Universidade do Estado do Rio de Janeiro

Centro de Tecnologia e Ciências

Faculdade de Engenharia

Anny Caroline Correa Chagas

Evaluation of transparent energy-saving mechanisms in embedded applications

Rio de Janeiro

2021

Anny Caroline Correa Chagas

Evaluation of transparent energy-saving mechanisms in embedded applications

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre em Ciências, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Redes de Telecomunicações.

Orientador: Prof. D.Sc. Francisco Sant'Anna

Rio de Janeiro

2021

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

C433 Chagas, Anny Caroline Correa.
Evaluation of transparent energy-saving mechanisms in embedded application / Anny Caroline Correa Chagas. – 2021.
69f.

Orientador: Francisco Figueiredo Goytacaz Sant'Anna.
Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

1. Engenharia eletrônica - Teses. 2. Sistemas embarcados (Computadores) - Teses. 3. Arduino (Controlador programável) - Teses. 4. Computadores digitais - Programação - Teses. I. Sant’Anna, Francisco Figueiredo Goytacaz. II. Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia. III. Título.

CDU 004.031.6

Bibliotecária: Júlia Vieira – CRB7/6022

Anny Caroline Correa Chagas

Evaluation of transparent energy-saving mechanisms in embedded applications

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre em Ciências, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Redes de Telecomunicações.

Aprovado em: 12 de novembro de 2021

Rio de Janeiro

2021

AGRADECIMENTO

Gostaria de agradecer ao meu orientador prof. Francisco Sant'Anna por todo o incentivo, ensinamentos e oportunidades. Agradeço também ao prof. Alexandre Sztajnberg pela minha passagem no Laboratório de Ciência da Computação (LCC/IME) e sua orientação durante o bacharelado.

RESUMO

CHAGAS, Anny Caroline Correa. *Avaliação de mecanismos transparentes de economia de energia em aplicações embarcadas.* 69 f. Dissertação (Mestrado em Engenharia Eletrônica) - Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, 2021.

Durante o desenvolvimento de um dispositivo embarcado, a tarefa de gerenciar os modos de economia de energia é normalmente delegada aos desenvolvedores das aplicações, principalmente quando o dispositivo possui recursos limitados e não há um sistema operacional. Essa abordagem acaba por tornar a aplicação difícil de ler e escrever e exigir um grande conhecimento sobre os hardwares utilizados. Em contrapartida, o gerenciamento de energia de dispositivos embarcados pode reduzir significativamente seu consumo e prolongar a vida útil de baterias. Nesse contexto, avaliamos o uso dos mecanismos transparentes de gerenciamento de energia da linguagem de programação Céu no desenvolvimento de aplicações embarcadas com recursos limitados. A semântica síncrona de Céu garante que reações ao ambiente sempre alcancem um estado ocioso, no qual a linguagem pode aplicar o modo de economia mais eficiente possível para cada hardware utilizado. A fim de avaliar a viabilidade de uso dessa linguagem, comparamos implementações em Céu e em Arduino de duas aplicações típicas: um sistema de iluminação inteligente e um dispositivo de coleta de dados de sensores. Para apoiar a implementação das aplicações em Céu, desenvolvemos drivers cientes de energia para as classes de sensores digitais e analógicos, além de um driver específico para o sensor de temperatura e umidade DHT11. Em ambas as aplicações as implementações em Céu se mostraram mais eficientes em relação ao consumo de energia em pelo menos 30%, com a penalidade do aumento de uso de memória. O aumento no uso de memória se mostrou significativo em uma das aplicações e indica uma limitação para a adoção da linguagem neste contexto. Em contrapartida, as implementações em Céu apresentaram uma melhor legibilidade.

Palavras-chave: Sistemas Embarcados. Economia de energia. Arduino. Programação reativa.

ABSTRACT

CHAGAS, Anny Caroline Correa. *Evaluation of transparent energy-saving mechanisms in embedded applications.* 69 f. Dissertação (Mestrado em Engenharia Eletrônica) - Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, 2021.

During the development of an embedded device, the task of managing power-saving modes is usually delegated to the application developers, especially when a device has limited resources and there isn't an operating system. This approach produces applications that are harder to read and write and requires developers to know more about the hardware used. In contrast, the power management of embedded devices can significantly reduce power consumption and extend battery life. In this context, we propose the use of transparent energy saving mechanisms of Céu programming language in the development of resource-constrained embedded applications. Its synchronous semantics ensures that reactions to the environment always reach an idle state, in which the language can apply the most efficient power mode for each hardware used. In order to evaluate the use of this language, we compared implementations in Céu and Arduino of two applications: an smart lighting system and a sensor data collection device. To support the implementation of the Céu applications, we have developed energy-aware drivers for digital and analog sensors, as well as a specific driver for the DHT11 temperature and humidity sensor. In both applications, the implementations in Céu proved to be more efficient in terms of energy consumption by at least 30%, with a penalty on increased memory usage. The increase in memory usage was significant in one of the applications and indicates a limitation for the adoption of the language in this context. In contrast, implementations in Céu showed better readability.

Palavras-chave: Embedded systems. Energy saving. Arduino. Reactive programming.

LIST OF FIGURES

Figure 1 Turning hardware on and off via MCU 14
Figure 2 Mini Ultra 8 MHz by Rocket Scream 17
Figure 3 Céu program interacting with real world thought an environment. Based at: https://ceu-lang.github.io/ceu/out/manual/v0.30 18
Figure 4 Circuit of the application that blinks an LED every second until a interrupt is received 21
Figure 5 Measuring current using a multimeter 30
Figure 6 Multimeter range settings 31
Figure 7 Pliers ammeter 31
Figure 8 Oscilloscope 32
Figure 9 Current waveform analyzer 32
Figure 10 DC power supplies 33
Figure 11 Arduino based current meter 33
Figure 12 Memory usage information provided by the Arduino compiler 35
Figure 13 d-sensor application circuit 41
Figure 14 DHT11 humidity and temperature sensor in a breakout board connected to the Mini Ultra prototyping board 44
Figure 15 Start of the communication between a microcontroller and a DHT11 sensor 45
Figure 16 DHT sends data to microcontroller 45
Figure 17 Simplified schematic of TX device 48
Figure 18 RX application states 52
Figure 19 Simplified schematic of RX device 52
Figure 20 Hardware used in the second application 59
Figure 21 Circuit of humidity and moisture sensor device 60

LIST OF TABLES

Table 1 ATmega328/p sleep modes and their main wake-up sources 13
Table 2 HC-05 power consumption 47
Table 3 Power consumption of the smart lighting TX application 50
Table 4 Memory usage of the smart lighting TX application 51
Table 5 Memory usage of the smart lighting RX application 57
Table 6 Power consumption of the smart lighting RX application 57
Table 7 Power consumption of the humidity and moisture sensor node 62
Table 8 Memory usage of the humidity and moisture sensor node 62

SUMMARY

INTRODUCTION 10

1 SAVING ENERGY ON EMBEDDED DEVICES 12

1.1 Hardware power saving mechanisms 12

1.2 Turning hardware off when not in use 14

1.3 Other mechanisms 15

1.4 Energy savings in embedded Operating Systems 15

1.5 Prototyping boards 16

1.6 Power saving techniques used in this dissertation 17

2 THE PROGRAMMING LANGUAGE CÉU 18

2.1 Interrupts and drivers 19

2.2 Power management mechanism 23

3 METHODOLOGY 25

3.1 Choosing the applications 25

3.2 Applications implementations 26

3.3 Power consumption analysis 27

3.3.1 Power comparison 29

3.3.2 Measuring instruments 29

3.3.3 Instruments used in this work 34

3.4 Memory consumption analysis 34

3.5 Readability comparison 35

3.5.1 Related works 36

3.5.2 The chosen approach 37

4 DEVELOPED DRIVERS 40

4.1 Driver d-sensor 40

4.2 Driver a-sensor 43

4.3 Driver dht 11 44

5 **COMPARISON OF APPLICATIONS IN CÉU AND ARDUINO** 47
5.1 **Smart lighting** 47
5.1.1 TX application 48
5.1.2 RX application 51
5.2 **Humidity and moisture sensor device** 59
CONCLUSÃO 64
REFERÊNCIAS 65

INTRODUCTION

Embedded software is characterized by running on devices that are not typically known as computers (e.g. microwaves, watches, cars, toys, etc.). Unlike general purpose computers, in embedded devices both software and hardware are designed to perform a specific predefined task. Because of this, it's possible to choose more specific hardware, minimizing costs and allowing the device to have a smaller physical size [1].

Although these devices are often battery-powered, most embedded programming languages and operating systems don't provide automatic energy saving mechanisms. In order to improve energy efficiency, application developers have to implement saving mechanisms manually. This task is often complicated and error-prone, and produces a software that is harder to read and write. However, it's indispensable for many projects, as it can make an application execute for a longer period of time using the same battery [2] [3] [4].

For embedded devices connected to power grid, the power consumption is usually not such a concern. In fact, they don't consume as much energy as other elements connected to the power grid (e.g. home appliances, heating system etc.). However, if we analyze it globally and consider that the number of these devices tends to grow in the coming years, the energy and environmental impact becomes considerable [5].

Regardless of how they are powered, embedded devices tend to remain idle for long periods of the time. Ideally, these devices should enter the deepest standby mode during idle periods, avoiding wasting energy. Nevertheless, identifying idle periods and inferring the most efficient power saving mode is not always simple. On connected devices (as the ones that use radios), energy efficiency during idle periods tends to be even more complicated, as more hardware resources typically remain on. These reasons have made standby for connected devices one of the six pillars of the G20 Energy Efficiency Action Plan [6].

In this context, the present work proposes to evaluate the usage of the programming language Céu [7] in the development of embedded applications. The language has been developed during the last 12 years and already has a stable implementation, being successfully adopted in the areas of wireless networks [8] [9], games [10] [11] and multimedia [12]. Due to its synchronous model, any application written in Céu always reaches an idle state susceptible to energy savings. With this, the language can identify which parts of the hardware should remain active in each idle period, and apply the best energy saving mode automatically. Previous works already presented their power-manage mechanism and propose their evaluation [13] [4].

Objectives

The main objective of this work is to evaluate the energy consumption of realistic applications written in Céu and compare them with implementations in Arduino, as suggested in a previous work [4]. Code readability and memory usage were also evaluated to discuss language adoption barriers. This dissertation also discusses how to measure the energy consumption of embedded devices and compare the readability of applications written in two different programming languages. The developed applications are:

- An smart lighting system consisting of two devices, a transmitter and a receiver, that communicate via Bluetooth. To assist its implementation, we developed the d-sensor driver, which allows a digital sensor to be turned on only when in use, improving the energy efficiency of the device.

- A device that collects air humidity and soil moisture, and transmits this information via radio using the nRF24l01+ transceiver. We developed the driver dht11 to retrieve air humidity from the sensor DHT11, and the driver a-sensor to read the soil moisture from an analog sensor. Analogous to the d-sensor driver, the a-sensor turns off the sensor when not in use.

Dissertation organization

The rest of the dissertation is organized as follows: Chapter 1 introduces different techniques to save energy on embedded devices, emphasizing automatic use of standby in SO-less resource-constrained embedded architectures. Chapter 2 introduces the programming language Céu and its power management runtime. Chapter 3 presents the methodology used in this work and discusses how to measure the embedded devices energy consumption and how to compare readability. Chapter 4 details the developed drivers and Chapter 5 compares applications written in Céu and Arduino, remarking code readability, energy consumption and memory usage.

1 SAVING ENERGY ON EMBEDDED DEVICES

This chapter discusses the main energy saving techniques used in embedded devices, and is organized as follows. Section 1.1 presents power saving mechanisms provided by different hardware, such as sensors, actuators, microcontrollers (MCUs) and radio transceivers. Section 1.2 explores the technique of completely turning off part of the device's hardware when not in use. Section 1.3 presents other saving techniques, such as adjusting the quality of service (QoS) and disabling application functionalities. It also discusses network protocols that aim to save power.

Section 1.4 comments about how some embedded operating systems (OSs) deal with energy savings. Section 1.5 discusses the usage of prototyping boards and its impact on power consumption. Finally, Section 1.6 elucidates the techniques used in this dissertation and finalizes the chapter.

1.1 Hardware power saving mechanisms

Saving energy on embedded devices requires both software and hardware efforts. When choosing which hardware (sensor, actuators, microcontroller, etc.) to use, developers should analyze their consumption and if they have power saving mechanisms.

Some hardware control their own power saving modes. An example is the DHT11 humidity and temperature sensor, which remains sleeping until it receives a signal requesting a measurement [14]. The radio transceiver nRF24l01+ has a similar behavior when acting as a transmitter. It remains in a sleep mode until it's time to transmit a message [15].

Instead of managing their own energy-saving states, some hardware delegates this task to the application. The gy-521 module is a good example. It's a breakout board that encapsulates the integrated circuit MPU-6050 and features a gyroscope and accelerometer [16]. This module has two power saving modes: in the most economical one, the MPU-6050 is sleeping and almost completely turned off. Only the I2C hardware remains on, as it's responsible for the module's communication with the MCU where the application is running. This way, the application can wake up the module even when it's sleeping. In its least economical mode (a.k.a cycle mode), the MPU-6050 wakes up from time to time to perform an accelerometer measurement [17]. The application can choose to use one of these modes by manipulating MPU-6050 registers.

MCUs are also a good example of hardware that has its power consumption managed by the application. In general, an MCU provides sleep modes. To choose each sleep mode to use, the developer should consider which resources should remain on during sleep, and its wake-up sources, i.e., which events can wake up the microcontroller. Some common events are external interrupts and timers.

