LFG Video Capture Card

Programmer’s Manual

Active Silicon Limited

v1.1

Disclaimer

While every precaution has been taken in the preparation of this manual, Active Silicon Ltd assumes no responsibility for errors or omissions. Active Silicon Ltd reserves the right to change the specification of the product described within this manual and the manual itself at any time without notice and without obligation of Active Silicon Ltd to notify any person of such revisions or changes.

Copyright Notice

Trademarks

Printed in the United Kingdom.

Version 1.1 January 2002

Part Number: LFG-LIB-MAN

Active Silicon Limited, Brunel Science Park, Kingston Lane, Uxbridge, Middx, UB8 3PQ, United Kingdom.

Tel: +44 1895 234254
Fax: +44 1895 230131
Web: www.activesilicon.co.uk
Email: support.lfg@activesilicon.co.uk

Revision History

Version	Date	Comments
1.0	8th January 2001	First release.
1.1	30th January 2002	Minor updates.

Table of Contents

Introduction

This manual describes Active Silicon's LFG software library which provides a simple interface to the LFG capture card. This library is supported on several operating systems, including Windows 98, NT, 2000, ME, MS-DOS and Linux. The low level driver layer is provided by Active Silicon's CDA driver architecture, also included in the SDK.

The LFG capture card uses the Conexant Fusion 878A video/audio acquisition chip (derived from the Brooktree Bt878). For those interested, the datasheet for this chip can be found in the LFG release media in the documentation area. However the LFG library has been designed to hide the physical details of the hardware and provide the user with a simple logical interface.

The library and driver have both been written specifically for professional applications in image acquisition, image processing, scientific imaging and machine vision, and are not based on the standard Fusion 878A drivers.

Digital video data is transferred into either host memory or directly to the display without any software overhead. The hardware handles all DMA scatter/gather and page table information automatically. The pixel format may also be converted in hardware to a number of standard RGB formats, including colour space conversion from YUV 4:2:2 to RGB colour space if required.

The functionality of the library consists of four main areas:

1. Card initialisation via an open call (and a close call to terminate access).
2. Card configuration for the desired video and acquisition mode. This includes video standard, image size, region of interest, field mode, temporal sampling and many other option settings.
3. Functions to start and stop acquisition. Notification of image acquisition may be interrupt driven (via a callback function) or polled.

Image display and image file format support is provided by the TMG imaging library supplied with the SDK. This imaging library is supplied on a licence basis for use with the LFG capture card only and therefore may only be used in a PC containing a LFG capture card. Also contained in the TMG imaging library are optimised JPEG compression routines capable of compressing full resolution video in real-time.

Multiple boards in a single PC are supported, limited only by the number of PCI slots.

Concepts

OVERVIEW

The LFG library consists of an application level programming API and a kernel mode driver. The library API is designed to be as simple as possible, yet able to provide a rich set of configuration and acquisition modes often required in complex machine vision applications.

The architecture of the library is based on Active Silicon’s “Logical-Physical Architecture” (LPA). The LPA architecture provides the user with the ability to set a series of logical settings, such as image size, video standard etc, and then call a single function to configure the hardware appropriately. This provides total abstraction from the details of the physical implementation (i.e. all the low level registers).

Internally the library takes these logical settings and maps them to their physical register level equivalents, but importantly it will only write any registers that have changed and then only write them via a single driver call. It does this by caching all writes and then sending them all to the low level driver. This has the benefit of being highly fast and efficient during on the fly reconfiguration of video modes. The LPA architecture also provides a method for by-passing the logical settings and writing directly to the physical hardware, which can sometimes for useful for advanced users.

For status information, the library does the reverse, that is, it takes physical status information from the capture card and maps these to simple logical settings.

OPEN AND CLOSE

The *LFG_Open* command is used to open the LFG library and device driver and establish communication with the capture card. *LFG_Close* is used to terminate access and free up internal library and driver resources.

CONFIGURATION AND STATUS OF THE CAPTURE CARD

Configuration of the capture card is done by setting various structure members to their required setting and then making the single call to *LFG_SetAndGet* which configures the capture card and returns status information about the card.

