**Bulletin of Electrical Engineering and Informatics**
Vol. 10, No. 3, June 2021, pp. 1687~1700
ISSN: 2302-9285, DOI: 10.11591/eei.v10i3.2526 

# Embedded operating system and industrial applications: a review

**Yew Ho Hee, Mohamad Khairi Ishak, Mohd Shahrimie Mohd Asaari, Mohamad Tarmizi Abu Seman**
School of Electrical and Electronics Engineering, Engineering Campus, Universiti Sains Malaysia,
Nibong Tebal, 14300, Penang, Malaysia

**Article Info**

***Article history:***

Received Apr 20, 2020
Revised Mar 5, 2021
Accepted May 3, 2021

***Keywords:***

Cooperative
Internet of things
Operating system
Real-time operating system
Super loop

**ABSTRACT**

The complexity of an embedded system is directly proportional to the requirements of industrial applications. Various embedded operating system (OS) approaches had been built to fulfil the requirements. This review aims to systematically address the similarities and differences of the embedded OS solutions and analyse the factors that will influence decision-making when choosing what solution to use in the applications. This paper reviews three standard solutions; super loop, cooperative, and real-time operating system (RTOS). These are commonly used in industrial applications. By grouping the tasks in the foreground and background execution region, the concept and working principle of each of them are reviewed. The unique feature of RTOS in the context of task switching was used to define the deterministic characteristic of meeting the deadlines. The importance and performance of this characteristic is addressed and compared among various solutions in this paper. Subsequently, this paper reviewed the internet of things (IoT) requirements, automotive, medical and consumer electronics industry. The influential factors on choosing the right embedded OS to be used are extracted based on the requirements. They are reviewed in the perspective of memory footprint, regulated standards, cost-effectiveness, time effectiveness, and scalability.



***Corresponding Author:***

Mohamad Khairi Ishak
School of Electrical and Electronic Engineering
Engineering Campus, Universiti Sains Malaysia
Nibong Tebal, 14300, Penang, Malaysia
Email: khairiishak@usm.my

## 1. INTRODUCTION

An embedded system integrates both electronic and software that is designed to run a defined function. A modern embedded system usually comes with a microcontroller, which could be programmed to perform various tasks such as temperature sensing, battery level sensing, and acceleration data retrieving from accelerometer. This system is used in many applications such as air conditioning, remote commander, car entertainment system, flight navigation system, robotic automation in factory, MP3 player, smartphone, and smart watch. Personal computers (PC) that run general-purpose operating systems such as Windows, Linux and Mac OS could accomplish a lot of tasks and are more resources hungry in high processing power, graphic processing, and memory usage. In contrast, embedded software is specifically designed for an application. For example, MP3 players could only do one job well that is audio playback while a PC could be used for video playback, audio playback, graphic editing, and text editing.

An embedded OS is typically designed for resource constrained microcontrollers, especially the memory footprint such as read-only-memory (ROM) and random-access-memory (RAM). A typical PC

usually comes with gigabytes of RAM and terabytes of hard disk space, whereas memory in microcontroller is extremely small when compared with a PC. For example, the 8-bit microcontroller STM8S003F3 has only 8 Kilo bytes (Kbytes) of ROM and 1 Kbyte of RAM [1], while the 32-bit microcontroller STM32F103R8 has 64 Kbytes ROM and 20 Kbytes RAM only [2]. This is why an embedded OS is naturally built to be resources sensitive and efficient. This paper will cover three types of embedded OS solutions, which are commonly used in applications with complexity ranging from mid-level until low-level. This can be applications such as MP3 player, TV, remote control, air conditioning, and refrigerator. The solutions are super loop, cooperative, and real time operating system (RTOS). The concept of how they work is explained from the big picture to the software level. The accuracy of meeting deadlines is also explained with example scenarios.

Lastly, this paper includes a review of how embedded OS is applied in industrial applications. The review encapsulated the internet of things (IoT), automotive, medical and customer electronics industry. Each industry has different standards and requirements, especially to regulate each application's robustness and safety issues. This will define the nature of how embedded OS will be designed to fit in accordingly. Apart from that, when the factors such as time to market and cost-effectiveness are included, the complexity of selecting the proper embedded OS will be increase. How these factors relate to the embedded OS in industrial applications is reviewed and explained.

## 2. EMBEDDED OPERATING SYSTEM

### 2.1. Super loop

#### 2.1.1. Concept of super loop

The operating system on small embedded systems is typically designed in a foreground and background pattern [3], [4]. As illustrated in Figure 1, the background region contains the tasks which are to be executed infinitely. When an interrupt is triggered, the background tasks are preempted and the software will switch over to the interrupt service routine (ISR), which conceptually belongs to the foreground region. This process is called preemption. After the ISR is served, it will go back to the exact point where it had stopped previously in the background region. This is exactly how super loop embedded OS works. The tasks in the background region are executed sequentially. The next task will be executed only when the previous task is served. When the last task in the sequence is served, it will go back to the first task and start over again from top to bottom.

#### 2.1.2. Working principle of super loop

When the program is started, it will go through the pin configuration, clock tree and component drivers before entering the infinite loop. After it enters into the infinite loop, it will never exit. The tasks inside will be executed cyclically as illustrated in Figure 2. There are three tasks (Task 1, Task 2 and Task 3) in the background region and each of them will take 4 milliseconds (ms), 2 ms and 3 ms to execute, respectively. The Systick timer is configured to be triggered every 3 ms. At the beginning, when the timer is triggered, Task 1 is halted and preempted while in the middle of execution and the program will go to serve IrqSystick in the foreground region. After IrqSystick is served, the program will resume the execution of Task 1 from the point where it had been halted previously. This preemption operation will repeat every 3 ms during runtime and the task that is to be preempted will depend on which task is running when the interrupt is triggered. In other words, this could be unpredictable.

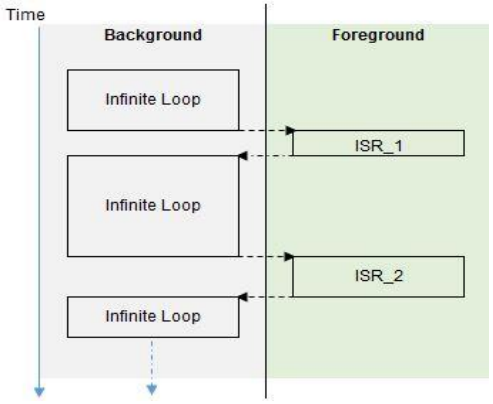



Figure 1. Concept diagram on how super loop work

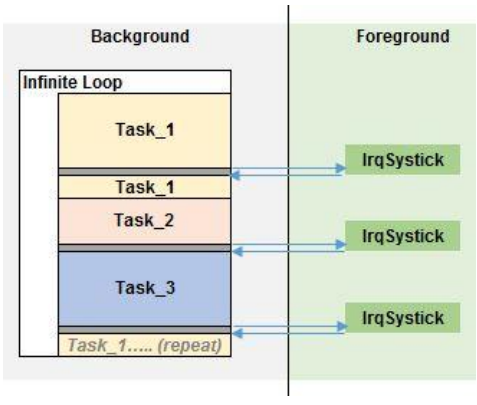



Figure 2. Task scheduling in super loop

Preemption is particularly important for the deterministic behavior in the super loop solution. For time critical tasks, it is usually handled by the ISRs in the foreground region to ensure that the deadline is not missed. This is because the preemption is fast and it will accurately respond to how the timing or interrupt is defined. This is particularly important if there are many heavy loaded tasks in the background region. As illustrated in Figure 3, there are 10 tasks and each of them takes 10 ms to execute. The background region will be loaded with 100 ms capacity, and in principle, each individual task will be served only every 90 ms. If there is an external signal which the system has to monitor every 10 ms, the deadline will be missed. This is particularly important for critical applications such as automation industry, with safety as the top priority. Thus, Super Loop is usually in-house and used in non-critical and minor application.

