Instituto Tecnológico
y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

Especialidad en Sistemas Embebidos

OCELOT: A USB-Based Industrial I/O Board for Cash Deposit Systems

---

TRABAJO RECEPCIONAL que para obtener el **DIPLOMA** de
ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: **OSCAR ALFREDO MERCADO RICO**

Asesor: **M. Sc. MARTÍN ALONSO SINSEL DUARTE**

Tlaquepaque, Jalisco. julio de 2026

OCELOT: A USB-Based Industrial I/O Board for Cash Deposit Systems

Oscar Alfredo Mercado Rico
Department of Electronics, Systems, and Computer Science
Western Institute of Technology and Higher Education (ITESO)
Tlaquepaque, Jalisco, México
alfredo.mercado@iteso.mx

Martín Alonso Sinsel Duarte
Department of Electronics, Systems, and Computer Science
Western Institute of Technology and Higher Education (ITESO)
Tlaquepaque, Jalisco, México
msinseld@iteso.mx

***Abstract*— Commercial off-the-shelf (COTS) I/O boards offer a convenient solution for embedded applications; however, they often incorporate unnecessary features, increase hardware dependencies, and complicate software integration. To address these limitations, this paper presents the design and implementation of the OCELOT I/O board, a custom embedded platform developed to replace the commercial Numato I/O board currently used in the RC-200 and RC-300 cash management systems. The proposed solution combines dedicated hardware with a layered RTOS-based firmware architecture. The firmware adopts an event-driven architecture that decouples communication, command processing, and hardware control, providing a modular and scalable software design. It implements a communication and control framework comprising a JSON-RPC host interface, dedicated drivers for digital inputs, digital outputs, and RGB LED control, together with a USB UF2 bootloader that enables firmware updates through a standard USB Mass Storage Class (MSC) interface. Functional validation verified the correct operation of the communication subsystem, peripheral drivers, and firmware update mechanism. The proposed platform demonstrates that a purpose-built embedded solution can successfully replace the previous commercial hardware while simplifying software integration and supporting future firmware expansion.**

Keywords*— *Embedded systems, Industrial I/O, Event-driven architecture, JSON-RPC, USB bootloader.

I. INTRODUCTION

Venteks is a company focused on the development of banking, point-of-sale (POS), and cash-handling solutions. Its Vsafe product line comprises several systems designed to automate payment processing, cash deposit operations, and cash recycling while enhancing security and operational efficiency in business environments [1]. Among these products are the RC-200 and RC-300, two high-volume cash management systems. These systems employ a PC-based controller that communicates with several embedded devices, including an I/O board responsible for monitoring sensors, controlling vault locks, and managing status indicators [2].

The current RC-200 and RC-300 systems employ the Numato I/O board, a commercial off-the-shelf solution that provides digital I/O and analog input functionality through USB serial communication and a command-based software interface [3]. However, this solution introduces challenges related to hardware availability, increased procurement costs, functionality not required by the target application, and communication interfaces that complicate software integration. To address these limitations, the OCELOT I/O board was developed as a dedicated embedded platform tailored to the requirements of the RC-200 and RC-300 systems. The proposed platform replaces the previous commercial solution with an application-specific hardware and firmware architecture designed to simplify software integration, reduce hardware dependency, and support future system evolution. This paper describes the design, implementation, and evaluation of the proposed platform.

The remainder of this paper is organized as follows: Section II describes the OCELOT hardware platform and firmware architecture. Section III presents the implementation results and future work. Finally, Section IV concludes the paper.

II. SYSTEM ARCHITECTURE

This section presents the system architecture of the OCELOT I/O board. It provides an overview of the hardware platform and its main interfaces, followed by the layered firmware architecture supporting host communication, event handling, and hardware control. The console, event handler, and driver layers are then described, highlighting their roles and interactions within the overall system.

A. OCELOT Hardware Platform

The OCELOT custom hardware was designed to meet the requirements of the RC-200 and RC-300 systems. The board´s main features include eight digital inputs with low-pass filtering, six relay-driven digital outputs, and a USB Type-B interface for communication with the host controller. These features provide functional equivalence to the previously used Numato I/O board.

In addition, OCELOT incorporates several auxiliary features, including a configurable LED output interface supporting SPI and PWM based devices, onboard EEPROM memory for non-volatile data storage, dedicated reset and boot buttons, status indication LEDs, expansion interfaces for I2C and RS-232 peripherals, and an SWD connector for firmware debugging and development [4]. *"Fig. 1"* presents an overview of the OCELOT hardware platform.

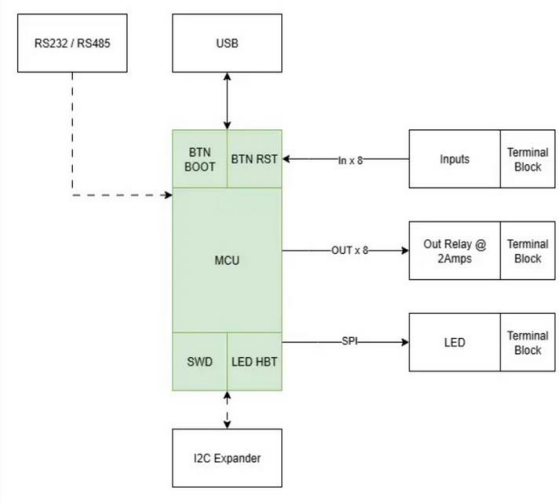

Fig 1. OCELOT hardware platform.

The OCELOT hardware platform is based on the STM32F411CCU6 microcontroller, which features an ARM Cortex-M4 core and provides the processing resources required for communication, I/O management, and real-time control. The device is supported by the STM32Cube development ecosystem, including hardware abstraction layer (HAL) drivers, middleware components, and compatibility with multiple real-time operating systems (RTOS) [5].

