# Physical Layer Format

## Packet format

Packets are constructed out of bytes from at least 6 bytes up to 21 long, with the payload varying from 0 to 16 bytes.

The header contains target and source node ID's. Target 255 (FFh) is information for all nodes. In addition, a message ID flag, payload size and *options* flag field exist. Packets are finalized with a CRC check.

The format of a packet is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Range | Description |
| Size | Uint8\_t | 5..21 | Length of header & payload, excluding CRC. |
| Source node | Uint8\_t | 0..254 | Node ID which packet originates from |
| Destination Node | Uint8\_t | 0..255 | Node ID targeted. 255 is broadcast |
| Message Type | NodeMsgType\_t | See enumeration (chapter 2.5) | Type of packet. |
| Options | Uint8\_t | N/A | Additional options. Reserved for future. |
| Payload | Uint8\_t[] | 0..16 length | *(Optional)* Data of the packet. |
| CRC | Uint8\_t | See CRC  (chapter 1.2) | Checksum to validate integrity of transmission of the packet. |

Options packet:

|  |  |  |
| --- | --- | --- |
| Bit | 0 | 1 |
| Data | Keep Alive |  |
| Type | bool |  |

Keep alive describes whether the node should go to rest. If set to 1 in any packet; the node should stay awake.

## CRC

The CRC is calculated from all bytes of the packet (Size up to up to CRC field). The CRC , as of revision 1 of the protocol, is calculated by a XOR chain of each data byte. The CRC initial value of 00h, and is computed as following:

The CRC is pending for improvement. At the current implementation this is nothing more than a checksum; where the order of bytes is not taken into account.

# Network Messages

## Node

A node is a client in the network which will receive and transmit messages on it's own unique node ID. Typical nodes are battery powered devices that can transmit information from sensors remotely.

Each node ID can only be used once. As such, up to 254 nodes can be held in the network, of which 1 must be a master that is in range of all nodes and manages the data streams.

All nodes will support the basic network protocol for firmware updates and time synchronization. All additional functionality, like sensor or actuator applications, are specific and described in a separate chapter.

## Types

In general, there are 2 types of messages defined:

* Unicast
* Broadcast

The packet can be identified as broadcast if it's destination node ID is 255 (FFh). All other messages are directed at specific nodes and by definition unicast.

It is not recommended to serve node ID's in the 0xF0-0xFE range, as it may be used in future revisions of the protocol for multicast support.

## Functionality

The scope of the basic network protocol is to provide the following basic functions to each node:

* Node power status  
  Nodes will broadcast battery information, online/offline status to the master.
* Time synchronization from master.   
  By using time acquired from a NTP server (on master node) plus software time base (1s ticker) it's possible for nodes to be aware of time even when shut down for long periods of time, and with the absence of a RTC.
* Firmware updates  
  The node will provide basic firmware update options.
* Security  
  Basic ciphering will be supported in future revisions (XXTEA, xor).

The network will be set-up without using any fancy mesh network (as of now). All nodes must be in reach of the master. Application specific details can be found in chapter 3.

## Time framing

Nodes must obey time framing and collision prevention rules to avoid receive buffer overflows and transmit buffer retransmits. Important messages contain an ACK type structure, where the remote client must send back a packet. If not done so, a retransmission of the packet is done up to 4 times.

To avoid other clients consuming the line before this ACK has been received, only once every 25ms a packet can be transmitted. Only packets in the 0xA0 through 0xAF do not experience this delay. This includes ping/pong (RTT test), acknowledges and miscellaneous 1-way traffic (which are not part of conversations and targeted to a master).

By avoiding more than 1 packet every 25ms and putting priority on these few packets, transmit buffers of nodes waiting for ACK should be cleared before other conversations can continue. These other conversations can be sensor data, actuator data, or bursts of firmware update data.

## Collision prevention

Packet collision prevention is implemented by polling the air status before transmit. If there is a signal present, transmission is delayed for another 10ms. Waiting is aborted after 100ms, and the packet is dropped.

How to test this?

## Firmware update

Nodes will support firmware update over the air. The node is a PIC16LF1508 controller, which has only got 4096 words of FLASH. While the RF stack takes ~1300 words, it's not possible to compile this stack a second time for the bootloader.

The firmware is therefore separated into a runtime and application part. The EEPROM is just big enough to contain the full firmware (4096 14b words = ~64kbit with 16-bit alignment). A flash loader application will reside in flash to update the runtime. Because a part of the firmware will be reserved for the application (i.e. 1024 words), the runtime will take , at peak, 48kbit of space. 16kbit of the 48kbit is shared with the application, which is flashed separately.

The firmware update messages will handle writing data to the EEPROM, requesting version information (like application name, runtime version, etc) and sending application commands.

16kbit of space would be enough to store up to 2kB of data, i.e. 512 x 2 x 16-bit raw sensor measurements. With a measurement taken every minute this will last for 8,5 hours. If 3/4 of the EEPROM is reserved for data (no runtime stored), there is space for 1 day of data.

