celepixel

CelePixel

CeleX-5 Chipset

SDK Reference

CelePixel Technology Co. Ltd.

Content

Version Control

Version	Date	Section	Description	Author
1.0	2018.11.08	All	New	Xiaoqin Hu
1.1	2019.05.21	All	Update	Xiaoqin Hu

1. Overview

1.1. Introduction of CeleX-5 Chipset

1.1.1 Basic working principle of CeleX-5 Chipset

The CeleX-5 chipset has two modes of operation: parallel port output data mode and MIPI serial port output data mode. This way of outputting data through the parallel port is the same as that of the CeleX-4 chipset, that is, the data of the Sensor is transmitted through an Opal Kelly FPGA processing board, and will not be described here.

This SDK uses the MIPI serial output mode as shown in Fig. 1-1, in which shows the basic working principle of the CeleX-5 Chipset. It needs a driver to convert MIPI data into USB3.0 data, then the PC applications could obtain the data from the USB3.0 Driver. Similarly, the PC can also configure the Sensor through the Driver.

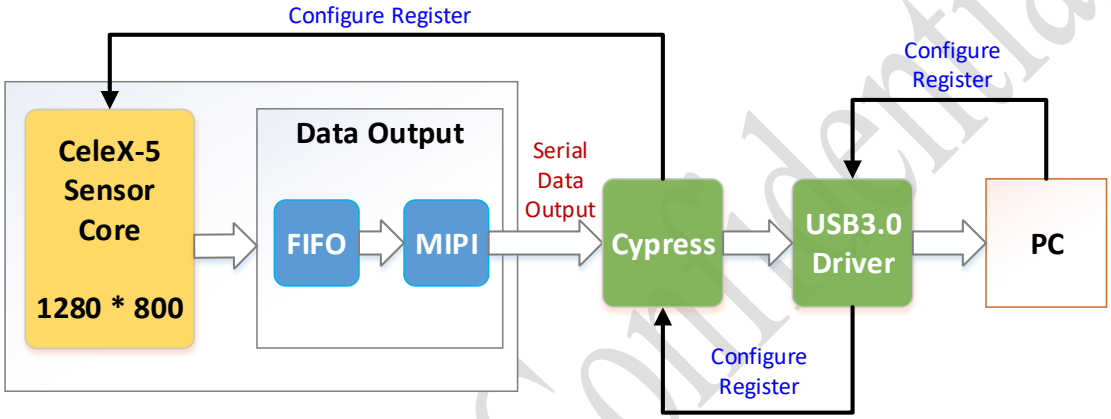


Fig. 1-1 Basic working principle of the CeleX-5 Chipset

1.1.2 Terminology

The following table lists some of the terms and their explanations that appear in this document.

No.	Terminology	Description
1	Event Mode	*Event Address-only Mode*, *Event Optical-flow Mode* and *Event Intensity Mode* are collectively referred to as Event Mode，in which the Sensor only detects the pixel whose intensity has changed and then marks it as an active pixel and outputs it.
2	Full-frame Mode	*Full-frame Picture Mode*, *Single Full-frame Optical-flow Mode* and *Multiple Full-frame Optical-flow Mode* are collectively referred to as *Full-frame Mode*, in which the Sensor outputs the information for each pixel in order (top to bottom, left to right) within a certain period of time.
3	Fixed Mode	For fixed mode, the sensor always works into only one of the six single modes unless being reconfigured.

4	Loop Mode	For loop mode, the sensor could automatically switch between three single modes.
5	Event Address-only Mode	A working mode of the CeleX-5 sensor, in which the sensor only outputs row/column address of the detected events. The pixel intensity and optical-flow information are not available in the mode.
6	Event Optical-flow Mode	A working mode of the CeleX-5 sensor, in which the sensor outputs row/column address of the detected events, together with the timing information. The pixel intensity information is not available in this mode.
7	Event Intensity Mode	A working mode of the CeleX-5 sensor, in which the sensor outputs row/column address of the detected events, together with the pixel intensity information. The optical-flow information is not available in this mode.
8	Full-frame Picture Mode	A working mode of the CeleX-5 sensor, in which the sensor could generate full-frame pictures with pixel intensity information, which is sampled at certain instant by all pixels simultaneously.
9	Single Full-frame Optical-flow Mode	A working mode of the CeleX-5 sensor, in which the sensor could generate a full-frame optical-flow picture after accumulating events in a specified time duration.
10	Multiple Full-frame Optical-flow Mode	A working mode of the CeleX-5 sensor, in which the sensor could generate several full-frame optical-flow pictures with pixel timing information within a time duration.
11	Event Binary Pic	The event binary frame output from the SDK
12	Event Gray Pic	The event fray frame from the SDK
13	Event Accumulated Gray Pic	The event accumulated gray frame output from the SDK
14	Event Count Pic	The event trigger number frame output from the SDK

1.2. Working Principle of CeleX-5 Sensors

CeleX™ is a family of smart image sensor specially designed for machine vision. Each pixel in CeleX™ sensor can individually monitor the relative change in light intensity and report an event if it reaches a certain threshold. Asynchronous row and column arbitration circuits process the pixel requests and make sure only one request is granted at a time in fairly manner when they received multiple simultaneous requests. The output of the sensor is not a frame, but a stream of asynchronous digital events. The speed of the sensor is not limited by any traditional concept such as exposure time and frame rate. It can detect fast motion which is traditionally captured by expensive, high speed cameras running at thousands of frames per second, but with drastic reduced amount of data. Besides, our technology allows post-capture change of frame-rate for video playback. One can view the video at 10,000 frames per second to see high speed events or at normal rate of 25 frames per second.

CeleX™ sensor can produce three kinds of outputs in parallel: logarithmic picture, motion, and full-frame optical flow. The sensor can greatly improve the performance for applications in broad areas including assisted/autonomous driving, UAV, robotics, surveillance, etc.

CeleX-5 is a multifunctional smart image sensor with 1Mega-pixels and some additional features integrating on-chip. The sensor supports several different output formats: pure binary address-events, address-events with either pixel intensity information or timing information.

In addition, the readout scheme of the sensor could either be asynchronous data stream or synchronous full frames. Different combinations of the output format and readout scheme lead to great flexibility of this sensor, which supports 6 separate operation modes in total.

To further meet the requirements of different applications, the sensor could also be configured into a loop-mode, in which it could automatically switch among three separate modes.

1.2.1. Fixed Mode of CeleX-5 Sensor

CeleX-5 Sensor provides six fixed working modes of CeleX-5 Sensors: *Event Address Only Mode*, *Event Optical-flow Mode*, *Event Intensity Mode*, *Full-frame Picture Mode*, *Single Full-frame Optical-flow Mode*, *Multiple Full-frame Optical-flow Mode.*

1.2.1.1. Event Mode with ADC Disabled (*Event Address Only Mode*)

In this mode, the sensor detects motions in field of view and outputs row/col address of the detected events. The column ADC is disabled in this mode, so the pixel intensity or timing information is not available.

The advantage of this mode is that short event latency is achieved as the time for AD conversion has been saved, thus the sensor could process the events in higher speed. Its working principle is shown in Figure 1-2.

1.2.1.2. Event Mode with Optical-Flow (*Event Optical-flow Mode*)

In this mode, the sensor detects motions in field of view and outputs row/col address of the detected events, together with the timing information indicating the instant when the event generated. The pixel intensity information is not available in this mode.

The address and timing information of the events could be used for further optical-flow analysis. Its working principle is shown in Figure 1-2.

1.2.1.3. Event Mode with Pixel Intensity (*Event Intensity Mode*)

In this mode, the sensor detects light intensity change and outputs row/col address of the detected events, together with the pixel intensity information sampled at the instant when the event generated. The timing information is not available in this mode. Its working principle is shown in Figure 1-2.

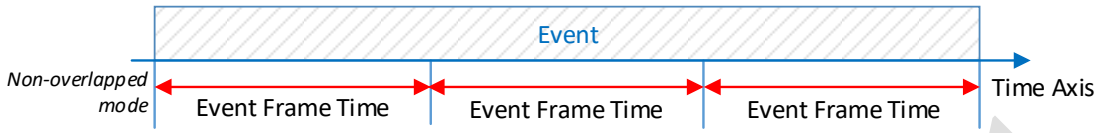


Fig. 1-2 Event data mode

The address and intensity information of the events could be used for further analysis.

For this mode, it could obtain several formats of frame simultaneously through APIs: one is event accumulated gray frame (the grayscale image accumulating the gray values of active pixels newly changed), event binary frame (the gray values of active pixels are marked as 255, while inactive pixels are 0), event gray frame and so on.

In the Event mode, the Event Frame Time can be adjusted (range: 1~1000ms), and you can modify this time by calling the API interface *setEventFrameTime*. The method of creating an Event image frame is described in section 1.2.4.2.

1.2.1.4. Full-Frame Picture Mode

In this mode, the sensor could generate full-frame pictures with pixel intensity information, which is sampled at certain instant by all pixels simultaneously. Because the sensor could generate continuous full frames, it could be used as a frame-based APS sensor.

The operating frequency of the Sensor is linear with the frame rate of the transmitted data, ie the frequency is x MHz, then fps is x frames per second. If the operating frequency of the Sensor is 100MHz, it means that the Sensor can generate about 100 full-frame image frames per second, that is, the Transmission Time is 10ms.

In this mode, you can modify the working frequency of the Sensor by calling the API interface *setClockRate* to modify the transmission time (or Frame Time), and also improve the image quality by calling the API interfaces *setBrightness* and *setContrast*.

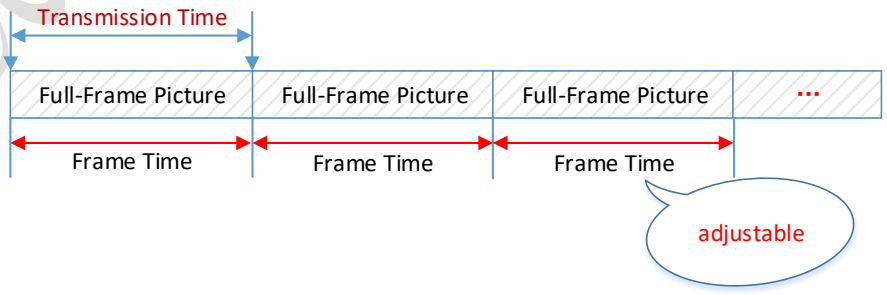


Fig 1-3. Full-Frame Picture Mode

In this mode, you can modify the Transmission Time (or Frame Time) by calling the API *setFullPicFrameTime*, for example, if it is set to 100ms, that means the sensor could generate about 100 full-frame pictures per second. You can also increase image quality by calling the API *setBrightness* and *setContrast*.

1.2.1.5. Single Full-Frame Optical-Flow Mode

In this mode, the sensor could generate a full-frame picture with pixel timing information. The sensor first accumulates events in a specified duration (*Accumulated Time*). After that, the sensor produces a full-frame picture with pixel timing information.

