Programming Embedded Systems with SenseVM

Anonymous author
Anonymous affiliation
Anonymous author
Anonymous affiliation
Anonymous author
Anonymous affiliation

Abstract

Programming embedded systems applications involves writing concurrent, event-driven and timing-aware programs. Traditionally such programs are written in low-level machine-oriented programming languages like C or Assembler. We present SenseVM, an alternative that offers high-level features to the programmer while delegating the low-level support needed to a runtime system and one-time-effort drivers.

SenseVM is a virtual machine (VM) for embedded systems that provides an API that supports (i) message-passing based concurrency, (ii) a message-passing based IO interface that translates between low-level interrupt based and memory-mapped peripherals and (iii) an operator to express timing behaviours. Programs for SenseVM are written in a Caml-inspired functional language that gets compiled to bytecodes interpreted by the VM. The VM supports Concurrent ML (CML) style concurrency by providing bytecode operations corresponding to CML's message-passing primitives.

All I/O is expressed as reads or writes on CML channels while a low-level I/O *bridge* interface handles interactions with interrupt service routines or memory-mapped I/O. For timing of operations, we add a variant of CML's `sync` operation called `syncT` that incorporates ideas from TinyTimber.

The devices targeted by SenseVM are microcontroller-based systems such as the STM32F4 or the NRF52. The particular models of these microcontrollers used in our evaluations come with 196 and 256kB of RAM and run at 168 and 64MHz. We evaluate the VM on a set of small programs suited to the peripherals available on each board and provide benchmarks estimating the VM's response time and jitter rates.

2012 ACM Subject Classification Computer systems organization → Embedded software; Software and its engineering → Runtime environments; Computer systems organization → Real-time languages; Software and its engineering → Concurrent programming languages

Keywords and phrases real-time, concurrency, functional programming, virtual machine

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.23

Funding Anonymous funding

1 Introduction

Embedded systems are ubiquitous and often used for performing control operations. For instance, a washing machine serves information to its user through a small LED-based display while taking input from the user in the form of control knobs and buttons. The main function of the system is, however, to perform a wash cycle consisting of heating of water, filling washing compartment with water, mix in laundry detergent at the right time and dosage, spinning the drum at various speeds at various time and so on. All of this is accomplished through actuation via microcontroller peripherals such as a timer generating a Pulse-width Modulation (PWM) signal of the correct frequency and duty cycle to drive a motor at the desired speed or controlling relays for turning pumps on and off. All the while, sensors provides information to the microcontroller about clogged up filters or other non-ideal conditions. All in all the application is concurrent and aware of a notion of time.

SenseVM is geared towards such applications, with the approximate performance of the NRF52 (64Mhz, 256kB RAM) microcontroller. These microcontroller-based applications like IoT controllers, industrial machinery, etc tend to embody three common characteristics:

1. They are predominantly *I/O bound* applications
2. They are *concurrent* in nature
3. A large subset of such programs are *timing-aware*

This list is by no means exhaustive, as discussed later in Section 1.2, but captures a prevalent theme among embedded applications. Programming these applications involve interaction with callback-based driver APIs like the following from the Zephyr RTOS[11]:

Listing 1 A callback-based GPIO driver API

```
int gpio_pin_interrupt_configure(const struct device *port
                                 , gpio_pin_t pin
                                 , gpio_flags_t flags);
void gpio_init_callback(struct gpio_callback *callback
                        , gpio_callback_handler_t handler
                        , gpio_port_pins_t pin_mask);
int gpio_add_callback(const struct device *port
                      , struct gpio_callback *callback);
```

Programming with the above API in low-level languages like C leads to complicated state machines, which even for relatively small programs result in difficult-to-maintain and complex state-transition tables. More modern language runtimes like MicroPython [12] and Espruino (Javascript) [36] support higher-order functions and handle callback-based APIs in the following way:

Listing 2 Driver interactions using Micropython

```
def callback(x):
#...callback body with nested callbacks...

extint = pyb.ExtInt(pin, pyb.ExtInt.IRQ_FALLING
                    , pyb.Pin.PULL_UP, callback)
ExtInt.enable()
```

The nested-callback handling, mentioned above, leads to a form of *accidental* complexity, colloquially termed as *callback-hell* [20]. Moreover, all of the above mentioned languages use error-prone shared-memory primitives like *semaphores* and *locks* to mediate interactions that occur between the callback-based driver handlers.

SenseVM attempts to address the concerns about callback-hell and shared-memory communication while targeting the three characteristics of embedded programs mentioned earlier by a combination of:

1. Concurrent ML style concurrency and message-passing.
2. An I/O *bridge* providing a message-passing interface to low-level peripherals.
3. A notion of time.

Concurrent ML (CML) [25] builds upon the message-passing based concurrency model CSP [15] but adds the feature of composable first-class events. These first-class events allow the programmer to tailor new concurrency abstractions and express application-specific protocols. A CML event is a value that represents the intention to possibly communicate and not the actual action of communication. The programmer decides when and if to turn an intent to communicate into the action of communication, which is done using the CML primitive `sync`.

SenseVM implements a low-level bridge interface that separates the runtime system from platform-dependent specifics. The runtime system is written in C99 with no dependencies on any threading subsystem or low-level hardware abstraction layer. The low-level bridge on the other hand, is implemented on top of Zephyr or ChibiOS to make use of their hardware abstractions. The bridge communicates with SenseVM application threads using the same channel abstraction as message-passing between threads, while on the low-level side of the bridge, each driver implements the memory-mapped IO or callback/interrupt-based interface with the respective hardware peripherals. Fig 1 provides an overview of SenseVM.

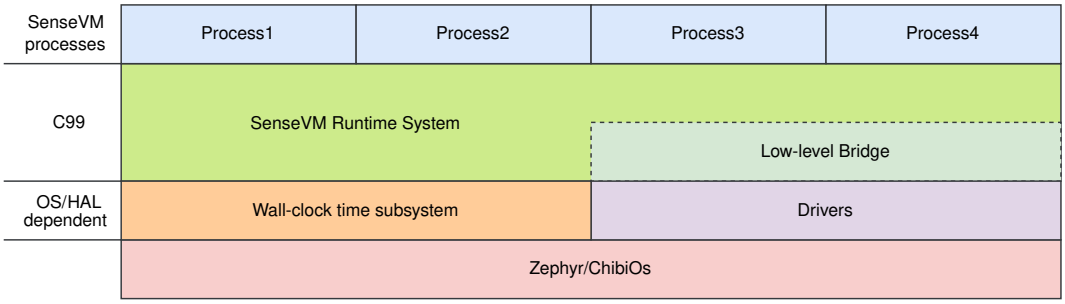


Figure 1 An overview of SenseVM

For timing, SenseVM takes inspiration from the TinyTimber kernel [19] that allows the specification of baseline and deadline windows for invocation of a method in a class. Just like TinyTimber, SenseVM applies an earliest-deadline-first (EDF) scheduling of tasks whose baseline have been reached. In TinyTimber, `WITHIN( B, D, &obj, meth, 123 );` expresses the desire that method `meth` should be run at the earliest at time `B` and finish within a duration of `D`. The SenseVM adaptation of TinyTimber `WITHIN` is `syncT` that takes a baseline, deadline and an event as argument.

1.1 Contributions

- We identify the three characteristic behaviours of being (i) I/O bound, (ii) concurrent, and (iii) timing-aware for embedded applications and propose a combination of ideas that mesh well with each other and address these requirements. We explain SenseVM's API in detail in Section 3.
- **Message-passing based IO.** We present a uniform message-passing framework that combines *concurrency* and *callback-based I/O* to a single interface. A software message or a hardware interrupt is identical in our programming interface, providing the programmer with a simpler message-based framework to express concurrent hardware interactions. We describe the API in Section 3, show illustrative examples in Sections 4 and 5 and look at the critical implementation details in Section 6.
- **Declarative state machines for embedded systems.** Combining CML primitives with our I/O interface allows presenting a declarative framework to express state machines, commonly found in embedded systems. We illustrate examples of representing finite-state machines on the SenseVM in Section 4.
- **Evaluation.** We provide empirical evaluations of several small embedded-system programs and an adaptation of a programming exercise from a real-time programming course. The programming exercise is a soft real-time music programming example running on the `STM32F4` microcontroller, presented in Section 5. We provide benchmarking results in Section 7.

1.2 Limitations

The characteristics of embedded programs that are not addressed in this work are:

- Power efficiency and lifetime while operating from a small battery, for example. This is challenging for a byte-code interpreting virtual machine as there will be interpretive overhead costs in battery life.
- Certain embedded applications have very tight timing requirements and should execute as fast as possible. This is very hard for a byte-code interpreting virtual machine to match.
- Reliable and consistent timing is central to many control applications. It is hard for a runtime system with garbage collection to *always* meet such timing requirements .

Our evaluations in Section 7 show the performance of SenseVM in these areas like interpretive and garbage-collection overheads. We discuss future work to address a number of these limitations in Section 8.

2 Motivation and Background

• **Concurrency and IO.** In embedded systems, concurrency takes the form of a combination of callback routines, interrupt service routines and possibly a threading system, for example threads as provided by ZephyrOS, ChibiOS or FreeRTOS. The callback style of programming is complicated but offers benefits when it comes to energy efficiency. Registering a callback with an Interrupt Service Routine (ISR) allows the processor to go to sleep and conserve power until the interrupt arrives. Listing 3 shows a cut-down code snippet that registers a callback for a button-press and release (as configured by `GPIO_INT_EDGE_BOTH` on line 11 of listing 3). The example uses ZephyrOS. When the button press or release happens, the callback routine sets an LED to either on or off. The `GPIO_INT_DEBOUNCE` configuration in the registering of the callback makes use of button debouncing hardware if implemented on the GPIOs on the micrcontroller unit (MCU); this means that at the time our callback is run, we should have a stable one or zero on the GPIO pin associated with the button.

Listing 3 Callback programming in C

```
1  void button_pressed(const struct device *dev,
2                      struct gpio_callback *cb, uint32_t pins)
3    { match_led_to_button(dev, led); }
4
5  void main(void){....
6    button = device_get_binding(SW0_GPIO_LABEL);
7    ret = gpio_pin_configure(button, SW0_GPIO_PIN, SW0_GPIO_FLAGS);
8    ret = gpio_pin_interrupt_configure(button,
9                                       SW0_GPIO_PIN,
10                                      GPIO_INT_DEBOUNCE |
11                                      GPIO_INT_EDGE_BOTH);
12
13   gpio_init_callback(&button_cb_data, button_pressed,
14                      BIT(SW0_GPIO_PIN));
15   gpio_add_callback(button, &button_cb_data);
16   ....}
17
18 static void match_led_to_button(const struct device *button,
19         const struct device *led){
20   bool val = gpio_pin_get(button, SW0_GPIO_PIN);
21   gpio_pin_set(led, LED0_GPIO_PIN, val);
22 }
```

The simple program above, in its entirety, is more than 100 lines of callback-based code [10]. We have omitted non-essential parts of the code. The control-flow of such callback-based programs is non-linear and reasoning about them complex. As the number of states in a system multiplies, the complexity of callback-based programs grows exponentially. An alternate pattern to restore the linear control flow of a program is the event-loop pattern.

As the name implies, an event-loop based program involves an infinitely running loop that handles interrupts and dispatches the corresponding interrupt-handlers. An event-loop based program involves some delicate plumbing that connects its various components. Listing 4 shows a tiny snippet of the general pattern.

Listing 4 Event Loop

```
void eventLoop(){
 while (1) {
  switch(GetNextEvent()) {
    case GPIO1 : GPIO1Handler();
    case GPIO2 : GPIO2Handler();
    ....
    default : goToSleep(); // no events
}}}

GPIO1Handler(){ ... } // must not block
GPIO2Handler(){ ... } // must not block

//when interrupt arrives write to event queue
GPIO1_IRQ(){....}
GPIO2_IRQ(){....}
```

Programs like the above are an improvement over callbacks, as they restore the linear control-flow of a program, which eases reasoning. However, such programs have a number of weaknesses - (i) they are highly *inextensible*, (ii) they enforce constraints on the blocking and non-blocking behaviours of the event handlers, (iii) programmers have to hand-code elaborate plumbings between the interrupt-handlers and the event-queue and (iv) they are instances of clearly concurrent programs that are written in this style due to lack of concurrency support in the language.

Although there are extensions of C to aid the concurrent behaviour of event-loops such as protothreads [7] or `FreeRTOS Tasks`, the first three listed problems still persist. In general, the main infinite event loop unnecessarily induces a tight coupling between unrelated code fragments (like the two separate handlers for GPIO1 and GPIO2) that additionally breaks down the abstraction boundaries between them.

• **Time.** Many embedded applications need to be aware of time. Take the earlier example with the washing machine that does certain things at certain points in time along the wash cycle as an example. Going further, there are applications where the timing requirements are even stronger. In hard real-time systems the completion time of an operation determines the correctness of the program. Real-time programs, while concurrent, differ from standard concurrent programs in that they allow the programmer to override the fairness of a *fair* scheduler. For instance, if we look at the `FreeRTOS Task` API:

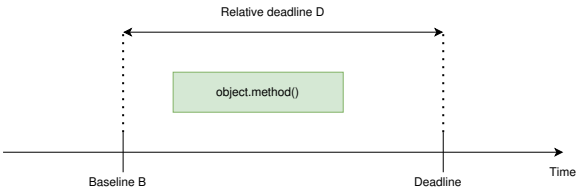
Listing 5 The FreeRTOS Task API

```
BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,
                       const char * const pcName,
                       configSTACK_DEPTH_TYPE usStackDepth,
                       void *pvParameters,
                       UBaseType_t uxPriority, // Task priority
                       TaskHandle_t *pxCreatedTask);
```

The fifth parameter in Line no. 5 accepts a static *priority* number, which a programmer uses to override the *fairness* of a task scheduler and customise the emergency of execution of each thread. However, with a limited set of priorities numbers (1 - 5) it is very likely for several concurrent tasks to end up with the same priority, leading the scheduler to order them fairly once again. A risk with priority-based systems is to run into the *priority inversion problem* [31], as explained in the Real-Time Java spec:

> If a lower-priority thread shares a resource with a higher-priority thread, and if that resource is guarded by a lock, the lower-priority thread may be holding the lock at the moment when the higher-priority thread needs it. In this case, the higher-priority thread is unable to proceed until the lower-priority thread has completed its work – and this can cause the higher-priority thread to miss its deadline.

