Technical Report: Embedded Systems 2 — Reliable and Robust Communication Pattern

Gerald Hilderink – Version V1.0, 2023-03-11

Table of Contents

This technical report is about applied reseach of how to create reliable and robust communication protocols via the serial port (UART). A simple sender and receiver of a message are designed such that they are essential for creating a reliable and robust communication protocols. The designs are based on state diagrams. Simplification techniques are applied that prevent state explosion when designing a communication protocols. Overall, the results of this research is useful for designing reliable and robust communication protocols over serial connections.

This document is an example of a technical report using AsciiDoc. It enables 'quick' writing technical reports in Visual Studio Code.

1. Introduction

The Arduino framework provides the `Serial` object that is used for communication over the serial port. Using patterns, like the ones below, are simple but they do not cover reliable and robust communication. For example, disconnections are not discovered nor handled. One should always take into account that communication via a serial connection can fail for whatever reason. A reliable and robust system must deal with failing communication at all times.

Sender side

```
Serial.print("$LedOn;");
```

Receiver side

```
  if (Serial.available()) {
    String msg = Serial.readStringUntil(‘;’);
    msg = msg.SubString(1); // remove '$' symbol.
    if (msg == “LedOn”) {
      digitalWrite(LedPin, HIGH);
  }
}
```

This observation requires further research to find an adequate solution that *ensures* reliable and robust serial communication. A research question is "What is required to develop a reliable and robust communication protocol?". The property *reliable* means "to be trusted" and this requires the detection and handling of communication problems to recover from disconnections and untrusted communication. The property *robust* means that the system must keep working properly in all circumstances and conditions (relevant to the application). To answer the question it is obvious that all relevant states need to be identified and nothing remains undefined. These properties are engaged during the design process using state-machines. This approach is described in the Procedure section.

The mission is to use state-diagrams. State diagrams are commonly used to discover, identify and structure behaviours. Communication protocols define a set of behaviours involving inputs and output (or sending and receiving messages) which can be designed with state diagrams. The lecture slides are used. The results are described in the Designs section. The designs are implemented in the C-language for embedded boards, like the Arduino Uno and ESP32.

The following requirement has to be taken into account. The performance must be high for embedded systems. The state machines should be unblocking, i.e. they must not delay and multiple state machines must be executed in an interleaving fashion that emulates parallel behaviour to retain real-time performance.

---

2. Procedure

The procedure was based on first design and then implementation — one way street. The design process was driven by the identification and structuring of states and state-transitions for the sender and receiver processes. Our assumption is that the `Serial` object provides the essential functionality of the serial port. It has to be seen if this assumption is to be true. Let’s put this aside and move on to thinking at a more abstract level in order to discover, identify and structure the building blocks for reliable and robust communication.

The use-case that is applied here is simply two Arduino’s communicating over the serial port. One Arduino sends a message "$LedOn;" to command the other arduino to turn on an led. Acknowledgement is an important mechanism to determine if the message was received and processed successfully or not. The design approach will discover what states are required to leave nothing uncertain.

The procedure has been divided into the design approach and the implementation of the functions.

2.1. Design approach

The design starts at the state LEDON at which the message "$LedOn;" will be send. At every state that is identified, the happy flow and unhappy flow (or corner cases) have to be considered. Unhappy flows must cover timeouts, corrupt messages and unknown messages. In Figure 1 in the Design section, the state diagram shows the initial design of the sender. After `SendMessage("$LedOn;")` the acknowledgment has to be received and handled. The functions `ResetTimer()` and `HasTimerExpired()` are used to start a timer and to wait for a timeout. The design stopped at finding the leaf-states. The leaf-states have to be worked out in the application design, which is out of scope. The leaf-states that were discovered are listed in Table 1. It took about 10 iteration to reach these states and being confident about the desired behaviour of the design.

Table 1, Leaf-states.

