Preprints.org

Article

Not peer-reviewed version

Timing and Performance Metrics for TWR-K70F120M device

George K. Adam *

Posted Date: 24 July 2023

doi: 10.20944/preprints202307.1587.v1

Keywords: single board computers; embedded systems; real-time; multithreading; performance metrics; time measurements; benchmarking; µClinux; TWR-K70F120M

Preprints.org is a free multidiscipline platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

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.

Article

Timing and Performance Metrics for TWR-K70F120M Device

George K. Adam

CSLab Computer Systems Laboratory, Department of Digital Systems, University of Thessaly, 41500 Larisa, Greece; gadam@uth.gr; Tel.: +30-2410-684-596

Abstract: Currently Single Board Computers (SBCs) are sufficiently powerful to run Real-Time Operating Systems (RTOSs) and applications with real-time attributes and requirements. SBCs serve as a foundation in Industrial Internet of Things (IIoT). The NXP Semiconductors produces a series of SBCs based on ARM-processors for a variety of industrial applications. The continuous increase in real-time data generated by IoT devices adds further research issues about the efficiency of such systems and applications. The purpose of this research was to investigate the timing performance of an NXP TWR-K70F120M device with μClinux OS on running concurrently tasks with real-time features and constraints. A custom-built multithreaded application with specific compute-intensive sorting and matrix operations was developed and applied to obtain measurements in specific timing metrics, including task's execution time, threads waiting time, and response time, under different threads variations. The performance of this device was additionally benchmarked and validated against favorite platforms, a Raspberry Pi4 and BeagleBone AI SBCs. The experimental results showed that this device stands well both in terms of timing and efficiency metrics. Execution times were quite lower than the others, by approximately 56% in the case of two threads, and by 29% in the case of thirty-two threads configurations.

Keywords: single board computers; embedded systems; real-time; multithreading; performance metrics; time measurements; benchmarking; μClinux; TWR-K70F120M

1. Introduction

Single Board Computers (SBCs) facilitate the Internet of Things (IoT) and can enable Fog and Edge compute applications to run efficiently on IoT computing and data generating nodes (e.g., sensors). In IoT architectures there is a tendency for the computational power to be pushed out closer to the edge, on smart devices such as SBCs. The volume of data generated from IoT devices, sensors and other equipment are forcing enterprises and manufacturers to design and develop SBCs with sufficient computing power for real-time analytics, and for applications with real-time attributes and requirements [1,2]. Single board computers are usually cost-effective and versatile commercial off-the-shelf (COTS) computer platforms, which offer significantly reduced time to implementation and efficient solutions in many sectors, including the Industrial IoT (IIoT) applications (e.g. Texas Instruments BeagleBone Black, NXP i.MX 8 series-based boards, Raspberry Pi Compute Module 4, etc.). ARM-based microcontrollers are commonly deployed in many SBCs, mobile phones, and different types of embedded devices and industrial applications. This is because they offer high performance and power efficiency at a reasonable price/performance ratio, and have low-power requirements [3].

An important feature of such devices in industrial solutions is the requirement for real-time capabilities. That is the capability to support real-time timing constraints (e.g., a deterministic response, minimized latencies, and bounded execution times). A real-time application must complete real-time tasks within a deterministic deadline. For example, the time elapsed from the appearance of an event to the actual system response often must be within a strict time range. Such timing requirements are quite important in industrial production and IIoT [4,5].

Single board computers deploy Operating Systems (OS) which make use of the minimum of the resources available by the embedded microcontroller, and usually with a small memory footprint.

© 2023 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

Real-Time Operating Systems (RTOS) are quite common in such devices and embedded systems applications [6,7]. Real-time kernels can be used for developing applications that perform multiple tasks (threads) simultaneously in a deterministic way. Certainly, real-time applications can be created without an RTOS, however, timing issues can be solved more efficiently with an RTOS.

The low-cost and power consumption of SBCs, their adaptability as well as their stability in availability by key market manufacturers such as Texas Instruments, Inc., Advantech Co., Digi International, Inc., NXP Semiconductors, Raspberry Pi Foundation, have enabled such products to be deployed into several cases where processing requirements are met by small and smart embedded devices [8,9]. The continuous increase in applications of SBCs has led also to the development of more powerful peripherals being included on such boards.

The need for SBCs that support real-time applications running multiple functions at guaranteed times, particularly in industry, will continue to rise. As a market research report by KBV Research foresees: the global Single Board Computer market size is expected to reach $3.8 billion by 2026, rising at a market growth of 7.2% CAGR (Compound Annual Growth Rate) during the forecast period [10]. In industrial automation this market growth is expected to be much higher than all other competitive sectors e.g., aerospace and defense, healthcare, and consumer electronics.

Therefore, their overall performance will continue to be assessed and taken into consideration according to each specific application and its requirements. In this direction, this research assessed the real-time computing capabilities and timing performance of a specific SBC, in particular the NXP TWR-K70F120M board [11]. This was motivated primarily by the interest of a machinery manufacturing company which has already procured some of these devices, and is looking to deploy them further on its machines' production [12]. One of the near future company's intentions is to deploy this board in a variety of control applications in the construction and production of automated machines for concrete products. This research investigated the capability of the NXP TWR-K70F120M board, running a μClinux distribution [13], to support applications with real-time constraints, and its overall performance in terms of timing metrics, such as task's execution time, waiting time and response time.

One of the contributions provided by this research were the performance measurements, carried out by the use of a custom-built multithreaded application with real-time attributes, developed for this purpose. In addition, a benchmark was applied that ran additional performance tests based on cryptography algorithms. As it is indicated in the next section of related research, to the best of our knowledge such information on timing performance metrics for this board are not available. Furthermore, beyond the assessment of the TWR-K70F120M platform with a benchmark and a specific multithreaded application, the same methodology and application assessment software was implemented and tested on other two popular single board computers, a BeagleBone AI and a Raspberry Pi4 Model B, with Linux OS support. The results were compared to those obtained by the NXP TWR-K70F120M board, and conclusions were drawn upon its performance.

This paper is structured as follows: Section 2 describes previous related work. Section 3 presents the methodology applied, the system's hardware and software infrastructure, and the performance metrics acquired in measurements. Section 4 presents the experimental framework used as the testbed, the results obtained on performance measurements of the NXP TWR-K70F120M board, and the comparative results with other SBC devices. Section 5 provides a brief discussion about the ideas in the paper and the research outcomes, and Section 6 provides concluding remarks.

2. Related Work

The last years, in several application sectors there is a tendency towards the use of more versatile, flexible, and low-cost devices based on ARM processors, provided by many vendors, such as the Nvidia Jetson Nano, the NXP SBC-S32V234, the BeagleBone AI-64 and the Raspberry Pi Compute Module 4 single board computers [14–16].

Real-time operating systems have also been employed in order to provide support to system applications running multiple tasks (threads) with real-time constraints, particularly in embedded systems [17–19]. For example, open-source real-time operating systems, such as μClinux and μC/OS-

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

III [20], which target microcontrollers without Memory Management Unit (MMU) support. A number of works make use of μClinux to create applications on microcontrollers [21,22].

The performance of real-time systems and multithreaded applications is analyzed and benchmarked with many different approaches. The techniques and tools used depend on the aspects of performance that are targeted for measurement and evaluation, most commonly schedulability and timing issues in real-time systems [23]. In general, there is considerable work on performance assessment of single board computers, however, there has not been much work on the NXP TWR-K70F120M platform.

This NXP TWR-K70F120M board can be found in several applications [24–26]. However, the works that implement performance assessment of this board, either with standard benchmarks or with specific software developed for such purposes, are very limited. An interesting research work is the work on performance analysis of an embedded system by Luigi Vicari [27]. This work presented debugging and tracing with the Lauterbach μTrace device in Trace32 environment of a μClinux signal processing application running on the TWR-K70F120M development board. However, the focus is on the performance analysis of the execution of two different FIR (Finite Impulse Response) filter implementations (with different complexities), and an FFT (Fast Fourier Transform) algorithm (with different number of points for which is computed the transform).

An interesting benchmark was applied, provided by wolfSSL Inc. [28], which examines the performance of the TWR-K70F120M platform using its in-house application wolfSSL package. This package includes a wolfCrypt benchmark application. Because the underlying cryptography is a very performance-critical aspect of SSL/TLS, this benchmark application runs performance tests on wolfCrypt's algorithms. The performance of the TWR-K70F120M platform was determined under the MQX RTOS and using the fastmath library and CodeWarrior 10.2 IDE. In our research work, beyond our performance measurements and assessment of the TWR-K70F120M platform on running multithreaded applications with real-time constraints, we have additionally applied this wolfCrypt benchmark for comparison purposes. In our case, the implementation of this benchmark was under μClinux (kernel v2.6.33) and the fastmath library, and the use of CodeWarrior 11.1 IDE for the application build [29].

3. Materials and Methods

3.1. Tower System Architecture

The tower system under investigation is based on the NXP's (ex Freescale) TWR-K70F120M board, which includes a Kinetis K70 family (MK70FN1M0VMJ12) microcontroller (MCU) having a 32-bit Arm Cortex-M4 core processor, and running an embedded version of Linux (μClinux).

