Usage of Modular Software Development for IoT Nodes—A Case Study

Petar Rajković [1], Anđelija Đorđević [1], Dejan Aleksić [2] and Dragan Janković [1]

[1] *University of Niš, Faculty of Electronic Engineering; Aleksandra Medvedeva 14, Niš, Serbia*
[2] *University of Nis, Faculty of Sciences and Mathematics, Department of Physics*

Abstract
The Internet of Things (IoT) nodes are considered one of the main drivers in Industry 4.0 and beyond. With their processing and connectivity power and small size, they become key components for data collection and remote monitoring in production facilities. With a constantly increasing number of features, they start to play a more important role in data processing and close the gap with Edge computers in terms of flexibility. Back in time, they were known as very limited devices with relatively poor programming environments and low flexibility when it came to the use of development paradigms. Since their processing and connectivity power increased in recent years, followed by the impressive system-on-a-chip components, the possibility to bridge the gap in programming techniques towards the computing devices of the higher level emerged. With this paper, we wanted to show that the well-known techniques, native to high-power computers, could be adapted and used in the lowest levels of the ISA-95 technology stack. To explore the possibilities of the changeable software configuration of a running node, we developed a dedicated software package for the ESP32-based node whose primary aim is to manage the sensor network. In this case study, the modular software design backed up with a feature-flag main execution loop implementation is presented. The key benefit of such an approach is the possibility that the IoT node could be partially updated, without the need to be fully shut down, or switch to maintenance mode.

Keywords
Internet of Things, resource awareness, feature flags, modular software design

1. Introduction

This research came out of the project to support off-grid industrial and storage facilities whose location is in remote and hazardous areas. The focal requirement was creating a general-purpose IoT node based on the standard components that monitor remote sensor networks. The node must be able to work long periods without any manual intervention, to be monitored and updated "over-the-air" and adapt its working node to spend as little as possible of energy.

The project is based on the edge computing paradigm, which allows computation to be performed at the edge of the network, between cloud services and IoT services. "Edge" represents any computing or network resources between data sources and cloud data centers. Besides computing, data storage, caching, and processing, Edge can distribute requests and delivery services from a cloud to a user. The usage of the edge computing paradigm has numerous benefits, including reduced response time and lower energy consumption. The network bandwidth could be saved if a larger portion of data is handled at the edge, instead of uploaded to the cloud[1].

The main challenge imposed by the environment where the IoT node shall be installed is that the node must run without any wiring[2]. This would make the IoT node independent both from the computer network and the power grid. However, the IoT node is part of the wider system and represents a single layer in the ISA-95 (International Society of Automation) model[3].

SQAMIA 2023: Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 10–13, 2023, Bratislava, Slovakia

petar.rajkovic@elfak.ni.ac.rs (P. Rajković); andjelija.djordjevic@elfak.ni.ac.rs (A. Đorđević); alexa@pmf.ni.ac.rs (D. Aleksić); dragan.jankovic@elfak.ni.ac.rs (D. Janković)

0000-0003-4998-2036 (P. Rajković); 0000-0002-4760-8641 (A. Đorđević); 0000-0002-5833-6194 (D. Aleksić); 0000-0003-1198-0174 (D. Janković)

The IoT nodes are part of the lowest levels in the ISA-95 model. They are connected directly to the sensors, and they are responsible for their monitoring and data capturing. Since the sensors are often located in harsh condition areas (high acidity, extreme temperature, high voltage, possibility for explosion or fire) where the possibility of the incident and equipment loss is high, the IoT node must be the cheapest possible, smallest possible, and to have a lowest possible effect on the environment.

To achieve such a goal, hardware, software, and battery charging routines must be developed in accordance [4]. All these elements are considered the key components of energy and process efficient IoT nodes. **Figure 1** presents IoT node architecture [18]. The IoT node consists of the ESP-32 "system-on-a-chip" component, which exposes interfaces to the sensor and communication network. It is enriched with the energy supply components that should guarantee its autonomy.

Running in special conditions, and with the low-processing elements, the software for the IoT nodes was usually rudimentary. The programs are organized in a loop which consists of a sequence of statements that continuously perform the same set of operations. With the increase in processing power and the operation system complexity, the IoT nodes became a platform that could benefit all the software development paradigms that are used in high-end devices. Modular software design, feature-flag deployment, over-the-air software updates, remote control of the software behavior, and runtime configuration change are well-known and widely used for standard software development. Now, we have the chance to apply these concepts to a new level – in the dark realm of devices with low processing power.

This paper is focused on software design, where the concepts of modular software development and feature flag-based implementation were brought from the manufacturing execution systems and enterprise resource planning, the pieces of software that run in the highest ISA-95 levels. This approach became available since the actual hardware components are much more potent than the previous generation.

