celepixel

CelePixel

CeleX™ Chipset

SDK Reference

CelePixel Technology Co. Ltd.

Content

Version Control

Version	Date	Section	Description	Author
1.0	2017.11.07	All	New	Guoping He
1.1	2017.12.11	All	Modified the format of Documentation	Xiaoqin Hu
1.2	2017.12.15	1.2	Added working mode of CeleX™ Sensor	Xiaoqin Hu
1.2	2017.12.15	1.3	Added how to use CeleX™ section	Xiaoqin Hu
1.2	2017.12.15	1.4	Added method of FPN generation section	Xiaoqin Hu
1.2	2017.12.15	2.2.6	Deleted SetCallBack interface Added getPixelData interface	Xiaoqin Hu
1.2	2017.12.15	2.2.8	Modified getEventPicBuffer interface	Xiaoqin Hu
1.2	2017.12.15	2.2.9	Added generateFPN interface	Xiaoqin Hu
1.2	2017.12.15	2.2.10	Added setFpnFile interface	Xiaoqin Hu
1.2	2017.12.15	2.2.11	Added setSensorMode interface	Xiaoqin Hu
1.2	2017.12.18	2.2.12	Added getSensorMode interface	Xiaoqin Hu
1.2	2017.12.18	2.2.12	Added isSensorReady interface	Xiaoqin Hu
1.2	2017.12.18	3	Added section 3	Xiaoqin Hu
1.3	2018.01.19	2.2.14 2.2.15	Added setTimeSlice interface Added getTimeSlice interface	Xiaoqin Hu
1.3	2018.01.19	2.2.16 2.2.17	Added setThreshold interface Added getThreshold interface	Xiaoqin Hu
1.3	2018.01.19	2.2.18 2.2.19	Added setContrast interface Added getContrast interface	Xiaoqin Hu
1.3	2018.01.19	2.2.20 2.2.21	Added setBrightness interface Added getBrightness interface	Xiaoqin Hu
1.3	2018.01.19	2.2.22	Added openSensor interface	Xiaoqin Hu
1.3	2018.02.13	All	Updated the Structure of Documentation	Xiaoqin Hu
1.3.1	2018.03.13	2.2.7	Modified the description of the API getEventPicBuffer	Xiaoqin Hu
1.4	2018.03.20	1.2.1	Added introduction of FullPicture_Event mode	Xiaoqin Hu
1.4	2018.03.20	2.2.29	Added setMotionTime interface	Xiaoqin Hu
1.4	2018.03.20	2.2.30	Added getMotionTime interface	Xiaoqin Hu
2.1	2018.05.22	1.2.2	Modified the number of bits in T (19 bits →17 bits)	Xiaoqin Hu
2.1	2018.05.22	1.2.4	Added section Optical-Flow	Xiaoqin Hu
2.1	2018.05.22	2.2.33~2.23 9	Added the interfaces of Optical-Flow	Xiaoqin Hu
2.1	2018.05.22	2.2.31~2.2.3 2	Added the interfaces of Sensor's clock rate	Xiaoqin Hu
2.1	2018.05.26	2.2.31~2.2.3 6	Added interface for setting and getting Frame Time in various data modes.	Xiaoqin Hu
2.1	2018.05.26	2.2.29 2.2.30	Deleted setTimeSlice interface Deleted setTimeSlice interface	Xiaoqin Hu
2.1	2018.05.30	All	Revised the principle description of CeleX™ Sensor	Xiaoqin Hu
2.1	2018.06.12	1.2.1	Revised the principle description of CeleX™ Sensor	Xiaoqin Hu
2.2	2018.08.02	2.2.12 2.2.14	Added getFullPicMat interface Added getEventPicMat interface	Hua Ren
2.2	2018.08.02	2.2.15	Added getEventDataVector interface.	Hua Ren
2.2	2018.08.02	2.2.40	Added setEventFrameParameters interface.	Hua Ren

2.2	2018.08.02	2.2.42 2.2.43	Added setTimeScale interface Added setEventCountStepSize interface	Hua Ren
2.2	2018.08.02	2.2.49 2.2.51 2.2.53	Added getOpticalFlowPicMat nterface Added getOpticalFlowDirectionPicMat interface Added getOpticalFlowSpeedPicMat interface	Hua Ren
2.2	2018.08.02	2.2.58	Added getAttributes interface.	Hua Ren
2.2	2018.08.02	2.2.59	Added convertBinToAVI interface.	Hua Ren
2.2	2018.08.03	2.2.60	Added startRecordingVideo interface	Xiaoqin Hu
2.2	2018.08.03	2.2.61	Added stopRecordingVideo interface	Xiaoqin Hu
2.2	2018.08.03	2.2.62	Added enableAutoAdjustBrightness interface	Xiaoqin Hu
2.2.1	2018.10.08	2.2.63 2.2.64 2.2.65	Added getIMUData interface Added getIMUDataSize interface Added setIMUIntervalTime interface	Xiaoqin Hu
2.2.1	2018.10.08	2.2.66 2.2.67	Added denoisingByTimeInterval interface Added denoisingAndCompresing interface	Hua Ren

1. Overview

1.1. Introduction of CeleX™ Chipset

1.1.1 Basic working principle of CeleX™ Chipset

Fig. 1-1 shows the basic working principle of the CeleX™ Chipset. Applications could configure the FPGA and eventually configure the sensor through the WIRE IN utility API provided by the Opal Kelly FPGA board. Similarly, Applications are able to configure those Control Registers residing in the Opal Kelly FPGA board. For more information about Opal Kelly, please refer to Opal Kelly's website by the following address:

https://library.opalkelly.com/library/FrontPanelAPI/classokCFrontPanel.html#a01d90a0ac728cf88b9637ab2b78546b5

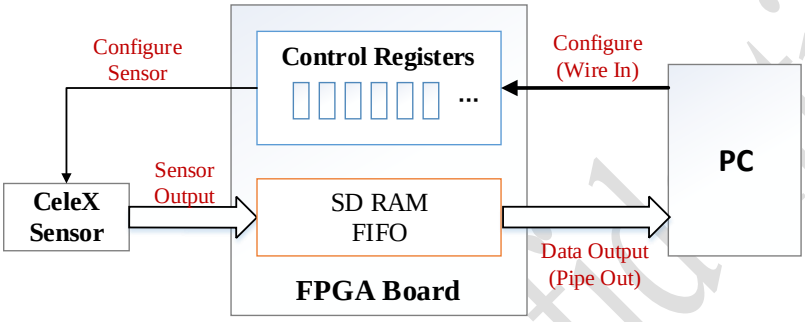


Fig. 1-1 Basic working principle of the CeleX™ Chipset

The working principle of the sensor and data format are illustrated in Section 1.2.

Please refer to Section 3 for the configuration of Control Registers.

1.1.2 Terminology

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

No.	Terminology	Description
1	Full-Picture Mode	A working mode of the Celex™ Sensor, in which the Sensor outputs the information for each pixel in order (top to bottom, left to right) within a certain period of time.
2	Event Mode	A working mode of the Celex™ Sensor, in which the Sensor only detects the pixel whose intensity has changed and then marks it as an active pixel and outputs it.
3	FullPic-Event Mode	A working mode of the Celex™ Sensor, in which the Sensor alternately outputs of Full-Picture and Event data.
4	Optical-Flow Mode	A working mode of the Celex™ Sensor, in which the Sensor outputs optical flow information.
5	Full Frame Transmission Time	The time it takes for a Celex™ Sensor to transmit a Full-Picture.

6	Time Block	A time period in the FPGA used to mark the timestamp of each event. It will be reset when it counts to T = 2^17.
7	FullPic Frame Time	The frame time of the Full-Picture data
8	Event Frame Time	The frame time of the Event data
9	FullPic-Event Frame Time	The frame time of the Full-Event data
13	Full Pic	The full frame output from the SDK
14	Event Binary Pic	The event binary frame output from the SDK
15	Event Gray Pic	The event fray frame from the SDK
16	Event Accumulated Gray Pic	The event accumulated gray frame output from the SDK

1.2. Working Principle of CeleX™ Sensors

1.2.1. CeleX™ Sensor's Working Modes

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.

This SDK provides three working modes of CeleX™ Sensors: *Full-Picture data*, *Event data*, and *Optical-Flow data*. *Full-Picture and Event data output alternately to create FullPic-Event data.*

1.2.1.1. **Full-Picture Mode**

Regarding of the Full-Picture mode (as shown in Fig.1-2), the Sensor could output the information of each pixel sequentially, from top to bottom then left to right. Similar to the conventional image cameras, we could also obtain a frame of Full Picture.

In the Full-Picture mode, the Full Frame Transmission Time of a frame is fixed. If the operating frequency of the Sensor is 25 MHz, the time is about 16 ms.

The FullPic Frame Time can be adjusted by software (range: 20~320ms). You can modify this time by calling the API *setFullPicFrameTime*. The method of creating Full-Picture frame is described in section 1.2.3.1.

