OpenThings Protocol

Universal Messaging Structure for Connected Devices

INTRODUCTION

Connected devices and home automation are young, fast growing markets served by a wide range of manufacturers and service providers, all over the world. For many buyers, the lack of product interoperability is an impediment to take-up. For many manufacturers, the lack of a suitable communications protocol adds to their product implementation costs.

The OpenThings protocol was originated by Sentec to address these problems. It defines message structures and essential information exchange sequences to enable manufacturers to collaborate and build interoperable products. It reduces implementation costs by providing a flexible, small footprint protocol, with helpful notes, that can be implemented quickly and easily into new products. It is also extensible, so that it can be adapted for new product types in the future.

Sentec has made OpenThings publicly available and wishes only to control one element that will benefit from central administration (the allocation of Manufacturer IDs). Sentec will not charge any fees, royalty, subscription or any other payment for the use of OpenThings.

TERMS OF USE

Sentec permits both commercial use and alteration to the protocol. However, the protocol is provided "as is" and users are solely responsible for their use of it. Sentec makes no representations about the protocol's quality or suitability for any purpose and disclaims all warranties and conditions relating to it. It may change the protocol at any time.

THE PROTOCOL

OpenThings is a lightweight messaging protocol for sending reports (e.g. temperature measurement, power reading) and commands (e.g. turn on socket, set dimmer level) between small sensors and connected devices. The protocol is intended for use in simple applications with point-to-point or star network topologies. In both cases one device is a master and remaining devices are slaves.

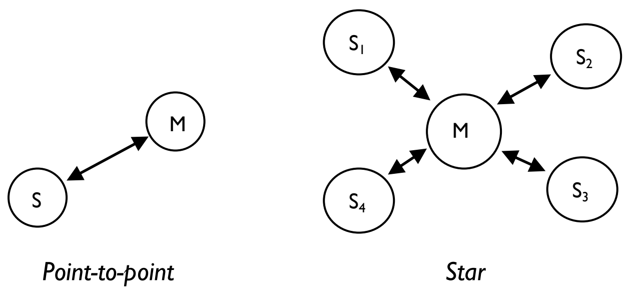

Figure 1: Network topologies

The protocol provides basic message structuring and data validation (CRC). Messages are transmitted in one direction at a time without acknowledgement. The protocol does not contain any encryption, quality of service, anti-collision or network routing elements. These can be implemented in additional network layers if required; the goal of this protocol is only to define a universal message structure and framework for how data is described, represented and sent. The protocol is hardware agnostic, it can be used with any suitable transport means; for example a 433MHz radio or serial bus connection.

TRANSMISSIONS

Each transmission has a defined messaging structure containing one or more records. Each record contains a parameter identifier (e.g. temperature, light level, voltage) and type description (e.g. integer, float, character) of the data value to be sent. In this way the protocol is extensible and allows new categories of data to be described and sent without any changes to the underlying protocol.

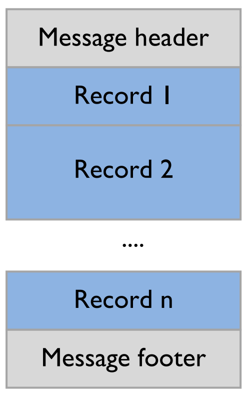

Figure 2: Example transmission

ADDRESSING

In OpenThings network topologies only slave devices have addresses. An address is the combination of the Manufacturer ID, Product ID and Sensor ID and must be unique. Slave devices will only act on messages sent containing their address and only originate messages with their address. Master devices listen to all messages sent on the network (other than messages they have sent themselves) and originate messages with the address of the slave they want to send the message to. The protocol does not allow broadcasting; all messages must be individually addressed.

This example shows the exchange of messages and addressing when a slave device reports a parameter:

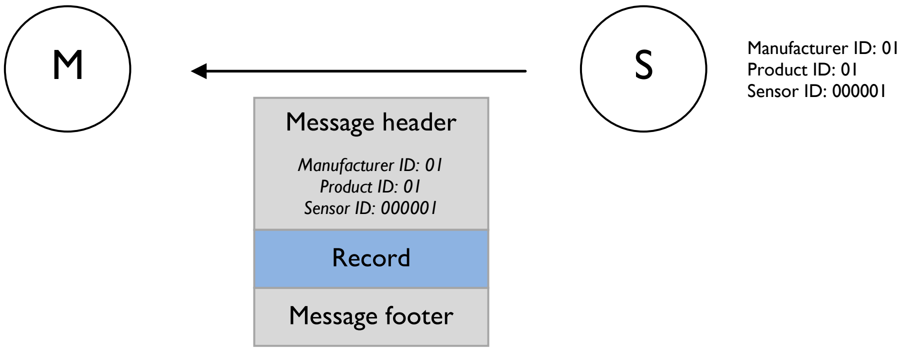

Figure 3: Report parameter example

This example shows the exchange of messages and addressing when a master commands a parameter on a slave device and the slave device is expected to respond.