EVENT DRIVEN ACQUISITION

For most applications, event driven acquisition will be the preferred solution. In this mode a user installed callback function is called each time a new image is acquired or on a number of other events, such as trigger input. The “new image” event can be one of several events, based on video fields or frames, temporal subsampling or one particular image being acquired in a circular acquisition buffer of programmable length. Generally two image buffers would be used (these are automatically created) allowing acquisition into one whilst processing and/or displaying from the other.

Polled acquisition may also be used if required - in this mode the user’s application requests the status of acquisition, effectively waiting for a new image to be written into host memory.

IMAGE DISPLAY, FILE FORMAT SUPPORT AND JPEG COMPRESSION

Image display, image file load and save, and optimised JPEG compression are supported by the TMG library provided with the SDK. Please refer to the TMG Library Programmer’s Manual for further details and the example applications in the SDK.

SOURCE FILES PROVIDED WITH THE SDK

The following source files are provided with the SDK as an aid to development of custom applications:

lfg_mode.c	This file, which is compiled into the library, shows how the top level video modes map onto the low level logical settings. This code provides a useful reference for how to generate customised video modes.
lfg_tmg.c	This file, which is not compiled into the library, shows how the LFG library may be interfaced to a typical image processing or display library. This interface file would normally be compiled in with application source code to provide a simple API from the application to an image processing and display library. The LFG capture card includes a single runtime licence for the TMG library and this interface file allows it to be used in a simple but effective manner.
lfg_tmg.cpp	A C++ version of the above designed for use with MFC applications.

The LFG include files are as follows:

lfg_api.h	This is the main application include file and contains the LFG structure (which contains all the logical settings available for the card). It is the only LFG include file that is required by the user to use the LFG functions. It includes other necessary include files listed below.
lfg_os.h	Definitions for various operating system dependencies including types.
lfg_err.h	Status, error returns codes and macros.
lfg_hw.h	Hardware register definitions and related.
lfg_pro.h	Function prototypes and external definitions.

Also provided are the include files for the CDA driver layer and TMG imaging library.

Different operating systems are supported using one of the compiler pre-processor directives:

- *_LFG_WIN32* for Windows 98, NT, 2000 and ME;
- *_LFG_LINUX* for Linux; and
- *_CDA_DOS32* for 32 bit MS-DOS support

See "lfg_os.h" for the latest supported operating systems.

Function Overview

This section gives a brief overview of each function available to the user. All functions are described in detail further on in this manual.

INITIALIZATION FUNCTIONS

LFG_Open	Establishes communication with the LFG capture card and configures the board into a default state. A specific PCI slot number may be selected or alternatively the function will scan for first available device. A error handler may also be passed into the function which will be called on any error condition and supplied with the error code and library function name in which the error occurred.
LFG_Close	This function closes a previously opened LFG capture card and frees all internal associated resources.

SETUP FUNCTIONS

LFG_EventHandlerInstall	Installs a callback function that is called when certain events occur such as when a new image has been acquired.
LFG_SetAndGet	"Sets" the capture card with configuration information and "gets" status information from the hardware.

ACQUISITION FUNCTIONS

LFG_AcquisitionStart	Starts acquisition. (DMA based with zero software overhead.)
LFG_AcquisitionStop	Stops acquisition.

IMAGING LIBRARY INTERFACE FUNCTIONS - SOURCE CODE PROVIDED

LFG_TMG_ImageCreate	Creates a TMG image that references the LFG's host video buffer.
LFG_TMG_ImageDestroy	Destroys a previously allocated TMG image.
LFG_TMG_ImageSet	Configures the TMG image's parameters to match the image format of the LFG capture card.

ERROR HANDLING AND RELATED FUNCTIONS

_LFG_Assert	A useful assert function that can be used during software development. Defined in "lfg_os.h".
_LFG_DebugString	Prints out a text message and a text string parameter. Useful during development.
_LFG_Debug	Prints out a text message and a numerical parameter. Useful during development.
_LFG_DebugPopup	Displays a popup window (under GUI based operating systems) and stops program execution.
LFG_ErrorHandlerInstall	Installs a user defined error handler. *LFG_ErrHandlerDefault* is the name of the default error handler.

