Author(s): Tan, Xinyu; Hakala, Ismo

Title: StateOS : A Memory-Efficient Hybrid Operating System for IoT Devices

Year: 2023

Version: Published version

Please cite the original version:

Tan, X., & Hakala, I. (2023). StateOS : A Memory-Efficient Hybrid Operating System for IoT Devices. IEEE Internet of Things Journal, 10(11), 9523-9533. https://doi.org/10.1109/jiot.2023.3234106

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3234106

StateOS: A Memory-Efficient Hybrid Operating System for IoT Devices

Xinyu Tan, Ismo Hakala*

***Abstract*—The increasing significance of operating systems (OSs) in the development of the internet of things (IoT) has emerged in the last decade. An event-driven OS is memory efficient and suitable for resource-constrained IoT devices and wireless sensors, although the program's control flow, which is determined by events, is not always obvious. A multithreaded OS with sequential control flow is often considered clearer. However, this approach is memory-consuming. A hybrid OS seeks to combine the strengths of the event-driven approach with multithreaded approach. An event-driven cooperative threaded OS represents a hybrid approach that supports concurrency by explicitly yielding control to another thread. Although this approach is memory efficient, as cooperative threads are not preemptive, it may not provide sufficient real-time performance.**

This article proposes a memory-efficient hybrid OS, called StateOS, for resource-constrained IoT devices. It is an event-driven cooperative threaded OS with partial real-time performance. StateOS implements a hybrid task scheduler that combines two cooperative threaded subsystems as kernel processes on a priority-based preemptive scheduler. This approach provides adequate real-time performance for IoT devices at a low memory cost.

***Index Terms*—cooperative programming, internet of things, IoT OS, hybrid operating system, wireless sensor network operating system, WSN OS**

I. Introduction

Internet of things (IoT) research has been very active over the past decade. This technology is expected to change people's daily lives by becoming part of the surrounding ambient objects [1]. In 2020, the number of IoT connections exceeded that of non-IoT connections for the first time by 12 billion [2].

Most of the deployed IoT devices are based on wireless sensors. These devices share similar restrictions as the nodes of wireless sensor networks (WSN), such as restricted resources, distant deployment, unreliable network connections, and dynamic network topology. Therefore, existing WSN operating systems (OSs), such as TinyOS [3] and Contiki [4], are also utilized in IoT devices.

The typical OS for resource-constrained IoT or WSN devices supports either an event-driven or a thread-based programming model. In an event-driven model, programs are collections of event handlers, and the execution of an event handler is triggered by events. This approach is well-suited for data-centric IoT applications. Event-driven OSs are memory efficient and, thus, attractive for use in resource-constrained IoT platforms. However, programming a complex system with an event-driven model may be challenging because of the manual control of the stack, the lack of blocking functions, and the events that determine the flow of the program [5], [6], [7], [8].

A thread-based model allows for sequential control flows of a thread. This model is attractive from a programmer's perspective, as the programming pattern is intuitive for the human mind. A typical multithreaded OS manages concurrent threads with preemptive task scheduling, in which the execution of a thread can interleave. Preemptive task scheduling offers certain advantages, such as automatic task switching and automatic stack management. In a preemptive multithreaded OS, each thread requires individual stack memory allocation. This entails a memory consumption problem in resource-constrained devices that may compromise their overall performance.

A hybrid model is a compromise solution for a memory-efficient, multithreaded OS. Many previous proposals merged event-driven systems and multithreaded systems in different combinations to obtain a balance between memory consumption and performance. Cooperative threaded programming, exemplified by Protothreads [5], is a hybrid model that supports cooperative threads in an event-driven system. These threads are specifically programmed to voluntarily hand over processor control to another thread at the yield point to enable concurrency between the threads. However, an event-driven cooperative threaded system can have problems with real-time requirements [7] because a time-sensitive task cannot obtain processor control until the current task reaches the yield point.

This study contributes to the literature by proposing a memory-efficient hybrid OS, StateOS, that offers an adequate real-time performance. StateOS implements macro-based application interfaces for programming event-driven WSN applications in a threaded fashion, a hybrid task scheduler that supports cooperative and preemptive task management, a hybrid memory management module that can alleviate fragmentation problems, semi-automated stack management interfaces for cooperative task management, and cross-layer network architecture to reduce communication overheads. With these features, this approach provides a memory-efficient OS for resource-constrained wireless devices to support increasingly complex IoT tasks.

StateOS is based on a cooperative threaded programming approach and a hybrid task-scheduling solution. The OS implements a hybrid task scheduler to support cooperative and preemptive task scheduling. A priority-based preemptive context switcher manages two kernel processes with different priorities. This allows the process with higher priority to

Xinyu Tan and Ismo Hakala are with the University of Jyväskylä, Kokkola University Consortium Chydenius, Kokkola, Finland. (e-mail:xinyu.tan@jyu.fi; ismo.hakala@jyu.fi).

* Corresponding author.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3234106

preempt the other. In both kernel processes, a cooperative task scheduler is implemented to manage the threads cooperatively. As a result, this hybrid approach can provide adequate real-time performance.

StateOS is oriented toward resource-constrained IoT and WSN devices. Therefore, it is designed to be a lightweight and modularized system that utilizes a microkernel architecture. In addition, the network protocol structure follows a cross-layer design to reduce the memory cost of network communication.

The remainder of this paper is structured as follows. Related research is reviewed in Section II. The system architecture and the kernel are proposed in Sections III and IV, respectively. A code example is shown in Section V, and the platform implementations are presented in Section VI. An evaluation of the proposed approach is given in Section VII. Finally, discussion and conclusion are provided in Section VIII.

II. Related work

Existing OSs for resource-restricted systems, such as WSN and IoT devices, support an event-driven programming model, a multithreaded programming model, or a hybrid model that combines event-driven and multithreaded models.

Event-driven OSs, exemplified by TinyOS [3], Contiki [4], OpenWSN [9], and SOS [10], are preferred over data-centric IoT/WSN applications for their event-based computational mechanisms and resource efficiency.

TinyOS was one of the earliest OSs to address the unique restrictions of WSN devices. It follows monolithic kernel architecture that is efficient but challenging to understand and maintain. Contiki applies a modularized design. The kernel implements an event scheduler that dispatches the event to the executing task. OpenWSN is an event-driven OS that focuses on providing network stack services. It retains a simple system design with a basic monolithic kernel architecture. SOS implements a modularized system structure using weakly linked components. The interactions between these components are accomplished by event-driven messages.

The event-driven programming style can be challenging [7], [11] due to associated programming difficulties, such as event-determined control flows and manual stack management. On the other hand, OSs that support a multithreaded programming model provide the programmer with a more familiar programming experience for sequential flow control, proactive task management, and automatic stack management. Typical examples of multithreaded OSs are MANTIS OS[12], RIOT [13], FreeRTOS [14], Zephyr [15], and Mbed OS [16].

