Specification

Most parts of the protocol should not be modified to enable interoperability. Some of the protocol decisions are found on this page and ongoing discussions will show whether some of these specifications should be changed.

Advertising Packets

FruityMesh uses a number of advertising packets, the most important one being the *JOIN_ME* packet that is used for discovery of nearby nodes that also run FruityMesh.

Advertising Packet with Manufacturer Specific Data

There are two types of advertising packets used by FruityMesh. One uses the manufacturer specific advertisement structure, the other uses the serviceData advertisement structure.

Table 1. Advertising Header With Manufacturer Specific Data

Bytes	Type	Name	Description
3	advStructureFlags	Flags	Mandatory Flags Adv Structure
4	advStructureManufacturer	Manufacturer Data	Manufacturer Specific Adv Structure with 0x024D as company identifier.
1	u8	*meshIdentifier*	Identifies this as a mesh advertising packet
2	NetworkId	*networkId*	The network ID of this node
1	u8	*messageType*	Used to identify a number of different advertising messages within the mesh

Manufacturer ID

The company ID 0x024D (M-Way Solutions GmbH) is registered with the Bluetooth SIG and may be used by any implementation that conforms to the latest specifications of the FruityMesh protocol.

Mesh Identifier

The mesh identifier is 0xF0. No other identifier shall be used. Future versions might change this identifier. The purpose of the mesh identifier is to enable different protocols under the same manufacturer ID.

Network Identifier

The network identifier can be changed by the user. In combination with the *networkKey*, this allows different meshes to co-exist in the same physical area. It is set either during flashing or during enrollment.

Message Type

The *messageType* is used to distinguish between different messages. The same *messageType* definitions are used for advertising packets and for connection packets. This allows us to send the same message over another "transport" if desired and compatible with the message format.

JOIN_ME Packet (MessageType 1)

The *JOIN_ME* packet contains all the information that other nodes can use to determine their best connection partner. The *messageId* in the header must be set to 1. Future *JOIN_ME* packets can have a different *messageId* and different values that can then be used in the Cluster Score Function. The current implementation uses the *clusterSize* and the number of *freeIn* and *freeOut* connections.

Bytes	**Type**	**Name**	**Description**
11	advHeaderManufacturerSpecific	*header*	The above depicted manufacturer specific header
2	NodeId	*sender*	The *nodeId* of the sending node
4	ClusterId	*clusterId*	Consists of the founding node's ID and the connection loss / restart counter
2	ClusterSize	*clusterSize*	The number of nodes in this cluster
3 bit	u8 : 3	*freeMeshInConnections*	Number of free in connections (as peripheral)
5 bit	u8 : 5	*freeMeshOutConnections*	Number of free out connections (as central)
1	u8	*batteryRuntime*	Contains the expected runtime of the device (1-59=minutes, 60-83=1-23hours, 84-113=1-29days, 114-233=1-119months, 234-254=10-29years, 255=infinite)
1	i8	*txPower*	Power of two's complement in dBm

Bytes	Type	Name	Description
1	u8	*deviceType*	cf. Device Types
2	u16	*hopsToSink*	Number of hops to the shortest sink
2	u16	*meshWriteHandle*	The GATT handle for the mesh communication characteristic
4	ClusterId	*ackField*	Contains the acknowledgement from another node for the slave connection procedure

Advertising Packet With Service Data

Another type of advertising packet used by FruityMesh uses the Service Data Adv Structure. This is important when dealing with mobile devices since some have hardware filtering with no support for manufacturer specific data.

Bytes	Type	Name	Description
Mandatory data			
3	Flags	*advStructureFlags*	(len, type, flags byte)
4	Service UUID complete	*advStructureUUID16*	(len, type, 2 byte UUID 0xFE12)
4	Service Data header	*advStructureServiceData*	(len, type, 2 byte UUID 0xFE12)
CustomDataHeader			
2	u16	*messageType*	Used to determine between different messages.

Other Advertising Packets

FruityMesh can be used to distribute all advertising packets that conform to the BLE specification. These can be Eddystone, iBeacon or any other kind of advertising messages. These are however not essential for FruityMesh itself and are therefore not documented here. Have a look at the AdvertisingModule for more information.