State name	Description	Flow
NEXT_STATE	The message was received and processed.	Happy
FAILED_ACK	The message was received but could not be processed.	Unhappy
UNKNOWN_ACK	The acknowledge token was unknown.	Unhappy
TIMEOUT_ACK	The acknowledge token was not received within specified time.	Unhappy
CORRUPT_ACK	The acknowledge token was corrupt.	Unhappy

The following function signatures were required. They are defined in an abstract manner and they can be straighforwardly translated to C functions.

Table 2, Used functions.

Function	return type	description
SendMessage(string message)	void	Send the specified message.
IsValidMessage(string message)	boolean	True if message is valid, otherwise false.
MessageAvailable()	boolean	True if message is available to read, otherwise false.
ReadMessage()	string	Read and return the message.
StripMessage(string message)	string	Strip and return the specified message.
ResetTimer()	void	Reset the timer to let is restart counting (ms).
HasTimerExpired(time delay)	boolean	True if the speficied time has expired since resetting the timer.

The problem with this design is that for every `SendMessage(message)` the states required for reading the acknowledgements must be added multiple times. These states must be given an unique name, of course. The number of states will be huge when lots of messages are send. This is very impractical. Therefore, the next step is to make the design simpler by nesting states in functions.

The states for reading acknowledgements can be put together, as shown in Figure 2 in the Design section. The states outside READ_ACK are the leaf-states which must be further designed. The READ_ACK state has been converted into the function `ReadAcknowledg()`. See Figure 3. The variable `result` is returned and has to be processed in the main design.

Using `ReadAcknowledge()` after `SendMessage(message)` simplifies the sender by a lot, see Figure 4. The `ResetTimer()` and `HasTimerExpired()` has disappeared and are used inside `ReadAcknowledge()`. The delay of the timeout must be given as a argument, e.g. `ReadAcknowledge(5s)` specifies a 5 seconds delay until the timer expires. When the function reaches the terminate state then it starts again at the initial state, Figure 3. At that moment, the timer is automatically resetted. The function `ReadAcknowledge()` is implemented with the functions of Table 2.

The design of the receiver is shown in Figure 5. Looking closely at the design it is not useful to further simplify the design, otherwise it will be over-simplified and becomes impractible. It is important that at every leaf-state the messages `$ACK;` or `$NACK;` are send back to the sender.

2.2. Implementation

The 7 functions in Table 2 have been implemented in separate source files (.h, .cpp). The source files `Messaging.h` and `Messaging.cpp` implement the top 5 functions. The source files `Timer.h` and `Timer.cpp` implement the bottom 2 functions.

The code patterns from the lecture were use to implement the state-machines. This was a straightforeward process. The problem that had to be solved was that various enumerations define states with same names. The C compiler results in errors. Therefore, state names in enumerations had to be made unique by adding prefixes. The designs were not adapted to the prefixes.

3. Designs

3.1. SendMessage initial design

The design departed from the LEDON state and SendMessage("$LedOn;"). See Figure 1. The design ended up in the leaf-states that cover the happy flow state (NEXT_STATE) and the unhappy flow states (FAILED_ACK, UNKNOWN_ACK, TIMEOUT_ACK and CORRUPT_ACK). Obviously, the design is incomplete because the leaf-states (NEXT_STATE, FAILED_ACK, UNKNOWN_ACK, TIMEOUT_ACK and CORRUPT_ACK) need to be further designed.