Parameters users could set are a) accumulated time (hardware parameter), b) transmission time (hardware parameters, related to main clock frequency).

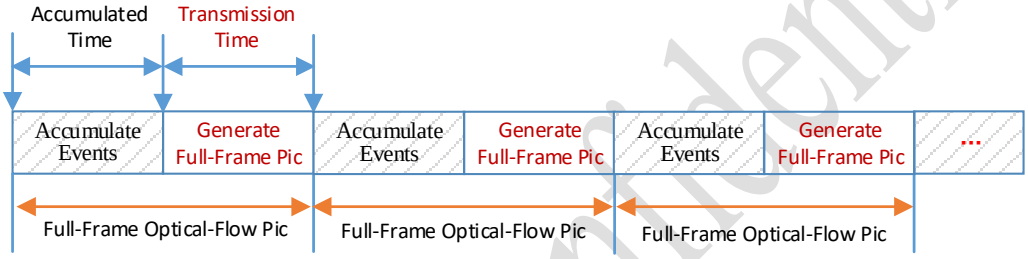


Fig 1-4. Full-Frame Optical-Flow (Single) Mode

1.2.1.6. Multiple Full-Frame Optical-Flow Mode

In this mode, the sensor could generate several full-frame pictures with pixel timing information within a time duration. Different from *Event Optical-flow Mode*, sensor operating in this mode produces full-frame pictures instead of asynchronous events stream. Different from *Single Full-frame Optical-flow Mode*, the sensor could produce multiple full-frame pictures within a time duration.

It's not recommended to use this mode as a fixed mode, it's good to use it as one of the three mode in the *Loop Mode*. You could modify the duration of this mode in the loop by calling the API interface *setPictureNumber*.

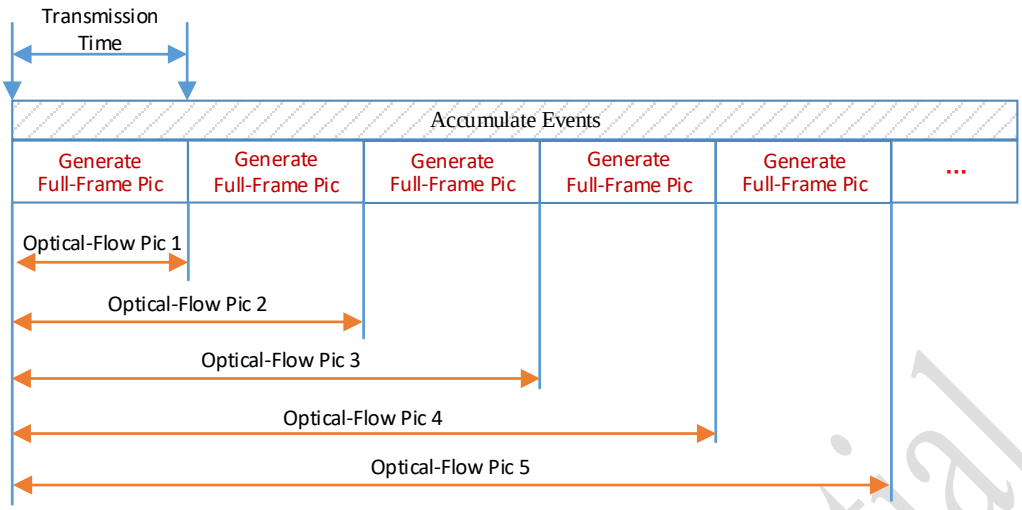


Fig 1-5. Full-Frame Optical-Flow (Multiple) Mode

1.2.2. Loop Mode of CeleX-5 Sensor

The detailed operation of each single mode has been illustrated above. For fixed mode, the sensor always works into only one of the six single modes unless being reconfigured. For loop mode, the sensor could automatically switch between three single modes.

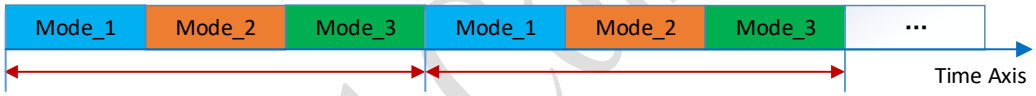


Fig. 1-6. Loop Mode

Fig. 1-6 shows the diagram of sensor operating in loop mode. After the select mode is configured to loop mode, the sensor would operate between Mode_1, Mode_2 and Mode_3 alternatively. This loop would continue until being reconfigured.

Here are some tips to select modes for the loop mode:

a) The best choice is one full-frame picture mode, one event mode and one full-frame optical-flow mode;

b) Only choose one full-fame optical-flow mode in the loop.

By default, the first loop (Mode_1) is *Full-frame Picture Mode*, in which the time duration is **10ms** and you could modify the duration of this mode by calling the API interface *setPictureNumber*.

The second loop (Mode_2) is *Event Mode*, in which the time duration is **20ms** and you could modify the event duration of this mode by calling the API interface *setEventDuration*.

The third loop (Mode_3) is *Full-frame Optical-flow Mode*, in which the time duration is **30ms** and you could modify the duration of this mode by calling the API interface *setPictureNumber*.

1.2.3. Data Format of CeleX-5 Sensor

1.2.3.1. MIPI Data Format

The data format of the *Full-frame Mode* is relatively simple, and it represents continuous light intensity information (ADC), as shown in the Fig. 1-9. The ADC values of two consecutive pixels can be parsed from three consecutive bytes.

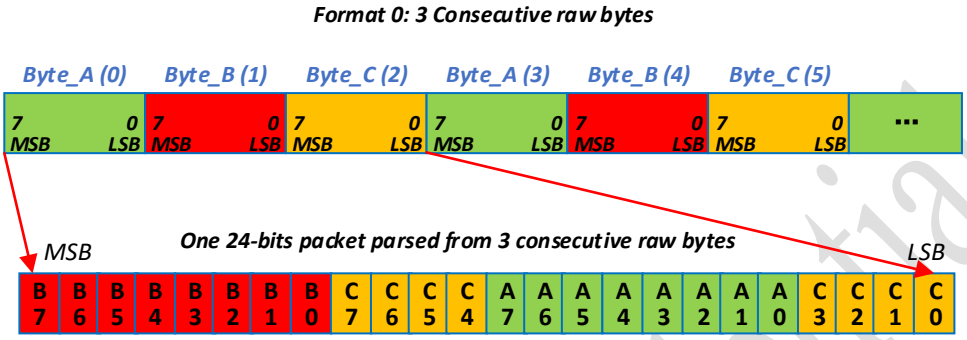


Fig. 1-9

In *Event Mode*, CeleX-5 sensor supports 3 data formats. You can set which data format to use by calling the API interface *setEventDataFormat*, which should be called before *openSensor*. Once the Sensor is enabled, calling this interface to modify the data format will not work.

If you don't care how to parse the data, you can skip the following sections, because the data has been parsed in the SDK and provided to the user in the form of images and (x, y, A, T) information.

The following sections describe these three data formats and how to parse out used information like: row address, column address, ADC, row timestamp, and Temperature.

1. Format 0

The Format 0 has 4 packages as shown in the below figure, each of which is 24-bits (3 bytes). The lowest 2 bits represent the type (ID) of the package. For example, if ID = 2b'10, it means Package_A, from which the row address and row timestamp could be parsed.

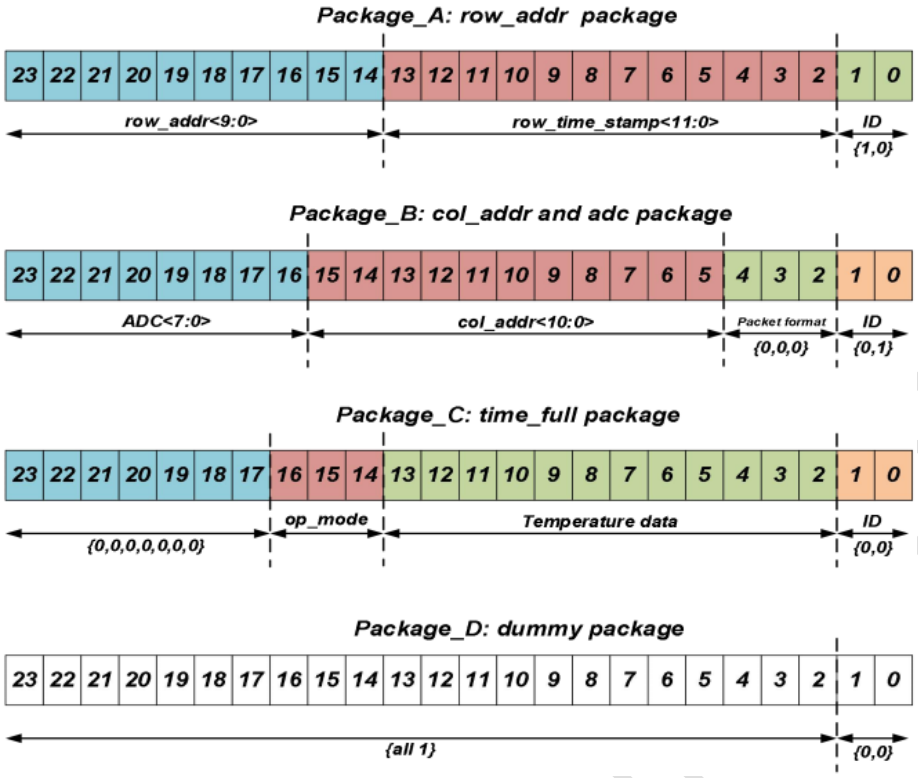


Fig. 1-10 Format 0: 24-bit packet with ADC data

It should be noted that this 24-bits data is not consecutive 24-bits in the original raw data, it needs to rearrange their order, the following figure shows how to parse this 24-bits data from three consecutive raw bytes.

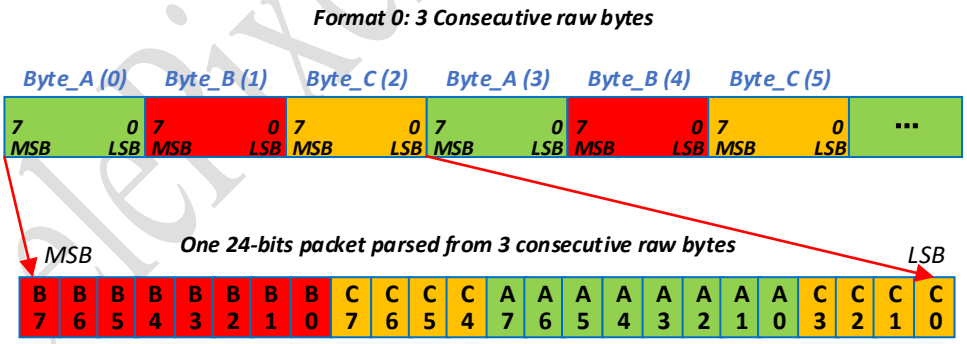


Fig. 1-11

2. Format 1

The Format 1 also has 4 packages as shown in Fig. 1-12, each of which is 28-bits. The lowest 2 bits represent the type (ID) of the package. For example, if ID = 2b'10, it means Package_A, from which the row address and row timestamp could be parsed. It differs from Format 0 in that the ADC information has 12 bits and the row timestamp has 16 bits.

