MDPI

Article

A Novel Approach to Managing System-on-Chip Sub-Blocks Using a 16-Bit Real-Time Operating System

Boisy Pitre * and Martin Margala *

School of Computing and Informatics, The University of Louisiana at Lafayette, 301 Lewis Street, Lafayette, LA 70503, USA
* Correspondence: boisy.pitre1@louisiana.edu (B.P.); martin.margala@louisiana.edu (M.M.)

Abstract: Embedded computers are ubiquitous in products across various industries, including the automotive and medical industries, and in consumer goods such as appliances and entertainment devices. These specialized computing systems utilize Systems on Chips (SoCs), devices that are made up of one or more main microprocessor cores. SoCs are augmented with sub-blocks that perform dedicated tasks to support the system. Sub-blocks contain custom logic or small-footprint microprocessors, depending upon their complexity, and perform support functions such as clock generation, device testing, phase-locked loop synchronization and peripheral management for interfaces such as a Universal Serial Bus (USB) or Serial Peripheral Interface (SPI). SoC designers have traditionally obtained sub-blocks from commercial vendors. While these sub-blocks have well-defined interfaces, their internal implementations are opaque. Without visibility of the specifics of the implementation, SoC designers are limited to the degree to which they can optimize these off-the-shelf sub-blocks. The result is that power and area constraints are dictated by the design of a third-party vendor. This work introduces a novel idea: using an open-source, small, multitasking, real-time operating system inside an SoC sub-block to manage multiple processes, thereby conserving code space. This OS is TurbOS, a new operating system whose primary goal is to provide the highest performance using the least amount of space. It is written in the assembly language of a new pipelined 16-bit microprocessor developed at the University of Florida, the Turbo9. TurbOS is derived from and incorporates the design benefits of an existing operating system called NitrOS-9, and reduces the code size from its progenitor by nearly 20%. Furthermore, it is over 80% smaller than the popular FreeRTOS operating system. TurbOS delivers a rich feature set for managing memory and process resources that are useful in SoC sub-block applications in an extremely small footprint of only 3 kilobytes.

Keywords: system on a chip; sub-block optimization; real-time operating system; 16-bit microprocessor; open source; pre-emptive multitasking

Citation: Pitre, B.; Margala, M. A Novel Approach to Managing System-on-Chip Sub-Blocks Using a 16-Bit Real-Time Operating System. *Electronics* **2024**, *13*, 1978. https://doi.org/10.3390/electronics13101978

Academic Editor: Lukáš Kohútka

Received: 20 April 2024
Revised: 15 May 2024
Accepted: 16 May 2024
Published: 18 May 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Embedded systems power devices across a range of fields, from safety to health [1]. The global market for these systems is expected to reach approximately USD 114 billion in 2024 [2]. Strategies exist to optimize these ubiquitous devices by reducing their area and the power they consume. At scale, such optimizations can result in quantifiable reductions in the energy consumed over time, as well as the materials (rare earth minerals and chemicals) required to produce them. The resulting savings yield conservation benefits that promote environmental sustainability, green goals, and stewardship of our planet.

SoCs are central actors in modern embedded systems. They are composed of one or more microprocessor cores and are often 32-bit or 64-bit in size. As the name implies, a System on a Chip is an entire integrated computer system that is made up of additional components called sub-blocks. They provide support functionality such as communica-

tion interface controllers and device testing components. Given the copious amount of functionality in SoCs, there is ample opportunity to optimize them for area and power.

One optimization opportunity is to reduce the number of sub-blocks by process consolidation through the employment of a real-time operating system (RTOS) [3] that manages multiple tasks and delivers predictable timing performance for applications. Another optimization strategy is reducing the sub-block's area and power utilization by employing smaller footprint microprocessors in them, such as a 16-bit or smaller architecture, versus a larger 32-bit one. While the main core or cores can still be 32-bit or even 64-bit for main applications, the narrow scope of work of sub-blocks makes them ideal for much smaller processors with minimal address spaces.

There is also an opportunity to open the current closed-source, opaque model, where SoC sub-blocks are purchased and integrated from vendors who preserve their intellectual property without providing any visibility into their implementation or their inner workings. A completely open-source RTOS and application suite for SoC sub-blocks can provide transparency and community-based opportunities for further area and power optimizations.

1.1. Evolution of Operating Systems and Their Sub-Block Potential

The operating system is at the heart of computers that run software applications. This critical software component offers important services such as input and output handling, memory and file management, process scheduling, and interprocess communication facilities. These services provide a base of common routines that the application software does not need to concern itself with implementing. The operating system acts as the arbiter and vendor of system resources through API calls, allowing the developer to focus on creating the application.

An RTOS is a specific class of operating system that promises to deliver execution on specific deadlines. RTOSs are common in critical applications that require precise timing. An RTOS is usually significantly smaller in size than an operating system designed for desktop or server computing.

In the early history of small-footprint microprocessors, operating systems were preceded by monitors. Monitors were system-level applications typically written in the assembly language of the processor. Monitors provided simplistic debugging services and some form of object code loading and execution. Small in size and usually existing onboard a read-only memory (ROM), the monitor provided enough intelligence for the microprocessor to execute code for a given task, but little more.