MANTIS OS is a multithreaded OS designed for WSN microsensor platforms. It implements a layered architecture based on a lightweight preemptive kernel. RIOT is a multithreaded OS that aims to provide a Linux-like programming experience. It has a microkernel architecture with a preemptive scheduler. FreeRTOS is a popular real-time OS for small embedded systems and has been ported to IoT platforms. It supports multithreaded programming using a preemptive task scheduler. In contrast to the OSs above that support traditional IoT platforms with 8-bit microcontroller units (MCUs), Zephyr and Mbed OS, by default, are used on platforms with 32-bit MCUs [15], [16]. Zephyr has two kernel implementations: a microkernel for less-constrained devices and a nanokernel for resource-limited devices. It supports multithreaded programming through different strategies, including priority-based, cooperative, earliest-deadline-first, preemptive, and non-preemptive scheduling. Mbed OS is a preemptive multithreaded OS that supports real-time software execution. It implements a kernel based on CMSIS-RTOS RTX [17] which is designed for Cortex-M processor-based platforms.

For certain resource-constrained devices, multithreaded OSs are heavyweight. However, complex IoT applications still prefer a multithreaded OS if the device supports it. Many programmers find a multithreaded OS to be more familiar and clearer for programming than an event-driven OS.

A hybrid OS is a compromise approach that combines event-driven and multithreaded systems. It aims to provide a memory-efficient OS with a thread-based programming style. Previous studies have attempted to create a balance between resource consumption and performance by assembling event-driven and multithreaded systems in different approaches.

One hybrid approach, exemplified by Protothreads [5] and TinyThreads [6], implements cooperative threaded APIs in an event-driven system. Event-driven tasks are explicitly programmed to perform yield operations as cooperative threads. Protothreads provide macro-based abstractions and allow a thread to yield when performing blocking operations. A protothread is stackless, so it is memory efficient but requires manual stack management. TinyThreads is a library extension of TinyOS. It supports cooperative threads by implementing a cooperative scheduler within a TinyOS kernel task. TinyThreads allocates individual thread stacks, which makes them heavier than Protothreads. This approach does not support real-time performance because there is no preemption between the threads.

The other hybrid approach, exemplified by TinyMOS [18], TOSThreads [19], SenSpire OS [20], Event-Bus [21], and OpenSwarm [22], combines event-driven and preemptive multithreaded systems to provide event-driven and multithreaded programming models.

TinyMOS implements a TinyOS subsystem in the primary thread of a MANTIS OS kernel. The subsystem manages event-driven tasks that can spawn slave threads to perform long-term operations. TOSThreads is the official designated multithreaded solution for TinyOS. It has a preemptive thread scheduler that maintains TinyOS as a subsystem in a high-priority thread. Long-term tasks are processed by application threads. SenSpire OS has a preemptive kernel that maintains two hierarchical event-driven subsystem threads. Including the interrupt routine, this forms a three-level event-driven system. Non-event-driven threads are low-priority threads that process long-term application tasks. Event-Bus has event-driven subsystems based on a preemptive scheduler. A subsystem maintains multiple cooperative subroutines to support a message-based, event-driven model called the *publish-subscribe* model. OpenSwarm[1] implements a hybrid kernel and natively supports preemptive

[1]OpenSwarm is a swarm robotic OS with computational restrictions that are similar to those of IoT sensor OSs.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3234106

and cooperative scheduling. The preemptive scheduler manages the thread-based program, and short reactive tasks are handled by the event handler functions.

This hybrid approach is flexible because the programmer can choose a suitable programming model for different tasks. However, this hybrid system structure inherits problems from both system models, such as the event-driven task needing to be run-to-complete and the preemptive threads being memory-consuming.

HybridKernel [23] takes an alternative approach that combines the event-driven cooperative threaded model and the preemptive multithreaded model. HybridKernel allows the creation of multiple event-driven Protothreads subsystems as preemptive threads. This approach solves the lack of the real-timeness of Protothreads by allowing preemption between the subsystems. Similar to Protothreads, HybridKernel requires manual stack management, which is a potential burden for the programmer. HybridKernel demands a fixed-sized stack memory allocation for each preemptive thread, which is similar to other preemptive multithreaded solutions. The stack memory allocation is a heuristic and involves stack overflow risk. As a result, programmers are inclined to allocate redundant stack memory, causing memory waste.

StateOS is related to the works above and addresses the problems revealed by the HybridKernel approach. StateOS's hybrid approach combines cooperative threaded and preemptive systems to provide partial real-time support to threads, which is similar to that offered by HybridKernel. Memory efficiency is achieved by introducing a memory-efficient hybrid task scheduler that consumes only one additional stack memory. In addition, cooperative stack management processes are semi-automatized by the task APIs.

III. StateOS: An Overview

StateOS is intended to provide a cooperative threaded OS with real-time capability for sensor-based IoT devices and wireless sensors. This OS implements a microkernel architecture, cross-layer network protocol design, and hybrid task scheduler to address resource constraint-related issues.

StateOS supports cooperative threaded programming through macro-based task APIs. These APIs are mainly designed for programming system modules and high-performance applications. This native programming model is not advocated as novice-friendly because of the system-specific language and system knowledge requirements. StateOS extensively supports a state machine-based visual programming model, statecharts, as a novice-friendly alternative approach. Statecharts are supported by *statechart middleware* and *action libraries*, in addition to StateOS. Readers are kindly referred to [11] for detailed information.

The StateOS architecture is depicted in Figure 1. The *kernel* implements a microkernel architecture with essential functions, including task APIs, task management services, and resource management functions.

The *cross-layer management entity* manages system services and sensor services as modules. These services are configurable according to the application requirements. The *database* maintains the system's global information, such as the system's dynamic parameters and network details. It is openly accessible to other components for performance optimization and system diagnosis.

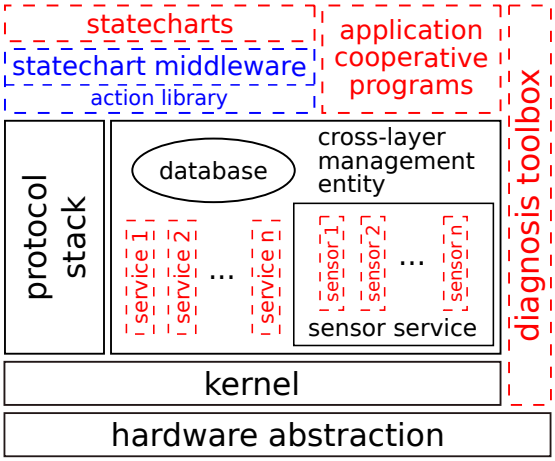


Figure 1. StateOS architecture. The components represented using a dashed line are the configurable modules.

The *protocol stack* collects the network protocol programs and provides network communication services. It provides cross-layer interfaces for efficient network communication.

The *diagnosis toolbox* is an optional component that collects diagnostic instruments, such as a debug message printer (via cable or radio), diagnosis shell, radio signal evaluator, executive time analyzer, and memory logger. These tools are intended to aid in the debugging process.