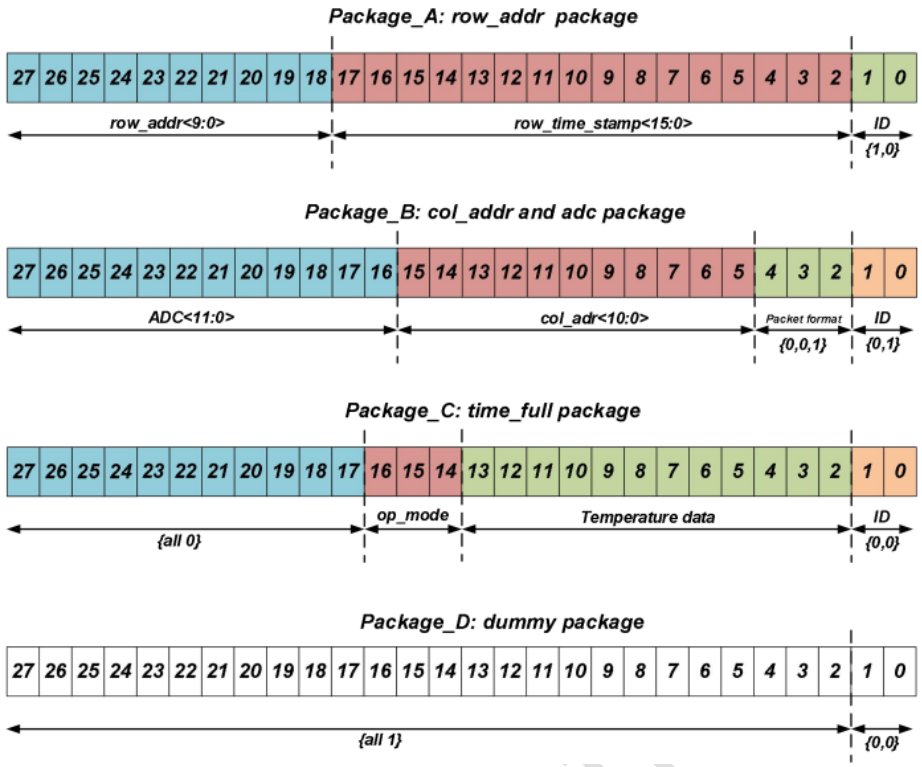


Fig 1-12 Format 1: 28-bit packet with ADC data

Similarly, the 28-bit data is not consecutive 28-bits in the original raw data, it needs to be rearranged in order. The following figure shows how to parse the two 28-bits data from seven consecutive raw bytes (56-bits).

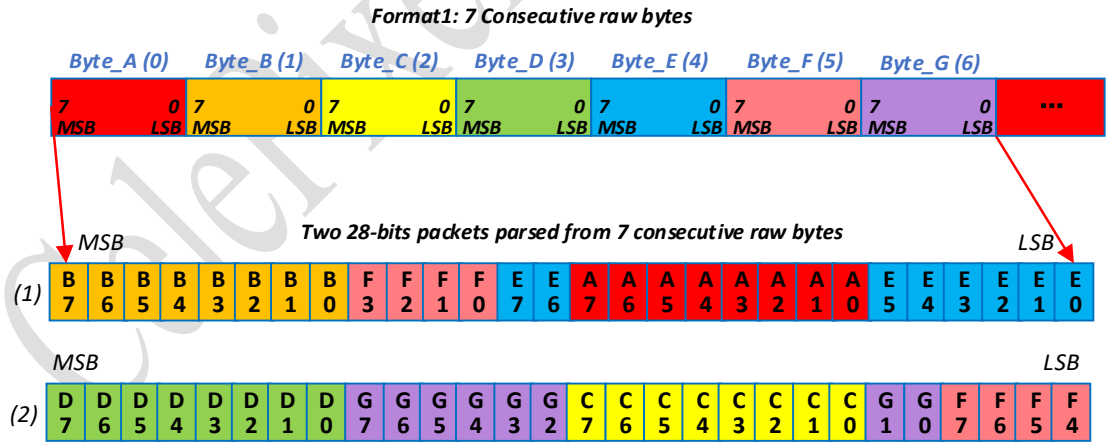


Fig. 1-13

2. Format 2

The Format 2 has 5 packages as shown in the below figure, each of which is 14-bits. The lowest 2 bits represent the type (ID) of the package. For example, if ID = 2b'10, it means Package_A, from

which only the row address could be parsed. Unlike the first two data formats, there is no ADC information in the format 2. This format cannot be used if you want to get the ADC information of the Event.

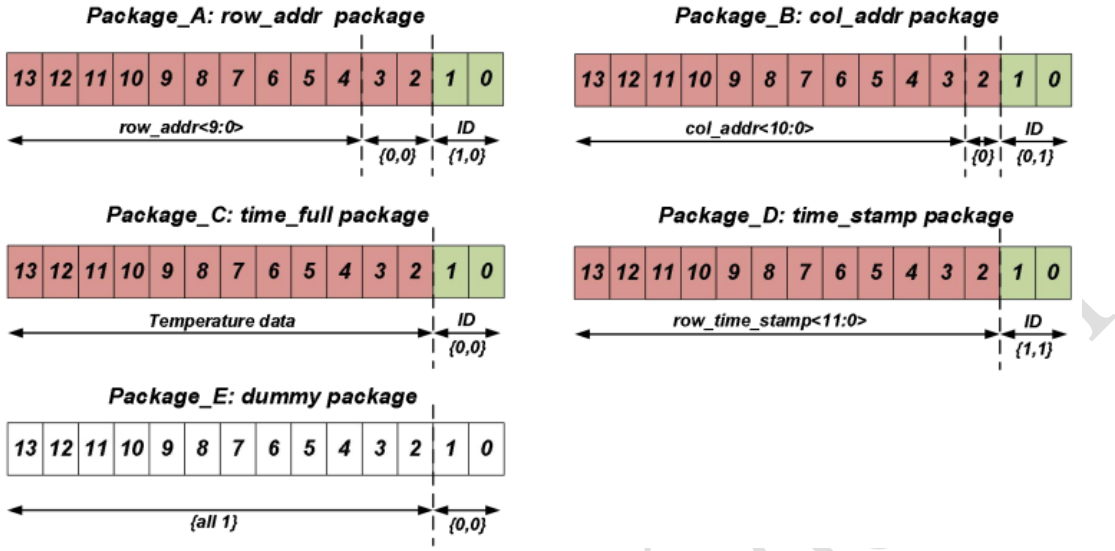


Fig 1-14 Format 2: 14-bit packet without ADC data

Similarly, the 14-bits data is not consecutive 28-bits in the original raw data, it needs to be rearranged in order. The following figure shows how to parse the four 28-bits data from seven consecutive raw bytes (56-bits).

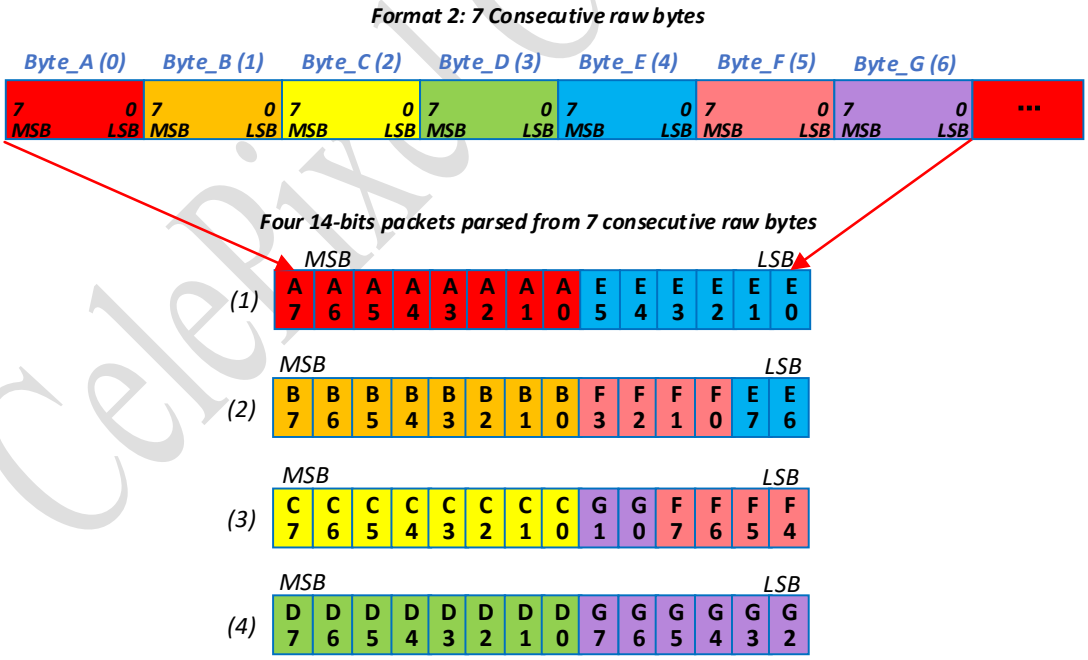


Fig. 1-15

1.2.3.2. Data Types Output by SDK

The data types output by the SDK are different when the CeleX-5 Sensor works in a different mode. The table below shows the types of data that users can obtain in different modes.

Table 1-1: Data Types Output by SDK

Sensor Mode	Data Types Output by SDK
Full-frame Picture Mode	Full Pic Buffer/Mat
Event Address-only Mode	Event Binary Pic Buffer/Mat Event Denoised Pic Buffer/Mat Event Count Pic Buffer/Mat Event Vector<row, col, timestamp>
Event Optical-flow Mode	Event Optical-flow Pic Buffer/Mat Event Binary Pic Buffer/Mat Event Vector<row, col, optical-flow info, timestamp>
Event Intensity Mode	Event Binary Pic Buffer/Mat Event Gray Pic Buffer/Mat Event Count Pic Buffer/Mat Event Accumulated Pic Buffer/Mat Event Superimposed Pic Buffer/Mat Event Vector<row, col, brightness, polarity, timestamp>
Single Full-frame Optical-flow Mode	Event Optical-flow Pic Buffer/Mat Event Optical-flow Direction Pic Buffer/Mat Event Optical-flow Speed Pic Buffer/Mat Event Binary Pic Buffer/Mat

xxx Pic Buffer is a single channel image array with row = 1280 and column = 800, and x***xx Pic Mat*** is a single channel OpenCV mat with row = 1280 and column = 800.

Additionally, the ***timestamp*** in the Event Vector has different range and time accuracy (T-Unit) at different frequencies or mode, as described below.

