Delft University of Technology
Master of Science Thesis in Computer Engineering and Embedded Systems

Reducing Programmer Burden in Task-Based Intermittent Computing Development

Marco Antonio van Eerden

TU Delft Delft University of Technology

Reducing Programmer Burden in Task-Based Intermittent Computing Development

Master of Science Thesis in Computer Engineering and Embedded Systems

Embedded Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Marco Antonio van Eerden

08-10-2025

Author
Marco Antonio van Eerden
()
Title
Reducing Programmer Burden in Task-Based Intermittent Computing Development
MSc Presentation Date
October 14th 2025

Graduation Committee

Dr. P. Pawełczak	Delft University of Technology
Dr. J.G.H. Cockx	Delft University of Technology

Abstract

Wireless sensors are used in many critical application field such as agriculture, healthcare and transportation. Such sensors often use batteries, which have a limited lifespan and contribute to environmental damage in the form of waste. Batteryless sensors solve these problems, but come with their own challenges; they frequently experience power failures because their energy storage is limited. To make sure the system can make forward progress, Intermittent Computing (ImC) techniques are used to back up the system state before the power fails. One such technique consists of splitting the program up into tasks, and backing up the system state after a task has fully executed. Such a task-based approach requires a non-standard programming model, often with many keywords that need to be inserted manually. This increases the burden on the programmer, moving focus to inserting the right keyword at the right time and not forgetting anything, instead of focusing on the things that really matter. To solve this problem, this work Code Instrumentation for Task-based Intermittent computing Development (CITID), a compile- and runtime ImC system that reduces the programmer effort of a state-of-the-art task-based ImC system by removing up to 96% of necessary manually placed keywords, while insignificantly affecting performance in most cases.

Preface

Intermittent computing (ImC) systems are a fascinating topic because they are so unconventional. We are used to our computers always having power (unless you forgot your laptop charger at home), but in ImC there is no such guarantee. When researching some of these systems, I was somewhat surprised at how unwieldy some of the implementations looked. User convenience was evidently not the first priority, which, of course, is to be somewhat expected. But I felt it could be a wall standing in front of more widespread adoption. So I endeavoured to try and see how possible it would really be to change an existing ImC system such that the burden that falls on the programmer is greatly reduced. This thesis is the culmination of these efforts.

Marco Antonio van Eerden

Delft, The Netherlands 15th November 2025

Acknowledgements
Many thanks to Dr. Przemyzław Pawełczak for the ideas, guidance and encouragement. Thanks to Dr. Jesper Cockx for being part of the thesis committee. Thanks to my family for the support, and to the people who came to my presentation. And the greatest of thanks to Zaza and Azraël, for being there when things were not so great.

Contents

Chapter 1

Introduction

The number of connected IoT devices globally is expected to increase to 41.1 billion by 2030 [34]. Many of these devices are wireless sensors, and are used in a variety of applications such as wildlife monitoring [2], agriculture [23, 31, 35], healthcare [3, 10], transportation [14] and industrial automation [27, 30]. Such wireless devices require an external power source, which is often a battery. However, batteries have a major downside for use in wireless sensing devices: their limited lifespan. This necessitates frequent battery replacement in sensors that are often far away or hard to reach [17]. In addition, spent batteries can cause environmental damage due to electronic waste [28]. Efforts have been made to extend the battery life in wireless sensors [5, 1], however, this does not solve the environmental issue.

One way to significantly reduce the environmental impact of wireless IoT devices is to eliminate batteries altogether. Such batteryless devices often use energy harvesters that use environmental power sources, such as solar radiation, RF signals or vibrations [33]. A side-effect of not having a battery is that the energy buffer is extremely small; this can cause the device to intermittently shut off due to a lack of power [37]. Such devices are called Intermittent Computing (ImC) systems.

A power failure can leave the system in an inconsistent state. For this reason, ImC systems contain Non-Volatile Memory (NVM) to save (part of) the system state between reboots [24]. When saving the system state, one must be careful: saving the state too often takes time away from useful computations, while not saving the state frequently enough may lead to a lack of forward progress. To help solve this problem, several solutions have been proposed in the form of code instrumentation and programming models. These solutions fall into two fundamental categories: checkpoint-based and task-based.

Checkpoint-based solutions use a compiler or other tool to instrument the code with checkpoints, or to determine the checkpoints at runtime. At each checkpoint, the state of the system is backed up into NVM. In task-based solutions, the program is divided into tasks, which denote checkpoint boundaries. Thus, the state of the system is backed up after each task execution. These two solution categories are discussed in more detail in Chapter 2. This work focuses on task-based ImC. Much research has already been done in this field, and the results range from C-like runtimes [26], to custom-tailored programming languages [13]. However, these systems have a major flaw: they only take into

account the development effort of the system in a limited capacity. The programmer must learn an entirely different language, or they must place runtime keywords in specific places, which is prone to error.

The developers of InK, a runtime ImC system and task scheduler, [42] tried to quantitatively measure the development effort in intermittent systems programming. InK was compared to Alpaca [25] and MayFly [13] in a user study. InK performed relatively poorly; only 59% of participants felt that it was easy to write a general-purpose ImC program in InK.

1.1 Problem statement and contributions

This work aims to reduce the required development effort for existing task-based ImC models. We choose to specifically improve InK because it is open source, was already studied in terms of development effort and has been used as the basis for multiple incremental works ([22, 4]). To this end, we introduce Code Instrumentation for Task-based Intermittent computing Development (CITID). This solution is a mixture of custom compile-time code instrumentation and the runtime task managment of InK. With the help of CITID, this work aims to answer the following research questions:

1. *How can we reduce development effort for a task-based ImC system while maintaining its performance?*
2. *How can we improve code instrumentation compared to state of the art open source alternatives for ImC systems?*
3. *How can we evaluate development effort for two task-based ImC systems using a Large Language Model (LLM)?*

In answering the above questions, this work contributes a more polished runtime system for task-based ImC, code instrumentation for task-based ImC that improves on previous work, and a novel evaluation using LLMs to assess development effort in CITID. We open-source our work for the research community to use [38].

1.2 Thesis outline

This section discusses the general outline of this work. We discuss related work in Chapter 2, including further elaboration of the concepts introduced in this chapter, such as batteryless devices and intermittent computing. Chapter 3 goes over the design choices of CITID, and Chapter 4 describes the implementation of this design. We compare CITID to InK in Chapter 5, both in terms of runtime performance and development effort. Lastly, Chapter 7 draws conclusions based on the results gathered, and discusses future work.

Chapter 2

Background

This chapter discusses related work in the field of ImC, with a special focus on ImC programming models.

2.1 Checkpoint-based ImC systems

The first category concerns checkpoint-based systems. These systems insert checkpoints into the code, such that the system state can be backed up when code execution reaches these checkpoints. One such system is So Far So Good (SFSG) [41]. Whenever a power failure occurs, SFSG records a trace where this happened at runtime and inserts a checkpoint slightly before this point. The checkpoint is moved further backwards until it is executed, at which time the state is backed up. A compiler is used to instrument the code to help with trace collection. The Liveness-Aware CheckpoinTing (LACT) [19] system uses the compiler to reason about intermittent data during checkpointing: if a value is not live when a checkpoint happens, it will not be copied to NVM. This reduces the overhead of checkpointing significantly. TICS [21] provides a programming model to deal with timing of sensor data. In addition, it is able to statically insert checkpoints into legacy code, removing the need for the programmer to rewrite that code for use in ImC. WARio [20] manages to reduce the number of checkpoints statically by leveraging Write After Read (WAR)-conflict detection. WARs happen when a variable in NVM is written after it is read, and then a power failure occurs (see Figure 3.1). By grouping write instructions to NVM variables, WARio is able to significantly reduce the number of checkpoints required for correct execution. NACHO [29] uses a data cache to generate checkpoints when a potential WAR can happen when a cache line about to be written back to the main persistent memory. RockClimb [7] goes a step further, and aims to eliminate checkpoints entirely. It uses the compiler to form power-failure-immune regions in the code and guarantees that no power failures will occur, since the runtime ensures that no region will execute before the on-board capacitor is fully charged.

2.2 Task-based ImC systems

The last category of ImC are task-based solutions. These often come in the form of a *programming model*, where non-standard programming structures are used to achieve correct intermittent execution at runtime. In most task-based systems, the system state is backed up after a task has finished executing in full. In essence, task boundaries effectively act as checkpoints.

Chain [8] is an early task-based ImC system. The programmer must divide the program into tasks using the Chain programming model. They can then use *channels* to manipulate non-volatile data and communicate between tasks. These channels are either incoming or outgoing, and the channel data is double-buffered internally when a task transition happens. Mayfly [13] focuses on the timeliness of its data. The sampling period of a sensor is kept track of between reboots, and some data may be discarded if it is too old (similar to TICS). This allows Mayfly to remove the task of manually programming the timing requirements. Alpaca [25] uses the compiler to automatically detect WAR conflicts. This allows it to automatically move data into NVM at compile time when required, removing the manual labour normally required by the programmer. Coala [26] reduces the overhead required for backing up NVM data by coalescing tasks when enough energy is available. Data is only backed up after all coalescing tasks execute, reducing the need for backups and thus reducing total backup overhead. InK [42] is a task-based runtime and task scheduler. It uses threads to encapsulate a group of tasks, and threads can be pre-empted by higher priority threads or interrupts. This reduces the response time required to respond to an external interrupt. Since InK is used as a basis for CITID, it is discussed in more detail in Section 6.2. InK also forms the basis for LATICS [22] and REHASH [4]. LATICS is similar to Coala; it coalesces tasks at runtime, but also uses static analysis to determine when the state must be backed up, regardless of runtime coalescing. REHASH adaptively schedules different threads depending on the available energy at runtime.

Chapter 3

Design

This chapter covers the CITID design. First, design considerations that need to be taken into account are discussed in Section 3.1. The design in accordance with these considerations is presented in Section 3.2.

3.1 Design considerations

The design considerations for CITID pertain to the research questions set out in Section 1.1. In addition to considerations for these goals, some other aspects of the system need to be taken into account, such as its intermittent nature.

3.1.1 Reducing programmer burden

The primary goal of this work is to reduce the development effort when programming an intermittent computing system using a task-based approach. In order to achieve this goal, we first state the following points about InK that require more development effort:

1. The `main` function is embedded into the InK library and contains code that is not relevant for InK to work, such as disabling the watchdog timer and intialising GPIOs for the LEDs on the evaluation board. This may not correspond to what the programmer wants to do. In addition, disabling the watchdog can be counter-productive when the computing system is used in an environment where automatic resets are required.

2. InK exposes nearly all of its internal functions to the programmer. The developer may accidentally use a function that should not be used, and must in any case be careful not to use them.

3. Publicly accessible InK functions and macro's are not prefixed with a namespace, which makes name collisions more probable. If this happens, the programmer must manually resolve the name conflicts.

4. InK requires explicit creation and signalling of threads at first boot. This needs to be done separately for every thread, which can lead to bugs if this is forgotten.

5. InK requires explicit getting and setting of task-shared variables using macro's. This is prone to introducing bugs, since there is no mechanism to alert the programmer that they have forgotten to use one of the access macro's. It also requires extra effort on the part of the programmer, since they have to use a macro for every task-shared variable access.

6. In InK, it is not trivial to initialise task-shared variables on first boot in a power-failure-safe way. Task-shared variables cannot be initialised at the global scope are only available to task functions. As such, initialising variables on first boot requires a separate task-shared flag and a check inside an entry task.

7. The programmer is required to explicitly mark all task functions. This is superfluous; since one task always follows another, it is possible to deduce which function is a task function if the root task function of a thread is known.

To solve issue 1, we can allow the programmer to create their own `main` function. The programmer then needs to manually activate the scheduler. One may argue that this increases development effort, but the call to activate the scheduler only needs to be done once. In addition, by using the `main` function in the way it is used in most applications, we avoid having the programmer need to learn about a new way to initialise the application.

Issues 2 and 3 can be solved by designing an API with namespaced functions and macro's. This API should restrict access to functions, macro's and variables that are not supposed to be used by the programmer. This consideration trades reduced flexibility for increased reliability.

A solution to issue 4 is a construction where thread creation is intertwined with the definition of task functions. When the programmer defines the first task function in a thread, they also need to define the thread priority and whether it will activate on first boot or not. In this way, it is harder for the programmer to forget about having to initialise the task.

Finally, issues 5, 6 and 7 can be solved by instrumenting traditional C code using a compiler. The programmer would be able to write normal C code, without having to worry about task-shared variables, task functions or restrictions on variable initialisation. Existing systems that employ this solution are Alpaca [25] and TICS [21]. However, Alpaca requires the program to be compiled using `clang`, while this work uses MSP430-GCC. Additionally, the TICS instrumentation works only for intermittent computing systems that are not task-based.

3.1.2 Maintaining performance

In addition to reducing programmer burden, we want to maintain system performance in the majority of cases. To aid in achieving this goal, we add some considerations that we must keep in mind when designing the system.

Consideration 1: when accessing task-shared data, keep the amount of instructions used to a minimum. Shared data can be accessed in many places, from a one-time access to multiple accesses inside a hot loop. In the first case, extra instructions will not affect significantly performance. But in the latter case, the extra instructions per access could have a significant effect.

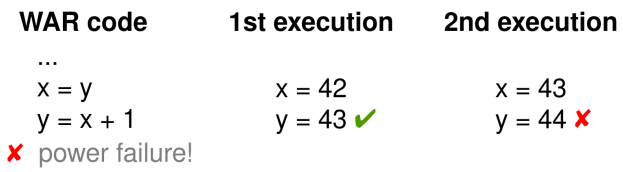

Figure 3.1: **This figure illustrates a WAR conflict inside a task. Here, `x` is a volatile variable, and `y` is non-volatile (i.e. persistent). On the first line, `y` is read, and on the second line, `y` is written. In the first execution of this task, `y` has the value 43, which is expected. However, a power failure occurs after the write, so the task re-executes from the beginning. Since `y` is persistent, it keeps its previous value; in the second execution, `y` has the value 44, which is incorrect.**

Consideration 2: when accessing any pointer, we must decide as quickly as possible if it is a pointer to task-shared data or not (see Section 3.1.4). For the same reason as in Consideration 1, slow pointer accesses could have a detrimental effect on performance.
Consideration 3: keep reboot initialisation code to a minimum. For the initial boot, we may take some time to set things up, as the system will not be in a running state. However, on reboot, we must be careful not to add too many instructions to the initialisation code. Reboot latency is important to get enough work done, since power failures are common.
Consideration 4: do not change the inner working of the scheduler. The point of this work is not to improve the runtime performance of InK, but to reduce the developer effort in developing applications of InK while *maintaining* runtime performance. As such, changing any of the inner workings of InK for the benefit of increasing performance could introduce a bias into the results, so this should not be done. Z

3.1.3 Intermittency

The intermittent nature of the computing system requires us to store task-shared variables in NVM. In addition, they must be double buffered, for the following reason. If a task-shared variable is read first, then written, and then a power failure occurs, we will have a WAR conflict. If there is no double buffering, that variable will have the wrong value once the task is re-run and the variable is read again. See Figure 3.1 for an illustration of this problem. If the variable is double buffered, however, we can instead choose to re-run the task with the untainted buffer, ensuring that correct values for the variables are used. It is possible to use undo-logging, where changes are undone on task re-execution, or redo-logging, where changes are committed only on task completion [39]. Since InK uses undo-logging, CITID will use this as well.