Connection Packets

The mesh uses a number of packets that are sent over connections. Most packets that are sent over connections must have this header. There are some exceptions to this (e.g. split packets use a two byte message header for less overhead. The *messageType* is used to identify if the *connPacketHeader* is used or not.

Connection Packet Header

Table 2. Format of a connPacketHeader

Bytes	Type	Name	Description
1	u8	*messageType*	Type of message
2	u16	*senderId*	Node ID of the sender
2	u16	*receiverId*	Node ID of the receiver

Modules

FruityMesh uses a concept of modules to group functionality into different parts. This works together nicely with the featuresets. A user is able to write his own modules that extend the basic functionality of FruityMesh. Each module is identified using a module id.

ModuleIds

There are two types of module ids: the standard **ModuleId** has a size of one byte and is solely used by standardized modules. The **VendorModuleId** on the other hand uses 4 bytes to allow different vendors to develop modules that can run together in a single network without clashing with other modules that were written by different vendors on different nodes in the same mesh network. This is done by including a vendor id as part of the VendorModuleId. This vendor id must be set to the company identifier that can be acquired from the BLE SIG. See this page (https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers/) for instructions on how to get this company identifier and for a complete list of known company identifiers.

VendorModuleIds were not part of FruityMesh prior to version 1.x.x. FruityMesh tries to keep the usage of VendorModuleIds as easy as possible and does most of the work when using VendorModuleIds. Some parts will e.g. use different message types for messages with ModuleId and VendorModuleId and other parts use the ModuleIdWrapper to be able to work with both module id types in the same data structure. As a convention, whenever a ModuleId is printed out as a string, it is printed as a decimal number, e.g. `123`. VendorModuleIds on the other hand are printed as a string like this `"0xABCD01F0"` to make the individual parts easier to read.

To get started, see Implementing a Custom Module.

ModuleId

In the implementation, the ModuleId is either used as a single byte or stored using the 4 byte ModuleIdWrapper with the subId and vendorId set to 0. This depends on the implementation of a feature.

VendorModuleId

The VendorModuleId is always composed of 4 bytes can be put together using the ModuleIdWrapper.

Bytes	Type	Name	Description
1	u8	prefix	This must be set to VENDOR_MODULE_ID_RESERVED(0xF0) to be recognized as a VendorModuleId

Bytes	Type	Name	Description
1	u8	subId	The user can use this to specify up to 254 different modules with his vendorId. 0x00 and 0xFF are reserved
2	u16	vendorId	The company identifier as assigned by the BLE SIG

Module Packet Header

Modules use an bigger message header to guarantee that there are no collisions between different functionality. The following describes the format of the header:

Table 3. Format of a connPacketModule or connPacketModuleVendor Header

Bytes	Type	Name	Description
5	connPacketHeader	*header*	MessageType must be one e.g. of the above.
1 or 4	u8 or u32	moduleId or vendorModuleId	Either a ModuleId of a core module or the VendorModuleId of a vendor specific module.
1	u8	*requestHandle*	A handle that can be used e.g. like a counter. Responses will always be returned with the same handle given in the request.

Bytes	Type	Name	Description
1	u8	*actionType*	This is the type of action that should be executed by the module. An individual list of *subCommands* is available for each of the *messageTypes* given above. E.g. there could be a `MODULE_TRIGGER_ACTION` message with the *actionType* set to 1 (PING) to execute a ping. The response would be a `MODULE_ACTION_RESPONSE` message with the *actionType* set to 1 (`PING_RESPONSE`).
…	u8[]	*data*	additional payload data for the command

The connPacketModule differs from the connPacketModuleVendor header in the size of the module id. They can be differentiated by checking the first byte, which is set to ModuleId::VENDOR_MODULE_ID_PREFIX (0xF0) in case of a VendorModuleId. While the connPacketModule has a size of 8 bytes, the connPacketModuleVendor needs 11 bytes.

