Erae API - V2

The ERAE API is a custom sysex library with messages enabling you to take full control of the ERAE product point detection and LEDs states. Each message is formatted as described below.

API Messages

Common definitions

Erae identifier prefix

The API V2 is designed to be compatible with different Erae products. The different products have different Erae Identifier as described in the following table:

Erae identifier prefix	**Product**
`0x00 0x21 0x50 0x00 0x01 0x00 0x02`	Erae 2
`0x00 0x21 0x50 0x00 0x01 0x00 0x01`	Erae Touch

Erae current MIDI network ID

Currently this is not used and it is always set to 0x01. It might be used in the future to select the Erae device in case of multiple Erae devices on the same MIDI chain.

Receiver prefix identifier

The receiver prefix is an arbitrary set of bytes you can choose when enabling the API mode. It is helpful when parsing the received sysex messages on the host side.

API zone identifier

The API provides messages to control the API zone elements. You can add API zone elements to your layout with the Erae Lab. Each API zone element has an API zone Identifier (0x00 to 0x7F), this identifier is used to control the different API Zone elements.

API Version request

Full sysex message:

```
0xF0 ERAE IDENTIFIER PREFIX 0x01 0x01 0x04 0x7F RECEIVER PREFIX BYTES 0xF7
```

Message break down:

`0xF0`	SysEx message begin
`ERAE IDENTIFIER PREFIX`	Erae identifier prefix
`0x01`	Erae current MIDI network ID
`0x01`	Erae Service
`0x04`	Erae [Service] API

0x7F	Erae [Service API] version request
BBs	Receiver Prefix Bytes
0xF0	SysEx message end

The reply to the API version reply message will begin with the chosen ‘Receiver Prefix Bytes’. It must be set at least to 1 byte and max 16 bytes.

API Version reply

Full sysex message:

```
0xF0 RECEIVER PREFIX BYTES 0x7F 0x02 VERSION 0xF7
```

Message break down:

0xF0	SysEx message begin
BBs	Receiver Prefix Bytes
0x7F	Non finger data byte
0x02	API Version reply
VERSION	version (1 byte)
0xF7	SysEx message end

The API Version reply message will begin with the ‘Receiver Prefix Bytes’ sent with the API Version request message.

API Mode enable

Full sysex message:

```
0xF0 ERAE IDENTIFIER PREFIX 0x01 0x01 0x04 0x01 RECEIVER PREFIX BYTES 0xF7
```

Message break down:

0xF0	SysEx message begin
ERAE IDENTIFIER PREFIX	Erae identifier prefix
0x01	Erae current MIDI network ID
0x01	Erae Service
0x04	Erae [Service] API
0x01	Erae [Service API] enable
BBs	Receiver Prefix Bytes
0xF0	SysEx message end

Messages sent by the Erae API to the receiver will begin with the chosen ‘Receiver Prefix Bytes’. You must set at least 1 byte and max 16 bytes.

Please send an API Mode disabled message before sending a new API Mode enable message.

API Mode disable

Full sysex message:

```
0xF0 ERAE IDENTIFIER PREFIX 0x01 0x01 0x04 0x02 0xF7
```

Message break down:

0xF0	SysEx message begin
ERAE IDENTIFIER PREFIX	Erae identifier prefix
0x01	Erae current MIDI network ID
0x01	Erae Service
0x04	Erae [Service] API
0x02	Erae [Service API] disable
0xF0	SysEx message end

API Fingerstream message

This message is sent by the Erae to your receiving device when the API mode is enabled. Messages are sent only when touching an API Zone.

Full sysex message:

```
0xF0 RECEIVER PREFIX BYTES ACTION ZONE_ID FIN1 … FIN10 XYZ1 ... XYZ14 CHKS 0xF7
```

Message break down:

0xF0	SysEx message begin
BBs	Receiver Prefix Bytes
ACTION	Action Index bytes
ZONE_ID	API zone identifier (1 byte)
FIN1 … FIN10	7-bitized uint64 Finger ID
XYZ1 … XYZ14	7-bitized 3 floats (X, Y, Z) position
CHK	Checksum (XOR) of the encoded XYZ bytes
0xF7	SysEx message end

The receiver prefix bytes are the ones given in the API Mode enable message.

