MagikEye API v1.0

Magik Eye Inc.

COLLABORATORS

	TITLE : MagikEye API v1.0		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Magik Eye Inc.	Jul 2020	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1702-00-EN-06	Jul 2020		MEI

Contents

List of Figures

1 Introduction

This document describes the Magik Eye API of version 1.0 (hereinafter referred to as the *MkE API*): the application programming interface that defines the communication between the Magik Eye depth sensor (hereinafter referred to as the *sensor*) and the sensors' user.

This document is divided into four parts. The first part describes the operational states of the sensor. The second part describes the communication protocol between the sensor and the sensor's user (hereinafter referred to as the *client*). The third part describes the data format of the 3D data provided by the sensor. The last part of this documents provides the full API reference.

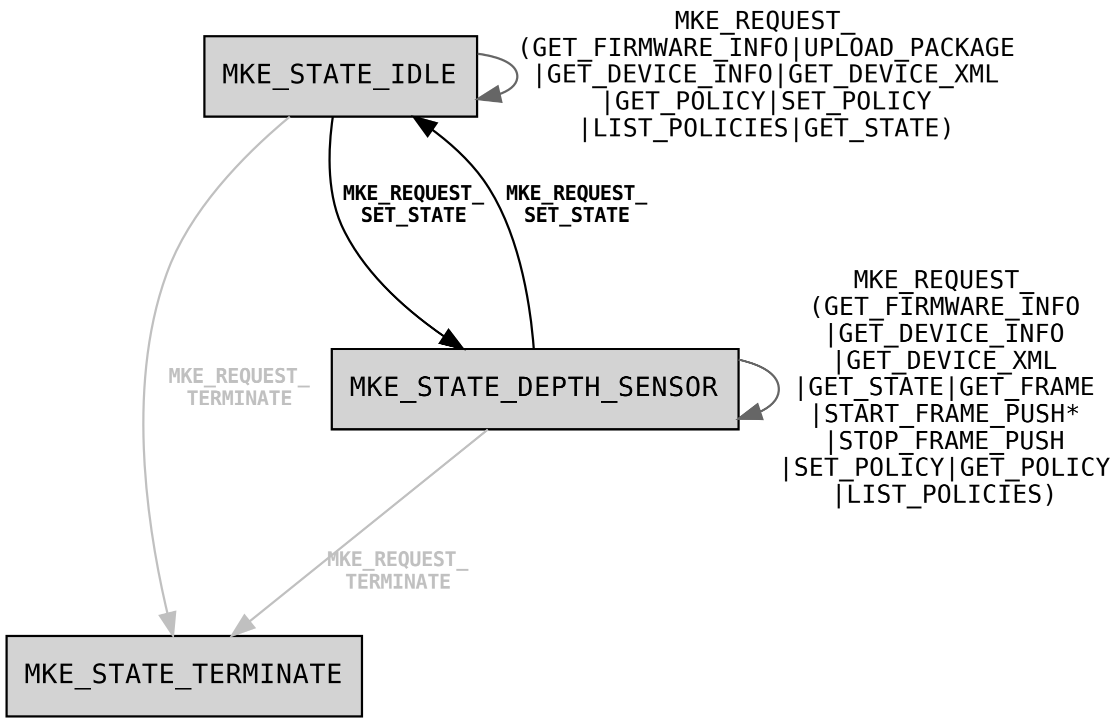

Figure 1: Sensor states

2 Sensor States

The sensor has two main states of operation, see Figure 1. After booting, the sensor will automatically move to the state `MKE_STATE_IDLE`. In this state, no 3D sensing is performed and the sensor consumes only limited power resources. In order to provide 3D measurements, the sensor needs to pass to the state `MKE_STATE_DEPTH_SENSOR`. In this state, the sensor computes the depth information and, upon request, provides this information to the client. The sensor can be, at any time, powered off or rebooted by passing it into the (pseudo) state `MKE_STATE_TERMINATE`. Optionally, the sensor can once again pass to the power saving `MKE_STATE_IDLE` state. The sensor and its client communicate by exchanging data packets called requests and replies. Figure 1 lists the request types available in each of the sensor's states.

3 Communication Protocol

The communication protocol between the sensor and the client is based on the client-server model. The client sends a fixed size data packet, a *request*, and the server (sensor) responds with variable-sized data packets, *replies*:

```
Client                          Server
---------------------------------
Request      ----------->
            <-----------       Reply
```

Both request and reply contain a human readable textual part and a binary part, where any parameters and payload data are contained.

Note
Note that all data is interpreted by the sensor using the *little endian encoding*.

3.1 Transport Layer

The MkE API describes the application layer of the communication protocol between the sensor and the client. The lower layers of the sensor-client communication are sensor specific. Practically, there are two main methods the client can use to physically connect to the sensor. The first is the UART interface, where no transport layer is needed. The second method is Ethernet where the TCP protocol is used as the transport layer. In the case the TCP connections are supported by a sensor, the MkE API server will usually listen on the port 8888.

3.2 Network Discovery

When Ethernet and TCP connections are supported by a sensor, the sensor can also optionally implement SSDP (Simple Service Discovery Protocol) for advertisement and discovery. SSDP is an internet standard and a part of UPnP (Universal Plug and Play) standard. The USN (Unique Service Name) URI of a MagikEye sensor will always contain "MkE" as a substring.