As demands for more application functionality increased, desktop operating systems began to emerge. A few examples are CP/M [4] and MS-DOS [5]. These operating systems were written in assembly language for the specific processor, as opposed to UNIX, which was written in C for portability.

Later, the advent of 32- and 64-bit processors provided increased memory sizes, both for the operating system itself and for the applications it managed. This in turn led to more complexity in both the features that the operating system offered, as well as the sophistication of the applications that ran under them.

At the same time, such increases in available resources led to the use of high-level languages such as C and C++ in the development of both the operating system and the applications. While these languages are near "bare-to-metal", they nonetheless contributed to the growth in size of the operating system and its related commands and tools. Recent research has focused on bringing full-featured desktop operating systems to the real-time embedded space [6] or implementing an RTOS completely in hardware [7]. Such research can benefit the area of SoC sub-blocks.

There is also the question of using a high-level language such as the C programming language for operating system development. The motivation for this is portability to other microprocessor architectures. While using a high-level language abstracts the architecture of the processor and makes porting both operating systems and software to other systems

more tenable, the resulting increase in code size adds pressure to the size of the system. In applications where smaller-memory-footprint microprocessors are desirable, with an operating system to manage resources and provide multi-tasking and process scheduling, the smallest size is achieved using assembly language. Portability becomes less of a concern when the application is based on a single microprocessor that is baked in silicon, as SoC sub-blocks are, and is used for a specific purpose.

1.2. Research Question and Contributions

There is a direct correlation between an SoC's area and its power consumption; reducing the former has a direct impact on the latter. The primary way to minimize the sub-block area is to focus on a reduction in logic gates. This can be accomplished by reducing the amount of memory that an SoC uses, both in terms of random access memory and read-only memory. This can be achieved by decreasing functionality, but this is not desirable. We seek to maximize the functionality of an SoC while at the same time decreasing its area. Power consumption will then naturally decrease. The research question we seek to answer in this work is: how might a reduction in both area and power utilization of an SoC and its sub-blocks be achieved while maintaining performance, meeting feature requirements, and promoting design transparency?

The work into answering this research question has yielded TurbOS: an open-source, small-footprint, pre-emptive multitasking operating system suited for SoC applications, along with a methodology for determining the appropriate hardware and software for SoC sub-blocks.

This paper reviews the previous literature on SoCs with regard to area and power; provides a survey of the Turbo9 microprocessor and its accompanying TurbOS operating system; provides a methodology for decreasing the size of TurbOS for further optimization; and presents a size comparison of FreeRTOS, a widely available open-source operating system used in a large number of modern embedded applications. The results and conclusions then follow.

2. Literature Review and Prior Work

Power and area optimizations for applications under real-time operating systems on SoCs have been explored in the literature, but none have focused specifically on sub-blocks within an SoC. Some have focused on the operating system running on the main SoC processor to manage specific use cases in critical applications, the critical applications themselves, SoC security, and design and verification tooling improvements.

Rane and Panem et al. [8] discuss the use of an RTOS to perform task management and traffic synchronization on a specific SoC known as a Network on a Chip (NoC). Vargas' tutorial on on-chip cross-layer infrastructure focuses on an RISC-V SoC with an RTOS for aerospace applications [9]. Krishnan and Wan et al. [10] explore domain-specific design SoCs (DSSoCs) for autonomous unmanned aerial vehicles (UAVs). In their work, they discuss the challenges of the complexity between sensors and computational power and propose a machine-learning-based methodology to explore efficient design. This work underscores the critical nature of optimizing the power and area of SoCs in applications such as self-driving cars and autonomous aerial vehicles, but only addresses the applications running on the main processor cores.

Specific real-time operating systems for SoC applications have also been explored. Shang focuses on the µC/OS-II RTOS [11] in their work [12]. Zhu, Zhou, Liu, and Dai employ a real-time operating system on a 32-bit SoC utilizing the stack-oriented Forth programming language [13]. Larrea and Barbalace propose Serverkernel, an operating system that blends ideas from Unix-like kernels and real-time operating systems for IoT devices [14]. All of these operating systems are written in the C programming language, and target a 32-bit SoC.

Security and performance are important areas affecting real-time operating systems in embedded applications. Wu, Zheng, Zeng, Gao, and Xiong propose a SystemC-based

cryptographic SoC virtual prototyping platform to accelerate the design and verification of embedded security devices [15]. While this work may have applicability in SoC sub-block applications, a tightly sealed silicon-based SoC sub-block with well-defined, stringent interfaces with the main SoC subsystem can avoid the overhead of additional security logic. The work of Orenes-Vera and Manocha et al. focuses on memory performance in SoCs that have a large number of cores [16]. Chi and Lin explore SoC simulation modeling that could be helpful in creating optimized SoCs along with an RTOS [17].

The literature is replete with SoC area and performance optimizations using 32-bit or larger processors, but there is a dearth of work focused specifically on using a real-time operating system to optimize power and area through the use of efficient process allocation on smaller 16-bit microprocessors. Moreover, there is not a focus on the most interesting part of the SoC that could benefit from such an optimization: the sub-block.