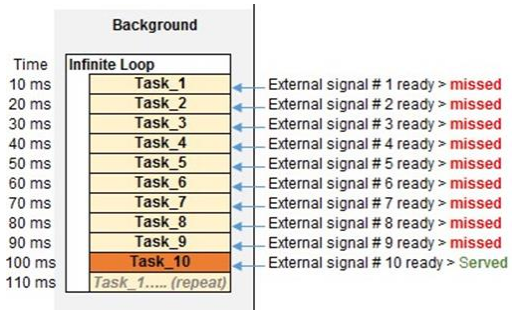



Figure 3. Super loop-external signal missed 9 times before it is served

## 2.2 Cooperative OSs

### 2.2.1. Concept of cooperative

Cooperative scheduler is probably one of the most used schedulers in embedded systems. The fundamental concept is the same as how the super loop scheduler works-with foreground and background region. Unlike the tasks which are executed sequentially and cyclically in super loop, the tasks are grouped by time slots. They will run only when the particular time slot is active. When the time slot is active, the tasks inside will be executed sequentially and they will be served only once. After that, they will remain idle until their time slot is active again. In principle, cooperative scheduler is more organized and more deterministic than a super loop [5], [6]. As illustrated in Figure 4, the timer is configured to keep track of time. When the timer interrupt is triggered, the background tasks will be preempted and the timer ISR routine in the foreground will be served to record the time [7]. For example, 1 second had passed, 2 seconds had passed, and so on. Depending on how the requirements are defined, a flag can be set to active when the specific time is reached inside the ISR. For example, Timer 1 flag is set to active when 1 second is reached, Timer 2 flag is set to active when 2 seconds is reached. After that, the program will return to the exact point where it had been halted previously in background region. With this mechanism, the scheduler could selectively execute the tasks that belong to specific time region. For example, Task 1 will be executed only when Timer 1 flag is active and Task 2 will be executed only when the Timer 2 flag is active. Contiki and TinyOS are among the OSs which adapted Cooperative mechanism.

### 2.2.2. Working principle of cooperative

In this example, the Systick timer will be triggered at every 1 ms, as illustrated in Figure5. When Systick times out, the background task will be preempted and the program will move to the IrqSystick ISR at the foreground region. Inside the ISR, the timer flags of 5 ms, 10 ms, 15 ms will be set to active respectively when the time hits. In this setup, the 5 ms flag will be activated every 5 ms, 10 ms flag will be activated and so on. As illustrated in Figure 5, Task 1 is located in the 5 ms time slot, Task 2 is located in the 10 ms time slot and Task 3 is located in the 15 ms time slot. After the program is started, Task 1 will be the first task to be executed when the 5 ms flag is activated. After that, the program will remain idle until the next time event is triggered. When 10 ms hits, both 5 ms flag and 10 ms flag will be activated on the timeline. Thus, Task 1 will be executed again for second time, and later followed by Task 2. Depending on how the orientation is designed, the sequence of task execution could be changed. On the timeline, when 15 ms hits, 5 ms flag will be activated together with the 15 ms flag. Thus, Task 1 will be executed again for the third time, followed by Task 3.

With this scheduling methodology, time resources can be defined and distributed easily, suggesting a more deterministic and controllable behaviour than the super loop scheduler. This is particularly crucial for tasks that need to be served consistently and accurately to ensure that the deadline is not missed. A good example will be key scanning which is very common in embedded systems. As shown in Figure 6, KeyScan

is designed to check the key input every 20 ms. Naturally, the key press event will not be missed within 20 ms timeframe because the time interval between key presss and key release will be more than 20 ms. With this mechanism, more detection patterns can be further developed, such as short-pressed, long-pressed, and press and hold.

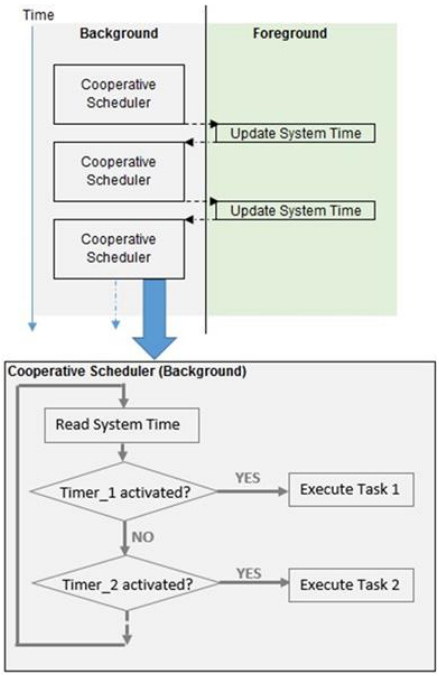



Figure 4. Concept diagram of how cooperative scheduler works

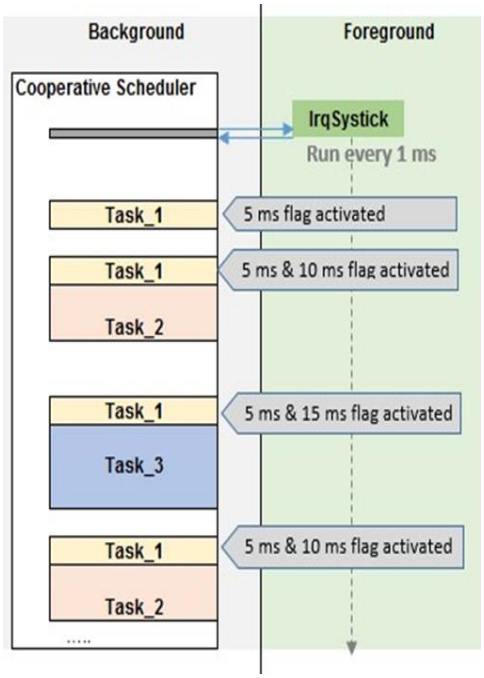



Figure 5. Tasks scheduling in cooperative scheduler

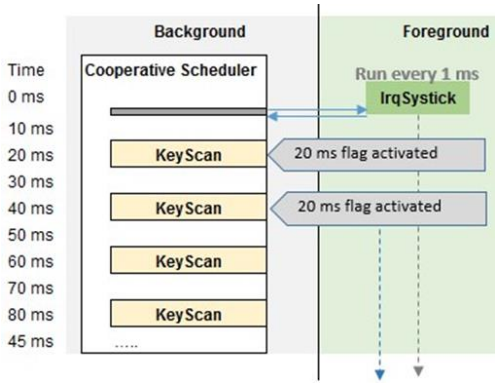



Figure 6. Key scanning in cooperative scheduler

Even though the key detection by external interrupt would be more accurate, the advantages such as matrix keypad scanning and freedom to use non-external interrupt supported general purpose input output (GPIO) pins suggest that the time-scheduled scanning is a better approach.

### 2.3 Real-time operating system

#### 2.3.1. Concept of RTOS

Real-time operating system (RTOS) is much more complicated than super loop and cooperative scheduler. It is all about the deterministic capability that makes it special and powerful. In principle, there are two types of RTOS: hard RTOS and soft RTOS [8]. The hard RTOS always meets the deadline, while the soft RTOS can meet the deadline most of the time. The hard RTOS will fit in for time critical real time

applications such as in automotive and military industry where missing any deadline would mean a disaster [9]. The car airbag is a good example. If the time requirement to activate the airbag must be less than 50 ms right after the impact is detected, then the system must respond before the deadline hits.

