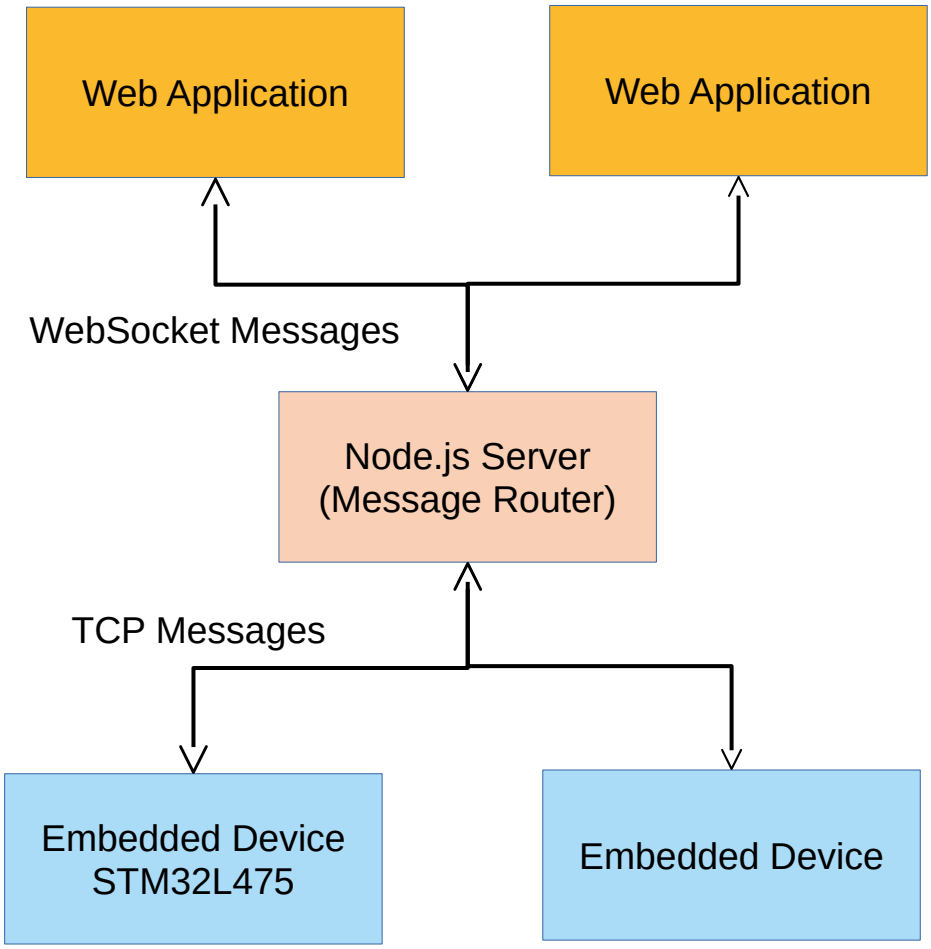
Module 7

Design and Optimization of Embedded and Real-time Systems

1 Introduction

In this class, we are going to connect our discovery board to the network. We will study how to interface with the WiFi module using SPI and understand basic communication protocol design.

2 Network Topology



First let's take a look at our network topology. The centerpiece of the network is a *message router* (a.k.a *message broker*), which is a Node.js application. Its job is to

- Establish and maintain connections to embedded devices on one end and to web applications on the other end.
- Route *messages* between different connection endpoints or nodes (e.g. between a web application and an embedded device).

The message router uses WebSocket to connect to web applications, which allows asynchronous message communication in both directions. It works just like state machines exchange events within our embedded framework.

The message router uses TCP to communicate with embedded devices, which is simple and works well with different types of WiFi adapters. Note, it can be enhanced to SSL/TLS for better security.

3 Message Protocol

3.1 Message Semantics

The message interface employs similar semantics as the event interface in our embedded framework. That is, a message must be of one of these types: REQ/CFM and IND/RSP. Like our events, a message can carry parameters. Some parameters such as source and destination IDs, as well as sequence number are mandatory for any messages.

Inside the embedded framework, we use "HSM Number (HSMN)" to indicate the source and destination of an event. For messages, we use the notion of "Node ID" as a unique identification of an endpoint (node) in the network.

The following list shows some messages that will be used in our demo project:

1. **SrvAuthReqMsg**

   This is a *request* to the *server* (message router) to *authenticate* a node. It carries the username, password and suggested node ID. (It is assumed SSL/TLS will be used).

2. **SrvAuthCfmMsg**

   This is a response to SrvAuthReqMsg. It carries the authentication result and, if success, the assigned node ID.

3. **SensorDataIndMsg**

   This is an indication from an embedded device (STM32L475 Discovery) to the server, carrying pitch and roll measurements (in degrees). The server routes it to the addressed destination, or broadcasts it all connected nodes if it is addressed to the server itself.

4. **SensorDataRspMsg**

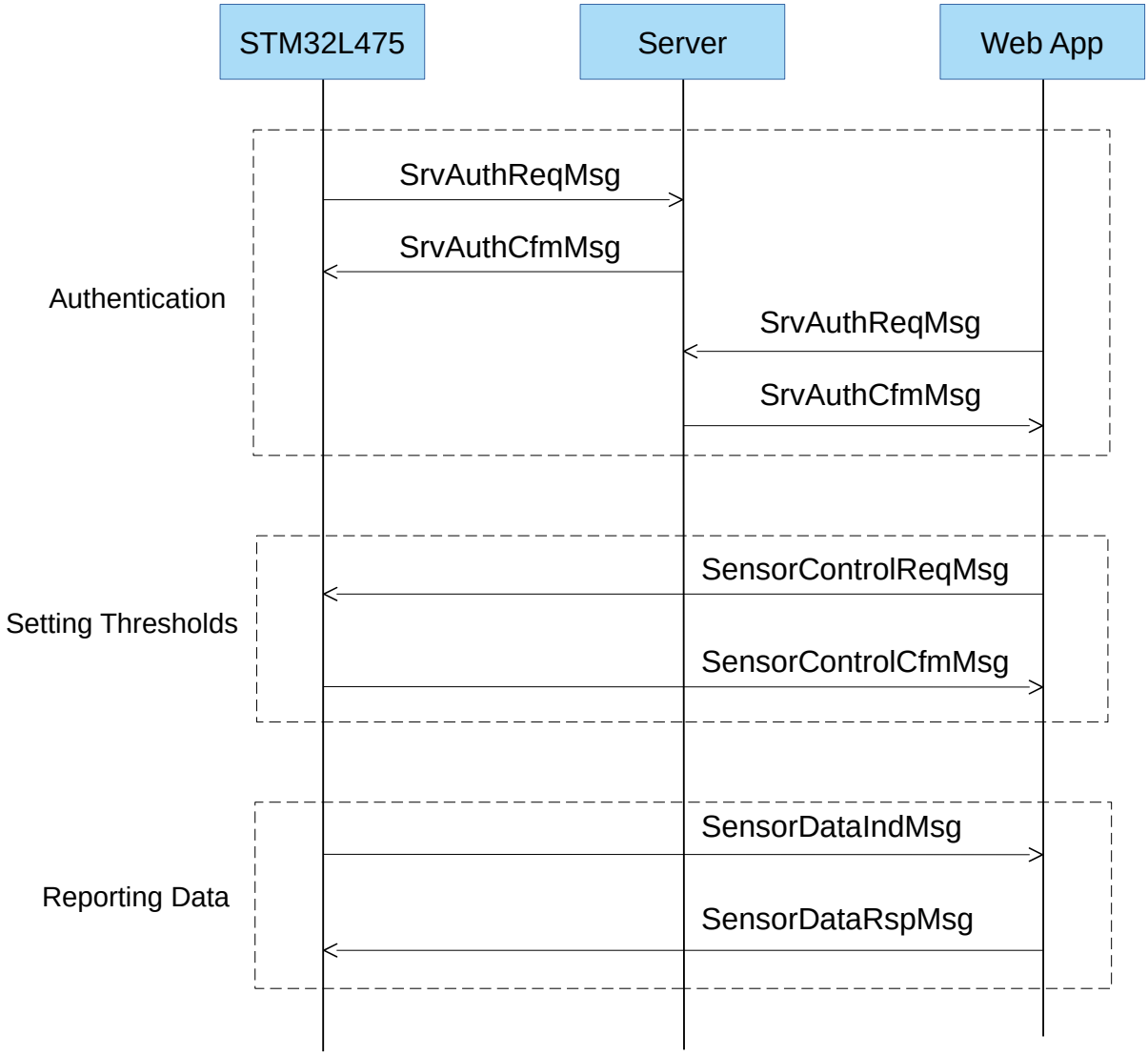
   This is a response to SensorDataIndMsg, acknowledging to the data originator that the indication has been received.

5. **SensorControlReqMsg**

   This is a request to an embedded device to set the warning pitch and roll thresholds. If the pitch or roll angle exceeds the set thresholds, the device shall alert the user (e.g. turn the background of the LCD red or sound an alarm).

6. **SensorControlCfmMsg**

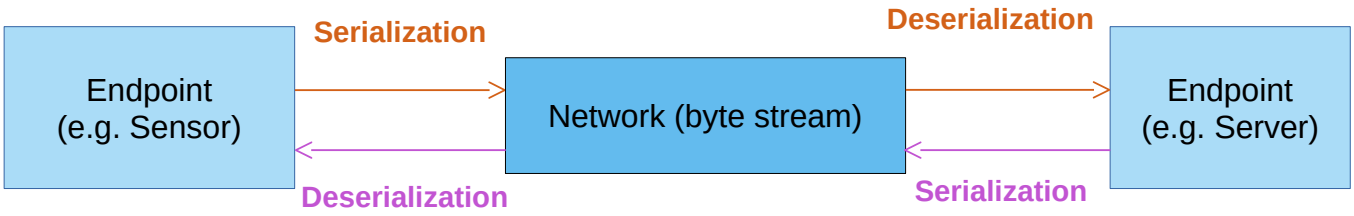
   This is a confirmation to SensorControlReqMsg, notifying to the request originator whether the thresholds have been set successfully.