The tower system platform consists of the main controller module (TWR-K70F120M), primary and secondary side elevators and a serial module (TWR-SER) (see Figure 1). The TWR-K70F120M controller module, beyond the Kinetis MCU, on-board includes 1GB DDR2 SDRAM memory, 2GB NAND Flash memory, various user controllable, LEDS, push buttons, switches, touch pads, a potentiometer, etc.. The Kinetis MK70FN1M0VMJ12 microcontroller includes a 32-bit Arm Cortex-M4 core with DSP instructions running at 120MHz, 1MB of program Flash, 128KB SRAM, 16-bit ADC, 12-bit DAC, various circuits for peripherals communication, and operates at a low voltage input range (1.7V-3.6V). The ARM Cortex-M4 does not support simultaneous multithreading (SMT).

The TWR-K70F120M board is powered up by connecting it to a host PC (running either Windows or Linux) through a mini-USB connector on the TWR-K70F120M board. For programming purposes, a serial interface is established to the host PC by plugging an RS-232 cable to the serial connector on the TWR-SER board. On the host PC side, the serial link provides a serial console device to the TWR-K70F120M. The software installed on the board is configured for a terminal (e.g. putty) with the following COM-port settings: 115200 8N1. On the Linux host, the serial console is available using a /dev/ttySn device. Network connectivity to the board, is provided by plugging a standard Ethernet cable into the TWR-SER 10/100 Ethernet connector. The board is pre-configured with an IP address of 192.168.0.2.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

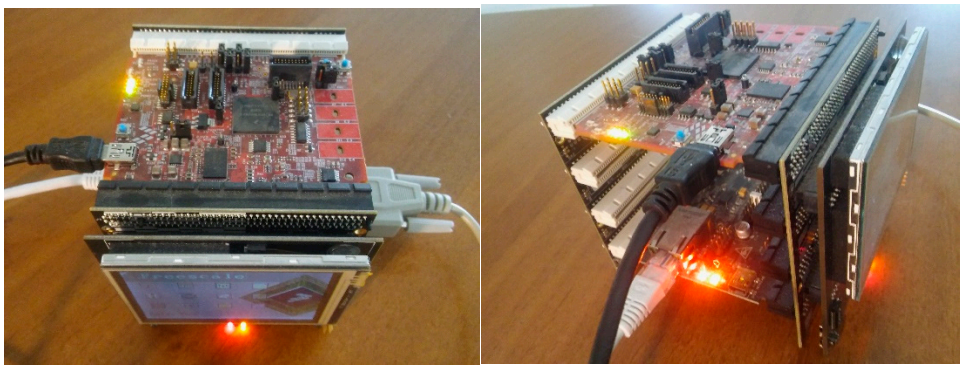

Figure 1. The Tower System.

3.2. Tower System Software Infrustructure

Linux is an open-source OS that can be installed on a variety of different types of architectures. A version of μClinux (kernel v2.6.33) intended for microcontrollers without MMUs is loaded into the external Flash (2GB) on the TWR-K70F120M board, to support the execution of the measurements software. This version supports POSIX Threads. This library supports real-time and preemptive scheduling. Pthread functions are used to set the threads real-time execution features including scheduling policy, CPU affinity, and timing.

On a power-on, the default configuration copies the Linux image from the external Flash to RAM and jumps to the Linux kernel entry point in RAM, to allow booting Linux. The Tower System provides a complete platform for developing and testing purposes. The CodeWarrior software was used for the development of the measurements multithreaded application. The application was loaded (with Kinetis Flash Programmer) into the Kinetis MK70FN1M0VMJ12 32-bit Arm Cortex-M4 microcontroller's internal Flash (1MB program Flash). The application uses the embedded SRAM (128KB) of the microcontroller as a storage for the data used in computations (sorting and matrix operations). The system application development flow is shown in Figure 2.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

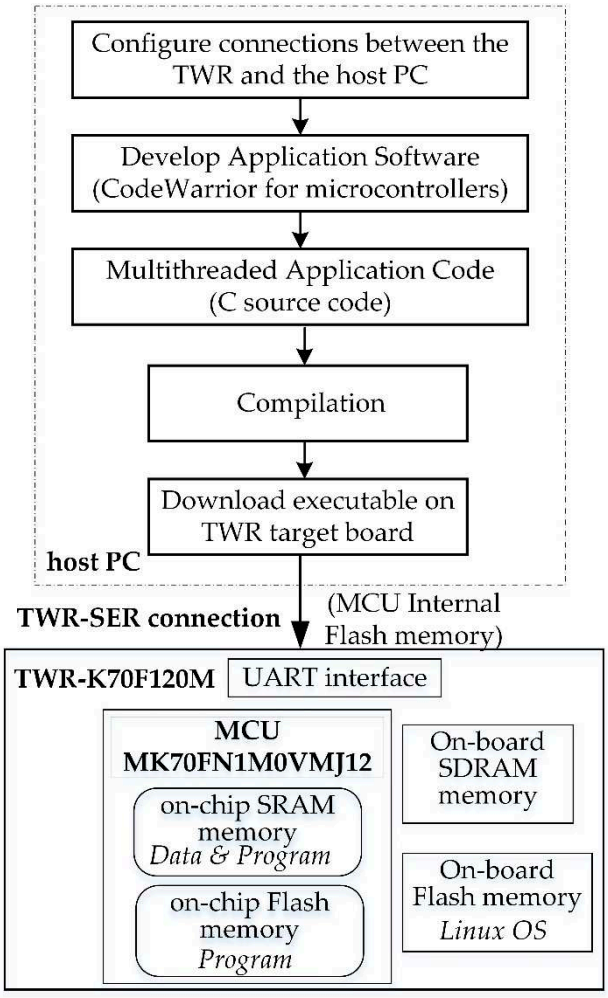

Figure 2. System application development flow.

3.3. Methodology

A primary goal of this research was to investigate some of the important performance metrics regarding timing constraints, such as task's execution time, waiting time, and response time (latency), of NXP TWR-K70F120M SBC, running a version of μClinux for embedded microcontrollers without MMUs.

For this purpose, a specific measurement application was developed and implemented as multithreaded software module. The design methodology was based on a dynamic and iterative multiple threads generation approach, by using POSIX Threads (an API defined by the standard IEEE POSIX.1c). Pthreads is preferred for the development of the multithreaded measurements software, than an API such as OpenMP, in order to have better low-level control. This measurement software enabled the estimation of certain performance metrics under real-time tasks (threads) concurrent execution, on μClinux kernel.

3.3.1. Multithreaded Application

The multithreaded design was based upon an algorithm which generates iteratively multiple processes (jobs) and threads (tasks), which ran concurrently in the same core. Thus, the Cortex-M4 core was shared by at least two software threads supported by a single hardware thread. The processes generated perform two types of computations: sorting and matrix operations. Each process generates multiple pairs of threads assigned to run concurrently on a single core. Therefore, on every execution run, initially two software threads, and later multiples of twos are instantiated.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

On the "sorting" process, each thread executes a computational intensive workload (function) that is either a selection sort or the quicksort algorithm. Selection sort is a simple sorting algorithm that divides the data array into a subarray of already sorted elements and a subarray of remaining elements to be sorted. The sorted array is finally populated based on an iteration procedure, upon which the smallest element from the unsorted subarray is placed at the end of the sorted array. Quicksort is a relatively more complex algorithm. It uses a divide-and-conquer strategy to divide the array into two subarrays. The sorted array is produced upon a procedure that reorders the array by moving all the elements less than a specific value (pivot) to its left and all the elements greater than it to its right.

On the "matrix" process, each thread performs either matrix addition or matrix multiplication operations. The multiplication of matrices is one of the basic computations used as a benchmark due to its very compute-intensive nature. For testing purposes, the threads generated are multiples of 2, 4, 8, 16 and 32 software threads.

Threads belonging to the same process share all the memory of the process and its resources. Therefore, the threads within each process share common data for their respective computation available in shared SRAM memory. Different data blocks are allocated for each process in the on-chip SRAM to support the execution of multiple threads which perform different operations. For sorting operations, data arrays of 32-bit integers of size 10^2 to 10^6 elements were allocated for each sorting thread in the on-chip shared memory to support the concurrent execution. Sorted data output was written out in the same memory space. For matrix operations, data sets were based on 512, 1024, 2048, 4096, 8192 arrays of short integers. A schematic view of application's structure is shown in Figure 3.