TinyTimber implements scheduling based on dynamically changing priorities [21]. The programmer can specify that an operation should take place within a window of time specified by a baseline and a deadline. The deadline time is used to prioritize operations with the earliest deadlines, so-called Earliest-Deadline-First scheduling.



The figure above shows a TinyTimber timing window specified by the following code:

```
1 WITHIN(B, D, &object, method, arg);
```

The code above says that the method `method` of the object `object` should be run no sooner than time B and preferably finish in an interval of time D. TinyTimber prioritises the execution of tasks with the earliest deadlines. It does not detect or capture a missed deadline. If catching missed deadlines is desired, watchdog tasks can be implemented by the programmer.

2.1 Key Idea

In SenseVM, we combine the concurrency and communication of CML with a possibility to specify timing windows inspired by TinyTimber. The channel-based communication of CML is also extended to encompass communication with interrupt-based and memory-mapped IO peripherals on the target microcontrollers.

3 Programming for SenseVM

Writing programs for SenseVM is currently done in a functional, eagerly-evaluated, statically-typed Caml-inspired language. Type declarations and signatures in the language syntactically resemble those of Haskell. The language is also indentation aware. Comments are expressed using `--`. In this section we show small examples to introduce the programming primitives. Most of the initial primitives are directly inherited from CML.

3.1 Spawning processes

Just like in CML, a process (or thread) is started, "spawned", using the `spawn` primitive with the following type signature.

```
spawn : (() → ()) → ThreadId
```

The `spawn` primitive takes a function of type `() -> ()` as argument and returns a thread identifier. We currently do not use this `ThreadId` in our interface but envision future use that could provide information on the lifetime of a process.

A function `f` that wishes to execute asynchronously is started by calling `spawn f` . In SenseVM, application-level processes are started by calling `spawn` from the main-method. The SenseVM scheduler is *cooperative* (`syncT` later makes this *preemptive*), where when a process encounters a synchronous, blocking message-passing call, it yields the control to the scheduler that then schedules other threads. Next, we introduce communication primitives that communicate via inter-process message-passing.

3.2 Message passing

The programming interface supports a synchronous style of message-passing adopted from Concurrent ML [25]. Synchronous message-passing involves communication along blocking constructs called *channels*. A channel can be created using the following function:

```
channel : () → Channel t
```

A `channel ()` call creates a typed *channel* along which a process can send or receive a message of type t.

The Concurrent ML API for sending and receiving messages differs from other synchronous message-passing programming models like CSP [15]. The central idea of Concurrent ML is to break the act of synchronous communication into its two constituent steps:

(i) Expressing the intent of communication.
(ii) Synchronising the communication between the sender and receiver.

In the first step the programmer creates a value of type `Event`. In Concurrent ML, an `Event`-value is a first-class citizen of the language akin to the treatment of higher-order functions in functional languages. Reppy describes an `Event`-value as a "first-class synchronous operation". Message sending and receiving in the context of these `Event` values have the following types:

```
send : Channel a → a → Event ()
recv : Channel a → Event a
```

The `Event` construct was introduced to resolve a fundamental conflict that arises between procedural abstraction and the selective communication operation of CSP. We discuss the issue after introducing the `choose` operator in Section 3.4. Note that the operations `send` and `recv` are non-blocking and do not perform any actual communication.

Given a value of type `Event`, the second step of synchronising between processes and initiating the actual communication is performed via the `sync` operation, whose type is:

sync : Event a → a

The `sync` operation takes a value of type `Event a` and blocks until the communication intent encoded in the `Event` can be performed. Synchronous communication requires matching a message sender with a message receiver. So the process of synchronisation involves finding two processes that are attempting to communicate along a common channel, and upon finding such processes passing the message from the sender to the receiver. Until synchronisation happens between the two parties, the processes remain blocked, waiting for synchronisation.

If we consider the standard type-signatures of message sending and receiving in CSP, they will be $sendMsg : Channel\ a \rightarrow a \rightarrow ()$ and $recvMsg : Channel\ a \rightarrow a$. Intuitively, we can draw an equivalence between the message passing in CSP and the Concurrent ML-based message passing using function composition:

```
sync . (send c) ≡ sendMsg c
sync . recv     ≡ recvMsg
```

In Listing 6 the operators `sync`, `send` and `recv` are used to communicate between the `foo` and `bar` processes. The `send` and `recv` operations create values of type `Event ()` and `Event Int` respectively. Note that in the function, the expression of a *communication intent* is separate from *actual communication* and an `Event` is simply a description of the intent. To turn the intent into actual communication, we use the `sync` operator.

Listing 6 Communicating tasks

```
chan : Channel Int
chan = channel ()

foo : () -> ()
foo _ =
  let _ = sync (recv chan) in
  foo ()

bar : () -> ()
bar _ =
  let _ = sync (send chan 1) in
  bar ()

main =
  let _ = spawn foo in
  let _ = spawn bar in
  ()
```

The illustration in Figure 2 shows the scheduling of the above program. Figure 2 shows the time-share of the processor owned by each process and also the interim time slots owned by the scheduler that decides which process to schedule when.

The parts of the chart shaded in blue are the time slots where the scheduler decides which process to schedule next. If it finds all processes are blocked at a certain instant it can choose to go to sleep. Next, before looking at the `choose` operation for selective communication, we take a quick look at so-called post-synchronisation operations.

3.3 Post-synchronisation operation

The `wrap` function can be used to attach an operation (of type $a \rightarrow b$) that should be executed once the `sync` operator has completed synchronising an event. The type signature of `wrap` is:

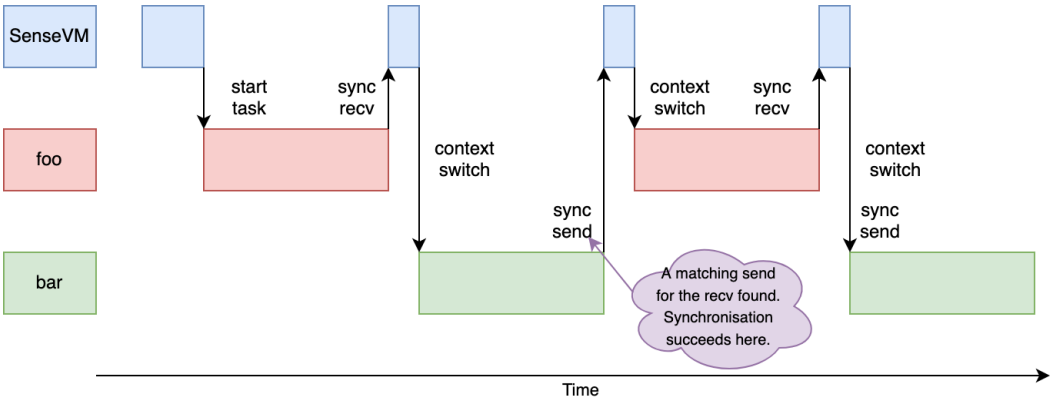


Figure 2 Scheduling the program in Listing 6

wrap : Event a → (a → b) → Event b

The important feature of `wrap` is that the result of wrapping a function over an event is an event. This becomes particularly important when constructing composite events using the `choose` operation that we look at next.

3.4 Selective communication

To enable multi-party communication, synchronous message-passing models introduce a *selective communication* operator that races between two operations and selects the one that completes first. This enables the handling of communication with multiple participants without being unnecessarily blocked by the synchronous nature of the communication.

However, Reppy identified a conflict that arises between selective communication and procedural abstraction [25]. The complication can be demonstrated via the example of a client-server communication protocol where the client follows the protocol - *first send a message along a channel, `reqCh`, and only upon the success of the send will it accept the server response along the channel, `respCh`.* Such a protocol can be expressed in synchronous models like CSP and then abstracted as a procedure like the following:

```
1 clientCall : (Channel a, Channel b) -> a -> b
2 clientCall (reqCh, respCh) a =
3   let _ = sendMsg reqCh a
4    in recvMsg respCh
```

Now imagine a scenario where the client is communicating with two servers and the first server is temporarily unavailable. In such an instance the `sendMsg` call in line no. 3 will block and as the `sendMsg` call has been abstracted away inside the procedure it is not possible to apply the *select* operation on it. Hence, for liveness the `sendMsg` operation should not be hidden in the procedure.

However, if we expose the `sendMsg` operation, it goes against the principles of software abstraction where the internal operations of a protocol are leaked and can authorize the programmer to write unsafe operations like a sequence of two `sendMsg` calls that violates the protocol invariant (a send should always be followed by a receive). The `Event` construct of Concurrent ML resolves this issue elegantly by programming the abstraction as follows:

```
clientCallEvt : (Channel a, Channel b) -> a -> Event b
clientCallEvt (reqCh, respCh) a =
  wrap (send reqCh a) (λ _ -> sync (recv respCh))
```

The `clientCallEvt` program above represents a server as a tuple of a request channel and a response channel. The use of `wrap` in `clientCallEvt` creates an event of type `Event b` (where `b` is the type of the values sent across the `respCh`). Send events have type `Event ()` so the function `(λ _ -> sync (recv respCh))` has type `() -> b`.

With the above abstraction, we can introduce the `choose` operator that allows selecting between two events and synchronises the event that become synchronisable first. The type signature of `choose` is as follows:

choose : Event a → Event a → Event a

Given two servers, `server1 = (server1ReqCh, server1RespCh)` and `server2 = (server2ReqCh, server2RespCh)`, multi-party communication can be expressed without breaking the procedural abstraction using `choose (clientCallEvt server1) (clientCallEvt server2)`.

The return type of a `choose` call will still be `Event`, allowing us to compose and *choose* among several synchronous operations like *choose* (*choose* (*choose* ev_1 ev_2) ev_3) ev_4... When `sync` is applied to such a composite event it will race among all the events, $ev_1, ev_2, ev_3, ..$, and synchronise on the operation that unblocks first.

In Listing 7, the `choose` operation is applied on two `recv` events on line no. 9 in `foo`. The function `foo` is notably non-terminating and it waits for a message along either one of the two channels. The message sending processes, `bar` and `baz` terminate immediately after sending one message each.

Listing 7 Selective communication

```
1  chanBar : Channel Int
2  chanBar = channel ()
3
4  chanBaz : Channel Int
5  chanBaz = channel ()
6
7  foo : () -> ()
8  foo _ =
9    let _ = sync (choose (recv chanBar) (recv chanBaz)) in
10   foo () -- non-terminating
11
12 bar : () -> ()
13 bar _ = sync (send chanBar 2) -- terminates
14
15 baz : () -> ()
16 baz _ = sync (send chanBaz 1) -- terminates
17
18 main =
19   let _ = spawn foo in
20   let _ = spawn bar in
21   let _ = spawn baz in
22   ()
```

Figure 3 shows the processor occupancy between the three processes and the scheduler of SenseVM for Listing 7. The process `foo` blocks and yields the control to the scheduler, which schedules `bar` to send a message. The process `bar` sends a message and terminates but enables the unblocking of `foo`. Next `foo` takes the message, completes the recursion and blocks again for a new message, which is sent by `baz` the second time around. An interesting point is the third iteration of `foo` where it doesn't have any sender, so it relinquishes control

to the scheduler that can decide to go to a power-saving mode.

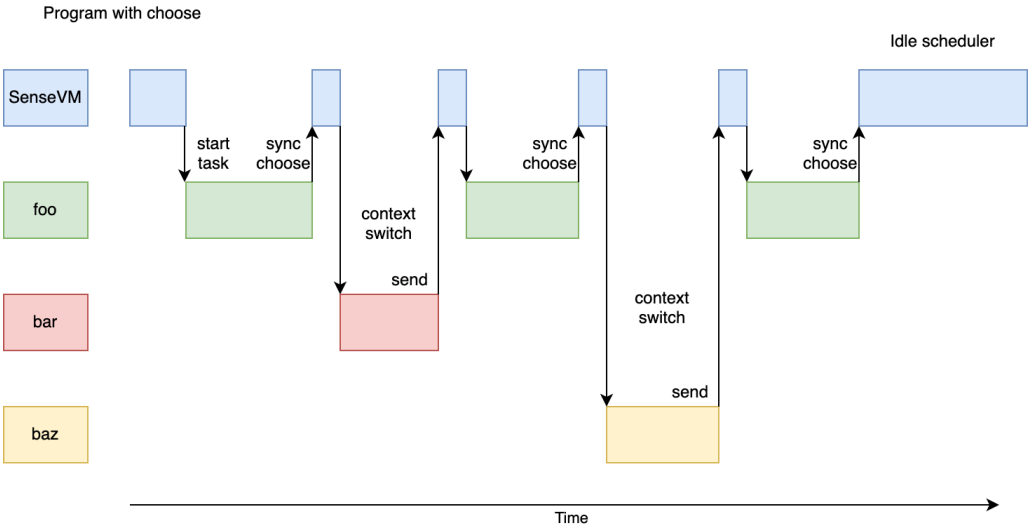


Figure 3 SenseVM using `choose`

3.5 Programming with IO

When programming for SenseVM, IO is expressed using the same events as are used for inter-process communications. Each IO device is connected to the running program using a primitive we call `spawnExternal` as a hint that the programmer can think of, for example, an LED as a process that can receive messages along a channel. Each "external" process denotes an underlying IO device that is limited to send and receive messages along one channel.