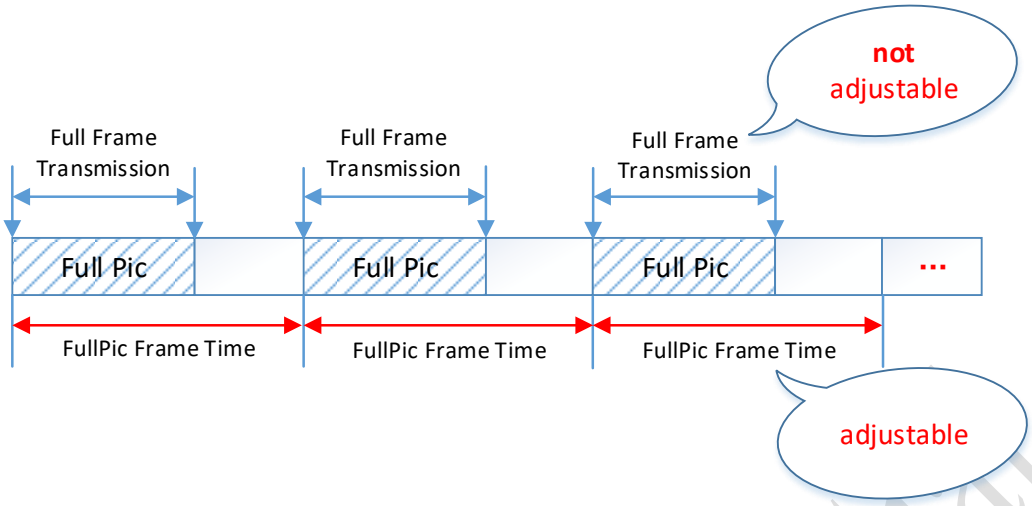


Fig. 1-2 Full-Picture data mode

1.2.1.2. **Event Mode**

Regarding of the Event mode (as shown in Fig.1-3), the Sensor only acquires the values of active pixels. Here, the pixels whose intensities have changed are marked as active pixels.

For this mode, it could obtain six 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 value of active pixels are marked as 255, while inactive pixels are 0), event gray frame and so on.

In the Event mode, there are two frame-creating methods, namely, overlapped mode and non-overlapped mode, as shown in Figure 1-3. In the case of overlapped mode, each frame of data is completely new, and the data between frames does not overlap. In the case of non-overlapped mode, except for the first frame, each subsequent frame of data is superimposed. You can set the time by calling the API *setOverlapTime*. The maximum value it can adjust is the Frame Time of the previous frame.

In the Event mode, the Time Block is fixed (it is a time period in the FPGA, T = 2^17). Similarly, 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 1.2.3.2.

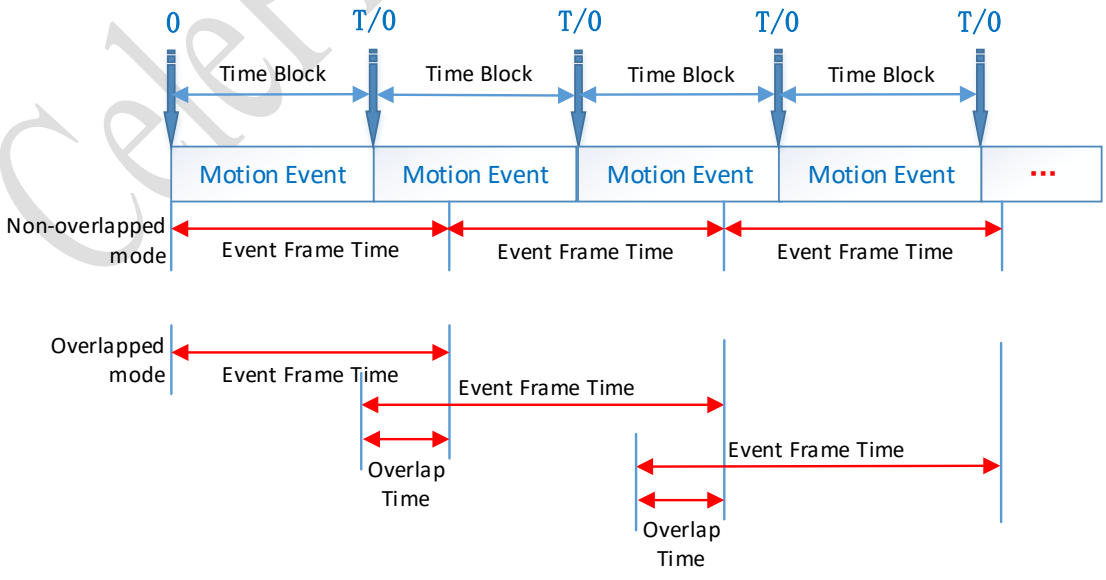


Fig. 1-3 Event data mode

1.2.1.3. **FullPic-Event Mode**

Regarding of the FullPic-Event mode (as shown in Fig.1-4), the Sensor could output Full-Picture and Event data alternately. That is, in each frame time, the Sensor will output one frame of Full-Picture data in sequence, followed by the output of Event data.

Similarly, Full Frame Transmission Time and Time Block are not adjustable, but the Frame Time can be adjusted (range: 40 to 1000 ms). You can modify this time by calling the API *setFEFrameTime*.

In this mode, users can obtain Full-Picture and Event pic frames at the same time. The method of creating Full-Picture frames is the same as the normal Full-Picture mode. See Section 1.2.3.1 for details. The method for creating Event frames is the same as the Event mode, see section 1.2.3.2.

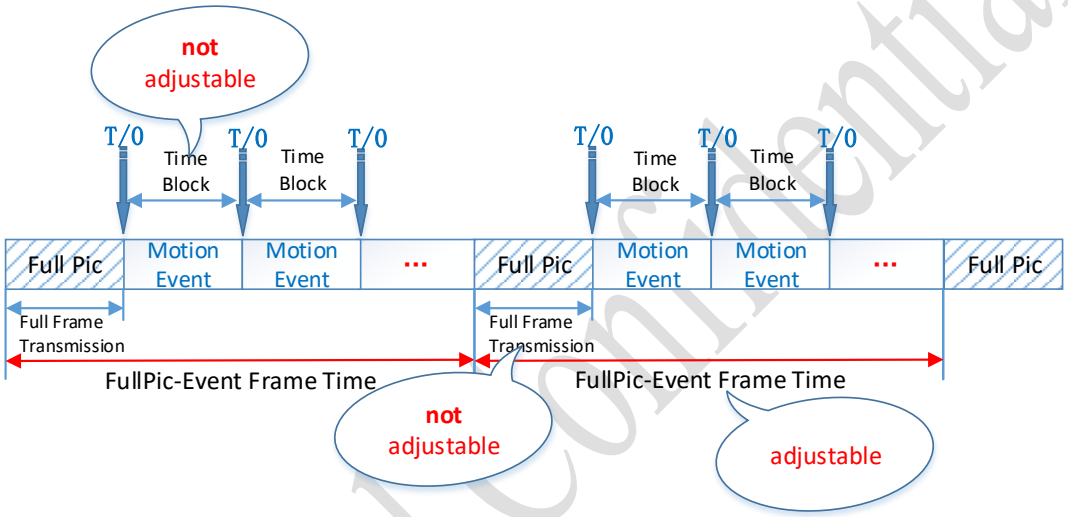


Fig. 1-4 FullPic-Event data mode

1.2.1.4. **Optical-Flow Mode**

When the Optical Flow data mode is selected in the SDK, the SDK will use Event data in the Event data mode to perform Optical Flow analysis which is described in Section 1.2.4.

1.2.1.5. **Mode Switching**

In terms of API application, both the two working modes of CeleX™ Sensors could be switched by calling the API *setSensorMode*, which is described in Section 2.2.10.

1.2.2. Data Format

The output of CeleX™ Chipset is a stream of pixel events. These events contain the information of pixel locations/coordinates (X, Y), the sample value of absolute brightness when the pixel event is triggered (A) and activation time (T). See Table 1-1 below for the detailed explanations of the notations X, Y, A and T.

Table 1-1 Detailed explanations of the notations X, Y, A and T

T continuously increases until it gets reset signal at the end of each time block. A more intuitive description of T is given in Fig. 1-5.

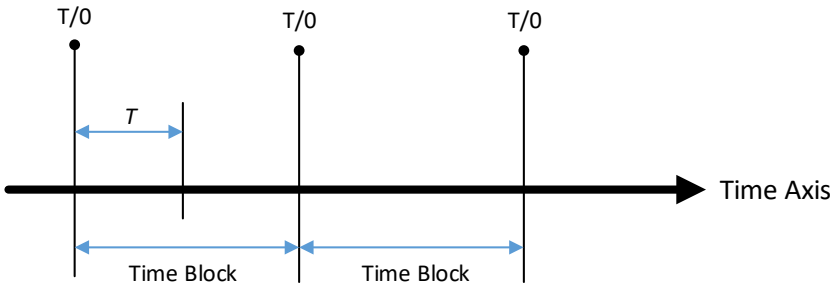


Fig. 1-5 Time Block

CeleX™ Chipset outputs three types of events, including Row Events, Column Events and Special Events. Row Events and Column Events will both carry partial data about X, Y, A and T. While Special Events indicate the end of a time block.

Fig. 1-6 shows the format of outputting row events and column events between two Special Events. Table 1-2 below shows the data structure and format of the event type.

As shown in the table, all the events are represented by 4-byte data. An event would be a Column Event if all first bits of each byte in that event are 0.

Table 1-2 Data structure and format of the event types

Event Type	**Size**	**Description**	**Comments**
Row Event	4 bytes (32 bits)	4 bytes Byte 0: {1'b1, Y[6:0]} Byte 1: {1'b1, Y[9,7], T[3:0]} Byte 2: {1'b1, T[10:4]} Byte 3: {2'b10, T[16:11]}	Row Events carry row address (Y) and the activation time (T) for the pixels on that row
Column Event	4 bytes (32 bits)	4 bytes Byte 0: {1'b0, X[6:0]} Byte 1: {1'b0, C[0], X[8,7], A[3:0]} Byte 2: {3'b000, A[8:4]} Byte 3: {8'b00000000}	Column Events carry the column address (X) and follow right after a row event. They carry the sample value of absolute brightness when the pixel event is triggered (A). The timing information can be retrieved from the preceding Row Event.
Special Event	4 bytes (32 bits)	4 bytes Byte 0: {8'b11111111} Byte 1: {8'b11111111} Byte 2: {8'b11111111} Byte 3: {8'b11111111}	Special Events represent the end of a time block. After this event, timer count resets to 0. The duration of the time block is configurable.