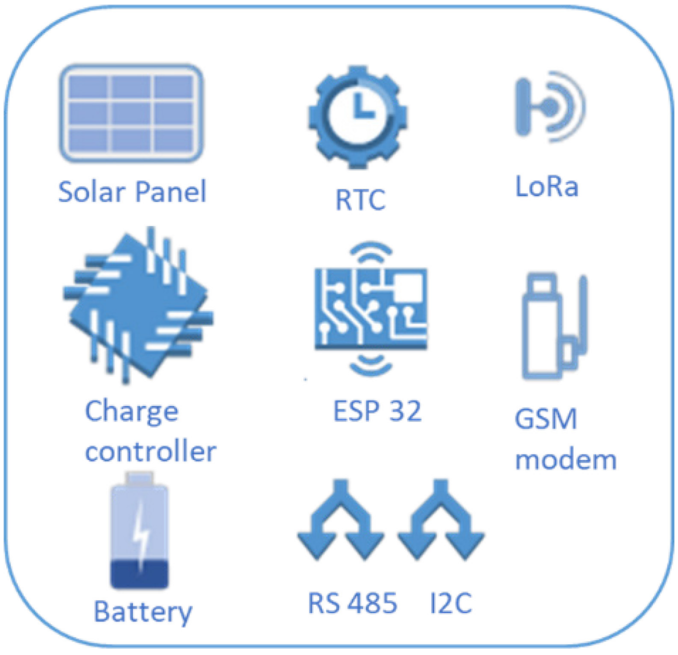


Figure 1: Main building blocks of IoT node architecture. Source: [18].

The IoT nodes are designed to run in cycles, where each cycle consists of periods of high activity (data collection, processing, and transmission) and periods when the node is in hibernate or sleep mode. The IoT node will spend some power while hibernating, then some power when collecting the data from the sensors, when processing collected data, and then it will spend power when transmitting the data to the ISA-95 Edge level.

The studies related to energy consumption in the IoT nodes [5]-[8] were used to define the design. One of the common challenges was the fact, that from time to time, some elements of node design must be replaced with a new component whose operation mode is a bit different than the previously used one. In that case, the node should be stopped, and the new software needs to be uploaded.

To address this challenge, every element of the hardware design is driven by a separate software module or task. Their execution is controlled by the main loop, which is designed with a feature flag approach. In this way, the main loop could easily switch off the part of the execution process while the targeted software module is getting updated, and then switch on again.

This paper represents the case study of the software development for low-level hardware, with the approaches used in the higher levels of the ISA-95 technology stack. Research goals are presented in section 1.1, and related work is described in section 2. Further sections describe the key elements of the software developed for the IoT node in detail. The closing section is a discussion of a proposed solution and closing remarks.

1.1. Research Goals

As stated before, the main goal for our running project is to create an IoT node primarily dedicated to working in remote off-grid areas. Its main purpose is to collect the signals from the sensor network, process collected data, and send them to the higher level – the Edge computer. All the benefits, explored before and presented in the earlier research [9][10], were incorporated into the proposed general design, which could be used as the model for similar implementations.

Besides many different custom-built solutions that could be found in the market and the literature, the main requirement was to stay with the widely used components which are easily affordable worldwide and backed up with wide support communities. Many of the existing (fully off-grid) designs were built on high-end components that are either too expensive, not easily replaceable, or without a wide enough support network. A description of the problem and some initial guidelines for the solution could be found in [11][12][13]. Having in mind maintainability, together with the focus on low energy consumption and high system readiness the main goal that drives this research is to:

- Create an easily adaptable software model that will allow node behavior change without installation or restart – to improve both maintainability and energy consumption.
- Support runtime changes of the working modes and make the system highly responsive to the update requests.
- Integrate the node into the digital twin to make the complete system more controllable (maintainability).

2. Related Work

To achieve stated goals, we had to rely on the known achievements, regardless of which domain of the IoT application they came from. For designing sensor networks, and routines about the data collection from the sensors, the research presented in [14] is interesting because of the sense of the design of the communication protocol.

Moreover, it deals with the sensor network used for medical purposes, and the problem of efficient energy management is common. The concept is based on the two-master node architecture, where one of them acts as the monitoring mode and the other as the transmitting mode. We decided to implement this concept by decoupling the data collection and processing from the data transmission routine.

Since data transmission uses a sizable part of the energy, this part was developed separately. The studies [15][16] give us an insight into the expected power consumption modes for the data transmission phase when different scenarios are employed. **Figure 2** presents a general state-based energy consumption model. The states of IoT nodes are compared in terms of energy consumption and time duration. It is important to point out that the sleep mode regularly takes more than 99% of the entire IoT node's uptime [15][16]. The energy model and battery lifetime estimation were used as the starting point in our research.

The IoT nodes are intended to work as a part of a wider system, and it is necessary to define the environment that would allow fast recovery in the cases when the IoT node needs to get replaced or refreshed. Therefore, we defined the set of recommendations for the software update processes in different IoT levels, so the establishment of a digital twin structure was recognized as a need to support development, testing, and later support when the system was in active usage [17][18]. During our research, the concept of dark launching expanded with feature flags deployment looked interesting with a possibility for wider application [19]. It was based on the concept that specific features in the software were enabled or disabled based on the value of the corresponding flags. The feature would be active only when the flag was set. The flag could be set or reset through the external interface and make the change in the software behavior without the need for restart or reinstalling. We applied this concept when designing the main loop of the ESP32 node [20][21].

