Master of Information and Communication
Electronic Systems

Final Work on Subject:
Wireless Communications

Software Design Patterns For Device Programming in WSN

Author
Vladislav Rykov

Professor
Dr.Gabriel Díaz Orueta

Academic Year 2019/2020

Abstract

Software developers around the world tend to follow the best practices established by the engineering community. Such practices aimed on effectively solving commonly faced problems are called Software Design Patterns.

Since there are many software development areas, i.e. Machine Learning (ML) or Embedded Software, design patterns were polished throughout every area. The area of wireless communication or Wireless Sensor Network (WSN) is not an exception. WSN nodes are devices that require low-level programming, and this work discusses design patterns that will help to organize the development.

Hardware access is a common action during WSN node development. There were conceived many hadrware access design patterns that improve code portability, multiple module support, and hardware modules interactions. Since WSN nodes often are charged with many tasks, concurrency and resource management patterns were made up for those contexts. Nodes actions, more often than not, are constrained to a certain sequence of states, and state machines govern state transitions. Thus, state machines patterns were brought into light.

Present work focuses on those three types of patterns: hardware access, concurrency and resource management, and state machines. The patterns are considered in the WSN area.

Keywords

Wireless Sensor Networks (WSN), Software Design Patterns, Embedded Systems, Firmware

Contents

Chapter 1

Introduction

Contents

The world today experiences unprecedented rates of technology development. Most of the development are related to the Information Technologies (IT) related fields. This exceptional growth was a product of a continuous evolution process started roughly 20 years ago.

During this period software developers faced numerous challenges presented in a form of problems to solve. Many of them were frequently repeated and solved by many talented engineers. Some experienced developers noticed this fact and made an intent to extract those problems with their contexts and solutions. General solutions for such problems were called Software Design Patterns. It is also referred to as formalized best practices [13] using which a developer is able to solve common problems designing software. However, this apparently straightforward formula for good designs can be questioned, and better understanding of the reason behind patterns can be observed.

1.1 Philosophical and Anthropological Foundations

Alan Shalloway and James Trott in [12] found philosophical and even anthropological argumentation to justify the use of patterns. They mention a prominent architect Christopher Alexander who started to question the objectiveness of quality. The question extended to a dilemma if a beauty was pertaining mostly to an individual observer or to a group of observers who would be able to agree that some things are beautiful and some not. In his book The Timeless Way of Building [1], he was projecting these meditations on the architectural field trying to respond if an objective basis to judge that a design is good can exist and be formalized.

Alexander declares that there is such an objective basis within architectural field. The judgement cannot be based on a taste only, a measurable objective basis can be determined.

Additionally, a cultural anthropology convergently points out that within one culture many individuals will agree to a considerably large extent on what is considered to be a good design [3]. In other words, the standards of beauty are established among a certain group of people.

The main point about the design patterns in software systems is a possibility to measure a system characteristics objectively. There certainly are some things present in a good design that make it good and in a bad design that make it bad.

Generally, every pattern can be described through its name, purpose that consists of a problem and how the solution is achieved, and resources that must be considered to reach the solution.

1.2 Design Patterns Emergence in Software

The mentioned work of Christopher Alexander, The Timeless Way of Building [1], affected many bright minds from the software development area. In 1990s, they started to ask themselves if described in Alexander's bool postulates can be applied to the software development as well.

In fact, many people were working on software design patterns during 1990s, but the first four whose work produced the greatest impact were Gamma, Helm, Johnson, and Vlissides. Their work called Design Patterns: Elements of Reusable Object-Oriented Software [6] ganged throughout decades and influenced generations of developers.

It is worth to note that [6] did not invent patterns, but they just identified them from thousand lines of existing code. Those patterns were based on what at that moment was considered high-quality solutions for particular problems.

A more complete list of features for a design pattern is presented in the following listing.

- **Name**

  Unique name is required for identification.

- **Intent**

  The pattern's purpose.

- **Problem**

  The problem that a patterns solves.

- **Solution**

  The way a patterns solves the posed problem.

- **Participants and Collaborators**

  Entities and components included in a pattern.

- **Consequences**

  The effects of a pattern application.

- **Implementation**

  The way a pattern is implemented.

- **Generic Structure**

  A diagram that visually shows a pattern's structure.

1.3 What are the Advantages?

There are a couple of obvious and obfuscate reasons for design patterns consideration during solution development.

The first reason is to reuse solutions. Doing so brings an initial or 'from-where-to-start' point during the solution development and helps to avoid hitches. A true way of wisdom is to learn from someone's experience. Applying design patterns a developer learns from experience of bright programmers. Finally, there is no need to reinvent the wheel. The same solutions are applied to the same kind of problems.

The second reason is to establish common terminology. This point is highly valuable when a team of developers works on a project. Using the same illustrations, vocabulary, common viewpoint of the problem will potentially improve the communication among team members.

The third reason is a possibility to improve code maintenance and modifiability. A positive consequence of applying patterns is a cleaner, better organized, easier modifiable code. Patterns are time-proven and tested solutions ready for potential future modifications.

Summarizing stated points, design patterns broaden developer's perspective on a problem and on how a solution can be designed and help to maintain a particular solution open for possible improvements and modifications.

1.4 Design Patterns and WSN

As it is known, wireless sensor nodes are the central part in a WSN [4]. These small devices are in charge of sensing, processing, and communication functions. The importance of high-quality firmware implementation for a WSN node should not be underestimated.

The WSN applications often require nodes installation on remote and not easily accessible locations. Sometimes the nodes are thrown into a field and some of them may result damaged and inoperative. Of course, an efficient network management must be applied to tackle

such situations. However, it must not escape from the readers attention that the very network management is implemented using a programming language on the nodes' side. Consequently, application of time-proven excellency approaches as design patterns will bring more reliability and efficiency to management functions of nodes.

An example of a node's architecture will help to identify and demonstrate more clearly what are the most common problems that the WSN programmers face coding nodes. Figure 1.1 represents a typical WSN node's architecture. Three subsystems can be observed: sensor, processor, and communication. Processor subsystem is in charge of executing instructions in order to orchestrate sensing and communication subsystems. It is the place where the program code is stored and main tasks execution carried out.

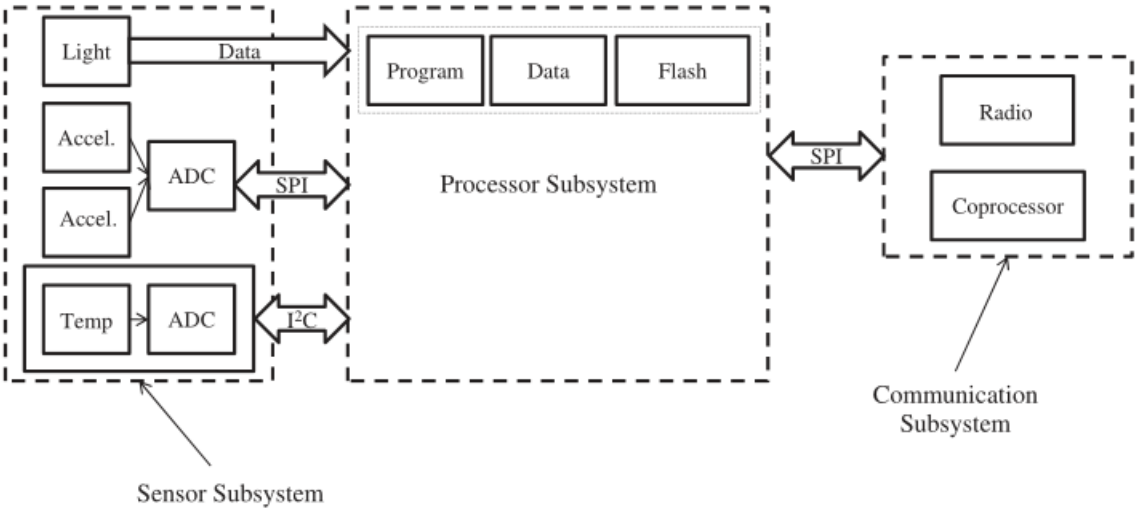

Figure 1.1: A common WSN node architecture [4]

Sensing subsystem helps to interface the system with the real physical environment. It is interfaced with the processing subsystem through well-defined commonly-agreed standard protocols such as Inter-Integrated Circuit (I^2C) [8] or Serial Peripheral Interface (SPI) [7]. Alternatively, some sensors can be straightforwardly connected to internal processing subsystem peripherals as Analog-to-Digital Converter (ADC) or General-Purpose Input/Output (GPIO).