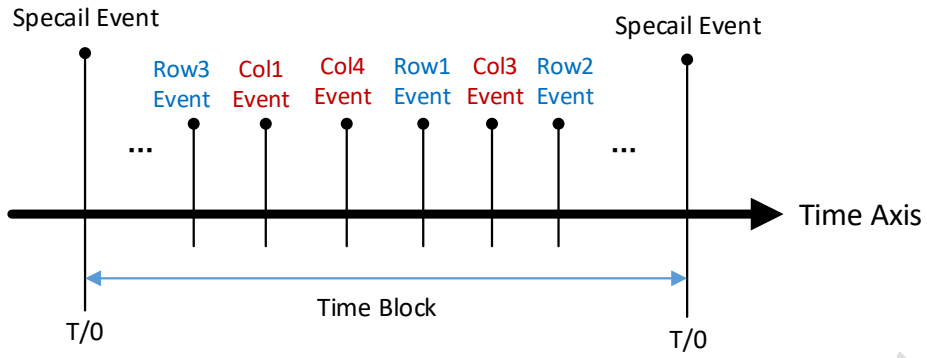


Fig. 1-6 The format of outputting row events and column events between two Special Events.

Notes:

- Byte 0 means the 1st byte in the event, Byte 1 means the 2nd byte in the event, and so are the third and forth.
- For Row Event, the 1st bit of each byte is set to be "1", and the 10-bit row address (Y) is comprised by Y[6:0] in the 1st byte and Y[9:7] in the 2nd byte. The 17-bit activation time (T) is comprised by T[3:0] in the 2nd byte, T[10:4] in the 3rd byte and T[16:11] in the 4th byte.
- For Column Event, the 1st bit of each byte is set to be "0", and the X[6:0] in the 1st byte and Y[8:7] in the 2nd byte are used to form a 9-bit column address(X). The 9-bit brightness (A) is comprised by A[3:0] in the 2nd byte and A[8:4] in the 3rd byte. The unused bit in the 4th byte is marked as "0".
- Besides, there's a column left/right half plane indicator in byte 1 of a Column Event, C[0]. When this control bit is "1", the actual column coordinate will be (767 - X). Then the actual column coordinate will be X when the control bit is "0".
- For Special Event, all 32 bits are marked as "1".
- Row/Column event could be easily distinguished by reading the 1st bit of that event data.

1.2.3. Methods to Create Full Picture and Event Frame

CeleX™ 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.

1.2.3.1. Method to Create Full Picture Frame

1) Call API *setFullPicFrameTime* to set the Full-Picture frame time, 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[640][768], which has 640 rows and each row consists of 768 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.

1.2.3.2. **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[640][768]，M2[640][768] and M3[640][768] 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 FPGA. 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. 1-7 and 1-8.

Fig. 1-7 and 1-8 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. 1-7 and event accumulated gray frame in Fig. 1-8.

“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).

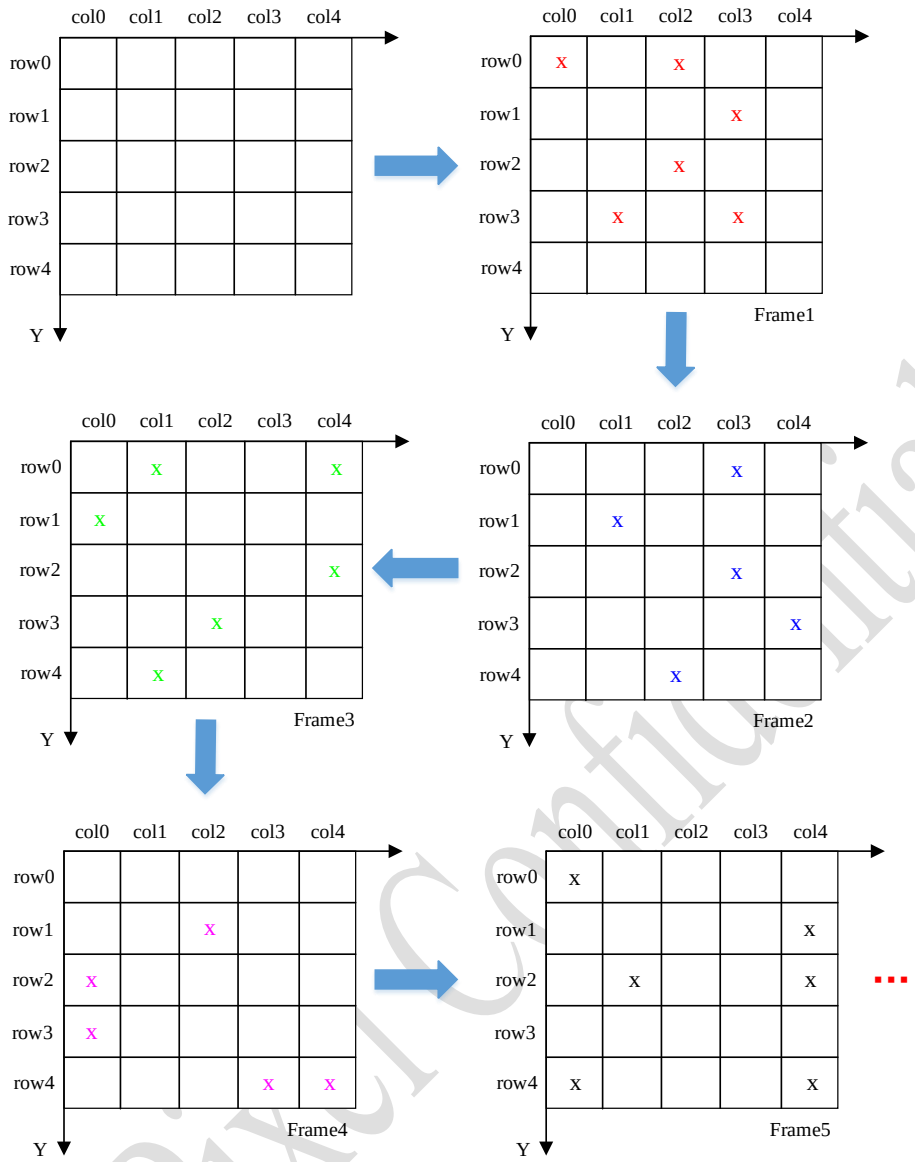


Fig. 1-7 Event Gray Pic in Event mode

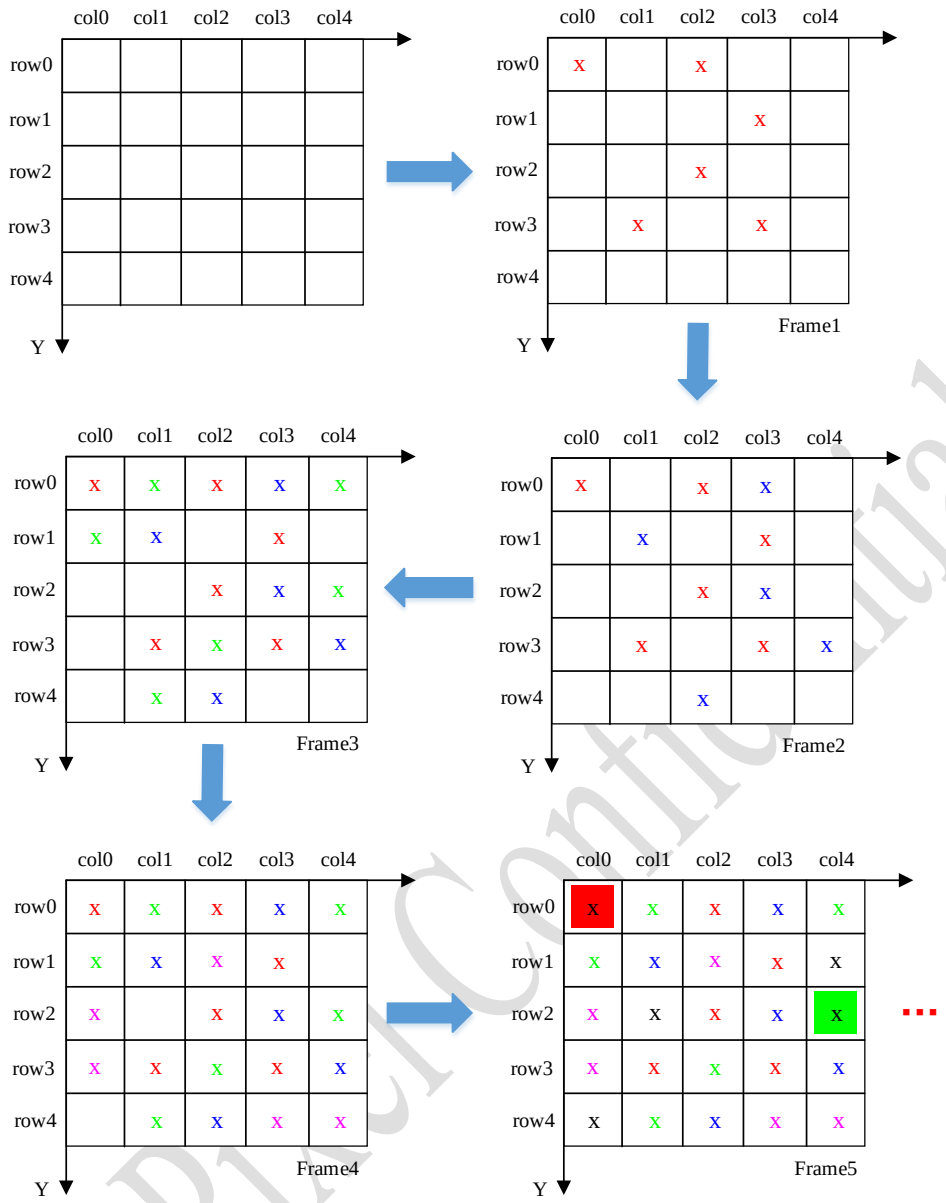


Fig. 1-8 Event Accumulated Gray Pic in Event mode

1.2.4. Optical Flow Data

Optical flow is to split the pixels changed in a certain time interval into N time slices in chronological order, and then set a weight value for each slice (which is a gray value corresponding to the image). An image frame with pixel weight value is then retrieved.