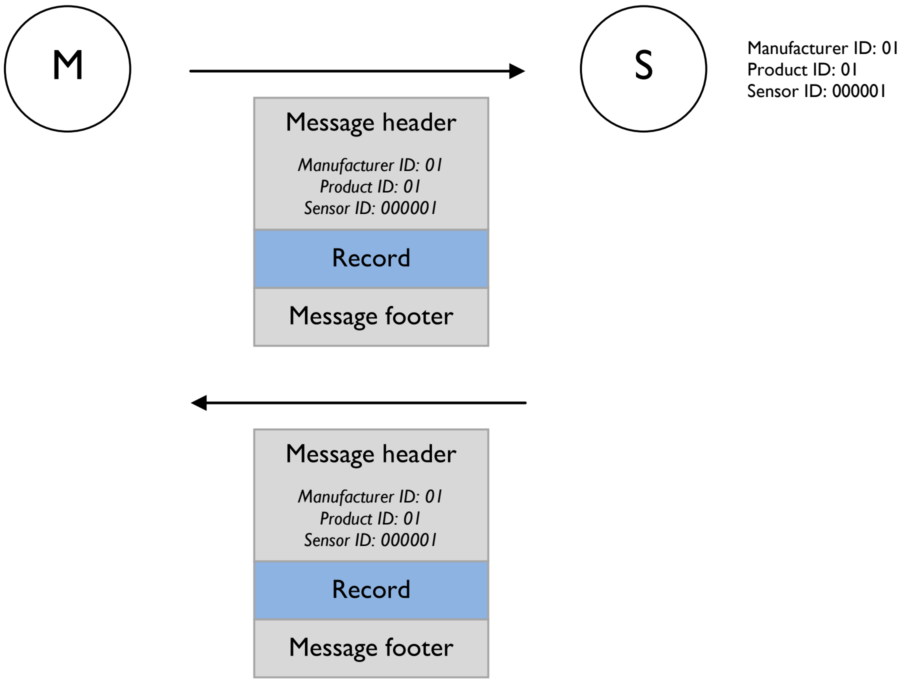

Figure 4: Command parameter example

DEFAULT DICTIONARY OF PARAMETER IDENTIFIERS

Table 1 below lists the parameters that can be reported and commanded as part of the default parameter identifier dictionary. Manufacturers may extend (but not modify) this to meet their requirements.

Parameter	**Char**	**Hex (Report)**	**Units**
Alarm	!	0x21	*See "Alarms"*
Debug Output	-	0x2D	*Reserved for debug*
Identify	?	0x3F	*See "Identify"*
Source Selector	@		*See "Multiple Commands"*
Water (Flood) Detector	A	0x41	1 = flooded, 0 = dry. *See "Binary Parameters"*
Glass Breakage	B	0x42	1 = broken, 0 = intact. *See "Binary Parameters"*
Closures	C	0x43	Curtains/blinds; 1 = open, 0 = closed. See *"Binary Parameters"*
Door Bell	D	0x44	1 = pressed, 0 = released. See *"Binary Parameters"*
Energy	E	0x45	kWh
Fall Sensor	F	0x46	1 = fall, 0 = no fall. *See "Binary Parameters"*
Gas Volume	G	0x47	m^3
Air Pressure	H	0x48	mbar (millibar)
Illuminance	I	0x49	Lux. *See "Illuminance/Light Level"*
Level	L	0x4C	*See "Generic Level"*
Rainfall	M	0x4D	mm
Apparent power	P	0x50	VA
Power factor	Q	0x51	
Report Period	R	0x52	s (seconds). See *"Sensor Report Frequency"*
Smoke Detector	S	0x53	1 = smoke detected, 0 = no smoke. *See "Binary Parameters"*
Time and Date	T	0x54	Seconds since Epoch (1 Jan 1970)
Vibration	V	0x56	1 = vibration detected, 0 = no vibration. *See "Binary Parameters"*
Water Volume	W	0x57	l (litres)
Wind Speed	X	0x58	m/s
Gas Pressure	a	0x61	Pa
Battery Level	b	0x62	V
CO Detector	c	0x63	1 = gas detected, 0 = no gas detected. *See "Binary Parameters"*
Door Sensor	d	0x64	1 = open, 0 = closed. *See "Binary Parameters"*
Emergency (Panic Button)	e	0x65	1 = emergency, 0 = no emergency. *See "Binary Parameters"*
Frequency	f	0x66	Hz
Gas Flow Rate	g	0x67	m^3/hr
Relative Humidity	h	0x68	%
Current	i	0x69	A
Join	j	0x6A	No data. See *"Joining"*
RF quality	k	0x6B	Received radio signal strength level
Light Level	l	0x6C	Unsigned int, 0 = off, max = full on. *See "Illuminance/Light Level"*

Motion Detector	m	0x6D	1 = motion, 0 = no motion. *See "Binary Parameters"*
Occupancy	o	0x6F	1 = room occupied, 0 = room not occupied. See "*Binary Parameters*"
Real Power	p	0x70	W
Reactive Power	q	0x71	VAR
Rotation Speed	r	0x72	RPM
Switch State	s	0x73	1 = on, 0 = off. *See "Binary Parameters"*
Temperature	t	0x74	Celsius
Voltage	v	0x76	V
Water Flow Rate	w	0x77	l/hr (litres/hour)
Water Pressure	x	0x78	Pa
Phase 1 Power	y	0x79	W
Phase 2 Power	z	0x7A	W
Phase 3 Power	{	0x7B	W
3 phase total Power	|	0x7C	W

Table 1: Default dictionary

Devices are not required to support all parameters and the dictionary does not in general define the type of parameters (See Table 10); whether Energy is reported as a 32-bit unsigned integer, 48.16 bit fixed point or even as an unhelpful character string is up to the manufacturer, as long as it is reported in kWh.