All of these sensors must be handled by the processing subsystem. Sensors' internal registers need to be written for configuration and read for sampling. In other words, hardware must be accessed. There is a number of important hardware access design patterns that alleviate node's operation. Hardware Proxy, Hardware Adapter, Mediator, and Observer are the most useful re-appearing solutions for common sensing subsystem problems. Chapter 3 will discuss them in detail.

Communication subsystem is another point to consider. Although, it is a distinct to sensing subsystem from hardware and functional viewpoints, the way it is interfaced with the processing subsystem is similar to the sensing subsystem.

Often, nodes in a WSN must simultaneously handle different subsystems, i.e. sensing and communication. It brings forth multiple programming challenges related to concurrency and resource management. Cyclic Executive, Static Priority, Queuing, and Rendezvous are pat-

terns that help to solve common challenges in resource management and concurrency. Chapter 4 meticulously deals with them.

Finally, WSN nodes usually have certain fixed behaviour. Periodically, they sense the environment, communicate data to the access point node, put the communication module in receive mode to serve as relays for other nodes, or go to low-power consumption sleep modes. Periods for every action are not necessarily the same. Those actions can be associated with node's states and are managed by state machines. If some relatively complex network protocol is deployed in a WSN, then even several state machines can coexist in the node.

Since it is usual to find state machines in WSN, Chapter 5 will be dedicated to the patterns that deal with typical problems in state machines implementation and management. Such patterns as Single Event Receptor, Multiple Event Receptor, State Table, and State patterns will be discussed.

1.5 Document Structure

Chapter 1 is dedicated to the introduction. Background and preceding ideas are exposed as foundation for design patterns usefulness. The advantages and the way design patterns can be applied in WSN are addressed.

Chapter 2 discusses language peculiarities and strategies for applying design patterns in non-object-oriented languages as C.

Chapters 3, 4 and 5 describe hardware access, concurrency and resource management, and state machines patterns respectively.

To conclude, Chapter 6 makes a conclusive meditation.

Chapter 2

Object-Oriented nature of patterns and C language

Contents

Microcontrollers (MCU) are the most suitable platform for WSN nodes [4]. Most MCUs are programmed using C programming language which is not object oriented. It does not inherently supports polymorphism neither subclassing.

C language offers programmers procedures, a set of primitive actions that lead to a concrete achievement, to form the behavioural aspect of the system (WSN node). They are usually called synchronously, though employing certain advanced techniques can be executed asynchronously as well.

Structures or C `struct` data types bring certain flexibility to a programmer allowing generally to organize data. Special case of C structures are bit-field which provides an abstraction to address a particular hardware device registers in a simple way.

However the design patterns normally require object-oriented programming (OOP) language. OOP languages fuse data attributes and procedures and encapsulate them into a structure called a class. Thus, a more versatile data structure appears and can be employed for the sake of abstractions.

In order to take advantage from the design patterns there must be a way in C language to emulate the most important properties of OOP languages. It is possible, and this chapter will briefly demonstrate it.

2.1 Classes and Objects

Classes can be represented in C using a `struct` that contains data attributes and pointers to functions (methods).

One way to create a class is using two separate files with public members located in a header file while the private variables and functions implementation in an implementation file.

To add more flexibility `struct` pointers that represent class objects will be passed to functions pertaining to the object's implementation as attributes. It will give a possibility to have multiple objects of the same class.

2.2 Polymorphism

Polymorphism is an OOP feature only and cannot be well-supported by the C language. It consists in a possibility of an object to have a function which has one implementation in one context and another implementation in a different context.

The only way to mimic this feature in C is sing conditional `if` or `switch` statements. This workout obliges to know in advance all possible contexts where the function will be executed.

Making a case where a communication module's interface of a WSN node is different, consider a function `communication_module_send` in the following listing

```
uint8_t communication_module_send(communication_module_t *me) {
    switch(me->interface_type) {
    case SPI:
        /* SPI access implementation */
        break;
    case: UART:
        /* UART access implementation */
        break;
    };
}
```

Depending on a kind of interface of a concrete communication module `SPI` or `UART` function will be executed. It is fairly poor polymorphism workout since in the real polymorphism interface types are not required beforehand.

2.3 Inheritance

Inheritance is an OOP language property that allows one class (parent) to be extended or specialized into another (child). Child class inherits the features of a parent class.

The possibility to reuse code is one of the main advantages of using inheritance. Code can be extended without modifying the parent class. Alternatively, some functions of the parent class can be redefined in the child class.

There are different ways to implement inheritance in C. A straightforward approach would be using member function pointers stored in a `struct` object as a way of specializing the parent class. Furthermore, the child class `struct` will contain parent's `struct`, and, in this manner, the parent class will be extended.

Sticking with an example of a communication module from the previous section, a good case would be a definition of a fixed communication module interface used in the main application code and concrete implementations of its two children - i.e. LoRa and ZigBee communication modules. In this fashion, the main code does not need to know which module will be invoked for data transmission. The developer will decide it just by changing a couple of lines of code.

This example depicts a concrete hardware access pattern called Hardware Adapter that will be described in the next chapter.

Chapter 3

Hardware Access

Contents

It is inherent property of all WSN node systems to access and manage hardware directly. As it was shown previously a WSN node consists of three parts - processing, sensor and communication subsystems.

Processing subsystem offers numerous infrastructural features as CPU, memory, timers, internal ADCs, GPIO, interrupts and so on. Communication and sensor subsystems require corresponding mechanisms and protocols that allow data extraction and transmission.

All of these subsystems must be initialized, configured, and tested. There are a plenty of ways to carried it out. This chapter presents design patterns that contribute to a smooth and maintainable hardware manipulation.

3.1 Hardware Proxy

Hardware Proxy Pattern's idea resides in an encapsulation of hardware implementation details into a software-object through which it can be accessed.

A `struct` is used to manage the access to hardware independently from its physical interface. Actual hardware can be anything from a communication module, sensor to an interrupt. Basically, Hardware Proxy exposes module services for reading from and writing into device memory. Moreover, it extends to managing the device and allowing its reconfiguration and operational changes.

Hardware Proxy removes issues related to direct hardware access. When an application is

designed directly accessing hardware, hardware changes will produce a need for serious code modifications of entire application. Hardware Proxy exposed to an application limits the effects of hardware changes, and thus, improves project maintainability.

Figure 3.1 provides an example of a communication interface problem put on Hardware Proxy rails. It is a simple straightforward pattern often intuitively employed in many Arduino projects.

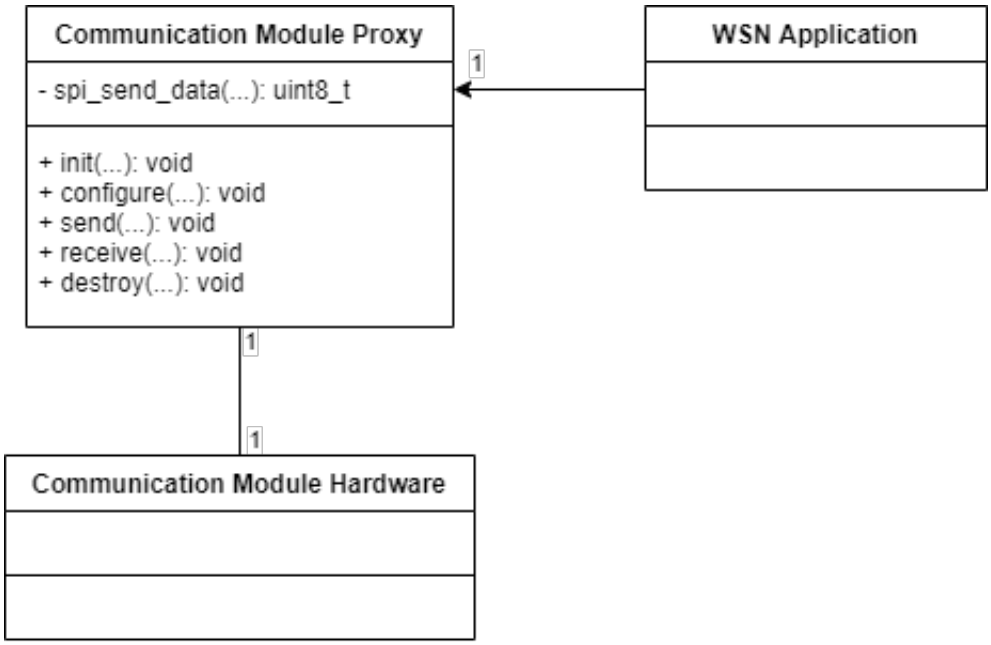

Figure 3.1: Hardware Proxy Pattern example

WSN application uses services offered by the communication module proxy. Communication module hardware is included in the figure for the sake of completeness, it does not represent software. Communication module proxy contains public and private data and functions that are adapted for a specific communication module.