3. Approach

In order to determine the feasibility of using a real-time operating system for SoC sub-blocks, several factors must be considered:

- Microprocessor selection.
- Operating system selection.
- An application survey.
- Optimization methods.
- Performance comparison.

3.1. Microprocessor Selection

The target microprocessor proposed in this work is the Turbo9 [18], a pipelined 16-bit addressable microprocessor with a complex instruction set computer (CISC) architecture. The Turbo9 is based on the architecture of the Motorola 6809 [19,20], an 8/16-bit microprocessor with a compact yet versatile orthogonal instruction set with a maximum addressable space of 64 K. The Turbo9 register file in Figure 1 shows two 8-bit accumulators (A and B which combine to form D), three 16-bit index registers (X, Y, and U), a 16-bit system stack pointer (S), and a 16-bit program counter (PC). There is an 8-bit direct page (DP) register that can be loaded with the upper 8 bits of a 16-bit address; this allows for shorter instruction lengths and fast access within a 256 byte region (known as a page). The Turbo9 adapts all of the addressing modes of the 6809 for accessing data directly from memory or indexed off of an index register. Branching instructions support both absolute and relative addressing; the latter allows for position-independent code to be written. Position-independent code can be loaded anywhere within the address space of the processor. The Turbo9 also improves upon the performance of 6809 by implementing an instruction execution pipeline that allows for interleaving of instructions as they are executed.

The Turbo9 offers a contrast to the 32-bit reduced instruction set computer (RISC) architecture that is used as default in designs today. This avenue is less represented in the literature; this is precisely why it is of interest to explore this model of microprocessor and see what benefits it may yield.

3.2. Operating System Selection

TurbOS is the operating system specifically designed for the Turbo9. It was derived from NitrOS-9 [21], which includes a number of innovative features for a small-footprint microprocessor, including pre-emptive multitasking, interprocess communication, and a module structure which allows code to be located anywhere in the memory and reused by multiple processes. It is loosely based on Unix and incorporates a number of fundamental choices in that operating system [22]. A summary of the major system calls and parameters is shown in Table 1.

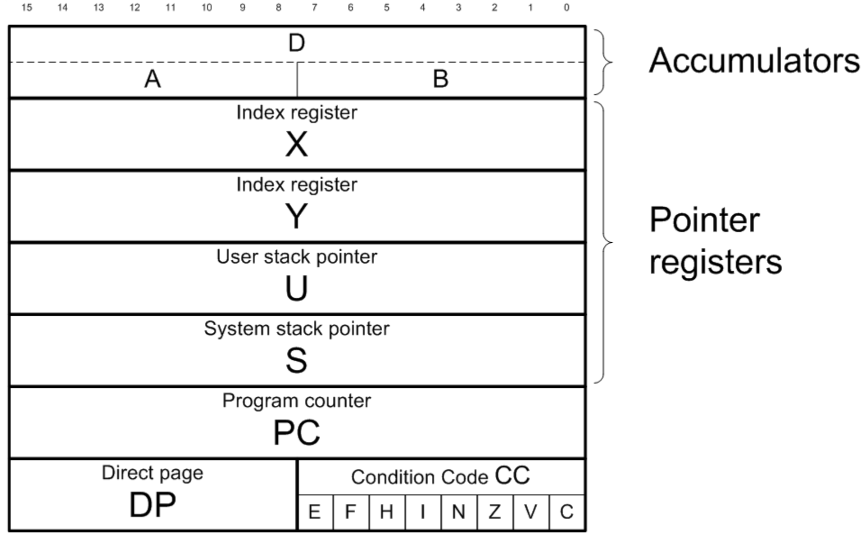

Figure 1. Turbo9 register file.

Table 1. Major system calls in TurbOS.

System Call	Parameters	Returns
`F$Fork`	module name	new process instance
`F$Wait`	process ID	process ID of terminated child
`F$Link`	module name	module pointer
`F$Unlink`	module pointer	error status
`F$All64`	number of pages	pointer to allocated pages
`F$Ret64`	pointer to allocated pages	error status
`F$SRqMem`	number of bytes	pointer to allocated bytes
`F$SRtMem`	pointer to allocated bytes	error status

Selecting an operating system that is open source encourages contributions from others for study and improvement. An open-source model for sub-blocks is opposite to the closed-source, black-box model of existing IP sub-block vendors. While the idea of an open-source framework for SoC applications has been explored before (Mantovani and Giri et al. explored an open-source SoC platform with Open ESP [23]), there is no 16-bit open-source project specifically focused on SoC sub-blocks. TurbOS is an open-source project that encourages collaboration and innovation from a worldwide group of contributors. The source code for both system-level software (kernel, support modules, etc.) and the application code is available on GitHub. GitHub offers a rich set of services that allow for collaboration on and extension of the TurbOS project. The preferred method is for an interested contributor to create a fork in the TurbOS repository. This fork gives the contributor a copy of the source code repository on their own GitHub account. They can experiment by making changes that they can easily discard or decide to improve upon. Once their changes are committed to their fork, they can create a pull request which offers the opportunity to integrate changes into the original (forked) repository. Principal contributors can review the pull request and either accept it as it is, reject it, or ask for revisions.