Note the debug and assert macros are only implemented if *_LFG_DEBUG* is defined. See the separate manual "LFG Error Handling" for further details of these functions.

Example Application

This section provides the source code for a complete application for the capture and subsequent saving to file of video data. This example application, called “simple.c” is installed as part of the SDK along with appropriate makefiles. Full error checking on all the return values of functions has been excluded for clarity, although the default error handler is used so any errors will be apparent. Again for clarity, this example is a simple console program and therefore does not provide image display. (The SDK contains other more comprehensive examples with full source code showing image display and processing.)

```
/* “simple.c” - Simple console example program.
 * This program configures the card, acquires 100 video frames, and
 * then writes last acquired image as a TIFF file.
 */
#include <lfg_api.h>
#include <tmg_api.h>   /* Used to save the image to a TIFF file */

static void EventHandler(ui32 hCard, ui32 dwEvent, ui32 dwIntStatus, void*
pv);

struct tMyApp  /* Put everything we need into a structure */
{
   struct tLFG  gsMyLFG;        /* LFG device structure            */
   struct tLFG *psLFG;          /* Convenient pointer to our LFG  */
   ui32 hCard;                  /* Handle to LFG PCI card         */
   ui32 hSrcImage1;             /* Source image 1 for DMA'ed data */
   ui32 hSrcImage2;             /* Source image 2 for DMA'ed data */
   volatile i32 nImageCount;    /* Count images that are DMA’ed   */
};

int main()
{
   struct tMyApp sApp;
   ui32 hRGB_Image;

   printf("\nLFG: Simple Test Program");

   memset(&sApp, 0, sizeof(struct tMyApp));
   sApp.psLFG = &(sApp.gsMyLFG);
   sApp.nImageCount = 0;
   LFG_Open( &(sApp.hCard), sApp.psLFG, LFG_ErrHandlerDefault);

   sApp.psLFG->sLog.dwVideoMode = LFG_50_PAL_384x288_F1;
   LFG_SetAndGet(sApp.hCard, sApp.psLFG, LFG_WRITE);

   LFG_TMG_ImageCreate( &(sApp.hSrcImage1) );
   LFG_TMG_ImageCreate( &(sApp.hSrcImage2) );
   LFG_TMG_ImageSet( sApp.hCard, sApp.psLFG, sApp.hSrcImage1, 1);
   LFG_TMG_ImageSet( sApp.hCard, sApp.psLFG, sApp.hSrcImage2, 2);

   sApp.psLFG->sLog.dwEvents = LFG_EVENT_NEW_IMAGE;
   sApp.psLFG->sLog.fPolledDrivenCallback = TRUE; /* For clarity here */
   LFG_SetAndGet(sApp.hCard, sApp.psLFG, LFG_WRITE);
   LFG_EventHandlerInstall(sApp.hCard, &EventHandler, &sApp);

   LFG_AcquisitionStart(sApp.hCard);

   while ( sApp.nImageCount < 100 )   /* Loop until finished */
   {
      /* The next line drives the event driven callback in polled mode */
      LFG_SetAndGet(sApp.hCard, sApp.psLFG, LFG_READ);
```

```
   }

   LFG_AcquisitionStop(sApp.hCard);

   /* Now convert from YUV 4:2:2 to RGB24 and save as a TIFF file */
   hRGB_Image = TMG_image_create();
   TMG_image_convert(sApp.hSrcImage1, hRGB_Image, TMG_RGB24, 0, TMG_RUN);
   TMG_image_set_outfilename(hRGB_Image, "out.tif");
   TMG_image_write(hRGB_Image, TMG_NULL, TMG_TIFF, TMG_RUN);
   TMG_image_destroy(hRGB_Image);

   LFG_TMG_ImageDestroy(sApp.hSrcImage1);
   LFG_TMG_ImageDestroy(sApp.hSrcImage2);
   LFG_Close(sApp.hCard);

}  /* End main() */


/*
 * Event handler - “dwEvent” is the logical bitwise event status.
 * This function is called each time a new image is acquired.
 * We use the default buffer size of 2 images, so we can process from one
 * image whilst we acquire into the other.
 */
static void EventHandler( ui32 hCard, ui32 dwEvent, ui32 dwIntStatus, void*
pv)
{
   struct tMyApp *psApp = (struct tMyApp*)pv;

   psApp->nImageCount++;

   if ( dwEvent & LFG_EVENT_NEW_IMAGE )
   {
     /* Make sure we know which image to process from our continuous
      * (circular) live acquisition sequence of 2 images.
      */
      if ( dwEvent & LFG_EVENT_END_SEQUENCE )
      {
         /* Process image 2 here */
      }
      else
      {
         /* Process image 1 here */
      }
   }
}
```