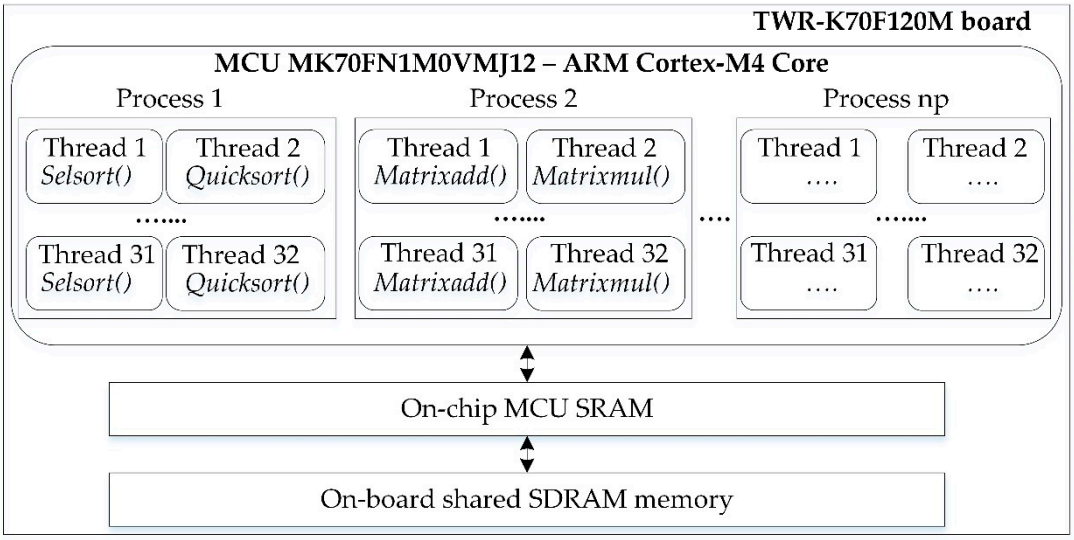

Figure 3. Multithreaded application structure.

3.3.2. Multiple Processes and Threads Generation

Multiple processes and threads were created repetitively. The repetitive functions (create_processes(), create_threads()) built a tree structure of multiple processes and concurrent threads, up to a certain quantity (np for processes and nth for threads). This tree structure is an aggregation of nodes consisted of processes and threads (see Figure 4). Each process contains a group of threads assigned to run concurrently, in synchronisation (pthread_join()). In the experiments, each process ran a variable number of threads, from a minimum two up to thirty-two threads. These multiples of threads resulted to a considerable contention for core resources simultaneously, an increase on thread switches, and threads delays on waiting for execution.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

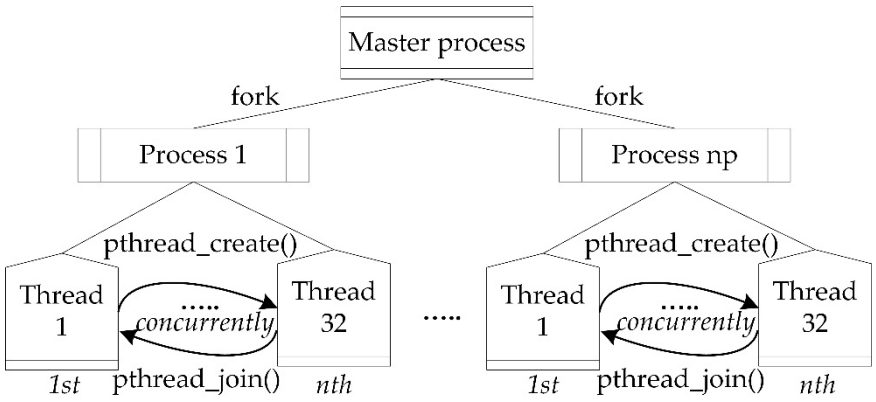

Figure 4. Processes and threads tree structure.

The algorithm for the iterative generation of each process distributes and allocates equally the workloads to each thread for optimal computation. A synopsis of the multiple threads and processes generation is shown as pseudocode in Algorithm 1 and as a flow chart in Figure 5.

