Release Notes: Rust Guaranteed Software Safety in HarSaRK Kernels

May 2025

1 Introduction

One of the important features of HarSaRK is that it enforces various software safety parameters with the help of kernel architecture and Rust language features without any runtime overhead. It is all achieved by static analysis at compile time. The different safety parameters and the bugs they address are :

- Memory safety: Buffer overflow, Use after free, Double free, Null pointer dereference, etc.
- Type safety: Unsafe typecasts, ensure coherence between address and data-type.
- Concurrency safety: Data races, Deadlocks, and Priority inversion.

The following subsections describe the various language constructs provided by Rust and how it helps HarSaRK guarantee various aspects of software safety.

1.1 Type Safety using Generics and Traits

Generics[1] and Traits[2] is the primary mechanism to achieve polymorphism and code-reuse in Rust. Rust's ownership system allows expressing and enforcing typestates at compile-time, which is beneficial in safety-critical applications [3].

1.1.1 Generics

Generics are used to create definitions for items like function signatures or structs, which can then be used with many different concrete data types [1]. Listing shows an example of the application of generics.

Kernel primitives such as the Messaging and Resource primitives enclose values in them. Their definition needs to be independent of the type in it. However, because C does not support generics, kernels written in C tend to use void pointers and typecasts to support type independence. However, this is a highly unsafe approach and can lead to bugs that cannot be traced easily;

Listing 1: Generics example

```
struct Sample<T> {
    field1: T,
    field2: T,
}

fn generic_function<T> (a: T, b: T) {
    ...
}

let inst1 = Sample {field1: 1, field2: 5}
let inst2 = Sample {field1: "hello" , field2: "all"};
```

pointer typecasts are hard to reason about and blocks static analysis tools from analyzing the code. For instance, the C compiler allows one to cast a *char** to *int** without throwing a compilation error. These bugs question the reliability of the system. Generally, it is not considered a good practice to work with raw pointers and perform manipulations on them [4].

The Message and Resource primitive of HarSaRK are designed using generics. When a Message variable is declared using HarSaRK, it is bound to a type at compile-time based on the declaration of the instance. So the compiler can guarantee at compile-time that when a message is received from a message instance, it is of that defined type. This allows the compiler to validate that the operations performed on the received value across the codebase at compile time.

This only implies that the type of the message received is determined at compile-time, not the size. For dynamically sized types, these variables exist behind a pointer to a memory location on the heap, and access to the pointer is safely handled by Message and Resource primitive.

1.1.2 Traits

A trait tells the Rust compiler about the functionality that a particular type will provide [2]. Generic functions can exploit this to specify bounds on the types they accept. For example, in the Listing below, *print_area()* is defined using generics. However, because of the bound on the generic T to implement HasArea trait, the compiler can verify that the argument *shape* has the *area()* method defined.

Listing 2: Traits and Trait bounds example

```
trait HasArea {
    fn area(&self) -> f64;
}
fn print_area <T: HasArea> (shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Traits help extend the polymorphism of types in Rust to a great extent without compromise on safety. The Message and Resource primitive can also be instantiated with trait objects [5] instead of a concrete type for supporting more freedom in its usage. So, now the Message primitive can be used to pass around values that adhere to a trait instead of a particular type. The following is a pseudocode showcasing the same:

Listing 3: Traits and Trait bounds example

```
trait SharedTrait {
    fn some_operation();
}
struct Type1 {
    // Fields
}
struct Type2 {
    // Fields
}
impl SharedTrait for Type1 {
    fn some_operation() {
        // definition for type 1
    }
}
impl SharedTrait for Type2 {
    fn some_operation() {
        // definition for type 2
    }
}

let message: Message<SharedTrait> = new_message();

spawn!(TASK_PRIORITY1, STACK_SIZE, {
    if(..) {
        let msg = Type1::new();
        message.send(msg);
    } else {
        let msg = Type2::new();
        message.send(msg);
    }
});