B. Firmware Architecture

The system firmware is developed in C using the STM32Cube development environment and HAL drivers. To support multitasking, real-time processing, and future scalability, the firmware integrates a real-time operating system (RTOS). The implementation is based on the FreeRTOS kernel due to its open-source nature, broad community support, simplicity, and compatibility with the STM32Cube ecosystem [6].

The firmware architecture is organized into five interconnected layers with clearly defined responsibilities: console, event handler, drivers, microcontroller peripherals, and hardware. The console layer provides the communication interface between the host system and the firmware. The event handler layer coordinates event processing and command routing. The driver layer implements the system functionalities and serves as an abstraction layer between the application logic and the underlying microcontroller peripherals and hardware platform.

Overall, the firmware follows an event-driven architecture in which host requests are translated into events that propagate through the software layers until the corresponding hardware action is executed. This layered approach provides a clear separation of responsibilities, centralizes event processing, and simplifies the integration of new functionalities [7]. *"Fig. 2"* and *"Fig. 3"* present an overview of the firmware architecture and the event flow among its layers.

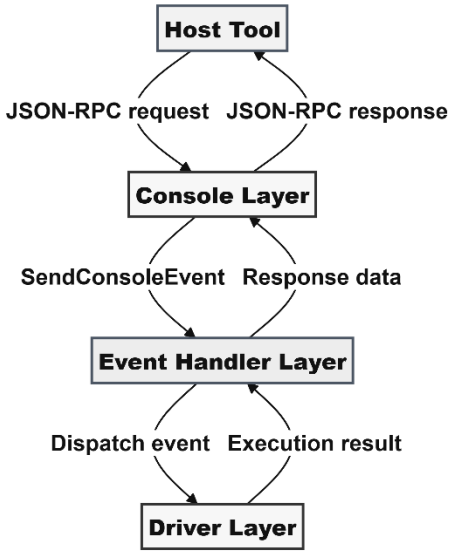

Fig 2. Firmware architecture.

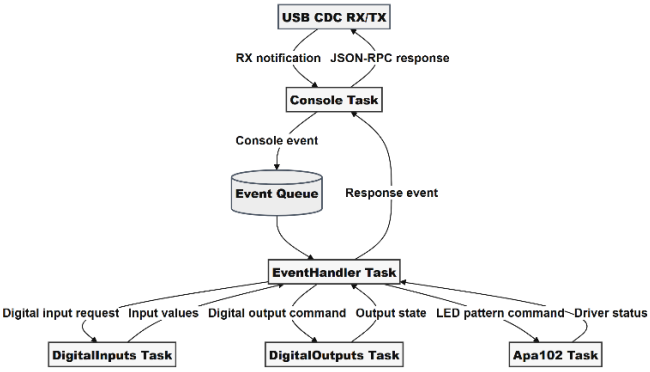

Fig 3. Firmware event flow.

C. Console Layer

The console layer manages the USB communication interface and serves as the entry point for host requests. Its main responsibilities include receiving incoming messages from the host, validating and parsing them according to the JSON-RPC request format, translating valid requests into internal processing events, and transmitting the corresponding responses back to the host system.

OCELOT implements its host communication interface over USB using the Communication Device Class (CDC), allowing the board to operate as a standard virtual serial port without requiring proprietary host-side drivers [8]. At the application layer, all requests and responses follow the JSON-RPC 2.0 format. Remote Procedure Call (RPC) is a communication model in which a host invokes a procedure on a remote system by sending the procedure identifier along with its arguments and receives the corresponding execution result in return [9]. JSON-RPC adopts this request–response model using JSON objects as the data exchange format, providing a lightweight and standardized interface between the host tool and the embedded firmware [10]. Each request contains the protocol version, a method name, associated parameters, and a request identifier, whereas each response

contains the protocol version, either a result or an error object, and the corresponding identifier. The supported OCELOT methods are grouped according to their associated functional domain, including digital inputs, digital outputs, LED control, and system-related operations, as summarized in Table I. The general JSON-RPC format and an example OCELOT request are illustrated in *"Fig. 4"* and *"Fig. 5"*, respectively.

Table I. OCELOT JSON-RPC requests.

Category	Method	Description
Digital Inputs	getInputs	Gets the current value of one or more digital inputs.
	getAllInputs	Gets the current value of all digital inputs.
	setDebounceTime	Configures the debounce time in milliseconds for digital input reading.
	getDebounceTime	Gets the current debounce time in milliseconds for digital input reading.
Digital Outputs	setOutputs	Sets the current value of one or more digital outputs.
	setAllOutputs	Sets the current value of all digital outputs.
	toggleOutputs	Toggles one or more digital outputs.
	toggleAllOutputs	Toggles all digital outputs.
	getOutputs	Gets the current value of one or more digital outputs.
	getAllOutputs	Gets the current value of all digital outputs.
LEDs	setLEDSegments	Configures one or more LED segments.
	getLEDSegments	Gets the configured LED segments.
	setLEDPattern	Configures the LED illumination pattern.
	getLEDStatus	Gets the current LED status.
System Reset	restart	Restarts the system.
System Information	getSystemInfo	Gets general system information.
Bootloader	enterBootloader	Executes the system bootloader.

Request

```
{
  "jsonrpc": "2.0",
  "method": "subtract",
  "params": [42, 23],
  "id": 1
}
```

Response

```
{
  "jsonrpc": "2.0",
  "result": 19,
  "id": 1
}
```

Fig 4. JSON-RPC general format.