Hardware heterogeneity is handled by *hardware abstraction* interfaces. The implementation of these interfaces is platform-dependent. Up to now, we have supported some MCUs of the ARM M0 series and the Microchip XMEGA A series. Moreover, hardware implementations include some useful sensors and radios.

StateOS applies a cross-layer network communication design, depicted in Figure 2, for resource efficiency and optimized network performance. This cross-layer design was proposed by [24], who distinguished inter-module communications as *asynchronous messages* and *synchronous function calling*.

Asynchronous messages are used to carry primitive data messages that cross vertically adjacent layers in a request-response manner. Similar to the traditional open systems interconnection module, network messages between application layers traverse all protocol layers with the necessary overhead. In contrast, system services, especially network services (e.g., clock synchronization and traffic control), can directly access the lower layers to reduce the overhead problem.

The control and management communications between parallel components are handled by horizontally synchronous function calls. Network services can control protocol layers for optimized network performance. Furthermore, applications and protocol programs can also efficiently access system services and databases.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3234106

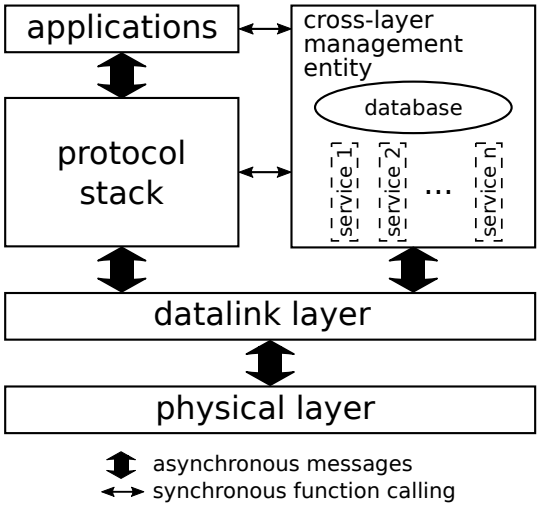


Figure 2. StateOS cross-layer network architecture

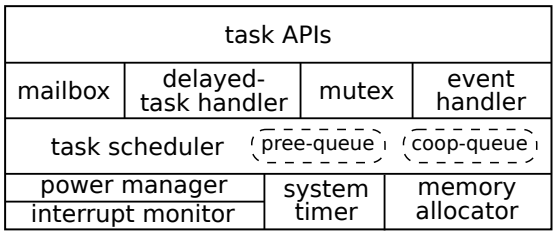


Figure 3. Kernel structure

IV. The Kernel

StateOS implements a lightweight kernel with reduced functionality. The kernel implements essential system functions, as shown in Figure 3, including task APIs, suspended task handlers, a task scheduler, and resource management services.

There are various suspended task handlers, such as *mailbox*, *delayed task handler*, *mutex*, and generic *event handler*. The suspended task is assigned to the corresponding handler based on the cause of the suspension.

Cooperative threads in the kernel are called *tasks*, which are managed by a lifecycle state model, as depicted in Figure 4. The state transition is driven by task APIs and the task scheduler. A task can be created by task-creating functions. This newly created task is in the state *new*. Depending on the function, a new task can be issued to the task scheduler and is queued for execution in the state *ready*. Alternatively, a new task can be sent to a suspended task handler as an event-driven task and labeled by the state *suspended*. An event-driven task can be triggered by events, and its state changes to *ready*. A task is in the state *running* during execution. A running task can cooperatively release processor control by performing yield or suspend operations, which change its state to *ready* or *suspended*, respectively. Finally, a finished task is in the state *end* before being destroyed.

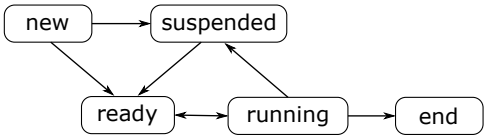


Figure 4. Task lifecycle state model. A task is created in the state *new*. The state *ready* indicates that a task is scheduled in the task queue and is ready to be executed. An executing task is in the state *running*. When the processing is completed, the task is in the state *end*. A task in the state *suspended* is suspended in the *suspended task handlers* and can be triggered to resume by an event.

Table I
Typical task APIs[ab]

Task prototype interfaces
TASK(type, name, *arg_type, arg_label ...*) STARTUP_TASK(name) STARTUP_DELAYED_TASK(name, delay) STARTUP_REPEATED_TASK(name, period) MAILBOX_MK(mailbox, mail_list)
Task return interfaces[c]
TASK_END(*retVal*) TASK_EXIT(*retVal*)
Task yield operations
TASK_WAIT_UNTIL(condition, *locVal ...*) TASK_WAIT_WHILE(condition, *locVal ...*) TAKS_YIELD(*locVal ...*)
Task suspend operations
TASK_WAIT_DELAY(delay, *locVal ...*) TASK_WAIT_EVT(evt, *locVal ...*) TASK_MUTEX_LOCK(mutex, *locVal ...*) TASK_SUSPEND(*locVal ...*)
Synchronized task-calling operation
TASK_CALL(task, *args ...*)(*locVal ...*) TASK_CALL_RETVAL(type)
Task-creating functions
os_add_task(task, *args ...*) os_add_pree_task(task, *args ...*) os_delayed_task(delay, task, *args ...*) os_repeated_task(period, task, *args ...*) os_evt_pending(evt, task, *args ...*)
TCB management functions
tcb_make(task, *args ...*) tcb_add_followup(master, followup) tcb_clone(original)

[a] This table collects the typical and generalized task APIs.
[b] The italic font indicates variadic arguments, i.e., it accepts any number of arguments of any type.
[c] The return interfaces accepts at most one argument *retVal*.

A. Task APIs

The kernel implements macro-based task APIs that support the cooperative threaded programming model. Typical APIs are categorized in Table I to illustrate their related functionalities.

The implementation of a task starts with the declaration of a task prototype interface and ends with a particular task return interface, *TASK_END*. Altogether, they complete a *switch-case* structure as the base of local continuation, whereas a yield point

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3234106

is implemented by a *case* statement that is labeled with the line number of the source code. The other task return interface, *TASK_EXIT*, is used to terminate a task in the middle of the process.

Code 1 primarily expands an example task implementation. The macro *TASK* is a basic prototype interface, as expanded in lines 13–18, which implements a part of the *switch-case* primitive structure. The variadic task argument *a* is automatically initialized when the task begins, as in lines 15. Between lines 22–27, it expands a basic yield operation *YIELD* that contains a local continuation structure as a yield point (lines 23–26). The macros *_VAR_SAVE* and *_VAR_RESTORE* in the *YIELD* expansion function preserve the local variable *b* across the yield point. In the end, the macro *TASK_END*, as expanded in lines 31–34, completes the primitive structure of the task.