The advantages of applying this pattern encompass all encapsulation benefits. It brings a good degree of flexibility since the WSN application does not have to care about which hardware is used underneath. All these details are placed in the Hardware Proxy. Unfortunately, for hard real-time systems it can have a negative impact since an additional abstraction layer is added and respective function calls and stack operations must be performed.

3.2 Hardware Adapter

Hardware Adapter Pattern allows to adapt already implemented hardware interface to another existent WSN application. It can be considered as an extension of Hardware Proxy Pattern and is useful in situations when the hardware provides one interface and the WSN application requires another. It consists in creation of an element that makes compatible the hardware interface with the application.

Hardware with comparable functions usually have similar interfaces, though, some functions may differ. Instead of re-implementing the application, an adapter with matching interface is created. The application uses created interface unaware of the included layer of adaption operations.

This pattern is useful when hardware may perform application required functions but has incompatible interface. Applying this pattern, required application modifications related to hardware changes will be reduced to minimum.

A virtual example for this pattern would be a situation where one temperature sensor might be replaced with a different one. The difference between both sensors would be an output format - Celsius and Fahrenheit. Figure 3.2 illustrates the diagram for this example.

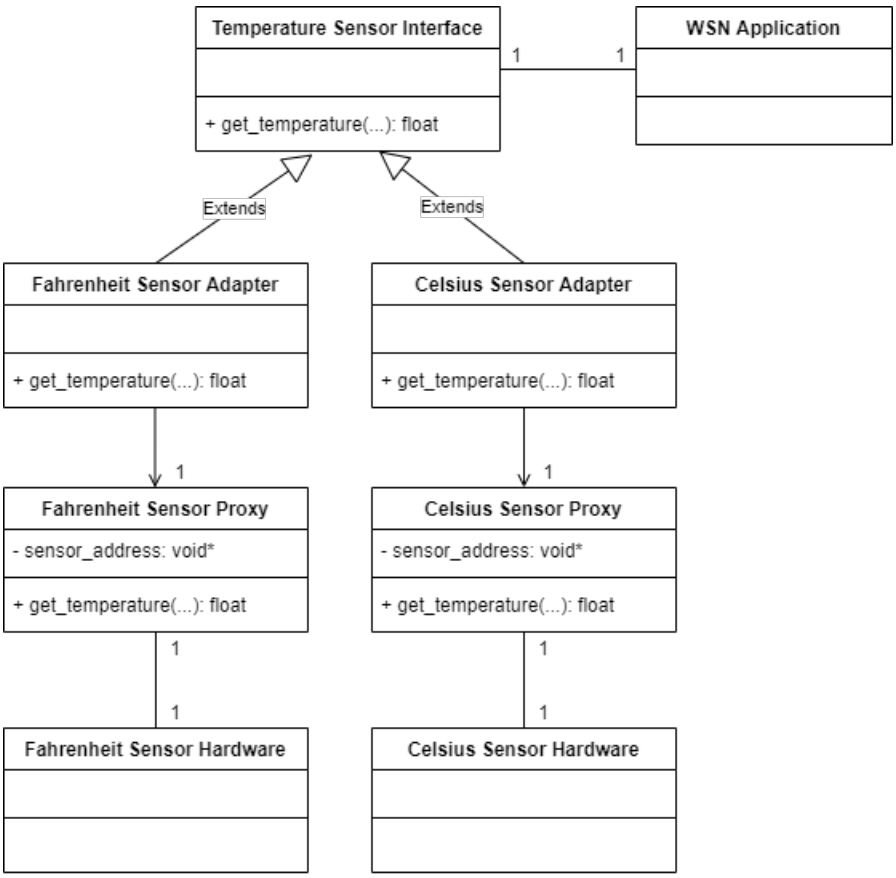

Figure 3.2: Hardware Adapter Pattern example

WSN application uses a fixed-defined interface that is extended by different adapters. Each adapter produces the application-specific output.

3.3 Mediator

Mediator Pattern bring up a way to organize complex interactions among a set of components. It appears to be quite useful when WSN node's hardware components must be coor-

dinated in a clearly defined but complicated ways.

This pattern is normally applied when a device serves primarily as an actuator. Despite it is unusual in WSN, it will be useful for a small set of WSN applications that can involve such operations.

The idea is to use a mediator class that will organize actions of a set of hardware components. Those hardware components are specific collaborators capable to notify the mediator of specific events occurrence. Figure 3.3 shows the generic pattern structure.

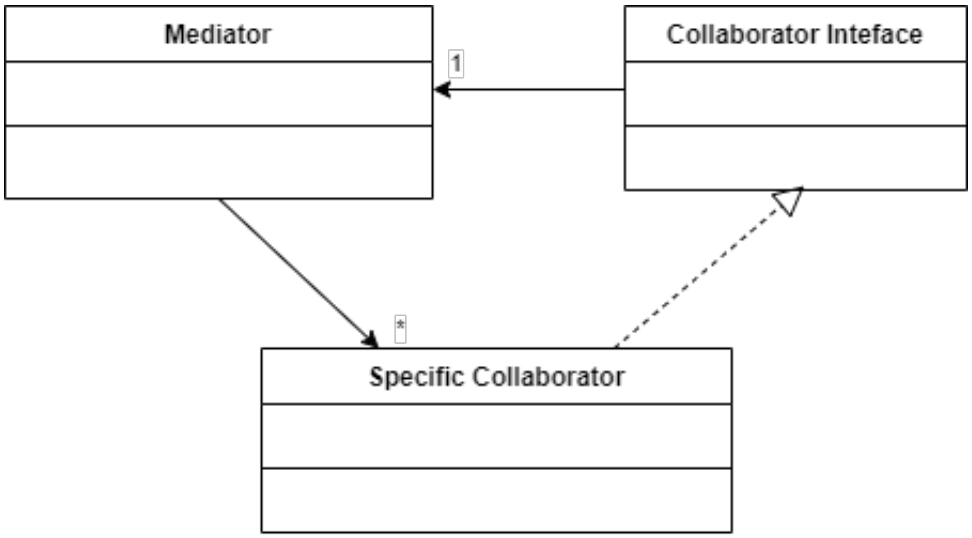

Figure 3.3: Mediator Pattern structure

Since the mediator coordinates the specific collaborators the overall design is simplified. The points of specific collaborators couplings reduced and all the coordination is concentrated in one object. Instead of a specific collaborator directly notifying another specific collaborators if some event of interest occurs, it notifies the mediator which is able to respond to the event in a wider system-wide fashion.

3.4 Observer or Publish/Subscribe

Observer is a very common pattern which can be associated to various scenarios inside as well as outside WSN. It brings possibility to avoid simple polling strategy and substitute it by providing clients with subscribing capability. In WSN is can be used for share important sensor data among different node's subsystems.

Clients will receive notifications about relevant for them data updates. They simply offer their subscription functions that will permit them to dynamically subscribe to and unsubscribe from the notification list. The data server can coordinate the notifications in different manner as the developer would desire. Clients can be notified periodically, when certain condition is met, or on new data arrival.

