

Next-Generation OS Architectures for Low-Power Smart Hardware: A Comprehensive Review

Ashish Sharma¹, Nausheen Khilji^{2,*}

Abstract

The ubiquity of smart devices has dramatically transformed how humans engage with technology. Devices with high characteristics do use standard operating systems like Linux, but those with low characteristics cannot do so because of extreme re-source constraints. The survey also offers a range of other results and analysis applicable to existing low-end operating systems as well (along with architectural highlights and constraints as applicable). A field survey of technical institutes is then done where major research problems in the area are outlined. On this understanding, we present a general framework that can be used to automatically create custom, light-weight operating systems in application-specific, resource-constrained hardware. As smart technology continues to expand in various spheres, such as education, industry and home automation, tools that facilitate access through hardware abstraction and resource optimization are essential. The proposed solution fills this gap by offering users an almost automatic approach to selecting application-specific peripherals and creating custom OS images in an automatic approach. To be universally adopted and iteratively extended, such a framework must be an open-source tool, one that people can use, build upon and modify. As such, the application of this generic framework and adherence to the Guidelines of our architectural design will be beneficial towards ease of use that transcends the complex hardware constraints and requirement of more intelligent domain-specific smart solution.

Keywords: Smart device, smart OS framework, low- end devices, lightweight operating system, generic OS framework

INTRODUCTION

The most appropriate field of application of human intelligence together with artificial intelligence [1]. is smart environments through the incorporation of intelligence into everyday lives. This paradigm shift helps improve human experience by incorporating devices and sensors. Intelligent lighting [2]. irrigation solutions [3]. medical product innovations [4]. and industrial/home systems [5, 6] are

examples of intelligent systems. These so-called smart technologies are made possible by web services, which enable people to use the same data across different locations and devices. Healthcare is the most common adopter of smart devices (47%), followed by security (34%), energy (30%), and manufacturing (28%), as shown in Figure 1. To best model low-end smart devices in this ecosystem, their architectural composition will first be specified, and their hardware classes will be separated.

*Author for Correspondence

Nausheen Khilji

E-mail: nausheen.khilji@jietjodhpur.ac.in

¹Professor, Department of Computer Science Engineering, (AIML/DS/CYS), JIET Universe, NH-62, Pali Road, Mogra, Jodhpur, Rajasthan, India

²Assistant Professor, Department of Computer Science, Engineering (AIML/DS/CYS), JIET Universe, NH-62, Pali Road, Mogra, Jodhpur, Rajasthan, India,

Received Date: June 29, 2026

Accepted Date: August 25, 2026

Published Date: August 31, 2026

Citation: Ashish Sharma, Nausheen Khilji. Next-Generation OS Architectures for Low-Power Smart Hardware: A Comprehensive Review. Journal of Operating Systems Development & Trends. 2026; 13(2): 15–27p.

Fundamental Smart Device Hardware

This involves a large range of hardware and software types, such as the integrated computing

platform part (Figure 2), an OS, communication modules (IoT), and application-specific sensors, for example, smart nodes, which must be exposed to efficient and optimized special memory and power. Hardware is the computing platform on which applications and operating systems execute, and as device agility becomes increasingly relevant to users, we must ensure that those functions occur at very high speeds on that hardware. This is complemented by Introduction/Output (I/O) units that offer the necessary communication between the user and the hardware that enables the entrance, storage, and retrieval of information.

The device uses special sensors embedded in it that exploit application-specific stimuli in the physical environment to perform tasks according to instructions developed in the prewriting stage. To obtain a real-time monitoring and control system, these sensors must be capable of capturing high-resolution data with minimal erroneous noise or interference [7]. The overall system was also improved in terms of data management through the use of a database and memory components. Memory [8] is the physical storage medium, which may be both temporary and permanent, that can then be used to multiply tasks, to decompose and perform complex tasks; and the database [9] assists in organizing this data in such a way that it is readily accessible, stored, and re-stored.

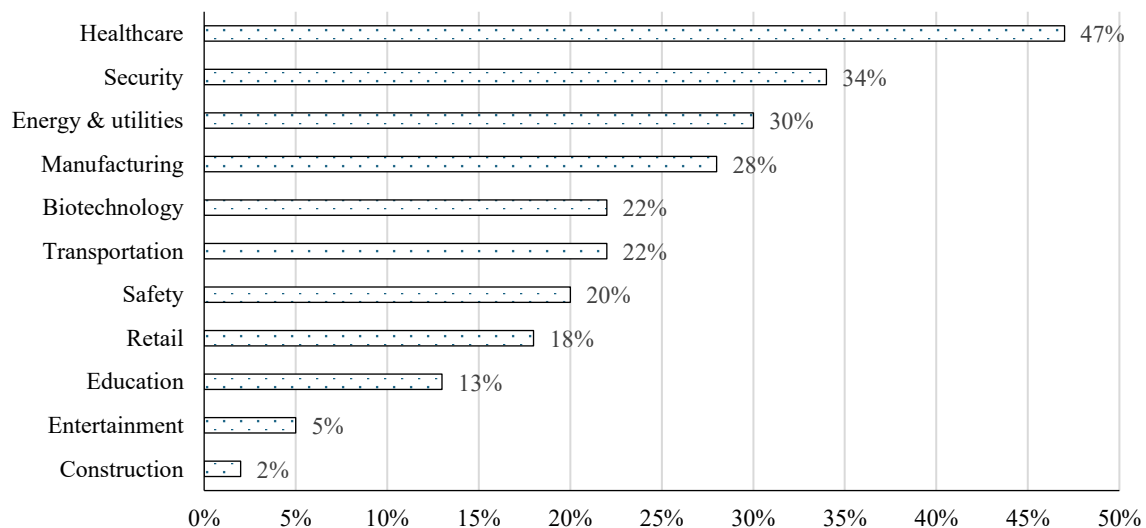


Figure 1. Smart device area distribution.