InK uses the principle of *data encapsulation*, which restricts the use of task-shared data to only one thread [42]. This keeps the variables from being manipulated by other threads, which could corrupt the shared data. CITID will use this principle as well, and additional checks can be added to alert the programmer if multiple threads can have access to the same shared data.

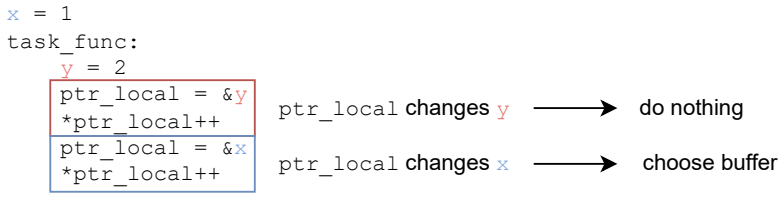

Figure 3.2: **This figure illustrates how pointers fit into the design of CITID. `x` is a tasks-shared persistent variable, and `y` is a local volatile variable. As shown, `ptr_local` can point to either the non-persistent variable `y` or the persistent variable `x`, and this can change at run time. So we must choose at runtime whether to do nothing, or to choose the correct buffer for the persistent variable.**

3.1.4 Pointers

Pointers require special consideration when instrumenting the C code written by the programmer. Because pointers can be created and manipulated at runtime, it may be impossible to know what a pointer points to at compile time. As such, *all* pointers inside task functions must be checked at runtime: if the pointer points to task-shared data, this data must be retrieved from the correct buffer. See Figure 3.2 for an illustration.

Task-shared data encapsulation also has an effect on how we deal with pointers. If there were limited encapsulation, there could be a memory region reserved for *thread*-shared memory. However, any task could create a pointer to any variable in this memory region at runtime, which would allow that task to indirectly manipulate the thread-shared variables. We could use the same construction as is used for task-shared data: if the pointer points to thread-shared data, retrieve the value from the correct buffer. This requires an additional check for any pointer that does not point to task-shared data. We must also avoid backing up the thread-shared data as much as possible to maintain performance. This requires an additional test for checking if the pointer is a write, and only then backing up the thread-shared data. Whether a pointer is a read or write can be checked at compile time, but this requires more effort to implement. Despite these disadvantages, the advantage of using a thread-shared memory region is that developers no longer have to worry about manual guarding of InK channels, which reduces development effort when sharing data between threads.

3.1.5 Local static variables

Even though locally declared static variables are only accessible in a single function, they do have global storage [6]. This means that the variable state will be saved across task executions. Because of this, not instrumenting local static variables can lead to data corruption, see Figure 3.3. In this scenario, one task execution writes to the variable. Then another task execution reads from the variable. If a power failure occurs between the two task executions, the variable will be reset and may not have the desired value; the data has been corrupted. In order to remove this corruption possibility, all locally declared

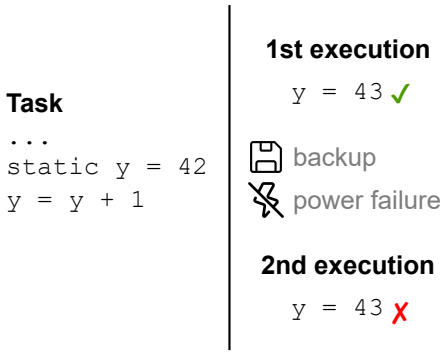

Figure 3.3: **This figure illustrates why static local variables must be persistent. Since the variable `y` has global storage, we expect it to increment every task iteration. However, if power fails between task iterations and `y` is *not* persistent, then `y` will have the wrong value on reboot.**

static variables inside task functions must be double buffered inside NVM, just as globally declared variables.

3.2 System design

The design of CITID consists of three major parts, the API (Section 3.2.1), the runtime (Section 3.2.2) and the code instrumentation (Section 3.2.3). The programmer interfaces with the runtime through the API. The code instrumentation adds CITID-specific code to the source files. These instrumented source files are then compiled into machine code, which can be run on the target hardware.

3.2.1 API

The API design of CITID is discussed in this section. The API design follows a minimalist approach, to keep the number of interfaces from the programmer to the runtime backend small. The API functions exposed to the user are also namespaced. This design approach resolves issues 2 and 3 listed in Section 3.1.1.

Scheduler: After the runtime is initialised, the scheduler should run. It is imperative that the programmer has control over when this happens, since some initialisation may be required on the application side before any task can run. As such, in the CITID design the programmer manually calls the InK scheduler. This may seem counter-intuitive to reduce development effort, but it allows full flexibility in initialisation *before* running the scheduler, without having to define a specific function as in InK. The scheduler itself remains unchanged, to account for Consideration 4 in Section 3.1.2.

Thread creation: To create a thread in CITID, just one macro needs to be called. In this macro, the programmer specifies the thread priority and whether

it activates on initial boot. Additionally, the programmer defines the first task function immediately afterwards; this does not need to be done explicitly. This implicit thread signalling covers point 4 in Section 3.1.1.

3.2.2 Runtime

Most of the CITID runtime is the same as the InK runtime, to be able to make a fair comparison when analysing performance figures. There are, however, a few changes in in the design pertaining to the runtime, which are discussed below.
Initialisation: Initialisation of the runtime is done implicitly. This hides the initialisation logic away from the programmer; they can be sure that the system is always properly initialised without having to call some initialisation function. The initialisation routine initialises all InK runtime components and creates the threads required by the application. This solves point 1 in Section 3.1.1. In order to satisfy Consideration 3 in Section 3.1.2, thread creation only happens on first boot.
Memory model: The design introduces a modified memory model in order to facilitate trivial initialisation of task-shared variables at first boot (point 6 in Section 3.1.1). The programmer declares and defines task-shared variables the same way as in normal C code. However, we still need a way to double-buffer the variables, as described in Section 3.1.3. To do this, we assign two task-shared memory regions to every InK thread. Figure 3.4 shows how these regions fit into the Micro Controller Unit (MCU) memory map. Since the number of threads is limited to 64, defining the memory regions is a feasible task. In addition, if a thread is not created, the size of its assigned memory regions is 0; there is no memory overhead for threads that are not used. All task-shared variables in a thread are placed in the first region. The second region acts as the double buffer.

For every thread, its two assigned regions are offset from each other by a fixed amount, which is the same for all threads. Just as in InK, one buffer is the active buffer, and the other is the backup buffer. The active buffer is used to write and read variables while a task is executing. If a power failure occurs, the backup buffer contents are copied to the active buffer to undo the changes of the interrupted execution. Hence, CITID uses undo-logging.

To avoid more copying of data, the active buffer pointer is swapped to the other buffer upon successful task completion. Therefore, upon accessing a variable, we must determine which buffer is the active one. If the first buffer is used, no offset will be added to the variable address. If the second buffer is used, the offset (see Figure 3.4) will be added to the variable address. This check must be done at run time, since we do not know in advance which buffer will be used during which task execution.

3.2.3 Instrumentation

Code instrumentation is an essential part of CITID and contributes greatly to the reduction in development effort. The programmer no longer needs to worry about which variables are task-shared and which are not; this is determined by the compiler. Figure 3.5 shows a global overview of the compiler pass phases.

The first pass is function labeling, where all task functions in a thread are marked with a label so they can be easily identified in subsequent passes. The

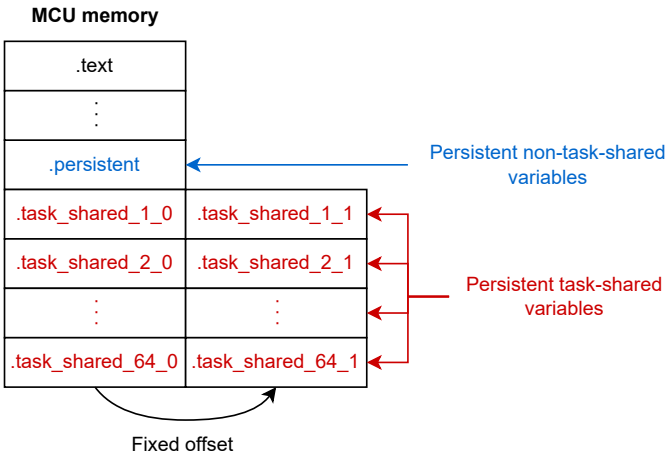

Figure 3.4: **This figure illustrates how CITID memory regions fit into the MCU memory. Each thread has two memory regions for double-buffering task-shared data:** `.task_shared_x_0` **and** `.task_shared_x_1`**, where** `x` **is the priority of the thread. For each thread, the buffer with number 1 is offset from number 0 by a fixed number of words. In addition to the task-shared regions, there is a single** `.persistent` **region for variables that need to be persistent but are not associated with any tasks.**

labeling is done in a top-down tree-like fashion, starting at the root task of a thread. Figure 3.6 illustrates this principle. Eventually, this compiler phase will find all task functions descending from the root task, solving point 7 in Section 3.1.1. The second pass is pointer instrumentation. This pass instruments all pointer dereference accesses inside tasks; this is necessary for the run-time checks as described in Section 3.1.4. The final pass instruments all task-shared variable accesses, including static local variables, as stated in Section 3.1.5. To maintain flexibility, it is possible for the programmer to explicitly mark a variable as non-task-shared. These variables are ignored in this pass.

3.3 Strengths and limitations

The strength of this design is that there are no constraints about how the programmer chooses to implement their application, as long as they respect the data encapsulation principle. Variables can be declared or defined anywhere and can be of any type. The majority of the application will be standard C code; the only CITID keywords and API functions the programmer needs to use are related to thread creation, timers and event handling. A minimum CITID program only contains one single API keyword for thread creation, and one function call for starting the scheduler.

Even though the design has some strengths, it also has some limitations. There is an additional constraint due to the nature of the implementation (see

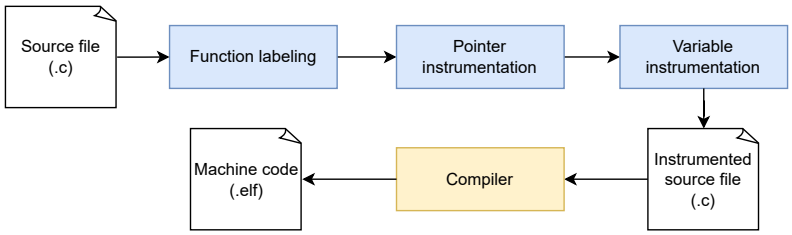

Figure 3.5: **How instrumentation works in CITID. The programmer writes the source file in C, then the instrumentation passes are used on it (in blue). This yields an instrumented source file, which is passed to the compiler (in yellow). This can be any desired compiler.**

```
task1:
    ...
    next = task2

task2:
    ...
    if x: next = task3
    else: next = task4

task3:
    ...
    next = task1

task4:
    ...
    END
```

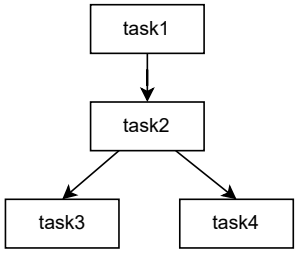

Figure 3.6: **Construction of a tree of task functions in the function labeling instrumentation pass. `task1` is the root task of the thread.**

Chapter 4); it is no longer possible to declare or define task-shared variables that have their type defined below the first task function. However, we believe that this is an acceptable compromise since global variables are declared at the beginning of a source file in most cases. Another limitation is that this design uses slightly more memory: one pointer for every task-shared variable. This corresponds to about 2 bytes of memory overhead per task-shared variable on 16-bit systems, or 4 bytes on 32-bit systems. Even though many intermittent computing systems are memory-constrained, we believe that the memory overhead of this design is small enough to be practical in the vast majority of use cases.

3.4 Theoretical future improvements

This section presents theoretical design improvements for the design discussed in Section 3.2. These improvements aim to further reduce runtime overhead, by directly using the task-shared variables instead of going through pointers. In addition, the task-shared variables should be passed to each individual task; this is how InK works as well. If both of these improvements are made, the

runtime performance should be nearly identical to that of InK, except when pointers are used.

An issue with this approach is that it is harder to implement; variables must be defined separately from their declarations. Splitting variable definitions and declarations is not trivial, as some variable definitions may depend on the value of other variables. In addition, it does not solve the constraint mentioned in Section 3.3, where task-shared variables cannot be declared with a type that is defined below the variable definitions. As such, the implementation of these improvements is left to future work.

Chapter 4

Implementation

This chapter discusses the implementation of CITID in three parts. Section 4.1 shows how the CITID API is implemented, including functions and keywords. The runtime implementation is shown in Section 4.2, which includes some corrections to the earlier InK work. Lastly, Section 4.3 discusses the implementation of code instrumentation for CITID.

4.1 API

The CITID API is implemented in C. Its functions and keywords are shown in Table 4.1. All functions and keywords are prefixed with a namespace to avoid name conflicts when CITID is used in a project. Only the API functions are exposed to the programmer; all CITID-internal functions cannot be used. Some functionalities of the API do not have an InK equivalent and require additional elaboration. As such, we further discuss these functionalities here.

The `scheduler_run` function is a non-returning function that must be executed once after all system initialisation. It activates the scheduler and runs any active tasks. The `CREATE_THREAD(p, a)` macro can only be used once per translation unit, to maintain consistency with the data encapsulation principle. It explicitly requires the programmer to state whether or not the thread is active at first boot, leading to fewer errors in this regard (see Section 5.1). In addition, it defines the entry task of the thread. Because the entry task is named internally, the `THREAD_ENTRY_TASK` macro must be used to refer to the entry task of the thread.

4.2 Runtime

The CITID runtime is fundamentally the same as that of InK, with a few exceptions. The runtime initialisation is different due to the different API, and is discussed in Section 4.2.2. The different CITID memory model is shown in Section 4.2.1. Finally, there are some correctness issues in the implementation of InK that had to be resolved; those are discussed in Section 4.2.3.