Algorithm 1 Iterative multiple processes and threads creation

```
Inputs: number of processes p [2:np], number of threads th [2:32]
create_processes(p) {
for number of processes np {
    set_process_attributes();
    set_process_policy(RoundRobin, 99);
    create_threads(th); {
    for number of threads nth {
        pthread_create(&threads[nth--], NULL, threads_functions[nth--], NULL);
        pthread_join(threads[nth--], NULL); }}}
threads_functions(nth); {
set_threads_attributes();
if (th%2<>0)
sort_ops(); } //sorting operations
else
matrix_ops(); } //matrix operations
perform_measurements();
Outputs: measurements [execution time, threads waiting time, response latency]
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

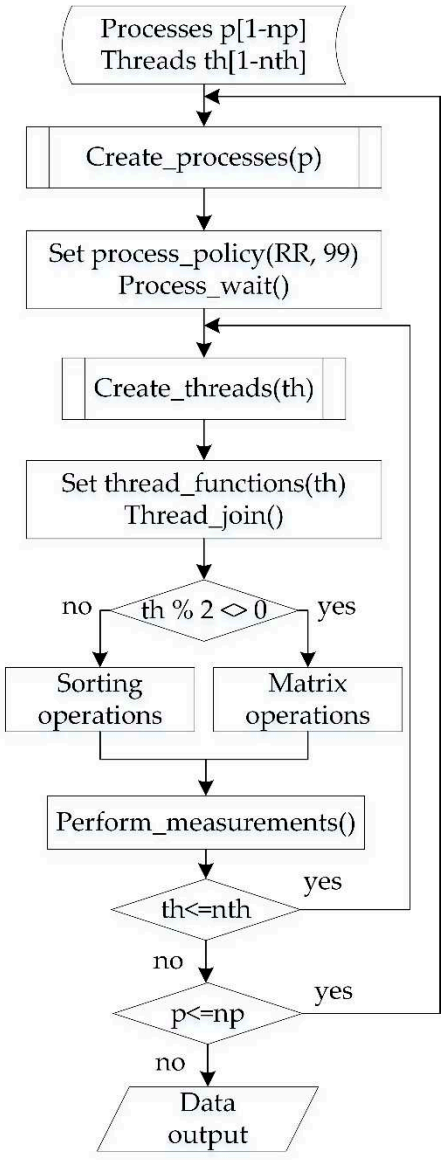

Figure 5. Iterative multiple processes and threads creation.

The threads were synchronized with mutexes. Therefore, data allocated in on-chip memory that support the execution of multiple threads were protected through software mutexes (pthread_mutex_lock(), pthread_mutex_unlock()). Mutexes guarantee exclusive access to the resources and facilitate threads synchronization. The threads attributes were configured to run with real-time execution features, including scheduling policy (SCHED_RR), CPU affinity (core 0), priority (99) and timing. Precise timing and reliable performance metrics require accurate timing source. The application software performed time performance measurements using the system call clock_gettime(), with the highest possible resolution, and the clock id set to CLOCK_MONOTONIC. The threads shared the same high priority which was set to 99. The SCHED_RR scheduling policy is preferred for real-time applications. Therefore, a Round Robin scheduler (SCHED_RR) was applied appropriately for threads real-time scheduling. The multithreaded application defined and implemented the threads scheduling policy and attributes. The source code in C of the experimental software module is available as an open-source project at GitHub [30]. The same application module and performance measurement methodology could be applied to other Linux-based systems and platforms.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

3.4. Performance Metrics

In Linux Operating Systems, a task is synonymous with a thread. In real-time applications, usually each task is scheduled as a thread, with real-time SCHED_FIFO or SCHED_RR policy and high priority. Time measurements are based upon the use of the system call clock_gettime() defined in POSIX timers header library implementation. This system call was invoked with the highest possible resolution, and with the clock id set to CLOCK_MONOTONIC. Each working thread during its execution recorded data for the performance metrics to be evaluated (including execution time, waiting time, response latency).

3.4.1. Execution time

For a real-time system a primary performance criteria under consideration is the amount of time it takes to complete a task, and consequently the system throughput. Therefore, the functions' execution time performing the sorting and matrix operations must be measured, in order to examine the tower (TWR) unit and μClinux OS in terms of real-time computing capability. An average value of the total execution time (t_{exec}) for a given number of execution runs (n iterations) was estimated by the following equation 1:

$$t_{exec} = \frac{\sum t_{end} - t_{begin}}{n}, \quad (1)$$

where t_{end} is the time it takes to finish the task's execution, and t_{begin} is the initial time the execution is started. The above is illustrated in Figure 6.

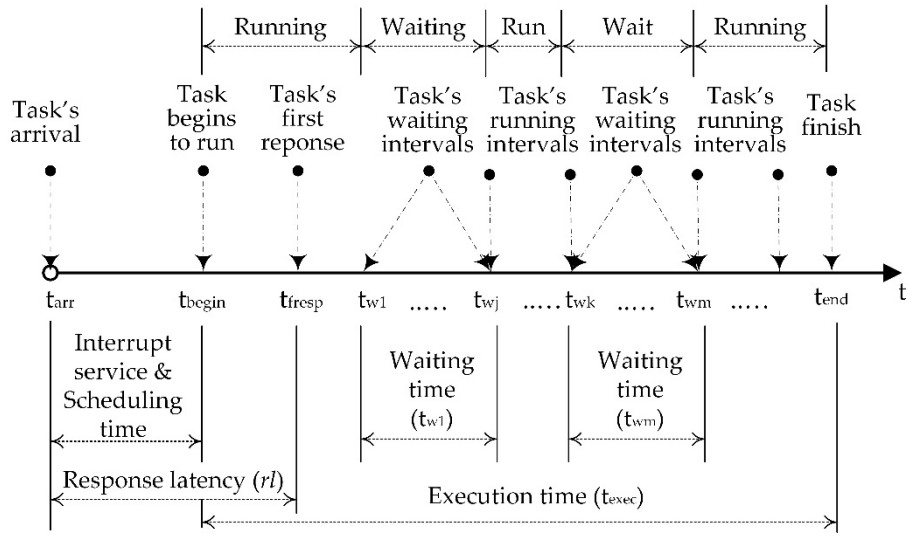

Figure 6. Task scheduling and running.

3.4.2. Threads waiting time

Each thread was assigned to run an independent task having its own memory stack and sharing process's space and time. The time it takes for a thread to execute a single task is essential in performance measurements. The threads were running concurrently on a single core. Therefore, there were frequent periods where any of the threads within a process is just waiting (idle thread) for execution resources (see Figure 6). In consequence, the time a thread spends awaiting for a resource or an event, is an important metric. Usually the time the thread is awaiting exceeds the time the thread spends using the resource. Reducing wait times (for resources) is a common strategy for improving the overall performance of a system. Therefore, in order to evaluate the performance of the threads dynamic operation in multithreaded applications it is essential to measure their waiting time. A simple but effective way of identifying when a thread is active or idle (waiting), is to use a thread-specific state variable, which is set when the thread exits the execution cycle, (e.g., waiting for a mutex

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

on a resource), and is unset respectively when it begins the execution again. In this way, a global time counter was used to count and add the total sum of the thread's waiting time (t_{wait}) on the basis of the following equation 2:

$$t_{wait} = \sum_{i=1}^{m} t_{w_i}, \quad (2)$$

where t_{wi} is the task's individual waiting time intervals out of the total m.

3.4.3. Response time

Applications having timing constraints should respond as soon as possible meeting any specified deadlines, an important aspect of a real-time system. For hard real-time systems the deadlines must always be met and the response times to be guaranteed. Therefore, it is important to measure the response time (or latency) of a task's running thread. A real-time task is characterized by its execution time, usually relevant to a deadline, and a maximum (worst-case) response latency as the upper bound. The worst-case response latency is a typical metric of the determinism of a real-time task. A task's response time or latency is defined as the overall time elapsed from the arrival of this task to the moment this task is switched to a running state, and receiving its first response. The overall response latency includes the interrupt latency that is the time it takes to appear upon the arrival of the task and its service time, and the task's scheduling latency that is the time it takes for the scheduler to run the task (see Figure 6). An average value of the response latency (rl) for a number of runs (n iterations) was calculated using the following equation 3:

$$rl = \frac{\sum_{n}^{i}(t_{fresp} - t_{arr})}{n}, \quad (3)$$