Code 1. Example of the simplified primary expansion of the task API. All irrelevant details have been omitted.

```
1  // The original macro-based task implementation
2  TASK(void, my_task, int, a)
3  {
4    int b = 0;
5
6    TASK_YIELD(b);
7    printf("%d\n", a + b);
8
9    TASK_END();
10 }
11
12 // The primarily expanded implementation
13 task_retval_t my_task(tcb_arg_t *_args)
14 {
15   int a = _tcb_argv(_args);
16   _TCB_PT_->resume = false;
17   switch (_TCB_PT_->line) {
18   case 0:
19   {
20     int b = 0
21
22     _VAR_SAVE(b);
23     _TCB_PT_->resume = true;
24     _TCB_PT_->line = __LINE__; case __LINE__:;
25     if (_TCB_PT_->resume)
26       return TASK_RETVAL_YIELD;
27     _VAR_RESTORE(b);
28
29     printf("%d\n", a + b);
30
31     _TASK_RETVAL();
32     }
33   }
34   return TASK_RETVAL_OK;
35 }
```

In addition to the macro *TASK*, task APIs provide interfaces that extend the macro *TASK* for particular purposes. The macro prefixed by *STARTUP* declares a startup task that is executed when the system starts. A mail handling task is declared by the macro *MAILBOX_MK*, which can automatically respond to system messages.

Task flow control APIs include task yield operations and suspend operations. These operations are extensions based on the *YIELD* operation (as introduced previously). The yield operations send a running task back to the scheduler and repeatedly estimate the condition until it is satisfied. Alternatively, a task with a suspended operation is dispatched to the associated suspended handler. For example, the operation *TASK_WAIT_DELAY* suspends a task in the *delayed task handler*, and a mutex lock failure by the operation *TASK_MUTEX_LOCK* suspends the task in the *mutex* handler.

Semi-automatic local variables are preserved by task flow control APIs. Unlike other stackless approaches, it is safe to use local variables across the local continuation structure in StateOS. However, task flow control APIs cannot detect the presence of local variables. Therefore, they must be introduced to the corresponding yield/suspend operations as variadic arguments, as shown in Code 1, line 6.

The APIs provide a synchronized task-calling mechanism that allows a task to call a subroutine task and wait until it is completed. This task-calling process is similar to calling a C function. The macro *TASK_CALL* creates and introduces a new task to the kernel and then suspends the running task until the called task is finished. Two pairs of parentheses follow the macro *TASK_CALL*. They are for the variadic task arguments and local variables because C language does not allow multiple variadic arguments in one set of parentheses. The subroutine task issued by *TASK_CALL* is capable of passing the *return value* to the caller. The caller should fetch this return value through the macro *TASK_CALL_RETVAL*.

A task is typically created and introduced to the kernel by a *task-creating function*. The functions *os_add_task* and *os_add_pree_task* can send a task directly to the scheduler as a regular or a preemptive task, respectively. The other task-creating functions can dispatch a newly created task to the relevant suspended handlers as an event-driven task.

Information about a task is maintained in a data structure, namely, a *Task Control Block (TCB)*. A TCB includes the task implementation address, state, command, identity, name, preemption, priority, task arguments, etc. Task APIs implement TCB management functions for more flexible flow control. For example, the function *tcb_add_followup* can link multiple TCBs in a daisy chain. When the previous task in the chain is completed, the following one is automatically invoked, and the function *tcb_clone* clones an existing TCB.

B. Suspended task handlers

Suspended tasks are event-driven because they are typically resumed by a specific event or signal. These tasks are suspended in the corresponding handlers until they are reactivated.

The *mailbox* system provides an interlayer communication approach between the system services and applications. A potential mail recipient can register a mail handler (with the macro *MAILBOX_MK*) in the mailbox system. Mail delivery can trigger the handler as a high-priority task. A typical use of the mailbox is to forward the radio messages received from the protocol stack to applications.

Delayed tasks are suspended in *delayed-task handlers*. This module is associated with a hardware timer that is used to count down task delays. A delayed-task queue sorts these tasks by their delay time. A timer event dispatches expired tasks to the scheduler.

The *mutex* module implements a locking mechanism for safe resource access. The failure of a mutex lock attempt causes task suspension. These suspended tasks are sorted by their priority in the associated mutex entries and wait for the required resources. Once the mutex is unlocked, relevant tasks are resumed.

The *event handler* manages generic event-suspended tasks. These tasks are registered in the associated event entries.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3234106

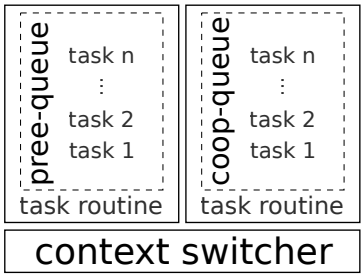


Figure 5. Task scheduler

Depending on the propriety, an event can trigger all relevant tasks at once, or it can exclusively trigger the first one.

C. Task scheduler

The kernel schedules ready tasks using a hybrid strategy. The task scheduler, as depicted in Figure 5, comprises two task queues: *pree-queue* and *coop-queue*. A *context switcher* empowers the pree-queue to preempt the coop-queue. In each task queue, a *task routine* is implemented as a scheduler to manage the tasks in a priority-based cooperative manner. This structure achieves hybrid task scheduling through preemptive task management between the task queues and cooperative task management between the tasks in the same queue.

Figure 6 demonstrates the task scheduling implementation in response to the hardware interrupt. The system interrupt *INT0* interrupts task *T0* and inserts a preemptive task *T1* in the pree-queue. When INT0 returns, the context switcher issues a context switch to the pree-queue, and *T1* is executed. In this demonstration, it is presumed that *T1* is the only task in the pree-queue at the moment. Therefore, the completion of task *T1* empties the pree-queue and causes another context switch that continues task *T0*.

Task *T2* is the subsequent task that is executed when *T0* ends. Task *T2* invokes preemptive task *T3*, causing an immediate context switch. The interrupt *INT1* interrupts the execution of *T3* and inserts another preemptive task, *T4* into the pree-queue. However, task *T4* must wait until *T3* is finished because it follows cooperative task management in the same task queue. Task *T2* is resumed once *T4* is completed.

Thus far, this proposed hybrid strategy is naïve because a blocked pree-queue task can prevent the coop-queue tasks, and a blocked high-priority task can prevent a lower-priority task. Therefore, a design principle is established that a preemptive task shall not contain a blocking operation. However, blocking a preemptive task is not prohibited. In such a case, a blocked preemptive task waives this privilege by descending to the coop-queue when it yields. In addition, inside a task queue, any yield operation downgrades the task to the lowest priority, and the task is rescheduled at the end of the task queue.

