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

Beyond State Machine: Modular Software Reference Architecture Design based on RTOS – An Industrial Experience Report

---

TRABAJO RECEPCIONAL que para obtener el **DIPLOMA** de

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: **ALDO ALONSO CORONADO MÉNDEZ**

Asesor: **M. Sc. SERGIO NICOLÁS SANTANA SÁNCHEZ**

Tlaquepaque, Jalisco. julio de 2021

Beyond State Machine: Modular Software Reference Architecture Design based on RTOS – An Industrial Experience Report

Aldo Coronado
Departamento de Electrónica, Sistemas e Informática
Especialidad en Sistemas Embebidos
Tlaquepaque, México
xe730053@iteso.mx

***Abstract* - Technology based industry is characterized by the accelerated need for innovation to meet market requirements, therefore, it is important to harness a solid framework for a faster and improved new product development. This article presents a novel reference architecture design to support applications with complex embedded software workflows. The design is focused on context management in the application layer and handling input data from peripherals. The architecture was successfully implemented in an electronic smart safe custom embedded hardware; however, it can be used in any embedded product with task concurrency and complex software workflow.**

***Index Terms* – reference architecture, embedded, RTOS.**

I. INTRODUCTION

In modern technology companies, it is a common practice to develop new projects based on previous generation products to reduce time-to-market and costs [1] [2]. Therefore, there is a need to create a shared reference architecture between several products with similar features, such as hardware components, software modules, or the application. However, over the years this architecture might become naturally obsolete. For this reason, the development of a reusable up-to-date architecture using modern technologies is fundamental for product evolution.

At Venteks, a company dedicated to developing vending and cash embedded solutions [3], the reference software architecture used in almost all products consists of a cooperative bare metal scheduler (without an operating system), and software modules based on state machines. Over time, the product design requirements regarding complexity and reliability increased, making it more difficult to sustain the legacy architecture. Consequently, maintaining, escalating, bug fixing, and new feature implementation processes may become inefficient. Alternatively, the broad range of modern and varied technologies, like Real-Time Operating Systems (RTOS) and communication protocols, provide more options to tackle these problems implementing an improved reference architecture.

The key features of Venteks' product portfolio are the user interface (e.g., screen and keyboard) and cloud connectivity. Therefore, the proposed reference architecture in this paper focuses on these two features. To the best of our knowledge, there are no design software guidelines using RTOS to date, in which state machines are an inherent part of the application context control and device drivers. For this reason, the design depicted in this paper is a novel pattern to run several context functionalities in an embedded application.

The product selected for this case study is an electronic Smart Safe considering the software involves a significant number of features, such as sensors for temperature, presence and vault instrusion, among others, in addition to human-machine interface through a screen and touch keypad, point-of-sale sofware connectivity over Local Area Network (LAN), cloud connectivity, bluetooth connection to a mobile app, and cash acceptors communication.

II. DESIGN METHODOLOGY

The design process for the architecture was first applied through a single existing product re-design, and this architecture is planned to roll-out over new products. With the objetive of tailoring the architecture to the available RTOS and conectivity tools, the RTOS and cloud conectivity protocol were selected before the architectural design process. Once

these components had been determined, the main purpose of the architecture is the migration from the finite state machine paradigm to a pattern compatible with the RTOS features.

A. RTOS Selection

From the RTOS available in the market, both propietary and open source, real-time and safety-critical hallmark characteristics are not a requirement in any of Venteks' products. For this reason, the RTOS selection was based on cost, usability and support. Furthermore, the system's main custom hardware operates under the STM32 microcontroller (ARM Cortex-M4 kernel), which incorporates readily accesible and well tested HAL modules, and the STM32 Cube Integraded Development Enviroment (IDE) provides direct integration and RTOS-aware debugging for Azure® ThreadX and Amazon FreeRTOS™ Kernel [4], owned and supported by cloud services market-leaders. Most notably, FreeRTOS is also supported by a large community of colaborators because of its open source nature [5].

B. Cloud Connectivity Protocol selection

Regarding cloud connectivity, there are more than 1500 smart safe units installed in the field and the forecast is predicted to reach three times this amount in the next couple of years; therefore, the need for a reliable, cost-effective, and scalable internet protocol is imperative. Cloud connectivity was formerly provided with a basic binary M2M protocol of small data packets (20 bytes in average) using the GPRS/LTE Quectel® BG96 module to connect to a socket server with simple reliability mechanism. The BG96 module supports a rich set of internet protocols, such as TCP/IP, HTTP and MQTT [6]. In comparison with the former protocol, HTTP and MQTT have a significant message overhead, which translates directly to data usage and thus cost increase. The selected cloud communication stack was MQTT, considering it is the growing market-leader in IoT applications [7], due to its lightweightness, and embedded Quality of Service.