Issue tracking and documentation for the project are also hosted on GitHub. Collaborators can open issues for investigation by principal contributors. This level of openness and transparency intends to foster a sense of purpose and community. It also structures code changes within a peer review system and enforces code discipline. Since the source code lies in a single repository, the entire project can be built on a modern computer using cross-hosted assemblers and compilers [24,25].

3.2.1. The Kernel

The heart of TurbOS is the *kernel*. The kernel creates an environment that processes reside and work in. It provides a number of common operating system services, including:

- Process creation, known as *forking*.
- Process scheduling and prioritization.
- Memory allocation and deletion.
- Module validation and management.
- Interrupt handling.
- Interprocess communication.

These services are available through system calls. System call names have a prefix of `F$`, for example, `F$Fork`, `F$Chain`, etc. System calls are made through the Turbo9's `swi2` instruction. This is a software interrupt instruction that suspends execution, pushes registers to the stack, then redirects execution to a location in the kernel that performs the system call. The byte following the `swi2` call represents the system call code. The assembly language source in Listing 1 demonstrates how this is achieved in TurbOS. The `F$Link` system call code, which is byte `$00`, follows the `swi2` instruction in the instruction stream.

Listing 1. TurbOS system call in Turbo9 assembly language.

```
some_routine:
          clra                      ; A = 0 (any module type/language)
          leax    modname,pcr       ; X = module name address
          swi2                      ; invoke system call
          fcb     F$Link            ; resolves to the constant $00
          rts                       ; return to the caller

modname   fcs     "testprog"        ; hard-coded name of the module
```

System call parameters are passed directly within the Turbo9's registers themselves. This provides a tight binding which saves code space and increases execution speed, compared to the C convention of the caller/callee using the stack for parameter placement and reference.

3.2.2. The Module

The fundamental organizational structure of a block of executable code or data is the *module*. As indicated in Figure 2, a module is composed of three parts: a *header*, a *body*, and a *footer*. The header contains information about the module such as its name and its identity (for example, is it object code or data). The body contains either executable code or data, and the footer is a 3-byte CRC (cyclic redundancy check) that the kernel uses to verify the integrity of the module. A module has a name, which uniquely identifies it in the system.

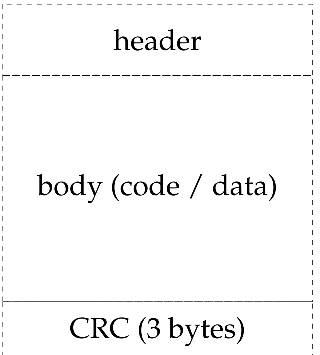

Figure 2. TurbOS module format.

Modules reside in the memory and are indexed in a *module directory*. The module directory contains a list of all of the modules in the memory. To obtain access to a module,

the application can link to it with the F$Link system call. This increases the module's link count, which is an indicator of how many times it is in use. Conversely, when the application finishes using a module, it calls F$Unlink to unlink it. This decreases the module's link count. When a module's link count reaches 0, it is removed from memory and the space is reclaimed.

3.2.3. Programs and Processes

A *program* is an executable block of code that resides inside a module. In other words, a program may also be considered an app. A program's purpose is to perform some useful work on the system, such as controlling inputs/outputs or performing calculations. Applications are a broad class of programs that communicate to the kernel through system calls.

On the other hand, a *process*, (also known as a *task*, in other operating systems such as FreeRTOS) is a single instance of a program. The kernel manages the running time of one or more processes through priority-based round-robin scheduling. The scheduling algorithm incorporates an age for each process to ensure that even lowest priority processes receives some run time. The kernel keeps track of the process with a data structure known as the *process descriptor*, (also known as a *task control block* in FreeRTOS), which is a 64-byte block of data. Figure 3 illustrates four sub-block processes interacting with the TurbOS kernel.

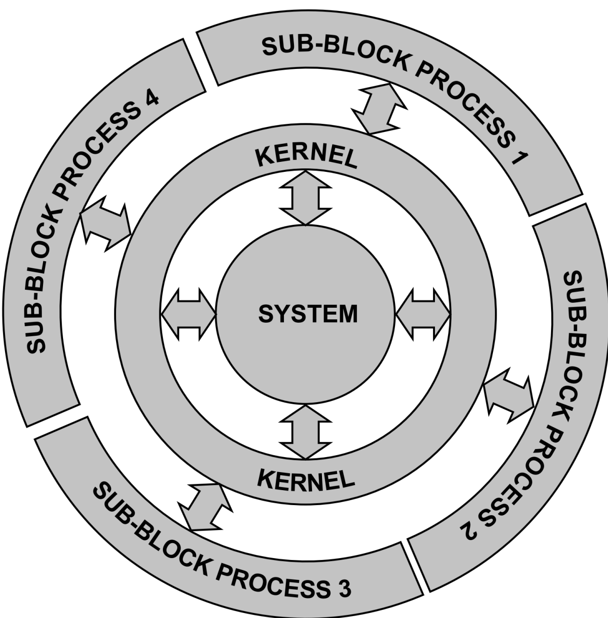

Figure 3. TurbOS process/kernel interaction.