Some parameters can also be set via a command from a master device or very occasionally by a slave. A command can be distinguished from a report by setting the top bit of the parameter identifier (i.e. adding 0x80 to the hex number). Thus a smart plug might report that its switch was off with parameter identifier report "0x73", and a gateway might command it to turn the switch on with the parameter identifier command "0xF3".

Report and Request

Parameters can either be requested or reported from a device. To report a parameter the device sends the type descriptor, parameter identifier and data. For a request of a parameter the length of data is zero and all that is sent is a parameter identifier.

To set a value of a certain parameter on a device the master sends the parameter identifier value + 0x80.

Example report, request and command messages

In these examples, for clarity, the message headers (Manufacturer ID, Product ID and sensor ID) and footers (null byte and CRC) are not presented.

Byte sequence	Value	Note
1	0x76	Parameter identifier "v", Voltage.
2	0x01	Unsigned normal integer, data length 1
3	0xFF	Data

Table 2: Node triggered Report from Node to Gateway

Byte sequence	Value	Note
1	0x76	Parameter identifier "v", Voltage.
2	0x00	Unsigned normal integer, data length 0. Length of 0 indicates that this is a request to the node.

Table 3: Gateway request to node

Byte sequence	Value	Note
1	0xF4	Parameter identifier "t", Temperature +0x80 indicates that this is a command.
2	0x01	Unsigned normal integer, data length 1
3	0xFF	Data

Table 4: Command from Gateway to set parameter on Node

Joining

The default joining protocol is simply a device announcement. The slave device sends a Join command (parameter identifier "j" command, 0xEA) and waits for a listening gateway to send it a Join report (parameter identifier 'j' report, 0x6A). Neither join record carries any data, so should have a type of 0x00 (See Table 10).

Keep-Alive

The default parameter identifier dictionary does not define any heartbeat or keep-alive record identifier. In the interests of brevity, devices wishing to send a keep-alive message that have nothing else to send can simply send a message with no records in it; in other words, they just send bytes 0 to 7 and x to x+2 from Table 8.

Identify

The "Identify" parameter (parameter identifier "?" report, 0x3F) may be used to report version numbers or model names. More usefully, as a command (0xBF) it commands the receiver to identify itself, by an out of band method, for example by flashing an LED or beeping for a few seconds.

Sensor Report Frequency

Some sensors make periodic reports of their data. The "Report Period" parameter (parameter identifier "R") allows the device to announce how often these reports will be made, and may allow a master device to command the time between reports if the slave device allows. By convention, a report period of zero indicates that a sensor will or should report parameter values when they change rather than at fixed intervals.

Multiple Reports

Some devices may have multiple sources of information. Consider monitoring a four-gang socket, for example; the device would need to report the voltage, current and power for each gang. This is done by sending multiple records with the same identifier in the same message. For example, a device would send a message containing a 'p' record containing the power on gang 1, another 'p' record containing the power on gang 2, and so on.

Multiple Commands

If a slave device has more than one commandable parameter, a master device may need to command only one parameter out of a set; for example to set the light level of one dimmer switch of a pair, or turn on one switch out of a multi-gang socket. To avoid any race conditions it must send a record containing the source selection parameter (parameter identifier "@" command) with a bit field of the indices of the parameter to command immediately before the part of the record setting the parameter.

For example to turn on the first (bit 0) and third (bit 2) switch of a multi-gang socket the master should send the following record (see Record Structure section for details of the elements):

Byte sequence	Value	Note
1	0xC0	Parameter identifier "@" command
2	0x01	Unsigned normal integer, data length 1
3	0x05	Bitfield, bits 0 and 2 set
4	0xF3	Parameter identifier "s" command
5	0x01	Unsigned normal integer, data length 1
6	0x01	Command switch on

Table 5: Example record to turn on first and third switch of a multi-gang socket

However to turn on the first switch and turn off the third switch a single source selection parameter cannot be used. Instead two source selection parameters commands must be sent in the same record:

Byte sequence	Value	Note
1	0xC0	Parameter identifier "@" command
2	0x01	Unsigned normal integer, data length 1
3	0x01	Bitfield, bit 0 set
4	0xF3	Parameter identifier "s" command
5	0x01	Unsigned normal integer, data length 1
6	0x01	Command switch on
7	0xC0	Parameter identifier "@" command
8	0x01	Unsigned normal integer, data length 1
9	0x04	Bitfield, bit 2 set
10	0xF3	Parameter identifier "s" command
11	0x01	Unsigned normal integer, data length 1
12	0x00	Command switch off

Table 6: Example record to turn on the first switch and turn off the third switch of a multi-gang socket

Illuminance/Light Level

"Illuminance" is intended for reports from light sensors, and gives the amount of illumination in Lux. "Light Level" is intended for reporting and controlling dimmer switch settings; its value is an unsigned integer where zero indicates that the switch is fully off, and the maximum value as defined by the type indicates that the switch is fully on.

Generic Level

"Generic Level" is intended for reporting and controlling generic devices; its value is an unsigned integer where zero indicates fully off, and the maximum value as defined by the type indicates fully on.

Alarms

The "Alarm" parameter (parameter identifier "!" report) is intended for conditions internal to the device that may need to be reported urgently. There is no obligation on a device to report these conditions. The parameter value is one or more characters defining the alarm conditions, as follows:

Condition	Char (set)	Hex (set)	Char (clear)	Hex (clear)	Comments
Low Battery	B	0x42	b	0x62	The device's battery is failing and must be replaced soon.
Power Fail	P	0x50	p	0x70	A mains-powered device is currently running off its battery backup.
Over Temperature	T	0x54	t	0x74	The device is running unexpectedly hot and needs attention.
Tamper	Z	0x5A	z	0x7A	The device's anti-tamper mechanism has been triggered.

