VOL. 11, NO. 1, JANUARY 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences

www.arpnjournals.com

AN ANALYTICAL STUDY ON RTOS AS THE ENGINE BEHIND INTERNET OF THINGS: CHOICES AND TRADE-OFFS

J. Umadevi, V. Kavitha and A. Srikrishnan
Department of Electronics and Communication Engineering, Dr. M.G.R Educational and Research Institute University, Chennai, India
E-Mail: umadevijp@yahoo.com

ABSTRACT

World is becoming increasingly data-driven and complexly connected due to the emergence of ever changing technological scenario such as cloud computing, Big Data, Internet of Things (IoT), etc. Internet of Things can be viewed as inter-connection of people, applications and smart devices, which transmit and receive data over already existing network. These operations are done in real-time within a deadline. In essence, IoT objects are small, networked embedded devices. Current ideology is that Android and Linux are the most suitable Operating Systems for IoT implementation. This paper introduces a viewpoint that a sophisticated, robust and efficient real-time operating system (RTOS) which is TCP/IP ready would be more suitable for IoT. The critical features of µC/OS III, Micrium's commercial RTOS and Embedded Linux are studied and a comparative analysis is done based on the study findings.

Keywords: IoT, RTOS, µC/OS III, Linux, RTAI, Real-time, embedded devices.

1. INTRODUCTION

1.1. Internet of things

The Internet of Things (IoT) is a paradigm shift and an ontological change. IoT is a communication between physical objects/ things that have built-in computing devices. These embedded objects have in-built processors, software and network connectivity for storing and sharing of data. Internet of Things enables embedded objects to be connected and operated remotely using already existing internet network. This creates tremendous opportunities for direct integration between the physical world and computing systems, resulting in increased performance and efficiency. It can be envisioned as a scenario in which human, animals or objects are embedded with unique identifiers and the ability to exchange information over a network without requiring human-to-human or human-to-machine interaction. The unique identifiers can be provided through RFID and SAAS. IoT has evolved from the convergence of wireless technologies, micro-electromechanical systems (MEMS) and the Internet. There are 4 major components in an IoT system:

- The **Embedded object-**Embedded devices make upthe key component/node/object of Internet of Things connectivity.
- The **local network -** This includes a gateway which translates communication protocols to internet protocols
- The **Internet-** Currently existing network connectivity
- **Back-end services -** Enterprise data services, PCs or Mobile phones

The illustration given in Figure-1 is an apt example of an IoT implementation. It shows a Wireless Sensor Network (WSN) installed in a factory setup. WSN nodes are connected to internet via a gateway. They go through the gateway for LAN connectivity as well. Data from WSN nodes are updated in the server using Cloud.

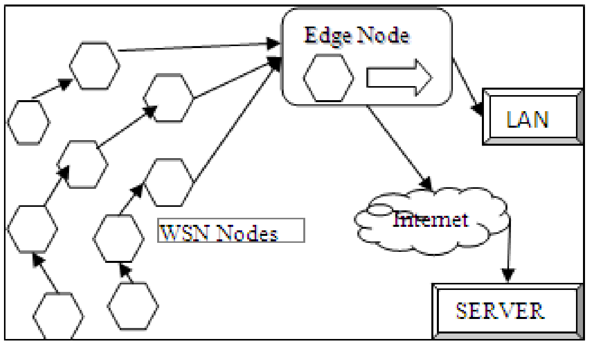

Figure-1. WSN of a factory.

The individual WSN nodes and the Edge Node/ Gateway can be considered as the embedded object/ device in an IoT system. These embedded objects which form the building blocks of IoT are computing devices with built-in processors, software and network connectivity. The software is the driving force of the IoT object, which has to provide real-time processing. A real time operating system (RTOS) like µC/OS III, VxWorks, etc has numerous special features and characteristics that can make them an appropriate choice for IoT implementation.

1.2. Which is the most preferrable o/s for IoT implementation?

Any common object ranging from Street lights, Washing machines, Automobiles to Nuclear reactors are empowered with electronic sensors and communication devices. These are the "things" that make up the IoT. To be suitable for IoT purpose, an operating system would need to be:

VOL. 11, NO. 1, JANUARY 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2016 Asian Research Publishing Network (ARPN). All rights reserved.
www.arpnjournals.com

- Small in memory size, because these are mostly tiny devices with restricted resources.
- Capable of detecting and working with a wide variety of different sensors and actuators.
- TCP/IP ready.

