Let us think of the concept of a single image from the camera. We will call this a *frame*. A frame is composed of some number of lines (the height) where each line is composed of some number of pixels (the width). A pixel is composed of some number of bits as a function of the color encoding. For example, a grayscale image is commonly 8 bits with 0 representing black and 255 representing white with corresponding graduations in between. For color images, the encodings can become more complex including 24 bits (8 bits for red/green/blue). For the cameras we are working with, color data is commonly encoded in 16 bits.

If we think of a frame in this context, we see that a frame is a large sequence of bytes. For example, a frame of 320 pixels by 240 lines would be 76,800 pixels which, at 2 bytes per pixel would be 153,600 bytes. This is also considered a small image. A large one might be 1920x1280 which would be 2,457,600 pixels or 4,915,200 bytes at 2 bytes per pixel. Again, realize that this is only one frame and we commonly want to achieve rates of 15 frames per second (or better). That is a lot of data.

For now, let's keep it simple and assume a 320x240 grayscale frame is our goal. That means we want to grab 76,800 bytes of data. In theory, we could use the CPU and read the bytes of data one at a time and stuff them in memory. That would indeed work and I can't see anything awfully wrong with it … however, there is a better way. The ESP32 support a technology called "Direct Memory Access" or DMA. The idea behind DMA is that data arriving from "outside" the ESP32 can be written directly to RAM without application oriented CPU instructions being performed. This means that we can just describe that we "want" the data and where to put it and the ESP32 will take care of the rest. In our camera story, this would mean taking the bytes that constitute the pixels and having the ESP32 write to RAM. Wonderful!!! However, life is rarely this simple. To use this technique, we need to understand a wealth of details and that is what we will focus on now.

Let us now bring in a partner component to DMA called I2S. While I2S is all about reading and writing serial audio data, in the ESP32 it is also the "peripheral" that is responsible for reading data from cameras. We will see that I2S and DMA are linked together in our story.

If we think of the I2S component as "sampling" data available on the input bus from the camera, what then is the unit of sample data. The answer, from an I2S perspective is 32 bits. That sounds good, that seems to say we can get a sample of 4 pixels at a time. Unfortunately, it isn't that easy.

There are three different possible encodings:

First, assume that the camera sends bytes: b1, b2, b3, b4, …

The first encoding we will look at is called `SM_0A0B_0B0C`. It encodes the data as :

```
[00 b1 00 b2] [00 b2 00 b3] [00 b3 00 b4] …
```

With this encoding, each sample (32 bits) includes only 1 new input byte for us to work with.

The second encoding is called `SM_0A0B_0C0D`. It encodes the data as:

```
[00 b1 00 b2] [00 b3 00 b4] …
```

With this encoding, each sample (32 bits) includes 2 new input bytes for us to work with.

The third encoding is called `SM_0A00_0B00`. It encodes the data as:

```
[00 b1 00 b0] [00 b2 00 00] [00 b3 00 00] [00 b4 00 00] …
```

With this encoding, each sample (32 bits) includes only 1 new input byte for us to work with.

Comparing these three schemes, we see that the encoding of `SM_0A0B_0C0D` gives us the highest density of usable data.

If we understand that I2S is responsible for reading samples from the camera, the next question is what does it do with these samples? The answer is that it places them in a First-In-First-Out (FIFO) queue. Think of this loosely as a "pipe". You start putting things in one end of the pipe and they come out in the same order as you put them in at the other end of the pipe. If I2S is placing its samples in the pipe, what then is at the other end? Now we get to come back to DMA. DMA is the consumer of the samples read from the pipe. Since DMA is responsible for taking data and placing it in RAM under hardware control, the next question is "where" does DMA place the data that it read from the FIFO queue? Here things get tricky again.

Now we introduce the notion of a DMA descriptor. This is a small C data structure that contains:

- Pointer to an area of allocated RAM into which the DMA retrieved data will be written.
- The size of the RAM associated with this descriptor.
- Pointer to the *next* descriptor.

The data structure described here is called `lldesc_t` which is an abbreviation of "linked list descriptor". This structure is defined in `rom/lldesc.h`. The description of this structure is reproduced here:

```
/* this bitfield is start from the LSB!!! */
typedef struct lldesc_s {
  volatile uint32_t size  :12,
                    length:12,
                    offset: 5, /* h/w reserved 5bit, s/w use it as offset in buffer */
                    sosf  : 1, /* start of sub-frame */
                    eof   : 1, /* end of frame */
                    owner : 1; /* hw or sw */

    volatile uint8_t *buf;       /* point to buffer data */
    union{
        volatile uint32_t empty;
        STAILQ_ENTRY(lldesc_s) qe;  /* pointing to the next desc */
    };
} lldesc_t;


SLC2 DMA Desc struct, aka lldesc_t
-------------------------------------------------------------
| own | EoF | sub_sof | 5'b0   | length [11:0] | size [11:0] |
-------------------------------------------------------------
|            buf_ptr [31:0]                                 |
-------------------------------------------------------------
|            next_desc_ptr [31:0]                          |
-------------------------------------------------------------
```

Notice the `length` field is 12 bits long. This means it can hold a value of 0 to 4095. This defines the length of the buffer pointed to by this `lldesc_t` instance. Since DMA is writing data in 32 bit (4 byte)

samples, the highest value we can have for a buffer length is 4092 bytes which would hold 1023 distinct samples.

The fields in `lldesc_t` are defined as follows:

- `size` - The size (in bytes) of the buffer.
- `length` - The number of bytes of DMA written data in the buffer.
- `sosf` - Not used by I2S DMA
- `eof` - End of file marker. Used to indicate that this is the end of the linked list of `lldesc_t` entries. We should not attempt to progress to the next entry.
- `owner` - Indication of who can write into this entry. If set to 1, then it is hardware (DMA) owned and DMA can write data into it. When the buffer is full, DMA will change the flag to 0 which means that it is now owned by the application.
- `buf_ptr` - Pointer to allocated RAM that can be written by DMA.
- `next_dsc_ptr` - Pointer to the next `lldesc_t` record in the linked list.

When DMA fills a `lldesc_t` buffer, the I2S subsystem generates an interrupt (IN_DONE).

To instruct DMA to do its job, we set some registers within the I2S peripheral. Specifically:

Register	**Field**	**Value**
I2S_RXEOF_NUM_REG rx_eof_num	N/A	Length of data expected to be received. Measured in samples.
I2S_IN_LINK_REG in_link	I2S_INLINK_START addr	Address of first entry in linked list of lldesc_t.
I2S_IN_LINK_REG in_link	I2S_INLINK_START start	Set this bit to start in-link descriptor. Set to 1.
I2S_INT_CLR_REG int_clr	val	Reset interrupt bits
I2S_INT_ENA int_ena	val	All the enable bits.
I2S_INT_ENA int_ena	I2S_IN_DONE_INT_ENA in_done	The I2S_IN_DONE_INT interrupt.
I2S_CONF_REG conf	I2S_RX_START rx_start	Set to 1 to start.