http://doi.org/10.35784/iapgos.7076 received: 14.01.2025 | revised: 04.09.2025 | accepted: 08.09.2025 | available online: 30.09.2025

IOT SYSTEM WITH FREQUENCY CONVERTERS OF PHYSICAL QUANTITIES ON FPGA

Oleksandr V. Osadchuk, Iaroslav O. Osadchuk, Valentyn K. Skoshchuk
Vinnytsia National Technical University, Vinnytsia, Ukraine

Abstract. *The paper presents an IoT system that ensures efficient collection, processing, and transmission of measurements from multichannel radio engineering systems based on FPGA. The developed system is designed for use in remote and hard-to-reach locations, where precision, reliability, and energy efficiency are critical. The implementation integrates the LilyGo LoRa32 module, which supports modern wireless communication protocols such as LoRa, WiFi, and Bluetooth. The core of data transmission is the compact CBOR format, which minimizes the volume of transmitted data and delays. The system includes a LoRa hub capable of receiving data from numerous collection devices, consolidating them into a centralized point, and transmitting them to a server via a WiFi connection. On the server side, a REST API has been developed based on the FastAPI framework, allowing the receipt of data in CBOR format, processing it, and storing it in an SQLite database. The use of the HTTPS protocol ensures the security of transmitted data, including confidentiality, authenticity, and integrity. The server also provides clients with access to data through multiple API interfaces, facilitating easy integration with other systems. A mobile application was separately developed using Kotlin and Android Studio, providing convenient access to the collected measurements. The application supports real-time dynamic data updates, allowing users to monitor connection status, select hubs and collection devices, as well as analyze and visualize the received results. The developed IoT system demonstrates high performance and versatility, making it suitable for a wide range of applications, including industrial automation, environmental monitoring, agriculture, and other IoT scenarios. Further development of this system includes improving data processing algorithms, increasing the hub's throughput capacity, and expanding functionality for real-time monitoring. This paves the way for the creation of innovative IoT solutions capable of meeting the demands of modern technologies in various fields.*

Keywords: FPGA, multichannel frequency meter, sensor with frequency output, IoT, LilyGo LoRa32, Nios II

SYSTEM IOT Z PRZEKSZTAŁTNIKAMI CZĘSTOTLIWOŚCI WIELKOŚCI FIZYCZNYCH NA FPGA

Streszczenie. *Artykuł przedstawia system IoT, który zapewnia wydajne gromadzenie, przetwarzanie i transmisję pomiarów z wielokanałowych systemów radiotechnicznych opartych na FPGA. Opracowany system jest przeznaczony do stosowania w odległych i trudno dostępnych lokalizacjach, gdzie kluczowe znaczenie mają precyzja, niezawodność i efektywność energetyczna. Implementacja integruje moduł LilyGo LoRa32, który obsługuje nowoczesne protokoły komunikacji bezprzewodowej, takie jak LoRa, WiFi i Bluetooth. Podstawą transmisji danych jest kompaktowy format CBOR, który minimalizuje objętość przesyłanych danych i opóźnienia. System zawiera koncentrator LoRa zdolny do odbierania danych z wielu urządzeń zbierających, konsolidowania ich w scentralizowanym punkcie i przesyłania do serwera za pośrednictwem połączenia WiFi. Po stronie serwera opracowano interfejs API REST oparty na frameworku FastAPI, umożliwiający odbiór danych w formacie CBOR, przetwarzanie ich i przechowywanie w bazie danych SQLite. Zastosowanie protokołu HTTPS zapewnia bezpieczeństwo przesyłanych danych, w tym poufność, autentyczność i integralność. Serwer zapewnia również klientom dostęp do danych poprzez wiele interfejsów API, ułatwiając integrację z innymi systemami. Oddzielnie opracowano aplikację mobilną przy użyciu Kotlin i Android Studio, zapewniającą wygodny dostęp do zebranych pomiarów. Aplikacja obsługuje dynamiczne aktualizacje danych w czasie rzeczywistym, umożliwiając użytkownikom monitorowanie stanu połączenia, wybór hubów i urządzeń zbierających dane, a także analizę i wizualizację otrzymanych wyników. Opracowany system IoT charakteryzuje się wysoką wydajnością i wszechstronnością, dzięki czemu nadaje się do szerokiego zakresu zastosowań, w tym automatyki przemysłowej, monitorowania środowiska, rolnictwa i innych scenariuszy IoT. Dalszy rozwój tego systemu obejmuje ulepszenie algorytmów przetwarzania danych, zwiększenie przepustowości koncentratora oraz rozszerzenie funkcjonalności monitorowania w czasie rzeczywistym. Otwiera to drogę do tworzenia innowacyjnych rozwiązań IoT, które są w stanie sprostać wymaganiom nowoczesnych technologii w różnych dziedzinach.*

Słowa kluczowe: FPGA, wielokanałowy miernik częstotliwości, czujnik z wyjściem częstotliwościowym, IoT, LilyGo LoRa32, Nios II

Introduction

The modern development of technologies demands the creation of precise, reliable, and energy-efficient measurement systems capable of operating in remote and hard-to-reach locations, such as mountainous areas or industrial zones. In the fields of the Internet of Things (IoT), industrial automation, environmental monitoring, and other domains, there is a growing need for multichannel measurement systems capable of parallel data collection and real-time transmission. However, existing approaches are often limited by the use of wired technologies, which reduce the mobility and scalability of solutions.

Moreover, integrating high-performance computing platforms like FPGA with modern wireless modules (LoRa, WiFi, Bluetooth) presents a challenge due to the need for optimizing data transmission formats and processing algorithms. Special attention should be given to the creation of a universal platform for collecting and transmitting data from multichannel measurement devices. Such systems must meet the following requirements:

1. Accuracy — ensuring high precision in measurement and data transmission.
2. Energy efficiency — minimizing energy consumption, especially for devices operating in hard-to-reach locations.
3. Scalability — the ability to support a large number of devices within the system without degrading performance.
4. Flexibility — support for various wireless data transmission technologies to adapt to usage conditions.