spawn!(TASK_PRIORITY2, STACK_SIZE, {
    let msg = message.receive();
    if let Some(val) = msg {
        val.some_operation(); // this is correct indifferent
             to the type of message as value returned will
            definitely implement the SharedTrait.
    }
});
```

One can define traits which don't have any methods defined in them, these

are called Marker traits. Rust uses marker traits to define basic properties of types [6]. Following are few marker traits used by HarSaRK:

- Copy: Types whose values can be duplicated simply by copying bits.
- Send: Types that can be transferred across thread boundaries.
- Sync: Types for which it is safe to share references between threads.

1.2 Strong types using Sum-types

In Rust, enums represent one of many possible variants. Rust goes a bit further and allow each variant to store data, each variant can hold data of different type. The combination of enum and union-like behavior is often referred to as a *tagged union* in languages like C, and functional languages refer to it as a sum type. Though the variants are disjoint, they collectively form a single type. The following is a sum-type representing IpAddr V4 and V6, where each variant of the enum stores data of different types. This subsection further talks about what makes sum-types so powerful. An important safety factor of a

Listing 4: Sum types

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}
```

programming language is how it handles null values. Most of the programming languages support the use of null pointers. Dereferencing null pointers leads to a runtime exception. The *Option* type encodes the very common scenario in which a value could hold something or nothing. The *Result* type helps represent cases where a function might not succeed, so the return value will either contain the value or the error [1]. Expressing this concept in terms of the type system means the compiler can check whether the developer has handled all the cases that should be handled instead of runtime crashes. *Option* and *Result* are better implementations of Null pointers and exception handling [1]. This is because operations on values that might be *Null* cannot be identified at compile-time, but operations on *None* can be as it is encoded into the type-system. Thus a large set of bugs can be warded off at compile time.

When the *receive* method is called on an instance of a *Message* primitive in HarSaRK, the method returns an *Option*<*T*>. If the task has received a message, then the method will return the message wrapped in *Some*<*T*> variant of the *Option*<*T*> enum, and if no message was sent to the task, then it returns None.

HarSaRK represents all kernel errors as an *enum*, and all kernel APIs which can possibly fail return a *Result*<*T, KernelError*>, where *T* will vary based on the functionality of the API. Enums combined with Rust's robust match expressions[7] make error handling in the application very clean.

Listing 5: Option and Result Enum

```
enum Option<T>{} {
    Some(T),
    None
}
enum Result<T,E>{
    Ok(T),
    Err(E)
}
```

Listing 6: Task Definition

```
let message: Message<SomeStruct> = init();

let msg = message.receive();
msg.some_method(); // error as receive returns a Option type
     not SomeStruct

if let Some(val) = msg {
    val.some_method(); // this will only be executed when
        receive returns a Some<SomeStruct>.
}
```

The following are few examples of application code calling kernel APIs and how errors can be handled:

Listing 8: Error handling

```
//Note that panic!() prints the error and exits the
    application.

// Error handling for task creation.
match spawn!(TASK_PRIORITY, STACK_SIZE, {
    hprintln!("TASK 1");
}) {
    Ok(_) => {
        // Task was created successfully, so we can continue
             execution.
    }
    Err(KernelError::NotFound) => {
        panic!("This task priority does not exist.");
    }
    Err(KernelError::StackTooSmall) => {
        panic!("Please increase the stack size.");
    }
    Err(KernelError::Exists) => {
        panic!("The task for this priority has already been
            configured");
```

Listing 7: Error enum

```
pub enum KernelError {
    NotFound,
    StackTooSmall,
    LimitExceeded,
    AccessDenied,
    Empty,
    Exists,
}
```

```
    }
    Err(err) => {
        panic!("Unexpected Error: {:?}", err);
    }
};

// Error handling for resource locking.
static res2: Resource<[u32; 2]> = Resource::new([4, 5],
    TaskMask::generate([task3]));

match res2.acquire(|res| {
    return res[0] + res[1];
}) {
    Ok(val) => {
        hprintln!("Output: {:?}", val);
    }
    Err(KernelError::AccessDenied) => {
        panic!("This task does not have access to the
            resource.");
    }
    Err(KernelError::LimitExceeded) => {
        panic!("PiStack is full.");
    }
    Err(err) => {
        panic!("Unexpected Error: {:?}", err);
    }
}
```

1.3 Concurrency Safety

Though every real-time kernel provides locking primitives, they cannot guarantee that the end application will be free from data races due to lack of language support. Concurrently executing tasks mutating shared data without any control over ordering can drive the system into an inconsistent state, and it might be impossible trace what exactly happened. Memory safety and concurrency bugs often come down to code section accessing data where it shouldn't, so Rust's memory protection model of ownership and borrowing helps guarantee

concurrency safety [8].

