IV
Demo applications

In this chapter, we present two demos that explore the high-level and safety capabilities of CÉU described in the previous chapter. Our goal is to present full commented applications that help understanding and getting familiar with the language. The applications are somewhat simple (70 and 170 lines), but complete enough to expose the programming techniques promoted by CÉU.

The first demo targets commercially available 16-bit WSN nodes, such as *micaZ* and *telosb*[1]. The second demo uses the Arduino open-source platform[2], in order to experiment with custom third-party hardware. Both platforms have low processing power and memory capacity (16Mhz CPU, 32Kb Flash, and 4Kb SRAM), showing that CÉU is applicable to highly constrained platforms.

IV.1 WSN ring

In the first demo, we implement a fixed-ring topology with N motes placed side-by-side which should all follow the same behavior: receive a message with an integer counter, show it on the LEDs, wait for 1 second, increment the counter, and forward it to the mote on its right. Because the topology constitutes a ring, the counter will be incremented forever while traversing the motes. If a mote does not receive a message within 5 seconds, it should blink the red LED every 500 milliseconds until a new message is received. The mote with *id=0* is responsible for initiating the process at boot time and recovering the ring from failures. On perceiving a failure, it should wait for 10 seconds before retrying the communication.

Figure IV.1 implements the communicating trail, which continuously receives and forwards the messages. The code is an endless loop that first awaits a radio message (line 2), gets a pointer to its data buffer (line 3), shows the received counter on the LEDs (line 4), and then awaits 1s (line 5) before

[1] `http://www.xbow.com`
[2] `http://arduino.cc`

```
1  loop do
2      var _message_t* msg = await RADIO_RECEIVE;
3      var int* cnt = _Radio_getPayload(msg);
4      _Leds_set(*cnt);
5      await 1s;
6      *cnt = *cnt + 1;
7      finalize
8          _Radio_send((_NODE_ID+1)%N, msg);
9      with
10         _Radio_cancel(msg);
11     end
12     await RADIO_SENDDONE;
13 end
```

Figure IV.1: Communicating trail for the WSN ring.

incrementing the counter in the message (line 6) and forwarding it to the next mote (line 7-12).

The program uses several services provided by the underlying operating system ([23]), which are all non-blocking *C* functions for LEDs and radio manipulation.

The finalization block (lines 7-11) ensures that regardless of how the communicating trail is composed with the rest of the application (and eventually aborted by it), the `msg` buffer will be safely released while waiting for a `RADIO_SENDDONE` acknowledge from the radio driver.

Because this code does not handle failures, it is straight to the point and easy to follow. Actually, this is the final code for this task, as error handling is placed in a parallel trail.

To handle failures, we define in Figure IV.2 a monitoring trail (lines 4-22) in parallel with the communicating trail. Lines 8 to 20 describe the network-down behavior. After 5 seconds of inactivity are detected in the sub-trails in parallel (lines 6 and 8), two new activities run in parallel: one that retries communication every 10 seconds by signaling the internal event `retry` (lines 8-11); and another that blinks the red LED every 500 milliseconds (lines 13-17).

The trick to restore the normal behavior of the network is to await event `RADIO_RECEIVE` (line 6) in the `par/or` (line 5) with the network-down behavior to abort it whenever a new message is received. By surrounding everything with a `loop` (line 4), we ensure that the error detection is continuous.

Finally, we implement in Figure IV.3 the initiating/retrying process that sends the first message from mote with *id=0*. Again, we place the code (lines 6-20) in parallel with the other activities. As this process is only handled by the mote with $id = 0$, we start by checking it (line 6). If this is not the case, we simply await forever on this trail (line 19). Otherwise, the `loop` (lines 7-17)

```
 1  par do
 2      <...> // COMMUNICATING TRAIL (previous code)
 3  with
 4      loop do
 5          par/or do
 6              await RADIO_RECEIVE;
 7          with
 8              await 5s;
 9              par do
10                  loop do
11                      emit retry; // only captured by mote 0
12                      await 10s;
13                  end
14              with
15                  _Leds_set(0);   // clear LEDs
16                  loop do
17                      _Leds_led0Toggle();
18                      await 500ms;
19                  end
20              end
21          end
22      end
23  end
```

Figure IV.2: Monitoring trail for the WSN ring.

sends the first message as soon as the mote is turned on (line 12). It then waits for a `retry` emit (line 16) to loop and resend the initial message. Remind that event `retry` is emitted on network-down every 10 seconds (line 10 of Figure IV.2).

