Using Low Power Coprocessors in an FRP Language for Embedded Systems

Go Suzuki[1,a)] Akihiko Yokoyama[1,b)] Sosuke Moriguchi[1,c)]
Takuo Watanabe[1,d)]

Abstract: A low power coprocessor in the microcontroller helps to save total power consumption. While the main processor is in a sleep state, the low power coprocessor can process the inputs and maintain responsiveness. However, inter-processor communication and processor power state management make development more complicated. In this paper, we address this problem by introducing a mechanism to switch a running processor to the functional reactive programming (FRP) language XStorm, which has an abstraction mechanism for modeling stateful behaviors. The proposed mechanism allows us to choose which processor to run in each state. Therefore, the switching of a running processor can be represented as a state transition. Our compiler can absorb differences in processor architectures and automatically generate programs for inter-processor communication and processor state management. As a result, developers can more easily describe system with coprocessors. We describe the proposed mechanism and report an evaluation on the power consumption and time of state transitions.

Keywords: functional reactive programming, state-transition model, embedded systems, low power coprocessor, asymmetric multicore processor

1. Introduction

Some microcontrollers have low power coprocessors with limited functionality and performance in addition to the main processor. For example, ESP32-S3 have the RISC-V Ultra Low Power Coprocessor (ULP coprocessor) [4]. The ULP coprocessor does not support SPI and PWM outputs, and has a limited number of GPIOs, but can use analog-to-digital converters and I^2C. Such a microcontoller reduces the power consumption while maintaining responsiveness by processing inputs in the ULP coprocessor while the main processor is in the sleep-state. However, inter-processor communication and processor power state management make development more complicated.

We extend XStorm [9], a functional reactive programming language for small-scale embedded systems with an abstraction mechanism for modeling stateful behaviors. XStorm can switch the relations between time-varying values depending on the state. This allows to describe embedded systems with stateful behavior by XStorm.

In this paper, we introduce a mechanism to choose which processor to run in each state. We implemented this mechanism for XStorm, and targeted ESP32-S3 with the ULP coprocessor. We show a simplified Theremin example using the proposed mechanism. We also evaluated the power consumption and the overheads of state transitions.

[1] Department of Computer Science, Tokyo Institute of Technology
[a)] gosuzuki@psg.c.titech.ac.jp
[b)] akihiko@psg.c.titech.ac.jp
[c)] chiguri@acm.org
[d)] takuo@acm.org

The rest of this paper is organized as follows. The following section describes XStorm, an FRP language that supports the description of state-dependent behavior, using a motivating example (Theremin). After describing the low-power coprocessor technologies used in this work in Section 3, the proposed language mechanism and its implementation are described in Sections 4 and 5, respectively. The motivating example is used again in these sections. Then, Sections 6 and 7 discuss the evaluation of power consumption and execution time, respectively. Section 8 discusses related work, and Section 9 concludes the paper.

2. XStorm

XStorm [9] is a functional reactive programming language designed for resource-constrained devices such as microcontrollers. It is based on Emfrp [15] and provides an abstraction mechanism for modeling stateful behaviors. This section first describes the concepts of FRP, then the language overview and implementation of XStorm.

2.1 Functional Reactive Programming

A *reactive system* is a computational system that continuously reacts to external inputs. Embedded systems are typical examples of reactive systems. *Functional Reactive Programming* (FRP) is a programming paradigm that describes a reactive system in terms of *time-varying values* (aka *signals*), which abstract values that change over time (e.g., sensor readings) [1]. By describing relations between time-varying values using FRP, we can avoid using techniques such as polling and callbacks often used in programming em-

Listing 1: A Simplified Theremin in XStorm

```
1  func next_volume_mode(v) = (v % 5) + 1
2  switchmodule Theremin {
3    in dist(0) : Int, btn(False) : Bool
4    out frequency : Int, volume : Int,
5        stateLed : Bool  # For Eval. 1
6    shared btn_released(False) : Bool,
7           volume_mode(1) : Int
8    init On
9
10   shared node btn_released = not(btn) && btn@last
11   shared node volume_mode =
12     if btn_released
13       then next_volume_mode(volume_mode@last)
14       else volume_mode@last
15   state On {
16     node dist_avg(0) =
17         (dist * 6 + dist_avg@last * 4) / 10
18     out node frequency = dist_avg
19     out node volume = volume_mode * 20
20     out node stateLed = True
21     switch: if dist_avg>=1500 then Off else Retain
22   }
23   state Off {
24     out node frequency = 0
25     out node volume = 0
26     out node stateLed = False
27     switch: if dist < 1500 then On else Retain
28   }
29 }
```

bedded systems and recognized as obstacles to readability, maintainability, and evolution.

FRP was initially proposed as an interactive program construction method for Haskell [2]. Since then, FRP has proven helpful in developing embedded systems, starting with its application to robot control [13]. Several FRP languages targeting resource-constrained embedded systems have also been proposed (e.g., [6], [12], [15]).

