Seminar Paper: Operating System Support for Embedded Devices

Jonas Wozar
Friedrich-Alexander-Universität Erlangen-Nürnberg

ABSTRACT

Living in a digital era, the terms "Internet of Things" and "embedded devices" are well known. An increasing number of products contain electronics and are internet-capable, making them part of the Internet of Things. Nowadays, these devices are capable of collecting and computing data. Furthermore, they can communicate via general-purpose protocols. Thus, operating systems are required to execute corresponding applications. While many general-purpose operating systems already exist for such devices, they may not be the optimal solution due to limited resources on these devices. This raises the question of how to save resources while still performing tasks performantly and reliably. Wang and Seltzer suggest that building a use-case-specific operating system represents a better alternative to general-purpose operating systems. Thereby they present Tinkertoy, a set of tools from which one can assemble a custom operating system from building blocks. Consequently, an operating system can be built providing only the necessary functionality to run a specific application, omitting unrequired parts of the system. Furthermore, Wang and Seltzer show that such an operating system can be assembled in only a few lines of code and provides a significant improvement in memory usage compared to established operating systems for embedded devices while performing virtually the same in terms of runtime.

1 INTRODUCTION

Low-powered devices able to collect and compute data have become increasingly popular over the last years. Nowadays, in addition to better performance, these embedded devices are often internet capable and use general-purpose protocols like 6LoWPAN as opposed to previous embedded devices which had to use non-commodity protocols like ZigBee for communication. Consequently, a number of these devices connected together build the Internet of Things (IoT). To execute the desired tasks, they run operating systems (OSs). Opposed to first-generation devices which often ran rather basic OSs like TinyOS [9] and Contiki [3], second-generation devices are capable of running OSs much more similar to well-known desktop OSs like Linux. Common OSs used on more modern devices are FreeRTOS [6] and Zephyr [11]. While offering many options, these are not the optimal way of operating all devices of the IoT. The world of IoT is changing rapidly and so do its devices and their respective fields of application. Additionally, many of these devices are still limited in performance and memory. Consequently, to run an OS like FreeRTOS or Zephyr, some features need to be removed beforehand resulting in a lot of work. Wang and Seltzer suggest that building a custom OS for specific use-cases provides a better alternative. Thereby, they present *Tinkertoy*, a collection of modules, enabling developers to construct their own OS from predefined building blocks. Consequently, Tinkertoy enables developers to tailor an OS to specific applications and use-cases. Moreover, Wang and Seltzer show that with such an OS up to 4 times less memory is required at no performance loss compared to existing alternatives like FreeRTOS and Zephyr.

2 BACKGROUND

In general, the purpose of an OS is to control the hardware and enable applications to be executed effectively. OSs separate applications from the hardware they use [1]. Consequently, an OS is a layer of software between applications and hardware. It provides abstractions for applications to interact with the hardware it is running on. To be able to perform these tasks, OSs include multiple components.

2.1 Key Components of Operating Systems

The core of an OS containing the core components is called the *kernel*. Its components include *process management*, *memory management*, *I/O management*, and *inter-process communication management* [1].

In order to manage multiple processes, the kernel must implement a *process scheduler*, determining when and how long processes are allowed to execute. To be scheduled reasonably, processes are often assigned priorities, from which the scheduler can determine a suitable amount of execution time and when it should be executed.

For memory management, kernels usually implement a *memory manager*. The memory manager determines how much memory is allocated to a process, when and where it is allocated, and what should be done in case the whole available memory is in use.

For I/O management, an *I/O manager* is implemented in kernels. Its purpose is to handle input and output requests from processes and enable processes to communicate with hardware devices in a well-organized manner.

2.2 Related Operating System Concepts

There are many different OS concepts for embedded devices. Some of these incluse *Unikernels*, *Exokernels*, *Library Operating Systems*, and *Real Time Operating Systems*

Unikernels are lightweight kernels designed to run a single application. By including solely the components necessary to run a given application they can be considered a resource-efficient type of OS [12]. Contrary to many other types of kernels, the focus of Unikernels lies on application-level management of resources and the removal of protection boundaries. However, as Unikernels are designed to run a single application only, they do not have a broad spectrum of use-cases.

Exokernels represent a different approach. In general, Exokernels are small kernels that analog to Unikernels focus on application-level resource management. However, Exokernels are not specifically designed to run a single application. By separating resource management from resource protection, Exokernels allow applications to manage resources, while protecting them from each other at the same time. Thereby, they significantly reduce the amount of abstraction by the OS and shift components like virtual memory

into userspace, enabling applications low-level access to hardware. As a result, the runtime of applications can be reduced significantly [4].