The `spawnExternal` primitive takes the channel to use for communication with software and a driver and returns a "ThreadId" for symmetry with `spawn`.

spawnExternal : Channel a → Driver → ExternalThreadId

The first parameter supplied to `spawnExternal` is a designated fixed channel along which the external process shall communicate. The second argument requires some form of an identifier to uniquely identify the driver. This identifier for a driver tends to be architecture-dependent. For instance, when using low-level memory-mapped I/O, reads or writes to a memory address (within the same address space as the program memory) are used to communicate with a peripheral. So the unique memory address would be an identifier in that case. On the other hand, certain real-time operating system (such as `FreeRTOS` or `Zephyr`) can provide more high-level abstractions over a memory address. In SenseVM, we currently number each peripheral in monotonically increasing order, starting from 0. So the SenseVM `spawnExternal` API becomes:

```
type DriverNo = Int
spawnExternal : Channel a -> DriverNo -> ExternalThreadId
```

In the rest of the paper, we will use suggestive names for drivers like `led0`, `uart1`, etc instead of plain integers for clarity. We have ongoing work, in SenseVM, to parse a file describing the target board/MCU system, automatically number the peripherals and emit typed declaration like $led0 = LED\ 0$ that can be used in the `spawnExternal` API.

Fig 4 provides an intuition of the I/O interactions within SenseVM. The software processes interact with each other as well as send and receive messages from the external hardware processes, using the very same message-passing API. The typed channels prevent the passing of ill-formed/garbage messages to the hardware processes. Certain drivers could be exclusively read-only/write-only, like the driver for a temperature sensor that is read-only. However, if a programmer attempts to send a message to such a driver, the failure handling is implementation-dependent. In SenseVM we ignore ill-formed messages.

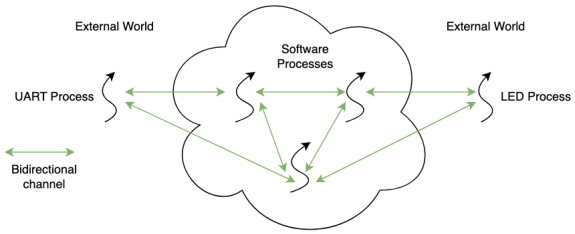


Figure 4 Software processes and hardware processes interacting

To demonstrate the I/O API for asynchronous drivers, we present a standard example of the *button-blinky* program from Listing 3. The program matches a button state to an led so that when the button is down, the LED is on, otherwise the LED is off:

Listing 8 Button-Blinky on the SenseVM

```
1  butchan : Channel Int
2  butchan = channel ()
3
4  ledchan : Channel Int
5  ledchan = channel ()
6
7  glowled : Int -> ()
8  glowled i = sync (send ledchan i)
9
10 foo : ()
11 foo =
12  let _ = sync (wrap (recv butchan) glowled) in
13  foo
14
15 main =
16   let _ = spawnExternal butchan 0 in
17   let _ = spawnExternal ledchan 1 in
18   foo
```

The above program represents an asynchronous, callback-based application in an entirely synchronous framework. The same application written in C, on top of the Zephyr OS, is more than 100 lines of callback-based code [10]. A notable aspect of the above SenseVM program is the lack of any callback-registration mechanism. We show the timing chart of Listing 8 below.

Comparing Listing 8 with Fig. 7, the program arrives at line 12 and blocks while awaiting a message from the button driver. At this point, the control of the program is relinquished to the scheduler and if it doesn't find any other threads ready to proceed, is free to go to sleep to reduce power usage. The button driver is internally implemented using interrupts that fire on both button-down and button-up hardware events. When an interrupt arrives, the driver notifies the scheduler and the scheduler relays the information about the button press to the process `foo`. `foo` resumes its execution from line 12 by message-passing the button input to the LED driver in line 8. The LED driver, being synchronous, accepts the

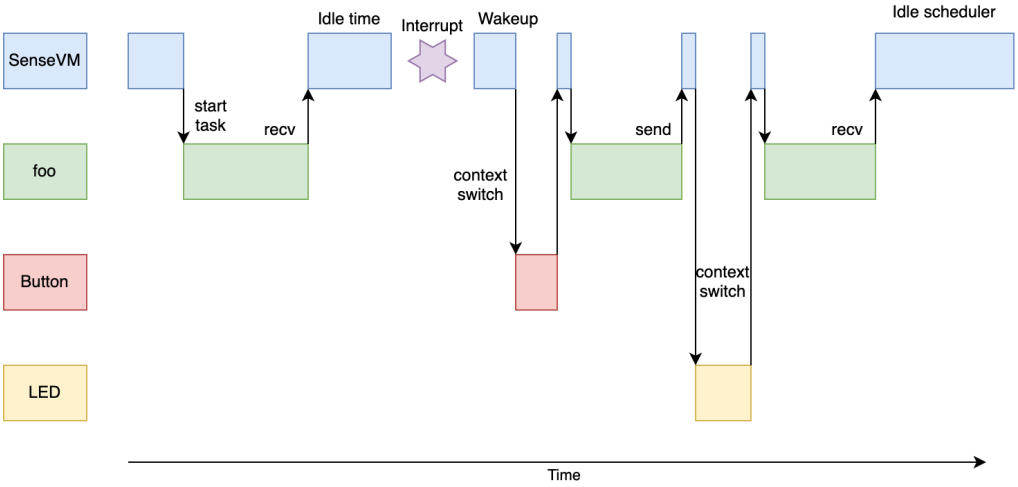


Figure 5 SenseVM using `spawnExternal`

input immediately, unblocking the program and proceeding to the tail-recursive call at line 13 that continues the loop infinitely.

3.6 Programming with Time

Real-time languages and frameworks generally provide APIs to override the fairness of a *fair* scheduler. A typical fair scheduler abstracts away the details of ordering its constituent processes (or threads). However, in a real-time scenario a programmer wants to precisely control the response-time of certain operations. So the natural intuition for real-time C-extensions like `Free-RTOS Tasks` or languages like Ada is to delegate the scheduling control to the programmer by allowing them to attach *priorities* to various processes. If we approach real-time programming with a similar API, we can attempt the following:

```
1 main = ...
2   spawn process1 HIGH;
3   spawn process2 MEDIUM;
4   spawn process3 MEDIUM
```

The `HIGH` and `MEDIUM` tags can indicate the order in which a tie is to be broken by the scheduler. However, with a limited number of tags (`HIGH`, `MEDIUM`, `LOW`) it is very likely for several processes to end up with the same priority, leading the scheduler to order them fairly once again.

Another complication that crops up in the context of priorities is the *priority inversion problem* [31]. Priority inversion is a form of resource contention where a high-priority thread gets blocked on a resource held by a low-priority thread, thus allowing a medium priority thread to take advantage of the situation and get scheduled first. The outcome of this scenario is that the high-priority thread gets to run after the medium-priority thread, leading to possible program failures.

The SenseVM approach to time takes inspiration from the TinyTimber kernel [21] that implements *dynamic* priorities based on the deadline of a task. Our programming interface allows a programmer to specify a *time window* (of the wall-clock time) at which we want a particular thread to run. The timing window consists of a baseline time and a deadline time. Once the baseline time has been reached the scheduler will be made aware that the thread

is ready to execute. Threads that are ready to execute are ordered according to deadline, implementing earliest deadline first scheduling. In essence, a thread becomes increasingly more prioritised as its deadline approaches. Just like the TinyTimber kernel, SenseVM does not track missed deadlines and is thus implementing a soft approach to real-time execution. Being a bytecode interpreting VM with a mark and sweep garbage collector, currently, is also a challenge for the implementation of application with very tight and precise timing requirements. We discuss this further in Section 8, Future Work.

We extend our interface by adding a single operator, the timed synchronisation operator - `syncT`, for expressing time. The type signature of `syncT` is given below:

syncT : Time → Time → Event a → a

Comparing the type signature of `syncT` with that of `sync` :

```
syncT : Time -> Time -> Event a -> a
sync  :                 Event a -> a
```

The two extra arguments to `syncT` specify a lower and upper bound on the *time of synchronisation* of an event. We assume the existence of a standard wall-clock time source that can reliably supply us with an exact timestamp at the desired level of granularity. The two arguments to `syncT`, of type `Time`, express the relative times calculated from the current wall-clock time. The first argument represents the *relative baseline* - the earliest time instant from which the event synchronisation should begin. The second argument specifies the *relative deadline* i.e the latest time instant (starting from the baseline) by which the event synchronisation should preferably complete. For instance, if we take an event, *timed_ev*, and write :

```
syncT (msec 50) (msec 20) timed_ev
```

The above means that the synchronisation of *timed_ev* should begin 50 milliseconds from *now* (what *now* means is explained below) and the synchronisation should complete within 20 milliseconds of its beginning. Synchronisation, in this framework, implies the entire process of synchronous communication, including sending a message along a channel, blocking till a receiver is found, passing the message from the sender to the receiver and then unblocking both the sender and the receiver. We illustrate the synchronisation timeline in Figure 6.

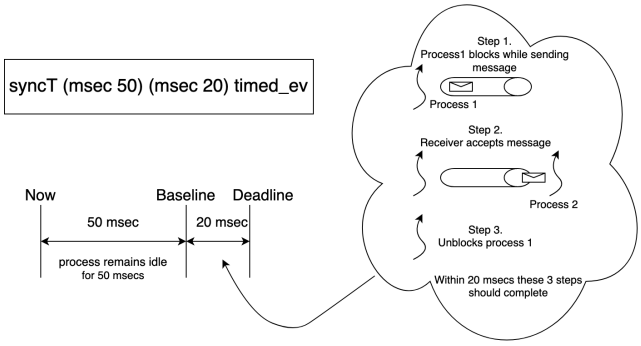


Figure 6 Timed synchronisation in action

The *now* concept is based on a thread's local view of what time it is. This thread-local time (T_{local}) is always less than or equal to wall-clock time ($T_{absolute}$). Wall-clock time is

maintained by a continuously running monotonically increasing timer that is started when the SenseVM system boots up. When a thread is spawned, its thread-local time, T_{local}, is set to the wall-clock time, $T_{absolute}$.

While a thread is running its local time is frozen and unchanged until the thread executes a timed synchronisation, a `syncT` operation where time progresses to $T_{local} + baseline$.

```
process1 _ =
  let _ = s1 in -- T_local = 0
  let _ = s2 in -- T_local = 0
  let _ = syncT (msec 50) (usec 10) ev1 in
  process1 () -- T_local = 50 msec
```

The example above illustrates that the untimed operations `s1` and `s2` have no impact on the thread's view of what time it is. In essence these operations are considered to take no time, which of course is not true. This is however an assumption that SenseVM shares with other systems such as ChucK [34] and the Sparse Synchronous Model [8]. Most importantly, this is an assumption that is very good to make, as it helps with controlling jitter in the timing as long as the timing windows specified on the synchronisation is large enough to contain the execution time of `s1`, `s2`, the synchronisation step and the recursive call.

So, the concept of *now* refers to the thread-local time (T_{local}) and executing a `syncT` progresses thread-local time to $now + baseline$. For this to make sense, local time and wall-clock time must match up at certain points. To ensure that thread-local time and wall-clock time matches up an alarm is set in the VM at the wall-clock time $now + baseline$ when a `syncT` is encountered. The thread is then yielded until that alarm goes off. When the alarm goes off, wall-clock time is exactly $now + baseline$ and the thread is made ready to execute again with updated thread-local time.

To see how this approach to timing helps with jitter, imagine if the alarm set at `syncT` was set to $wallclock\ time + 50ms$. In this case time progresses as we execute `s1` and `s2` as well as in the tail-recursive call to `process1`, any variability in this would manifest as jitter on the synchronisation and would keep on adding with each recursive iteration. With our approach, however, and as long as any time spent on untimed operations in `process1` is less than the timing window, jitter should be under control. To guarantee this on a larger scale with many involved processes and timing windows is not easy and would require worst-case execution time and schedulability analyses, which is beyond the scope of this work.

3.6.1 Blinky

We present the popular *blinky* example, which constitutes blinking an LED at a certain frequency. The program switches the LED ON for 1 second and then switches it OFF for 1 second and so on. Following show the program that runs on the `STM32F4`-discovery board -

Listing 9 Blinky with `syncT`

```
not : Int -> Int
not 1 = 0
not 0 = 1

ledchan : Channel Int
ledchan = channel ()

sec : Int -> Int
sec n = n * 1000000

usec : Int -> Int
usec n = n -- the unit-time in SenseVM

```

```
foo : Int -> ()
foo val =
 let _ = syncT (sec 1) (usec 1) (send ledchan val) in
 foo (not val)

main =
  let _ = spawnExternal ledchan 1 in
  foo 1
```

In the above program, we have `foo` as the only software process and one external hardware process for the LED driver that is restricted to communicate along the `ledChan` channel.

Figure 7 below demonstrates the timing chart of the above program. This chart is more involved as it has two clocks. The actual wall-clock time, $T_{absolute}$, is represented along the X-axis while the process-local clock, T_{local}, for the process `foo` is shown inside the body of green chart representing `foo`.