An example for this pattern application would be a fire alarm with sprinklers WSN. Figure 3.4 provides the pattern structure. It can be deployed in big warehouses for fire detection and extinguishing. A node will combine a smoke detector, sprinkler actuator, and the communication subsystems. Sprinkler actuator and the communication subsystems will subscribe to the smoke detector data.

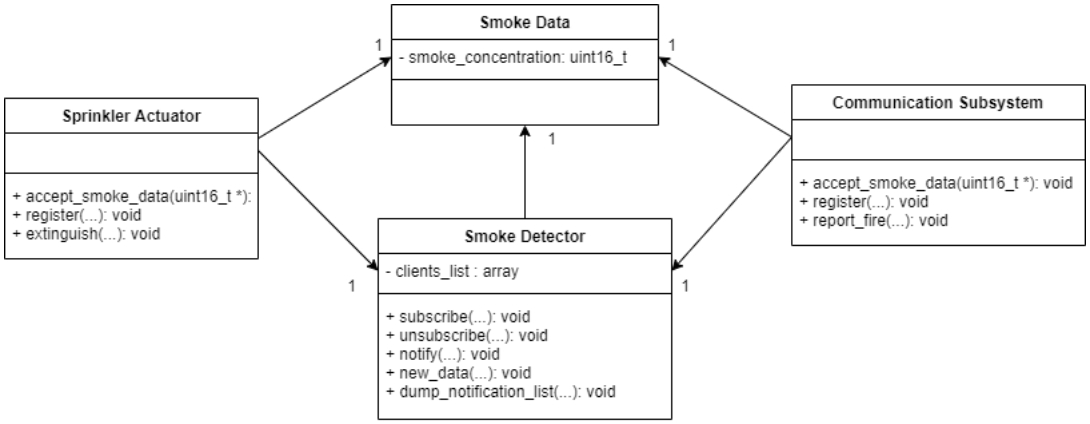

Figure 3.4: Observer fire protection WSN example WSN

Unfortunately, implementation details will not be included in the current work due to the time constraints.

Observer Patterns makes easier data distribution among different clients. It resembles a client/server paradigm with a configuration flexibility mechanisms. It allows to reduce computing requirements of the node since the interested subsystems receive data when it become available.

Chapter 4

Concurrency and Resource Management

Contents

Often WSN nodes have to perform concurrently multiple actions or tasks. Network protocol state updating and sensing the physical environment which can comprise multiple tasks, event detection routines are threads that must be executed in parallel.

In order to fulfill the requirement of concurrent tasks execution there are many options that can be exploited. Among them the most common ones were selected and presented in this chapter.

4.1 Cyclic Executive

Cyclic Executive Pattern is a fairly simple approach to schedule multiple tasks. Its simplicity is its main advantage, though it is a quite rigid strategy and that is likely to loose an urgent deadline. Fairness is incorporated here giving all tasks the same opportunity to run.

When it is determined what mission a WSN will have and which tasks the nodes will have to perform, it will be possible to analyze if this pattern will be suitable. The deadline for each task should be estimated and must be greater than the sum of all worst-case executions of the preceding tasks.

This pattern does not bring the real concurrency, but rather a pseudoconcurrency. The scheduler is just a simple infinite loop that calls tasks sequentially. Tasks are functions executed by

the scheduler until `return` statement.

Figure 4.1 presents a simple Cyclic Executive pattern. The Cyclic Executive agent serves a scheduler with a control loop calling sequentially the tasks.

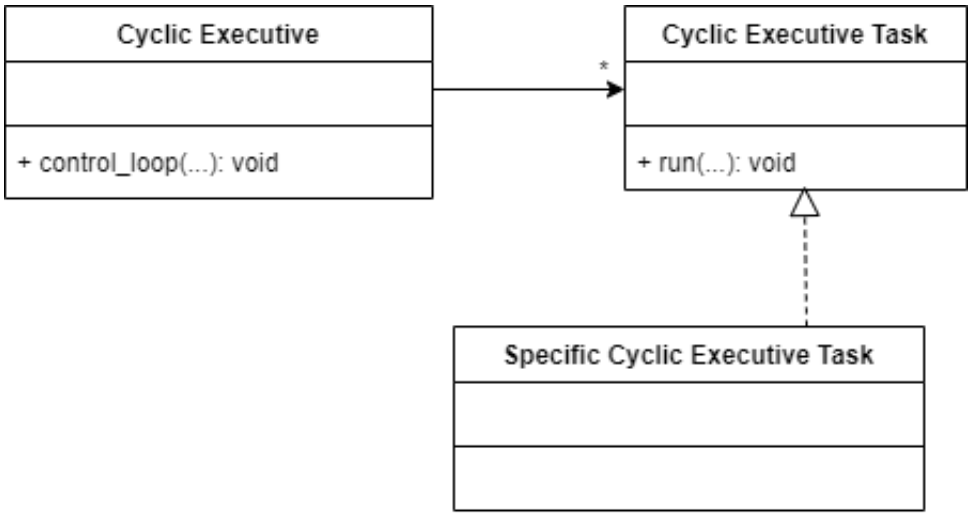

Figure 4.1: Cyclic Executive structure

Since it can be easily certified, it is widely used in high-safety systems [5]. On the other hand, if node's hardware does not allow to run a full-fledged Real-Time Operating System (RTOS), it will serve as a good substituent.

4.2 Static Priority

Static Priority Pattern is implemented by the most RTOS since it has a considerably good responsiveness for tasks with high priority. Additionally, it performs well with a large number of tasks.

There are different criteria for priorities assignment. Usually, they are assigned according to event criticality or urgency. There is an important condition that the system must meet in order to know if the tasks can be scheduled successfully. Equation 4.1 expresses the condition [9].

$$\sum_{j} \frac{C_j}{Tj} + max(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}) \leq utilization(n) = n(2^{\frac{1}{n}} - 1) \tag{4.1}$$

where

- C_j is the worst case execution time for task j
- T_j is the time period for task j

- B_j is the worst case time task j can remain blocked
- n is the total amount of tasks

An important assumption is made that all tasks are periodic. However, in case of aperiodic tasks, minimum time between successive calls should be estimated and used.

Static Priority is widely supported by virtually any RTOS. It is a good match for small WSN nodes which tasks are extremely predictable and almost no aperiodic tasks are present. Schedulability can be easily analyzed during the compile time.

Figure 4.2 presents a generic structure of the Static Priority pattern. Static Task Control Block is matched with corresponding Static Priority Task, and it holds the information necessary for preemption and scheduling support. Additionally it contains a `start_address` value that stores the starting address of that task or the address where it has been preempted. Static Priority Scheduler organizes the whole execution process always running ready tasks with the highest priority. Every task must have a Stack for storing return addresses and passed functions parameters. Priority Based Queue is a queue of Static Task Control Blocks ordered by priority. Two queues are required for management of ready and blocked tasks. Finally Mutex and Shared Resource elements were added for diagram completeness and awareness of these system components.