The proposed IoT system for multichannel data collection based on FPGA with LoRa, WiFi, and Bluetooth aims to address these challenges. The integration of the CBOR format for data transmission enables the creation of compact packets for efficient information exchange, while the use of a LoRa hub significantly simplifies data processing and ensures reliable delivery to servers or mobile devices.

The implementation of such a system represents a significant contribution to the development of IoT measurement solutions, enabling their deployment in areas where precision, mobility, and cost-efficiency are critical.

1. Analysis of recent research and publications

Previous research forming the basis of this study focuses on the development of multichannel frequency measurement systems based on FPGA and the integration of modern wireless technologies. Analysis of this research highlights key achievements and identifies prospects for further development.

In [10] which builds on earlier work on FPGA-based frequency metering a multichannel frequency measurement system is described using an Altera Cyclone IV EP4CE10F17C8 FPGA. Fig. 1 shows the block diagram of the finalized setup. Its core functions are to measure the frequency of input channels,

assemble packets with the results, and transmit these data via UART.

Consistent with established practice reported in prior literature, [10] employs a NIOS II soft-core within the multichannel frequency meter. This adopted approach provides flexibility for on-chip preliminary data processing and filtering and enables dynamic adjustment of the number of frequency meters without redesigning the data-processing algorithms – capabilities inherited from the earlier designs on which [10] is based.

In [10], the system's functionality was expanded by integrating an interface for interaction with digital sensors via I2C. For this purpose, an NIOS II core interface for I2C was created, the I2C protocol was implemented as a hardware block, and software was developed to support operations on the I2C bus.

In [11] which builds on earlier work, a prototype of a measurement collection device based on the LilyGo LoRa32 module for a multichannel radio engineering system on FPGA is presented (Fig. 2).

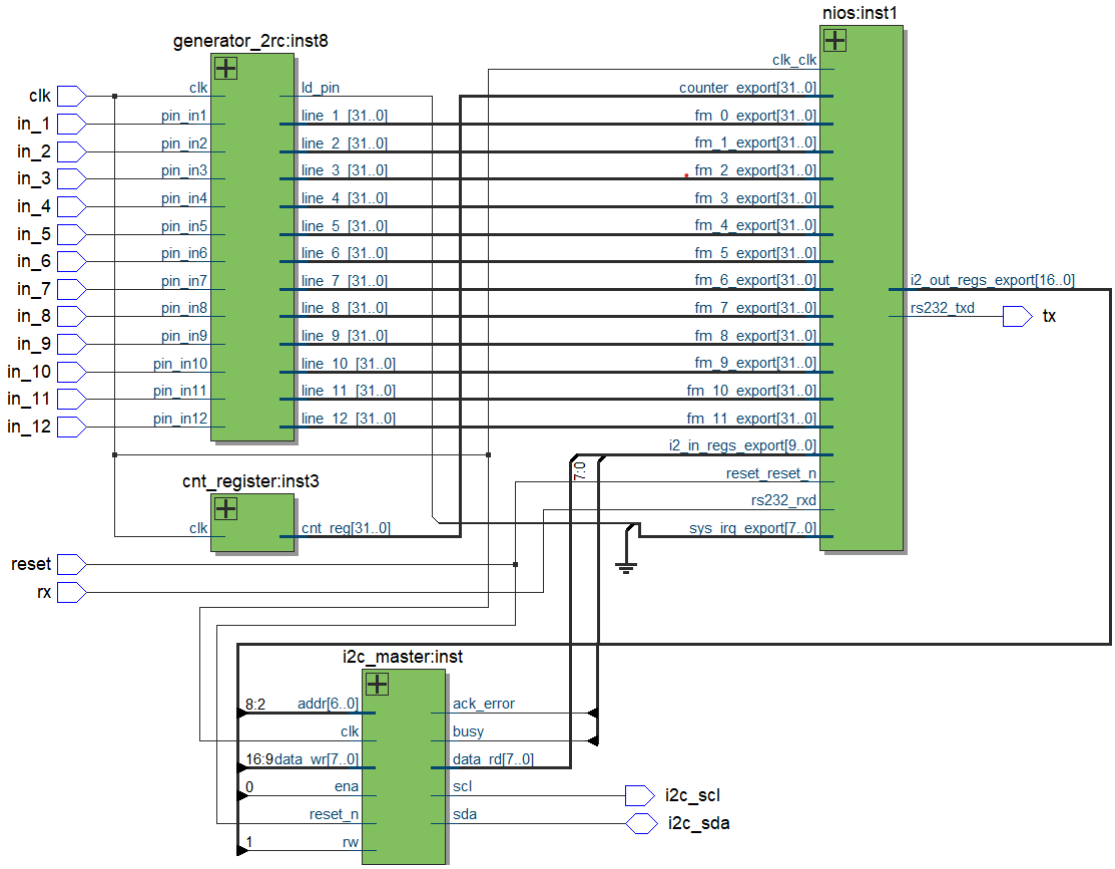


Fig. 1. A multichannel frequency meter scheme using the NIOS II core with I2C protocol support

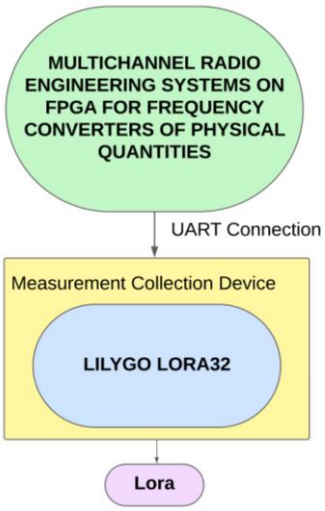


Fig. 2. Block diagram of the measurement collection system with FPGA

This system is intended for frequency converters of physical quantities. The main goal was to enable preliminary processing and wireless data transmission via the LoRa communication channel. The development environment was set up using VS Code and PlatformIO, a UART driver was created for reading packets, along with algorithms for parsing, processing, and error checking, as well as methods for forming compact packets for further transmission. This paper emphasizes the importance of using FPGA to ensure high measurement accuracy and demonstrates the advantages of integrating the Lilygo LORA32 module into multichannel measurement systems.

In [11], a prototype of a measurement collection system using a hub based on the LilyGo LoRa32 module for gathering data from multichannel radio engineering systems on FPGA is presented (Fig. 3).