As Exokernels provide resource protection, but allow resource management to be handled on application level, operating system abstractions can be customized. Library Operating Systems use this interface, enabling developers to build application-specific operating systems by extending, specializing, or replacing libraries [4]. Therefore, Library Operating Systems often use existing libraries from which developers can assemble the OS to their needs.

A further Concept for IoT OSs are Real-Time Operating Systems (RTOS). As the name suggests, RTOS are aimed at time-sensitive operations. For example in health care, wearable electrocardiograms should be able to detect irregularities in the patient's heart rhythm and notify the doctor as soon as possible [14]. Another example of an area of application would be autonomous plant watering systems, as one expects a plant to be watered as soon as the soil is dry and not to be watered in case the soil moisture is high enough. Examples of RTOS are FreeRTOS and Zephyr.

2.3 FreeRTOS and Zephyr

FreeRTOS is a widely used thread-based open-source RTOS implemented in C with a small kernel. Within FreeRTOS, applications are encapsulated as tasks and have their own stack. With the use of C macros, its kernel can be modified providing some level of customization. Developers are offered the option to enable or disable kernel APIs using *define* macros in a given header file.

Equally to FreeRTOS, Zephyr is an RTOS, thread-based and implemented in C. Sharing a lot of design features with Linux, Zephyr offers multiple scheduling algorithms. Additionally, Zephyr provides more options for customization than FreeRTOS by using a Kconfig interface [2], similar to Linux. Thus, developers can customize scheduling algorithms, device drivers, and more.

As both FreeRTOS and Zephyr allow for disabling unrequired functionality, they were compared to Tinkertoy in section 4. Furthermore both OSs have an offical port for the emulated boards, Wang and Seltzer used to evaluate their work.

3 TINKERTOY

Tinkertoy is a set of prebuild and customizable modules written in C++20 [10] by Wang and Seltzer to enable developers to build their own custom kernel for *low-end devices* as classified by IETF [7]. Comparably to the concept of Exokernels, Tinkertoy puts the creation and choice of abstractions in the hands of developers. However, Tinkertoy approaches this by enabling developers to design the OS from a set of modules. Instead of exposing hardware to applications while still keeping the core functionality of the OS, Tinkertoy represents a set of modules from which a use-case-specific OS can be built to deal with resources efficiently. While FluxOSKit[5] has a similar design concept by making it possible to build a new system from a set of components, it does so by importing components from other systems. As most of these components are not resource-efficient enough for the devices Tinkertoy is targeted at, Tinkertoy's components are designed from the ground up.

3.1 Structural Composition

To enable developers to assemble their own OS, Tinkertoy offers a set of modules, made up from the following:

- Constraints
- Scheduler
- Memory Allocator
- Context Switcher
- Execution State
- System Call
- Dispatcher
- Kernel Service Runtime
- Execution Models
- Task Control Block

These modules consist of a number of components, some of which can be built from predefined building blocks. For example, the scheduler consists of three components, *Scheduling Policy*, *Event Handler*, and *Task Constraints*. The developer is enabled to build each component from a set of generic building blocks resulting in components behaving exactly the way the developer intends them to.

3.2 C++ Features Used for Code Reusability and Flexibility

To implement these generic building blocks, Wang and Seltzer use three C++ specific features: *templates*, *concepts* and *functors*. Using these concepts, Tinkertoy provides reasonable flexibility, high reusability, and easy composability for its building blocks. Furthermore, Tinkertoy's runtime overhead is kept small at the cost of compile time.

Templates are used to design the modules as generic as possible. With the help of templates, components like the scheduler can be defined generically, enabling it to schedule different kinds of tasks on different systems.

Concepts define constraints on types. Multiple of them can be assigned to a single template, enabling the developer to bind a set of requirements to a type. Thereby, Tinkertoy formulates concepts as specific as possible while still allowing for considerable potential. As a result, concepts are used for example to allow a priority-based scheduler to accept only task types overloading comparison operators.

Functors represent the last C++ feature used for the genericity of Tinkertoy's code. Functors are C++ classes that overload the function call operator and can be constrained by concepts. Furthermore, they can be inlined by compilers, providing a performance benefit compared to indirect function calls at runtime. Tinkertoy's building blocks are encapsulated in functors, allowing for the creation of new building blocks from existing ones at compile time.