It is industry belief that *Android* has great potential to provide the unifying power needed for IoT than any other single operating system. *Linux* seems to be the other popular choice. The Internet of Things is too big to be contained by one or two operating systems. Interoperability is the mandate for IoT. Actually we may very likely find that building the right APIs and making sure those APIs have all the right gateways and features is more important than the operating systems itself.

1.3. RTOS as the driving force of an IoT object

It is common knowledge that IoT objects are small networked systems, i.e. low end microcontroller based systems with even 8 or 16 bit processors and small sensors whose *memory* unit is tiny. The major disadvantage of *Linux*being the driver behind IoT is its memory footprint, which is huge to fit inside these tiny IoT objects. Another critical factor needed for IoT objects is *Real-Time processing* of data. The following list shows a wide range of application areas which require the stringent principles and concepts of Hard Real-time system.

- **Aerospace:** Jet Engine controls, Weapons systems
- **Automobile industry:** GPS, Engine Control, Antilock Braking systems
- **Process control:** Factory Automation, Chemical Plants, Food Processing

These small real-time microcontroller based IoT systems uses 8, 16 or 32-bit processors. Most of these systems use a Real Time Operating system (RTOS) as the driving software. RTOS can be custom-built for a specific application or can be a generic one. A few of the commercially available RTOS are RTAI, Free RTOS, VxWorks, µC/OS III etc. A Real Time Operating System (RTOS) generally contains a real-time kernel and other higher-level services such as file management, protocol stacks, Graphical User Interface (GUI), and other components. Most additional services revolve around I/O devices. As the name suggests, there is a deadline associated with tasks and an RTOS adheres to this deadline as missing a deadline can cause reactions ranging from undesired to catastrophic. Basic features of an RTOS are as follows:

- **Context switching:** The time taken for saving the context of current task and switching over to another task should be extremely negligible.
- **Interrupt latency:** The switch-over time between executing the final instruction of an interrupted task and the first instruction of interrupt handler should be less and deterministic..
- **Interrupt dispatch latency:** The switch-over time between the final instruction of the interrupt handler and the next task should be predictable and limited.
- **Inter-process mechanisms:** Should have Reliable and time bound system in place for processes to interact with each other.
- **Task preemption:** The ability to stop the currently executing task and switch to a higher priority task should be available.
- **Kernel preemption:** kernel should have the ability to preempt the current process for a higher priority process.
- **Network support:** Readymade APIs should be available for internet connectivity and networking support.

The RTOS, which can cater to the needs of an IoTdevice, must contain the following characteristics:

- **Scalability:** To accommodate a wide range of different classes of devices.
- **Modularity:** The option to choose only the essential components in order to meet tight RAM requirements.
- **Connectivity:** For movement of data in and out of the device via Wi-Fi, Ethernet, USB, or Bluetooth.
- **Security:** The pervasive connectivity in IoT results in substantial exposure to threats.
- **Safety:** Safety is paramount in many embedded operating systems because they control machines that can endanger life.
- **Reliability:** Foremost for devices used in safety-critical applications.

The illustration given in Figure-2 depicts the software architecture of an IoT object.

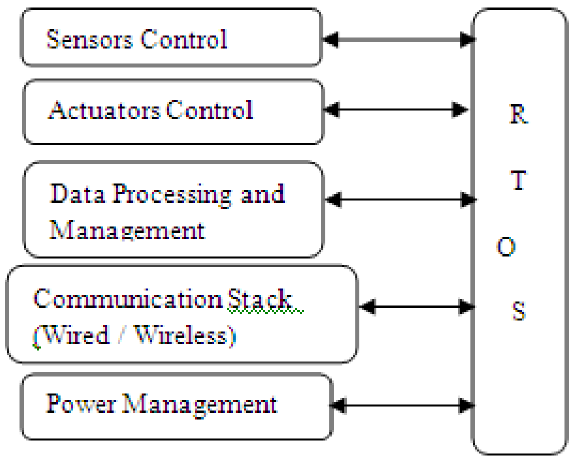
Sensors Control
Actuators Control
Data Processing and Management
Communication Stack (Wired / Wireless)
Power Management
R T O S

Figure-2. Software architecture of an IoT object.