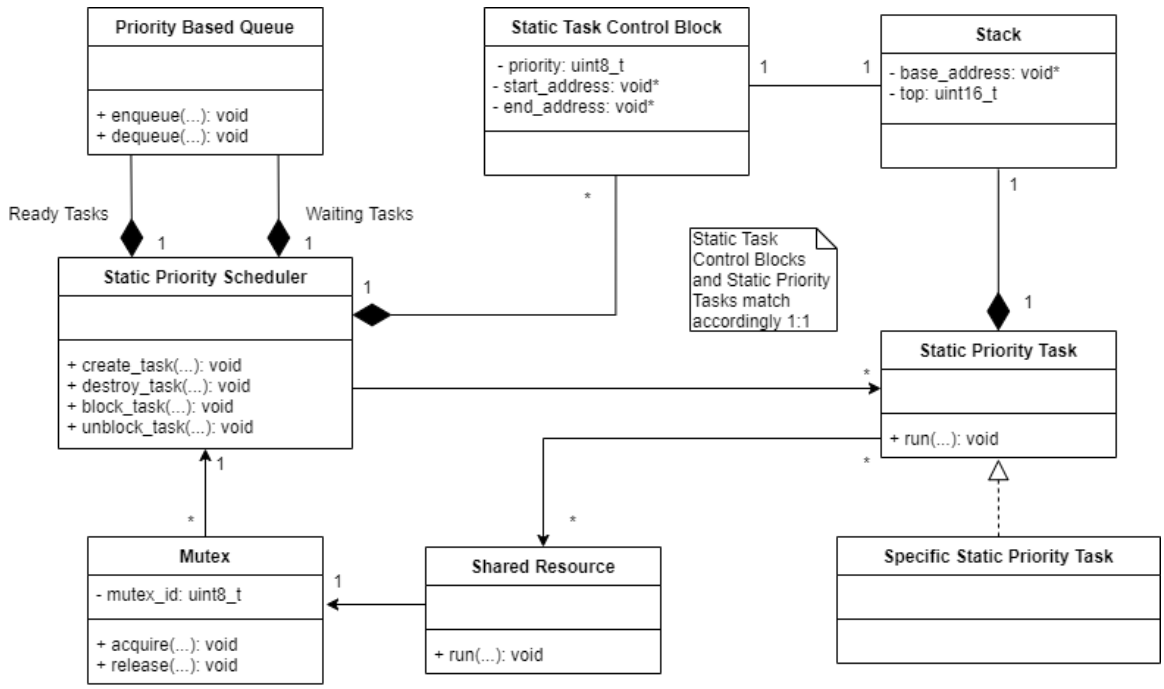

Figure 4.2: Static Priority Pattern structure

A straightforward implementation of this pattern will have shared resource management issue known as Priority Inversion [2]. It is a setting in which a high-priority task must wait for a low-priority task to release the shared resource. Appropriate measures such as in described in [10] and [11] must be taken to eradicate this dangerous scenario.

4.3 Queuing

Queuing Pattern provides a commonly employed method of asynchronous tasks communication. It consists of a first-in-first-out data structure shared between the sender and receiver. The sender stores a message in a queue and the receiver polls it at later moment. Alternatively, it can be used as a mutual exclusion mechanism for consistent accesses to shared resources.

It is worth to note that all messages in the queue are passed by value, not by reference. It solves occasional resource corruption issues found in other systems when the messages are passed by reference. The receiver takes the message and makes a local copy. Then, it can be modified in the most appropriate way according to the receiver.

The pattern aims to solve two problems. Primarily, it allows tasks to synchronize and sharing the data as well. Secondly, the shared information is corruption-free, and race conditions are avoided.

Figure 4.3 shows the Queuing Pattern simple structure. Message Queue stores the data and coordinates exclusive accesses to the data using the Mutex. The Messages is shared among many Queuing Tasks. When a Queuing Task wants to read or write data it acquires the Mutex, perform the action, and releases the Mutex.

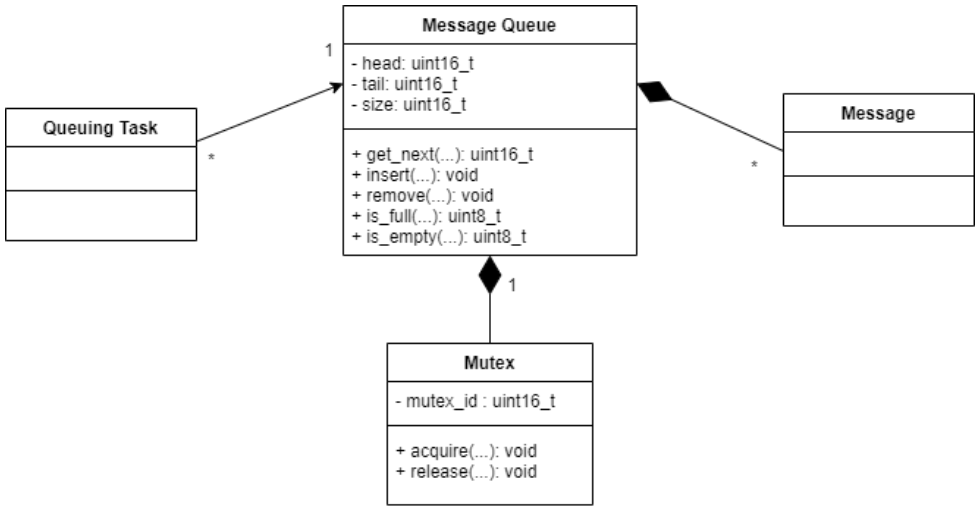

Figure 4.3: Queuing Pattern structure

A negative side of the pattern is that there is no Observer Pattern incorporated. It can lead to possible data processing delays. Another downside is that the queue size can possibly exceed the available memory. It can be remedied establishing a fixed queue size that must be estimated according to adequate throughput requirements and node's memory capacity.

4.4 Rendezvous

There exist different ways for tasks synchronization, semaphores and queuing are two very common examples. These synchronization examples are based on a sharing data or a simple function call. However, they are not always the most convenient ways. There can be cases where a more complex conditional synchronization needed. Rendezvous Pattern provides a time-proven solution that accommodates the most cases for complex synchronization.

It is based on concept of a precondition, a condition that is specified to be met before an action happens. This pattern allows a developer to define preconditions for task synchronizations. It helps to ensure that a defined set of preconditions is met during an application execution.

Figure 4.4 presents the pattern structure. It has a fairly simple structure. Rendezvous object contains data and functions for tasks synchronization. Once a task is initialized it registers in Rendezvous Object calling `register` function. Right after it has been registered it becomes blocked until a condition is met. When necessary to run preconditions are met, then the task is released for running.

There can be implemented different precondition models. It is possible to have only one precondition for all tasks, individual preconditions for every task, or both. Individual preconditions is a much flexible approach, though every WSN application must be examined and the best option might be implemented.

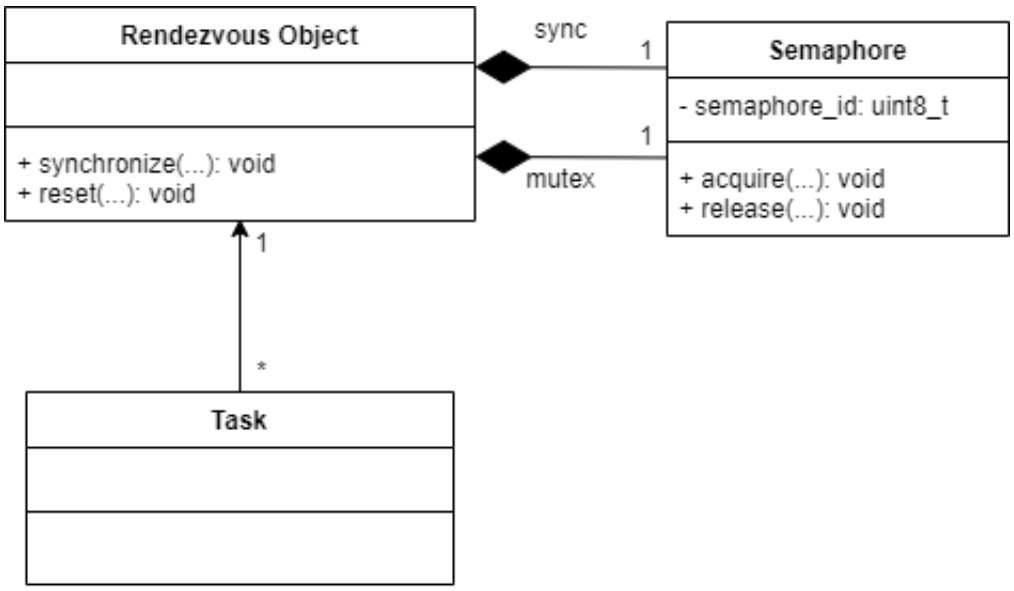

Figure 4.4: Rendezvous Pattern structure

The `synchronize` function of the Rendezvous Object is the core piece of code that will determine when the tasks will be released. A possible implementation can be realized using the Observer pattern notifying the tasks to run when the precondition is met.

Chapter 5

State Machines

Contents

Every device in a WSN follows a particular model of behaviour according to its purpose. Once programmed, they decide by themselves what, how, and when to take actions. A WSN node must understand the network state and behave respectively. It should know when to get data from sensors and when to send them. How it can be described and programmed? What is the most clear and concise way to design a WSN node that must have a particular behaviour?