3.3 Details on More Customizable Modules

Some of Tinkertoy's modules are highly customizable and offer a variety of building blocks to be constructed from. One of those modules is its scheduler which is constructed from the three components *Scheduling Policy*, *Event Handler* and *Task Constraints* as previously mentioned in section 3.1. Each of those components can be built differently from a set of building blocks, and put together using a builder class provided by Tinkertoy, resulting in a scheduler behaving in exactly the desired way. Figure 1 shows the composition of the scheduler from three components, each of which can be made up of a set of building blocks.

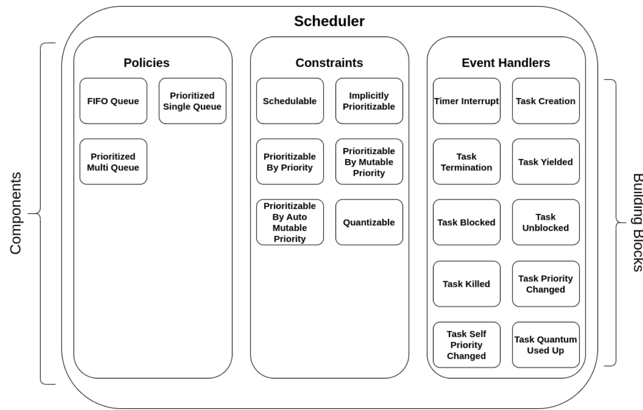

Figure 1: Module Scheduler which can be made up from three components Policies, Constraints and Event Handlers. Each Component can be constructed from a set of Building Blocks.

At its core, the scheduler decides which task should be executed and for how long it should be executed. Thereby, the scheduler watches over ready tasks held in one or multiple queues and chooses the next one based on its scheduling policy. As an example, a simple First In First Out (FIFO) scheduler can be assembled by using the FIFO queue for the policy. Furthermore, schedulable tasks must inherit from the class *Schedulable* to be enqueued to the scheduler's queue. To enable the scheduler to react to scheduling events, the event handler component needs to be specified. As can be seen in Figure1, Tinkertoy provides 10 building blocks for the event handler component, enabling developers to decide to which events the scheduler can respond. For example, when building a kernel that allows processes to create other processes and wait for them to yield, or terminate, one would have to use *Task Creation*, *Task Termination*, *Task Yielded*, *Task Blocked* and *Task Unblocked* as event handlers. An example implementation of a scheduler for this purpose is shown in Figure 2.

```
using Policy = PolicyWithEnqueueExtensions<FIFO, Counter>;
class CustomFIFOScheduler : public SchedulerAssembler<Policy,
  TaskCreation::Cooperative::KeepRunningCurrent<Task>,
  TaskTermination::Common::RunNext<Task>,
  TaskBlocked::Common::RunNext<Task>,
  TaskUblocked::Cooperative::KeepRunningCurrent<Task>,
  TaskYielded::Common::RunNext<Task>> {}
```

Figure 2: Example implementation of a scheduler for a system without a timer, allowing processes to create other processes and wait for them to finish.

For reasonable handling of tasks, Tinkertoy provides the module *Task Control Block*, which also can be built from a set of building blocks. Each task needs its own task control block, providing information like its priority or identifier. Tinkertoy provides a set of components from which developers can build task control blocks according to their needs. Furthermore, Tinkertoy provides components to build task control block initializers and finalizers for these task control blocks. Task control blocks have to meet a set of constraints, which can be achieved by the construction of components provided by Tinkertoy. Thereby, task control blocks must specify a type of stack with the help of stack components (e.g. Shared Stack) and whether tasks are prioritizable with the help of further components (e.g. Priority Level). In the running kernel, for task creation and termination, corresponding task control blocks have to be allocated or released. For that purpose, Tinkertoy provides initializer and finalizer components for each task control block component.

Another customizable module of Tinkertoy is its *Execution Model*. Tinkertoy generally supports two types of Execution Models, thread-based Execution Model and event-driven Execution Model. After defining task control blocks, developers can specify the execution model by exposing the respective system calls like thread creation for a thread-based model or register/unregister events for an event-driven model. In a thread-based Execution model, a typically large number of threads can handle tasks concurrently. In addition, these threads are often short-lived [8]. As can be seen in section 4, the thread-based execution model usually is a good choice for devices like gateways where multiple concurrent threads can translate messages. Unlike thread-based systems, event-driven systems do not implement a large number of threads. In event-driven systems, the control flow is determined by a set of events that can occur and trigger the execution of corresponding tasks. Thus, systems that can be expressed as state machines are usually best implemented as event-driven systems which can be seen in section 4. For each execution model, developers first need to define task control blocks to enable the kernel to manage the execution of tasks.