The pree-queue and the coop-queue are, de facto, two kernel processes scheduled by the context switcher. Thus, both queues require individual memory stacks. The preemptive tasks in the pree-queue typically require small memory stacks, as they are short-lived. Therefore, the stack memory assigned to the pree-queue can be small (e.g., 128 or 256 bytes). As shown in

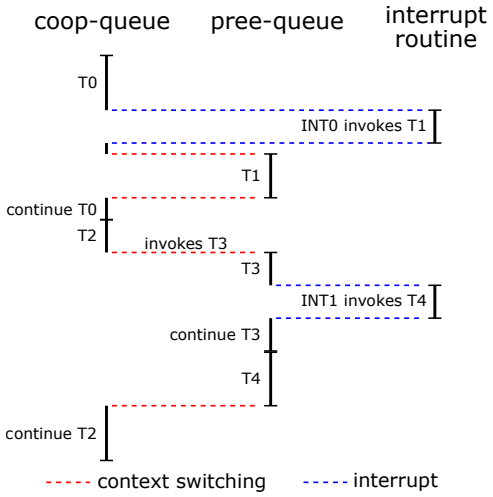


Figure 6. Task-scheduling demonstration, where the label *T0* and *T2* denote regular tasks from the coop-queue; *T1*, *T3*, and *T4* denote the preemptive tasks from the pree-queue; and *INT0* and *INT1* denote the hardware interrupts

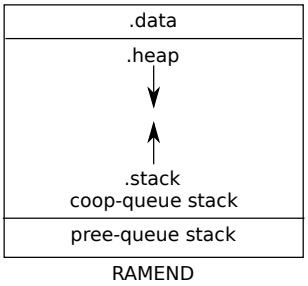


Figure 7. Stack memory distribution

Figure 7, the pree-queue stack is allocated to the RAMEND (the end of random-access memory). In this way, the wasted memory used to initialize the system is reused as the pree-queue stack. The coop-queue uses a native memory stack, and the stack size is dynamic until it overlaps with the stack *.heap*.

D. Resource management

Resource management modules are essential for managing hardware resources. The *power manager* controls the system's power-saving level based on the kernel status. Primitive events/signals from hardware are managed/filtered by the *interrupt monitor*. The *system timer* provides system timing functions, such as the system tick service and the primitive timer event.

StateOS applies dynamic memory management using a *memory allocator*. The memory allocator distinguishes the memory allocating requests as long-term and short-lived requests and applies different algorithms to each. This strategy effectively alleviates internal and external fragmentation problems.

The traditional heap-based *malloc* algorithm (as implemented in the standard C library) allocates memory in a dynamically growing data segment. This algorithm is efficient in processing long-term requests. However, frequent short-lived requests can cause the external fragmentation problem [25]. In contrast, *buddy system* [26] divides the memory pool into fix-sized

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3234106

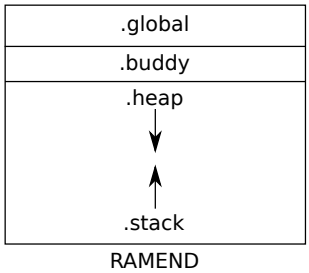


Figure 8. Memory sections

memory blocks to alleviate the external fragmentation problem [27]. However, when long-term memory requests occupy the memory pool, this can lead to an internal fragmentation problem.

StateOS allocates separate memory sections to fulfill short-lived and long-term requests. As shown in Figure 8, the dynamically growing memory pool *.heap* satisfies long-term requests using a heap-based *malloc* algorithm, and the static memory pool *.buddy* implements a variant of the buddy system for short-lived requests. This hybrid strategy improves efficiency with regard to dynamic memory management by alleviating fragmentation problems.

The buddy variant maintains a binary tree that monitors the states of the memory blocks. The operations of the binary tree have a time complexity $O(logN)$, where N is the number of memory blocks. The worst-case scenario is allocating/freeing memory that is smaller than a memory block because the operation has to traverse the tree to reach the bottom leaf. It is inefficient to repeatedly issue small memory requests to a buddy allocator because the algorithm processing time diminishes the system's performance. To alleviate this problem, a memory recycling system is implemented to temporarily hold the recently freed small memory chunks without restoring them to the binary tree. These memory chunks can be quickly reassigned to new requests. However, holding these memory spaces in the long term may cause an unbalanced binary tree and increase internal fragmentation. For this reason, the recycled memory blocks are dumped periodically at the system's convenience.

V. A CODE EXAMPLE

An example code, shown as Code 2, is a code snippet that initializes a radio transceiver. It contains the radio-initializing task *radio_init_task* and the startup task *start_demo*.

The task implementation begins with the task prototype interface *TASK*, followed by the return type, task name, and variadic arguments. The task prototype *TASK(int, radio_init_task, int, timeout)*, in line 1, contains the return value type *int*, the task name *radio_init_task*, and an argument *int, timeout*. The reader may notice that the argument's type *int* and label *timeout* are separated by a comma. This is because the type and label of an argument are treated as a pair of parameters in the prototype.

The task *STARTUP_TASK(start_demo)*, as in line 17, implements a startup task. This task calls the subroutine task *radio_init_task* in line 19 and is blocked until the called task is completed.

The task *radio_init_task* contains the local variable *ts* across the yield operation *TASK_WAIT_WHILE*, as in line 10. This local variable must be introduced to the yield operation for preservation; otherwise, the value of the variable is no longer guaranteed when the task resumes.

The task invoked by the synchronized task-calling macro *TASK_CALL* can return the result as a *return value*. In the example, the task *radio_init_task* has the return value of an integer *int*. There are two return operations in the example, as can be seen in lines 13 and 14. The return value is read by the macro *TASK_CALL_RETVAL*.

Code 2. A radio initialization program example

```
1  TASK(int, radio_init_task, int, timeout)
2  {
3    time_t ts = os_get_time() + timeout;
4
5    radio_init();
6
7    TASK_WAIT_WHILE(
8      (radio_state() != RADIO_READY) &&
9      (ts < os_get_time()),
10     ts);
11
12   if (radio_state() != RADIO_READY)
13     TASK_EXIT(-1);
14   TASK_END(0);
15 }
16
17 STARTUP_TASK(start_demo)
18 {
19   TASK_CALL(radio_init_task, 100)();
20   int res = TASK_CALL_RETVAL(int);
21   if (res == 0) dbg_printf("radio OK\n");
22
23   TASK_END();
24 }
```

This example demonstrates that StateOS's cooperative task APIs provide a practical approach to cooperative threaded programming. It is flexible, memory efficient, and easy to use.