The Rust compiler requires that, variables that are accessed by different functions when not passed as arguments should implement the Sync trait, else the code will not compile. Sync trait is a marker trait that when implemented hints the compiler that this type is safe to be accessed by multiple threads of execution. This rule will apply to data shared between tasks and between a task and an interrupt as they both are implemented as functions. Immutable variables and constants can be shared between tasks without the risk of safety. Mutable variables do not implement the Sync trait, thus compiler won't allow to share them across tasks. When mutable variables need to be shared across tasks, they need to be wrapped inside a concurrency safe data structure (like Mutex). Such concurrency safe data structures implement the Sync trait. However, constructs like Mutex are dependent on the Operating system, and are not available when programming directly on hardware.

As HarSaRK is designed for single-core systems, 2 tasks running in parallel and mutating shared data is a not-issue. But the tasks in real-time systems are driven by device interrupts raised by sensors based on change in environmental state. These interrupts can mutate global data or will enable tasks defined by the application, and if this enabled task has a higher priority than the task that the interrupt preempted, then after exiting the interrupt handler, the kernel will execute this new task. Here comes the problem, as interrupts can be raised at any point in time, so it is very much possible that an interrupt is raised *while* a task was modifying some shared data-structure. The problem here is that, at this point when the interrupt accesses this shared data-structure, it is in an inconsistent state and any code reading/modifying it as part of this interrupt, or the higher priority task enabled by this interrupt will drive the whole system into an inconsistent state. Hence we need to ensure that after every kind of context switch, the shared data structures are in an consistent state.

The 3 synchronization primitives provided by HarSaRK: Resource, Semaphores and Message primitive implements the Sync trait. Thus the Application code can use them to synchronize between tasks. Section **??** explains the design of these primitives, and how the APIs are designed such that it is impossible to misuse them.

1.4 Enforce Invariants

In mathematics, an invariant is a property of a mathematical object which remains unchanged after operations or transformations of a certain type are applied to the objects [9]. Similarly, to ensure functional safety of the system, various invariants need to be guaranteed. An excellent example of an invariant is if a resource primitive is locked, then it must be unlocked after use and before the task exits. If this is not ensured, it can lead to many tasks getting blocked forever. Breaking of invariants like this can open-up safety-critical issues in the system. Unless statically enforced, violations of such invariants can be difficult to catch as it is hard to trace it manually in case of complicated control flows; even worse, these erroneous control flows might be so rare and specific that they

might not show up during the testing phase of development.

Rust's verbose type system, ownership and borrowing model, and support for modern language constructs allow HarSaRK to enforce few important constraints at compile-time.

1.4.1 Task declaration

Task definition in HarSaRK requires that the tasks be infinitely running and do not terminate. If not, then every time the task exits, the task stack has to be cleared, and the new task's stack has to be initialized appropriately. This adds considerable overhead to task switch time. Infinitely looped tasks are a general approach taken by other real-time kernels too (eg. FreeRTOS). There are two main issues with this; the first is that some specific control flow can lead to the line of control to get out of the infinite loop defined. The other issue is that this is very constrictive by design and can lead to starvation of lower priority tasks as higher priority tasks can end-up looping forever.

We address the first issue by using the *Never* type (represented by *!*) [10]. When a function returns a *Never* type, it implies that, the function shall never return, i.e. an infinite loop. This fits perfectly for defining tasks. So while defining tasks the developer could define them as *fn task() ->!*. One of the important steps during the static analysis is the return value validation. In this step the Rust compiler performs strict checks for all branch and loop statements in a function to validate if the type of value returned by the function matches the function definition. For example the following function definitions will result in a compilation error:

Thus when the function returns Never type in the definition, the Rust compiler can guarantee at compile time that the function will not return under any circumstances. In this way, we can statically determine the correct definition of tasks. Tasks are created using the kernel API *create_tasks()* which accepts several arguments, one of which is a pointer to the task definition (a function pointer). The function declaration is such that it only accepts a function returning Never type.

To address the second issue the kernel provides a *task_exit()* API, this function notifies the kernel that the currently running task has ended and it has to schedule a new task. So the tasks can be defined as follows : This design works great, except for the fact that there developer will have to deal with the boilerplate code now. We address this by defining a macro for creating tasks called *spawn!*. Now, the developer provides the task definition as a function closure inside the spawn macro. The compiler expands it into the desired format, thus abstracting away the implementation details, making the API easier and cleaner to the developers. An example of task definition with *spawn!*:

1.4.2 Access Control of wrapped data

While Semaphores allow a task/interrupt to notify other tasks, Resource and Message primitive share data between tasks. So access to the wrapped data

