MySensors Documentation

Release 1.5.0

MySensors.org

Nov 08, 2017

Contents

Welcome to MySensors' unofficial documentation. We are trying to gather relevant information and API documentation which will help new users and developers.

MySensors is a library built focusing on Arduino platform to make home automation easier. There is a set of third-party libraries which complements the whole library.

If you don't find what you want in this page, you always can go to the MySensors' official website and its Forum

Protocol version 1.6

Contents:

CHAPTER 1

Getting Started

To start building your own Home Automation network, you need to setup at least one Gateway.

Since protocol version 1.6 Gateways can be used as sensor nodes, so you don't need to setup other nodes to get your networking working.

CHAPTER 2

Supported Hardware

MySensors library doesn't have any specific hardware to be used in. However, it was built around Arduino Uno that uses Atmega 328p, so, it's most likely that you will get your networking working if you use any board that uses the same microcontroller.

2.1 Tested Hardware

- Arduino Uno
- Arduino Nano
- Arduino Mini Pro
- ESP8266

CHAPTER 3

Bootloaders

Bootladers are softwares that run at the boot of your device to perform some tasks like upload your new code.

MySensors library works officially with three different bootloaders:

- *Arduino Bootloader*
- *MYSBootloader*
- *Dual Optiboot*

3.1 Arduino Bootloader

Official Link: Arduino Bootloader

It is the official Arduino Bootloader and every Arduino comes with it. There is no special feature on it.

Advantage:

- Upload over Serial

Disadvantage:

-

3.2 MYSBootloader

Offial Link: MYSBootloader

3.3 Dual Optiboot

Official Link: Dual Optiboot

CHAPTER 4

Protocol

The protocol used between the Gateway and the Controller is a simple semicolon separated list of values. The last part of each "command" is the payload. All commands ends with a newline. The serial commands has the following format:

```
<node-id>;<child-sensor-id>;<message-type>;<ack>;<sub-type>;<payload>\n
```

4.1 Message Structure Elements

Message Part	Description
node-id	The unique id of the node that sends or should receive the message (address)
child-sensor-id	Each node can have several sensors attached. This is the child-sensor-id that uniquely identifies one attached sensor
message-type	Type of message sent - See table below ack The ack parameter has the following meaning: Outgoing: 0 = unacknowledged message, 1 = request ack from destination node Incoming: 0 = normal message, 1 = this is an ack message
sub-type	Depending on messageType this field has different meaning. See tables below
payload	The payload holds the message coming in from sensors or instruction going out to actuators.

Warning: The maximum payload size is 25 bytes!

The NRF24L01+ has a maximum of 32 bytes. The MySensors library (version 1.5) uses 7 bytes for the message header.

4.2 Message Types

Type	Value	Comment
presentation	0	Sent by a node when they present attached sensors. This is usually done in setup() at startup.
set	1	This message is sent from or to a sensor when a sensor value should be updated.
req	2	Requests a variable value (usually from an actuator destined for controller).
internal	3	This is a special internal message. See table below for the details.
stream	4	Used for OTA firmware updates.

4.3 Message Sub-types

Note: If you feel that these sub-types don't fit your needs, there are some custom varaibles that you can use. However, you think that people maybe have the same issue, you can modify the liberary and add your variable, then, pull your modifications to the development branch.

4.3.1 Presentation

When a presentation message is sent from a sensor, sub-type can one the following:

The payload of presentation message will be set to the library version (node device) or an optional description for the sensors.

Type	Value	Comment	Variables
S_DOOR	0	Door and window sensors	V_TRIPPED, V_ARMED
S_MOTION	1	Motion sensors	V_TRIPPED, V_ARMED
S_SMOKE	2	Smoke sensor	V_TRIPPED, V_ARMED
S_LIGHT	3	Light Actuator (on/off)	V_STATUS (or V_LIGHT), V_WATT
S_BINARY	3	Binary device (on/off), Alias for S_LIGHT	V_STATUS (or V_LIGHT), V_WATT
S_DIMMER	4	Dimmable device of some kind	V_STATUS, V_DIMMER, V_WATT
S_COVER	5	Window covers or shades	V_UP, V_DOWN, V_STOP, V_PERCENTAGE
S_TEMP	6	Temperature sensor	V_TEMP, V_ID
S_HUM	7	Humidity sensor	V_HUM

Continued on next page

Table 4.1 – continued from previous page

Type	Value	Comment	Variables
S_BARO	8	Barometer sensor (Pressure)	V_PRESSURE, V_FORECAST
S_WIND	9	Wind sensor	V_WIND, V_GUST
S_RAIN	10	Rain sensor	V_RAIN, V_RAINRATE
S_UV	11	UV sensor	V_UV
S_WEIGHT	12	Weight sensor for scales etc.	V_WEIGHT, V_IMPEDANCE
S_POWER	13	Power measuring device, like power meters	V_WATT, V_KWH
S_HEATER	14	Heater device	V_HVAC_SETPOINT_HEAT, V_HVAC_FLOW_STATE, V_TEMP
S_DISTANCE	15	Distance sensor	V_DISTANCE, V_UNIT_PREFIX
S_LIGHT_LEVEL	16	Light sensor	V_LIGHT_LEVEL, V_LEVEL
S_ARDUINO_NODE	17	Arduino node device	
S_ARDUINO_REPEATER_NODE	18	Arduino repeating node device	
S_LOCK	19	Lock device	V_LOCK_STATUS
S_IR	20	Ir sender/receiver device	V_IR_SEND, V_IR_RECEIVE
S_WATER	21	Water meter	V_FLOW, V_VOLUME
S_AIR_QUALITY	22	Air quality sensor e.g. MQ-2	V_LEVEL, V_UNIT_PREFIX
S_CUSTOM	23	Use this for custom sensors where no other fits.	
S_DUST	24	Dust level sensor	V_LEVEL, V_UNIT_PREFIX
S_SCENE_CONTROLLER	25	Scene controller device	V_SCENE_ON, V_SCENE_OFF
S_RGB_LIGHT	26	RGB light	V_RGB, V_WATT
S_RGBW_LIGHT	27	RGBW light	V_RGBW, V_WATT
S_COLOR_SENSOR	28	Color sensor	V_RGB
S_HVAC	29	Thermostat/HVAC device	V_HVAC_SETPOINT_HEAT, V_HVAC_SETPOINT_COLD, V_HVAC_FLOW_STATE, V_HVAC_FLOW_MODE, V_HVAC_SPEED
S_MULTIMETER	30	Multimeter device	V_VOLTAGE, V_CURRENT, V_IMPEDANCE
S_SPRINKLER	31	Sprinkler device	V_STATUS, V_TRIPPED
S_WATER_LEAK	32	Water leak sensor	V_TRIPPED, V_ARMED

Continued on next page

Table 4.1 – continued from previous page

Type	Value	Comment	Variables
S_SOUND	33	Sound sensor	V_LEVEL(dB), V_TRIPPED, V_ARMED
S_VIBRATION	34	Vibration sensor	V_LEVEL(Hz), V_TRIPPED, V_ARMED
S_MOISTURE	35	Moisture sensor	V_LEVEL, V_TRIPPED, V_ARMED
S_INFO	36	LCD text device	V_TEXT
S_GAS	37	Gas meter	V_FLOW, V_VOLUME
S_GPS	38	GPS Sensor	V_POSITION

4.3.2 Set & Req

When a set or request message is being sent, the sub-type has to be one of the following:

Type	Value	Comment	Used by
V_TEMP	0	Temperature	S_TEMP, S_HEATER, S_HVAC
V_HUM	1	Humidity	S_HUM
V_STATUS	2	Binary status. 0=off 1=on	S_LIGHT, S_DIMMER, S_SPRINKLER, S_HVAC, S_HEATER
V_PERCENTAGE	3	Percentage value. 0-100(%)	S_DIMMER
V_PRESSURE	4	Atmospheric Pressure	S_BARO
V_FORECAST	5	Whether forecast. One of “stable”, “sunny”, “cloudy”, “unstable”, “thunderstorm” or “unknown”	S_BARO
V_RAIN	6	Amount of rain	S_RAIN
V_RAINRATE	7	Rate of rain	S_RAIN
V_WIND	8	Windspeed	S_WIND
V_GUST	9	Gust	S_WIND
V_DIRECTION	10	Wind direction	S_WIND
V_UV	11	UV light level	S_UV
V_WEIGHT	12	Weight (for scales etc)	S_WEIGHT
V_DISTANCE	13	Distance	S_DISTANCE
V_IMPEDANCE	14	Impedance value	S_MULTIMETER, S_WEIGHT

Continued on next page

Table 4.2 – continued from previous page

Type	Value	Comment	Used by
V_ARMED	15	Armed status of a security sensor. 1=Armed, 0=Bypassed	S_DOOR, S_MOTION, S_SMOKE, S_SPRINKLER, S_WATER_LEAK, S_SOUND, S_VIBRATION, S_MOISTURE
V_TRIPPED	16	Tripped status of a security sensor. 1=Tripped, 0=Untripped	S_DOOR, S_MOTION, S_SMOKE, S_SPRINKLER, S_WATER_LEAK, S_SOUND, S_VIBRATION, S_MOISTURE
V_WATT	17	Watt value for power meters	S_POWER, S_LIGHT, S_DIMMER, S_RGB, S_RGBW
V_KWH	18	Accumulated number of KWH for a power meter	S_POWER
V_SCENE_ON	19	Turn on a scene	S_SCENE_CONTROLLER
V_SCENE_OFF	20	Turn off a scene	S_SCENE_CONTROLLER
V_HVAC_FLOW_STATE	21	Mode of header. One of "Off", "HeatOn", "CoolOn", or "AutoChangeOver"	S_HVAC, S_HEATER
V_HVAC_SPEED	22	HVAC/Heater fan speed ("Min", "Normal", "Max", "Auto")	S_HVAC, S_HEATER
V_LIGHT_LEVEL	23	Uncalibrated light level. 0-100%. Use V_LEVEL for light level in lux.	S_LIGHT_LEVEL
V_VAR1	24	Custom value	Any device
V_VAR2	25	Custom value	Any device
V_VAR3	26	Custom value	Any device
V_VAR4	27	Custom value	Any device
V_VAR5	28	Custom value	Any device
V_UP	29	Window covering. Up.	S_COVER
V_DOWN	30	Window covering. Down.	S_COVER
V_STOP	31	Window covering. Stop.	S_COVER

Continued on next page

Table 4.2 – continued from previous page

Type	Value	Comment	Used by
V_IR_SEND	32	Send out an IR-command	S_IR
V_IR_RECEIVE	33	This message contains a received IR-command	S_IR
V_FLOW	34	Flow of water (in meter)	S_WATER
V_VOLUME	35	Water volume	S_WATER
V_LOCK_STATUS	36	Set or get lock status. 1=Locked, 0=Unlocked	S_LOCK
V_LEVEL	37	Used for sending level-value	S_DUST, S_AIR_QUALITY, S_SOUND (dB), S_VIBRATION (hz), S_LIGHT_LEVEL (lux)
V_VOLTAGE	38	Voltage level	S_MULTIMETER
V_CURRENT	39	Current level	S_MULTIMETER
V_RGB	40	RGB value transmitted as ASCII hex string (I.e "ff0000" for red)	S_RGB_LIGHT, S_COLOR_SENSOR
V_RGBW	41	RGBW value transmitted as ASCII hex string	S_RGBW_LIGHT
V_ID	42	Optional unique sensor id (e.g. OneWire DS1820b ids)	S_TEMP
V_UNIT_PREFIX	43	Allows sensors to send in a string representing the unit prefix to be displayed in GUI. This is not parsed by controller! E.g. cm, m, km, inch.	S_DISTANCE, S_DUST, S_AIR_QUALITY
V_HVAC_SETPOINT_COOL	44	HVAC cold setpoint	S_HVAC
V_HVAC_SETPOINT_HEAT	45	HVAC/Heater setpoint	S_HVAC, S_HEATER

Continued on next page

Table 4.2 – continued from previous page

Type	Value	Comment	Used by
V_HVAC_FLOW_MODE	46	Flow mode for HVAC (“Auto”, “ContinuousOn”, “PeriodicOn”)	S_HVAC
V_TEXT	47	Text message to display on LCD or | controller device	S_INFO
V_CUSTOM	48	Custom messages used for controller | /inter node specific commands, | preferably using S_CUSTOM device type.	S_CUSTOM
V_POSITION	49	GPS position and altitude. Payload: | latitude;longitude;altitude(m). | E.g. “55.722526;13.017972;18”	S_GPS
V_IR_RECORD	50	Record IR codes S_IR for playback	S_IR

4.3.3 Internal

The internal messages are used for different tasks in the communication between sensors, the gateway to controller and between sensors and the gateway.

When an internal messages is sent, the sub-type has to be one of the following:

Type	Value	Comment
I_BATTERY_LEVEL	0	Use this to report the battery level (in percent 0-100).
I_TIME	1	Sensors can request the current time from the Controller using this message. The time will be reported as the seconds since 1970
I_VERSION	2	Used to request gateway version from controller.
I_ID_REQUEST	3	Use this to request a unique node id from the controller.
I_ID_RESPONSE	4	Id response back to sensor. Payload contains sensor id.
I_INCLUSION_MODE	5	Start/stop inclusion mode of the Controller (1=start, 0=stop).
I_CONFIG	6	Config request from node. Reply with (M)etric or (I)mperal back to sensor.
I_FIND_PARENT	7	When a sensor starts up, it broadcast a search request to all neighbor nodes. They reply with a I_FIND_PARENT_RESPONSE.
I_FIND_PARENT_RESPONSE	8	Reply message type to I_FIND_PARENT request.
I_LOG_MESSAGE	9	Sent by the gateway to the Controller to trace-log a message
I_CHILDREN	10	A message that can be used to transfer child sensors (from EEPROM routing table) of a repeating node.
I_SKETCH_NAME	11	Optional sketch name that can be used to identify sensor in the Controller GUI
I_SKETCH_VERSION	12	Optional sketch version that can be reported to keep track of the version of sensor in the Controller GUI.
I_REBOOT	13	Used by OTA firmware updates. Request for node to reboot.
I_GATEWAY_READY	14	Send by gateway to controller when startup is complete.
I_REQUEST_SIGNING	15	Used between sensors when initialting signing.
I_GET_NONCE	16	Used between sensors when requesting nonce.
I_GET_NONCE_RESPONSE	17	Used between sensors for nonce response.
I_HEARTBEAT	18	
I_PRESENTATION	19	
I_DISCOVER	20	
I_DISCOVER_RESPONSE	21	

4.3.4 Stream

Type	Value	Comment
ST_FIRMWARE_CONFIG_REQUEST	0	
ST_FIRMWARE_CONFIG_RESPONSE	1	
ST_FIRMWARE_REQUEST	2	
ST_FIRMWARE_RESPONSE	3	
ST_SOUND	4	Used to transfer sound to controller
ST_IMAGE	5	Used to transfer image to controller

4.4 Examples

Received message from radio network from one of the sensors: Incoming presentation message from node 12 with child sensor 6. The presentation is for a binary light S_LIGHT. The payload holds a description of the sensor. Gateway passes this over to the controller.

```
12;6;0;0;3;My Light\n
```

Received message from radio network from one of the sensors: Incoming temperature V_TEMP message from node 12 with child sensor 6. The gateway passed this over to the controller.

```
12;6;1;0;0;36.5\n
```

Received command from the controller that should be passed to radio network: Outgoing message to node 13. Set V_LIGHT variable to 1 (=turn on) for child sensor 7. No ack is requested from destination node.

```
13;7;1;0;2;1\n
```

Note: There are some messages which are processed by the MySensors library. Which means that you don't have to implement an action for them.