Table 4.1: **Overview of CITID API functions and keywords. Functions are lowercase, keywords are uppercase. In the actual implementation, all API functions and keywords are prefixed with a namespace. This is omitted in the table for brevity. API functions for the timers are the same as in InK, so these are omitted as well.**

CITID API	InK equivalent	Description
`is_first_boot`	—	If system is in first boot
`is_initialized`	—	If the runtime is initialized
`scheduler_run`	—	Run the scheduler
`CREATE_THREAD(p, a)`	`__CREATE` `__SIGNAL`	Create a with priority `p`, activated at first boot if `a` is `true`
`activate_thread(p)`	`__SIGNAL`	Activate thread with priority `p`
`THREAD_ENTRY_TASK`	—	Pointer to the thread entry task function
`signal_event_isr(p, e)`	`__SIGNAL_EVENT` `__EVENT_BUFFER_FULL`	Signal thread with priority `p` using event `e`
`IGNORE`	—	If a variable is annotated with this macro it is ignored by the instrumentation compiler
`PERSISTENT`	`__nv`	If a variable is annotated with this macro it is ignored by the instrumentation compiler and put into non-volatile memory

4.2.1 Memory model

As described in Section 3.2.2, each thread has its own task-shared memory region defined in the linker script. These memory regions are split into two buffers, with a fixed offset. On initialisation, each buffer is associated with its thread (see Section 4.2.2). Double-buffering is implemented as two `struct`s of pointers to shared data. The pointers in one `struct` point to the data in the active buffer, and the pointers to the other `struct` point to the data in the backup buffer. An overview of this model is shown in Figure 4.1.

Which pointer structure to use is determined at runtime. Every time a task executes, it looks up the current active buffer index inside the scheduler's internal state and chooses the pointer structure corresponding to that buffer. The task-shared variables are then accessed using a dereference operator. An exception to this is arrays, which are accessed directly.

Another thing that needs to be taken into account are static local variables. These variables must be instrumented, as the same variable can be accessed between different task executions. However, adding these variables to the pointer `struct` may lead to name conflicts, since it is legal for global task-shared variables to have the same name. To solve this issue a unique number is added to every variable in the instrumentation phase.

An additional exception for static variables is that its address is not available in the global scope. As such, the runtime initialisation cannot initialise the pointer to this variable. The runtime *must* therefore initialise the pointer to any static local variable *inside* the task function, after the static variable is declared.

Lastly, access to pointers outside of the `struct` is tricky, since pointers may point to any place in memory, including task-shared memory. As such, we must

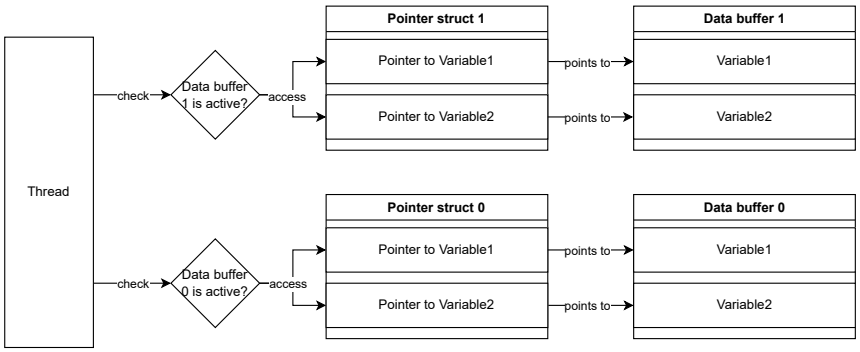

Figure 4.1: **High-level overview of the CITID memory model implementation. When a thread wants to access some task-shared variable, it first needs to access the correct pointer `struct` by checking which data buffer is the active one. The pointer inside the pointer `struct` then points to the variable in the correct buffer.**

determine if an accessed pointer points to task-shared memory *at runtime*. The algorithm used to do this is shown in Algorithm 1. The runtime calls a function that checks if the pointer address is within the task-shared memory region of the current thread. If it does, an offset may be added to the pointer address if the active buffer is different from the buffer that the pointer points to.

Algorithm 1 Algorithm for accessing the underlying variables of pointers in CITID.

```
1: buffersStart                          ▷ Start of the task-shared buffer region
2: buffersEnd                            ▷ End of the task-shared buffer region
3: bufferOffset                          ▷ Offset between task-shared buffers
4:
5: procedure PointerAccess(address, bufferIndex)
6:     if address ≥ buffersStart and address < buffersEnd then
7:         return address + bufferOffset * bufferIndex
8:     else
9:         return address
```

4.2.2 Initialisation

CITID runtime initialisation is implemented such that the programmer does not have to do anything for the runtime to be properly initialised. To achieve this, all intitialisation code is implemented using C constructor functions with `__attribute__((constructor(p)))`, where `p` is the priority of that constructor. There are four types of constructors in CITID. The relationships between them are shown in Figure 4.2.

- **Runtime initialisation begin**: this constructor function initialises the CITID internal runtime components, including the FRAM, scheduler, event system and timers. It also resets the flag used in the `is_first_boot`

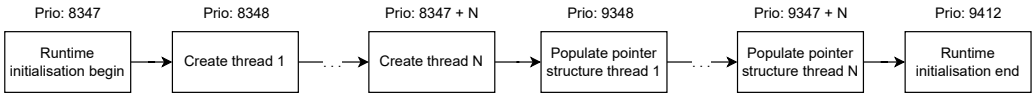

Figure 4.2: **Runtime initialisation of CITID. Constructor priorites are displayed above each block.**

API function to make sure this function behaves properly. This constructor has the lowest (base) priority. Its priority is an arbitrary number, reduce the probability of clashing with other potential constructors in user code.

- **Thread creation**: each thread has its own creation constructor. Thread creation is only done once on first boot, and extracts the relevant task-shared memory region from the linker script. It then creates the thread internally in the same way InK does. The constructor priority is `base + thread_priority`.
- **Pointer structure population**: populates the pointer structures for each thread at first boot. The constructor priority is `base + 1000 + thread_priority`.
- **Runtime initialisation end**: sets the flag for the *is_initialized* API function. The constructor priority is `base + 1000 + 64`, ensuring that it executes last.

4.2.3 InK correctness violations

In addition to the features mentioned above, CITID implements solutions for some correctness violations in the implementation of InK. In order to discuss these solutions, we must first understand the underlying issues. A pseudocode for InK thread handling is shown in Algorithm 2. There are two points where InK correctness could fail, which are highlighted in red text.

The first point at which InK can become incorrect is the `return` statement on line 13. This yields control back to the scheduler, which tries to select the highest-priority task to execute. The issue arises when a lower-priority task activates a higher-priority task, and then the thread code yields to the scheduler. The scheduler will select the higher priority task without properly committing the results of the lower priority task. This problem is illustrated in Figure 4.3 and can lead to an incorrect system state, in this case a deadlock. To solve this problem, we introduce a new `TASK_PRECOMMIT` state, which combines the `TASK_RELEASE_EVENT` and `TASK_FINISHED` states. This way, it is no longer necessary to break out of the procedure early.

The second point occurs between setting the `Thread.NextTask` and setting the `State` to `TASK_FINISHED` (lines 11-12). If a power failure occurs between the execution of these lines, the `Thread.NextTask` will be set to the next task in the thread, but the `State` will not have changed. Thus, the next task will be executed before the previous task has committed its changes, potentially leaving the system in an incorrect state. This issue is further illustrated in Figure 4.4. The solution to this problem is to introduce a temporary variable

Algorithm 2 Pseudocode for InK thread handling. The `TickThread` procedure is called inside the scheduling loop. The `State` variable is persistent, so the program can resume at that state on reboot. Code that can lead to an inconsistent or corrupt system state on power failure is highlighted in red.

```
 1: State ← TASK_READY
 2:
 3: procedure TickThread(Thread)
 4:     if State == TASK_READY then
 5:         CopyBackupToActiveBuffer()
 6:         if IsEntryTask(Thread.NextTask) then
 7:             Event ← TakeEvent()
 8:             Thread.NextTask ← Thread.NextTask.Call(Event)
 9:             State ← TASK_RELEASE_EVENT
10:         else
11:             Thread.NextTask ← Thread.NextTask.Call(NULL)
12:             State ← TASK_FINISHED
13:             return
14:     if State == TASK_RELEASE_EVENT then
15:         ReleaseEvent()
16:         State ← TASK_FINISHED
17:     if State == TASK_FINISHED then
18:         BufferIndexTemp ← BufferIndex xor 1
19:         State ← TASK_COMMIT
20:     if State == TASK_COMMIT then
21:         BufferIndex ← BufferIndexTemp
22:         State ← TASK_READY
23:         return
```

`NextTaskTemp`. Now if a power failure occurs between the previously discussed lines, the `NextTask` will not have changed and the task is simply re-executed. `NextTask` is then actually set inside the new `TASK_PRECOMMIT` state, see Algorithm 3.

4.3 Instrumentation

The CITID source code instrumentation is implemented as a compiler plugin for `clang` version 20.1.4. Fundamentally, the instrumentation consists of threes stages; function labeling (Section 4.3.1), pointer instrumentation (Section 4.3.2) and variable instrumentation (Section 4.3.3). We also show how our implementation of code instrumentation improves on previous work; TICS [21] and ALPACA [25].

4.3.1 Function labeling

Function labeling is the first phase of instrumentation. Task functions are labeled, so they can be easily identified in the subsequent phases. The function labelling algorithm pseudocode is shown in Algorithm 4. First, the root task of the thread is identified. This is done by using an `annotate` attribute, which

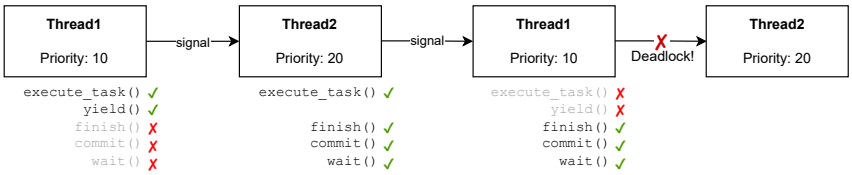

Figure 4.3: **This figure illustrates the problem with the early return in the InK implementation. A task from Thread1 is executed, and then yields early to the scheduler. But in the mean time, Thread2 has been activated. Instead of selecting Thread1 to schedule, the scheduler selects Thread2. This means that the finish and commit steps are skipped for the task in Thread1. Thread2 then tries to signal Thread1. But instead of executing, Thread1 finishes its commit and waits for Thread2's signal, which will never come. Both threads are waiting for each other: a deadlock.**

is embedded into the `CREATE_THREAD` macro (see Section 4.1). There may only be a single root task inside a translation unit; if there are multiple, the instrumentation program will give an error. Next, all return statements inside the task function are collected, represented by the `CollectReturnStatements()` function. For each return statement, the program first checks if the return expression refers to a function declaration, i.e. is a valid function pointer. If the function pointer is valid, the program checks if the original function declaration has static storage duration and is not in any header file. Only then the function is a valid task function. If we would not enforce this, it would be possible to use the task function externally, which could violate correctness. In short, task functions must be a) declared inside the same translation unit as the root task, b) declared with static storage duration and c) not declared in a header file. If a function returned from a task does not satisfy all these constraints, it is considered invalid and the instrumentation program returns an error. For every correct function pointer returned from the task, the same algorithm is applied to the function to which that pointer points. In essence, we have a recursive algorithm that builds a tree of task functions.

4.3.2 Pointer instrumentation

The pointer instrumentation is the second phase, and is separated into two sub-phases. In the first phase, pointer access is instrumented for dereference and array subscript operators on pointers inside task functions. An overview of how this is done is presented in Figure 4.5. Any pointer operations outside of task functions are ignored. When implementing pointer instrumentation, it is important to keep in mind that entire expressions may be dereferenced. As such, we must be careful not to evaluate that expression twice when instrumenting since that may lead to unintended side-effects. The second sub-phase deals with pointer dereference via member access, using the `->` operator. The sub-phases had to be separated because doing both in a single pass led to conflicts where the same pointer access was instrumented multiple times.

In addition to basic pointer instrumentation, pointer read and write detection

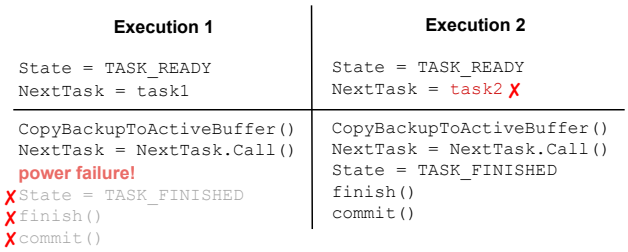

Figure 4.4: **This figure illustrates the issue with the processing of the next task in InK. In the first execution, `task1` is executed and the result (`task2`) is stored in `NextTask`. Then, a power failure occurs. On reboot, the TickThread procedure continues at the `TASK_READY` state, which re-executes the task. However, instead of `task1`, `task2` is executed. `task1` never had the chance to commit its results, so the `CopyBackupToActiveBuffer()` call of `task2` overwrites the results of `task1`.**

was added in anticipation of future work. This means that it is possible for pointer accesses to be instrumented differently based on it being a read or write. This could, for example, be used to detect WAR conflicts.

Pointer instrumentation instruments the pointers at compile time. But pointers must be processed at runtime, as they can point to any value. To do this, a function is called that returns `void*`. This is necessary because different pointers may have different types, and `void*` acts as a general type for all pointers. In order for the code to have the same behaviour and not introduce any warnings, we must cast this `void*` to the original pointer type. To do this, we can deduce the type of the pointer expression at compile time, and then cast the result of the runtime function to that type.

Finally, a limitation of the CITID implementation for pointer instrumentation is that pointers to anonymous structs or unions are not supported. We believe that this is acceptable, as the use cases for such pointers are extremely limited. They also require the memory to be dynamically allocated, which is often not desirable in small, critical embedded devices such as wireless sensors.

4.3.3 Variable instrumentation

The final instrumentation phase is variable instrumentation. Here, task-shared variable accesses are instrumented so the correct buffer is used at run time. To start, variables must be detected as being task-shared. There are a few conditions for this, namely a) the variable must not be marked with `CITID_IGNORE` or `CITID_PERSISTENT`, b) the variable must not be marked as `const`, c) the variable must not be marked as `volatile`, d) the variable must have static storage duration and e) the variable must not be declared in a header file. If a global or static local variable satisfies all these conditions, it is added to the pool of task-shared variables.