3.3 Request

The client request data packet is *always at least* 24 bytes long. The following C-style structure describes the inner structure of the request data packet:

```
struct MkERequest {
  char magik[8];
  char type[4];
  uint32_t reqid;
  MkERequest_Params params;
};
```

The `MkERequest` fields have the following meaning:

Field	*Definition*
`char magik[8]`	MkE API request packet identifier, must be set to: `MKERQ100`.
`char type[4]`	Request type as a zero padded decimal string, e.g., `0020` for request type 20, see Section 6.1
`uint32_t reqid`	ID number of a request. A respective response will have the same ID.
`MkERequest_Params params`	8 -byte long data structure describing the respective MkE API request parameters.

Note that the API does not enforce any special values or sequence of data passed by `reqid` ID. This identification will simply be a part of the sensor's response. It is up to the client to decide if and how to use this information.

3.4 Reply

The sensor reply data packet is *always at least* 48 bytes long. The following C-style structure describes the inner structure of the first 48 bytes of the reply:

```
struct MkEReply {
  char magik[8];
  char type[4];
  char status[4];
  uint32_t reqid;
  uint32_t num_bytes;
  MkEReply_params params;
};
```

The `MkEReply` fields have the following meaning:

Field	*Definition*
`char magik[8]`	MkE API reply packet identifier. Must be set to: `MKERP100`.
`char type[4]`	Request type of the original Mke API request that this data packet replies to, e.g., `0020` for request type 20.
`char status[4]`	Reply status as a zero padded decimal string, e.g., `0200` for `MKE_REPLY_OK`, see Section 6.2.
`uint32_t reqid`	ID number of a request that this data packet replies to.
`uint32_t num_bytes`	Size of the additional payload data directly following this reply. If there is no additional payload, this must be set to 0.
`MkEReply_params params`	24-bytes long data structure describing the respective MkE API reply parameters.

The `num_bytes` field indicates that an appropriate number of bytes will directly follow the first 48 bytes of the reply.

Note

Note that since each request may vary in its processing time by the sensor, there is no guarantee that replies will arrive in the same order as requests were sent by the client. It is up to the client to account for this fact, e.g., using the `reqid` field.

3.5 Examples

This section presents several examples of client and server data packet exchanges.

3.5.1 Querying the Sensor State

Let us assume that the sensor was just powered on. After the booting process, the sensor will pass into the `MKE_STATE_IDLE` state. As a user, you can check this by quering the sensor via the request `MKE_REQUEST_GET_STATE` (numerical code 20), see Section 6.1.5. The `MKE_REQUEST_GET_STATE` request type, together with `reqid` set to `0A`, lead to the following request data packet:

```
00000000  4D 4B 45 52 51 31 30 30 30 30 32 30 MKERQ1000020
0000000C  0A 00 00 00 00 00 00 00 00 00 00 00 ............
```

Here, the first number in the row specifies the hexadecimal data offset, followed by 12 data bytes. This is followed by the same 12 data bytes interpreted as an ASCII string. To such a request, the sensor will reply with the following data packet:

```
00000000  4D 4B 45 52 50 31 30 30 30 30 32 30 MKERP1000020
0000000C  30 32 30 30 0A 00 00 00 00 00 00 00 0200........
00000018  01 00 00 00 00 00 00 00 00 00 00 00 ............
00000024  00 00 00 00 00 00 00 00 00 00 00 00 ............
```

Here, the sensor replied to the request of type `MKE_REQUEST_GET_STATE` and `reqid` set to `0x0A` with the reply status `MKE_REPLY_OK` (Numerical code 200), indicating that the request was successfully handled. Further, it provided the response via the `params` part of the reply data packet as `MKE_STATE_IDLE` (numerical code 1).

3.5.2 Setting the Sensor State

To change the sensor state from `MKE_STATE_IDLE` to `MKE_STATE_DEPTH_SENSOR`, use the `MKE_REQUEST_SET_STATE` request type (numerical code 21), see Section 6.1.6. Assuming we incremented `reqid` to `0x0B`, the corresponding request data packet will look like this:

```
00000000  4D 4B 45 52 51 31 30 30 30 30 32 31 MKERQ1000021
0000000C  0B 00 00 00 02 00 00 00 00 00 00 00 ............
```

If all goes well, the sensor will readily reply with the following data packet with reply status `MKE_REPLY_OK`:

```
00000000  4D 4B 45 52 50 31 30 30 30 30 32 31 MKERP1000021
0000000C  30 32 30 30 0B 00 00 00 00 00 00 00 0200........
00000018  00 00 00 00 00 00 00 00 00 00 00 00 ............
00000024  00 00 00 00 00 00 00 00 00 00 00 00 ............
```

3.5.3 Powering Off the Sensor

To request a physical shutdown of the sensor, use `MKE_REQUEST_TERMINATE` request type (numerical code 10) with parameter `method` set to `MKE_TERMINATE_BY_SHUTDOWN`, see Section 6.1.1. Assuming we once again incremented the `reqid` to `0x0C`, the corresponding request data packet will look like this:

```
00000000  4D 4B 45 52 51 31 30 30 30 30 31 30 MKERQ1000010
0000000C  0C 00 00 00 02 00 00 00 00 00 00 00 ............
```

The sensor will reply with `MKE_REPLY_OK` status in the case it registered the request and is on its way to perform the termination action. Of course, it cannot confirm a successful shutdown after termination action is completed.