Request

```
{
  "version": "1.0",
  "method": "setLEDPattern",
  "params": {
    "segments": [
      {
        "segmentId": 0,
        "pattern": "solid",
        "color": "blue",
        "duration": 10000
      },
      {
        "segmentId": 1,
        "pattern": "blink",
        "color": "red",
        "duration": 10000,
        "config": {
          "delay": 50
        }
      }
    ]
  },
  "id": 1
}
```

Response

```
{
  "version": "1.0",
  "method": "setLEDPattern"
  "result": "ack",
  "id": 1
}
```

Fig 5. OCELOT JSON-RPC format example.

The USB CDC stack provided by the STM32Cube environment is used to manage low-level USB communication through configuration structures, transmission functions, and a reception callback [11]. Whenever a raw message is received from the host, the reception callback stores the data in an input buffer and notifies the console task. The console task then processes the message in two validation stages. First, a general parser verifies that the received message conforms to the JSON-RPC request structure, checking the presence and syntax of the required fields (*jsonrpc*, *method*, *params*, and *id*) as well as whether the requested method is supported by the firmware. If the request is valid, it is forwarded to a method-specific parser, which validates the contents of the *params* object according to the selected command. This second stage verifies parameter syntax, expected data types, and valid value ranges.

Once the request has been successfully validated, the console task translates it into an internal event structure containing the event type and its associated parameters. This event is then posted to the event handler queue, allowing the corresponding driver action to be dispatched by the event handler and executed by the appropriate driver module. If an invalid request format, unsupported method, or incorrect parameter set is detected at any stage of the parsing process, the console layer generates an error response and returns it to the host. After the requested operation is processed by the event handler and the corresponding driver module, the resulting status or output data are routed back to the console layer, which formats and transmits the final JSON-RPC response. An overview of this processing flow is shown in *"Fig. 6"*.

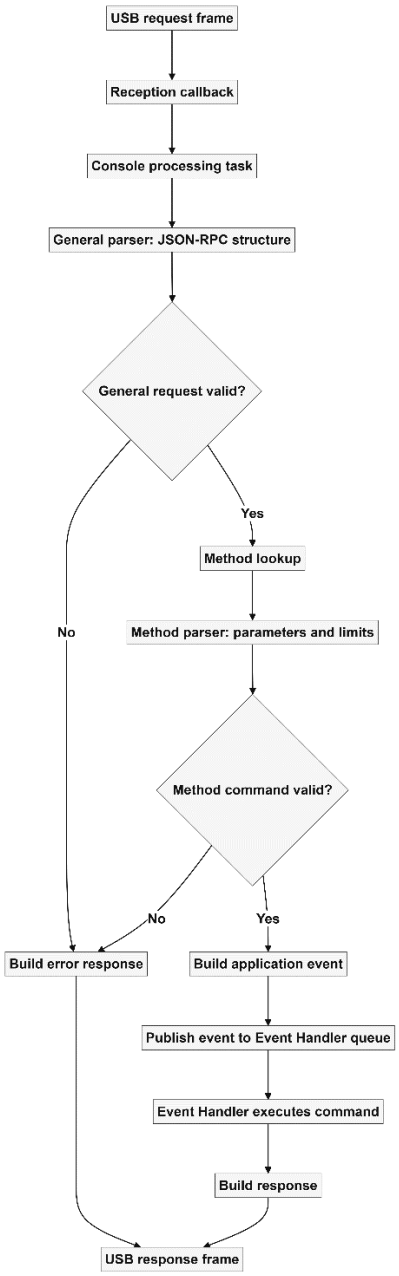

Fig 6. Console layer flow.

D. Event Handler Layer

The event handler layer acts as the central event-dispatch module of the firmware architecture. Its primary function is to receive the internal events generated by the console layer, map each event to its corresponding system operation, and invoke the appropriate driver module to execute the requested action. In this way, the event handler decouples the communication interface from the application drivers and centralizes the execution flow of host requests within the firmware.

As described in the console layer section, each valid JSON-RPC request is translated into a dedicated internal event and published to the event handler queue. This queue operates in a first-in, first-out (FIFO) manner, preserving the order in which host requests are accepted by the console layer. The event handler is implemented as a dedicated RTOS task that continuously monitors the queue using a blocking receive operation with a configurable timeout [7]. If no event is available within this period, the task executes a short delay and resumes queue monitoring. When an event is detected, the task decodes its type and extracts the associated parameters that were previously validated by the console layer.

Based on the event type, the event handler routes the request to the corresponding driver module, such as the digital input, digital output, LED control, or system management driver. The selected driver then executes the requested operation using the parameters contained in the event structure and returns the corresponding status or output data. The event handler collects this execution result and forwards it to the console layer, which is responsible for building and transmitting the final JSON-RPC response to the host. After processing the event, the event handler resumes monitoring the queue and continues with the next available event. This event-driven dispatch mechanism provides a centralized control point for command execution, preserves modular separation between communication and application logic, and simplifies the integration of new firmware functionalities. An overview of the event handler task flow is shown in *"Fig. 7"*.