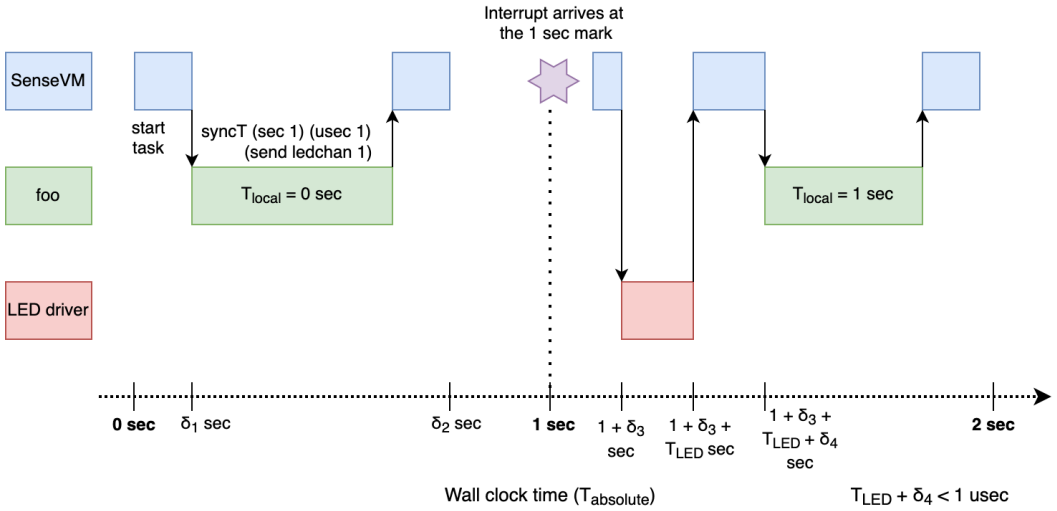


Figure 7 SenseVM timeline for the *blinky* program

As shown in Fig 7 the T_{local} clock is initialised to the time at the very beginning of the system that is 0. When the program arrives at the `syncT` statement, an alarm is set for the time at which the VM should begin attempting communication with the LED driver. The alarm is set exactly at the 1-second mark, calculated from the T_{local} clock, which removes the jitter associated with other statement executions and VM overheads.

The timer interrupt arrives at the 1-second mark and the VM begins communication with the LED driver taking δ_3 seconds. Once the communication is initiated, the deadline counter becomes effective. The LED driver process take T_{LED} seconds to execute, and the scheduler takes an additional δ_4 time units to unblock the process `foo`. So, the deadline requested to the runtime follows the relation - $\delta_4 + T_{LED} < 1\ usec$. Finally, T_{local} is incremented by the relation $T_{local} = T_{local} + baseline$, where the baseline is 1 second in our case and the program continues.

Having introduced the concurrency, I/O and timing APIs of SenseVM, we shall next look at example applications running on the VM.

4 Finite-State Machines on the SenseVM

We shall now illustrate two larger examples of expressing state machines involving callback-based APIs running on the SenseVM. For running our examples, we choose the `NRF52840DK` microcontroller board, which comes equipped with four buttons and four LEDs. We particularly choose the button peripheral because its drivers have a callback-based API that could lead to non-linear control-flows within the program. The programs presented in this section notably doesn't elide any part of the program and can be compiled and run unmodified on an `NRF52840DK` board.

4.1 Four-Button-Blinky

We build on the *button-blinky* program from Listing 8 presented in Section 3.5. The original program, upon a single button-press, would light up an LED corresponding to that press and switch off upon the button release. We now extend that program to produce a one-to-one mapping between four LEDs and four buttons such that button1 press lights up LED1, button2 press lights up LED2, button3 press lights up LED3 and button4 press lights up LED4 (while the button releases switch off the corresponding LEDs). Figure 8 shows the state-machine diagram of this application.

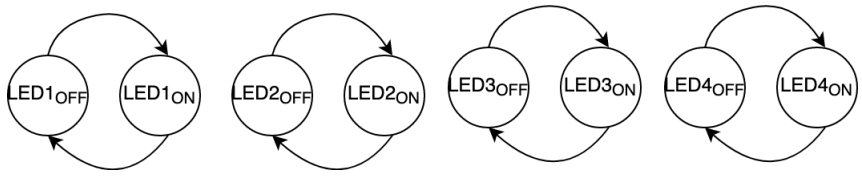


Figure 8 The Four-Button-Blinky FSM

Fig 8 shows a case where four separate state machines, operating on a single MCU board, can be composed together into a single function on SenseVM. As each button-LED combination has its own state machine we can use the `choose` operator to compose the four combinations. Listing 10 show the entire *four-button-blinky* program running on SenseVM.

Listing 10 The Four-Button-Blinky program running on the SenseVM

```
butchan1 = channel ()
butchan2 = channel ()
butchan3 = channel ()
butchan4 = channel ()

ledchan1 = channel ()
ledchan2 = channel ()
ledchan3 = channel ()
ledchan4 = channel ()

press1 = wrap (recv butchan1) (λ x -> sync (send ledchan1 x))
press2 = wrap (recv butchan2) (λ x -> sync (send ledchan2 x))
press3 = wrap (recv butchan3) (λ x -> sync (send ledchan3 x))
press4 = wrap (recv butchan4) (λ x -> sync (send ledchan4 x))

anybutton = choose press1 (choose press2 (choose press3 press4))

program : ()
program =
 let _ = sync anybutton in
 program

main =
```

```
  let _ = spawnExternal butchan1 0 in
  let _ = spawnExternal butchan2 1 in
  let _ = spawnExternal butchan3 2 in
  let _ = spawnExternal butchan4 3 in
  let _ = spawnExternal ledchan1 4 in
  let _ = spawnExternal ledchan2 5 in
  let _ = spawnExternal ledchan3 6 in
  let _ = spawnExternal ledchan4 7 in
  program
```

Listing 10 utilises eight hardware process, four to model the LED drivers and the remaining four for the buttons. The key composition happens on Line no. 16 where we compose the four state machines together using the `choose` operator. The main body of the program synchronises the resulting event (`anybutton`) and keeps the main-loop alive by calling itself.

4.2 A more intricate FSM

We now construct a more intricate finite-state machine involving intermediate states that can move to an error state if the desired state-transition buttons are not pressed. We configure our button driver now to send only one message per button press-and-release. So there is no separate button-on and button-off signal but one signal per button.

In this FSM, we glow the LED1 upon consecutive presses of button1 and button2. We use the same path to turn LED1 off. However, if a press on button1 is followed by a press of button 1 or 3 or 4, then we move to an error state indicated by LED3. We use the same path to switch off LED3. In a similar vein, consecutive presses of button3 and button4 lights up LED2 and button3 followed by button 1 or 2 or 3 lights up the error LED - LED3. Figure 9 demonstrates the FSM diagram of this application omitting self-loops in the OFF state.

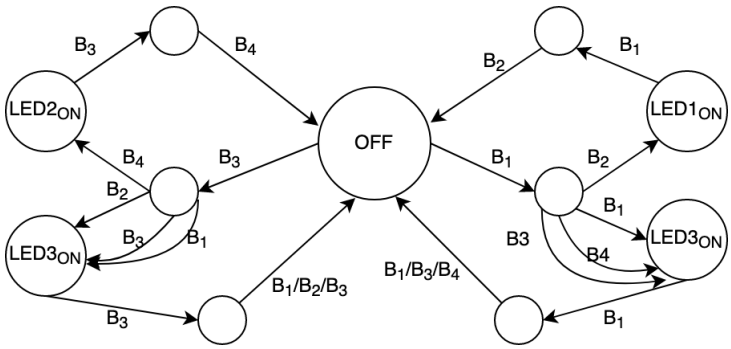


Figure 9 A complex state machine

We encode the FSM of Fig 9 on the SenseVM in Listing 11. This FSM can be viewed as a composition of two separate finite state machines, one on the left side of the OFF state involving LED2 and LED3 and one on the right side involving LED1 and LED3. Once again we utilise `choose` operator to compose these two state machines.

Listing 11 The complex state machine running on the SenseVM

```
butchan1 : Channel Int
butchan1 = channel ()
butchan2 : Channel Int
butchan2 = channel ()
butchan3 : Channel Int
butchan3 = channel ()
butchan4 : Channel Int
butchan4 = channel ()
```

```
ledchan1 : Channel Int
ledchan1 = channel ()
ledchan2 : Channel Int
ledchan2 = channel ()
ledchan3 : Channel Int
ledchan3 = channel ()
ledchan4 : Channel Int
ledchan4 = channel ()

not : Int -> Int
not 1 = 0
not 0 = 1

errorLed x = ledchan3

fail1ev = choose (wrap (recv butchan1) errorLed)
         (choose (wrap (recv butchan3) errorLed)
                 (wrap (recv butchan4) errorLed))

fail2ev = choose (wrap (recv butchan1) errorLed)
         (choose (wrap (recv butchan2) errorLed)
                 (wrap (recv butchan3) errorLed))

led1Handler x =
   sync (choose (wrap (recv butchan2) (\x -> ledchan1)) fail1ev)

led2Handler x =
   sync (choose (wrap (recv butchan4) (\x -> ledchan2)) fail2ev)

led : Int -> ()
led state =
  let fsm1 = wrap (recv butchan1) led1Handler in
  let fsm2 = wrap (recv butchan3) led2Handler in
  let ch = sync (choose fsm1 fsm2) in
  let _ = sync (send ch (not state)) in
  led (not state)

main =
  let _ = spawnExternal butchan1 0 in
  let _ = spawnExternal butchan2 1 in
  let _ = spawnExternal butchan3 2 in
  let _ = spawnExternal butchan4 3 in
  let _ = spawnExternal ledchan1 4 in
  let _ = spawnExternal ledchan2 5 in
  let _ = spawnExternal ledchan3 6 in
  let _ = spawnExternal ledchan4 7 in
  led 0
```

In Listing 11, the `led1Handler1` and `ledHandler2` functions capture the intermediate states after one button press, when the program awaits the next button press. The error states are composed using the `choose` operator in the functions `fail1ev` and `fail2ev`.

The compositional nature of our framework is visible in line no. 43 where we compose the two state machines, `fsm1` and `fsm2`, using the `choose` operator. Synchronising on this composite event returns the LED channel (demonstrating a higher-order approach) on which the process should attempt to write. This program is notably a highly callback-based, reactive program that we have managed to represent in an entirely synchronous framework.

In the next section we look at a music playing application example that notably uses the `syncT` operator and the rest of the functions from our programming interface.

5 A soft-realtime music playing example

We present a soft-realtime music playing exercise from a Real-Time Systems course, running on the SenseVM. We choose a simple tune, the first fourteen notes of the popular nursery rhyme - "Twinkle, Twinkle, Little Star". At the end of the fourteen notes, our program circles back and start playing the tune from the beginning.

We run this program on the `STM32F4`-discovery board that comes with a 12-bit digital-to-analog converter (DAC), which we connect to a speaker as a peripheral. We can write a value between 0 to 4095 to the DAC driver that gets translated to a voltage between 0 to 3V on the DAC output pin.

To produce a sound note we need to periodically write a sequence of 1's and 0's to the DAC driver. However, to make the produced note sound musical to the human ear, the *periodic rate* at which our process writes to the DAC driver is very important, and this is where the real-time aspect of the application comes in. The human ear recognises a note produced at a certain frequency as a musical note. Frequency is related to the periodic rate of a process by the relation:

$$Period = 1/Frequency$$

For instance, the musical note A occurs at a frequency of 440 Hz, which implies it has a time period of 2272 μseconds. From the point of view of the software, we are actually writing two values, a 1 and a 0, so we need to further divide the value by 2 to determine our rate of each individual write. If we call the rate of our writes as $Time_{Write}$, we get the relation -

$$Time_{Write} = Period/2 = 1/(2 * Frequency)$$

Now that we know how to calculate the periodicity of our write in relation to the frequency, we need to know (i) what are the musical notes that occur in the "Twinkle, Twinkle" rhyme and (ii) what are the frequencies corresponding to those notes so that we can calculate the $Time_{Write}$ value from the frequency. The musical notes of the "Twinkle, Twinkle" tune are well known and is given below:

Twinkle, Twinkle - **C C G G A A G F F E E D D C**

Given the above notes, the frequency of each of these notes are also well known. In Table 1 we show our calculation of the $Time_{Write}$ value for the various musical notes.

Note	Frequency (Hz)	Period (μsec)	Time$_{Write}$ (μsec)
C	261	3830	1915
D	294	3400	1700
E	329	3038	1519
F	349	2864	1432
G	392	2550	1275
A	440	2272	1136
B	493	2028	1014

Table 1 Musical notes, their frequencies and time periods

Now we need to specify the time duration of each note. At the end of each note's duration period, we change the frequency of writes to the DAC driver. For instance, consider the transition from the second to the third note of the tune from C to G. If the note duration for C is 1000 milliseconds then that implies our writing frequency should be 261 Hz for 1000 milliseconds, and then at the 1001st millisecond the frequency changes to 392 Hz (G's frequency).

When describing a proper musical etude, each note should be ideally mapped to its distinct duration in the program. However, to keep our illustration slightly simpler we set a static note duration of 500 milliseconds for each note.

Listing 12 shows the entire program running on the SenseVM that cyclically plays the "Twinkle, Twinkle, Little Stars" tune. The first 15 lines consists of declarations initialising a `List` data type and other standard library functions. Lines 39 - 60 consist of the principal logic of the program. Listing 12 can be compiled and run, **unaltered**, on an STM32F4-discovery board.