You can call *enableOpticalFlow* interface to enable the Optical Flow function. By default, this function is disabled. By calling the *geOpticalFlowDirectionPicBuffer* and *getOpticalFlowSpeedPicBuffer* interfaces, you can obtain Optical Flow data, which is the speed and direction of each pixel. The specific process is as follows:

1) The data of the Event mode is cut into frames on the time axis, and there may be overlaps between the frames in a certain time domain, as shown in the following Optical-Flow Frame1, Optical-Flow Frame2;

2) Slicing sequentially within sustaining time of each frame (Latency). Each slice does not overlap with each other. Each Latency in this SDK has 255 slices.
3) Assign values in ascending order for data points that fall in different time slices according to the time sequence. The data points that belong to the same slice are assigned the same value. For example, the weights assigned to pixels in Slice1 are all 1, pixels that fall in Slice2 are given a weight of 2, and so on. Pixels that fall in Slice 255 have a weight of 255;
4) Map these weights to the M[640][768] array to obtain an Optical-Flow weighted frame, where a larger value indicates that the changing moment of pixel occurs later, the smaller the earlier.
5) Optical-Flow is identified in the space as shown in Figure 1-10 below. With the trajectory of the object, a slope is formed whose direction is the same as the direction of movement of the object. The slope is also related to the speed of movement. The faster the object moves, the smaller the slope. Two built-in algorithms for extracting the slope and its aspect are provided, namely *getOpticalFlowDirectionPicBuffer* and *getOpticalFlowSpeedPicBuffer*. Users can also develop their own algorithms to obtain speed and direction information.

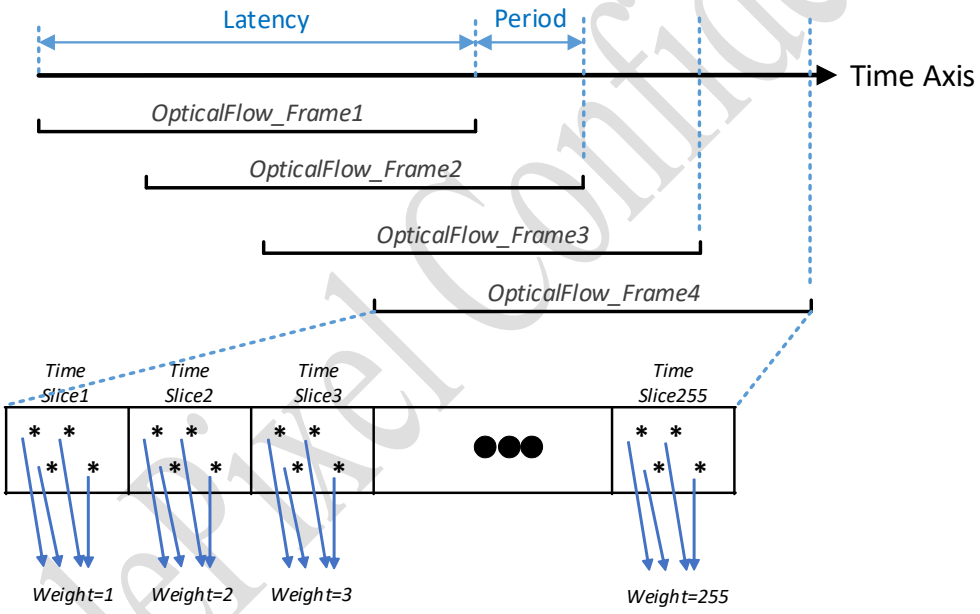


Fig. 1-9 The basic principle of Optical Flow

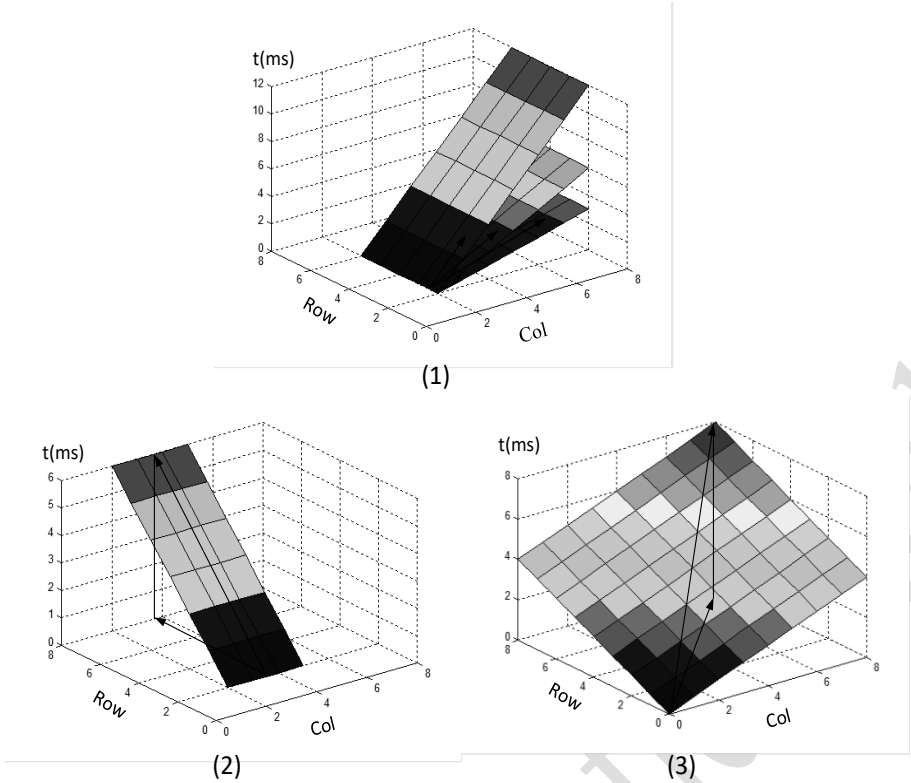


Fig. 1-10 The slope and aspect of Optical-Flow

2. CeleX™ API Reference

2.1. Overview

The CeleX API provide a C++ interface to communicate with CeleX™ 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 CeleX4 which encapsulated communication the CeleX™ Sensor. You can first call *openSensor* to open the sensor that has connected to your PC, then call *pipeOutFPGAData* to pipe out the data from the FPGA FIFO, and finally 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 CeleX4DataManager and override its virtual *onFrameDataUpdated* methods to receive the notification that the frame buffers are ready.

In version 2.2, **dvs** namespace was added to place the image processing classes and functions. Notice that earlier versions of the library used classes outside of this namespace.

Here are enums and structs with brief descriptions:

num emSensorMode {
FullPictureMode = 0,
EventMode = 1,
FullPic_Event_Mode = 2
};
	FullPictureMode - A working mode of the Celex™ Sensor, in which the Sensor outputs the information for each pixel in order (top to bottom, left to right) within a certain period of time.
	EventMode - A working mode of the Celex™ Sensor, in which the Sensor only detects the pixel whose intensity has changed and then marks it as an active pixel and outputs it.
	FullPic_Event_Mode - A working mode of the Celex™ Sensor, in which the Sensor alternately outputs of Full-Picture and Event data.

enum emEventPicMode { EventBinaryPic = 0, EventAccumulatedPic = 1, EventGrayPic = 2, EventSuperimposedPic = 3, EventDenoisedBinaryPic = 4, EventDenoisedGrayPic = 5, EventCountPic = 6, EventDenoisedByTimeBinaryPic	**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) 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

