Asynchronous Data Processing Using QStateMachine

Binoy Kurikaparambil Revi

Independent Researcher, USA
binoyrevi@live.com

Abstract

In embedded system design, one of the prominent challenges lies in the efficient handling of data. This challenge arises particularly in scenarios where data must be awaited from external sources. The crux of the issue is ensuring that while the software is engaged in processing the data that has already been received, it does not miss any incoming data that may arrive simultaneously. To address this, every software module may need to be meticulously designed to accommodate asynchronous processing. This involves creating mechanisms that allow the system to respond to incoming data streams promptly, ensuring data integrity, and efficiently designing the data processing module. Effective design in this area is crucial for maintaining the reliability and performance of the embedded system. QT frameworks [4], with the magic of Signal and Slot mechanisms and the QStateMachine[1], provide an exceptional design solution for embedded system development to handle asynchronous data processing.

Keywords: Data Processing, QT Programming, Data Communication

Introduction:

Effective concurrent data processing handling is crucial for multi-threaded applications that perform data communication functions in the background. This capability is particularly important in various embedded systems and Internet of Things (IoT) use cases, where responsiveness and efficiency are essential to meet specific system requirements. In the context of hard real-time systems and critical applications, such concurrent processing is not just beneficial but necessary to ensure seamless operation and reliability.

The QT framework effectively manages data interactions from various devices and networks by utilizing a system of signals and slots. Additionally, the QStateMachine component offers a structured state machine that governs transitions between distinct states based on user-defined signals. By aligning the functionalities of the QStateMachine with the data reading mechanisms, which operate on the principles of signals, slots, and state transition mappings, developers can create a cohesive and efficient data processing module. This integration represents a well-thought-out design strategy for embedded system software, enhancing responsiveness and robustness.

Problem Description:

The problem faced by the software architect in the past and present in handling the incoming data and how the data receiver buffer is monitored for incoming data. The simple primitive solution was an infinite loop(Figure 1). However, this comes with significant drawbacks.

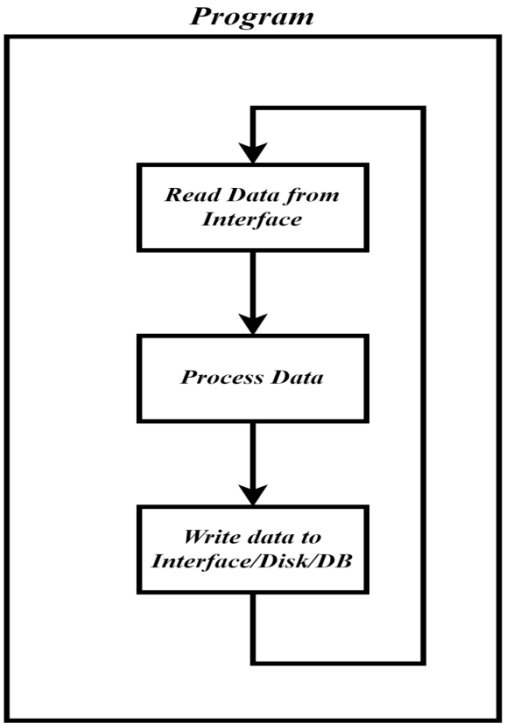

Figure 1: Primitive Data Processing - Not efficient

The issue with this is that the data read depends on data processing time, which can create a significant bottleneck. If write operations to disk or interfaces are included, then we face an even bigger problem. This has led software designers to adopt a threading mechanism combined with polling techniques. This new approach is more effective than previous methods because it incorporates polling with a defined time interval. However, the interval introduced a minimum practical system delay, preventing the system from polling data beyond a certain limit. Figure 2 demonstrates how the polling in threading is done.

