SILICONSOFTWARE GmbH

High Performance Image Acquisition

A Functional Description of the DMA engine

Application Note

Summary
In this application note, the DMA engine of the Silicon Software frame grabber series microEnable-III and microEnable-IV will be explained. First, a functional description of the data transfer methods as well as the memory management will be presented. Following to this, the methods to achieve the maximum performance rates at a superior stability of the Silicon Software frame grabbers are explained. Finally the document comprises a short overview on the interface models available and the user interfaces.
This document is an extension to the existing Silicon Software runtime documentations.

Keywords
- DMA
- SDK
- Buffer

Version 2

September 2009

www.silicon-software.com

Table of Contents

1. Introduction

The DMA engine's task is to transfers images from the frame grabber device to the PC memory.

In general, the frames are defined by the size of the image sensor and thus have a fixed length. Frames of line scan cameras however are defined by a logic concatenation of image lines (i.e. by the trigger system) and will have variable size. The situation is even worse if the framegrabber is used for image analysis. In this case the size of an image (frame) may strongly depend on the image content.

For better understanding, in the following the term frame is used for any of the conditions described and may consist in a number of bytes that have to be determined after acquisition.

Frames are transferred via Direct Memory Access (DMA) from the frame grabber to the PC memory. To do this a protected memory area has to be allocated and locked before the actual transfer. This memory is located in user space and thus seen in a continuous address space. In real life and due to the management by the Operating System the memory is paged. Each frame which is completely transferred to the PC causes an interrupt.

2. Acquisition Modes

In general a minimum of two memory sections i.e. memory buffers are required. Each of these buffers has to be large enough to store a complete frame – for frames with variable size it must be large enough for the worst case.

While one buffer is active as target of the DMA transfer the other buffer is used for image post-processing in the user application on the PC. In real life usually more than two buffers are used. Depending on the application there might be thousands of buffers.

The memory management of the Silicon Software DMA model is designed to hold a variable number of memory buffers. Furthermore, it offers two completely different ways of arranging the memory.

The first method (ACQ_STANDARD) arranges the memory buffers using a round-robin model. This model is called the ring buffer and is the most common method. In this approach the first frame is transferred to buffer number one. The second frame is transferred to buffer number two etc. If the last buffer is filled, buffer number one is used again. In this concept there is a direct map between image and memory buffer (image buffer = image number modulo number of buffers).

The second method of arranging memory (ACQ_BLOCK) fills each buffer using the FIFO (first in first out) principle. At the beginning, all buffers in the FIFO are available and may be filled with frame data. If one of the buffers is filled, it is deleted from the list of available buffers and hence, cannot be overwritten. As soon as the image data in a buffer is post-processed and hence not needed anymore, it is added to the FIFO list and may be reused again. It is important to note that the frame buffers can be locked and freed in random order. Declaring a frame buffer as free and thus adding it back to the list of free buffers is entirely controlled by the user application.

The selection of the desired method is done with the routine **Fg_Acquire()** that starts the DMA transfer.

There are two ways to allocate the frame buffer memory. It can be either done by Silicon Software SDK routines or by the user himself. In the second case the allocation is done using functions such a malloc() or similar ones.

Whatever method is choosen, always memory in user space is used. This gives the highest flexibility and does not need a continuous memory address space. Because the user memory can be freely accessed by the application there is no need for double buffering.

Another advantage is that nearly the complete physical memory can be allocated for the frame buffers. There is no constraint in either the number of sub-buffers (in the ring or FIFO buffer) or the allocated memory itself. Except the memory which is used by the Operating System itself the physical memory is usable completely. In 64 bit Operating Systems ring buffers with 32 Gigabyte were allocated successfully and experiences today are only constraint by the physical memory provided by the PC mainboards.

3. Performance Optimizations

For data transfer from the frame grabber to the PC main memory bus mastering is always used. This method ensures a maximum data transfer rate from the device to the host. The drivers and the DMA engine are designed for maximum possible data transfer rates and very high frame rates (> 100.000 frames/sec[1]).

Internally, the DMA is realized using so called scatter-gather lists which are used for complete images. Hence, no driver interaction is needed once a frame transfer is started. This maximizes the performance and stability.

If the transfer of one frame is completed, the transfer of the next one is automatically started by the hardware. This is done without intervention of the driver. This increases the performance, minimizes the latency of the software and facilitates the processing of very high frame rates.

The data transfer is always performed without double-buffering i.e. the original image data is always processed without a need of copying them into the driver internally. This increases the overall performance of the frame grabber and reduces the memory load of the system.

An image is transferred from camera to the grabber and from the grabber to the PC via DMA transfer. The point in time in which the DMA transfer starts is efficiently set to reduce the latency to a minimum. In general, the DMA engine starts the transfer of the image data when the first line is completely captured from the camera and after it is processed in the grabber. The transfer does now wait until the image has been completely transferred to the frame grabber.

The data flow between the camera, the grabber and the PC is comparable to a pipeline. The data is directly transferred upon availability. Therefore, the latency between the image capturing and transfer completion is less than 10µs.

The three main points for optimized performance are:

1. The DMA transfer is completely hardware-controlled
2. There is no double buffering.
3. The latency between image capture and memory transfer is minimized.

[1] In order to get highest frame rates on small frames at least 16 buffers should be allocated in the ring buffer.

4. Emergency Situations

The DMA engine is designed for both highest performance and maximum stability. However in demanding applications there is the danger of critical situations which may even lead to image loss. Unfortunately those situations have real impact and are principle problems, i.e. they are *not* a bug in the implementation but somehow unavoidable. Although there is no chance of recovery the system provides a mechanism to detect those situations securely.

The two critical situations are:

1. The camera bandwidth is higher than the PC interface bandwidth.
2. The application software is slower than the image acquisition.