1. Event data format = 1 (used in *Event Intensity Mode* and *Event Optical-flow Mode*)

(1) The default frequency of the Sensor is 70MHz, and the adjustable range is 20MHz ~ 70MHz.

(2) The event timestamp range: 0 ~ 65535 (16 bits).

(3) By default, the SDK creates a frame of Event images every 30ms, so the output timestamp range is 0 ~ 2142.

2. Event data format = 2 (used in *Event Address Only Mode*)

(1) The default frequency of the Sensor is 100MHz, and the adjustable range is 20MHz ~ 100MHz.

(2) The event timestamp range: 0 ~ 4095 (12 bits).

(3) By default, the SDK creates a frame of Event images every 30ms, so the output timestamp range is 0 ~ 1500.

Table 1-2: Time Accuracy and Timestamp Range at Different Frequencies

Clock (MHz)	T-Unit (us)	The number of time units corresponding to 30ms 30*1000 / T-Unit
20	T_Unit = 25	30*1000 / 25 = 1200
30	T_Unit = 22	30*1000 / 22 = 1363
40	T_Unit = 25	30*1000 / 25 = 1200
50	T_Unit = 20	30*1000 / 20 = 1500
60	T_Unit = 17	30*1000 / 17 = 1764
70	T_Unit = 14	30*1000 / 14 = 2142
80	T_Unit = 25	30*1000 / 25 = 1200
90	T_Unit = 22	30*1000 / 22 = 1363
100	T_Unit = 20	30*1000 / 20 = 1500

1.2.4. Methods to Create Full Picture and Event Frame

CeleX-5 sensor can output a continuous stream of pixel events after powering-up. As illustrated in the last section, the X, Y, A, T information can be decoded from the events. Next, we will introduce how to create a visual frame with (X, Y, A, T) information.

1.2.4.1. Method to Create Full Picture Frame

1) Call the API interface *setClockRate* to set the working frequency of the Sensor, indicating that the Sensor generates a complete frame (Full Picture) after user-set Full-Picture Frame Time. This mode is compatible with the traditional Sensor, that is, regardless of whether there is a change of intensity on the pixel or not, its gray value can be retrieved.

2) Switch the sensor mode to Full-Picture mode.

3) Construct a 2D array to represent the Full Pic, regarded as M[800][1280], which has 800 rows and each row consists of 1280 pixels. Initialize every pixel's brightness value to 0 at first.

4) When an event E is decoded into X, Y, A and T, the value of each pixel on M[Y][X] is given brightness value of A, i.e., M[Y][X] = A. Repeat the same process for each decoded event. In the Full-Picture mode, the (T) information is invalid.

5) Repeat step 4 in the next Full-Picture frame time.

The above frame creation process is implemented by the API. You can directly call the API *getFullPicBuffer* to obtain the data array of the Full Picture frame.

1.2.4.2. Method to Create Event Picture Frame

There are many ways to create image frames in Event mode. The following content only describes the method to create Event Binary Pic, Event Gray Pic and Event Accumulated Gray Pic in the SDK.

1) Call the API *setEventFrameTime* to set the Event frame time, indicating that the SDK will combine the Event outputs from the Sensor after a certain time interval (i.e., the Event Frame Time set by the user) to generate an Event binary frame.

2) Switch the sensor mode to Event mode.

3) Construct three 2D array M1[800][1280]， M2[800][1280] and M3[800][1280] to represent the Event Binary Pic, Event Gray Pic and Event Accumulated Gray Pic. Initialize every pixel's brightness value to 0 at first.

4) Parse the data obtained from the data stream from the USB driver. When each event E is decoded as (X, Y, A, T), the value of the pixel on M1[Y][X] is 255, and the value of the pixel on M2[Y] [X] and M3[Y][X] is the brightness value A of the pixel.

5) In the new Event Frame Time, first set the value of each pixel in the 2D arrays M1 and M2 to 0. M3 remains unchanged. Then repeat step 4. That is, in each Event Frame Time, M1 and M2 are cleared each time, and the M3 array is updated based on the array created by the previous Event Frame Time.

The above frame creating process is implemented by the API. The user can directly call the API *getEventPicBuffer* to obtain the data arrays of Event frames. To help users understand the difference between Event Gray Pic and Event Accumulated Gray Pic, the above process is further illustrated in Fig. 1-16 and 1-17.

Fig. 1-16 and 1-17 describes the process above with a 5*5 array. We only list the first five frames, and each frame is an event gray frame in Fig. 1-16 and event accumulated gray frame in Fig. 1-17.

"x" represents gray value in event gray frame and event accumulated gray frame. The red "x" indicates the pixels that changed in the first Event Frame Time, the blue "x" indicates the second, the green "x" indicates the third, and the purple "x" indicates the fourth, and the black "x" indicates the fifth.

In Fig. 1-8, it should be noted that when a pixel has changed before and it changed again, we replace the old gray value with the new one directly, which are marked in location (row0, col0) and location (row2, col4).

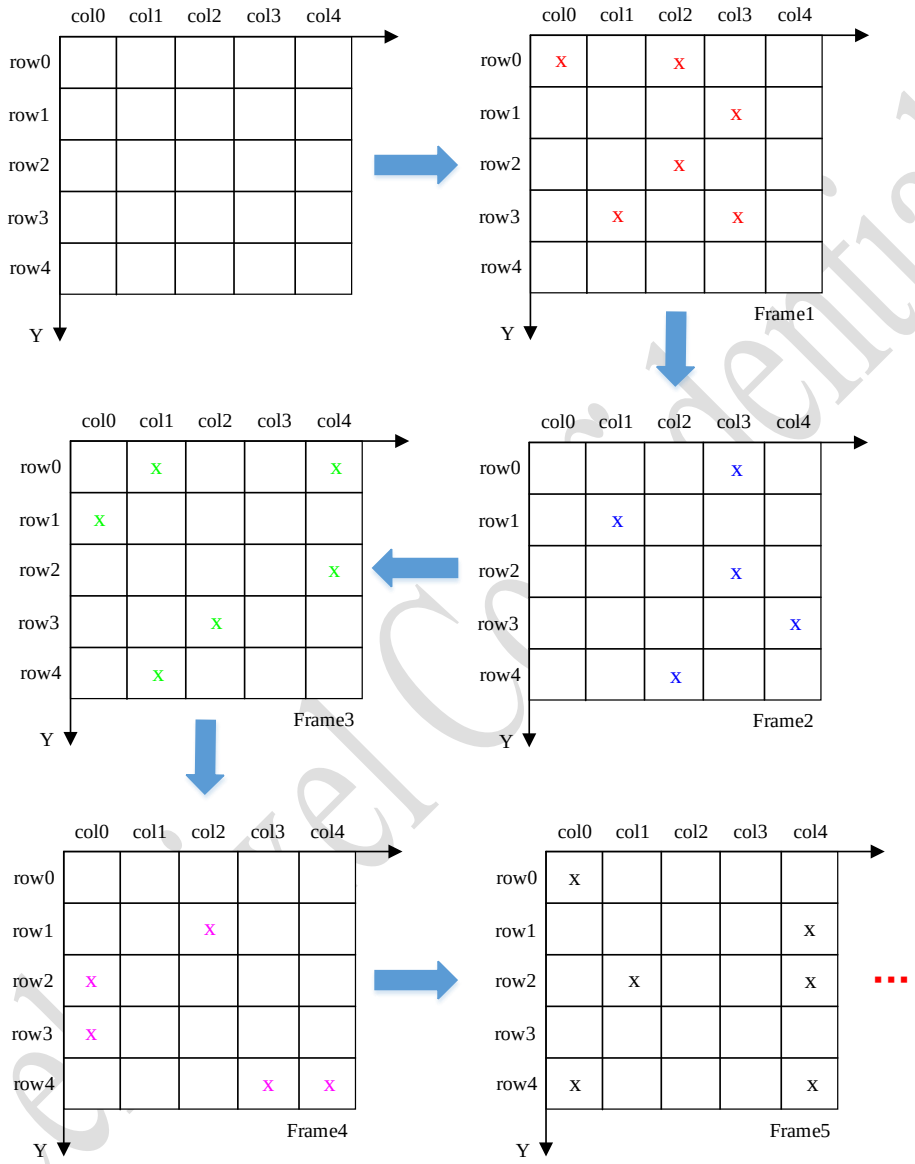


Fig. 1-16 Event Gray Pic in Event mode

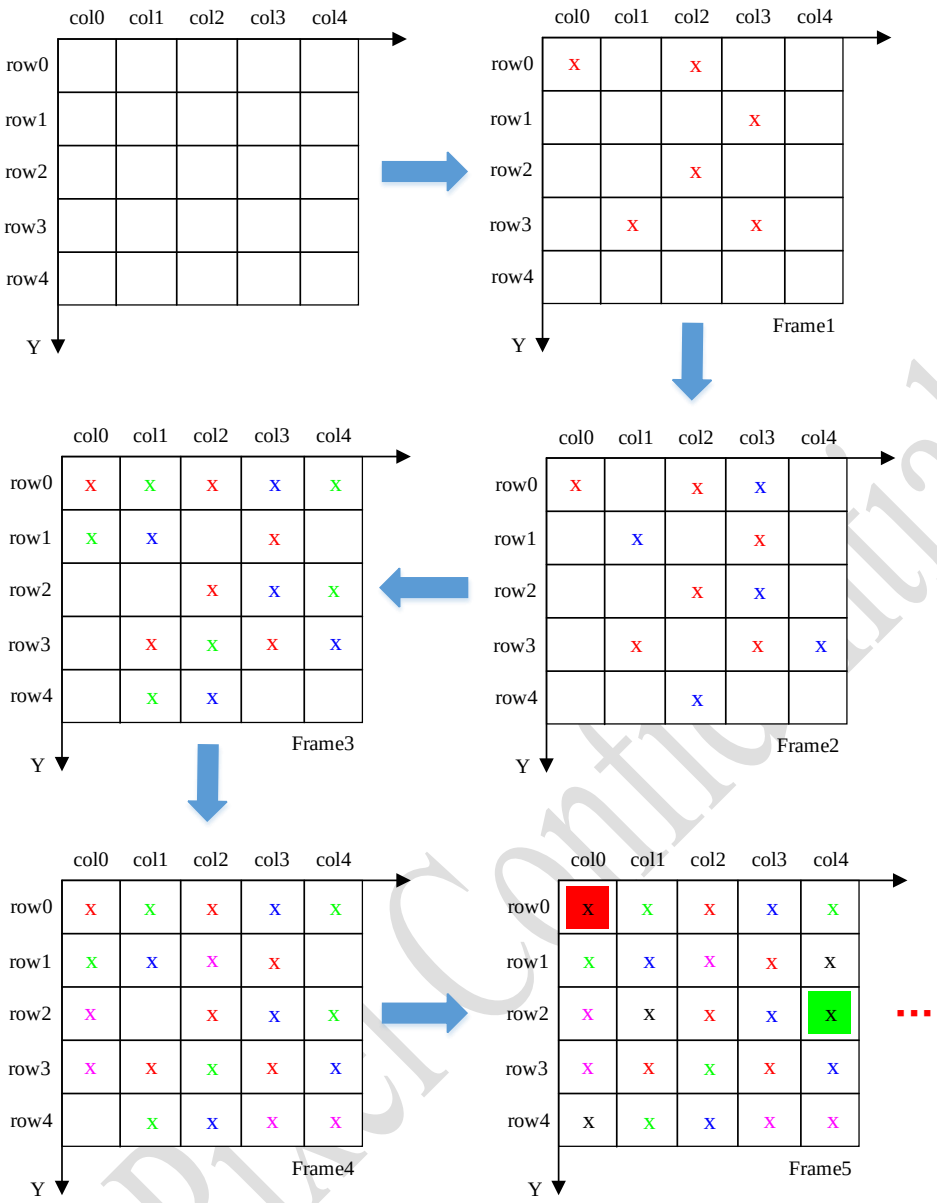


Fig. 1-17 Event Accumulated Gray Pic in Event mode

1.3. Data Structure of Bin File

1.3.1. Data structure of the bin file without IMU data

The bin file data structure without IMU data includes a header (*Bin Header*) and a specific image packet (*Image Package*), as shown in Figure 1-18.