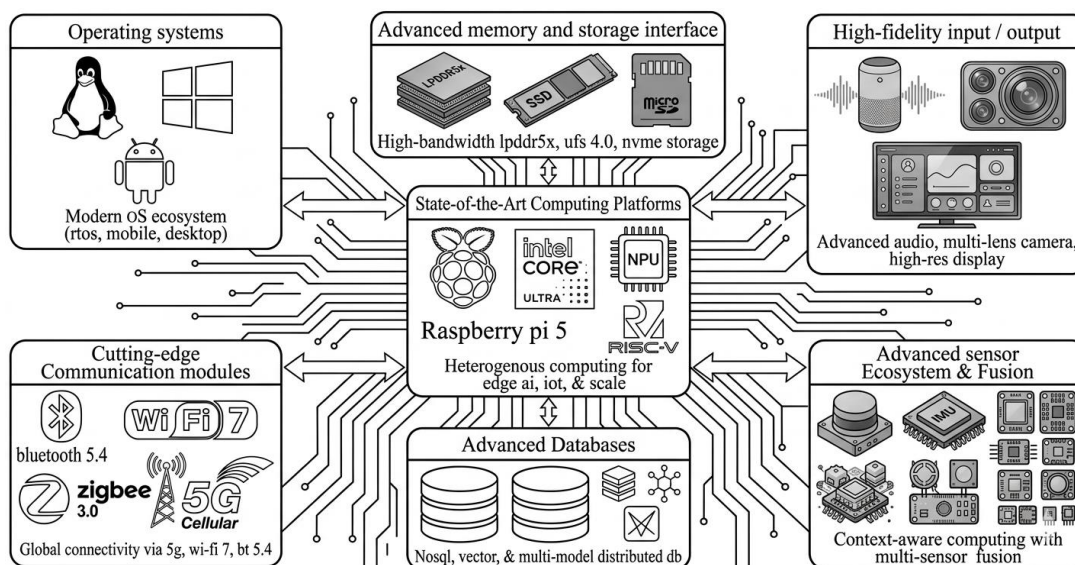


Figure 2. Basic components of a smart device.

The communication module and power element provide the device with the capability to remain operational and connected. The power supply, which can be as simple as batteries and power banks, or as complex as solar panels or a master supply, should be reliable enough to keep all inner peripherals running. Conversely, with regard to integration of IoT, protocols incorporated in the communication module facilitate the ability of the device to exchange information with other nodes within its smart environment. It is the operating system that controls all of this, and it is the software that governs the hardware resources and brings the device to a current level of connectivity, communication and security standards.

Classification and Characteristics of Low-End Smart Devices

On the low-end smart-devices, it is noteworthy (perhaps obvious) to state that because of resource-constrained conditions, they are significantly limited in three key fronts which are computational power (CPU), energy and memory. Such hardware platforms include, but are not limited to, Arduino Due, Atmel SAM R21, Econotag, IoT-LAB M3, OpenMote, TelosB, and Zolertia Z1. Nevertheless, the Internet Engineering Task Force (IETF) has categorized these devices into three classes, namely Class 0, Class 1, and Class 2, based on their memory and code storage limits, to standardize the discussion of this subject [10].

Class 0 devices are the smallest devices, usually having less than 10 KB RAM and 100 KB Flash memory. As a result of these restrictive constraints, Class 0 devices are likely to implement an architecture that uses 8-bit MCUs and cannot safely send or receive data directly to or from the Internet without the aid of gateways or proxies. The middle ground between Class 1 devices with typically 10KB data memory and 100KB code size, and Class 2 devices at the high end of the low-end with a typical 50KB RAM and between 250 and 500KB Flash. Although the latter two classes may be used to provide 8-bit–32-bit architectures, the point of convergence is that their resources are too limited to provide resource-based traditional operating systems such as Linux [11].

As industry trends shift toward smaller, cheaper, and more energy-efficient hardware, lightweight operating systems are a necessity. Various open-source and proprietary operating systems have been developed to enable these devices to be Internet-enabled, but the architecture is quite different. This paper discusses the basic needs of this type of software and presents abstract architecture that can guide researchers in constructing application-oriented lightweight operating systems. The remainder of this paper is divided into the following sections: Section 2 describes the design features of low-end OSs; Section 3 presents the research method and literature analysis; Section 4 proposes a framework for low-end OSs; and Section 5 concludes the paper.

OPTIMIZING OS DESIGN FOR RESOURCE-LIMITED INTELLIGENT NODES

The motivation to develop a cohesive smart space lies in the capability of various gadgets to operate and interact between them through firmware and dynamic software on the operating system tier [2]. Conversely, a smart ecosystem operating system must be developed to ensure minimum memory and other power requirements compared with conventional computing systems. In contrast to hardware or older software architecture, operating systems can be viewed as a layered design in which the microkernel [12] or monolithic architecture attempts to trade-off between modularity and performance. In addition, the programming model must be accessible to programmers who generally use high-level languages to write application logic, with assembly language remaining an attractive choice for low-level hardware interfacing and tuning [12, 13].

Schedulers are also key to these systems, and in addition to offering real-time scheduling capabilities, the kernel should offer time-based process priorities to manage critical environmental data [13, 14]. Access to the Internet is an essential need. Therefore, the OS must observe that local Wireless Sensor Networks (WSNs) protocols, for example Bluetooth, Z-Wave and ZigBee are insufficient to meet communication needs of smart environment and must also meet communication needs. To address these

issues, a range of modern low-power protocols, including RPL, CoAP, and 6LoWPAN, have been directly integrated into the system architecture [12]. However, a smart device OS, when taken down to the bare bones, should continue to be as portable as possible to various hardware platforms, but be open source to allow ample room for customization/fine-tuning to ensure that application requirements can be met [12, 14].