Please refer to the 7-bit-izing section for more information about the unpacking logic for Finger ID and Finger position data.

Zone Boundary Request message

Full sysex message:

```
0xF0 ERAE IDENTIFIER PREFIX 0x01 0x01 0x04 0x10 ZONE_ID 0xF7
```

Message break down:

0xF0	SysEx message begin
ERAE IDENTIFIER PREFIX	Erae identifier prefix
0x01	Erae current MIDI network ID
0x01	Erae Service
0x04	Erae [Service] API
0x10	Erae [Service API] boundary request
ZONE_ID	API zone identifier (1 byte)
0xF7	SysEx message end

Zone Boundary Reply message

The API mode must be enabled for this message to be sent by the Erae to your receiving device.

Full sysex message:

```
0xF0 RECEIVER PREFIX BYTES 0x7F 0x01 ZONE_ID WIDTH HEIGHT 0xF7
```

Message break down:

0xF0	SysEx message begin
BBs	Receiver Prefix Bytes
0x7F	Non finger data byte
0x01	Zone boundary reply byte
ZONE_ID	API zone identifier (1 byte)
WIDTH HEIGHT	Width & height of the zone (2 bytes)
0xF7	SysEx message end

The returned size of the zone is (7F, 7F) if the zone is not used in the Erae.

Clear Zone Display

Full sysex message:

```
0xF0 ERAE IDENTIFIER PREFIX 0x01 0x01 0x04 0x20 ZONE_ID 0xF7
```

Message break down:

0xF0	SysEx message begin
ERAE IDENTIFIER PREFIX	Erae identifier prefix
0x01	Erae current MIDI network ID
0x01	Erae Service
0x04	Erae [Service] API
0x20	Erae [Service API] clear zone
ZONE_ID	API zone identifier (1 byte)
0xF7	SysEx message end

Draw pixel

Full sysex message:

```
0xF0 ERAE IDENTIFIER PREFIX 0x01 0x01 0x04 0x22 ZONE_ID XPOS YPOS WIDTH HEIGHT RED GREEN BLUE 0xF7
```

Message break down:

0xF0	SysEx message begin
ERAE IDENTIFIER PREFIX	Erae identifier prefix
0x01	Erae current MIDI network ID
0x01	Erae Service
0x04	Erae [Service] API
0x21	Erae [Service API] draw pixel
ZONE_ID	API zone identifier (1 byte)
XPOS YPOS	Coordinates of the pixel (2 bytes)
RGB	Color of the pixel (3 7-bits bytes)
0xF7	SysEx message end

Draw rectangle

Full sysex message:

```
0xF0 ERAE IDENTIFIER PREFIX 0x01 0x01 0x04 0x22 ZONE_ID XPOS YPOS WIDTH HEIGHT RED GREEN BLUE 0xF7
```

Message break down:

0xF0	SysEx message begin

ERAE IDENTIFIER PREFIX	Erae identifier prefix
0x01	Erae current MIDI network ID
0x01	Erae Service
0x04	Erae [Service] API
0x22	Erae [Service API] draw rectangle
ZONE_ID	API zone identifier (1 byte)
XPOS YPOS	Coordinates of the pixel (2 bytes)
WIDTH HEIGHT	Width and Height (2 bytes)
RGB	Color of the pixel (3 7-bits bytes)
0xF7	SysEx message end

Draw image

Full sysex message:

```
0xF0 ERAE IDENTIFIER PREFIX 0x01 0x01 0x04 0x23 ZONE_ID XPOS YPOS WIDTH HEIGHT RED GREEN BLUE 0xF7
```

Message break down:

0xF0	SysEx message begin
ERAE IDENTIFIER PREFIX	Erae identifier prefix
0x01	Erae current MIDI network ID
0x01	Erae Service
0x04	Erae [Service] API
0x23	Erae [Service API] draw rectangle
ZONE_ID	API zone identifier (1 byte)
XY	Coordinates of the pixel (2 bytes)
WH	Width and Height (2 bytes)
BIN	7-bitized 24 bits RGB data of the pixels
CHK	Checksum (XOR) of all the ‘BIN’ bytes
0xF7	SysEx message end

When displaying a large image, it will be best to break down the image into subimages with no more than 32 pixels each. This keeps the messages short enough for it to be managed by your operating system properly.