The static analysis of Céu correctly warns about concurrent calls to `_Radio_send` (line 12) *vs.* `_Leds_set` and `_Leds_led0Toggle` (lines 15,17 of Figure IV.2), which all execute after the program detects 5 seconds of inactivity (line 6 of Figure IV.2). However, because these functions affect different devices (i.e. radio *vs.* LEDs), they can be safely executed concurrently. The following annotation (to be included in the program) states that these specific functions can be called concurrently with deterministic behavior, allowing the program to be compiled without warnings:

```
safe _Radio_send with
     _Leds_set, _Leds_led0Toggle;
```

This example shows how complementary activities in an application can be written in separate and need not to be mixed in the code. In particular, error handling (monitoring trail) need not interfere with regular behavior (communicating trail), and can even be incorporated later. To ensure that parallel activities exhibit deterministic behavior, the Céu compiler rejects harmful concurrent *C* calls by default.

```
par do
    <...> // COMMUNICATING TRAIL
with
    <...> // MONITORING TRAIL
with
    if _NODE_ID == 0 then
        loop do
            var _message_t msg;
            var int* cnt = _Radio_getPayload(&msg);
            *cnt = 1;
            finalize
                _Radio_send(1, &msg);
            with
                _Radio_cancel(&msg);
            end
            await retry;
        end
    else
        await FOREVER;
    end
end
```

Figure IV.3: Retrying trail for the WSN ring.

As a final consideration, we can extend the idea of compositions by combining different *applications* together. In the context of WSNs, it is usually difficult to physically recover motes in a deployed network, and by combining multiple applications in a single image, we can switch their execution remotely via radio. The archetype in Figure IV.4 illustrates this idea. The input event `SWITCH` (line 1) is used to request application switches remotely.[3] Initially, the code behaves as application 1 (lines 7-9), but is also waiting for a `SWITCH` request in parallel (line 5). Whenever a new request occurs, the `par/or` terminates, aborts the running application, and restarts as the requested application. The statement `await FOREVER` (line 13) ensures that a terminating application does not reach the end of the `par/or` and restarts itself.

The same idea can be used to *reboot* a mote remotely, in the case of a strange behavior in an application.

IV.2 Spaceship game

In the next demo, a spaceship game, we control a ship that moves through space and has to avoid collisions with meteors until it reaches the finish line. Although this application is not networked, it is still embedded and reactive, using timers, buttons, and an LCD with real-time feedback. We use an Arduino

[3] We are assuming the existence of an hypothetical high-level event `SWITCH` that abstracts the radio protocol for requests to change the current running application.

```
input int SWITCH;
var int cur_app = 1;
loop do
    par/or do
        cur_app = await SWITCH;
    with
        if cur_app == 1 then
            <...>  // CODE for APP1
        end
        if cur_app == 2 then
            <...>  // CODE for APP2
        end
        await FOREVER;
    end
end
```

Figure IV.4: Retrying trail for the WSN ring.

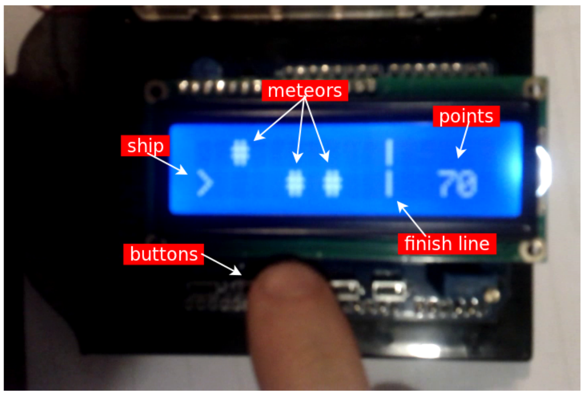

Figure IV.5: The “spaceship” game

connected to a third-party two-row LCD display with two buttons to exhibit and control the spaceship. Figure IV.5 shows the picture of a running quest.

We describe the behavior of the game, along with its implementation, following a top-down approach. The outermost loop of the game, in Figure IV.6, is constituted of `CODE 1`, which sets the game attributes such as globals `code` and `dt`; `CODE 2` with the central game loop; and `CODE 3` with the “game over” animation. Every time the loop is executed, it resets the game attributes (line 5), generates a new map (line 7), redraws it on screen (line 8), and waits for a starting key (line 9). Then, the program executes the main logic of the game (line 11), until the spaceship reaches the finish line or collides with a meteor. Depending on the result held in `win`, the “game over” code (line 13) may display an animation before restarting the game.