E.g.: I_REBOOT.

CHAPTER 5

Controllers

Controllers are softwares which are responsible for answer the requests from the nodes and send commands made by users to nodes.

Example of tasks made by controllers:

- Auto assign an ID to a node
- Keep tracking of node on the network

All the magic is made by the controllers, the nodes are just responsible for collect and send the data. Then, the controller gather the all data and "think" how it should respond.

You can control your nodes send commands over your gateway. However, if you really want to automate your home, you must use a controller. If you don't know how or you don't want to developer a new one, you can use one of them listed below.

5.1 List of controllers

- Ago Control
- Calaos
- Domoticz
- DomotiGa
- FHEM
- Freedomotic
- Home Assistant
- Homeseer
- Indigo Domotics
- Jeedom

- MajorDoMo
- MyController.org
- MyNetSensors
- OpenHAB
- openLuup
- PiDome
- Pimatic
- Vera

5.2 Other tools

- MYSController
- Yveaux NRF24 Sniffer

CHAPTER 6

Nodes

Test

CHAPTER 7

Security

Signing support has now been implemented and is available from release 1.5 of the MySensors library.Since doxygen documentation is not yet available for the master branch from which releases really made, I have documented signing support in this post. If you use the development branches, I strongly urge you to read the "live" documentation which is available here. You can find the signing documentation under "modules" and "MySensors internal APIs and functionalities > MySigning".

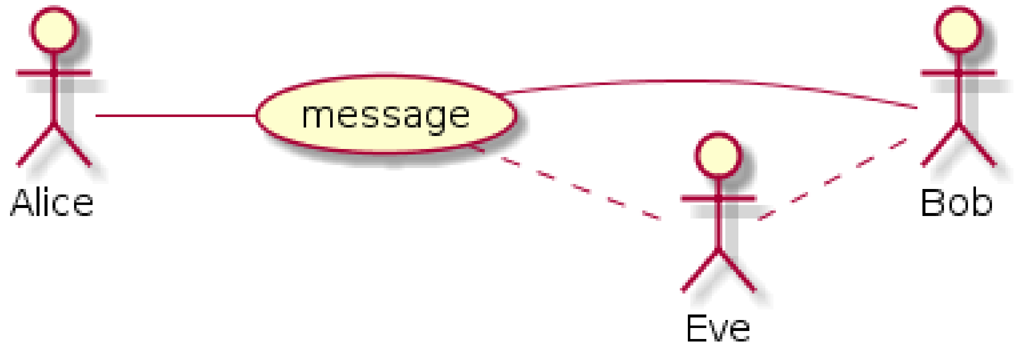
In an effort to normalize the technical playing field for all forum members I will describe what this means, and why it is important. And I will also briefly touch on the generic concept of authentication and verification, motivate the need for this (and explain why encryption is not part of this) and go on to explain the architecture used for MySensors and how to enable and use this new feature.

7.1 Background and concepts

Suppose two participants, Alice and Bob, wants to exchange a message. Alice sends a message to Bob. In MySensors "language" Alice could be a gateway and Bob an actuator (light swich, electronic lock, etc). But to be generic, we will substitute the term "gateway" with Alice and a "node" with Bob (although the reverse relationship is also supported).

Alice sends a message to Bob. This message can be heard by anyone who wants to listen (and also by anyone that is within "hearing" distance). Normally, this is perhaps not a big issue. Nothing Alice says to Bob may be secret or sensitive in any way.

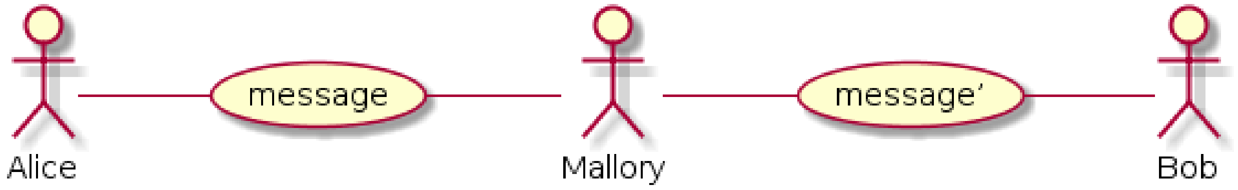
However, sometimes (or perhaps always) Bob want to be sure that the message Bob receives actually came from Alice. In cryptography, this is known as authenticity. Bob need some way of determining that the message is authentic from Alice, when Bob receives it. This prevent an eavesdropper, Eve, to trick Bob into thinking it was Alice that sent a message Eve in fact transmitted. Bob also need to know how to determine if the message has been repeated. Eve could record a message sent by Alice that Bob accepted and then send the same message again. Eve could also in some way prevent Bob from receiving the message and delay it in order to permit the message to arrive to Bob at a time Eve chooses, instead of Alice. Such an attack is known as a **replay attack**. *Authenticity* permits Bob to determine if Alice is the true sender of a message.



It can also be interesting for Bob to know that the message Alice sent has not been tampered with in any way. This is the *integrity* of the message. We now introduce Mallory, who could be intercepting the communication between Alice and Bob and replace some parts of the message but keeping the parts that authenticate the message. That way, Bob still trusts Alice to be the source, but the contents of the message was not the content Alice sent. Bob needs to be able to determine that the contents of the message was not altered after Alice sent it.

Mallory would in this case be a **man-in-the-middle** attacker.

Integrity permits Bob to verify that the messages received from Alice has not been tampered with. This is achieved by adding a signature to the message, which Bob can inspect to validate that Alice is the author.



The signing scheme used, needs to address both these attack scenarios. Neither Eve nor Mallory must be permitted to interfere with the message exchange between Alice and Bob.

The key challenge to implementing a secure signing scheme is to ensure that every signature is different, even if the message is not. If not, **replay attacks** would be very hard to prevent.