4 EVALUATION

In this section, Wang and Seltzer's work is evaluated and compared with FreeRTOS and Zephyr on the basis of memory usage and runtime performance. For the benchmarking an automatic watering system for a plant was chosen as a plausible setup that could take place in the real world.

4.1 Watering System Setup

The watering system consists of three devices instantiated with emulated Stellaris LM3S811 boards. The three devices are a *monitor*, an *actuator*, and a *gateway*, each one running its own kernel built with Tinkertoy. Since Tinkertoy currently does not support networking, the devices communicate via UART.

In this setup, the monitor device has to keep track of the moisture level in a pot for plants and inform the actuator whether it has to start watering or it has to stop. It runs an event-driven kernel and uses three events and three event handlers. Firstly, *Periodic Timer* and the corresponding event handler *Sensor Reader*. Consequently, the monitor device is able to measure the moisture level in timed intervals. To inform the actuator device whether it has to water the plant or not, two events *Dry Soil* and *Wet Soil* and corresponding event handlers *Dry Handler* and *Wet Handler* are used.

The second device is the actuator controlling the gate of a water bottle to start or stop watering the pot. Similarly to the Monitor, it is realized using an event-driven kernel with three events and three corresponding event handlers. The first two events are *Start Watering* and *Stop Watering* and trigger the event handlers *Open Gate* and *Stop Gate* resulting in the pot being watered or not. In case

the water bottle is empty, the event *No Water* triggers the event handler *Signal Alert* which then informs the gateway device.

The third device is the gateway device. The gateway uses a thread-based kernel and runs three concurrent threads to translate incoming Constrained Application Protocol (CoAP) or Hypertext Transfer Protocol (HTTP) messages to be able to inform a user about the watering system's status. For the gateway kernel, a cooperative scheduler able to handle task creation and blocked and unblocked events was used. Furthermore, the kernel has a memory allocator to provide the threads with respective stacks. Besides other modules, an interrupt handler is needed to maintain a queue of threads and inform the scheduler about the thread's states.

4.2 Comparison with Alternatives

The three operating systems were evaluated in memory footprint, flash footprint, active stack usage, and the round trip time of the gateway kernel. As can be seen in Figure 3, the kernels implemented with Tinkertoy use significantly less memory than the kernels implemented in FreeRTOS and Zephyr. Accordingly, as shown in Figure 4, Tinkertoy has a considerably lower flash footprint. This reduced memory footprint is the result of Tinkertoy enabling developers to tailor the kernel to their needs with only the required functionality. Consequently, the task control blocks of the monitor kernel implemented using Tinkertoy are 12 bytes in size, while on FreeRTOS and Zephyr, they take up 64 and 112 bytes. Although Tinkertoy provides a better stack usage, the measured benefit was not as significant as in flash and memory usage. Lastly, Wang and Seltzer measured Tinkertoy's gateway performance based on the round trip time, as shown in Figure 5, to be comparable to FreeRTOS and Zephyr.

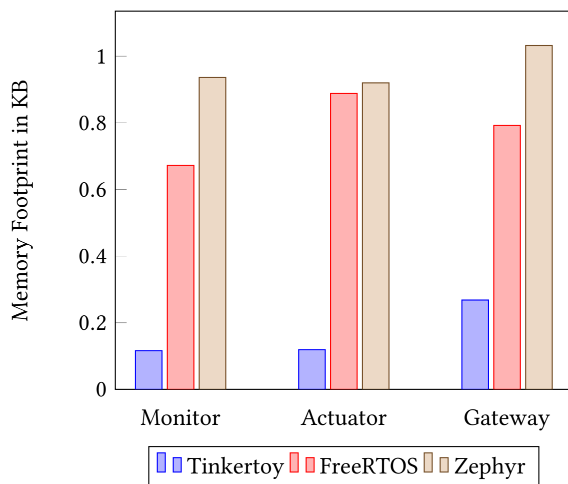

Figure 3: Memory Footprint in KB of the Monitor, Actuator, and Gateway device for a plant watering system, implemented with Tinkertoy, FreeRTOS and Zephyr running on an Stellaris LM3S811 board emulate by the ARM Fast Models.