These systems are designed for frequency converters of physical quantities. The main achievements include:

1. Integration of CBOR as a data transmission format, enabling a compact and efficient way to transmit measurements while minimizing delays and data volume.
2. Development and implementation of a LoRa hub, capable of consolidating data from multiple collection devices into a central point. This hub ensures reliable data collection even in systems with a large number of sensors.
3. Capability of using WiFi and Bluetooth protocols to transmit data from the hub to external servers or mobile devices, allowing the system to scale for different use cases.
4. Implementation of a multi-level packet formation algorithm, including adding hub information to the transmitted data. This simplified information processing on server-side applications.

Additionally, the paper details the system's hardware architecture, configurations, and main software development stages, such as the creation of a packet formation and decoding algorithm and the development of an asynchronous data transmission mechanism via LoRa (Fig. 4). Consequently, [8] marks a significant step forward in the development of wireless IoT systems based on FPGA, combining high performance with energy-efficient data transmission capabilities in remote environments. It lays the groundwork for further improvements in platforms for monitoring and controlling physical parameters.

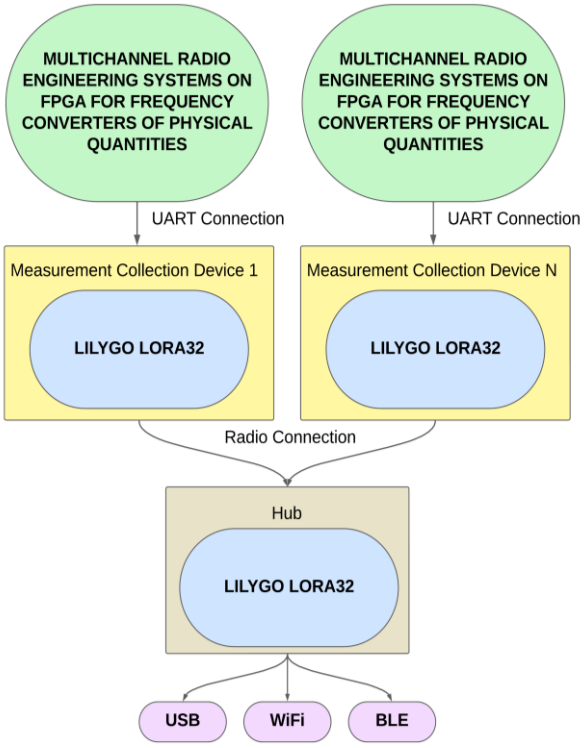


Fig. 3. Block diagram of the measurement collection and transmission system using the LoRa protocol

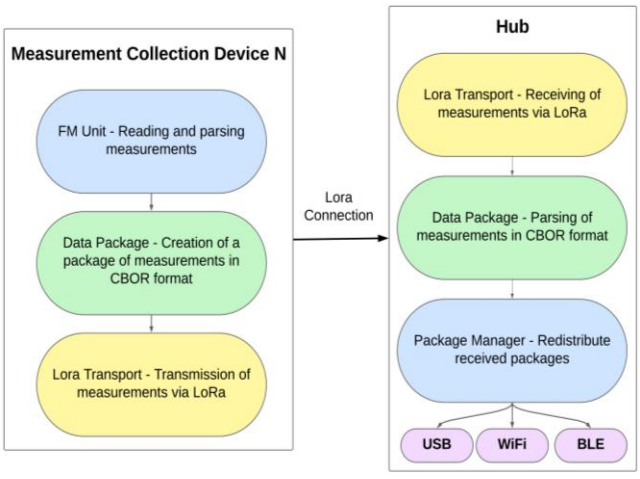


Fig. 4. Measurement processing sequence

2. Formulation of the problem

The aim of this work is to create a universal IoT system based on a multichannel radio engineering platform using FPGA for frequency converters of physical quantities. The system must ensure reliable data collection, preliminary processing, and secure transmission through modern wireless communication channels such as LoRa, WiFi, and Bluetooth.

To achieve this goal, the following key tasks are planned:

1. Defining the system structure and key components: Develop an architecture for the IoT system that includes a server, a mobile application, and data collection nodes, with detailed clarification of the functions of each component and mechanisms for their interaction via REST API.
2. Development of a LoRa hub: Enable the collection and centralization of data from numerous measurement devices, integrating the capability for subsequent transmission via WiFi.
3. Integration of the server-side component: Implement a REST API for receiving, storing, and processing data in CBOR format, utilizing the HTTPS protocol to ensure data confidentiality, authenticity, and integrity.
4. Development of a mobile application: Provide a tool for convenient real-time access to measurements, allowing for data visualization, analysis, and monitoring of the IoT system's status.
5. Optimization of energy efficiency: Leverage FPGA and LoRa technologies to achieve high performance with minimal energy consumption.

The proposed system should combine the advantages of high-performance FPGA computing platforms with modern wireless data transmission technologies, ensuring high accuracy, scalability, and energy efficiency.

3. Structure and key components of the IoT system

Before describing the implementation of the IoT system, it is important to provide a general understanding of its structure. To achieve this, the block diagram shown in Fig. 3 has been updated. Several new components have been developed in the enhanced system, significantly expanding its functionality. The server acts as a centralized platform for data processing and storage. It receives packets through the REST API [6], unpacks them in CBOR format, stores them in an SQLite database [12], and provides users access to the information. The server also processes requests from mobile devices, providing them with information about available hubs, measurement devices, and the measurements themselves. This architecture ensures system scalability and effective management of large volumes of data.

Mobile devices integrate into the system as tools for convenient access to stored information. They interact with the server via the REST API, making requests to obtain lists of hubs, measurement devices, and measurements. The retrieved data is presented to users in an intuitive format, enabling real-time monitoring and analysis of results. By supporting automatic data updates, mobile devices ensure continuous access to current information (Fig. 5).