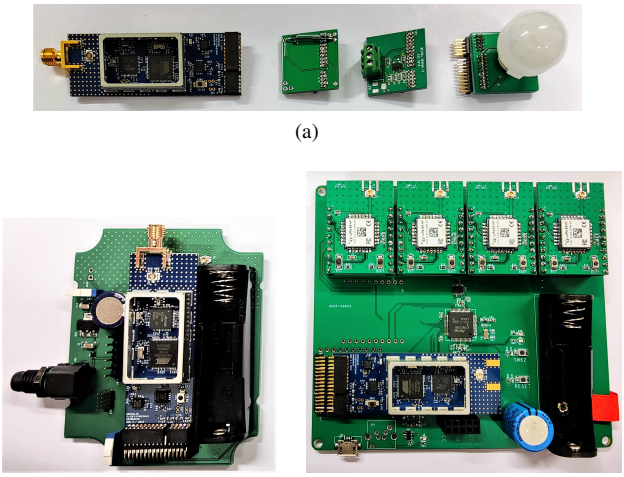
VI. IMPLEMENTATION

StateOS has been implemented as an IoT solution on different platforms. Figure 9a shows the baseboard and several sensor modules, including a relay-based magnet sensor, a pressure sensor, and a motion sensor. The baseboard includes an XMEGA256A3Bu MCU and an AT86RF215M transceiver. Using this modularized design, an IoT sensor device can be assembled simply by attaching the sensor module to the baseboard. These sensor modules are supported by the sensor service modules of StateOS.

Network management is achieved using autonomous network services, which include a network scheduler and clock synchronization. The network scheduler synchronizes the radio activities of all IoT devices on the network. This reduces a device's power consumption by minimizing the radio's active period. The clock synchronization service fine-tunes the device's local clock according to the central clock.

The database of the cross-layer management entity maintains a network status list *neighbor-table* by recording broadcasts from neighboring devices. The table collects information about these devices, which includes their battery level, signal strength, link quality, and traffic throughput. This information can be used by network services, such as topology management and traffic control services. Furthermore, the neighbor-table is uploaded to the server for network diagnosis.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3234106



(a) (b) (c)

Figure 9. The StateOS platform examples include (a) a baseboard with sensor modules, (b) a LoRa gateway board, and (c) a heavy-duty LoRa gateway board.

The LoRa [28] gateway board, shown in Figure 9b, is an extension of the baseboard that supports LoRa communication using an RN2483 LoRa module. It is mainly used to upload gateway data to the service. The LoRa protocol has limited bandwidth, especially over a poor-quality network link. Thus, StateOS further implements a data aggregation module and a compression module to reduce LoRa network traffic.

Some applications demand increased bandwidth that can exceed the LoRa capacity. Therefore, for larger network traffic throughput, we implemented heavy-duty gateway boards, as shown in Figure 9c. This platform is mounted with four LoRa-E5 modules and an ESP8266 Wi-Fi module. This gateway has its own MCU (which also runs StateOS), ARM M0+ ATSAML21G18B, for network management.

VII. Evaluation

In this section, StateOS is evaluated based on its technical properties, scalability, and performance. The technical properties of an IoT OS include the kernel architecture, scheduling strategy, programming paradigm, programming language, and real-time capability. The scalability of an OS is measured by comparing the data memory consumption and program memory engagement. The performance is determined by calculating the processing time of the kernel operations.

Traditional resource-constrained IoT/WSN platforms are powered by an 8-bit MCU, and the typical IoT/WSN-oriented OSs were originally designed under the 8-bit computing architecture. Therefore, in the scalability and performance subsections, evaluations are performed by comparing operating systems that support 8-bit processor families (e.g., AVR and PIC processors) with comparable performance metrics. The evaluation data for StateOS were obtained on the platform specified in Table II. Evaluation data for other OSs were obtained in literature research.

A. Technical properties

Table III lists the technical properties of different IoT OSs. The kernel architecture choice significantly influences an OS's overall architecture and modularity. StateOS applies a microkernel architecture for a small kernel size and a modularized structure. The system modules are loosely coupled, which achieves a flexible and robust architecture.

Table II
Specifications of the evaluated StateOS platform

Platform	StateOS sensor baseboard
MCU	Xmega256A3Bu
CPU clock	32MHz
pree-queue stack	128 bytes
memory pool	1024 bytes
minimum memory block	8 bytes
memory recycling pool	20 bytes (10 entries)

StateOS implements a hybrid task-scheduling strategy to support cooperative threaded programming at a small memory cost while maintaining adequate real-time capability. The cooperative threaded interfaces in StateOS are provided by the system-specific language. It can be challenging for novice programmers. Therefore, StateOS extensively supports state-charts as a state machine-based visual programming model.

B. Scalability

The scalability of an OS is evaluated by memory usage for handling concurrent tasks/threads. The evaluation is conducted with a methodology similar to [23]. In the evaluation, we run 16 cooperative tasks on StateOS, which are contained by two kernel threads (the pree-queue and the coop-queue).

A typical StateOS configuration takes 46 bytes of static data memory, which is its kernel's memory footprint that includes the control blocks of two kernel threads (the pree-queue and the coop-queue). The task scheduler typically allocates 128 bytes of stack memory for pree-queue context saving. Additionally, the buddy memory module requires extra management memory of 19 bytes and binary tree memory of M/B bytes, where M is the memory pool size, and B is the memory block size. It is typical to configure a dynamic memory pool of 1024 bytes with a block size of 8 bytes. Thus, the buddy module takes 147 ($19+1024/8$) bytes of heap memory as the management cost. Furthermore, the memory recycling system can be optimized to take 20 bytes to implement ten recycling entries. In summary, a functional StateOS requires 341 ($46+128+147+20$) bytes of data memory (the memory footprint of the hardware implementation is not counted).

In StateOS, a TCB takes a minimum of 22 bytes of memory. Therefore, a running StateOS with 16 concurrent tasks consumes 693 ($22 * 16 + 341$) bytes of memory, which includes 352 bytes of dynamic memory from 16 TCBs and 341 bytes of system memory consumption. However, this estimation is based on the minimum task profile, with no arguments nor local variables, and the results are suggestive of estimating the system's memory usage.

StateOS takes a minimum of 13K bytes of flash memory, which primarily involves kernel implementation. However, a typical configuration of StateOS consumes more memory to satisfy the application requirements. For example, the StateOS implementation in Section VI consumes 59K bytes of flash

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3234106

Table III
COMPARISON OF TECHNICAL PROPERTIES

	Architecture	Scheduling	Programming paradigm	Language	Real-time	Supported processor families
StateOS	microkernel	hybrid	cooperative threaded, statecharts[a]	C, statecharts	partial supported	AVR, ARM Cortex-M
Contiki	modular	cooperative	cooperative threaded[b], multithreaded[c], event-driven	C	partial supported	AVR, ARM Cortex-M, MSP430, PIC, etc.
RIOT	microkernel	preemptive	multithreaded	C, C++	supported	AVR, ARM Cortex-M/A9, Tensilica Xtensa, RISC-V, etc.
FreeRTOS	microkernel	preemptive	multithreaded	C	supported	AVR, ARM Cortex-M, Tensilica Xtensa, RISC-V, etc.
OpenWSN	monolithic	cooperative	event-driven	C	not supported	AVR, ARM Cortex-M
HybridKernel	monolithic	hybrid	cooperative threaded[d]	C	partial supported	PIC
TinyOS	monolithic	cooperative	cooperative threaded[e], multithreaded[f], event-driven	nesC	not supported	AVR, ARM Cortex-M, MSP430, px27ax, etc.
MANTIS OS	layered	preemptive	multithreaded	C	supported	AVR
SenSpire	monolithic	hybrid	event-driven	CSpire[g]	partial supported	AVR, MSP430
Zephry	monolithic	preemptive	multithreaded	C, C++	supported	ARM Cortex-M/R, ARC, MIPS RISC-V, SPARC, ARC Tensilica Xtensa, NIOS II, etc
Mbed OS	monolithic	preemptive	multithreaded	C, C++	supported	ARM Cortex-M