The game attributes (`CODE 1` in Figure IV.7) change depending on the result of the previous iteration of the outermost loop. For the first game execution and whenever the spaceship collides with a meteor, variable `win` is

```
 1  var int dt;       // inverse of game speed
 2  var int points;   // number of steps alive
 3  var int win = 0;  // starting the game
 4
 5  loop do
 6      <...> // CODE 1: set game attributes
 7
 8      _map_generate();
 9      _redraw(x, y, points);
10      await KEY;  // starting key
11
12      <...> // CODE 2: the central loop
13
14      <...> // CODE 3: game over
15  end
```

Figure IV.6: Outermost loop for the game.

```
 1      // CODE 1: set game attributes
 2      var int y = 0;          // ship coordinates
 3      var int x = 0;          //  restart every phase
 4
 5      if not win then
 6          dt     = 500;
 7          points = 0;
 8      else
 9          if dt > 100 then
10              dt = dt - 50;
11          end
12      end
```

Figure IV.7: Sets the game attributes.

false, hence, the attributes are reset to their initial values (lines 6-7) Otherwise, if the player reached the finish line, then the game gets faster, keeping the current points (lines 9-11).

The central loop of the game (`CODE 2` in Figure IV.8) moves the spaceship as time elapses and checks whether the spaceship reaches the finish line or collides with a meteor. The code is actually split in two loops in parallel: one that runs the game steps (lines 3-19), and the other that handles input from the player to move the spaceship (lines 21-29). Note that we want the spaceship to move only during the game action, this is why we did not place the input handling in parallel with the whole application.

The game steps run periodically, depending on the current speed of the game (line 4). For each loop iteration, `x` is incremented and the current state is redrawn on screen (lines 5-6). Then, the spaceship is checked for collision with meteors (lines 8-11), and also with the finish line (lines 13-16). In either

```
 1  // CODE 2: the central loop
 2  par/or do
 3      loop do
 4          await (dt)ms;
 5          x = x + 1;
 6          _redraw(x, y, points);
 7
 8          if _MAP[y][x] == '#' then
 9              win = 0;  // a collision
10              break;
11          end
12
13          if x == _FINISH then
14              win = 1;  // finish line
15              break;
16          end
17
18          points = points + 1;
19      end
20  with
21      loop do
22          var int key = await KEY;
23          if key == _KEY_UP then
24              y = 0;
25          end
26          if key == _KEY_DOWN then
27              y = 1;
28          end
29      end
30  end;
```

Figure IV.8: The game central loop.

of the cases, the central loop terminates with `win` set to the proper value, also canceling the input handling activity. The points are incremented before each iteration of the loop (line 18).

To handle input events, we wait for key presses in another loop (line 22) and change the spaceship position accordingly (lines 24, 27). Note that there are no possible race conditions on variable `y` (i.e., lines 6,8 *vs.* 24,27) because the two loops in the `par/or` statement react to different events (i.e., time and key presses).

After escaping the central loop, we run the code of Figure IV.9 for the "game over" behavior, which starts an animation if the spaceship collides with a meteor. The animation loop (lines 6-13) continuously displays the spaceship in the two directions, suggesting that it has hit a meteor. The animation is interrupted when the player presses a key (line 3), proceeding to the game restart. Note the use of the `_lcd` object, available in a third-party *C++* library shipped with the LCD display.

This demo makes extensive use of global variables, relying on the de-

```
1      // CODE 3: game over
2      par/or do
3          await KEY;
4      with
5          if !win then
6              loop do
7                  await 100ms;
8                  _lcd.setCursor(0, y);
9                  _lcd.write('<');
10                 await 100ms;
11                 _lcd.setCursor(0, y);
12                 _lcd.write('>');
13             end
14         end
15     end
```

Figure IV.9: The “game over” behavior for the game.

terministic concurrency analysis guaranteed by the CÉU compiler. We used a top-down approach to illustrate the hierarchical compositions of blocks of code. For instance, the “game over” animation (lines 6-13) is self-contained and can be easily adapted to a new behavior without considering the other parts of the program.