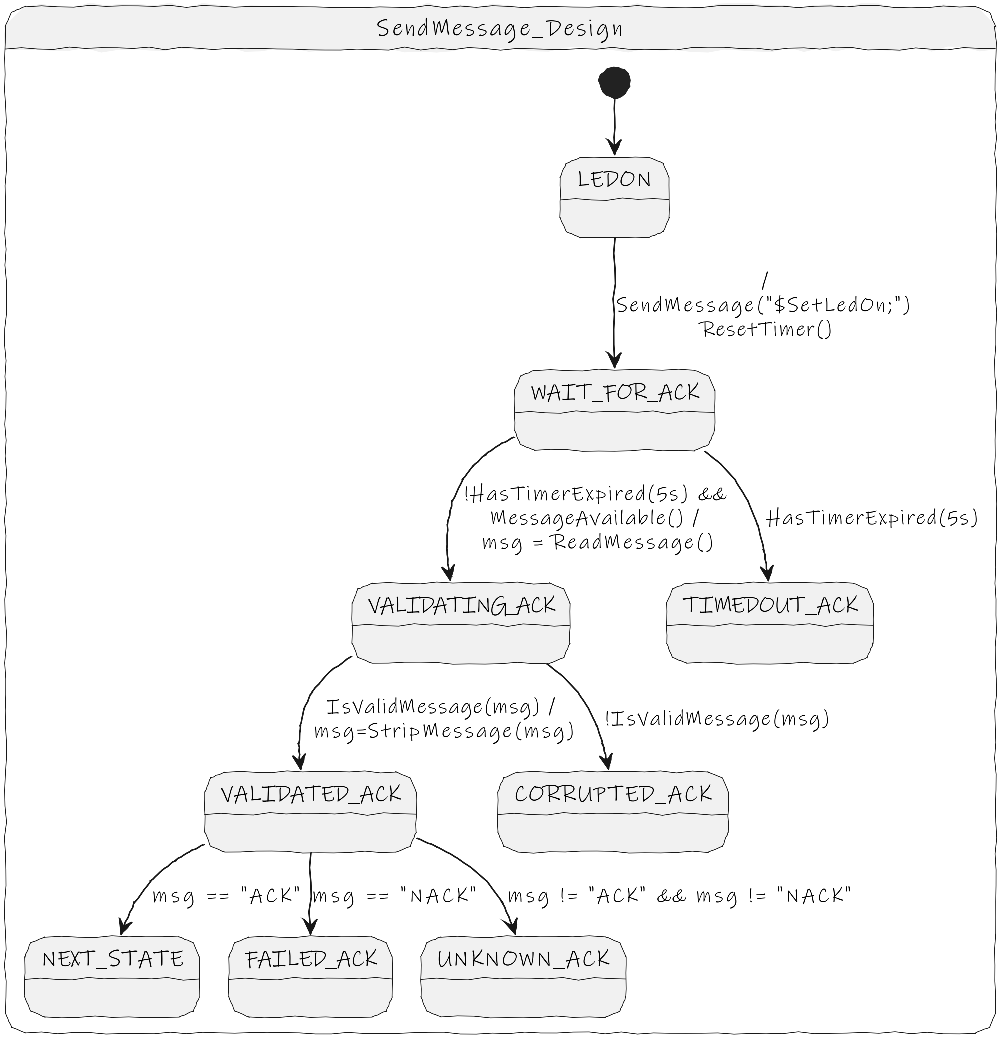


Figure 1, First design started from LEDON state.

3.2. SendMessage simplification design

The state diagram of Figure 1 can be simplified by nesting states into a function. Figure 2 shows an intermediate design with the help of the READ_ACK state. READ_ACK is a sub-state-machine. The approach helped to see the implication of nesting states and the effects on the leaf-states.