Methodology

The study design was a multistage procedure (see Figure 3). Here, we formalize the findings using two methods: a general literature review and a formal user feedback analysis using questionnaires. The review literature has focused on the assessment of the available operating systems designed to work with smart devices; therefore, the main purpose of this study was to determine the strengths and weaknesses of the architectures. Through the collection of this information, he established the fundamental needs that many sites followed.

LITERATURE SURVEY

An extensive non exhaustive literature survey of different popular operating systems (OS) that can be used in low-end smart devices to construct a theoretical framework and study the state of art was conducted. The intention is to reveal the building capabilities, networking opportunities, and directional constraints of open-source and proprietary systems [15, 16].

Contiki is a free operating system intended to run on resource constrained sensor nodes with minimal memory. It is a written C event-driven kernel that provides preemptive multi-threading. The generality of Contiki can be seen through its capability to execute on a wide variety of platforms: 8051 SoCs with power, MSP430, Atmel AVR and nearly all ARM architectures. It also includes support for Ethernet, full IP stack including TCP and UDP, and multiple IoT protocols including 6LoWPAN, RPL, and CoAP. It also provides a lightweight communications stack, RIME, a security platform, ContikiSec [17], and the Cooja simulator for testing application code.

Another known open-source OS is TinyOS, which is designed specifically to run on very low-power wireless devices in smart building automation and pervasive computing [18]. It uses a domain-specific C variant named nesC that builds off a small base framework (less than 400 bytes). It now supports processors based on Atmel and Texas Instruments MSP-series, and ARM and Intel 8051 support is being developed. TinyOS natively employs a FIFO task scheduler, which was not originally designed to support real-time; a scheduler called Earliest Deadline First (EDF) is employed to do so [19]. They include multi-hop protocols, IPv6 support, the TinyDB database tool [20], and TinySec communication security [21]. Applications can be simulated using the TOSSIM environment [22].

Mantis is an open-source wireless micro-sensor system lightweight operating system that only requires approximately 500 bytes of memory [23]. It is C-based and offers a preemptive multithreading facility that can be used with Mica2, MicaZ, and Telos. Cross-platform testing is another characteristic of Mantis and gives developers the ability to test how an application will run on a PC or a PDA prior to the deployment of the hardware [23-25]; The AVRORA simulator supports this [26].

On a proprietary, commercial front, VxWorks is a long-time industry mainstay with long-standing and relevant support [27], supporting Intel, MIPS, and ARM-based architectures. Although IPv6 support is extensive, 6LoWPAN is not distributed as a natively constrained networking stack. Similarly, a widely used commercial microkernel-based OS is QNX (currently owned by BlackBerry) [28]. The latest variant is called QNX Neutrino, which is based on various high-performance processors, such as PowerPC and x86. Android was developed as a Linux-based generalized operating system originally designed to support high-end mobile devices, but in recent years, it has been focused on constrained space with Brillo (since Android Things) [29, 30]. Brillo is a slimmed-down Android application compatible with low-power, memory-restricted devices running on ARM, x86, and MIPS.

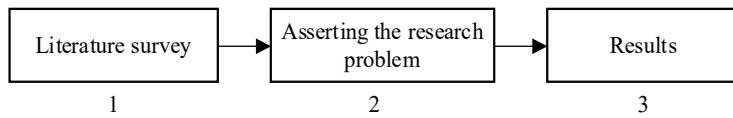


Figure 3. Research design.

Finally, sophisticated real-time operating systems (RTOS), such as EmbOS and FreeRTOS, offer extremely optimized scheduling of embedded units. EmbOS is an ideal memory consumption of 8-bit to 32-bit MCUs is a priority based pre-emptive multitasking system with ideal memory use of the RAM and ROM [31]. Simple compilation tools can be used to create applications that are executable on this RTOS [32]. It supports hardware porting and can run tickless mode with the most aggressive low power workloads, but with third-party stacks required to add the glitz of an actual internet connection.

Certainly. Below is the table converted into editable text/Markdown format, preserving the same headings, rows, symbols, and footnotes as closely as possible.

The features and hardware compatibility highlights of these surveyed systems are analyzed and compared (as in Table 1 and Table 2), and architectural highlights graphicalized as in Figure 4. Based on the data analyzed above, the following conclusions can be drawn:

- *Scheduling*: Standard priority-based scheduling is used in most current smart device operating systems (OSs).
- *Language*: C, if supported and developed, would be the language of this field.
- *Licensing*: Most research-grade systems are open-source and can be tailored for education and experimentation.
- *Hardware support*: ARM processors can be found on literally nearly every surveyed platform.
- *Isolated help*: Despite many of the more popular operating systems having wider adoption, there appears to be split support over IDEs and extensive, native real-time integration.

Problem Identification and Definition

Most low-end smart devices can support a variety of operating systems, and the abundance of different operating systems can pose a significant challenge to developers looking to select the best operating system for their particular use-case scenario. To formalize the requirements of these constrained operating systems, a subset of these operating systems was studied. In addition, the name of the Design and Development of Smart Device was taken as a source of user feedback survey in questionnaire-based workshops [33]. The survey was developed to respond to the practical outlook and requirements of developers regarding the lifecycle of OS design, specifically relating to those aspects that influence the efficiency of the development and performance of the system.