Each firmware data packet is 8 bytes long. Up to 64kB FLASH is supported. This in turn means that, for programming 64kB of FLASH, it takes 8192 \* 2 (ACK) packets to complete. This should mean a FLASH update takes 3 minutes. As of now, the worst-case scenario is that the whole 64kbit EEPROM needs to be written, which takes 1024\*2 packets, which can take up to 20 seconds.

## List of Messages

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Name | Direction | Description |
| A0 | Ping/pong | Any | Echo packet for round-trip time measurements and a means of polling online/offline status or RSSI measurement. |
| A1 | Acknowledgement | Any | Acknowledgement for packets that require data to be delivered onto target. |
| A2 | Time synchronize | M > S | Master updates network of new timestamp. |
| A3 | Power Status | S > M | Unicast message to indicate power status of node, like online, uptime and battery status. |
| B0 | Configuration CMD | M > S  S > M | List configuration parameters Reload configuration Load defaults |
| B1 | Write Configuration | M > S | Write a configuration value to the node |
| B2 | Read Configuration | M > S  S > M | Read a configuration value from the node |
| C0a | Firmware Action Request | M > S | Request to node to read constants or undertake action. Like, flash runtime, flash application, calculate checksum, etc. |
| C0b | Firmware Action  Response | S > M | Response to C0a. |
| C1a | Firmware Data Write | M > S | Send new firmware |
| C1b | Firmware Data Verify | S > M | Verify data to firmware, after it was send & programmed |
| C2a | Firmware Data Read | M > S | Request to read data, implement only when Dx not implemented. |
| C2b | Firmware Data Read | S > M | Response to read data |
| D0 | EEPROM Read *(opt)* | M > S | Read data from node EEPROM |
| D1 | EEPROM Write *(opt)* | M > S | Write data to node EEPROM |
|  |  |  |  |
| FF | NOP | Any | No action taken. |

**Application specific:**

Temperature/humidity sensor node

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Name | Direction | Description |
| 00 | Sample | S > M | Sends 2 channels worth of sample data. |
|  |  |  |  |
|  |  |  |  |

### A0. Ping/pong

**ID:** A0

**Size:** 16 bytes

**Traffic direction:** Any (Master -> Slave / Slave -> Master)  
**Side effects:** none  
**Acknowledge:** No

**Packet Format:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Data | Type | Randomized Data | | | | | | | | | | | | | | |
| Type | u8 | u8 | | | | | | | | | | | | | | |

1. **Type**: Value 1 is Request. Value 2 is Response. Any other is invalid and should be dropped.
2. **Data:** Bytes 1 through 15 are randomized data which must be echo'ed in a response packet.

Node must reply to packet when type is 1 ASAP. This measures the RTT .

### A1. Acknowledge

**ID:** A1

**Size:** 2 bytes

**Traffic direction:** Any (Master -> Slave / Slave -> Master)  
**Side effects:** acknowledges reception of packet on client  
**Acknowledge:** No

**Packet Format:**

|  |  |  |
| --- | --- | --- |
| Byte | 0 | 1 |
| Data | ID | CRC |
| Type | u8 | u8 |

1. **ID**: Original packet ID being acknowledged.
2. **CRC:** CRC of packet being acknowledged.

It's important this ACK is send ASAP back, as the TX buffer of the remote client is held busy until this packet has been received. Please read the time framing paragraph (2.4).

### A2. Time synchronization

**ID:** A2

**Size:** 2 bytes

**Traffic direction:** Master -> Slave   
**Side effects:** sets high-word time stamp in node

**Packet Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Byte | 0 | 1 | 2 | 3 |
| Data | Time | | | |
| Type | U32 | | | |

1. **Time:** seconds since 1970

### A3. Power Status

**ID:** A3

**Size:** 8 bytes

**Traffic direction:** Slave -> Master  
**Side effects:** power status of node  
**Acknowledge:** Yes

**Packet Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Data | Online |  | Battery indicator | | Uptime | | | |
| Type | u8 |  | U16 | | U32 | | | |

1. **Online:** indicates if node was reset since last power message
2. Reserved
3. **Battery indicator:** current battery value. Measures internal voltage reference (2.048V) with VCC as reference. This reference only works with >2.5V VCC
4. **Uptime:** current uptime (seconds).

### C0a. Firmware Request

**ID:** C0a

**Size:** 1 to 9 bytes

**Traffic direction:** Master -> Slave  
**Side effects:** request constants about firmware and application of node  
**Acknowledge:** No

**Packet Format:**

|  |  |  |
| --- | --- | --- |
| Byte | 0 | 1..8 |
| Data | Action | Data |
| Type | U8 | U8 |

1. **Request:** request ID to request
2. **Data:** see request table for format

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Name | Format | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 00 | NOP |  |  |  |  |  |  |  |  |
| 01 | Read Firmware ID |  |  |  |  |  |  |  |  |
| 02 | Read EEPROM State |  |  |  |  |  |  |  |  |
| 03 | Prepare EEPROM | State |  |  |  |  |  |  |  |
| 04 | Flash Runtime |  |  |  |  |  |  |  |  |
| 05 | Flash Application |  |  |  |  |  |  |  |  |
| 06 | Boot Application |  |  |  |  |  |  |  |  |
| 07 | Application Run State |  |  |  |  |  |  |  |  |