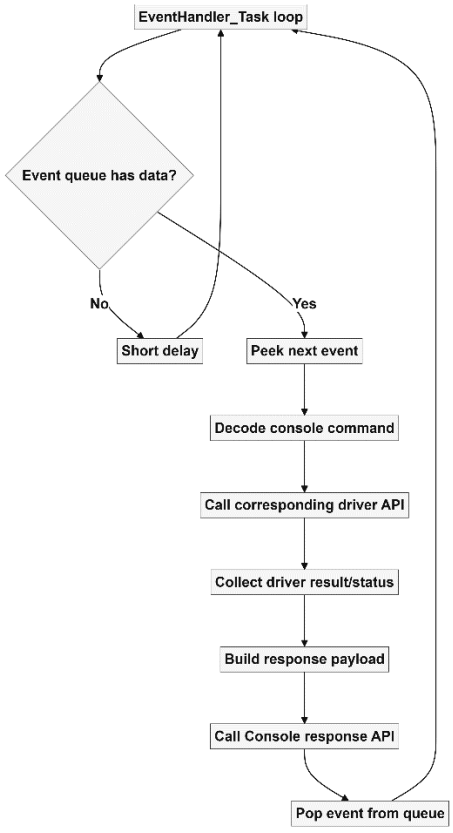

Fig 7. Event handler layer flow.

E. Low Level Drivers Layer

The following section describes the system drivers, focusing on their role in executing instructions dispatched by the event handler and their interaction with the microcontroller peripherals and the underlying hardware platform.

1) Digital Inputs

The digital input driver provides access to the eight onboard pull-up digital input pins and serves as the abstraction layer between the application and the hardware input interface. It supports both single-pin and multi-pin read operations, as well as a configurable software debounce mechanism. These features enable reliable monitoring of sensors and switches while filtering transient signal fluctuations caused by contact bounce or electrical noise.

The driver is implemented as a dedicated RTOS task that periodically samples all digital inputs using STM32 HAL GPIO APIs. The sampling period is defined by the configured debounce interval. During the first execution, the sampled values are stored as the initial reference state for each input. In subsequent iterations, each newly acquired sample is compared against its corresponding reference state and is updated only when a mismatch is detected; otherwise, it remains unchanged. As a result, only debounced and stable input states are exposed to the upper software layers. *"Fig. 8"* illustrates the execution flow of the digital input driver task.

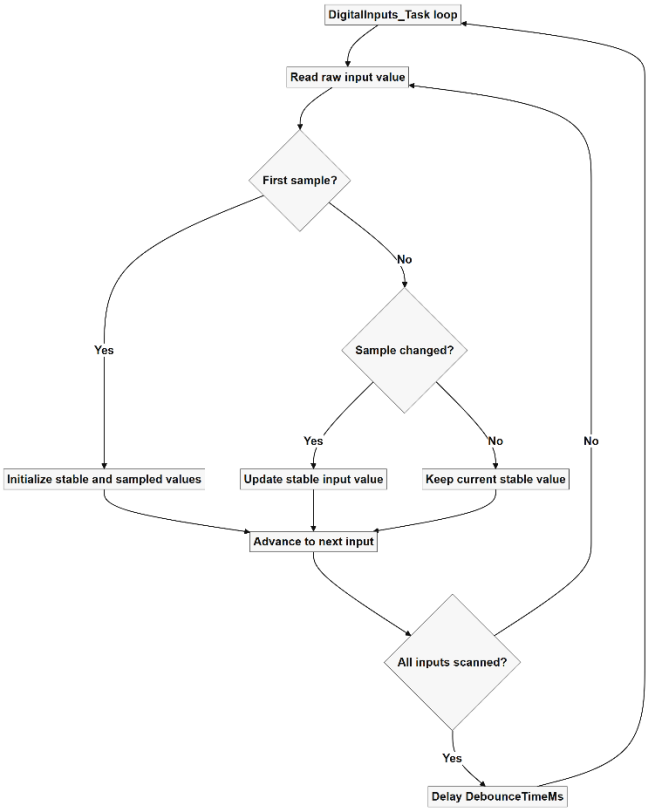

Fig 8. Digital input task flow.

The driver supports requests for reading either individual inputs or multiple inputs simultaneously, both of which return the current debounced input states. In addition, the driver provides a method for configuring the debounce interval. Increasing this interval improves immunity to noise and switch bouncing at the expense of response time, whereas shorter intervals provide faster detection of input transitions. When the debounce interval is configured to zero, the periodic polling task is suspended and input requests access the GPIO pins directly, bypassing the software debounce mechanism. This optimization eliminates the overhead associated with periodic sampling when input filtering is not required.

2) Digital Outputs

The digital output driver provides control of the six onboard relay-driven digital output channels and serves as the abstraction layer between the application and the hardware output interface. These outputs allow the board to control external actuators, including solenoids, relays, gates, and other open-drain compatible devices. The driver supports individual and multi-output write operations, output state monitoring, and configurable toggle operations, providing a flexible interface for digital output control.

The driver is implemented as a dedicated RTOS task that executes periodically or whenever a new output request is received from the event handler layer. During each execution, the task sequentially processes all digital outputs and checks whether a pending operation has been scheduled for each channel. Depending on the requested action (set, clear, or toggle), the driver invokes the corresponding STM32 HAL GPIO API to update the output state. Once the operation has been completed, the processed request is cleared and the resulting output state is stored for subsequent read operations. Toggle requests are managed using a software timing mechanism based on the RTOS system tick. For each active toggle operation, the driver monitors the elapsed time and updates the output state whenever the configured toggle period expires. If the operation corresponds to a time-limited toggle, the driver also tracks its total duration and automatically returns the output to its inactive state once the programmed interval has elapsed. After processing all output channels, the task blocks while waiting for either a notification generated by a new output request or the expiration of its periodic timeout. This event-driven scheduling mechanism minimizes CPU utilization during idle periods while still providing low-latency processing of new output requests. The complete driver task flow is illustrated in *"Fig. 9"*.