Please refer to the 7-bit-izing section for more information about the packing / unpacking logic of the RGB data.

Example:

Draw an image on API Zone 1 at location (bottom left) x = 5, y = 3 of size, width = 2, height = 2. We want to send 24 bit rgb data “white, red, green, blue” (0xFFFFFF, 0xFF0000, 0x00FF00, 0x0000FF) from left to right and bottom to top. The RGB data to be bitized is

```
0xFF 0xFF 0xFF 0xFF 0x00 0x00 0x00 0xFF 0x00 0x00 0x00 0xFF
```

Full sysex message for Erae 2:

```
0xF0 0x00 0x21 0x50 0x00 0x01 0x00 0x02 0x01 0x01 0x04 0x23 0x01 0x05 0x03 0x02 0x02 0x78 0x7F 0x7F 0x7F 0x7F
0x00 0x00 0x00 0x44 0x7F 0x00 0x00 0x00 0x7F 0x3C 0xF7
```

Message break down

SysEx message begin	`0xF0`
Erae Identifier Prefix for Erae 2	`0x00 0x21 0x50 0x00 0x01 0x00 0x02`
Draw an image	`0x01 0x01 0x04 0x23`
API zone identifier (1 byte)	`0x01`
Position x = 5, y = 3	`0x05 0x03`
Width = 2, height = 2	`0x02 0x02`
7bitized 24 bits RGB data	`0x78 0x7F 0x7F 0x7F 0x7F 0x00 0x00 0x00 0x44 0x7F 0x00 0x00 0x00 0x7F`
Checksum of the bitized data	`0x3C`
SysEx message end	`0xF7`

7-bit-izing logic

MIDI requires all data bytes to be 7-bit values (0–127). To safely transmit full 8-bit binary data, we use a technique called 7-bit encoding with MSB extraction (commonly used in SysEx messages).

For every **7 bytes of original 8-bit data**, we create an **8th byte** that stores the **most significant bit (MSB)** of each of those 7 bytes. This lets us "hide" the MSBs in a separate byte, so all other transmitted bytes are guaranteed to be 7-bit-safe.

Encoding (7-Bitizing):

- Take 7 bytes of 8-bit data.
- Extract the MSB (bit 7) from each byte.
- Pack those 7 MSBs into one new byte — this becomes the prefix byte.
- Clear the MSB of each original byte (mask with 0x7F) and send them.
- Transmit the prefix byte before or after the group (based on protocol).

Decoding (Reversing the Process):

- Read the prefix byte and extract its 7 bits.
- For each data byte, restore its MSB using the bits from the prefix.
- Combine the MSB and 7-bit value to reconstruct the original 8-bit byte.

Example:

```
Unset
Original 8-bit block: [0x95, 0xD3, 0x82, 0xFF, 0x74, 0x8A, 0x61]

MSBs:               [1,    1,     1,     1,     0,     1,     0] → 0b1111010 =
0x7A

7-bitized output:
  Prefix byte:      0x7A
  Data bytes:       [0x15, 0x53, 0x02, 0x7F, 0x74, 0x0A, 0x61]

Full 7-bitized:     [0x7A, 0x15, 0x53, 0x02, 0x7F, 0x74, 0x0A, 0x61]
```