4 3D Data

Once the sensor is in the `MKE_STATE_DEPTH_SENSOR` state, it is ready to provide 3D measurements to the client. The 3D data is provided as a reply data payload, which is in this context called a *frame*. There are two ways the client can decide to request frames:

- by *client polling* via `MKE_REQUEST_GET_FRAME` request, or
- by *sensor pushing* via `MKE_REQUEST_(START|STOP)_FRAME_PUSH` requests.

The frame itself consists of a variable number of *frame items*--which conceptually correspond to detections, i.e., 3D points—and a *frame footer*, containing the CRC-32 checksum (ITU-T V.42) of the whole frame. There are two types of data items: `MKE_FRAME_TYPE_1` and `MKE_FRAME_TYPE_2`.

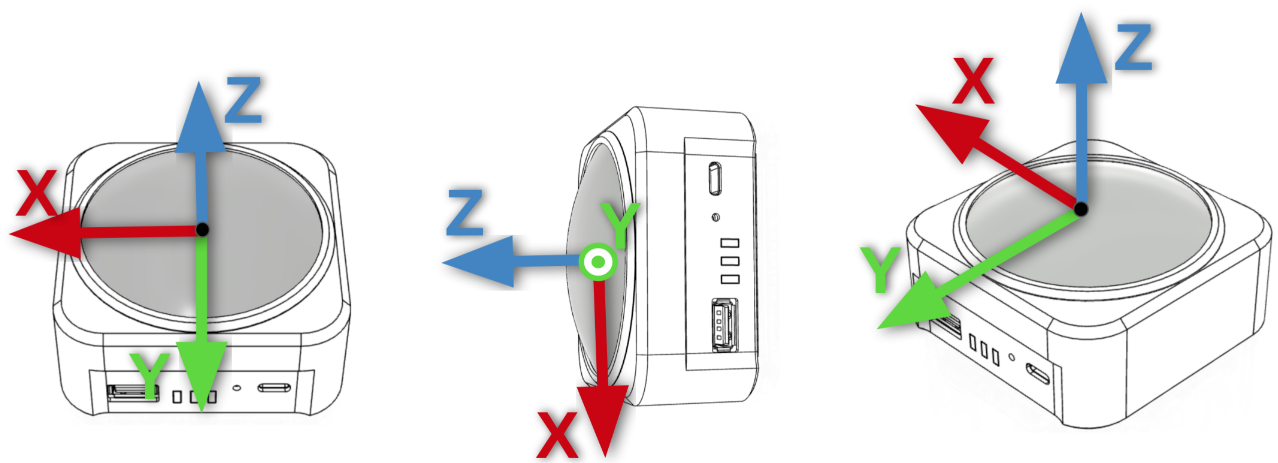

Figure 2: An example of sensor axes placement

4.1 3D Data Frame

If a reply data packet contains a data payload, it signals the same via the non-zero reply field `num_bytes`, see Section 3.4. This allows the client to wait for the appropriate number of bytes before attempting to parse its contents. As it is practically impossible to detect all theoretical detection points in each image captured by the sensor's camera, the values of `num_bytes` fields of reply packets carrying frames will generally differ. This will depend on the number of 3D points the sensor was able to recover for each given camera image.

4.1.1 `MkEReply` Parameters

A frame-carrying reply 24-byte field `params` has a specific structure described by the following C-style structure:

```
struct MkEReply_Frame {
  uint64_t timer;
  uint64_t seqn;
  uint32_t data3d_type;
  uint16_t frame_type;
  uint16_t num_data;
};
```

The `MkEReply_Frame` fields have the following meaning:

Field	*Definition*
`uint64_t timer`	Time of the end of the camera image exposure in milliseconds elapsed from the boot. This can be used to measure time elapsed between different frame exposures.
`uint64_t seqn`	Sequence number of a frame. This counter is incremented every time a frame is successfully processed by the sensor in the `MKE_STATE_DEPTH_SENSOR` state.
`uint32_t data3d_type`	Determines the units of the 3D coordinates of the frame items. Currently, there are five possible types, see Section 4.1.2.
`uint16_t frame_type`	Determines the frame item type: either `MKE_FRAME_TYPE_1` (numerical code 1) or `MKE_FRAME_TYPE_2` (numerical code 2).
`uint16_t num_data`	Number of frame items.

4.1.2 3D Data Types

Currently, there are five possible 3D data types (units) that the sensor can use to encode the 3D coordinate values. The 3D data type is determined by the sensor and cannot be changed by an `MkERequest`. However, it is possible for the sensor to change the 3D data type during its operations.

Type	*Value*	*Units*
`MKE_DATA3D_MM`	0	1 millimeter
`MKE_DATA3D_MM2`	1	1/2 millimeter
`MKE_DATA3D_MM4`	2	1/4 millimeter
`MKE_DATA3D_MM8`	3	1/8 millimeter
`MKE_DATA3D_MM16`	4	1/16 millimeter

4.1.3 Frame Items

Immediately after the first 48 bytes of MkEReply follow the num_data frame items. Every item in a given frame has the item type given by the frame_type field.

The following C-style structure MkEFrameItem1 describes the inner structure of frame item type MKE_FRAME_TYPE_1:

```
struct MkEFrameItem1 {
  uint16_t uid;
  int16_t  x, y, z;
};
```

The MkEFrameItem1 frame item is 8 bytes long and its fields have the following meaning:

Field	*Definition*
uint16_t uid	Universal ID key of the detection.
int16_t x, y, z	3D coordinates of the detection in the units determined by data3d_type in the coordinate system connected with the sensor, see Figure 2.