where t_{arr} is the task's arrival time, t_{fresp} is the task's first response time, and n is the number of performed iterations.

4. Results

4.1. The Experimental Framework

The reliability of the TWR-K70F120M platform (TWR) to run efficiently the multithreaded application was examined with experimental measurements. As it was stated earlier, this platform integrates a Kinetis MK70FN1M0VMJ12 microcontroller, which includes a 32-bit Arm Cortex-M4 core with DSP instructions running at 120MHz. In addition, for comparison purposes, the same experimental measurements were executed on other two popular single board computers, a Raspberry Pi4 (RPi4) and a BeagleBone AI (BBAI) (see Figure 7). These low-cost, low-power, and stand-alone single-board computers are being extensively used for embedded applications. The Raspberry Pi4 board integrates a Broadcom BCM2711 SoC having an ARM Cortex-A72 4-core processor running at 1.5 GHz. The BeagleBone AI is built upon Texas Instruments Sitara AM5729 SoC having a dual ARM Cortex-A15 processor running at 1 GHz. Neither of the processors support simultaneous multithreading (SMT). One of the goals of SMT is to keep the shared pipelines full where they would otherwise stall or have bubbles. The 32-bit ARM Cortex-M4 architecture is based on a 3-stage small and efficient pipeline that does not have considerable load-store latencies. The interrupt latency takes 12 cycles. The ARM Cortex-A15 and Cortex-A72 core architectures have a 15-stages pipeline. All the above system platforms run Linux versions. The TWR runs μClinux (kernel v2.6.33) and the other two RPi4 and BBAI platforms run UBUNTU (kernel 4.14.74-v7+).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

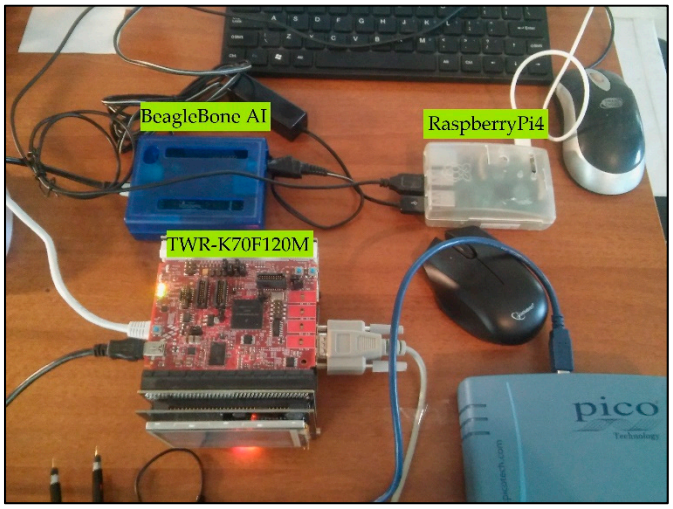

Figure 7. Experimental setup.

4.2. Performance Measurements on TWR-K70F120M Platform

A number of experiments were carried out to evaluate the performance of the tower architecture and verify its feasibility for running efficiently multithreaded applications with real-time features. The application module developed and used in the experimental runs scheduled the execution initially of two pairs of computational tasks concurrently on a single core, as pairs of two software threads, and later their multiples (2, 4, 8, 16 pairs). Thus, the core resources are shared by at least two threads running concurrently. This core architecture by default allows one-to-one threads to core mapping. This running scheme of having multiple threads assigned to run on the core with the same scheduling policy and priorities added substantially to the workload.

Each process scheduled two tasks as software threads which ran concurrently and performed different computations. The computational tasks consist of sorting and matrix operations. The sorting algorithms (selection and quick sort) were executed on a single core with multiple threads configurations (two to thirty-two) and datasets of 32-bit integers of size 10^2 to 10^6 (100 to 1 M) elements. Similarly, were executed matrix operations (addition and multiplication) with data sets based on 512, 1024, 2048, 4096, 8192 arrays of short integers.

The experiments were executed multiple times, approximately for a few thousands iterations, in order to obtain sufficient number of values for averages estimation. Table 1 presents the results obtained for the execution times for selection and quicksort algorithms for a variety of datasets and threads configurations.

Table 1. Selection sort and quicksort execution times.

Threads	Sort Alg	Datasets (of 32-bit integers)							
		10^2	10^3	10^4	10^5	10^6			
		Execution Time (ms)					avg	mean	ratio
2	Selection	0.25	0.45	1.52	2.56	5.90	2.136	1.83	-
	Quick	0.18	0.30	1.09	1.79	4.30	1.532		
4	Selection	0.22	0.37	1.27	2.12	5.25	1.846	1.67	-9%
	Quick	0.17	0.31	1.10	1.67	4.20	1.490		
8	Selection	0.20	0.29	1.10	1.98	4.97	1.708	1.58	-14%
	Quick	0.18	0.32	1.12	1.56	4.11	1.458		
16	Selection	0.17	0.25	1.02	1.88	4.56	1.576	1.48	-19%
	Quick	0.17	0.29	1.01	1.45	3.97	1.378		
32	Selection	0.16	0.23	0.97	1.82	4.48	1.532	1.44	-22%
	Quick	0.15	0.25	1.00	1.43	3.87	1.340		

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

The results show at first the differences in structure and complexity of each sorting algorithm. The quicksort is relatively more complex algorithm, however, it was faster than the selection sort. E.g., for a 1000 of iterations the selection sort requires $O(n^2)$ or about 1000000 operations where quicksort requires only about 10000. In all cases, the quicksort algorithm achieved lower execution times than selection sort.

As it is depicted in Figure 8, the results indicate that regarding the execution time data sorting takes shorter time as the number of threads increased. The increase in the number of threads indeed resulted in lower execution times, but execution did not become substantially more efficient. The ratio of the execution time on a single core with thirty-two threads to that on two threads is lower by approximately 22%, in all cases of sorting operations.

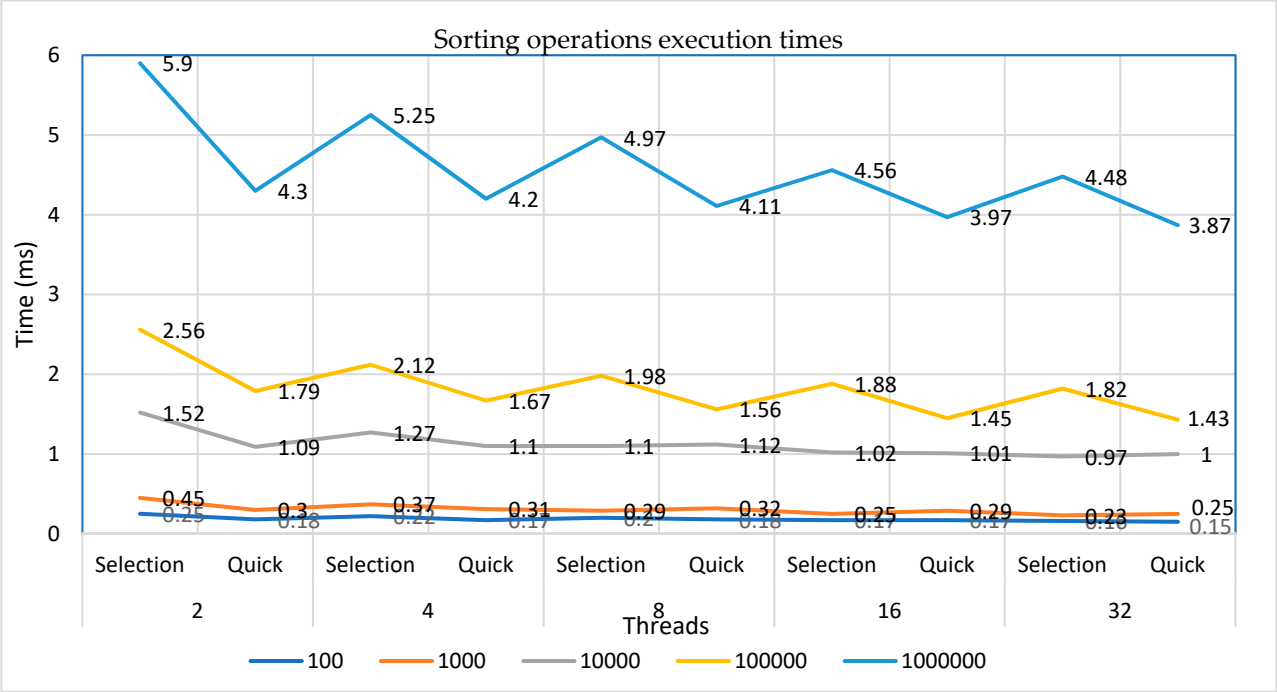

Figure 8. Sorting operations execution times.

Table 2 presents the results for the execution times for matrix additions and multiplications for a variety of datasets and threads.

Table 2. Matrix addition and multiplication execution times.

Threads	Matrix	Matrices (size of short integers) 512	1024	2048	4096	8192	avg	mean	ratio
		Execution Time (ms)							
2	Addition	0.02	0.04	0.09	0.20	0.32	0.134	0.20	-
	Multiplication	0.05	0.09	0.21	0.29	0.67	0.262		
4	Addition	0.02	0.03	0.07	0.20	0.31	0.126	0.19	-2%
	Multiplication	0.04	0.11	0.22	0.29	0.65	0.262		
8	Addition	0.04	0.05	0.06	0.21	0.34	0.140	0.21	7%
	Multiplication	0.06	0.13	0.24	0.32	0.66	0.282		
16	Addition	0.05	0.07	0.09	0.25	0.38	0.168	0.24	20%
	Multiplication	0.08	0.15	0.26	0.35	0.69	0.306		
32	Addition	0.07	0.10	0.11	0.29	0.44	0.202	0.28	41%
	Multiplication	0.12	0.20	0.31	0.40	0.75	0.356		

The multiplication of matrices even with a few thousands of elements is a heavy loaded function. Such a workload can impact the execution of the threads running concurrently on the same core.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

Regarding the core's execution performance with multiple threads, the results produced are illustrated in Figure 9. As it is evident, there is a tendency of increase on execution times for all matrix operations, as the number of threads increased. This is quite evident in the case of thirty-two threads combination. The ratio of the execution time on a single core with thirty-two threads to that on two threads is higher by approximately 41%, in all cases of matrix operations. Although the computations' workload was distributed equally across all the threads, however, the threads execution suffered from heavy contention for the same shared core resources by multiple threads.