C. Architectural Design

The smart safe functionalities available to users can be accessed through input peripherals and the activation of some sensors. The system workflow is modelled in a state-machine diagram, where state transitions are triggered by inputs or the system events. The proposed design is a 5-layer architecture (HAL, RTOS, Device Drivers, Services/Middleware, and Application Layer). Using event-driven and producer-consumer paradigms, a single event queue is used to control peripheral inputs, messages, and system transitions.

1. Applets and Scenes

Each smart safe functionality is represented in the form of an applet, using the same criteria applied to the state definition in a finite state-machine, and may internally incorporate any number equal or greater than 2 scenes in an array form. The concept of an applet is a context routine that can somewhat independently exist from the rest of the program. Likewise, applets are further divided into scenes, which is the smallest system portion able to receive and process events, acting as the consumer. The purpose of scenes is to abstract and define behavior of each screen inside the system context. The elements associated to the scene definition are:

- Scene ID
- Variable Element Enable (e.g., Time banner)
- Associated Screen Print Function
- Scene to go after timeout
- Events Subscription Array

If necessary, the *applet* module could contain *entry* and *exit* routines to configure *applet* inner context before execution. Additionally, to provide context control for the system workflow, previous and current *applet-scenes* are contained in a structure used by the *event handler* for state transitions. *Scenes* cannot be executed concurrently, therefore, the *applet-scene* code is executed from the *event handler* task context.

2. Event Handler

The available event in the queue is compared to the current *scene* subscriptions and if the subscription is correct, the event type and data is then delivered to the *scene* to be processed. After processing, the *scene* will reply if the event can be discarded or should be retained in the queue with the objective of being propagated to the next executing *applet-scene*. The *scene* can also send an event if a state transition is required and specify which *applet-scene* will execute next. These special *applet* events are only processed by the *event handler*.

3. System Diagnostic

The mission of this software component is to perform a periodical check for system errors (e.g., bill jam,

keyboard disconnection). These errors are provided as driver error flags, and to deliver an event if the system cannot operate normally anymore, in other words, go out of service. If the system recovered from the error, it could then send the event to inform that the system can operate normally again.

4. Input Events

Software modules such as peripheral drivers send notifications (e.g., key pressed, door opened, bill inserted) through events to the executing *applet-scene* as shown in Figure 1, acting as producers. These events can be discarded or processed depending on the *scene* subscription list.

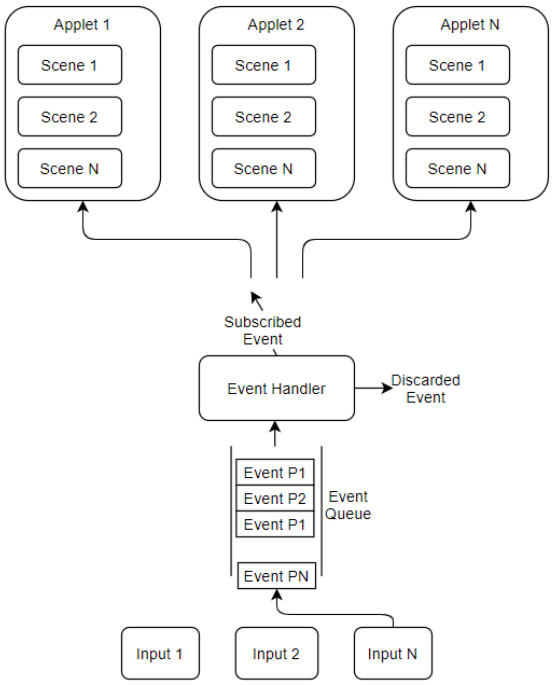

Fig. 1. Event handling method

III. EXPERIMENTAL RESULTS

This section portrays the experimental results of implementing the modular architecture in the smart safe embedded software development process. We present the modular architecture of the smart safe software implementing the applet partition paradigm and then, the result of an event propagation instance within the smart safe software. Finally, the development process improvement due to the reference architecture implementation.

A. *Software Architecture*

The reference architecture was applied to the smart safe embedded software, containing originally 4 *applets*: Main Menu, Money Insertion, Password Validation, and Open Vault State as shown in Figure 2. To increase portability, the software modules that can generate events were located on driver wrappers, or the middleware layer. This way, it is possible to migrate to another cloud connectivity protocol or bill validator technology and keep the application layer unaffected.