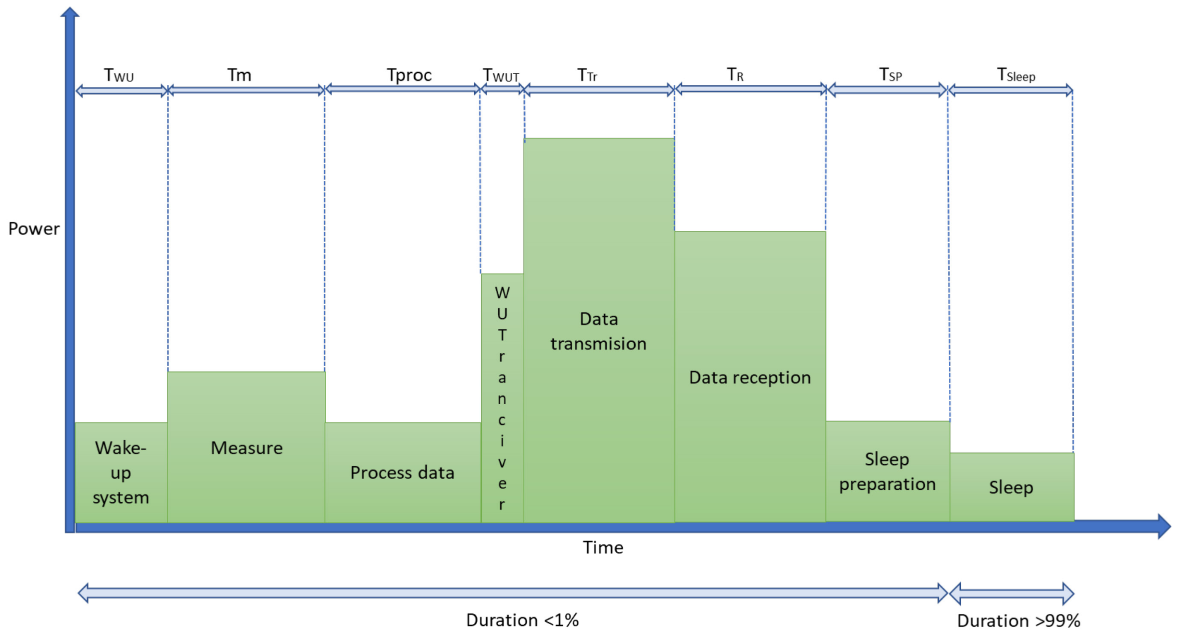


Figure 2: General state-based consumption model as described in [15][16].

In the examined case, the IoT nodes could be in remote locations where it is not possible to have a backup on site. For this reason, we decided to go with the concept of a digital twin in the cloud. The digital twin is a software structure that represents the physical network or system, and which allows faster and more effective governance, maintenance, or recovery [22].

3. Software Platform Structure

Software platform realization is based on the multitasking abilities that come from the system on a chip (SoC) based on ESP-32 which runs FreeRTOS [23]. Except for the main program, there are eight more different tasks and each of them supports a different system function. Each task could be treated as a separate software module, which could be independently replaced when needed.

Software implementation is organized around the main loop (main task) which could be followed by more tasks. All of them could be either controlled by the main task or run as the reaction to a specific signal from the environment.

Figure 3 displays a scheme of all the tasks implemented to support the operation of the developed IoT node. These tasks are divided into three major groups – setup and maintenance (represented by the graphic elements with a red border in **Figure 3**), sensor communication (green-bordered elements), and data transmission tasks (elements having a yellow border). Namely:

- all_param task – the set of routines and data structures responsible for the management of the system setup parameters. It is used to set control flags and enable or disable certain aspects of the system. This task backs up the feature flag approach which was used to define the main loop and to simplify the transition between different execution modes (transition from active to sleep and back)
- GPS_comm task – the task that controls the communication with the GPS (Global Positioning System) module. GPS_comm task is especially valuable for cases when the node is installed on some moving object and the correct position of the device is needed (i.e., in a barge used to transport crude oil in the rivers)
- LoRa_comm task – supervises the communication between the IoT node and Edge computer using the LoRaWAN protocol.
- GSM_comm task – oversees the backup communication channel between the IoT node and the Edge computer.
- MQTT_SN_comm task – It oversees the capacity and occupancy of the MQTT_SN queue. Also, it coordinates write processes from the data producers and read processes from data consumer tasks.
- I2C_comm task – The IoT node uses I2C to exchange data with the connected sensors. This task is responsible to keep this process and to support exchange routines promptly.
- RS485_comm task – like with the previous task, this one is responsible for supporting another way of communication between the IoT node and the sensors. The communication between the IoT node and sensors is executed under the RS 485 protocol in the master/detail mode.
- battery_charger task – it is used to manage the complex procedure of battery charge using the solar panel. This task controls the battery charging process and ensures that the IoT node should have enough power for uninterrupted operation.

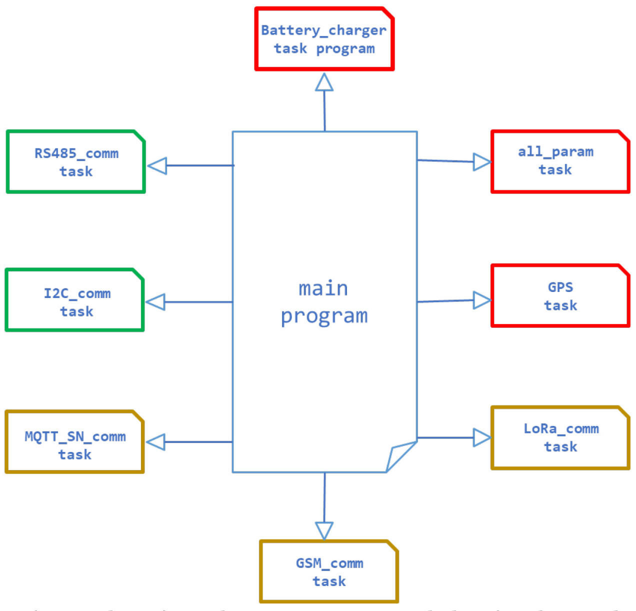


Figure 3: Main program (main loop) and supporting modules/tasks in the software design.