2.2 Language Overview

Listing 1 is an example application in XStorm that implements a simplified Theremin[*1] that equips a distance sensor and a button. When a performer moves her/his hand closer to the distance sensor, the frequency of the instrument's sound changes according to the distance between the hand and the sensor. It stops the sound if the hand is outside the measurable range of the sensor. The volume of the sound can be adjusted to five levels with the button.

The module `Theremin` (lines 2–29) is a *switch module* (an XStorm program component that defines stateful reactive behaviors) with two states. In lines 3–5, the module declares two inputs (`dist` and `btn`) and three outputs (`frequency`, `volume` and `stateLed`) as time-varying values: `dist` represents the distance sensor measurement, `btn` represents the button status, `frequency` controls the sound frequency, `volume` controls the output volume, and `stateLed` controls an LED for Evaluation 1 (Section 6). Their initial values are written within parentheses. For example, `dist` is initialized as `0`.

In XStorm, as in Emfrp, time-varying values are called *nodes* and are classified into *input*, *output*, and *intermediate* nodes. The values of input nodes are determined by external devices (e.g., sensors), while the values of other nodes are specified by *node definitions* expressed using the keyword `node`. We say that node A depends on node B if B appears (without `@last`, discussed below) on the right-hand side of `=` in the definition of A. Note that no input nodes depend on any other nodes. The nodes in the module should form a directed acyclic graph (DAG) whose edges are dependencies between nodes. XStorm's runtime system updates the values of the nodes as follows. It first determines the values of the nodes with input degree 0 and then updates each node once along the DAG up to the nodes with output degree 0. This update process is called an *iteration*. The runtime system realizes the reactive behavior defined in the module by repeating the iteration. In other words, the execution of a module means the updating of the nodes.

Many embedded systems have state-dependent behaviors. Therefore, their design often uses state transition models such as Statecharts [5]. If we try to represent state-dependent behaviors in Emfrp, time-varying values representing states will appear everywhere, significantly reducing program readability, extensibility, and maintainability. The most significant feature of XStorm is that it provides a language mechanism for describing state-dependent behaviors.

In XStorm, keyword `state` introduces a state and defines a state block. `Theremin` module has two States (`On` and `Off`) and corresponding state blocks (lines 15–22 and lines 23–28).

With the introduction of states, intermediate nodes are further classified into *state local* and *shared* nodes. A definition of an intermediate node in a state block defines a state local node that is updated while the switch module is in the state.

For example, the node `dist_avg` which represents the moving average of `dist`, is updated only while the module is in the state `On`. The time-varying values `frequency` and `volume` are depending on the inputs while the state is `On`, and are `0` while the state is `Off`. The keyword `switch` (lines 21 and 27) defines a state transition by specifying the next state. `Retain` represents staying in the current state. For example, line 21 means that if `dist_avg` is greater than 1500, the state of the switch module will become `Off`; otherwise, it will remain `On`. This transition means that the performer's hand is outside the measurable range of the distance sensor. Shared nodes should be declared in the switch module header separately (lines 6–7) from their definitions outside the state blocks (lines 10–14). They are updated regardless of the state of the module.

The expression `btn@last` in the definition of the shared node `btn_released` (line 10) refers to the *last value* of the node `btn`. This means the value in the previous iteration. In this example, `btn_released` becomes `True` when a falling edge of `btn` is detected.

XStorm is a statically typed language. In addition to `Int` and `Bool`, there are floating point types and tuples, and abstract data types can also be defined. However, recursive data types (e.g., lists and trees) are not allowed. Recur-

[*1] Theremin is an electronic musical instrument that controls its sound without physical contact by detecting the position and movement of the performer's hands with antennas.

Listing 2: Generated Functions for Listing 1 (abridged)

```
extern void input(int * dist, int * btn);
extern void output(int * frequency, int * volume,
                   int * stateLed);

static void activate(void) {
  INITIALIZE;
  while(1) {
    input(&(mem->dist), &(mem->btn));
    ITERATION;
    output(&(mem->frequency), &(mem->volume),
           &(mem->stateLed));
  }
}

// An example of the entry point.
void main(void) {
  // Setup peripherals.
  setup_peripherals();
  activate();
}
```

sive functions are also prohibited, and there are no loop statements.

2.3 Generated Code

The XStorm compiler generates the function `activate` (Listing 2). This function initializes constants (*INITIALIZE*), and then it inputs, do an iteration (*ITERATION*), and outputs repeatedly. The extern functions `input` and `output` are called for inputs and outputs of the main module. A developer complete these extern functions by using platform-specific APIs.

2.4 Memory Management

Since XStorm prohibits recursive functions and recursive data types, it is easier to estimate the maximum memory consumption. The generated C program allocates tuples, abstract data types and some `structs` for modules on arrays declared as global variables. The length of the array is determined according to the estimated memory consumption. Therefore, a XStorm program does not allocate the memory dynamically at the runtime. It avoids running out of memory and can run on memory-constrained devices such as the ULP coprocessors.