LFG_AcquisitionStart

USAGE

ui32 LFG_AcquisitionStart(ui32 hCard)

ARGUMENTS

hCard Handle to a LFG capture card.

DESCRIPTION

This function starts video acquisition which in turn will cause the hardware DMA engine to start transferring video data into memory.

The function *LFG_Open* must be called before this function in order to initialise and configure the capture card ready for acquisition. Typically *LFG_SetAndGet* would also be used to configure the card from its default to the desired acquisition mode. (The default acquisition mode is 50Hz colour PAL video at resolution of 384 x 288 at 25 frames per second.)

RETURNS

LFG_OK on success or an error code as defined in the programmer's manual "LFG Error Handling".

EXAMPLES

The following code will start acquisition. See also the complete example in the "Example Application" section.

```
LFG_AcquisitionStart(hCard);
```

BUGS / NOTES

None.

SEE ALSO

LFG_EventHandlerInstall, *LFG_AcquisitionStop.*

LFG_AcquisitionStop

USAGE

ui32 LFG_AcquisitionStop(ui32 hCard)

ARGUMENTS

hCard Handle to a LFG capture card.

DESCRIPTION

This function stops video acquisition and turns of the hardware DMA engine.

RETURNS

LFG_OK on success or an error code as defined in the programmer's manual "LFG Error Handling".

EXAMPLES

The following code will stop acquisition. See also the complete example in the "Example Application" section.

```
LFG_AcquisitionStop(hCard);
```

BUGS / NOTES

None.

SEE ALSO

LFG_AcquisitionStart.

LFG_Close

USAGE

ui32 LFG_Close(ui32 hCard)

ARGUMENTS

hCard	Handle to a LFG capture card.

DESCRIPTION

This function closes a previously opened LFG capture card and frees internal resources associated with the card.

RETURNS

LFG_OK on success or an error code as defined in the programmer's manual "LFG Error Handling".

EXAMPLES

The following code terminates access to the capture card. See also the complete example in the "Example Application" section.

```
if ( (dwErrCode = LFG_Close(hCard)) != LFG_OK )
{
   printf("Failed to close LFG card (Error Code = %08x)\n", dwErrCode);
}
```

BUGS / NOTES

None.

SEE ALSO

LFG_Open.

LFG_EventHandlerInstall

USAGE

*ui32 LFG_EventHandlerInstall(ui32 hCard, void (*pFnHandler)(ui32, ui32, ui32, void*), void* pv)*

ARGUMENTS

hCard	Handle to a LFG capture card.
pFnHandler	User callback function.
pv	Pointer to an application specific "context" structure which will be passed to the callback function.

DESCRIPTION

This function installs a callback function that it is called by the LFG library on any event - either hardware or software that is setup in advance by a call to *LFG_SetAndGet*, using the LFG logical structure member *dwEvents.*

The four parameters passed to the callback function are as follows:

hCard Handle to the LFG capture card that caused the event.

dwEvent The event that caused the handler to be called, along with other useful status information.

The events are (bitwise):

LFG_EVENT_NEW_IMAGE	A new image has been acquired. With no temporal subsampling, this event is essentially a field or frame interrupt depending on the acquisition mode.
LFG_EVENT_TRIGGER	The trigger has been asserted. This event is signalled immediately the trigger is detected. A more practically useful version is listed next.
LFG_EVENT_TRIGGER_THIS_IMAGE	This event is synchronised with the *LFG_EVENT_NEW_IMAGE* event to indicate that the trigger has happened within the last image acquisition period (i.e. the last field or frame depending on the acquisition mode).
LFG_EVENT_VSYNC	A vertical synchronisation event has occurred (the start of a new video field).