The questionnaire explored various essential research facets to identify the most prevalent gaps in development and user needs. This assignment was to decide on the favorite operating systems and programming languages and define the most demanded communication protocols to integrate devices. Also, this survey tries to rank top three major challenges encountered in developing, and also estimates target memory footprints of common solutions and potential methods of writing lightweight application-specific operating systems. This research seeks to offer a roadmap that can guide the development of an automated strategy for selecting and personalizing OS for resource-constrained hardware.

RESULTS

Retention- Two workshops were asked to complete a structured survey [34], which is summarized in the above section. Figure 5 summarizes all the obtained data on 90 subjects in total [35, 36]. In the final list of the most salient characteristics of low-end smart device operating systems, the technical results of Section 3.1 are synthesized with the survey responses obtained in an empirical manner. Moreover, the

Table 1. Key features of an existing operating system for low-end smart device.

Features - >Smart Device OS↓	Design objectives	Licence	Architecture	Scheduling algorithm	Platforms	Language supported	Support in IDE
Contiki	Commercial	BSD/Free educational	Monolithic	Interrupts execute w.r.t priority	Tmote, AVR Series, ARM	C	Supported by Eclipse IDE
TinyOS	Educational/Research	BSD/Free educational	Monolithic	FIFO	Atmel AT90L-series, Atmel ATmega-series, and T1, Tmote	nesC	No Support
Mantis	Educational/Research	BSD	Layered	Pre-emptive priority-based scheduling	Mica2, MicaZ, Telos	C	No Support
VxWorks	Commercial	Proprietary	Layered	Pre-emptive and Round Robin Scheduling	ARM, Intel 64, MIPS, PowerPC, SH-4; Strong ARM, xScale	C/C++	Supported by Toolbox IDE
QNX Neutrino	Commercial	Proprietary	Microkernel	Priority-Pre- emptive Scheduling	MIPS, PowerPC, SH-4, ARM, xScale	C/C++ / Java	Supported by Toolbox IDE
Android	Commercial / Educational / Research	GPL/Free educational purpose	Monolithic	Fair Scheduling	ARM, X36, MIPS	C/C++ / Java	Eclipse
Embos	Commercial	Proprietary	Modular	Priority based pre- emptive	8/16/32 bit processors/ARM cortex, ARM 7/9/11	C	No support
FreeRTOS	Personal Interest based	Modified GNU GPL	Microkernel RTOS	Priority Based pre- emptive	ARM (ARM7, ARM9, Cortex-M3, Cortex-M4), Atmel AVR, AVR32, SuperH, Nios II, Cortex-R4	C	No support

Table 2. Support of basic operating system requirements.

OS requirements → Smart device OS ↓	Open source (a)	Portability (b)	Real Time Requirement (c)	High Reliability (d)	Robustness (e)	Failure Handling (f)	Security (g)	Scalability (h)	Multitasking (i)	IoT (j)	Scheduling (k)	Network Support (l)	Memory management (m)
Coatiki	✓	✓	*	×	×	×	×	✓	✓	✓	✓	✓	✓
Tiny OS	✓	×	*	×	×	×	×	✓	×	✓	✓	✓	✓
Mantis OS	✓	✓	*	×	×	×	×	✓	✓	✓	✓	✓	✓
VxWorks	✓	✓	*	✓	×	×	✓	✓	✓	✓	✓	✓	✓
QNX Neutrino	✓	✓	*	✓	×	×	✓	✓	✓	✓	✓	×	✓
Android	✓	✓	×	*	✓	✓	*	✓	✓	✓	✓	✓	✓
EmbOS	×	×	✓	✓	×	×	×	×	✓	×	✓	✓	✓
FreeRTOS	✓	✓	*	×	×	×	×	✓	✓	✓	✓	✓	✓

✓ – full support, × – no support, * – partial support

The generic architecture of these devices is shown in Figure 6. The survey concluded that over 65 percent of those surveyed would want a generic, common framework to automatically generate fully fledged application specific operating systems led by the user. The primary needs identified based on this two-stage analysis are detailed below.

