International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Multithreading: Threading State Machine

Binoy Kurikaparambil Revi

Independent Researcher
binoyrevi@live.com

Abstract:
State Machines are instrumental when designing software with an event-driven execution pattern. However, when a complete state machine is just a tiny part of the bigger system, implementing the State Machine inside a thread benefits the overall program execution under certain conditions. QThread and QStateMachine are Thread and State Machine classes, respectively, provided by the QT framework that can be used by software programs to implement this technique to handle such scenarios, especially when sending the data from the State Machine to the main program becomes inevitable.

Introduction:
State Machine works using the idea that the State Machine at any point in time will be in one of the well-defined states that execute functions assigned to the state. State Machine moves to another state when it receives a predefined command. The command needs to be understood by the State Machine and should apply to the current state. In a software design, a State Machine provides better clarity to the program execution if the requirements align with the State Machine concept. State Machine also provides the flexibility to quickly scale the program by adding more states or reducing states that are no longer needed.

State Machine Design:
It's important to understand that the State Machine runs in an even loop. That means the State Machine will be in some state at any given time. The following steps summarizes State Machine[1] design:

1. Define States: From a software design perspective, states can be considered a function that must be executed when the State Machine moves to that state. Even after the function execution is complete, the state doesn't change unless the state machine receives the command or signal, which is predefined to move the current state to another state. So technically, once the states are defined, the program is expected to have a set of functions that correspond to the set of states.
2. Define Signals: Signals are commands that are defined to command the transition of the current state of the State Machine to another state. No rule restricts any state to have only a certain number of signals that can be interpreted.
3. Define Transitions: Transitions are the wiring that informs the State Machine during initialization about the Signal and State relations. This can be interpreted as an action the State Machine must take when a Signal is received. The State Machine is expected to follow the rules defined by the transitions to move from one state to another. There can be complex cases where a State has multiple transitions defined using various signals to move to numerous states.
4. Define Initial State: This is an essential step. It tells the state machine to be in a predefined State at the program's start or start-up.

Consider the following State Machine with four states, as shown in Figure 1. Let us define the initial state as the "Initial State," which is the state of the State Machine on start-up.