The following C-style structure MkEFrameItem2 describes the inner structure of frame item type MKE_FRAME_TYPE_2:

```
struct MkEFrameItem2 {
  uint16_t uid;
  int16_t  x, y, z;
  uint16_t lid, did;
};
```

The MkEFrameItem2 frame item is 12 bytes long and its fields have the following meaning:

Field	*Definition*
uint16_t uid	Universal ID key of the detection.
int16_t x, y, z	3D coordinates of the detection in the units determined by data3d_type in the coordinate system connected with the sensor, see Figure 2.
uint16_t lid, did	Reserved for future use.

4.1.4 Frame Footer

Finally, immediately after the last frame item follows the frame footer:

```
struct MkEFrameFooter {
  uint32_t crc32;
};
```

The MkEFrameFooter structure has only one field as below:

Field	*Definition*
`uint32_t crc32`	CRC-32 check (ITU-T V.42) of the frame items bytes.

4.1.5 Example

Let us inspect the following reply data packet:

```
00000000  4D 4B 45 52 50 31 30 30 30 30 32 36 MKERP1000026
0000000C  30 32 30 30 01 00 00 00 24 00 00 00 0200....$...
00000018  AD 0D AC BA 00 00 00 00 02 00 00 00 ............
00000024  00 00 00 00 00 00 00 00 01 00 04 00 ............
00000030  07 00 AE FF E4 FF 4F 00 0B 00 A1 FF ......O.....
0000003C  E4 FF 40 00 0C 00 B7 FF E5 FF 56 00 ..@.......V.
00000048  12 00 A8 FF E4 FF 47 00 99 38 6B BA ......G..8k.
```

We can see that this data packet is a successful reply, `MKE_REPLY_OK` (numerical code 200), to a `MKE_REQUEST_GET_FRAME` (numerical code 26) request, see Section 6.1.12, with `reqid` = 1. Now, because this data is a reply to `MKE_REQUEST_GET_FRAME`, we should interpret the reply `params` as `MkEReply_Frame`. Here, the fields of `MkEReply_Frame` have the following values: `timer` = `0xBAAC0DADul` = 3131837869, `seqn` = 2, `frame_type` = `MKE_FRAME_TYPE_1`=1, and `num_data` = 4.

Besides the 48 bytes of `MkeReply`, the data packet also contains `num_bytes` = 36 bytes of additional data payload. This corresponds to `num_data` = 4 times 8 bytes of `MkeFrameItem1`, plus 4 bytes of `MkEFrameFooter`. The frame parses into four 3D points (-82, -28, 79), (-95, -28, 64), (-73, -27, 86), and (-88, -28, 71) with four respective IDs 7, 11, 12, and 18. Finally, the frame ends with CRC-32 checksum `crc32` = `0xBA6B3899u`.

4.2 Requesting Frames

Requesting a single frame from the sensor is quite straightforward. First, the sensor must be in the `MKE_STATE_DEPTH_SENSOR` state. Then, to a request of type `MKE_REQUEST_GET_FRAME`, see Section 6.1.12, the sensor responds with a single reply data packet. The reply status `MKE_REPLY_OK` determines that this data packet carries a valid frame.

To receive a stream of frames from the sensor, one can simply place the `MKE_REQUEST_GET_FRAME` request call into a loop. Since the sensor will not respond to `MKE_REQUEST_GET_FRAME` request *before* a new frame is available, it is guarantied that a client that waits for a sensor's reply to such a call before sending another request will never receive the same frame twice. On the other hand, if the client does not request the next frame in time, maybe because it spent too much time processing the current one, it is possible it will miss the chance to receive it and it will receive a frame after that instead. To detect such situations, differences of the respective fields `timer` and `seqn` from subsequent `MkEReply_Frame's can be used.

Another way to receive a stream of frames is via the `MKE_REQUEST_START_FRAME_PUSH` request. Once the sensor receives this request, it will automatically send every subsequent available frame. To tell the sensor to stop the frame stream, the client must send the `MKE_REQUEST_STOP_FRAME_PUSH` request. The following diagram explains this communication scheme in more detail:

```
Client's Request                Sensor's Reply
------------------------------------------------------------------
MKE_REQUEST_START_FRAME_PUSH
  (reqid = 1)
                                MKE_REPLY_DATA_WILL_START
                                  (reqid = 1)
                                MKE_REPLY_DATA_WILL_CONTINUE
                                  (reqid = 1, frame data)
                                ...
                                MKE_REPLY_DATA_WILL_CONTINUE
                                  (reqid = 1, frame data)
MKE_REQUEST_STOP_FRAME_PUSH
  (reqid = 2)
                                ...
                                MKE_REPLY_DATA_WILL_CONTINUE
                                  (reqid = 1, frame data)
                                MKE_REPLY_OK
                                  (reqid = 2)
                                MKE_REPLY_DATA_STOPPED
                                  (reqid = 1)
------------------------------------------------------------------
```