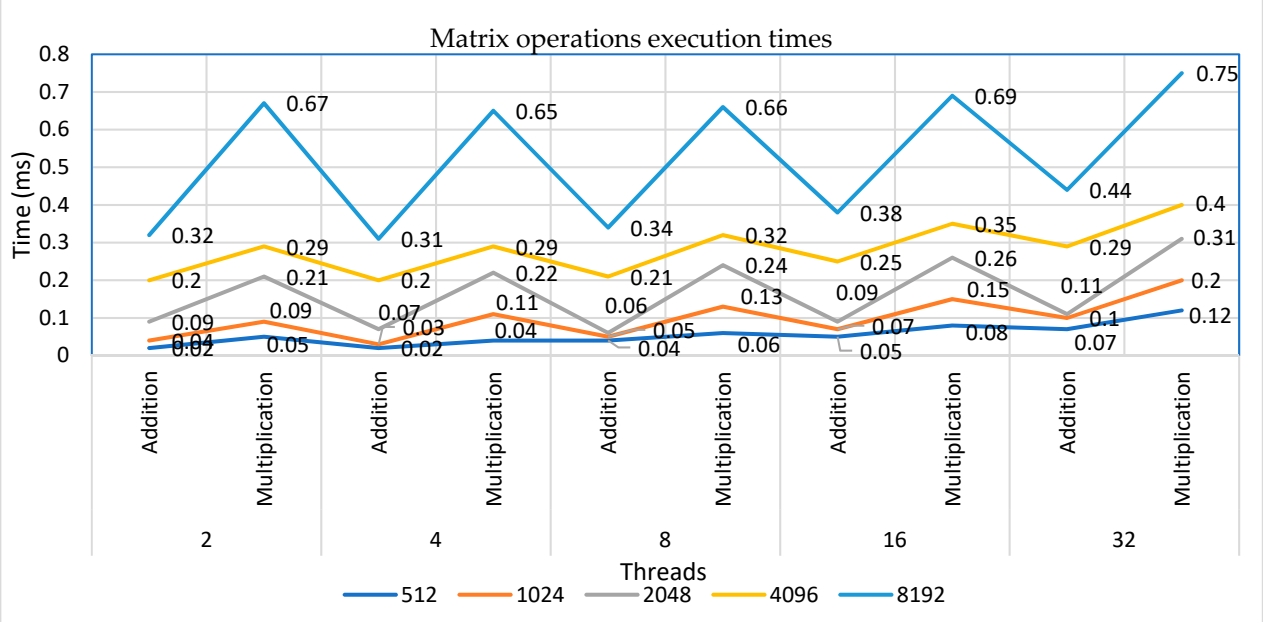

Figure 9. Matrix operations execution times.

Table 3 presents the averages of execution times, threads waiting times and response times for both sorting and matrix operations. These are illustrated in Figure 10.

Table 3. Average execution times, threads waiting times and response times.

Threads	Average execution time (ms)				Average threads waiting time (ms)				Average response time (ms)			
	Sorting ops	Matrix ops	mean	ratio	Sorting ops	Matrix ops	mean	ratio	Sorting ops	Matrix ops	mean	ratio
2	1.834	0.198	1.016	-	0.877	0.041	0.459	-	0.454	0.024	0.239	-
4	1.668	0.194	0.931	-8%	0.686	0.052	0.369	-20%	0.470	0.015	0.243	1%
8	1.583	0.211	0.897	-12%	0.824	0.046	0.435	-5%	0.512	0.023	0.268	12%
16	1.477	0.237	0.857	-16%	0.765	0.060	0.413	-10%	0.531	0.010	0.271	13%
32	1.436	0.279	0.858	-16%	0.785	0.072	0.429	-7%	0.560	0.022	0.291	22%

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

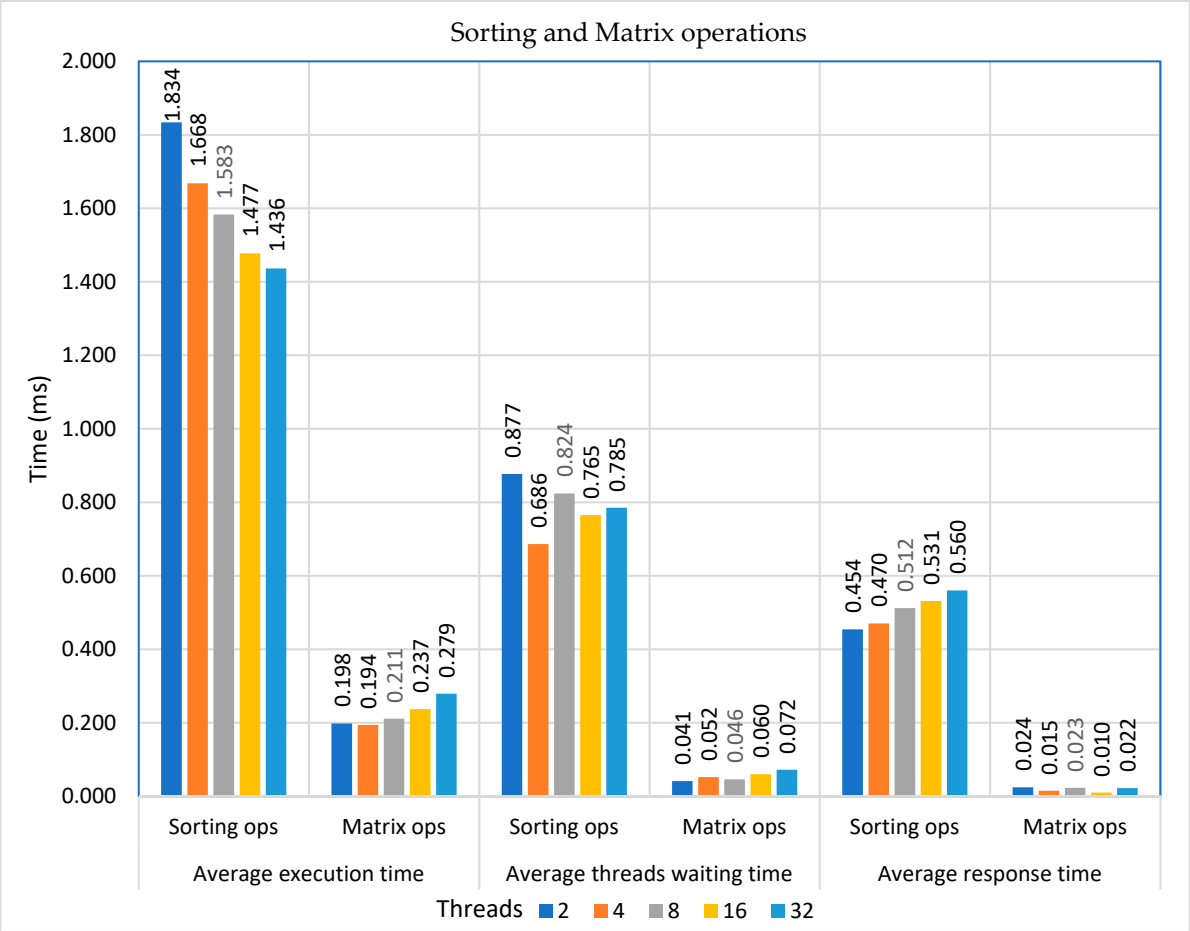

Figure 10. Sorting and matrix operations timing metrics.

As the number of threads increases up to thirty-two the average execution time decreases up to 16%, the average threads waiting time decreases up to 7%, while the average first response time increases up to 22% for all operations. The overall results show the minimum response time to be at an average value of 239 μs (case of two threads), and the maximum worst-case response (latency) to be at an average value of 291 μs (case of thirty-two threads). Therefore, a value of about 300 μs, as an upper bound, could be an acceptable safety margin for such applications in most real-time systems. These are illustrated in Figure 11. It is obvious that multiple pairs of threads, particularly in the case of thirty-two threads, led to further contention for execution resources, resulting in greater delays of the first response.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

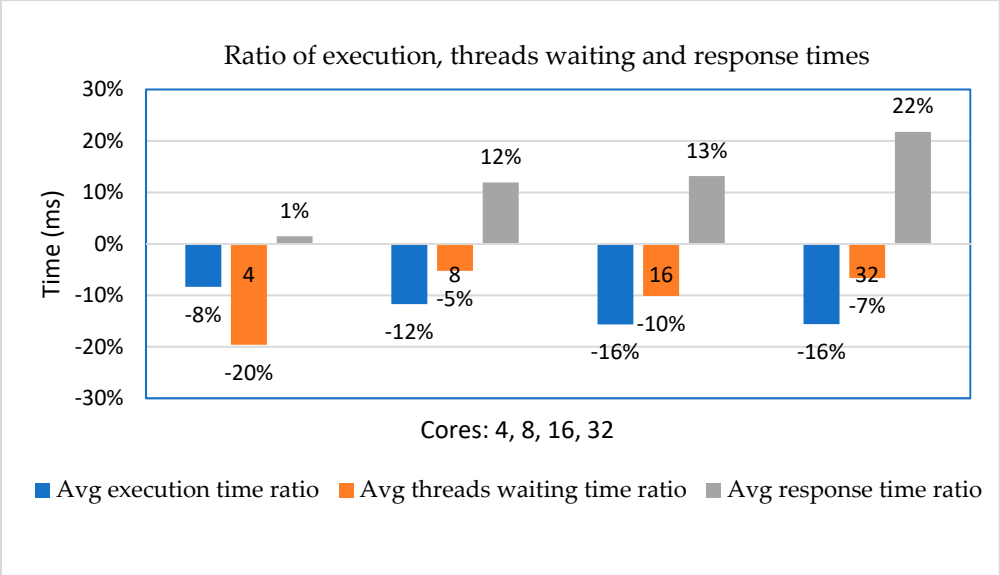

Figure 11. Ratio of execution, threads waiting and response times.

4.3. Comparative Performance Measurements

An interesting question could be how the Raspberry Pi4 and BeagleBone AI SBCs behave upon the execution of such a software module. For comparison purposes, the same application module was imported and executed on these two single board computers. Although all the devices, including the tower system (TWR-K70F120M platform) have embedded microcontrollers which are based on ARM processors and run versions of Linux distributions, however, are quite different on architecture and other specifications. Therefore, it is hard to provide generic performance conclusions. The intention was to reconfirm that the results obtained with the TWR platform are equally acceptable and comparable, rather than providing a fair comparison, since the hardware architectures and OS kernel versions are quite different.

The RPi4 and BBAI are a 4-core and 2-core devices respectively. Since none of these cores supports simultaneous multithreading, each core can only run one hardware thread at a time. However, each core can switch between several (software) threads of the running application. For this reason, in order to make the comparisons more fair and reliable, both the RPi4 and BBAI, a 4-core and 2-core devices respectively, were forced to use only a single core, as the TWR is based upon a single core processor. That is the affinity was set only to a single core (core 0).

Table 4 provides the results obtained running the application for sorting and matrix operations on those platforms, and Table 5 provides a summary of the average values for both operations.

Table 4. Timing metrics results for sorting and matrix operations on different platforms.

Platform	Linux Kernel	Threads	Average execution time (ms)		Average threads waiting time (ms)		Average response time (ms)	
			Sorting ops	Matrix ops	Sorting ops	Matrix ops	Sorting ops	Matrix ops
TWR	2.6	2	1.834	0.198	0.877	0.041	0.454	0.024
RPi4	4.14		3.12	0.23	1.3	0.8	0.9	0.008
BBAI	4.14		2.79	0.2	0.9	0.7	0.5	0.007
TWR	2.6	4	1.668	0.194	0.686	0.052	0.470	0.015
RPi4	4.14		2.89	0.17	1.2	0.6	0.8	0.006
BBAI	4.14		2.36	0.19	0.6	0.5	0.5	0.007
TWR	2.6	8	1.583	0.237	0.824	0.046	0.512	0.023
RPi4	4.14		2.5	0.14	1.2	0.7	0.7	0.005

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

BBAI	4.14		2.08	0.18	0.5	0.5	0.4	0.006
TWR	2.6		1.477	0.237	0.765	0.060	0.531	0.010
RPi4	4.14	16	2.39	0.12	1.1	0.6	0.6	0.004
BBAI	4.14		1.98	0.16	0.3	0,4	0.5	0.006
TWR	2.6		1.436	0.279	0.785	0.072	0.560	0.022
RPi4	4.14	32	2.3	0.11	1.1	0.7	0.7	0.005
BBAI	4.14		1.87	0.15	0.3	0.5	0.6	0.006

Table 5. Summary of average values on timing metrics on different platforms.

Platform	Threads	Average execution time (ms)	Average threads waiting time (ms)	Average response time (ms)
TWR		1.016	0.459	0.239
RPi4	2	1.675	1.050	0.454
BBAI		1.495	0.800	0.254
TWR		0.931	0.369	0.243
RPi4	4	1.530	0.900	0.403
BBAI		1.275	0.550	0.254
TWR		0.897	0.435	0.268
RPi4	8	1.320	0.950	0.353
BBAI		1.130	0.500	0.203
TWR		0.857	0.413	0.271
RPi4	16	1.255	0.850	0.302
BBAI		1.070	0.350	0.253
TWR		0.858	0.429	0.291
RPi4	32	1.205	0.900	0.353
BBAI		1.010	0.400	0.303

In general the results show that the TWR performs relatively well compared to the other platforms. The performance of the multithreaded application running at the TWR platform was comparable to that running at the RPi4 and BBAI, particularly in terms of execution times. The execution times in the TWR platform were quite lower than those on the other platforms, by approximately 56% in the case of two threads, and by 29% in the case of thirty-two threads. As illustrated in Figure 12, the TWR performed better as the number of threads remained small. This was expected to some extent due to the single core architecture, optimized to run more efficiently with smaller amount of threads. The threads waiting time and response time were also well comparable. The outcome is that overall, the TWR platform is indeed reliable in handling efficiently the execution of the multithreaded application.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

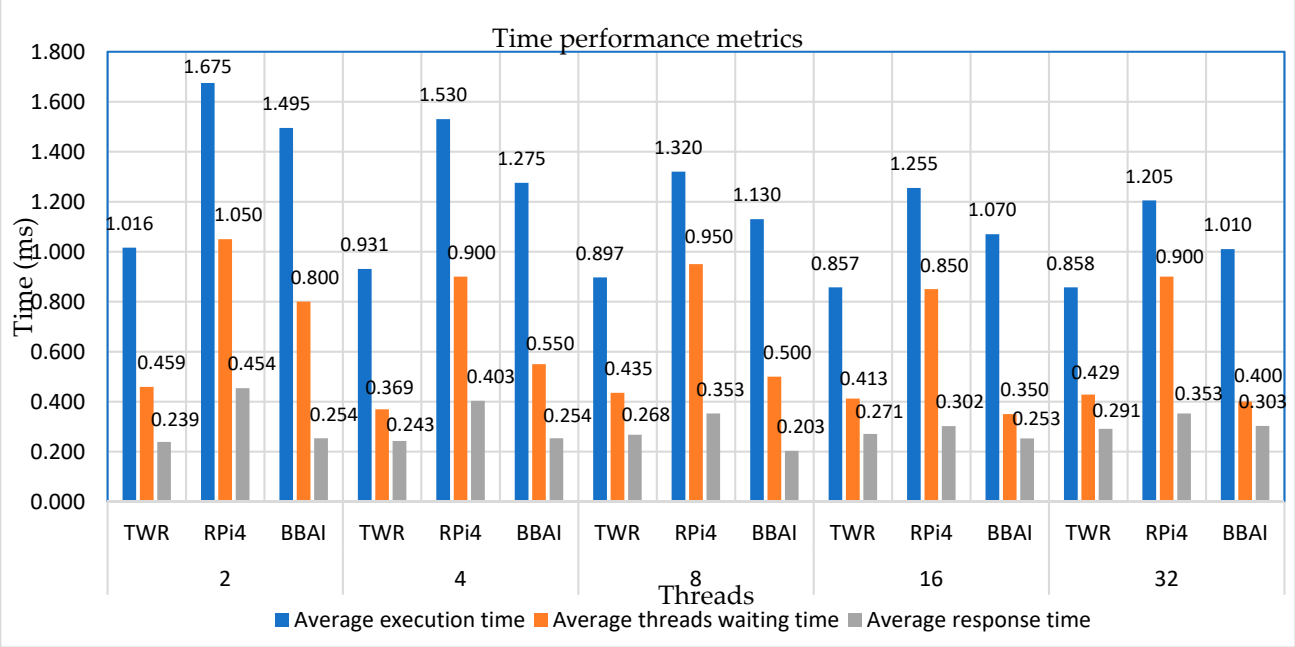

Figure 12. Comparison of average values of timing metrics on different platforms.

4.4. Benchmarking