Listing 9: Task Definition

```
fn task() -> i32 {
    // ERROR : no return
}
fn task() -> i32 {
    let x = 6;
    if x > 6 {
        return 3;
    } else {
        x = 8;
        // ERROR : no return in this branch
    }
}
fn task() -> ! {
    let x = 9;
    for i in 0..100 {
        if x>i {
            break;
        }
    }
    // ERROR : function returns when Never return is defined
}
fn task() -> ! {
    loop {
        /* CODE */
        break;
    }
    // ERROR : function returns when Never return is defined
}
```

needs to be controlled by the primitive to ensure safe access. This is achieved through the concept of ownership. The Resource/Message primitives are made the owners of the shared data, any task wanting a reference to this data will have to go via the methods exposed by the primitives:

- Resource primitive: the *lock* method returns a reference to the shared data.
- Message primitive: the *receive* method gives the task a copy of the shared data. We are returning a copy instead of a reference as the same message has to be passed to multiple tasks, and giving each of them their own copy means they can manipulate it without being affected by mutations to it made by other tasks.

The Resource primitive provides methods *lock()* and *unlock()*. If due to some unexpected control flows unlock is not called by the task after using the resource, then the resource remains locked thus keeping many other tasks blocked, which in turn affects the responsiveness and questions the real-time guarantees of the

Listing 10: Task Definition

```
fn task() -> ! {
    loop {
        {
         // task definition
        }
        task_exit();
    }
}
```

Listing 11: Task Definition

```
const TASK_PRIORITY: u32 = 5; // task priority
const STACK_SIZE: usize = 256; // task stack size

spawn!(TASK_PRIORITY,
       STACK_SIZE,
       (|| {
            // task definition
       })
);
```

system. HarSaRK address this by making *lock()* and *unlock()* private to the resource primitive and providing an *acquire()* method on Resource which accepts a function closure [11]. *acquire()* internally calls *lock()* on the resource, if access to the resource was granted then it calls the closure with the value protected by the Resource else the closure is not executed. Once the function closure completes execution it calls unlock and returns. Exposing only acquire allows us to enforce another important invariant. SBPC requires that the locks should be released opposite to the order in which they were held. When using *acquire*, the inner-most lock will be released first and following that, as each scope ends the corresponding locks are released, this is opposite to the order in which the lock was acquired. Though function closures seems similar to function pointers, they are much more powerful. They allow access to values in the scope of the closure definition. C doesn't support closures thus making it impossible to achieve this feature in C based kernels. An example:

Listing 13: Task Definition

```
static res :Resource<Peripherals> = // initialize

spawn!(TASK_PRIORITY,
       STACK_SIZE,
       (|| {
            // instructions
            let x = 9;
            res.acquire(|perf| {
                if(perf.is..) {
                    x = 10; // This function closure has
```

Listing 12: Task Definition

```
let perip = init_peripherals();
static res :Resource<Peripherals> = Resource::new(perip)

// task definition
spawn!(TASK_PRIORITY,
       STACK_SIZE,
       (|| {
            perip.GPIO;
            // leads to compilation error as the instance of
                 Peripherals locked by Resource primitive is
                private and can be accessed only via the lock
                 method.
       })
);



                      access to x.
                  // Not possible with a function pointer
               }
            });
            // instructions
        })
);
```

Well defined variable lifetimes in Rust combined with the scoped locking and releasing model of *acquire*, ensures that there is no way the task can maintain reference to the shared data outside the closure passed to *acquire*. It can obviously make a copy of the shared data, but mutations to the copy won't affect the actual shared data, so this is safe. This also ensures another important invariant that all modifications to the shared data should be completed before the lock is released. As there is no reference to the shared-data after the lock has been released, we can statically guarantee that the shared data will not be mutated after it.

1.4.3 Limiting access to Synchronisation primitives

As mentioned in Section **??**, resource primitives should not be accessed by interrupts, rather interrupts will trigger tasks and pass messages to them if required, and they will handle mutating the shared resources. Similarly, Section **??** and **??**, explains why only tasks should be allowed to call *receive* on semaphores and messages. Though the developer might not intend to use these primitives inside the interrupt handler's directly, they might be calling some functions which internally are using them. Such bugs are hard to trace, thus HarSaRK enforces these restrictions statically:

- A struct named *Context* has been defined which has a integer field corresponding to the priority of the task called *priority*.

