

# 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 inputs to outputs. 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, 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. Peripherals can be abstracted in a single component, *the environment*, which connects with the application through an event loop: the application sits idle until the environment awakes it on an input; the application reacts and generates back outputs which affect the environment; the application becomes idle and the loop restarts.

Reactive languages provide dedicated syntax and semantics to deal with events and simplify the development of applications in this domain. However, the environment is still 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.

The event-based interface between the reactive language and the environment is arguably inevitable, but 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, 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 reactive language CÉU [7]. ISRs empower reactive applications with their own device drivers which self-generate inputs, bypassing the need for a host environment. CÉU targets OS-less embedded architectures, such as Arduino-compatible microcontrollers. Applications in CÉU 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 to properly disable interrupts and turn off peripherals completely. 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, the microcontroller sleeps automatically at optimal sleeping modes after each input reaction. We implemented drivers for a variety of peripherals, such as GPIO, A/D converter, USART, and nRF24L01 transceiver.

Our work is largely inspired by TinyOS [4], a low-power OS for sensor networks, which also provides asynchronous events with data race detection [3], and automatic energy management [5]. We adapted these ideas to the structured reactive model of CÉU, and refined them with stronger guarantees due to its support for lexically-scoped lines of execution with automatic finalization.

### 2 Céu: Structured Synchronous Reactive Programming

CÉU [7] is a Esterel-based [2] 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 [6].

---

```

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.

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. 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

---

```

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.

block goes out of scope. The **par/or** is regarded as an *orthogonal preemption primitive* [1] 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 environment can only generate a new input after the application yields control and closes the event 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).

## 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.

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. 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.

---

```

221 1 // OUTPUT DRIVER
222 2
223 3 { pinMode(13, OUTPUT); }
224 4 output (high/low v) PIN_13 do
225 5     { digitalWrite(13, @v); }
226 6 end
227 7
228 8 // INPUT DRIVER
229 9
230 10 input high/low PIN_02;
231 11 {
232 12     pinMode(2, INPUT_PULLUP);
233 13     EICRA |= (1 << ISC00); // sets INTO
234 14     EIMSK |= (1 << INTO); // turns on INTO
235 15 }
236 16 spawn async/isr [INT0_vect] do
237 17     emit PIN_02({digitalRead(2)});
238 18 end

```

---

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

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 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).

---

```

276 1 // A/D DRIVER (adc.ceu)
277 2
278 3 output int ADC_REQUEST;
279 4 input int ADC_DONE;
280 5
281 6 do finalize with
282 7 {
283 8     ADCSRA &= B01111111; // disables A/D converter
284 9     ACSR = B10000000; // disables comparator
285 10    DIDRO |= B00111111; // disables pins
286 11 }
287 12 end
288 13 ... // driver initialization
289 14 ... // input / output implementations
290 15
291 16 // APPLICATION (main.ceu)
292 17
293 18 loop do
294 19     var int v;
295 20     do
296 21         #include "adc.ceu" // driver contents above
297 22         emit ADC_REQUEST(A0);
298 23         v = await ADC_DONE;
299 24     end
300 25     ... // uses sensor value "v"
301 26     await 1h;
302 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.

The example illustrates the clear separation between the application and the driver through an event-driven interface. 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 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.

```

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

```

Listing 5. Shared x Listing 6. Shared y

**Figure 1.** Accesses to shared  $x$  never concurrent. Accesses to shared  $y$  are concurrent but still deterministic.

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.

## 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 1, 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 [7].

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

```

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 potential race condition. (*The driver and application code are in separate files.*)

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 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.



---

```

441 1 // USART INTERFACE
442 2
443 3 code Usart_RX (var&[] byte buf, var int n) -> none;
444 4
445 5 // DRIVER
446 6
447 7 input none USART_RX;
448 8 var[32*] byte rx_buf;
449 9
450 10 spawn async/isr [USART_RX_vect] do
451 11     rx_buf = rx_buf .. [{UDRO}];
452 12     emit USART_RX;
453 13 end
454 14
455 15 code Usart_RX (var&[] byte buf, var int n) -> none
456 16 do
457 17     loop do
458 18         atomic do
459 19             buf = buf..rx_buf;
460 20             $rx_buf = 0;
461 21         end
462 22         if $buf >= n then
463 23             break;
464 24         end
465 25         await USART_RX;
466 26     end
467 27 end
468 28
469 29 // APPLICATION
470 30
471 31 loop do
472 32     var[255] byte buf;
473 33     await Usart_RX(&buf, 10);
474 34     // uses buf
475 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.*)

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.

### 2.3 Energy-Efficient Runtime

Our runtime 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`

---

```

496 1 evt_t queue[MAX];           // input queue
497 2 void main () {
498 3     ceu_start();             // "boot reaction"
499 4     while (1) {
500 5         evt_t evt;
501 6         if (ceu_input(&evt)) { // query input queue
502 7             ceu_sync(&evt);    // execute synchronous
503 8         } else {
504 9             ceu_pm_sleep();    // nothing to execute
505 10        }
506 11    }
507 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).