First, the client elicits the frame stream via `MKE_REQUEST_START_FRAME_PUSH` request. If the sensor is ready to start sending the frame stream, it will reply with the `MKE_REPLY_DATA_WILL_START` data packet along with the corresponding `reqid` value. This data packet will not yet contain any frame data. After this initial reply, the sensor will start sending every available frame in a separate reply data packet with reply type `MKE_REPLY_DATA_WILL_CONTINUE`. Once the client decides to stop the frame stream, it issues the `MKE_REQUEST_STOP_FRAME_PUSH` request. After this request, the client may still receive one or more `MKE_REPLY_DATA_WILL_CONTINUE` replies with frame data, as some may have already been send before the sensor received the `MKE_REQUEST_STOP_FRAME_PUSH` request. The fact that the stream was stopped is signaled by the `MKE_REPLY_DATA_STOPPED` reply. Once this reply is received, it is guaranteed that no more `MKE_REPLY_DATA_WILL_CONTINUE` requests will follow. Finally, however, not necessarily in this order, the sensor will also receive a `MKE_REPLY_OK` reply with `reqid` value corresponding to the `MKE_REQUEST_STOP_FRAME_PUSH` request.

5 Sensor Policies

The MkE API does not allow the client to change the specific sensor parameters, such as exposure time, gain, etc. However, it provides a concept of *policies*. Policies are sets of predefined sensor parameters that are available to the client to chose from and set. The actual parameters are not exposed to the client via the MkE API. The policies are accessible via their names, *e.g.*, 'SUNLIGHT', 'INDOORS', *etc*. These names must not be longer than 8 ASCII characters and are sensor specific.

The current policy can be queried via the `MKE_REQUEST_GET_POLICY` request, see Section 6.1.7. The sensor policy can be changed via the `MKE_REQUEST_SET_POLICY` request, see Section 6.1.8. The list of the available sensor policies can be queried using the `MKE_REQUEST_LIST_POLICIES` request, see Section 6.1.9.

6 Reference

6.1 Request Type

The following table lists the valid client requests (request types) and the respective numerical codes:

Request type	*Numerical code*
MKE_REQUEST_TERMINATE	10
MKE_REQUEST_GET_FIRMWARE_INFO	11
MKE_REQUEST_GET_DEVICE_INFO	12
MKE_REQUEST_GET_DEVICE_XML	13
MKE_REQUEST_GET_STATE	20
MKE_REQUEST_SET_STATE	21
MKE_REQUEST_GET_POLICY	22
MKE_REQUEST_SET_POLICY	23
MKE_REQUEST_START_FRAME_PUSH	24
MKE_REQUEST_STOP_FRAME_PUSH	25
MKE_REQUEST_GET_FRAME	26
MKE_REQUEST_LIST_POLICIES	27
MKE_REQUEST_UPLOAD_PACKAGE	2001

6.1.1 MKE_REQUEST_TERMINATE

MKE_REQUEST_TERMINATE request is used to shutdown or reboot the sensor programatically. The following C-style structure MkERequest_Terminate describes the inner structure of request field params:

```
struct MkERequest_Terminate {
  uint32_t  method;
  uint8_t   undefined[4];
};
```

The MkERequest_Terminate structure has only one valid field with the following meaning:

Field	*Definition*
uint32_t method	Method of termination, valid methods are MKE_TERMINATE_BY_REBOOT (Numerical code 1) and MKE_TERMINATE_BY_SHUTDOWN (Numerical code 2).

The sensor will reply with MKE_REPLY_OK status in the case it registered the request and is on its way to perform the termination action. Otherwise, the sensor will reply with an error reply status, see Section 6.2.

6.1.2 MKE_REQUEST_GET_FIRMWARE_INFO

Use MKE_REQUEST_GET_FIRMWARE_INFO to query the sensor's firmware version and various other information. The MKE_REQUEST_GET_FIRMWARE_INFO request has no parameters. This means that the bytes of the request's params field should be set to zero. The following C-style structure MkEReply_FirmwareInfo describes the inner structure of MkEReply field params:

```
struct MkEReply_FirmwareInfo {
  int64_t   posix_time;
  uint32_t  git_commit;
  uint8_t   rt_ver_major;
  uint8_t   rt_ver_minor;
  uint8_t   rt_ver_patch;
  uint8_t   fw_ver_major;
  uint8_t   fw_ver_minor;
  uint8_t   fw_ver_patch;
  char      undefined[6];
};
```

The `MkEReply_FirmwareInfo` structure has 8 valid fields with the following meaning:

Field	*Definition*
`int64_t posix_time`	POSIX time in the time of firmware compilation.
`uint32_t git_commit`	Short git hash of the latest firmware commit.
`uint8_t rt_ver_major`	Runtime version - major part
`uint8_t rt_ver_minor`	Runtime version - minor part
`uint8_t rt_ver_patch`	Runtime version - patch part
`uint8_t fw_ver_major`	Firmware version - major part
`uint8_t fw_ver_minor`	Firmware version - minor part
`uint8_t fw_ver_patch`	Firmware version - patch part

In the case of success, the sensor will reply with `MKE_REPLY_OK` status. Otherwise, the sensor will reply with an error reply status, see Section 6.2.