Listing 14: Task Definition

```
static res :Resource<Peripherals> = // initialize

spawn!(TASK_PRIORITY,
       STACK_SIZE,
       (|| {
            // instructions
            let x = 9;
            res.acquire(|perf| {
                x = perf;
                /*
                Fails at compilation. ‘perf‘ is a reference
                    to the shared data protected by the
                    resource primtive ‘res‘. ‘perf‘ is
                    dropped when the closure exits, so
                    attempts to copy perf, i.e. reference to
                    the shared data into an outer scope is
                    treated as invalid by the compiler.
                */
            });
            // instructions
        })
);
```

- The constructor and the *priority* field of *Context* is made private to the kernel library.
- For each task defined, an instance of *Context* is created with the priority of this task. An immutable reference to the *Context* instance corresponding to this task is passed as an argument to the task.
- The *acquire* method of the resource primitive, and *receive* method of semaphore and message primitive now take an immutable reference to a *Context* instance as an argument to it.
- *Context* does not implement Copy trait, and no other method other than the constructor method is defined for it.

The following is an example code snippet of a task trying to call *acquire()* over a resource:

The above defined system guarantees that *Context* can be constructed only inside the kernel library code, and that in the application code only the tasks have access to an instance of *Context*; also, the tasks cannot leak this instance of *Context* outside the task code:

- Because the constructor method of *Context* and the *priority* field is private to the kernel library, the compiler ensures that new instances of Context cannot be created outside the kernel code.

Listing 15: Task Definition

```
static res :Resource<Peripherals> = // initialize
spawn!(TASK_PRIORITY,
       STACK_SIZE,
       (|cxt: &Context| {
            // instructions

            res.acquire(cxt, |peripherals| {
                peripherals.initalize();
            });
            // Note how acquire takes reference to Context
                to work.

            res.acquire(|peripherals| {
                peripherals.initalize();
            });
            // This section will fail to compile as acquire
                function call was expecting reference to an
                instance of Context.

            // instructions
        })
);
```

- *Context* does not implement *Copy* trait so it cannot be copied, ensuring that new instances of Context cannot be created from the ones passed on to the task handler. This also implies that any attempts to assign a global variable of *Context* type with this instance will move it, i.e. delete the context instance inside the task and assign this instance to the global variable. But as this instance is not owned by the task, but rather has reference given by the task manager, the compiler won't allow the task to move this. This is the product of the strict ownership rules in Rust.
- The reference to the instance of *Context* given to the tasks are scoped to their task code, hence this reference is dropped at the end of task code. Thus any attempts to leak this reference outside the task will violate the lifetime checks made by the compiler and the code will not compile. [12]

This implies statically that interrupt handlers cannot call *lock* or *receive*.

1.5 Memory Safety

Memory safety for static variables and data allocated on stack is straight forward as they are self-managed. Static variables exist for the entirety of the application and do not need to be deleted, and stack allocations are automatically handled when entering and exiting a function.

The kernel has no hard dependency on an allocator, it does not use dynamic/heap memory as heap memory usage comes with considerable overhead [13]. The kernel data-structures are all defined as static variables and are created at compile-time. The kernel primitives themselves are required to be created as static variables, thus they are also allocated during compilation. This allows the users to build applications which does not use dynamic allocation.

As the users might require support for dynamic memory as part of their application development, HarSaRK optionally configures an off-the-shelf memory allocator that can be used by the application. We use Rust's support for conditional compilation so that if the user does not want dynamic memory, then we won't have the allocator compiled with the binary, and simultaneously support for dynamic data-structures will be retracted from the application code. The reason for this is that the Rust system manages heap memory statically with minimal overhead.

1.6 Safe Hardware abstractions

Microcontrollers consist of sections of silicon which are used for interacting with systems outside of it. For example, sensors, actuators, user interfaces such as a display, etc. These components are collectively known as Peripherals. Communication with these external devices is done via various ports. Configuring these embedded platforms involved reading and writing to various registers. But, directly writing values to registers is not considered a good approach because:

- Concurrent execution of tasks, multiple tasks can end up reading/writing to the hardware simultaneously, which can put the hardware in an inconsistent state.
- Not all the registers are the same, some are read-only, and some are read-write.

A safer design pattern is to design a structure representing the hardware peripheral and hold private fields corresponding to registers and methods that operate on them. These methods internally read from / write to the registers, and they keep track of which registers are readable and writable. The Rust embedded working group has developed libraries that define peripheral structs for various machine architectures and microcontrollers. These structs follow the singleton design pattern, i.e., only one instance of the peripherals struct can exist across the codebase. Any other attempt to initialize the peripherals will error out during compilation. This brings in a great number of safety guarantees with it. The only issue is sharing this common Resource safely across various tasks without leaving it in an inconsistent state. This issue can be easily addressed by wrapping the peripherals instance in a Resource primitive. Through this approach, HarSaRK can be used to access and program hardware safely by treating it like a sharable Resource.