The answer to all these question is using state machines. A Finite State Machine (FSM) formally represents a directed graph that has three main components: states, transactions, and actions. A state is a condition that describes device's current setting. A transition is a trail from one state to another. It is often triggered by an external intervention or when some condition is met. An action represents a concrete sequence of actions carried out by the system. Actions can be performed on state entry or exit, or on transitions.

Most WSN nodes are delimited by stateful behaviour. That is, the node's behavior will depend upon freshly extracted data that imply change in a physical environment or timing event related to the network state and the current state in which the node remains.

FSM can be based on OR-States or AND-States. OR-State FSM poses a rigid rule that the system can be in one and only one state in a given moment. OR-State FSM can be added additional level of abstraction allowing nesting states inside a state. But the main rule stated previously cannot be violated. AND-State FSM is a more complex kind of FSM that adapts better to the real world situations. It is suitable when the system has independent features. Consider a WSN that must follow the network state and detect certain event which is not trivially derived, that is, the event detection involves multiple phases or states. Then, two independent features are present and the node state is expressed as a combination of the net-

work and event-detection states. AND-State FSM are complex to describe and will not be convered in the current work.

This chapter discovers the behavioral design patterns that are most commonly found in practice.

5.1 Single Event Receptor

Single Event Receptor Pattern exposes a single event reception interface that allows to link or communicate the state machine with the clients. The event receptor must accept a received event data type and data that come therewith. It can be employed for synchronous and asynchronous events.

The pattern structure for asynchronous events will slightly differ from that of synchronous. It will have a queue for storing the events, and the state machine class will have an infinite loop that traverses the queue. Synchronous version will directly receive events and take corresponding actions.

As an example will be used a WSN network protocol that has three states: Initialization, Global Wakeup, and Right-Hand Rule Forwarding. Initialization state will have two substates: Discovery and Sync. Figure 5.1 represents the pattern structure for synchronous events and Figure 5.2 for asynchronous events.

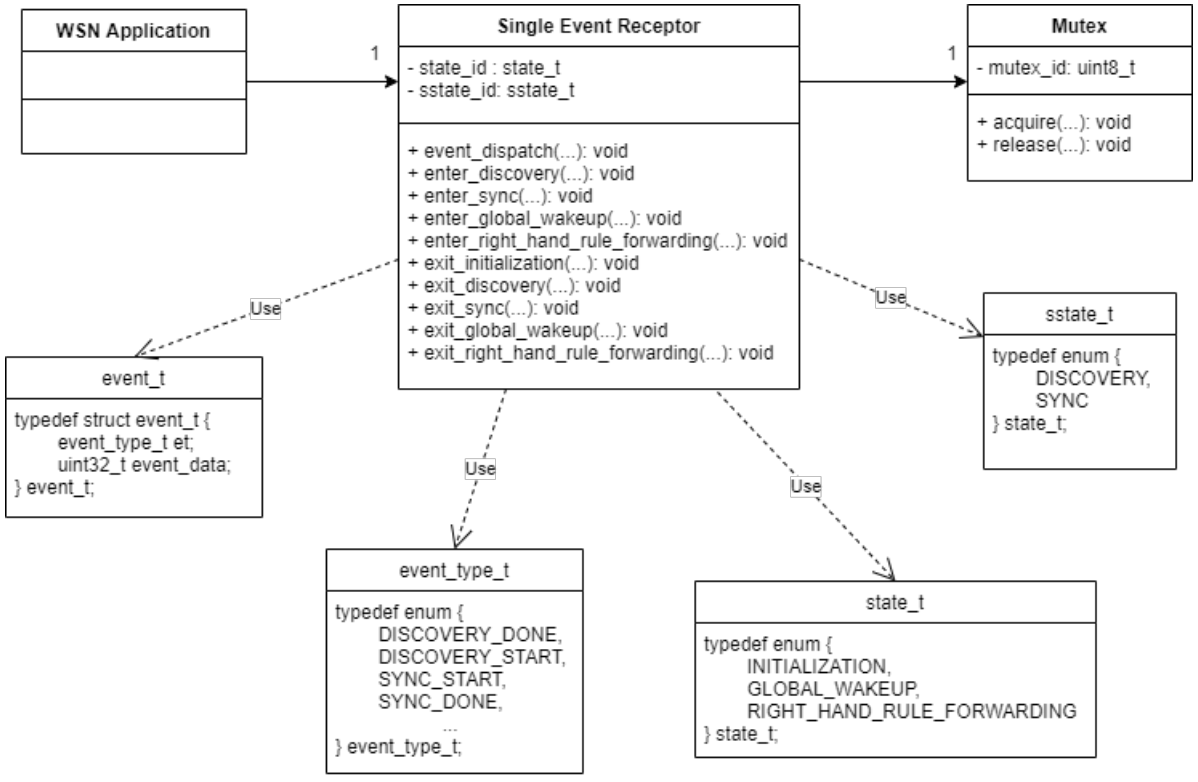

Figure 5.1: Single Event Receptor for synchronous events structure

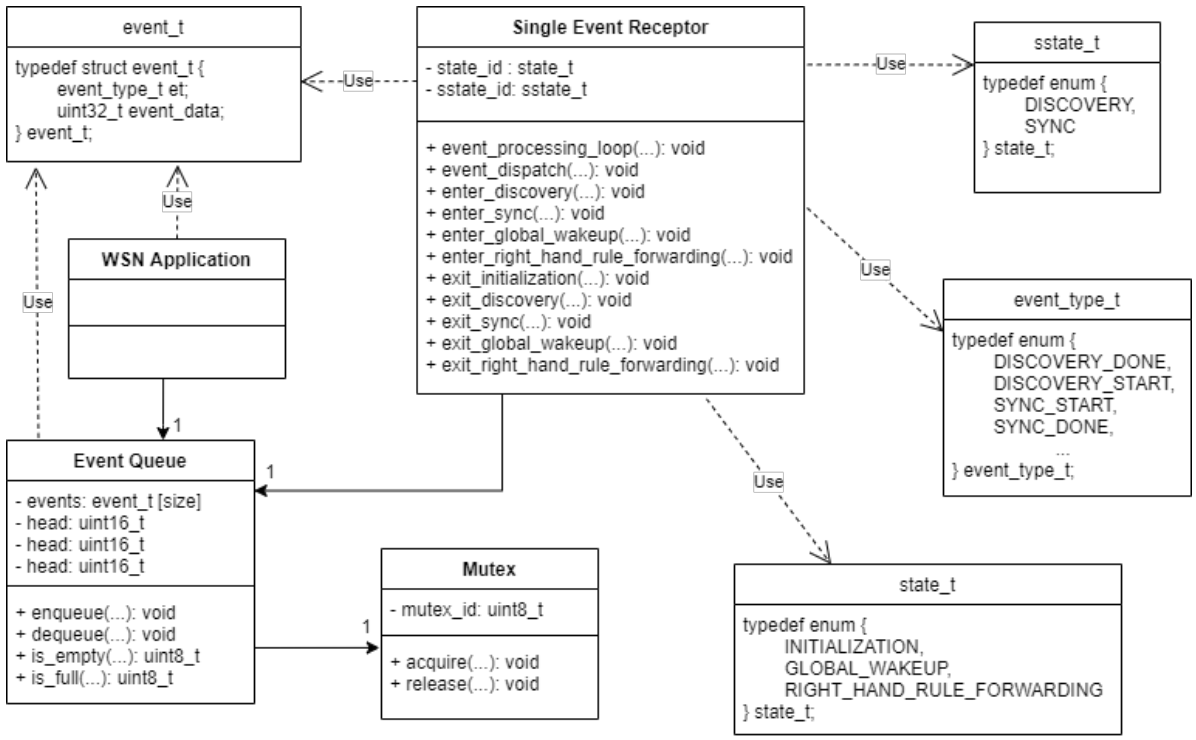

Figure 5.2: Single Event Receptor for asynchronous events structure

Single Event Receptor is the actor that holds the implementation of the state machine. It contains state variables that define the current machine state and functions that are used for event dispatch and for invocation on previous state exit and next state entry. The asynchronous version is differentiates that the WSN application enqueues events into Events Queue and the Single Event Receptor has additional function `event_processing_loop` for traversing the Event Queue.