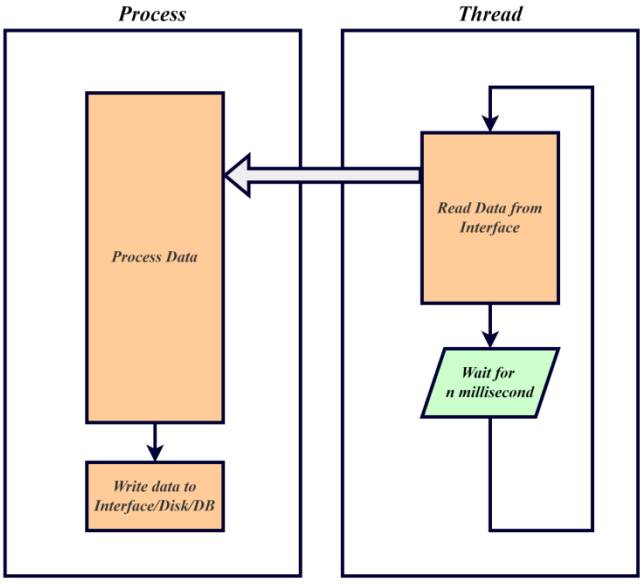

Figure 2: Data read by polling from thread

Even today, there are systems functioning worldwide that rely on this particular methodology. As long as these systems do not impose stringent requirements on the frequency of data reading, this approach tends to operate effectively and reliably. When data needs to be processed immediately upon arrival, it can be

challenging for the system to keep up due to its limitations in minimizing delays that are less than those of a practical programmable sleep time. This situation calls for a more robust system to synchronize data arrival at the interface and th data read.

Event-Driven Programming:

Event-driven programming has become essential for embedded system programmers, providing them with a powerful alternative to traditional polling methods. This programming paradigm allows systems to respond instantly to external events or data rather than continuously checking for changes or buffers. Developers can create more efficient and responsive systems that conserve resources and improve performance by utilizing event-driven programming. This shift away from polling reduces CPU usage and enhances the overall reliability and functionality of embedded applications, making it a vital technique in modern embedded system design. There are different frameworks that provide techniques for developing event-driven programming. QT Framework is one of the most mature embedded system frameworks available for developing embedded system programming. Common use cases for the even-driven programming[3] are listed below:

1. Handling Asynchronous Data: In this scenario, the incoming data is managed by an external entity, making it impossible to predict the data traffic to the system. Additionally, the system needs to perform other functions, some of which are time-sensitive. Therefore, instead of constantly checking for new data, an event-driven approach is the most effective way to handle this situation. With this approach, the application will receive a notification or the incoming data itself as soon as it arrives.
2. Handling devices: Reading data from devices or connected sensors requires careful attention to the frequency of data retrieval. Often, the system only gets a brief time slice that is insufficient for thorough processing. In such cases, the read method must quickly read the data, pass it along for further handling, and proceed to the next read. An event-driven approach is highly effective for managing this scenario.
3. UI Interaction: No one wants to deal with a frozen user interface (UI) after performing an operation. Actions should be processed immediately, and the UI should remain responsive for subsequent actions. An event-driven approach is an effective way to manage this situation.
4. State Machine: A state machine is designed to execute a specific set of tasks in one state and then wait for an event to trigger a transition to another state, where it will perform a different set of tasks. This design is an excellent example of event-driven programming.