The ATmega328/P is a MCU widely used in the Arduino community. It has six sleep modes as shown in Table 1. Among them, "power down" is the most economical mode, while "idle" is the most energy consuming. As an example, consider an application that waits a presence detection to turn on an LED. In this case, the ATmega328/P can enter into the deepest sleep mode while waiting [1]. On the other hand, if the application is waiting to receive a message via SPI interface instead of a presence detection, the microcontroller can't go into "power down" mode, because the SPI circuit is disabled in it. Alternatively, the developer can choose the least economical mode, the "idle" mode.

	Wake-up source							
Sleep mode	INT and PCINT	TWI Address Match	Timer 1	Timer 2	SPI	SPM/EEPROM Ready	ADC	Watch dog timer
Idle	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ADC noise reduction	Yes	Yes		Yes		Yes	Yes	Yes
Power-down	Yes	Yes						Yes
Power-save	Yes	Yes		Yes				Yes
Standby	Yes	Yes						Yes
Extended standby	Yes	Yes		Yes				Yes

Table 1: ATmega328/p sleep modes and their main wake-up sources

As the microcontroller usually consumes a lot of power when it's active (i.e. when not in a sleep mode), developers often prefer to keep sensors and peripheral hardware active and wake up the microcontroller using them. Finally, it's worth mentioning some other characteristics that can influence the power consumption of a hardware.

- **Operating voltage**: embedded hardware usually works with different voltages (usually 3.3V and/or 5V). In order to save energy, it's better to choose a lower voltage.

- **Clock speed**: in general, the lower the clock, the lower the consumption. In this way, making MCUs run at a lower clock rate can save energy.

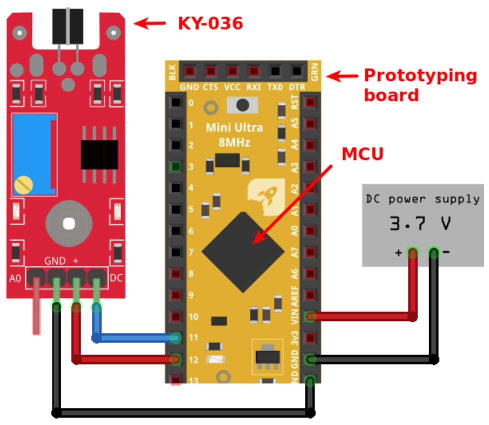
[1]For this to be possible, the presence detection sensor needs to be connected to the MCU through an interrupt pin. This way, when a presence is detected, an interrupt (INT or PCINT) is generated and the microcontroller can be woken up.

- **Radio configurations**: The size of the transmission radius is directly proportional to its power consumption, as is the number of retransmission attempts.

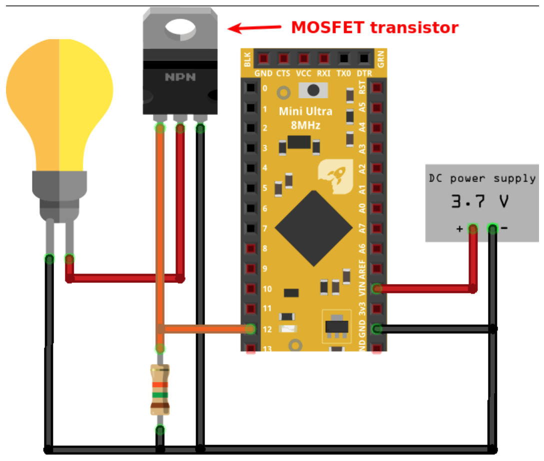
1.2 Turning hardware off when not in use

Instead of using a power saving mode, there is the possibility of completely turning off a hardware while it's not in use. This is especially interesting when the hardware doesn't have any power saving modes. However, it's important to consider how long the application should wait before using the hardware after powering it on.

One way to implement this technique is to connect the hardware VCC to a MCU digital output pin. In Figure 1(a), the ky-036 touch sensor is connected to pin 11 of the MCU[2]. In this way, the application running on the MCU can turn on or off the hardware using this pin.



(a) ky-036 touch sensor powered by a MCU digital output pin



(b) Using a MOSFET transistor to turn on a DC lamp that consumes a lot of current

Figure 1: Turning hardware on and off via MCU

The MCU pins have a limitation on the current they can supply, so not all hardware can be powered this way. To control hardware that consumes more current, a MOSFET transistor can be used as shown in Figure 1(b). However, most sensors used in embedded software do not consume as much current, and the MOSFET ends up being more used to control motors, LED strips and DC lamps.

[2]A prototyping board was used to facilitate the use of the MCU. The board used was the Mini Ultra 8MHz, compatible with the Arduino platform.

1.3 Other mechanisms

There are other saving techniques that do not aim at using sleep modes in idle periods, but by adjusting the quality of service (QoS) [18] of the application or its functionalities to meet the demands of battery levels, for example, disabling certain features [19]. There are also network protocols aimed at energy saving, such as LoraWAN and Bluetooth Low Energy [20]. However, these savings are restricted to parts related to the network, not extending to sensors and actuators, for example.

1.4 Energy savings in embedded Operating Systems

When the software is responsible for applying energy saving techniques, this task is usually performed manually by the application itself. It's known as an error-prone task, which affects the readability of the application and makes it difficult to maintain. However, there is the possibility of using an embedded operating system. The OS can facilitate the power management, or even provide automatic saving mechanisms. On the downside, they often require more hardware resources. The following list comments about some embedded OSs, and their main energy saving mechanisms.

- µCLINUX (and other UNIX based systems) doesn't provide a native API to power management, but some of their drivers provide some power management functions [21].

- TinyOS is commonly used in wireless sensor networks and already has an automatic power mechanism. Among the related works, this is the one that most resembles the mechanism offered by the Céu language. However, TinyOS uses a complex callback-based programming model. This complexity inspired previous works to compare applications written in Céu and nesC, the language used in TinyOS. As a result, Céu used 50% fewer lines of code, with a small increase in memory usage [22].

- Unlike TinyOS, Contiki does not offer automatic energy saving: applications are responsible for saving energy by observing the size of the event queue. When there are no events scheduled, the processor can sleep until an interrupt wakes it up [23]. It also provides power consumption estimates, which can be used by the app [24], and has a duty cycle mechanism for wireless receivers called ContikiMAC. Wireless receivers must remain turned on if any signal is received. As they consume a lot of energy, the idea is to reduce their work cycles, keeping them in standby and waking them up from time to time to receive some message. Obviously, this increases the chance of missing messages, as the receiver can be in standby when a message arrives, preventing it from receiving it. It's up to the developer to decide how often the receivers should be woken up in order to keep the network usable and save energy. According to the literature, this mechanism allows saving something around 10% and 80%, depending on how often the transceivers are woken up [25].

1.5 Prototyping boards

As already mentioned, the choice of which hardware to use has a big impact on the device's power consumption. The developer has the option of building the entire device manually, or using a prototyping board. Using a custom device is generally a cheaper and more power economical option, as it uses only the necessary hardware. However, this requires some knowledge in electronics.

An alternative is to use prototyping boards. They have become quite popular precisely for allowing the creation of embedded applications without the need of complicated circuits. As the name suggests, they're often used for prototyping, but it's not uncommon to see projects in production using one of these boards.

In this dissertation, we prefer to use a prototyping board to simplify the applications development. However, most of the prototyping boards available on the market consumes a lot of energy, which ends up harming the visualization of power savings implemented by the application. Because of this, we used the Mini Ultra 8 Mhz board from Rocket Scream (Figure 2), which is an Arduino Pro mini compatible board that implements some hardware modifications to consume less energy. When choosing a low power board, it's also important to note its voltage and clock frequency. The smaller are these values, the less is the power consumption.

Instead of using a custom board, like the Mini Ultra 8Mhz, it's also possible to modify an existing one. The most common modifications found in the Arduino community are based at the Arduino Pro Mini, and reduce clock speed, remove indicator LEDs and change the voltage regulator.

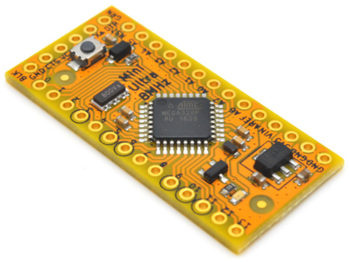


Figure 2: Mini Ultra 8 MHz by Rocket Scream

1.6 Power saving techniques used in this dissertation

The main power saving technique used in this dissertation is applying sleep modes to device hardware during idle periods. To implement this technique, the software must identify idle periods and apply the deepest sleep mode possible. Manually implementing it is considered a complex and error-prone task, which justifies the need for automatic and transparent mechanisms. In this context, we used the already existing power-saving mechanism of the Céu programming language, and evaluate it by measuring the power consumption. Also worth mentioning that TinyOS is a good alternative in this case. However, it uses a callback-based programming model, that is known as complex.

Another technique used in this work is turn off device hardware while not in use. We have implemented two drivers (d-sensor and a-sensor) that allow digital and analog sensors (respectively) to be turned off automatically while not in use. As all the hardware used could be powered directly by the prototyping board, we didn't need to use MOSFET transistors. At last, we also used hardware that manages power automatically, such as the temperature and humidity sensor DHT11, the radio transceiver nRF24l01+ and the Bluetooth module HC-05.

The use of techniques that adjust the QoS or functionalities of the application is not the objective of this work, and neither of the Céu programming language. Energy harvesting techniques, such as the use of solar panels, and energy-aware network protocols are also outside the scope of this work.

2 THE PROGRAMMING LANGUAGE CÉU

Céu is a reactive programming language based on Esterel [26]. As a **reactive** language, Céu requires an environment to communicate with the outside world and expose input and output events that the applications can use [27].

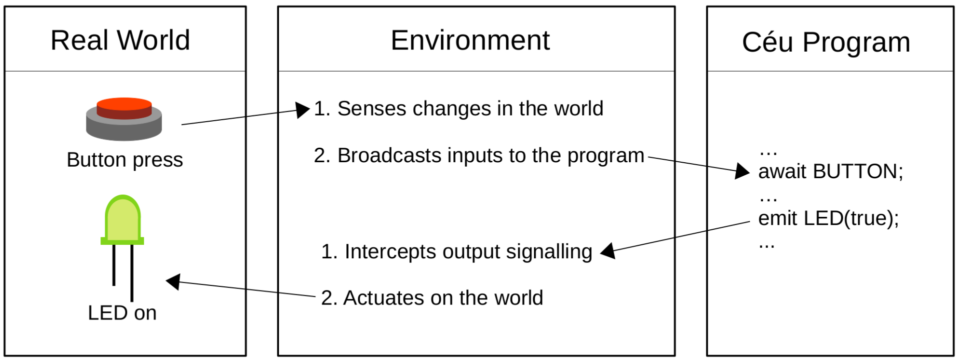


Figure 3: Céu program interacting with real world thought an environment. Based at: https://ceu-lang.github.io/ceu/out/manual/v0.30

Céu environments are developed separately from Céu core code, allowing the same core to be used in different contexts. This dissertation is using the Céu-Arduino environment, which supports the development of Arduino and embedded applications in the programming language Céu, targeting resource-constrained embedded systems. In a previous work [4], this environment was expanded with energy aware drivers, which add transparent standby mechanism to the language.

To exemplify the main features of the language, Listing 1 presents an application that blinks two LEDs at different rates. This example is complicated to be implemented in the Arduino language, and ends up requiring the use of control variables [3]. Using Céu, this implementation is straightforward. The program uses the structured control mechanism `par` to create two lines of execution (known as trails), each one blinking a different LED. The trails execute synchronously, i.e., there is no preemption or real parallelism.

Communication with the environment is made through `await` and `emit` statements. In this application, the `emit` is used to turn LEDs on and off, and the `await` statement halts the running trail until the specified event occurs, in this case, the passage of a certain time (250ms or 1s). In Céu, events received from the environment control the execution of the application, which characterizes a reactive language.

[3]Using variables to control the application flow usually impairs the sequential reading of the application

```
#include "out.ceu"
#include "wclock.ceu"

output high/low OUT_01; // LED connected to pin 1
output high/low OUT_02; // LED connected to pin 2

par do
    loop do
        emit OUT_01(high);
        await 250ms;
        emit OUT_01(low);
        await 250ms;
    end
with
    loop do
        emit OUT_02(high);
        await 1s;
        emit OUT_02(low);
        await 1s;
    end
end
```

Listing 1: Céu application that blinks two LEDs in parallel at different rates

2.1 Interrupts and drivers

Interrupts are signals that interrupt what is running on the processor. An embedded application can define interrupt handling routines (also known as Interrupt Service Routines - ISRs) to be called when an interrupt occurs. After an ISR is executed the application resumes running where it left off. The ISRs are available in both Arduino and Céu language and introduce a assyncronous behavior to the application.

Céu deals with the asynchronicity of interrupts and ISRs just as the environment deals with the outside world (Figure 3). Perceived changes in ISRs are informed to the main application through events (via `await` and `emit` primitives). That way, the changes will be perceived like any other event, and the main application can still execute synchronously [13].

As ISRs introduce asynchronous behavior, race conditions can be created due to shared memory access. The Céu compiler can identify risky situations and generate a compilation error. Céu also supports the `atomic` primitive, which allows a code block to be executed atomically without being interrupted. However, low-level implementations like ISRs are usually not explicitly used in applications, but encapsulated in drivers which constitute the environment.

Drivers are write-once components developed by embedded systems specialists that are reused in most applications [13]. They encapsulate the low-level complexity, asynchronous behavior, and hardware details. As an example, we present the int0 driver, starting with its use in an example application and, then, detailing its implementation.