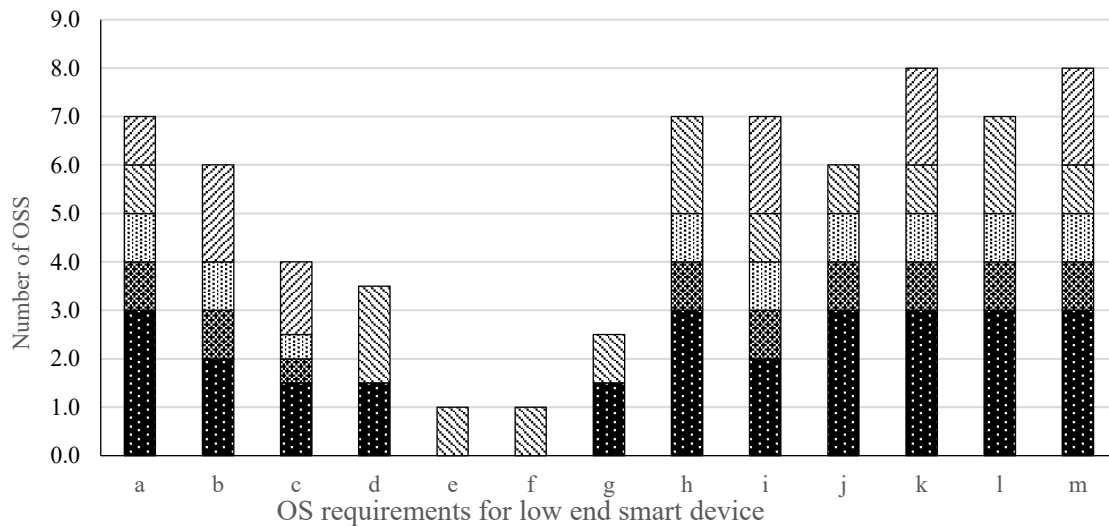
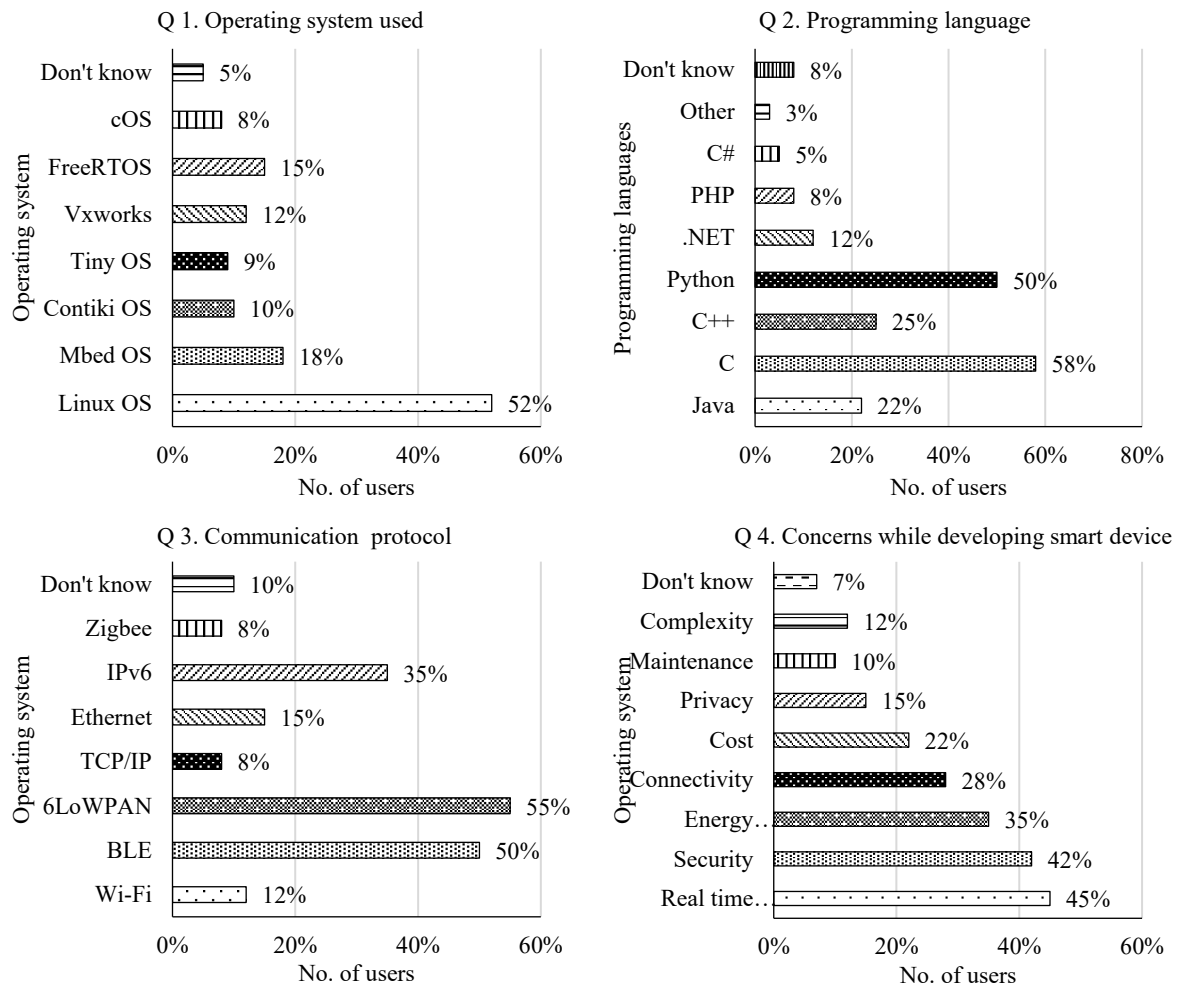


Figure 4. Support of basic OS requirements.