= 7, EventDenoisedByTimeGrayPic = 8 };	Gray value(0). **EventSuperimposedPic** - The frame buffer (matrix) that superimposes event binary picture onto event accumulated picture in Event mode and full picture in FullPic-Event mode. **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. **EventDenoisedGrayPic** - The frame buffer (matrix) is similar to the event gray buffer, except that it is a denoised event gray buffer with a simple custom algorithm. **EventCountPic** - The event count frame buffer (matrix). **EventDenoisedByTimeBinaryPic** - Denoised event binary buffer with a time interval based algorithm. **EventDenoisedByTimeGrayPic** - Denoised event gray buffer with a time interval based algorithm.
typedef struct BinFileAttributes { int hour; int minute; int second; emSensorMode mode; long length; }BinFileAttributes;	**hour, minute, second** - The time length of the bin file recording. **mode** - The sensor mode for bin file recording. **length** - The length of the bin file.
typedef struct EventData { uint16_t col; uint16_t row; uint16_t brightness; uint32_t t; int8_t p; }EventData;	**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. **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;	**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

double y_ACC; double z_ACC; uint32_t t_ACC; double x_GYROS_OFST; double y_GYROS_OFST; double z_GYROS_OFST; uint32_t t_GYROS_OFST; double x_ACC_OFST; double y_ACC_OFST; double z_ACC_OFST; uint32_t t_ACC_OFST; uint64_t frameNo; } IMUData;	**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_GYROS_OFST** - Offset of the angular rate in the X-axis (gyroscopes) **y_GYROS_OFST** - Offset of the angular rate in the Y-axis (gyroscopes) **z_GYROS_OFST** - Offset of the angular rate in the Z-axis (gyroscopes) **t_GYROS_OFST** - Timestamp when angular rate offset is received **x_ACC_OFST** - Offset of the acceleration in the X-axis (accelerometer) **y_ACC_OFST** - Offset of the acceleration in the Y-axis (accelerometer) **z_ACC_OFST** - Offset of the acceleration in the Z-axis (accelerometer) **t_ACC_OFST** - Timestamp when acceleration offset is received **frameNo** - Frame number

2.2. CeleX4 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:

API Name	Description
openSensor	Start sensor via calling this interface
isSensorReady	Check whether CeleX™ Sensor is successfully initialized
isSdramFull	Check whether data is blocked in SD RAM
setSensorMode	Configure working mode of CeleX™ Sensor
getSensorMode	Get the working mode of CeleX™ Sensor
setFpnFile	Configure FPN used for creating a visual frame
generateFPN	Generate FPN

pipeOutFPGAData	Read data from FPGA
getFPGADataSize	Get the number of bytes in the FPGA FIFO
readDataFromFPGA	Read the specified length of data in the FPGA buffer
getFullPicBuffer	Acquire a visual frame in Full-Picture mode
getFullPicMat	Obtain the mat form of full picture buffer.
getEventPicBuffer	Acquire a visual frame in Event mode
getEventPicMat	Obtain the mat form of each event type picture buffer.
getEventDataVector	Get vector of event data in each frame time.
setThreshold	Configure the threshold value where the event triggers (the change of light intensity happen to a pixel is greater than the threshold, the pixel can be marked as an event/active pixel)
getThreshold	Get the threshold value for triggering an event
setContrast	Configure the contrast
getContrast	Return the contract
setBrightness	Configure the brightness
getBrightness	Get the brightness
setLowerADC	Configure the lower limit of variable brightness range
getLowerADC	Get the lower limit of ADC value
setUpperADC	Configure the higher limit of variable brightness range
getUpperADC	Get the upper limit of ADC value
resetFPGA	Clear the data in the FPGA FIFO buffer
resetSensorAndFPGA	Disable the sensor and clear the data in the FPGA FIFO buffer
enableADC	Enable or Disable output of light intensity by a configurable parameter
trigFullPic	Force to generate a visual frame at Motion mode, by capturing and sending out all the pixels' value at the time
setClockRate	Set the clock rate of the Sensor
getClockRate	Obtain the clock rate of the Sensor
setFullPicFrameTime	Set the frame time when CeleX™ Sensor is in the Full-Picture mode
getFullPicFrameTime	Obtain the frame time when CeleX™ Sensor is in the Full-Picture mode
setEventFrameTime	Set the frame time when CeleX™ Sensor is in the Event mode
getEventFrameTime	Obtain the frame time when CeleX™ Sensor is in the Event mode
setFEFrameTime	Set the frame time when CeleX™ Sensor is in the FullPic-Event mode

getFEFrameTime	Obtain the frame time when CeleX™ Sensor is in the FullPic-Event mode
setOverlapTime	Set the overlap time between the current frame and the previous frame when creating Event frame
getOverlapTime	Obtain the overlap time
setEventFrameParameters	Set event frame time and interval time before getting frame buffer.
setFrameTimeRange	Set the start ratio and the end ratio of the event frame to the event length
setTimeScale	Set scale for time remapping.
setEventCountStepSize	Set step size of the event count image.
enableOpticalFlow	Enable or Disable the function Optical-Flow
isOpticalFlowEnabled	Check whether the Optical-Flow function is enabled
setOpticalFlowLatencyTime	Set the latency time of optical flow frame, the default value is 100ms
setOpticalFlowSliceCount	Set the number of optical flow slices, the default value is 255
getOpticalFlowPicBuffer	Obtain the optical flow raw frame buffer.
getOpticalFlowPicMat	Obtain the mat form of frame buffer.
getOpticalFlowDrirectionPicBuffer	Obtain the direction frame buffer of each pixel calculated on the optical flow raw frame. In order to display it on the image directly, we map the angel value from [0,359] to the [0,255]
getOpticalFlowDirectionPicMat	Obtain the mat form of direction frame buffer.
getOpticalFlowSpeedPicBuffer	Obtain the speed frame buffer of each pixel calculated on the optical flow raw frame. In order to display it on the image directly, we map the speed to the [0,255]
getOpticalFlowSpeedPicMat	Obtain the mat form of speed frame buffer.
startRecording	Start to record the raw data of sensor and saved as bin format
stopRecording	Stop to record the raw data of sensor
openPlaybackFile	Open the bin file in the user-specified directory
readPlayBackData	Read data from the opened bin file
getAttributes	Obtain attributes of the bin file.
convertBinToAVI	Convert the bin file to video.
startRecordingVideo	Start to record the processed frame data of the sensor and save it as a video
stopRecordingVideo	Stop to record video data of the sensor
enableAutoAdjustBrightness	Enable or disable the auto-adjust image brightness function
getIMUData	Obtain the IMU data packets

getIMUDataSize	Obtain the number of IMU data already stored in the vector
setIMUIntervalTime	Set the time interval at which the Sensor sends IMU data packets
denoisingByTimeInterval	Obtain binary pic buffer after denoising.
denoisingAndCompresing	Obtain gray pic buffer after denoising and compressing.

2.2.1 openSensor

CeleX4::ErrorCode CeleX4::openSensor(string str)

Parameters

[in] **str** The name of CeleX™ Sensor

Returns

NoError - Sensor is open correctly.

InitializeFPGAFailed - Fail to initialize the CeleX™ sensor.

PowerUpFailed - Fail to power up the CeleX™ sensor.

ConfigureFailed - Fail to configure the CeleX™ sensor.

This method is used to start up the CeleX™ sensor. When multiple Sensors work together (please configure their names before opening sensor). If there is only one Sensor device, it is okay to pass an empty string in the name of the sensor. For example,

```
//create a CeleX object
CeleX *pCelex = new CeleX;
//now, open the sensor device
if (pCelex != NULL)
    pCelex->openSensor("");
```

See also

isSensorReady

2.2.2 isSensorReady

bool CeleX4::isSensorReady()

Returns

The state of the CeleX™ Sensor.

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

See also

openSensor

2.2.3 isSdramFull

bool CeleX4::isSdramFull()

Returns

The state of the SDRAM.

This method is used to check whether data is blocked in SDRAM or not. If data is blocked, the data cannot be read out from FPGA correctly. It returns true if the SDRAM is blocked, or it returns false.

2.2.4 setSensorMode

void CeleX4::setSensorMode(emSensorMode mode)

Parameters

[in] **mode** The working mode of CeleX™ Sensor.

This method is used to set the working mode of CeleX™ Sensor including FullPictureMode, EventMode and FullPic_Event_Mode. For more details, please refer to the explanation of *enSensorMode*.

See also

getSensorMode

2.2.5 getSensorMode

emSensorMode CeleX4::getSensorMode()

Returns

The working mode of CeleX™ Sensor.

This method is used to obtain the working mode of CeleX™ Sensor.

See also

setSensorMode

2.2.6 setFpnFile

bool CeleX4::setFpnFile(const string &fpnFile)

Parameters

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

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

2.2.7 generateFPN

void CeleX4::generateFPN(std::string fpnFile)

Parameters

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

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

2.2.8 pipeOutFPGAData

void CeleX4::pipeOutFPGAData()

This method is used to read data from FPGA. You need to pipe out the data from FPGA regularly to avoid blocking FPGA, as the image data is written into FPGA continuously.

See also

getFPGADataSize

readDataFromFPGA

2.2.9 getFPGADataSize

long CeleX4::getFPGADataSize()

Returns

The number of bytes in the FPGA FIFO.

This method is used to get the number of bytes in the FPGA FIFO.

See also

pipeOutFPGAData

readDataFromFPGA

2.2.10 readDataFromFPGA

long CeleX4::readDataFromFPGA(long length, unsigned char *data)

Parameters

[in] **length** The number of bytes to be read.

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

Returns

The number of bytes read or -1 if the read failed.

This method is used to read the specified length of data in the FPGA buffer.

See also

pipeOutFPGAData

getFPGADataSize

2.2.11 getFullPicBuffer

unsigned char *CeleX4::getFullPicBuffer()

Returns

The frame buffer (matrix) in Full-Picture mode.

This method is used to obtain a FullPic buffer in Full-Picture or FullPic-Event mode. You can get the buffer if the sensor is opened successfully and the data is piped out. For example,

```
//now, open the sensor device
if (pCelex != NULL)
    pCelex->openSensor("");
pCelex->setSensorMode(FullPictureMode);//set sensor mode
pCelex->setFpnFile("FPN.txt");//set FPN file
while (true)
{
    pCelex->pipeOutFPGAData();
    //get the fullpic buffer
    unsigned char* pFullPicBuffer = pCelex->getFullPicBuffer();
}
```

See also

getFullPicMat

2.2.12 getFullPicMat

cv::Mat CeleX4::getFullPicMat()

Returns

The frame buffer (cv::Mat) in Full-Picture mode.