[a] Statecharts are supported by the statechart framework [11]
[b] Contiki supports cooperative threaded programming using Protothreads [5] .
[c] Contiki supports multithreaded programming by a library on top of the event-based kernel [5].
[d] HybridKernel supports cooperative threaded programming using Protothreads [23] .
[e] TinyOS accepts cooperative threaded programming using the library extension TinyThreads [6] .
[f] There are many multithreaded solutions for TinyOS. The *official* one is TOSThreads.
[g] CSpire is an object-oriented programming language that extends C++ [20].

memory, which includes the kernel (13K bytes), hardware implementation (15K bytes), network services (24K bytes), sensor services (2K bytes), and miscellaneous components (5K bytes).

In Table IV, we compare the multiple task overhead of StateOS to a multithreaded solution (MANTIS OS) and three hybrid solutions (Contiki with Protothreads, TinyOS with TOSthreads, and HybridKernel). In this evaluation, the hybrid solutions (including StateOS) are evaluated with 16 cooperative components and two preemptive components, and the multithreaded solution executes 16 preemptive components. To distinguish between processes and threads in this evaluation, we define threads as being cooperative and consuming memory from TCBs, whereas processes are preemptive and consume memory from Process Control Blocks (PCB). The results suggest that StateOS is a memory-efficient approach to implementing multitask systems.

C. Performance

The performance of StateOS is evaluated based on the processing time of the task APIs. Most task operations involve dynamic memory management. The approximate processing time for a short-lived and small-sized memory allocation is 40 μs and for memory free is 41 μs. This performance can be promoted by the memory recycling system to obtain a memory allocation of 16 μs and a memory free of 30 μs.

When the scheduler dispatches a task, it takes 12 μs to establish the task. In addition, preparing an 8 or 16-bit task argument takes less than 1 μs. However, preparing a 32-bit argument consumes a higher amount of processing time of 3 μs. The task yield operations take an average of 4 μs to release the processor control. The other task flow control operations, such as *TASK_WAIT_DELAY* and *TASK_CALL*, can have a processing time of 40–60 μs. Furthermore, if local variables are saved during the operation, the processing time increases because of the dynamic memory operations.

The cooperative task-switching procedure includes a flow control operation, a task establishment operation, and possible stack management operations. In summary, the processing time required for a cooperative task-switching operation can be a minimum[2] of 16 μs and a maximum of more than 100 μs. Cooperative task-switching operations are usually issued by tasks with low urgency levels. Therefore, its processing speed is sufficient for such tasks.

Compared to cooperative task switching, preemptive context switching between the coop-queue and the pree-queue is faster. It takes about 4 μs to switch between the task queues, which is quick enough for a time-sensitive task to be executed in time.

The results for the comparison of the scheduling overhead of task switch operations in cooperative and preemptive scheduling are shown in Table V. The overhead values are platform-dependent. Therefore, the comparison is feasible if the results are unified with the CPU clock cycles.

Cooperative switch time is the scheduling overhead between consecutive cooperative tasks. StateOS APIs provide semi-automatic stack management and versatile task controls (e.g., task concatenation, task callback, and mutex) that are processed between tasks. This user-friendly approach requires more executive time than simple task switches. However, typical cooperative tasks can tolerate longer latency in exchange for flexibility.

[2]This includes a basic yield operation (4 μs) and a task establishment operation (12 μs).

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3234106

Table IV
MULTITASK HANDLING MEMORY COMPARISON[a]

(in bytes)	Static memory	PCB	PCB total	Stack	Stack total	TCB	TCB total	Sum
StateOS	341	-	-	-[b]	-	22	352 (16 * 22)	693
Contiki[c]	262[29] + 128[d][4]	8[4]	16 (2 * 8)	128	256 (2 * 128)	15[29]	240 (16 * 15)	902
TinyOS[e]	16[30]	43[31]	129 (3 * 43)	128	384 (3 * 128)	46[23]	736 (16 * 46)	1265
MANTIS OS[f]	144[32]	10[32]	160 (16 * 10)	128	2048 (16 * 128)	-	-	2352
HybridKernel	89[23]	28[23]	56 (2 * 28)	128	256 (2 * 128)	31[23]	496 (16 * 31)	897

[a] This comparison is conducted by executing 16 concurrent and two preemptive parts in the system. Some OSs may require an additional thread or stack in the kernel.
[b] The StateOS' pree-queue stack is counted as static memory.
[c] Contiki with Protothreads and a multithreaded library.
[d] Contiki kernel requires an extra stack.
[e] TinyOS with TOSThreads requires an extra kernel thread hosting the event system. Thus, it counts as three preemptive parts in total.
[f] MANTIS OS executes 16 preemptive threads because it supports no cooperative tasks.

Table V
COMPARISON OF SCHEDULER OVERHEAD

	Cooperative switch (in clock cycles)	Preemptive switch (in clock cycles)
StateOS	480	120
TinyOS[a]	≈10[b]	184[19]
Contiki[c]	106[5]	-
MANTIS OS	-	≈400[32]
SenSpire	130[20]	400∼420[20]
HybridKernel	106[d][5]	360[33]

[a] TinyOS is evaluated with TOSThreads.
[b] TinyOS has a first-in-first-out task queue with a run-to-the-end strategy. We had trouble searching the literature results for the task switch time between the consecutive tasks in the task queue. However, a task switch operation has a few instructions about memory copying and function calling, similar to event posting and command calling operations (both of them take 10 clock cycles). Therefore, we assume that the cooperative switch in TinyOS is around 10 clock cycles.
[c] Contiki is evaluated with Protothreads.
[d] HybridKernel manages cooperative tasks using Protothreads. The results are taken from Protothreads.

Real-time capability can be estimated by the performance of preemptive task switch operations. This reflects the guaranteed maximum time of the system's responsiveness. Of the results for the solutions shown in the table, StateOS takes the least amount of preemptive task-switching time. This shows that StateOS provides sufficient real-time performance for IoT/WSN applications.

VIII. DISCUSSION AND CONCLUSION

This paper presents an embedded OS targeting IoT and wireless sensor devices with distinctive features. The kernel provides a set of macro-based cooperative task APIs that allows for the modeling of event-driven systems in a threaded paradigm. The hybrid task scheduler supports a mix of various scheduling algorithms, such as cooperative, priority-based, and preemptive scheduling. It allows the programmer to balance resource usage and real-time performance based on the application's specifications. The dynamic memory allocator in StateOS implements two separate strategies for long- and short-lived allocation requests. This method can alleviate the fragmentation problem and improve the robustness of the system in the long term. Automatic stack management is typically the privilege of multithreaded OSs. The users of event-driven and cooperative OSs have to manually pass the parameters between tasks and protect local variables. StateOS' APIs provide semi-automatic stack management that automatizes the process of parameter passing and local variable preservation.