The wolfSSL Incorporation has investigated the performance of the TWR-K70F120M platform using its in-house application wolfSSL package. This package includes a wolfCrypt benchmark application. Because the underlying cryptography is a very performance-critical aspect of SSL/TLS, this benchmark application runs performance tests on wolfCrypt's algorithms. The performance of the TWR-K70F120M platform was determined under the MQX RTOS and using the fastmath library and CodeWarrior 10.2 IDE.

In our research we applied the same benchmark, however under µClinux and using the fastmath library and CodeWarrior 11.1 IDE for the build. Typical results output shows the time it took to run each benchmark (duration in secs) and the throughput in MB/s. These are summarized in Table 6. As illustrated in Figure 13, the results obtained under µClinux are very close to those under MQX RTOS regarding the duration and throughput. This reconfirms the optimum performance of the TWR under µClinux OS.

Table 6. wolfCrypt Benchmarking summary of results for the TWR-K70F120M.

wolfCrypt Benchmark	OS	Duration (sec)	Throughput (MB/s)
AES 5120 kB	MQX	9.059	0.55
	µClinux	10.120	0.49
ARC4 5120 kB	MQX	2.190	2.28
	µClinux	2.201	2.27
DES 5120 kB	MQX	18.453	0.27
	µClinux	19.897	0.25
MD5 5120 kB	MQX	1.396	3.58
	µClinux	1.411	3.54
SHA 5120 kB	MQX	3.635	1.38
	µClinux	3.712	1.35
SHA-256 5120 kB	MQX	9.145	0.55
	µClinux	9.356	0.53

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

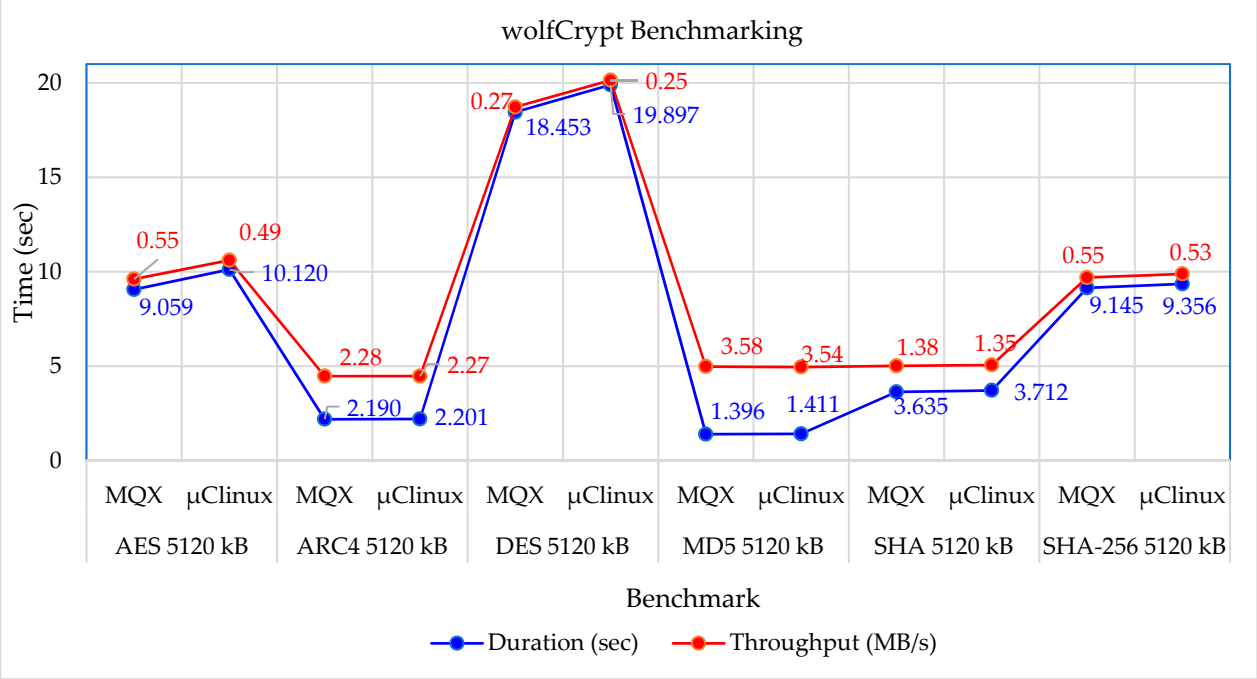

Figure 13. wolfCrypt Benchmarking results for the TWR-K70F120M under MQX and µClinux.

5. Discussion

The experiments ran efficiently for a few thousands iterations on the NXP TWR-K70F120M board, for a variety of datasets and threads configurations with equally distributed computational workload. The resources of the single Cortex-M4 core were shared concurrently by multiple threads having the same real-time scheduling policy attributes and priorities. The computations consisting of sorting and matrix operations were sufficient as workload. In addition, the differences in structure and complexity of sorting algorithms, as well the intensive matrix operations have contributed towards a comprehensive view of the execution behavior of the device under investigation.

The running scheme applied gradually increased the number of threads involved in execution of the tasks. This added substantially to the contention of the threads for the core's shared execution resources. This impact became quite obvious in matrix operations, resulting to an increase in average execution times. The ratio of the execution time on a single core with thirty-two threads to that on two threads was higher by approximately 41%, in all cases of matrix operations. On the other hand, in sorting operations the increase in the number of threads indeed resulted in lower execution times, however execution did not become substantially more efficient. The ratio of the execution time on a single core with thirty-two threads to that on two threads was lower by approximately 22%, in all cases of sorting operations.

As a general outcome, for both operations is that the average of execution time decreased, as the number of threads increased. In the case of thirty-two threads the average execution time decreased up to 16%, the average threads waiting time decreased up to 7%, while the average first response time increased up to 22% for all operations. Taking into consideration that the average execution time decreased up to 16%, the increase on the initial first response time by 22% shows that the threads gradually performed better despite the contention for shared execution resources, particularly at the first rounds of executions.

Another important metric is the maximum worst-case response (latency). This was shown to be at an average value of 291 µs (case of thirty-two threads), and 239 µs (case of two threads). Therefore, a value of about 300 µs, as an upper bound, could be an acceptable safety margin for such applications in most real-time systems.

The comparative results with RPi4 and BBAI showed that the TWR-K70F120M performed relatively well compared to these platforms, particularly in terms of execution times. Actually, the

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

execution times in the TWR were quite lower than the other devices, by approximately 56% in the case of two threads, and by 29% in the case of thirty-two threads. In general, the TWR performed better as the number of threads remained small. This was expected to some extent due to the TWR's single core architecture, optimized to run more efficiently with smaller amount of threads. The threads waiting time and response time were also well comparable.

The application of wolfCrypt benchmark on the TWR-K70F120M platform under μClinux, regarding its duration and throughput, has also reconfirmed the appropriate performance of the tower. Overall, the TWR platform is indeed reliable in handling efficiently the execution of applications with multiple threads and real-time features.

6. Conclusions

In this paper it was shown that the TWR-K70F120M platform is reliable in handling efficiently the execution of multithreaded applications with real-time attributes. This was documented primarily through experimental runs of a custom-built software application which generated and scheduled for execution multiple pairs of threads of equal workload, and with the same real-time policy. This application takes measurements on timing metrics, including task's execution time, threads waiting time and first response time. As it was investigated, to the best of our knowledge such information on timing performance metrics for this board are not available in the literature.

Comparative results on the other two favorite platforms, a Raspberry Pi4 and a BeagleBone AI, have also reconfirmed its efficient operation. Actually, the execution times in the TWR-K70F120M platform were quite lower than the others, by approximately 56% in the case of two threads, and by 29% in the case of thirty-two threads. In addition, wolfCrypt benchmark under μClinux, regarding its duration and throughput, has also reconfirmed its appropriate performance.

It is also important to state that the same application module and performance measurement methodology could be applied to other Linux-based systems and platforms. Research is already undertaken into the implementation of this device in the automation and control of machines for concrete products. The outcome of such real case studies will provide valuable feedback for further future research.