Mutex is used in distinct manners. For asynchronous events it maintains data coherency of the queue. For synchronous events it guards the Single Queue Receptor when state entrance of exit is being performed.

A very common implementation strategy for this pattern is to create a huge `switch` structure that handles all the events inside the `event_dispatch` function.

A downside of this approach lays in the fact the all logic is encapsulated in the Single Event Receptor what limits the system scalability.

5.2 Multiple Event Receptor

Multiple Event Receptor Pattern has a very similar approach to the Single Event Receptor Pattern. However, it is used practically only for synchronous events. This peculiarity is given due to the fact a WSN application will know which events it will send to the state machine. Thus, instead of having a common `event_dispatch` function it will have a dedicated func-

tion for every state that will be invoked from the application when the event occurs. In this manner, there is a single event receptor point for every event issued from the application.

It is the most common type of implementation of synchronous event state machines. Since each event receptor function takes actions dedicated to the event only according the current system state, it has simpler implementation than the Single Event Receptor.

Figure 5.3 presents the Multiple Event Receptor Pattern applied to the same WSN protocol example as described in the previous section.

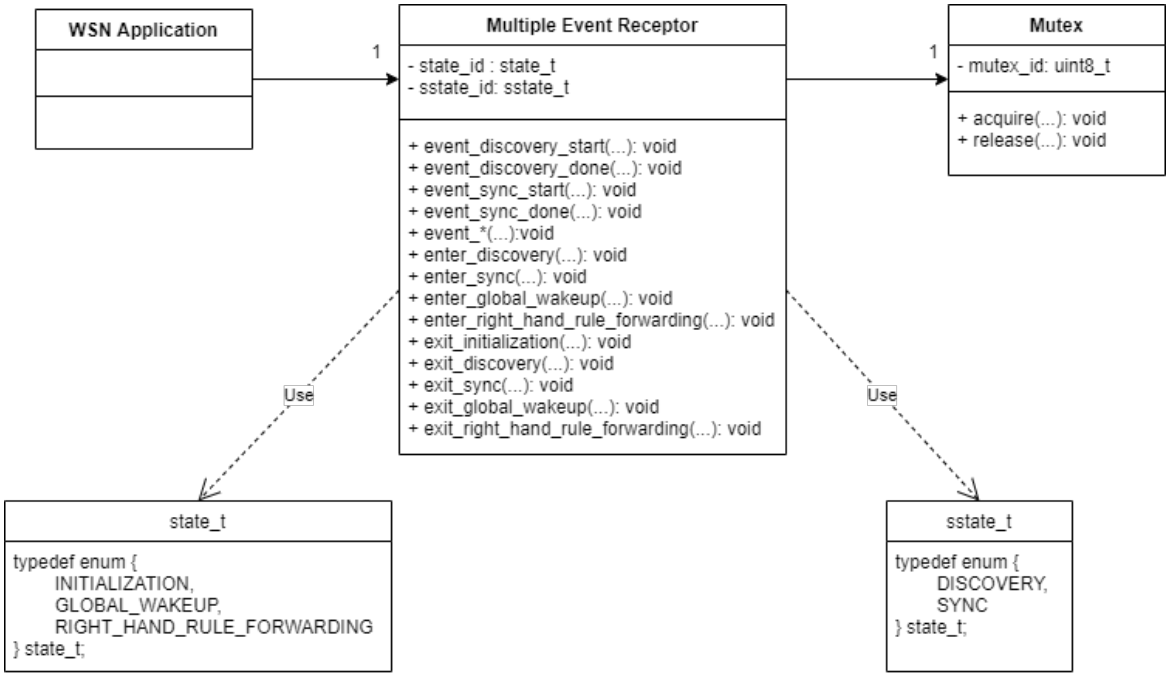

Figure 5.3: Multiple Event Receptor Pattern applied to WSN protocol example

It can be appreciated that Multiple Event Receptor has `event_*` receptor function for each possible system state and state transition functions with associated actions. There is no need for `event_type_t` and `event_t` data types since corresponding receptors are directly callable the data is passed as an argument.

Multiple Event Receptor simplifies the actual implementation splitting the `event_dispatch` logic into a number of receptor functions. Every receptor function has a `switch` structure nested and effectuates appropriately to the current state behavior.

5.3 State Table

State Table Pattern was designed for large and flat state machines. It is not suitable for AND-States nor for nesting states. Constant computational complexity performance is guaranteed during the run-time, though with a relatively slow initialization process. State Table is easily extends for new states.

The core of the pattern is a two-dimensional array that stores all necessary information about all state transitions. Current state ID and event ID are indices of the array. Each entry holds a data structure with references to the corresponding actions and guards, and new state IDs. Thus, there is a need for more developed data structures than previously described patterns.

Figure 5.4 presents a structure of the pattern applied to the previously defined WSN protocol example. State Table contains the state machine implementation along with the table containing all relative information for `event_dispatch` function and current system state.

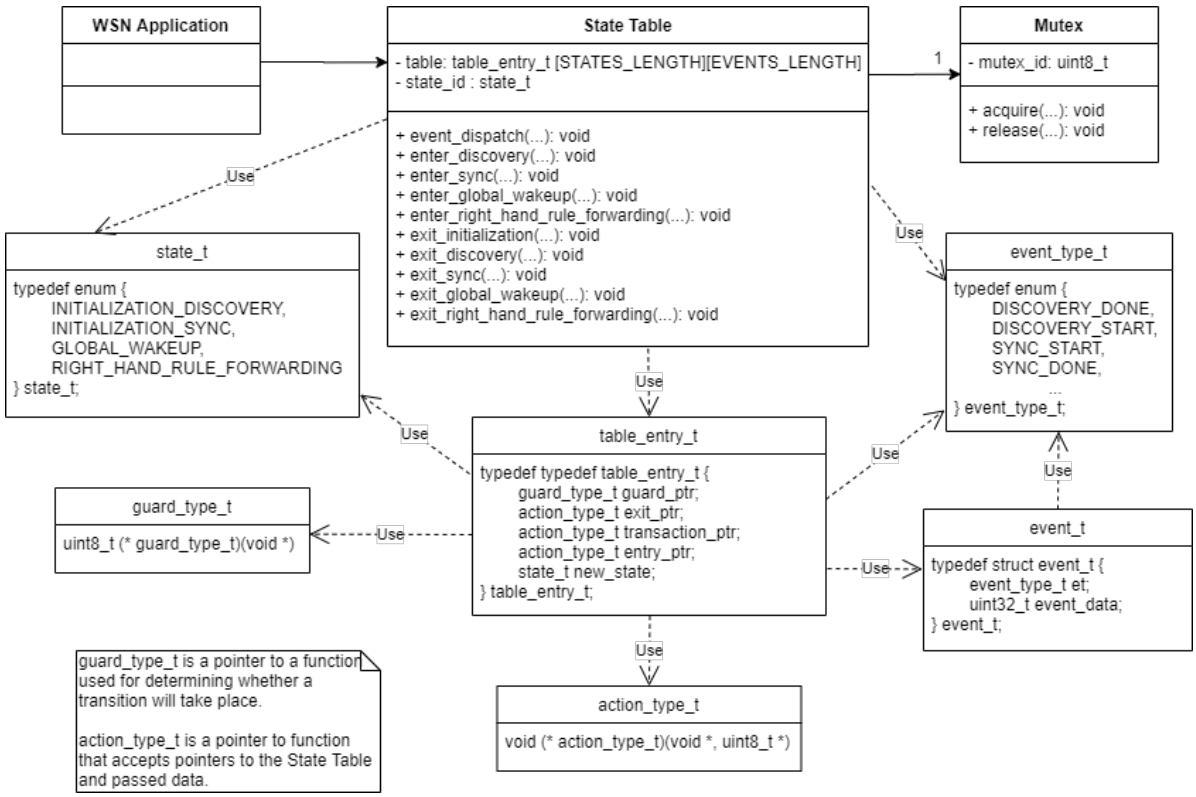

Figure 5.4: State Table Pattern applied to WSN protocol example

It can also be observed how the state machine was flattened compared to the Single Event Receptor Pattern; the sub-states were transferred into the states.

