" occurs during `boot-0` but, because time is discrete, its corresponding reaction only executes at logical instant `A-1`. Similarly, event `B` occurs during `A-1` and its reaction is postponed to execute at `B-2`. Event `C` also occurs during `A-1` but its reaction must also wait for `B-2` to execute at `C-3`. Finally, event `D` occurs during an idle period and can start immediately at `D-4`.

In order to guarantee responsiveness, the synchronous model relies on the *synchronous hypothesis*, which states that reactions must be significantly faster than the rate of inputs. For this reason, Céu (like most synchronous languages) refuses unbounded loops at compile time. This restriction guarantees that all reactions to the environment are computed in bounded time [14], ensuring that applications are always responsive to upcoming events.

2.1 Lexically-Scoped Interrupt Service Routines

Interrupts service routines (ISRs) are software entry points that execute in response to hardware interrupts from peripherals such as timers and GPIOs. ISRs are at the lowest interface layer between the hardware and software and are the absolute source of inputs to programs. An ISR starts to execute as soon as the hardware interrupt occurs, suspending the ongoing program flow abruptly. Such asynchronous behavior reflects the inherent concurrent nature of peripherals interacting with the external world.

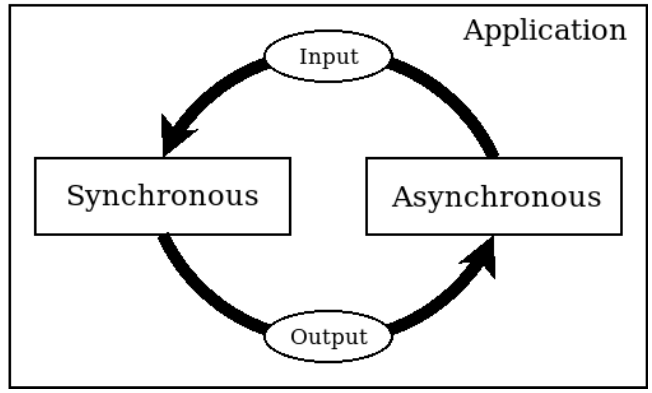

Figure 3. An application in Céu has a concurrent asynchronous side and a predictable synchronous side that receives and reacts to inputs atomically.

```
1 output high/low PIN_13;  // connected to an LED
2 input  high/low PIN_02;  // connected to a button
3 #include "gpio.ceu"      // GPIO driver
4
5 emit PIN_13(low);
6 loop do
7     var high/low v = await PIN_02;
8     emit PIN_13(v);
9 end
```

Listing 2. Synchronous application that turns the LED on whenever the button is pressed and off whenever it is unpressed.

We propose to extend Céu with asynchronous ISRs. However, asynchronous execution confronts the synchronous mindset of Céu since the assumption that reactions are atomic no longer holds. Not only this may lead to race conditions at a fine grain, but may also affect the ordering of events at a coarse grain: the effect of an earlier event may be perceived after the effect of a later event. Still, our goal is to preserve the well-behaved interaction between the Céu application and the environment even in the presence of asynchronous ISRs. Figure 3 adapts the event loop of Figure 1, with the original application represented as the synchronous side and the environment as the asynchronous side, all inside the same Céu application. Our idea is to push all subtleties of asynchronous execution into device drivers, which emit input events to regular synchronous code in Céu, exactly as before.

As a first example, Listing 2 and 3 uses GPIOs to connect a button (pin 02) to an LED (pin 13) via software such that the LED is on whenever the button is pressed and off whenever it is unpressed.

Listing 2, the synchronous side, first declares its interface with the external world and includes the asynchronous side as a driver (ln 1–3). It then starts with the

```
1 // OUTPUT DRIVER
2
3 { pinMode(13, OUTPUT); }
4 output (high/low v) PIN_13 do
5     { digitalWrite(13, @v); }
6 end
7
8 // INPUT DRIVER
9
10 input high/low PIN_02;
11 {
12     pinMode(2, INPUT_PULLUP);
13     EICRA |= (1 << ISC00);  // sets INT0
14     EIMSK |= (1 << INT0);   // turns on INT0
15 }
16 spawn async/isr [INT0_vect] do
17     emit PIN_02({digitalRead(2)});
18 end
```

Listing 3. Asynchronous driver for GPIO which, on hardware interrupts, emits an input event into a queue.