3. Low Power Coprocessor

Inter-processor communication and processor power state management are essential to use the low power coprocessor. This section describes about the RISC-V ULP coprocessor of ESP32-S3, because there are various implementations and limitations of low power coprocessors.

Since the ULP coprocessor has less available memory space than the main processor, the main processor must transfer data that the ULP coprocessor reads and writes to the memory space that is accessible to the ULP coprocessor. ESP32-S3 has several memories, including 512KiB Internal RAM and 8KiB RTC SLOW Memory. The main processor uses mainly the 512KiB Internal RAM, which is invisible to the ULP coprocessor. Data read and written by the ULP coprocessor must be copied to the 8KiB RTC SLOW Memory.

Listing 3: Theremin using the Proposed Mechanism

```
switchmodule Theremin {
  in dist(0) : Int, btn(False) : Bool
  out frequency : Int, volume : Int,
      stateLed : Bool   # For Eval. 1
  shared btn_released(False) : Bool,
         volume_mode(1) : Int
  init On

  shared node btn_released = not(btn) && btn@last
  shared node volume_mode =
    if btn_released
      then next_volume_mode(volume_mode@last)
      else volume_mode@last
  state On on main {
    node dist_avg(0) =
        (dist * 6 + dist_avg@last * 4) / 10
    out node frequency = dist_avg
    out node volume = volume_mode * 20
    out node stateLed = True
    switch: if dist_avg>=1500 then Off else Retain
  }
  state Off on ulp {
    out node frequency = 0
    out node volume = 0
    out node stateLed = False
    switch: if dist < 1500 then On else Retain
  }
}
```

The RTC SLOW Memory is slower than the Internal RAM because of its slower clock. In addition, the start address of the RTC SLOW Memory is different between the ULP coprocessor and the main processor. In order to transfer data containing pointers, addresses must be properly translated prior to data transfer.

The main processor power states can be classified into two states: active and sleep. ESP32-S3 have two sleep states: Light-sleep and Deep-sleep. Light-sleep retains the contents of the Internal RAM and the program counter of the main processor, but Deep-sleep does not. Thus Deep-sleep saves more power than Light-sleep. According to the data sheet [3], the typical power consumption is 8 µA versus 240 µA. They are much lower than 13.2 mA of the main processor, which runs at 40 MHz (minimum frequency).

4. Proposed Mechanism

The proposed mechanism provides a way to choose which processor to run on for each state of the switch module. The program switches processors to run exclusively depending on the state.

Listing 3 is a Theremin program that uses the ULP coprocessor by the proposed mechanism and is derived from Listing 1. The switch module `Theremin` has two states: `On` running on the main processor and `Off` running on the low power coprocessor. There are few changes in the XStorm program. `on` modifier are added in the `state` definitions. States with `on main` run on the main processor and states with `on ulp` run on the low power coprocessor. This modifier is effective only on the toplevel module. XStorm can instantiate modules within a module (a *submodule* [9], [15]), but this modifiers on the submodules are ignored.

Listing 4: Generated Functions for the Main Processor (abridged)

```
extern void input(int * dist, int * btn);
extern void output(int * frequency, int * volume,
                   int * stateLed);
extern void into_sleep(void);
extern void into_active(void);

static void switch_processor(void) {
  if(!SWITCH_REQUIRED) return;
  // beginning of (1)
  DATA_TRANSFER_MAIN_TO_ULP;
  // end of (1) & beginning of (2)
  into_sleep();  // The main processor halts.
  // end of (3) & beginning of (4)
  DATA_TRANSFER_ULP_TO_MAIN;
  // end of (4)
  into_active();
}
static void activate_main(void) {
  while(1) {
    input(&(mem->dist), &(mem->btn));
    ITERATION;
    output(&(mem->frequency), &(mem->volume),
           &(mem->stataeLed));
    switch_processor();
  }
}
static void activate(void) {
  INITIALIZE;
  activate_main();
}
static void activate_deep(void) {
  // end of (3) & beginning of (4)
  DATA_TRANSFER_ULP_TO_MAIN;
  // end of (4)
  into_active();
  activate_main();
}

// An example of the entry point.
void main(void) {
  // Enables to wake up from the interruption
  // by the ULP coprocessor.
  esp_sleep_enable_ulp_wakeup();
  // Peripherals (e.g. LED PWM Controller)
  // are reset after Deep-sleep.
  setup_peripherals();
  // Check whether the main processor have been
  // woke up by the ULP coprocessor.
  if(esp_sleep_get_wakeup_cause()
                  == ESP_SLEEP_WAKEUP_ULP) {
    activate_deep();
  } else {
    // Load the ULP program.
    ulp_riscv_load_binary(
      bin_start, (bin_end-bin_start));
    activate();
  }
}
```

Listing 5: Generated Functions for the ULP Coprocessor (abridged)

```
extern void input(int * dist, int * btn);
extern void output(int * frequency, int * volume,
                   int * stateLed);
extern void into_active(void);

static void switch_processor(void) {
  if(!SWITCH_REQUIRED) return;
  // beginning of (3)
  into_active(); // The ULP coprocessor terminates.
}
static void activate(void) {
  while(1) {
    input(&(mem->dist), &(mem->btn));
    ITERATION;
    output(&(mem->frequency), &(mem->volume),
           &(mem->stateLed));
    switch_processor();
  }
}
// An example of the entry point.
void main(void) {
  // end of (2)
  activate();
}
```

5. Implementation

There are some changes in the generated programs to target ESP32-S3. This section describes these changes.