1.4. Advantages of using commercial RTOS for IoT implementation

IoT devices require a solid software infrastructure, including a real-time kernel along with additional services like TCP/IP, Wi-Fi, Bluetooth stacks, USB connectivity as well as cloud services and the ability to put it all together. Developers will hugely benefit if GUI development, communications middleware and wireless connectivity can also be integrated into the design process. An RTOS that can integrate wireless (Ethernet, WiFi and Bluetooth) protocols and USB connectivity into their OS will have a fair advantage over other vendors. Modeling can enable rapid prototyping, code reuse as well as integration of legacy software. Commercial off-the-shelf solutions offer quicker time-to-market, lesser risk and cost, and long term maintenance and support. A list of Proprietary RTOSand free RTOS are given in Table-1.

Table-1. List of commonly available RTOS.

	List of RTOS
Free RTOS	FreeRTOS, ucLinux, eCos, coscox
Proprietary RTOS	SafeRTOS, VxWorks, µC/OS II & III, ThreadX, QNX, embOS

2. µC/OS III AS THE CORE OF IOT

This section analyses the major features of µC/OS III, a commercially available RTOS from Micrium Inc.as an engine behind IoT object. µC/OS-III is a third generation real-time kernel which offers the services expected from a modern age RTOS such as resource management, synchronization, inter-task communication, scheduling and much more. The architecture of µC/OS-III is shown in Figure-3.

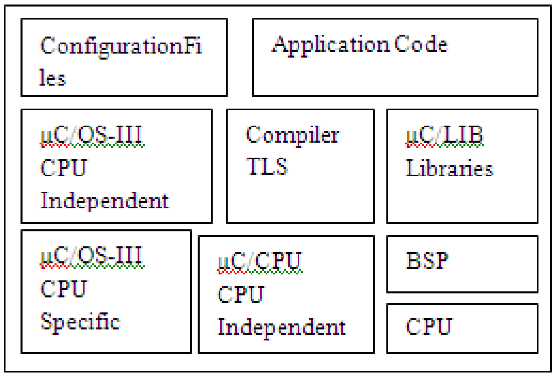

Figure-3. Architecture of µC/OS-III.

Notable features of µC/OS-III are explained in the following section:

- Running on the largest number of processor architectures, their performance is measured at run-time. Also, can directly signal or send messages to tasks
- Can handle unlimited number of tasks with the processor's memory capacity being only constraint.
- Supported priority level is also unlimited, generally between 8 and 256 different priority levels.
- Facility for mutexes, event flags, semaphores, message queues, timers and memory partitions.
- Multiple tasks can run at the same priority level at the same time. For equal priority tasks that are ready-to-run, µC/OS-III runs each for a user-specified time period. The facility for each task to define its own time quanta and give up its time slice if it does not require the full time quanta is also available.
- Stack growth of tasks can be monitored.
- Extensive range checking facility which can be disabled during compile time is available.
- It can check for many features such as if NULL pointers are passed in API calls, task level services aren't called from ISRs, arguments are within allowable range and valid options are specified. Each API function provides an error code with respect to the outcome of the function call.
- µC/OS-III's memory footprint can be scaled to accommodate only the specific features needed for an application, typically 6–24 KB of code space residing in memory.

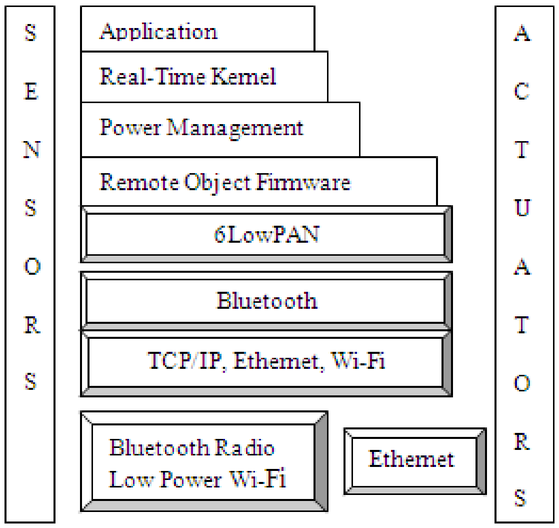

Figure-4. Software stack of WSN.