Additionally, we will consider the main components through which the data flows, starting from its acquisition on the FPGA to its display on the mobile device. To illustrate this, the block diagram shown in Fig. 4 has been updated. On the server, new blocks have been added to process the data after it is received through the REST API. The first stage involves the "CBOR Parser," which unpacks compact messages in CBOR format, extracting key information for further storage. The next stage is storing the processed data in the "SQLite DB – table measurements," which efficiently organizes access to the information. Interaction with mobile devices is implemented via the REST API. Before obtaining measurements, it is necessary to retrieve information about the available hubs and connected measurement devices, after which a request is sent to obtain the data, which is then displayed on the screen (Fig. 6).

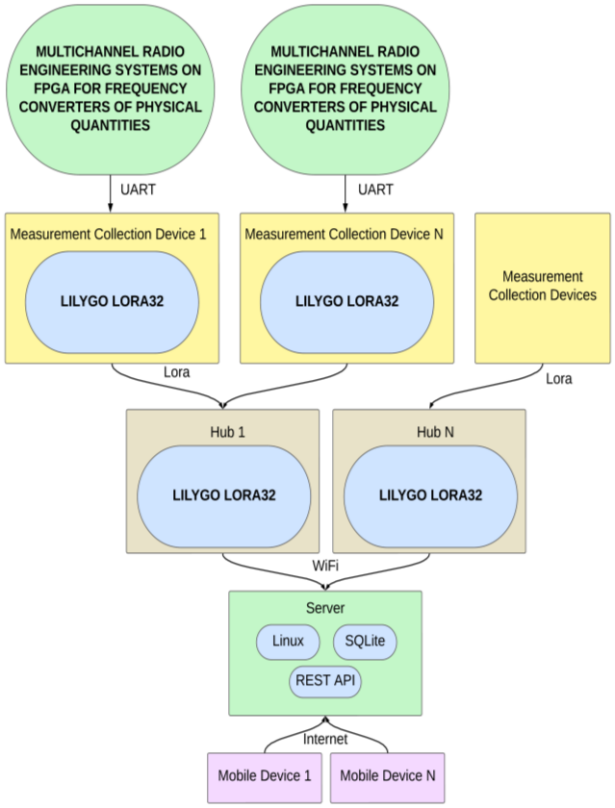


Fig. 5. Block diagram of the IoT system for measurement collection, transmission, and analysis

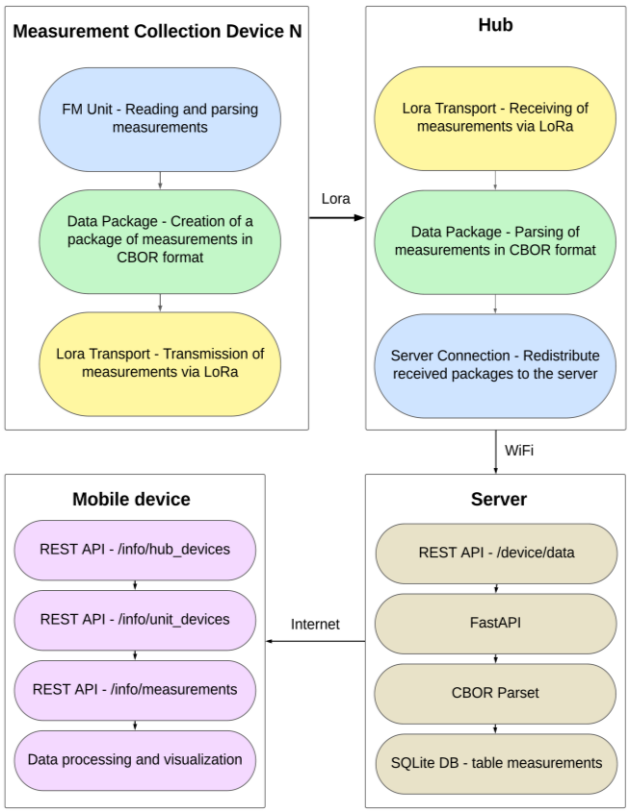


Fig. 6. Sequence of measurement processing in the IoT system

4. Development of the key system components

The first step in implementing the described IoT system is ensuring the hub's access to the internet to enable the transmission of accumulated data. To achieve this, the hub's implementation was expanded, and support for data transmission to a designated server via WiFi was added.

A simple interface was created to allow:

1. Connecting to a WiFi access point.
2. Establishing a secure connection with the server.
3. Sending measurements to the server.

```
[1] namespace ServerConn {
[2]  bool begin();
[3]  bool isConnected();
[4]  void sendData(const std::vector<uint8_t> &data);
[5] }
```

When *ServerConn::begin()* is called, the WiFi module is initialized and connected to the specified access point, with auto-reconnection to the access point enabled in case of connection loss.

```
[1] WiFi.begin(WIFI_SSID.c_str(), WIFI_PASSWORD.c_str());
[2] WiFi.setAutoReconnect(true);
```

A queue of 50 messages is created to store the measurements that will be sent to the server.

```
[1] _payloadQueue = xQueueCreate(PAYLOAD_QUEUE_SIZE,
      sizeof(Payload));
```

The final step in the *ServerConn::begin()* function is creating a separate thread on core 0, which allows parallel reception of measurements via the LoRa protocol and transmission of these measurements to the server without mutual blocking.

```
[1] xTaskCreatePinnedToCore(serverTask, "server", 10000,
      NULL, 5, &_serverTaskHandle, 0);
```

To better understand, the *_payloadQueue* queue is created to manage the data transfer between different parts of the program, specifically between the part that receives the measurements and the part that sends them to the server. From an architectural perspective, using a queue has several advantages:

1. Asynchrony: The queue separates the data reception process from the transmission process. This means the program can continue performing other tasks while the data awaits transmission in the queue.
2. Buffering: The queue acts as a buffer, allowing data to accumulate if the rate of data generation exceeds the rate of transmission.
3. Thread Safety: Using a queue ensures safe data exchange between different threads in a multithreaded environment such as FreeRTOS [6]. *ServerConn::isConnected()* function returns the connection status of the WiFi access point:

```
[1] WiFi.status() == WL_CONNECTED
```

The *ServerConn::sendData()* function creates a copy of the data to be sent and writes it to the queue, where it will be processed by the "server" thread at the earliest opportunity.

```
[1] Payload payload;
[2] payload.data = new uint8_t[data.size()];
[3] payload.size = data.size();
[4] memcpy(payload.data, data.data(), data.size());
[5] if (xQueueSend(_payloadQueue, &payload,
      PAYLOAD_QUEUE_TIMEOUT) != pdPASS) {
[6]  LOGE(TAG, "Failed to send payload to queue");
[7]  delete[] payload.data;
[8]  return;
[9] }
```

This concludes the description of the ServerConn interface, and we can move on to the internal implementation, namely the serverTask() function. The serverTask() function is a key part of the IoT system implementation, responsible for processing the _payloadQueue data queue and transmitting it to the server via the HTTPS protocol [14]. This frates in a dedicated thread,

ensuring asynchronous and efficient data transmission without blocking other processes.