Unlike super loop and cooperative scheduler which can be easily and quickly made in-house, the industry usually relies on third-party solutions to RTOS. The core of RTOS is called the kernel or scheduler. It is responsible to manage the tasks in the system. With the scheduling methodology such as round robin with time slicing, a task can be purposely halted during execution when timed out to have another task with the same priority level to start executing [10]. This is also called multitasking. It creates an illusion that multiple tasks are running in parallel even with a single core processor, but in fact only a single task is being executed in a single time. Figure 7 demonstrates how round robin with time slicing works. Task A, Task B, and Task C are created with equal priority. Task A, Task B, and Task C require three, two, and one time slots respectively, to finish execution. The time slots are divided equally (T1=T2=T3)

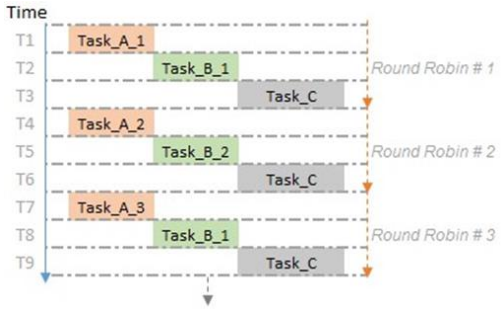



Figure 7. Tasks are time sliced and scheduled in round robin pattern

The industrial RTOS's scheduler is far more sophisticated than just time slicing. Figure 8 presents how the whole system looks like in a commercial RTOS-µC/OS-III [11]. In principle, it is encapsulated in a background and foreground design pattern. The tasks are located in the background region while the ISR is located in the foreground region. The low priority tasks are scheduled in the round robin time slicing pattern. When an interrupt occurs, the running task is preempted, and the program will move to the ISR to serve the interrupt. The ISR will trigger a high priority task to active. Right after the ISR is served, the scheduler will notice that a higher priority task is being activated. Thus, it will serve the higher priority task rather than go back to the previous halted task. With this mechanism, the deadline can be guaranteed. Once the higher priority task is served, the previous halted task will be resumed.

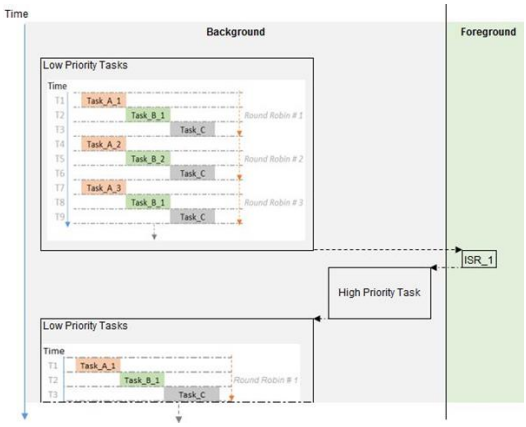



Figure 8. An overview of how RTOS works

### 2.3.2. Working principle of RTOS

Figure 9 demonstrates how the car airbag can be triggered with the RTOS solution. The priorities for each task and execution time are shown in Table 1. TriggerAirBag has the highest priority among all the

tasks as the deadline must not be missed. The time slicing is configured to 2 ms. Hence, Task 1, Task 2, and Task 3 which is configured with the same priority level will be executed sequentially with a task switch at every 2 ms.

Table 1. Tasks priority and execution time information for car airbag example

| Tasks | Priority | Execution Time |
|---|---|---|
| TriggerAirBag | 1-High | ~ 1 ms |
| BuzzerOn | 2-Mid | ~ 2 ms |
| Task_1 | 3-Low | ~ 1 ms |
| Task_2 | 3-Low | ~ 2 ms |
| Task_3 | 3-Low | ~ 2 ms |

When the program starts, Task 1 is executed twice before it switches to Task 2 as its execution time is 1 ms. Task 2 will be executed until it finishes as the execution time is 2 ms. Afterward, Task 3 will start and at the middle of execution (timeline 5 ms), an interrupt is triggered as the system detected that the safety belt on driver seat is not locked. The product requirements request that the system shall respond with the beeper sounding in this scenario.

Subsequently, Task 3 is preempted and the software will move to serve the ISR. The BuzzerOn task will be set to active by the ISR in order to generate the beeper sound. When the program returns, it does not move back to Task 3 as the scheduler detected that a higher priority task (BuzzerOn) is activated. BuzzerOn will now be served, but shortly after that, the system detected a hit impact on timeline 6 ms. BuzzerOn will be preempted and the system will move again to ISR [10]. The task TriggerAirBag will be set to active and when the program returns, it will move to TriggerAirBag as it has higher priority than BuzzerOn. BuzzerOn will be served to finish only after TriggerAirBag is served. Task 3 will be resumed again after the program returns from BuzzerOn.

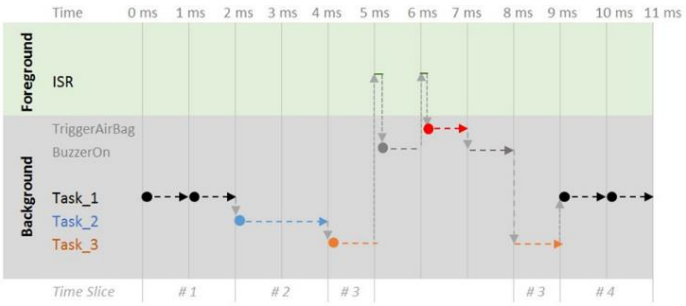



Figure 9. Example of how the car airbag triggering works in RTOS

With this mechanism, the deadline of the task with higher priority such as TriggerAirBag can be a safeguard as it would be served right after it is activated. The swiftness of response will not be guaranteed if the same scenario is handled by a cooperative scheduler, as shown in Figure 10. In this design, the TriggerAirBag will be called at every 5 ms and it will be triggered only if there is a hit impact. The hit impact is detected at timeline 12 ms, but the system will experience 3 ms delay to respond as TriggerAirBag will be served again only at timeline 15 ms.

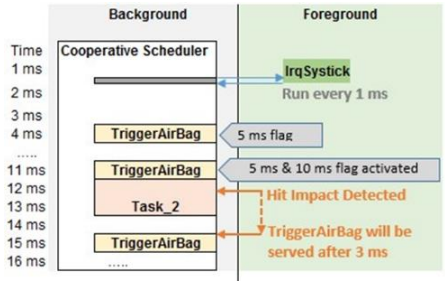



Figure 10. TriggerAirBag could not be triggered instantly in cooperative scheduler

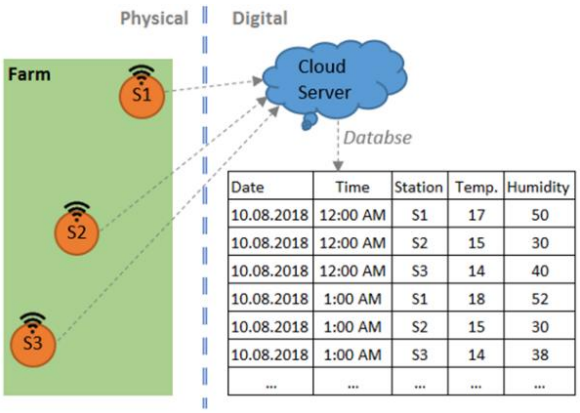
## 3. INDUSTRIAL APPLICATIONS

The world gone through a few waves of the industrial revolution (IR) before reaching this high technological level as what we could produce today. The first wave started about year 1700 in Great Britain where the industry experienced a transformation from hand manufacturing methods to machine, which was driven by steam power [12]. The second wave began in the beginning of the 20th century when electricity became the primary source of power as it is more consistent and cleaner than water and steam [12]. The industrial applications expanded with the invention of electrical systems and correlated new designs. It grew rapidly until the last few decades of the 20th century. Industrial 3.0 kick started with a higher level of automation in machinery. The invention of transistor and later the integrated circuit with microcontroller solutions had greatly driven this automation. The higher the automation is, the greater the productivity is as there is lesser uncertainty from human error.