Listing 12 The *Twinkle, Twinkle* tune running on the SenseVM

```
1  data List a where
2    Nil  : List a
3    Cons : a -> List a -> List a
4
5  head : List a -> a
6  head (Cons x xs) = x
7
8  tail : List a -> List a
9  tail Nil = Nil
10 tail (Cons x xs) = xs
11
12 not : Int -> Int
13 not 1 = 0
14 not 0 = 1
15
16 msec : Int -> Int
17 msec t = t * 1000
18
19 usec : Int -> Int
20 usec t = t
21
22 after : Int -> Event a -> a
23 after t ev = syncT t 0 ev
24
25 twinkle : List Int
26 twinkle = Cons  1915 (Cons 1915 (Cons 1275 (Cons 1275 (Cons 1136
27           (Cons 1136 (Cons 1275 (Cons 1432 (Cons 1432 (Cons 1519
28           (Cons 1519 (Cons 1700 (Cons 1700 (Cons 1915 Nil)))))))))))))
29
30 dacC : Channel Int
31 dacC = channel ()
32
33 noteC : Channel Int
34 noteC = channel ()
35
36 noteDuration : Int
37 noteDuration = msec 500
38
39 playerP : List Int -> Int -> () -> ()
40 playerP melody n void =
41   if (n == 14)
42   then let _ = after noteDuration (send noteC (usec (head twinkle))) in
43        playerP twinkle 2 void
44   else let _ = after noteDuration (send noteC (usec (head melody))) in
45        playerP (tail melody) (n + 1) void
```

```
46
47
48 tuneP : Int -> Int -> () -> ()
49 tuneP timePeriod vol void =
50   let newtp =
51       after timePeriod (choose (recv noteC)
52                                (wrap (send dacC (vol * 4095))
53                                      (λ _ -> timePeriod))) in
54   tuneP newtp (not vol) void
55
56 main =
57   let _ = spawnExternal dacC 0 in
58   let _ = spawn (tuneP (usec (head twinkle)) 1) in
59   let _ = spawn (playerP (tail twinkle) 2) in
60   ()
```

Our application consists of two software processes and one external hardware process. The $Time_{Write}$ values of each of the fourteen notes are represented as the list `twinkle` on Lines 25-28. We use two channels - `dacC` to communicate with the DAC and `noteC` to communicate between the two software processes. Looking at what each software process is doing -

- `playerP`. The `playerP` process runs at the rate of 500 milliseconds per cycle. It wakes up every 500 millisecond, traverses the list `twinkle` one element at a time and sends the value of that element (the $Time_{Write}$ value) along the `noteC` channel. As the list contains 14 notes, `playerP` circles back to the head of the list after 14 notes.
- `tuneP`. The `tuneP` process is responsible for actually creating the sound. Its running rate varies depending on the note that is being played. For instance, when playing note C, it will write to the DAC at at a rate of 1915 microseconds-per-write. However, upon receiving a new value of $Time_{Write}$ along `noteC`, it changes its write frequency to the new $Time_{Write}$ value resulting in changing the note of the song.

Owing to the different time periods of the two processes, their T_{local} clock progresses at different rates. In Figure 10 we visualise the message passing that occurs between the two software process and the hardware process when transitioning from the second note C to the third note G.

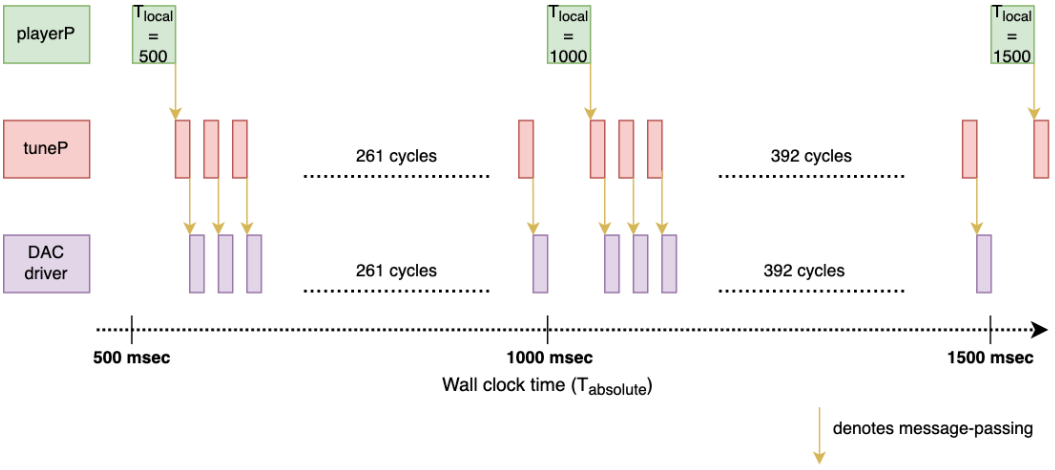


Figure 10 Moving from the note C to note G

As the `playerP` process runs once every 500 milliseconds, the `tuneP` process completes $500 * 10^3/1915 = 261 cycles$ when playing the note C. For the next note, G, the $Time_{Write}$ value changes to 1432 microseconds and the corresponding write frequency changes to 392 cycles and the process cyclically carries on.

There is an additional factor of the *pitch* at which we play our song. We play this tune at the standard `A440` Stuggart pitch that corresponds to playing the note A at a frequency of 440 Hz (see Table 1). If we wish to play the music at a higher pitch then that would involve increasing the frequency of our writes, however, there exists a threshold frequency at which SenseVM cannot meet the real-time deadlines and jitters start getting introduced into the tune. We discuss such evaluations on our application's performance in Section 7. In the next section we shall look at the design and implementation of the various components of SenseVM.

6 Design and Implementation

We discuss the design and implementation of our bytecode-interpreted virtual machine - SenseVM. The execution unit of SenseVM is based on the Categorical Abstract Machine (CAM) [6], as explained by Hinze [14]. CAM supports the cheap creation of closures, as a result of which SenseVM can support a functional language quite naturally. We start by giving a general overview of the compilation and runtime pipeline of SenseVM.

6.1 System Overview

Figure 11 shows the architecture of SenseVM. The whole pipeline consists of three major components - (i) the frontend, (ii) the middleware and (iii) the backend.

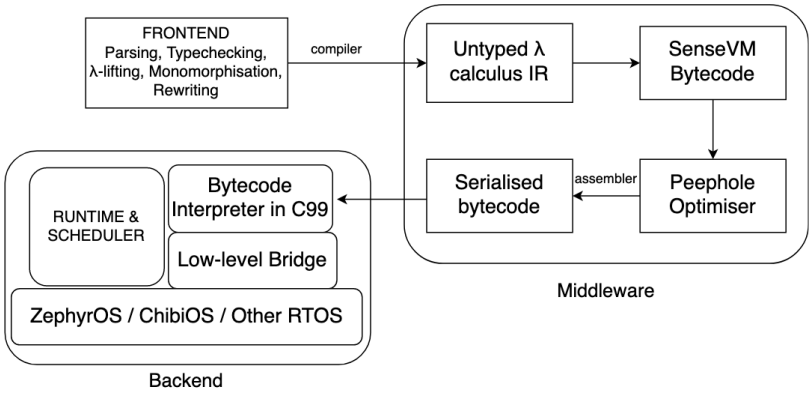


Figure 11 The compiler and runtime for SenseVM

Frontend. We support a statically-typed, eagerly-evaluated, Caml-like functional language on the frontend. The language comes equipped with Hindley-Milner type inference. The polymorphic types are monomorphised at compile-time. The frontend module additionally runs a lambda-lifting pass to reduce the heap-allocation of closures.

Middleware. The frontend language gets compiled to an untyped lambda-calculus-based intermediate representation. This intermediate representation is then further compiled down to the actual bytecode that gets interpreted at runtime. The generated bytecode emulates operations on a stack machine with a single environment register that holds the final value of a computation. This module generates specialised bytecodes that reduce the environment

register-lookup using an operational notion called *r-free* variables described by Hinze [14]. Followed by the bytecode generation, a peephole-optimisation pass is run on the bytecode that applies further optimisations like β-reduction and *last-call optimisation* [14] (a generalisation of tail-call elimination).

Backend. The backend module is the principal component of the SenseVM. It can be further classified into two distinct segments - (i) the high-level modules, (ii) the low-level OS/driver support and a low-level bridge connecting the two segments.

- High-Level Modules. The high-level modules include an interpreter written in C99 (for portability) that operates on a fixed-size stack with an environment register. To emulate multiple threads/processes, SenseVM consists of multiple *contexts*, where each context holds its own fixed-size stack, the environment register and a program counter to indicate which bytecode is being interpreted. Owing to memory constraints in embedded systems, we restrict the VM to contain a fixed number of contexts. A context also holds two additional registers - one to indicate the process-local clock (T_{local}) and the second register to hold the deadline of that specific context (or thread).
  An important part of the high-level modules is a garbage-collected heap to support closures and other composite values like tuples, algebraic data types, etc. The heap is structured as a list made of uniformly-sized tuples. For garbage collection, SenseVM utilises a mark-and-sweep algorithm with the Hughes lazy-sweep optimisation [16] that frees the memory on demand to keep the mark-and-sweep pass shorter. Another optimisation applied to our GC is the Deutsch-Schorre-Waite algorithm [18, 26] that enables a pointer-reversal-based algorithm that doesn't require extra stack space to complete the marking phase of our garbage collector.
- Low-level OS/drivers. The lowest level of SenseVM uses a real-time operating system that provides drivers for interacting with the various peripherals. The low-level is not restricted to use any particular OS, as demonstrated in our examples using both the Zephyr-OS and ChibiOS. The low-level interacts with the high-level components via a *bridge* interface that we discuss in depth in Section 6.6.

6.1.1 Concurrency, I/O and Timing bytecodes

For accessing the operators of our programming interface as well as any general runtime-based operations, SenseVM has a dedicated bytecode - `CALLRTS n`, where `n` is a natural number to disambiguate between operations. Table 2 shows the bytecodes corresponding to our programming interface.

A notable aspect is the handling of the `syncT` operation, which gets compiled into two separate bytecodes. The first bytecode interpretation (CALLRTS 8) calculates the timing window, in terms of absolute time, at which a thread should run. It then schedules the thread to run at that exact time such that when it finally runs, it encounters the synchronisation bytecode (CALLRTS 4) as the next bytecode to be interpreted (discussed in depth in Section 6.4). We next look at the runtime implementation of the message passing-operations within the `Event` framework.

6.2 Message-passing with events

All forms of communication and I/O in SenseVM operate via synchronous message-passing. However, a distinct aspect of SenseVM's message-passing is the separation between the *intent* of communication and the actual communication. When we describe the intent of communication we create a value of type `Event`.

Operation	Bytecode
spawn	CALLRTS 0
channel	CALLRTS 1
send	CALLRTS 2
recv	CALLRTS 3
sync	CALLRTS 4
choose	CALLRTS 5
spawnExternal	CALLRTS 6
wrap	CALLRTS 7
syncT	CALLRTS 8; CALLRTS 4

Table 2 Concurrency, I/O and Timing bytecodes

An event-value, like a closure, is a concrete runtime value allocated on the heap. The fundamental event-creation primitives are `send` and `recv`, which Reppy calls base-event constructors [25]. The event composition operators like `choose` and `wrap` operate on these base-event values to construct larger events. When a programmer attempts to send or receive a message, an event-value captures the channel number on which the communication was desired. When this event-value is synchronised (synchronisation is discussed in the next section), we use the channel number as an identifier to match between prospective senders and receivers. Listing 13 shows the heap representation of an event-value as the the type `event_t` and the information that the event-value captures on the SenseVM.

Listing 13 Representing an `Event` in SenseVM

```
typedef enum {
   SEND, RECV
} event_type_t;

typedef struct {
  event_type_t e_type; //  8 bits
  UUID channel_id;     //  8 bits
} base_evt_simple_t;

typedef struct {
  base_evt_simple_t  evt_details; // stored with 16 bits free
  cam_value_t wrap_func_ptr; // 32 bits
} base_event_t;


typedef struct {
  base_event_t bev; // 32 bits
  cam_value_t  msg; // 32 bits; NULL for recv
} cam_event_t;

typedef heap_index event_t;
```

An event is essentially a linked list, and the composition operation `choose` adds more nodes to this list. Each element of the list captures (i) the message that is being sent or received, (ii) any function that is wrapped around the base-event using `wrap`, (iii) the channel being used for communication and (iv) an enum to distinguish whether the base-event is a `send` or `recv`. Fig 12 visualises an event upon allocation to the SenseVM heap.

The linked-list, as shown above, is the canonical representation of an `Event`-type. It can represent any complex composite event. For instance, if we take an example composite event

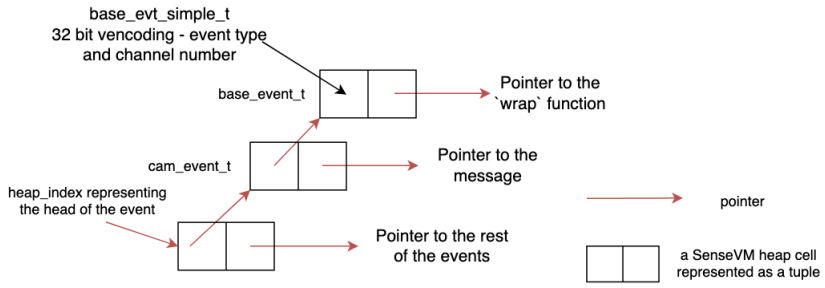


Figure 12 An event on the SenseVM heap

that is created using the base-events, e_1, e_2, e_3 and a wrapping function wf_1, it can always be rewritten to its canonical form.

```
choose e1 (wrap (choose e2 e3) wf1)

-- Rewrite to canonical form --

choose e1 (choose (wrap e2 wf1) (wrap e3 wf1))
```

The `choose` operation can simply be represented as *cons*ing on the event list. After expressing the intent of communication as events, the next operation involves synchronising an event via `sync`.

6.3 Synchronising events

The synchronisation operation `sync` is the most elaborate operation on the SenseVM. It accepts an event as an argument, which is represented as an event-list. The algorithm traverses this event list, detects the base-event that has a sender or receiver blocked on communication and passes the message between the two parties. Algorithm 1 provides a high-level, bird's eye view of the synchronisation algorithm.