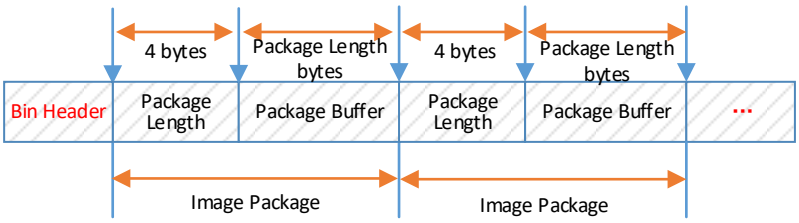


Fig. 1-18 The data structure of the bin file without IMU data

The format of the *Bin Header* is shown in the structure below, where the bit1 of *data_type* is used to indicate whether or not the IMU data is included.

```
typedef struct BinFileAttributes
{
    uint8_t     data_type; //bit0: 0: fixed mode; 1: loop mode
                           // bit1: 0: no IMU data; 1: has IMU data
    uint8_t     loopA_mode;
    uint8_t     loopB_mode;
    uint8_t     loopC_mode;
    uint8_t     event_data_format;
    uint8_t     hour;
    uint8_t     minute;
    uint8_t     second;
    uint32_t    package_count;
} BinFileAttributes;
```

The *Package Length* refers to the size of a *Package Buffer*, and the *Package Buffer* stores the data in the MIPI format as described in section 1.2.3.1 (the default size of the Event data is 357001, and the default size of the Full-frame data is 153661).

1.3.2. Data structure of the bin file with IMU data

The bin file data structure with IMU data includes a header (*Bin Header*), a specific image packet (*Image Package*) and IMU data (*IMU Data*), as shown in Figure 1-19.

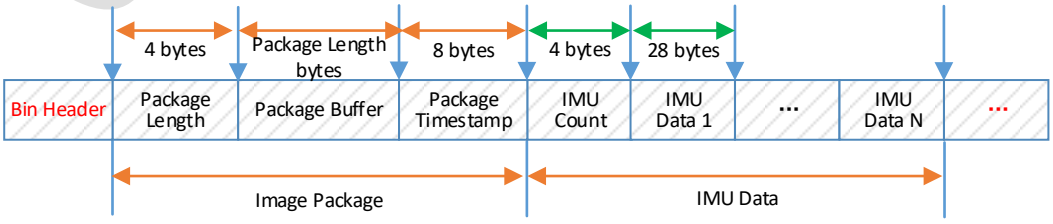


Fig. 1-19 The data structure of the bin file with IMU data

The format of the *Bin Header*, *Package Length*, and *Package Buffer* is as described in the previous section, where *Package Timestamp* is the timestamp of receiving the *Image Package*.

IMU Data includes the number of IMU data and specific information of each IMU data, where the *IMU Count* is the number, and the *IMU Data N* is the Nth IMU data (including the specific IMU data and timestamp information).

Users can obtain IMU data through the API interface getIMUData without having to parse the IMU data themselves, so the data format of *IMU Data N* is not introduced here.

2. CeleX-5 API Reference

2.1. Overview

The CeleX API provide a C++ interface to communicate with CeleX-5 Sensor.

To build an application using this API, you need to include the files in the *include* directory in you project. These contain all the functions that will make calls to the shared library. You will also need to include the CeleX.dll (Windows) library in the same directory as your executable or libCeleX.so (Linux) in the lib directory (e.g. /usr/local/lib).

To use the library, you could create an instance of CeleX5 which encapsulated communication the CeleX-5 sensor. You can call *openSensor* to open the sensor that has connected to your PC and then you can call *getFullPicBuffer* and *getEventPicBuffer* to get various of frames that calculated using the raw data the sensor outputted.

You can also obtain the processed frame buffer by deriving your own data manager subclass from class CeleX5DataManager and override its virtual *onFrameDataUpdated* methods to receive the notification that the frame buffers are ready.

Here are enums and structs with brief descriptions:

enum CeleX5Mode {
Unknown_Mode = -1,
Event_Address_Only_Mode = 0,
Event_Optical_Flow_Mode = 1,
Event_Intensity_Mode = 2,
Full_Picture_Mode = 3,
Full_Optical_Flow_S_Mode = 4,
Full_Optical_Flow_M_Mode = 6,
};
	Unknown_Mode - Unknown mode
	Event_Address_Only_Mode - Event Mode with ADC Disabled (*Event Address Only Mode*)
	Event_Optical_Flow_Mode - Event Mode with Optical-Flow (*Event Optical-flow Mode*)
	Event_Intensity_Mode - Event Mode with Pixel Intensity (*Event Intensity Mode*)
	Full_Picture_Mode - Full-frame picture mode
	Full_Optical_Flow_S_Mode - Single full-frame optical-flow mode
	Full_Optical_Flow_M_Mode - Multiple full-frame optical-flow mode

enum DeviceType {
Unknown_Devive = 0,
CeleX5_MIPI = 1,
CeleX5_OpalKelly = 2,
CeleX5_ZYNQ = 3
};
	Unknown_Device - Unknown device
	CeleX5_MIPI - CeleX5 MIPI device
	CeleX5_OpalKelly - CeleX5 OpalKelly device
	CeleX5_ZYNQ – CeleX5 ZYNQ device

enum emEventPicMode { EventBinaryPic = 0, EventAccumulatedPic = 1, EventGrayPic = 2,	**EventBinaryPic** - The frame buffer (a matrix) contains Gray values(255) of active pixels obtained from Sensor, and other inactive pixels value is filled with Gray value(0). **EventAccumulatedPic** - The frame buffer (a matrix)

EventCountPic = 3, EventDenoisedBinaryPic = 4, EventSuperimposedPic = 5 };	contains light intensity of active pixels obtained from Sensor, and other inactive pixels value is filled by their existing light intensity value. **EventGrayPic -** The frame buffer (a matrix) contains light intensity (not 255) of active pixels obtained from the Sensor, and other inactive pixels value is filled with Gray value(0). **EventCountPic -** The event trigger number frame buffer (matrix). **EventDenoisedBinaryPic -** The frame buffer (matrix) is similar to the event binary buffer, except that it is a denoised event binary buffer with a simple custom algorithm. **EventSuperimposedPic -** The frame buffer (matrix) that superimposes event binary picture onto event accumulated picture in Event mode.

enum emFullPicType {
Full_Optical_Flow_Pic = 0,
Full_Optical_Flow_Speed_Pic = 1,
Full_Optical_Flow_Direction_Pic = 2
};
	Full_Optical_Flow_Pic – The full-frame optical flow buffer.
	Full_Optical_Flow_Speed_Pic - The speed frame buffer of each pixel calculated on the optical flow raw frame.
	Full_Optical_Flow_Direction_Pic - The direction frame buffer of each pixel calculated on the optical flow raw frame.

typedef struct BinFileAttributes { uint8_t data_type; uint8_t loopA_mode; uint8_t loopB_mode; uint8_t loopC_mode; uint8_t event_data_format; uint8_t hour; uint8_t minute; uint8_t second; uint32_t package_count; } BinFileAttributes;	**data_type -** bit0: 0: fixed mode; 1: loop mode. - bit1: 0: no IMU data; 1: has IMU data. **loopA_mode -** The first working mode of the loop. **loopB_mode -** The second working mode of the loop. **loopC_mode -** The third working mode of the loop. **event_data-format -** The event data format used. **hour, minute, second -** The time length of the bin file recording. **package_count -** The package count of the recorded bin file.
typedef struct EventData { uint16_t col; uint16_t row; uint16_t brightness; uint32_t t; int8_t p;	**col -** The column information of one pixel in the frame buffer. **row -** The row information of one pixel in the frame buffer. **brightness** - The brightness of one pixel in the frame buffer.

}EventData;	**t** - The time of one pixel in the frame buffer. **p** - The polarity of one pixel in the frame buffer.(-1: intensity weakened; 1: intensity is increased; 0 intensity unchanged)
typedef struct IMUData { double x_GYROS; double y_GYROS; double z_GYROS; uint32_t t_GYROS; double x_ACC; double y_ACC; double z_ACC; uint32_t t_ACC; double x_MAG; double y_MAG; double z_MAG; uint32_t t_MAG; double x_TEMP; uint64_t frameNo; std::time_t time_stamp; } IMUData;	**x_GYROS** - Angular rate in the X-axis (gyroscopes) **y_GYROS** - Angular rate in the Y-axis (gyroscopes) **z_GYROS** - Angular rate in the Z-axis (gyroscopes) **t_GYROS** - Timestamp when angular rate is received **x_ACC** - Acceleration in the X-axis (accelerometer) **y_ACC** - Acceleration in the Y-axis (accelerometer) **z_ACC** - Acceleration in the Z-axis (accelerometer) **t_ACC** - Timestamp when acceleration is received **x_MAG** - Magnetometer component in the X-axis **y_MAG** - Magnetometer component in the Y-axis **z_MAG** - Magnetometer component in the Z-axis **t_MAG** - Timestamp when magnetometer is received **frameNo** - Frame number **time_stamp** – Timestamp when the IMU data received