The waves of industrial revolution had not only changed the way on how we produce things, it has influenced on what we could produce as well. The capability to design and the capability to produce a product is correlated with each other as they grew and expanded together. Aside from improving productivity, we could now design products with higher complexity such as cars, airplanes, and air conditioning, which was unthinkable before the industrial revolution. The more complex a product is, the more demanding the software is. The capability of what an embedded OS could do evolved with the needs of the application. This is how the embedded OS started from the simple super loop and gradually developed to the RTOS solution. What an embedded OS is best fit for depends on a lot of factors. The sections below will cover how industry had used the super loop, cooperative driven and RTOS in applications. IoT, automotive, medical and consumer electronics domain are discussed as most related to most daily lives.

### 3.1. Internet of things

The revolutionary move of industry did not end at 3.0 with the achievement in automation. IR 4.0 had started years back, focusing on integrating the physical, digital, and biological field. Since then, breakthrough and advancement in technologies such as the IoT, artificial intelligence (AI), interconnected automation, and biotechnology is flourishing. IoT is the integration of an embedded system with sensors and connectivity to the Internet for data collection. Various improvements could be achieved with post-processed information. Figure 11 illustrates an example of how IoT could be applied for agricultural use. Embedded systems with temperature and humidity sensors (S1, S2, and S3) are installed at different locations in the farm. The systems are connected to a cloud server and the data is transmitted hourly and daily. Later, database in the cloud server is filled with a huge amount of data. With the help of post data processing, the change in temperature and humidity level inside the farm could be visualized and potential problems such as temperature too high at any particular time could be identified. Thus, accurate improvement action could be rolled out and this helps to improve productivity. The farmers in China already apply IoT for dairy cows. Sensors that gather data such as temperature and heart rate were installed and the data helped to improve milk production. It helped farmers to make an extra USD420 each year with the profit increased by 50% annually [13].




| Date | Time | Station | Temp. | Humidity |
|---|---|---|---|---|
| 10.08.2018 | 12:00 AM | S1 | 17 | 50 |
| 10.08.2018 | 12:00 AM | S2 | 15 | 30 |
| 10.08.2018 | 12:00 AM | S3 | 14 | 40 |
| 10.08.2018 | 1:00 AM | S1 | 18 | 52 |
| 10.08.2018 | 1:00 AM | S2 | 15 | 30 |
| 10.08.2018 | 1:00 AM | S3 | 14 | 38 |
| ... | ... | ... | ... | ... |

Figure 11. Sensors installed on farm to monitor the changes of temperature and humidity over time

IoT is an important subject in the industry nowadays. According to the embedded market survey which was conducted by AspenCore in 2017 [14], 50% among 1234 respondents voted that IoT development

will be important to themselves and their organization in the next 12 months. At the same time, 63.5% of respondents suggested that they will have one or more projects devoted to IoT. The survey also suggests that IoT is applied mostly for industrial application (19.7%) such as connected robotic automation, and subsequently followed by the sensor-driven application (18.3%) [14].

IoT will not be possible without connectivity. Various wireless interfaces and protocols are deployed on IoT applications. IEEE 802.15.4, Bluetooth Low Energy and LPWAN are among the common wireless interfaces for IoT devices [15], [16]. IEEE 802.15.4 defines the protocol and compatible interconnection for data communication devices using low-data-rate, low-power, and low-complexity short-range radio frequency (RF) transmissions in a wireless personal area network (WPAN). Various protocols built upon IEEE 802.15.4 standard, such as IPv6 over low-power wireless personal area networks (6LoWPAN), routing protocol for LLNs (RPL), and constrained application protocol (CoAP) [17], [18]. They are standardized by internet engineering task force (IETF) and applicable to be used on home automation, wireless sensor networking, and manufacturing IoT applications. Besides that, non-IETF standardized protocols designed upon IEEE 802.15.4 are also available, such as ZigBee, wireless highway addressable remote transducer (HART), MiWi by Microchip Technology, and ISA 100 by International Society of Automation (ISA) [18], [19].

Low-power wide-area network (LPWAN) is designed to have long-range communication at low baud rate among connected objects. Various protocols are designed for LPWAN, such as long range (LoRa), SigFox, random phase multiple access (RPMA), and symphony link. Bluetooth low energy (BLE) is designed for constrained low power application compared to classic Bluetooth. There are many profiles available on top of BLE, such as mesh profiles, health care profiles, and generic sensors [18], [19]. Table 2 lists some IoT oriented embedded OS with supported IoT protocols. The solutions that come with network-ready protocols could save implementation time. Simultaneously, the design that is naturally dedicated for constrained devices with lower power and small memory footprint could fit into small systems. Table 3 lists some IoT oriented solution with memory footprint information. From the Table, RIOT-OS and TinyOS memory footprint is very small in comparison to other solutions. They are occupying only 5 Kbytes and 4 Kbytes of ROM code respectively. This could easily fit into many small-sized microcontrollers.

Table 2. Embedded OS with supported IoT protocols and scheduler type

| Name | Scheduler | IEEE 802.15.4 | | | | LPWAN | BLE Beacon | License | Ref. |
|---|---|---|---|---|---|---|---|---|---|
| | | 6LoWPAN | LoRa | CoAP | ZigBee | LoRa | | | |
| Contiki | Cooperative | Yes | Yes | Yes | - | - | - | BSD | [20],[21],[22] |
| RIOT-OS | RTOS | Yes | Yes | Yes | - | - | - | GNU LGPL | [20],[21],[22] |
| Mbed | RTOS | Yes | - | - | - | Yes | Yes | MIT | [20],[21] |
| TinyOS | Cooperative | Yes | Yes | - | - | - | - | BSD | [20],[21],[22] |
| LiteOS | RTOS | Yes | - | - | Yes | - | Yes | BSD | [20],[21],[22] |
| Zephr | RTOS | Yes | Yes | Yes | - | - | Yes | BSD | [23],[24] |

Table 3. IoT oriented solution is designed for resources constrained devices with small memory footprint

| Name | Scheduler | Memory Footprint | | Ref. |
|---|---|---|---|---|
| | | ROM (Kbytes) | RAM (Kbytes) | |
| Contiki | Cooperative | 30 | 10 | [20],[21],[22] |
| RIOT-OS | RTOS | 5 | 1.5 | [20],[21],[22] |
| Mbed | RTOS | 16 | 4 | [20],[21] |
| TinyOS | Cooperative | 4 | 1 | [20],[21],[22] |
| LiteOS | RTOS | 26 | 6 | [20],[21],[22] |
| Zephr | RTOS | 50 | 8 | [23],[24] |

To build the IoT application more effectively, IoT-LAB [25] testbeds could be facilitated. It is an environment where users could test the connectivity performance and build the application code without having the actual hardware setup at their site. The tests will be done on the setup at IoT-LAB site. It offers a combination of both 16-bit and 32-bit nodes, which basically covered most use case on the field. RIOT, Contiki and Zephyr are among the RTOS which is available to the users on IoT-LAB testbeds.

### 3.2. Automotive industry

Selecting the right embedded OS to use is important for the automotive industry as their requirements are much stricter compared to consumer electronics due to higher safety requirements. As it is a time critical application, RTOS will naturally fit into automotive industry. On top of the strict requirements not to miss the deadline, the requirements on the code's reliability are also regulated. Automotive companies and academic units had formed a few consortiums to define the standards in the embedded operating system.