The system call responsible for creating a new process is F$Fork. When a new process is forked, the kernel provides it with its own data area and stack, then places it in the active queue for scheduled execution. The process calling F$Fork becomes the parent, and the new process becomes the child. If the parent chooses to wait for the child process to finish, it calls the F$Wait system call, which blocks the parent process for the duration of the child process' lifetime.

New processes can also be created through chaining. When a process calls F$Chain, it effectively becomes the new process. There is no parent/child relationship when chaining occurs. This is useful when attempting to keep system resource usage to a minimum.

3.2.4. Process Scheduling

The kernel is responsible for determining when a process can run and obtain CPU time. The basic unit of time is called a tick and is usually 1 millisecond in length. The unit of time that a process can run on the CPU at an interval is called the time slice. A time slice

is a configurable multiple of a tick. A time slice is nominally 10 ticks or 10 milliseconds given a 1 millisecond tick.

The kernel uses a priority-based round-robin scheduling algorithm. Every process has an 8-bit priority and an 8-bit age. The highest value for priority and age is 255 and the lowest is 0. The default priority and age value for a new process is 128.

In order to manage access to the microprocessor, known as providing *CPU time*, the kernel defines a number of different queues that a process can be in at any given time.

3.2.5. The Active Queue

The active queue is a singly linked list of process descriptors that represents processes which are actively expecting to use the CPU. When a process is created through the `F$Fork` system call, the kernel places it in the active queue. The active queue is sorted by a process' age in order to ensure that lower priority processes receive some CPU time. As each process receives a slice of time on the CPU, the kernel increases its age. When the age reaches the maximum value of 255, the process' priority value is copied into its age. The kernel then reinserts the process into the active queue based on its new age value.

3.2.6. The Wait Queue

The wait queue is a singly linked list of process descriptors that represents processes which are waiting on one or more children to complete execution. A process enters this queue by calling the `F$Wait` system call.

3.2.7. The Sleep Queue

The sleep queue is a singly linked list of process descriptors that represents processes which are expecting to not use the CPU. A process can call the `F$Sleep` system call to give CPU time to another process for a given period of time.

3.2.8. Memory Allocation and Management

Given that memory is a constrained resource, the kernel manages memory in both 256-byte and 64-byte chunks. Some kernel data structures are small enough to only need 64 bytes, while others require 256 bytes, hence the separate system calls for obtaining different sizes.

The kernel divides the memory up primarily into 256-byte *pages*. The page size of 256 bytes conveniently fits into the 64K address space of the Turbo9, as 64K can accommodate exactly 256 pages of 256 bytes.

The kernel is responsible for managing memory resources on behalf of processes, but also claims some memory for its own use. Starting at the very low end of the memory map, the first 256-byte page from $0000 to $00FF is the area known as the system globals. The kernel uses this area for its own housekeeping. The next three 256-byte pages between $0100 and $03FF contain allocation bitmaps, the module directory, the system call table, and the system stack.

The top of the memory map contains the kernel and other system modules, as well as any program modules necessary to perform work. The free memory exists in between, and is available for both the kernel and programs to draw from.

The kernel provides four system calls that vend memory to and recover memory from processes. For memory allocations with 256-byte increments, there is `F$SRqMem`, which allocates memory on behalf of a process, and `F$SRtMem`, which returns allocated memory back to the system's free memory pool. The system's free memory pool is managed by the kernel using a 256-bit (32 byte) allocation bitmap located in the system globals area. Each bit in the 256-bit table represents one 256-byte page of memory starting at address 0 and ending at address 65280. It is the responsibility of the process to return the memory once it has finished using it; however, if the process terminates without deallocating memory, it is automatically returned to the system's free memory pool.

For finer grained memory allocation, two additional system calls are available for 64 byte allocation and deallocation: `F$All64` allocates memory on behalf of a process, and `F$Ret64` returns allocated memory back to the system's free memory pool. These two calls rely on `F$SRqMem` and `F$SRtMem`, respectively, to request and return 256-byte pages that are subdivided into four 64-byte blocks. TurbOS uses these fine grained calls for certain system data structures that fit within 64 bytes to utilize memory more efficiently.

3.2.9. Interrupt Handling

Interrupts are external triggers that arrest the microprocessor's execution and direct its path to a new execution area. Interrupts are instigated by hardware sources like clock generators, communications circuits, and human interface devices such as buttons or keys. The nature of interrupts is that they may arrive at any time, and must be serviced as soon as possible. Only after the interrupt is serviced will the original execution path resume at the interrupted location.

A key component to an RTOS's response is minimizing the time between the generation of the interrupt and the time at which it is handled. It is important that this span of time, known as the *interrupt latency*, be as short as possible.

In TurbOS, interrupts are directed into the kernel, where they are checked against an *interrupt polling table*. A process that expects to service interrupts must call the `F$IRQ` system call with a pointer to a polling table entry data structure, which includes the address of the interrupt service routine and other information that the kernel uses to verify the party responsible for servicing that interrupt.