LED off (ln 5) and enters a loop (ln 6–9) that, whenever the button changes (ln 7), toggles the state of the LED with the new value (ln 8).

Listing 3, the asynchronous side, implements the driver for the output and input events. An `output` implementation (ln 4–6) is similar to a parameterized subroutine: whenever the application invokes `emit`, the output body executes atomically. Céu supports inline C between curly braces (ln 3,5) with interpolation to evaluate Céu expressions (e.g., `@v`). This allows drivers to take advantage of existing libraries of embedded toolchains, such as Arduino. In the example, when the driver is included, it configures pin 13 as output (ln 3) and sets its new state whenever `PIN_13` is emitted (ln 5). An `input` event implementation uses an ISR registered with the `spawn async/isr` primitive (ln 16–18), which is automatically invoked whenever the associated interrupt occurs (e.g., `INT0_vect`). An ISR in Céu will typically emit an input event inside its body to awake the synchronous side (ln 17). However, although the ISR executes asynchronously when the interrupt occurs, the `emit` goes into a queue (to be discussed) and does not interrupt the ongoing (suspended) reaction on the synchronous side. In the example, the input driver also configures pin 2 to behave as input and to generate external interrupts on level transitions (ln 11–15).

The example illustrates the clear separation between the application and the driver through an event-driven interface that reflects the desired architecture of Figures 1 and 3. Now, the whole application is written in Céu: the synchronous side remains well behaved with no low-level calls, and the asynchronous side deals with the complexity of device drivers. Nevertheless, drivers are

```
1 // A/D DRIVER (adc.ceu)
2
3   output int ADC_REQUEST;
4   input  int ADC_DONE;
5
6   do finalize with
7     {
8       ADCSRA &= B01111111;  // disables A/D converter
9       ACSR    = B10000000;  // disables comparator
10      DIDR0  |= B00111111;  // disables pins
11    }
12  end
13  ...      // driver initialization
14  ...      // input / output implementations
15
16 // APPLICATION (main.ceu)
17
18  loop do
19    var int v;
20    do
21      #include "adc.ceu" // driver contents above
22      emit ADC_REQUEST(A0);
23      v = await ADC_DONE;
24    end
25    ...                    // uses sensor value "v"
26    await 1h;
27  end
```

Listing 4. The A/D driver (ln 1–14) is included in the application inside a lexically-scoped block (ln 20–24), which turns off all driver functionalities automatically on termination.

typically write-once components developed by embedded specialists, which are reused in most applications. Note that each supported architecture requires a mapping between the `async/isr` and the actual interrupt vector table (which we provide for the *AVR/ATmega* and *ARM/Cortex* microcontrollers).

The example in Listing 4 illustrates the use of a lexically-scoped driver for the A/D converter. Note that the driver (ln 1–14) and application (ln 16–27) are actually in separate files. The application is a typical periodic sensor sampling loop, but which designates an explicit `do-end` block (ln 20–24) to include the driver (ln 21) and use it (ln 22–23). The driver specifies a finalization code (ln 6–12) which executes unconditionally whenever its enclosing block goes out of scope (ln 24). The finalize completely disables all A/D functionality to save energy. As discussed in the previous section, the finalizer would also execute if aborted from a `par/or` enclosing the driver. Note that the driver interface (ln 3–4) is visible only inside the block and thus cannot be inadvertently used outside it.

```
1  input void A;
2  input void B;
3  var int x = 1;
4  par/and do
5      await A;
6      x = x + 1;
7  with
8      await B;
9      x = x * 2;
10 end
```

Listing 5. Shared `x`

```
1  input void A;
2  // (empty line)
3  var int y = 1;
4  par/and do
5      await A;
6      y = y + 1;
7  with
8      await A;
9      y = y * 2;
10 end
```

Listing 6. Shared `y`

Figure 4. Accesses to shared `x` never concurrent. Accesses to shared `y` are concurrent but still deterministic.