The illustration given in Figure-4 shows the software stack for an industrial wireless sensor node (WSN) using µC/OS-III as the software driving this IoT device. Wireless communication over short distances is achieved using Bluetooth, low-power Wi-Fi or Ethernet. Such a device typically uses a Cortex-M3/M4 or a Cortex-A processor.

A few functionalities of µC/OS-III which are highly essential for IoT implementation such as WSN are discussed in detail in the following section:

Network protocols: Accessibility to Wi-Fi is a necessity for many applications. However, Wi-Fi needs good amount of power. In the case of remotely located sensors, where it is difficult to get supply from the grid, battery operated devices are the mandate. Hence, IPv6 is key for IoT devices. IPv6's addressing scheme provides more addresses than any other option - some have calculated that it could be as high as 10^{30} addresses. With IPv6, it is much easier for an IoT device to get a global IP address, which facilitates efficient peer-to-peer communication. µC/TCP-IP is a compact, reliable, high-performance TCP/IP protocol stack from Micrium that can be used for IoT implementation. It is optimized for embedded systems, and features dual IPv4 and IPv6 support. µC/OS-III real-time kernel takes care of task scheduling and mutual exclusion and µC/TCP-IP module handles IPv6 support. The specifications of µC/TCP-IP such as supported processors and protocols, Interface type, Transport layer, etc are given in detail in Table-2.

Table-2. Specifications of µC/TCP-IP.

Supported processors	**Most of the commercial processors**
Real-time Kernel	Supports Micrium'sµC/OS-III or µC/OS-II
Transport Layer	TCP and/or UDP
Protocols supported	Multicast transmission and reception (IGMPv2), IPv6 Multicast (MLD), ICMPv6, ARP, Neighbor Discovery Protocol (NDP)
Interface Type	Ethernet (802.3 and Ethernet), Wifi, Loopback
Socket API	Two sets of socket APIs: one proprietary and another standard BSD socket.

The uC/USB Device and Host modules is available with robust implementations of many popular classes, including Audio, Mass Storage, Human Interface Device, Communications Device, and Personal Healthcare Device.

The real-time kernel of µC/OS-III handles the major functions such as Task scheduling, Time management, Inter task communication, etc. The functions are discussed in detail in the following section:

Task scheduling: µC/OS-III can manage any number of tasks at a given point of time. The kernel reserves 4 highest priority and 4 lowest priority tasks for its internal use. When the priority value is low, actually, the priority of the task is higher. The task priority number also doubles as the task identifier. The process cycle is shown in Figure-5.

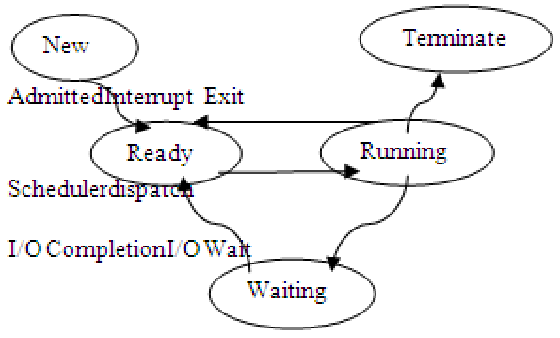

Figure-5. Process cycle.

After task creation, the task is loaded onto a stack where the data is stored. A stack usually has contiguous memory locations. The kernel determines how much stack space a task actually uses. Deleting a task returns the task to its dormant state. However, the code for the task will be retained. The calling task can delete the code. If another task tries to delete the current task, the resources are not freed and thus are lost. Hence, the task should delete oneself after it uses its resources.

Memory management: Each memory partition is made of several fixed sizedblocks. A task must create a memory partition before it can be used and obtains memory blocks from the partition. Allocation and de-allocation of these fixed-sized memory blocks is done in constant time and is deterministic. Multiple memory partitions can exist, which enables a task to obtain memory blocks of varying sizes. Each memory block should be returned to the respective partition it came from.