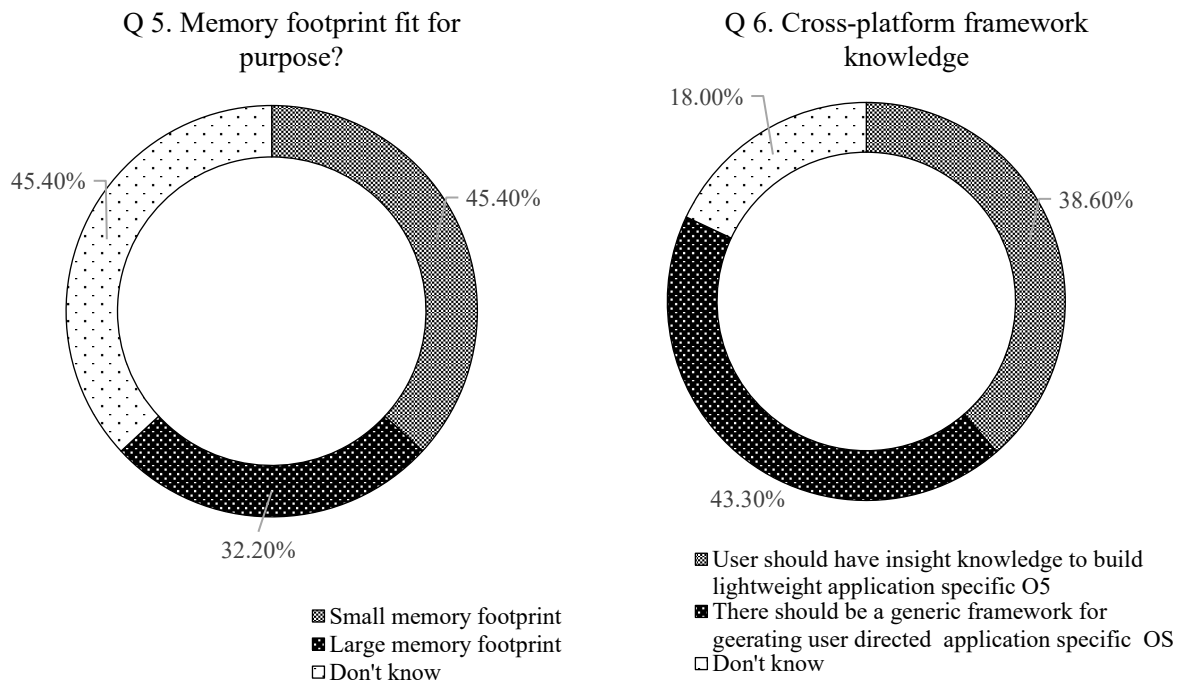


Figure 5. Responses of participants gathered based upon survey questionnaire.

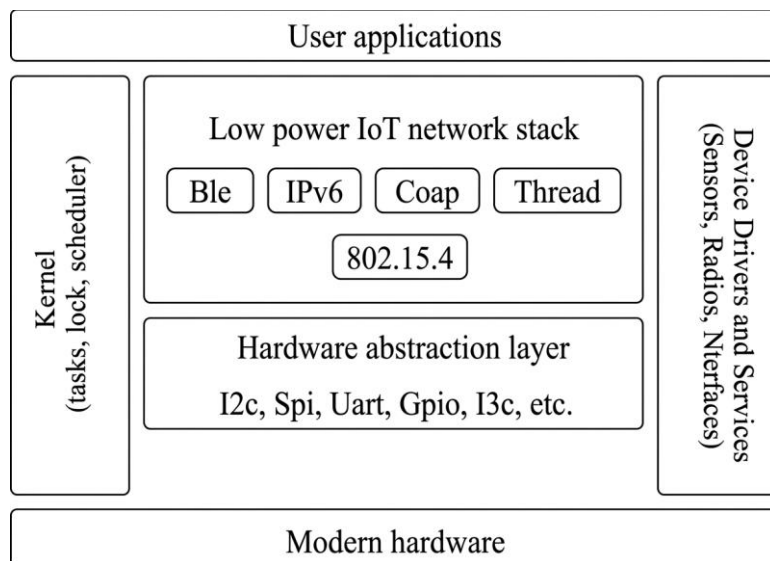


Figure 6. Generic Architecture of an OS for low-end smart devices.

A lot to Support Heterogeneous Hardware: A smart environment would be made up of a range of use cases which would require the operating system to support a wide range of hardware architectures. Future-proof systems should be capable of operating smoothly across highly heterogeneous platforms, even though the existing literature indicates that OS support is currently highly biased towards ARM, which also supports Atmel AVR and Texas Instruments MSP series processors.

Programming Model and Memory Footprint: A programming environment must support industry-standard languages (ANSI C, C++, etc.) that allow the creation of a memory-efficient application. Because smart devices are smaller and less expensive, they are typically programmed with a limited number of kilobytes. These libraries must be optimized to maintain a low OS footprint with minimal loss of functionality.

Communication protocols and standards: The interoperability of devices requires integration with standard communication protocols and standards (commonly used) [11]. These protocols should have low power consumption and high security levels. Every smart device will require networking protocol stack implementations over Internet Protocol (IP), including IPv6 and 6LoWPAN [11, 37], with the underlying operating system platform required to be compatible with both IEEE 802.15.4 radio systems and Bluetooth Low Energy (BLE) as a low-power networking protocol stack.

Real-time Responsiveness and Power consumption: In applications where the stakes are high, such as in smart robotics, healthcare monitoring, and car networks, timing and execution must be highly accurate. In this respect, the OS must be capable of facilitating real-time processing of data streams entering it without latency [38]. In addition, because most low-end devices are battery-operated and should last for years, particularly in industrial or building automation, energy efficiency is essential. The operating system must be able to run on ultra-low-power run-time to attain high hardware longevity [39].

Proposed Solution: Automated OS Generation Framework

In an effort to narrow down the general-purpose OS on smart devices, embedded research is on the rise. In such scenarios, developers struggled with the idea of constructing light weighted, domain-specific systems due to severe resource limits or due to the complexity of explicit kernel optimizations. The current paper, thus, suggests a generic new framework that can simplify the development of application specific operating systems that cut near what developers hope to observe as a perfect finish. The heart of this structure is a prolific configurable, intelligent graphical user interface (GUI) to enable the granular selection of modular peripheral interfaces, including the computing platform, core OS characteristics, and sensor and communications module, thereby eliminating huge heavyweight standard distributions.

The fundamental needs of this framework are based on the literature survey and empirical analysis presented in Section 3.3. Consequently, the operating system image developed using this framework encompasses these basic properties along with optimized application code, special APIs, and necessary device drivers of the hardware chosen. A developer, say of a smart home automation system with an ARM microcontroller, will then need libraries to combine light sensors, temperature sensors, ultrasonic sensors with Wi-Fi and camera sensors. Writing optimized code by hand to choose the numerous available peripherals and handle the relationships is tedious, but the suggested framework automates these steps. Generates a custom, minimal, OS image compatible with the ARM platform and the peripherals chosen by the user with the minimum possible overhead to provide an optimum configuration.