A design principle of StateOS is memory efficiency. To this end, several technologies are applied, including a cross-layer communication structure, modularized services, and microkernel architecture. Therefore, the memory occupation of the system is configurable, depending on the application's requirements.

StateOS works perfectly with a statechart visual programming framework [11] that supports modeling and programming wireless sensor programs using graphic statechart diagrams. This combination provides an alternative visual programming approach that can aid developers in creating IoT applications efficiently.

The hybrid approach to kernel design is, in fact, a compromised approach that combines event-driven and multithreaded systems. The system may consume more memory and executive time than event-driven solutions. On the other hand, multithreaded systems need no attention from users for stack management. Compared to this, StateOS implements a semi-automatic stack management solution, requiring users to manually identify local variables. Furthermore, StateOS adopts a dynamic memory management strategy. It has a trade-off of the overhead of managing memory spaces.

StateOS was initially designed for WSN-based solutions, where the gateway handles internet protocol (IP)-based network traffics. Our following works include extending the network stack to support low-power IP protocols such as 6LoWPAN [34], allowing individual access to WSN nodes through an IP-based IoT network directly. It will enable the use of IP-based IoT application protocols, such as Thread [35] and Matter [36], and emerge StateOS as a part of the modern IoT ecosystem. Moreover, the hardware implementations are limited to a few MCU models. We will extend the hardware implementations to other popular IoT MCUs and platforms in the following works.

In conclusion, we proposed a hybrid approach to program-

ming IoT and wireless sensor applications in a threaded paradigm with less memory consumption. We expect the proposed solution to be a viable instrument that aids modern IoT application development.

References

[1] Y. Wu, P. Wang, and C. Xu, "Improving visible light backscatter communication with delayed superimposition modulation," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–3.

[2] L. Knud, Lasse, "State of the iot 2020: 12 billion iot connections, surpassing non-iot for the first time," *IoT analytics*, 2020. [Online]. Available: https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/

[3] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*, "Tinyos: An operating system for sensor networks," in *Ambient intelligence*. Springer, 2005, pp. 115–148.

[4] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 2004, pp. 455–462.

[5] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems*. Acm, 2006, pp. 29–42.

[6] W. P. McCartney and N. Sridhar, "Abstractions for safe concurrent programming in networked embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006, pp. 167–180.

[7] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 289–302.

[8] O. Kasten and K. Römer, "Beyond event handlers: Programming wireless sensors with attributed state machines," in *Proceedings of the 4th international symposium on Information processing in sensor networks*. IEEE Press, 2005, p. 7.

[9] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister, "Openwsn: a standards-based low-power wireless development environment," *Transactions on Emerging Telecommunications Technologies*, vol. 23, no. 5, pp. 480–493, 2012.

[10] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005, pp. 163–176.

[11] I. Hakala and X. Tan, "A statecharts-based approach for wsn application development," *Journal of Sensor and Actuator Networks*, vol. 9, no. 4, p. 45, 2020.

[12] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han, "Mantis: System support for multimodal networks of in-situ sensors," in *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. ACM, 2003, pp. 50–59.

[13] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot: An open source operating system for low-end embedded devices in the iot," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[14] "Freertos," https://www.freertos.org/, accessed: 2021-12-01.

[15] "Zephyr," https://zephyrproject.org/, accessed: 2021-12-01.

[16] "Mbed os," https://os.mbed.com/mbed-os/, accessed: 2021-12-01.

[17] "Keil rtx5," https://www2.keil.com/mdk5/cmsis/rtx/, accessed: 2021-12-01.

[18] E. Trumpler and R. Han, "A systematic framework for evolving tinyos," in *IEEE Workshop on Embedded Networked Sensors*. Citeseer, 2006, pp. 61–65.

[19] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-Elefteri, P. Levis, A. Terzis, and R. Govindan, "Tosthreads: thread-safe and non-invasive preemption in tinyos." in *SenSys*, vol. 9, 2009, pp. 127–140.

[20] W. Dong, C. Chen, X. Liu, Y. Liu, J. Bu, and K. Zheng, "Senspire os: A predictable, flexible, and efficient operating system for wireless sensor networks," *IEEE transactions on computers*, vol. 60, no. 12, pp. 1788–1801, 2011.

[21] Y. Guan, J. Guo, and Q. Li, "Formal verification of a hybrid iot operating system model," *IEEE Access*, vol. 9, pp. 59 171–59 183, 2021.

[22] S. M. Trenkwalder, Y. K. Lopes, A. Kolling, A. L. Christensen, R. Prodan, and R. Groß, "Openswarm: An event-driven embedded operating system for miniature robots," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 4483–4490.

[23] T. Laukkarinen, V. A. Kaseva, J. Suhonen, T. D. Hamalainen, and M. Hannikainen, "Hybridkernel: Preemptive kernel with event-driven extension for resource constrained wireless sensor networks," in *2009 IEEE Workshop on Signal Processing Systems*. IEEE, 2009, pp. 161–166.

[24] I. Hakala and M. Tikkakoski, "From vertical to horizontal architecture: a cross-layer implementation in a sensor network node," in *Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*. ACM, 2006, p. 6.

[25] A. Bohra and E. Gabber, "Are mallocs free of fragmentation?" in *USENIX Annual Technical Conference, FREENIX Track*, 2001, pp. 105–117.

[26] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, no. 10, pp. 623–624, Oct. 1965. [Online]. Available: http://doi.acm.org/10.1145/365628.365655

[27] J. L. Peterson and T. A. Norman, "Buddy systems," *Communications of the ACM*, vol. 20, no. 6, pp. 421–431, 1977.

[28] "Lora," https://www.lora-alliance.org, accessed: 2021-12-01.

[29] "Contiki source code," https://github.com/contiki-os/contiki, accessed: 2021-12-01.

[30] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *ACM Sigplan notices*, vol. 35, no. 11, pp. 93–104, 2000.

[31] "Tinyos source code," https://github.com/tinyos/tinyos-main, accessed: 2021-12-01.

[32] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 563–579, 2005.

[33] T. Laukkarinen, "Abstracting application development for resource constrained wireless sensor networks," Ph.D. dissertation, Tampere University of Technology, 2015.

[34] G. Mulligan, "The 6lowpan architecture," in *Proceedings of the 4th workshop on Embedded networked sensors*, 2007, pp. 78–82.

[35] "Thread," https://www.threadgroup.org/, accessed: 2022-11-17.

[36] "Matter," https://github.com/project-chip/connectedhomeip/, accessed: 2022-11-17.