The status information flags are:

LFG_EVENT_END_SEQUENCE	The last image acquired was the last one in the circular image buffer of N images, where the default for N is 2, but may be set to anything.

dwIntStatus The actual contents of the Fusion 878A interrupt status register. This may be of use to advanced users.

pv A *void** pointer to user installed context data. Typically this would be used for an application structure that contains the relevant information about the application and capture card.

Under pre-emptive multi-tasking operating systems, such as the supported Windows operating systems, the foreground process would normally sleep (using no CPU time) or alternatively be performing other application specific tasks.

However it is possible to simulate the callback environment without the use of hardware interrupts (which are required for the usual callback mode of operation). This can sometimes be useful when debugging or troubleshooting when sharing the interrupt line with other devices. (The LFG driver is designed to operate with either shared as well as exclusively allocated interrupt lines.) To do this, the logical flag in the LFG structure, *fPolledDrivenCallback* is set to *TRUE* and then the function *LFG_SetAndGet* called in a polling loop. See the section "Example Application" for an example of how to do this.

RETURNS

LFG_OK on success or an error code as defined in the programmer's manual "LFG Error Handling".

EXAMPLES

The following code is an example of how to install a callback function. See also the complete example in the "Example Application" section.

```
/* Install an event handler to be called each time a new image is acquired
 * and ready for processing.
 */
 sApp.psLFG->sLog.dwEvents = LFG_EVENT_NEW_IMAGE;
 LFG_SetAndGet(sApp.hCard, sApp.psLFG, LFG_WRITE);
 LFG_EventHandlerInstall(sApp.hCard, &EventHandler, &sApp);



/* Event handler
 * -------------
 */
static void EventHandler(ui32 hCard, ui32 dwEvent, ui32 dwIntStatus, void* pv)
{
  struct tLFG *psLFG = (struct tLFG*)pv;
  (void)hDevice;
  (void)dwData;

   if ( dwEvent & LFG_EVENT_NEW_IMAGE )
   {
     /* Make sure we know which image to process from our continuous
      * (circular) live acquisition sequence of 2 images.
      */
      if ( dwEvent & LFG_EVENT_END_SEQUENCE )
      {
         /* Process image 2 here */
      }
      else
      {
         /* Process image 1 here */
      }
   }
}
```

BUGS / NOTES

None.

SEE ALSO

LFG_SetAndGet.

LFG_Open

USAGE

*ui32 LFG_Open(ui32 *phCard, struct tLFG *psLFG, void (*pFnErrHandler)(ui32, const char*, ui32, const char*))*

ARGUMENTS

phCard	The address of (pointer to) a user allocated 32 bit unsigned integer, filled in by the LFG library, that becomes the handle used to reference the LFG capture card that has been opened.
psLFG	Pointer to a user allocated LFG structure, used to transfer information to and from the library.
pFnErrHandler	Function pointer to an error handler, called on any errors encountered whilst executing library code. The default error handler may be used by passing in the parameter *LFG_ErrHandlerDefault* (the name of the default error handler). A custom error handler may also be used instead of the default one. See the LFG Library Error Handling manual for further details.

DESCRIPTION

This function is used to open the capture card, that is to establish communication and provide a handle through which the capture card may be accessed.

The pointer *psLFG* must point to a user allocated structure (see example below). In order to select a particular LFG capture card (if multiple cards are used in one PC), the LFG structure member *dwDeviceAddr* is set to 0 to automatically select the first available card, or else 1, 2, 3…etc is used to select a specific PCI device number. (Note that often the physical layout of PCI slots does not follow an ascending pattern - for example a four slot PCI motherboard may have the physical slots laid out as 2134.)

RETURNS

LFG_OK on success or an error code as defined in the programmer's manual "LFG Error Handling".

EXAMPLES

The following code fragment shows the use of the open function:

```
int main()
{
   struct tLFG sMyLFG;     /* LFG device structure     */
   ui32 hCard;             /* Handle to LFG PCI card   */

   if ( LFG_Open( &hCard, &sLFG, LFG_ErrHandlerDefault) == LFG_OK )
   {
      printf("LFG Capture Card opened OK");
   }
}
```

BUGS / NOTES

None.

