ADVACAM
Imaging the Unseen

The RawTpx3Pixel structure in memory

Online version:
https://wiki.advacam.cz/wiki/The_RawTpx3Pixel_structure_in_memory

Contents

Warning

- There is not how to use RawTpx3Pixel.
- There is:
  - Warning about potential complications with using RawTpx3Pixel.
  - Structure explanation (But the primary purpose of creating this page was to warn about problems with using the structure.)
- See the pxcMeasureTpx3DataDrivenMode if you want use online pixels processing.

The structure

Declaration:

```
#pragma pack(push, 1)
typedef struct _RawTpx3Pixel
{
    u32 index:      24;
    u64 toa:        64;
    byte overflow:   1;
    byte ftoa:       5;
    u16 tot:        10;
} RawTpx3Pixel;
#pragma pack(pop)
```

Potential problem source

But #pragma pack may not be 100% respected by the compiler, for example due to mandatory alignment.

Warning: Use the pxcGetMeasuredRawTpx3Pixels only in special, exceptional cases.

Explanation of the struct

- The index is index of the pixel. On the Minipix Tpx3 is 0 at the left-down.
- The toa is time of arrival in units 25 ns, mod by limit specific by device type.

  For example Minipix 2^{64} (14600y), Advapix-single 2^{30} (26s), Advapix-Quad 2^{28} (6.5s).
  Note: The ToA on-chip implementation in the pixels is limited to 14 bits (409.6 µs).

The toa item is extended by device. But there is inherent uncertainty around the borders. These values may be incorrectly assigned. Users not comfortable with our extension can apply AND with (uint64)16383 to extended ToA to get original ToA from the chip.

- The tot is time over threshold in units 25 ns.
- The ftoa stands for "fine ToA" and it is the finest step of the ToA measurement. To properly account for this step in the conversion of ToA to time, it is necessary to subtract the amount of counts of fToA in the following manner:

Time [ns] = 25*ToA - (25/16)*fToA

The original range of this fToA value in the chip is 4 bits, or 16 values. This is extended in the post-processing of the data into 5 bits, or 32 values to include a correction for the delay of the clock propagation in the chip. The final value exported in the ftoa item has a range of 5 bits, or 32 values, but the previous equation still stands.

- The overflow is sign of data transfer overflow. If the line has this 1:

index = 116 (0x74): start of lost data
index = 117 (0x75): end of lost data, toa is length of the missing time

(this can occurs with rates over megahits per seconds for Minipix)

The test

1. Add testbox, checkbox and test code to the Tpx3-1 example from the AdvacamApiExamples package.

2. Add global declarations:

```
RawTpx3Pixel dataDrivenPxRaw[32000000];
int dataDrivenPxRawOvrFirst = -1;
int dataDrivenPxRawLen = -1;
bool showRawPx = false;
```

3. Add this code to begin of the initMeasParams() function:

```
showRawPx = checkRawPixels->Checked;
dataDrivenPxRawLen = -1;
```

```
txtRawData->Visible = showRawPx;
```

4. Show the data sample function, add above the timer1_Tick function:

```
void showRawPxFn() {
     if (dataDrivenPxRawLen < 1) return;
     int start = (dataDrivenPxRawOvrFirst > 0) ? dataDrivenPxRawOvrFirst : 0;
     if (start > 1) start-=2;
     String^ s = String::Format("start adr: {0}, OvrFirst: {1}, siz: {2}\r\n", start,
dataDrivenPxRawOvrFirst, sizeof(RawTpx3Pixel)) + "index 24 / toa 64 / overflow 1 / ftoa 5 /
tot 10\r\n";

     for (int n = 0; n < 20; n++) {
             int idx = start + n;
             s += String::Format("{0}\t{1}\t{2}  {3}\t{4}\t", dataDrivenPxRaw[idx].index,
dataDrivenPxRaw[idx].toa, dataDrivenPxRaw[idx].overflow, dataDrivenPxRaw[idx].ftoa,
dataDrivenPxRaw[idx].tot);
            s += "B:";
            unsigned char* arr = (unsigned char*)(dataDrivenPxRaw + idx);
            for (int i=0; i < 16; i++) {
                    s += String::Format(" {0}", arr[i]);
            }
            s += "\r\n";
    }
    txtRawData->Text = s;
}
```

5. Add the line

```
if (showRawPx) showRawPxFn();
```

to the

```
if (dataDrivenPixelsCount != 0) {
```

section in the timer1_Tick function

6. Add this section to end of the clbDataDriven function:

```
if (showRawPx) {
     if (dataDrivenPixelsCount > 1000000) dataDrivenPixelsCount = 1000000;
     rc = pxcGetMeasuredRawTpx3Pixels(deviceIndex, dataDrivenPxRaw, dataDrivenPixelsCount);
     dataDrivenPxRawOvrFirst = -1;
```

```
     dataDrivenPxRawLen = 0;
     if (rc != 0) {
             dataDrivenPxRawLen = 0;
             clbError = rc;
             return;
    }
    dataDrivenPxRawLen = dataDrivenPixelsCount;
    for (int n = 0; n < dataDrivenPixelsCount; n++) {
            if (dataDrivenPxRaw[n].overflow != 0) {
                    dataDrivenPxRawOvrFirst = n;
                    break;
            }
    }
}
```