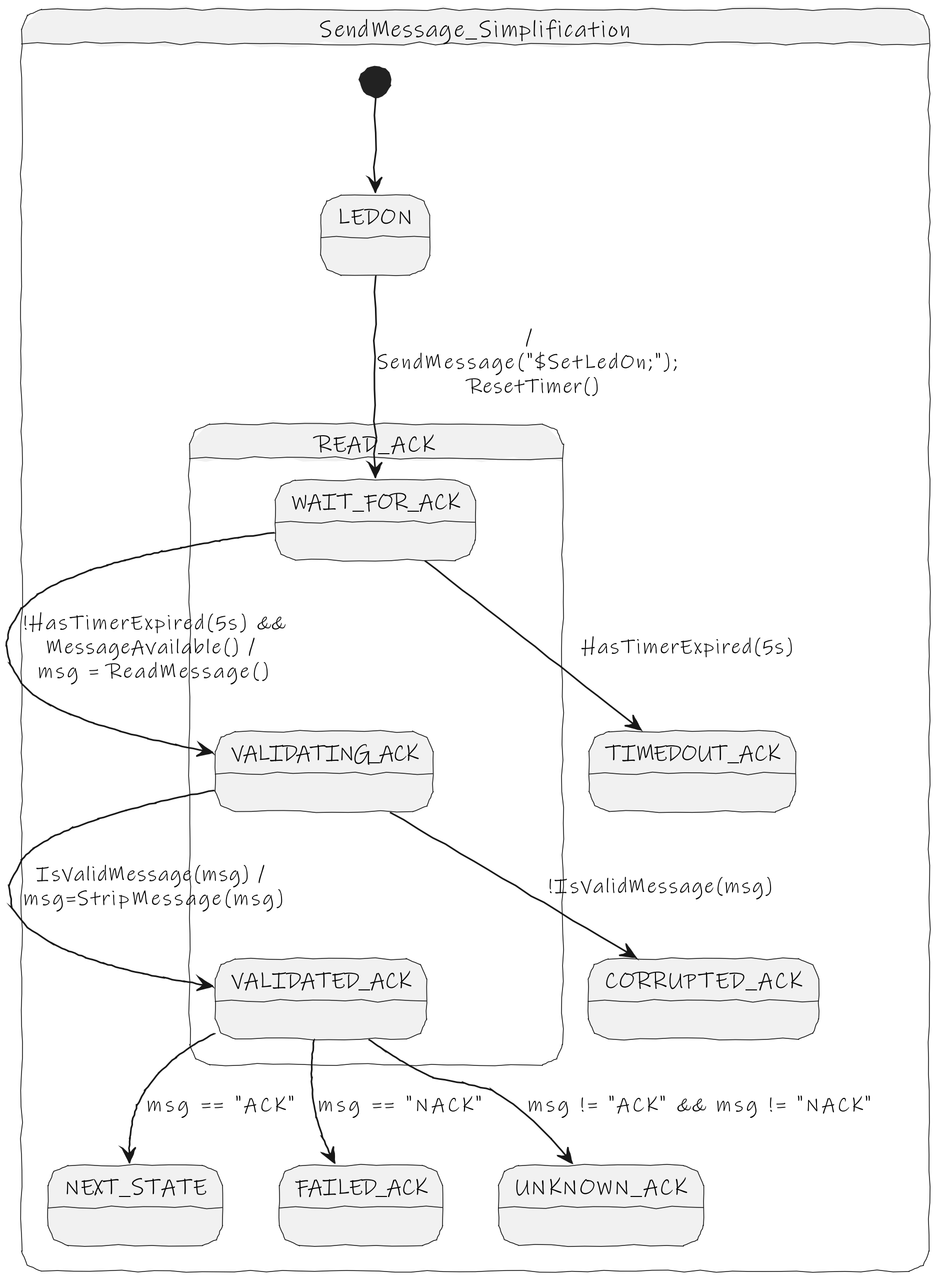