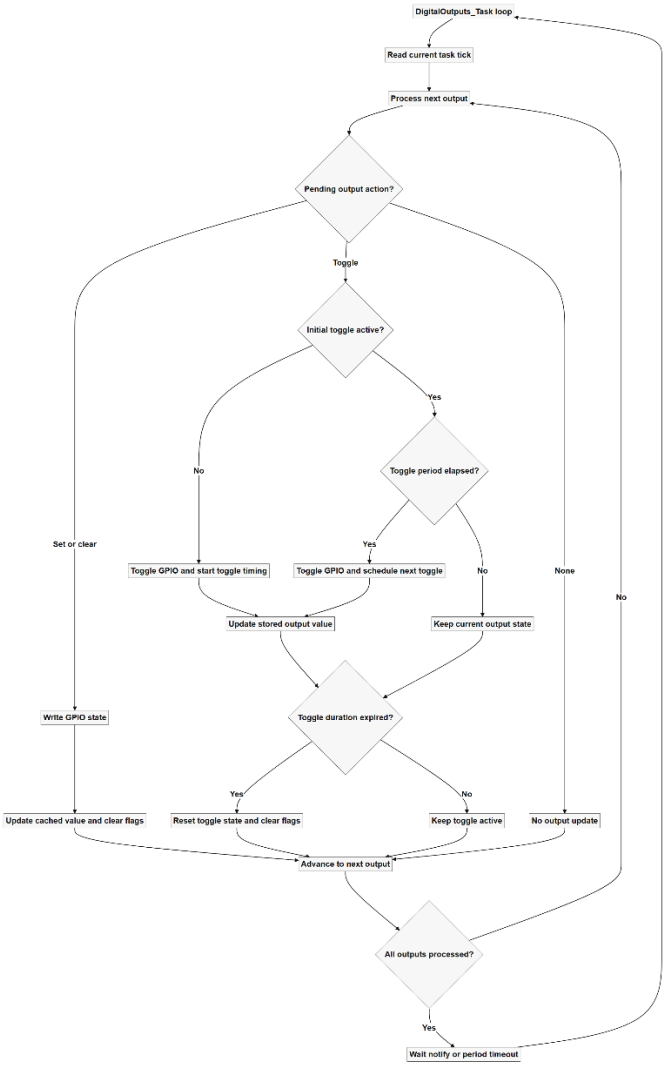

Fig 9. Digital output task flow.

The driver supports requests for writing the state of individual outputs or multiple outputs simultaneously. It also provides continuous and time-limited toggle operations for one or more output channels, as well as requests for reading the current output states. Since output requests are processed asynchronously by the driver task, the application remains decoupled from the low-level GPIO control while maintaining deterministic and centralized management of all digital output operations.

3) RGB Control

The RGB LED driver provides control of addressable APA102 LED strips and serves as the abstraction layer between the application and the underlying LED hardware. The APA102 is an addressable RGB LED with an integrated controller that communicates through independent clock and data lines, enabling high-speed communication and efficient implementation using standard SPI peripherals [12]. Rather than controlling individual LEDs directly, the driver organizes the strip into configurable segments, where each segment is identified by a unique identifier, an initial LED index, and the number of LEDs it contains. This segmentation allows multiple independent lighting regions to coexist on the same LED strip. Once a segment has been defined, different colors and animation patterns can be applied to it without modifying its physical boundaries. By separating segment definition from animation control, the driver allows the same LED layout to be reused across multiple visual effects through a simple host interface. This functionality enables the firmware to present customizable visual indications, including system status, user feedback, and operating conditions.

The driver is organized into three software layers, each responsible for a specific aspect of LED control. The addressable LEDs layer exposes a high-level API that provides animation-specific functions to the event handler layer, including static color, blinking, breathing, fill, and back-and-forth effects. These requests are forwarded to the APA102 segments layer, which constitutes the core of the driver. This layer validates segment configurations, maintains the state of all registered segments, manages animation and timing parameters, and generates the corresponding LED update commands. Whenever a segment configuration is modified or animation requires a visual update, the APA102 segments layer builds a command containing the required LED data and places it into the APA102 task queue. Dynamic animation patterns additionally generate periodic updates to maintain continuous visual effects. The APA102 layer is implemented as a dedicated RTOS task responsible for communicating with the LED strip. The task remains blocked while waiting for commands from the APA102 queue. Upon receiving a new command, it generates the complete SPI data frame corresponding to the current state of all configured LED segments and transmits it using the STM32 HAL SPI driver with DMA support. Once the DMA transfer is completed, the task updates the driver status, handles possible communication errors, refreshes the timing information for active animations, and returns to the waiting state. This queue-based architecture decouples animation processing from the hardware communication layer while allowing efficient and non-blocking SPI transfers with minimal CPU utilization. An overview of the complete driver processing flow is illustrated in *“Fig. 10”*.