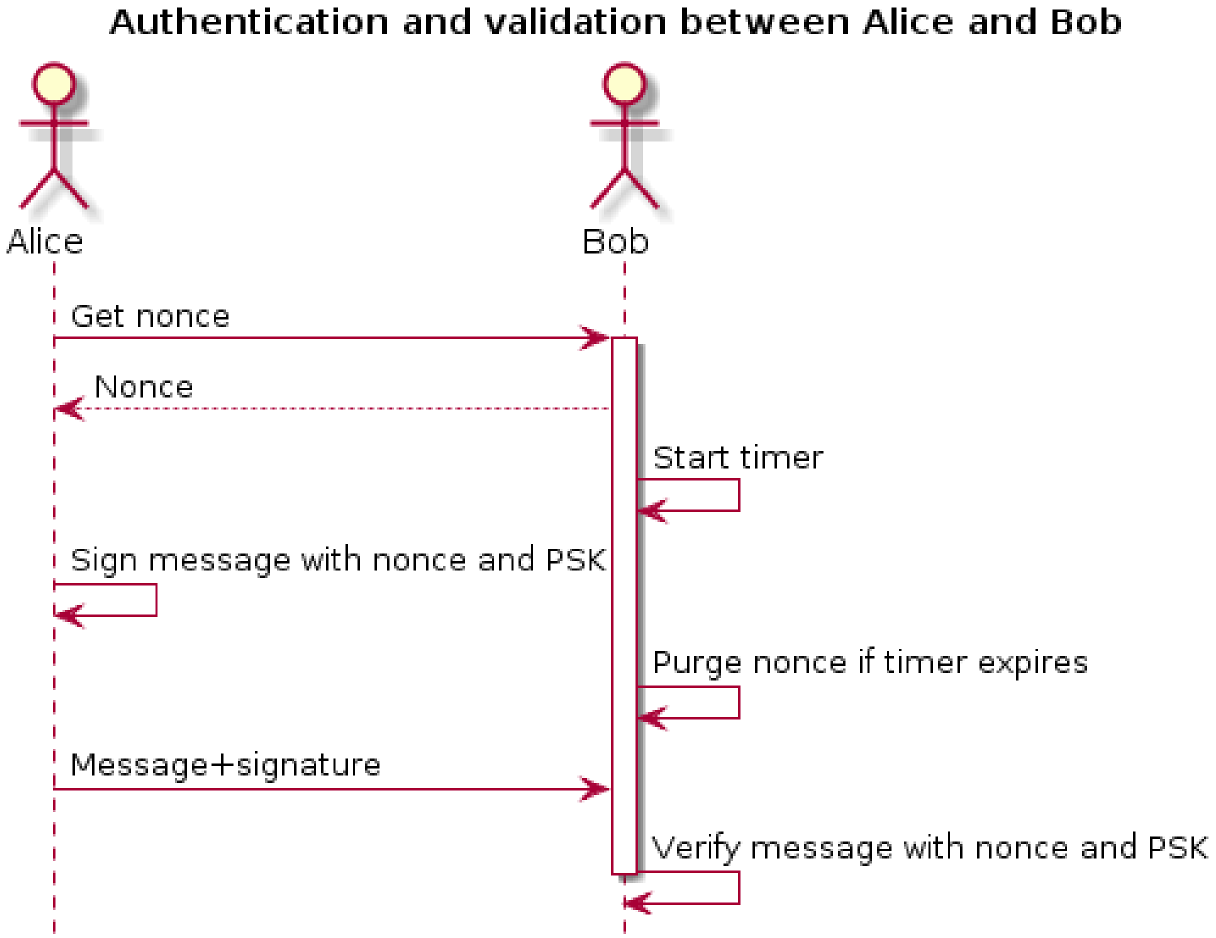
One way of doing this is to increment some counter on the sender side and include it in the signature. This is however predictable.

A better option would be to introduce a random number to the signature. That way, it is impossible to predict what the signature will be. The problem is, that also makes it impossible for the receiver (Bob) to verify that the signature is valid. A solution to this is to let Bob generate the random number, keep it in memory and send it to Alice. Alice can then use the random number in the signature calculation and send the signed message back to Bob who can validate the signature with the random number used. This random number is in cryptography known as a nonce or salt.

However, Mallory might be eavesdropping on the communication and snoop up the nonce in order to generate a new valid signature for a different message. To counter this, both Alice and Bob keep a secret that only they know. This secret is never transmitted over the air, nor is it revealed to anybody. This secret is known as a pre-shared key (PSK).

If Even or Mallory are really sophisticated, he/she might use a **delayed replay attack**. This can be done by allowing Bob to transmit a nonce to Alice. But when Alice transmits the uniquely signed message, Mallory prevents Bob from receiving it, to a point when Mallory decides Bob should receive it. An example of such an attack is described here. This needs to be addressed as well, and one way of doing this is to have Bob keep track of time between a transmitted nonce and a signed message to verify. If Bob is asked for a nonce, Bob knows that a signed message is going to arrive “soon”. Bob can then decide that if the signed message does not arrive within a predefined timeframe, Bob throws away the generated nonce and thus makes it impossible to verify the message if it arrives late.

The flow can be described like this:


Authentication and validation between Alice and Bob
Alice
Bob
Get nonce
Nonce
Start timer
Sign message with nonce and PSK
Purge nonce if timer expires
Message+signature
Verify message with nonce and PSK

The benefits for MySensors to support this are obvious. Nobody wants others to be able to control or manipulate any actuators in their home.

7.2 Why encryption is not part of this

Well, some could be uncomfortable with somebody being able to snoop temperatures, motion or the state changes of locks in the environment. Signing does **not** address these issues. *Encryption* is needed to prevent this.

It is my personal standpoint that encryption should not be part of the MySensors “protocol”. The reason is that a gateway and a node does not really care about messages being readable or not by “others”. It makes more sense that such guarantees are provided by the underlying transmission layer (RF solution in this case). It is the information transmitted over the air that needs to be secret (if user so desires). The “trust” level on the other hand needs to go all the way into the sketches (who might have different requirements of trust depending on the message participant), and for this reason, it is more important (and less complicated) to ensure authenticity and integrity at protocol-level as message contents is still readable throughout the protocol stack. But as soon as the message leaves the “stack” it can be scramble into “garbage” when transmitted over the air and then reassembled by a receiving node before being fed in “the clear” up the stack at the receiving end.

There are methods and possibilities to provide encryption also in software, but if this is done, it is my recommendation that this is done after integrity- and authentication information has been provided to the message (if this is desired).

Integrity and authentication is of course not mandatory and some might be happy with only having encryption to cover their need for security. I, however, have only focused on integrity and authenticity while at the same time keeping the current message routing mechanisms intact and therefore leave the matter of secrecy to be implemented in the "physical" transport layer. With the integrity and authenticity handled in the protocol it ought to be enough for a simple encryption (nonce-less AES with a PSK for instance) on the message as it is sent to the RF backend. Atmel does provide such circuits as well but I have not investigated the matter further as it given the current size of the ethernet gateway sketch is close to the size limit on an Arduino Nano, so it will be difficult to fit this into some existing gateway designs.

Also it is worth to consider that the state of a lock can just as readily be determined by simply looking at the door in question or attempting to open it, so obfuscating this information will not necessarily deter an attacker in any way. Nevertheless, I do acknowledge that people find the fact that all information is sent "in the clear" even if it require some technical effort for an intruder to obtain and inspect this information. So I do encourage the use of encrypting transport layers.

This is however not covered by this topic nor my implementation.

7.3 How this is done

There exist many forms of message signature solutions to combat Eve and Mallory. Most of these solutions are quite complex in term of computations, so I elected to use an algorithm that an external circuit is able to process. This has the added benefit of protecting any keys and intermediate data used for calculating the signature so that even if someone were to actually steal a sensor and disassembled it, they would not be able to extract the keys and other information from the device.

A common scheme for message signing (authenticity and integrity) is implemented using HMAC which in combination with a strong hash function provides a very strong level of protection.

The Atmel ATSHA204A is a low-cost, low-voltage/current circuit that provides HMAC calculation capabilities with SHA256 hashing which is a (currently) virtually unbreakable combination. If SHA256 were to be hacked, a certain cryptocurrency would immediately be rendered worthless. The ATSHA device also contain a random number generator (RNG) which enables the generation of a good nonce, as in, non-predictable. As I acknowledge that some might not want to use an additional external circuit, I have also implemented a software version of the ATSHA device, capable of generating the same signatures as the ATSHA device does. Because it is pure-software however, it does not provide as good nonces (it uses the Arduino pseudo-random generator) and the HMAC key is stored in SW and is therefore readable if the memory is dumped. It also naturally claims more flash space due to the more complex software. But for indoor sensors/actuators this might be good enough for most people.

7.4 How to use this

Before we begin with the details, I just want to emphasize that signing is completely optional and although it is enabled by default, it will use a default backend that does not require signing and does not enforce signature checks. However, if you really do not want any additional "cost" in program space related to signing, you can disable MY_SIGNING_FEATURE in MyConfig.h.