Table 7: Default Alarm Values

The device may report that an alarm condition no longer applies by sending an alarm with the lower case version of the condition character in its parameter value. For example, a device whose ventilation has been fixed might send the record "0x21 0x71 0x74" to indicate that its temperature is back within normal operating parameters.

Alarms can be cancelled from a master device by sending an alarm parameter identifier "!" command.

Binary Parameters

A number of the parameters describe binary states, such as a switch being on or off. All these parameters are "active high", i.e. they report their active state ("the switch is on") with a value of 1 and their inactive condition with a value of 0.

Some slave devices may have multiple sources for the same binary parameter, such as individually controllable switches on a multi-gang socket. A slave may compress the different instances of the binary parameter into a single bitfield, with each bit reporting a different instance. So for example a strip of six switches could report that switches 0 and 3 were on and all the others off with a data value of 0x09 (binary 00001001). Binary parameters cannot be commanded using this shorthand; to avoid race conditions, master devices must use the "Source selection" parameter as described above.

MESSAGE STRUCTURE

Each transmission is sent according to the following message structure:

	Byte Num.	Data Name	Remark
Message header	0	Remaining length [7:0]	Number of bytes in whole message excluding this byte(x+2) – *see Note 1*
	1	Reserved bit [7]	Must be zero – see *Note 2*
		Manufacturer ID [6:0]	Manufacturer identifier – see *Note 3*
	2	Product ID [7:0]	Product identifier – see *Note 4*
	3	Reserved [15:8]	Reserved, default 0x0000 – see *Note 5*
	4	Reserved [7:0]	
	5	Sensor ID [23:16]	Unique Sensor ID to allow differentiation between end devices – see *Note 6*
	6	Sensor ID [15:8]	
	7	Sensor ID [7:0]	
Records	ˆ		
	Bytes 8 to x-1 contain one or more Records as shown in Table 9		
	ˇ		
Message footer	x	End of data [7:0]	NULL (0x00) to indicate end of data
	x+1	CRC [15:8]	CRC-16-CCITT – see *Note 7*
	x+2	CRC [7:0]	

Table 8: Message structure

The message header and footer are mandatory for each message transmission. Each message may contain one or more records, up to a maximum message length of 256 bytes. Bit numbers are shown in square brackets "[ ]".

Note 1: This byte can be used by the receiver to determine the length of the message to be received. Some radio chipsets with a packet handler can use this byte to automatically receive variable length messages.

Note 2: This reserved bit will always be zero in this version of the protocol.

Note 3: The Manufacturer ID is a globally unique number from 0 to 127 inclusive. It is allocated by Sentec on request and will be published on the OpenThings website.

Note 4: The Product ID should be used to identify the type/model of the product (e.g. smart plug, temperature sensor, PIR sensor etc). There is no reservation on Product IDs and manufacturers are expected to allocate their own; however, devices with Product IDs from 0 to 127 inclusive are expected to use a subset of the default parameter dictionary detailed above.

Note 5: These bytes are reserved for application/manufacturer specific functions. For example they could be used for an encryption seed. If not used the default is 0x0000.

Note 6: The Sensor ID must be unique to each end device of a given Product ID within the same radio network. To ensure this is the case it is recommended that the Sensor ID should be incremented (+1) sequentially in production for a given Product ID by the manufacturer.

Note 7: See Appendix A for more details on the cyclic redundancy check ("CRC") validation. The CRC is calculated from byte number 5 to byte number x inclusive, i.e. over the Sensor ID, the message body and the end of data marker. If the message is encrypted, the CRC should be performed over the unencrypted data.

RECORD STRUCTURE

Each record has the following structure:

Byte Num.	Data Name	Remark
1	Parameter identifier [7:0]	The identifier of the data being sent – see *Note 8*
2	Type description [7:0]	The type of the data being sent including its length z – see *Note 9*
ˆ		
Bytes 3 to 3+(z-1) contain the data value for the parameter		
ˇ		

Table 9: Record structure

Note 8: The parameter identifier indicates the physical measurement or control being communicated (e.g. power, frequency, switch state). Devices with Product IDs between 0 and 127 inclusive are expected to use parameter identifiers from the default parameter identifier dictionary, otherwise manufacturers are expected to define their own dictionary of parameter identifiers for each product.

The parameter identifier "j" (0xEA as a command, 0x6A as a report) has been reserved as a special parameter to allow slave devices to request to join/pair to a master device (typically a display or gateway) and must be included in each dictionary. If manufacturer products have a joining/pairing process it should be implemented around this command and response pair. The parameter identifier of value 0x00 is forbidden.