5.1 Generated Program

The XStorm compiler implementing the proposed mechanism generates two C/C++ source code files: one for the main processor (Listing 4) and one for the ULP coprocessor (Listing 5). In each file, the *ITERATION* for the other processor is eliminated. For example, the C source code file for the ULP coprocessor does not contain the *ITERATION* under the state running on the main processor (`On` state).

In Section 2.3, we mentioned that the developer must fill in the functions `input` and `output`. In addition to these functions for the main processor, the developer also must write the functions `input` and `output` for the ULP coprocessor. In ESP32-S3, the ULP coprocessor has a different I/O API than the main processor.

In addition to `input` and `output`, the extern functions `into_sleep` and `into_active` have been added. These functions are called when the running processor is switched. The developer must fill in `into_sleep` for only the main processors, and `into_active` for the both. For example, `into_sleep` starts the ULP coprocessor and make self (the main processor) and some peripherals sleep. `into_active` for the ULP coprocessor wakes the main processor, and that for the main processor wakes some peripherals.

After each iteration, the function `swtich_processor` is called. It checks whether the running processor must be switched (*SWITCH_REQUIRED*). When the running processor is switched from the main processor to the ULP coprocessor, the function copies the necessary data from the main processor to the ULP coprocessor (*DATA_TRANSFER_MAIN_TO_ULP* in Listing 4). Then calling `into_active` wakes the ULP coprocessor and halts the main processor. After the running processor must be the main processor from the ULP coprocessor, `switch_processor` of the ULP coprocessor (Listing 5) only calls the function `into_active` that wakes the main processor and halts the ULP coprocessor. After waking the main processor, the following procedure is different in the sleep state. If the main processor was in Light-sleep, the program counter remains, so `switch_processor` of the main processor (Listing 4) is reentered. It copies the necessary data from the main processor to the ULP coprocessor (*DATA_TRANSFER_ULP_TO_MAIN*), and then calls `into_active` to wake some peripherals. If the main processor

was in Deep-sleep, it executes from the entry point. The function `esp_sleep_get_wakeup_cause` (an ESP32-S3 API) returns `ESP_SLEEP_WAKEUP_ULP`, then `activate_deep` is called (lines 47–51 in Listing 4). `activate_deep` copies the necessary data (*DATA_TRANSFER_ULP_TO_MAIN*), and then starts iterations.

5.2 Data Transfer between Processors

In the Theremin example, the values of the shared nodes (`btn_released` and `volume_mode`) must be copied when the state transitions from `On` to `Off`. These values are stored in the struct of the switch module, and its instance are allocated in the array explained in Section 2.4. In ESP32-S3, the ULP coprocessor does not have access to the Internal RAM, as explained in Section 3. Therefore, in order to reference these (e.g., `volume_mode@last`) from the ULP coprocessor after a state transition, the main processor must copy these values into the RTC SLOW Memory.

In our implementation, we defined arrays on the RTC SLOW Memory for the ULP coprocessor as well as the main processor, as explained in Section 2.4. In state transitions, reachable values from the struct of the switch module `Theremin` are copied. The program traces the current and previous values of each node from the main module. Primitive types, such as numbers and boolean values, are copied straightforward. For types referenced through pointers, such as tuples, abstract data types and submodule instances, the instance allocated on the array as a global variable is copied, and the address of the pointer is translated because the start address of the RTC SLOW Memory is different between the main processor and the ULP coprocessor. Address translation is defined as a preprocessor macro (`#define`) or a C function. The developer can adapt to the different memory mappings (e.g., future ESP32s).

6. Evaluation 1 : Power Consumption

Using the ULP coprocessor, it is expected to reduce the power consumption. We measured power consumption using the Theremin example in three configurations:

- No-Sleep. Uses only the main processor, no sleep state.
- Light-sleep.
- Deep-sleep.