Motor industry software reliability association (MISRA) was formed in response to the UK Safety Critical Systems Research Programme. It produced coding guidelines to ensure safety, security, portability and reliability for embedded software [26], [27]. Currently, the guidelines are available for C and C++ programming language. For the software to be compliant to MISRA, all the mandatory rules must be met. It will be a heavy task to manually check the code compliance line by line as it is not uncommon for a standard mid-range car to have over a million code lines. There are automated tools developed to check the code for compliancy. There are also industrial RTOS that is MISRA compliant. The certified RTOS will help save the cost and time needed to validate the OS part in software [27]. In principle, the bigger the system is, the greater the time and cost could be saved. Table 4 lists some of the industrial RTOS which is compliant with MISRA C.

OSEK/VDK and AUTOSAR are other well-known automotive standards. OSEK is a German abbreviation for "open systems and their interfaces for electronics in motor vehicle" [28]. It was founded in 1993 by a German automotive company consortium (BMW, Robert Bosch GmbH, DaimlerChrysler, Opel, Siemens, and Volkswagen Group) and the University of Karlsruheand [28], later joined by the French automotive company consortium (Renault and PSA Peugeot Citroën) [28] called vehicle distributed executive (VDX) in 1994. Thus, it is called OSEK/VDK. It is later encapsulated in AUTOSAR, which defined a broader standard for automotive systems. Unlike MISRA C or C++, which focuses on how the code is written, OSEK/VDK focuses on producing the software's requirements such as the key characteristic needed for a safety-critical system. All the objects are required to be built during build time and no dynamic objects during runtime will be created. This leads to minimizing the uncertainty during runtime and promote a safer system [27], [29]. There is a handful of industrial RTOS compliant to OSEK/VDK standard as listed in Table 4.

Table 4. Embedded RTOS with the automotive software standards certified

| RTOS | MISRA C | ISO 26262 | OSEK/VDX | AUTOSAR | License | Ref. |
|---|---|---|---|---|---|---|
| Erika Enterprise | Yes | - | Yes | Yes | GPL and GPL linking exception | [30] |
| FreeOSEK | - | - | Yes | - | GNU GPLv3 | [31] |
| Arc Core | - | - | Yes | Yes | GPL/proprietary | [32] |
| OpenOSEK | - | - | Yes | Yes | LGPL | [33] |
| Elektrobit tresos | - | Yes | Yes | Yes | Proprietary | [34] |
| ThreadX | Yes | Yes | - | - | Proprietary | [35] |
| RTX5 | Yes | - | - | - | Proprietary, royalty free | [36] |
| SafeRTOS | Yes | Yes | Yes | - | Proprietary | [37] |
| µC/OS-III | Yes | - | - | - | Proprietary | [38] |

### 3.3. Medical industry

For medical industry, IEC 62304 (Medical Device Software-Software Life Cycle Processes) is a widely adopted standard. It defines the life cycle requirements for medical device software. It covers the beginning stage where the development planning starts until the software is released [39], [40]. Similar to the automotive industry, making medical software is relatively regulated. The more complicated the products is, the more complex the embedded OS will be. The higher complexity in software is directly proportional to the difficulty level to meet the standard. Thus, using the IEC 62304 certified embedded OS will accelerate the product development progress [41].

Even not all medical devices require deterministic behaviour, but commercial embedded OS usually comes with RTOS approach. The characteristic of RTOS can handle both deterministic and non-deterministic operations and the small memory footprint requirement, promote the scalability to run on simple and complex applications. For instance, SafeRTOS only require 6 kB to 15 kB ROM, 500 bytes RAM and 400 bytes/task Stack to run [38]. This could easily fit into a small microcontroller such as STM32F030K6, which contains 32kB ROM and 4kB RAM. Various embedded RTOS is certified with IEC 62304, Table 5 lists some of them. Aside scalability, the support level which provided by RTOS supplier is also a major consideration. Along the product development, support services such as prototype bring-up activies, issues solving, consultation, training, attentiveness, and responsiveness to query are major concerns when selecting the supplier, aside from licensee cost.

Table 5. Embedded RTOS with IEC62304 certified

| RTOS | IEC 62304 | License | Ref. |
|---|---|---|---|
| SafeRTOS | Yes | Proprietary | [38] |
| embOS Safe | Yes | Proprietary | [40] |
| µC/OS-III | Yes | Proprietary | [39] |
| QNX | Yes | Proprietary | [42] |
| ThreadX | Yes | Proprietary | [36] |

### 3.4. Consumer electronics

Comparing to automotive and medical industry, the software for consumer electronics is much less restricted and regulated. For example, if the home air conditioning system is late for 200 ms to serve the received remote commander code to turn off, it will not have a big impact at all to the user. As long as it turns off, it is fine.

Decision on the embedded OS to be used is correlated to cost effectiveness. The simpler the solution, the lower the cost. The saved cost is especially visible for products with large quantities. A huge amount will amplify a reduction of 10 cent USD in proportion to the quantity. Super loop OS and cooperative scheduler will consume a minimal amount of memory footprint (almost none) compared to the industrial RTOS solution. The memory footprint in this context refers to both read-only-memory (ROM) and random-access-memory (RAM). For example, the QP-Nano [43] which is one of the lightest RTOS solutions already require 2 kB of ROM space to house it. The memory footprint requirements for typical RTOS solution is even more demanding-10 kB of ROM and 10 kB of RAM [43]. If the application is planned to use an 8-bit microcontroller with 8kB of ROM, the QP-Nano already occupies 25% of ROM space. In this context, super loop and cooperative scheduler will be a better option as long as the product requirements can be met.

What will be the scenario to consider RTOS? Aside from the deadline requirement, the industry used it to integrate complex software modules [44]. More and more component makers such as the Bluetooth module and WiFi module provide software modules with an RTOS framework. The effort to integrate them will be easier if the software platform is similar. Some industrial RTOS such as FreeRTOS provides an integrated solution for FAT (file allocation table) file system, TCP/IP network stack, graphical user interface (GUI) libraries, and USB stacks. FAT is used on the application with a mass storage component, such as serial flash for the purpose of file writing, reading and deleting. GUI library is usually composed of a toolset which allows the user to draw an image on the display, such as a straight line, a curve line, and a round shape filled with blue color. This would simplify and speed up the product development and ensure the product is delivered on time to the market. Table 6 lists the industrial RTOS with their integrated solutions.

At the same time, applying RTOS in the industry which is already certified by industrial standards such as IEC 60730 (Safety Standard for Household Appliances) and IEC 60335 (Household and similar electrical appliances-Safety) is also one of the main considerations [45]. Example of home appliances for both of these standards are washing machines, refrigerators, power tools, air conditioning, and electric water heater. The consideration is that certified embedded OS could save time and cost whereas the certification process can be daunting. Table 6 lists some of the industrial RTOS which is certified with IEC 60730 and IEC 60335.