Helper code

Python

Python

```python
from functools import reduce

def bitized7size(length: int) -> int:
	# Get size of the resulting 7 bits bytes array obtained when using the bitize7 function
	return length // 7 * 8 + ((1 + length % 7) if (length % 7 > 0) else 0)


def unbitized7size(length: int) -> int:
	# Get size of the resulting 7 bits bytes array obtained when using the bitize7 function
	return length // 8 * 7 + ((length % 8 - 1) if (length % 8 > 0) else 0)


def checksum(data: list[int]) -> int:
	return reduce(lambda x, y: x ^ y, data)


def bitize7chksum(data: list[int], append_checksum: bool = True) -> list[int]:
	# 7-bitize an array of bytes and add the resulting checksum
	bitized_arr = sum(([sum((el & 0x80) >> (j+1) for j, el in enumerate(data[i:min(i+7, len(data))]))] + [el & 0x7F for el in data[i:min(i+7, len(data))]] for i in range(0, len(data), 7)), [])

	if append_checksum:
		return bitized_arr + [checksum(bitized_arr)]
	else:
		return bitized_arr


def unbitize7chksum(bitized_data: list[int], checksum: int = None) -> list[int]:
	# Rebuild the 8-bit data from the 7-bit segments
	original_data = [0] * unbitized7size(len(bitized_data))

	i = 0
	outsize = 0
	inlen = len(bitized_data)
	while i < len(bitized_data):
		for j in range(7):
```

```
                if (j + 1 + i < inlen):
                        original_data[outsize + j] = ((bitized_data[i] <<
      (j + 1)) & 0x80) | bitized_data[i + j + 1]
                outsize = outsize+7
                i = i + 8

      # Validate checksum (should match the XOR of all the bytes in the
bitized data)
      if checksum is not None:
            calculated_checksum = checksum(bitized_data)
      if checksum != calculated_checksum:
            print("Warning: Checksum mismatch! {} {}".format(
            checksum, calculated_checksum))

      return original_data
```

C++

C/C++

```
#include <cstdint>
#include <cstddef>

/**
* @brief Get size of the resulting 7 bits bytes array obtained when using
the bitize7 function
*/
constexpr size_t bitized7size(size_t len)
{
 return len / 7 * 8 + (len % 7 ? 1 + len % 7 : 0);
}

/**
* @brief Get size of the resulting 8 bits bytes array obtained when using
the unbitize7 function
*/
constexpr size_t unbitized7size(size_t len)
{
 return len / 8 * 7 + (len % 8 ? len % 8 - 1 : 0);
}

/**
* @brief 7-bitize an array of bytes and get the resulting checksum
*
* @param in Input array of 8 bits bytes
```

```
* @param inlen Length in bytes of the input array of 8 bits bytes
* @param out An output array of bytes that will receive the 7-bitized bytes
* @return the output 7-bitized bytes XOR checksum
*/
constexpr uint8_t bitize7chksum(const uint8_t* in, size_t inlen, uint8_t*
out)
{
 uint8_t chksum = 0;
 for (size_t i{0}, outsize{0}; i < inlen; i += 7, outsize += 8)
 {
   out[outsize] = 0;
   for (size_t j = 0; (j < 7) && (i + j < inlen); ++j)
   {
     out[outsize] |= (in[i + j] & 0x80) >> (j + 1);
     out[outsize + j + 1] = in[i + j] & 0x7F;
     chksum ^= out[outsize + j + 1];
   }
   chksum ^= out[outsize];
 }
 return chksum;
}

/**
* @brief 7-unbitize an array of bytes and get the incomming checksum
*
* @param in Input array of 7 bits bytes
* @param inlen Length in bytes of the input array of 7 bits bytes
* @param out An output array of bytes that will receive the 7-unbitized
bytes
* @return the input 7-bitized bytes XOR checksum
*/
constexpr uint8_t unbitize7chksum(const uint8_t* in, size_t inlen, uint8_t*
out)
{
 uint8_t chksum = 0;
 for (size_t i{0}, outsize{0}; i < inlen; i += 8, outsize += 7)
 {
   chksum ^= in[i];
   for (size_t j = 0; (j < 7) && (j + 1 + i < inlen); ++j)
   {
     out[outsize + j] = ((in[i] << (j + 1)) & 0x80) | in[i + j + 1];
     chksum ^= in[i + j + 1];
   }
 }
 return chksum;
}
```