6.1.3 MKE_REQUEST_GET_DEVICE_INFO

Use `MKE_REQUEST_GET_DEVICE_INFO` to query the sensor's ID information. The `MKE_REQUEST_GET_DEVICE_INFO` request has no parameters. This means that the bytes of the request's `params` field should be set to zero. The following C-style structure `MkEReply_DeviceInfo` describes the inner structure of `MkEReply` field `params`:

```
struct MkEReply_DeviceInfo {
  uint16_t  device_id;
  char      unit_id[8];
  char      undefined[14];
};
```

The `MkEReply_DeviceInfo` structure has 2 valid fields with the following meaning:

Field	*Definition*
`uint16_t device_id`	Identification code of the device. This value is shared with other devices of the same model.
`char unit_id[8]`	Serial number of the device.

In the case of success, the sensor will reply with `MKE_REPLY_OK` status. Otherwise, the sensor will reply with an error reply status, see Section 6.2.

6.1.4 MKE_REQUEST_GET_DEVICE_XML

Use MKE_REQUEST_GET_DEVICE_XML to query the sensor's ID information in the XML file format. This XML file is sensor specific and its content is not part of the MkE API.

6.1.5 MKE_REQUEST_GET_STATE

Use MKE_REQUEST_GET_STATE to query the sensor's current state. There are two states the sensor can be in. The following C-style enum lists these states and their respective numerical codes:

```
enum MkEStateType {
  MKE_STATE_IDLE = 1,
  MKE_STATE_DEPTH_SENSOR = 2,
};
```

The MKE_REQUEST_GET_STATE request has no parameters. This means that the bytes of the request's params field should be set to zero. The sensor will respond with MkEReply where num_bytes = 0. The following C-style structure MkEReply_State describes the inner structure of MkEReply field params:

```
struct MkEReply_State {
  uint32_t  state;
  char      undefined[20];
};
```

The MkEReply_State structure has only one valid field with the following meaning:

Field	*Definition*
uint32_t state	The current sensor's state as a valid value of MkEStateType.

In the case of success, the sensor will reply with MKE_REPLY_OK status. Otherwise, the sensor will reply with an error reply status, see Section 6.2.

6.1.6 MKE_REQUEST_SET_STATE

Use MKE_REQUEST_SET_STATE to change the sensor's current state. The following C-style structure MkERequest_SetState describes the inner structure of request field params:

```
struct MkERequest_SetState {
  uint32_t  new_state;
  uint8_t   undefined[4];
};
```

The MkERequest_SetState structure has only one valid field with the following meaning:

Field	*Definition*
uint32_t new_state	The new sensor's state as a valid value of MkEStateType, see Section 6.1.5.

The sensor will respond with MkEReply where num_bytes = 0. The reply field status will be set to MKE_REPLY_OK in the case the state has been successfully changed. The sensor will reply with MKE_REPLY_CLIENT_REQUEST_DOES_NOT_APPLY in the case the requested state is identical to the current state.

6.1.7 MKE_REQUEST_GET_POLICY

Use MKE_REQUEST_GET_POLICY to query the current sensor policy, see Section 5. The MKE_REQUEST_GET_POLICY request has no parameters. This means that the bytes of the request's params field should be set to zero. The following C-style structure MkEReply_GetPolicy describes the inner structure of MkEReply field params:

```
struct MkEReply_GetPolicy {
  char       profile_name[8];
  char       undefined[16];
};
```

The MkEReply_GetPolicy structure has one valid field with the following meaning:

Field	*Definition*
char policy_name[8]	Name of the current policy as a C-style string, *i.e.*, zero terminated string. In the case the name is exactly 8 characters long, the terminating zero character is not added.

In the case of success, the sensor will reply with MKE_REPLY_OK status. Otherwise, the sensor will reply with an error reply status, see Section 6.2.

6.1.8 MKE_REQUEST_SET_POLICY

Use MKE_REQUEST_SET_POLICY to set the sensor policy, see Section 5. The following C-style structure MkERequest_SetPolicy describes the inner structure of request field params:

```
struct MkERequest_SetPolicy {
  char       policy_name[8];
};
```

The MkERequest_SetPolicy structure has only one valid field with the following meaning:

Field	*Definition*
char policy_name[8]	Name of the policy to set as a C-style string, *i.e.*, zero terminated string. In the case the name is exactly 8 characters long, the terminating zero character is not added. Use MKE_REQUEST_LIST_POLICIES to query the available policies, see Section 6.1.9.

In the case of success, the sensor will reply with MKE_REPLY_OK status. Otherwise, the sensor will reply with an error reply status, see Section 6.2.

6.1.9 MKE_REQUEST_LIST_POLICIES

Use MKE_REQUEST_LIST_POLICIES to list the available policies. The MKE_REQUEST_LIST_POLICIES request has no parameters, *i.e.*, the bytes of the request's params field should be set to zero.

The following C-style structure MkEReply_ListPolicies describes the inner structure of MkEReply field params:

```
struct MkEReply_ListPolicies {
 uint32_t  num_policies;
 char      undefined[20];
};
```

The MkEReply_ListPolicies structure has only one valid field with the following meaning:

Field	*Definition*
uint32_t num_policies	The number of available sensor specific policies.

The actual names of the policies are provided via the reply's data payload signaled by non-zero reply field num_bytes. This data payload will contain a zero-byte separated list of num_policies policy names.

In the case of success, the sensor will reply with MKE_REPLY_OK status. Otherwise, the sensor will reply with an error reply status, see Section 6.2.

6.1.10 MKE_REQUEST_START_FRAME_PUSH

Use MKE_REQUEST_START_FRAME_PUSH type to elicit frame stream from the sensor. The structure of the request's params field is described by the C-style structure MkERequest_GetFrame, see Section 6.1.12.