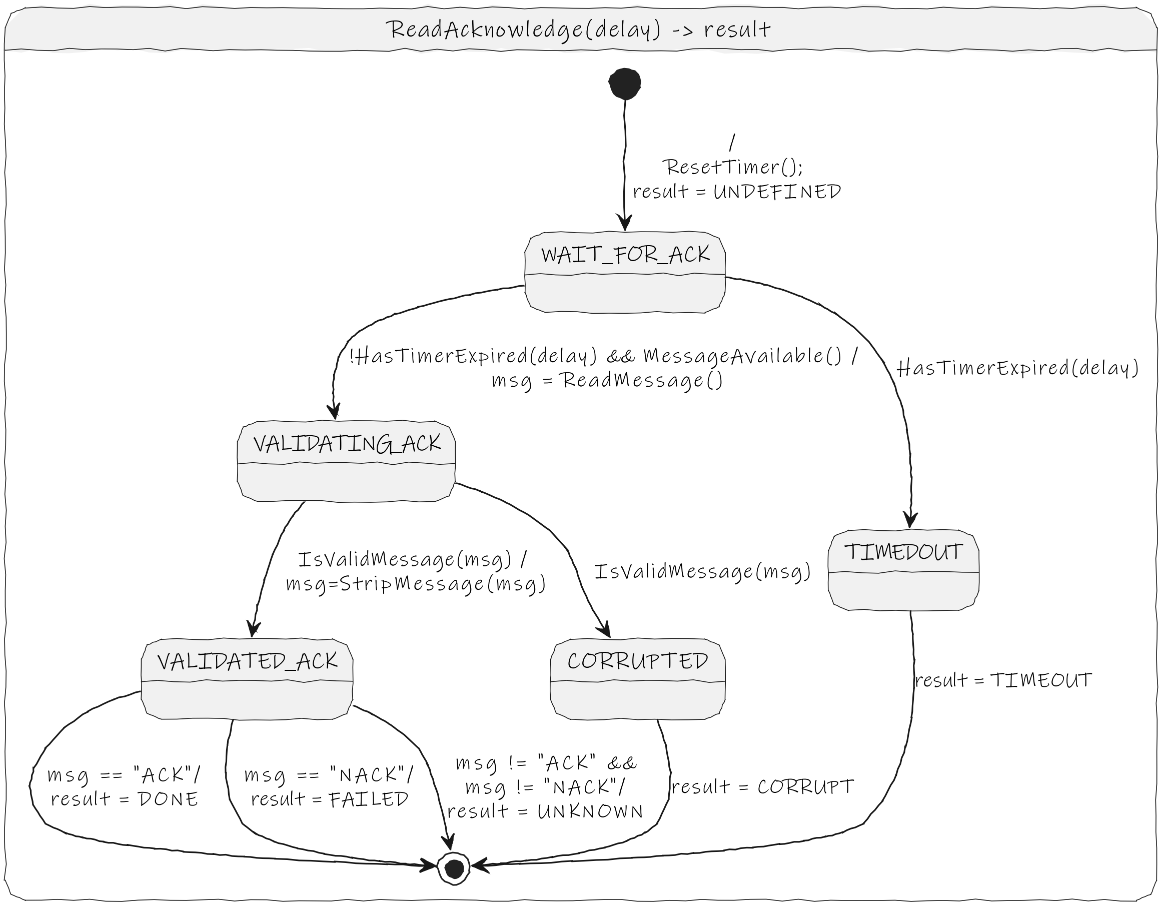