Once all candidate task-shared variables have been added to the pool, the processing of this pool occurs in multiple phases, which are shown in Figure

Algorithm 3 Pseudocode for corrected InK thread handling. The `TASK_RELEASE_EVENT` and `TASK_FINISHED` states have been combined into a `TASK_PRECOMMIT` state.

```
1:  State ← TASK_READY
2:
3:  procedure TickThread(Thread)
4:      if State == TASK_READY then
5:          CopyBackupToActiveBuffer()
6:          if IsEntryTask(Thread.NextTask) then
7:              Event ← TakeEvent()
8:              NextTaskTemp ← Thread.NextTask.Call(Event)
9:          else
10:             NextTaskTemp ← Thread.NextTask.Call(NULL)
11:         State ← TASK_PRECOMMIT
12:     if State == TASK_PRECOMMIT then
13:         ReleaseEvent()
14:         Thread.NextTask ← NextTaskTemp
15:         BufferIndexTemp ← BufferIndex xor 1
16:         State ← TASK_FINISHED
17:     if State == TASK_COMMIT then
18:         BufferIndex ← BufferIndexTemp
19:         State ← TASK_READY
20:         return
```

4.6. In phase (1), the usage pattern of all variables is analysed. If a variable is never written, it is discarded from the pool, since no WAR conflicts can occur for a variable that is read-only. Phase (2) moves all task-shared variables to the thread-assigned memory region. Phase (3) instruments all the variable uses with run-time code to determine which of the two buffer regions should be used to access them. Phase (4) is the last phase. This phase checks if the address of any non task-shared global variable is used to initialise a task-shared variable. If we do not do this, it could lead to inconsistencies, see Figure 4.7. The address of an uninstrumented variable could be used to initialise a task-shared pointer. If phase (4) was not done, the underlying pointer data would not have been marked as task-shared, but could still be shared between tasks. In addition, the underlying pointer data would not live in NVM, so it is not power-failure safe. Phase (4) makes sure all variables that are used inside the initialisation expression of a task-shared variable are marked as task-shared.

In the last part of the variable instrumentation, the pointer structure mentioned in Section 4.2.1 is inserted into the source code at the start of the file. In addition, the function that initialises these pointers is added to the end of the file. For static local variables, the pointer initialisation instrumentation is done in phase (2) of the task-shared variable pool.

Algorithm 4 Algorithm for function labeling in CITID code instrumentation.

```
1: TaskList ← ∅
2: procedure GetRootTask
3:     if AnnotatedTasks.length > 1 then
4:         Error()
5:     return AnnotatedTasks.first
6:
7: procedure CollectChildTasks(RootTask)
8:     ReturnStatements ← CollectReturnStatements()
9:     for Statement in ReturnStatements do
10:        if Statement is not function_pointer then
11:            Error()
12:        TaskFunction ← Statement.GetFunction
13:        if TaskFunction.StorageDuration is not static then
14:            Error()
15:        if not TaskFunction.IsInMainFile() then
16:            Error()
17:        TaskList ∪ TaskFunction
18:        CollectChildTasks(TaskFunction)
19:
20: procedure FunctionLabeling
21:     RootTask ← GetRootTask()
22:     CollectChildTasks(RootTask)
23:     for Task in TaskList do
24:         AddLabelToTask(Task)
```

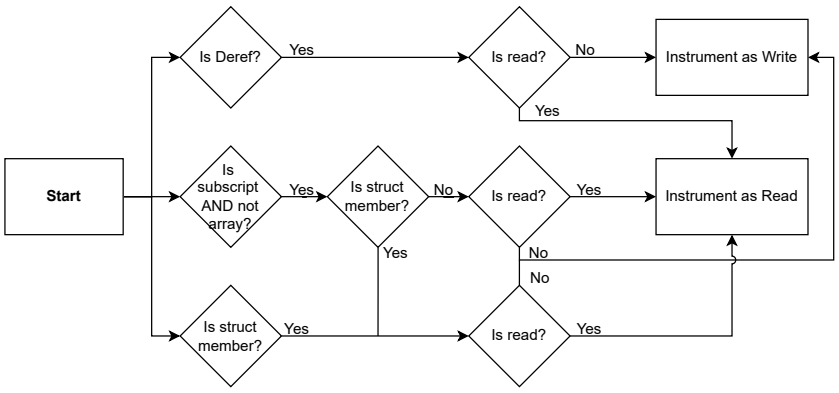

Figure 4.5: **Overview of how pointer instrumentation is implemented in CITID. There are three possible ways to dereference a pointer: dereference operator, array subscript and struct/union member access. For each type, the instrumentation determines if it is a read or write.**

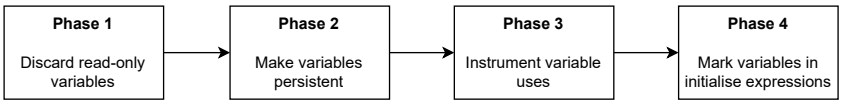

Figure 4.6: **Different phases of variable instrumentation in CITID.**

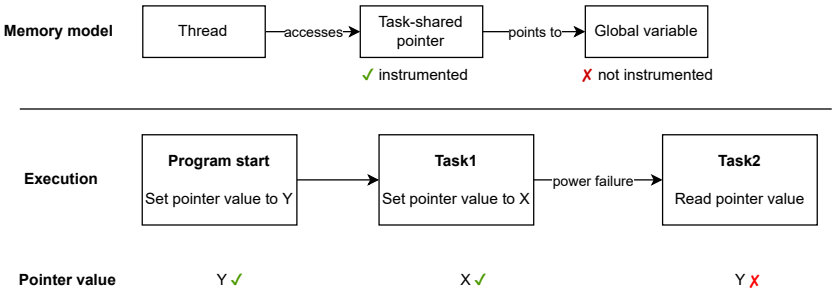

Figure 4.7: **If a global variable is pointed to by a pointer and is *not* transitively instrumented, it can lead to an inconsistent state when a power failure occurs. In this example, The global variable is set to `Y` at the start of the program, and then set to `X` by *Task1*. Then, a power failure occurs and the variable value is reset to `Y`, because the global variable has not been instrumented and is therefore not persistent.**

Chapter 5

Evaluation

In this chapter, we evaluate CITID on two main factors. The development effort is discussed in section 5.1. It is evaluated by training a LLM with both InK and CITID programming models. We then ask the LLM to generate code for both programming models. We evaluate the output of the LLM on correctness and the number of prompts necessary to obtain a correct output.

The next main factor is performance. As mentioned in 1, one of the main goals of this work is to maintain system performance while reducing development effort. To this end, we run multiple benchmarks on real hardware with InK and CITID. We evaluate the systems on execution time and overhead under various power conditions for all benchmarks. We observe that the additional overhead and increased execution time of CITID is not significant in most cases.

5.1 Development effort

5.1.1 Methodology

Development effort evaluation of CITID compared to InK is done in three ways. The first way is to compare the lines of code for each benchmark. Although this is not a great measurement of development effort, it may lead to valuable insights. The number of lines of code (#LOC) excludes comments, whitespace and include directives, and excludes code that only needs to be written once for a given application (the `main()` function for CITID, and the `__app_init` function declaration and full `__app_reboot` function for InK). All code was formatted with the same settings using `clang-format`, such that the the same standards are applied to all of the #LOC measurements.

The second way is to measure the amount of system-specific keywords for CITID compared to InK. This represents the number of times the programmer needs to deviate from standard C programming and thus needs to spend extra effort on remembering and applying the keywords. It also signifies the number of points of failure: for every keyword use, there is a chance that the developer may forget to use it, which introduces a bug into the code.

Lastly, CITID and InK are compared by instructing an LLM to instrument existing code to make it compliant with CITID or InK and to generate a simple program in CITID or InK. We chose to use a LLM for this task because these models are widely used today for code generation [11, 43]. If CITID is easier for

the model to instrument and create code with, this could be beneficial for LLM-aided programming, and CITID's instrumentation compiler could catch errors that the model makes in calling the system's runtime functions or macro's. This could be especially useful in further research into the automatic conversion of large, existing code bases into CITID-compliant code.

The results from the LLM are achieved by running experiments multiple times. First, the model was given a description of InK or CITID, the context it operates in and the API functions that can be used. The prompt is displayed in Appendix A, and it only differs in the API explanation, to make the comparison as fair as possible. Next, the instrumentation experiment, we check how many mistakes were made over five separate runs. The benchmarks are in the public repository [38]. Each benchmark builds upon the previous one, and introduces one more edge case for either InK, CITID or both. For the program generation, we check how many times the program was correct and which mistakes were made over five runs. The prompt for this generation can be found in Appendix A.

5.1.2 Experimental setup

The LLM used was Microsoft 365 Copilot. This model was chosen because it is used for general purpose tasks, and allows the use of a TU Delft enterprise account, where data is kept local and not sent to Microsoft.

5.1.3 Results

Keywords and lines of code

The relation between the #LOC, number of keywords, the benchmarks and CITID and InK is displayed in Table 5.1. We observe that the #LOC are similar between InK and CITID, but CITID consistently has fewer #LOC. This happens because InK threads require explicit initialisation, whereas CITID threads do not. Declaring shared-variables also increased the #LOC for InK, especially when persistent variables must be initialised only once on the first boot. To illustrate: in CITID, one can simply initialise the global variable as one would normally do. But in InK, this is not possible since the `__shared()` macro for declaring shared variables generates a struct. One cannot initialise struct variables inside this macro, so the initialisation has to be done on a new line, inside the thread initialisation function. This is the reason why `rsa` has comparatively fewer #LOC than the other benchmarks.

As can be seen in Table 5.1, the number of keywords for CITID is significantly reduced by using CITID. One reason for the reduction is that 82% to 96% of keywords used in the InK benchmarks are used for accessing variables or defining tasks, which are eliminated by the CITID instrumentation. If we subtract all of those keywords from the total for all InK benchmarks, we are indeed left with nearly the same number of keywords as in CITID. The keywords in CITID consist of creating threads (1), calling the scheduler (1), calling the entry task (1) and specifying persistent non-task-shared variables (1-3).

Table 5.1: **List of benchmarks used to evaluate InK and CITID, with corresponding number of keywords and lines of code (#LOC). The number of keywords is significantly lower when using CITID compared to InK, but the number of lines of code is only slightly lower.**

		InK		CITID	
Benchmark	**Source**	**#Keywords**	**#LOC**	**#Keywords**	**#LOC**
ar	[42]	136	565	5	556
bitcount	[42]	89	347	5	338
cem	[42]	120	301	4	293
cuckoo	[42]	100	331	4	322
dijkstra	[26]	54	157	5	150
fft	[26]	47	405	6	398
rsa	[26]	106	299	4	281
sort	[26]	29	154	5	147

LLM evaluation

Instrumentation (no examples): Failure rates for the edge-case benchmarks are shown in Table 5.2. In the first case, no example code is given to the LLM. We see that for both InK and CITID, the LLM makes mistakes in all cases. For CITID, the LLM consistently failed to correctly place the macro for creating a thread. In addition, it failed to use a macro for making a variable that was not used in tasks persistent. The maximum number of mistakes the LLM made for a given benchmark in CITID is 2, which happened 21 times. The total number of mistakes made was 71.

For InK, the LLM did not always signal the thread to start at first boot. It also did not apply the macro's for task definition and declaration correctly in all cases. In some cases, the LLM used macro's for accessing task-shared variables outside of tasks, which is illegal. Lastly, it failed to take into account the minimum required size of the task-shared variables and did not initialise the task-shared variables, if they were initialised. The maximum number of mistakes made in a benchmark for InK is 4, which happened 8 times. The total number of mistakes made was 119.

These results show that the number of benchmark failures is equal for CITID and Ink when not using any code examples. The number of mistakes made by the LLM when using CITID is lower than for InK. We believe that this is the case because there are simply fewer mistakes to make when using CITID compared to InK.

Instrumentation (with example code): When using the LLM with a code example (see Appendix A), the number of mistakes made and the failure rate drops dramatically for both InK and CITID (see Table 5.2). When the LLM used CITID, the `trivial`, `recursive` and `datatype` did not contain any mistakes. For InK, this is the case for `trivial` and `recursive` as well, but not for `datatype`, where the LLM again struggled with the minimum required size. For `extravar`, there were 9 failures for both InK and CITID. When using CITID, the LLM found it difficult to use the macro for making a variable persistent but not task-shared. However, it is surprising to note that the LLM found the application of this macro easier for the `init` benchmark, leading to fewer failures.

Table 5.2: **Failure rate for edge-case benchmarks for InK and CITID. This failure rate is based on 10 runs for each benchmark. The table is split into failure rate with no examples given, and failure rate with example code given.**

	No examples		With example	
Benchmark	**InK**	**CITID**	**InK**	**CITID**
trivial	10/10	10/10	0/10	0/10
recursive	10/10	10/10	0/10	0/10
datatype	10/10	10/10	7/10	0/10
extravar	10/10	10/10	9/10	9/10
init	10/10	10/10	10/10	4/10
Total	50/50	50/50	26/50	13/50

The failures for `init` when using InK are explained by the fact that the LLM did not correctly initialise the task-shared variables. In the end, CITID has 2x fewer failures than InK on 50 benchmark runs. The total number of mistakes of the LLM when using CITID is 15, and when using InK it is 37. As such, we can conclude that, given this experiment, very similar prompts and example code, it was easier for the LLM to instrument code using CITID than using InK.

Generation: The model was instructed to generate a simple sense, compute and transmit program. The complete prompt for this generation is given in Appendix A. When no example was given, the LLM failed to provide working code in all cases for CITID and InK. A total of 20 mistakes were made when using InK, compared to 19 for CITID. The number of mistakes made per run is also comparable. The types of mistakes are similar to the instrumentation results when not using example code; when using InK, the LLM fails to forward-declare task functions, and when using CITID, the LLM did not place the macro for creating a thread correctly.

When using the example code, the output of the LLM for both CITID and InK contained no errors over the 10 runs done. This is to be expected, as the example code is very similar to the ideal output code for the generation prompt. With the above results, we conclude that the LLM has a comparable mistake and failure rate for CITID and InK.

5.2 Performance results

5.2.1 Experimental setup

Hardware

For evaluating the systems, we must choose a processor with on-board NVM, since InK does not support external NVM [42]. We choose the MSP430FR5969 processor [36], which was also used in the original InK experiments. It is a platform with 64kB of available on-board FRAM NVM and a maximum clock speed of 16MHz. The clock speed was set to 1MHz in our experiments, to mirror the experiments done in the InK paper.