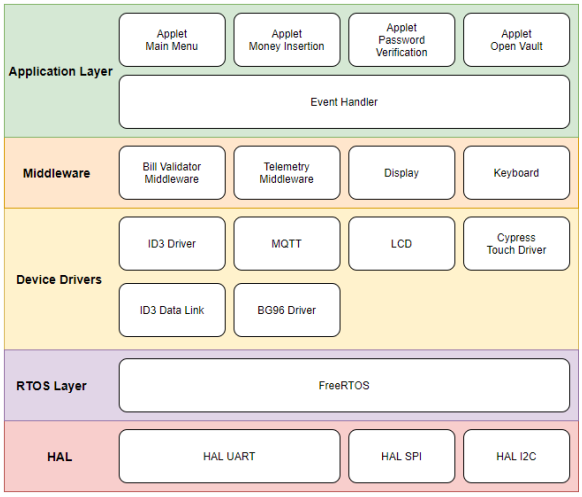

Fig. 2. Smart Safe layered architecture

B. *Event Propagation Results*

An instance of event propagation in the smart safe software was necessary under the following condition: while the main screen is active (main menu applet), a numeric key is pressed on the keyboard. The processing of this event generates the applet transition request onto password validation applet. This applet transition event is then sent back to the event handler in combination with the signal to propagate the last received event (numeric key pressed), so the first number can be printed on the display while typing the password. This propagation scenario sequence is shown in Figure 3 in numbered order.

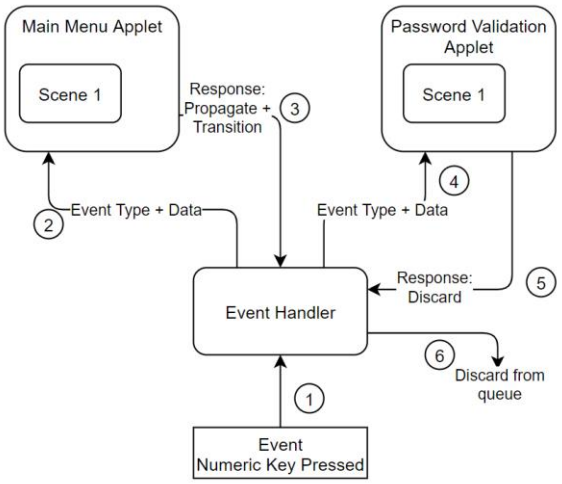

Fig. 2. Event propagation sequence

C. *Modularity and time-to-market*

Although the integration of HAL and RTOS layers accounts for a significant program footprint increase, since the implementation of the application-layer modular architecture, the number of written lines of code necessary for this project were notably reduced and consequently, achieving a time-to-market cut in approximately 40%.

While the smart safe project development was under way, the reference architecture was successfully tested in another product development. Remarkably, 35% of software components from the smart safe project were able to be reused seamlessly, particularly driver stacks such as cash validation and telemetry, among others. Therefore, successful outcome for the reference architecture regarding portability and reusability was proven.

CONCLUSION

This article presented a novel modular reference architecture for the design of embedded systems with a complex workflow within the product software. The architecture was proven successful, particularly, in the application layer, where the abstraction of context through *applets* and *scenes* interacting with events sent from inputs, allows a solid and reliable method for the software to take actions only derived from relevant events within the application current context.

REFERENCES

[1] B. Graaf, H. v. Dijk and A. v. Deursen, "Evaluating an embedded software reference architecture - industrial experience report," *IEEE*, 2005.

[2] M. Galster, "Software reference architectures: related architectural concepts and challenges," in *1st International Workshop on Exploring Component-based Techniques for Constructing Reference Architectures (CobRA)*, Montreal, QC, Canada, 2015.

[3] Venteks, Venteks, 2016. [Online]. Available: www.venteks.com. [Accessed 01 07 2021].

[4] STMicroelectronics, ST, 2021. [Online]. Available: https://www.st.com/en/development-tools/stm32cubeide.html. [Accessed 01 07 2021].

[5] Amazon Web Services, Inc., [Online]. Available: https://www.freertos.org/partners/overview.html. [Accessed 07 07 2021].

[6] Quectel, "LTE BG96 Cat M1/NB1/EGPRS," [Online]. Available: https://www.quectel.com/product/lte-bg96-cat-m1-nb1-egprs/. [Accessed 13 07 2021].

[7] A. K. Biswajeeban Mishra, "The Use of MQTT in M2M and IoT Systems: A Survey," *IEEE Access*, pp. 201071 - 201086, 4 November 2020.