2.2. CeleX5DataManager Class Reference

This class allows to be notified about the processed frame buffer that is ready. To receive these notifications, you need to derive your own data manager subclass from this class and override its virtual *onFrameDataUpdated* methods.

```
#include <opencv2/opencv.hpp>
#include <celex5/celex5.h>
#include <celex5/celex5datamanager.h>
#include <celex5/celex5processeddata.h>

#ifdef _WIN32
#include <windows.h>
#else
#include<unistd.h>
#endif

#define FPN_PATH       "./FPN.txt"
```

```
class SensorDataObserver : public CeleX5DataManager
{
public:
	SensorDataObserver(CX5SensorDataServer* pServer)
	{
		m_pServer = pServer;
		m_pServer->registerData(this, CeleX5DataManager::CeleX_Frame_Data);
	}
	~SensorDataObserver()
	{
		m_pServer->unregisterData(this, CeleX5DataManager::CeleX_Frame_Data);
	}
	virtual void onFrameDataUpdated(CeleX5ProcessedData* pSensorData);//overrides Observer operation
	CX5SensorDataServer* m_pServer;
};

void SensorDataObserver::onFrameDataUpdated(CeleX5ProcessedData* pSensorData)
{
	if (NULL == pSensorData)
		return;
	CeleX5::CeleX5Mode sensorMode = pSensorData->getSensorMode();
	if (CeleX5::Full_Picture_Mode == sensorMode)
	{
		//get fullpic when sensor works in FullPictureMode
		if (pSensorData->getFullPicBuffer())
		{
			cv::Mat matFullPic(800, 1280, CV_8UC1, pSensorData->getFullPicBuffer()); //full pic
			cv::imshow("FullPic", matFullPic);
			cv::waitKey(1);
		}
	}
	else if (CeleX5::Event_No_ADC_Mode == sensorMode)
	{
		//get buffers when sensor works in EventMode
		if (pSensorData->getEventPicBuffer(CeleX5::EventBinaryPic))
		{
			cv::Mat matEventPic(800, 1280, CV_8UC1,
			pSensorData->getEventPicBuffer(CeleX5::EventBinaryPic)); //event binary pic
			cv::imshow("Event Binary Pic", matEventPic);
			cvWaitKey(1);
		}
	}
	else if (CeleX5::Event_Intensity_Mode == sensorMode)
```

```
        {
            //get buffers when sensor works in FullPic_Event_Mode
            if (pSensorData->getEventPicBuffer(CeleX5::EventBinaryPic))
            {
                cv::Mat matEventPic1(800, 1280, CV_8UC1,
                pSensorData->getEventPicBuffer(CeleX5::EventBinaryPic)); //event binary pic
                cv::Mat matEventPic2(800, 1280, CV_8UC1,
                pSensorData->getEventPicBuffer(CeleX5::EventGrayPic)); //event gray pic
                cv::imshow("Event Binary Pic", matEventPic1);
                cv::imshow("Event Gray Pic", matEventPic2);
                cvWaitKey(1);
            }
        }
}

int main()
{
        CeleX5 *pCeleX = new CeleX5;
        if (NULL == pCeleX)
            return 0;
        pCeleX->openSensor(CeleX5::CeleX5_MIPI);
        pCeleX->setFpnFile(FPN_PATH);
        CeleX5::CeleX5Mode sensorMode = CeleX5::Event_Intensity_Mode;
        pCeleX->setSensorFixedMode(sensorMode);
        SensorDataObserver* pSensorData = new SensorDataObserver(pCeleX->getSensorDataServer());

        while (true)
        {
#ifdef _WIN32
            Sleep(5);
#else
            usleep(1000 * 5);
#endif
        }
        return 1;
}
```

2.3. CeleX5 Class Reference

This is the class that encapsulates the methods to get the data from the sensor as well as the functionality to adjust the working mode and configuration register parameters of the sensor.

Public Member Functions:

No.	API Name	Description
1	openSensor	Start sensor via calling this interface
2	isSensorReady	Check whether CeleX-5 sensor is successfully initialized
3	getMIPIData	Read MIPI data from the USB3.0 driver
4	getFullPicBuffer	Obtain a visual frame in Full-Picture mode
5	getFullPicMat	Obtain the mat form of full picture buffer.
6	getEventPicBuffer	Obtain a visual frame in Event mode
7	getEventPicMat	Obtain the mat form of each event type picture buffer.
8	getOpticalFlowPicBuffer	Obtain the optical flow raw frame buffer.
9	getOpticalFlowPicMat	Obtain the mat form of frame buffer.
10	getEventDataVector	Get vector of event data in each frame time.
11	getIMUData	Obtain the IMU data packets
12	setSensorFixedMode	Set the fixed working mode of CeleX-5 sensor
13	getSensorFixedMode	Obtain the fixed working mode of CeleX-5 sensor
14	setFpnFile	Configure FPN used for creating a visual frame
15	generateFPN	Generate FPN
16	setClockRate	Set the clock rate of the sensor
17	getClockRate	Obtain the clock rate of the sensor
18	setThreshold	Configure the threshold value where the event triggers (the
19	getThreshold	Get the threshold value for triggering an event
20	setBrightness	Configure the brightness
21	getBrightness	Get the brightness
22	setISOLevel	Set the ISO level which is related to image contrast and dynamic range
23	getISOLevel	Obtain the ISO level which related to image contrast and dynamic range
24	getFullPicFrameTime	Obtain the time of generating a full-frame picture when the CeleX-5 sensor works in *Full-frame Picture Mode*
25	setEventFrameTime	Set the frame time when CeleX-5 sensor is in the *Event Mode*
26	getEventFrameTime	Obtain the frame time when CeleX-5 sensor is in the *Event Mode*

27	setOpticalFlowFrameTime	Set the time of generating a full-frame picture when the CeleX-5 sensor works in *Full-frame Optical-flow Mode*
28	getOpticalFlowFrameTime	Obtain the time of generating a full-frame optical-flow picture when the CeleX-5 sensor works in *Full-frame Optical-flow Mode*
29	reset	Reset the sensor and clear the data in the FIFO buffer
30	pauseSensor	Pause the data output of all sensors
31	restartSensor	Restart the data output of all sensors
32	stopSensor	Stop the data output of all sensors
33	setSensorAttribute	Set the attribute of the CeleX-5 Sensor (Master or Slave)
34	getSensorAttribute	Obtain the attribute of the CeleX-5 Sensor (Master or Slave)
35	getSerialNumber	Obtain the sensor device serial number which is unique
36	getFirmwareVersion	Obtain the firmware version of the sensor
37	getFirmwareDate	Obtain the release date of the used firmware
38	setEventShowMethod	Set the image frame construction way
39	getEventShowMethod	Obtain the way of image frame construction
40	setRotateType	Set the rotation type of the image
41	getRotateType	Obtain rotation type of the image
42	setEventCountStepSize	Set the count step size of the Event Count Pic image
43	getEventCountStepSize	Obtain the count step size of the Event Count Pic image
44	setEventDataFormat	Set the event data format to be used
45	getEventDataFormat	Obtain the event data format have been used
46	startRecording	Start to record the raw data of sensor and saved as bin format
47	stopRecording	Stop to record the raw data of sensor
48	openBinFile	Open the bin file in the user-specified directory
49	readBinFileData	Read data from the opened bin file
50	getBinFileAttributes	Obtain attributes of the bin file.
51	setRowDisabled	Modify the resolution of the image output by CeleX-5 Sensor (Disable outputting data for the specified rows)

2.3.1 openSensor

bool CeleX5::openSensor(DeviceType type)

Parameters

[in] **type** The device type of CeleX-5 sensor.

Returns

The state whether the CeleX-5 sensor is successfully started up.

This method is used to start up the CeleX-5 sensor. If the currently used CeleX-5 chipset supports serial output, type is *CeleX5_MIPI*, if it supports parallel output is supported, type is *CeleX5_Parallel*. For more details about type, please refer to the explanation of *DeviceType*.

```
#include <celex5/celex5.h>
{
    CeleX5 *pCeleX = new CeleX5;
    if (pCeleX == NULL)
        return 0;
    pCeleX->openSensor(CeleX5::CeleX5_MIPI);
}
```

See also

isSensorReady

2.3.2 isSensorReady

bool CeleX5::isSensorReady(int device_index)

Parameters

[in] **device_index** 设备的索引值（如果只连接单个设备，该参数可忽略，索引默认设置为 0，当连接多个设备时，则需要通过 **device_index** 来进行区分，0 表示 Master Sensor，1 表示 Slave Sensor）

Returns

The state of the CeleX-5 sensor

This method is used to check whether the CeleX-5 sensor is successfully started up. It returns true if the sensor is ready, or it returns false.

See also

openSensor

2.3.3 getMIPIData

void CeleX5::getMIPIData(vector<uint8_t> &buffer, int device_index)

void CeleX5::getMIPIData(vector<uint8_t> &buffer, std::time_t& **time_stamp_end**,
vector<IMURawData>& **imu_data**, int **device_index)**

Parameters

[out] **buffer** The vector buffer to store the data have been read.

[out] **time_stamp_end** The timestamp when the packet was obtained.

[out] **imu_data** IMU data packet

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to read MIPI data from the USB3.0 driver.

2.3.4 getFullPicBuffer

void getFullPicBuffer(unsigned char* buffer, int device_index)

Parameters

[in] **buffer** The frame buffer (size is 1280 * 800) in *Full-frame Picture Mode*.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to obtain a full-frame picture buffer when sensor works in the *Full-frame Picture Mode*. You can get the buffer if the sensor is opened successfully. For example,

```
#include <opencv2/opencv.hpp>
#include <celex5/celex5.h>
int main()
{
    CeleX5 *pCeleX = new CeleX5;
    if (NULL == pCeleX)
        return 0;
    pCeleX->openSensor(CeleX5::CeleX5_MIPI);
    pCeleX->setFpnFile(FPN_PATH);
    pCeleX->setSensorFixedMode(CeleX5::Full_Picture_Mode);
    int imgSize = 1280 * 800;
    unsigned char* pBuffer1 = new unsigned char[imgSize];
    while (true)
    {
        if (sensorMode == CeleX5::Full_Picture_Mode)
        {
            //get fullpic when sensor works in Full-frame Picture Mode
            pCeleX->getFullPicBuffer(pBuffer1); //full pic
```

```
                cv::Mat matFullPic(800, 1280, CV_8UC1, pBuffer1);
                cv::imshow("FullPic", matFullPic);
                cvWaitKey(10);
            }
        }
}
```

See also

getFullPicMat

2.3.5 getFullPicMat

cv::Mat CeleX5::getFullPicMat(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The frame buffer (cv::Mat(800, 1280)) in *Full-frame Picture Mode*.

This method is used to obtain a full-frame picture mat. It is similar to call *getFullPicBuffer*, for example,

```
#include <opencv2/opencv.hpp>
#include <celex5/celex5.h>
int main()
{
    CeleX5 *pCeleX = new CeleX5;
    if (pCeleX == NULL)
        return 0;
    pCeleX->openSensor(CeleX5::CeleX5_MIPI);
    pCeleX->setFpnFile(FPN_PATH);
    pCeleX->setSensorFixedMode(CeleX5:: Full_Picture_Mode);
    while (true)
    {
        if (sensorMode == CeleX5::Full_Picture_Mode)
        {
```

```
            if (!pCeleX->getFullPicMat().empty())
            {
                cv::Mat fullPicMat = pCeleX->getFullPicMat();
                cv::imshow("FullPic", fullPicMat);
                cv::waitKey(10);
            }
        }
    }
}
```

See also

getFullPicBuffer

2.3.6 getEventPicBuffer

void getEventPicBuffer(unsigned char*** buffer,** emEventPicType type, int **device_index)**

Parameters

[in] **type** The event frame buffer (size is 1280 * 800) according to the type you specify.