### C0b. Firmware Response

**ID:** C0b

**Size:** 1 to 9 bytes

**Traffic direction:** Slave -> Master  
**Side effects:** response to C0 message  
**Acknowledge:** Yes

**Packet Format:**

**Response**

|  |  |  |
| --- | --- | --- |
| Byte | 0 | 1..8 |
| Data | Action | Data |
| Type | U8 | U8 |

1. **Response:** response ID to request
2. **Data:** see request table for format

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Name | Format | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 00 | NOP |  |  |  |  |  |  |  |  |
| 01 | Firmware ID Report | Runtime Major | Runtime Minor | App Major | App Minor |  |  |  |  |
| 02 | EEPROM State | State |  |  |  |  |  |  |  |
| 03 | Prepare EEPROM |  |  |  |  |  |  |  |  |
| 04 | Flash Runtime |  |  |  |  |  |  |  |  |
| 05 | Flash Application |  |  |  |  |  |  |  |  |
| 06 | Boot Application |  |  |  |  |  |  |  |  |
| 07 | Application State | Flags |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

#02. State

* 0 = Application Data
* 1 = Application Loader
* 2 = Runtime Loader

#07. Flags:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Field | Flashed | Running | Fault |  |  |  |  |  |
| States | 1=flashed  0=empty | 1=running 0=idle | 1=fault  0=none |  |  |  |  |  |

### C1a Firmware Data Write

**ID:** C1a

**Size:** 16 bytes

**Traffic direction:** Master -> Slave

**Side effects:** writes data to the destination EEPROM  
**Acknowledge:** No

**Packet Format:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Data | Address | | | |  |  |  |  | Data | | | | | | | |
| Type | U32 | | | |  |  |  |  | U8 | | | | | | | |

The packet allows programming chips up to 64kB of FLASH. 8 bytes of data is programmed per packet.

Fields 4-7 are reserved, and could be used to have individual encryption and decryption keys per piece of data. This data is also applied to the address field.

1. **Address:** raw address that needs to be written to EEPROM/program memory. Only bytes 2-3 are used.
2. Reserved
3. **Data: 8** bytes of program data

### C1b Firmware Data Verify

**ID:** C1b

**Size:** 10 bytes

**Traffic direction:** Slave -> Master

**Side effects:** calculates checksum of destination EEPROM  
**Acknowledge:** Yes

**Packet Format:**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Data | Address | | | |  |  |  |  | Checksum | |
| Type | U32 | | | |  |  |  |  | U16 | |

This packet contains the checksum for **8** bytes of data of the EEPROM. It's automatically sent, next to an ACK, after C1a was received.

The reserved section can be used to generate a encryption key, based on the data (for example).

1. **Address:** raw address that was written. Only bytes 2-3 are used.
2. Reserved
3. **Checksum:** checksum of address that was programmed

The checksum is calculated according to XOR of 16-bit words:

### C2a Firmware Data Read

**ID:** C2a

**Size:** 4 bytes

**Traffic direction:** Master -> Slave

**Side effects:** read data of EEPROM/firmware  
**Acknowledge:** No

**Packet Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Byte | 0 | 1 | 2 | 3 |
| Data | Address | | | |
| Type | U32 | | | |

The packet requests to read data from the specified address. The device will always read 8 bytes. Only bytes 2-3 are used.

### C2b Firmware Data Read

**ID:** C2b

**Size:** 16 bytes

**Traffic direction:** Slave -> Master

**Side effects:** data read from the EEPROM  
**Acknowledge:** Yes

**Packet Format:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Data | Address | | | |  |  |  |  | Data | | | | | | | |
| Type | U32 | | | |  |  |  |  | U8 | | | | | | | |

This packet responds to C2a, which reads data from the device EEPROM/firmware image.

The reserved section can be used to generate a encryption key, based on the data (for example).

1. **Address:** raw address that needs to be written to EEPROM/program memory. Only bytes 2-3 are used.
2. Reserved
3. **Data: 8** bytes of program data

### 00. Sample data

**ID:** 00

**Size:** 6+ bytes

**Traffic direction:** Slave > Master

**Side effects:** will store data in master  
**Acknowledge:** Yes

**Packet Format:**

|  |  |  |
| --- | --- | --- |
| Byte | 0..3 | 4..15 |
| Data | Timestamp | Data |
| Type | U32 | Channel[] |

**Channel format:**

|  |  |  |
| --- | --- | --- |
| Byte | 0 | 1..5 |
| Data | ID | Value |
| Type | U8 | U8/U16/U32 |

The channel format repeats itself. For 2 channels, of a given type ID, the values will append each other. The type ID will determine the value width.

1. **Timestamp:** time of sample moment
2. Data: channels sampled

For example: the format of a weather node

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Byte | 0..3 | 4 | 5..6 | 7 | 8..9 |
| Data | Timestamp | Temp ID | Temperature | Humidity ID | Humidity |
| Type | U32 | U8 | U16 | U8 | U16 |