Table 6. Embedded RTOS with supported featues (TCP/IP, File System, GUI, USB) and IEC 60730, IEC 60335 certified

| RTOS | TCP/IP | File System | GUI | USB | IEC 60730 | IEC 60335 | License | Ref. |
|---|---|---|---|---|---|---|---|---|
| FreeRTOS | Yes | Yes | - | - | - | - | MIT | [46] |
| µC/OS-III | Yes | Yes | Yes | Yes | - | - | Proprietary | [39] |
| eCos | Yes | - | - | Yes | - | - | GNU GPL | [47] |
| embOS | Yes | Yes | Yes | Yes | - | - | Proprietary | [40] |
| Nucleus RTOS | Yes | Yes | Yes | Yes | - | - | Proprietary | [48] |
| ThreadX | - | - | - | - | Yes | Yes | Proprietary | [36] |

Besides that, the huge selection of portable microcontroller is also one of the major considerations when it comes to RTOS selection. Table 7 lists ported microcontrollers with µC/OS-III and FreeRTOS RTOS solution. The more variety of microcontrollers ported to a particular embedded OS solution, the more choices it is when it comes to changing the microcontroller. The effort to port the RTOS to a new microcontroller can be saved. For example, suppose a product uses Infineon XMC1000 ARM Cortex-based microcontroller with FreeRTOS as the software platform. When it comes to changing the microcontroller in the future maybe due to cost effectiveness, there is a massive selection of FreeRTOS ported microcontroller to choose from as shown in Table 7.

As a summary, the dependency among application complexity and memory footprint is shown in Figure 12. It is divided into four quadrants. Quadrant 1 represents the scenario when the RTOS is being deployed in low complexity application. This will lead to higher demand on memory footprint and reduce cost-effectiveness. The higher the memory footprint is, the less the cost-effectiveness will be. For example, the price for STM32F103T4 microcontroller (16 kBytes ROM, 6 kBytes of RAM) is USD 1.7325 [49], while the price for the same family microcontroller STM32F103TB (128 kBytes ROM, 20 kBytes of RAM) is USD 2.3581 [50]. In contrast, development time could be reduced if the selected RTOS carry the needed software solutions.

Table 7. Ported microcontrollers with uC/OSIII and FreeRTOS RTOS solution

| MCU Maker | µC/OS-III Ported Microcontroller [51] | FreeRTOS Ported Microcontroller [52] |
|---|---|---|
| Altera | Nios II, SoC FPGA (Cortex-A) | Nios II, Cyclone V SoC (ARM Cortex-A9) |
| Analog Devices | Blackfin, ADSP-CM4xx (Cortex-M) | - |
| ARM | ARM7, ARM9, ARM11, Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15, Cortex-A17, Cortex-A53, Cortex-A57, Cortex-R4, Cortex-R5, Cortex-R7, Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4(F), Cortex-M7 | - |
| ATMEL | AVR, AVR32, SAM3, SAM4, SAM7, SAM9, SAMA5 (ARM Cortex-based) | AVR, AVR32, ARM Cortex-based (ATSAMD20, SAMV7, SAME7, AT91SAM4, AT91SAM3, ATSAMA5, AT91SAM7S, AT91SAM7X, AT91SAM9) |
| Cypress | PSoC 4, PSoC 5 (Cortex-M) | PSoC 5 (Cortex-M) |
| EnSilica | eSi-RISC | |
| Imagination/MIPS | M14K | 24 K, 34 K,74 K,1004 K,1074 K, M4 K, M14 K, microAptiv, interAptiv, proAptiv, M5100, M5150, M6200, M6250, P5600 |
| Infineon | XMC4000 (Cortex-M) | TriCore, ARM Cortex-based (XMC1000, XMC4000) |
| Microchip | PIC24, PIC32 | PIC18, PIC24, PIC32, MEC14xx, CEC13xx, CEC17xx, MEC17xx, MEC51xx (ARM Cortex-M4F) |
| Microsemi | SmartFusion2 (Cortex-M) | SmartFusion A2F, SmartFusion2 M2S |
| Microsoft | Win32 | |
| NXP | ColdFire, HCS12, i.MX, Kinetis (Cortex-M), LPC (Cortex-M), LPC (ARM7 / ARM9), MPC5xxx, MPC8xxx, VFxxx (Cortex-A & Cortex-M) | ColdFire, HCS12, i.MX, Kinetis (Cortex-M), LPC (Cortex-M), LPC (ARM7) |
| Renesas | H8S, 78K0R, R32C, RL78, RX100, RX200, RX600, RX700, RZ/A (Cortex-A), RZ/T1 (Cortex-R & Cortex-M), R-IN32 (Cortex-M), SuperH-2A, V850E/2/S | H8S, 78K0R, V850ES, RL78, RX100, RX200, RX600, RX700, RZ/A (Cortex-A), RZ/T (Cortex-R & Cortex-M), SuperH-2A, |
| Silicon Labs | Gecko (Cortex-M) | Gecko (Cortex-M), EFM32G890F128, Cygnal 8051 |
| ST Microelectronics | STM32F (Cortex-M), STM32L (Cortex-M), STR9 | STM32F (Cortex-M), STR7, STR9 |
| Texas Instruments | C28x, MSP430 (Cortex-M), MSP432 (Cortex-M), Hercules RM (Cortex-R), Hercules TMS570 (Cortex-R) | SimpleLink IoT, MSP430 (Cortex-M), MSP432 (Cortex-M), Stellaris (Cortex-M), Hercules Safety |
| Xilinx | MicroBlaze, Zynq-7000 (Cortex-A), Zynq Ultrascale+ MPSoC (Cortex-A & Cortex-R) | MicroBlaze , Zynq, Zynq UltraScale MPSoC, PowerPC 405, PowerPC 440 |
| Cadence Tensilica | - | Xtensa Processors |
| Cortus | - | Cortus APS3 |
| Spansion | | FM3 ARM Cortex-M3, MB91460, MB96340 |

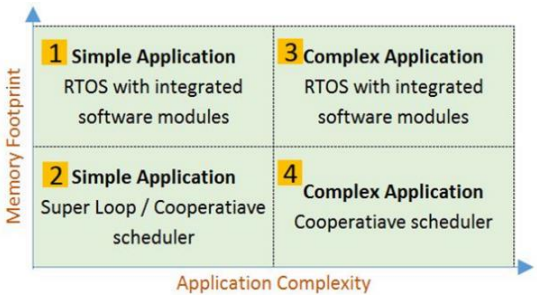



Figure 12. Application complexity versus memory footprint

Quadrant 2 represent the scenario where the super loop or cooperative scheduler is deployed in low complexity application. In principle, the complexity of application in this quadrant is lower than quadrant 1. The microcontroller with lower complexity, such as an 8-bit or 16-bit microcontroller [50], could be used in this scenario as the memory resources that the schedulers occupied very small and could be ignored. In this perspective, quadrant 2 will have the highest cost-effectiveness among all quadrants. For example, the price for STM8S003K3 microcontroller (8 kBytes ROM, 1 kBytes of RAM) is USD 0.4242 [53]. Quadrant 3 represents the scenario where deterministic behavior is required and development efforts could be reduced with RTOS deployment in the high complexity application. The main advantage is the availability of integrated software solutions such as FAT, GUI, and TCP/IP network stacks. For the complex application with no third-party software solutions available, quadrant 4 will fit in. Thus, the cooperative scheduler could handle the complexity with the tasks being grouped and controlled in time slots.

## 4. CONCLUSION

Each embedded OS approach has their strongpoints and weaknesses. Super loop and cooperative scheduler are small in memory footprint but less deterministic, and in contrast, RTOS is naturally deterministic but require more memory. Selecting the right embedded OS for an application will depend upon a lot of factors. At the norm, the more complex the application is, the higher the complexity of embedded OS. This might not be always true when external elements such as cost and quantity are factored in. In this case, super loop and cooperative scheduler with a naturally small memory footprint will be preferred as they could fit into smaller microcontrollers, which lead to higher cost-effectiveness. Another factor will be the product's requirements to fulfill the compliance of regulated standards such as OSEK/VDK, MISRA C, ISO 26262 in the automotive industry, and IEC 62304 in the medical industry. The certified embedded OS will help save cost and time needed to validate the OS part in software. Variety of supported features such as network protocols, GUI, USB, file system for IoT and the consumer electronics industry is also an important factor. Choosing the right embedded OS helps to simplify the integration of complex software modules.