If the sensor is ready to start sending the frame stream, it will reply with MKE_REPLY_DATA_WILL_START data packet with the corresponding reqid value. This data packet will not yet contain any frame data. After this initial reply, the sensor will start sending every available frame in a separate reply data packet with reply type MKE_REPLY_DATA_WILL_CONTINUE. These data packets will contain 3D data frames, see Section 4.1.

The frame stream will end with a MKE_REPLY_DATA_STOPPED reply, if it has been correctly stopped by a MKE_REQUEST_STOP_FRAME_PUSH request. Otherwise it will stop with an error reply.

6.1.11 MKE_REQUEST_STOP_FRAME_PUSH

Use MKE_REQUEST_STOP_FRAME_PUSH type to stop the frame stream elicited by previous MKE_REQUEST_START_FRAME_PUSH request. MKE_REQUEST_STOP_FRAME_PUSH request has no parameters. This means that the bytes of the request's params field should be set to zero. The sensor will respond with MkEReply where num_bytes = 0. If successful, the sensor will reply with MKE_REPLY_OK status. Otherwise, the sensor will reply with an error reply status, see Section 6.2.

6.1.12 MKE_REQUEST_GET_FRAME

Use MKE_REQUEST_GET_FRAME type to request a single frame from the sensor. The sensor can request frame items of two types. The following C-style enum lists their respective numerical codes:

```
enum MkEFrameType {
  MKE_FRAME_TYPE_1 = 1,
  MKE_FRAME_TYPE_2 = 2,
};
```

The following C-style structure `MkERequest_GetFrame` describes the inner structure of request field `params`:

```
struct MkERequest_GetFrame {
  uint16_t   frame_type;
  uint8_t    undefined[6];
};
```

The `MkERequest_GetFrame` structure has only one valid field with the following meaning:

Field	*Definition*
`uint16_t frame_type`	The requested frame type as a valid value of `MkEFrameType`.

The sensor will not respond to `MKE_REQUEST_GET_FRAME` request until a new frame is available. If successful, the sensor will reply with the `MKE_REPLY_OK` status. Otherwise, the sensor will reply with an error reply status, see Section 6.2.

6.1.13 MKE_REQUEST_UPLOAD_PACKAGE

The `MKE_REQUEST_UPLOAD_PACKAGE` request is used to upload general variable-sized data to the sensor. The format and interpretation of the uploaded data is not part of the MkE API. In practice, this request is used to upload firmware binary package or other updates. The size of the uploaded data may be limited by the sensor. The `MKE_REQUEST_UPLOAD_PACKAGE` request is available only in the `MKE_STATE_IDLE` state.

The following C-style structure `MkERequest_UploadPackage` describes the inner structure of request field `params`:

```
struct MkERequest_UploadPackage {
  uint32_t   payload_size;
  uint32_t   crc32;
};
```

The `MkERequest_UploadPackage` structure has two valid fields with the following meaning:

Field	*Definition*
`uint32_t payload_size`	Size of the data payload in bytes
`uint32_t crc32`	CRC-32 checksum (ITU-T V.42) of the data payload

The `MKE_REQUEST_UPLOAD_PACKAGE` request must be immediately followed by `payload_size` bytes of the payload.

If successful, the sensor will reply with the `MKE_REPLY_OK` status. In the case the payload size surpasses the request payload size limit of the server, the `MKE_REPLY_CLIENT_REQUEST_PAYLOAD_TOO_LONG` will be returned. If the sensor fails to validate the CRC-32 checksum, the MKE_REPLY_CLIENT_MALFORMED_REQUEST reply will be returned. Otherwise, the sensor will reply with MKE_REPLY_SERVER_ERROR error reply status.

6.2 Reply Status

The sensor replies are conceptually divided into four classes:

1xx Replies with numerical codes 1xx are reserved for frame stream replies, see Section 4.2.

2xx Replies with numerical codes 2xx signalize successful completion of the reply.

4xx Replies with numerical codes 4xx signalize client side errors.

5xx Replies with numerical codes 5xx signalize server side errors.

The following table lists the valid sensor reply statuses and the respective numerical codes:

Reply status	*Numerical code*
MKE_REPLY_DATA_WILL_START	100
MKE_REPLY_DATA_WILL_CONTINUE	101
MKE_REPLY_DATA_STOPPED	102
MKE_REPLY_OK	200
MKE_REPLY_CLIENT_ERROR	400
MKE_REPLY_CLIENT_MALFORMED_REQUEST	401
MKE_REPLY_CLIENT_ILLEGAL_REQUEST_TYPE	402
MKE_REPLY_CLIENT_REQUEST_DOES_NOT_APPLY	403
MKE_REPLY_CLIENT_REQUEST_PAYLOAD_TOO_LONG	404
MKE_REPLY_SERVER_ERROR	500
MKE_REPLY_SERVER_REQUEST_INTERRUPTED	501
MKE_REPLY_SERVER_BUSY	502
MKE_REPLY_SERVER_INSUFFICIENT_RESOURCES	503
MKE_REPLY_SERVER_FATAL_ERROR	504

6.2.1 MKE_REPLY_DATA_WILL_START

The MKE_REPLY_DATA_WILL_START reply signals the successful initialization of the frame streaming process. The data packet does not yet contain any data. At least one more reply will follow.