Algorithm 1 The synchronisation algorithm

Data: *eventList*
$ev \leftarrow findSynchronisableEvent(eventList)$;
if $ev \neq \varnothing$ **then**
  $syncNow(ev)$;
else
  $block(eventList)$;
  $dispatchNewThread()$;
end

We will begin explaining the synchronisation algorithm by looking at the*findSynchronisableEvent* function first. To describe this function we need to understand the structure of a channel. A channel constitutes of two queues, (i) a send queue and (ii) a receive queue, which track the thread-id of the senders and receivers respectively. So the actual message is never held by a channel. For communication with a hardware peripheral, a channel tracks the driver number of the communicating driver. With this information we can now describe *findSynchronisableEvent* in Algorithm 2.

The *llBridge* mentioned above refers to the low-level bridge that facilitates reading and writing from hardware peripherals. We discuss it in more detail in Section 6.6.

Algorithm 2 The findSynchronisableEvent function

```
Data: eventList
Result: A synchronisable event or ∅
foreach e ∈ eventList do
    if e.channelNo communicating with driver then
        if llBridge.driver readable/writeable? then
            return e;
        end
    else
        if e.baseEventType == SEND then
            if ¬isEmpty(e.channelNo.recvq) then
                return e
            end
        else if e.baseEventType == RECV then
            if ¬isEmpty(e.channelNo.sendq) then
                return e
            end
        else return ∅;                                  /* Impossible case */
    end
end
return ∅ ;                              /* No synchronisable event found */
```

When the *findSynchronisableEvent* function is unable to find any threads ready for synchronisation it blocks all the base-events that make up an event. Algorithm 3 shows the *block* function, which does the blocking.

Algorithm 3 The block function

```
Data: eventList
foreach e ∈ eventList do
    if e.baseEventType == SEND then
        e.channelNo.sendq.enqueue(currentThread);
    else if e.baseEventType == RECV then
        e.channelNo.recvq.enqueue(currentThread);
    else Do nothing;                                    /* Impossible case */
end
```

Once the *block* function registers the current thread-id in the respective channels associated with the base events, the next operation is dispatching a new thread. The *dispatchNewThread* function does that by dequeueing off an important data structure of the SenseVM - the *ready queue* or `readyQ`. It is a priority queue containing thread-ids, arranged in the ascending order of their respective deadlines. So the thread with the most urgent deadline will be the first element on the `readyQ`. Threads with no deadline or timing requirements are arranged in a FIFO order. When the `readyQ` is empty, the VM relinquishes the control back to the underlying OS, which can then choose to run on a low-power mode as all threads are blocked.

When two threads are communicating, the first one to be scheduled will be blocked as it did not find any corresponding thread for communication. However, when *dispatchNewThread* dispatches the second thread, the *findSynchronisableEvent* function will return

Algorithm 4 The dispatchNewThread function

Data: *eventList*
if $readyQ \neq \varnothing$ **then**
  $threadId \leftarrow dequeue(readyQ)$;
  $currentThread = threadId$;
else
  relinquish control to the underlying OS
end

a synchronisable event and the *syncNow* operation does the actual message passing. We describe the algorithm of *syncNow* in Algorithm 5 below.

This completes our overview of the `sync` operation. We next discuss how we handle the timing aspect of SenseVM.

6.4 Timed synchronisation of events

The timed-synchronisation operator, `syncT`, handles the timing aspects of SenseVM. As discussed in Section 6.1.1, `syncT` is compiled down to two bytecodes. The first bytecode operation, referred to as the `TIME` bytecode, reads the relative baseline and deadline values and accordingly schedules the thread's execution.

An important data structure for interpreting the `TIME` bytecode is the wait queue or `waitQ`. Just like the `readyQ`, this is another priority queue that holds thread-ids but unlike `readyQ`, it is arranged by the ascending order of baseline. So the thread that is supposed to begin the earliest will be at the head of the `waitQ`.

Another important runtime function is the `setAlarm` function built upon the SenseVM timing subsystem (Section 6.7). As the name implies, this operation allows setting an alarm for the $T_{absolute}$ value at which we want to wake a thread. An alarm interrupt arrives at that time and the thread gets woken up. We can now present Algorithm 6 that illustrates how the SenseVM runtime handles the `TIME` bytecode. An important heuristic used in our algorithm is the `SET_ALARM_AFTER` variable that we currently set at a value of 30000 μseconds. When the alarm setting time is less than this value, the overheads of bytecode interpretation and garbage collection coupled with setting an alarm and receiving an interrupt might become too expensive. So when the thread wake-up time is less than `SET_ALARM_AFTER` time-units away, we straightaway move the thread to the `readyQ` to begin its execution.

A notable aspect of Algorithm 6 is the usage of the T_{local} clock to set the wakeup time (T_{wakeup}). This avoids the jitter introduced by executing other operations. When the wall-clock time moves to T_{wakeup} timeunit, a timer interrupt arrives and we handle that via the low-level bridge (Section 6.6). The interpreter checks after executing each bytecode if there are any new timer interrupt messages present in the low-level bridge. For all such messages SenseVM uses the `handleTimerMsg` function.

Algorithm 7 describes the `handleTimerMsg` function. This algorithm primarily involves updating the T_{local} clock of the thread for which the interrupt arrives. However, we apply an optimisation, where a check is made to see if the next thread ($timedThread_2$) in the `waitQ` has a baseline that is near execution. If that is the case, the next thread ($timedThread_2$) is also scheduled for execution and all the concerned T_{local} clocks are adjusted.

Additionally, before scheduling the thread, $timedThread$, we check if the deadline of the currently running thread is approaching sooner than the deadline for $timedThread$. In such

Algorithm 5 The syncNow function

```
Data: A base-event value - event
if ¬hardware(event) then
    if event.baseEventType == SEND then
        threadIdR ← dequeue(event.channelNo.recvq);
        recvEvt ← threadIdR.envRegister;
        event.Message → threadIdR.envRegister ;
        threadIdR.programCounter → recvEvt.wrapFunc ;
        currentThread.programCounter → event.wrapFunc;
        sendingThread = currentThread;
        currentThread = threadIdR;
        readyQ.enqueue(sendingThread);
    else if event.baseEventType == RECV then
        threadIdS ← dequeue(event.channelNo.sendq);
        sendEvt ← threadIdS.envRegister;
        sendEvent.Message → currentThread.envRegister ;
        threadIdS.programCounter → sendEvt.wrapFunc ;
        currentThread.programCounter → event.wrapFunc;
        readyQ.enqueue(threadIdS);
    else Do nothing;                                   /* Impossible case */
else if hardware(event) then
    if event.baseEventType == SEND then
        llBridge.write(event.Message) → driver ;
    else if event.baseEventType == RECV then
        currentThread.envRegister ← llBridge.read(driver) ;
    else Do nothing;                                   /* Impossible case */
    currentThread.programCounter → event.wrapFunc;
else Do nothing;                                       /* Impossible case */
```

a case we enqueue *timedThread* in the `readyQ` as we want to run the program with the earliest deadline first.

In the next section, we will briefly discuss the SenseVM scheduler.

6.5 The scheduler

SenseVM's scheduler is a hyrbid of cooperative and preemptive scheduling. When programming applications that do not use `syncT` the scheduler is cooperative in nature. Initially the threads are scheduled in the order that the main method calls them. For eg:

```
main =
  let _ = spawn thread1 in
  let _ = spawn thread2 in
  let _ = spawn thread3 in
  ...
```

The above would schedule the threads in the order of `thread1`, followed by `thread2` and finally `thread3`. As the program proceeds, the scheduler relies on the threads to yield control back to the scheduler so that it can run the next thread. The synchronisation algorithm (Algorithm 1) shows when the scheduler is unable to find a matching thread for the currently

Algorithm 6 The time function

Data: Relative Baseline = $baseline$, Relative Deadline = $deadline$
$T_{wakeup} = currentThread.T_{local} + baseline$;
if $deadline == 0$ **then**
  $T_{finish} = Integer.MAX$;
else
  $T_{finish} = T_{wakeup} + deadline$;
end
$currentThread.deadline = T_{finish}$;
$baseline_{absolute} = T_{absolute} + baseline$;
$deadline_{absolute} = T_{absolute} + baseline + deadline$;
$cond1 = T_{absolute} > deadline_{absolute}$;
$cond2 = (T_{absolute} \geq baseline_{absolute})\&\&(T_{absolute} \leq deadline_{absolute})$;
$cond3 = baseline < SET\_ALARM\_AFTER$;
if $baseline == 0 \vee cond1 \vee cond2 \vee cond3$ **then**
  $readyQueue.enqueue(currentThread)$;
  $dispatchNewThread()$;
end
$setAlarm(T_{wakeup})$;
$waitQ.enqueue(currentThread).orderBy(T_{wakeup})$;
$dispatchNewThread()$;

running thread that is ready to synchronise the communication, it blocks the current thread and calls the *dispatchNewThread*() function to run other threads. On the other hand, when synchronisation succeeds, the scheduler puts the message-sending thread to the `readyQ` and the message-receiving thread starts running.

The preemptive behaviour of the scheduler occurs when using `syncT`. For instance, when a particular *untimed* thread is running and the baseline time of a timed thread has arrived, the scheduler then preempts the execution of the *untimed* thread and starts running the timed thread. A similar policy is also observed when the executing thread's deadline is later than a more urgent thread, the thread with the earliest deadline is chosen to be run at that instance. Algorithm 7 shows the preemptive components of the scheduler.

The SenseVM scheduler also handles hardware driver interactions via message-passing. The structure that is used for messaging is shown below:

Listing 14 A SenseVM hardware message

```
typedef struct {
  uint32_t sender_id;
  uint32_t msg_type;
  uint32_t data;
  Time timestamp;
} svm_msg_t;
```

The `svm_msg_t` type contains a unique sender id for each driver that is the same as the number used in `spawnExternal` to identify that driver. The 32 bit `msg_type` field can be used to specify different meanings for the next field, the `data`. The `data` is a 32 bit word. The `timestamp` field of a message struct is a 64 bit entity, explained in detail in Section 6.7.

When the SenseVM scheduler has all threads blocked, it uses a function pointer called `blockMsg`, which is passed to it by the OS that starts the scheduler, to wait for any interrupts from the underlying OS (more details in Section 6.6). Upon receiving an interrupt, the

Algorithm 7 The handleTimerMsg function

Data: Wakeup Interrupt
$timedThread \leftarrow dequeue(waitQ)$;
$T_{now} = timedThread.baseline$;
$timedThread.T_{local} = T_{now}$;
if $waitQ \neq \varnothing$ **then**
  $timedThread_2 \leftarrow peek(waitQ)$; `/* Does not dequeue */`
  $cond1 = T_{absolute} > timedThread_2.deadline$;
  $cond2 = (T_{absolute} \geq timedThread_2.baseline)\&\&(T_{absolute} \leq timedThread_2.deadline)$;
  $cond3 = (T_{absolute} - timedThread_2.baseline) < SET\_ALARM\_AFTER$;
  if $cond1 \vee cond2 \vee cond3$ **then**
    $timedThread_2 \leftarrow dequeue(waitQ)$; `/* Actually dequeue the thread */`
    $readyQ.enqueue(timedThread_2)$;
    $T_{now} = timedThread2.baseline$;
    $timedThread.T_{local} = T_{now}$;
    $timedThread_2.T_{local} = T_{now}$;
  else
    $setAlarm(timedThread_2.baseline)$;
  end
end
if $currentThread == \varnothing$ **then**
  $currentThread = timedThread$;
  $currentThread.deadline = timedThread.deadline$;
else
  if $timedThread.deadline < currentThread.deadline$ **then** `// earlier deadline`
    $currentThread.baseline = T_{now}$;
    $readyQ.enqueue(currentThread)$;
    $currentThread = timedThread$;
    $currentThread.deadline = timedThread.deadline$;
  else
    $readyQ.enqueue(timedThread)$;
    $currentThread.logicalTime = T_{now}$; `/* Avoids too much time drift */`
  end
end

scheduler uses the SenseVM runtime's `handleMsg` function to handle the corresponding message. The function internally takes the message and unblocks the thread for which the message was sent. We show the general structure of SenseVM's scheduler in Algorithm 8.

Algorithm 8 The SenseVM scheduler

Data: $blockMsg$ function pointer
$\forall threads\ set\ T_{local} = T_{absolute}$;
$svm\_msg\_t\ msg$;
while *True* **do**
  if *all threads blocked* **then**
    $blockMsg(\&msg)$; `/* Relinquish control to OS */`
    $handleMsg(msg)$;
  else
    $interpret(currentThread.PC)$;
  end
end

Notable above is the initialisation of the T_{local} clock for each thread at the beginning of the scheduler. Also notable is the `blockMsg` function that relinquishes control to the underlying OS, allowing it to save power. When the interrupt arrives, the `handleMsg` function unblocks certain thread(s) so that when the $if..then$ clause ends, in the following iteration the *else* clause is executed and bytecode interpretation continues. We next discuss the low-level bridge connecting SenseVM to the underlying OS.

6.6 The Low-Level Bridge

The low-level bridge specifies a set of two interfaces that should be implemented when writing peripheral drivers to use with SenseVM. The first interface contains functions for reading and writing data synchronously to and from a driver. The other interface is geared towards interrupt-based drivers that asynchronously produce data.

The C-`struct` below contains the interface functions for reading and writing data to a driver as well as functions for checking of the availability of data.

```
typedef struct ll_driver_s{
  void *driver_info;
  bool is_synchronous;
  uint32_t (*ll_read_fun)(struct ll_driver_s *this, uint8_t*, uint32_t);
  uint32_t (*ll_write_fun)(struct ll_driver_s *this, uint8_t*, uint32_t);
  uint32_t (*ll_data_readable_fun)(struct ll_driver_s* this);
  uint32_t (*ll_data_writeable_fun)(struct ll_driver_s* this);

  UUID channel_id;
} ll_driver_t;
```

The `driver_info` field in the `ll_driver_t` struct can be used by a driver that implements the interface to keep a pointer to lower-level driver specific data. For interrupt-based drivers, this data will contain, among other things, an *OS interoperation* struct. These OS interoperation structs are explained further down. A boolean indicates if the driver is synchronous or not. Next the struct contains function pointers to the low-level driver's implementation of the interface. Lastly, a `channel_id` identifying the channel along which the driver is allowed to communicate with processes running on top of SenseVM.