## ACKNOWLEDGEMENTS

This work is supported by the Universiti Sains Malaysia RUI Grant (PELECT/8014049).

## REFERENCES

[1] D. Chalagulla, J. Jayateertha, T. Giri and V. Sailaja, "Gesture Controlled Bomb Diffusing Mobile Robot," *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*, 2018, pp. 1-5, doi: 10.1109/ICCONS.2018.8662838.

[2] M. Petrvalsky, M. Drutarovsky and M. Varchola, "Differential power analysis attack on ARM based AES implementation without explicit synchronization," *2014 24th International Conference Radioelektronika*, 2014, pp. 1-4, doi: 10.1109/Radioelek.2014.6828434.

[3] S. Fischmeister and P. Lam, "Time-Aware Instrumentation of Embedded Software," *in IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 652-663, Nov. 2010, doi: 10.1109/TII.2010.2068304.

[4] K. W. Batcher and R. A. Walker, "Interrupt Triggered Software Prefetching for Embedded CPU Instruction Cache," *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, 2006, pp. 91-102, doi: 10.1109/RTAS.2006.24.

[5] Nahas, M., "Implementation of highly-predictable time-triggered cooperative scheduler using simple super loop architecture," *International Journal of Electrical & Computer Sciences*, vol. 11, no. 4, pp. 33-38, 2011.

[6] M. J. Pont, S. Kurian, H. Wang, & T. Phatrapornnant, "Selecting an appropriate scheduler for use with time-triggered embedded systems," *Embedded Systems Laboratory, University of Leicester*, pp. 1-24, 2007.

[7] S. Kurian &, M. J. Pont, "The maintenance and evolution of resource-constrained embedded systems created using design patterns," *Journal of Systems and Software*, vol. 80, no. 1, pp. 32-41, 2017.

[8] P. Hambarde, R. Varma, & S. Jha, "The survey of real time operating system: RTOS," *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, 2014, pp. 34-39, doi: 10.1109/ICESC.2014.15

[9] Luis Fernando Friedrich, & Mario A. R. Dantas, "A Review of Operating System Infrastructure for Real-Time Embedded Software," *Journal of Communication and Computer*, vol. 12, no. 6, pp. 273-285, 2015, doi: 10.17265/1548-7709/2015.06.001.

[10] M. A. Mohammed, M. AbdulMajid, B. A. Mustafa and R. F. Ghani, "Queueing theory study of round robin versus priority dynamic quantum time round robin scheduling algorithms," *2015 4th International Conference on Software Engineering and Computer Systems (ICSECS)*, 2015, pp. 189-194, doi: 10.1109/ICSECS.2015.7333108.

[11] X. Guan, Q. Xing and L. Feng, "Implementation of embedded system platform based on μC/OS-II and S3C44B0X microprocessor," *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, 2011, pp. 2205-2208, doi: 10.1109/MEC.2011.6025929.

[12] Clark, G. (2014). The industrial revolution. In Handbook of economic growth, vol. 2, pp. 217-262. Elsevier.

[13] Ken Hu, "Rethinking The Internet Of Things," VOIZ ASIA, [Online]. Available: https://voiz.asia/en/33760. [Accessed: 12-August-2019].

[14] A. Omer, M. K. Ishak, M. K. L. Bhatti, "Adaptive clear channel assessment (A-CCA): Energy efficient method to improve wireless sensor networks (WSNs) operations," *AEU-International Journal of Electronics and Communications*, vol. 131, 2021, Art. no. 153603, doi: 10.1016/j.aeue.2020.153603.

[15] O. Hahm, E. Baccelli, H. Petersen and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720-734, 2016, doi: 10.1109/JIOT.2015.2505901.

[16] F. Javed, M. K. Afzal, M. Sharif and B. Kim, "Internet of Things (IoTs) Operating Systems Support, Networking Technologies, Applications, and Challenges: A Comparative Review," *IEEE Communications Surveys Tutorials*, vol. 20, no. 3, pp. 2062-2100, 2018, doi: 10.1109/COMST.2018.2817685.

[17] C. Perera, C. H. Liu, S. Jayawardena and M. Chen, "A Survey on Internet of Things From Industrial Market Perspective," *IEEE Access*, vol. 2, pp. 1660-1679, 2014, doi: 10.1109/ACCESS.2015.2389854.

[18] S. M. R. Islam, D. Kwak, M. H. Kabir, M. Hossain and K. Kwak, "The Internet of Things for Health Care: A Comprehensive Survey," *IEEE Access*, vol. 3, pp. 678-708, 2015, doi: 10.1109/ACCESS.2015.2437951.
[19] C. Sabri, L. Kriaa and S. L. Azzouz, "Comparison of IoT Constrained Devices Operating Systems: A Survey," *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, Hammamet, 2017, pp. 369-375, doi: 10.1109/AICCSA.2017.187.
[20] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. IEEE communications surveys & tutorials, 17(4), 2347-2376.
[21] O. Hahm, E. Baccelli, H. Petersen and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *in IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720-734, Oct. 2016, doi: 10.1109/JIOT.2015.2505901.
[22] F. Javed, M. K. Afzal, M. Sharif and B. Kim, "Internet of Things (IoT) Operating Systems Support, Networking Technologies, Applications, and Challenges: A Comparative Review," *in IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2062-2100, thirdquarter 2018, doi: 10.1109/COMST.2018.2817685.
[23] M. Silva, D. Cerdeira, S. Pinto and T. Gomes, "Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking," *in IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10375-10383, Dec. 2019, doi: 10.1109/JIOT.2019.2939008.
[24] Zikria, Y.B.; Kim, S.W.; Hahm, O.; Afzal, M.K.; Aalsalem, M.Y., "Internet of Things (IoT) Operating Systems Management: Opportunities, Challenges, and Solution," *Sensors 2019*, vol. 19, pp. 1793, doi: https://doi.org/10.3390/s19081793
[25] C. Adjih et al., "FIT IoT-LAB: A large scale open experimental IoT testbed," *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015, pp. 459-464, doi: 10.1109/WF-IoT.2015.7389098.
[26] D. D. Ward, "MISRA Standards for Automotive Software," *2006 2nd IEE Conference on Automotive Electronics*, London, 2006, pp. 5-18.
[27] F. Fabbrini, M. Fusani, G. Lami and E. Sivera, "Software Engineering in the European Automotive Industry: Achievements and Challenges," *2008 32nd Annual IEEE International Computer Software and Applications Conference*, 2008, pp. 1039-1044, doi: 10.1109/COMPSAC.2008.140.
[28] D. John, "OSEK/VDX history and structure," *IEE Seminar on OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523)*, London, UK, 1998, pp. 2/1-2/14, doi: 10.1049/ic:19981073.
[29] A. Zahir and P. Palmieri, "OSEK/VDX-operating systems for automotive applications," *IEE Seminar on OSEK/VDX Open Systems in Automotive Networks (Ref.No. 1998/523)*, London, UK, 1998, pp. 4/1-418, doi: 10.1049/ic:19981075,
[30] C. Tsai, K. Tsai and M. Hsu, "An implementation of the enhanced-CAN BUS network connection in CAR real-time embedded software system," *2012 12th International Conference on Control, Automation and Systems*, 2012, pp. 277-283.
[31] P. O. Ridolfi, "Extension of the FreeOSEK RTOS for Asymmetric Multiprocessor Systems," *2016 IEEE Biennial Congress of Argentina (ARGENCON)*, 2016, pp. 1-6, doi: 10.1109/ARGENCON.2016.7585243.
[32] J. Axelsson, A. Kobetski, Z. Ni, S. Zhang and E. Johansson, "MOPED: A Mobile Open Platform for Experimental Design of Cyber-Physical Systems," *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2014, pp. 423-430, doi: 10.1109/SEAA.2014.38.
[33] F. C. Braescu, L. Ferariu and C. Lazar, "OSEK-based multiple controllers with schedule feasibility self-testing," *SPEEDAM 2010*, 2010, pp. 1237-1242, doi: 10.1109/SPEEDAM.2010.5542060.
[34] F. C. Braescu, L. Ferariu and C. Lazar, "OSEK-based multiple controllers with schedule feasibility self-testing," *SPEEDAM 2010*, 2010, pp. 1237-1242, doi: 10.1109/SPEEDAM.2010.5542060.
[35] C. Dong and H. Zeng, "Minimizing stack memory for hard real-time applications on multicore platforms," *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014, pp. 1-6, doi: 10.7873/DATE.2014.041.
[36] "RTX5," [Online]. Availabe: http://www2.keil.com/mdk5/cmsis/ rtx [Accessed: 28-Jan-2020].
[37] "SafeRTOS," [Online]. Availabe: https://www.highintegritysyst ems.com/safertos [Accessed: 18-Jan-2020].
[38] Kabra, A., Karmakar, G., & Joseph, J., "ST to MISRA-C translator and proposed changes in IEC61131-3 standard," International Journal of Information and Electronics Engineering, vol. 2, no. 4, p. 575, 2012.
[39] P. Jordan, "Standard IEC 62304 - Medical Device Software - Software Lifecycle Processes," 2006 IET Seminar on Software for Medical devices, London, 2006, pp. 41-47, doi: 10.1049/ic:20060141.
[40] Nelson, R., "Design through test technologies boost real-world intelligence," *EE-Evaluation Engineering*, vol. 57, no. 4, pp. 12-17, 2018.
[41] Hobbs, C., "Embedded software development for safety-critical systems," CRC Press, 2019.
[42] J. A. Coy, J. H. Pfeiffer, Y. S. Krieger, J. -H. Mehrkens, K. Bötzel and T. C. Lueth, "Mechatronic device for the optimization of the DBS-electrode placement," *2016 6th IEEE International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, 2016, pp. 214-219, doi: 10.1109/BIOROB.2016.7523625.
[43] Samek, M.," Practical UML statecharts in C/C++: event-driven programming for embedded systems," CRC Press, 2018.
[44] R. Le Moigne, O. Pasquier and J. -. Calvez, "A generic RTOS model for real-time systems simulation with systemC," *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, 2004, pp. 82-87 vol.3, doi: 10.1109/DATE.2004.1269211.
[45] D. Roman, "Introduction to IEC 60335-Household and similar electrical appliances-Safety," *2015 IEEE Symposium on Product Compliance Engineering (ISPCE)*, Chicago, IL, 2015, pp. 1-6, doi: 10.1109/ISPCE.2015.7138702.