Figure 2, Simplification design of initial design.

The function that implements this sub-state-machine is `ReadAcknowledge()`. See Figure 3.



ReadAcknowledge(delay) -> result
/ ResetTimer(); result = UNDEFINED
WAIT_FOR_ACK
!HasTimerExpired(delay) && MessageAvailable() / msg = ReadMessage()
HasTimerExpired(delay)
VALIDATING_ACK
TIMEDOUT
IsValidMessage(msg) / msg=StripMessage(msg)
IsValidMessage(msg)
VALIDATED_ACK
CORRUPTED
result = TIMEOUT
msg == "ACK"/ result = DONE
msg == "NACK"/ result = FAILED
msg != "ACK" && msg != "NACK"/ result = UNKNOWN
result = CORRUPT

Figure 3, Sub-state-machine performed by function ReadAcknowledge().

3.3. SendMessage final design

Using `ReadAcknowledge()` after `SendMessage(message)` simplifies the sender by a lot, see Figure 4.

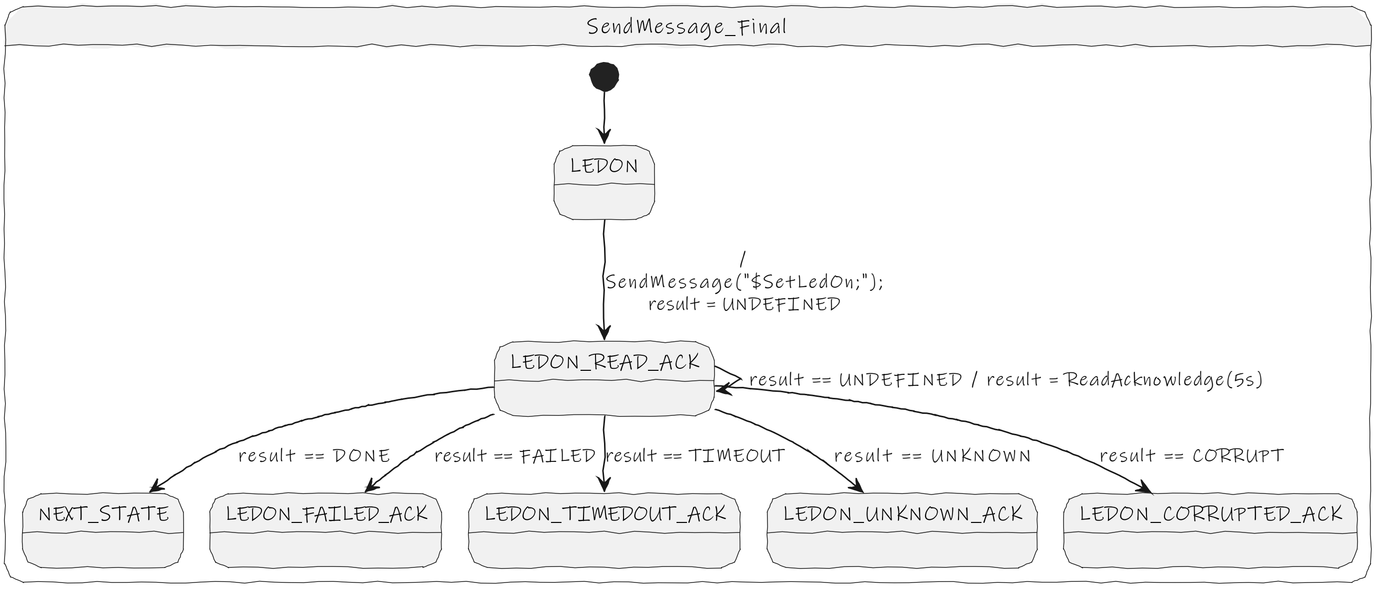


Figure 4, Final design of using SendMessage and handling acknowledgement.

Receiving acknowledgement is required when messages are send to serial port connections. Acknowedgements (ACK, NACK) enable timeouts and corrupt communication, which allow detecting issues that should be handled. The function `ReadAcknowledge()` is non-blocking. It doesn't delay the flow of the state machine. When `ReadAcknowledge()` returns UNDEFINED then the acknowledment has not been completed. It is important that this function is invoked by a self-loop. Simultaneous state machines can continue when this state machine is in the self-loop. The design of `ReadAcknowledge()` illustrates that a state machine can be used to design a non-blocking function using blocking functions like `readStringUntil()` and detecting timeout. The function is state-machine ready, since it can performs a state-machine and can be invoked from a parent state-machine.

3.4. Receive message example

The receiver design is shown in Figure 5. The handling of two messages, namely `$LedOn;` and `$LedOff;`, is illustrated. The design is incomplete. The leaf-states (LED_IS_ON, LED_IS_OFF, UNKNOWN_MSG and CORRUPTED_MSG) must be further designed. Timeout on receiving messages is not part of this design.