Note 9: The first 4 bits of a record type description byte define the type of the data value being sent (e.g. integer, float). The last 4 bits define the length of the data value being sent (in bytes):

7	6	5	4	3	2	1	0
Type				Length			

Table 10: Type description byte

7	6	5	4	Type	Decimal point
0	0	0	0	Unsigned integer	x.0 normal integer
0	0	0	1		x.4 fixed point integer
0	0	1	0		x.8
0	0	1	1		x.12
0	1	0	0		x.16
0	1	0	1		x.20
0	1	1	0		x.24
0	1	1	1	Characters	
1	0	0	0	Signed integer	x.0 normal integer
1	0	0	1		x.8 fixed point integer
1	0	1	0		x.16
1	0	1	1		x.24
1	1	0	0	Enumeration	Used for record enumeration, length byte defines number and this byte is followed by another type description byte, containing the type and length as normal.
1	1	0	1	Reserved	
1	1	1	0	Reserved	

1	1	1	1	Floating point	IEEE 754-2008

Table 11: Type definition within type description byte

It is entirely legal for any record to have a zero length data value, like the example "join" record below. Any type may be used, but a type description of 0x00 (unsigned integer) is conventional. Multi-byte numerical data values are sent big-endian, i.e. with the most significant byte first; see the example "power" record below.

Should a single node have several reports of the same type you can enumerate each report by using the type descriptor and the length as the enumeration value you then should follow this with a second report type and length of the payload.

Example records

Byte num.	7	6	5	4	3	2	1	0	Note
1	1	1	1	0	1	0	1	0	Parameter identifier "j" command
2	0	0	0	0	0	0	0	0	Unsigned normal integer, data length 0

Table 12: Example "join" command record

Byte num.	7	6	5	4	3	2	1	0	Note
1	0	1	1	1	0	0	0	0	Parameter identifier "p" report
2	1	0	0	0	0	0	1	0	Signed normal integer, data length 2 bytes
3	x	x	x	x	x	x	x	x	Power [15:8]
4	x	x	x	x	x	x	x	x	Power [7:0]

Table 13: Example "power" report record

Byte num.	7	6	5	4	3	2	1	0	Note
1	0	1	0	1	0	0	1	1	Parameter identifier "S" report
2	0	0	0	0	0	0	0	1	Unsigned normal integer, data length 1
3	0	0	0	0	0	0	0	x	State [0]: 1 = smoke detected, 0 = no smoke.

Table 14: Example "smoke detector" report record

Byte num.	7	6	5	4	3	2	1	0	Note
1	0	1	1	1	0	0	0	0	Parameter identifier "p" report
2	1	1	0	0	0	0	0	1	Enumerated report, report 1.
3	0	0	0	0	0	0	0	1	Unsigned normal integer, data length 1
4	x	x	x	x	x	x	x	x	Power report 1
2	1	1	0	0	0	0	1	0	Enumerated report , report 2
3	0	0	0	0	0	0	0	1	Unsigned normal integer, data length 1
4	x	x	x	x	x	x	x	x	Power report 2, end of enumeration

Table 15: Enumeration example

DOCUMENT REVISION

Revision	Comments
1.00	Initial release
1.01	Updated with: Enumeration, new dictionary entries, AES encryption, improved Joining procedure and authentication and improved examples of message structures.
1.02	Typographical style cleanups following review.

APPENDIX A – ADDITIONAL INFORMATION

CRC validation

The CRC can be used by the application layer to check the validity of a message. The CRC-16-CCITT checksum is implemented using the divisor polynomial of 0x1021 or ($x^{16} + x^{12} + x^5 + 1$). The `crc` function will calculate the 16-bit CRC for an array of message bytes using a code space efficient algorithm. Other algorithms are available optimised for speed but considering the low data rate, this algorithm is preferable.

```
int16_t crc(uint8_t const msg[], size_t size)
{
        uint16_t rem = 0;
        size_t   byte, bit;

        for (byte = 0; byte < size; ++byte)
        {
                rem ^= (msg[byte] << 8);
                for (bit = 8; bit > 0; --bit)
                {
                        rem = ((rem & (1 << 15)) ? ((rem << 1) ^ 0x1021) : (rem << 1));
                }
        }
        return rem;
}
```

Figure 5: CRC validation code extract

Example radio settings

These are the radio settings used as default by Sentec's MicroMonitor reference designs.

Transmission frequency	434.300 MHz
Deviation	± 30kHz
Radio FSK data rate	2400 baud Manchester (encoded at 4800 bits/s)
Message transmission interval	10s
Preamble	3 bytes (0xAA , 0xAA, 0xAA)
Sync bytes	2 bytes (0x2D , 0xD4)

Table 16: Radio physical settings

For radio applications, it is it is recommended that the total message length is less than 64 bytes.

Example product listing

This is an example product listing that a manufacturer may choose to share if they want people to develop other devices compatible with their product:

Manufacturer ID:	0x01 (Sentec)
Product ID:	0x13 (Temperature sensor)
Parameters:	t, j
Encryption:	None
Radio:	434.300 MHz ± 30kHz

APPENDIX B - LINEAR SHIFT ENCRYPTION

The OpenThings protocol includes provision for a lightweight 16-bit linear shift encryption algorithm. In this approach, the message includes all the information required to decrypt the message payload, without any knowledge of the previous messages in the stream. Each message is encoded using a random seed which is transmitted in the header of the message. A cyclic redundancy check (CRC) provides confirmation that the decryption of the message is successful.

The encryption is not designed to provide secure communication, as there are many methods for a cryptographic attack on a linear shift register based encoding. The approach will however provide protection from casual observation by third parties. For more robust encryption, OpenThings also offers provision for AES128.