[46] A. Musaddiq, Y. B. Zikria, O. Hahm, H. Yu, A. K. Bashir and S. W. Kim, "A Survey on Resource Management in IoT Operating Systems," *in IEEE Access*, vol. 6, pp. 8459-8482, 2018, doi: 10.1109/ACCESS.2018.2808324.
[47] Massa, A. J.," Embedded software development with eCos," *Prentice Hall Professional*, 2002.
[48] Pothuganti, K., Haile, A., & Pothuganti, S." A Comparative Study of Real Time Operating Systems for Embedded Systems," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 4, no. 6, 2016.
[49] "STM32F103T4," [Online]. Available: https://www.st.com/content/st com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32-mainstream-mcus/stm32f1-series/stm32f103/stm32f103t4.html [Accessed: 12-Feb-2020].
[50] "STM32F103TB," [Online]. Available: https://www.st.com/content/st com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32-mainstream-mcus/stm32f1-series/stm32f103/stm32f103tb.html [Accessed: 12-Feb-2020].
[51] uC/OS RTOS & Stacks, "Real-Time Kernels: µC/OS-II and µC/OS-III," [Online]. Available: https://www.micrium.com/ rtos/kernels. [Accessed: 12-Feb-2020].
[52] FreeRTOS, "FreeRTOS Ports," [Online]. Available: https://www.freertos.org/ a00090.html. [Accessed: 12-Feb-2020].
[53] "STM8S003K3," [Online]. Available: https://www.st.com/content/st com/en/products/microcontrollers/stm8-8-bit-mcus/stm8s-series/stm8s-value-line/stm8s003k3.html [Accessed: 12-Feb-2020].

## BIOGRAPHIES OF AUTHORS

**Yew Ho Hee** received the B.S degree in Electrical and Electronics engineering from Multimedia University, Melaka, Malaysia, in 2004. He is currently pursuing the Master of Business Administration (MBA) at Universiti Sains Malaysia, Penang, Malaysia. Since 2004, he had been working as firmware engineer in consumer electronics, industrial tools and automotive product. Currently, he is a feature responsible who lead a team of six members to realize the firmware update over the air (FOTA) feature for instrument cluster. His research interests including Internet of Things, network topology design and applications, scalability in software abstraction, accelerometer applications, over the air firmware updatability (OTA), User Experience (UX), and Real Time Operating System.

**Mohamad Khairi Ishak** received the B.Eng degree in Electrical and Electronics Engineering from the International Islamic University Malaysia (IIUM), Malaysia, the MSc. in Embedded System, from the University of Essex, United Kingdom and PhD from the University of Bristol, United Kingdom. He is a member of IEEE and a registered graduate engineer with the Board of Engineers Malaysia (BEM). Currently, he is a Senior Lecturer in Mechatronics Engineering at School of Electrical and Electronic Engineering, Universiti Sains Malaysia (USM). His research interests are Embedded System, Real-Time Control Communications and Internet of Things (IoT). Emphasis is given towards the development of theoretical and practical methods which can be practically validated. Recently, significant research effort has been directed towards important industrial issues of embedded networked control systems and IoT.

**Mohd Shahrimie Mohd Asaari** completed his PhD in Science (Physics) from Universiteit Antwerpen, Belgium in 2019. He obtained Master of Science (Electrical and Electronics) from Universiti Sains Malaysia in 2012 and Bachelor of Engineering Hons. (Electrical) from Universiti Teknologi Mara in 2009. In his PhD research, he concentrates in plant-related study, where the project aims for early stress detection in plants using the analysis of close-range hyperspectral imaging. Currently he is a senior lecturer at School of Electrical and Electronics Engineering, Universiti Sains Malaysia. His research interest includes Image processing, Computer vision, Machine learning and Close-range hyperspectral imaging.

**Mohamad Tarmizi Abu Seman** has experience 17 years as a lecturer/senior lecturer at School of Electrical and Electronic, Universiti Sains Malaysia (USM), Engineering Cmpus in Mechanical and Mechatronic Engineering. He was graduated from UiTM (Degree), USM (Master and PhD) in an area of Mechanical Engineering with focusing on Computational Fluids Dynamic (CFD), Control, NDT, M&E, IOT, and Embedded System. He is also a Chairman of Malaysian of Technical Doctorate Association (MTDA). Recently, he has supervised ongoing 2 PhD Student and 1 master student. He also has obtained a cumulative grant from a various sector with a total amount of RM 220,000 from 2018 to 2021. He was awarded as a Professional Engineer (Ir.) in a field of Mechanical Engineering (P119216) by the Board of Engineer Malaysia (BEM) and IEM members (61958) in Mechanical from The Institution of Engineer Malaysia (IEM).