Table 1 shows the evaluation environment. The evaluation board ESP32-S3-DevKitC-1 has an always-on LED and an addressable RGB LED. The always-on LED indicates that the board is powered on even if the main processor is in the sleep states. On version 1.0 boards, the addressable RGB LED may be lit due to noise. We powered on this board not to turn on this RGB LED carefully. All resources are available: `https://github.com/psg-titech/apris-2023-experiments`

6.1 Method

Fig. 1 illustrates the evaluation method. We used the Theremin program in Listing 3. The inputs `dist` and `btn` were generated virtually on the function `xsinput`. `dist` was input a triangle wave ranging from 0 to 3000. `btn` was always

Table 1: The Evaluation Environment

	Name	Revision
Evaluation board	ESP32-S3-DevKitC-1	N8, v1.0
SDK for ESP32-S3	ESP-IDF	v5.1.0
Ammeter (Eval. 1)	Nordic Power Profiler Kit 2	
FPGA (Eval. 2)	ZYBO Z7-20	Vivado 2023.1.1

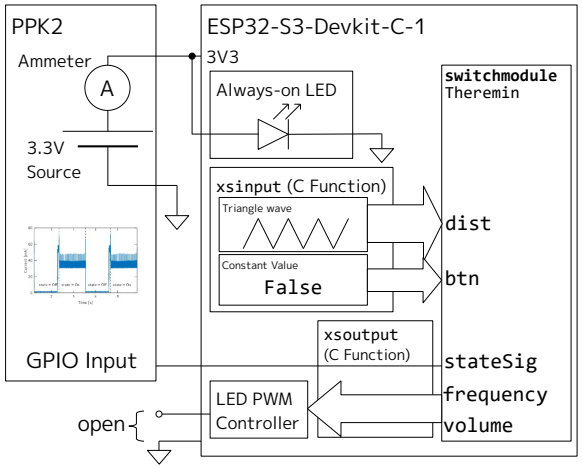

Fig. 1: The Configuration for Evaluation 1

`False`. The outputs `freq` and `volume` were outputs to the LED PWM Controller. The LED PWM Controller output a PWM signal to an GPIO pin, which was open. In the No-sleep configuration, the LED PWM Controller was disabled while the state is `Off`, like the sleep states.

To measure power consumption, we used Nordic Power Profiler Kit 2 (PPK2). The sample rate was 10^5 samples per second, and we sampled for 5 minutes. PPK2 supports GPIO inputs. The output `stateSig` was connected to the GPIO inputs of PPK2 to distinguish the state. We also used PPK2 as a voltage source (Source Meter Mode). We connected the 3V3 pin in ESP32-S3-DevKitC and the power output is 3.3 V. The ESP32-S3 operated at 160 MHz. We used the functions `usleep` on the main processor and `ulp_riscv_delay_cycles` on the ULP coprocessor to wait for 33 µs between iterations.

We disabled all logger outputs because the logger causes slower boot. We set following parameters by the menuconfig:

- `CONFIG_BOOTLOADER_LOG_LEVEL_NONE=y`
- `CONFIG_BOOTLOADER_LOG_LEVEL=0`

6.2 Results and Discussion

Fig. 2 shows an extracted result and Table 2 shows maximum, minimum, and average current. The states displayed at the bottom of the figures come from the output `stateSig`.

The typical current consumption ranges from 27.6 mA to 54.6 mA [3]. The No-sleep results in Table 2b also range from 29.91 mA to 50.36 mA. In Table 2a, in each configuration, the results are similar in the state `On`. The typical current consumption in Light-sleep and Deep-sleep are 240 µA and 8 µA, respectively [3]. Looking at the minimum on the state `Off` in Table 2b, the difference between Light-sleep and Deep-sleep is 0.26 mA. This result is close to the data sheet.

In the state `Off`, the current consumption of Light-sleep

and Deep-sleep are much lower than on No-sleep. The averages of Light-sleep and Deep-sleep are 12 and 25 times lower than No-sleep in Table 2a, respectively. However, spikes are also observed in Light-sleep and Deep-sleep. In Table 2b, their maximum values are higher than No-sleep. As can be seen in Fig. 2, the spikes occur during state transitions of Light-sleep and Deep-sleep. The spikes in Deep-sleep is much longer than in Light-sleep. This is because the same process as the normal boot process runs during the state transitions of Deep-sleep. Therefore, the highest consumption is observed in the state `Off` and Deep-sleep in Table 2b because the signal `stateSig` is changed after the boot process. In this evaluation, the state is switched frequently. As a result, the average of Deep-sleep is worse than Light-sleep in Table 2b.

There was a significant difference in power consumption. We observed that using a low power coprocessor in FRP reduces power consumption. However, power consumption during state transitions is high if the sleep states are used. We have to choose Light-sleep or Deep-sleep while taking the frequency of the state transition into consideration.

Table 2: Maximum, Minimum, and Average Current Consumption (mA)

(a) Between the 10th and 90th Percentiles

	No-sleep		Light-sleep		Deep-sleep	
State	On	Off	On	Off	On	Off
Minimum	31.52	31.52	31.52	2.57	31.27	1.21
Maximum	32.28	32.28	32.38	2.68	32.08	1.35
Average	31.89	31.89	31.94	2.63	31.65	1.27

(b) All Results

	No-sleep		Light-sleep		Deep-sleep	
State	On	Off	On	Off	On	Off
Minimum	30.00	29.91	28.67	0.95	29.62	0.69
Maximum	50.36	48.33	67.01	67.01	59.26	76.08
Average	31.95	31.95	32.01	2.65	31.72	4.41

7. Evaluation 2: Elapsed Time During the State Transition

We observed that state transitions with switching the running processor take longer than without switching, because there are overheads: processor power state transition and data transfer. We measured the overheads in state transitions with switching using the Theremin example in two configurations:

- Light-sleep
- Deep-sleep

Table 1 shows the evaluation environment.