Module Packet Header for Actions / Responses and Events

The most common usage of the connPacketModule header is to provide the possibility to trigger actions, get responses and to fire events. This is a command and response based schema that is very well suited to communicate between different nodes in a mesh network. The following MessageTypes are used for this:

Table 4. Module MessageTypes for Actions / Responses and Events

MessageType	Name	Description
51 / 0x33	MODULE_TRIGGER_ACTION	A request for a node to perform an action
52 / 0x34	MODULE_ACTION_RESPONSE	Response message for a previous request
53 / 0x35	MODULE_GENERAL	An event that does not need a response

Actuator and Sensor Messages

Often, FruityMesh must tunnel a different protocol such as Modbus because it is installed on a controller that is attached to a 3rd party controller. In this case, it is necessary to have a tunneling protocol that is generic enough to provide access to all functionality while still allowing a platform or MeshGateway to interpret the received data in a useful manner. This is why we have introduced the `component_sense` and `component_act` message types. You should use them whenever you report sensor data or when you want to write data to the node that should be written to a specific "address".

Table 5. Module MessageTypes for Actuators and Sensors

MessageType	Name	Description
58 / 0x3A	COMPONENT_ACT	A request to trigger an actuator
59 / 0x3B	COMPONENT_SENSE	A response from a sensor

Take a look at the detailed documentation for Sensors and Actuators.

Raw Data

Another use-case is to tunnel any kind of data through FruityMesh where the nodes do not need to parse or process the data at all, e.g. a smartphone might want to send data to a MeshGateway or Backend. This could also be data of an entirely different protocol such as HTTP, etc... This could be a small data packet or a rather large file that needs to be split into several parts. To do this, we have introduced `raw_data` and `raw_data_light` which are documented in a lot of detail at the Raw Data page.

Table 6. Module MessageTypes for Raw Data

MessageType	Name	Description
54 / 0x36	MODULE_RAW_DATA	Used to transmit large files or large amounts of data (acknowledged, similar to TCP)
55 / 0x37	MODULE_RAW_DATA_LIGHT	Used to transmit up to "MAX_MESH_PACKET_SIZE" bytes of data (see `Config.h`) (unacknowledged, similar to UDP)

Module Configuration Messages

Another MessageType is dedicated to configure modules and to get more information about them.

Table 7. Module MessageTypes for Configuring Modules

MessageType	Name	Description
50 / 0x32	MODULE_CONFIG	Used to retrieve or set a module configuration or get more information about modules

Node IDs

A NodeId is a way of addressing devices in a network. Each device in a network must have a unique *nodeId* assigned to it that must not clash with the node ID of another device.

There are different node ID ranges that are used for different purposes:

Table 8. Node ID ranges

Name	Node ID	Usage

Name	Node ID	Usage
NODE_ID_BROADCAST	0	Broadcast address to reach all nodes in a network
NODE_ID_DEVICE_BASE	1…1999	Uniquely address devices (nodes, sinks, …) given by enrollment
NODE_ID_VIRTUAL_BASE	2000…19999	Virtual addresses to address smartphones connected to the mesh (dynamically assigned, aka. virtual partner id)
NODE_ID_GROUP_BASE	20000…20999	Address groups of devices (statically assigned at compile time)
NODE_ID_LOCAL_LOOPBACK	30000	Address for the current node itself (similar to localhost)
NODE_ID_HOPS_BASE	30001…30999	Specify the number of hops that a packet can travel. (30 001 e.g. specifies that the packet must only reach the direct neighbours)
NODE_ID_SHORTEST_SINK	31000	Used to send a packet to all sink nodes (see Device Types)
NODE_ID_APP_BASE	32000	NodeId given to non-mesh devices that can connect via a MeshAccessConnection (e.g. a Smartphone). This is replaced by a virtual partner id during communication.
NODE_ID_GLOBAL_DEVICE_BASE	33000…39999	Assign nodeIds uniquely over multiple meshes for the same organization (used for assets that roam between different networks)
NODE_ID_INVALID	65535	Invalid node ID, which is used for internal errors

Name	Node ID	Usage
-	others	All other node IDs are currently reserved

Serial Numbers / SerialNumberIndex

The serial numbers use a special alphabet of ASCII characters that is easily readable (`BCDFGHJKLMNPQRSTVWXYZ123456789`) and will not result in funny words because it does not contain vocals. Serial numbers are either 5 or 7 characters long. Each serial number can be converted to a serialNumberIndex which is represented as an unsigned 32 bit integer. If a serial number is human readable, it can be printed in its ASCII representation, but if sending a serialNumber over the network or using it in code, the serialNumberIndex should be used. Check out the appropriate methods (*GenerateBeaconSerialForIndex* and *GetIndexForSerial*) in the *Utility* class on how to convert the serial number. Also see the appropriate tests in the *TestUtility* class.