The `event_dispatch` task has a clear task on new event arrival. It will access the `table[state_id][et]` and check the guard condition calling `guard_ptr` function. If it will return 'true', then `exit_ptr`, `transision_ptr`, and `entry_ptr` functions will be executed sequentially and, finally, the `new_state` will be assign to the `state_id` State Table variable. Otherwise, if the guard will return 'false', nothing will happen.

State Table Pattern has a great performance due to the directly indexed `table` of `table_entry_t` entries by `state_id` and `et` (`event_type_t`). However, the `table` must be populated with all necessary pointer values for every state and event combination. It is a time-consuming operation, though, once finished, it will have excellent performance. A potential downside could be a wasted memory if the state space is spare, that is, many states are left unused.

5.4 State

State Pattern allows optimization of run-time performance at cost of data-memory usage or vice versa. It has flexibility to suit dense as well as spare state spaces keeping the initialization time low. Active use of dynamic memory and polymorphism makes it a slightly cumbersome for implementation in C, though, as it has been demonstrated, possible.

The basic idea behind this pattern is to create state objects for each state. Object that will manage the state machine is called Context and stores the list of all state objects along with the index of the current state within the list. Event dispatching receptor functions for every event are grouped into a single structure and passed to the state object that represents the current state.

Every state object implements dedicated to that state behaviour. States thus become easily modifiable and extensible. Figure 5.5 depicts the structure of the pattern applied to the previously mentioned WSN protocol example.

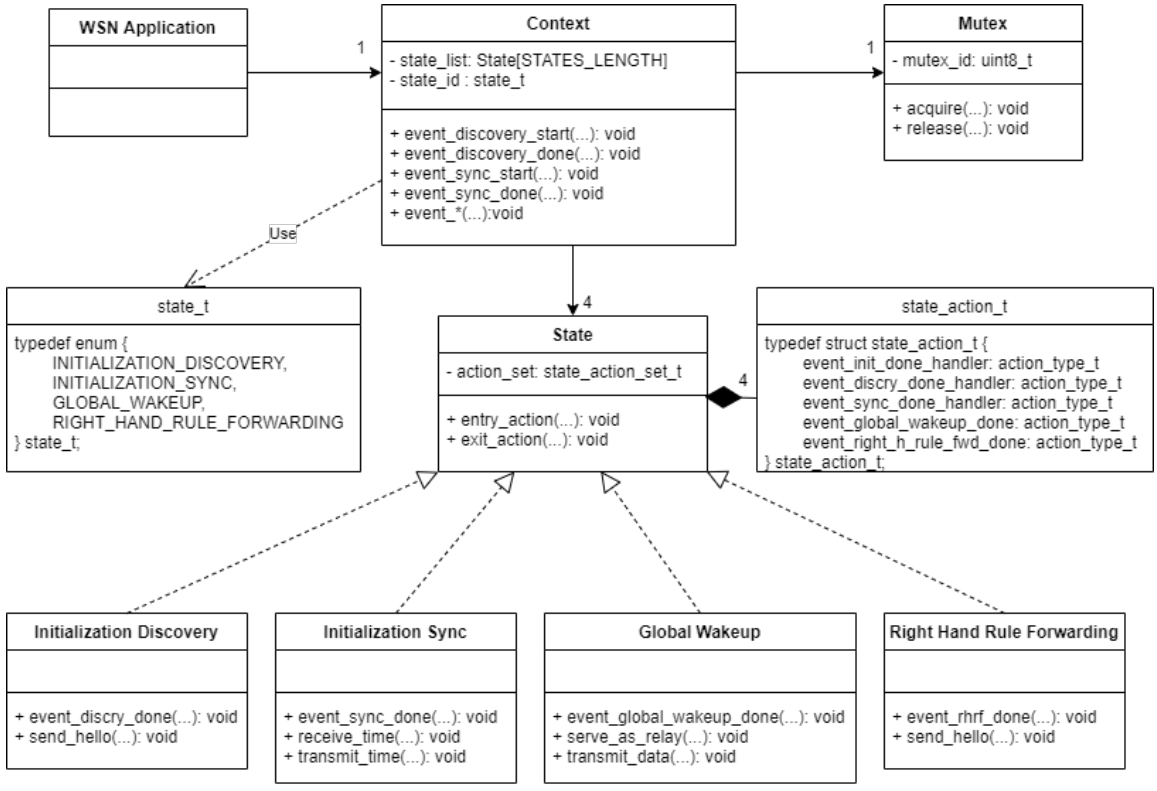

Figure 5.5: State Pattern applied to WSN protocol example

It is observed that all states have the same interface with common to all states functions as well as the specific to state functions. On new event arrival Context passes the control to the corresponding state object. Once all actions have been finished, the control returns back to the Context.

Context object offers a number of event handlers where each of them commissions the control to the current state object. Once the processing has been done, the `state_id` is updated to

a new value, and the control passed back to the Context. Each State has an instance of the `state_action_t` object that stores pointers to every event handler specific it that state. It is set up with all pointers during the initialization. Polimorphism is actively involved during the `action_set` use since each event handler has a pointer (`action_type_t`) that should be dereferenced to execute a corresponding to the current state function. All references were set during state objects initialization.

State Pattern encapsulates the behavior of a state to a single object. It allows to easily extend possible states and introduce modifications effectively. However, it requires more memory than State Table Pattern due to the additional objects with associated functions and data.

Chapter 6

Conclusions

Design patterns ought to be seen as a way for learning from decades of developers' experience. They widen the understanding of a particular problem, help to architect sophisticated yet clear solutions, and should be selected in the light of a problem-specific context. Generally, programming task can even be seen as a puzzle that can be solved recognizing different design patterns that, once put together, lead to a well-designed and highly-reliable solution.

Design patterns from three different areas were presented. Of course, it is not an exhaustive list of existent patterns. Neither all areas were covered. But, given the context of WSN applications, these patterns will help to develop structured solutions related to hardware access, concurrency and resource management, and state machines realms.

Hardware Proxy, Hardware Adapter, Mediator and Observer patterns will allow a developer to effectively design code for accessing the hardware and even sharing the data produced thereby. Cyclic Executive, Static Priority, Queuing, and Rendezvous patterns provide a basis for organized and controlled execution of tasks and additionally allowing their synchronization. Finally, Single Event Receptor, Multiple Event Receptor, State Table, and State patterns provide a plethora of ways the behavior of WSN nodes can be determined and programmed.

Personally, I have considerably enriched my understanding of many common problems found designing firmware, and, as an Embedded Engineer, I will extensively apply those methods whenever it will be suitable for a challenge I will face.

Bibliography

[1] Christopher Alexander. *The timeless way of building*, volume 1. New York: Oxford University Press, 1979.

[2] Özalp Babaoğlu, Keith Marzullo, and Fred B Schneider. A formalization of priority inversion. *Real-Time Systems*, 5(4):285–303, 1993.

[3] Ruth Benedict. The sword and the chrysanthemum, 1946.

[4] Waltenegus Dargie and Christian Poellabauer. *Fundamentals of wireless sensor networks: theory and practice*. John Wiley & Sons, 2010.

[5] Calvin Deutschbein, Tom Fleming, Alan Burns, and Sanjoy Baruah. Multi-core cyclic executives for safety-critical systems. *Science of Computer Programming*, 172:102–116, 2019.

[6] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[7] Motorolla Inc. Spi block guide. `https://web.archive.org/web/20150413003534/http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf`. [Published on January 21, 2000].

[8] NXP Inc. I2c-bus specification and user manual. `https://www.nxp.com/docs/en/user-guide/UM10204.pdf`. [Inquired on September 23, 2020].

[9] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[10] Douglass Locke, Lui Sha, R Rajikumar, John Lehoczky, and Greg Burns. Priority inversion and its control: An experimental investigation. *ACM SIGAda Ada Letters*, 8(7):39–42, 1988.

[11] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[12] Alan Shalloway and James R Trott. *Design patterns explained: A new perspective on object-oriented design, 2/E*. Pearson Education India, 2005.

[13] Wikipedia. Software design pattern. `https://en.wikipedia.org/wiki/Software_design_pattern` (22.09.2020).