Firstly, you need to make sure MY_SIGNING_FEATURE is enabled in MyConfig.h. You then select which backend to use by passing the appropriate handle when constructing the MySensor object. The handle is passed as the third argument (example here uses the real ATSHA without whitelisting):

```
#include <MySigningAtsha204.h>

MyTransportNRF24 radio;  // NRFRF24L01 radio driver
MyHwATMega328 hw; // Select AtMega328 hardware profile
```

```
MySigningAtsha204 signer; // Select HW ATSHA signing backend

MySensor gw(radio, hw, signer);
```

If the software equivalent if the ATSHA is desired instead do

```
#include <MySigningAtsha204Soft.h>

MyTransportNRF24 radio;  // NRFRF24L01 radio driver
MyHwATMega328 hw; // Select AtMega328 hardware profile

// Change the soft_serial value to an arbitrary value for proper security
uint8_t soft_serial[SHA204_SERIAL_SZ] = {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09}
↪;
MySigningAtsha204Soft signer(true, 0, NULL, soft_serial);  // Select SW ATSHA signing␣
↪backend

MySensor gw(radio, hw, signer);
```

It is legal to mix `MySigningAtsha204` and `MySigningAtsha204Soft` backends in a network. They work together.

Secondly, you need to verify the configuration for the backend. Currently, only `MySigningAtsha204` and `MySigningAtsha204Soft` backends have a specific configuration. For MySigningAtsha204 it is the pin the device is connected to. In `MyConfig.h` there are defaults for sensors and gateways which you might need to adjust to match your personal build. The setting is defined using MY_ATSHA204_PIN and the default is to use pin A3. For `MySigningAtsha204Soft`, an unconnected analog pin is required to set a random seed for the pseudo-random generator. It is important that the pin is floating, or the output of the pseudo-random generator will be predictable, and thus compromise the signatures. The setting is defined using `MY_RANDOMSEED_PIN` and the default is to use pin A7.

Thirdly, if you use the `MySigningAtsha204Soft` backend, you need to create/set a HMAC key to use. This key is 32 bytes wide and should be an arbitrarily chosen value. A string is OK, but as this key never needs to be "parsed" a completely random number is recommended. The key is stored in our sketch and is passed when constructing the `MySigningAtsha204Soft` object.

If you use the "real" ATSHA204A, before any signing operations can be done, the device needs to be personalized. This can be a daunting process as it involves irreversibly writing configurations to the device, which cannot be undone. I have however tried to simplify the process as much as possibly by creating a helper-sketch specifically for this purpose in libraries/MySensors/examples/Sha204Personalizer/sha204_personalizer.ino

The process of personalizing the ATSHA204A involves

- Writing and locking chip configuration
- (optionally) Generating and (mandatory) writing HMAC key
- (optionally) Locking data sections

First execute the sketch without modifications to make sure communications with the device can be established. It is highly recommended that the first time this is done, a device with serial debug possibilities is used. When this has been confirmed, it is time to decide what type of personalization is desired. There are a few options here.

Firstly, enable `LOCK_CONFIGURATION` to allow the sketch to lock the chip configuration. The sketch will write the default settings to the chip because these are fine for our purposes. This also enables the RNG which is required to allow the sketch to automatically generate a PSK if this is desired. Now it is possible to execute the sketch to lock the configuration and enable the RNG.

Next step is to decide if a new key should be generated or an existing key should be stored to the device. This is determined using `USER_KEY_DATA`, which, if defined, will use the data in the variable user_key_data. If

USER_KEY_DATA is disabled, the RNG will be used to generate a key. This key obviously need to be made available to you so you can use it in other devices in the network, and this key is therefore also printed on the serial console when it has been generated. The key (generated or provided) will be written to the device unless SKIP_KEY_STORAGE is set. As long as the data zone is kept unlocked the key can be replaced at any time. However, Atmel suggests the data region to be locked for maximum security. On the other hand, they also claim that the key is not readable from the device even if the data zone remains unlocked so the need for locking the data region is optional for MySensors usage.

For devices that does not have serial debug possibilities, it is possible to set SKIP_UART_CONFIRMATION, but it is required to set :code'USER_KEY_DATA' if this option is enabled since a generated and potentially unknown key could be written to the device and thus rendering it useless (if the data zone is also locked).

For devices with serial debug possibilities it is recommended to not use SKIP_UART_CONFIRMATION as the sketch without that setting will ask user to send a "space" character on the serial terminal before any locking operations are executed as an additional confirmation that this irreversible operation is done. However, if a number of nodes are to undergo personalization, this option can be enabled to streamline the personalization. This is a condensed description of settings to fully personalize and lock down a set of sensors (and gateways): Pick a "master" device with serial debug port.

Set the following sketch configuration of the personalizer:

Enable LOCK_CONFIGURATION
Disable LOCK_DATA
Enable SKIP_KEY_STORAGE
Disable SKIP_UART_CONFIGURATION
Disable USER_KEY_DATA

Execute the sketch on the "master" device to obtain a randomized key. Save this key to a secure location and keep it confidential so that you can retrieve it if you need to personalize more devices later on.

Now reconfigure the sketch with these settings:

Enable LOCK_CONFIGURATION
Enable LOCK_DATA (if you are sure you do not need to replace/revoke the key, this is the most secure option to protect from key readout according to Atmel, but they also claim that key is not readable even if data region remains unlocked from the slot we are using)
Disable SKIP_KEY_STORAGE
Enable SKIP_UART_CONFIGURATION
Enable USER_KEY_DATA
Put the saved key in the user_key_data variable.

Now execute the sketch on all devices you want to personalize with this secret key. That's it. Personalization is done and the device is ready to execute signing operations which are valid only on your personal network.

In case you want to be able to "whitelist" trusted nodes (in order to be able to revoke them in case they are lost) you also need to take note of the serial number of the ATSHA device. This is unique for each device. The serial number is printed in a copy+paste friendly format by the personalizer for this purpose.

Signing in the MySensors network is driven from the receiving nodes. That means, if a node require signing it will inform the gateway of this. To instruct a node to require signing by the gateway, provide a suitable backend to the library constructor. Both MySigningAtsha204 and MySigningAtsha204Soft backends will by-default require signing when used. The default constructors for these backends can be overridden to disable signing requirements if the node does not require signed messages but still need the ability to verify messages (like a gateway).

Example for a node that uses ATSHA and require signing:

```
#include <MySigningAtsha204.h>
MyTransportNRF24 radio;  // NRFRF24L01 radio driver
MyHwATMega328 hw; // Select AtMega328 hardware profile
MySigningAtsha204 signer; // Select ATSHA204A physical signing circuit
MySensor gw(radio, hw, signer);
```

Example for a gateway that uses ATSHA signing in software and do not require signing from nodes:

```
#include <MySigningAtsha204Soft.h>
MyTransportNRF24 radio;  // NRFRF24L01 radio driver
MyHwATMega328 hw; // Select AtMega328 hardware profile
uint8_t soft_serial[SHA204_SERIAL_SZ] = {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09}
↪;
MySigningAtsha204Soft signer(false, 0, NULL, soft_serial);  // Select ATSHA204A␣
↪software signing backend
MySensor gw(radio, hw, signer);
```