SEE ALSO

LFG_Close.

LFG_SetAndGet

USAGE

*ui32 LFG_SetAndGet(ui32 hCard, struct tLFG *psLFG, ui32 dwBitsOptions)*

ARGUMENTS

hCard	Handle to a LFG capture card.
psLFG	Pointer to the user defined LFG structure.
dwBitsOptions	A bitwise variable that accepts *LFG_WR*ITE to set information and *LFG_READ* to get information. These options may be used together by ORing them as follows: *LFG_WRITE* | *LFG_READ*

DESCRIPTION

This function provides a method of configuring the LFG capture card and reading back status information through the use of a single structure and single function call.

Various logical structure members may be used to set the functionality and similarly various logical members are set by the library according to the status of the card. Hence this function "Sets" the capture card and "Gets" status information.

The parameter *dwBitsOptions* determines whether the library is to set information or get status or both. This parameter is set to the bitwise flag *LFG_WRITE* in order to set information and *LFG_READ* to get status information. These flags may be used independently or together by ORing them:
i.e. "*LFG_WRITE* | *LFG_READ*".

The include file "lfg_api.h" contains the definition for the LFG structure.

The structure is composed of two sub-structures - one is a logical representation of the capture card and the other a physical representation of all the required values. The *LFG_SetAndGet* function intelligently generates the physical representation from the logical one and then only writes to the card any register values that have changed for optimum driver efficiency. And to make things even more efficient, the appropriate physical registers are only re-generated if a logical setting that effects that particular register is changed. This is done by comparing the application's LFG structure to a private internal one to determine exactly what has changed. This has the benefit that if required, the user may set some or all of the physical registers themselves to override or perhaps access a special mode not supported through the usual logical settings.

The logical members are accessed through the structure "sLog" and the physical registers (if so desired) through the structure "sReg".

For example to set increase the contrast from the default value of 128, the following code would be used:

```
psLFG->sLog.bContrast = 200;
```

The registers are not listed here but are fully documented in the Fusion 878A datasheet supplied as part of the SDK. The register may be accessed through the LFG API by using the sReg structure - for example to set the "A Delay" register directly:

```
psLFG->sReg.bAdelay = 0x20;
```

When setting registers directly, the capture card should first be setup with the closest configuration using the logical settings with a call to *LFG_SetAndGet* and then by fine tuning the register values whilst leaving the logical settings alone.

The *LFG_SetAndGet* function is also used to "drive" the callback functionality when configured for "polled" (non-interrupt driven) mode. In order to do this the logical parameter *fPolledDrivenCallback* is set to *TRUE* and *LFG_SetAndGet* polled in a loop - perhaps in a separate thread. See the application example in the "Example Application" section.

The following table lists all logical settings and their associated values:

fPolledDrivenCallback	Set to *TRUE* for polled driven callbacks. Default is *FALSE*.
dwEvents	A bitwise OR field to determine the events that get signalled to the installed callback function:

`LFG_EVENT_NEW_IMAGE`	Called each time a new image is acquired.
`LFG_EVENT_TRIGGER`	Called each time a trigger event occurs.
`LFG_EVENT_TRIGGER_THIS_IMAGE`	Same as *LFG_EVENT_TRIGGER* but synchronised with *LFG_EVENT_NEW_IMAGE*.
`LFG_EVENT_VSYNC`	Called on each video vertical sync.

dwVideoSrc — Selects one of the following video sources:

`LFG_VID_SRC_CM0`	Select composite/mono input 0 (default).
`LFG_VID_SRC_CM1`	Select composite/mono input 1.
`LFG_VID_SRC_CM2`	Select composite/mono input 2.
`LFG_VID_SRC_CM3`	Select composite/mono input 3.
`LFG_VID_SRC_YC0,1,2,3`	Select the S-video input (Luma on 0,1,2,3).

dwVideoMode — Selects an overall video mode that automatically configures many of the other logical settings for simplicity. The file that takes this video mode setting and fills in the other logical settings from it, is provided as a source code example so that additional video modes may easily be added. The file is "lfg_mode.c" and is installed as part of the LFG SDK.