At each iteration, the function checks the WiFi connection status and the presence of data in the queue. If the WiFi connection is unavailable or the queue is empty, the function pauses for 1 ms, allowing other threads to continue working.

```
[1] if (!isWifiConnected() ||
      xQueueReceive(_payloadQueue, &payload, 0) !=
      pdPASS) {
[2]  vTaskDelay(1);
[3]  continue;
[4] }
```

After retrieving data from the queue, the WiFiClientSecure is initialized, setting up the CA_CERT certificate required to establish a secure connection.

```
[1] WiFiClientSecure client;
[2] client.setCACert(CA_CERT);
```

A connection to the server is established via HTTPS. The path *"https://server_ip:8080/device/data"* is used, which includes the server's IP address, port, and API endpoint. The HTTP request includes a header *"Content-Type: application/octet-stream"*, indicating the binary format of the transmitted data. The data is sent using the POST [7] method. The server's response code is stored in the *httpResponseCode* variable, and appropriate messages are logged to the terminal based on the result.

```
[1] HTTPClient https;
[2] if (https.begin(SERVER_DATA_PATH.c_str())) {
[3]  https.addHeader("Content-Type", "application/octet-
      stream");
[4]  int httpResponseCode = https.POST(payload.data,
      payload.size);
[5]  LOGI(TAG, "HTTP Response: %d", httpResponseCode);
[6]  if (httpResponseCode == HTTP_CODE_OK) LOGI(TAG,
      "POST Success");
[7]  else LOGE(TAG, "POST Failed, Error: %d (%s)",
      httpResponseCode,
      https.errorToString(httpResponseCode).c_str());
[8]  https.end();
[9] }
```

After the data transfer is completed, the memory allocated for storing the data in the queue is released.

```
[1] delete[] payload.data;
```

The *serverTask()* function plays a key role in ensuring reliable and secure data transmission in the IoT system, making it adaptable to the demands of modern network solutions. To secure data transmission within the system, the HTTPS (Hypertext Transfer Protocol Secure) protocol is used. The main security aspects are implemented as follows:

1. Data Encryption: Data transmitted between the device and the server is encrypted using SSL/TLS, ensuring confidentiality and protection against interception by malicious actors.
2. Certificate Verification: The system uses a trusted certificate (CA_CERT), ensuring the server's authenticity. This guarantees that data is sent to a legitimate server rather than a counterfeit node.
3. Data Integrity: HTTPS includes mechanisms for verifying data integrity, ensuring that transmitted data has not been altered during transport.
4. Two-Way Authentication: The server's certificate confirms its authenticity. If necessary, client authentication mechanisms can be added for an even higher level of security.
5. Protection Against MITM [15] Attacks: Using a secure protocol with certificate verification significantly reduces the risk of attacks where an attacker could intercept or alter data.

These measures ensure reliable and secure measurement transmission, preserving the confidentiality, integrity, and authenticity of the information within the system.

The next step is to set up the server and implement the REST API. To do this, we had to:

1. Rented cloud computing on the AWS platform [2].
2. Create an *Instance* of the *t2.micro* type with the Linux operating system [5].
3. Provided access to port 8080 for incoming requests, which will be used by the HTTP server.
4. Create a certificate that will be used to establish a secure communication channel between the server and all other devices: *sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout selfsigned.key -out selfsigned.crt*.

After setting up the server, the application launch environment was configured and the SQLite database was installed.

```
[1] sudo apt install sqlite3
[2] sudo apt install python3-pip
[3] sudo apt install python3-venv
[4] python3 -m venv iotenv
[5] source iotenv/bin/activate
[6] pip install -r requirements.txt
```

To implement the server-side component, the Python programming language was used [17]. For the REST API, the FastAPI framework [13] was chosen, providing a convenient way to create an interface for handling requests and supporting asynchronous operations for efficient processing of incoming data. The FastAPI application was deployed using uvicorn [16], a high-performance ASGI server. The use of uvicorn allows the server to handle multiple requests simultaneously, maintaining low latency and high performance even with a large number of connections. The primary purpose of the server-side component is to receive measurements from hubs, store this data in a database, and provide access to it via the API.

```
[1] # Create the FastAPI app
[2] app = FastAPI()
[3] # Run the server
[4] uvicorn.run(
[5]  "server:app",
[6]  host="0.0.0.0",
[7]  port=8080,
[8]  ssl_certfile="ssl_keys/selfsigned.crt",
[9]  ssl_keyfile="ssl_keys/selfsigned.key"
[10] )
```

When the server starts, the SQLite database is initialized, and a *measurements* table is created if it does not already exist.

```
[1] # SQLite connection
[2] # Table: measurements
[3] # Columns: id(INTEGER), db_timestamp_ms(INTEGER),
      hub_mac_addr(TEXT), unit_mac_addr(TEXT),
      pkg_id(TEXT), pkg_timestamp_ms(INTEGER),
      fm_data(TEXT)
[4] db_conn = sqlite3.connect("data.db")
[5] db_conn.execute(table_description)
[6] db_conn.commit()
```

The structure of the REST API includes several interfaces: */device/data*, */info/hub_devices*, */info/unit_devices*, */info/measurements*. Each of these interfaces has been implemented, and we will describe them in more detail.