We ran the benchmarks on an MSP-EXP430FR5969 development board [16]. This board exposes several GPIO pins; some of these pins were used to measure

the execution time of a piece of code, by toggling said pin on and off. A signal analyser measures the signals coming from the pins and sends them to the attached PC for further processing and analysis.

Power failure traces were simulated using a Raspberry Pi Pico [12] attached to the `RST` (Reset) pin of the development board. The Pico was programmed to pull the `RST` low when the voltage on the power trace would fall below the minimum operating voltage of the processor. In the case of the MSP430FR5969, this threshold voltage is 0.7V when shutting down, and 0.79V when rebooting (see [36], Section 5.12.1).

Compiler and toolchain

CITID uses a customised linker script that cannot be used with Code Composer Studio (CCS), because CCS does not provide a way to provide linker symbols to allow the system to work. Therefore, we opted to use the MSP430-GCC compiler [15], version 9.3.1, instead. In our CMake build system, we used `CMAKE_BUILD_TYPE=Release` and `CMAKE_BUILD_TYPE=MinSizeRel` to compile InK, CITID and all benchmarks, in order to ensure a fair comparison.

Benchmarks

For evaluating CITID, we use multiple benchmarks and run them with both InK and CITID. These benchmarks are displayed in Table 5.1.

5.2.2 Methodology

Even if the development effort was reduced, we must also be sure that the performance of CITID is not significantly affected by its design. To evaluate this performance, we run three types of experiments:

- **Benchmark** experiments: we run each of the benchmarks in Table 5.1 five times for both InK and CITID (except for `dijkstra`, which is run four times). We note the execution times of the benchmark runs and average them. In addition, we measure the task execution time and task initialisation time for every benchmark run. The benchmark experiments are discussed in Section 5.2.3.

- **Overhead** experiments: we run one benchmark on both systems five times (except for dijkstra, which is ran four times), with optimisaton flags `-O3` and `-Os`. We then note the overhead that the system runtime has, compared to the benchmark execution time. Measured overheads include initial boot, reboot, scheduling, task commit and task activation. Task initialisation overhead is measured by running a separate application on the MSP430FR5969 MCU. This application uses different amounts of shared data to evaluate the task initialisation overhead: 10 bytes, 1 kB, and 10 kB. Lastly, we measure the executable sizes for all benchmarks in both systems, to account for memory overhead. The results of the overhead experiments are shown in Section 5.2.4.

- **Power failure** experiments: we run a selection of benchmarks on both systems using five of the power traces provided in [32]. All of these power

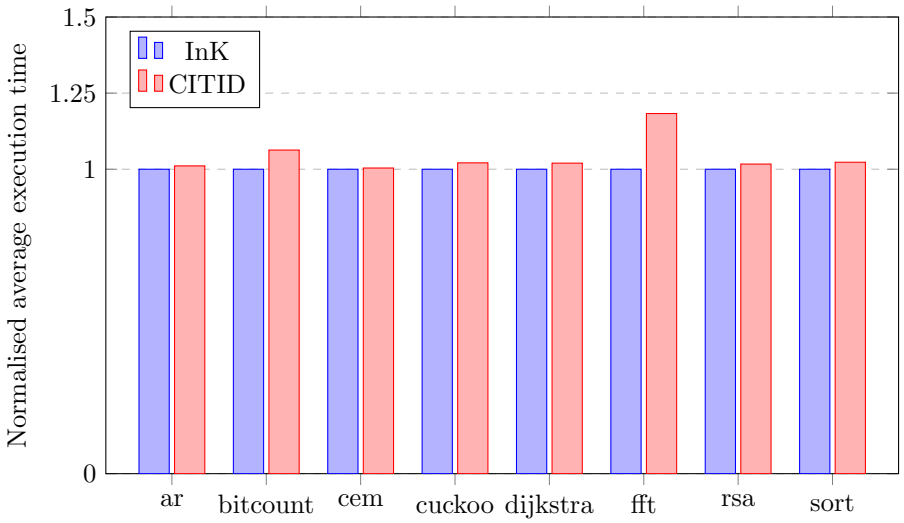

Figure 5.1: **Average benchmark execution time of CITID, normalised to the average benchmark execution time of InK, for each benchmark. The benchmark runs for this data were optimised with the `-O3` compiler flag. This figure contains no error bars, since the benchmarks are deterministic and each run deviated a negligible amount from the average.**

traces ran for 90 seconds, so that all of the power traces could run for at least one cycle. Power traces that lasted less than 90 seconds were repeated over the execution duration. All power traces ran at optimisation level `-Os` on benchmarks `dijkstra`, `rsa` and `fft`. These benchmarks were chosen because their execution times differed roughly by an order of magnitude, and so it is possible to find out if the benchmark execution times have an influence on power failure results. We measure the average benchmark time, average time that the MCU is on (on-time), and total reset latency over five runs for each trace. Section 5.2.5 discusses the power failure experiment results.

5.2.3 Benchmark results

This section discusses the results of the benchmark experiments, namely the difference in the average execution time between InK and CITID, for each benchmark. This metric is important because it pertains to one of the goals for this research; decreasing programmer burden while **maintaining performance**. The benchmarks were run according to the experiment described in Section 5.2.2. The `dijkstra` benchmark includes four runs instead of five, because this benchmark has a period of four runs where the results become the same again (all other benchmarks have a period of one run). In addition, the benchmark experiments were done using the `-Os` and `-O3` flags in different runs, so different optimisations can be compared.

Figure 5.1 displays the results for the normalised average benchmark execution time of CITID compared to InK using optimisation level `-O3`. For all

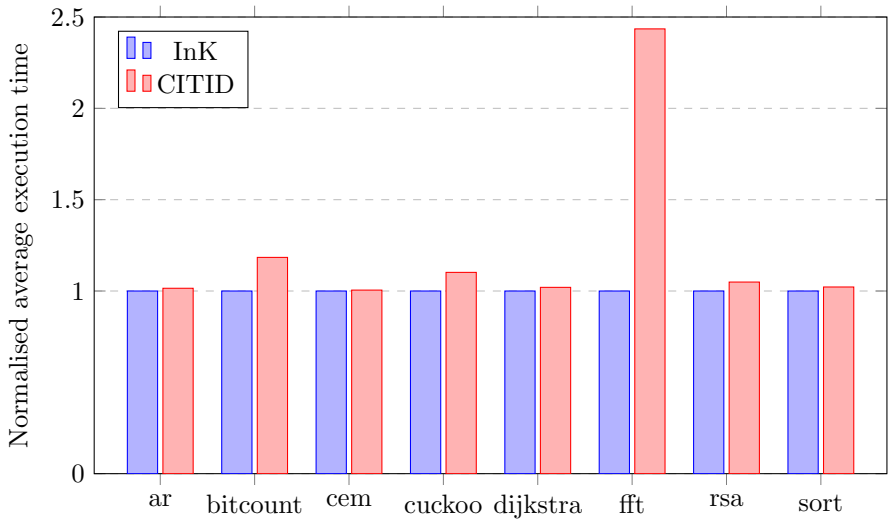

Figure 5.2: **Average benchmark execution time of CITID, normalised to the average benchmark execution time of InK, for each benchmark. The benchmark runs for this data were optimised with the `-Os` compiler flag. This figure contains no error bars, since the benchmarks are deterministic and each run deviated a negligible amount from the average.**

benchmarks except `bitcount` and `fft`, we can see that the difference in average execution time is not significant (at most a 2.1% difference for the `cuckoo` benchmark). The `bitcount` and `fft` show a higher difference between the average benchmark execution time of CITID and InK, they are 6.3% and 18.3% slower respectively.

The results of the benchmark execution time measurements with optimisation level `-Os` are shown in Figure 5.2, to compare with the results for `-Os` in Figure 5.2. We observe that for the benchmarks `ar`, `cem`, `dijkstra`, `rsa` and `sort`, the average execution time is a little less, but not significantly so. The interesting results come from benchmarks `bitcount` (18.4% slower), `cuckoo` (10.2% slower) and `fft` (143.5% slower). The difference for `fft` between `-Os` and `-O3` is especially stark; when using `-Os`, the difference in execution time between InK and CITID is approximately 6.8 times higher.

The differences in average benchmark execution time between InK and CITID for both optimisaton levels can be partially explained by looking at Table 5.3. This table displays the percentage of the difference that occurred due to a difference in task execution time. The results in this table show that the vast majority of the difference in average benchmark execution time occurs due to the average difference in task execution time, for all benchmarks on both optimisation levels. This does not explain, however, where exactly this difference in task execution time comes from.

To explain the difference, we must examine the assembly code emitted by the MSP430-GCC compiler. Listings 5.1 and 5.1 show the assembly code for the `bitcount` benchmark that is generated when a task-shared variable is set

Table 5.3: **The average difference in benchmark execution time between InK and CITID due to task execution, in %. We calculate this using the following method: 1) multiply the average task execution time difference between InK and CITID of a benchmark with the number of tasks executed to get the total average task execution time difference. 2) Divide the number obtained from step 1 by the average difference in benchmark execution time. This yields the fraction of the benchmark execution time difference that is spent on executing tasks.**

	Average difference in benchmark execution time due to task execution (%)	
Benchmark	**-Os**	**-O3**
ar	99%	100%
bitcount	102%	107%
cem	87%	100%
cuckoo	99%	105%
dijkstra	96%	97%
fft	103%	101%
rsa	101%	99%
sort	89%	90%

when using the `-Os` optimisation level. Listing 5.1 shows that setting a task-shared variable in CITID takes 12 instructions, compared to just one instruction in InK. The compiler does the address calculation based on the current buffer index for every variable set operation. This happens because the current buffer index is moved into `r12`, and this overwrites the address of the previous task-shared variable. As such, the address of the new variable must be computed again. This happens in most cases where a task-shared variable is set or read. When using pointers, this increase in instructions is even greater, as in addition it needs to call the function that determines whether the pointer falls into the task-shared memory regions or not. In contrast, the compiler never re-computes the address when using the `-O3` optimisation level. This is why the `fft` and `bitcount` benchmarks execute much slower on `-Os` than on `-O3`.

The increased number of instructions also accounts for the fact that CITID is slower than InK for the `fft` and `bitcount` benchmarks on `-O3`. Since the struct in the implementation of CITID contains *pointers* to the task-shared data, and not the data itself, we must use an additional instruction if we want to access the task-shared data. If this happens in hot loops, as is the case in the `fft` benchmark, the extra instructions can compound into an increase in execution time. If a task with task-shared variable accesses is executed many times, as is the case with `bitcount` and `cuckoo`, the same effect occurs. These two benchmarks also have a relatively quick execution time, around 600ms for and 500ms respectively. Because of this, the extra instructions have a bigger effect than they have on the execution times of the other benchmarks, which already take longer to execute.

```
; Move current buffer index to r12
mov     r10,        r12
; Set r13 to the size of the struct
mov.b   #22,        r13
; Call hardware multiplier
call  #29136
    push    r2
    dint
    nop
    mov     r12,        &0x04c0
    mov     r13,        &0x04c8
    mov     &0x04ca,    r12
    reti
; Get variable address
mov     17680(r12), r12
; Store value inside variable
mov.b   #1,         0(r12)
```

Listing 5.1: **Assembly code for setting a task-shared variable in the `bitcount` benchmark in the CITID system with optimisation level `-Os`.**

```
; Store value inside variable. The address in r12 is
; passed to the task function and stays in r12.
mov.b   #1,       0(r12)
```

Listing 5.2: **Assembly code for setting a task-shared variable in the `bitcount` benchmark in the InK system with optimisation level `-Os`.**

5.2.4 Overhead results

Execution time overhead

Tables 5.4 and 5.5 show the execution time overhead for CITID compared to InK. All times were measured using a signal analyser and averaged over 5 runs. We note that the difference between all measured times is insignificant, except for the reboot case. In this case, CITID spends about 100μs or about 7% more time rebooting. This can be explained by the constructor functions running on every reboot for CITID. In the measurements, there are three such functions; one for creating a thread, one for initialising the runtime and one for signalling the end of initialisation. The function for initialising the runtime is not relevant, as this function also runs when using InK. The other two functions, however, increase the reboot time. The functions for creating a thread each individually check whether the system is in its first boot or not, while in InK this is only checked once. This means that the reboot time scales with the number of threads; the more threads, the slower the reboot. This is also the case for InK, but the increase for CITID is more significant.

Memory overhead

Table 5.6 shows the binary size overhead of CITID compared to InK. The overhead for `dijkstra`, `fft`, `rsa` and `sort` is small, and decreases when using `-Os`. The overhead for the other benchmarks is larger, from 9% for `bitcount` to almost 11% for `ar`. This is due to two factors. Firstly, there are the extra instructions required for CITID when accessing a task-shared variable or a pointer

Table 5.4: **InK and CITID overhead for various scenario's when compiled with optimisation level `-O3`. The differences between InK and CITID are mostly insignificant, except for the reboot overhead.**

	Overhead (μs)		
Overhead type (`-O3`)	**InK**	**CITID**	**Difference (%)**
Initial boot	89262	89726	0.52%
Reboot	1470	1571	6.84%
Scheduling & thread selection	63	63	0.00%
Task initialisation (10B)	127	130	1.75%
Task initialisation (1kiB)	1746	1749	0.17%
Task initialisation (10kiB)	16460	16462	0.01%
Task commit	94	94	0.00%
Task activation	11	11	0.00%

Table 5.5: **InK and CITID overhead for various scenario's when compiled with optimisation level `-Os`. The differences between InK and CITID are mostly insignificant, except for the reboot overhead.**

	Overhead (μs)		
Overhead type (`-Os`)	**InK**	**sysname**	**Difference (%)**
Initial boot	91124	91403	0.31%
Reboot	1471	1575	7.07%
Scheduling & thread selection	94	94	0.00%
Task initialisation (10B)	130	128	-1.54%
Task initialisation (1kiB)	1748	1750	0.11%
Task initialisation (10kiB)	16468	16468	0.00%
Task commit	88	88	0.00%
Task activation	13	13	0.00%

(see Section 5.2.3). Secondly, the additional API functions take up quite some space, especially the `__ink_create_thread` function, which contains a sizable switch-case statement.