This method returns the full picture buffer in cv::Mat form when it is called. This method is similar to get the fullpic buffer, for example,

```
//now, open the sensor device
if (pCelex != NULL)
    pCelex->openSensor("");
pCelex->setSensorMode(FullPictureMode);//set sensor mode
pCelex->setFpnFile("FPN.txt");//set FPN file
while (true)
{
    pCelex->pipeOutFPGAData();
    //get the fullpic mat
```

```
    cv::Mat fullPicMat = pCelex->getFullPicMat();
}
```

See also

getFullPicBuffer

2.2.13 getEventPicBuffer

unsigned char *CeleX4::getEventPicBuffer(emEventPicMode mode)

Parameters

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

Returns

The event frame buffer (matrix) according to the type you specify.

This method is used to obtain various frames for the Sensor to work in Event mode. For more details, please refer to the explanation of *emEventPicMode*. You can get the event frame buffer in the specified picture mode. For example,

```
//now, open the sensor device
if (pCelex != NULL)
    pCelex->openSensor("");

pCelex->setFpnFile("FPN.txt");//set FPN file

while (true)

{

    pCelex->pipeOutFPGAData();

    //get the event buffer in the form of binary pic
    unsigned char* pEventBuffer = pCelex->getEventPicBuffer(EventBinaryPic);

}
```

See also

getEventPicMat

2.2.14 getEventPicMat

cv::Mat **CeleX4::**getEventPicMat(emEventPicMode **picMode**)

Parameters

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

Returns

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

This method gets corresponding cv::Mat form of the event picture buffer when it is called with a certain mode provided. This method can obtain seven different types of pictures include binary, gray, count etc. For more details about these types, please refer to *emEventPicMode*.

See also

getEventPicBuffer

2.2.15 getEventDataVector

bool CeleX4::getEventDataVector(std::vector<EventData>& data)

Parameters

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

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 60ms. 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. For example,

```
//now, open the sensor device
if (pCelex != NULL)
    pCelex->openSensor("");
while (true)
{
    pCelex->pipeOutFPGAData(); //get event data in real-time
    std::vector<EventData> v;  //vector to store the event data
    pCelex->getEventDataVector(v); //get the event data
}
```

2.2.16 setThreshold

void CeleX4::setThreshold(uint32_t value)

Parameters

[in] **value** Threshold value.

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 25 to 200.

The threshold value only works when the Celex™ Sensor is in the Event mode, however, the Sensor still outputs a complete image regardless of the threshold value when it works in Full-Picture mode.

See also

getThreshold

2.2.17 getThreshold

uint32_t CeleX4::getThreshold()

Returns

Threshold value.

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

See also

setThreshold

2.2.18 setContrast

void CeleX4::setContrast(uint32_t value)

Parameters

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

This method is used to configure register parameter, which controls the contrast of the image Celex™ Sensor generated. It could be adjusted from 0 to 1023.

See also

getContrast

2.2.19 getContrast

uint32_t CeleX4::getContrast()

Returns

The register value associated with the contrast of the image.

This method is used to get the register value associated with the contrast of the image that the CelexTM Sensor generated.

See also

setContrast

2.2.20 setBrightness

void CeleX4::setBrightness(uint32_t value)

Parameters

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

This method is used to configure register parameter, which controls the brightness of the image Celex™ Sensor generated. It could be adjusted from 0 to 1023.

See also

getBrightness

.

2.2.21 getBrightness

void CeleX4::getBrightness(uint32_t value)

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™ Sensor generated.

See also

setBrightness

2.2.22 setLowerADC

void CeleX4::setLowerADC(uint32_t value)

Parameters

[in] **value** The lower limit of ADC value.

This method is used to set the lower limit of ADC value that is the sample value of absolute brightness when the pixel event is triggered. If the ADC value is less than the lower limit, it will be assigned to 0, whereas the other values between the lower and upper limit will be linearly mapped to [0,255].

See also

getLowerADC

2.2.23 getLowerADC

uint32_t CeleX4::getLowerADC()

Returns

The lower limit of ADC value.

This method is used to get the lower limit of ADC value that is the sample value of absolute brightness when the pixel event is triggered.

See also

setLowerADC

2.2.24 setUpperADC

void CeleX4::setUpperADC(uint32_t value)

Parameters

[in] **value** The lower limit of ADC value.

This method is used to set the upper limit of ADC value that is the sample value of absolute brightness when the pixel event is triggered. If the ADC value exceeds the upper limit, it will be assigned to 255, whereas the other values between the lower and upper limit will be linearly mapped to [0,255].

See also

getUpperADC

2.2.25 getUpperADC

uint32_t CeleX4::getUpperADC()

Returns

The upper limit of ADC value.

This method is used to get the upper limit of ADC value that is the sample value of absolute brightness when the pixel event is triggered.

See also

setUpperADC

2.2.26 resetFPGA

void CeleX4::resetFPGA()

This method is used to clear the data in the FPGA FIFO buffer.

2.2.27 resetSensorAndFPGA

void CeleX4::resetSensorAndFPGA()

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

2.2.28 enableADC

void CeleX4::enableADC(bool enable)

Parameters

[in] **enable** The value to control whether or not to output the ADC.

This method is used to enable or disable output of light intensity by a configurable parameter. True means enable and false means disable.

2.2.29 trigFullPic

void CeleX4::trigFullPic()

This method is used to generate a visual frame at Event mode compulsorily by capturing and sending out all the pixels' value at a time.

2.2.30 setClockRate

void CeleX4::setClockRate(uint32_t value)

Parameters

[in] **value** The clock rate of the Celex™ Sensor, unit is MHz

This method is used to set the clock rate of the Sensor. By default, the Celex™ sensor works at 25 MHz and the range of clock rate is from 2 to 50.

See also

getClockRate

2.2.31 getClockRate

uint32_t CeleX4::getClockRate()

Returns

The clock rate of the Celex™ Sensor, unit is MHz

This method is used to obtain the clock rate of the Celex™ Sensor. The range of clock rate is from 2 to 50.

See also

setClockRate

2.2.32 setFullPicFrameTime

void CeleX4::setFullPicFrameTime(uint32_t msec)

Parameters

[in] **msec** The frame time of Full-Picture, unit is ms.

This method is used to set the frame time when CeleX™ Sensor is in the Full-Picture mode. It changes the hardware parameters of the sensor, and the sensor will output FullPic frames according to the frame length.

See also

getFullPicFrameTime

2.2.33 getFullPicFameTime

uint32_t CeleX4::getFullPicFrameTime()

Returns

The frame time of Full-Picture, unit is ms.

This method is used to obtain the frame time when CeleX™ Sensor is in the Full-Picture mode.

See also

setFullPicFrameTime

2.2.34 setEventFrameTime

void CeleX4::setEventFrameTime(uint32_t msec)

Parameters

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

This method is used to set the frame time when CeleX^{TM} 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

2.2.35 getEventFrameTime

uint32_t CeleX4::getEventFrameTime()

Returns

The frame time of Event mode, unit is ms.

This method is used to obtain the frame time when CeleX^{TM} Sensor is in the Event mode.

See also

setEventFrameTime

2.2.36 setFEFrameTime

void CeleX4::setFEFrameTime(uint32_t msec)

Parameters

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

This method is used to set the frame time when CeleX^{TM} Sensor is in the FullPic-Event mode. It changes the hardware parameters of the sensor, then the sensor will output a FullPic frame and a period of time event data according to this frame length. Since the time of outputting a FullPic frame is fixed, it actually modifies the length of outputted event data when users modify the frame time of FullPic-Event mode.

See also

getFEFrameTime

2.2.37 getFEFrameTime

uint32_t CeleX4::getFEFrameTime()

Returns

The frame time of FullPic-Event, unit is ms.

This method is used to obtain the frame time when CeleX^{TM} Sensor is in the FullPic-Event mode.

See also

setFEFrameTime

2.2.38 setOverlapTime

void CeleX4::setOverlapTime(uint32_t msec)

Parameters

[in] **msec** The overlapping time, unit is ms.

This method is used to set the overlapping time between the current frame and the previous frame when creating Event frame. The argument msec is the time in milliseconds.

See also

getOverlapTime

setEventFrameParameters

2.2.39 getOverlapTime

uint32_t CeleX4::getOverlapTime()

Returns

The overlapping time of a frame.

This method is used to obtain the overlapping time. The return value is the time in milliseconds.

See also

setOverlapTime

2.2.40 setEventFrameParameters

void CeleX4::setEventFrameParameters(uint32_t frameTime,

uint32_t intervalTime

)

Parameters

[in] **frameTime** Frame time for creating event buffer.

[in] **intervalTime** Interval time of two event frame data.

This method is used to set frame time and interval time for event buffer in Event mode. You can also set the overlapping time by adjusting the frame time and interval time through this method. There is no overlap if frame time equals to interval time. Otherwise, the overlapping time is the difference between frame time and interval time. These two arguments are both the time in milliseconds.