7.1 Method

Fig. 3 illustrates the evaluation method. We used the Theremin program in Listing 3. The inputs and outputs of the switchmodule `Theremin` were same as Section 6.1. We disabled all logger outputs as explained in Section 6.1. The ESP32-S3 operated at 160 MHz.

We measured time by using an FPGA. The FPGA and the evaluation board are connected through GPIO. The evaluation board always outputs a gray code. The output changes at the beginning and the end of the overheads being measured. The FPGA measures the time that the value is being output. The FPGA operated at 50 MHz.

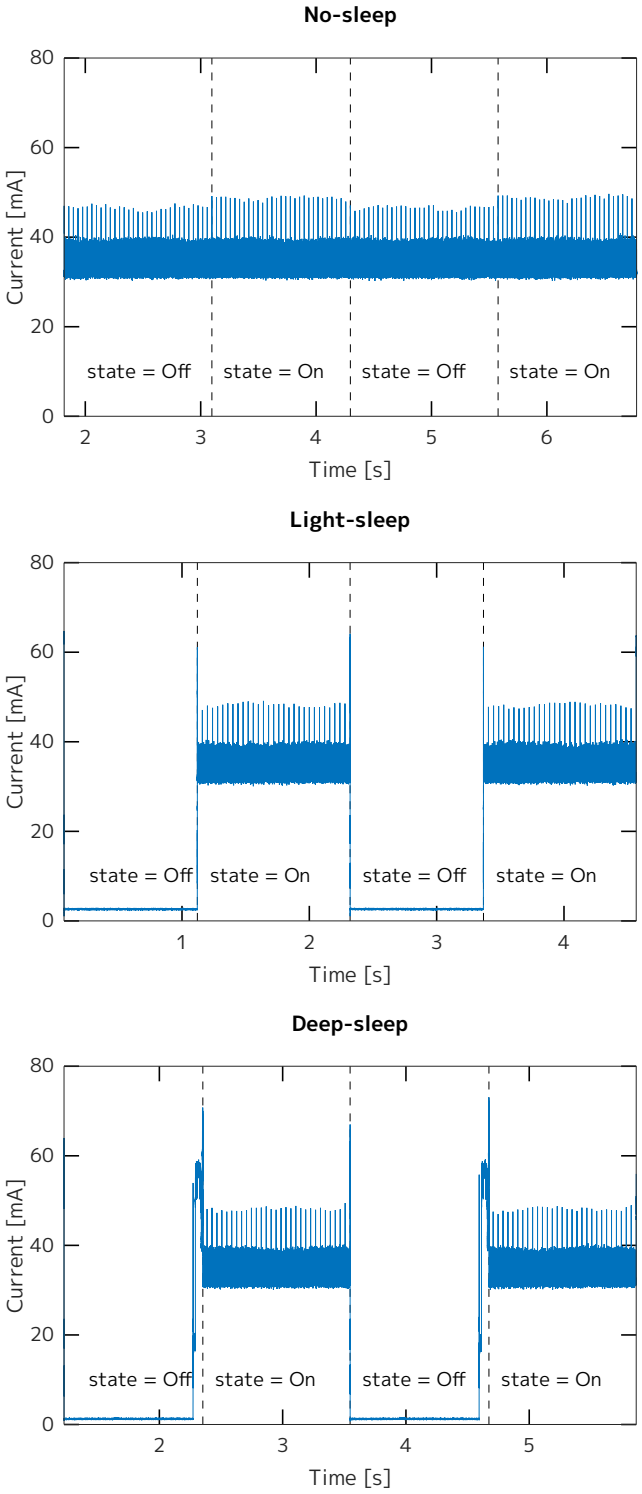

Fig. 2: Current Consumption

The overheads measured in this evaluation are:

(1) Data transfer from the main processor to the ULP coprocessor

(2) The ULP coprocessor wake-up

(3) The main processor wake-up
(4) Data transfer from the ULP coprocessor to the main processor

Each overhead is showed in Listing 4 and 5. These overheads do not occur if a state transition does not require switching the running processor. We sampled 1000 samples for each overhead.