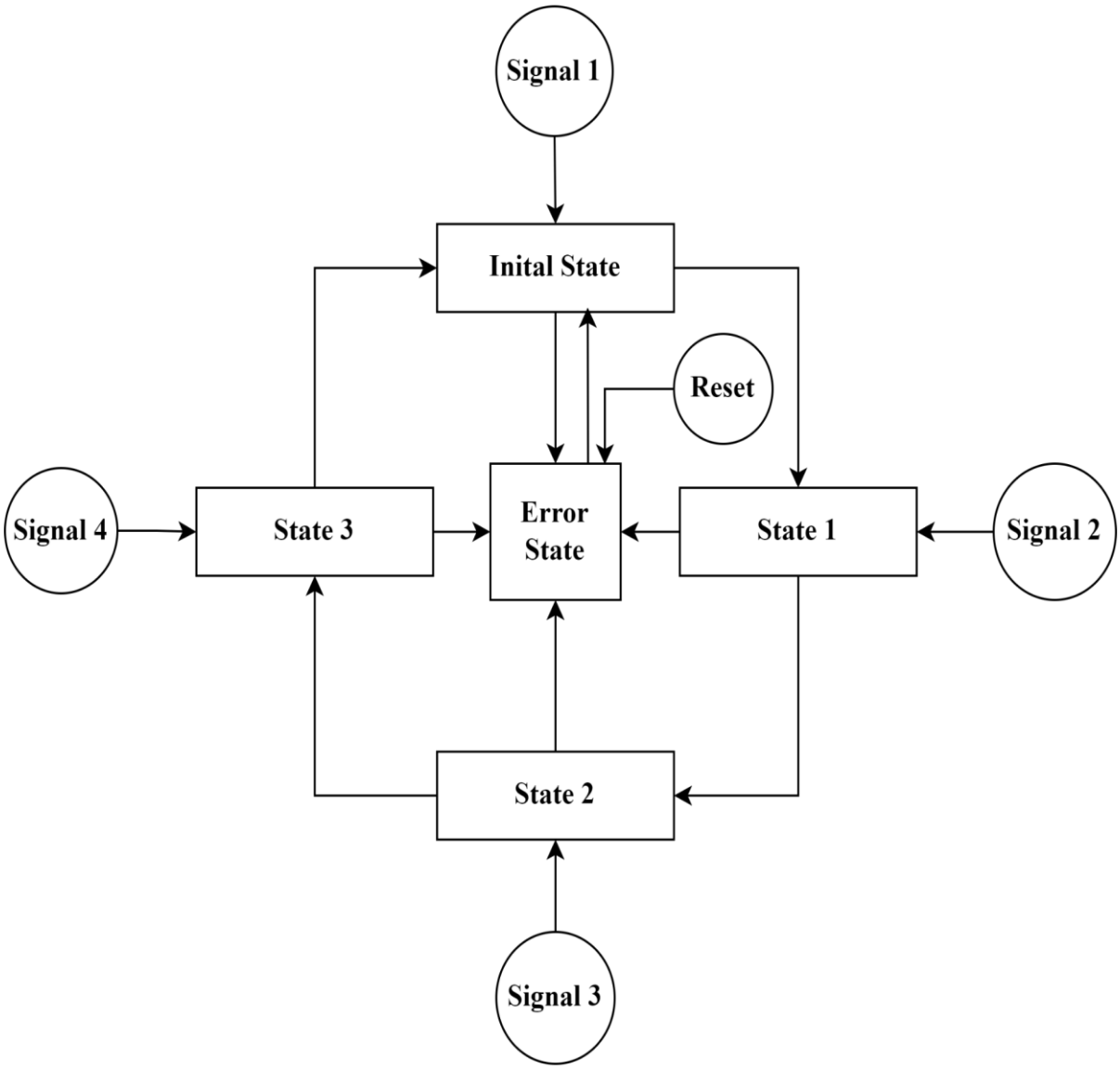

Figure 1: Example of a typical State Machine

In this example, the State Machine will maintain its current state unless it receives a Signal that is understood by the current state; at this point, the State is transitioned to a different state. For example, if the State Machine is in "State 1", only "Signal 2" or an Error Signal can change the State of the State Machine. If the State Machine gets the "Signal 2", the State is moved to "State 2". If an Error Signal is received, the State is forced to Error State. It is essential to understand that if the State Machine receives any other Signal (Signal 1 or Signal 4 or Signal 3), the State is not moved from State 1. The transition of the State Machine is defined as in Figure 2.

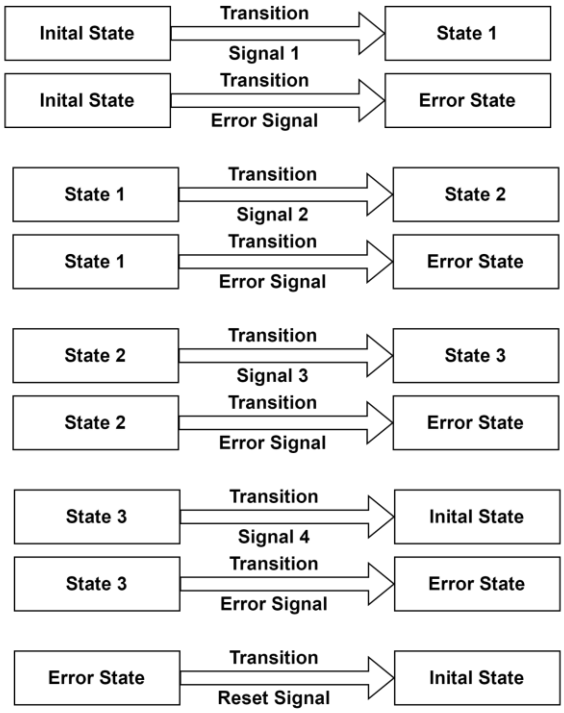

Figure 2: Transition Definitions

Threading State Machine:

A thread is a lightweight process that shares key resources, like memory, with the main process. Threading enables concurrent execution of multiple tasks within the program[4]. Even if the thread is running parallel with the main program, the execution of the thread depends on the priority assigned to it. If the main process runs some critical or real-time tasks and the thread is designed to run low-priority tasks, the thread can be assigned a low priority before starting the thread. This threading technique can run the State Machine as a low-priority task inside the thread. Moreover, as the thread shares data and resources, it is easy to transfer the data from the thread to the main program. Figure 3 demonstrates the State Machine running inside the thread.

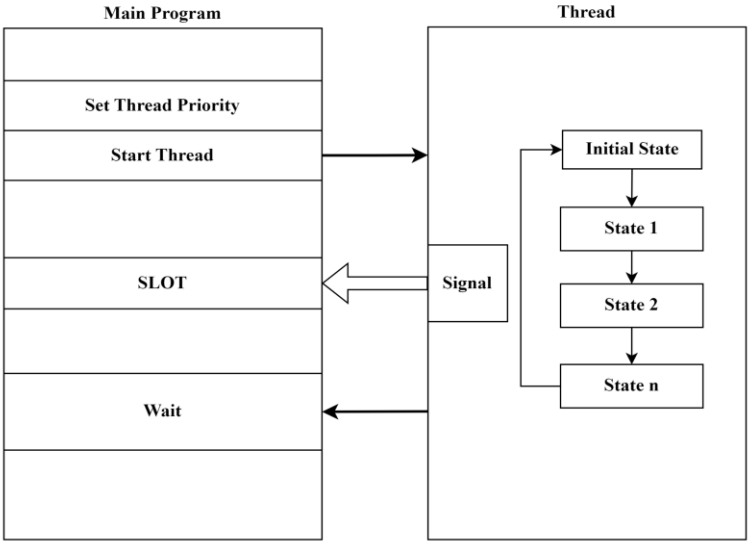

Figure 3: Setting up a State Machine inside a Thread

I used QThread[2] and QStatemachine[3] from the QT framework to implement this. The reason is that QT offers an excellent non-blocking Signal-Slot mechanism that can be used to transmit data from the thread to the main process. Implementation starts by creating the user thread class derived from the QThread class and implementing the State Machine in the run function of the user thread class. QT provides well-defined classes for State(QState) and a simple way to define the state transitions for the QStateMachine. If data or messages are generated that are expected to be transmitted to the parent process while running tasks in the States, these data or messages can be transmitted to the parent process using Signal. In the parent process, these signals emitted by the thread are captured, and the data is processed by the Slots connected to these Signals.

Conclusion:

Threading the State Machine is a useful technique for implementing a system like a real-time backend application with some form of state-driven data source that is not critical to the application. This technique is helpful when it is required to run the State Machine tasks with a low priority. Using QThread and QStateMachine to implement this technique proved to give better performance than running the QStateMachine in the main process.

References:

1. Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167
2. QThread. https://doc.qt.io/qt-5/qthread.html
3. QStateMachine. https://doc.qt.io/qt-5/qstatemachine.html
4. John R. Graham. 2007. Integrating parallel programming techniques into traditional computer science curricula. SIGCSE Bull. 39, 4 (December 2007), 75–78. https://doi.org/10.1145/1345375.1345419