3.2.10. Interprocess Communication through Signaling

At times, multiple processes may need to pass information between each other. Interprocess communication facilities such as pipes, events, message queues, and signals are mechanisms for this type of cross-communication in many operating systems.

TurbOS provides a fast and simplistic method of interprocess communication known as signaling. A signal is an 8-bit value that a process receives and acts upon. When a process receives a signal, a routine, known as a signal handler, is invoked. The code to address and act upon the signal value is executed inside the signal handler.

Once the signal handler routine is complete, execution resumes at the point in the process where it left off. Signals act in a similar fashion to interrupts, but are sourced from processes instead of hardware.

A number of signals are reserved for specific behavior. `S$Kill` is the kill signal; any process that receives it is immediately terminated. `S$Wake` is the wakeup signal. When a process receives this signal, the kernel puts the process into the active queue if it is not already there. The signal handler is not called, and is not even necessary, for a process receiving this signal.

A process can install a signal handler to process signals by using the `F$Icpt` system call. If a process does not install a signal handler, any signal it receives (except for `S$Wake`) immediately causes its termination; its parent then receives the signal value as the exit code.

The remaining signal values from 4 to 255 are known as user-defined signals, and can be used to pass information from one process to another. Cooperating applications can define their own behavior when receiving such a signal, and use the `F$Send` system call to send a signal to another process.

3.3. Application Survey

This work does not broach the area of specific applications; that will be studied in the next steps in the research. However, it is worth noting that when focusing on the smallest possible area and power consumption for a given task, hand-written assembly language programming for both the operating system and the applications yields the smallest size. A microprocessor, such as the Turbo9, with a rich instruction set that encapsulates

more functionality per instruction aids in hand-written assembly language, and intimate knowledge of the microprocessor's architecture gives the designer an advantage over an implementation of that operating system in a high-level language such as C. As noted earlier, this trade-off comes at the expense of code portability.

4. Methodology

With the Turbo9 and TurbOS selected, a validation step is conducted to compare its performance against other known microprocessors and operating systems. There are a number of open-source RTOSs written in the C programming language, but none are available that have binary compatibility with Turbo9. A popular C-based RTOS is FreeRTOS [26] and this is the operating system that is used for comparison. Like TurbOS, FreeRTOS offers pre-emptive scheduling, multiple process management, and interrupt handling facilities. Unlike TurbOS, it is arranged in a single monolithic artifact and has no modular organization. The Turbo9 port of FreeRTOS [27] resides in its own repository and is compiled using the CMOC C compiler [25].

In order to bring as much parity as possible to the comparison, it is necessary to optimize certain features in TurbOS. Knowing where to optimize requires a complete understanding of the operating system and where the dependencies are. The modular design of TurbOS provides a level of granularity that can help pare down the memory footprint; unneeded functionality can be discarded by simply removing the module that contains it. The kernel itself, however, is a monolithic object, and as such, it needs successive "cuts" to remove unneeded functionality. Therefore, the first step to optimizing TurbOS is to identify and eliminate operating system features that may not be needed for SoC sub-block management. Eliminating these features saves code space and ultimately contributes to a smaller on-chip area.

4.1. Interrupt Processing

Interrupt processing is crucial for the Turbo9 to process external events that may happen at any time. External interrupts can arrive on the Turbo9 in two ways: through a regular interrupt and a fast interrupt. The regular interrupt causes all of the Turbo9's registers to be pushed onto the stack; then, control is transferred to the interrupt handler, which processes the interrupt and exits with an `rti` instruction. The fast interrupt only pushes the condition code and program counter before transferring control to the fast interrupt handler.

Under TurbOS, processes register for interrupt processing through a system call. The kernel itself receives the interrupt and implements a polling mechanism to determine the source of the interrupt. If the interrupt is deemed to belong to a pre-registered process, the interrupt service routine is called to service that interrupt. Otherwise, the kernel continues requesting other handlers.

FreeRTOS does not contain a polling table, so for testing, it was conditionally removed from TurbOS for this analysis.

4.2. Module Verification

TurbOS performs a CRC validation check when loading a module into memory from an external storage device. This time-consuming task is justified to ensure that the integrity of the module is intact and that none of its contents have changed.

In an SoC context, modules are not loaded from an external storage device, but are present at the start of the operating system. Moreover, TurbOS and any running applications are immutable and essentially frozen in silicon, so module verification is not needed. It was conditionally removed for this analysis.

FreeRTOS does not have a module structure or perform any module validation, so for testing, this feature was eliminated in TurbOS.

4.3. Unified I/O

One of the features of the TurbOS operating system is its unified I/O model, which is very similar to UNIX. A device is accessible through a file interface, with common entry points labeled as Init, Read, Write, and Terminate. System calls enforce access to these devices and data are read or written via a path.

In embedded applications where external storage does not exist, the value of a unified I/O model is negligible. Moreover, FreeRTOS does not have this feature, so it was conditionally removed from TurbOS for this analysis.

4.4. Boot Support

In the same vein as unified I/O, boot support in TurbOS relies on the fact that the operating system's core resides in flash or ROM, and the rest of the system is loaded from external storage. For embedded applications where storage is not present, there is no need for boot support, and FreeRTOS does not have this feature, so it was conditionally removed from TurbOS for this analysis.