/device/data – handles HTTP POST requests from hubs. It receives measurement packets in CBOR format, decodes them using the cbor2 module [1], and stores them in the SQLite database. The saved data includes the MAC address of the collection device, the MAC address of the hub, the packet ID, the timestamp, and the measurements.

```
# Receive the payload as binary
payload = await request.body()
# Decode CBOR payload
decoded_payload = cbor2.loads(payload)
unit_mac_addr_str = ":".join(f"{byte:02X}" for byte
    in decoded_payload[DP_UNIT_MAC_ADDR_ID])
hub_mac_addr_str = ":".join(f"{byte:02X}" for byte in
    decoded_payload[DP_HUB_MAC_ADDR])
# Save the data to the database
db_conn.execute("INSERT INTO measurements
    (db_timestamp_ms, hub_mac_addr, unit_mac_addr,
    pkg_id, pkg_timestamp_ms, fm_data) VALUES (?, ?, ?,
    ?, ?, ?)", parameters)
db_conn.commit()
```

/info/hub_devices – handles HTTP GET requests to retrieve a list of unique MAC addresses of hubs registered in the database. This allows clients to obtain a list of available hubs for further interaction.

```
hub_devices = db_conn.execute("SELECT DISTINCT
    hub_mac_addr FROM measurements").fetchall()
```

/info/unit_devices – this interface returns a list of collection devices connected to a specific hub. The request includes the hub's MAC address, after which the server executes an SQL query to retrieve all devices that have interacted with it.

```
if hub_mac_addr:
    unit_devices = db_conn.execute("SELECT DISTINCT
    unit_mac_addr FROM measurements WHERE hub_mac_addr
    = ?", parameters).fetchall()
else:
    unit_devices = db_conn.execute("SELECT DISTINCT
    unit_mac_addr FROM measurements").fetchall()
```

/info/measurements – this interface is used to retrieve measurements. The request can include parameters such as the hub's MAC address, the collection device's MAC address, or the number of recent records. The server processes the request and returns a response in JSON format [4].

```
if hub_mac_addr and unit_mac_addr:
    measurements = db_conn.execute("SELECT
    pkg_timestamp_ms, fm_data FROM measurements WHERE
    hub_mac_addr = ? AND unit_mac_addr = ? ORDER BY id
    DESC LIMIT ?", parameters).fetchall()
elif hub_mac_addr:
    measurements = db_conn.execute("SELECT
    pkg_timestamp_ms, fm_data FROM measurements WHERE
    hub_mac_addr = ? ORDER BY id DESC LIMIT ?",
    parameters).fetchall()
elif unit_mac_addr:
    measurements = db_conn.execute("SELECT
    pkg_timestamp_ms, fm_data FROM measurements WHERE
    unit_mac_addr = ? ORDER BY id DESC LIMIT ?",
    parameters).fetchall()
else:
    measurements = db_conn.execute("SELECT
    pkg_timestamp_ms, fm_data FROM measurements ORDER
    BY id DESC LIMIT ?", parameters).fetchall()
```

To ensure a secure connection, the server uses an SSL certificate generated during the setup process. All HTTP requests are performed over HTTPS, providing secure data transmission between the server and other system components. The SQLite database serves as the storage for the received measurements. The main database table includes fields such as the reception timestamp, hub MAC address, collection device MAC address, packet ID, and measurement data. Interaction with the database is handled using the *sqlite3* library.

Thus, the server-side implementation ensures centralized storage and processing of data, as well as convenient access to it through the REST API. This enables the integration of the server with the system and mobile devices, ensuring scalability and efficiency.

The final step in the development of the IoT system is creating mobile software that enables the visualization and analysis of measurements. It was decided to use Kotlin [9] and the Android Studio development environment for implementation. The implementation is divided into two parts: one for receiving data from the server and the other for displaying the received data.

To ensure secure interaction with the server, the *OkHttp* library was used, which supports encrypted and certified HTTP requests. The primary purpose of this part is to check the server's availability and retrieve data via the REST API. To establish a secure connection, an SSL certificate stored in the application's resources is used. During client initialization, the certificate in X.509 format is loaded and added to the *KeyStore*. Then, using the *TrustManagerFactory* and *SSLContext*, a secure SSL context is created, which is used to handle all HTTPS requests.

```
[1] val cf = CertificateFactory.getInstance("X.509")
[2] val certInput: InputStream =
      context.resources.openRawResource(R.raw.selfsigned)
[3] val ca: Certificate =
      cf.generateCertificate(certInput)
      certInput.close()
[4] val keyStore =
      KeyStore.getInstance(KeyStore.getDefaultType()).appl
      y {
[5]  load(null, null)
[6]  setCertificateEntry("ca", ca)
[7] }
[8] val tmf =
      TrustManagerFactory.getInstance(TrustManagerFactory.
      getDefaultAlgorithm()).apply {
[9]  init(keyStore)
[10] }
[11] val sslContext = SSLContext.getInstance("TLS").apply
      {
[12]  init(null, tmf.trustManagers, null)
[13] }
```

The *NetworkHelper* class provides essential methods for server interaction. The *isServerReachable* method is used to check server availability. It performs an HTTP HEAD request and analyzes the server's response code, allowing the application to promptly notify the user about the connection status.

```
[1] val request =
      Request.Builder().url(url).head().build()
[2] val response: Response =
      client.newCall(request).execute()
[3] response.isSuccessful
```

The get method is used to retrieve data from the server. It supports the addition of query parameters to the request URL, enabling flexible interaction with the REST API. The method processes the server's response, including status code, headers, and body. Upon a successful request, it returns the received data as a text string.

```
[1] val httpUrlBuilder =
      url.toHttpUrlOrNull()?.newBuilder()
[2] queryParams?.forEach { (key, value) ->
[3]  httpUrlBuilder?.addQueryParameter(key, value)
[4] }
[5] val finalUrl = httpUrlBuilder?.build().toString()
[6] val request = Request.Builder().url(finalUrl).build()
[7] val response: Response =
      client.newCall(request).execute()
[8] val responseBody = response.body?.string()
      responseBody
```

This part of the mobile application implements basic functionality for secure data transmission between the server and the mobile device, forming the foundation for further analysis and visualization of the received measurements.

The primary purpose of the second part is the interactive display of data received from the server and providing a user-friendly interface. The implementation includes monitoring the server connection status, updating device lists (hubs and collection devices), and dynamically displaying measurements in a table.