Time management: µC/OS-III increments a 32-bit counter for every clock cycle. Starts at zero, the counter rolls over to 4, 294, 967, 295(2^32-1). The facility to delay a task can and then resume the delayed task is available.

Five services related to time management are:

- OSTimeDLY()
- OSTimeDLYHMSM()
- OSTimeDlyResume()
- OSTimeGet()
- OSTimeSet()

Inter-Task communication: µC/OS III handles Inter-task or inter process communication using one of the options such as Semaphores, Message mailbox, Message queues, Tasks and Interrupt service routines (ISR), which interact with each other through an ECB (event control block). Figure-6 shows the multi task waiting and signaling operation.

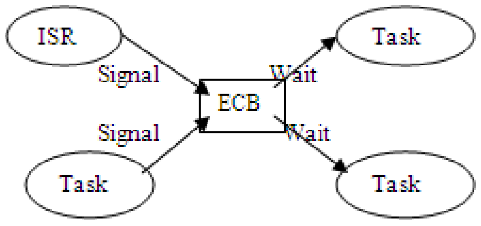

Figure-6. Inter-task communication.

Development environment: µC/GUI is aversatile development environment that comes along with µC/OS III. It supports almost all commercially available processors from 8-bit upwards.

- **Window manager:** Windows that are created using the GUI are managed by this facility. It handles all mouse and keyboard related events. Since µC/OS III provides touch screen facility, those are also managed by window manager.
- **Frame buffer:** The image of a window is drawn on this buffer and when the drawing is complete, it is copied to the touch screen. This helps in preventing screen flickering.
- **Anti-aliasing:** This polishes the fonts and othergraphical entities. It gives a pleasant affect to the eye instead of the rugged look.
- **Touch screen:** Touch Screen drivers, LCD drivers.
- **Widgets and dialogs:** A library to create the widgets (Check boxes, buttons, textboxes, etc) and dialog boxes is readily available. This reduces lot of effort to build everything using pixels and lines.
- **Font converter:** It converts a general font format (ttf, utf, etc) to the µC/GUI compatible fonts.
- **Bitmap converter:** It converts the 32-bit bitmap images to a compatible image that is used on µC/GUI.

Hard real-time operating system: µC/OS III is a hard RTOS. In critical applications, missing a deadline often results in catastrophe. µC/OS III is highly deterministic in nature, where the execution times of all functions and tasks are predictable. Hence, missing deadlines does not happen that easily in this environment.

3. RTAI

Linux is a time-sharing OSthat provides good average performance and highly sophisticated services. Linux was not built for real time support. To obtain a timing correctness behavior, it is very much needed to make a few modifications in the kernel sources, i.e. in interrupt handling techniques and scheduling policies. In this way, one can have a real time platform, with low latency and high predictability requirements, within full non real time Linux environment (access to TCP/IP, graphical display and windowing systems, file and data base systems, etc.).Real Time Application Interface (RTAI) is not a real time operating system, such as VXworks or QNX. It is based on the Linux kernel, providing the ability to make it fully pre-emptable. RTAI is also an open source like Linux. According to Kim and Ambike, RTAI has real time performance comparable to RTOS such as VxWorks and QNX [Kim *et al*. 2006], having sufficient determinism to replace them [Barbalace et al. 2008], however it does not offer any certification or guarantee to its users, as it is open source. RTAI provides deterministic response to interrupts, POSIX-compliant and native RTAI real-time tasks. RTAI supports various architectures, including IA-32 (with and without FPU and TSC), x86-
64, PowerPC, ARM (StrongARM and ARM7: clps711x-family, Cirrus Logic EP7xxx, CS89712, PXA25x), and MIPS. In RTAI, the operating system is divided into domains, making the real time domain as the one with highest priority. Whenever a real time task needs to be executed, a tiny scheduler, called nanokernel, schedules this task, freezing the whole remaining system. When there are no pending real times jobs to be done, the other parts of the system, such as GUI and user interface are executed [Barabanov 1997]. This solution is very practical, as most real time systems consists of a combination of tasks with soft and hard real time requirements [Labrosse 2002]. This enables the use of a single computer to control both real time critical functions and user interface, networking or others features that can be added at any time without changing the real time performance. RTAI mainly consists of two parts: an Adeos-based and a plethora of services. The Adeos-based patch to the Linux kernel introduces a hardware abstraction layer. Adeos is a kernel patch comprising an Interrupt Pipeline where different OS domains register interrupt handlers. RTAI versions 3.0 and above uses one which is slightly modified in the x86 architecture case, providing additional abstraction and much reduced dependencies on the "patched" OS. This way, RTAI can transparently take over interrupts while leaving the processing of all others to Linux kernel. It offersthe following services to the applications:

- Hardware management layer which deals with interrupts and event polling
- Task Scheduling classes that deals with process activation, time slice, priorities, etc
- Inter-application communication

The main disadvantage of Linux is its memory footprint. It requires minimum of 1MB of on board RAM and ROM size for it to operate. Hence, it cannot run on 8 or 16-bit MCUs and even many newer 32-bit MCUs do not have enough on board RAM for the Linux kernel. The ARM Cortex-M series is a good example.

VOL. 11, NO. 1, JANUARY 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2016 Asian Research Publishing Network (ARPN). All rights reserved.

4. COMPARATIVE STUDY

An elaborate analysis of the features of μC/OS III was covered in Section 2. The features of RTAIwere discussed in Section 3.This section consolidates the key features of μC/OS III and RTAI that are essential for Internet of Things implementation. Table-3 gives the comparison study on the key programming features forμC/OS III and RTAI.

Table-3. Comparative Study on programming features of μC/OS III and RTAI.

	μC/OS III	**RTAI**
Min RAM	1K B	1 MB
Min ROM	6K-24K B	1 MB
C Support	Yes	Yes
C++ Support	Yes	Yes
Multi-Threading	Yes	Yes
Response time	Close to Zero	Moderate

The graph illustrated in Figure-7 shows the Comparative minimum RAM/ROM requirements of μC/OS III and RTAI. Minimum memory requirements are measured in units of KB.

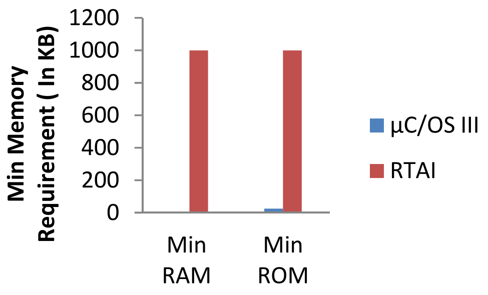

Figure-7. Comparison on minimum RAM/ROM requirements for μC/OS III and RTAI.

It can be inferred from Figure-7 that when compared to RTAI, the memory requirements of μC/OS III are negligible. The characteristics that are mandatory for an OS to operate as an efficient driver for IoT are Scalability, Modularity, Security, Safety, Network Support, Determinism, extensive feature set and memory footprint. The comparative study report based on the analysis of the above mentioned key features are given in Table-4.

Table-4. Comparative study on μC/OS III and Linux for IoT implementation.

	μC/OS III	**RTAI**
Scalability	Supports 8, 16 and 32-bit processors	No support for 8 and 16 bit processors
Modularity	Option to choose only the needed components to meet tight RAM requirements	Modularity is better with the advent of Linux 2.4.
Security	Has μC/SSL which supports Java,Perl,PhP and Python	Same security features as Linux
Network Support	APIs for Ipv6, Wi-Fi,Ethernet,USB, Bluetooth connectivity	Need to know shell programming to include IPv6 support
Extensive Features set	MicroEJ – a SDK for design and development	Same development and debugging tools as Linux
Safety	Encrption, firewall and other safety measures	Watchdog facility for safe development environment
Determinism	Execution times of all functions and services are deterministic. Hence highly predictable	Not designed for Real-time processing. So, lacks true real-time support
Memory Footprint	Occupies less memory	High memory footprint

Since RTAI is open-source, the features are attractive from a licensing perspective. Development tools are available along with distribution. However, if things go wrong, there is no vendor to blame. Post development maintenance and support of Linux cannot be compared to that of a commercial RTOS. Based on the study given in Table-3 and $, taking into consideration the complete package of features and post implementation support, μC/OS III emerges as a better choice for design, development and maintenance of IoT objects.