2.2.41 setFrameLengthRange

void CeleX4::setFrameLengthRange(float startRatio,

float endRatio

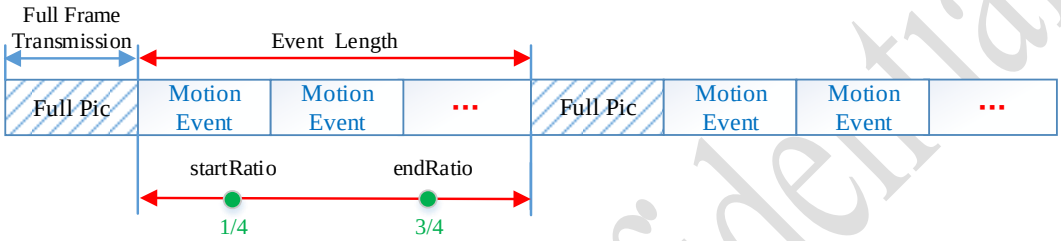
)

Parameters

[in] **startRatio** The ratio of the event frame length to event length.

[in] **endRatio** The ratio of the event frame length to event length.

This method is used to set the start ratio and the end ratio of the event frame to the event length (as shown in the following figure) in the FullPic-Event mode. You can specify the period of event data needed to create frame. The minimum ratio is zero and the maximum ratio is 1.



2.2.42 setTimeScale

void CeleX4::setTimeScale(float scale)

Parameters

[in] **scale** Time scale for time remapping.

This method sets a time scale so that the time of each event data can be redefined. The argument scale is the time in milliseconds.

2.2.43 setEventCountStepSize

void CeleX4::setEventCountStepSize(uint32_t size)

Parameters

[in] **size** The step size of an event count per change.

This method is used to set step size of event count per change. Once the pixel brightness changes, the step will be added once. You can set different step size for better visualization. Default step size is 9.

2.2.44 enableOpticalFlow

void CeleX4::enableOpticalFlow(bool enable)

Parameters

[in] **enable** The state whether the Optical-Flow function is enabled.

This method is used to enable or disable the function Optical-Flow. True means enable, and false means disable. For more details of the Optical-Flow, see section 1.2.4.

See also

isOpticalFlowEnabled

2.2.45 isOpticalFlowEnabled

bool CeleX4::isOpticalFlowEnabled()

Returns

The state whether the Optical-Flow function is enabled.

This method is used to check whether the Optical-Flow function is enabled, for more details of the Optical-Flow, see section 1.2.4.

See also

enableOpticalFlow

2.2.46 setOpticalFlowLatencyTime

void CeleX4::setOpticalFlowLatencyTime(uint32_t msec)

Parameters

[in] **msec** The latency time of Optical-Flow frame, unit is ms.

This method is used to set the latency time of Optical-Flow frame. The default value is 100ms. For more details of the Optical-Flow, see section 1.2.4.

See also

setOpticalFlowSliceCount

2.2.47 setOpticalFlowSliceCount

void CeleX4::setOpticalFlowSliceCount(uint32_t count)

Parameters

[in] **count** The number of Optical-Flow slices

This method is used to set the number of Optical-Flow slices. The default value is 255. For more details of the Optical-Flow, see section 1.2.4.

See also

setOpticalFlowLatencyTime

2.2.48 getOpticalFlowPicBuffer

unsigned char* CeleX4::getOpticalFlowPicBuffer()

Returns

The Optical-Flow raw frame buffer (matrix).

This method is used to obtain the Optical-Flow raw frame buffer. For more details of the Optical-Flow, see section 1.2.4. Before getting the Optical-Flow frame, you need to enable Optical flow. For example,

```
//now, open the sensor device
if (pCelex != NULL)
    pCelex->openSensor("");
pCelex->enableOpticalFlow(true); //enable optical-flow option
while (true)
{
    pCelex->pipeOutFPGAData();
    //get the optical-flow buffer
    unsigned char* pFullPicBuffer = pCelex->getOpticalFlowPicBuffer();
}
```

See also

getOpticalFlowPicMat

getOpticalFlowDirectionPicBuffer

getOpticalFlowSpeedPicBuffer

2.2.49 getOpticalFlowPicMat

cv::Mat **CeleX4::** getOpticalFlowPicMat ()

Returns

The optical flow frame buffer (cv::Mat).

This method returns the optical flow picture buffer in cv::Mat form when it is called. For more details about the Optical-Flow, see section 1.2.4.

See also

getOpticalFlowPicBuffer

getOpticalFlowDirectionPicMat

getOpticalFlowSpeedPicMat

2.2.50 getOpticalFlowDirectionPicBuffer

unsigned char* CeleX4::getOpticalFlowDirectionPicBuffer()

Returns

The direction frame buffer (matrix) of all frame pixels.

This method is used to Obtain the direction frame buffer of each pixel calculated on the Optical-Flow raw frame. In order to display it on the image directly, we map the angle from [0,359] to the [0,255]. For more details of the Optical-Flow, see section 1.2.4.

See also

getOpticalFlowDirectionPicMat

getOpticalFlowPicBuffer

getOpticalFlowSpeedPicBuffer

2.2.51 getOpticalFlowDirectionPicMat

cv::Mat CeleX4::getOpticalFlowDirectionPicMat()

Returns

The direction frame buffer (cv::Mat) of all frame pixels.

This method returns the optical flow direction picture buffer in cv::Mat form when it is called. The direction is calculated on the Optical-Flow raw frame. In order to display it on the image directly, we map the angle from [0,359] to the [0,255]. For more details about the Optical-Flow, see section 1.2.4.

See also

getOpticalFlowDirectionBuffer

getOpticalFlowPicMat

getOpticalFlowSpeedPicMat

2.2.52 getOpticalFlowSpeedPicBuffer

unsigned char* CeleX4::getOpticalFlowSpeedPicBuffer()

Returns

The speed frame buffer (matrix) of all frame pixels.

This method is used to obtain the speed frame buffer of each pixel calculated on the Optical-Flow raw frame. In order to display it on the image directly, we map the speed to the [0,255]. For more details of the Optical-Flow, see section 1.2.4.

See also

getOpticalFlowSpeedPicMat

getOpticalFlowPicBuffer

getOpticalFlowDirectionPicBuffer

2.2.53 getOpticalFlowSpeedPicMat

cv::Mat CeleX4:: getOpticalFlowSpeedPicMat ()

Returns

The speed frame buffer (cv::Mat) of all frame pixels.

This method returns the optical flow speed picture buffer in cv::Mat form when it is called. The speed is calculated on the Optical-Flow raw frame. In order to display it on the image directly, we map the speed to the [0,255]. for more details of the Optical-Flow, see section 1.2.4.

See also

getOpticalFlowSpeedBuffer

getOpticalFlowPicMat

getOpticalFlowDirectionPicMat

2.2.54 startRecording

void CeleX4::startRecording(std::string filePath)

Parameters

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

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

2.2.55 stopRecording

void CeleX4::stopRecording()

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

See also

startRecording

2.2.56 openPlaybackFile

bool CeleX4::openPlaybackFile(string filePath)

Parameters

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

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

readPlayBackData

2.2.57 readPlayBackData

bool CeleX4::readPlayBackData(long length)

Parameters

[in] **length** The number of bytes to be read from the file. The default value is 1968644.

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

openPlaybackFile

2.2.58 getAttributes

BinFileAttributes CeleX4:: getAttributes(std::string& binFile)

Parameters

[in] **binFile** The path of the bin file.

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*.

2.2.59 convertBinToAVI

void CeleX4:: convertBinToAVI (std::string binFile,
emEventPicMode picMode,
uint32_t frameTime,
uint32_t intervalTime,
cv::VideoWriter writer
)

void CeleX4:: convertBinToAVI (std::string binFile,
cv::VideoWriter writer
)

Parameters

[in] **binFile** The path of the bin file.

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

[in] **frameTime** Frame time for creating event buffer.

[in] **intervalTime** Interval time of two event frame data.

[in] **writer** VideoWriter for writing the video.

This method converts bin file to video by processing data in the bin file. The path of bin file should be provided. You can specify the picture mode, frame time and interval time. The frame time and interval time are both the time in milliseconds. For more details about event pic mode, refer to *emEventPicMode*.

Note

Notice that the frame time and interval time are only available for the data in the Event mode. If you want to get the full picture in the Full-Picture mode or FullPic-Event mode, you can use the interface with two parameters.

2.2.60 startRecordingVideo

```
void CeleX4::startRecordingVideo(std::string filePath,
                                 std::string fullPicName,
                                 std::string eventName,
                                 int fourcc,
                                 double fps
                                 )
```

Parameters

[in] **filePath** The path of the output video file.

[in] **fullPicName** The Full Pic data name of the output video file.

[in] **eventName** The Event Pic data name of the output video file.

[in] **fourcc** 4-character code of codec used to compress the frames, opencv parameter

[in] **fps** Framerate of the created video stream, opencv parameter

This method is used to start recording the processed frame data of the sensor and save it as a video. When the Sensor works in FullPic-Event mode, both the Full Pic and Event data are recorded at the same time. In Full-Picture mode, only Full Pic video is recorded and in Event mode, only Event video is recorded.

See also

stopRecordingVideo

2.2.61 stopRecordingVideo

```
void CeleX4::stopRecordingVideo()
```

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

See also

startRecordingVideo

2.2.62 enableAutoAdjustBrightness

void CeleX4::enableAutoAdjustBrightness(bool enable)

Parameters

[in] **enable** The state to enable or disable the auto-adjust image brightness function.

This method is used to enable or disable the auto-adjust image brightness function. By default, this function is not turned on. After this function is enabled, the internal API will automatically call the *setBrightness* to adjust the brightness of the image outputted by the Sensor according to the external light intensity.

See also

setBrightness

2.2.63 getIMUData

int CeleX4::getIMUData(int count, std::vector<IMUData>& data)

Parameters

[in] **count** The number of IMU data packets to obtain.