There are two types of serial number ranges. There is a range of two billion serial numbers that is exclusively managed by M-Way Solutions. This range contains serial numbers with 5 characters and also serial numbers with 7 characters. Parts of this range are licensed to partners. The second range can be used for projects working with the open source version of FruityMesh completely free of charge. This is always a 7 character serial number.

M-Way Solutions Proprietary Serial Number Range

The serialNumberIndex of this range starts with 0 for the serial number `BBBBB` and increments up to `0x7FFFFFFF` for the serial number `D8PJQ8K` . This range should not be used without consulting us beforehand because serial numbers will clash otherwise. We assign sub-ranges to our partners which allows them to use serial numbers with 5 characters.

Open Source Testing Range

To provide beginners with an easy method for testing FruityMesh, we decided to open source a part of our proprietary range that contains nicely readable serial numbers. This range starts from `2673000` (`FMBBB`) and ranges until `2699999` (`FM999`).By default, the serial number is generated randomly (from the unique deviceId stored in the chipset) within this testing range if there is no UICR data available (see *generateRandomSerialAndNodeId* in the Config class). This makes it easier to start testing FruityMesh. You should not distribute products that use a serial number from this range as it will clash with others.

Open Source Serial Number Range for Products

The open sourced serial number range that you can use for final products uses serial numbers with 7 characters to avoid a clash between different vendors. A serial number is constructed by setting the most significant bit of the serialNumberIndex to 1. The next 16 bits must be set to a two byte company identifier assigned by the Bluetooth SIG (https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers/). The remaining 15 bits are used to generate the individual serial numbers for this vendorId. This gives each vendor the possibility to generate around 32 thousand unique serial numbers. (See *VendorSerial* type).

If you do not have a Bluetooth SIG company identifier yet, you can use the manufacturer id from M-Way Solutions (`0x024D`). This range starts from `0x81268000` (`D9HCK3N`) and ranges until `0x8126FFFF` (`D9HDSHW`). You should not publicly distribute products that use this serial number range as it could clash with other serial numbers. If you want to distribute a product, you should register with the Bluetooth SIG to get a free company identifier that you can set as the MANUFACTURER_ID in the configuration.

Encryption Keys

There are a number of different keys used throughout FruityMesh. These are all 128-bit keys and are used for AES encryption between the nodes, as well as for communication with smartphones or other devices.

No Key (FM_KEY_ID_ZERO = 0)

Can only be used if a node is not enrolled and uses a key filled with all 0x00 for encryption.

Node Key (FM_KEY_ID_NODE = 1)

This key is used for the lifetime of a device and is uniquely generated during production. It must be kept secure because it allows full configuration access, e.g. enrolling and removing the enrollment.

Network key (FM_KEY_ID_NETWORK = 2)

The network key is shared between all nodes that belong to a mesh network. Whoever is in posession of this key can configure all nodes in the network and can send any message. It is important to keep this key secret, but it is possible to change it if it is compromised.

UserBase Key (FM_KEY_ID_BASE_USER = 3)

This is a key that can't be used to connect. It is used to derive all other user keys.

Organization Key (FM_KEY_ID_ORGANIZATION = 4)

The organization key is shared between all networks of an organization. It allows access to a limited set of functionality, e.g. necessary for tracking assets between differen meshes. If the organization key leaks, it is necessary to reconfigure all meshes of the organization.

Restrained Key (FM_KEY_ID_RESTRAINED = 5)

The restrained key is generated based on the node key. It is a node key with limited access rights.