[in] **type** The event picture type.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to obtain an event picture buffer by the given event pic type when the sensor works in *Event Mode*. For more details, please refer to the explanation of *emEventPicMode*.

```
#include <opencv2/opencv.hpp>
#include <celex5/celex5.h>
int main()
{
    CeleX5 *pCeleX = new CeleX5;
    if (NULL == pCeleX)
        return 0;
    pCeleX->openSensor(CeleX5::CeleX5_MIPI);
    pCeleX->setFpnFile(FPN_PATH);
    pCeleX->setSensorFixedMode(CeleX5::Event_Address_Only_Mode);
    int imgSize = 1280 * 800;
```

```
    unsigned char* pBuffer1 = new unsigned char[imgSize];
    while (true)
    {
        if (sensorMode == CeleX5:: Event_Address_Only_Mode
        {
            //get buffers when sensor works in Event Mode
            pCeleX->getEventPicBuffer(pBuffer1 , CeleX5::EventBinaryPic);
            cv::Mat matEventPic(800, 1280, CV_8UC1, pBuffer1);
            cv::imshow("Event-EventBinaryPic", matEventPic);
            cvWaitKey(10);
        }
    }
}
```

See also

getEventPicMat

2.3.7 getEventPicMat

cv::Mat **CeleX5::**getEventPicMat(emEventPicType **type**, int **device_index**)

Parameters

[in] **type** The event picture type.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The event frame buffer (cv::Mat(800, 1280)) according to the type you specify.

This method is used to obtain an event picture mat by the given event pic type when the sensor works in *Event Mode*. This method can obtain seven different types of pictures include binary, gray, count etc. For more details about these types, please refer to *emEventPicType*.

See also

getEventPicBuffer

2.3.8 getOpticalFlowPicBuffer

void CeleX5::getOpticalFlowPicBuffer(unsigned char* **buffer,**

emFullPicType **type**,

int **device_index)**

Parameters

[in] **buffer** The Optical-Flow frame buffer (size is 1280 * 800).

[in] **type** The full-frame optical flow type.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to obtain the Optical-flow frame buffer. For more type of the Optical-Flow, see *emFullPicType*.

For example,

```
#include <opencv2/opencv.hpp>
#include <celex5/celex5.h>
int main()
{
    CeleX5 *pCeleX = new CeleX5;
    if (NULL == pCeleX)
        return 0;
    pCeleX->openSensor(CeleX5::CeleX5_MIPI);
    pCeleX->setSensorFixedMode(CeleX5::Full_Optical_Flow_1_Mode);
    int imgSize = 1280 * 800;
    unsigned char* pOpticalFlowBuffer = new unsigned char[imgSize];
    while (true)
    {
        //get optical-flow data when sensor works in EventMode
        //optical-flow raw data - display gray image
        pCeleX->getOpticalFlowPicBuffer(pOpticalFlowBuffer, CeleX5::
Full_Optical_Flow_Pic);
        cv::Mat matOpticalRaw(800, 1280, CV_8UC1, pOpticalFlowBuffer);
        cv::imshow("Optical-Flow Buffer - Gray", matOpticalRaw);
        cvWaitKey(10);
    }
    return 1;
```

```
}
```

See also

getOpticalFlowPicMat

2.3.9 getOpticalFlowPicMat

cv::Mat **CeleX5::** getOpticalFlowPicMat (**emFullPicType type**, int **device_index**)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The optical flow frame buffer (cv::Mat(800, 1280)).

This method returns the optical flow picture buffer in cv::Mat form when it is called. For more type of the Optical-Flow, see *emFullPicType*.

See also

getOpticalFlowPicBuffer

2.3.10 getEventDataVector

bool CeleX5::getEventDataVector(std::vector<EventData>& **data**,
int **device_index)**

bool CeleX5::getEventDataVector(std::vector<EventData> &**vector,**
uint64_t& **frameNo**,
int **device_index)**

Parameters

[out] **data** The vector of event data at each frame time.

[out] **frameNo** The frame number of the event data vector.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

If the vector of event data is not empty, return true. Otherwise it will return false**.**

This method is used to obtain vector of event data at each frame time. Default frame time is 30ms. Each event data contains rows, columns, brightness and time information. For more details, please refer to the explanation of *EventData***.** It can be used in real-time or in the offline bin files.

2.3.11 getIMUData

int CeleX5::getIMUData(std::vector<IMUData>& **data**, int **device_index)**

Parameters

[out] **data** Obtained IMU data packets.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The number of IMU data packets actually obtained.

This method is used to obtain the IMU data packets. For a detailed explanation of the IMU data packet, see *IMUData*.

2.3.12 setSensorFixedMode

void CeleX5::setSensorFixedMode(CeleX5Mode **mode**, int **device_index)**

Parameters

[in] **mode** The fixed working mode of CeleX-5 sensor.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to set the fixed working mode of CeleX-5 sensor including *Event Address-only Mode, Event Optical-flow Mode, Event Intensity Mode, Full-frame Picture Mode, Single Full-frame Optical-flow Mode and Multiple Full-frame Optical-flow Mode*. For more details, please refer to the explanation of *CeleX5Mode.*

See also

getSensorFixedMode

2.3.13 getSensorFixedMode

CeleX5Mode CeleX5::getSensorFixedMode(int **device_index)**

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The fixed working mode of CeleX-5 sensor.

This method is used to obtain the fixed working mode of CeleX-5 sensor.

See also

setSensorFixedMode

2.3.14 setFpnFile

bool CeleX5::setFpnFile(const string &fpnFile, int **device_index)**

Parameters

[in] **fpnFile** The directory path and file name of FPN file required.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The state of loading FPN file.

This method is used to set the FPN path, then the API will use this FPN to calculate the full frame picture. It returns true if the FPN file load successfully, or it returns false.

See also

generateFPN

2.3.15 generateFPN

void CeleX5::generateFPN(std::string fpnFile, int device_index)

Parameters

[in] **fpnFile** The directory path and file name of generated FPN file to be saved.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to generate FPN file. Normally we are intending to name the generated FPN as "FPN.txt" and save it in the directory where the execution application is running.

FPN, known as Fixed Pattern Noise, is the term given to a particular noise pattern on digital imaging sensors often noticeable during longer exposure shots where particular pixels are susceptible to giving brighter intensities above the general background noise.

See also

setFpnFile

2.3.16 setClockRate

void CeleX5::setClockRate(uint32_t value, int device_index)

Parameters

[in] **value** The clock rate of the CeleX-5 sensor, unit is MHz.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to set the clock rate of the sensor. By default, the CeleX-5 sensor works at 100 MHz and the range of clock rate is from 20 to 100, step is 10.

See also

getClockRate

2.3.17 getClockRate

uint32_t CeleX5::getClockRate(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The clock rate of the CeleX-5 sensor, unit is MHz

This method is used to obtain the clock rate of the CeleX-5 sensor. The range of clock rate is from 20 to 100.

See also

setClockRate

2.3.18 setThreshold

void CeleX5::setThreshold(uint32_t **value**, int **device_index)**

Parameters

[in] **value** Threshold value.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to configure the threshold value where the event triggers (the light intensity change of a pixel exceeds this threshold, the pixel can be marked as an event or active pixel). The large the threshold value is, the less pixels that the event will be triggered (or less active pixels). It could be adjusted from 50 to 511, and the default value is 171.

The threshold value only works when the CeleX-5 sensor is in the *Event Mode*, however, the Sensor still outputs a complete image regardless of the threshold value when it works in *Full-frame Picture Mode*.

See also

getThreshold

2.3.19 getThreshold

uint32_t CeleX5::getThreshold(int **device_index)**

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

Threshold value.

This method is used to get threshold value where the event triggers.

See also

setThreshold

2.3.20 setBrightness

void CeleX5::setBrightness(uint32_t value, int **device_index)**

Parameters

[in] **value** The register value associated with the brightness of the image.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to configure register parameter, which controls the brightness of the image CeleX-5 sensor generated. It could be adjusted from 0 to 1023, and the default value is 150.

See also

getBrightness

2.3.21 getBrightness

uint32_t CeleX5::getBrightness(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The register value associated with the brightness of the image.

This method is used to get the register value associated with the brightness of the image that the CeleX-5 sensor generated.

See also

setBrightness

2.3.22 setISOLevel

void CeleX5::setISOLevel(uint32_t value, int device_index)

Parameters

[in] **value** The register value associated with the contrast of the image.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to adjust the ISO level which is related to image contrast and dynamic range. The larger the value, the lower the contrast and the higher the dynamic range. If the ISO level is modified, it needs to regenerate the FPN. It could be adjusted from 1 to 4, and the default value is 2.

See also

getISOLevel

2.3.23 getISOLevel

uint32_t CeleX5::getISOLevel(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

ISO level

This method is used to obtain the ISO level which related to image contrast and dynamic range.

See also

setISOLevel

2.3.24 getFullPicFrameTime

uint32_t CeleX5::getFullPicFrameTime(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The time of generating a full-frame picture, unit is ms.

This method is used to obtain the time of generating a full-frame picture when the CeleX-5 sensor works in *Full-frame Picture Mode*.

2.3.25 setEventFrameTime

void CeleX5::setEventFrameTime(uint32_t msec, int device_index)

Parameters

[in] **msec** The frame time of *Event Mode*, unit is ms.

This method is used to set the frame time when CeleX-5 sensor is in the *Event Mode*. It modifies the frame length when the software creates event frames without changing the hardware parameters.

See also

getEventFrameTime

2.3.26 getEventFrameTime

uint32_t CeleX5::getEventFrameTime(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The frame time of *Event Mode*, unit is ms.

This method is used to obtain the frame time when CeleX-5 sensor is in the *Event Mode*.

See also

setEventFrameTime

2.3.27 setOpticalFlowFrameTime

void CeleX5::setOpticalFlowFrameTime(uint32_t msec, int device_index)

Parameters

[in] **msec** The time of generating a full-frame optical-flow picture, unit is ms.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to set the time of generating a full-frame picture when the CeleX-5 sensor works in *Full-frame Optical-flow Mode*. It changes the hardware parameters. The default value is

30ms, which means the sensor could generate about 33 full-frame optical-flow pictures per second. For more details about this time, please refers section 1.2.1.5.

See also

getOpticalFlowFrameTime

2.3.28 getOpticalFlowFrameTime

uint32_t CeleX5::getOpticalFlowFrameTime(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The time of generating a full-frame optical-flow picture, unit is ms.

This method is used to obtain the time of generating a full-frame optical-flow picture when the CeleX-5 sensor works in *Full-frame Optical-flow Mode*.

See also

setOpticalFlowFrameTime

2.3.29 reset

void CeleX5::reset (int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to reset the sensor and clear the data in the FIFO buffer.