3.2 Message Format

The above examples show that message communication in a network is very similar to event communication within an embedded system (among state machines).

While events within an embedded system can simply be C++ *objects* or C *structures*, messages traveling in network must first be serialized. Data going across a network are just a sequence or stream of bytes, and there are no concepts to *objects* or *structures*. We must therefore apply explicit formatting to the data, which is also known as *serialization*.



There are many different formats for serialization, including:

1. Binary (Fixed Offsets)

   This is suitable for resource-constrained embedded devices, since they *speak* in *binary data* as a native language. Encoding can be done by direct *copying* a binary data structure (C++ message object) to an outgoing packet buffer. Decoding can be done by *casting* a received packet buffer to the corresponding C++ message class.

   While this is simple and efficient, it lacks flexibility. It requires all senders and receivers agree on a common packet format. If it ever gets changed, all parties must be updated at the same time.

   For the purpose of demonstration, we will use this technique for the communication between our discovery boards and the server.

   This is an example of a binary message defined in **Src/app/Node/SensorMsgInterface.h**:

```
class SensorDataIndMsg final: public Msg {
public:
    SensorDataIndMsg(float pitch = 0.0, float roll = 0.0) :
        Msg("SensorDataIndMsg"), m_pitch(pitch), m_roll(roll) {
        m_len = sizeof(*this);
    }
    uint32_t GetPitch() const { return m_pitch; }
    uint32_t GetRoll() const { return m_roll; }
protected:
    float m_pitch;  // Pitch in degree.
    float m_roll;   // Roll in degree.
} __attribute__((packed));
```

   where base class **Msg** is defined in **Src/framework/include/fw_msg.h**:

```
class Msg {
public:
    Msg(char const *type = MSG_UNDEF, char const *to = MSG_UNDEF,
        char const *from = MSG_UNDEF, uint16_t seq = 0):
        m_seq(seq), m_len(sizeof(*this)) {
        // m_len to be updated in derived class constructors.
        FW_MSG_ASSERT(type && to && from);
        STRBUF_COPY(m_type, type);
        STRBUF_COPY(m_to, to);
        STRBUF_COPY(m_from, from);
    }
    // Must be non-virtual to ensure no virtual table pointer is added.
    ~Msg() {}

    char const *GetType() const { return m_type; }
    char const *GetTo() const { return m_to; }
    char const *GetFrom() const { return m_from; }
    uint16_t GetSeq() const { return m_seq; }
    uint32_t GetLen() const { return m_len; }
    bool MatchSeq(uint16_t seq) const { return m_seq == seq; }
    enum {
        TYPE_LEN = 38,     // Role(16) ServicePrimitive(22)
        TO_LEN = 16,
        FROM_LEN = 16,
    };
    void SetTo(char const *to) {
        FW_MSG_ASSERT(to);
        STRBUF_COPY(m_to, to);
    }
    void SetFrom(char const *from) {
        FW_MSG_ASSERT(from);
        STRBUF_COPY(m_from, from);
    }
    void SetLen(uint32_t len) { m_len = len; }
    void SetSeq(uint16_t seq) { m_seq = seq; }
protected:
    char m_type[TYPE_LEN];
    char m_to[TO_LEN];
    char m_from[FROM_LEN];
    uint16_t m_seq;
    uint32_t m_len;        // Total message length including base and derived parts.
} __attribute__((packed)); // It must be 4-byte aligned.
```

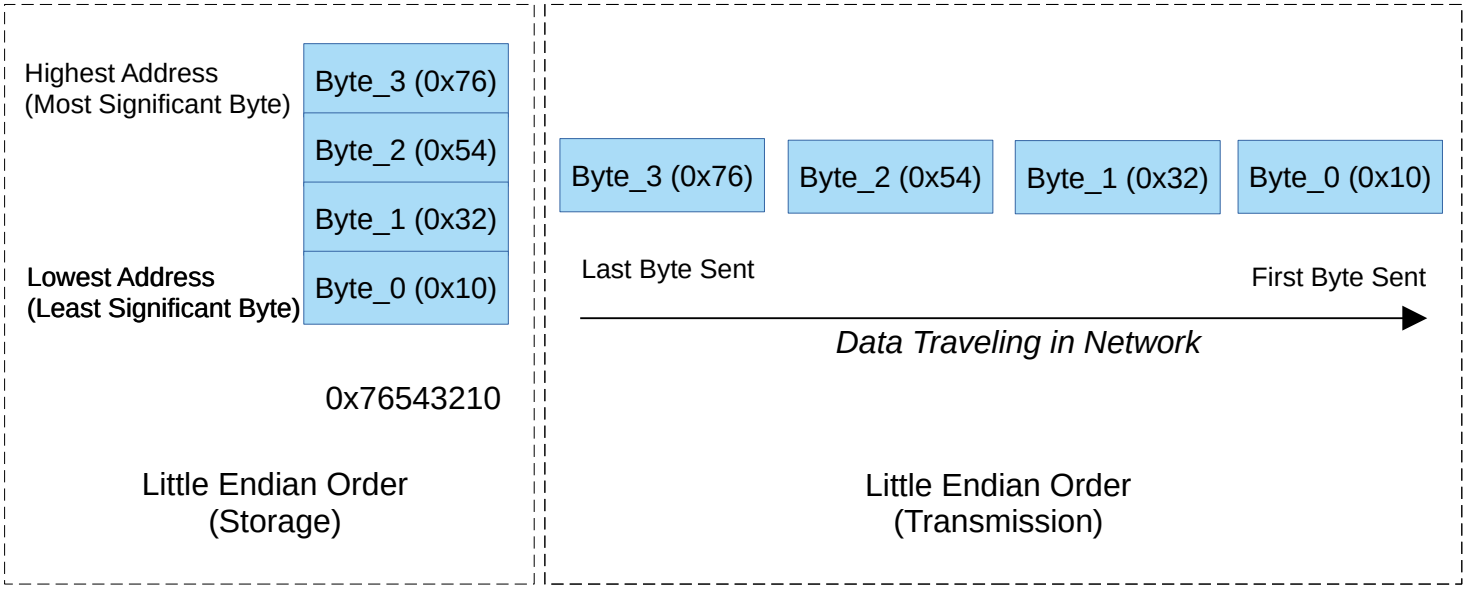
A memory buffer containing a **SensorDataIndMsg** object looks like this:

Offset	Field
0	m_type: char [38]
38	m_to: char [16]
54	m_from: char [16]
70	m_seq: uint16_t (2 bytes)
72	m_len: uint32_t (4 bytes)
76	m_pitch: float (4 bytes)
80	m_poll: float (4 bytes)

Total length = 84 bytes.

When transmitted to a network, fields in the message buffer are sent from the lowest offset (address) to the highest. For fields of multiple-byte types (e.g. float or uint32_t), they can be sent in *ascending significance* order (called **little-endian**, lowest significant byte sent first) or in *descending significance* order (called **big-endian**, highest significant byte sent first).

Traditionally data are sent to a network in big-endian order, which is also called *network-order*. However since our STM32L475 stores its internal data in little-endian order (lowest address storing lowest significant byte), we simply use little-endian order when transmitting data to a network to avoid unnecessary *endian conversion*.



2. JSON

   https://www.json.org/json-en.html

   JSON stands for Javascript Object Notation which is a very popular format for data communications. It is supported in Javascript (obvious), C/C++ (with libraries), Python, etc.

   JSON is more flexible than the fixed binary format. Each *member* of an object is a name/value pair, and therefore the order of members (or fields) in a message is not hard-coded.. New fields can be added to a message or old ones can be removed while maintaining backward compatibility.

   Since JSON is the native language of Javascript, in which both our Node.js server and web application are written, it's natural to use JSON as the message exchange format between the server and the web application.

   This is an example of the same **SensorDataIndMsg** as we have seen before in binary format. This time it is presented in JSON format:

```
SensorDataIndMsg {
  type: 'SensorDataIndMsg',
  to: 'Srv',
  from: 'LevelMeter',
  seq: 0,
  len: 84,
  pitch: -1.526681900024414,
  roll: -0.621920108795166
}
```

3. Protocol Buffer

   https://developers.google.com/protocol-buffers

4. ASN.1

   https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx

4 Component Design

To support network communications with a server (message router), we need to decide which additional components are needed, what their roles are and how they interact with each other.

Using our framework, each component is a *hierarchical state machin*e (HSM), and they interface with each other via *events*. These are the roles of the additional HSMs:

- **Node**

  It is an *active object* responsible for the *session layer* between the embedded device itself (node) and the server (message router). It is responsible for:

  - Establishing connection to a server and authentication.
  - Maintaining connection with ping req/cfm and recovery.
  - Transmitting messages from other state machines to the server.
  - Receiving messages from the server and forwarding them to other state machines for processing.