The example application is presented in Listing 2. It blinks an LED every second until a button is pressed (ln 8), finalizing with the LED off. As in the example in Listing 1, the program creates two execution trails, but this time using the `par/or` statement. While the `par` never rejoins, the `par/or` rejoins when any of the trails terminate, aborting all other trails. It's regarded as an orthogonal preemption primitive because the trails need not to be tweaked to affect each other [13].

```
 1 #include "out.ceu"
 2 #include "int0.ceu"
 3 #include "wclock.ceu"
 4
 5 output high/low OUT_13; // an LED is connected to the pin 13
 6
 7 par/or do
 8     await INT0;
 9 with
10     do finalize with
11         emit OUT_13(low);
12     end
13     loop do
14         emit OUT_13(high);
15         await 1s;
16         emit OUT_13(low);
17         await 1s;
18     end
19 end
```

Listing 2: Blinks an LED every second until a interrupt is received

The trail defined between lines 10 and 18 never terminates and blinks an LED every second inside an endless loop. The trail defined at line 8, on the other hand, terminates whenever the application receives a `INT0` event, which indicates when the button was pressed. When this happens, the entire `par/or` block terminates and the other trail is aborted. The `finalize` clause (ln 10-12) ensures that its enclosing trail always finalizes with the LED off, even if it's aborted.

The button is connected to the MCU via pin 2 (as shown in Figure 4). On the

Mini Ultra 8MHz prototyping board, pin 2 can be used as an external interrupt source, the `INT0`. The int0 driver encapsulates all interrupt configuration details and emits the `INT0` event to the application whenever the value received by pin 2 changes. Due to the use of pull-up resistors, the application receives a high value on this pin by default, and only receives a low value when the button is pressed.

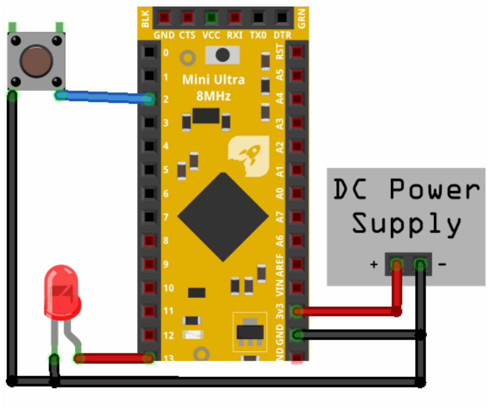


Figure 4: Circuit of the application that blinks an LED every second until a interrupt is received

The “out” and “wclock” drivers are also used. The wall clock driver (wclock) controls the passage of time according to the real world. The “out” driver is responsible for receiving emitted events from the application and output the event payload (which can be `high` or `low`) through digital pins to the real world. In this case, the application is emitting the output event `OUT_13` (ln 5), that is being used to turn on and off an LED connected to pin 13.

Listing 3 presents the int0 driver implementation. At first line, we declare an input event `INT0`, that caries no values (`none`). An input event is always emitted from the environment or from an ISR to the application. The ISR is declared between lines 17 and 19 using the `spawn async/isr` primitive. It’s executed asynchronously whenever the associated interruption occurs (`INT0_vect`). In this case, it simply emits the `INT0` event. This way, even if the ISR is called asynchronously, the `INT0` event is scheduled into the event queue and treated synchronously by Céu application.

Céu is designed to interoperate seamlessly with the C programming language [13]. One of these interoperations allows native symbols defined externally in C to be used inside a Céu program. In order to do so, the external symbols must be declared before their first use using the `native` statement and should be prefixed with an underscore [27]. At line 5, the int0 driver declares that it wants to use the `INT0_vect`, and uses the

```
1  input none INT0;
2
3  #define INT0_PIN 2
4
5  native/const _INT0_vect;
6
7  code/call INT0_Get (none) -> high/low do
8      escape _digitalRead(INT0_PIN) as high/low;
9  end
10
11 {
12     pinMode(INT0_PIN, INPUT_PULLUP);
13     EICRA = (EICRA & ~((1<<ISC00) | (1<<ISC01))) | (CHANGE << ISC00);
14     EIMSK |= (1 << INT0);
15 }
16
17 spawn async/isr [_INT0_vect] do
18     emit INT0;
19 end
```

Listing 3: int0 driver

modifier `/const` so it can't be reassigned. That variable is included by the Céu-Arduino environment and it's based on the Arduino library.

It's also possible to include C code between curly braces. The code block between lines 11 and 15 uses the `pinMode` function from the Arduino library to set the pin associated with `INT0` as input and turn on the pull-up registers[4] [28]. Lines 13 and 14 show a low-level code that manipulates the MCU registers to enable the `INT0` interrupt (ln 14) and configure it to happen on change (ln 13).

The `INT0_GET code/call` is a code abstraction that returns (using `escape` statement) the value of the pin associated with the `INT0` interrupt. A `code/call` is similar to a C function, and can be defined as a subprogram that runs to completion and can't contain synchronous control statements (e.g. `await`, `spawn`) [27]. The `INT0_Get` uses the C `digitalRead` function to retrieve the value from pin 2, and converts its return value from `boolean` to `high/low`. The way this interpolation works is very similar to `INT0_vect`, but this time it's not necessary to declare the symbol using `native` because Céu-Arduino environment already does that. As the `INT0` event is emitted to the application every time the pin changes, the `INT0_GET code/call` is used so the application can identify if

[4]When an input pin is not connected to any source, Arduino reads high or low randomly. Configuring the input pin as INPUT_PULLUP includes a set of resistors into the input circuit, and makes Arduino reads high if no source is connected

the pin changed from `low` to `high` or from `high` to `low`.

2.2 Power management mechanism

Céu energy saving mechanisms are based on automatically identifying idle periods in the application and applying the most economical sleep mode to the MCU automatically. Looking for an application written in Céu, it's easy to identify the idle periods due to the `await` statement. As an example, we can identify three idle moments in Listing 2: while waiting for `INT0` interrupt (ln 8) and waiting for 1 second to pass at lines 15 and 17.

To better illustrate how Céu identifies the idle periods, Listing 4 presents an overview of the Céu runtime. It maintains a event queue (ln 1), in which new events are stored. To ensure that an MCU can sleep, all incoming events are generated through ISRs. At line 3, the application starts, and reaches the `await` statement in the multiple trails of the program (known as "boot reaction"). After that, the application verifies the event queue in an endless loop known as "event loop" (ln 4-11) [13].

Upon receiving an event, the Céu runtime resumes executing the trails that are waiting for this event (ln 7). Input events emitted asynchronously by ISRs are stored in the event queue and queried in subsequent iterations of the loop [13]. When there is no event to react to, i.e., when the application is waiting, the MCU sleeps (ln 9). To choose each sleep mode to apply, the `ceu_pm_sleep` instruction verifies which interruptions may wake up the MCU (Listing 5).

```
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)) { // queries input queue
7              ceu_sync(&evt); // executes synchronous
8          } else {
9              ceu_pm_sleep(); // nothing to execute
10         }
11     }
12 }
```

Listing 4: Céu runtime architecture

To illustrate this verification, consider an application that waits for a `INT0` interrupt to transmit a radio message. In this case, the MCU remains in the deepest sleep mode waiting for an event arrive and awake it up. According to Table 1, all sleep modes of the MCU ATMega328/p can be woken up from an external interrupt (indicated as INT in the table). As all of them can be used, Céu chooses the most power efficient, the "power down" mode.

If the driver uses some wake-up source that is not available in deepest sleep mode, it should use the `ceu_pm_set` instruction. This instruction allows the driver to inform Céu if a certain interruption should or shouldn't wake-up the MCU. Céu language power management mechanism uses this information to choose the sleep mode (as shown in Listing 5). Drivers are also responsible for indicating when the interrupts are no longer needed. At this point, it may be interesting to use the `finalize` statement, to ensure that they are removed regardless of the way the driver finalizes (because it terminates to execute or because it was aborted).

```
void ceu_pm_sleep (void) {
    if (ceu_pm_get(CEU_PM_TIMER1) || ...) {
        sleep_mode_idle(...);
    } else if (ceu_pm_get(CEU_PM_ADC)) {
        sleep_mode_adc(...);
    } ...
    } else {
        sleep_mode_power_down(...);
    }
}
```

Listing 5: Céu language power management engine: choosing the sleep mode based at the wake-up sources

An example of a driver that uses the `ceu_pm_set` instruction is the wclock. It configures an interrupt to happen after a predefined period of time. This interruption is generated by a MCU internal timer "timer 1". This timer is off during most ATmega328/p sleep modes, which prevents the interrupt from being generated. The only sleep mode that allows the hardware related to timer 1 to remain on is "idle". Therefore, timer 1 can only be used as a wake-up source if the MCU is in "idle" mode.

3 METHODOLOGY

As described at the Introduction, this work aims to evaluate the power-saving mechanism of the Céu programming language, and analyze their possible barriers to adoption. In summary, we achieve this by analyzing the power consumption of typical applications in the context of embedded software implemented in Céu and comparing them to their implementation in Arduino, as suggested in a previous work [4]. To discuss the adoption barriers, we compared their memory usage and code readability.

The rest of this chapter is organized as follows: Section 3.1 discusses the process of choosing the applications. Section 3.2 describes the different implementations of each application, Section 3.3 discusses the embedded devices power consumption analysis and explains how we measured, estimated and compared it. At last, Section 3.4 explains how we analyzed the memory consumption and Section 3.5 discusses about the code readability comparison.

3.1 Choosing the applications

Previous work [4] suggested evaluating the power consumption of realistic applications by rewriting open-source projects developed by the Arduino community in Céu, and comparing the time to rewrite, the resulting program structure, and the actual energy efficiency. However, we found out some difficulties in this approach:

- As Arduino has a great educational appeal, a large part of the applications found in the community are too simple, and many do not have their intellectual property clearly explained;
- "Time to rewrite" analysis would be highly influenced by the developer's prior knowledge of Arduino and Céu languages. To mitigate this influence, it would be necessary to conduct the study with a large group of developers. Furthermore, as Céu is not very well known and used language, it would be necessary to provide training for these developers. At last, the quality of that training would also be an aggressive influence on this analysis.

Because of these factors, we adapt the proposal by removing the "time to rewrite" analysis. Also, instead of rewriting the already existing Arduino code, we implemented both Arduino and Céu applications. The chosen applications represent common patterns in the development of embedded software, such as sensor reading, actuator control, radio and Bluetooth communication and concurrent behavior [5], and illustrate common code patterns that hamper readability (explained at Section 3.5). These applications also illustrates common hardware and software power-saving techniques used in SO-less resource-constrained devices that are related to standby mode. These techniques are described at the Chapter 1, and include:

- Choosing hardware that control their own power consumption, such as the HC-05 Bluetooth module and the DHT11 sensor;
- Turning hardware off when not in use. This technique was applied to the soil humidity sensor and for the ky-036 sensor;
- Controlling hardware power saving modes. In our case we controlled the MCU sleep modes, but other hardware (sensor, actuators, radios etc.) need to have their power-saving modes controlled via software

The chosen applications are:

- An smart lighting system consisting of two devices, a transmitter and a receiver, that communicate via Bluetooth.
- A device that collects air humidity and soil moisture, and transmits this information via radio.

3.2 Applications implementations

We wrote three implementations for each application. The first one, in Arduino, illustrates the applications normally found in the community. They generally follow the structured and sequential style proposed by the Arduino platform, which allows simple and readable code, and avoids the complexity of dealing with asynchronous code and Interrupt Service Routines (ISRs). However, it's common to use blocking operations, which impair reactivity and prevent operations from executing concurrently [4] [29]. In this implementation there are no energy savings at software level.

[5]For example, waiting for a radio message to arrive while waiting a presence detection

In the second implementation, also in Arduino, we performed all energy saving manually. These types of implementations are typically used in more professional projects, especially when creating battery-powered devices. They are characterized by being complex and difficult to read, and containing ISRs, control variables and low-level code, such as register manipulation. It's also common for them to violate the structured and sequential style proposed by Arduino to accommodate concurrency.

The last implementation, in Céu, demonstrates the transparent power saving mechanism provided by the language, and its structured reactive style that allows an easy-to-read code. At this point, it was necessary to assess the need to create additional drivers. As already mentioned, we created three new drivers.

3.3 Power consumption analysis

After developing the applications, we estimated the power consumption of each of their implementations. Estimating the energy consumption of an embedded device usually starts even before a prototype is created. Developers analyze each hardware element separately, and verify their power consumption by consulting datasheets or manually measuring it. Although simple, this analysis already helps in choosing the hardware and contributes to a better project [30].

When a prototype is already available it can be measured as a black box the same way the hardware elements are. In this moment, the measurement is useful not only for estimating consumption, but also for detecting energy leakage before creating the final version of the device [30]. Before or after a prototype exists, the power measurements can be used to estimate the power consumption and battery life.

Our analysis mixes these two approaches and was performed in five steps:

1. **Analyzing each hardware individually** to identify the order of magnitude of its consumption (mA, µA , nA etc.), and how quickly it changes. This step is crucial to define which measuring instruments can be used.

2. **Identifying the application states, and estimate their duration**. Some common states of embedded applications are: waiting for some time to pass, collecting data from a sensor, transmitting messages via radio and waiting for a user interaction. We modified the applications to time and display the duration of most states

(this modification was removed later). For states where the application waits for a predefined period, it was not necessary to time it. At last, for applications that depend on user interaction, a typical usage routine was considered.

3. **Measuring the consumption of each state**. The most common measuring instruments allow readings between several orders of magnitude (ranging between some microamps and some few amps). However, it's necessary to adjust the measuring range manually, which can prevent continuous measurement of an application that has large variations in power consumption. Thus, in this dissertation, we prefer to measure each application state individually. This allows each state to be measured with the most suitable instrument and avoids the need of instruments that allow uninterrupted reading, which are generally more expensive and less accessible. Sections 3.3.2 and 3.3.3 better discuss the challenges of measuring power consumption of embedded devices, and which instruments were chosen for this dissertation.

4. **Estimating the average power consumption** by calculating:

$$\frac{(state1power \cdot state1duration) + ... + (stateXpower \cdot stateXduration)}{state1duration + ... + stateXduration}$$

As an example, consider an application that transmits a message via radio every second. Also consider that the device consumes 5 µA while waiting one second, and 20 mA during transmission, which lasts 55ms.

$$\frac{(5\mu A \cdot 1s) + (20mA \cdot 55ms)}{1s + 55ms} = \frac{(5\mu A \cdot 1000ms) + (20000\mu A \cdot 55ms)}{1000ms + 55ms}$$

$$= 1047.39\mu A \approx 1mA$$

Therefore, the average consumption of this application is approximately 1 mA.