5.2.5 Power failure results

For the power failure experiments, we used five different power traces. These traces were selected to provide the most diverse fractions of on and off time. Execution time results are shown in Figure 5.3. If we compare this to Figure 5.2, the results are very similar. In fact, the difference between these results is at most 5.5%. We attribute the relatively small difference to two factors. The first factor is the reset time. The average reboot latency over all benchmarks is only 0.07% of the total on-time. Thus, the reboots have an extremely small effect on the execution time. This is expected, since the reboot latency difference between InK and CITID is only about 100μs. The second and most significant factor is the arbitrary start time of the measurement. If the measurement always starts at the same time, we would always get the same results. However, the start time is not always the same. This leads to some random fluctuations, which influence the final outcome. Because of these two factors, we can conclude that

Table 5.6: **Memory overhead of CITID when compiling various benchmarks with optimisation level `-O3` or `-Os`. The first four benchmarks show a larger increase in the memory overhead, compared to the last four benchmarks.**

	Memory overhead of CITID	
Benchmark	`-O3`	`-Os`
ar	10.87%	10.38%
bitcount	8.99%	8.30%
cem	9.27%	8.57%
cuckoo	10.31%	9.42%
dijkstra	3.69%	2.99%
fft	2.60%	2.07%
rsa	4.16%	3.54%
sort	2.61%	1.89%

CITID does not have a significant performance penalty under a realistic power failure trace compared to InK.

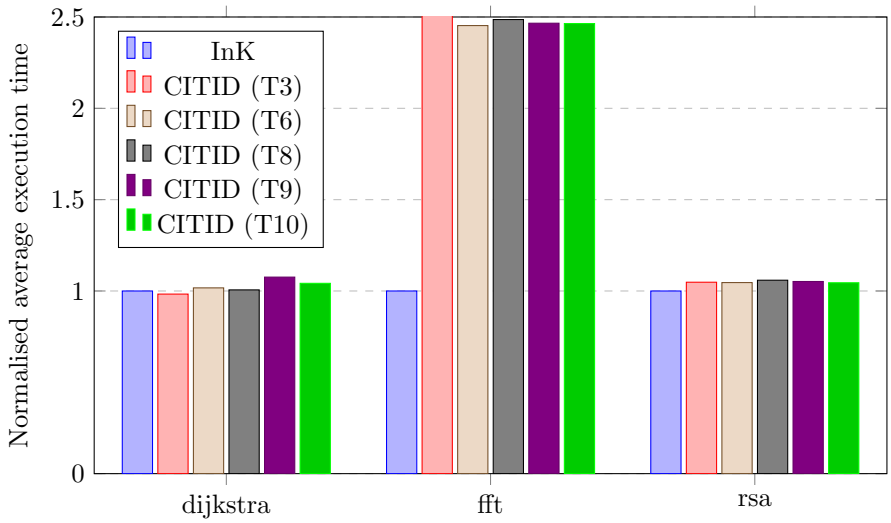

Figure 5.3: **Exeution times of InK without power failure and CITID with power failure. Five different power traces (T3, T6, T8, T9, T10) are used to simulate power failures in CITID. The figure shows that CITID performs similarly for all power traces.**

Chapter 6

Related work

This chapter compares CITID to some open source state-of the art ImC programming models, including Alpaca [25], TICS [21] and InK [42].

6.1 Comparison to Alpaca and TICS

This section compares CITID with three open source state-of-the-art ImC programming models. The first model is Alpaca [25]. This model uses a combination of run time and compile time features to allow for task-based ImC development. Alpaca's compile time instrumentation is implemented as a compiler pass for the clang compiler. On the one hand, this means it has more detailed information available about the source code, which allows it to make more intricate decisions for what to instrument. On the other hand, it means that all of the code must be compiled using clang, which is not always desirable; it is not compiler-agnostic. In comparison, CITID implements its instrumentation as a clang *plugin*. This allows instrumentation to be treated as an entirely separate process from compilation, meaning different compilers can be used. Also, CITID automatically constructs a tree of task functions for each thread, whereas Alpaca requires the use of prefixes in task function names. This gives the programmer a bit more flexibility when using CITID, and reduces the probability of getting errors.

The second programming model is TICS [21]. This model mainly uses compile-time code instrumentation to insert task boundaries and process variable reads and writes. Like CITID, TICS is implemented as a clang plugin, but its instrumentation capabilities are less extensive. Table 6.1 illustrates where TICS fails while CITID succeeds. TICS has numerous edge cases that are not covered, especially when instrumenting pointers. CITID improves on this by correctly instrumenting these cases where TICS does not.

6.2 Comparison to InK

The final open source ImC programming model that we look at is InK [42]. As mentioned in Section 2.2, InK uses *threads* to group multiple tasks together. Each thread has a collection of persistent *task-shared* data that are shared between the tasks of that thread. All task-shared data in a thread are backed

Table 6.1: **Comparison of where TICS and CITID instrumentation succeed. [1] [3] Some pointer or array expressions are not instrumented, in most cases this happens when compound assignment or pre/post increment/decrement is used. [2] TICS does not instrument nested structs or unions beyond a single level of nesting. [4] TICS does not correctly instrument global variables when they are used in macro's.**

Instrumentation type	**TICS**	**CITID**
Pointers	$\sim^{1}$	✓
Structs/unions	$\sim^{2}$	✓
Arrays	$\sim^{3}$	✓
Global variables	$\sim^{4}$	✓

up when the executing task changes. This allows some granularity, since only data of one thread is backed up at the same time.

In addition to persistent data, threads have a priority that is used by InK to schedule the threads accordingly. Higher priority threads can pre-empt lower priority threads at task level. This means that, when a task finishes executing, InK checks if there is a higher priority thread waiting to execute. If there is, the current thread is paused and the firsts task of the higher priority thread runs.

A unique aspect of InK is that it incorporates interrupts into its design, reducing the time needed to respond to external events. When an interrupt happens, the current task execution is interrupted. Inside the interrupt, the event that happened is stored in NVM, and the task that must respond to this interrupt is *signalled* to wake up. The current task finishes executing, and then the task that responds to the interrupt is scheduled by the scheduler (if it has higher priority).

InK threads can also talk to each other via *channels*. These channels contain data and a timestamp, but they are not robust against power failures by themselves. The programmer must ensure that the data in channels is not corrupted when a power failure occurs.

All operations mentioned above are done at runtime. On the one hand, this is good because it requires no extra software infrastructure to compile the program, and InK works as-is. On the other hand, it means the programmer must manually insert all calls to the InK programming model, which is a tedious and error-prone task. CITID improves on InK by adding compile-time instrumentation, which removes up to 96% of manually written keywords. This removes the tedious task of instrumenting much of the code, and reduces the chance of programmer-induced errors.

Chapter 7

Conclusions

The goal of this work is finding a way to reduce programmer burden for development in an intermittent computing environment, while not significantly altering runtime performance. The results in Section 5.2 show that we achieved the goal of preserving performance for most of the tested benchmarks. In addition, the evaluation in Section 5.1 shows that using CITID leads to fewer errors made by the LLM, which indicates a reduction in required programmer effort.

7.1 Limitations and future work

Despite the contributions of this work, there are some limitations that can be improved by future work, both experimentally and design-wise.

7.1.1 Design limitations and improvements

As shown in Section 5.2.3, runtime execution time is changed significantly by CITID compared to InK for some benchmarks. To reduce the impact of CITID on the runtime performance of the system, some improvements can be made to the design. One improvement, as mentioned in Section 3.4, is the direct usage of task-shared variables instead of accessing them through pointers. This could improve performance, and bring it more in line with InK. In addition to this improvement, there are a few other optimizations for the runtime, which apply to both CITID and InK. First, it is possible to only back up task-shared memory if there is at least one WAR violation during the task execution, or between task executions. Secondly, it may be beneficial to integrate parallel state backup into InK as described in [44]. This could further reduce execution time.

In addition to performance improvements, there are ways to improve the flexibility of current CITID system. For example, CITID currently assumes an on-chip NVM. However, not all processors possess this feature. As such, supporting external off-chip NVM would allow CITID to be used on any processor that supports an external memory chip. Also, a major part of embedded systems applications are written in C++ [18], while CITID only works with C. Adding C++ support would increase the number of applications InK can be used in.

Furthermore, there is still one major burden bestowed on the programmer:

dividing the program into tasks. Some progress has already been made on automatically dividing a program into tasks [9, 40]. One of these solutions could be integrated with CITID, so the programmer does not need to manually divide their application into tasks. This could be especially useful for larger codebases, which would require more manual labour.

Lastly, the programmer has no control over the CITID scheduler beyond specifying the priorities of each thread. In order to facilitate more elaborate thread scheduling, such as in [26], the API could be extended to add the ability for arbitrary user thread data. This data could then be used in custom scheduling rules to schedule the tasks.

7.1.2 Experimental limitations and improvements

In our evaluation of development effort using LLMs (see Section 5.1), only a single Artificial Intelligence (AI) model is used. It may be beneficial to use multiple different models, to see if the results still hold. Another limitation is that the number and type of experiments are limited. Experiments could be done with more repetitions to increase data reliability. In addition, it could be interesting to explore how the LLMs respond to correcting mistakes in InK or CITID code, and if there is any difference in the results between InK and CITID.

Bibliography

[1] Michael Abner, Peter Kok-Yiu Wong, and Jack C.P. Cheng. Battery lifespan enhancement strategies for edge computing-enabled wireless bluetooth mesh sensor network for structural health monitoring. *Automation in Construction*, 140:104355, 2022. URL: https://www.sciencedirect.com/science/article/pii/S092658052200228X, doi:10.1016/j.autcon.2022.104355.

[2] Th. Arampatzis, J. Lygeros, and S. Manesis. A survey of applications of wireless sensors and wireless sensor networks. In *Proceedings of the 2005 IEEE International Symposium on, Mediterrean Conference on Control and Automation Intelligent Control, 2005.*, pages 719–724, 2005. doi:10.1109/.2005.1467103.

[3] Rani Baghezza, Kévin Bouchard, Abdenour Bouzouane, and Charles Gouin-Vallerand. From offline to real-time distributed activity recognition in wireless sensor networks for healthcare: A review. *Sensors*, 21(8), 2021. URL: https://www.mdpi.com/1424-8220/21/8/2786, doi:10.3390/s21082786.

[4] Abu Bakar, Alexander G. Ross, Kasim Sinan Yildirim, and Josiah Hester. Rehash: A flexible, developer focused, heuristic adaptation platform for intermittently powered computing. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 5(3), September 2021. doi:10.1145/3478077.

[5] Mukesh Bathre and Pradipta Kumar Das. Smart dual battery management system for expanding lifespan of wireless sensor node. *International Journal of Communication Systems*, 36(3):e5389, 2023. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/dac.5389, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/dac.5389, doi:10.1002/dac.5389.

[6] C standard committee. ISO/IEC 9899:202x. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf, 2018. Last accessed: Jul. 16, 2025.

[7] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. Compiler-directed high-performance intermittent computation with power failure immunity. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 40–54, 2022. doi:10.1109/RTAS54340.2022.00012.

[8] Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. *SIGPLAN Not.*, 51(10):514–530, October 2016. doi:10.1145/3022671.2983995.

[9] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*, CC '18, page 116–127, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3178372.3179525.

[10] Harsh Doshi and Achyut Shankar. Wireless sensor network application for iot-based healthcare system. In T. P. Singh, Ravi Tomar, Tanupriya Choudhury, Thinagaran Perumal, and Hussain Falih Mahdi, editors, *Data Driven Approach Towards Disruptive Technologies*, pages 287–307, Singapore, 2021. Springer Singapore.

[11] Yunhe Feng, Sreecharan Vanam, Manasa Cherukupally, Weijian Zheng, Meikang Qiu, and Haihua Chen. Investigating code generation performance of chatgpt with crowdsourcing social data. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 876–885, 2023. doi:10.1109/COMPSAC57700.2023.00117.

[12] Raspberry Pi Foundation. Raspberry Pi Pico. https://www.raspberrypi.com/products/raspberry-pi-pico/. Last accessed: Sep. 9, 2025.

[13] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3131672.3131673.

[14] Victoria J. Hodge, Simon O'Keefe, Michael Weeks, and Anthony Moulds. Wireless sensor networks for condition monitoring in the railway industry: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 16(3):1088–1106, 2015. doi:10.1109/TITS.2014.2366512.

[15] Texas Instruments. MSP430-GCC-OPENSOURCE: GCC - Open Source Compiler for MSP Microcontrollers. https://www.ti.com/tool/MSP430-GCC-OPENSOURCE. Last accessed: Sep. 9, 2025.

[16] Texas Instruments. MSP430FR5969 LaunchPad™ development kit. https://www.ti.com/tool/MSP-EXP430FR5969. Last accessed: Sep. 9, 2025.

[17] Muhammad Ali Jamshed, Kamran Ali, Qammer H. Abbasi, Muhammad Ali Imran, and Masood Ur-Rehman. Challenges, applications, and future of wireless sensors in internet of things: A review. *IEEE Sensors Journal*, 22(6):5482–5494, 2022. doi:10.1109/JSEN.2022.3148128.

[18] JetBrains. Embedded - The state of Devleoper Ecosystem 2021. https://www.jetbrains.com/lp/devecosystem-2021/embedded/, 2021. Last accessed: Sep. 9, 2025.

[19] Youngbin Kim, Yoojin Lim, and Chaedeok Lim. Lact: Liveness-aware checkpointing to reduce checkpoint overheads in intermittent systems. *Journal of Systems Architecture*, 153:103213, 2024. URL: https://www.sciencedirect.com/science/article/pii/S1383762124001504, doi:10.1016/j.sysarc.2024.103213.

[20] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. Wario: efficient code generation for intermittent computing. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 777–791, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523454.

[21] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 85–99, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378476.

[22] Songran Liu, Wei Zhang, Mingsong Lv, Qiulin Chen, and Nan Guan. Latics: A low-overhead adaptive task-based intermittent computing system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3711–3723, 2020. doi:10.1109/TCAD.2020.3012214.

[23] Jaime Lloret, Sandra Sendra, Laura Garcia, and Jose M. Jimenez. A wireless sensor network deployment for soil moisture monitoring in precision agriculture. *Sensors*, 21(21), 2021. URL: https://www.mdpi.com/1424-8220/21/21/7243, doi:10.3390/s21217243.

[24] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent Computing: Challenges and Opportunities. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SNAPL.2017.8, doi:10.4230/LIPIcs.SNAPL.2017.8.

[25] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133920.

[26] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Trans. Sen. Netw.*, 16(1), February 2020. doi:10.1145/3360285.

[27] Mamoona Majid, Shaista Habib, Abdul Rehman Javed, Muhammad Rizwan, Gautam Srivastava, Thippa Reddy Gadekallu, and Jerry Chun-Wei Lin. Applications of wireless sensor networks and internet of things frameworks in the industry revolution 4.0: A systematic literature review.

Sensors, 22(6), 2022. URL: https://www.mdpi.com/1424-8220/22/6/2087, doi:10.3390/s22062087.