4.5. Code Base Reduction

In addition to reducing the code footprint of the operating system, the TurbOS repository is composed of 37 assembly source files. The NitrOS-9 repository from which it derives currently consists of 1092 source files. Much of this code is third-party drivers, add-ons, and apps that are not needed in a targeted application like SoC sub-blocks. These extraneous files have been eliminated in the TurbOS repository.

5. Results

Table 2 shows the size with specific and all features eliminated from TurbOS. Individually, feature elimination has a small impact on the resulting size of the kernel; together, they provide a 16.5% size reduction. This size does not include any application code; it is just the operating system.

Table 2. Feature elimination (bytes).

Feature	Before	After	% Reduced
interrupt polling	3435	3268	4.8%
module verification	3435	3270	4.8%
unified I/O	3435	3259	5.1%
boot support	3435	3351	2.4%
all of the above	3435	2865	16.6%

Table 3 shows the size of the artifact for TurbOS and FreeRTOS. It is expected that FreeRTOS, which is written in C, will be larger than an operating system written in assembly language. Setting aside the portability of the operating system as a requirement, the >80% reduction in size directly impacts the SoC sub-block area considerably, and weighs heavily in favor of using a hand-tuned assembly-language-based operating system for such applications.

Table 3. Operating system artifact size (bytes).

TurbOS	FreeRTOS	% Reduced
2865	16,379	82.5%

6. Conclusions and Future Work

As shown in the results, TurbOS is considerably smaller in size than FreeRTOS while providing the same functionality (e.g., process/task creation and destruction, memory allocation, and task switching). While the kernel is demonstrably small for the amount of

features it presents, its size can be further minimized with more hand-tuned instruction optimizations and careful excision of unneeded features.

Further space savings can be realized on the hardware by removing higher address lines of the Turbo9. For example, eliminating both the highest and second highest address lines provides a maximum addressing size of 32 KB in a 15-bit system or 16 KB in a 14-bit system, which is still large enough to fit the kernel and a number of robust applications.

Additionally, there are a number of areas in this work that should be explored in further research, including:

- Analysis of physical metrics (die area, power consumption) for a set of SoC sub-block applications on TurbOS vs. operating systems on other architectures such as RISC-V and ARM.
- System call, interrupt handling, and application bench marking comparisons against other operating systems on the Turbo9, such as FreeRTOS and FUZIX [28].
- Designing new instructions for the Turbo9 to increase performance.
- A complete test bench and validation suite for TurbOS.

Author Contributions: Conceptualization, B.P. and M.M.; methodology, B.P. and M.M.; software, B.P.; validation, B.P.; formal analysis, B.P.; investigation, B.P.; resources, B.P. and M.M.; writing—original draft preparation, B.P.; writing—review and editing, B.P. and M.M.; supervision, M.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The software created for this paper is available in [29].

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
IoT	Internet of Things
IRQ	Interrupt Request
RTOS	Real-Time Operating System
SoC	System on a Chip
SPI	Serial Peripheral Interface
SWI	Software Interrupt
USB	Universal Serial Bus

Glossary

Forking	The act of creating a new process in TurbOS.
FreeRTOS	An open-source real-time operating system.
Link count	A count of the number of times a TurbOS module is in use.
Module	A set of bytes in TurbOS that contains a header, a body, and a CRC.
Process	A process in TurbOS that is analogous to a task in FreeRTOS.
SoC	System on a Chip; a single or multicore processor with sub-blocks.
Task	A process in FreeRTOS that is analogous to a process in TurbOS.
Turbo9	An open-source, 16-bit pipelined microprocessor modeled after the Motorola 6809.
TurbOS	An open-source, 16-bit real-time operating system designed for the Turbo9.
Sub-block	A unit of logic in an SoC that provides specific support functionality.