The `ll_driver_t` struct contains all the data associated with a drivers configuration in

one place and allows for the definition of a set of platform and driver independent functions to be defined for use in the runtime system, shown below:

```
uint32_t ll_read(ll_driver_t *drv, uint8_t *data, uint32_t data_size);
uint32_t ll_write(ll_driver_t *drv, uint8_t *data, uint32_t data_size);
uint32_t ll_data_readable(ll_driver_t *drv);
uint32_t ll_data_writeable(ll_driver_t *drv);
```

The OS interoperation structs mentioned above, are essential for drivers that asynchronously produce data and are shown below, both the Zephyr and the ChibiOS versions.

```
typedef struct zephyr_interop_s {
  struct k_msgq *msgq;
  int (*send_message)(struct zephyr_interop_s* this, svm_msg_t msg);
} zephyr_interop_t;
```

```
typedef struct chibios_interop_s {
  memory_pool_t *msg_pool;
  mailbox_t *mb;
  int (*send_message)(struct chibios_interop_s* this, svm_msg_t msg);
} chibios_interop_t;
```

In both cases, the struct contains the data that functions need to set up low-level message-passing between the driver and the OS thread running the SenseVM runtime system. Zephyr provides a message-queue abstraction that can take fixed-size messages, while ChibiOS supports a mailbox abstraction that receives messages that are the size of a pointer. Since ChibiOS mailboxes cannot receive data that is larger than a 32-bit word, a memory pool of messages is employed in that case. The structure used to send messages from the drivers is the already introduced `svm_msg_t` struct, described in Listing 14.

Another important class of interrupts that the scheduler needs to handle are alarms from the wall-clock time subsystem, which arise from the `syncT` operation. The next section discusses that component of SenseVM.

6.7 The wall-clock time subsystem

Programs running on SenseVM that make use of the timed operations rely on there being a monotonically increasing timer. The wall-clock time subsystem emulates this by implementing a 64bit timer that would take almost 7000 years to overflow at 84MHz frequency or about 36000 years at 16MHz. The timer frequency of 16MHz is used on the NRF52 board, while the timer runs at 84MHz on the STM32.

SenseVM requires the implementation of the following functions for each of the platforms (such as ChibiOS and Zephyr) that it runs on :

```
bool      sys_time_init(void *os_interop);
Time      sys_time_get_current_ticks(void);
uint32_t  sys_time_get_alarm_channels(void);
uint32_t  sys_time_get_clock_freq(void);
bool      sys_time_set_wake_up(Time absolute);
Time      sys_time_get_wake_up_time(void);
bool      sys_time_is_alarm_set(void);
```

The timing subsystem uses the same OS interoperation structs as drivers do and thus have access to a communication channel to the SenseVM scheduler. The interoperation is provided to the subsystem at initialisation using `sys_time_init`.

The key functionality implemented by the timing subsystem is the ability to set an alarm at an absolute 64-bit point in time. Setting an alarm is done using `sys_time_set_wake_up`. The runtime system can also query the timing subsystem to check if an alarm is set and at what specific time.

The low-level implementation of the timing subsystem is highly platform dependent at the moment. But on both Zephyr and ChibiOS the implementation is currently based on a single 32-bit counter configured to issue interrupts at overflow, where an additional 32-bit value is incremented. Alarms can only be set on the lower 32-bit counter at absolute 32-bit values. Additional logic is needed to translate between the 64-bit alarms set by SenseVM and the 32-bit timers of the target platforms. Each time the overflow interrupt happens, the interrupt service routine checks if there is an alarm in the next 32-bit window of time and in that case, enables a compare interrupt to handle that alarm. When the alarm interrupt happens, a message is sent to the SenseVM scheduler in the same way as for interrupt based drivers, using the message queue or mailbox from the OS interoperation structure.

6.8 Comparison with Asynchronous Message-Passing

SenseVM chooses a synchronous message-passing model in contrast with actor-based systems like Erlang that support an asynchronous message-passing model with each process containing a mailbox. We believe that a synchronous message-passing policy is better suited for embedded systems for the following reasons:

1. Embedded systems are highly memory-constrained and asynchronous send semantics assume the *unboundedness* of an actor's mailbox, which is a poor assumption in the presence of memory constraints. Bounding the size of a mailbox results that when the mailbox gets filled, message-sending becomes blocking, which is already the default semantics of SenseVM.
2. Acknowledgement is implicit in synchronous message-passing systems, in contrast to explicit message acknowledgement in asynchronous systems that leads to code bloat. Additionally, if a programmer forgets to remove acknowledgement messages from an actor's mailbox, it leads to memory leaks.
3. Actor-based embedded systems runtimes like Medusa [2] incur an extra cost of tagging messages to identify which message should be routed to which actor. SenseVM's dedicated channel-per-driver API (via `spawnExternal`) avoids this extra cost.

7 Evaluation

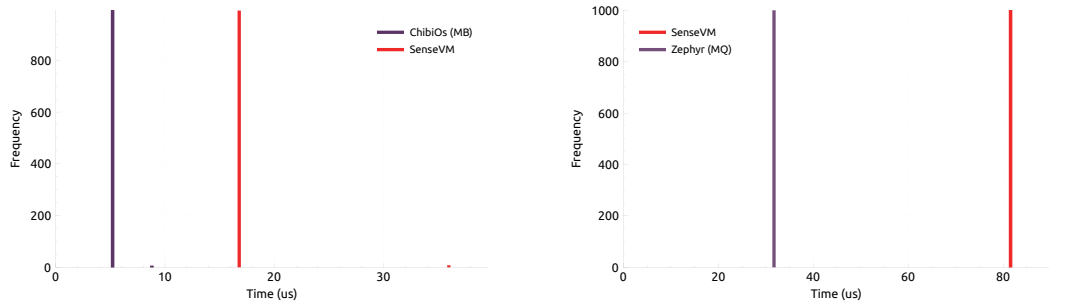
7.1 Interpretive overhead measurements

To characterize the overhead of executing programs on top of SenseVM compared to running them directly on either Zephyr or ChibiOS, we implement *button-blinky* directly on top of these operating systems and measure the response-time differences.

The button-blinky program copies the state of a button onto an LED, something that could be done very rapidly at a large CPU utilization cost by continuously polling the button state and writing it to the LED. Instead, the Zephyr and ChibiOS implementations are interrupt-based and upon receiving a button interrupt (for either button being pressed down or released), send a message to a thread that is kept blocking until such messages arrive. When the thread receives a message indicating a button down or up, it sets the LED to on or off. This approach keeps the low-level implementation in Zephyr and ChibiOS similar to SenseVM and indicates the interpretive and other overheads in SenseVM.

The data in charts presented here are collected using an STM32F4 microcontroller based testing system connected to either the NRF52 or the STM32F4 system under test (SUT). The testing system provides the stimuli, setting the GPIO (button) to either active or inactive and measures the time it takes for the SUT to respond on another GPIO pin (symbolising

the LED). The testing system connects to a computer displaying a GUI and generates the plots used in this paper. The plots are created by placing all response times measured into buckets of similar time and plots the number of samples falling in a bucket as a vertical bar. Each button is labelled with the average time of the samples it contains.



(a) Response time comparison between a C-code implementation using ChibiOS against the same program on SenseVM (running on ChibiOS). Data obtained on the STM32F4 microcontroller.

(b) Response time comparison between a C-code implementation using the Zephyr OS against the same program on SenseVM (running on Zephyr). Data obtained on the NRF52 microcontroller.

Figure 13 Button-blinky response times comparison between C and SenseVM

Figure 13a shows the SenseVM response time in comparison to the implementation of the program running on ChibiOS using its mailbox abstraction (MB). There the overhead is about 3x. Figure 13b, the chart on the right shows the SenseVM response time in comparison to the Zephyr message queue based implementation (MQ) and shows an overhead of 2.6x.

In the measurements relative to ChibiOS (Figure 13a), there are outliers both when running on ChibiOS directly and when running SenseVM on top of ChibiOS. In the ChibiOS (MB) data there are 6 outliers where a response takes 1.6 times longer than an average non-outlier response. For SenseVM, 8 outliers take 2.1 times longer than an average non-outlier SenseVM response.

Outliers are expected in the SenseVM data series due to the garbage collection. The outliers apparent in the plain ChibiOS code has not been pinpointed. Outliers, when running on SenseVM, are studied in more detail below.

7.2 Effects of Garbage Collection

This experiment measures the effects of garbage collection on response time by repeatedly running the 1000 samples test for different heap-size configurations of SenseVM. A smaller heap should lead to much more frequent interactions with the garbage collector and the effects of the garbage collector on the response time becomes magnified.

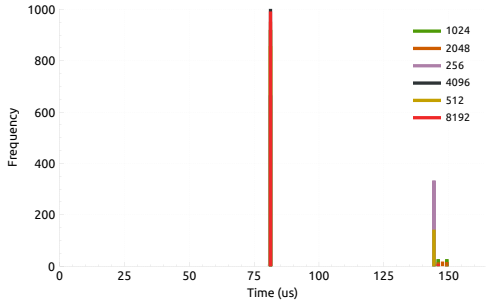
As a smaller heap is used the number of outliers should increase if the outliers are due to garbage collection. The following table shows the number of outliers at each size configuration for the heap used, and there seems to be an indication that GC is the cause of outliers.

Heap size	256	512	1024	2048	4096	8192
Outliers NRF52 on Zephyr	333	142	80	48	0	7
Outliers STM32 on ChibiOs	346	156	99	67	18	9

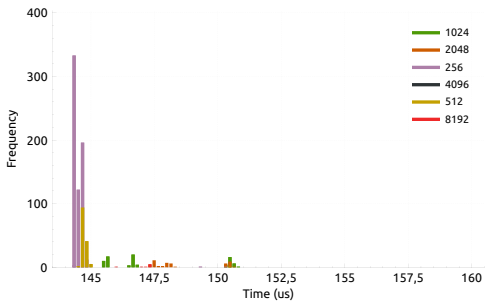
From the figures 14a and 14c, we can see that the GC involvement in the measurements

can make them take roughly twice the amount of time. Note that in the chart, the plots overlap and all non-outliers are overlaying each other in the single tall red bar.

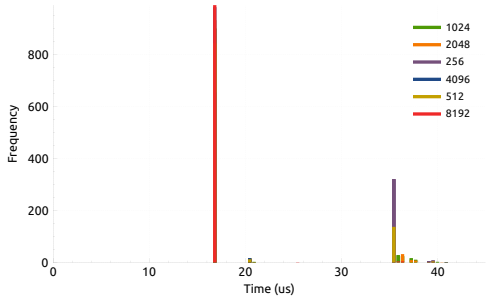
Zooming in on the outliers shows their internal relationships and spread. See figures 14b and 14d. Here it is clear that there are more outliers as the heap is made smaller and maybe a vague indication that garbage collection in a small heap is cheaper than in a larger one.



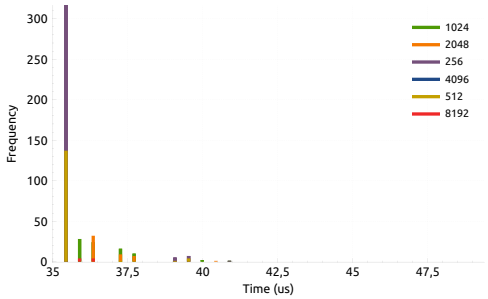
(a) Response time measurements at different sizes of the heap (8192, 4096, 2048, 1024, 512 and 256 Bytes) to identify effects of garbage collection. This data is collected on the NRF52 microcontroller running SenseVM on top of the Zephyr OS



(b) Zooming in on the outliers when measuring response time at different size of the heap. This data is collected on the NRF52 microcontroller running SenseVM on top of the Zephyr OS



(c) Response time measurements at different sizes of the heap to identify effects of garbage collection. This data is collected on the STM32F4 microcontroller running SenseVM on top of ChibiOS



(d) Zooming in on the outliers when measuring response time at different size of the heap. This data is collected on the STM32F4 microcontroller running SenseVM on top of the ChibiOS

Figure 14 Effects of garbage collection on the response times for various heap sizes

7.3 Precision and jitter

This section evaluates how the timing subsystem works in relation to jitter. Jitter can be defined as the deviation from true periodicity of a presumably periodic signal, often in relation to a reference clock signal. We want to evaluate how our claims of `syncT` reducing jitter pans out in practice.

The program below is written in a naive way to illustrate how jitter manifests in programs. Figure 15b shows what the oscilloscope draws, set to persistent mode drawing while sampling the signal the Raspberry Pi outputs.

```
while (1) {

   uint32_t state = GPIO_READ(23);
   if (state) {
     GPIO_CLR(23);
   } else {
     GPIO_SET(23);
   }

   usleep(400);
 }
```

The Raspberry Pi program reads the status of a GPIO pin and then inverts its state back to that same pin. The program then goes to sleep using `usleep` for 400us. The goal frequency was 1kHz and sleeping for 400us here gave a roughly 1.05kHz signal. The more expected sleep time of 500us to generate a 1kHz signal lead, instead, to a much lower frequency. So, the 400us value was found experimentally. Figure 15a shows the oscilloscope measurement outputs of the roughly 1kHz wave.

The SenseVM program for a 1kHz frequency generator is shown below. Note that in this case specifying a baseline of 500us, actually leads to a 1kHz wave (compared to the 400us used above that together with additional delays of the system give a roughly 1kHz wave).