2.3.30 pauseSensor

void CeleX5::pauseSensor()

This method is used to pause the data output of all Sensors.

2.3.31 restartSensor

void CeleX5::restartSensor()

This method is used to restart the data output of all Sensors.

2.3.32 stopSensor

void CeleX5::stopSensor()

This method is used to stop the data output of all Sensors.

2.3.33 setSensorAttribute

void CeleX5::setSensorAttribute(SensorAttribute attribute, int device_index)

Parameters

[in] **attribute** Sensor Attribute (Master or Slave)

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to set the attribute of the CeleX-5 Sensor. You can specify the device_index device to be set to Master or Slave.

See also

getSensorAttribute

2.3.34 getSensorAttribute

SensorAttribute CeleX5::getSensorAttribute(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

Sensor Attribute (Master or Slave)

This method is used to obtain the attribute of the CeleX-5 Sensor.

See also

setSensorAttribute

2.3.35 getSerialNumber

std::string CeleX5::getSerialNumber (int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

Sensor serial number

This method is used to obtain the sensor device serial number which is unique.

See also

getFirmwareVersion

getFirmwareDate

2.3.36 getFirmwareVersion

std::string CeleX5::getFirmwareVersion (int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The firmware version of the sensor.

This method is used to obtain the firmware version of the sensor.

See also

getSerialNumber

getFirmwareDate

2.3.37 getFirmwareDate

std::string CeleX5::getFirmwareDate(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The release date of the used firmware.

This method is used to obtain the release date of the used firmware.

See also

getSerialNumber

getFirmwareVersion

2.3.38 setEventShowMethod

void CeleX5::setEventShowMethod(EventShowType type, int value, int device_index)

Parameters

[in] **type** The way of image frame construction.

[in] **value** Parameters corresponding to different frame construction methods.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to set the image frame construction way. Currently, there are three ways of creating frames by time, creating frames by number of events, and creating frame by rounds of scanning (one round means that sensor scans from top to bottom). The settings corresponding to the value parameter are time (us), number, and number of rounds of scanning.

See also

getEventShowMethod

2.3.39 getEventShowMethod

EventShowType CeleX5::getEventShowMethod(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The way of image frame construction.

This method is used to obtain the way of image frame construction.

See also

setEventShowMethod

2.3.40 setRotateType

void CeleX5::setRotateType(int type, int device_index)

Parameters

[in] **type** Rotate type

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to set the rotation type of the image. It can be used to flip the image up and down or flip the image left and right.

See also

getRotateType

2.3.41 getRotateType

int CeleX5::getRotateType(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

Rotate type

This method is used to obtain rotation type of the image.

See also

setRotateType

2.3.42 setEventCountStepSize

void CeleX5::setEventCountStepSize(uint32_t size, int device_index)

Parameters

[in] **size** step

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to set the count step size of the Event Count Pic image. Since the number the event triggered is small in some locations, a better visible image can be obtained by increasing the step size.

See also

getEventCountStepSize

2.3.43 getEventCountStepSize

uint32_t CeleX5::getEventCountStepSize(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

Event count step size.

This method is used to obtain the count step size of the Event Count Pic image.

See also

setEventCountStepSize

2.3.44 setEventDataFormat

void CeleX5::setEventDataFormat(int format, int device_index)

Parameters

[in] **format** The envet data format to be used.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to set the event data format to be used.

See also

getEventDataFormat

2.3.45 getEventDataFormat

int CeleX5::getEventDataFormat(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The event data format to be used.

This method is used to obtain the event data format have been used.

See also

setEventDataFormat

2.3.46 startRecording

void CeleX5::startRecording(std::string filePath, int device_index)

Parameters

[in] **filePath** The directory path to save the bin file

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

This method is used to start recording the raw data of the sensor and save it as a bin file. The type in which data will be saved depends on which mode Sensor is working in.

See also

stopRecording

2.3.47 stopRecording

void CeleX5::stopRecording()

This method is used to stop recording the raw data of the sensor.

See also

startRecording

2.3.48 openBinFile

bool CeleX5::openBinFile(string filePath, int device_index)

Parameters

[in] **filePath** The directory path and name of the bin file to be played.

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The value whether the bin file is opened.

This method is used to open the bin file in the user-specified directory. It returns true if the bin file opens successfully, otherwise it returns false.

See also

readBinFileData

2.3.49 readBinFileData

bool CeleX5::readBinFileData(int device_index)

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

The value whether the bin is read over.

This method is used to read data from the opened bin file. If reaching the end of the bin file it will return true, otherwise it will return false. Before reading the bin file, you need to open the file first.

See also

openBinFile

2.3.50 getBinFileAttributes

BinFileAttributes CeleX5:: getBinFileAttributes(int **device_index)**

Parameters

[in] **device_index** Sensor device index, 0 means Master Sensor, 1 means Slave Sensor.

Returns

A structure of the file attributes**.**

This method is used to obtain the attributes of the bin file. It will return a structure which includes hour, minute, second, mode, length, and clock rate when we call this method. For more details, please refer to the explanation of *BinFileAttributes*.

2.3.51 setRowDisabled

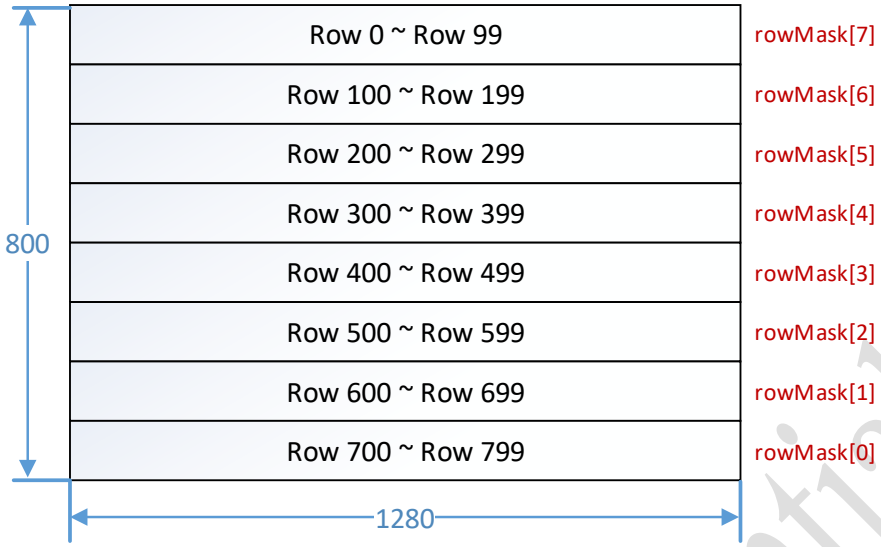
void setRowDisabled(uint8_t rowMask);

Parameters

[out] **rowMask** Bit parameters for the specified rows

This method is used to modify the resolution of the image output by CeleX-5 Sensor (Disable outputting data for the specified rows).

As shown in the figure below, the default image resolution of the CeleX-5 Sensor is 1280*800, which can be divided into 8 separate horizontal blocks. The size of each block is 1280*100, and each block has a corresponding control bit (rowMask[7] to rowMask[0] from top to bottom). For example, if you want to close the top 100 rows, just set rowMask[7] to 1 and the other control bits to 0, that is, set rowMask to 128 (b'10000000).



3. Appendix

Table 3-1: Sensor Operation Mode Control Parameters

Addr	Name	Width	Default	Note
50	SWITCH_RESET_GAPA	[7:0]	200	Set gap time during mode switching
51	SWITCH_RESET_GAPB	[7:0]	250	Set gap time during mode switching
52	SWITCH_RESET_GAPC	[7:0]	200	Set gap time during mode switching
53	SENSOR_MODE_1	[2:0]	0	The operation mode in fixed mode, or the first operation mode in loop mode
54	SENSOR_MODE_2	[2:0]	3	The second operation mode in loop mode
55	SENSOR_MODE_3	[2:0]	4	The third operation mode in loop mode
57	EVENT_DURATION	[7:0]	20	Duration of event mode when sensor operates in loop mode -- Low byte
58	EVENT_DURATION	[1:0]	0	Duration of event mode when sensor operates in loop mode -- High byte
59	PICTURE_NUMBER_1	[7:0]	1	Number of pictures to acquire in Mode_D
60	PICTURE_NUMBER_2	[7:0]	1	Number of pictures to acquire in Mode_E
61	PICTURE_NUMBER_3	[7:0]	1	Number of pictures to acquire in Mode_F
62	PICTURE_NUMBER_4	[7:0]	3	Number of pictures to acquire in Mode_G
63	PICTURE_NUMBER_5	[7:0]	3	Number of pictures to acquire in Mode_H
64	SENSOR_MODE_SELECT	[0]	0	Sensor operation mode select: 0: fixed mode / 1: loop mode

Table 3-2: Sensor Data Transfer Parameters

Addr	Name	Width	Default	Note
70	EXTERNAL_DATA	[5:0]	0	Data from external sensor --- high byte
71	EXTERNAL_DATA	[7:0]	0	Data from external sensor --- middle byte
72	EXTERNAL_DATA	[7:0]	0	Data from external sensor --- low byte
73	EVENT_PACKET_SELECT	[1:0]	2	Event packet format select
74	MIPI_PIXEL_NUM_EVENT	[7:0]	254	Number of pixels in one row at event mode = 4*(this value+1)
75	ADC_RESOLUTION_SEL	[0]	1	Sensor ADC resolution select: 1: 12bit / 0: 8bit
76	MIPI_PIXEL_NUM_FRAME	[6:0]	5 (fixed)	Number of pixels in one row at full-frame mode --- high byte
77	MIPI_PIXEL_NUM_FRAME	[7:0]	0 (fixed)	Number of pixels in one row at full-frame mode --- low byte
78	MIPI_DATA_SOURCE_SELECT	[0]	1	For event mode, MIPI data source select: 1: internal data / 0: external data
79	MIPI_ROW_NUM_EVENT	[7:0]	0	Number of rows in one frame at event mode --- high byte
80	MIPI_ROW_NUM_EVENT	[7:0]	200	Number of rows in one frame at event

				mode --- low byte
81	MIPI_VIR_CHANNEL_ID	[5:0]	0	MIPI parameter, virtual channel ID
82	MIPI_HD_GAP_FULLFRAME	[2:0]	2	In full-frame mode, the interval between the last long packet and EOF -- high byte
83	MIPI_HD_GAP_FULLFRAME	[7:0]	132	In full-frame mode, the interval between the last long packet and EOF -- low byte
84	MIPI_HD_GAP_EVENT	[2:0]	2	In event mode, the interval between the last long packet and EOF -- high byte
85	MIPI_HD_GAP_EVENT	[7:0]	89	In event mode, the interval between the last long packet and EOF -- low byte
86	MIPI_GAP_EOF_SOF	[2:0]	0	High byte
87	MIPI_GAP_EOF_SOF	[7:0]	100	Low byte