,

2.2 Safe Shared-Memory Concurrency

In Céu, when multiple trails awake in the same reaction, they execute in lexical order, i.e., in the order they appear in the source code. In Figure 4, both examples define a shared variable (ln 3) and assign to it in parallel trails (ln 6, 9). In Listing 5, the two assignments to `x` can only execute in reactions to different events `A` and `B` (ln 5,8), which cannot occur simultaneously in accordance with the synchronous model. In Listing 6, the two assignments to `y` are simultaneous because they execute in reaction to the same event `A`. Nevertheless, because Céu follows lexical order, the execution is still deterministic, and `y` always becomes `4` (`(1+1)*2`). Even so, Céu performs (optional) concurrency checks at compile time to detect conflicting accesses to shared variables: if a variable is written in a trail segment, then a concurrent trail segment cannot access that variable [12].

The addition of asynchronous ISRs poses real threats concerning race conditions since they interrupt programs at arbitrary points. Listing 7 illustrates a minimum USART application (ln 13–22) that consumes incoming bytes (ln 18–21) as they arrive (ln 16). The USART API (ln 1–4) exposes an input event to signal incoming data (ln 3) and a circular buffer to prevent data loss (ln 4). The buffer is indispensable because the microcontroller may cope with the USART speed. The driver ISR (ln 6–11) simply appends incoming bytes to the end of the buffer (ln 9) and signals the application (ln 10). As a possible race condition, note that the ISR may fire and append a new byte to the buffer (ln 9) just before the application clears it (ln 21), in which case the new byte will be lost forever.

Céu treats an `async/isr` as a block of code that runs in parallel with the rest of the program (hence the required prefix `spawn`). This way, it is clear for the compiler that the accesses to `rx_buf` in Listing 7 may lead to a

```
1  // USART INTERFACE
2
3  input none USART_RX;
4  var[32*] byte rx_buf;              // '*' = circular
5
6  // DRIVER
7
8  spawn async/isr [USART_RX_vect] do
9      rx_buf = rx_buf .. [{UDR0}];
10     emit USART_RX;
11 end
12
13 // APPLICATION
14
15 loop do
16     await USART_RX;
17     var int i;
18     loop i in [0 -> $rx_buf[ do    // '$' = length ($)
19         // uses rx_buf[i]
20     end
21     $rx_buf = 0;
22 end
```

Listing 7. The USART buffer is shared between the ISR and the application, resulting in a pontential race condition. *(The driver and application code are in separate files.)*

race condition (ln 9,21), and Céu raises a compile-time error. The programmer is responsible to protect concurrent memory accesses with `atomic` blocks, which disables interrupts (supposedly) for a short period of time.

However, we do not expect that application programmers should be required to resolve race conditions. Céu supports reactive abstractions (similar to coroutines) that help hiding the concurrency complexity inside the driver. Listing 8 changes the USART API (ln 1–3) to expose a code abstraction that expects a reference to a buffer and a number of bytes to receive. Now, the application (ln 29–35) invokes the abstraction (ln 33) passing a local buffer (ln 32). The driver (ln 5–27) now hides the low level interface of the previous example (ln 7–8) and implements the code abstraction (ln 15–27) that protects the access to the shared buffer with an `atomic` block (ln 18–21). The abstraction copies the driver buffer into the application buffer (ln 19) up to the requested number of bytes (ln 22). On the one hand the extra copying incurs a memory and runtime overhead, but on the other hand it prevents race conditions with the ISR.

Support for ISRs in the same language and memory space of the application allows programmers to choose between efficiency and safety during development by providing multiple interfaces with different levels of abstraction.

```
 1 // USART INTERFACE
 2
 3 code Usart_RX (var&[] byte buf, var int n) -> none;
 4
 5 // DRIVER
 6
 7 input none USART_RX;
 8 var[32*] byte rx_buf;
 9
10 spawn async/isr [USART_RX_vect] do
11     rx_buf = rx_buf .. [{UDR0}];
12     emit USART_RX;
13 end
14
15 code Usart_RX (var&[] byte buf, var int n) -> none
16 do
17     loop do
18         atomic do
19             buf = buf..rx_buf;
20             $rx_buf = 0;
21         end
22         if $buf >= n then
23             break;
24         end
25         await USART_RX;
26     end
27 end
28
29 // APPLICATION
30
31 loop do
32     var[255] byte buf;
33     await Usart_RX(&buf, 10);
34     // uses buf
35 end
```

Listing 8. The USART driver now exposes a safer (and more friendly) interface to the application. *(The driver and application code are in separate files.)*

2.3 Energy-Efficient Runtime

Our runtime implements the event loop proposed in Figure 3, which alternates between synchronous, well-behaved execution, and asynchronous, unpredictable ISRs. Since the language now has full control over the event loop and only executes from interrupts, it is possible for the runtime to enter in sleep mode automatically to save energy.