DETAILS ON ENCRYPTION

Each transmission is sent according to the following message structure:

	Byte Num.	Data Name	Remark
Message header	0	Remaining length [7:0]	Number of bytes in whole message excluding this byte(x+2)
	1	Reserved bit [7]	Must be zero
		Manufacturer ID [6:0]	Manufacturer identifier
	2	Product ID [7:0]	Product identifier
	3	Encryption pip [15:8]	Pip for decoding encryption
	4	Encryption pip [7:0]	
	5	Sensor ID [23:16]	Unique Sensor ID to allow differentiation between end devices
	6	Sensor ID [15:8]	
	7	Sensor ID [7:0]	
Records	ˆ		
	Bytes 8 to x-1 contain one or more Records		
	ˇ		
Message footer	x	End of data [7:0]	NULL (0x00) to indicate end of data
	x+1	CRC [15:8]	CRC-16-CCITT
	x+2	CRC [7:0]	

Table 17: Linear encryption message example

The first five bytes of the message are not encrypted (bytes 0 to 4). The remainder of the message is encrypted prior to being sent.

The encryption is based on a random seed, and a pip of that seed is sent with each message. This enables each message to be decoded in isolation from the overall message stream.

The pip is a modified form of the seed based on the Encryption ID. The Encryption ID should be known by both sides of the link and enables the same protocol to be used across different manufacturer specific devices; for devices to be cross-compatible they must share the same Encryption ID.

The standard Sentec Encryption ID is 0x01 and this should be used for products that are intended to be open and cross-compatible with other manufacturers' products. Otherwise, a manufacturer-specific Encryption ID should be used, which can be allocated by Sentec on request.

HOW TO IMPLEMENT LINEAR SHIFT ENCRYPTION

The algorithm to implement encryption requires the seed to be randomised from another random source for each message. In practice, measuring the lowest bit of an analogue to digital conversion and shifting the result into a noise register gives a very good noise source. The noise source is then used in the `randomise_seed` function.

The pip can then be calculated from the current seed using the `generate_pip` function and added into the message to be sent.

Finally the remaining bytes of the message should be modified before transmission by feeding them through the `encrypt_decrypt` encryption routine in turn.

```
#define LOWER_BYTE(uint16_value) ((uint8_t) (uint16_value & 0xff))
#define UPPER_BYTE(uint16_value) ((uint8_t) (uint16_value >> 8))


/* Function to encrypt a whole outgoing message.
 * The length field in the messaged is used to determine
 * the end point of the message.
 */
void encrypt_message(uint8_t *buf, uint16_t noise)
{
      uint8_t  length;
      uint16_t message_pip;
      size_t   i;

      length = buf[MESSAGE_OFFSET_LENGTH];

      randomise_seed(noise);
      message_pip = generate_pip(ENCRYPTION_ID);

      buf[MESSAGE_OFFSET_PIP_HIGH] = UPPER_BYTE(message_pip);
      buf[MESSAGE_OFFSET_PIP_LOW] = LOWER_BYTE(message_pip);

      for (i = MESSAGE_OFFSET_ENCRYPTION_START;
            i <= length; ++i)
      {
            buf[i] = encrypt_decrypt(buf[i]);
      }
}
```

Figure 6: Linear shift encrypt code extract

HOW TO IMPLEMENT DECRYPTION

The algorithm to implement the decryption requires initialisation by calling the `seed` function with the Encryption ID and the Encryption pip. This is followed by calling `encrypt_decrypt` for each of the remaining bytes of the message.

```
/* Function to decrypt an incoming message.
 */
void decrypt_message(uint8_t *buf)
{
      uint8_t  length;
      uint16_t message_pip;
      size_t   i;

      length = buf[MESSAGE_OFFSET_LENGTH];

      message_pip = ((((uint16_t) buf[MESSAGE_OFFSET_PIP_HIGH]) << 8)
                                    & buf[MESSAGE_OFFSET_PIP_LOW]);

      seed(ENCRYPTION_ID, message_pip);

      for (i = MESSAGE_OFFSET_ENCRYPTION_START;
            i <= length; ++i)
      {
            buf[i] = encrypt_decrypt(buf[i]);
      }
}
```

Figure 7: Linear shift decrypt code extract

ENCRYPTION HELPER FUNCTIONS

```
/* Static variable maintaining the value of the linear shift
 * random number generator for the next transmission or
 * reception.
 */
static uint16_t random;

/* Function to randomise the seed based on a noise source
 * such as an analogue to digital converter.
 */
void randomise_seed(uint16_t noise)
{
      random = random ^ noise;
      if (random == 0)
      {
            random = 1; /* the seed must not be zero */
      }
}

/* Function to generate the pip to be sent in an encrypted
 * transmitted message, based on the current seed and eid,
 * the encryption id.
 */
uint16_t generate_pip(uint8_t encryptionid)
{
      return (random ^ ((uint16_t) encryptionid) << 8));
}

/* Function to update the seed to match the pip received in
 * the message.
 */
void seed(uint8_t encryptionid, uint16_t pip)
{
      random = (((uint16_t) encryptionid) << 8) ^ pip);
}

/* Function to encrypt or decrypt the next byte of data in the stream.
 */
uint8_t encrypt_decrypt(uint8_t dat)
{
      size_t i;

      for (i = 0; i < 5; ++i)
      {
            random = (random & 1) ? ((random >> 1) ^ 62965U) : (random >> 1);
      }
      return (uint8_t)(random ^ dat ^ 90U);
}
```

Figure 8: Linear shift helper functions

APPENDIX C - AES128 ENCRYPTION

Radio modules such as the RFM69W[1] from HopeRF provide on-chip support for AES128, enabling the use of the protocol with no significant overhead on the processor. The particular choice of the implementation of AES128 has been tailored to work with the RFM69W message structure.