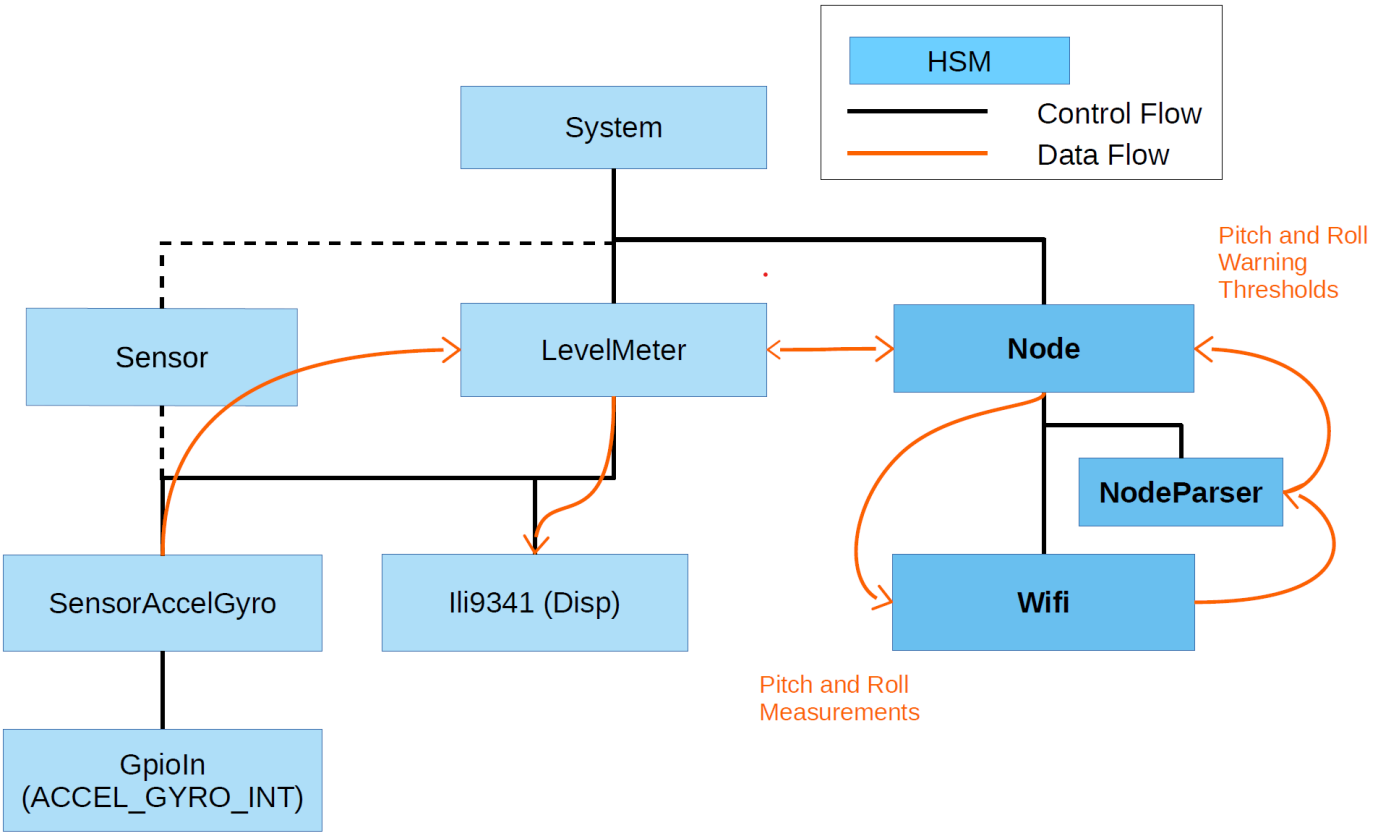
- **NodeParser**

  It is an *orthogonal region* under **Node** responsible for parsing raw TCP data from WiFi to form complete messages. It runs in the same thread as **Node**.

- **Wifi**

  It is an *active object* responsible for interfacing with the **Inventek Systems ISM43362** WiFi module over SPI. It manages the *network layer* between the node and the server, including joining an access point (AP) and establishing a TCP socket connection to the server.

You can find them under the **Src/app** directory in our project. The block diagram below illustrates the overall control hierarchy:



5 Node Active Object

5.1 Event Interface

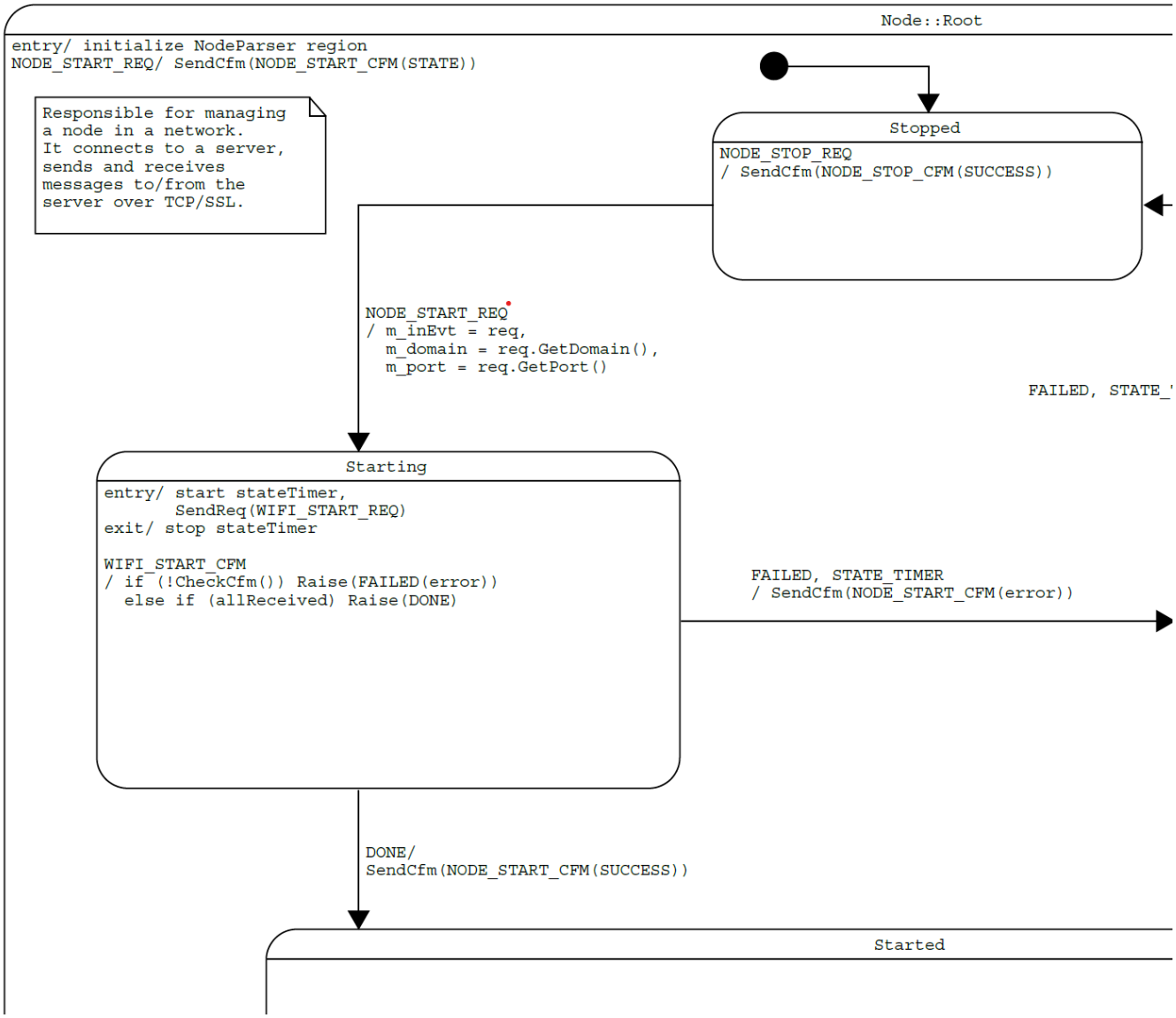
The following table lists the event interface of **Node**. The *NODE_START_REQ* automatically connects to the specified server.

Event	Description
NODE_START_REQ	Starts the state machine passing in the IP address and port number of the server to connect to.
NODE_START_CFM	
NODE_STOP_REQ	Stops the state machine.
NODE_STOP_CFM	

The main behaviors are *specified* in the following statechart extracts.