5. **Estimate battery life:** The average consumption obtained from the previous step would be enough to compare the applications. However, we also estimated the battery life, as we understand this to be a more meaningful information for the reader. Accurately estimating how long a battery will last can take into account several factors, such as temperature and its composition, discharge rate and cutoff

voltage. For this work, we will consider ideal conditions for simplicity, following the formula:

$$BatteryLife = \frac{BatteryCapacity(mAh)}{AverageDeviceConsumption(mA)}$$

As an example, the device from the previous step would remain on for eight days and eight hours if powered by a 200 mAh battery.

$$BatteryLife = \frac{200mAh}{1mA} = 200h = 8\ days\ and\ 8\ hours$$

3.3.1 Power comparison

Arduino and Céu implementations were compared to illustrate the transparent power saving mechanism of the Céu programming language. To obtain the power savings percentage, we used the average consumption of each implementation as follows:

$$PercentageOfPowerSavings = (1 - (Ceu/Arduino)) \cdot 100$$

As an example, consider an application whose Arduino and Céu implementations consume, respectively, 30 and 15 mA. This means that the implementation in Céu consumes 50% less energy than the implementation in Arduino.

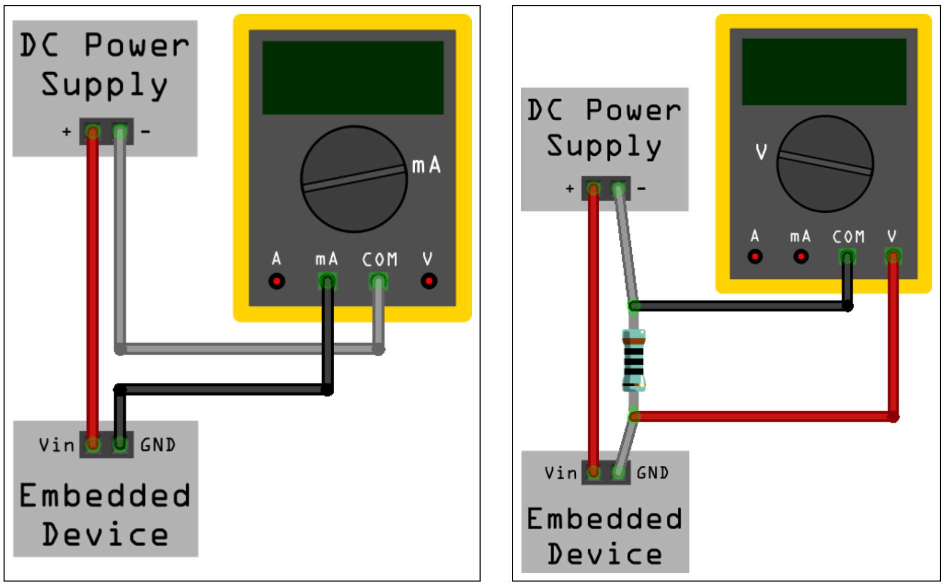
The Arduino with manual power management and Céu implementations were also compared based on their energy consumption. Ideally, they should have the same power consumption, otherwise it means that either the Arduino application can still be optimized, or that the Céu language is not applying the most efficient power-saving mode.

3.3.2 Measuring instruments

Measuring the consumption of embedded devices can be a challenge, as the consumption is generally small (usually no more than some milliamps) and varies quickly between several orders of magnitude: an embedded software application can easily consume tens of milliamps at its peak and a few microamps during its most economical mode [30].

Several instruments can be used for this, from the most generic ones, such as digital multimeters and oscilloscopes, to more specific instruments, such as current waveform analyzers. During this work, some instruments were analyzed:

- **Digital multimeters** are relatively inexpensive and easy to find instruments. Basically, there are two ways of measuring embedded devices current using a multimeter. The first one is using the multimeter as an ammeter to directly measure the current. In this case, the multimeter is introduced in the circuit in series (Figure 5(a)). Another possibility is to use a shunt resistor (Figure 5(b)). This technique consists of inserting a low-resistance resistor into the circuit and measuring the voltage across it. Thus, it's possible to obtain the current using ohm's law.

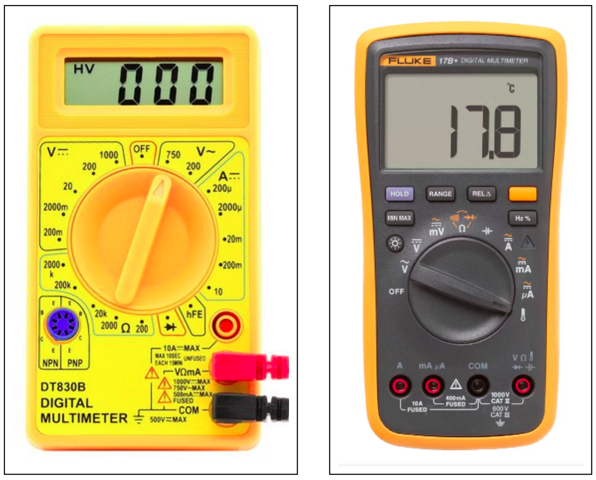


(a) Using ammeter functionality (b) Using a shunt resistor

Figure 5: Measuring current using a multimeter

Using a multimeter to measure an application that varies greatly in power consumption requires some precautions. Although it allows measuring currents from a few milliamperes to amperes, it's necessary to manually choose a range to measure. Ranges vary between different multimeter models as shown in (Figure 6).

As an example, the DT830B multimeter has 5 range settings for measuring current (200µA, 2000µA, 20mA, 200mA, 10A). This means that, for reading a 100µA, we need to choose the first setting, but for reading a 10mA, we should chose the third one. The Fluke 17B+, by the other hand, has only three ranges for measuring



(a) DT830B (b) Fluke 17B+

Figure 6: Multimeter range settings

current (µA, mA and A). To measure different power modes (varying between milliamperes and microamperes, for example), it's necessary to manually change the settings and taking separate recordings to patch together a full picture of the power consumption.

Multimeters are not suitable instruments for analyzing rapid changes in consumption, such as consumption peaks of a few microseconds. As the goal of multimeters is to measure and display a practically constant value (either current or voltage), these variations are difficult to notice and can lead to measurement errors.

- **Pliers ammeters** use the Hall effect to measure current non-invasively (Figure 7). However, this instrument can't measure small currents like those consumed in embedded devices (usually µA to few mA).

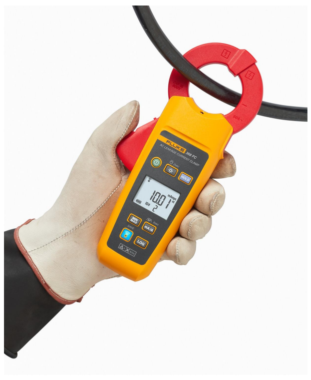


Figure 7: Pliers ammeter

- **Oscilloscopes** can be used to measure current through common probes by looking at the differential voltage around a shunt resistor (in the same way as used in multimeters). A more accurate solution (but which requires a greater financial investment) is to use a specific probe for current measurement. Oscilloscopes produce a graph with the measured value, which allows a better visualization of the application's consumption pattern.

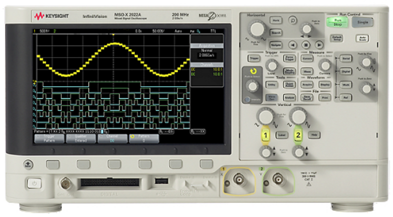


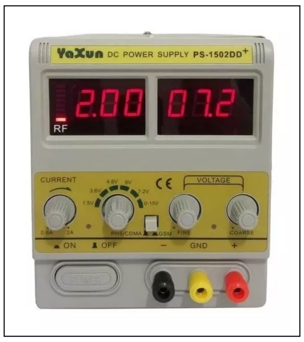
Figure 8: Oscilloscope

As with the multimeter, manually changing settings (such as changing shunt resistors and probes) and taking separate recordings may be required to prevent small differences in the measured values from being omitted because of the scope's resolution.

- **Embedded devices current waveform analyzers** are specific instruments for measuring embedded devices power consumption. It's possible to measure a wide range of current (from picoamps to a few amps) and generate more accurate graphics than oscilloscopes. In addition, there is no need to manually change the measuring range, which allows the measurement to be done without interruptions. However, the major limitation for using theses instruments is its high price [31].

Figure 9: Current waveform analyzer

- **DC bench power supply** usually have a display that indicates the amount of current supplied (Figure 10(a)). However, this measurement is generally not accurate. More professional models have a more accurate measurement. The Keithley 2280S (Figure 10(b)), for example, even allows the creation of accurate consumption graphs [32], just like current waveform analyzers does.


(a) YaXun PS-1502DD

(b) Keithley 2280S

Figure 10: DC power supplies

- **Arduino based current meters** are cheap and accessible instruments, which allow the measured value to be logged via serial. They can be analyzed in a PC to create graphs. It's also possible to visualize the measured values in real time using serial plotters software.

  Although there are several Arduino-compatible current sensors available on the market, most are not capable of measuring currents as small as those consumed by embedded devices. The alternative is to measure current indirectly, using a shunt resistor (Figure 11).

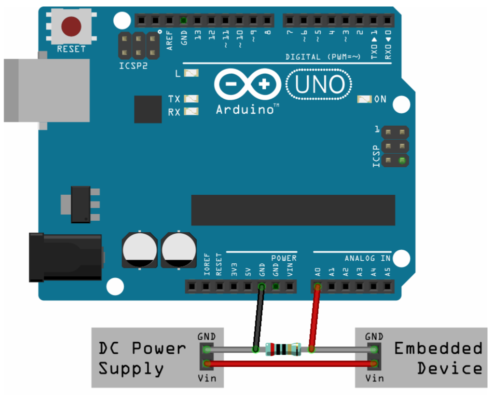


Figure 11: Arduino based current meter

Using an Arduino for current measurement requires attention when choosing the shunt resistor and the analog reference value of the analog to digital converter. These two settings influence the resolution.

3.3.3 Instruments used in this work

In this dissertation we use the Fluke 17B+ digital multimeter (Figure 6(b)), and a current meter based on Arduino. These instruments are accessible and cheap when compared to other ones.

The multimeter was used to measure application states where there is little variation in power consumption and that last long enough to be perceived by the instrument. In contrast, the current meter in Arduino was used to analyze consumption peaks and measure shorter states. As an example, we can mention the active state of the DHT11 temperature sensor, which only lasts about 25ms. Such a short duration wasn't abble to be measured by the multimeter.

With the current meter based on Arduino it was also possible to observe the power consumption peak during the transmission of the radio nRF24l01+. This peak lasts about 1 millisecond and consumes 11mA. In this case, the Arduino didn't behave so well and ignored some transmission peaks. However, the observed peaks indicate a consumption similar to that presented in the transceiver datasheet and to the consumption measured by members of the Arduino community [33] [34].

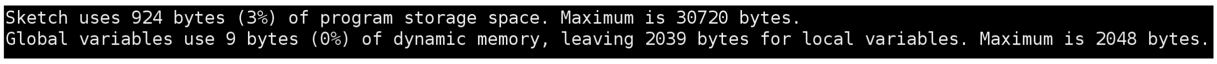
It's also worth mentioning that we prefer to use a power supply instead of batteries to perform the measurements. In our case, we use YaXun 1502DD, but any other could be used. When discharging, batteries decrease the voltage supplied to the circuit, which ends up influencing the measured current.

3.4 **Memory consumption analysis**

Our memory analysis used the information provided by the Arduino compiler, which indicates the storage and memory usage against the specifications of the selected board. As Céu-Arduino compiles a Céu program to C and uses the Arduino compiler to upload it to the MCU, we could obtain the memory usage information in the same way for all implementations.

Figure 12 shows an example of the memory usage information provided by the

Arduino compiler. At the first line it indicates the used "program store space", a ROM Flash memory where the Arduino program and the bootloader are stored[6]. The second line indicates the used SRAM, which is where the program creates and manipulates variables when it runs [35].



```
Sketch uses 924 bytes (3%) of program storage space. Maximum is 30720 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.
```

Figure 12: Memory usage information provided by the Arduino compiler

We expect Céu applications to consume more memory, since Céu-Arduino stores information that are used by the language to provide the synchronous behavior and the automatic power-saving mechanism. Examples of this information are: current MCU sleep mode and the input events queue. However, we advocate that Céu compiler can still be optimized to use less memory. To validate it, we performed an initial manual optimization in which unused code was manually removed.

3.5 **Readability comparison**

In software development, "readability" is a software quality metric that illustrates how easy a software is to read and understand. This term can overlap with other terms such as "legibility", "understandability" and "comprehension" and its definition varies by author [36]. It's worth mentioning that readability is closely related to writeability and maintainability, mostly because the majority of the development time is spent on debugging and understanding an already existing code.

Several works try to analyze and measure code readability, and are mostly based on three approaches. **Human-centered studies** analyze code readability by performing experiments with a group of developers. This type of study is good to illustrate the readability perception of a certain group. As a downside, it usually involves a large number of developers, and their background, previous programming experience and familiarity with the evaluated programming language have a great impact on the result. When comparing two or more programming languages, the developer experience have even a greater impact since they're likely to know more about one language than the other, which could influence his readability analysis.

[6]Although important, bootloaders usually don't consume a lot of memory. The ATMega328p, for example, it consumes only about .5k bytes

Model-based studies create, execute and/or analyze readability models. These models analyze characteristics of the code, such as number of operators, program size and number of indentations to generate a score that represents the readability. These models have great appeal in the industry, as they allow a quick verification of code quality and are easily integrated with continuous integration tools. Common uses are to automatically check whether a code modification improves or decreases the code quality of a project, and identify the hardest-to-read files so developers can analyze and propose refactorings.