Encryption key sharing

Both master and slave devices must have a shared key. A key is used by a master to connect to all slave devices; the same key is used for all slaves in one system. OpenThings specifies a key sharing scheme to allow slave devices to receive the system key securely while pairing.

The Long sensor ID (LSID) is made up of the Manufacturer ID, Product ID and Sensor ID.

Byte	0	1	2	3	4
Identity	Manufacturer ID	Product ID	Sensor ID[0]	Sensor ID[1]	Sensor ID[2]

Figure 9: Long sensor ID

Each Master device (Hub) is able to query a list of LSIDs that it is allowed to pair with, with the list stored either on the gateway itself, or checked remotely. The pairing process then proceeds as follows:

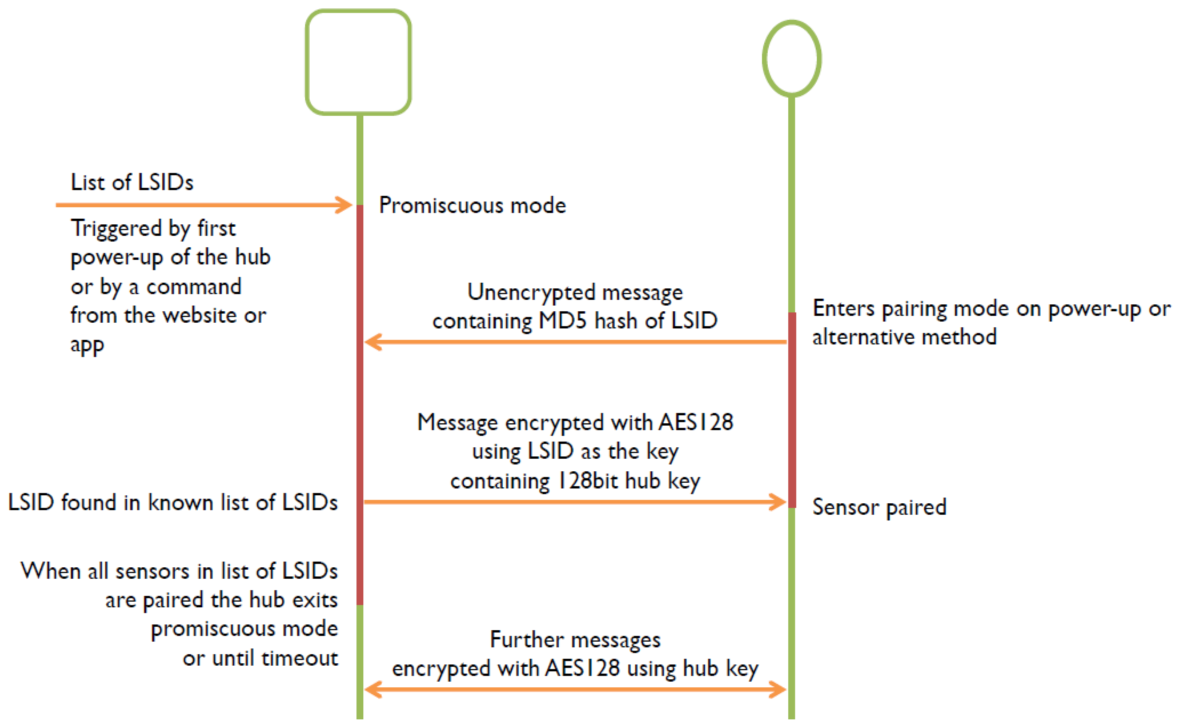
List of LSIDs
Triggered by first power-up of the hub or by a command from the website or app
Promiscuous mode
Unencrypted message containing MD5 hash of LSID
Enters pairing mode on power-up or alternative method
Message encrypted with AES128 using LSID as the key containing 128bit hub key
LSID found in known list of LSIDs
Sensor paired
When all sensors in list of LSIDs are paired the hub exits promiscuous mode or until timeout
Further messages encrypted with AES128 using hub key

Figure 10: Pairing state flow

Pairing messages from the slave device are repeated every 5 seconds until a response is received from hub. If the received LSID is not in the hub list, the hub should not send a key. To guard against brute force attacks, the hub should exit from promiscuous mode on receiving 10 incorrect LSIDs.

[1] http://www.hoperf.com/rf_transceiver/modules/RFM69W.html

AES encryption Scope

Each transmission is sent according to the following message structure:

	Byte Num.	Data Name	Remark
Message header	0	Remaining length [7:0]	Number of bytes in whole message excluding this byte(x+2)
	1	Reserved bit [7]	Must be zero
		Manufacturer ID [6:0]	Manufacturer identifier
	2	Product ID [7:0]	Product identifier
	3	Random salt [15:8]	Random salt for improving encryption diversity
	4	Random salt [7:0]	
	5	Sensor ID [23:16]	Unique Sensor ID to allow differentiation between end devices
	6	Sensor ID [15:8]	
	7	Sensor ID [7:0]	
Records	ˆ		
	Bytes 8 to x-1 contain one or more Records		
	ˇ		
Message footer	x	End of data [7:0]	NULL (0x00) to indicate end of data
	x+1	CRC [15:8]	CRC-16-CCITT
	x+2	CRC [7:0]	

Table 18: Message structure showing encrypted section in orange

The length byte of the message and the final CRC are not encrypted. The remainder of the message is encrypted prior to being sent. Table 8 shows the structure of an OpenThings message indicating in orange the part of the message encrypted under AES128. Bytes 3 and 4 are reserved for a random salt that maintains diversity in the message contents to improve security.