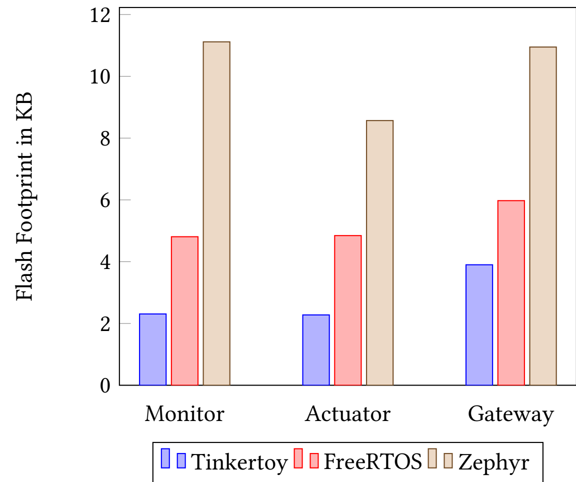

Figure 4: Flash Footprint in KB of the Monitor, Actuator, and Gateway device for a plant watering system, implemented with Tinkertoy, FreeRTOS and Zephyr running on an Stellaris LM3S811 board emulate by the ARM Fast Models.

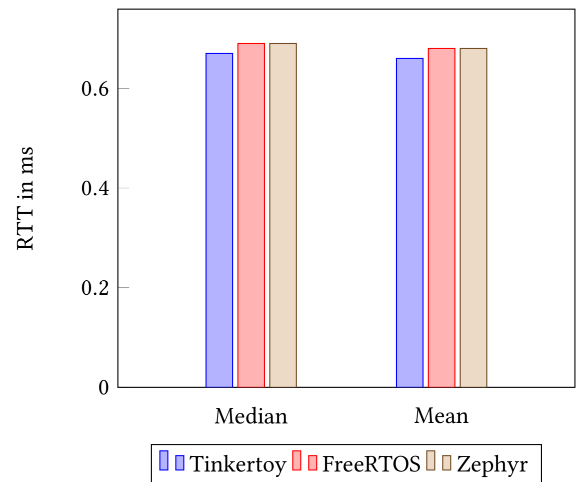

Figure 5: Median and mean round trip time of the gateway kernel calculated from 1000 measured samples.

5 CONCLUSION

Tinkertoy by Wang and Seltzer, besides sharing some ideas with Exokernels [4] and the FluxOSKit [5] and other IoT operating systems, provides a novel approach in OS design for IoT devices by enabling developers to build a custom OS from a set of predefined modules. Therefore, the code is written generically and highly reusable. Consequently, with the help of Tinkertoy, developers can build a use-case-specific OS, implementing solely the required functionality in only a few lines of code. By only including the components required for a specific use-case, Tinkertoy has a memory footprint significantly smaller than FreeRTOS and Zephyr with no performance loss. However, Tinkertoy still allows for improvements like

support for nested hardware interrupts, multiple kernel stacks, and networking. Furthermore, synchronization primitives like mutexes are not yet provided by Tinkertoy.

REFERENCES

[1] Harvey M Deitel, Paul J Deitel, David R Choffnes, et al. 2004. *Operating systems*. Pearson/Prentice Hall.

[2] Zephyr Project Documentation. 2022. *Configuration System (Kconfig)*. https://tinyurl.com/zephyrkconfig

[3] A. Dunkels, B. Gronvall, and T. Voigt. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*. 455–462. https://doi.org/10.1109/LCN.2004.38

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[5] Bryan Ford, Godmar Back, Gregory Benson, Jay Lepreau, Albert Lin, and Olin Shivers. 1997. The Flux OSKit: A Substrate for Kernel and Language Research. *ACM SIGOPS Operating Systems Review* 31 (08 1997), 38–51. https://doi.org/10.1145/269005.266642

[6] FreeRTOS. [n. d.]. *FreeRTOS*. https://www.freertos.org

[7] IETF. 2014. *RFC 7228: Terminology for Constrained-Node Networks*. https://tools.ietf.org/html/rfc7228

[8] Hugh C. Lauer and Roger M. Needham. 1979. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.* 13, 2 (apr 1979), 3–19. https://doi.org/10.1145/850657.850658

[9] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. 2005. *TinyOS: An Operating System for Sensor Networks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 115–148. https://doi.org/10.1007/3-540-27139-2_7

[10] C++ Reference Manual. 2024. *Constraints and concepts*. https://en.cppreference.com/w/cpp/language/constraints

[11] Zephyr Project. 2023. *Zephyr*. https://zephyrproject.org

[12] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-Application Library Operating Systems.

[13] Bingyao Wang and Margo Seltzer. 2022. Tinkertoy: Build Your Own Operating Systems for IoT Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4028–4039. https://doi.org/10.1109/TCAD.2022.3198907

[14] Guangyu Xu. 2020. IoT-Assisted ECG Monitoring Framework With Secure Data Transmission for Health Care Applications. *IEEE Access* 8 (2020), 74586–74594. https://doi.org/10.1109/ACCESS.2020.2988059