VOL. 11, NO. 1, JANUARY 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2016 Asian Research Publishing Network (ARPN). All rights reserved.

www.arpnjournals.com

5. CONCLUSIONS

This paper analyzed the basic characteristics that are required for embedded objects/things that make up the key component of IoT. IoT is an immensely complex and complicated system which must be equipped to handle heterogenous requirements of diverses hardware platforms and application scenarios, provide an adaptive IP network stack, and offer a standard developer-friendly API. Further, the OS suited for IOT must have reliable microkernel architecture, occupy less memory and have a highly adaptive network stack. It was established beyond doubt that the software needs of an IoT object can be satisfied by the RTOS, µC/OS III. Micrium'sµC/OS III, a commercial RTOS was taken up for study and its characteristic features in the context of IoT implementation were discussed. Further, the merits and demerits of RTAI/Linux were also discussed and the study findings were tabulated. Based on the study findings, µC/OS III, with its impeccable post development support and maintenance, emerges as the preferred choice of RTOS for IoT implementation. For future scope of work, design and simulation of an IoT embedded device with µC/OS III as the OS can be achieved.

REFERENCES

[1] Emmanuel Baccelli and Oliver Hahm INRIA Saclay, France, Mesut Gunes and Matthias Wahlisch, FreieUniversitat Berlin, Germany and Thomas C. Schmidt, HAW Hamburg, Germany. 2013. Operating Systems for the IoT - Goals, Challenges, and Solutions. Workshop Interdisciplinaire sur la S´ecurit´e Globale (WISG2013), Troyes, France.

[2] Emmanuel Baccelli, Oliver Hahm, Mesut G¨unes, Matthias W¨ahlisch, Thomas Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. The 32nd IEEE International Conference on Computer Communications (INFOCOM 2013), Turin, Italy.

[3] 2014. The RTOS as the engine powering the Internet of Things. By Bill Graham and Michael Weinstein.

[4] D. McCullough. 2004. uCLinux for Linux Programmers. In Linux Journal.

[5] RIOT OS - An Operating System for the IoT 2012. [Online]. Available: http://www.riot-os.org

[6] H. Will, K. Schleiser, and J. H. Schiller. A real-time kernel for wireless sensor networks employed in rescue scenarios. In: Proc. of the 34th IEEE Conference on Local Computer Networks (LCN)

[7] M. O. Farooq and T. Kunz. 2001. "Operating systems for wireless sensor networks: A survey," Sensors Journal.

[8] Ambike A., Kim, W.-J. and Ji, K.. 8-10 June 2005. Real-time operating environment for networked control systems. American Control Conference.

[9] Rafael V. Aroca, GlaucoCaurin. A Real Time Operating Systems (RTOS) Comparison.

[10] Cedeno, W. and Laplante P. A. 2007. An overview of real-time operating systems. Journal of the Association for Laboratory Automation.

[11] Barbalace A., Luchetta A., Manduchi G., Moro M., Soppelsa A. and Taliercio C. 2008. Performance comparison of vxworks, linux, rtai, and xenomai in a hard realtime application. Nuclear Science, IEEE Transactions on. 55(1):435-439.

[12] Gerd Kortuem and FahimKawsar Lancaster University, Daniel Fitton. University of Central Lancashire, Vasughi Sundramoorthy University of Salford. Smart objects as building blocks for the Internet of Things. Published by IEEE computer society.

[13] Srikanth, Narayanaraju Samunuri. 2013. RTOS Based Priority Dynamic Scheduling for Power Applications through DMA Peripherals. International Journal of Engineering Trends and Technology (IJETT). 4(8):

[14] http://doc.micrium.com/display/osiiidoc/uc-OS-III documentation home, µC/OS III the Real Time Kernel User's Manual, 2015[Online].

[15] http://micrium.com/iot/iot-rtos/The Internet of Things and the RTOS, 2015.[Online]http://www.rtai.org.[Online].