The last approach is to analyze **code patterns that hamper readability**. An example of one of these patterns is the problem known as "prop drilling", easily found in projects that use the React framework. In this case, a property is passed through a component hierarchy just to be used by the final one. It's argued that this problem impairs the readability since the definition and usage of a variable can be very far away, but mainly because several components in the hierarchy needs to know a property that they shouldn't know only due to implementation details. Therefore, it's useful to identify this pattern in a code project and try to use some alternative. When comparing two different programming languages, this analysis can also be very helpful to illustrate problems that are likely to happen at language A and discuss why they are not likely to happen when using language B.

The next sections present some related works that analyze readability (Section 3.5.1), and discuss the chosen approach for this dissertation (Section 3.5.2).

3.5.1 Related works

This section presents several related works that were used as reference for defining the readability analysis methodology for this dissertation.

Work [37] analyzes several commits related to code readability improvements and tries to answer some questions, including: "Are state-of-the-art readability models able to capture readability improvements in practice?". To answer this question, three existing state of the art readability models were investigated: Scalabrino, Dorn and a Combined model, which combines the first two models as well as Buse Weimer's model and Posnett's model. The results confirm that those models fail to capture readability improvements and thus do not appear to be suitable for day to day maintenance tasks.

The article [38] promoted a study where several programmers were challenged to implement a solution to one single problem using one of the following programming languages: C, C++, Java, Perl, Python, Rexx, and Tcl. After that, their implementation were compared based on the following metrics with the objective of comparing the programming languages features: programming length, amount of comments, program reliability (i.e. if the implementation produces the expected result), robustness (if the implementation doesn't fail when received unexpected inputs) and work time and productivity. This article raises concerns about comparing different programming languages and the reliability of their experiments. One of these concerns is that "Any programming language comparison based on actual sample programs is valid only to the degree to which the capabilities of the respective programmers using these languages are similar".

The paper [39] conducted a survey where 120 students evaluated 100 snippets of Java code according to their readability. The survey results were compared to a set of local code features (such as number of blank lines and number of comments) in order to understand which of these metrics are important, how important they are, and whether they influence positively or negatively the quality of the code. Using those features, they construct a readability measure.

At last, [10] rewrote the video game Pingus from C++ to the language Céu, and discussed how the Céu characteristics helped eliminating patterns that hamper readability usually found at C++ video games implementations. One of the mentioned patterns is the "lifespan hierarchies", which illustrates that memory and visibility of game entities are managed through class fields and containers. In Céu, all entities have an associated lexical scope, similarly to local variables with automatic memory management, so there is no need to use variables and to manage memory explicitly. Together with the analysis of other patterns, the article concluded that most difficulties in implementing control-flow behavior in game logic is not inherent to this domain, but a result of accidental complexity due to the lack of structured abstractions and an appropriate concurrency model to develop event-based applications.

3.5.2 The chosen approach

Among the approaches, we discard the use of models as some works (such as [37]) already consider them unrepresentative. Furthermore, more commercial models such as those provided by the SourceMeter[7] and SonarQube[8] tools are not available for Arduino and Céu.

We also chose to not execute human-centered studies due to the various threats to their validity, as pointed at [38]. Among of the threads, the fact that the language Céu is not yet well known and that its synchronous and event-based paradigm is not so commonly used could negatively influence the readability analysis of Céu programs. We also believe that the participants would have to receive a training about the Céu language, and the quality of the training could be a big influence in the experiment results.

To analyze the readability, we chose the "code pattern" approach. The first step was to identify some code patterns that hamper readability while developing Arduino applications. The patterns found were:

- **ISRs:** Although ISRs are easily found in embedded applications, using them represents a break in the sequential reading of the code. They also introduce asynchronous behavior, which makes the code more error prone and harder to debug. Finally, ISRs usually require the usage of global variables to allow the communication between ISRs and the main application.

- **Register manipulation:** Registers often do not have an easy-to-read name and require the developer to know details about the hardware used. We can also consider that the operators used for register manipulation are less known/used by the Arduino community and can also hamper a fast code reading;

- **Global variables:** Using global variables usually represents a tight coupling of code and harms code encapsulation;

- **Control variables:** They are normal variables that are used together with control statements to store the application states and control the application flow. This pattern impairs sequential reading, and is usually found when the application has some concurrent behavior. It can also be found when the application needs to store states between the executions `loop` function and when an ISR needs to communicate with the main application.

[7]https://sourcemeter.com/
[8]https://www.sonarqube.org/

After that, we identified these patterns in the implementations of the applications developed in this dissertation in order to understand if using the Céu language helped to reduce the occurrence of those patterns.

4 DEVELOPED DRIVERS

Three drivers were developed throughout the dissertation in order to enable the development of the applications presented in Chapter 5.

4.1 Driver d-sensor

Several embedded applications receive values from digital sensors, but often this reading does not occur throughout all the application life cycle. Therefore, the circuit responsible for reading the digital sensor can be switched off at times to save energy. The d-sensor driver uses the lexical scope of the Céu language to infer when a sensor circuit is no longer used and turn it off.

The circuit responsible for reading digital sensors can be customized for a particular project or be a shelf module like the ky-036 or ky-038. These modules (touch and microphone sensor, respectively) have, in addition to the sensor, other elements that facilitate the acquisition of the sensor data. In the case of ky-036, for example, the module has two indicative LEDs, a potentiometer to control the sensitivity of the sensor, an amplifier and a comparator. Some of these elements consume energy, and can be turned off when not in use.

```
#include "dsensor-int0.ceu"
#include "wclock.ceu"
#include "out.ceu"

output high/low OUT_13;

par/or do
    var& DSensor_INT0 mysensor = spawn DSensor_INT0(7, _, _);
    var high/low v = await mysensor.change until v==high;
with
    loop do
        emit OUT_13(high);
        await 1s;
        emit OUT_13(low);
        await 1s;
    end
end
```

Listing 6: d-sensor application example

Listing 6 shows, as an example, an application that blinks an LED every second (ln 11-16) while waiting for a digital sensor to detect a high value (ln 8-9). This sensor is being powered by pin 7 (Figure 13), and it's the d-sensor driver that manages when it should be on. When the high value is detected, the `await` at line 9 can proceed, and, since there are no more instructions, its enclosing trail terminates and the scope of `mysensor` is lost. In this moment, the sensor is automatically turned off, i.e. the output pin 7 becomes low.

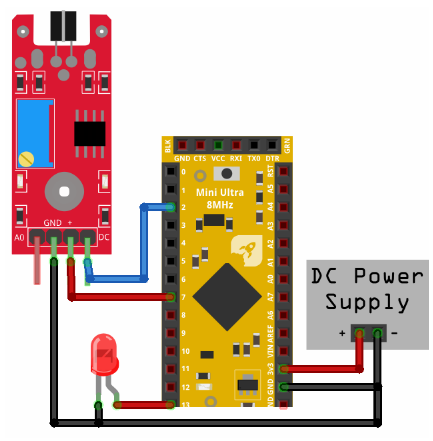


Figure 13: d-sensor application circuit

Listing 7 presents a second application, which waits for a high value from the touch sensor to transmit a radio message (the radio transmission was omitted for simplicity). The `do-end` (ln 4-7) statement creates an explicit block.

```
1  #include "dsensor-int0.ceu"
2
3  loop do
4      do
5          var& DSensor_INT0 mysensor = spawn DSensor_INT0(7, 1, 500);
6          var high/low v = await mysensor.change until v==high;
7      end
8      // sends a radio message
9  end
```

Listing 7: d-sensor application example - manually scoped

Block declarations are present in several programming languages (C, JavaScript, etc.) and using them is considered a good practice to delimit scope. In the example in Listing 7, the variables `mysensor` and `v` are limited to the `do-end` block and, then, can't be used outside it. As the Céu language and the d-sensor driver use the scope of

DSensor_INT0 to turn the sensor on and off on, not using blocks can negatively influence the application's energy consumption. In Listing 7, if there were no blocks, the sensor would remain on during the radio transmission.

Listing 8 presents a simplified version of the driver implementation. As soon as DSensor_INT0 is spawned, the driver sets the power pin as output (ln 9) and turns the sensor on (ln 10). Before going to the next step, the d-sensor waits some milliseconds for the circuit to stabilize. This time varies from circuit to circuit, and should be evaluated based on the hardware used. For off-the-shelf modules, this value is usually present in datasheets.

```
1  #include "int0.ceu"
2  #include "wclock.ceu"
3
4  code/await DSensor_INT0(var int energyPort, var int time, var int
       debounce) -> (event high/low change) -> NEVER do
5      do finalize with
6          _digitalWrite(energyPort, low);
7      end
8
9      _pinMode(energyPort, OUTPUT);
10     _digitalWrite(energyPort, HIGH); //turn on
11     await (time!)ms; // waits for the circuit to stabilize
12
13     loop do
14         await INT0;
15         await (debounce!)ms;
16
17         var high/low v = call INT0_Get();
18         emit change(v);
19     end
20 end
```

Listing 8: d-sensor driver

Between lines 13 and 19, the driver waits to receive the INT0 event, which indicates when pin 2 state changes. After receiving the event, the driver waits some milliseconds for debounce before emitting the internal event change (ln 17). This event caries the value of the pin 2, i.e., if the pin is receiving a high or low value. Unlike input events, internal events don't need to be emitted from ISRs and serves as signaling and communication mechanisms among trails [27].

4.2 Driver a-sensor

The a-sensor driver works similarly to the d-sensor, but for analog sensors. Listing 9 presents its implementation. As with the d-sensor, the a-sensor driver manages the informed power pin, turning it on when it starts (ln 10) and turning it off when it finalizes (ln 5-7). Its main difference is in obtaining sensor data. While the d-sensor waits for pin 2 value changes in a loop, the a-sensor performs a single analog reading (ln 13-14). After that, the `ASensor_Get` returns the read value using the `escape` statement[9].

```
 1 #include "wclock.ceu"
 2 #include "adc.ceu"
 3
 4 code/await ASensor_Get(var int dataPort, var int energyPort, var int
       time) -> int do
 5     do finalize with
 6         _digitalWrite(energyPort, low);
 7     end
 8
 9     pinMode(@energyPort!, OUTPUT);
10     digitalWrite(@energyPort!, HIGH);
11     await (time!)ms;
12
13     spawn Adc();
14     var int value = await Adc_Conversion(dataPort);
15
16     escape value;
17 end
```

Listing 9: a-sensor driver

Listing 10 exemplifies its use with an application that transmits readings from an analog sensor every 2 seconds. The `ASensor_Get` (ln 5) turns on the sensor, waits for the circuit to stabilize, reads the sensor value and turns it off right after. While the application transmits the value via radio and waits for two seconds, the sensor remains off.

[9]In this case, escape statement works as the C and Java return statement, i.e., terminating the deepest matching enclosing block and possibly carrying a value - in this case the variable “value”

```
#include "asensor.ceu"
#include "wclock.ceu"

loop do
    var int v = await ASensor_Get(_A0, 7, 1);
    // transmit via radio
    await 2s;
end
```

Listing 10: a-sensor application

4.3 Driver dht 11

The dht11 driver was created to enable the use of the DHT11 temperature and humidity sensor in applications written in Céu. This sensor can be easily connected to the Arduino, and it only requires a resistor. However, in this dissertation we used the breakout board shown in Figure 14 to facilitate the connection.

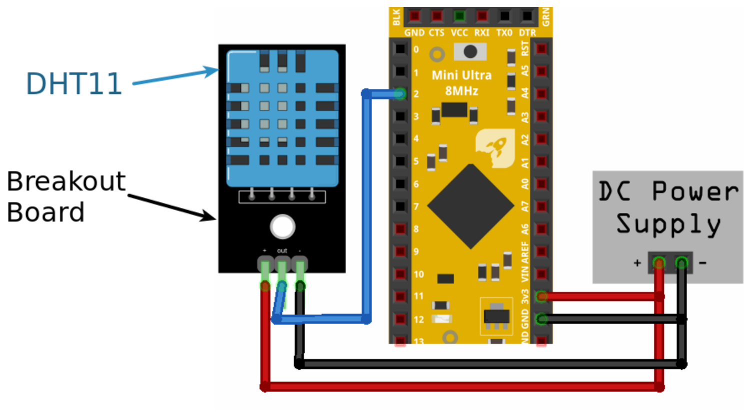


Figure 14: DHT11 humidity and temperature sensor in a breakout board connected to the Mini Ultra prototyping board

DHT11 uses a single digital pin to send the measured data to the MCU. In a simplified way, the sensor remains in standby mode until it receives a request from the MCU. Then, it transmits 40 bits of data: 16 bits for humidity, 16 bits for temperature and 8 bits for a checksum. After the transmission, the sensor returns to the standby mode and goes back to waiting for a new request.

The data request step is shown in Figure 15. The MCU starts the communication by pulling the data-bus low[10] and waiting at least 18 milliseconds before pulling it high

[10]The data bus between the MCU and DHT11 should be kept high by default. That's why we use a pull-up resistor. In our case, it is encapsulated inside the DHT11 breakout board.

again to request for a sensor response. Then, the sensor pulls down the wire for 80 microseconds as a response signal and indicates that data transmission will begin by pulling it up again for another 80 microseconds.