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 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

---

```

551 1 // POWER MANAGER (in C)
552 2
553 3 enum {
554 4     CEU_PM_ADC,
555 5     CEU_PM_TIMER1,
556 6     CEU_PM_USART,
557 7     ...
558 8 };
559 9
559 10 void ceu_pm_sleep (void) {
560 11     if (ceu_pm_get(CEU_PM_USART) || ...) {
561 12         sleep_mode_1(...);
562 13     } else if (ceu_pm_get(CEU_PM_ADC)) {
563 14         sleep_mode_2(...);
564 15     } else {
565 16         sleep_mode_3(...);
566 17     }
567 18 }
568 19
568 20 // USART DRIVER (in Ceu)
569 21
570 22 code Usart_RX (var&[] byte buf, var int n) -> none
571 23 do
572 24     {ceu_pm_set(CEU_PM_USART, 1);}
573 25 do finalize with
574 26     {ceu_pm_set(CEU_PM_USART, 0);}
575 27 end
576 28 ... // same as in Listing 7
577 29 end
578 30
578 31 // APPLICATION (in Ceu)
579 32
580 33 input high/low PIN_02; // connected to a button
581 34 loop do
582 35     par/or do
583 36         await PIN_02;
584 37     with
585 38         var[255] byte buf;
586 39         await Usart_RX(&buf, 10);
587 40         // uses buf
588 41     end
589 42     await PIN_02;
590 43 end

```

---

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

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 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.

### 3 Conclusion

We extend the synchronous language CÉU with asynchronous interrupt service routines (ISRs) to take control of the whole event loop in reactive systems, from inputs to outputs. ISRs empower reactive applications to also implement their own device drivers and self-generate inputs, bypassing the need for a host environment. However, asynchronous execution confronts the well-behaved semantics of synchronous languages with possible race conditions. To mitigate this threat, we extend the static concurrency checks of CÉU to detect data races between ISRs and the application.

Our approach for writing drivers relies on the lexical structured mechanisms of CÉU, such as parallel compositions and abortion of lines of execution, to offer stronger guarantees. For instance, the visibility of drivers can be delimited to scoped blocks to avoid resource leaks after usage. The same mechanism allows drivers to signal the power manager that associated peripherals are no longer required, allowing applications to use optimal sleep modes to save energy automatically. Our initial tests show optimal energy savings for applications in idle states.

### References

- [1] Gérard Berry. 1993. Preemption in Concurrent Systems.. In *FSTTCS (LNCS)*, Vol. 761. Springer, 72–93.
- [2] Frédéric Boussinot and Robert De Simone. 1991. The Esterel language. *Proc. IEEE* 79, 9 (Sep 1991), 1293–1304.
- [3] David Gay et al. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of PLDI'03*. 1–11.
- [4] Jason Hill et al. 2000. System architecture directions for networked sensors. *SIGPLAN Notices* 35 (November 2000), 93–104. Issue 11.
- [5] Kevin Klues et al. 2007. Integrating Concurrency Control and Energy Management in Device Drivers. In *Proceedings of*

661	<i>SOSP'07</i> . ACM, New York, NY, USA, 251–264.	716
662	[6] Ingo Maier, Tiark Rompf, and Martin Odersky. 2010. <i>Depre-</i>	717
663	<i>cating the observer pattern</i> . Technical Report.	718
664	[7] Francisco Sant’Anna et al. 2013. Safe System-level Concurrency	719
665	on Resource-Constrained Nodes. In <i>Proceedings of SenSys’13</i> .	720
666	ACM.	721
667		722
668		723
669		724
670		725
671		726
672		727
673		728
674		729
675		730
676		731
677		732
678		733
679		734
680		735
681		736
682		737
683		738
684		739
685		740
686		741
687		742
688		743
689		744
690		745
691		746
692		747
693		748
694		749
695		750
696		751
697		752
698		753
699		754
700		755
701		756
702		757
703		758
704		759
705		760
706		761
707		762
708		763
709		764
710		765
711		766
712		767
713		768
714		769
715		770