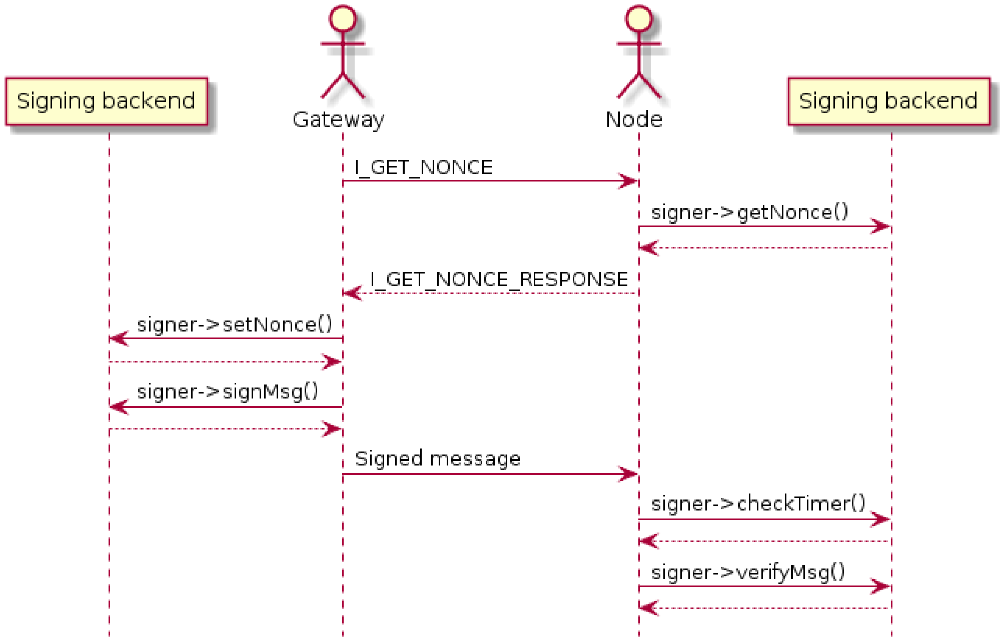
If a node does require signing, any unsigned message sent to the node will be rejected. This also applies to the gateway. However, the difference is that the gateway will only require signed messages from nodes it knows in turn require signed messages.

A node can also inform a different node that it expects to receive signed messages from it. This is done by transmitting an internal message of type `I_REQUEST_SIGNING` and provide a boolean for payload, set to true.

All nodes and gateways in a network maintain a table where the signing preferences of all nodes are stored. This is also stored in EEPROM so if the gateway reboots, the nodes does not have to retransmit a signing request to the gateway for the gateway to realize that the node expect signed messages. Also, the nodes that do not require signed messages will also inform gateway of this, so if you reprogram a node to stop require signing, the gateway will adhere to this as soon as the new node has presented itself to the gateway.

The following sequence diagram illustrate how messages are passed in a MySensors network with respect to signing:

Authentication and validation with MySensors (Gateway transmitting a signed message to a Node)



None of this activity is “visible” to you (as the sensor sketch implementor). All you need to do is to set your preferences in `MyConfig.h`, depending on chosen backend, do personalization or key configurations and set the `requestSignatures` parameter to true. That is enough to enable protection from both Eve and Mallory in your network (although because of the lack of encryption, Eve can eavesdrop, but not do anything about, your messages).

7.5 Whitelisting and node revocation

Consider the situation when you have set up your secure topology. We use the remotely operated garage door as an example:

- You have a node inside your garage (therefore secure and out of reach from prying eyes) that controls your garage door motor. This node require signing since you do not want an unauthorized person sending it orders to open the door.
- You have a keyfob node with a signing backend that uses the same PSK as your door opener node.

In this setup, your keyfob can securely transmit messages to your door node since the keyfob will sign the messages it sends and the door node will verify that these were sent from a trusted node (since it used the correct PSK). If the keyfob does not sign the messages, the door node will not accept them.

One day your keyfob gets stolen or you lost it or it simply broke down.

You know end up with a problem; you need some way of telling your door node that the keyfob in question cannot be trusted any more. Furthermore, you maybe locked the data region in your door nodes ATSHA device and is not able to revoke/change your PSK, or you have some other reason for not wanting to replace the PSK. How do you make sure that the "rogue" keyfob can be removed from the "trusted chain"?

The answer to this is whitelisting. You let your door node keep a whitelist of all nodes it trusts. If you stop trusting a particular node, you remove it from the nodes whitelist, and it will no longer be able to communicate signed messages to the door node.

This is achieved by 'salting' the signature with some node-unique information known to the receiver. In the case of ATSHA204A this is the unique serial number programmed into the circuit. This unique number is never transmitted over the air in clear text, so Eve will not be able to figure out a "trusted" serial by snooping on the traffic. Instead the value is hashed together with the senders NodeId into the HMAC signature to produce the final signature. The receiver will then take the originating NodeId of the signed message and do the corresponding calculation with the serial it has stored in it's whitelist if it finds a matching entry in it's whitelist.

Whitelisting is an optional alternative because it adds some code which might not be desirable for every user. So if you want the ability to provide and use whitelists, as well as transmitting to a node with a whitelist, you need to enable `MY_SECURE_NODE_WHITELISTING` in `MyConfig.h`. The whitelist is provided when constructing the signing backend as follows (example is a node that require signing as well):

```
#include <MySigningAtsha204.h>
MyTransportNRF24 radio;  // NRFRF24L01 radio driver
MyHwATMega328 hw; // Select AtMega328 hardware profile
#ifdef MY_SECURE_NODE_WHITELISTING
whitelist_entry_t node_whitelist[] = {
  { .nodeId = 55, // Just some value, this need to be changed  to the NodeId of the␣
↪trusted node
    .serial = {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09} } // This need to change␣
↪to the serial of the trusted node
};
MySigningAtsha204 signer(true, 1, node_whitelist);  // Select ATSHA204A software␣
↪signing backend with one entry in the whitelist
#else
MySigningAtsha204 signer;  // Select ATSHA204A software signing backend
#endif
MySensor gw(radio, hw, signer);
```

The “soft” backend of course also support whitelisting. However, since it does not contain a unique identifier, you have to provide an additional constructor argument when you enable whitelisting as illustrated in this example:

```
#include <MySigningAtsha204Soft.h>
MyTransportNRF24 radio;  // NRFRF24L01 radio driver
MyHwATMega328 hw; // Select AtMega328 hardware profile
#ifdef MY_SECURE_NODE_WHITELISTING
// Change the soft_serial value to an arbitrary value for proper security
uint8_t soft_serial[SHA204_SERIAL_SZ] = {0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01}
↪;
whitelist_entry_t node_whitelist[] = {
  { .nodeId = 55, // Just some value, this need to be changed to the NodeId of the␣
↪trusted node
    .serial = {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09} } // This need to change␣
↪to the serial of the trusted node
};
MySigningAtsha204Soft signer(true, 1, node_whitelist, soft_serial);  // Select␣
↪ATSHA204A software signing backend with one entry in the whitelist and our unique␣
↪serial
#else
MySigningAtsha204 signer;  // Select ATSHA204A software signing backend
#endif
MySensor gw(radio, hw, signer);
```

For a node that should transmit whitelisted messages but not receive whitelisted messages, you can simply skip the whitelist arguments (1 and node_whitelist above). For the “soft” backend, you can set these to 0 and NULL since you then need to provide the soft_serial buffer.

It is important to emphasize that you do not have to provide a whitelist that has entries for all nodes that transmit signed messages to the node in question. You only need to have entries for the nodes that in turn have enabled `MY_SECURE_NODE_WHITELISTING`. Nodes that does not have this option enabled can still transmit “regular” signed messages as long as they do not match a NodeId in the receivers whitelist.

7.6 The technical stuff

How are the messages actually affected by the signing?