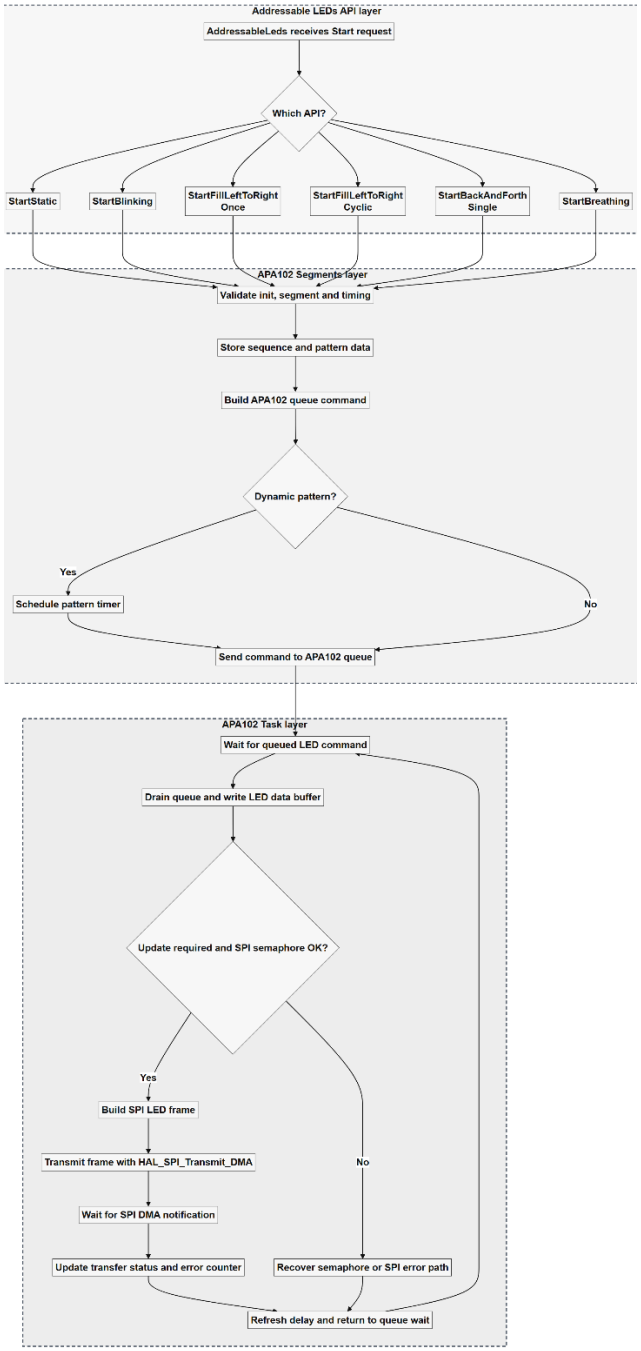

Fig 10. LED driver flow.

The driver supports requests for configuring one or multiple LED segments, obtaining the current segment configuration, assigning animation patterns to one or multiple segments, retrieving the active animation status, and obtaining the overall driver status. Animation requests are accepted only for previously configured segments, ensuring that all visual effects operate on valid LED regions. By separating segment configuration from animation control, the driver provides a scalable and flexible interface that simplifies the implementation of complex lighting effects while remaining independent of the physical LED layout.

F. Firmware Update Mechanism

The firmware incorporates an independent USB bootloader that enables in-field firmware updates without requiring external programming tools. Unlike the application firmware, which implements the JSON-RPC communication interface and peripheral drivers, the bootloader executes as a standalone operating mode dedicated exclusively to firmware programming. The update mechanism is based on the USB Flashing Format (UF2), a block-based firmware format specifically designed for USB MSC bootloaders. UF2 encapsulates firmware data into self-contained blocks that include addressing and validation information, allowing each block to be independently validated and programmed [13]. This format was selected because it provides a robust update mechanism by exposing the device as a standard USB MSC volume, eliminating the need for vendor-specific flashing utilities or dedicated programming software. As a result, new firmware images can be deployed by simply copying a UF2 file to the exposed USB drive, providing a straightforward and operating-system-independent update procedure.

To support this mechanism, the internal flash memory is partitioned into three independent regions, as illustrated in Table II and *"Fig. 11"*. The first 32 kB are reserved for the bootloader, ensuring that the update mechanism remains protected from application modifications. The following 16 kB sector stores shared information exchanged between the bootloader and the application, including boot request flags and update status variables. The remaining 208 kB are allocated to the main application firmware. This memory organization guarantees that the bootloader remains operational even if an application update is interrupted or corrupted, while also providing a synchronization mechanism between both execution modes.

Table II. STM32F11CCU6 Flash memory sectors.

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

```
/* Memories definition */
MEMORY
{
  RAM    (xrw)    : ORIGIN = 0x20000000,  LENGTH = 128K
  USBLOADER (rx)  : ORIGIN = 0x8000000,   LENGTH = 32K    /* Bootloader */
  SHARED  (rx)    : ORIGIN = 0x8008000,   LENGTH = 16K    /* Shared data/flags */
  FLASH   (rx)    : ORIGIN = 0x800C000,   LENGTH = 208K   /* Application */
}
```

Fig 11. Flash memory partition.

During every system reset, execution begins in the bootloader, which first evaluates the shared boot request flag stored in the shared flash sector. By default, this flag remains cleared, causing the bootloader to immediately transfer execution to the application firmware. When the application receives the *enterBootloader* command through the JSON-RPC interface, the request is processed by the console and event handler layers, as illustrated in *"Fig. 12"*. The corresponding event handler routine sets the shared boot request flag and performs a software reset. During the subsequent startup, the bootloader detects the active flag and enters firmware update mode instead of launching the application.