6.2.2 MKE_REPLY_DATA_WILL_CONTINUE

The MKE_REPLY_DATA_WILL_CONTINUE reply signals that the frame stream will continue with at least one more data packet. At the same time the data payload of the reply contains the 3D frame data.

6.2.3 MKE_REPLY_DATA_STOPPED

The MKE_REPLY_DATA_STOPPED reply signalizes that the frame stream has been successfully stopped via MKE_REQUEST_STOP_FRAME_PUSH request. This data packet does not contain any data payload. No more data packets pertinent to this frame stream will follow.

6.2.4 MKE_REPLY_OK

The MKE_REPLY_OK reply status signals that a request has been successfully handled. No more data packet will follow MKE_REPLY_OK reply.

6.2.5 MKE_REPLY_CLIENT_ERROR

The MKE_REPLY_CLIENT_ERROR reply signals a general client side error.

6.2.6 MKE_REPLY_CLIENT_MALFORMED_REQUEST

The MKE_REPLY_CLIENT_MALFORMED_REQUEST reply signals a sensor's problem with parsing a request. For example, the field magik of MkERequest, see Section 3.3, does not contain string MKERQ100, or a request parameters are out of bounds.

6.2.7 MKE_REPLY_CLIENT_ILLEGAL_REQUEST_TYPE

The MKE_REPLY_CLIENT_ILLEGAL_REQUEST_TYPE reply signals that the client issued a request type not available in the current state. Figure 1 lists the available sensor states and pertinent requests types.

6.2.8 MKE_REPLY_CLIENT_REQUEST_DOES_NOT_APPLY

The MKE_REPLY_CLIENT_REQUEST_DOES_NOT_APPLY reply signals a situation where a client requested resources that were not available in the sensor's current state. For example, the client issued MKE_REQUEST_GET_FRAME request while in MKE_STATE_IDLE state.

6.3 MKE_REPLY_CLIENT_REQUEST_PAYLOAD_TOO_LONG

The MKE_REPLY_CLIENT_REQUEST_PAYLOAD_TOO_LONG reply signals that the MKE_REQUEST_UPLOAD_PACKAGE request payload size surpassed the maximum request payload size allowed by the server.

6.3.1 MKE_REPLY_SERVER_ERROR

The MKE_REPLY_SERVER_ERROR reply signals a general sensor side error.

6.3.2 MKE_SERVER_REQUEST_INTERRUPTED

The MKE_REPLY_SERVER_REQUEST_INTERRUPTED reply signals that a sensor's work on a reply has been externally interrupted. For example, the client requested a state change from MKE_STATE_DEPTH_SENSOR to MKE_STATE_IDLE, but did not correctly stop an ongoing frame stream via MKE_REQUEST_STOP_FRAME_PUSH.

6.3.3 MKE_REPLY_SERVER_BUSY

The sensor will issue the MKE_REPLY_SERVER_BUSY reply in situations where the client requests an operation that is already being processed by the sensor. For example, the client issued two MKE_REQUEST_START_FRAME_PUSH requests without correctly stopping the first one via MKE_REQUEST_STOP_FRAME_PUSH.

6.3.4 MKE_REPLY_INSUFFICIENT_RESOURCES

The MKE_REPLY_INSUFFICIENT_RESOURCES reply signals a fatal problem with memory resources on the sensor's side. This reply data packet is "read-only", i.e., the sensor is unable to set reqid and status fields, see Section 3.4, before sending the reply data packet. This problem might be caused, for example, by quickly sending several requests without waiting for the sensor's reply. The inner request queue is limited in length and the sensor will issue MKE_REPLY_INSUFFICIENT_RESOURCES reply in the case it is full. Once the sensor has processed any outstanding items in this queue, it will be ready to receive new requests, again.

6.3.5 MKE_REPLY_SERVER_FATAL_ERROR

The MKE_REPLY_SERVER_FATAL_ERROR reply signals a fatal problem encountered during the sensor startup and runtime. This problem may have been caused by hardware issues or by an unsuccessful firmware update. The MKE_REPLY_SERVER_FATAL_ERROR reply contains another supplementary error code specifying the problem. The following C-style structure MkEReply_ServerFatal describes the inner structure of MkEReply field params:

```
struct MkEReply_ServerFatal {
  uint32_t   err_code;
  char       undefined[20];
};
```

The MkEReply_ServerFatal structure has only one valid field:

Field	*Definition*
uint32_t err_code	Supplementary error code specifying the problem as a valid value of MkEFatalErrorType.

The following C-style MkEFatalErrorType enum lists possible values of err_code:

```
enum MkEFatalErrorType {
  MKE_FATAL_UNDEF                = 0,
  MKE_FATAL_BADCONFIG            = 1,
  MKE_FATAL_DETECTORINIT         = 2,
  MKE_FATAL_BADCAMERA            = 3,
  MKE_FATAL_RUNTIME              = 4,
};
```

The meaning of the supplementary error err_code is listed in the following table:

Enum string	*Numerical code*	*Meaning*
`MKE_FATAL_UNDEF`	0	Unspecified fatal error
`MKE_FATAL_BADCONFIG`	1	Corrupted device configuration
`MKE_FATAL_DETECTORINIT`	2	Unable to initialize the detector
`MKE_FATAL_BADCAMERA`	3	The device has encountered a problem with the camera connection
`MKE_FATAL_RUNTIME`	4	Unspecified fatal error during runtime