Listing 9 is a realization of this architecture: The runtime defines an input queue (ln 1) in which ISRs enqueue new events (e.g., Listing 8, ln 12). In the `main` function, we first generate the *boot reaction* (ln 3), which starts the program, spawns the ISRs, and reaches the first `await` statements in the multiple lines of code. Then, the runtime enters an infinite loop (ln 4–11) that queries the input queue (ln 6) to awake the program (ln 7). Note

```
 1 evt_t queue[MAX];           // input queue
 2 void main () {
 3   ceu_start();              // "boot reaction"
 4   while (1) {
 5     evt_t evt;
 6     if (ceu_input(&evt)) {  // query input queue
 7       ceu_sync(&evt);       // execute synchronous
 8     } else {
 9       ceu_pm_sleep();       // nothing to execute
10     }
11   }
12 }
```

Listing 9. Overall runtime architecture of Céu with an input queue (ln 1), event loop (ln 4–11), synchronous execution (ln 7), and standby mode (ln 9).

that ISRs execute asynchronously with the loop, but they typically only emit an input event into the queue which will be queried in a subsequent iteration. Note also that if the queue is empty, the runtime enters in sleep mode to save energy (ln 9), and will only awake on a new hardware interrupt.

Microcontrollers typically support multiple levels of sleep mode, each progressively saves more energy at the expense of keeping less functionality active. As an example, the least efficient mode of the *ATmega328P* allows for timer interrupts since it keeps its internal clock active, while the most efficient mode turns off all peripherals and can only awake from external interrupts.

Our runtime supports three compile-time configurations for `ceu_pm_sleep` (ln 9): The first configuration is a *nop* which simply keeps the event loop running all the time without ever sleeping. This configuration is useful when introducing new platforms. The second configuration chooses a (inefficient) sleep mode that keeps all peripherals active. Although this configuration is not the most energy efficient, at least, it requires no extra assistance from the drivers. The third configuration keeps a bit vector of the active drivers during runtime and chooses the most efficient mode, querying this vector every time `ceu_pm_sleep` is called.

The third configuration achieves optimal efficiency but requires a tight interaction between the drivers and the power management runtime. Listing 10 unveils the power manager (ln 1–18), the modified USART driver (ln 20–29), and an illustrative application (ln 31–43). The application is a loop that awaits in parallel for a button press (ln 36) or receiving 10 bytes (ln 39). Let's call this initial state `STATE-A`. The `par/or` terminates when either of them occurs, going to the next line that awaits another button click (ln 42). Let's call this other state `STATE-B`. After the button click, the program loops back to `STATE-A`. While in `STATE-A`, the program can awake from USART

```
1 // POWER MANAGER (in C)
2
3 enum {
4     CEU_PM_ADC,
5     CEU_PM_TIMER1,
6     CEU_PM_USART,
7     ...
8 };
9
10 void ceu_pm_sleep (void) {
11     if (ceu_pm_get(CEU_PM_USART) || ...) {
12         sleep_mode_1(...);
13     } else if (ceu_pm_get(CEU_PM_ADC)) {
14         sleep_mode_2(...);
15     } else {
16         sleep_mode_3(...);
17     }
18 }
19
20 // USART DRIVER (in Ceu)
21
22 code Usart_RX (var&[] byte buf, var int n) -> none
23 do
24     {ceu_pm_set(CEU_PM_USART, 1);}
25     do finalize with
26         {ceu_pm_set(CEU_PM_USART, 0);}
27     end
28     ...   // same as in Listing 7
29 end
30
31 // APPLICATION (in Ceu)
32
33 input high/low PIN_02;      // connected to a button
34 loop do
35     par/or do
36         await PIN_02;
37     with
38         var[255] byte buf;
39         await Usart_RX(&buf, 10);
40         // uses buf
41     end
42     await PIN_02;
43 end
```