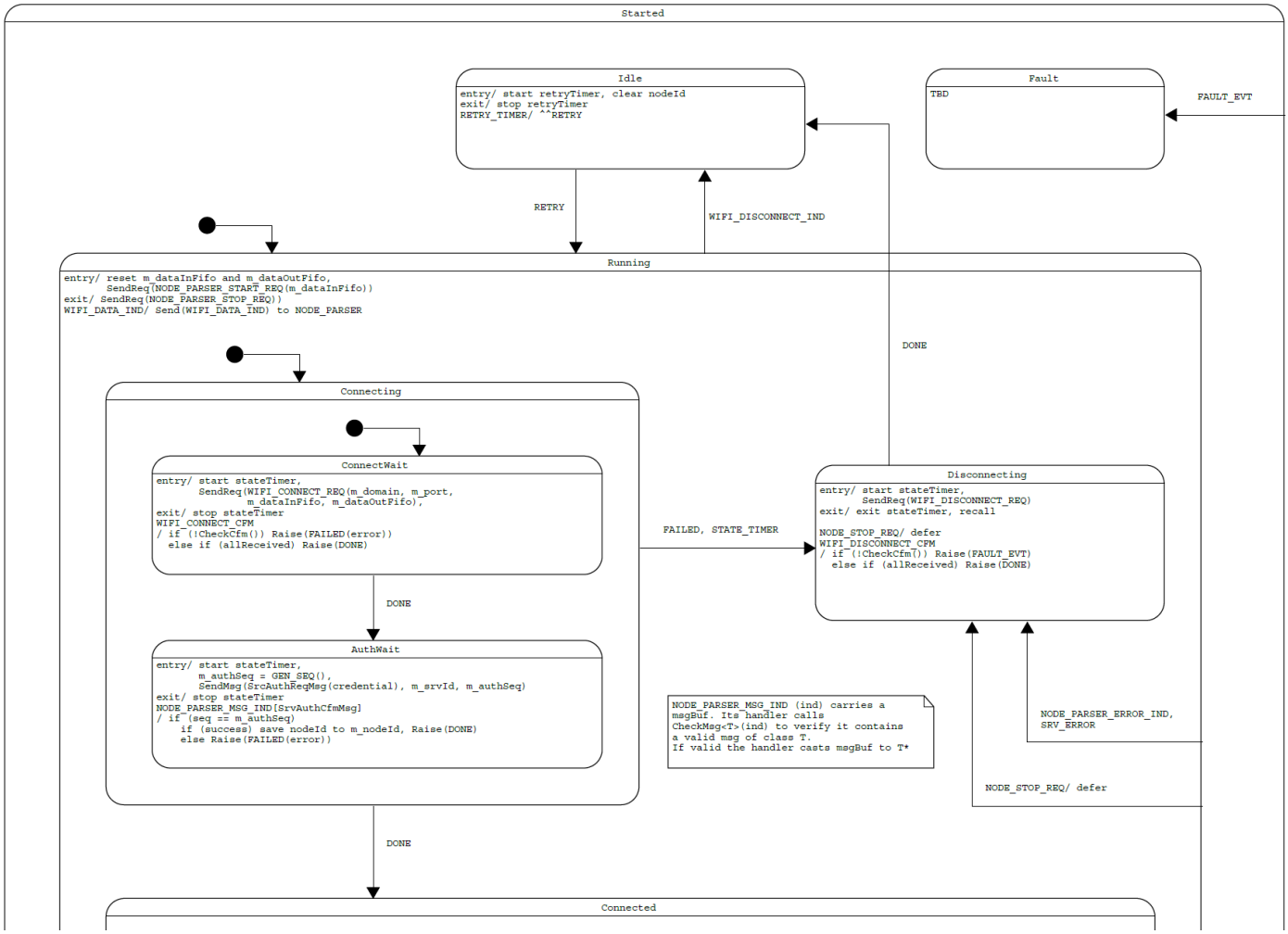
5.2 Starting State

Node starts the **Wifi** HSM in its *Starting* state. It makes sure it gets the positive confirmation before proceeding to the *Started* state.



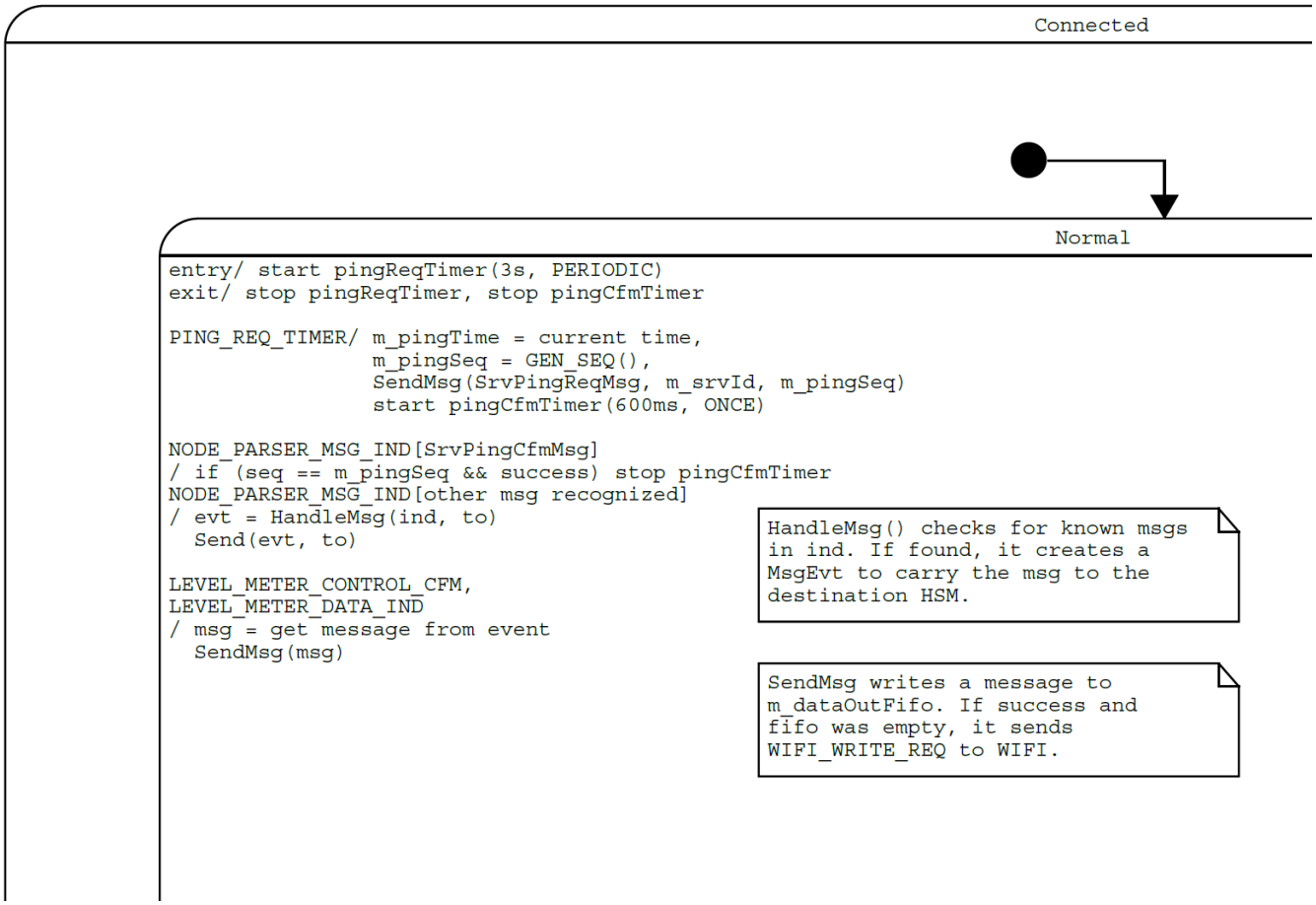
5.3 Started State

In the *Started* state, Node manages connection, authentication, disconnection and automatic re-connection. See how the division of substates at each level forms a "partition" of the state-space (i.e. non-overlap and without gaps).



5.4 Connected/Normal State

The *Normal* state within *Connected* is where *events* are converted to *messages*, and vice versa. It serves as a *gateway* to the outside world/network.



5.4.1 Incoming Messages

For incoming messages, the event *NODE_PARSER_MSG_IND* carries a buffer containing a complete message. Since TCP is a streaming protocol, there is NO guarantee a single read from the socket yields a complete message. The **NodeParser** HSM is an *orthogonal region* (parallel state-machine) of **Node** that helps with message delineation. (Read the statechart of NodeParser to learn more about it.)

Upon NODE_PARSER_MSG_IND, Node handles the SrvPingCfmMsg itself. For other messages, it converts them into *events* via the helper method **HandleMsg()**, which are sent to the destination HSMs for processing.

The HandleMsg() function is listed below:

```
MsgEvt *Node::HandleMsg(NodeParserMsgInd const &ind, Hsmn &to) {
    to = HSM_UNDEF;
    if (auto m = CHECK_MSG_LOG(SensorControlReqMsg, ind)) {
        to = LEVEL_METER;
        return new LevelMeterControlReq(*m);
    }
    if (auto m = CHECK_MSG_LOG(SensorDataRspMsg, ind)) {
        to = LEVEL_METER;
        return new LevelMeterDataRsp(*m);
    }
    // @todo Add other message handling here.
    return nullptr;
}
```

This is the mapping from incoming messages to events:

SensorControlReqMsg → LevelMeterControlReq

SensorDataRspMsg → LevelMeterDataRsp

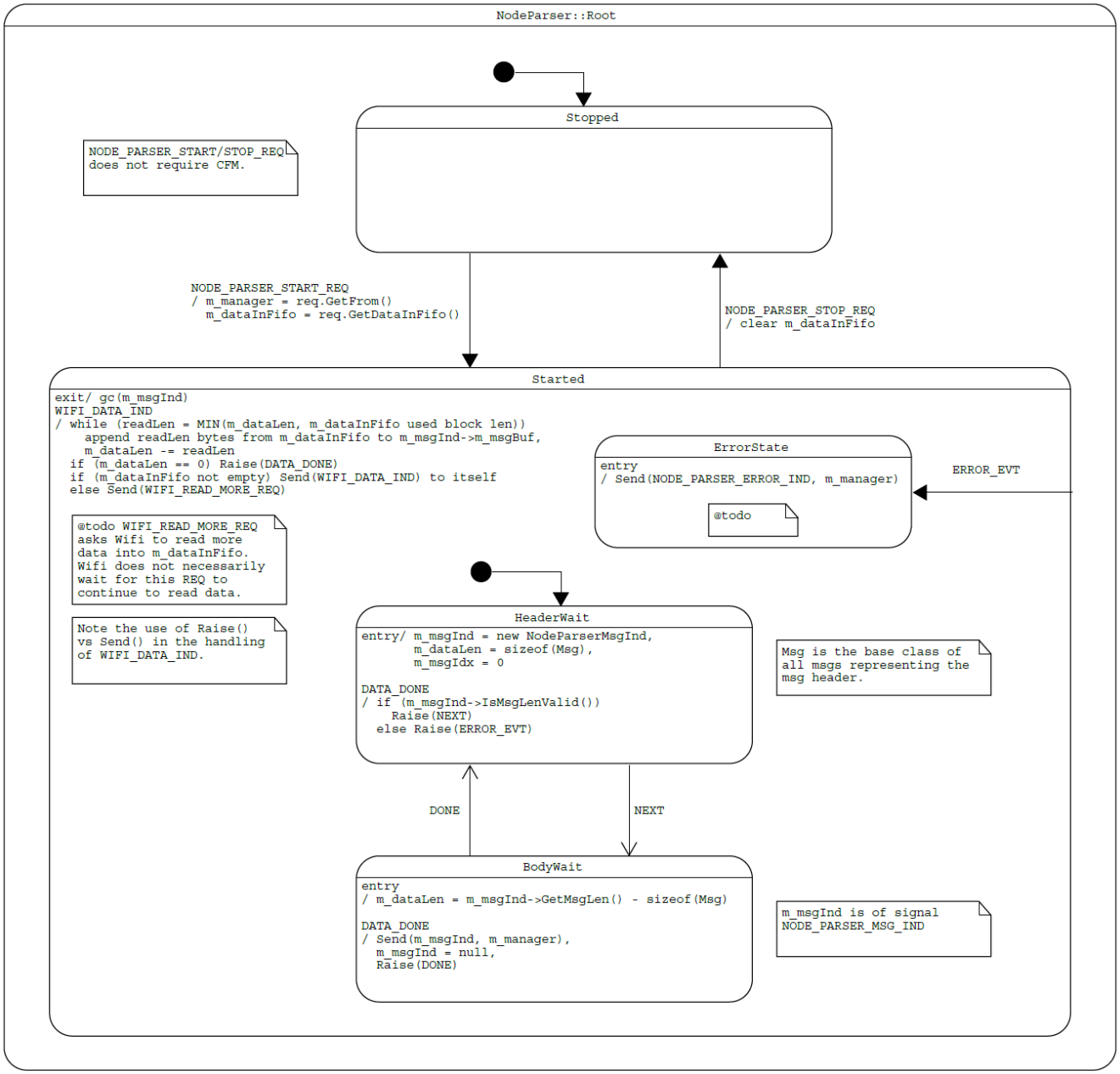
The event simply carries the message object as an event parameter:

```
class LevelMeterControlReq : public MsgEvt {
public:
    // Must pass 'm_msg' as reference to member object and NOT the parameter 'msg'.
    LevelMeterControlReq(SensorControlReqMsg const &r) :
        MsgEvt(LEVEL_METER_CONTROL_REQ, m_msg), m_msg(r) {
        LEVEL_METER_INTERFACE_ASSERT(&GetMsgBase() == &m_msg);
    }
    float GetPitchThres() const { return m_msg.GetPitchThres(); }
    float GetRollThres() const { return m_msg.GetRollThres(); }
protected:
    SensorControlReqMsg m_msg;
};
```