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

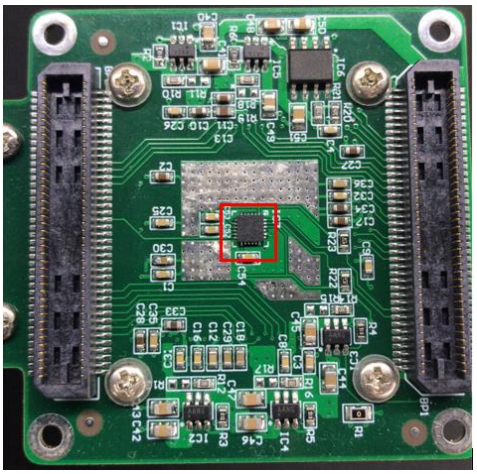
Returns

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*. It returns the entire data in the vector if the specified length is greater than the data size of the vector.

Note

1. Before calling this method, it is necessary to confirm whether there is an IMU hardware module. There are two ways to determine it:
   a) Check the label attached to the Sensor board. If it is the CMH series, it means there is an IMU module.
   b) Separate the Sensor board and the FPGA processing board, and then check if there is a chip marked with a red box in the figure below, and if so, it has an IMU module.



2. In this method, the maximum size of the vector is 1000, when the data size reaches 1000, the earliest IMU data packet will be cleared.

2.2.64 getIMUDataSize

int CeleX4::getIMUDataSize()

Returns

The number of IMU data already stored in the vector.

This method is used to obtain the number of IMU data already stored in the vector.

2.2.65 setIMUIntervalTime

void CeleX4::setIMUIntervalTime(uint32_t value)

Parameters

[in] **value**: The time interval at which IMU data is sent. Unit: 4096*0.04us.

This method is used to set the time interval at which the Sensor sends IMU data packets. The value ranges from 50 to 255 and the default value is 50 (about 8ms).

2.2.66 denoisingByTimeInterval

bool CeleX4::denoisingByTimeInterval(std::vector<EventData> vec, cv::Mat &mat)

Parameters

[in] **vec** The raw event data vector.

[out] **mat** The binary image after denoising.

This method is used to get the binary image after denoising.

2.2.67 denoisingAndCompresing

bool CeleX4::denoisingAndCompresing(std::vector<EventData> vec, float compressRatio, cv::Mat &mat)

Parameters

[in] **vec** The raw event data vector.

[in] **compressRatio** Pre-compression ratio.

[out] **mat** The binary image after denoising and compressing.

This method is used to get the binary image after denoising and compressing.

2.3. CeleX4DataManager 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 "celex.h"
#include "celexdatamanager.h"
#include "celex4processeddata.h"
#include <opencv2/opencv.hpp>
#include <Windows.h>
#define FPN_PATH    "./FPN.txt"
using namespace std;
using namespace cv;
class SensorDataObserver : public CeleX4DataManager
{
public:
    SensorDataObserver(CX4SensorDataServer* pServer)
    {
        m_pServer = pServer;
        m_pServer->registerData(this, CeleX4DataManager::CeleX_Frame_Data);
    }
    ~SensorDataObserver()
    {
        m_pServer->registerData(this, CeleX4DataManager::CeleX_Frame_Data);
    }
    virtual void onFrameDataUpdated(CeleX4ProcessedData* pSensorData);//overrides
Observer operation

    CX4SensorDataServer* m_pServer;
};

void SensorDataObserver::onFrameDataUpdated(CeleX4ProcessedData* pSensorData)
{
    if (NULL == pSensorData)
        return;
    //get buffers when sensor works in EventMode
```

```
    if (pSensorData->getEventBasePic(EventBinaryPic))
    {
        cv::Mat matFullPic(640, 768, CV_8UC1,
pSensorData->getEventPicBuffer(EventAccumulatedPic)); //event accumulative pic
        cv::Mat matEventPic(640, 768, CV_8UC1,
pSensorData->getEventPicBuffer(EventBinaryPic)); //event binary pic
        cv::imshow("FullPic", matFullPic);
        cv::imshow("EventPic", matEventPic);
        cvWaitKey(30);
    }
}

int main()
{
    CeleX* pCelexSensor = new CeleX;
    if (NULL == pCelexSensor)
        return 0;
    pCelexSensor->openSensor("");
    pCelexSensor->setFpnFile(FPN_PATH);
    emSensorMode sensorMode = FullPic_Event_Mode; //EventMode, FullPic_Event_Mode or
FullPictureMode
    pCelexSensor->setSensorMode(sensorMode);
    SensorDataObserver* pSensorData = new
SensorDataObserver(pCelexSensor->getSensorDataServer());
    while (true)
    {
        pCelexSensor->pipeOutFPGAData();
        Sleep(10);
    }
    return 1;
}
```

2.4. CeleX NameSpace Reference

All the image processing classes and functions are placed into the **dvs** namespace. Therefore, to access this functionality from your code, use the **dvs::** specifier or **using namespace dvs**; directive:

```
#include "eventproc.h"
…
dvs::segmentationByMultislice(multislicebyte, ratio, segimage);
…
```

or

```
#include "eventproc.h"
```

```
using namespace dvs;
…
segmentationByMultislice(multislicebyte, ratio, segimage);
…
```

2.4.1 denoisingByNeighborhood

void dvs::denoisingByNeighborhood(const cv::Mat& countEventImg,

cv::Mat& denoisedImg

)

Parameters

[in] **countEventImg** Image of pixel assigned by event count.

[out] **denoisedImg** Binary image after denoising.

This method can be used to denoise the event count image. The image is denoised in both the spatial domain and the spatial domain.

2.4.2 denoisingMaskByEventTime

int dvs::denoisingMaskByEventTime(const cv::Mat& countEventImg,

double timelength,

cv::Mat& denoiseMaskImg

)

Parameters

[in] **countEventImg** Image of pixel assigned by event count.

[in] **timelength** Time length of frame.

[out] **denoisedImg** Binary image after denoising.

This method is used to denoise the event count image. The threshold for denoising is related to the setting of the time length. Noise with a gray value less than the threshold will be eliminated.

3. Appendix

3.1. Control Registers in the Opal Kelly FPGA

Applications are able to configure those Control Registers residing in the Opal Kelly FPGA board. And these Control Registers will configure the behavior of the sensor. Table 3-1 lists the configurable registers used.

Table 3-1 Configurable Registers List

Registor	Bit	Command Name	Address	Value	Mask	Description
0x00	[0]	Reset All	0x00	0x01	0x01	Reset Sensor & FPGA
		Dereset All	0x00	0x00	0x01	
	[1]	Reset FPGA	0x00	0x02	0x02	Reset FPGA
		Dereset FPGA	0x00	0x00	0x02	
	[4]	Forcefire ON	0x00	0x10	0x10	Trigger a full picture
		Forcefire OFF	0x00	0x00	0x10	
	[5]	set GAIN to #value#	0x00	value	0x20	
	[6]	SetADC Enalbe	0x00	0x40	0x40	ADC enable
		SetADC Disable	0x00	0x00	0x40	ADC disable
0x02	[7:0]	set Clock_M to #value#	0x02	value	0x00FF	Clock_M
	[15:8]	set Clock_D to #value#	0x02	value	0xFF00	Clock_D
	[16]	Clock_APPLY OFF	0x02	0x00	0x10000	Clock_APPLY
		Clock_APPLY ON	0x02	0x10000	0x10000	
0x03	[0]	VDD_IO_EN OFF	0x03	0x00	0x01	VDD_IO_EN
		VDD_IO_EN ON	0x03	0x01	0x01	
	[1]	VDD_RB_EN OFF	0x03	0x00	0x02	VDD_RB_EN
		VDD_RB_EN ON	0x03	0x02	0x02	
	[2]	VDD_CORE_EN OFF	0x03	0x00	0x04	VDD_CORE_EN
		VDD_CORE_EN ON	0x03	0x04	0x04	
0x04	[17:8]	set resolution to #value#	0x04	value	0x3FF00	SPI value
		set EVT_VL to #value#	0x04	value	0x3FF00	0x3FF00
		set EVT_VH to #value#	0x04	value	0x3FF00	0x3FF00
		set RAMP- to #value#	0x04	value	0x3FF00	0x3FF00
		set RAMP+ to #value#	0x04	value	0x3FF00	0x3FF00
		set REF- to #value#	0x04	value	0x3FF00	0x3FF00
		set REF+ to #value#	0x04	value	0x3FF00	0x3FF00
		set REF-H to #value#	0x04	value	0x3FF00	0x3FF00

		set REF+H to #value#	0x04	value	0x3FF00	0x3FF00
		set EVT_DC to #value#	0x04	value	0x3FF00	0x3FF00
		set LEAK to #value#	0x04	value	0x3FF00	0x3FF00
		set CDS_DC to #value#	0x04	value	0x3FF00	0x3FF00
		set CDS_V1 to #value#	0x04	value	0x3FF00	0x3FF00
		set PixBias to #value#	0x04	value	0x3FF00	0x3FF00
	[23:20]	set DAC_CHANNEL to #fixed value#	0x03	value	0xF00000	Sensor SPI address
0x05	[0]	DAC_APPLY OFF	0x05	0x00	0x01	DAC_APPLY
		DAC_APPLY ON	0x05	0x01	0x01	
	[1]	SPI_DEFAULT OFF	0x05	0x00	0x02	SPI_DEFAULT
		SPI_DEFAULT ON	0x05	0x02	0x02	