3.1. Main Program/Loop

The main program consists of two routines – setup and continuous loop. In standard implementations, setup is executed at the beginning of the work – either when the node is connected to the power, or when the hardware reset gets executed. To enable the feature flag approach, the setup routine is modified to enable and disable control flags in the continuous loop.

When discussing its default behavior, the setup method is designed to perform a minimal system initialization. It creates and sets the initial parameter for the series of events that control inter-task communication related to the main loop execution and the operation change to sleep mode.

After the initialization was completed, the conditions for the transition to sleep mode were set up. The last step in the initialization is starting the internal real-time clock (RTC) and setting up the remaining flags which will enable or disable functional blocks of the main loop.

In this way, the setup process could get executed instantly, and its main purpose is then to pass the set of configuration parameters to the predefined variables. Also, the setup process could be controlled over the air, which means that the device from a higher ISA-95 level could easily control and reconfigure the targeted IoT node.

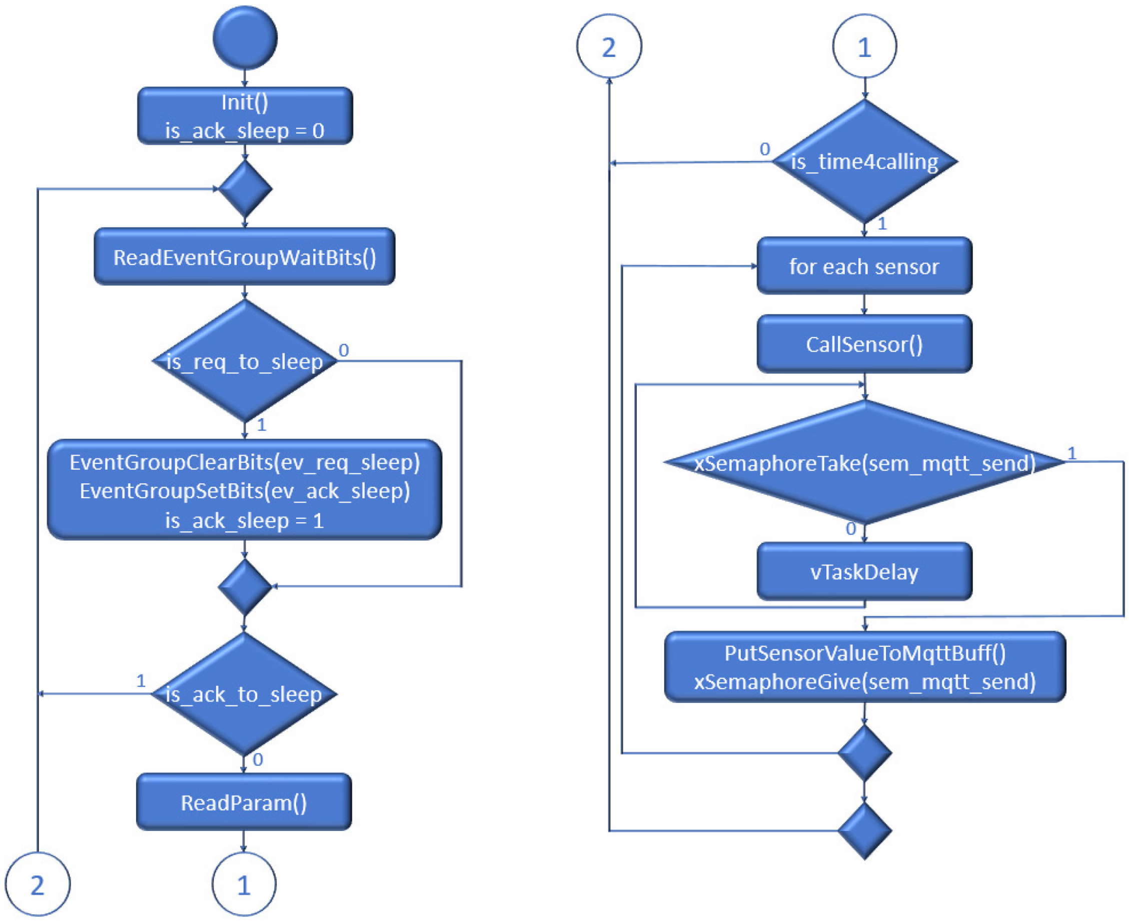


Figure 4: Main program – main blocks

As with the standard IoT nodes, after the setup routine gets executed, the loop routine starts. It ensures that the system works correctly and with minimal energy use. As it has been mentioned that the IoT node runs on a rechargeable battery, the crucial point is ensuring that the system works with the lowest possible energy usage. The loop routine executes the following steps (**Figure 4**):

- call_sensor, where the IoT node accesses the sensors and sets up a connection.
- read_data, where the IoT node reads the data from the sensor, stores them locally, and closes the connection.
- send_data, where the IoT node sends collected data from sensors to the Edge computer.