where **MsgEvt** is the base class of all message-carrying events providing common utilities:

```
// Abstract base class for message-carrying events to provide easy access to msg members.
class MsgEvt : public Evt {
public:
    Msg &GetMsgBase() const { return m_msgBase; }
    char const *GetMsgType() const { return m_msgBase.GetType(); }
    char const *GetMsgTo() const { return m_msgBase.GetTo(); }
    ...
    void SetMsgTo(char const *to) { m_msgBase.SetTo(to); }
    void SetMsgSeq(uint16_t seq) { m_msgBase.SetSeq(seq); }
protected:
    MsgEvt(QP::QSignal signal, Msg &msg) : Evt(signal), m_msgBase(msg) {}
    MsgEvt(Msg &msg, QP::QEvt::StaticEvt /*dummy*/) :
        Evt(QP::QEvt::STATIC_EVT), m_msgBase(msg) {}
    ...
    Msg &m_msgBase;
};
```

The statechart of **NodeParser** orthogonal region is shown below for reference.



5.4.2 Output Messages

For outgoing messages, **Node** receives *events* from other HSMs and sends the messages carried by those events to the network through the **Wifi** active object. The following list shows the mapping of outgoing events to messages:

LevelMeterControlCfm → SensorControlCfmMsg

LevelMeterDataInd → SensorDataIndMsg

The listing below shows the definition of the **LevelMeterDataInd** event class. Just like event classes for incoming messages, outgoing event classes are derived from **MsgEvt** and contain the corresponding message objects as members:

```
class LevelMeterDataInd : public MsgEvt {
public:
    // Must pass 'm_msg' as reference to member object and NOT the parameter 'msg'.
    LevelMeterDataInd(SensorDataIndMsg const &r) :
        MsgEvt(LEVEL_METER_DATA_IND, m_msg), m_msg(r) {
        LEVEL_METER_INTERFACE_ASSERT(&GetMsgBase() == &m_msg);
    }
    uint32_t GetPitch() const { return m_msg.GetPitch(); }
    uint32_t GetRoll() const { return m_msg.GetRoll(); }
protected:
    SensorDataIndMsg m_msg;
};
```

The *Normal* state of **Node** sends these outgoing messages to the **Wifi** HSM via the helper method **SendMsg()**:

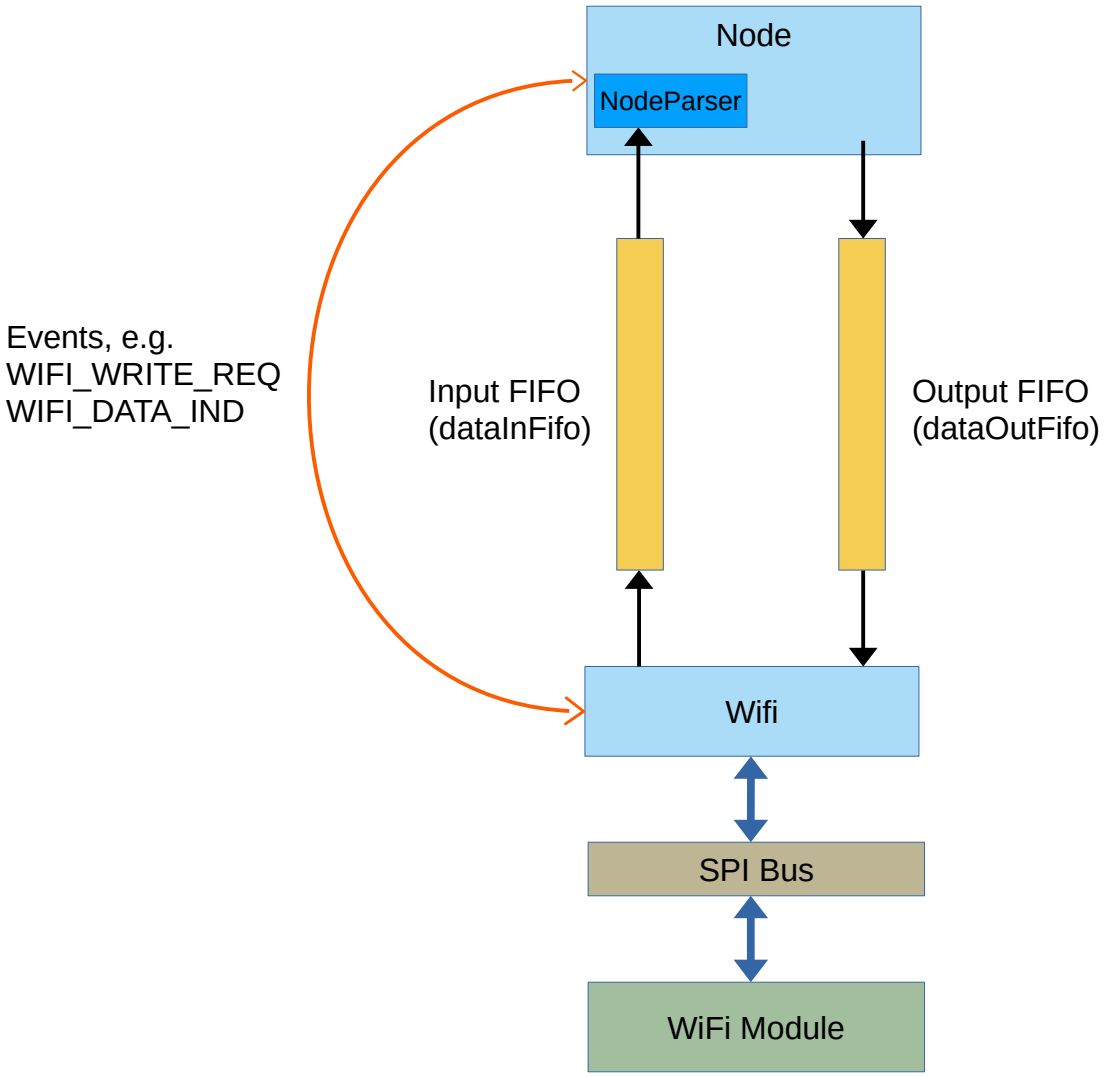
```
QState Node::Normal(Node * const me, QEvt const * const e) {
    switch (e->sig) {
        ...
        case LEVEL_METER_CONTROL_CFM:
        case LEVEL_METER_DATA_IND: {
            EVENT(e);
            auto const &msgEvt = static_cast<MsgEvt const &>(*e);
            me->SendMsg(msgEvt.GetMsgBase(), msgEvt.GetMsgLen());
            return Q_HANDLED();
        }
    }
    return Q_SUPER(&Node::Connected);
}
```

Through the common base class **MsgEvt**, the code above gets the address and length of a message buffer *generically*, regardless to the specific message type.