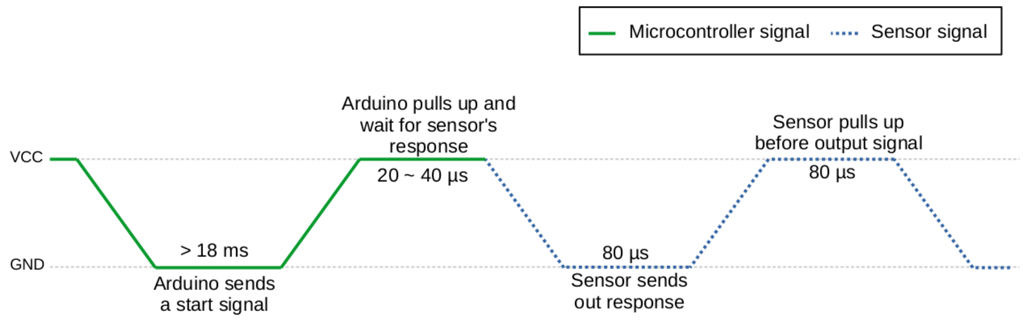

Microcontroller signal
Sensor signal
VCC
GND
> 18 ms
Arduino sends a start signal
Arduino pulls up and wait for sensor's response
20 ~ 40 µs
80 µs
Sensor sends out response
Sensor pulls up before output signal
80 µs

Figure 15: Start of the communication between a microcontroller and a DHT11 sensor

All bit transmission starts with a low signal of 50 microseconds, followed by a high signal. The duration of each high signal indicates whether it represents bit “0” (between 26 and 28 microseconds) or “1” (70 microseconds) - Figure 16.

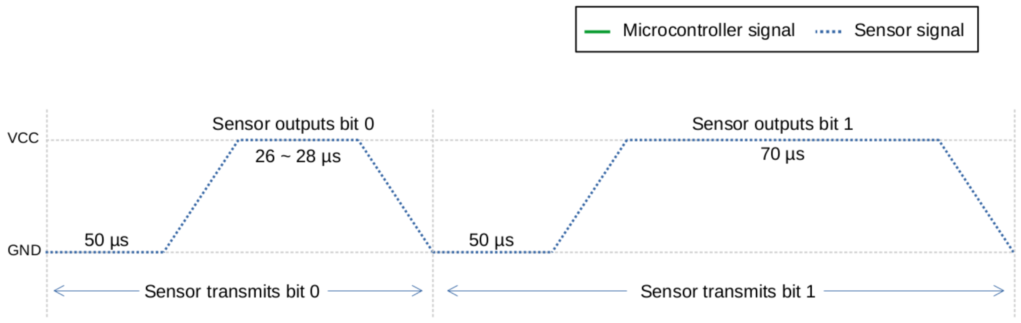


Figure 16: DHT sends data to microcontroller

The dht11 driver implements this protocol in a single `code/call`, the `DHT_Read`. Listing 11 shows an example of how this `code/call` is used. The example application performs the reading on line 8, passing as a parameter to `DHT_Read` the pin that connects the sensor to the Mini Ultra (pin 2), and the `temp` and `hum` variables. The `&` operator allows these variables to be changed inside the `code/call`.

```
#include "wclock.ceu"
#include "dht11.ceu"
loop do
    await 2s;

    var real temp=0;
    var real hum=0;
    call DHT_Read(2, &temp, &hum);

    // Format and transmit temperature and humidity data
end
```

Listing 11: Application that reads temperature and humidity data and transmits them via radio

5 COMPARISON OF APPLICATIONS IN CÉU AND ARDUINO

5.1 Smart lighting

The smart lighting application was inspired by previous work [29], and aims to reduce unnecessary lighting in a environment. The application monitors the presence in the environment, and turns on the lighting when a presence is detected. After some time, the lighting is turned off. There is also the possibility to turn off presence detection and keep the lights off. Preventing the lights from being turned on due to presence detection can be interesting during periods of intense daylight or at times when the user wants to keep the environment dark, such as while sleeping or watching a movie.

As the lighting and presence sensor may be located at an hard to reach place (such as at the ceiling of a room), the application was divided into two devices, which communicate via Bluetooth. The transmitter (TX) is composed by a switch, and the receiver device (RX) is composed by the presence sensor and the lighting module, which is responsible for lighting the environment and can be composed of any type of lighting, such as LEDs or lamps.

We chose to use Bluetooth to facilitate a possible replacement of the switch module with a mobile application that runs on a cell phone or tablet. Both devices use the HC-05 Bluetooth module with a breakout board for communication. It has 3 energy saving modes, as shown in Table 2. These modes are controlled automatically by the HC-05. The breakout board LED remains flashing throughout the application, which justifies the variation in the module's energy consumption.

	Power consumption (mA)	Duration
Waiting paring	41-43	Until the module is paired
Active mode - TX, RX and right after paring	18-23	5s
Sleeping	2.8-8	Until it loses paring, or finish transmitting or receiving

Table 2: HC-05 power consumption

5.1.1 TX application

The TX device uses a switch, and remains idle most of the time waiting for a user interaction to send a message to the RX. When the switch is turned off, the TX sends the message “0” to disable the presence detection and turn off the lighting. When it’s turned on, it sends the message “1” so the RX starts detecting presence again.

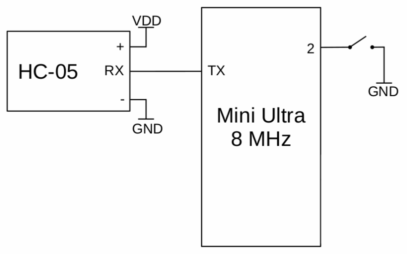


Figure 17: Simplified schematic of TX device

```
const int lightSwitchPin = 2;
boolean lightSwitchState;
boolean lastLightSwitchState;

void setup() {
  Serial.begin(9600);
  pinMode(lightSwitchPin, INPUT_PULLUP);
}

void loop() {
  lightSwitchState = digitalRead(lightSwitchPin);

  if (lightSwitchState != lastLightSwitchState) {
    if (lightSwitchState == HIGH){
      Serial.print("1");
    } else if (lightSwitchState == LOW){
      Serial.print("0");
    }
  }

  lastLightSwitchState = lightSwitchState;
}
```

Listing 12: Smart lighting application - Arduino implementation

Figure 17 presents a simplified schematic of the device circuit. The switch is connected to the MCU via pin 2, which allows the use of the external interrupt `INT0`. We used the MCU’s internal pull-up resistors to ensure that pin 2 receives a high value if the switch is open.

The Arduino implementation is shown in Listing 12. In this implementation, the application is always polling and checking whether the switch is on or off (ln 11 and 13) to send a message via Bluetooth (ln 14-18). The communication between the Arduino and the HC-05 module is done via UART serial communication (ln 6, 15 and 17).

Listing 13 shows another way to implement the same application in Arduino. Instead of always keeping the device active, we put its MCU in the deepest sleep mode (ln 15). It only exits this mode if it receives an external interrupt that indicates a change in the switch pin value. When coming out of the sleep mode, the application can then check if the switch has been turned on or off (ln 17) and send the corresponding message to the Bluetooth module (ln 19 and 21). It was also necessary to add a delay (ln 24) to wait for the message to be sent via serial before the MCU goes back into sleep mode.

```
 1 #include "LowPower.h"
 2
 3 const int lightSwitchPin = 2;
 4 boolean lightSwitchOn;
 5
 6 void lightSwitchChangedISR() {}
 7
 8 void setup() {
 9   Serial.begin(9600);
10   pinMode(lightSwitchPin, INPUT_PULLUP);
11 }
12
13 void loop() {
14   attachInterrupt(digitalPinToInterrupt(lightSwitchPin),
         lightSwitchChangedISR, CHANGE);
15   LowPower.powerDown(SLEEP_FOREVER, ADC_OFF, BOD_OFF);
16
17   lightSwitchOn = digitalRead(lightSwitchPin);
18   if (lightSwitchOn) {
19     Serial.print("1");
20   } else {
21     Serial.print("0");
22   }
23
24   delay(50); // serial await
25 }
```

Listing 13: Smart lighting application - Arduino implementation with manual power saving

The implementation in Céu (Listing 14) waits, at line 6, for a `INT0` input event, which indicates that the switch has changed state. At this point, the application reaches an idle state and is automatically put into the deepest sleep mode. After a change happens, the application can go to the next line, where the pin value is read (ln 8) and the message is sent to the Bluetooth module via UART (ln 18).

```
 1 #include "usart.ceu"
 2 #include "int0.ceu"
 3 #include "string.ceu"
 4 
 5 loop do
 6     await INT0;
 7     do
 8         var high/low v = call INT0_Get();
 9 
10         spawn USART_Init(9600);
11 
12         var[2] byte str = [];
13         if (v == high) then
14             call String_Append_STR(&str, "1");
15         else
16             call String_Append_STR(&str, "0");
17         end
18         await USART_Tx(&str);
19     end
20 end
```

Listing 14: Smart lighting application - Céu implementation

Table 3 presents the power consumption of each implementation. We observed that the implementation in Céu had the same consumption of the implementation in Arduino that saves energy manually. This represents that the language Céu was able to apply the deepest sleep mode possible during the idle period.

	Power consumption (mA)			
	Arduino	Power-saving Arduino	Céu	Duration
Waiting pairing	44-46	41-43	41-43	Until the Bluetooth is paired
After paired	22-26	18-23	18-23	5s
Waiting for switch change	6.6-12	2.9-6	2.9-6	Until the user interacts with the switch (2 hours)
TX	22-26	22-26	22-26	5s

Table 3: Power consumption of the smart lighting TX application

To calculate the average consumption of each implementation, we disregarded the “Waiting pairing” and “After paired” states because we understand that these states would only happen in an initial moment of the application. We also assume that the user interacts with the application every two hours. With that, we obtained an average consumption of 9.3 mA for the Arduino implementation and 4.46 mA for the Céu implementation. Considering a 2600 mAh battery, the application in Céu could execute for 24 days, while the one in Arduino (Listing 12) would last 11 days.

During the analysis of memory usage, we observed that the implementation in Céu consumes about 29% of the device's total memory while Arduino implementations consumed less than 8% (Table 4). As discussed at Chapter 3, we believe that Céu compiler can be optimize for memory usage. To illustrate it, we perform a manual memory optimization by removing unused code, which reduced the Céu code size from 9140 bytes to 6114 bytes, resulting of 19.6% of the device's total memory instead of 29%.

	ROM	RAM	Total
Arduino	6.4 %	9.3 %	6.5 %
Power-saving Arduino	7.2 %	9.3 %	7.3 %
Céu	29.7 %	18.4 %	29 %
Céu with manual memory optimization	19.9 %	15.3 %	19.6 %

Table 4: Memory usage of the smart lighting TX application

Regarding code readability, we can mention that the Arduino implementations used global variables to store the current and previous state of the switch. Managing states introduces more complexity to the code, making it more difficult to read and maintain. Furthermore, the very use of global variables may impair code readability, as discussed in Chapter 3. In the Céu implementation, however, the use of global variables was not necessary. It was also not necessary to store the last state of the switch, since the synchronous reactive model of the Céu language already allows the identification of changes in the switch without the need for state variables.

5.1.2 RX application

As discussed, the RX application turns on the lighting if a presence is detected, and turns it off after a timeout (Figure 18). The application also waits, in parallel, to receive a message via Bluetooth. If the message is “0”, the lights will remain off until a “1” message is received and the application goes back to monitoring presence in the environment.

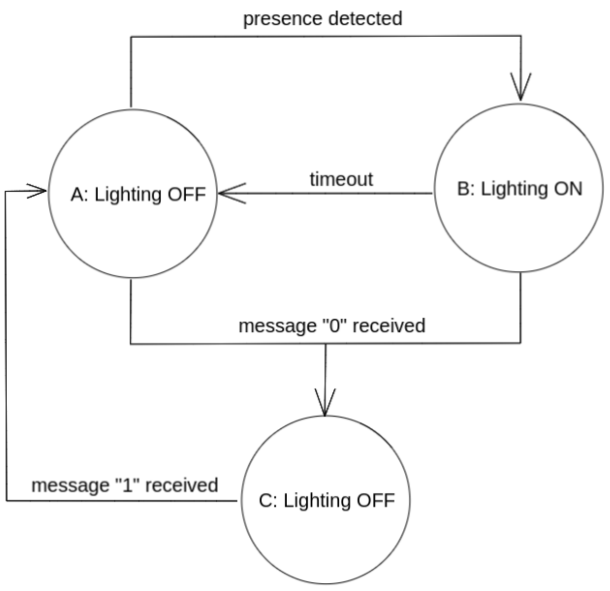


Figure 18: RX application states

We used the ky-036 sensor to simulate presence detection behavior and a LED to represent the lighting module[11]. Figure 19 presents a simplified schematic of the device. The ky-036 sensor is powered by the MCU digital output pin 7, and informs the application if a presence is detected or not using pin 2.