References

1. Lin, T.J.; Liao, C.Z.; Hu, Y.J.; Hsu, W.C.; Wu, Z.X.; Wang, S.Y.; Huang, C.M.; Lai, Y.H.; Yeh, C.; Wang, J.S. A 40 nm CMOS SoC for Real-Time Dysarthric Voice Conversion of Stroke Patients. In Proceedings of the 27th Asia and South Pacific Design Automation Conference, Taipei, Taiwan, 17–20 January 2022; pp. 7–8. [CrossRef]
2. Embedded Systems Market Size, Share, Growth Report 2030. Available online: https://www.zionmarketresearch.com/report/embedded-systems-market (accessed on 3 May 2024).
3. Comparison of Real-Time Operating Systems. Available online: https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems (accessed on 3 May 2024).
4. CP/M. Available online: https://computerhistory.org/blog/early-digital-research-cpm-source-code/#code (accessed on 5 May 2024).
5. MS-DOS. Available online: https://github.com/microsoft/MS-DOS (accessed on 5 May 2024).
6. Muchow, O.; Ustarbowski, D.; Hammouda, I. An Investigation of Migrating from Proprietary RTOS to Embedded Linux. In Proceedings of the 11th International Symposium on Open Collaboration, San Francisco, CA, USA, 19–21 August 2015. [CrossRef]
7. Nakano, W.; Shinohara, Y.; Ishiura, N. Full Hardware Implementation of FreeRTOS-Based Real-Time Systems. In Proceedings of the TENCON 2021—2021 IEEE Region 10 Conference (TENCON), Auckland, New Zealand, 7–10 December 2021; pp. 435–440. [CrossRef]
8. Rane, U.V.; Panem, C.; Abhyankar, G.; Gad, R.S. Network on Chip(NoC) Mesh Topology FPGA Verification: Real Time Operating System Emulation Framework. In Proceedings of the 2024 Fourth International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT), Bhilai, India, 11–12 January 2024; pp. 1–5. [CrossRef]
9. Vargas, F.L. On-Chip Cross-Layer Infrastructure to Leverage System Reliability for Aero-Space Applications: Embedded Tutorial. In Proceedings of the 2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS), Kielce, Slovakia, 3–5 April 2024; pp. 116–117. [CrossRef]
10. Krishnan, S.; Wan, Z.; Bhardwaj, K.; Whatmough, P.; Faust, A.; Neuman, S.; Wei, G.Y.; Brooks, D.; Reddi, V.J. Automatic Domain-Specific SoC Design for Autonomous Unmanned Aerial Vehicles. In Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture, Chicago, IL, USA, 1–5 October 2023; pp. 300–317. [CrossRef]
11. µC/OS. Available online: https://www.osrtos.com/rtos/uc-os-ii/ (accessed on 5 May 2024).
12. Shang, X. Implementation of Embedded Real-Time Operating System and Application Software based on Smart Chip. In Proceedings of the 2022 3rd International Conference on Electronics and Sustainable Communication Systems (ICESC), Coimbatore, India, 17–19 August 2022; pp. 1290–1293. [CrossRef]
13. Zhu, H.; Zhou, Y.; Liu, H.; Dai, H. Design and Implementation of a Stack-based SoC Supporting Real-time Forth Operating System. In Proceedings of the 2023 4th International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE), Guangzhou, China, 25–27 August 2023; pp. 263–267. [CrossRef]
14. Larrea, J.; Barbalace, A. The serverkernel operating system. In Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking, Heraklion, Greece, 27 April 2020; pp. 13–18. [CrossRef]
15. Wu, J.; Zheng, X.; Zeng, S.; Gao, H.; Xiong, X. High-Performance Cryptographic SoC Virtual Prototyping Platform Based on RISC-V VP. In Proceedings of the 6th International Conference on High Performance Compilation, Computing and Communications, Jilin, China, 23–25 June 2022; pp. 84–90. [CrossRef]
16. Orenes-Vera, M.; Manocha, A.; Balkind, J.; Gao, F.; Aragón, J.L.; Wentzlaff, D.; Martonosi, M. Tiny but mighty: Designing and realizing scalable latency tolerance for manycore SoCs. In Proceedings of the 49th Annual International Symposium on Computer Architecture, New York, NY, USA, 18–22 June 2022; pp. 817–830. [CrossRef]
17. Chi, Y.; Lin, X.; Zheng, X. Design of High-performance SoC Simulation Model Based on Verilator. In Proceedings of the 2022 5th International Conference on Algorithms, Computing and Artificial Intelligence, Sanya, China, 23–25 December 2022. [CrossRef]
18. Phillipson, K.; Rywalt, M. Turbo9. Available online: https://www.turbo9.org (accessed on 5 May 2024).
19. Motorola. *M6809-M6809E Programmers Manual*, 1st ed.; Motorola: Chicago, IL, USA, 1981.
20. Ritter, T.; Boney, J. The 6809 Design Philosophy. *Byte Mag.* **1979**, *4*, 14–42.
21. NitrOS-9. Available online: https://github.com/n6il/nitros9 (accessed on 5 May 2024).
22. Ritchie, D.M.; Thompson, K. The UNIX time-sharing system. *Commun. ACM* **1974**, *17*, 365–375. [CrossRef]
23. Mantovani, P.; Giri, D.; Di Guglielmo, G.; Piccolboni, L.; Zuckerman, J.; Cota, E.G.; Petracca, M.; Pilato, C.; Carloni, L.P. Agile SoC development with open ESP. In Proceedings of the 39th International Conference on Computer-Aided Design, Storrs, CT, USA, 24–27 October 2021. [CrossRef]
24. Astle, W. LWTools. 2024. Available online: http://www.lwtools.ca/ (accessed on 5 May 2024).
25. CMOC. Available online: http://perso.b2b2c.ca/~sarrazip/dev/cmoc.html (accessed on 5 May 2024).
26. FreeRTOS. Available online: https://freertos.org (accessed on 5 May 2024).
27. FreeRTOS Port for the Turbo9. Available online: https://github.com/boisy/FreeRTOS_Turbo9 (accessed on 5 May 2024).
28. Cox, A. FUZIX. Available online: https://github.com/EtchedPixels/FUZIX (accessed on 5 May 2024).
29. Pitre, B.G. TurbOS. Available online: https://github.com/boisy/turbos (accessed on 5 May 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.