6 Wifi Active Object

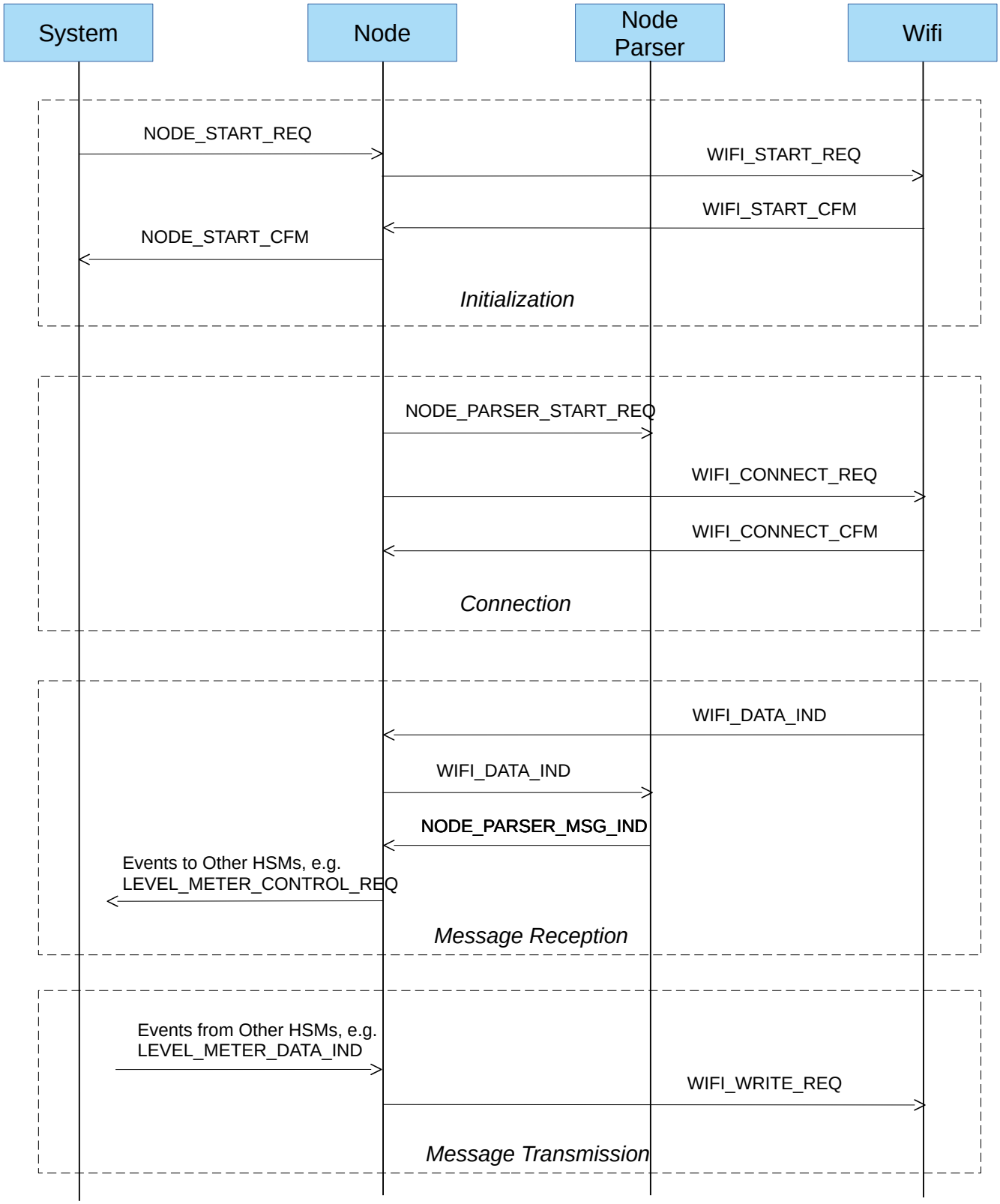
6.1 Event Interface

The role of the **Wifi** active object is to establish a connection with a TCP server and send/receive data to/from the server. For efficiency, data are passed via FIFOs (pipes). Pointers to the input and output FIFOs (owned by **Node**) are injected into the Wifi object through the *WIFI_CONNECT_REQ* event. The block diagram and event interface are shown below:



Event	Description
WIFI_START_REQ	Request to start this state machine.
WIFI_START_CFM	
WIFI_STOP_REQ	Request to stop this state machine.
WIFI_STOP_CFM	
WIFI_INTERACTIVE_ON_REQ	Request to enter interactive mode to send AT commands from the serial console to the WiFi module directly. This is mainly used for debugging and testing purposes. (Not available with the ISM43362 module)
WIFI_INTERACTIVE_ON_CFM	
WIFI_INTERACTIVE_OFF_REQ	Request to exit interactive mode.
WIFI_INTERACTIVE_OFF_CFM	
WIFI_CONNECT_REQ	Request to connect to a server at the specified IP address and port. It passes in the input and output data FIFOs for efficient streaming of payload data.
WIFI_CONNECT_CFM	
WIFI_DISCONNECT_REQ	Request to disconnect from a connected server.
WIFI_DISCONNECT_CFM	
WIFI_DISCONNECT_IND	Indication to a user HSM that the connection has been dropped. It is the responsibility of a user to re-connect.
WIFI_WRITE_REQ	Request to send any data in the output FIFO to a connected server.
WIFI_WRITE_CFM	
WIFI_EMPTY_IND	Indication to a user HSM that the output FIFO has been emptied. Upon this event, a user HSM may write more data to the output FIFO. (It is currently not used by **Node**, which discards a message when the FIFO is full.)
WIFI_DATA_IND	Indication to a user HSM that data have been received from a connected server and are available in the input FIFO for processing.
WIFI_READ_MORE_REQ	Request to read more data from a connected server once a user HSM has read all data from the input FIFO. (It is currently not used since it uses polling to read data from the ISM43362 module.)

6.2 Sequence Diagram

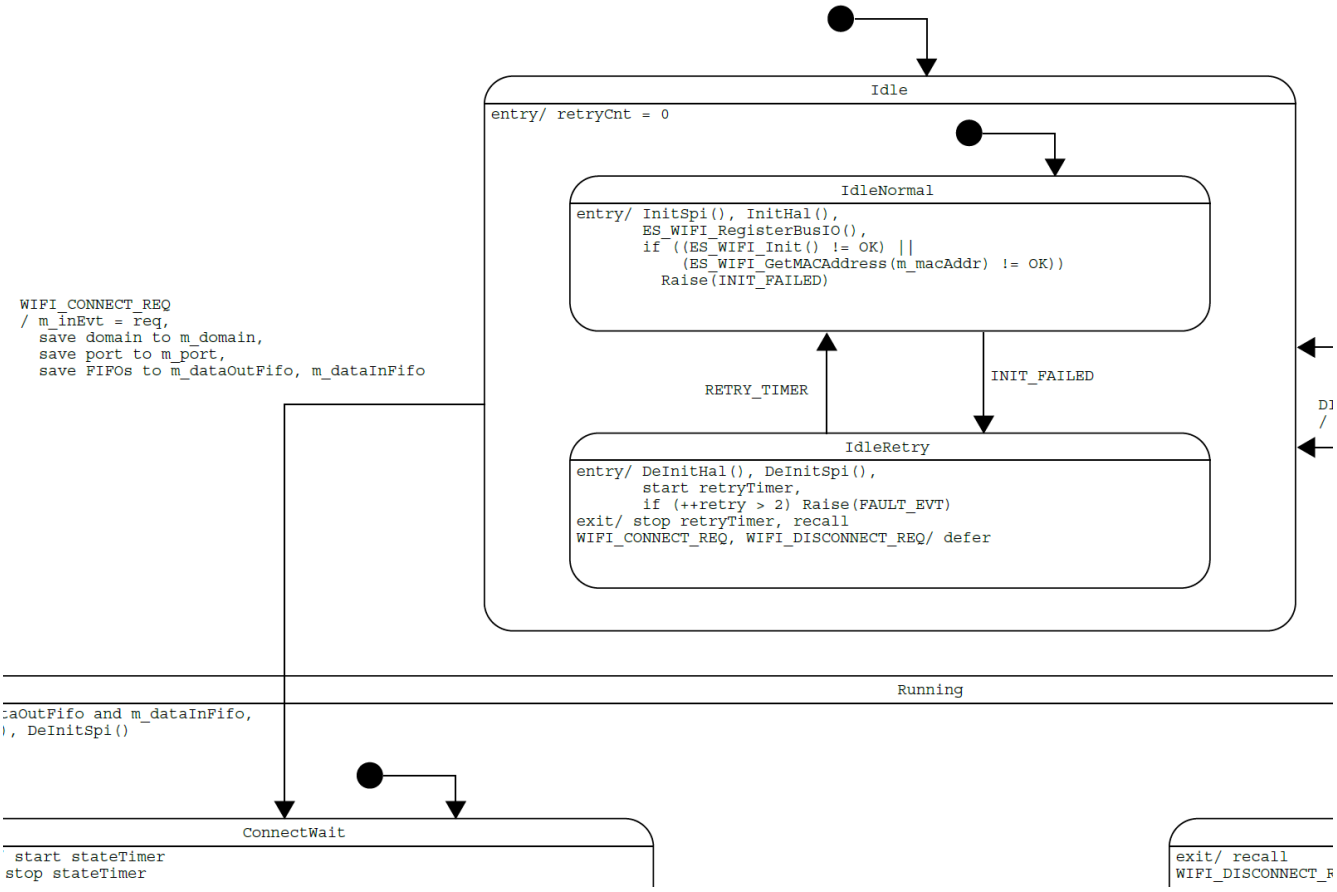


6.3 Idle State

The substates in **Idle** automatically handle the case when the WiFi module fails to initialize successfully. This is particular important when interfacing with an external module of which its firmware is beyond our control.

Note how it saves the parameters of the **WIFI_CONNECT_REQ** to member variables of the state machine for use later.