The flowchart provided in Figure 7 illustrates that the operation logic of the framework is constructed in the form of sequential modules. First, the system asks the user to provide simple parameters, such as the name of the device, type of processor architecture, and preferred type of operating system. The structure then inquires whether to load a formal or light picture within the container. The former, for example, leads to a regular download, with the latter triggering an application-specific granular selection process of peripherals. Once these requirements have been specified, the user can compile and perhaps even optimize these components using the framework, which then provides a custom-built image that can be flashed onto any storage media and booted straight onto the hardware used.

CONCLUSION AND FUTURE SCOPE

After a rigorous literature review and user work, we developed the official requirements for smart device operating systems. This kind of demand shows that there is the need to have a generic framework that can automatically generate light weight and custom build operating system depending on the application. As smart technology continues to expand in various spheres, such as education, industry and home automation, tools that facilitate access through hardware abstraction and resource optimization are essential.

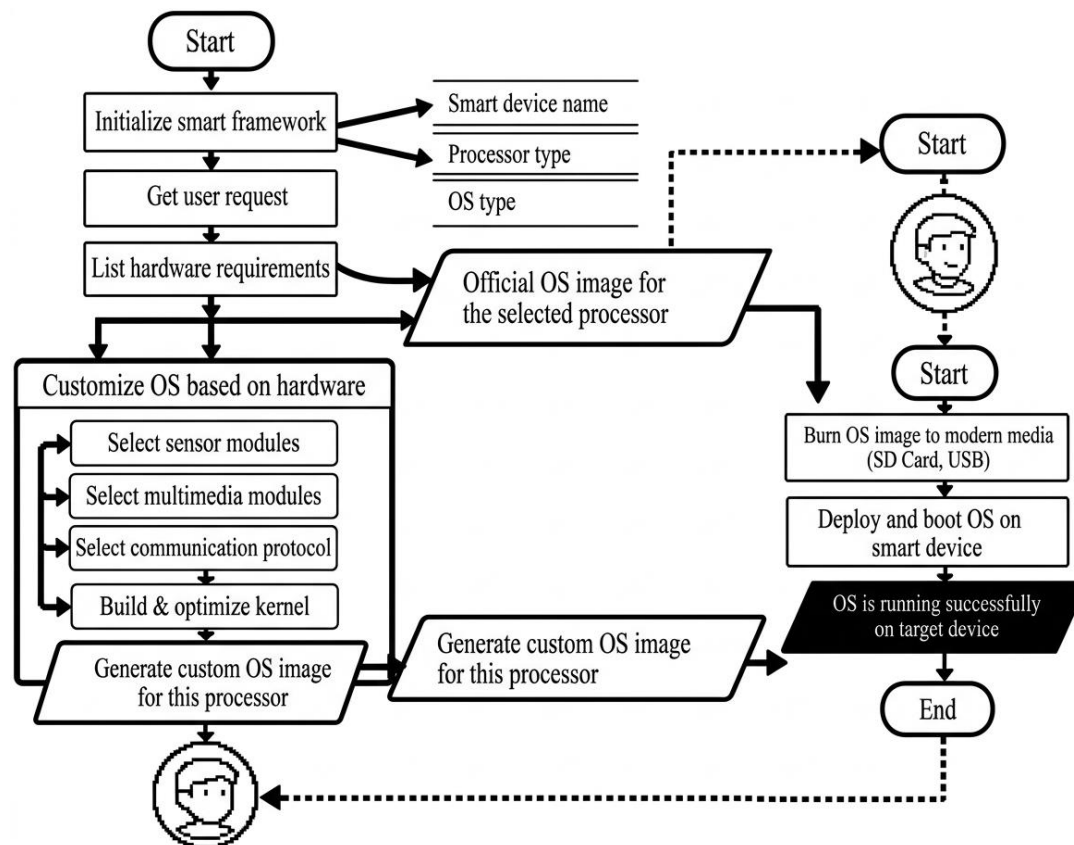


Figure 7. Generic flow chart of the targeted framework.

The proposed solution fills this gap by offering users an almost automatic approach to selecting application-specific peripherals and creating custom OS images in an automatic approach. To be universally adopted and iteratively extended, such a framework must be an open-source tool that people can use, build upon, and modify. As such, the application of this generic framework and adherence to the guidelines of our architectural design will be beneficial for ease of use that transcends the complex hardware constraints and requirements of more intelligent domain-specific smart solutions.

Declaration of Interest

The author(s) declare(s) that there is no conflict of interest regarding the publication of this manuscript.

REFERENCES

1. Davy A. Components of a smart device and smart device interactions. Telecommunications Software and Systems Group. 2003 Sep. p. 1-8.
2. Smart light bulb [Online]. Google Patents; 1998. Available from: <https://patents.google.com/patent/US6528954B1/en?q=Lys+I%2c+Muller+GG%2c+inventors.+Smart+light+bulb.+United+States+patent+US+6%2c528%2c954.+2003+Mar+4>
3. Popper S, Friedman R, Litbak A, Petel Y, inventors; Computerized Smart Faucet Ltd, assignee. Smart device and system for improved domestic use and saving of water. United States patent US 6,895,985. 2005 May 24.
4. Mohaideen AH, inventor. System and method for providing health care services using smart health cards. United States patent application US 12/171,953. 2009 Oct 8.
5. Lawson DC, Reichard DJ, Harkulich JA, Hessmer R, Chand S, Farchmin DW, inventors; Rockwell Automation Technologies Inc, assignee. Smart device for industrial automation. United States patent application US 13/725,660. 2013 Aug 15.