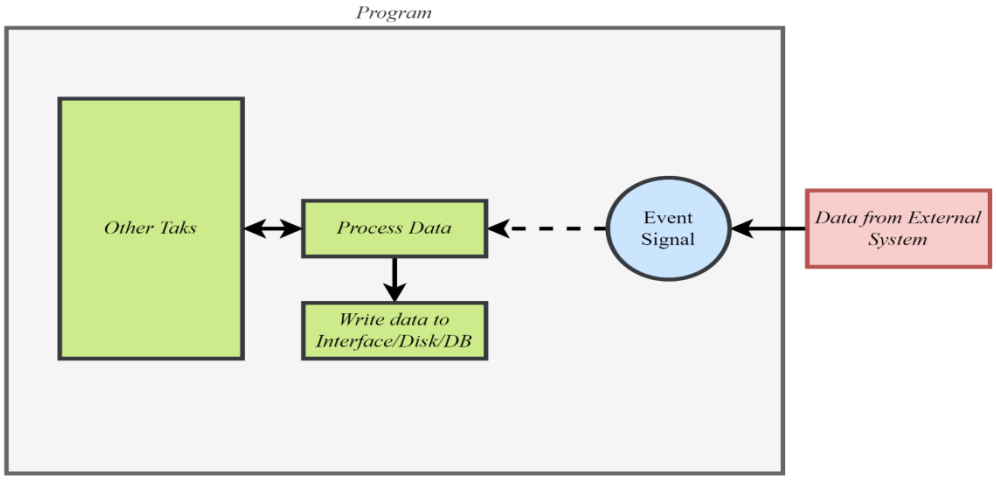

Figure 3: Even Driven Processing of data

QT Signal Slot Mechanism for event-driven programming:

QT offers a powerful signal-slot[4] mechanism that enables the design of asynchronous data interactions, fundamentally based on event-driven techniques. This approach begins by establishing connections between signals and slots. Essentially, a signal is an indication that something has occurred, while a slot is a method designed to respond to that event. By defining these connections, the program can intelligently react when a specific signal is emitted from one object, automatically invoking the appropriate method in a different class. This creates a responsive and efficient communication system within the application, allowing for seamless event handling and collaboration between various components.

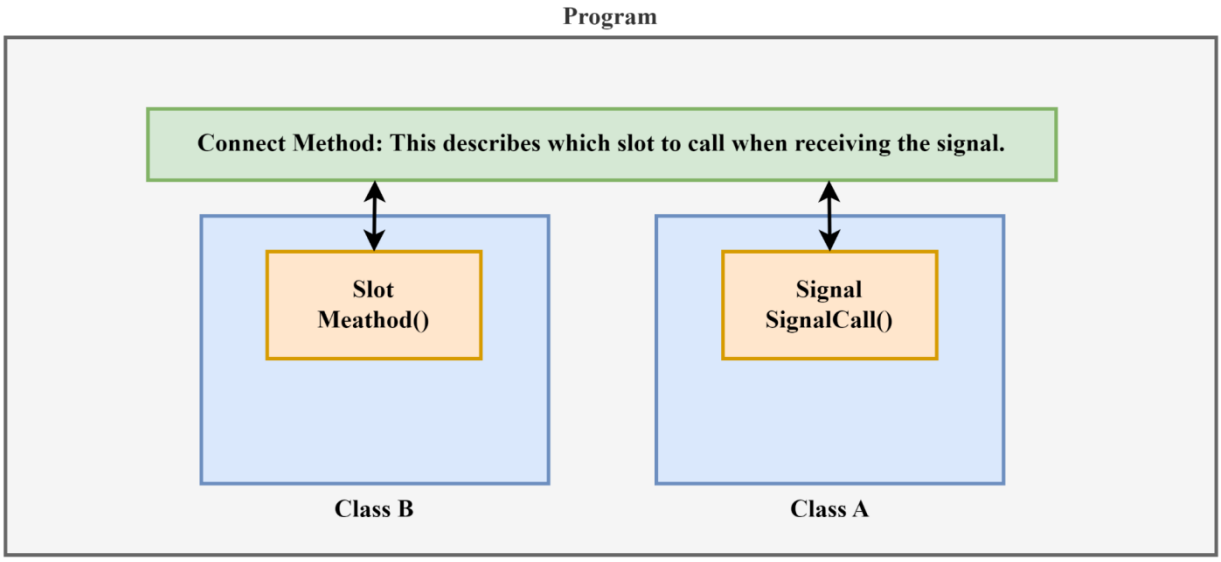

Figure 4: Signal -Slot Mechanism

Imagine a specialized class designed to handle incoming data from a device. Whenever new data becomes available, the system generates a signal to notify the program. By implementing a connection method that is set up to listen for these signals from the data source, the corresponding slot—essentially a method that is invoked in response to the signal—will be executed, taking the incoming data as its input. This setup ensures that every time new data is provided, the method responsible for processing that data can be triggered automatically, allowing for efficient and timely handling of the information.