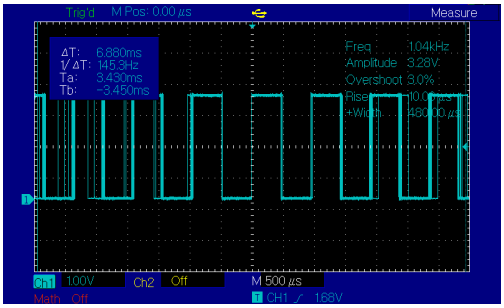
```
not : Int -> Int
not 1 = 0
not 0 = 1

ledchan : Channel Int
ledchan = channel ()

foo : Int -> ()
foo val =
 let _ = syncT 500 0 (send ledchan val) in
 foo (not val)

main =
  let _ = spawnExternal ledchan 1 in
  foo 1
```

Figure 15d shows the persistent display of the 1kHz wave generated by SenseVM and Figure 15c shows the oscilloscope measurement outputs.

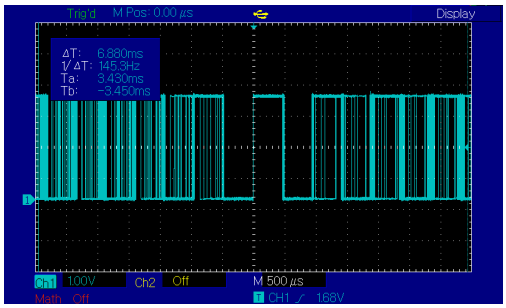
8 Limitations and Future Work

We have already discussed some of the subsets of embedded systems applications that are not currently addressable by SenseVM in Section 1.2. In this section, we look at the specific components of SenseVM that can be improved and propose future work to improve them.

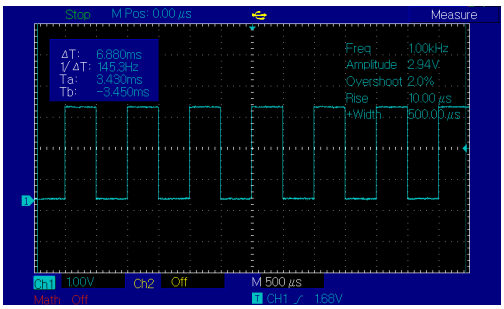
- Memory managememt. A primary area of improvement is upgrading our stop-the-world mark and sweep garbage collector and investigating real-time garbage collectors like Schism [23]. Another relevant future work would be investigating static memory-management schemes like regions [30] and techniques combining regions with GC [13].
- Interpretation overhead. A possible approach to reducing our interpretation overhead could be pre-compiling our bytecode to machine code (AOT compilation). Similarly, dynamic optimization approaches like JITing could be an area of investigation.
- Deadline miss API. Currently SenseVM doesn't have an API to represent actions that should happen if a task were to miss its deadline. We envision adapting the negative acknowledgement API of CML to represent missed-deadline handlers for SenseVM.



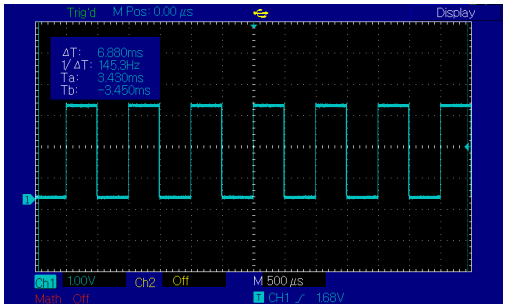
(a) A roughly 1kHz square wave generated using an Raspberry Pi 3 model B+ running Linux



(b) Illustrating the amount of jitter on the square wave generated from the Raspberry Pi by setting the oscilloscope display in persistent mode.



(c) A 1kHz square wave generated using SenseVM running on the STM32F4



(d) Leaving the Oscilloscope display in persistent mode shows a much more stable picture for the SenseVM implementation

Figure 15 Evaluations on the 1 KHz frequency generator

- Priority inversions. Although TinyTimber-style dynamic priorities might reduce priority inversion occurrences, they can still occur on the SenseVM. Advanced approaches like priority inheritance protocols [27] need to be experimented with on the SenseVM scheduler.
- Static analysis and error message. The programs presented in our examples, by convention, take care to call the dynamic `spawn` function solely from the main method. We envision static analysis passes that report warnings when a programmer writes too dynamic programs, which could cause unrestricted memory growth. A similar line of work would be improving the general state of error messages and warnings on our frontend language.

9 Related Work

Among functional languages, there exists a number of virtual machines designed for micro-controllers such as OMicroB [33] supporting OCaml, Picobit [29] supporting Scheme and AtomVM [4] supporting Erlang. SenseVM differs from these projects in the respect that we identify certain fundamental characteristics of embedded systems and accordingly design a runtime to address those demands. Our frontend language's APIs arise from the requirements enforced by the VM in contrast with general-purpose languages like Scheme.

The Medusa [2] language and runtime is the inspiration behind SenseVM's uniform framework of concurrency and I/O. Medusa, however, does not provide any timing based APIs, and their message-passing framework is based on the actor model. We have already

compared the actor model's design choices with ours in Section 6.8

There exist virtual machines like the VeloxVM [32] that enable specifying security policies for safe execution of programs. The security of embedded systems remains a prospective research area where we envision using information-flow control policies to monitor the messages passed between various processes, drawing inspiration from OSes like HiStar [37].

In the real-time space, a safety-critical VM that can provide hard real-time guarantees on Real-Time Java programs is the FijiVM [24] implementation. A critical innovation of the project was the Schism real-time garbage collector [23] from which we hope to draw inspiration for future work on memory management.

RTMLton [28] is another example of a real-time project supporting a general-purpose language like SML. RTMLton adapts the MLton runtime [35] with ideas from FijiVM to enable handling real-time constraints in SML. CML is available as an SML library, so RTMLton provides access to the event framework of CML but lacks the uniform concurrency-I/O model and the `syncT` operator of SenseVM.

The Timber language [5] is another functional/object-oriented language that inspired the `syncT` API of SenseVM. The TinyTimber kernel that we have discussed so far is the runtime of the Timber language. Timber was designed for hard real-time scenarios and their theoretical work on estimating the heap space bounds for real-time programs [17] is a prospective area of future research.

The WebAssembly project (WASM) has spawned sub-projects like WebAssembly Micro Runtime (WAMR) [1] that allows running languages that compile to WASM to run on microcontrollers. Notable here is that while several general-purpose languages like JavaScript can execute on ARM architectures by compiling to WebAssembly, they lack the native support for the concurrent, I/O-bound and timing-aware programs that is naturally provided by SenseVM. As such reactive extensions of Javascript, like HipHop.js [3], are being envisioned to be used for embedded systems.

Another related line of work is embedding domain-specific languages like Ivory [9] and Copilot [22] in Haskell to generate C programs that can run on embedded devices. This approach differs from ours in that two separate languages dictate the programming model when using an EDSL, the first being the DSL itself and the second being the host language (Haskell). We assess that having a single language (like on SenseVM) provides a more uniform programming model to the programmer but on the other hand, EDSLs have very little runtime overheads, and when properly optimised, can run as fast as C.

10 Conclusion

In this paper, we have presented SenseVM, a virtual machine targeted towards embedded systems programming. We identified three essential characteristics of embedded applications, namely being concurrent, I/O-bound and timing-aware, and correspondingly designed our VM to address all three concerns. Our evaluations, conducted on the STM32 and NRF52 microcontrollers, suggest that SenseVM's response time is within the range of 2-3x times that of native C programs, which is an encouraging result and allows us to envision research avenues to further improve our numbers through smarter garbage collection strategies and AOT compilation optimisations. We also demonstrated the ease of expressing state machines, common in embedded systems, through our examples. Finally, our timing API was demonstrated by expressing a soft real-time application, and we expect further theoretical investigations on the worst-case execution time and schedulability analysis on SenseVM.

References

1. Wamr - webassembly micro runtime, 2019. URL: https://github.com/bytecodealliance/wasm-micro-runtime.
2. Thomas W. Barr and Scott Rixner. Medusa: Managing concurrency and communication in embedded systems. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 439–450. USENIX Association, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/barr.
3. Gérard Berry and Manuel Serrano. Hiphop.js: (a)synchronous reactive web programming. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 533–545. ACM, 2020. doi:10.1145/3385412.3385984.
4. Davide Bettio. Atomvm, 2017. URL: https://github.com/bettio/AtomVM.
5. Andrew P Black, Magnus Carlsson, Mark P Jones, Richard Kieburtz, and Johan Nordlander. Timber: A programming language for real-time embedded systems. Technical report, OGI School of Science and Engineering, Oregon Health and Sciences University, Technical Report CSE 02-002. April 2002, 2002.
6. Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 1985. doi:10.1007/3-540-15975-4_29.
7. Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In Andrew T. Campbell, Philippe Bonnet, and John S. Heidemann, editors, *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006, Boulder, Colorado, USA, October 31 - November 3, 2006*, pages 29–42. ACM, 2006. doi:10.1145/1182807.1182811.
8. Stephen A Edwards and John Hui. The sparse synchronous model. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–8. IEEE, 2020.
9. Trevor Elliott, Lee Pike, Simon Winwood, Patrick C. Hickey, James Bielman, Jamey Sharp, Eric L. Seidel, and John Launchbury. Guilt free ivory. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 189–200. ACM, 2015. doi:10.1145/2804302.2804318.
10. Zephyr examples. Zephyr button blinky, 2021. URL: https://pastecode.io/s/szpf673u.
11. The Linux Foundation. Zephyr RTOS. https://www.zephyrproject.org/. Accessed 2021-11-28.
12. Damien George. Micropython, 2014. URL: https://micropython.org/.
13. Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 141–152. ACM, 2002. doi:10.1145/512529.512547.
14. Ralf Hinze. The categorical abstract machine: Basics and enhancements. Technical report, University of Bonn, 1993.
15. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.
16. R John M Hughes. A semi-incremental garbage collection algorithm. *Software: Practice and Experience*, 12(11):1081–1082, 1982.
17. Martin Kero, Pawel Pietrzak, and Johan Nordlander. Live heap space bounds for real-time systems. In Kazunori Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2010. doi:10.1007/978-3-642-17164-2_20.

18 Donald Ervin Knuth. *The art of computer programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998. URL: https://www.worldcat.org/oclc/312994415.

19 Per Lindgren, Johan Eriksson, Simon Aittamaa, and Johan Nordlander. Tinytimber, reactive objects in c for real-time embedded systems. In *2008 Design, Automation and Test in Europe*, pages 1382–1385, 2008. doi:10.1109/DATE.2008.4484933.

20 Tommi Mikkonen and Antero Taivalsaari. Web applications - spaghetti code for the 21st century. In Walter Dosch, Roger Y. Lee, Petr Tuma, and Thierry Coupaye, editors, *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, 20-22 August 2008, Prague, Czech Republic*, pages 319–328. IEEE Computer Society, 2008. doi:10.1109/SERA.2008.16.

21 Johan Nordlander. *Programming with the TinyTimber kernel*. Luleå tekniska universitet, 2007.

22 Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2010. doi:10.1007/978-3-642-16612-9_26.

23 Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 146–159. ACM, 2010. doi:10.1145/1806596.1806615.

24 Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real time java on resource-constrained platforms with fiji VM. In M. Teresa Higuera-Toledano and Martin Schoeberl, editors, *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009, Madrid, Spain, September 23-25, 2009*, ACM International Conference Proceeding Series, pages 110–119. ACM, 2009. doi:10.1145/1620405.1620421.

25 John H. Reppy. Concurrent ML: design, application and semantics. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer, 1993. doi:10.1007/3-540-56883-2_10.

26 Herbert Schorr and William M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967. doi:10.1145/363534.363554.

27 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

28 Bhargav Shivkumar, Jeffrey C. Murphy, and Lukasz Ziarek. Rtmlton: An SML runtime for real-time systems. In Ekaterina Komendantskaya and Yanhong Annie Liu, editors, *Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20-21, 2020, Proceedings*, volume 12007 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2020. doi:10.1007/978-3-030-39197-3_8.

29 Vincent St-Amour and Marc Feeley. PICOBIT: A compact scheme system for microcontrollers. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages - 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers*, volume 6041 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009. doi:10.1007/978-3-642-16478-1_1.

30 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997. doi:10.1006/inco.1996.2613.

31 Hideyuki Tokuda, Clifford W. Mercer, Yutaka Ishikawa, and Thomas E. Marchok. Priority inversions in real-time communication. In *Proceedings of the Real-Time Systems Symposium -*

1989, Santa Monica, California, USA, December 1989, pages 348–359. IEEE Computer Society, 1989. doi:10.1109/REAL.1989.63587.

32 Nicolas Tsiftes and Thiemo Voigt. Velox VM: A safe execution environment for resource-constrained iot applications. *J. Netw. Comput. Appl.*, 118:61–73, 2018. doi:10.1016/j.jnca.2018.06.001.

33 Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. A generic virtual machine approach for programming microcontrollers: the omicrob project. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, 2018.

34 Ge Wang and Perry R. Cook. Chuck: A concurrent, on-the-fly, audio programming language. In *Proceedings of the 2003 International Computer Music Conference, ICMC 2003, Singapore, September 29 - October 4, 2003*. Michigan Publishing, 2003. URL: http://hdl.handle.net/2027/spo.bbp2372.2003.055.

35 Stephen Weeks. Whole-program compilation in mlton. In Andrew Kennedy and François Pottier, editors, *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, page 1. ACM, 2006. doi:10.1145/1159876.1159877.

36 Gordon Williams. Espruino, 2012. URL: http://www.espruino.com/.

37 Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 263–278. USENIX Association, 2006. URL: http://www.usenix.org/events/osdi06/tech/zeldovich.html.