The main class, *MainActivity*, is used for interactivity. It initializes interface elements:

1. A text field to display the connection status (*connTextView*)
2. Dropdown menus for selecting hubs (*hubSpinner*) and collection devices (*unitSpinner*)
3. A table for displaying measurements (*TableLayout*).

All elements are dynamically updated based on the data received from the server. Connection monitoring is implemented through a coroutine that periodically queries the server. The *startMonitoringServer* method checks the server's availability via the API and retrieves lists of available hubs (*/info/hub_devices*) and collection devices (*/info/unit_devices*). If the server is unavailable, the connection status is updated in the interface, and device lists are cleared. If the connection is successful, the data is loaded and displayed in the respective interface elements.

```
[1] if(!networkHelper.isServerReachable(serverAddr)
    ) {
[2]  withContext(Dispatchers.Main) {
[3]   updateConnectionStateUI(false)
[4]  }
[5]  delay(1000)
[6]  Continue
[7] } else {
[8]  withContext(Dispatchers.Main) {
[9]   updateConnectionStateUI(true)
[10]   }
[11]  }
[12]  val hubDevRsp =
     networkHelper.get(apiHubDevices)
[13]  if (hubDevRsp == null) {
[14]   continue
[15]  }
[16]  val hubMacAddresses =
     mutableListOf(anyItemInTheList) +
     parseMacAddresses(hubDevRsp)
[17]  withContext(Dispatchers.Main) {
[18]   updateSpinner(hubSpinner,hubMacAddresses)
[19]  updateSpinner(unitSpinner,listOf(anyItemInThe
     List))
[20]  }
[21]  var queryParams = emptyMap<String, String>()
[22]  if (selectedHubMac != anyItemInTheList) {
[23]  queryParams = mapOf("hub_mac_addr" to
     selectedHubMac)
[24]  }
[25]  val unitDevRsp =
     networkHelper.get(apiUnitDevices,
     queryParams)
[26]  if (unitDevRsp == null) {
[27]   continue
[28]  }
[29]  val unitMacAddresses =
     mutableListOf(anyItemInTheList) +
     parseMacAddresses(unitDevRsp)
[30]  withContext(Dispatchers.Main) {
[31]   updateSpinner(unitSpinner, unitMacAddresses)
[32]  }
```

NetworkHelper performs HTTP GET requests to retrieve device lists and measurements. Device lists are updated using Spinner adapters, and measurements are added to the table using the *clearTableExceptHeaders* and *updateTable* methods. Measurement data is requested via the API (*/info/measurements*) with parameters such as the hub's MAC address, the collection device's MAC address, or the number of recent measurements. The received data is processed and displayed in the table with corresponding keys and values. The table dynamically shows measurements from the IoT system, and it supports automatic updates every 3 seconds, ensuring data freshness for the user. Updates are performed asynchronously to avoid interface delays.

```
[1]  val queryParams = emptyMap<String,
       String>().toMutableMap()
[2]  if (selectedHubMac != anyItemInTheList) {
[3]   queryParams += mapOf("hub_mac_addr" to
       selectedHubMac)
[4]  }
[5]  if (selectedUnitMac != anyItemInTheList) {
[6]   queryParams += mapOf("unit_mac_addr" to
       selectedUnitMac)
[7]  }
[8]  queryParams += mapOf("last_n_measurements" to
       measurementsCnt.toString())
[9]  val measurements =
       parseMeasurements(networkHelper.get(apiMeasurements,
       queryParams) ?: "")
[10] withContext(Dispatchers.Main) {
[11]  clearTableExceptHeaders(table)
[12]  updateTable(table, measurements)
[13] }
```

Thus, the second part of the mobile application implements a full-fledged graphical interface for interacting with the IoT system, providing users with real-time data access and an intuitive way to view measurements. This completes the main functionality of the application, combining data retrieval and visualization, as shown in Fig. 7, 8.

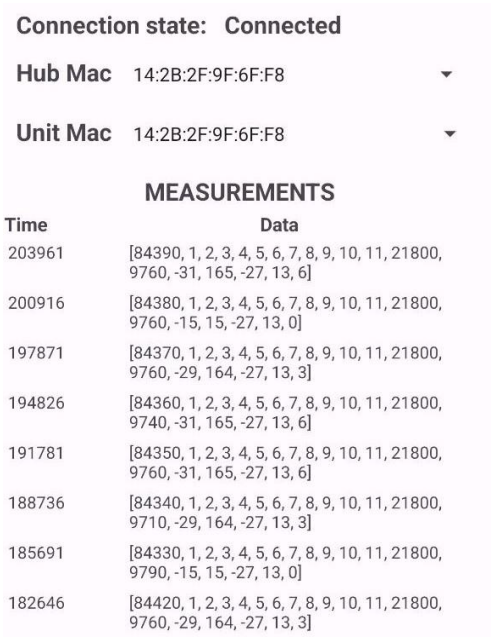


Fig. 7. Mobile application interface layout

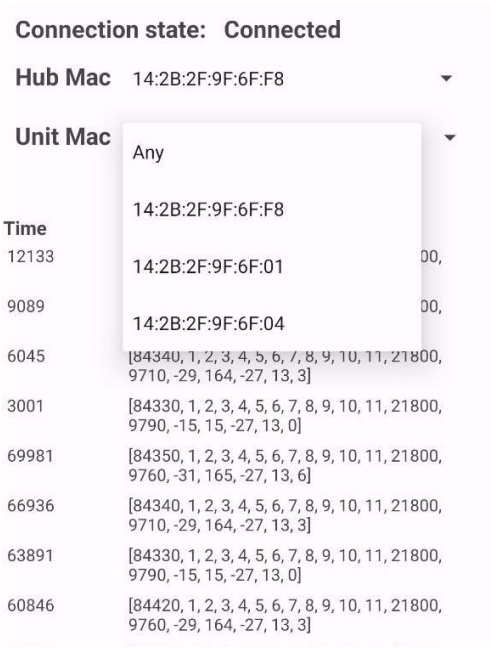


Fig. 8. Method of selecting a measurement device

5. Conclusions

This study presents the development of an IoT system for collecting, processing, and transmitting measurements from multichannel radio engineering systems based on FPGA, offering a modern approach to addressing measurement technology challenges. The primary goal was to create a universal, scalable, and energy-efficient system capable of operating in remote and hard-to-reach areas. Through the integration of the Lilygo LORA32 module and modern wireless communication protocols such as LoRa and WiFi, a multi-layer architecture was developed that combines high performance with flexibility.