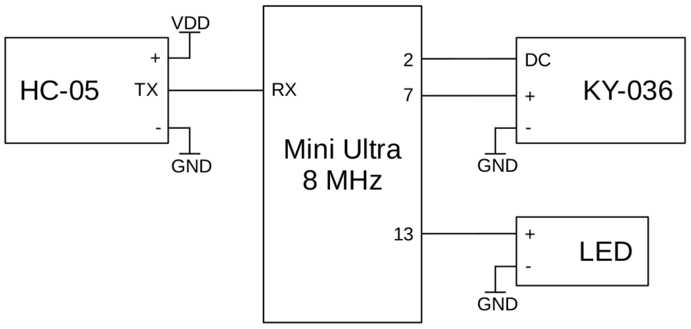


Figure 19: Simplified schematic of RX device

A simplified version of the Arduino implementation is presented in Listing 15. It uses polling to check if there is a message to be read (ln 4-12), if the LED has to be kept off regardless of presence detection (ln 12-15) and if any presence was detected (ln 15-23). The sequential and synchronous structure encouraged by the Arduino language does not prove to be a good choice for this application due to its asynchronous nature (checking a sensor **while** waiting for a message). To implement the asynchronous behavior it was

[11]Alternatively we could use a digital presence detection sensor like Seeed's PIR motion sensor instead of the ky-036 touch sensor and a relay module instead of the LED.

necessary to use a global control variable (`presenceDetectionEnabled`) and decision structures, improving code complexity and decreasing its writability and readability.

```
boolean presenceDetectionEnabled = true;

void loop() {
  if (Serial.available() > 0) {
    String incomingByte = Serial.readString();
    if (incomingByte.equals("1")) {
      presenceDetectionEnabled = true;
    } else if (incomingByte.equals("0")) {
      presenceDetectionEnabled = false;
      digitalWrite(ledPin, LOW);
    }
  } else if (!presenceDetectionEnabled){
    // always keeps the LED off if the last message received was 0
    digitalWrite(ledPin, LOW);
  } else {
    // turn the LED on if a presence is detected
    boolean presenceDetected = digitalRead(sensorDataPin);
    if (presenceDetected) {
        digitalWrite(ledPin, HIGH);
        delay(3000);
        digitalWrite(ledPin, LOW);
    }
  }
}
```

Listing 15: Smart lighting application RX - Arduino implementation

In the Céu implementation (Listing 16) it was not necessary to use control variables or conditional structures. The `par/or` allowed the application's parallel behavior to be implemented directly.

```
output high/low OUT_13;

spawn USART_Init(9600);

loop do
   par/or do
      await USART_Rx_Str("0"); // waits until receiving the value 0
      emit OUT_13(low);
   with
      loop do
         do
            var& DSensor_INT0 mysensor = spawn DSensor_INT0(7, _, 0);
            var high/low v = await mysensor.change until v==high;
         end

         emit OUT_13(high);
         await 3s;
         emit OUT_13(low);
      end
   end

   await USART_Rx_Str("1"); // waits until receiving the value 1
end
```

Listing 16: Smart lighting application RX - Céu implementation

The Arduino implementation with energy saving is divided in three Listings (17, 18, and 19) that are responsible for the application startup routine, the ISR that is executed when a message is received and the application loop, respectively. Lines 2 to 7 of Listings 17 configure the serial communication between the MCU and HC-05 Bluetooth module and enable interrupts. This is a hard to read code that requires hardware knowledge and register manipulations.

The main change to enable power saving was to put the MCU in sleep mode in line 3 of Listing 19. The MCU can be woken from this sleep mode when an INT0 interrupt occurs (enabled on line 2) or when a message is received via serial. To make this last case possible, we inform the parameter `USART0_ON` for the function that puts the MCU in sleep mode, which indicates that the hardware related to the USART communication must be kept on during sleep.

```
void setup() {
  UCSR0A = 1 << U2X0;
  UBRR0H = (BAUD_USART(9600)>>8); // set baud rate
  UBRR0L = (BAUD_USART(9600));
  UCSR0C = (1<<USBS0)|(3<<UCSZ00); // 8data, 2stop-bit
  UCSR0B = (1<<RXEN0) | (1<<TXEN0) // enable RX/TX
         | (1<<RXCIE0) | (1<<TXCIE0); // enable interrupts

  pinMode(sensorPowerPin, OUTPUT);
  digitalWrite(sensorPowerPin, HIGH); // turn sensor ON

  pinMode(sensorDataPin, INPUT_PULLUP);

  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, LOW);
}
```

Listing 17: Smart lighting application RX - Arduino with manual energy saving - setup function

A second modification was to manually turn off the presence sensor when not in use. In the Céu implementation, the driver d-sensor already turns off the sensor based on its lexical scope. Finally, the application puts the MCU in sleep mode while waiting three seconds (ln 21-22) using the watch dog timer to control the passage of time. The MCU can also be woken up via serial.

```
ISR(USART_RX_vect) {
  char data = UDR0;
  if (data == '1') {
    presenceDetectionEnabled = true;
  } else if (data == '0') {
    presenceDetectionEnabled = false;
  }
  usartReceived = true;
}
```

Listing 18: Smart lighting application RX - Arduino with manual energy saving - ISR

```
void loop() {
  attachInterrupt(digitalPinToInterrupt(sensorDataPin),
     presenceChangedISR, RISING);
  LowPower.idle(SLEEP_FOREVER, ADC_OFF, TIMER2_OFF, TIMER1_OFF,
     TIMER0_OFF, SPI_OFF, USART0_ON, TWI_OFF);

  if (usartReceived) {
    if (presenceDetectionEnabled) {
      digitalWrite(ledPin, HIGH);
      digitalWrite(sensorPowerPin, LOW);
    } else {
      digitalWrite(ledPin, LOW);
      digitalWrite(sensorPowerPin, HIGH);
    }

    usartReceived = false;
  } else {
    detachInterrupt(0);

    digitalWrite(sensorPowerPin, LOW); // turn sensor off
    digitalWrite(ledPin, HIGH); // turn LED on

    LowPower.idle(SLEEP_2S, ADC_OFF, TIMER2_OFF, TIMER1_OFF,
       TIMER0_OFF, SPI_OFF, USART0_ON, TWI_OFF);
    LowPower.idle(SLEEP_1S, ADC_OFF, TIMER2_OFF, TIMER1_OFF,
       TIMER0_OFF, SPI_OFF, USART0_ON, TWI_OFF);

    digitalWrite(ledPin, LOW);
    digitalWrite(sensorPowerPin, HIGH);
  }
}
```

Listing 19: Smart lighting application RX - Arduino with manual energy saving - loop

Céu implementation used 43% of the total device memory while the Arduino implementations used a maximum of 12% (see Table 5). Our manual memory optimization decreased the 43% memory usage to 31.7%. From a code readability point of view, Céu implementation has 32 fewer code lines when compared to the power-saving Arduino implementation, and it does not introduce hardware complexity, register manipulation, energy savings, control variables and ISR into the application code.

	ROM	RAM	Total
Arduino	12%	10%	12%
Power-saving Arduino	5%	0.7%	5%
Céu	44%	22%	43%
Céu with manual memory optimization	31.7%	20.5%	31%

Table 5: Memory usage of the smart lighting RX application

To estimate the average power consumption of this application, we consider the following usage routine:

1. The application remains idle for two hours, in its initial state;
2. A presence is detected and the LED turns on for 3 seconds;
3. The application returns to its initial state;
4. After an hour, it receives the message “0”. The presence detection functionality is disabled and LED is turned off;
5. After an hour, the device receives a “1” message.

Table 6 presents the consumption of each application state considering the usage routine presented[12]. As with the TX module, we disregard the initial states of the HC-05 Bluetooth module. The most power-consuming states are those in which HC-05 is active receiving a message. The energy savings achieved in these states are due to the ky-036 turning off. In other states, power savings are due to the MCU’s sleep modes.

		Power consumption (mA)			
Figure 18 state	Application state	Arduino	Power-saving Arduino	Céu	Duration
A	Idle	10.5	7.5	7.5	2hrs
B	LED ON because of a presence detection	15.75	10	10	3s
A	Idle	10.5	7.5	7.5	1h
-	Receiving message “0”	26	21	21	5s
C	LED OFF	10.5	5.6	5.6	1h
-	Receiving message “1”	26.65	23.5	23.5	5s

Table 6: Power consumption of the smart lighting RX application

[12]As with the TX module, the consumption of each state is not static due to the HC-05 Bluetooth module. However, to simplify data visualization, Table 6 shows the average consumption for each state.

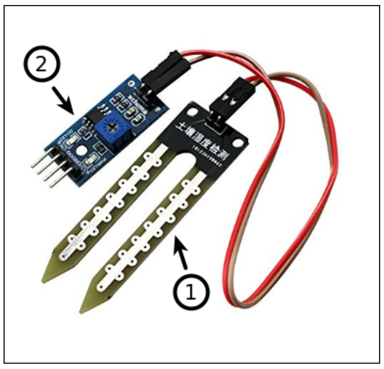
The implementation in Céu consumes approximately 33% less energy than the Arduino implementation, and has a battery life of fifteen days if we consider a 2600mAh battery. The Arduino implementation would execute for ten days.

Céu implementation saved almost as much energy as the implementation in Arduino with manual power management, despite turning off different MCU elements during state B (Figure 18). While in this state, the MCU goes into sleep mode leaving the USART hardware and a timer on. The implementation in Céu and Arduino use different timers (respectively the watch dog time and timer 1). Even both implementations applying the same sleep mode, keeping different hardware on generate a difference in the consumption, however, they are still very similar, with the difference of some microamps.

5.2 Humidity and moisture sensor device

The second application collects the air humidity and the soil moisture and transmits it via radio. This type of application is used in agriculture and several works propose the implementation of cheap and energy efficient alternatives [40] [41] [42]. We used these works as support for choosing each hardware to use here.

The sensor used to obtain air humidity is the DHT11, already presented in Section 4.3. For the soil moisture, we used the FC-28 (Figure 20(a)), which is composed by two pieces: a sensor probe and a comparator module. The sensor probe (indicated by number 1 in the figure) must remain buried in the soil and measures the potential difference that varies depending on moisture.



(a) FC-28 soil moisture sensor

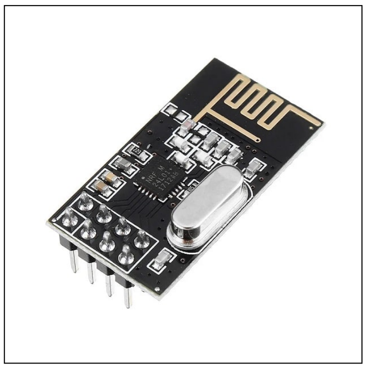
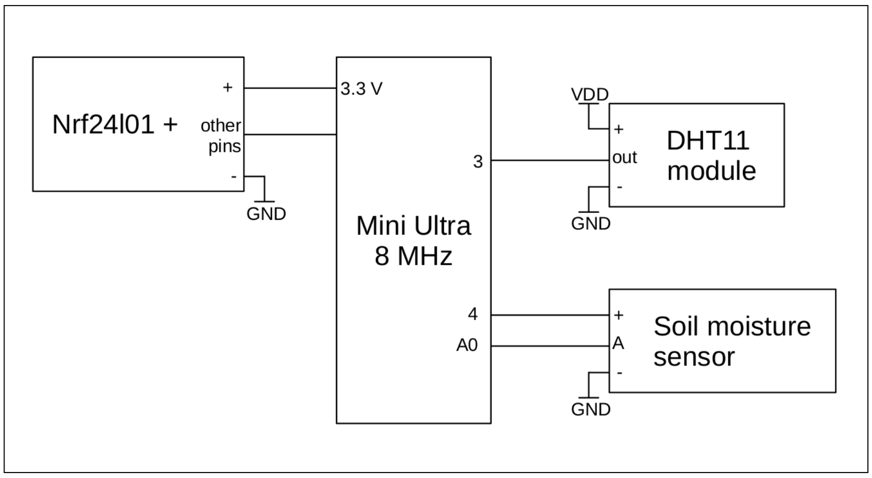

(b) nRF24l01+

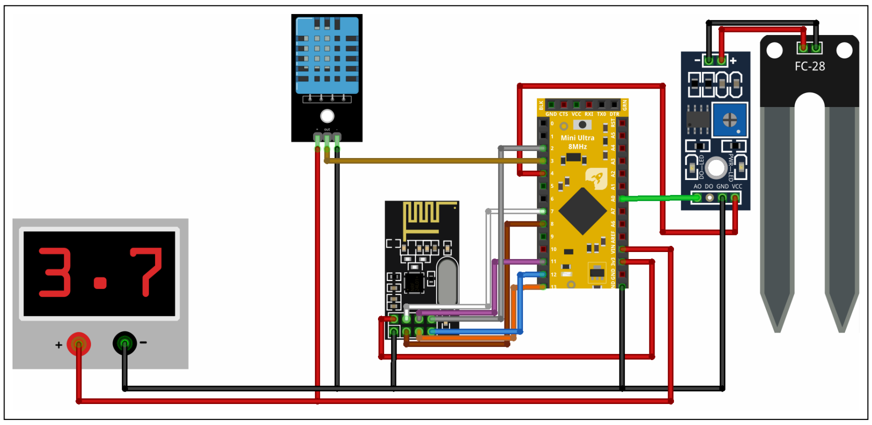
Figure 20: Hardware used in the second application

The second piece (indicated by number two) is a comparator module that checks the value measured by the sensor probe and generates a digital signal based on a predefined threshold. This threshold can be customized using the module's potentiometer. In addition to the digital output, the module also has an analog output. For this application, we only used the analog pin. For transmission we used the nRF24l01+ transceiver (Figure 20(b)).

Figure 21(a) presents a simplified schematic of the device, and Figure 21(b) shows a more detailed view of the circuit. The humidity sensor is powered directly by 3.7 volts as well as the Mini Ultra 8Mhz. However, 3.7 volts could damage the nRF24l01+ module, as its recommended maximum voltage is 3.6 volts [15]. Therefore, the radio is powered by a prototyping board pin that provides 3.3 volts. At last, the moisture sensor is powered by a digital output pin, which allows the application to turn it on and off.



(a) Simplified schematic



(b) Detailed circuit

Figure 21: Circuit of humidity and moisture sensor device

Listings 20, 21 and 22 present the three implementations of the application. All of them have had parts omitted for simplicity. The omitted parts are responsible for importing libraries/drivers and configuring the nRF24l01+ radio and the DHT11 sensor. The three implementations turned out to be very similar, and all can be considered easy to read. This can be justified due to the sequential nature of the application.