[28] Elda M. Melchor-Martínez, Rodrigo Macias-Garbett, Alonso Malacara-Becerra, Hafiz M.N. Iqbal, Juan Eduardo Sosa-Hernández, and Roberto Parra-Saldívar. Environmental impact of emerging contaminants from battery waste: A mini review. *Case Studies in Chemical and Environmental Engineering*, 3:100104, 2021. URL: https://www.sciencedirect.com/science/article/pii/S2666016421000268, doi:10.1016/j.cscee.2021.100104.

[29] Sourav Mohapatra, Vito Kortbeek, Marco Antonio van Eerden, Jochem Broekhoff, Saad Ahmed, and Przemysław Pawełczak. *Data Cache for Intermittent Computing Systems with Non-Volatile Main Memory*, page 227–243. Association for Computing Machinery, New York, NY, USA, 2025. URL: https://doi.org/10.1145/3676641.3715989.

[30] Rita de Fátima Muniz and Antônio Clécio Thomaz. Revolutionizing industrial automation with wireless sensor networks. *Big Data and Computing Visions*, 3(3):76–83, 2023. URL: https://www.bidacv.com/article_190167.html, arXiv:https://www.bidacv.com/article_190167_8c6e2996d0b385689a8000b5fcf815c1.pdf, doi:10.22105/bdcv.2023.190167.

[31] S. J. Suji Prasad, M. Thangatamilan, M. Suresh, Hitesh Panchal, Christober Asir Rajan, C. Sagana, B. Gunapriya, Aditi Sharma, Tusharkumar Panchal, and Kishor Kumar Sadasivuni. An efficient lora-based smart agriculture management and monitoring system using wireless sensor networks. *International Journal of Ambient Energy*, 43(1):5447–5450, 2022. doi:10.1080/01430750.2021.1953591.

[32] Benjamin Ransford, Jacob Sorber, and Kevin Fu. MementOS power traces. https://github.com/ransford/mspsim/tree/mementos/traces, 2011. Last accessed: Sep. 9, 2025.

[33] Teodora Sanislav, George Dan Mois, Sherali Zeadally, and Silviu Corneliu Folea. Energy harvesting techniques for internet of things (iot). *IEEE Access*, 9:39530–39549, 2021. doi:10.1109/ACCESS.2021.3064066.

[34] Satyajit Sinha. State of IoT 2024: Number of connected IoT devices growing 13% to 18.8 billion globally. https://iot-analytics.com/number-connected-iot-devices/, 2024. Last accessed: Sep. 1, 2025.

[35] Aristotelis C. Tagarakis, Dimitrios Kateris, Remigio Berruto, and Dionysis Bochtis. Low-cost wireless sensing system for precision agriculture applications in orchards. *Applied Sciences*, 11(13), 2021. URL: https://www.mdpi.com/2076-3417/11/13/5858, doi:10.3390/app11135858.

[36] Texas Instruments. TI MSP430FR596x Datasheet. https://www.ti.com/lit/gpn/msp430fr5969, 2018. Last accessed: Jul. 9, 2025.

[37] Sumanth Umesh and Sparsh Mittal. A survey of techniques for intermittent computing. *Journal of Systems Architecture*, 112:101859,

2021. URL: https://www.sciencedirect.com/science/article/pii/S1383762120301430, doi:10.1016/j.sysarc.2020.101859.

[38] Marco van Eerden. CITID repository. https://github.com/mavaneerden/low-effort-task-based-intermittent-computing, 2021. Last accessed: Oct. 14, 2025.

[39] Hu Wan, Youyou Lu, Yuanchao Xu, and Jiwu Shu. Empirical study of redo and undo logging in persistent memory. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, 2016. doi:10.1109/NVMSA.2016.7547178.

[40] Shuo Xu, Wei Zhang, Mengying Zhao, Zimeng Zhou, and Lei Ju. Cache-aware task decomposition for efficient intermittent computing systems. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3649329.3657382.

[41] Bahram Yarahmadi and Erven Rohou. So far so good: Self-adaptive dynamic checkpointing for intermittent computation based on self-modifying code. In *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '21, page 29–34, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3493229.3493300.

[42] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, page 41–53, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3274783.3274837.

[43] Maxwell Zeff. GitHub Copilot crosses 20M all-time users. https://techcrunch.com/2025/07/30/github-copilot-crosses-20-million-all-time-users/, 2025. Last accessed: Sep. 9, 2025.

[44] Wei Zhang, Qianling Zhang, Mingsong Lv, Songran Liu, Zimeng Zhou, Qiulin Chen, Nan Guan, and Lei Ju. Adaptive task-based intermittent computing system with parallel state backup. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(6):1798–1809, 2023. doi:10.1109/TCAD.2022.3213989.

Appendix A

LLM evaluation prompts

A.1 CITID

A.1.1 Prompt for description without examples

The following paragraphs describe 'CITID', an API and runtime framework. Later on, you will use this description of CITID to transform C code into CITID-compliant code. So remember the following description for later. 'CITID' is a runtime and API meant for programming task-based intermittent computers. Intermittent computers sometimes run out of power and shut down. CITID allows the computer to continue where it shut off, by periodically backing up the state of the computer into Non-Volatile Memory (NVM). NVM does not lose data when powered off, which allows the state of the computer to be saved until the next power on. In order to periodically back up the state, the CITID runtime needs to know when this can be done. To make this clear, the program is split into tasks by the programmer. These tasks take the form of C functions. Each task does computations as described inside the function body, and then returns a pointer to the next task to be executed. If there is no next task, the return pointer is NULL. After a task is done executing, CITID will back up the system state. The CITID runtime will then select the next task to execute. The system state is represented by some global variables in C. These variables are shared between tasks within a thread. When a variable is part of the system state, it is called a 'task-shared' variable. The task-shared variables can only be used inside tasks; using them outside tasks is illegal. A unique thing about CITID is that it uses threads, which are distinct from tasks. Each thread contains multiple tasks, and there can be a maximum of 64 threads in a program. Threads are explicitly created by the programmer using an API call (described below). A thread can be either 'active' or 'dormant'. If a thread is active, its tasks can be executed by the CITID runtime. If a thread is dormant, the CITID runtime needs to wait for an event. Events are signalled by the programmer inside interrupts and are stored internally inside a persistent queue. There is one queue for every thread. If the event is triggered, the thread goes from the 'dormant' state to the 'active' state; the thread is 'activated'. When this thread is selected, the event that triggered the state change is passed to the first task in the thread; this task is called the 'entry task'. The CITID runtime uses priority scheduling to select which thread to execute. As such, each thread has

a specified priority, which is a number from 1 up to and including 64. Different threads cannot have the same priority. This limits the maximum number of threads to 64. Next to the CITID runtime, there is the CITID compiler. Task-shared variables are detected automatically by the compiler, by checking if a variable with global storage is used inside one or more task functions. If it is, it is marked as task-shared; no instrumentation is required by the programmer. If the variable does not have global storage, or is not used inside any task function, it is ignored by the CITID compiler, and marked as non-task-shared and not in NVM. To facilitate the backing up of state into NVM in this task-based programming environment, the CITID runtime requires hints from the programmer. These hints take the form of API functions. The following bullet list describes the API functions and instructions on where and when to use them. - API functions for interfacing. After system initialisation, the CITID runtime needs to be called using the 'citid_scheduler_run()' function. This can be done in the 'main' function of the program. - API functions for creating threads. A thread can be created using the 'CITID_CREATE_THREAD(prio, activate_on_first_boot)' macro. The 'prio' parameter is a number representing the priority of the thread. The 'activate_on_first_boot' parameter is a boolean value. If set to true, the created thread is activated on first boot. If false, the created thread remains dormant until woken up by an event. In addition, the 'CITID_CREATE_THREAD' macro defines the thread's entry task. The function pointer to the entry task created by the 'CITID_CREATE_THREAD' macro can be accessed using the 'CITID_THREAD_ENTRY_TASK' macro. The 'CITID_THREAD_ENTRY_TASK' macro takes no arguments; the thread priority is automatically deduced. - API functions for directing the compiler. To explicitly exclude a variable from being flagged as task-shared by the compiler, you can place the 'CITID_IGNORE' macro before the variable definition. This also does not store the variable into NVM. To exclude variables from being flagged as task-shared by the compiler, **but** still store them in NVM, you can place the 'CITID_PERSISTENT' macro before the variable definition. The 'CITID_IGNORE' and 'CITID_PERSISTENT' macro's override the default compiler behaviour. - API functions for events. To signal an event, the programmer can use 'citid_signal_event_isr(prio, event)'. This function should only be called inside interrupt service routines! The 'prio' parameter is the priority of the thread that the programmer wants to activate using the event. The 'event' parameter is a pointer to the event of type 'citid_event_t', used to activate the thread. The struct 'citid_event_t' has the following instance variables: 1) 'void *data', a pointer to the data associated with the event. This data should be allocated inside NVM to make sure it is persistent across power failures; 2) 'uint32_t size', the size of the data buffer in bytes; 3) 'uint32_t timestamp', timestamp at which the event took place. Since we do not use the timer functionality of InK, this remains unused and can be set to 0. When a thread is activated by an event, the pointer to that event can be accessed using the 'citid_event' variable. This variable can only be accessed inside the entry task! - API functions for activating threads. To activate a thread from another thread, the programmer can call the 'citid_activate_thread(prio)' function. Here, 'prio' is a number representing the priority of the thread we want to activate.

The API header file is 'citid.h'.

A.1.2 Prompt for description with some examples

The following paragraphs describe 'CITID', an API and runtime framework. Later on, you will use this description of CITID to transform C code into CITID-compliant code. So remember the following description for later. 'CITID' is a runtime and API meant for programming task-based intermittent computers. Intermittent computers sometimes run out of power and shut down. CITID allows the computer to continue where it shut off, by periodically backing up the state of the computer into Non-Volatile Memory (NVM). NVM does not lose data when powered off, which allows the state of the computer to be saved until the next power on. In order to periodically back up the state, the CITID runtime needs to know when this can be done. To make this clear, the program is split into tasks by the programmer. These tasks take the form of C functions. Each task does computations as described inside the function body, and then returns a pointer to the next task to be executed. If there is no next task, the return pointer is NULL. After a task is done executing, CITID will back up the system state. The CITID runtime will then select the next task to execute. The system state is represented by some global variables in C. These variables are shared between tasks within a thread. When a variable is part of the system state, it is called a 'task-shared' variable. The task-shared variables can only be used inside tasks; using them outside tasks is illegal. A unique thing about CITID is that it uses threads, which are distinct from tasks. Each thread contains multiple tasks, and there can be a maximum of 64 threads in a program. Threads are explicitly created by the programmer using an API call (described below). A thread can be either 'active' or 'dormant'. If a thread is active, its tasks can be executed by the CITID runtime. If a thread is dormant, the CITID runtime needs to wait for an event. Events are signalled by the programmer inside interrupts and are stored internally inside a persistent queue. There is one queue for every thread. If the event is triggered, the thread goes from the 'dormant' state to the 'active' state; the thread is 'activated'. When this thread is selected, the event that triggered the state change is passed to the first task in the thread; this task is called the 'entry task'. The CITID runtime uses priority scheduling to select which thread to execute. As such, each thread has a specified priority, which is a number from 1 up to and including 64. Different threads cannot have the same priority. This limits the maximum number of threads to 64. Next to the CITID runtime, there is the CITID compiler. Task-shared variables are detected automatically by the compiler, by checking if a variable with global storage is used inside one or more task functions. If it is, it is marked as task-shared; no instrumentation is required by the programmer. If the variable does not have global storage, or is not used inside any task function, it is ignored by the CITID compiler, and marked as non-task-shared and not in NVM. To facilitate the backing up of state into NVM in this task-based programming environment, the CITID runtime requires hints from the programmer. These hints take the form of API functions. The following bullet list describes the API functions and instructions on where and when to use them. - API functions for interfacing. After system initialisation, the CITID runtime needs to be called using the 'citid_scheduler_run()' function. This can be done in the 'main' function of the program. - API functions for creating threads. A thread can be created using the 'CITID_CREATE_THREAD(prio, activate_on_first_boot)' macro. The 'prio' parameter is a number representing

the priority of the thread. The ‘activate_on_first_boot‘ parameter is a boolean value. If set to true, the created thread is activated on first boot. If false, the created thread remains dormant until woken up by an event. In addition, the ‘CITID_CREATE_THREAD‘ macro defines the thread’s entry task. For example, if we want to create a thread with priority ‘15‘ which is activated on first boot, and which has an entry task that prints ‘Hello World‘ with the next task being ‘task_next‘, we would use ‘CITID_CREATE_THREAD(15, true) printf(”Hello World”); return task_next; ‘. The function pointer to the entry task created by the ‘CITID_CREATE_THREAD‘ macro can be accessed using the ‘CITID_THREAD_ENTRY_TASK‘ macro. The ‘CITID_THREAD_ENTRY_TASK‘ macro takes no arguments; the thread priority is automatically deduced. - API functions for directing the compiler. To explicitly exclude a variable from being flagged as task-shared by the compiler, you can place the ‘CITID_IGNORE‘ macro before the variable definition. This also does not store the variable into NVM. To exclude variables from being flagged as task-shared by the compiler, **but** still store them in NVM, you can place the ‘CITID_PERSISTENT‘ macro before the variable definition. The ‘CITID_IGNORE‘ and ‘CITID_PERSISTENT‘ macro’s override the default compiler behaviour. - API functions for events. To signal an event, the programmer can use ‘citid_signal_event_isr(prio, event)‘. This function should only be called inside interrupt service routines! The ‘prio‘ parameter is the priority of the thread that the programmer wants to activate using the event. The ‘event‘ parameter is a pointer to the event of type ‘citid_event_t‘, used to activate the thread. The struct ‘citid_event_t‘ has the following instance variables: 1) ‘void *data‘, a pointer to the data associated with the event. This data should be allocated inside NVM to make sure it is persistent across power failures; 2) ‘uint32_t size‘, the size of the data buffer in bytes; 3) ‘uint32_t timestamp‘, timestamp at which the event took place. Since we do not use the timer functionality of InK, this remains unused and can be set to 0. When a thread is activated by an event, the pointer to that event can be accessed using the ‘citid_event‘ variable. This variable can only be accessed inside the entry task! - API functions for activating threads. To activate a thread from another thread, the programmer can call the ‘citid_activate_thread(prio)‘ function. Here, ‘prio‘ is a number representing the priority of the thread we want to activate.

The API header file is ‘citid.h’.