1. The first step involved expanding the functionality of the LoRa hub to enable measurement transmission to the server via WiFi. The use of WiFi ensured high-speed data transfer to a centralized server, playing a key role in the system.

2. On the server side, a REST API was implemented using the FastAPI framework, providing flexibility in data management. The server processes incoming packets in CBOR format and stores the decoded information in an SQLite database. Furthermore, the server provides access to the measurements through multiple API interfaces, enabling seamless integration with other systems and devices. The implementation of the secure HTTPS protocol ensures protected data transmission, preventing interception or modification.

3. The final stage was the development of a mobile application for displaying and analyzing measurements. Using Kotlin and Android Studio, a user-friendly interface was created, allowing users to monitor connection status, select collection devices and hubs, and retrieve and analyze measurements in real time. The mobile application supports dynamic data updates, ensuring the information remains current and accessible to users.

4. The developed IoT system demonstrates high accuracy and reliability, making it a universal solution for various applications, including industrial automation, environmental monitoring, smart agriculture, and other IoT scenarios. By combining FPGA with wireless modules, the system achieves high energy efficiency, enabling its use in challenging conditions with limited access to energy resources.

Future advancements of this system include improving data processing algorithms, increasing hub throughput, and integrating new features for real-time monitoring. This paves the way for the development of innovative IoT solutions capable of meeting the demands of modern industrial, environmental, and infrastructure systems. Thus, the developed IoT platform lays the foundation for further innovations in measurement technology and the Internet of Things, unlocking new opportunities for efficient technology utilization across various sectors.

References

[1] Agronholm A.: cbor2: Python CBOR (de)serializer with extensive tag support. GitHub Repository, 2024 [https://github.com/agronholm/cbor2].

[2] Amazon Web Services: Overview of Amazon Web Services. AWS Whitepaper, 2023 [https://docs.aws.amazon.com].

[3] Barry R.: Mastering the FreeRTOS Real Time Kernel – A Hands-On Tutorial Guide. Real Time Engineers Ltd., 2016.

[4] Bray T.: The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.

[5] Corbet J., Rubini A., Kroah-Hartman G.: Linux Device Drivers. 3rd ed., O'Reilly Media, 2005.

[6] Fielding R.: Architectural Styles and the Design of Network-Based Software Architectures (Chapter 5). Ph.D. dissertation. University of California, Irvine 2000 [https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm].

[7] Fielding R., Reschke J.: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014.

[8] IEEE Standard for Information Technology: Telecommunications and Information Exchange between Systems – Local and Metropolitan Area Networks – Specific Requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications [https://web.archive.org/web/20220206191751/https://standards.ieee.org/ieee/802.11/7028/].

[9] JetBrains: Kotlin Programming Language. 2024 [https://kotlinlang.org/].

[10] Osadchuk O. V., Osadchuk Y. O., Skoshchuk V. K.: Improvement of a Multichannel Radio Engineering System on FPGA for Frequency Converters of Physical Quantities with Support for Digital Sensors. Measurement and Computing Equipment in Technological Processes No. 2, 2023, 72–82 [https://doi.org/10.31891/2219-9365-2023-74-10].

[11] Osadchuk O., Skoshchuk V.: Wireless FPGA-Based Data Acquisition System for Frequency Converters of Physical Quantities Using LoRa. Herald of Khmelnytskyi National University. Technical Sciences 347(1), 2025, 375–386 [https://doi.org/10.31891/2307-5732-2025-347-51].

[12] Owens M.: The Definitive Guide to SQLite. Apress, 2010.

[13] Ramírez S.: FastAPI: Modern, Fast (High-Performance) Web Framework for Building APIs with Python 3.6+. 2018 [https://fastapi.tiangolo.com/].

[14] Rescorla E.: HTTP Over TLS. RFC 2818, May 2000 [https://www.rfc-editor.org/rfc/rfc2818].

[15] Rescorla E., Korver B.: Guidelines for Writing RFC Text on Security Considerations. RFC 3552, July 2003.

[16] Uvicorn Documentation: An ASGI Web Server for Python. 2023 [https://www.uvicorn.org/].

[17] Van Rossum G., Drake F. L.: Python Tutorial. Python Software Foundation, 2001.

Prof. Oleksandr Osadchuk
e-mail: osadchuk.av69@gmail.com

Doctor of Technical Sciences, professor, Head of the Department of Information Radioelectronic Technologies and Systems of Vinnitsia National Technical University, Academician of the Academy Metrology Ukraine. Author of over 950 publications, including 35 monographs, 18 textbooks, 320 patents for inventions, more than 500 scientific articles in professional journals, of which 75 are in the scientometric databases Scopus and Web of Science.

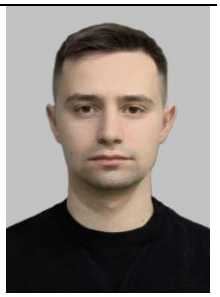
https://orcid.org/0000-0001-6662-9141

Ph.D. Iaroslav Osadchuk
e-mail: osadchuk.j93@gmail.com

Candidate of Technical Sciences, associate professor of Information Radioelectronic Technologies and Systems of Vinnytsia National Technical University. Author of more than 250 publications, including 10 monographs, 60 patents for inventions and more than 130 scientific articles in professional journals, of which 32 are in scientometric databases Scopus and Web of Science.

https://orcid.org/0000-0002-5472-0797

M.Sc. Valentyn Skoshchuk
e-mail: skoschuk999@gmail.com



Postgraduate student at the Department of Information Radioelectronic Technologies and Systems of Vinnytsia National Technical University. Author of 13 publications, including 1 patent for an invention and 4 scientific articles in professional journals.

https://orcid.org/0009-0008-9762-2397