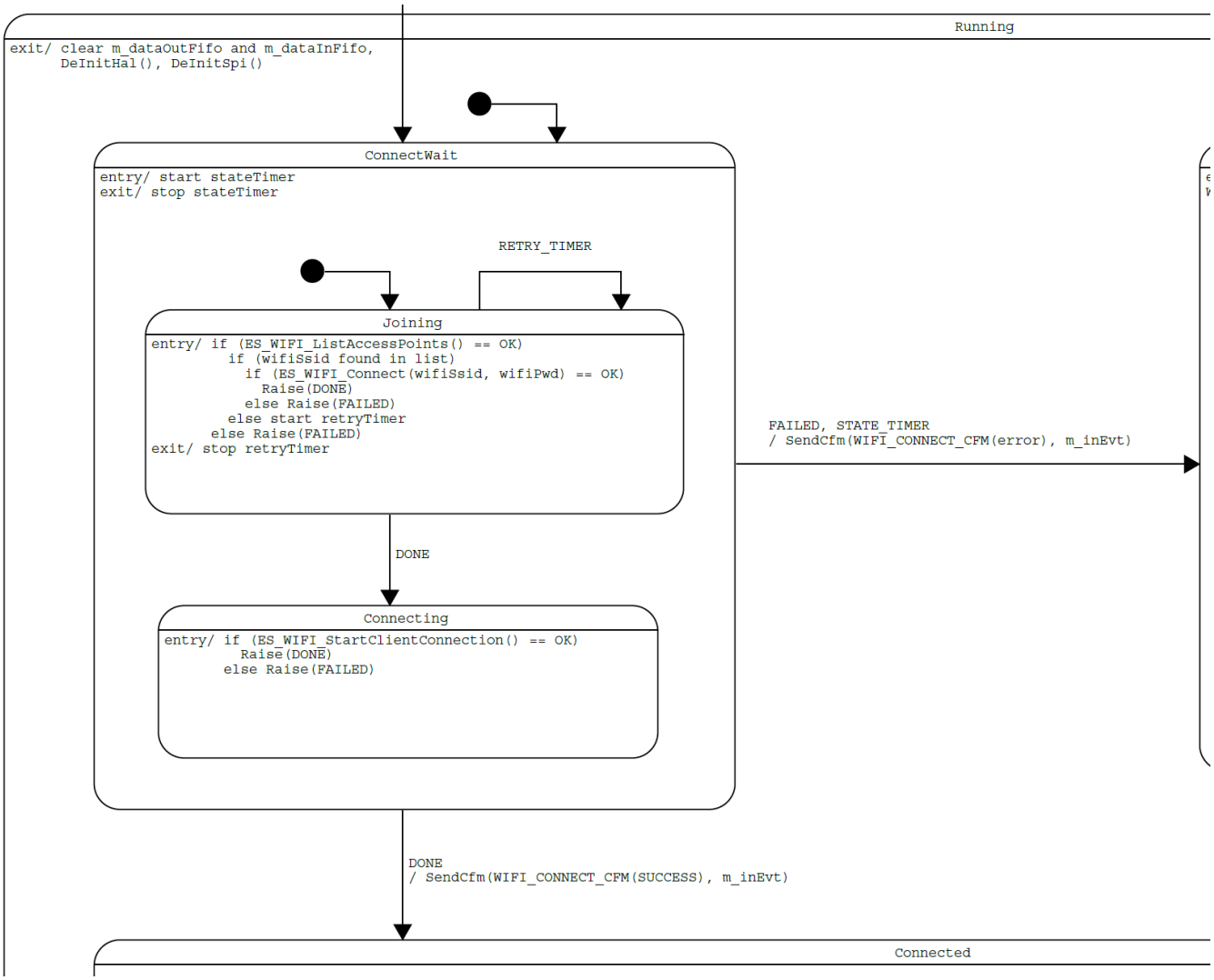
ES_WIFI_RegisterBusIO() and **ES_WIFI_Init()** are vendor-provided API functions to initialize the WiFi module. They are located in **system/BSP/Components/es_wifi/es_wifi.cpp**.



6.4 ConnectWait State

The substates in **ConnectWait** first joins the specified access point (AP) and then connects to the TCP server. It is critical that it handles any joining or connection failures such that it will automatically retry after cleaning up. The IP address and port number of the server have been saved to member variables upon **WIFI_CONNECT_REQ**.

ES_WIFI_ListAccessPoints(), ES_WIFI_Connect() and **ES_WIFI_StartClientConnection()** are vendor-provided API functions to scan APs, join an AP and connect to a TCP server respectively. They are located in **Src/system/BSP/Components/es_wifi/es_wifi.cpp**. (Note – it is a *synchronous* API.)

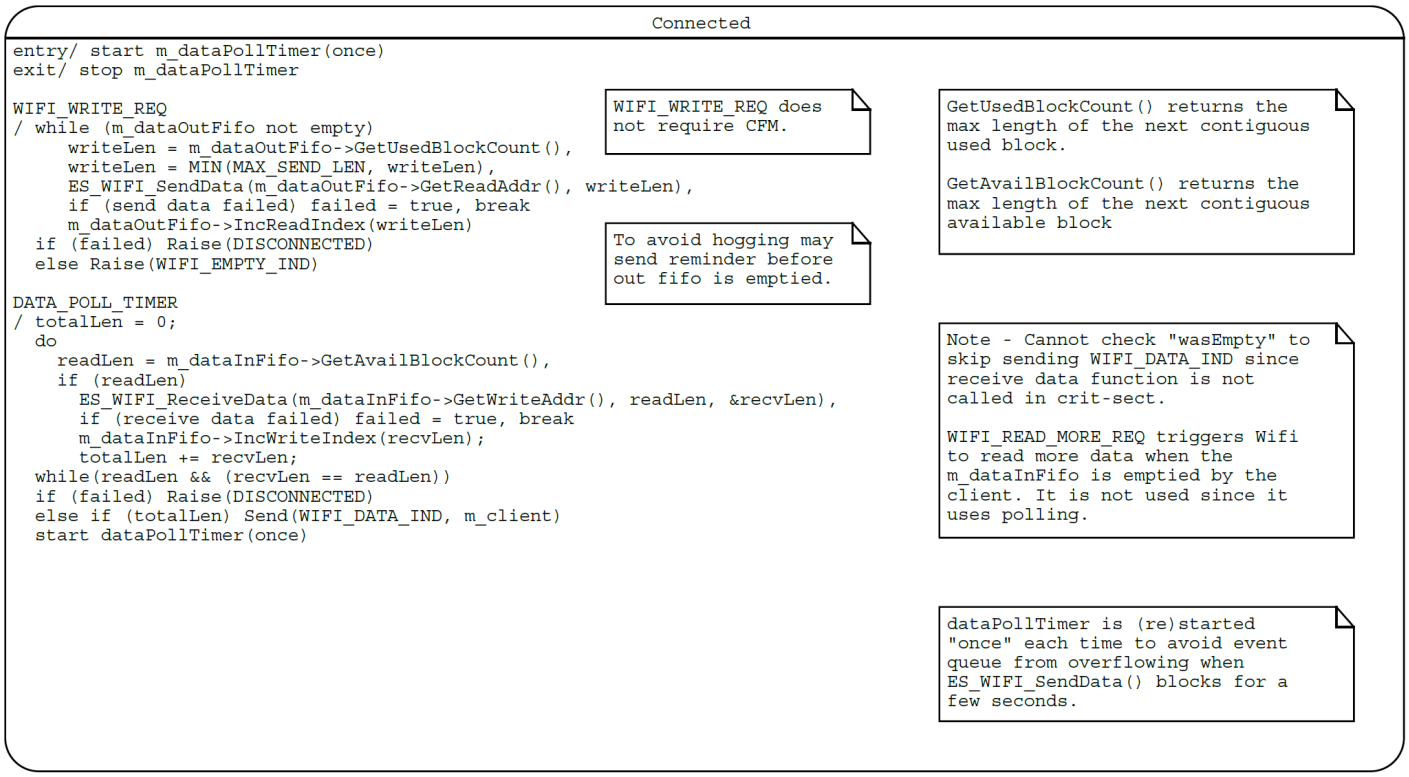


6.5 Connected State

This is the stable state after a connection to a server has been made.

It handles the **WIFI_WRITE_REQ** event to send any data stored in the m_dataOutFifo (output FIFO) to the WiFi module via the API function **ES_WIFI_SendData()**. Note the **Wifi** HSM handles the network layer of the protocol stack, and therefore it streams data out without regard to application message formats and boundaries.

It handles the **DATA_POLL_TIMER** event to periodically read any received data from the WiFi module via the API function **ES_WIFI_ReceiveData()**. We have to use *polling* since the ISM43362 WiFi module does *not* support *data-ready interrupts* to notify the MCU when data have been received. Note data are read directly into the empty space of the m_dataInFifo (input FIFO) to avoid unnecessary copying. It notifies the user HSM (**Node** active object) of data being available via the **WIFI_DATA_IND** event.



7 WiFi Module

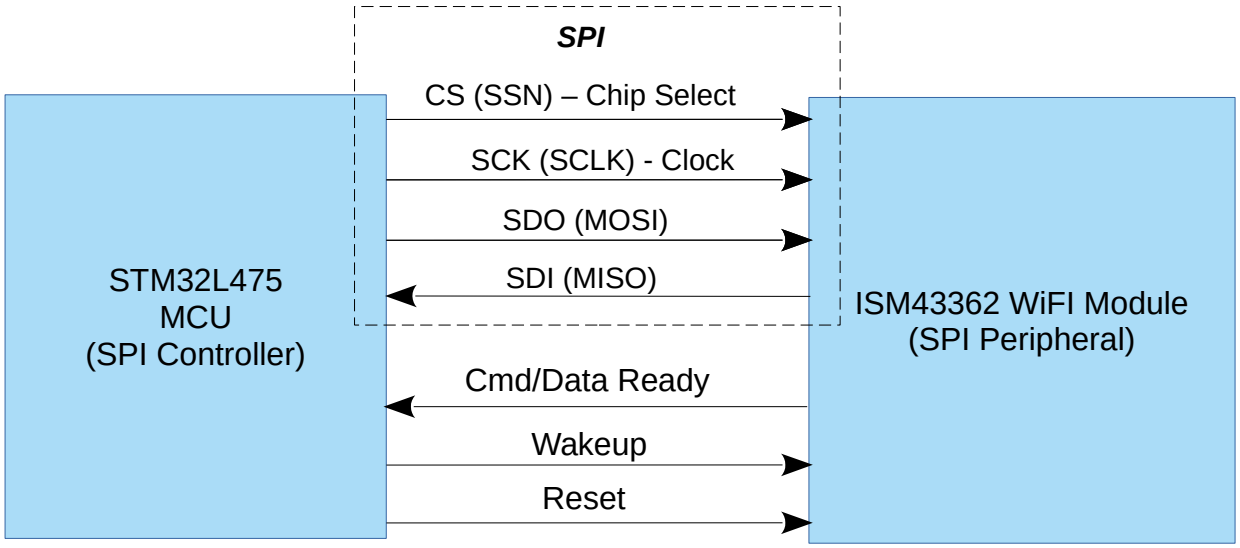
Our discovery board is equipped with an Inventek Systems ISM43362-M3G-L44 WiFi module. You may download its functional specification from this link:

https://www.inventeksys.com/wp-content/uploads/ISM43362_M3x_l44_Functional_Spec.pdf