The "restrained key" can be derived from the node key by using an AES-128 bit encryption by encrypting the ASCII-String "RESTRAINED_KEY00" (without terminating 0) using the node key as the AES key. Example values are:

Table 9. Examples of node keys and "restrained keys"

Node Key	Restrained Key
00:11:22:33:44:55:66:77:88:99:AA:BB:CC:DD:EE:FF	2A:FC:35:99:4C:86:11:48:58:4C:C6:D9:EE:D4:A2:B6
FF:EE:DD:CC:BB:AA:99:88:77:66:55:44:33:22:11:00	9E:63:8B:94:65:85:91:99:A9:74:7D:A7:40:7C:DD:B3
DE:AD:BE:EF:DE:AD:BE:EF:DE:AD:BE:EF:DE:AD:BE:EF	3C:58:54:FC:29:96:00:59:B7:80:6B:4C:78:49:8B:27
00:01:02:03:04:05:06:07:08:09:0A:0B:0C:0D:0E:0F	60:AB:54:BB:F5:1C:3F:77:FA:BC:80:4C:E0:F4:78:58

User Keys (FM_KEY_ID_USER_DERIVED_START = 10 to UINT32_MAX / 2)

The user base key is used to generate millions of user keys that can be given to users or user groups. A user key allows access to a limited set of commands and can be restricted in functionality depending on the use case. If the *userBaseKey* leaks, all *userKeys* have to be regenerated and distributed to users.

A key that is filled with 0xFF is considered invalid and cannot be used.

Device Types

There are different device types that are given to nodes with specific functionality:

Table 10. List of Device Types

DeviceType	Name	Description
0	DEVICE_TYPE_INVALID	Not used

DeviceType	Name	Description
1	DEVICE_TYPE_STATIC	A node that is installed somewhere with a position that will not change much over time.
2	DEVICE_TYPE_ROAMING	A node that can move around freely.
3	DEVICE_TYPE_SINK	A node that is installed at a fixed place and collects all the data (typically a MeshGateway).
4	DEVICE_TYPE_ASSET	A node that moves around and broadcasts its presence so that it can be detected by a mesh.
5	DEVICE_TYPE_LEAF	A node that will only connect to the mesh as a leaf but will not relay any data (Useful if its position changes but it needs a constant data connection)

For more explanation, see Device Types.

UICR

The UICR is a special persistant storage that is used to store factory defaults once a node is flashed. The NRF_UICR→CUSTOMER area is used to store the data on nRF chips.

If you want to store a serial number, *nodeKey*, etc. for a node, you must write the UICR during flashing. The NRF_UICR→CUSTOMER area is used for that purpose and starts at 0x10001080. You can use srec_cat (http://srecord.sourceforge.net/) to produce a .hex file containing the desired UICR data. This can then be merged with the SoftDevice and Application or you can flash each one separately. For detailed instructions, see our Developers chapter.

FruityMesh will boot with random data (random *nodeId* / *serialNumber* / ...) if no data is present in the UICR. The data will however be persistent across reboots as it is generated according to the internal chip id from the FICR. Layout of UICR memory:

Table 11. Layout of UICR memory

Offset	Size (Bytes)	Name	Description
0	4	MAGIC_NUMBER	Must be set to 0x00F07700 when UICR data is available

Offset	Size (Bytes)	Name	Description
4	4	BOARD_TYPE	Accepts an integer that defines the hardware board that FruityMesh should be running on (*boardId*, a.k.a. *boardType*)
8	8	SERIAL_NUMBER	Deprecated: This contained the given serial number as ASCII (zero terminated) but is not used anymore (since 12.05.2020). Must now be set to FFFF....FFFF. The serial number is instead calculated from the SERIAL_NUMBER_INDEX
16	16	NODE_KEY	Should be securely and randomly generated
32	4	MANUFACTURER_ID	Set to manufacturer ID according to the BLE company identifiers (https://www.bluetooth.org/en-us/specification/assigned-numbers/company-identifiers)
36	4	DEFAULT_NETWORK_ID	0: unenrolled; 1: using an enrollment network; other: default enrollment
40	4	DEFAULT_NODE_ID	Node ID to be used while not enrolled
44	4	DEVICE_TYPE	Type of device according to Device Types
48	4	SERIAL_NUMBER_INDEX	Unique index that represents the serial number
52	16	NETWORK_KEY	Default network key if pre-enrollment is used

Heap usage

Heap usage (malloc / new) is prohibited in the FruityMesh codebase. To ensure that this rule is followed, a linker flag for ld is used that generates a linker error if malloc is used. The error looks something like this:

```
Make: new_op.cc:(.text._Znwj+0xe): undefined reference to `__wrap_malloc'
```

If this happened to you, you have to remove the malloc / new usage.