Asynchronous Processing using QTStateMachine:

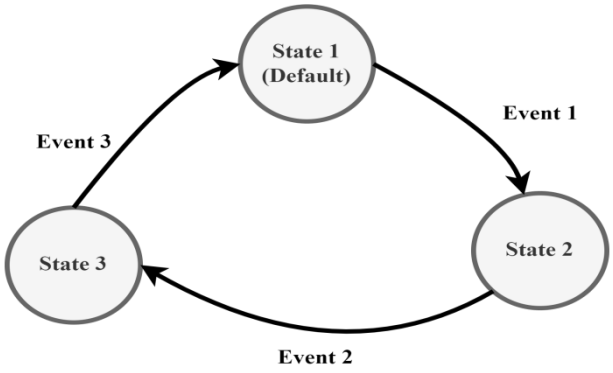

Figure 5: State Machine

The flowchart illustrating the state machine is depicted in Figure 5. Initially, the state machine[2] resides in the default state. For the purposes of this example, let's consider that this default state is labeled as State 1. Upon receiving a specific event trigger, the state transitions to State 2. From a programming perspective, this transition signifies that the method associated with State 2 is executed, effectively changing the current state from State 1 to State 2. The state machine serves as a robust design framework, particularly suited for programs that require intricate state transition logic or that operate as wizard-like workflows. This structure facilitates clear and organized management of various states and their corresponding actions, allowing for more efficient and maintainable code.

The Qt State Machine[1] framework is a powerful tool for implementing state machines. Here are the steps to create a Qt State Machine:

1. Create a `QStateMachine` instance.[1]
2. Create `QState` instances for each state in your machine.[1]
3. Define signals that will serve as event triggers.
4. Implement transitions that define how the state machine moves from one state to another.
5. Specify functions that need to be executed upon entering each state.
6. Set the default state
7. Start the state machine.

Conclusion:

The QTStateMachine and the Signal-Slot mechanism are powerful programming techniques offered by the Qt framework, particularly useful for designing highly responsive real-time embedded systems. This approach effectively eliminates the need for traditional polling mechanisms, instead embracing an event-driven programming model that leads to more efficient handling of events and data. The Signal-Slot mechanism is a robust communication method that allows different components of a program to interact in a decoupled manner. By emitting signals when certain conditions are met and connecting these signals to slots, Qt enables system to respond dynamically to user interactions or system events. This promotes a streamlined architecture where components can operate independently while still being able to communicate when necessary.

On the other hand, the QStateMachine class provides a structured and clean implementation of state machines. State machines are essential for modeling the behavior of systems that go through various states based on events or conditions. The QStateMachine simplifies the transition between different states and makes it easier to define complex state-dependent behaviors. This capability is particularly valuable in real-time applications, where the system's response must be both timely and predictable.

Together, these powerful features of the Qt framework make it well-suited for developing soft and hard realtime applications, especially in embedded systems where performance is critical.

References

1. C. Yu et al., "The implementation of IEC60870-5-104 based on UML statechart and Qt state machine framework," 2015 IEEE 5th International Conference on Electronics Information and Emergency Communication, Beijing, China, 2015, pp. 392-397, doi: 10.1109/ICEIEC.2015.7284566.
2. W. Said, J. Quante and R. Koschke, "On State Machine Mining from Embedded Control Software," 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 2018, pp. 138-148, doi: 10.1109/ICSME.2018.00024.

3. L. Guo, A. S. Vincentelli and A. Pinto, "A complexity metric for concurrent finite state machine based embedded software," 2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES), Porto, Portugal, 2013, pp. 189-195, doi: 10.1109/SIES.2013.6601491.
4. Lazar, Guillaume, and Robin Penea. Mastering Qt 5: Create stunning cross-platform applications using C++ with Qt Widgets and QML with Qt Quick. Packt Publishing Ltd, 2018.