1.7 Unsafe Rust

There are various scenarios in low-level system programming, for which full control over the function stack layout is necessary. A typical example is the implementation of *context switch*. This section of code is generally then written in assembly. However, this breaks the development workflow, and the Rust compiler cannot perform validations in these parts [**unikernel˙Rust**]. Thus, such code regions have to be explicitly marked unsafe by enclosing these sections in unsafe blocks. Rust recently added support for annotated inline assembly. These annotations allow the Rust compiler to perform static analysis on the assembly code and validate its safety to some extent. Further work is going on in this domain [14].

It is the responsibility of the developer to ensure well-defined behavior inside the unsafe blocks. Suppose the developer can guarantee that code inside the unsafe blocks does not lead to undefined behavior. In that case, the compiler can proceed to verify the rest of the codebase, assuming that this unsafe block has manually been verified to be safe [15].

References

[1] Carol Nichols Steve Klabnik. *Generic Data Types - The Rust Programming Language*. Accessed 2020-02-05. 2020. URL: https://web.archive.org/web/20200918035429/https://doc.rust-lang.org/book/ch10-01-syntax.html.

[2] Carol Nichols Steve Klabnik. *Traits: Defining Shared Behavior - The Rust Programming Language*. Accessed 2020-02-05. 2020. URL: https://web.archive.org/web/20200820231553/https://doc.rust-lang.org/book/ch10-02-traits.html.

[3] Matthias Erdin, Vytautas Astrauskas, and Federico Poli. "Verification of Rust Generics, Typestates, and Traits". MA thesis. 2018.

[4] CVE-Details. *Linux Kernel : CVE security vulnerabilities, versions and detailed reports*. Accessed 2020-02-04. 2020. URL: https://web.archive.org/web/20200918032929/https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[5] *Using Trait Objects That Allow for Values of Different Types - The Rust Programming Language*. 2021. URL: https://web.archive.org/web/20210510185018/https://doc.rust-lang.org/book/ch17-02-trait-objects.html (visited on 05/10/2021).

[6] *std::marker - Rust*. 2021. URL: https://web.archive.org/web/20210508144457/https://doc.rust-lang.org/std/marker/index.html (visited on 05/08/2021).

[7] *Match expressions - The Rust Reference*. 2021. URL: https://web.archive.org/web/20210216182152/https://doc.rust-lang.org/reference/expressions/match-expr.html (visited on 05/09/2021).

[8] *Critical section - Wikipedia*. 2021. URL: https://web.archive.org/web/20210508145437/https://en.wikipedia.org/wiki/Critical_section (visited on 05/08/2021).

[9] *Invariant (mathematics) - Wikipedia*. 2021. URL: https://web.archive.org/web/20210508185204/https://en.wikipedia.org/wiki/Invariant_%5C%28mathematics%5C%29 (visited on 05/08/2021).

[10] Steve Klabnik. *Never type - The Rust Reference*. Accessed 2020-04-17. Apr. 2020. URL: https://web.archive.org/web/20200417152240/https://doc.rust-lang.org/reference/types/never.html.

[11] Neal Gafter. *Neal Gafter's blog: A Definition of Closures*. Sept. 2020. URL: https://web.archive.org/web/20200918042437/http://gafter.blogspot.com/2007/01/definition-of-closures.html (visited on 09/18/2020).

[12] *Validating References with Lifetimes - The Rust Programming Language*. 2021. URL: https://web.archive.org/web/20210508145257/https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html (visited on 05/08/2021).

[13] *Dynamic Memory Allocation - Just Say No - Embedded Computing Design*. 2021. URL: https://web.archive.org/web/20210509170436/https://www.embeddedcomputing.com/technology/software-and-os/ides-application-programming/dynamic-memory-allocation-just-say-no (visited on 05/09/2021).

[14] Rust-lang team. *Inline-Asm Rust-lang RFC*. Accessed 2020-08-08. Aug. 2020. URL: https://web.archive.org/web/20200918042807/https://github.com/rust-lang/rfcs/blob/master/text/2873-inline-asm.md.

[15] Wicher Heldring. "An RTOS for embedded systems in Rust". English. PhD thesis. University of Amsterdam, June 2018. URL: https://esc.fnwi.uva.nl/thesis/centraal/files/f155044980.pdf.