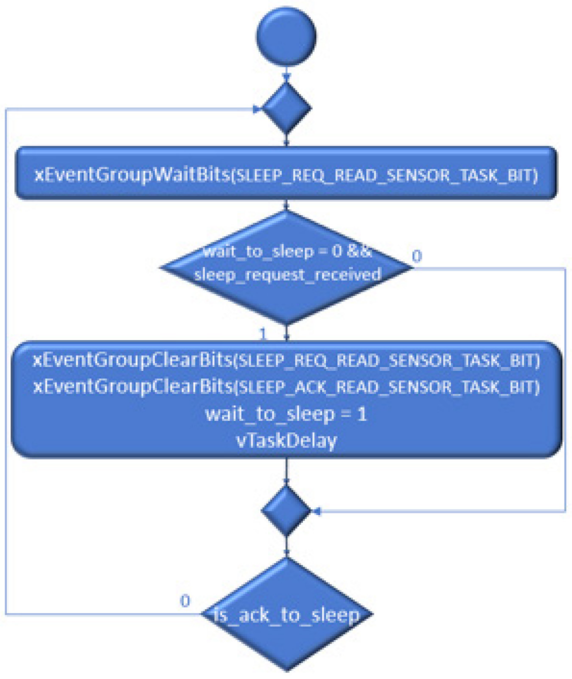
- go_to_sleep, where the IoT node changes its operation mode to reduce the energy consumption until it must re-execute the loop routine.

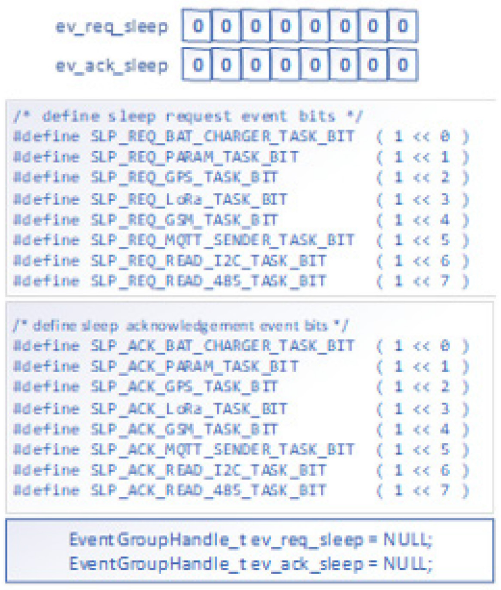
It is worth mentioning that every step in the main loop is controlled with the feature flag and can be easily turned on or off. In this way, the cost of replacement of the software component is only the time needed to upload a new version of the software, and to re-enable the feature flag. In most cases, this operation could be done even during the period when the IoT node is in inactive (sleep, hibernate) mode. As stated previously, sleep mode takes regularly more than 99% of the entire IoT node's uptime [15][16].

When the loop routine executes the call_sensor step, it contacts them one after another and reads the data. The data are then either collected until all the sensors from the list get contacted or sent at once to the Edge computer. The data has been sent, and the IoT node switches to sleep mode. The system stays in this stage until it should wake up and then it re-executes all the steps from the loop routing. The sleep period and the time for the wake of the system are controlled by the RTC, which uses a predefined pin to send the signal to the ESP32. As it has been mentioned before, the ESP32 has one auxiliary core whose role is to control sleep mode only and the sequence of commands that will drive the wake-up process.

4. Transition to Sleep Mode

Since the system operation is distributed between different tasks (**Figure 5**), the switch between active and sleep mode must be carefully implemented. The fact is that the IoT node must not forcibly close any active task since it could result in a wrongly sent signal to the sensors, or wrongly collected data. For this reason, every active task must be pre-informed about the potential switch to sleep mode. Before the switchover to sleep mode, each task must finish all the started actions and come to the state where a safe switchover is possible.


xEventGroupWaitBits(SLEEP_REQ_READ_SENSOR_TASK_BIT)
wait_to_sleep = 0 && sleep_request_received
0
1
xEventGroupClearBits(SLEEP_REQ_READ_SENSOR_TASK_BIT)
xEventGroupClearBits(SLEEP_ACK_READ_SENSOR_TASK_BIT)
wait_to_sleep = 1
vTaskDelay
0
is_ack_to_sleep



```
/* define sleep request event bits */
#define SLP_REQ_BAT_CHARGER_TASK_BIT   ( 1 << 0 )
#define SLP_REQ_PARAM_TASK_BIT         ( 1 << 1 )
#define SLP_REQ_GPS_TASK_BIT           ( 1 << 2 )
#define SLP_REQ_LoRa_TASK_BIT          ( 1 << 3 )
#define SLP_REQ_GSM_TASK_BIT           ( 1 << 4 )
#define SLP_REQ_MQTT_SENDER_TASK_BIT   ( 1 << 5 )
#define SLP_REQ_READ_I2C_TASK_BIT      ( 1 << 6 )
#define SLP_REQ_READ_485_TASK_BIT      ( 1 << 7 )

/* define sleep acknowledgement event bits */
#define SLP_ACK_BAT_CHARGER_TASK_BIT   ( 1 << 0 )
#define SLP_ACK_PARAM_TASK_BIT         ( 1 << 1 )
#define SLP_ACK_GPS_TASK_BIT           ( 1 << 2 )
#define SLP_ACK_LoRa_TASK_BIT          ( 1 << 3 )
#define SLP_ACK_GSM_TASK_BIT           ( 1 << 4 )
#define SLP_ACK_MQTT_SENDER_TASK_BIT   ( 1 << 5 )
#define SLP_ACK_READ_I2C_TASK_BIT      ( 1 << 6 )
#define SLP_ACK_READ_485_TASK_BIT      ( 1 << 7 )

EventGroupHandle_t ev_req_sleep = NULL;
EventGroupHandle_t ev_ack_sleep = NULL;
```