The following illustration shows what part of the message is signed, and where the signature is stored:

The first byte of the header is not covered by the signature, because in the network, this byte is used to track hops in the network and therefore might change if the message is passing a relay node. So it cannot be part of the signature, or the signature would be invalid when it arrives to its destination. The signature also carries a byte with a signing identifier to prevent false results from accidental mixing of incompatible signing backends in the network. Thus, the maximum size for a payload is 29-7 bytes. Larger payloads are not possible to sign. Another thing to consider is that the strength of the signature is inversely proportional to the payload size.

As for the ATSHA204SOFT backend, it turns out that the ATSHA does not do “vanilla” HMAC processing. Fortunately, Atmel has documented exactly how the circuit processes the data and hashes thus making it possible to generate signatures that are identical to signatures generated by the circuit.

The signatures are calculates in the following way:

Exactly how this is done can be reviewd in the source for the ATSHA204SOFT backend and the ATSHA204A datasheet.

In the MySensors protocol, the following new internal messagetypes has been added for handling signature requirements and nonce requests:

```
I_REQUEST_SIGNING
I_GET_NONCE
I_GET_NONCE_RESPONSE
```

Also, the version field in the header has been reduced from 3 to 2 bits in order to fit a single bit to indicate that a message is signed.

7.7 Known limitations

It is very important to emphasize that with the current implementation of message signing, OTA firmware updates are transmitted unsigned. In other words, it is technically possible for Mallory to reset a node by cutting power or some other attack, spoof a gateway and push his/her own custom firmware to the node even if the old sketch was requiring signing. The current architecture of the OTA solution prevents signing support from being implemented in the bootloader due to size constraints.

It is still possible to use OTA bootloader with a signing gateway and node, but it is important to understand this potentially provides an attack vector for compromising the security of the node.

Also, due to the limiting factor our our Arduino nodes, the use of diversified keys is not implemented. That mean that all nodes in your network share the same PSK (at least the ones that are supposed to exchange signed data). It is important to understand the implications of this, and that is hopefully covered in the “Typical usecases” chapter below.

Also be reminded that the strength of the signature is inversely proportional to the size of the message. The larger the message, the weaker the signature.

7.8 Typical usecases

“Securely located” in this context mean a node which is not physically publicly accessible. Typically at least your gateway.

“Public” in this context mean a node that is located outside your “trusted environment”. This includes sensors located outdoors, keyfobs etc.

7.8.1 Securely located lock

You have a securely located gateway and a lock somewhere inside your “trusted environment” (e.g. inside your house door, the door to your dungeon or similar).

You should then keep the data section of your gateway and your lock node unlocked. Locking the data (and therefore the PSK) will require you to replace at least the signing circuit in case you need to revoke the PSK because some other node in your network gets compromised.

7.8.2 Patio motion sensor

Your gateway is securely located inside your house, but your motion sensor is located outside your house. You have for some reason elected that this node should sign the messages it send to your gateway.You should lock the data (PSK) in this node then, because if someone were to steal your patio motion sensor, they could rewrite the firmware and spoof your gateway to use it to transmit a correctly signed message to your secure lock inside your house. But if you revoke your gateway (and lock) PSK the outside sensor cannot be used for this anymore. Nor can it be changed in order to do it in the future. You can also use whitelisting to revoke your lost node. This is an unlikely usecase because it is really no reason to sign sensor values. If you for some reason want to obfuscate sensor data, encryption is a better alternative.

7.8.3 Keyfob for garage door opener

Perhaps the most typical usecase for signed messages. Your keyfob should be totally locked down. If the garage door opener is secured (and it should be) it can be unlocked. That way, if you loose your keyfob, you can revoke the PSK in both the opener and your gateway, thus rendering the keyfob useless without having to replace your nodes. You can also use whitelisting to revoke your lost keyfob.

CHAPTER 8

Api

In this page you will find about the functions you must implement and the objects you may use on your sketches.

8.1 Functions

8.1.1 presentation

```
void presentation()
```

This function must be implemented (at least present) in your sketch. It will be called at the start of your node and every time your nome receive an I_PRESENTATION.

@params: None

@return: None

8.1.2 receive

```
void receive(const MyMessage &msg)
```

If you want to handle messages you must implement this function on your sketch. If you don't implement it, you won't be able to handle the messages and they will just be ignored, if they are not "special" messages which are handled behind the scenes.

Then, you will use the MyMessage object to see what is inside the message.

@params:

- msg: It takes a reference of a MyMessage that you will to manipulate.

@return: None

8.1.3 receiveTime

`void receiveTime(unsigned long time)`

This function will be called every time your node receives time from controller. So if you want to handle it, you must implement this functions.

@params:

- time:

@return: None

8.1.4 getConfig

`ControllerConfig getConfig()`

It takes the most recent node configuration received from controller

@params: None

@return: ControllerConfig struct with all configuration

8.1.5 getNodeId

`uint8_t getNodeId()`

@params: None

@return: Node's id

8.1.6 loadState

`uint8_t loadState(uint8_t pos)`

Load a state (from local EEPROM).

@params:

- pos: The position to fetch value from 0 to 255

@return: Value that was stored in position

8.1.7 present

```
void present(uint8_t sensorId, uint8_t sensorType, const char* description="",
bool ack=false)
```

Each node must present all attached sensors before any values can be handled correctly by the controller. It is usually good to present all attached sensors after power-up in setup().

@params:

- sensorId: Select a unique sensor id for this sensor. Choose a number between 0-254.
- sensorType: The sensor type. See sensor typedef in MyMessage.h.
- description: A textual description of the sensor.
- ack: Set this to true if you want destination node to send ack back to this node. Default is not to request any ack.

- description: A textual description of the sensor.

@return: None

8.1.8 request

```
void request(uint8_t childSensorId, uint8_t variableType, uint8_t
destination=GATEWAY_ADDRESS)
```

It sends an requesting package.

@params:

- childSensorId: The variable's node's id you want to request.
- variableType: The type of variable you are requesting.
- destination: Destination node. The default destination is the Gateway.

@return: None

8.1.9 requestTime

```
void requestTime()
```

Requests time from controller. Answer will be delivered to receiveTime function in sketch.

@params: None

@return: None

8.1.10 saveState

```
void saveState(uint8_t pos, uint8_t value)
```

Save a state (in local EEPROM). Good for actuators to "remember" state between power cycles. You have 256 bytes to play with. Note that there is a limitation on the number of writes the EEPROM can handle (~100 000 cycles on ATMega328).

@params:

- pos: The position to store value in 0 to 255
- value: Value to store in position

@return: None

8.1.11 send

```
bool send(MyMessage &msg, bool ack=false)
```

Sends a message to gateway or one of the other nodes in the radio network

@params:

- msg: It takes a reference to a Message object to send.
- ack: Set this to true if you want destination node to send ack back to this node. Default is not to request any ack.

@return: Returns true if message reached the first stop on its way to destination.

8.1.12 sendBatteryLevel

```
void sendBatteryLevel(uint8_t level, bool ack=false)
```

Send this nodes battery level to gateway.

@params:

- level: Level between 0-100(%)
- ack: Set this to true if you want destination node to send ack back to this node. Default is not to request any ack.

@return: None

8.1.13 sendHeartbeat

```
void sendHeartbeat()
```

Send a heartbeat message (I'm alive!) to the gateway/controller. The payload will be an incremental 16 bit integer value starting at 1 when sensor is powered on.

Allows node to send heartbeat and controller to ping nodes.

@params: None

@return: None

8.1.14 sendSketchInfo

```
void sendSketchInfo(const char* name, const char* version, bool ack=false)
```

It sends sketch meta information to the gateway. Not mandatory but a nice thing to do.

@params:

- name String containing a short Sketch name or NULL if not applicable
- version String containing a short Sketch version or NULL if not applicable
- ack Set this to true if you want destination node to send ack back to this node. Default is not to request any ack.