The detailed AT command set is available at:

https://www.inventeksys.com/iwin/wp-content/uploads/IWIN_Command_Set_Users_Manual.pdf

7.1 Physical Interface



The physical interface between the MCU and the WiFi module consists of 4 SPI bus signals and 3 addition control lines. All GPIO pin usage in the system is commented in **Src/periph.cpp**. Those used for the WiFi module are specified in the constant configuration table named *CONFIG* in **Src/app/Wifi/Wifi.cpp**:

```
// Define SPI and interrupt configurations.
Wifi::Config const Wifi::CONFIG[] = {
    {
      WIFI, SPI3, SPI3_IRQn, SPI3_PRIO,
      // SCK, MISO, MOSI spi pins.
      GPIOC, GPIO_PIN_10, GPIOC, GPIO_PIN_11, GPIOC, GPIO_PIN_12, GPIO_AF6_SPI3,
      // CS, Wakeup and Reset output pins.
      GPIOE, GPIO_PIN_0, GPIOB, GPIO_PIN_13, GPIOE, GPIO_PIN_8,
      // CmdDataReady input pin.
      EXTI1_IRQn, EXTI1_PRIO, GPIOE, GPIO_PIN_1
    }
};
```

Here are some notes on the signal:

- CS (*active-low*) is used by the MCU to select an SPI peripheral to communicate with, allowing multiple peripherals to be connected to the same SPI bus.
- SCK is driven by the MCU to provide clock edges for an SPI peripheral to sample data on the SDO pin and to output data on the SDI pin.
- SDO is used by the MCU to send AT commands and TX payload data to the WiFi module.
- SDI is used by the MCU to receive AT responses and RX payload data from the WiFi module.
- Cmd/Data Ready (*active-high*) is used by the WiFi module to notify the MCU one of the following conditions:
  - It is ready to receive an AT command (including TX payload data) from the MCU (command phase), or
  - Data (AT response or RX payload data) is available to be read by the MCU (data phase).

  Note that the terms "Cmd" vs "Data" aren't very precisely defined.
- Reset (*active-low*) performs a hardware reset of the WiFi module, which is very useful to recover it from any unexpected errors within the module itself.

7.2 SW Interface

7.2.1 High-level API

ST provides a BSP library for the on-board WiFi module. It has been imported into our project under **Src/system/BSP/Components/es_wifi**. It contains:

- **es_wifi.cpp**
  It implements the top level API functions to interface with the WiFi module. These are the functions we call in our **Wifi** active object, e.g. **ES_WIFI_Connect()**, **ES_WIFI_SendData()**.
- **es_wifi.h**
- **es_wifi_config.h**
  Its allows us to customize the library e.g. the maximum number of AP it reports.

The above BSP library requires a user application to provide low level *hook* functions to interface with the physical interface, e.g. SPI bus, GPIO pins, OS mutexes, etc.

7.2.2 Low-level Hooks

The BSP library defines the prototypes of the hook functions for a user application to implement. They are located under **Src/app/Wifi**. It contains:

- **es_wifi_io.cpp**
- **es_wifi_io.h**

The hooks include the following macros and functions. ST provides sample implementation from which our implementation is modified:

```
#define WIFI_RESET_MODULE()               APP::Wifi::ResetModule()
#define WIFI_ENABLE_NSS()                 APP::Wifi::EnableCs()
#define WIFI_DISABLE_NSS()                APP::Wifi::DisableCs()
#define WIFI_IS_CMDDATA_READY()           APP::Wifi::IsCmdDataReady()

void    SPI_WIFI_MspInit(SPI_HandleTypeDef* hspi);
int8_t  SPI_WIFI_DeInit(void);
int8_t  SPI_WIFI_Init(uint16_t mode);
int8_t  SPI_WIFI_ResetModule(void);
int16_t SPI_WIFI_ReceiveData(uint8_t *pData, uint16_t len, uint32_t timeout);
int16_t SPI_WIFI_SendData( uint8_t *pData, uint16_t len, uint32_t timeout);
void    SPI_WIFI_Delay(uint32_t Delay);
```

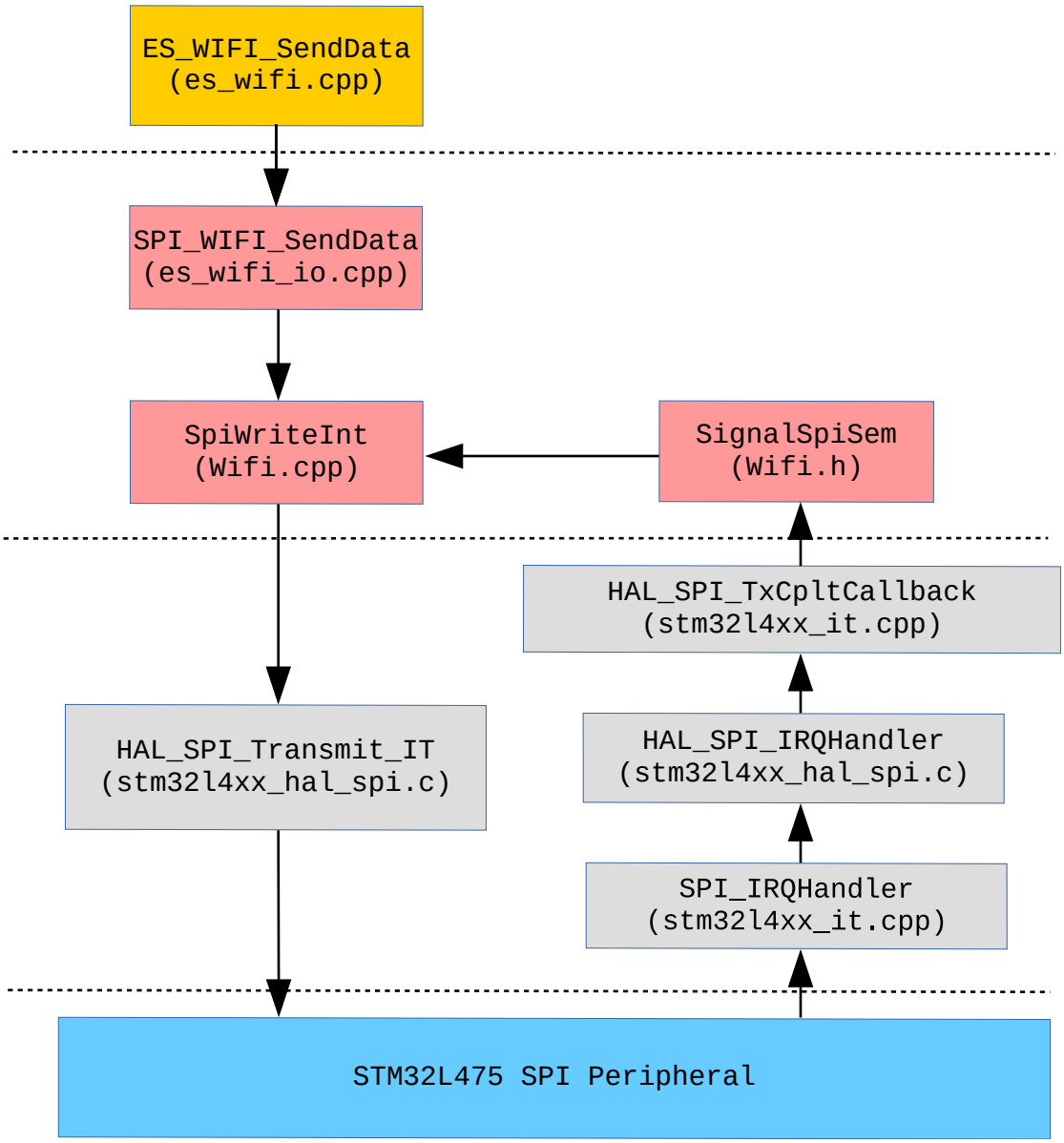
These hook functions are called by the high-level API functions to initialize the WiFi module and to send/receive data to/from the WiFi module. These hook functions are implemented in a synchronous manner, i.e. it blocks for the **Cmd/Data Ready** pin to become active (high) and for the SPI transaction to complete.

Being able to integrate our *asynchronous* state machines with *synchronous* 3rd-party libraries (low-level drivers) are essential. This opens up the possibilities of using a vast source of 3rd-party libraries most of which are written in the traditional synchronous manner. For some low-level operations, coding in synchronous style often yields more straight-forward logic.

As long as those synchronous functions do not block for too long (e.g. < 100s ms), it may be OK to call them from our state machines. The downside is the state machine won't be responsive to any other events while it is blocked by a synchronous function (Which functions are blocking in *ConnectWait* state in **Wifi**. What is the implication?). As in many engineering problems, an *optimal* solution is sometimes a *hybrid* one. We have been using this hybrid pattern in our previous examples of interfacing with sensors and LCD display.

7.2.3 Block Diagram

The following block diagram shows the data transmission path from the high-level API function **ES_WIFI_SendData()** to the SPI bus. It sends the AT command and payload data to the WiFi module.



The following block diagram shows the data receiving path initiated by the high-level API function **ES_WIFI_SendData()**. Although this API function is to send data to the network, it still needs to read an AT response back from the WiFi module to check the status of the data-sending AT command.