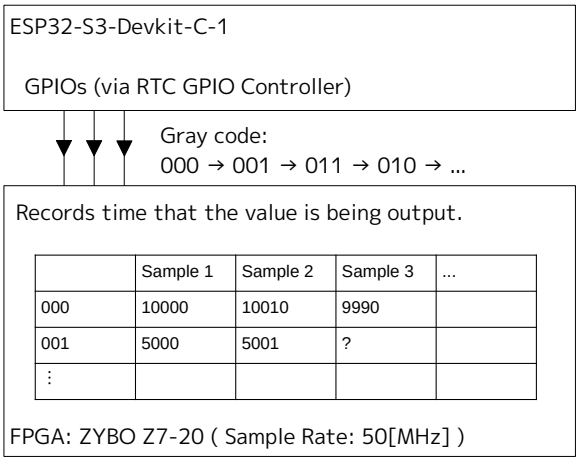

Fig. 3: The Configuration for Evaluation 2

7.2 Result and Discussion

Table 3 shows the average wall-clock time for the overheads. Comparing two configurations, every overheads in Deep-sleep is worse than Light-sleep. Due to the complete boot process, the wake-up time of the main processor from the Deep-sleep state is more than 135 times the wake-up time from the Light-sleep state. In this example, the iterations are executed every 33 ms, but the wake-up time of the main processor from the Deep-sleep state takes about 80 ms. Thus, timings of two iterations are not met. By the way, it is unclear why data transfer and wake-up time of the ULP coprocessor in Deep-sleep take longer than in Light-sleep. The state of the interconnect may be different between Light-sleep and Deep-sleep.

When comparing data transfer and processor wake-up time, the processor wake-ups take more than 10 times the data transfers. Since the RTC SLOW Memory is limited, we can consider that data transfer does not take much longer than processor wake-up time.

Table 3: Average Wall-Clock Time of the Overheads (µs)

	(1)	(2)	(3)	(4)
Light-sleep	9.63	164.00	579.29	14.64
Deep-sleep	13.91	179.75	79669.60	39.43

8. Related Work

8.1 FRP Languages for Embedded Systems

Several FRP languages for embedded systems are proposed, e.g., Hailstorm [14] and Juniper [6]. Hailstorm is an instance of Arrowised FRP languages. Thus, the relations between time-varying values are described by combining *signal functions*. The language provides `switch#` combinator for modeling stateful behaviors. It takes two arguments: the first is a signal function whose output value is used to switch between the various signal functions, and the second is a function that returns a signal functions depending on the result of the first argument. The second argument function should be a combination of the `if` expressions.

Juniper combines several FRP concepts. Time-varying values in Juniper are first class citizens. This means that stateful behaviors can be described by switching the time-varying values with the `if` or `match` expression. Listing 6[*2] shows an example of stateful behavior in Juniper.

In these language, it is difficult to discover an `if` expression eligible to switch the running processor because the whole program must be analyzed. In contrast, our implementation only seeks the `state` declaration in the toplevel `switchmodule`. To use a low power coprocessor in these languages, a new primitive for switching the running processor or such a complicated analyzer is required.

Listing 6: A Stateful Behavior in Juniper

```
1  fun main() : unit = (
2    setup();
3    while true do (
4      let inputSig = ...
5      let mainSig =
6      Signal:foldP(fn (inputs, state) ->
7        case state of
8        | On => ( ... )  // state On  { ... }
9        | Off => ( ... ) // state Off { ... }
10       end
11     end, inputSig)
12     Signal:sink(fn state -> ... end, mainSig)
13   ) end
14 )
```

XCios [16] is derived from XStorm and has a mechanism for dynamically switching peripherals. In each state of a `switchmodule`, XCios allows to describe what state each I/O device is in. The extern functions `input` and `output` are prepared for each I/O time-varying variable. In addition to `input` and `output`, XCios automatically calls extern functions called *hook functions* when each I/O time-varying variable changes its state. The hook function manages the power mode of the peripheral to save the power consumption of the peripheral device. It is not difficult to combine XCios with the proposed mechanism in this paper.

8.2 Related Technologies

8.2.1 Other Microcontrollers

Our current implementation only supports the ESP32-S3, but can be easily adapted to other microcontollers satisfying:

- A main processor and coprocessor*s* are communicating via memory mapped I/O.
- A main processor can enter the sleep states by any C function. It pauses during the coprocessor is working, and then it restarts from the entry point or resumes from the last position before sleeping.

*2 A similar example is `NeopixelRing.jun` on the Juniper repository with the commit hash `da2725428d1`.

- Memory spaces accessible from coprocessors are accessible from the main processor before the coprocessors wake up, or the coprocessors can wait for data transfer (e.g., by using spinlocks).

For example, the PSoC6 series [7] and the LPC43xx series [10] have a Cortex-M4 (CM4) as a main processor and a Cortex-M0(+) as a low power coprocessor, and i.MX RT 1160 [11] has a Cortex-M7 (CM7) and a CM4. They have shared memories for communication between processors and the memory map for each processor is the same. All of the memories are visible from both processors. Such a microcontroller can be applied the proposed mechanism straightforward. Because of the same memory map, no address translation is required. However, despite of the same memory map, most microcontrollers does not allow to access to part of memories in the sleep state.

8.2.1.1 PSoC6 Series

Both processors use the shared memory as a main memory and the entire memory is visible even if it is in the sleep state. Thus, our implementation can be easily adapted to the PSoC6 series by disabling data transfer.

8.2.1.2 LPC43xx Series

When the main processor is in the sleep state, the accessible range of the memory map is limited. Therefore, shared data must be copied to the accessible range before the main processor halts. Moreover, in this case, it is expected that allocating shared data within the accessible range beforehand allows to reduce data transfer size.