Figure 5: Transition to sleep mode – supporting flags and events

In devices such as IoT nodes, the tasks are executed independently and in parallel with the main program. This architecture requires the introduction of synchronizing inter-task mechanisms since all running tasks must inform the main loop when the switchover is safe. To achieve the required synchronization and have the controlled transition to sleep mode, the mechanism based on the event groups implemented inside FreeRTOS is used. Two 8-bit groups are defined - ev_req_sleep and ev_ack_sleep. Each bit from an event group is related to a single task within the system (**Figure 5** - left).

When the system switches over to sleep mode, the main program sets all bits in the ev_req_sleep event group and in this way requests all tasks to go to sleep mode. Each task then receives the request and starts the preparation for the transition. This is reflected by resetting the proper bit from ev_req_sleep. In this way, it informs the main program that the request is received. After the remaining operations within the task have finished, it sets the related bit in the event group (**Figure 5** - right).

The loop routine of the main program constantly checks bits from the ev_ack_sleep event group until all the tasks become ready to switch to sleep mode. When this happens, the main program calls the doit_sleep routine which controls the change in the system operation from active to sleep mode.

When it comes to the transition process to sleep mode, the tasks could be classified into three groups by the implementation approach. The simplest is the realization for tasks battery_charger and all_param. They could at once send a confirmation when the request for entering sleep mode was received. They have no actions or commands that must wait or interact with other software instances and thus could change state at the same point when it was asked. Furthermore, the task responsible for battery charging could operate independently and keep the battery in healthy condition.

The second group consists of tasks used to read data from the sensor network - I2C_comm, RS485_comm, and GPS tasks. They must finish checking all the sensors from their queues and read the data. After all these actions have been completed, tasks from the second group are ready for transition and could send the confirmation to the main program.

The third group is made of tasks that should wait until other tasks from the implementation confirm the transition to sleep mode. Only when dependent tasks are ready, the ones from the third group can finish their actions and confirm the possibility for transition. An example of such is MQTT_SN_comm which should wait for I2C_comm and RS485_comm to confirm their readiness. Only after that, it can confirm its readiness and then start with the creation of the data packages which will be stored in the synchronization buffer. After that, the mentioned packages can be uploaded to the Edge computer. When the upload process is finished, MQTT_SN_comm can confirm that it is ready to enter sleep mode.

5. Node Update and Re-Configuration

Implementation of such a system is characterized by a wide variety of different nodes. They could differ both in the topology and internal structure of Edge computers and IoT nodes. Each IoT node could be connected to various sensors that measure different physical values. Furthermore, internal software in the node could process measured values in separate ways. The complete system has such a level of complexity that makes the configuration of every single node different. For this reason, the configuration of the IoT nodes during installation and their re-installation after the updates are assumed as a complex organizational and logic problem.

To solve this problem, the Over-the-Air (OTA) mechanism, which controls a parameter change for every IoT node, has been implemented. The ESP32 has built-in 256B a Non-Volatile memory which means that the data stored remains even when the ESP32 undergoes a reset or power is cycled. Configuration parameters are stored in the mentioned memory. Initially, that memory will store only two pieces of information – the MAC address of the device and the IP address of the Edge computer which is the IoT node itself connected. After the initial start, every IoT node will contact the Edge computer from the configured address and download additional configuration parameters.

These parameters are stored in the database in the cloud. Since the MAC address of the IoT node is unique data, they are used for the exact identification of the IoT nodes. Owing to this, the IoT node will send its MAC address to the Edge computer, Edge computer will access the database in the cloud and take the configuration associated with the requesting IoT node. The retrieved configuration is then pushed from the Edge computer to the IoT node to complete the initial setup. These parameters could be specific for each IoT node, but the following is needed in most cases: several sensors connected to the IoT node, type of the physical interface, sensor addresses, data collection period, key, and boundary values of the measured parameters on the connected sensors, etc.

Internal configuration management in the IoT node is implemented in the all_param task. Two main reasons for such an approach are the controlled read/write of parameters from/to Non-Volatile memory and the centralized loading, storing, and manipulation of the configuration parameters. It is well-known that flash memories have a limited number of write operations. Data could be read Flash as many times as you want, but most devices are designed for about 100k to 1M write operations. Because of this, the all_param task before each write request should read the current value in Non-Volatile memory, and then perform write only if the new value differs from the one which is currently stored. Task all_param manages the update process of the configuration parameters. It synchronizes their values in the IoT node with the values in the cloud and reduces the need for contact between the IoT nodes and the cloud. The benefit of such an approach is reduced traffic and power consumption in the IoT layer. On the other hand, the drawback is that the remaining processes could not change configuration parameters which requested higher programming effort. However, since the focus was on execution efficiency, this was a calculated cost.