IMPLEMENTATION

For the implementation of the AES encryption using the HopeRF RFM69 module, we can allow the module to manage the preamble, start message synchronisation, the Manchester encoding, the packet length management, manufacturer ID filtering, encryption/decryption and the cyclic redundancy check. The management of the product ID and device ID filtering will be the responsibility of the application processor.

Preamble

The preamble is used to synchronise the clock timing between the transmitter and the receiver. In order to automatically transmit a preamble of alternating zeros and ones, the RegPreambleMsb and RegPreambleLsb registers are used to set up the length of the transmit preamble.

RegPreambleMsb	0x2C	PreambleSize [15:8]	MSB of the length of the preamble
RegPreambleLsb	0x2D	PreambleSize [7:0]	LSB of the length of the preamble

Table 19: Preamble registers

The default value of 3 bytes of preamble is appropriate for the OpenThings messages.

Message Start Synchronisation

The message start is indicated using a bit sequence that clearly identifies the end of a preamble. The sync control is turned on using the RegSyncConfig register. This register also configures the length (stored as length-1) and error management of the sync.

RegSyncConfig	0x2E	7 SyncOn 6 FifoFillCondition 5-3 SyncSize 2-0 SyncTol	Configuration of the sync testing for the start of messages

Table 20: Sync configuration registers

The value of the sync comparison is stored in up to eight bytes of the RegSyncValue registers. If the SyncSize is less than eight, the most significant bytes are used.

RegSyncValue1	0x2F	SyncValue [63:56]	MSB of sync
RegSyncValue2	0x30	SyncValue [55:48]	2nd byte of sync
RegSyncValue3	0x31	SyncValue [47:40]	3rd byte of sync
RegSyncValue4	0x32	SyncValue [39:32]	4th byte of sync
RegSyncValue5	0x33	SyncValue [31:24]	5th byte of sync
RegSyncValue6	0x34	SyncValue [23:16]	6th byte of sync
RegSyncValue7	0x35	SyncValue [15:8]	7th byte of sync
RegSyncValue8	0x36	SyncValue [7:0]	LSB of key

Table 21: Sync value registers

OpenThings uses a SyncValue of 0x2DD4. The registers are therefore set to RegSyncConfig = 0x88, RegSyncValue1=0x2D, RegSyncValue2=0xD4, with the rest set to zero.

Packet Length Management

The remaining packet length is sent as the first byte of the actual message. For this to be automatically interpreted, the PacketFormat setting in RegPacketConfig1 must be set to variable length:

RegPacketConfig1	0x37	7 PacketFormat 6-5 DcFree 4 CrcOn 3 CrcAutoClearOff 2-1 AddressFiltering	Configuration of the packet automation

Table 22: Packet configuration registers

In addition, for receiving, the PayloadLength that defines the maximum length of the payload must be set:

RegPayloadLength	0x38	PayloadLength	In variable length packet receive mode this is the maximum payload length of the packet

Table 23: Payload configuration registers

In OpenThings, we set the RegPacketConfig1 to 0xA0 and the RegPayloadLength to 0x40. The length byte is maintained as part of the delivered packet.

Encryption

The AES implementation using the HopeRF RFM69 module is turned on using bit 0 of the RegPacketConfig2 register (0x3D):

RegPacketConfig2	0x3E	7-4 InterPacketRxDelay 2 RestartRx 1 AutoRxRestartOn 0 AesOn	Configuration of packet management including the setting for the AES encryption and decryption

Table 24: Packet configuration registers for encryption

The key is stored in the AES key registers:

RegAesKey1	0x3E	AES Key [127:120]	MSB of key
RegAesKey2	0x3F	AES Key [119:112]	2nd byte of key
RegAesKey3	0x40	AES Key [111:104]	3rd byte of key
RegAesKey4	0x41	AES Key [103:96]	4th byte of key
RegAesKey5	0x42	AES Key [95:88]	5th byte of key
RegAesKey6	0x43	AES Key [87:80]	6th byte of key
RegAesKey7	0x44	AES Key [79:72]	7th byte of key
RegAesKey8	0x45	AES Key [71:64]	8th byte of key
RegAesKey9	0x46	AES Key [63:56]	9th byte of key
RegAesKey10	0x47	AES Key [55:48]	10th byte of key
RegAesKey11	0x48	AES Key [47:40]	11th byte of key
RegAesKey12	0x49	AES Key [39:32]	12th byte of key
RegAesKey13	0x4A	AES Key [31:24]	13th byte of key
RegAesKey14	0x4B	AES Key [23:16]	14th byte of key
RegAesKey15	0x4C	AES Key [15:8]	15th byte of key
RegAesKey16	0x4D	AES Key [7:0]	LSB of key

Table 25: Key registers

These values are maintained when the module is in sleep mode.

Cyclic Redundancy Check

The cyclic redundancy check (CRC) is also configured using the RegPacketConfig1 register:

RegPacketConfig1	0x37	7	PacketFormat	Configuration of the packet automation
		6-5	DcFree	
		4	CrcOn	
		3	CrcAutoClearOff	
		2-1	AddressFiltering	

Table 26: Packet configuration registers for CRC

In OpenThings, we set the RegPacketConfig1 to 0xA0 as described above. In this mode, the CRC is stripped off the payload being delivered.