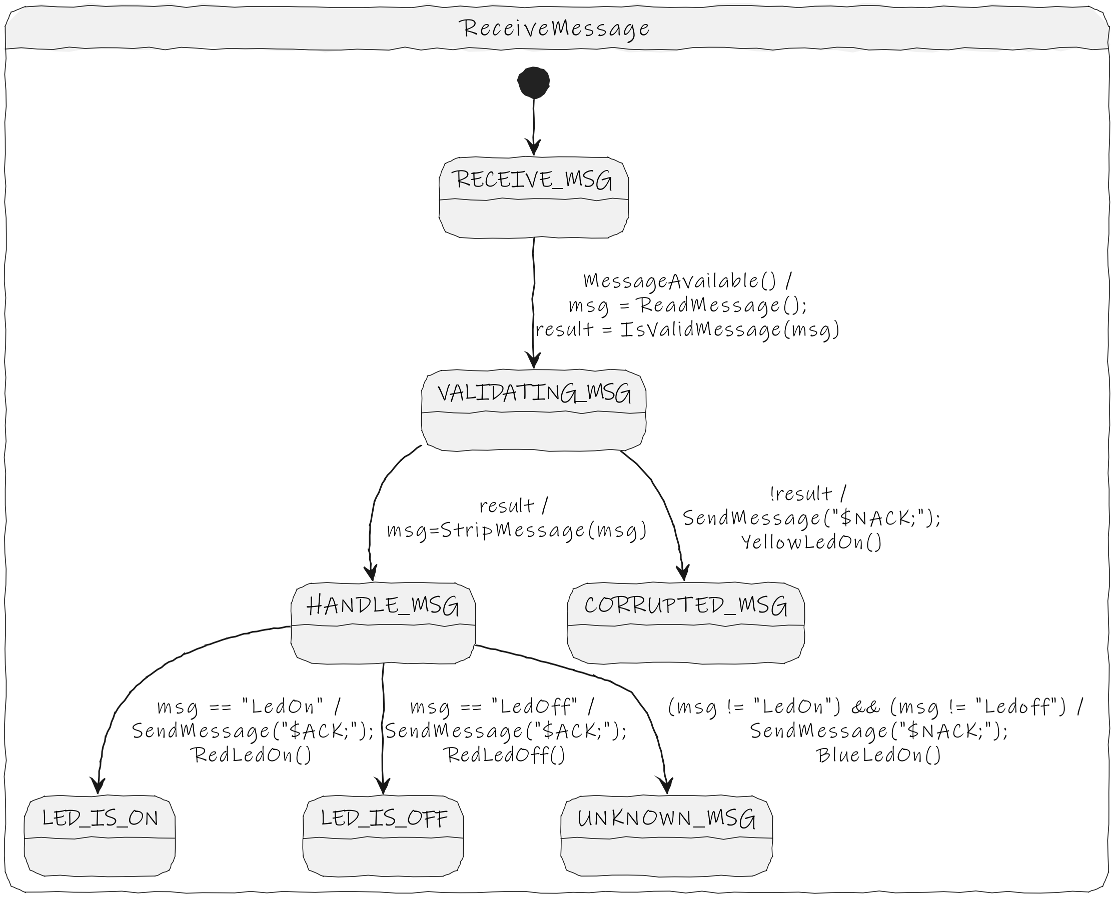

ReceiveMessage
RECEIVE_MSG
MessageAvailable() / msg = ReadMessage(); result = IsValidMessage(msg)
VALIDATING_MSG
result / msg=StripMessage(msg)
!result / SendMessage("$NACK;"); YellowLedOn()
HANDLE_MSG
CORRUPTED_MSG
msg == "LedOn" / SendMessage("$ACK;"); RedLedOn()
msg == "LedOff" / SendMessage("$ACK;"); RedLedOff()
(msg != "LedOn") && (msg != "Ledoff") / SendMessage("$NACK;"); BlueLedOn()
LED_IS_ON
LED_IS_OFF
UNKNOWN_MSG

Figure 5, Design of the receiver side.

4. Conclusions

Reliable and *robust* communication require lots of states and makes the sender and receiver more complex. The approach of using state-machines was useful. The attempt to using state diagrams gave sufficient guidance to end up with the desired and complete behaviour of the sender and receiver. These are essential building blocks for designing communication protocols.

The functions that are discovered from the designs are implemented in C. Thanks to the state-design approach, the `ReadAcknowledge()` is non-blocking and state-machine-ready. The performance is high without delays and multiple state-machines are allowed to run in interleaving fashion.

Using AsciiDoc for writing this technical report takes some practice and understanding of the formatting. Lots of googling helped to find the possibilities. At one hand writing adocs was quick and easy and on the other hand it was complicated when using fancy formats, like used to in MS-Word. It must be concluded that AscciDoc isn't as powerfull as writing in MS-Word. What about writing technical report in AsciiDoc? Well, this technical report is proof that AsciiDoc is useful for writing technical reports. In an Agile approach this way of writing technical reports in Visual Studio Code looks promising.

5. References

…to be done…