6. Discussion

The default software update method for IoT nodes implies complete node downtime. This downtime could last several execution cycles, and during that period, the node would be completely out of order. After the software update, the node needs to be restated and verified, which takes a certain time. One cycle is defined as the sum of time spent in active and in sleep mode (according to **Figure 2**). The average total time needed to stop the node, update the software, restart, and verify is around ten cycles.

With the introduction of modules and feature flag approach, the downtime could be significantly reduced. Depending on the task that should be updated, the overall effect varies (Table 1).

If the part related to data collection needs to be updated, one of the I2C or RS485 communication tasks will be replaced. To achieve this operation, a data collection flag needs to be switched off, and the targeted software module could be updated. Depending on the volume of the updated module, downtime will last 2 cycles at most, and it will affect only the data collection part. If buffering is enabled, data processing and data transmission will continue to execute.

In case some of the data transmission modules should be updated (LoRa or GSM) downtime could be avoided. Since one channel is active, before the software update, the node will switch to an alternative channel, and the software update will be executed without interruption. It is worth mentioning that in the case when the node switches from LoRa to GSM, it will result in higher energy consumption during the update, but the node will continue to operate as designed.

When the GPS communication node should be updated, its feature flag will be disabled, and the system will continue to run without interruption. The consequence of this update process will be that data packages transmitted from the node will have no geolocation associated. This problem will last for one or two cycles, depending on which point of the cycle the node acquires GPS coordinates.

Updates of battery chargers and setup tasks do not affect execution. These tasks run in a separate thread, and even in the separate core of ESP32 which is dedicated to supporting the sleep node. In that sense, they are completely safe to be updated at any given moment.

When the task dedicated to the message queue needs to be updated, the system could continue to work regularly, but without buffering capability. This means that all data stored in the buffer will be lost, and during the update, the system will be able to process and transmit only the data acquired in the current cycle.

Table 1
Effects on the IoT node when some software modules are updated.

Task name	Downtime (in execution cycles)	Downtime - effect
all_param_task	0	No effect on execution
GPS_comm_task	0	The data package transmitted from the node will have no geolocation info during the update
LoRa_comm_task	0	Optionally data transmission switches to an alternative channel
GSM_comm_task	1-2	Optionally data transmission switches to an alternative channel
MQTT_SN_comm_task	2	Buffering disabled, the node will be able to send only data acquired in the current cycle
I2C_comm_task	2	Data collection inactive
RS485_comm_task	2	Data collection inactive
battery_charger_task	0	No effect on execution
Setup_task	0	No effect on execution
Main	2	Total
Standard software update	8-10	Total

With the proposed software model, the only part of the update process that would require total downtime is the update of the main loop. It will cause the system to stop working for two cycles. In one cycle, the software itself will be replaced, and in the next, the setup task will start the node again. Since the IoT node is part of the wider systems, and integrated into a cloud-based digital twin, the complete control of the update process could be done over the air [24] and remotely with a minimal impact on the node's operations.

Due to the significant increase in processing power and storage space in the components used for the IoT nodes, it is expected that soon, all software development procedures become closer to the development of large-scale information systems where many already proven approaches and paradigms will make the software development easier. With this case study we wanted to investigate how far and how fast development could be improved at each moment, and looking at the results, it seems promising.

7. Conclusion

Moving the software developing paradigm closer to higher-level components, the software in IoT nodes could be designed and maintained more effectively. Since the standard components used as the base for IoT nodes, such as ESP32, become more powerful and closer to standard computers, their software will look more like the software developed for “real” computers. This will be followed with the standardized development environment, methodologies, and paradigm, and will ease software development in the entire industrial landscape.

This paper describes usage and gives tips for modular software development for IoT nodes. The paper provides energy efficiency measurements and instructions for reducing node downtime. One of the main benefits that this study pointed out is the ability to significantly reduce node downtime while the software update process is running. By dividing software into modules, a separate update becomes possible. Depending on a module that is updated, the effect on the system readiness is minor compared to the several cycles of the total downtime as with the standard approach. Feature flags, implemented as the part of main loop give the possibility to partially stop some segments of the node's execution process leaving the node in a stable and operational state. The negative effect of the update is visible only in one or two execution cycles, and after that, the node continues to operate as expected.

With the significant improvement in the design and processing power of the system-on-a-chip components used in IoT node design, it becomes possible to bring all the benefits of the standard desktop and Web programming. Many approaches, that are successfully used in higher programming languages become possible in low-level programming for controllers. With this paper, we intend to show the benefits that could be achieved in IoT node design when the proper components are used and when well-known concepts, such as modular programming, are used in a new area – in the programming of low-power processing units and IoT nodes.

Acknowledgments

This work has been supported by the cost action CA 19135 CERCIRAS (Connecting Education and Research Communities for an Innovative Resource Aware Society).

This work is part of the research that has been funded by the Ministry of Education, Science, and Technological Development of the Republic of Serbia, grant number 451-03-68/2022-14/200102.

References

[1] Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637-646, doi: 10.1109/JIOT.2016.2579198.