A.1.3 Prompt for description with example code

Starts with the prompt from Section A.1.1.
The following is an example application of how to use CITID. Remember this example for use in instrumenting code later. In this example, there is a thread with priority 42, which is activated on first boot. There is also an event, which is triggered by a hardware timer. The data in this event is a counter, which is incremented every time the interrupt triggers. The entry task takes the event data and stores it in the task-shared variable counter. Task ’task1’ then calls an external function with the counter value as its argument. The thread then waits again for the next event. When the microcontroller reboots, it is necessary to set up the hardware again. In this example, it is done using the hardware_setup() function.

```
1 #include "example.h"
```

```c
#include "citid.h"

#define THREAD_PRIORITY 42

CITID_PERSISTENT int counter_isr = 0; // Persistent non task-shared event data variable
citid_event_t counter_isr_event = { .data = &counter_isr, .size = sizeof(int), .timestamp = 0 }; // Event struct

int counter; // Task-shared variable

// Forward task declarations
void* task1(void);

// Entry task
CITID_CREATE_THREAD(THREAD_PRIORITY, true) {
    counter = *(int*)citid_event->data; // Get data from event and store into task-shared variable

    return task2; // The next task is task2
}

void* task1(void) {
    do_something_with_counter(counter);

    return NULL; // This is the last task to execute in the thread: the thread lays dormant until awoken by an event.
}

int main(void) {
    hardware_setup(); // Included from "example.h"

    citid_scheduler_run();
}

void __attribute__ ((interrupt(TIMER0_A0_VECTOR))) TIMER0_A0_ISR(void) {
    counter_isr++;

    citid_signal_event_isr(THREAD_PRIORITY, &counter_isr_event);
}
```

A.2 InK

A.2.1 Prompt for description without examples

The following paragraphs describe 'InK', an API and runtime framework. Later on, you will use this description of InK to transform C code into InK-compliant code. So remember the following description for later. 'InK' is a runtime and API meant for programming task-based intermittent computers. Intermittent computers sometimes run out of power and shut down. InK allows the computer to continue where it shut off, by periodically backing up the state of the computer into Non-Volatile Memory (NVM). NVM does not lose data when powered off, which allows the state of the computer to be saved until the next power on. In order to periodically back up the state, the InK runtime needs to know when this can be done. To make this clear, the program is split into tasks by the programmer. These tasks take the form of C functions. Each task does

computations as described inside the function body, and then returns a pointer to the next task to be executed. If there is no next task, the return pointer is NULL. After a task is done executing, InK will back up the system state. The InK runtime will then select the next task to execute. The system state is represented by some global variables in C. These variables are shared between tasks within a thread. When a variable is part of the system state, it is called a ‘task-shared‘ variable. The task-shared variables can only be used inside tasks; using them outside tasks is illegal. A unique thing about InK is that it uses threads, which are distinct from tasks. Each thread contains multiple tasks, and there can be a maximum of 64 threads in a program. Threads are explicitly created by the programmer using an API call (described below). A thread can be either 'active' or 'dormant'. If a thread is active, its tasks can be executed by the InK runtime. If a thread is dormant, the InK runtime needs to wait for an event. Events are signalled by the programmer inside interrupts and are stored internally inside a persistent queue. There is one queue for every thread. If the event is triggered, the thread goes from the 'dormant' state to the 'active' state; the thread is 'activated'. When this thread is selected, the event that triggered the state change is passed to the first task in the thread; this task is called the 'entry task'. By default, a thread is in the 'dormant' state on creation. The InK runtime uses priority scheduling to select which thread to execute. As such, each thread has a specified priority, which is a number from 1 up to and including 64. Different threads cannot have the same priority. This limits the maximum number of threads to 64. To facilitate the backing up of state into NVM in this task-based programming environment, the InK runtime requires hints from the programmer. These hints take the form of API functions. The following bullet list describes the API functions and instructions on where and when to use them. - API functions for interfacing. When the system boots up for the first time, threads need to be created. To facilitate this creation, the InK runtime calls the ‘void __app_init()‘ function on first boot. This function must be defined by the programmer. There is also the ‘void __app_reboot()‘ function, which is called when the system reboots. This can be used to specify actions that need to happen at every reboot. - API functions for creating threads. The ‘__CREATE(prio, entry_task)‘ function creates a thread. The ‘prio‘ argument is a number that represents the priority of the thread. The ‘entry_task‘ argument is a pointer to the function that is the entry task of the thread to be created. The ‘__CREATE(prio, entry_task)‘ function should always and only be called when the system boots for the first time. - API functions for activating threads. To activate a thread, one can use the ‘__SIGNAL(prio)‘ function. The ‘prio‘ argument is a number that represents the priority of the thread to activate. - API functions for specifying tasks. To specify that a function is an InK task, the entire function signature should be replaced by ‘TASK(¡func_name¿)‘, where ‘¡func_name¿‘ is the name of the original function. This should be done for both the function declaration and function definition. The entry task for a thread should use ‘ENTRY_TASK‘ instead of ‘TASK‘. - API functions for specifying when a global variable is task-shared. The ‘__shared(...)‘ macro. This macro should be called in the global scope, before any function definition or declaration. The ‘...‘ in this macro is to be replaced a ';' delimited list of variable declarations for task-shared variables. Variable definitions are not allowed. The ‘__shared(...)‘ macro will mark these variables as task-shared. NOTE: there must be at least 2 bytes of data inside the ‘__shared(...)‘ macro, otherwise there

will be a compiler error. - API functions for specifying when a task-shared variable is read. Every time a task-shared variable is read inside a task function, the name of that variable should be replaced by '__GET(¡var_name¿)', where '¡var_name¿' is the name of the variable to be replaced. - API functions for specifying when a task-shared variable is written. Every time a task-shared variable is written inside a task function, the assignment expression should be replaced by '__SET(¡var_name¿, ¡value¿)', where '¡var_name¿' is the name of the variable, and '¡value¿' is the value that is written to the variable. - API functions for events. To check if the event queue is full, the programmer can use '__EVENT_BUFFER_FULL(prio)', where the 'prio' parameter corresponds to the priority of the thread that we want to check the event queue for. To signal an event, the programmer can use '__SIGNAL_EVENT(prio, event)'. This function should only be called inside interrupt service routines! The 'prio' parameter is the priority of the thread that the programmer wants to activate using the event. The 'event' parameter is a pointer to the event of type 'isr_event_t', used to activate the thread. The struct 'isr_event_t' has the following instance variables: 1) 'void *data', a pointer to the data associated with the event. This data should be allocated inside NVM to make sure it is persistent across power failures; 2) 'uint32_t size', the size of the data buffer in bytes; 3) 'uint32_t timestamp', timestamp at which the event took place. Since we do not use the timer functionality of InK, this remains unused and can be set to 0. When a thread is activated by an event, the pointer to that event can be accessed using the '__event' variable. This variable can only be accessed inside the entry task! - API for setting a variable as persistent non-task shared. If the programmer wants to have a variable that is stored in NVM, but is not supposed to be task-shared, they can use the '__nv' macro. This macro is to be placed in front of the variable declaration.

The API header file is 'ink.h'.

A.2.2 Prompt for description with some examples

The following paragraphs describe 'InK', an API and runtime framework. Later on, you will use this description of InK to transform C code into InK-compliant code. So remember the following description for later. 'InK' is a runtime and API meant for programming task-based intermittent computers. Intermittent computers sometimes run out of power and shut down. InK allows the computer to continue where it shut off, by periodically backing up the state of the computer into Non-Volatile Memory (NVM). NVM does not lose data when powered off, which allows the state of the computer to be saved until the next power on. In order to periodically back up the state, the InK runtime needs to know when this can be done. To make this clear, the program is split into tasks by the programmer. These tasks take the form of C functions. Each task does computations as described inside the function body, and then returns a pointer to the next task to be executed. If there is no next task, the return pointer is NULL. After a task is done executing, InK will back up the system state. The InK runtime will then select the next task to execute. The system state is represented by some global variables in C. These variables are shared between tasks within a thread. When a variable is part of the system state, it is called a 'task-shared' variable. The task-shared variables can only be used inside tasks; using them outside tasks is illegal. A unique thing about InK is

that it uses threads, which are distinct from tasks. Each thread contains multiple tasks, and there can be a maximum of 64 threads in a program. Threads are explicitly created by the programmer using an API call (described below). A thread can be either 'active' or 'dormant'. If a thread is active, its tasks can be executed by the InK runtime. If a thread is dormant, the InK runtime needs to wait for an event. Events are signalled by the programmer inside interrupts and are stored internally inside a persistent queue. There is one queue for every thread. If the event is triggered, the thread goes from the 'dormant' state to the 'active' state; the thread is 'activated'. When this thread is selected, the event that triggered the state change is passed to the first task in the thread; this task is called the 'entry task'. By default, a thread is in the 'dormant' state on creation. The InK runtime uses priority scheduling to select which thread to execute. As such, each thread has a specified priority, which is a number from 1 up to and including 64. Different threads cannot have the same priority. This limits the maximum number of threads to 64. To facilitate the backing up of state into NVM in this task-based programming environment, the InK runtime requires hints from the programmer. These hints take the form of API functions. The following bullet list describes the API functions and instructions on where and when to use them. - API functions for interfacing. When the system boots up for the first time, threads need to be created. To facilitate this creation, the InK runtime calls the 'void __app_init()' function on first boot. This function must be defined by the programmer. There is also the 'void __app_reboot()' function, which is called when the system reboots. This can be used to specify actions that need to happen at every reboot. - API functions for creating threads. The '__CREATE(prio, entry_task)' function creates a thread. The 'prio' argument is a number that represents the priority of the thread. The 'entry_task' argument is a pointer to the function that is the entry task of the thread to be created. The '__CREATE(prio, entry_task)' function should always and only be called when the system boots for the first time. For example, if we want to create a thread with priority '15' and entry task function 't_init', we call '__CREATE(15, t_init)'. - API functions for activating threads. To activate a thread, one can use the '__SIGNAL(prio)' function. The 'prio' argument is a number that represents the priority of the thread to activate. For example, if we want to activate the thread with priority '15', we call '__SIGNAL(15)'. - API functions for specifying tasks. To specify that a function is an InK task, the entire function signature should be replaced by 'TASK(¡func_name¿)', where '¡func_name¿' is the name of the original function. This should be done for both the function declaration and function definition. The entry task for a thread should use 'ENTRY_TASK' instead of 'TASK'. For example, if we want to convert the function declaration 'void* some_func()' to a task, we replace it with 'TASK(some_func)'. - API functions for specifying when a global variable is task-shared. The '__shared(...)' macro. This macro should be called in the global scope, before any function definition or declaration. The '...' in this macro is to be replaced a ';' delimited list of variable declarations for task-shared variables. Variable definitions are not allowed. The '__shared(...)' macro will mark these variables as task-shared. NOTE: there must be at least 2 bytes of data inside the '__shared(...)' macro, otherwise there will be a compiler error. - API functions for specifying when a task-shared variable is read. Every time a task-shared variable is read inside a task function, the name of that variable should be replaced by '__GET(¡var_name¿)', where '¡var_name¿' is the name of

the variable to be replaced. - API functions for specifying when a task-shared variable is written. Every time a task-shared variable is written inside a task function, the assignment expression should be replaced by '__SET(¡var_name¿, ¡value¿)', where '¡var_name¿' is the name of the variable, and '¡value¿' is the value that is written to the variable. - API functions for events. To check if the event queue is full, the programmer can use '__EVENT_BUFFER_FULL(prio)', where the 'prio' parameter corresponds to the priority of the thread that we want to check the event queue for. To signal an event, the programmer can use '__SIGNAL_EVENT(prio, event)'. This function should only be called inside interrupt service routines! The 'prio' parameter is the priority of the thread that the programmer wants to activate using the event. The 'event' parameter is a pointer to the event of type 'isr_event_t', used to activate the thread. The struct 'isr_event_t' has the following instance variables: 1) 'void *data', a pointer to the data associated with the event. This data should be allocated inside NVM to make sure it is persistent across power failures; 2) 'uint32_t size', the size of the data buffer in bytes; 3) 'uint32_t timestamp', timestamp at which the event took place. Since we do not use the timer functionality of InK, this remains unused and can be set to 0. When a thread is activated by an event, the pointer to that event can be accessed using the '__event' variable. This variable can only be accessed inside the entry task! - API for setting a variable as persistent non-task shared. If the programmer wants to have a variable that is stored in NVM, but is not supposed to be task-shared, they can use the '__nv' macro. This macro is to be placed in front of the variable declaration.

The API header file is 'ink.h'.

A.2.3 Prompt for description with example code

Starts with the prompt from Section A.2.1.
The following is an example application of how to use InK. Remember this example for use in instrumenting code later. In this example, there is a thread with priority 42, which is activated on first boot. There is also an event, which is triggered by a hardware timer. The data in this event is a counter, which is incremented every time the interrupt triggers. task1 takes the event data and stores it in the task-shared variable counter. task2 then calls an external function with the counter value as its argument. The thread then waits again for the next event. When the microcontroller reboots, it is necessary to set up the hardware again. In this example, it is done using the hardware_setup() function.

```
#include "example.h"
#include "ink.h"

#define THREAD_PRIORITY 42

__nv int counter_isr = 0; // Persistent non task-shared event data variable
isr_event_t counter_isr_event = { .data = &counter_isr, .size = sizeof(int), .timestamp = 0 }; // Event struct

__shared(
    int counter; // Task-shared variable, int is 4 bytes, this satisfies the 'at least 2 bytes' rule
)
```

```
12
13 // Forward task declarations
14 ENTRY_TASK(task1)
15 TASK(task2)
16
17 // Entry task
18 ENTRY_TASK(task1) {
19     __SET(counter, *(int*)(__event->data)); // Get data from event
       and store into task-shared variable
20
21     return task2; // The next task is task2
22 }
23
24 TASK(task2) {
25     do_something_with_counter(__GET(counter));
26
27     return NULL; // This is the last task to execute in the thread:
        the thread lays dormant until awoken by an event.
28 }
29
30 void __app_init(void) {
31     __CREATE(THREAD_PRIORITY, task1); // Create thread with
       priority 42
32     __SIGNAL(THREAD_PRIORITY);        // Activate thread on first
       boot
33 }
34
35 void __app_reboot(void) {
36     hardware_setup(); // Included from "example.h"
37 }
38
39 void __attribute__ ((interrupt(TIMER0_A0_VECTOR))) TIMER0_A0_ISR(
       void) {
40     counter_isr++;
41
42     if (!__EVENT_BUFFER_FULL(THREAD_PRIORITY)) {
43         __SIGNAL_EVENT(THREAD_PRIORITY, &counter_isr_event);
44     }
45 }
```