The first problem is a rather simple bandwidth problem. If the camera is faster than the framegrabber, there will be data loss in the framegrabber itself. However this does not refer to situation where a very fast camera outputs an image in maximum data rate and then waits a long time for the next trigger (**temporary** data rate). It refers to the situation where the **mean** data rate of the camera is higher than the PC interface bandwidth.

The Silicon Software framegrabber are able to handle temporary high data rates using their internal DRAM. This DRAM works as FIFO and can buffer usually about 100 MByte and more. But when this situation lasts for a long time, the DRAM will fill up more and more until there is no more memory left. Then problem 1 occurs and the framegrabber will shorten or drop frames. There are always complete lines or frames dropped.

This problem can only by fixed by reducing the mean data rate accordingly. If this cannot guaranteed this situation is detected the following way. In the framegrabber each frame is marked with a (hardware) image number. This number is called tag and can be read by SDK routines. The image gets a second image number (actually the main image number) by software. In a system without frame loss both numbers will always match. If image loss occurs the number of lost images can be measured by the difference between software and hardware image number.

For each frame the frame size (transfer length) can be read by software. If image lines are dropped the expected image size will not match with the transfer length.

The second problem is caused by insufficient speed of the application software. It occurs if the time for image processing exceeds the acquisition time. If this is a temporary situation there is no problem. The acquisition will use the next frame buffer and the ring buffer will become more and more crowded. But if this situation lasts for a longer time there will be no free frame buffer left.

Now there are two possible reactions. The first possibility is to use the next buffer which is not processed yet and overwrite the old image by the new one. The second possibility is to skip the new frame. It is a decision between loosing either the new or the old frame. The better (or less worse) alternative depends on the application.

In this situation the two memory models of chapter 2 behave differently. The ring buffer model will give priority to the new frame and will use the next buffer. Here the old image is lost.

In the FIFO buffer model the new buffer will not find an empty slot in the buffer list. Thus the new frame is dropped.

In both model the image loss situation can be detected (see next chapter for details).

5. Transfer Status and Interface Model

From software side, a DMA transfer always requires an explicit start (**Fg_Acquire()** or **Fg_AcquireEx()**). Both functions can be parametrized to use either ACQ_STANDARD or ACQ_BLOCK acquisition model and how many frames should be grabbed. The frame count limit can also be set to GRAB_INFINITE to advise the grabber to send frames until explicitly stopped. Stopping the grabbing can be done at any time using **Fg_stopAcquire()** or **Fg_stopAcquireEx()**.

The state of the acquisition i.e. the number of currently transferred images can be read using three different mechanisms.

The first mechanism is a non-blocking request for the last completely transferred frame (**Fg_getLastPicNumber()**, **Fg_getLastPicNumberEx()**, **Fg_getImage()** or **Fg_getImageEx()**).

The second mechanism is a blocking wait call for a respective image (**Fg_getLastPicNumberBlocking()**, **Fg_getLastPicNumberBlockingEx(), Fg_getImage()**[2]**,** or **Fg_getImageEx()**). This mechanism blocks the current thread i.e. the thread is in sleep state and therefore does not consume processor time. If the requested image has been fully transferred, the thread is reactivated and continues. If a time limit passed to the function call is exceeded without reaching the desired image number, the function returns a time-out.

The third mechanism uses APCs (asynchronous procedure calls) for notification of a complete image transfer. Using this method, the user can specify a function which is called upon the complete transfer of a frame by calling **Fg_registerApcHandler()** before calling **Fg_Acquire()** or **Fg_AcquireEx()**.

Detecting frame loss

Frame loss using ACQ_STANDARD

When ring buffer acquisition mode is used frame loss will happen when the grabber “overlaps” the user application. Since this is not detectable by the grabber as he has no way of knowing which image the user application is currently processing the frame loss detection has to be done by the application itself. A frame loss will happen when the grabber has grabbed more images as it could store in additional buffers, i.e. the frame number is greater or equal than the applications frame number + the number of available subbuffers. Checking for a frame loss uses exactly this equation: check the current frame number of the grabber (e.g. by using **Fg_getLastPicNumber()**) and check if it is greater or equal the next frame number your application would process + the number of subbuffers created.

[2] Fg_getImage() and Fg_getImageEx() can be used both blocking and non-blocking depending on the parameters passed to those functions.

Frame loss using ACQ_BLOCK

Since the situation of running out of buffers is easily detectable by the grabber some statistic counters (including a lost frames counter) are available in blocking mode. Those can be queried using **Fg_getStatus()** and **Fg_getStatusEx()**.

6. Transfer Info Interface

For every acquired frame a special set of values are stored to track the frame transmission. These values are:

- the amount of data written to the buffer
- the exact time when the interrupt signaling the completion of the transfer was received
- the image tag of the transferred image

These values can be queried using **Fg_getParameter()** and **Fg_getParameterEx()**. **Fg_getParameter()** will be mapped internally to call **Fg_getParameterEx()** so calling **Fg_getParameterEx()** directly will give slightly more efficient code.

The values can be queried as follows[3]:

- The DMA transfer length: **Fg_getParameter**(.., FG_TRANSFER_LENGTH, ...). Please note that the variable used to store that value is of type *size_t*, i.e. passing a pointer to a 32 bit value on 64 bit systems will lead to memory corruption.
- The image tag: **Fg_getParameter**(.., FG_IMAGE_TAG, ...). This tag contains a hardware generated number which can be used for DMA transfer consistency check.
- The timestamp: **Fg_getParameter**(.., FG_TIMESTAMP, ...) or **Fg_getParameter**(..., FG_TIMESTAMP_LONG, ...).

Please see the general SDK documentation for exact description of these functions and their arguments.

[3] For simplicity only one query function is used in the examples.