[2] Guidelines for integrated risk assessment and management in large industrial areas, https://www-pub.iaea.org/MTCD/publications/PDF/te_994_prn.pdf, last accessed on June 22nd, 2023

[3] ISA-95 standard page, available online: https://www.isa.org/standards-and-publications/isa-standards/isa-standards-committees/isa95, last accessed on June 26th, 2023

[4] Kumar, S., Tiwari, P., & Zymbler, M. (2019). Internet of Things is a revolutionary approach for future technology enhancement: a review. Journal of Big Data, 6(1), 1-21.

[5] Andres-Maldonado, P., Lauridsen, M., Ameigeiras, P., & Lopez-Soler, J. M. (2019). Analytical modeling and experimental validation of NB-IoT device energy consumption. IEEE Internet of Things Journal, 6(3), 5691-5701

[6] Mocnej, J., Miškuf, M., Papcun, P., & Zolotová, I. (2018). Impact of edge computing paradigm on energy consumption in IoT. IFAC-PapersOnLine, 51(6), 162-167.

[7] Dos Anjos, J. C., Gross, J. L., Matteussi, K. J., González, G. V., Leithardt, V. R., & Geyer, C. F. (2021). An algorithm to minimize energy consumption and elapsed time for IoT workloads in a hybrid architecture. Sensors, 21(9), 2914.

[8] Li, W., & Kara, S. (2017). Methodology for monitoring the manufacturing environment by using wireless sensor networks (WSN) and the internet of things (IoT). Procedia CIRP, 61, 323-328, DOI: doi.org/10.1016/j.procir.2016.11.182

[9] Aleksic, D. S., Jankovic, D. S., & Rajkovic, P. (2017). Product configurators in SME one-of-a-kind production with the dominant variation of the topology in a hybrid manufacturing cloud. The International Journal of Advanced Manufacturing Technology, 92, 2145-2167.

[10] Aleksić, D. S., Janković, D. S., & Stoimenov, L. V. (2012). A case study on the object-oriented framework for modeling product families with the dominant variation of the topology in the one-of-a-kind production. The International Journal of Advanced Manufacturing Technology, 59, 397-412.

[11] Hamied, A., Mellit, A., Zoulid, M. A., & Birouk, R. (2018, November). IoT-based experimental prototype for monitoring of photovoltaic arrays. In *2018 International conference on applied smart systems (ICASS)* (pp. 1-5). IEEE.

[12] Hussain, F., Hussain, R., Hassan, S. A., & Hossain, E. (2020). Machine learning in IoT security: Current solutions and future challenges. *IEEE Communications Surveys & Tutorials*, *22*(3), 1686-1721.

[13] Cheddadi, Y., Cheddadi, H., Cheddadi, F., Errahimi, F., & Es-sbai, N. (2020). Design and implementation of an intelligent low-cost IoT solution for energy monitoring of photovoltaic stations. *SN Applied Sciences*, *2*(7), 1165.

[14] Alkhayyat, A., Thabit, A. A., Al-Mayali, F. A., & Abbasi, Q. H. (2019). WBSN in IoT health-based application: toward delay and energy consumption minimization. Journal of Sensors, 2019. DOI: doi.org/10.1155/2019/2508452

[15] Bouguera, T.; Diouris, J.-F.; Chaillout, J.-J.; Jaouadi, R.; Andrieux, G. Energy Consumption Model for Sensor Nodes Based on LoRa and LoRaWAN. Sensors 2018, 18, 2104. https://doi.org/10.3390/s18072104

[16] Rajab, H., Cinkler, T., & Bouguera, T. (2021). Evaluation of Energy Consumption of LPWAN Technologies, available at Research Square, DOI: 10.21203/rs.3.rs-343897/v1

[17] Rajković, P.; Aleksić, D.; Djordjević, A.; Janković, D. Hybrid Software Deployment Strategy for Complex Industrial Systems. Electronics 2022, 11, 2186. https://doi.org/10.3390/electronics11142186

[18] Rajković, P., Aleksić, D., Janković, D., Milenković, A., & Đorđević, A. (2021, September). Resource Awareness in Complex Industrial Systems–A Strategy for Software Updates. In Proceedings of the First Workshop on Connecting Education and Research Communities for an Innovative Resource Aware Society (CERCIRAS), Novi Sad, Serbia (Vol. 2).

[19] Fowler, M. DarkLaunching, April 2020.

[20] ESP32-WROOM-32 Datasheet, available online: https://cdn-shop.adafruit.com/product-files/3320/3320_module_datasheet.pdf last accessed on February 25th, 2023

[21] ESP32 Series Datasheet, available online: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf last accessed on February 25th, 2023

[22] Liu, M., Fang, S., Dong, H., & Xu, C. (2021). Review of digital twin about concepts, technologies, and industrial applications. Journal of Manufacturing Systems, 58, 346-361, ISSN 0278-6125, https://doi.org/10.1016/j.jmsy.2020.06.017.

[23] FreeRTOS resource page, available online: https://www.freertos.org/, last accessed on February 26th, 2023

[24] Bauwens, J., Ruckebusch, P., Giannoulis, S., Moerman, I., & De Poorter, E. (2020). Over-the-air software updates in the internet of things: An overview of key principles. IEEE Communications Magazine, vol. 58, no. 2, pp. 35-41, doi: 10.1109/MCOM.001.1900125.