First test and result

Check the checkRawPixels and click the data-driven with callback button.

```
start adr: 0, OvrFirst: -1, sizeof: 15
index 24 / toa 64 / overflow 1 / ftoa 5 / tot 10
242        196318        0  20        14        B: 242 0 0 0 222 254 2 0 0 0 0 0 40 14 0
245
245   238506        0  9        3        B: 245 0 0 0 170 163 3 0 0 0 0 0 18 3 0 238
750  386872        0  19        20        B: 238 2 0 0 56 231 5 0 0 0 0 0 38 20 0 238
750  445251        0  17        17        B: 238 2 0 0 67 203 6 0 0 0 0 0 34 17 0 238
750  477832        0  14        8        B: 238 2 0 0 136 74 7 0 0 0 0 0 28 8 0 238
750  572804        0  12        11        B: 238 2 0 0 132 189 8 0 0 0 0 0 24 11 0 6
4614 631863        0  15        21        B: 6 18 0 0 55 164 9 0 0 0 0 0 30 21 0 6
4614        636812        0  25        27        B: 6 18 0 0 140 183 9 0 0 0 0 0 50 27 0 6
```

- Length of the declared structure is: 24+64+1+5+10 = 104 bits = 13 bytes
- Length returned by the sizeof function is: 15 bytes
- The bytes list confirms this, it can be seen that the next record starts from the 16th byte.
- It is unlikely that the lowest byte of the ToA is always zero, so we see that the ToA starts with the 5th byte, so the pixel index is 4 bytes length.
- Assume that the ToA is indeed 8 bytes. So the next entry starts with the 13th byte.
- 13th byte at first data line has 40. It contains overflow=0 and ftoa=20.

Bits 7-0 in this byte is: offffff00 (o overflow, f ftoa, fill zeroes)

- 14th and 15th byte at first data line is 14,0. This is word containing 10 bits of tot value and 6 bits filled by 0.

  First byte contains lower 8 bits of tot, second byte has remaining highest 2 bits and zeroes.

- 16th (last) byte of the line is the first byte of the next line.

Second test and result

Now add the lines

```
rc = pxcSetThreshold(deviceIndex, 0, 0.0);
errorToList("pxcSetThreshold 0 ******* test ****", rc);
```

to end of the initMeasParams() function
This will cause a dramatic increase in noise and subsequent occurrences of data overflow events.

```
start adr: 0, OvrFirst: 2, siz: 15, dLen: 543665
index 24 / toa 64 / overflow 1 / ftoa 5 / tot 10
10174        147457        0  16        4        B: 190 39 0 0 1 64 2 0 0 0 0 0 32 4 0 192
9920 147457        0  15        5        B: 192 38 0 0 1 64 2 0 0 0 0 0 30 5 0 116
116        0          1  0        0        B: 116 0 0 0 0 0 0 0 0 0 0 1 0 0 117
117         83          1  0        0        B: 117 0 0 0 83 0 0 0 0 0 0 1 0 0 47
57391 147457        0  24        8        B: 47 224 0 0 1 64 2 0 0 0 0 0 48 8 0 43
57643        147457        0  24        5        B: 43 225 0 0 1 64 2 0 0 0 0 0 48 5 0 45
57389        147457        0  24        5        B: 45 224 0 0 1 64 2 0 0 0 0 0 48 5 0 53
57397       147457        0  21        7        B: 53 224 0 0 1 64 2 0 0 0 0 0 42 7 0 49
```

In the listing we see pairs of lines that have overflow=1, pixel indexes 116 and 117. The first one announces the beginning of the area of missing data, the second the end and has the missing time in the toa item.

Warning: The bad threshold value will save to the config file after the pixet core correctly exit. This causes that even if you remove the line with the threshold setting, the device will continue to noise at next runs. Then you have to load factory config, copy the file from somewhere else, or edit this item manually.

The essence of the problem

Different compilers, for different platforms and with different settings, may have different "opinions" on the interpretation of the #pragma pack and bit-sized items in a structs. This may result in incompatibility.

- If the compiler generating the library interprets the structure in the same way as the compiler generating the user program, standard access to the structure's items in memory, such as array[index].item, works correctly.
- But if the interpretations differ, a problem arises.
- Therefore, it is necessary to carefully test whether the data access works correctly when using it for the first time. If not, the entries must be accessed differently, for example using bit-shifts and ANDs.

Related

- Binary core API: pxcMeasureTpx3DataDrivenMode
- Binary core API: pxcGetMeasuredRawTpx3Pixels
- Binary core API