8.2.1.3 i.MX RT 1160

There are TCMs, memories associated with the Cortex processor. The CM7 TCM is visible from the CM4 during the CM7 is on. The CM4 TCM is visible from the CM7 even if the CM4 is sleeping, but power consumption can be reduced by turning off the CM4 TCM. In addition, access from the CM7 to the CM4 TCM is slower than the CM7 TCM because they are not directly connected.

8.2.2 OpenAMP

OpenAMP (Open Asymmetric Multi-Processing) [8] supports life cycle management (remoteproc) and inter-processor communication (RPMsg) capabilities for handling remote compute resources. It provides a library for Linux, RTOS (Real-Time Operating System) and bare-metal. With RPMsg, packets are sent by a `send`-like function and are received by a callback. Our implementation may be adapted to RPMsg APIs (not straightforward).

9. Conclusion

We proposed a new mechanism allowing to use low power coprocessors with XStorm, a FRP language. This mechanism allows to write the running processor in each state. Managing the processor power state makes easier by this mechanism, thanks for an abstraction mechanism for modeling stateful behaviors in XStorm. We implemented a compiler and evaluated a Theremin example with ESP32-S3. Our compiler can automatically generate a data transfer program between processors. In our evaluation, it shows that the proposed mechanism reduces power consumption. However, the overhead of state transitions with switching the running processor is also observed. We need to consider these overhead when choose whether we use the sleep state or not.

As explained in 8.2.1.2, allocating the shared data within a range accessible from both of a main and a low power processor beforehand is a future work. It is expected to reduce data transfer size and latency when using LPC43xx-like microcontrollers.

Acknowledgments This work is supported in part by JSPS KAKENHI Grant Numbers 21K11822 and 22K11967.

References

[1] Bainomugisha, E., Carreton, A. L., Van Cutsem, T., Mostinckx, S. and De Meuter, W.: A Survey on Reactive Programming, *ACM Computing Surveys*, Vol. 45, No. 4, pp. 52:1–52:34 (online), DOI: 10.1145/2501654.2501666 (2013).

[2] Elliott, C. and Hudak, P.: Functional Reactive Animation, *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*, ACM, pp. 263–273 (online), DOI: 10.1145/258949.258973 (1997).

[3] Espressif Systems: *ESP32-S3 Series Datasheet* (2023).

[4] Espressif Systems: *ESP32-S3 Technical Reference Manual* (2023).

[5] Harel, D.: Statecharts: a visual formalism for complex systems, *Science of Computer Programming*, Vol. 8, No. 3, pp. 231–274 (online), DOI: 10.1016/0167-6423(87)90035-9 (1987).

[6] Helbling, C. and Guyer, S. Z.: Juniper: A Functional Reactive Programming Language for the Arduino, *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*, FARM 2016, New York, NY, USA, ACM, pp. 8–16 (online), DOI: 10.1145/2975980.2975982 (2016).

[7] Infenion Technologies AG: *PSoC 6 MCU: CY8C61x4, CY8C62x4 Architecture Technical Reference Manual (TRM)* (2023).

[8] Linaro: https://openamp.readthedocs.io/en/v2023.04.0/. (Accessed in 2023-09-13).

[9] Matsumura, A. and Watanabe, T.: An Abstraction Machanism for Modeling Stateful behaviors in an FRP Language for Embedded Systems, *IPSJ Transactions on Programming (PRO)*, Vol. 13, No. 2, pp. 1–13 (2020). (in Japanese).

[10] NXP Semiconductors N.V.: *UM10503 LPC43xx/LPC43Sxx ARM Cortex(R)-M4/M0 multi-core microcontroller* (2019).

[11] NXP Semiconductors N.V.: *i.MX RT1160 Processor Reference Manual* (2021).

[12] Oeyen, B., De Koster, J. and De Meuter, W.: Reactive Programming on the Bare Metal: A Formal Model for a Low-Level Reactive Virtual Machine, *9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '22)*, ACM, pp. 50–62 (online), DOI: 10.1145/3563837.3568342 (2022).

[13] Pembeci, I., Nilsson, H. and Hager, G.: Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages, *4th International Conrefernce on Principles and Practice of Declarative Programming (PPDP 2002)*, ACM, pp. 168–179 (online), DOI: 10.1145/571157.571174 (2002).

[14] Sarkar, A. and Sheeran, M.: Hailstorm: A Statically-Typed, Purely Functional Language for IoT Applications, *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*, PPDP '20, New York, NY, USA, ACM, pp. 1–16 (online), DOI: 10.1145/3414080.3414092 (2020).

[15] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, *MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity*, ACM, pp. 36–44 (online), DOI: 10.1145/2892664.2892670 (2016).

[16] Takimoto, S., Moriguchi, S. and Watanabe, T.: Mode Management of Peripherals Based on State hook Model in FRP Language for Embedded Systems, *Proceedings of the 40th JSSST Annual Conference* (2023). (in Japanese).