Funding: This research received no external funding.

Data Availability Statement: The experimental software module is available as an open-source project at GitHub https://github.com/gadam2018/NXP-TWR (accessed on 22 July 2023).

Acknowledgments: The author would like to thank the Computer Systems Laboratory (CSLab, https://cslab.ds.uth.gr/) (accessed on 12 June 2023) in the Department of Digital Systems, University of Thessaly, Greece, for the technical support and the resources provided for this experimental research.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. Fernández-Cerero, D.; Fernández-Rodríguez, J.Y.; Álvarez-García, J.A.; Soria-Morillo, L.M.; Fernández-Montes, A. Single-Board-Computer Clusters for Cloudlet Computing in Internet of Things. Sensors 2019, 19, 3026. https://doi.org/10.3390/s19133026.
2. Adam, G.K. Real-time performance analysis of distributed multithreaded applications in a cluster of ARM-based embedded devices. International Journal of High Performance Systems Architecture 2022, 11, 2, 105–116. https://dx.doi.org/10.1504/IJHPSA.2022.10052765.
3. Coelho, P.; Bessa, C.; Landeck, J.; Silva, C. The Potential of Low-Power, Cost-Effective Single Board Computers for Manufacturing Scheduling. Procedia Computer Science 2023, 217, 904-911. https://doi.org/10.1016/j.procs.2022.12.287.
4. Galkin, P.; Golovkina, L.; Klyuchnyk, I. Analysis of Single-Board Computers for IoT and IIoT Solutions in Embedded Control Systems. In Proceedings of the International Scientific-Practical Conference Problems of Infocommunications, Kharkiv, Ukraine, 9-12 October 2018, pp. 297-302. https://doi.org/10.1109/INFOCOMMST.2018.8632069.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

5. Prashanth, K.V.; Akram, P.S.; Reddy, T.A. Real-time issues in embedded system design. In Proceedings of the International Conference on Signal Processing and Communication Engineering Systems, Guntur, India, 2-3 January 2015, pp. 167-171. https://doi.org/10.1109/SPACES.2015.7058239.
6. Hee, Y.H.; Ishak, M.K.; Asaari, M.S.M.; Seman, M.T.A. Embedded operating system and industrial applications: a review. Bulletin of Electrical Engineering and Informatics 2021, 10, 1687-1700. https://doi.org/10.11591/EEI.V10I3.2526.
7. Ungurean, I. Timing Comparison of the Real-Time Operating Systems for Small Microcontrollers. Symmetry 2020, 12, 592. https://doi.org/10.3390/sym12040592.
8. Costa, D.G.; Duran-Faundez, C. Open-Source Electronics Platforms as Enabling Technologies for Smart Cities: Recent Developments and Perspectives. Electronics 2018, 7, 404. https://doi.org/10.3390/electronics7120404.
9. Rodrigues, J.M.F.; Cardoso, P.J.S.; Monteiro, J. Smart Systems Design, Applications, and Challenges. IGI Global: Hershey, USA, 2020.
10. KBV Research: Single Board Computer Market Size. Available online: www.kbvresearch.com/single-board-computer-market/ (accessed on 5 July 2023).
11. NXP Semiconductors: Kinetis K70 120 MHz Tower System Module. Available online: www.nxp.com/design/development-boards/tower-development-boards/mcu-and-processor-modules/kinetis-modules/kinetis-k70-120-mhz-tower-system-module:TWR-K70F120M (accessed on 4 February 2023).
12. Adam Co.: Automatic Press Machines. Available online: https://www.adam.com.gr/ (accessed on 15 March 2023).
13. µClinux. Available online: https://en.wikipedia.org/wiki/%CE%9CClinux (accessed on 11 April 2023).
14. ModBerry: Industrial Raspberry Pi with Compute Module 4. Available online: https://modberry.techbase.eu/tag/compute-module-4/ (accessed on 25 April 2023).
15. NXP Semiconductors: S32V2 Vision and Sensor Fusion low-cost Evaluation Board. Available online: www.nxp.com/design/development-boards/automotive-development-platforms/s32v-mpu-platforms/s32v2-vision-and-sensor-fusion-low-cost-evaluation-board:SBC-S32V234 (accessed on 16 January 2023).
16. RevPi: Open Source IPC based on Raspberry Pi. Available online: https://revolutionpi.com/ (accessed on 5 December 2022).
17. Wang, K.C. Embedded and Real-Time Operating Systems. Springer: Cham, Switzerland, 2017.
18. Seo, S.; Kim, J.; Kim, S.M. An Analysis of Embedded Operating Systems: Windows CE Linux VxWorks uC/OS-II and OSEK/VDX. Int. J. Appl. Eng. Res. 2017, 12, 7976–7981.
19. Adam, G.K. Co-Design of Multicore Hardware and Multithreaded Software for Thread Performance Assessment on an FPGA. Computers 2022, 11, 76. https://doi.org/10.3390/computers11050076.
20. MicroC/OS: Micro-Controller Operating Systems. Available online: https://en.wikipedia.org/wiki/Micro-Controller_Operating_Systems (accessed on 7 March 2023).
21. Aysu, A.; Gaddam, S.; Mandadi, H.; Pinto, C.; Wegryn, L.; Schaumont, P. A design method for remote integrity checking of complex PCBs. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, Dresden, Germany, 14-18 March 2016, pp. 1517-1522.
22. Musaddiq, A.; Zikria, Y.B.; Hahm, O.; Yu, H.; Bashir, A.K.; Kim, S.W. A Survey on Resource Management in IoT Operating Systems. IEEE Access 2018, 6, 8459-8482. https://doi.org/10.1109/ACCESS.2018.2808324.
23. Belleza, R.R.; Freitas, E.P. Performance study of real-time operating systems for internet of things devices. IET Softw. 2018, 12, 176–182. http://dx.doi.org/10.1049/iet-sen.2017.0048.
24. Petrellis, N.A Scalar Interpolator for the Improvement of an ADC Output Linearity. International. Journal of Engineering Science and Innovative Technology 2014, 3, 3, 591-600.
25. Arrobo, G.E.; Perumalla, C.A.; Hanke, S.B.; Ketterl, T.P.; Fabn, P.J.; Gitin, R.D. An innovative wireless Cardiac Rhythm Management (iCRM) system. In Proceedings of the Wireless Telecommunications Symposium, Washington, DC, USA, 9-11 April 2014, pp. 1-5. https://doi.org/10.1109/WTS.2014.6835035.
26. Deschambault, O.; Gherbi, A.; Legare, C. Efficient Implementation of the MQTT Protocol for Embedded Systems. J Inf Process Syst 2017, 13, 1, 26-39. https://doi.org/10.3745/JIPS.04.0028.
27. Vicari, L. Performance analysis of an embedded system. MSc Thesis, Politecnico Di Torino, Torino, Italy, 2018.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 July 2023 doi:10.20944/preprints202307.1587.v1

28. wolfSSL Inc.: Benchmarking wolfSSL and wolfCrypt. Available online: www.wolfssl.com/docs/benchmarks/ (accessed on 5 June 2023).
29. NXP Semiconductors: CodeWarrior Embedded Software Development Tools. Available online: www.nxp.com/design/software/development-software/codewarrior-development-tools:CW_HOME (accessed on 8 December 2022).
30. NXP-TWR. Available online: https://github.com/gadam2018/NXP-TWR (accessed on 20 July 2023).

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.