The Arduino implementation uses the blocking operation `delay` to wait two seconds and keeps the sensor on all the time. The energy-saving implementation, on the other hand, manually keeps the sensor off when not in use, setting its power pin as `HIGH` and `LOW` manually. The two-second delay has also been optimized: instead of using `delay`, we put the Arduino into its deepest sleep mode ("power down") for two seconds. Finally, the Céu implementation uses the a-sensor driver to keep the analog sensor on only during its use, transparently. Furthermore, the language puts the MCU in sleep mode during the two-second waiting.

```
void loop() {
  delay(2000);

  float hum = dht.readHumidity();

  int sensorValue = analogRead(A0);

  char str[9];
  sprintf(str, "%4d%4d", (int)(hum*100), sensorValue);
  radio.write(&str, 9);
}
```

Listing 20: Humidity and moisture sensor node - Arduino implementation

```
void loop() {
  LowPower.powerDown(SLEEP_2S, ADC_OFF, BOD_OFF);

  float hum = dht.readHumidity();

  digitalWrite(4, HIGH);
  delay(50);
  int sensorValue = analogRead(A0);
  digitalWrite(4, LOW);

  char str[9];
  sprintf(str, "%4d%4d", (int)(hum*100), sensorValue);
  radio.write(&str, 9);
}
```

Listing 21: Humidity and moisture sensor node - Arduino implementation with manual power saving

```
loop do
    await 2s; // replace with Wdt(2000); to use the watchdog timer
    call DHT_Read(3,&temp,&hum);

    var int asensor_value;
    do
        asensor_value = await ASensor_Get(_A0, 4, 50);
    end

    var[9] byte str = [];
    call String_Format(&str, hum, asensor_value);
    await NRF24L01_Tx(&nrf, &str);
end
```

Listing 22: Humidity and moisture sensor node - Céu implementation

The consumption of each implementation is shown in Table 7. We could observe that the second implementation (in Arduino with energy savings) managed to save more power than the implementation in Céu during the waiting period. This happens because the Céu wall clock driver uses TIMER 1, that is not available in the MCU deepest sleep mode (power down mode). This way, the Céu power management driver puts the MCU in the deepest sleep mode that still keeps TIMER 1 on: the "idle" mode.

The Arduino implementation uses the watchdog timer, which remains on while the microcontroller is in "power-down" mode. Alternatively, we could use the Céu watch dog timer driver to put the MCU into the "power down" mode during the two second waiting. However, this brings energy consumption concerns to the application layer. To avoid this, future works could try to use the watchdog timer transparently through the wall clock driver.

	Power consumption (mA)			
	Arduino	Power-saving Arduino	Céu	Duration
Waiting	9	2.4	4.16	2s
Sensor reading	9.87	9.87	5.2	25ms or 75ms
Radio TX	20.3	18.6	16.3	1ms

Table 7: Power consumption of the humidity and moisture sensor node

The implementation in Céu occupied 82.9% of the device memory, against 17% occupation of Arduino implementation. This increase can limit the adoption of the Céu language in memory-constrained projects. It can also limit the implementation of more complex communication protocols, as they require the construction of bigger drivers.

	ROM	RAM	Total
Arduino	18%	3.80%	17%
Céu	85.7%	40.7%	82.9%
Céu with manual memory optimization	85.1%	40.7%	82.4%

Table 8: Memory usage of the humidity and moisture sensor node

Overall, we observed a higher memory usage by Céu applications that use a lot of drivers. This application, for example, uses the wclock, dht11, asensor, int0, nrf24l01 drivers directly, in addition to spi and asensor drivers indirectly[13]. The nrf24l01 and spi drivers are used to send the message via radio, and among the drivers used, they have the most impact on memory consumption. To illustrate it, removing the radio communication

[13]They are used by nrf24l01 and asensor drivers, respectively

from the application resulted at 37% of program storage space usage (ROM) and 20% of dynamic memory usage (RAM).

Finally, our memory optimization was not very effective, saving only 172 bytes. The savings were due to the removal of unused `code/calls` from the wclock driver. We believe that deeper optimizations can be performed in future works.

CONCLUSION

This dissertation evaluated the use of the programming language Céu and its power-management mechanism in the development of embedded applications. This analysis was done with the implementation of two embedded software applications: a smart lighting system, and a moisture collector node. They can be considered simple applications, but they illustrate common patterns in embedded software applications. Both of them were implemented in Céu and Arduino, a widely used programming language in the embedded software community. The implementations were then compared in relation to their power consumption, memory usage and code readability.

It was possible to verify that applications that have some concurrent behavior, such as blinking two LEDs at different frequencies, become easier to read and write when implemented in Céu. This improvement was observed in the receiver module of the smart lighting application, which waits to receive a message via Bluetooth while waiting for a sensor to detect a presence. We also noted that all applications written in Céu used more memory than the ones written in Arduino. The biggest increase was observed in the moisture collector application, where the implementation in Céu uses 82.9% of device's memory, while the implementation in Arduino only used 17%. This increase can limit the language adoption in the development of resource-constrained devices. It can also hamper the implementation of more complex drivers, such as radio drivers with complex network protocols.

Regarding power savings, in both applications the language Céu was able to identify the idle periods and apply a power saving mode, making the application save energy transparently. Additionally, the developed drivers for digital and analog sensors kept the sensors off when not in use, saving even more energy. In the moisture sensor application, however, the Arduino was not put into the deepest sleep mode when using the wall clock time driver. Alternatively, we could use the watchdog timer driver for optimal savings.

Future works include memory usage optimization, development of drivers for more peripherals (such as RFID-RC522), and use the watchdog timer transparently, allowing optimal energy savings in a completely transparent way. It's also important to mention that Céu-Arduino lacks documentation, which also hinders the adoption of the language.

REFERÊNCIAS

[1] STROUSTRUP, B. Abstraction and the C++ machine model. In: SPRINGER. *International Conference on Embedded Software and Systems.* [S.l.], 2004. p. 1–13.

[2] EMBEDDED.COM. *Using power analysis to optimize battery life in IoT devices.* Abril 2019. Disponível em: <https://www.embedded.com/using-power-analysis-to-optimize-battery-life-in-iot-devices/>. Acesso em: 7 out. 2020.

[3] SCHWARTZ, M. *How to Run an Arduino for Years on a Battery.* Julho 2020. Disponível em: <https://makecademy.com/arduino-battery>. Acesso em: 7 out. 2020.

[4] SANT'ANNA, F. et al. Transparent standby for low-power, resource-constrained embedded systems: a programming language-based approach (short wip paper). In: ACM. *ACM SIGPLAN Notices.* [S.l.], 2018. v. 53, n. 6, p. 94–98.

[5] EDNA/IEA. *Energy Efficiency of Internet of Things - Policy Opinions.* Julho 2016. Disponível em: <https://www.iea-4e.org/document/388/energy-efficiency-of-the-internet-of-things-policy-options>. Acesso em: 23 jul. 2019.

[6] IEA. *G20 Energy Efficiency Action Plan: Networked Devices.* Disponível em: <https://www.iea-4e.org/projects/g20-energy-efficiency-action-plan-networked-devices>. Acesso em: 23 jul. 2019.

[7] SANT'ANNA, F. *Safe System-level Concurrency on Resource-Constrained Nodes with Céu.* Orientador: Roberto Ierusalimschy. Co-orientadora: Noemi de La Roque Rodriguez. 2013. 86f. Tese (Doutorado) - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2013.

[8] BRANCO, A. et al. Terra: Flexibility and safety in wireless sensor networks. *ACM Trans. Sen. Netw.*, ACM, New York, NY, USA, v. 11, n. 4, p. 59:1–59:27, set. 2015. ISSN 1550-4859. Disponível em: <http://doi.acm.org/10.1145/2811267>.

[9] SANT'ANNA, F. et al. Safe system-level concurrency on resource-constrained nodes. In: ACM. *Proceedings of SenSys'13.* [S.l.], 2013.

[10] SANT'ANNA, F. Structured synchronous reactive programming for game development-case study: On rewriting Pingus from C++ to céu. In: IEEE. *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. [S.l.], 2018. p. 240–24009.

[11] SANT'ANNA, F. et al. *Structured Reactive Programming with Céu*. 2014. Workshop on Reactive and Event-based Languages & Systems (REBLS'14).

[12] SANTOS, R. et al. Céu-Media: Local Inter-Media Synchronization Using Céu. In: *Proceedings of WebMedia'16*. New York, NY, USA: ACM, 2016. p. 143–150. ISBN 978-1-4503-4512-5. Disponível em: <http://doi.acm.org/10.1145/2976796.2976856>.

[13] SANT'ANNA, F.; SZTAJNBERG, A. *Where Do Events Come From? — Reactive and Energy-Efficient Programming From The Ground Up*. 2018. Workshop on Reactive and Event-based Languages & Systems (REBLS'18).

[14] DHT-11 datasheet. Disponível em: <https://www.filipeflop.com/img/files/download/Datasheet_DHT11.pdf>. Acesso em 8 ago. 2021.

[15] SEMICONDUCTOR, N. *nRF24l01+ datasheet*. Disponível em: <https://www.sparkfun.com/datasheets/Components/SMD/nRF24L01Pluss_Preliminary_Product_Specification_v1_0.pdf>. Acesso em: 21 ago. 2021.

[16] SCHOEFFLER, M. *How to use the GY-521 module (MPU-6050 breakout board) with the Arduino Uno*. Disponível em: <https://mschoeffler.com/2017/10/05/tutorial-how-to-use-the-gy-521-module-mpu-6050-breakout-board-with-the-arduino-uno/>. Acesso em 25 out. 2021.

[17] INVENSENSE. *MPU-6000/MPU-6050 Register Map and Descriptions*. Disponível em: <https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf>. Acesso em: 8 ago. 2019.

[18] BAEK, W.; CHILIMBI, T. M. A framework for supporting energy-conscious programming using controlled approximation. In: ACM. *ACM Sigplan Notices*. [S.l.], 2010. v. 45, n. 6, p. 198–209.

[19] FLINN, J.; SATYANARAYANAN, M. Managing battery lifetime with energy-aware adaptation. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 22, n. 2, p. 137–179, 2004.

[20] BARKER, P.; HAMMOUDEH, M. A survey on low power network protocols for the internet of things and wireless sensor networks. In: ACM. *Proceedings of the International Conference on Future Networks and Distributed Systems*. [S.l.], 2017. p. 44.

[21] FRÖHLICH, A. A. A comprehensive approach to power management in embedded systems. *International Journal of Distributed Sensor Networks*, SAGE Publications Sage UK: London, England, v. 7, n. 1, p. 807091, 2011.

[22] SANT'ANNA, F. Ceu: A reactive language for wireless sensor networks. In: *Proceedings of the ACM SenSys*. [S.l.: s.n.], 2011. v. 11.

[23] GAUR, P.; TAHILIANI, M. P. Operating systems for iot devices: A critical survey. In: IEEE. *2015 IEEE Region 10 Symposium*. [S.l.], 2015. p. 33–36.

[24] SITE do Sistema Operacional Contiki. Disponível em: <http://www.contiki-os.org/>. Acesso em: 8 ago. 2019.

[25] DUNKELS, A. The contikimac radio duty cycling protocol. Swedish Institute of Computer Science, 2011.

[26] BERRY, G. Real time programming: Special purpose or general purpose languages. 1989.

[27] SANT'ANNA, F. *Céu programming language documentation*. 2018. Disponível em: <https://ceu-lang.github.io/ceu/out/manual/v0.30>. Acesso em: 19 out. 2020.

[28] ARDUINO. *Arduino language reference*. Disponível em: <https://www.arduino.cc/reference/en/>. Acesso em: 26 jul. 2019.

[29] SIMAS, G. *Aplicação em Sistemas Distribuídos utilizando biblioteca e driver próprios, baseados em interrupções desenvolvido em Céu para o microcontrolador Arduino*. Orientadora: Ana Lúcia de Moura. 2018. 39f. Projeto Final (Graduação de Engenharia da Computação) - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

[30] BOSTON, E. *Estimating and measuring power in embedded systems.* September 2010. Disponível em: <https://youtu.be/Ryib2tT2-bA>. Acesso em: 19 out. 2020.

[31] TECNOLOGIES, K. *Device current waveform analyzer.* Disponível em: <https://www.keysight.com/en/pc-2633352/device-current-waveform-analyzers?cc=USlc=eng>. Acesso em: 27 out. 2020.

[32] KEITHLEY. *Determining power consumption and battery life in low power portable IoT devices webinar.* April 2020. Disponível em: <https://www.tek.com/webinar/determining-power-consumption-and-battery-life-in-low-power–portable-iot–devices-webinar>. Acesso em: 20 out. 2020.

[33] CORGITRONICS. *Power use of an Arduino and nRF24L01 using a Keithley 2208S.* Fevereiro 2015. Disponível em: <https://www.youtube.com/watch?v=8ECNJPVHqe0t=293s>. Acesso em: 13 out. 2021.

[34] FORCETRONICS. *Reducing the power consumption of the nRF24L01 transceiver.* Maio 2015. Disponível em: <https://www.youtube.com/watch?v=MvjpmsH2wKIlist=WLindex=28>. Acesso em: 13 out. 2021.

[35] ARDUINO. *Arduino foundations, memory.* Disponível em: <https://www.arduino.cc/en/Tutorial/Foundations/Memory>. Acesso em: 1 jan. 2022.

[36] OLIVEIRA, D. et al. Evaluating code readability and legibility: An examination of human-centric studies. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME).* [S.l.: s.n.], 2020. p. 348–359.

[37] FAKHOURY, S. et al. Improving source code readability: Theory and practice. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC).* [S.l.: s.n.], 2019. p. 2–12.

[38] PRECHELT, L. An empirical comparison of seven programming languages. *Computer*, v. 33, n. 10, p. 23–29, 2000.

[39] BUSE, R. P.; WEIMER, W. R. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, v. 36, n. 4, p. 546–558, 2010.

[40] CARVALHO, M. S. de. *Sensor para monitoramento de umidade do solo utilizando energia solar*. 2016. 46 f. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software) - Curso de Engenharia de Software, Universidade Federal do Ceará, Quixadá, 2016.

[41] DINIZ, A. M. *Sistema automatizado de aquisição, em tempo real, de umidade e temperatura do solo na irrigação*. Orientador: Márcio Antônio Vilas Boas. 2017. 60 f. Tese (Doutorado) - Programa de Pós-Graduação em Engenharia Agrícola, Universidade Estadual do Oeste do Paraná, Cascavel, 2017.

[42] AZETA, J. et al. Design of a wireless communication drip irrigation system using nRF24L01 technology.