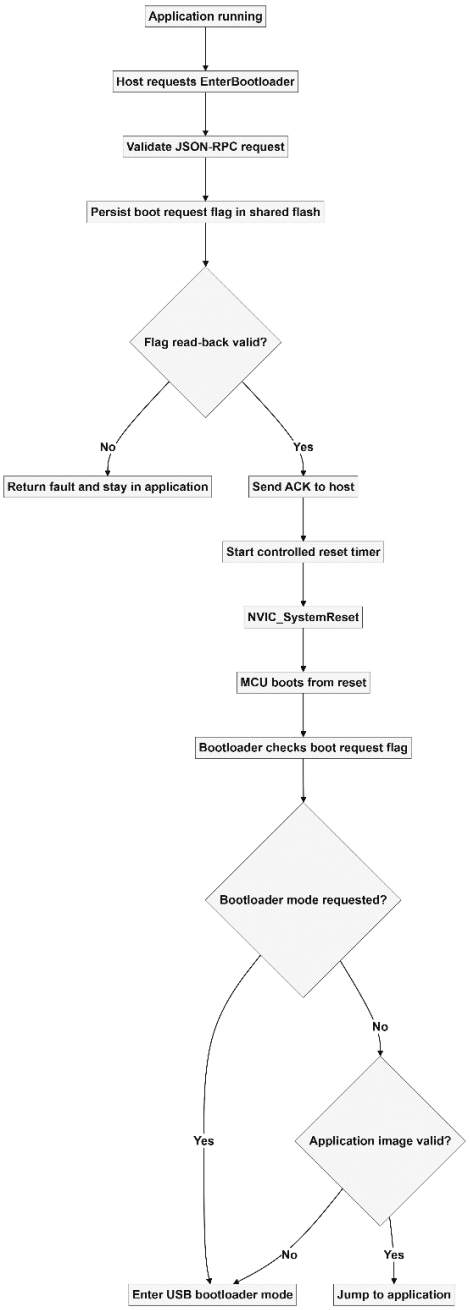

Fig 12. Bootloader mode request flow.

Once bootloader mode has been entered, the USB MSC stack and the GhostFAT UF2 filesystem are initialized, exposing a removable storage device named "OCELOT_IO" to the host computer. The bootloader then waits for incoming UF2 blocks, as summarized in *"Fig. 13"*. Each received block is first validated by checking its UF2 header, family identifier, and target flash address. If the validation succeeds and the target address belongs to the application region, the corresponding flash sector is erased when necessary and the payload is programmed. Otherwise, the block is discarded without affecting the existing firmware. This validation process prevents accidental modification of the bootloader region and guarantees that only application memory is updated. After the last UF2 block has been successfully programmed, the bootloader marks the update as complete, clears the shared boot request flag, deinitializes the USB services, and transfers execution to the newly programmed application. To avoid remaining indefinitely in update mode, the bootloader also implements a one-minute inactivity timeout. If no valid UF2 blocks are received within this interval, the boot request flag is automatically cleared and control is transferred to the existing application firmware. This recovery mechanism guarantees that the device always returns to operational mode even when a firmware update is aborted before completion.

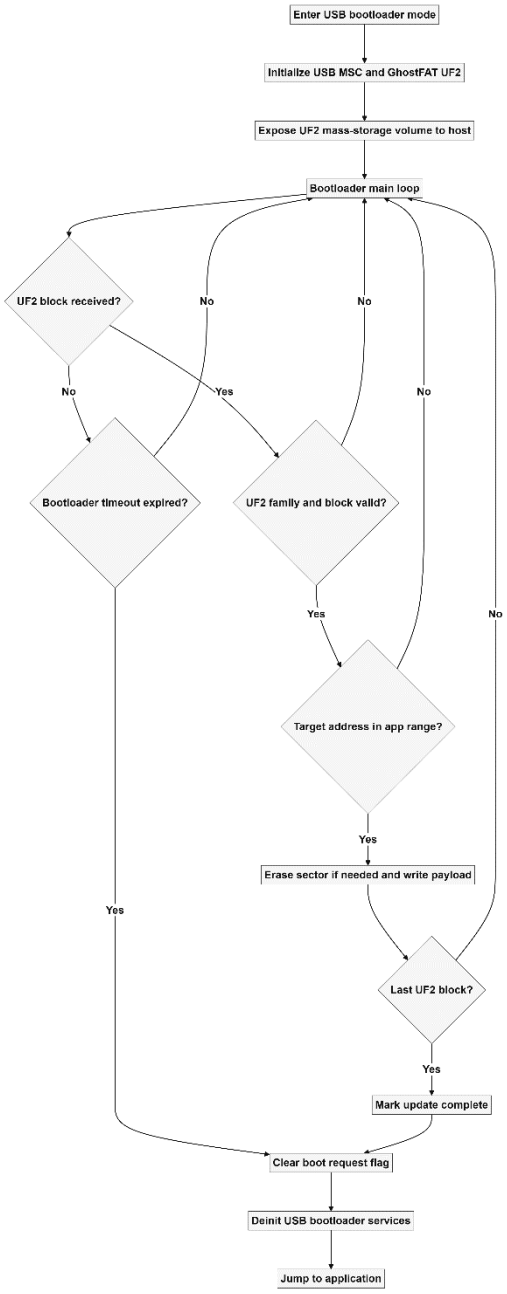

Fig 13. Bootloader UF2 update flow.

III. RESULTS

The OCELOT I/O board was functionally validated by testing its communication interface, peripheral drivers, and firmware update mechanism. The evaluation confirmed the correct operation of the proposed firmware architecture.

The USB CDC communication interface correctly processed all supported JSON-RPC commands. Valid requests were parsed, dispatched by the event handler, and executed by the corresponding driver modules, while invalid requests generated the appropriate error responses. Representative communication examples are shown in *"Fig. 14"*.

Fig 14. USB communication.

The digital input, digital output, and RGB LED drivers operated as expected during functional testing. The input driver correctly performed single- and multi-pin read operations with configurable software debounce. The output driver executed set, clear, and toggle operations, while the RGB LED driver correctly managed segment configuration and concurrent animation patterns on APA102 LED strips. Representative driver operations are illustrated in *"Fig. 15"* and *"Fig. 16"*.