@return: None

8.1.15 sleep

```
void sleep(unsigned long ms)
```

Sleep (PowerDownMode) the MCU and radio. Wake up on timer.

@params:

- ms: Number of milliseconds to sleep.

@return: None

8.1.16 smartSleep

```
void smartSleep(unsigned long ms)
```

@params:

- ms: Number of milliseconds to sleep.

8.1.17 sleep

```
bool sleep(uint8_t interrupt, uint8_t mode, unsigned long ms=0)
```

Sleep (PowerDownMode) the MCU and radio. Wake up on timer or pin change. See: http://arduino.cc/en/Reference/attachInterrupt for details on modes and which pin is assigned to what interrupt. On Nano/Pro Mini: 0=Pin2, 1=Pin3

@params:

- interrupt: Pin that should trigger the wakeup
- mode: RISING, FALLING, CHANGE
- ms: Number of milliseconds to sleep or 0 to sleep forever

@return: True if wake up was triggered by pin change and false means timer woke it up.

8.1.18 smartSleep

```
bool smartSleep(uint8_t interrupt, uint8_t mode, unsigned long ms=0)
```

@params:

- interrupt: Pin that should trigger the wakeup
- mode: RISING, FALLING, CHANGE
- ms: Number of milliseconds to sleep or 0 to sleep forever

@return: True if wake up was triggered by pin change and false means timer woke it up.

8.1.19 sleep

```
int8_t sleep(uint8_t interrupt1, uint8_t mode1, uint8_t interrupt2, uint8_t
mode2, unsigned long ms=0)
```

Sleep (PowerDownMode) the MCU and radio. Wake up on timer or pin change for two separate interrupts. See: http://arduino.cc/en/Reference/attachInterrupt for details on modes and which pin is assigned to what interrupt. On Nano/Pro Mini: 0=Pin2, 1=Pin3

@params:

- interrupt1 First interrupt that should trigger the wakeup
- mode1 Mode for first interrupt (RISING, FALLING, CHANGE)
- interrupt2 Second interrupt that should trigger the wakeup
- mode2 Mode for second interrupt (RISING, FALLING, CHANGE)
- ms Number of milliseconds to sleep or 0 to sleep forever

@return: Pin number wake up was triggered by pin change and negative if timer woke it up.

8.1.20 smartSleep

`int8_t smartSleep(uint8_t interrupt1, uint8_t mode1, uint8_t interrupt2, uint8_t mode2, unsigned long ms=0)`

@params:

- interrupt1 First interrupt that should trigger the wakeup
- mode1 Mode for first interrupt (RISING, FALLING, CHANGE)
- interrupt2 Second interrupt that should trigger the wakeup
- mode2 Mode for second interrupt (RISING, FALLING, CHANGE)
- ms Number of milliseconds to sleep or 0 to sleep forever

@return: Pin number wake up was triggered by pin change and negative if timer woke it up.

8.1.21 wait

`void wait(unsigned long ms)`

Wait for a specified amount of time to pass. Keeps process()ing. This does not power-down the radio nor the Arduino. Because this calls process() in a loop, it is a good way to wait in your loop() on a repeater node or sensor that listens to messages.

@params:

- ms: Number of milliseconds to sleep.

@return: None

8.2 Objects

8.2.1 MyMessage

`MyMessage msg1(CHILD_ID, CHILD_TYPE);` `MyMessage msg2();`

This object will handle incoming and outcoming messages. You must create one message for each sensor you have in your node.

Attributes

`uint8_t last`

8 bit - Id of last node this message passed

`uint8_t sender`

8 bit - Id of sender node (origin)

`uint8_t destination`

8 bit - Id of destination node

`uint8_t version_length`

2 bit - Protocol version

1 bit - Signed flag

5 bit - Length of payload

`uint8_t command_ack_payload`

3 bit - Command type

1 bit - Request an ack - Indicator that receiver should send an ack back.

1 bit - Is ack messsage - Indicator that this is the actual ack message.

3 bit - Payload data type

`uint8_t type`

8 bit - Type varies depending on command

`uint8_t sensor`

8 bit - Id of sensor that this message concerns.

`char data[MAX_PAYLOAD + 1];`

That is the message's payload

Methods

getCommand

```
uint8_t getCommand()
```

@params: None

@return: It returns the value of command (type of message). E.g.: C_SET, C_REQ, ...

isAck

```
bool isAck()
```

@params: None

@return: It return if it is an ack or not.

set

```
MyMessage& set(void* payload, uint8_t length)

MyMessage& set(const char* value)

MyMessage& set(float value, uint8_t decimals)

MyMessage& set(uint8_t value)

MyMessage& set(uint32_t value)

MyMessage& set(int32_t value)

MyMessage& set(uint16_t value)

MyMessage& set(int16_t value)
```

This function set the message's payload. There are a lot of overwritten in this function, which allow you to send many types of value.

@params:

- payload:
- length:
- value:
- decimals:

@return: It returns a reference to your MyMessage object.

setDestination

```
MyMessage& setDestination(uint8_t destination)
```

It sets the destination of the package.

@params:

- destination

@return: It returns a reference to your MyMessage object.

setSensor

```
MyMessage& setSensor(uint8_t sensor)
```

It sets the sensor's id which will be send on the protocol header.

@params:

- sensor

@return: It returns a reference to your MyMessage object.

setType

```
MyMessage& setType(uint8_t type)
```

It sets the sensor's type which will be send on the protocol header.

@params:

- type

@return: It returns a reference to your MyMessage object.

What's new:

- Library size reduced ~20% by removing a lot of C++ overhead.
- Full configuration of library behaviours and features directly from sketch code.
- No more MySensors constructor mess with radio drivers and signing backends. All setup and initialization is handled "behind-the-scene".
- Calls to process() is handled in the background automatically.

- Structural change: Embedding of drivers and libraries in a structured way instead of using the weird arduino-build-system util-folder which includes everything when building.
- All gateway variants available as examples without any need for re-configuration (E.g. SOFT-SPI automatically enabled for W5100).
- Gateway just another sensor node! So now you can have wired ethernet sensors and wireless ESP8266 sensors without any radio attach if you want.
- AES encryption (RF24)
- Introducing a presentation() function in sketch - This allows controller to re-request presentation and do re-configuring node after startup.
- Optional receive() function in sketch replaces begin(callback).
- sendHeartbeat() - Allows node to send heartbeat and controller to ping nodes.
- Ethernet/ESP8266 gateway supports setting static ip and using dhcp.
- Ethernet/ESP8266 gateway allowing communication using UDP.
- Ethernet/ESP8266 gateway can now act as a client which opens a socket to controller at startup. NOTE: This has to be supported by controller.
- MQTT client gateway on ESP8266 or Arduino+W5100 Ethernet adapter.
- Serial transport layer (RS232/RS485) with dePin management.
- Added variable V_CUSTOM - For sending/requesting custom data to/from controller and between nodes. Preferable using S_CUSTOM device. This variable-type is controller specific so use it with care in the officially provided examples.
- New sensors types and variables: S_INFO, S_GAS, S_GPS, V_TEXT, V_CUSTOM, V_POSITION.
- New command I_DISCOVER - retrieve active nodes and topology.
- Detection of mis-wired RFM69 radios.
- MyMessage:getCommand() (get message type, e.g.: internal, stream, set, ...)
- “Smart sleep” variants of all sleep methods that allows nodes to receive buffered messages/commands from controller that was issued while node was sleeping. When calling the smartSleep() variant, the node immediately sends a heartbeat mesage when waking up (informing controller that node is awake). Before going back to sleep it waits MY_SMART_SLEEP_WAIT_DURATION to process any incoming buffered messages from the controller. NOTE: Controller must support this feature.