Listing 10. Interaction between the power manager, USART driver, and application (each code is actually in a separate file).

and external interrupts, which means that the power manager should choose a sleep mode in which the USART remains operational. While in `STATE-B`, only external interrupts should awake the program, which means that the power manager may use the most efficient sleep mode. The power manager enumerates all microcontroller's peripherals (ln 3–8) and, before sleeping, queries their states (ln 11,13) to choose the most appropriate sleep mode (ln 12,14,16). The USART driver needs to be extended with calls to enable and disable the USART state inside the power manager: just before awaiting, the driver enables the USART (ln 24) and creates a finalization block to unregister it on termination (ln 25–27). Termination may occur either directly after receiving the requested number of bytes, or indirectly if the `par/or` in the application terminates on a button click (ln 36).

Note that applications need not to be explicit about energy management to take advantage of sleep modes. All happens automatically because of the synchronous-reactive execution model and energy-aware runtime of Céu.

TODO: energy savings

3 Related Work

Our work is largely inspired by TinyOS [7] and its companion language nesC [6], which is a callback-driven dialect of C. By default, callbacks are synchronous but can be annotated as `async` when triggered directly from interrupts. In this case, a static data-race detector ensures that concurrent accesses are protected by atomic sections. Since the TinyOS runtime is also based on an event loop, it can easily detect idle periods and put the CPU to sleep. Device drivers also cooperate with the runtime to designate the best sleep mode [8].

In comparison to nesC, our proposal takes advantage of the structured mechanisms of Céu to offer even stronger guarantees. For instance, nesC cannot provide a safe abortion mechanism, since the notion of control flow is entirely dynamic, typically encoded inside ordinary global variables that represent state machines. For this reason, some manual bookkeeping is required to avoid that resources eventually leak (e.g., memory or energy). As an example, if the result of a long chain of split-phase I/O is no longer required due to a cancellation event, the cancellation callback is itself responsible for aborting the ongoing unnecessary operations. Céu relies on the `par/or` and `finalize` structured mechanisms to manage resources automatically.

- CRP

- finalize - structured programming - lexical scoping vs manual - no turn off

- citar ICEM o mais cedo nas duas ocasioes (async, energy) - citar TOS/ICEM na introducao como maior influencia - ja no abstract?

- falar que async/isr nao contem nada de Ceu, apenas shared memory e nocao de estar em paralelo - compartilhamento de memoria - velocidade - ajuda do compilador p/ detectar acesso concorrente

- finite is can be still much greater than 0 (order of milliseconds) - which is a lot of time

- artigo synchronous languages

- shim

- electre

References

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

[2] Gérard Berry. 1993. Preemption in Concurrent Systems.. In *FSTTCS (LNCS)*, Vol. 761. Springer, 72–93.

[3] Frédéric Boussinot and Robert De Simone. 1991. The Esterel language. *Proc. IEEE* 79, 9 (Sep 1991), 1293–1304.

[4] Evan Czaplicki. 2018. Elm and JavaScript Interop. https://guide.elm-lang.org/interop/javascript.html (accessed in Jul-2018).

[5] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs.. In *Proceedings of PLDI'13*. 411–422.

[6] 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.

[7] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. 2000. System architecture directions for networked sensors. *SIGPLAN Notices* 35 (November 2000), 93–104. Issue 11.

[8] Kevin Klues, Vlado Handziski, Chenyang Lu, Adam Wolisz, David Culler, David Gay, and Philip Levis. 2007. Integrating Concurrency Control and Energy Management in Device Drivers. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 251–264. https://doi.org/10.1145/1294261.1294286

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

[10] Dumitru Potop-Butucaru, Stephen A Edwards, and Gerard Berry. 2007. *Compiling esterel*. Vol. 86. Springer Science & Business Media.

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

[12] Francisco Sant'Anna et al. 2013. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM.

[13] Francisco Sant'anna et al. 2017. The Design and Implementation of the Synchronous Language Céu. *ACM TECS* 16, 4, Article 98 (July 2017), 26 pages. https://doi.org/10.1145/3035544

[14] Rodrigo C. M. Santos et al. 2018. A Memory-Bounded, Deterministic and Terminating Semantics for the Synchronous Programming Language CéU. In *Proceedings of LCTES'18*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/3211332.3211334