6. Gunge VS, Yalagi PS. Smart home automation: a literature review. *Int J Comput Appl.* 2016;975(8887-8891):24679-568.
7. WIPRO [Online]. Academia.edu; 2017. Available from: <https://www.academia.edu/31955135/WIPRO>
8. Gillis AS, Lutkevich B, Hughes A. Database (DB) [Online]. TechTarget; 2024. Available from: <https://www.techtarget.com/data-technologies/definition/database-DB>
9. Beal V. Memory (computers) [Online]. Webopedia; 2010. Available from: <https://www.webopedia.com/definitions/memory/>
10. Bormann C, Ersue M, Keranen A. Terminology for constrained-node networks. 2014 May 14.
11. Hahm O, Baccelli E, Petersen H, Tsiftes N. Operating systems for low-end devices in the Internet of Things: a survey. *IEEE Internet Things J.* 2016;3(5):720-34.
12. Gaur P, Tahiliani MP. Operating systems for IoT devices: a critical survey. In: 2015 IEEE Region 10 Symposium; 2015 May 13. IEEE; 2015. p. 33-6.
13. Baccelli E, Hahm O, Günes M, Wählisch M, Schmidt T. RIOT OS: towards an OS for the Internet of Things. In: 2013 IEEE Conference on Computer Communications (INFOCOM); 2013 Apr 14. IEEE; 2013.
14. Chandra TB, Verma P, Dwivedi AK. Operating systems for Internet of Things: a comparative study. In: Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies; 2016 Mar 4. p. 1-6.
15. Dunkels A, Grönvall B, Voigt T. Contiki—a lightweight and flexible operating system for tiny networked sensors. In: 29th Annual IEEE International Conference on Local Computer Networks (LCN); 2004 Nov 16. IEEE; 2004. p. 455-62.
16. Sesli E, Hacıoğlu G. Contiki OS usage in wireless sensor networks (WSNs). *Turk J Electromech Energy.* 2017;2(2):1-6.
17. Casado L, Tsigas P. Contikisec: a secure network layer for wireless sensor networks under the Contiki operating system. In: Nordic Conference on Secure IT Systems; 2009 Oct 14. Berlin: Springer; 2009. p. 133-47.
18. Levis P, Madden S, Polastre J, Szewczyk R, Whitehouse K, Woo A, et al. TinyOS: an operating system for sensor networks. In: Ambient Intelligence. Berlin: Springer; 2005. p. 115-48.
19. Montenegro G, Kushalnagar N, Hui J, Culler D. Transmission of IPv6 packets over IEEE 802.15.4 networks. 2007 Sep 25.
20. Madden SR, Franklin MJ, Hellerstein JM, Hong W. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans Database Syst.* 2005;30(1):122-73.
21. Karlof C, Sastry N, Wagner D. TinySec: a link layer security architecture for wireless sensor networks. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems; 2004 Nov 3. p. 162-75.
22. Levis P, Lee N, Welsh M, Culler D. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems; 2003 Nov 5. p. 126-37.
23. Bhatti S, Carlson J, Dai H, Deng J, Rose J, Sheth A, et al. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob Netw Appl.* 2005;10(4):563-79.
24. Leopold M, Dydenborg MB, Bonnet P. Bluetooth and sensor networks: a reality check. In: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems; 2003 Nov 5. p. 103-13.
25. Gellersen H. Smart-its: computers for artifacts in the physical world. *Commun ACM.* 2005;48(3):66.
26. Titzer BL, Lee DK, Palsberg J. Avrora: scalable sensor network simulation with precise timing. In: IPSN 2005: Fourth International Symposium on Information Processing in Sensor Networks; 2005 Apr 15. IEEE; 2005. p. 477-82.
27. Vytla SJ. VxWorks application programmers guide 6.2 [Online]. Scribd; 2026. Available from: <https://www.scribd.com/doc/131953382/Vxworks-Application-Programmers-Guide-6-2>
28. QNX Everywhere [Online]. QNX; 2026. Available from: <https://qnx.software/en/developers/get-started/qnx-everywhere>

29. Android overview [Online]. Open Handset Alliance; 2026. Available from: https://www.openhandsetalliance.com/android_overview.html
30. Nurmi I. Application layer protocol support in IoT cloud platforms [Online]. Iiro Nurmi Blog; 2017. Available from: <https://iinurmi.wordpress.com/2017/04/11/application-layer-protocol-support-in-iot-cloud-platforms/>
31. embOS-Classic real-time operating system user guide & reference manual for embOS-Classic and embOS-Classic-MPU [Online]. SEGGER; 2026. Available from: https://doc.segger.com/UM01001_embOS.html
32. FreeRTOS [Online]. FreeRTOS; 2023. Available from: <https://www.freertos.org/>
33. Reddy SRN, Kaur J, Chande S, Challa RK. My smartphone kit: design and development of an integrated platform for innovation and product design in engineering education. *Comput Appl Eng Educ*. 2018;26(3):642-54.
34. Kaur J, Reddy SRN. Operating systems for low-end smart devices: a survey and a proposed solution framework. *Int J Inf Technol*. 2018;10:49-58. doi:10.1007/s41870-017-0044-5.
35. Jain D, Chakraborty P, Chakraverty S. Smartphone apps for teaching engineering courses: experience and scope. *J Educ Technol Syst*. 2018;47(1):4-16.
36. Instructables. Build your own smartphone [Online]. Instructables; 2015. Available from: <https://www.instructables.com/Build-Your-Own-Smartphone/>
37. Eeckhout L. Is Moore's law slowing down? What's next? *IEEE Micro*. 2017;37(4):4-5.
38. Adam GK, Petrellis N, Doulos LT. Performance assessment of Linux kernels with PREEMPT_RT on ARM-based embedded devices. *Electronics (Basel)*. 2021;10(11):1331.
39. Min R, Bhardwaj M, Cho SH, Chandrakasan A. Energy-centric enabling technologies for wireless sensor networks. *IEEE Wirel Commun*. 2002;9(4):28-39.