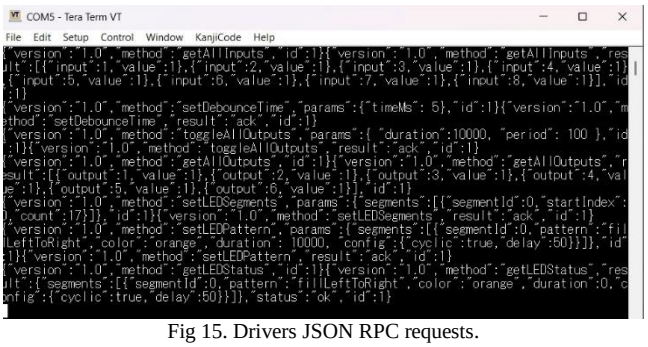

Fig 15. Drivers JSON RPC requests.

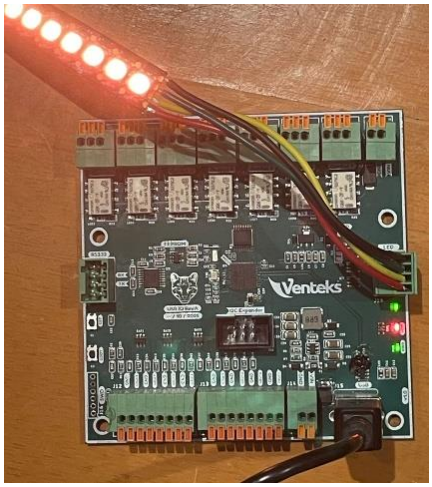

Fig 16. OCELOT with LED pattern.

The firmware update mechanism correctly entered bootloader mode upon receiving the dedicated JSON-RPC command, exposed the USB Mass Storage device, programmed the UF2 firmware image, and automatically launched the updated application. The implemented timeout mechanism also restored normal application execution when no firmware update was initiated. The firmware update process is presented in *"Fig. 17"*.

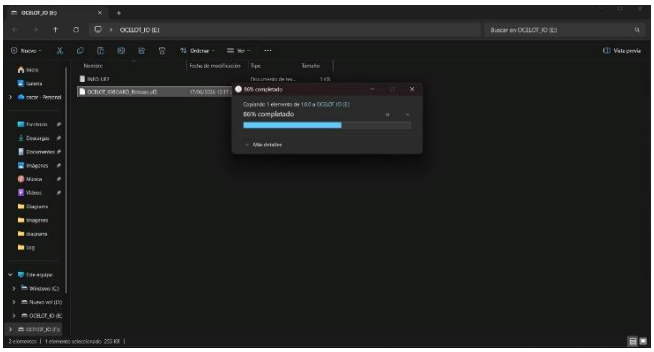

Fig 17. UF2 firmware update.

IV. CONCLUSION

This paper presented the design and implementation of the OCELOT I/O board, a dedicated replacement for the commercial I/O solution previously used in the RC-200 and RC-300 cash management systems. The proposed platform integrates a layered RTOS-based firmware architecture comprising a JSON-RPC communication subsystem, dedicated peripheral drivers, and a USB UF2 bootloader for simplified firmware updates. Functional validation confirmed the correct operation of the communication subsystem, peripheral drivers, and firmware update mechanism. By replacing the previous commercial solution with a purpose-built platform, OCELOT reduces hardware dependency, simplifies software integration, and provides a scalable and maintainable embedded platform for future development of the RC-200 and RC-300 systems.

REFERENCES

[1] Venteks, "VSafe Cash Solutions." [Online]. Available: https://vsafecash.com.mx/. Accessed: May 21, 2026.

[2] Venteks, "Cashflow RC200 User Manual," Internal document, 2025.

[3] Numato Lab, "8 Channel USB GPIO Module with Analog Inputs." [Online]. Available: https://numato.com/docs/8-channel-usb-gpio-module-with-analog-inputs/. Accessed: May 21, 2026.

[4] Venteks, "USB_IO_RevA_Sch," Internal schematic, Oct. 2025.

[5] STMicroelectronics, "STM32 Embedded Software." [Online]. Available: https://www.st.com/content/st_com/en/stm32-mcu-developer-zone/embedded-software.html. Accessed: Jun. 13, 2026.

[6] Amazon Web Services, "FreeRTOS Overview." [Online]. Available: https://www.freertos.org/Documentation/00-Overview. Accessed: Jun. 14, 2026.

[7] A. Coronado, "Beyond State Machine: Modular Software Reference Architecture Design Based on RTOS – An Industrial Experience Report," Instituto Tecnológico y de Estudios Superiores de Occidente (ITESO), Guadalajara, Mexico, 2021. [Online]. Available: https://rei.iteso.mx/bitstreams/f74643e0-ca80-4081-99e2-cb5227ab6837/download

[8] Microchip Technology Inc., "CDC Overview." [Online]. Available: https://onlinedocs.microchip.com/oxy/GUID-38BB0804-1305-4C34-99DC-04025E667766-en-US-5/GUID-56B2B02B-BB4F-41F1-8774-BCA3B46CD210.html. Accessed: Jun. 18, 2026.

[9] F. Mauroner and M. Baunach, "Remote Instruction Call: An RPC Approach on Instructions for Embedded Multi-Core Systems," Graz University of Technology, Graz, Austria, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8352392

[10] JSON-RPC Working Group, "JSON-RPC 2.0 Specification." [Online]. Available: https://www.jsonrpc.org/specification. Accessed: Jun. 18, 2026.

[11] STMicroelectronics, "Introduction to USB with STM32." [Online]. Available: https://wiki.st.com/stm32mcu/wiki/Introduction_to_USB_with_STM32. Accessed: Jun. 19, 2026.

[12] Shenzhen LED Color Opto Electronic Co., Ltd., "APA102 Super LED Datasheet." [Online]. Available: https://www.szledcolor.com. Accessed: Jun. 19, 2026.

[13] Microsoft, "UF2 File Format Specification." [Online]. Available: https://microsoft.github.io/uf2/. Accessed: Jun. 19, 2026.