There are 64 predefined video modes covering 50 and 60Hz video, colour (PAL/NTSC/SECAM) or monochrome, various standard image sizes and whether to acquire fields 1, 2 or both.

For example, *LFG_50_PAL_768x576_F12* means 50Hz video, colour PAL, 768 x 576 resolution, acquire both fields.

Another example would be *LFG_60_MONO_320x240_F1* which means 60Hz video monochrome at a resolution of 320 x 240 acquiring field 1 only.

Listed below is the set for colour PAL to give an idea of the options available. (The modes are actually generated from the bitwise ORing of several sub-options. See "lfg_api.h" for the full list, and "lfg_mode.c" for how these modes affect the logical members in the LFG structure).

There is also a setting *LFG_VMODE_USER* that allows the user to setup the options that would otherwise be automatic from *dwVideoMode*. For example to scale to particular resolutions or to read out a region/area of interest.

`LFG_50_PAL_768x576_F12`	50Hz video, colour PAL standard, 768 x 576 resolution, both fields acquired and re-interlaced to generate a full frame.
`LFG_50_PAL_640x480_F12`	As above but 640 x 480 resolution as a scaling example.
`LFG_50_PAL_768x288_F1`	50Hz, colour PAL, field 1 only, 25 images/sec.
`LFG_50_PAL_768x288_F2`	50Hz, colour PAL, field 2 only, 25Hz images/sec.
`LFG_50_PAL_768x288_F12`	50Hz, colour PAL, fields 1 and 2, 50Hz images/sec.
`LFG_50_PAL_384x288_F1`	As above, but 384 x 288, field 1 only (default).
`LFG_50_PAL_384x288_F2`	As above but field 2 only.

LFG_50_PAL_384x288_F12	As above but acquire both fields.
LFG_50_PAL_192x144_F1	As above, but 192 x 144, field 1 only.
LFG_50_PAL_192x144_F2	As above but field 2 only.
LFG_50_PAL_192x144_F12	As above but acquire both fields.

This next block of settings are set automatically by the previous setting, *dwVideoMode*, unless *dwVideoMode* is set to *LFG_VMODE_USER* in which case this next set may be set directly.

f50Hz — Set *TRUE* for 50Hz video and *FALSE* for 60Hz. Default is 50Hz.

dwVideoDecodeStd — Set to one of the following:

LFG_VID_STD_NTSC	NTSC-M as used in the USA and others.
LFG_VID_STD_NTSC_J	NTSC as used in Japan.
LFG_VID_STD_PAL	PAL-I, D, B, G, H as used in UK, Ireland, South Africa, Western Europe, China and others (default).
LFG_VID_STD_PAL_M	PAL-M as used in Brazil.
LFG_VID_STD_PAL_NC	PAL as used in Argentina.
LFG_VID_STD_PAL_N	PAL-N as used in Paraguay and Uruguay.
LFG_VID_STD_SECAM_4406	France, SECAM as used in France, the Middle East and Eastern Europe.
LFG_VID_STD_SECAM_4250	SECAM but based on a sub-carrier of 4.25MHz rather than 4.406MHz.
LFG_VID_STD_MONO	Monochrome video.

nHorzStart — Horizontal start of the image readout in un-scaled pixels. For full region of interest acquisition, set to *LFG_50HZ_HORZ_START* for 50Hz video (default) or *LFG_60HZ_HORZ_START* for 60Hz video.

nVertStart — Vertical start of the image readout in un-scaled lines. For full region of interest set to *LFG_50HZ_VERT_START* for 50Hz video (default) or *LFG_60HZ_VERT_START* for 60 Hz video.

fsHorzScaling — Set to a floating point number between 1.0 and 0.0625 (1/16) for horizontal scaling (default 0.5).

fsVertScaling — Set to a floating point number between 1.0 and 0.0625 (1/16) for vertical scaling (default 0.5).

fVertFieldAlign — Set to *TRUE* to re-align sequential odd and even fields in the vertical plane. Used if 50/60 frames per second acquisition is required for sub-sampled video. (Default *TRUE).*

fReInterlace — Set to *TRUE* if the DMA engine should "re-interlace" sequential video fields whilst transferring digitised video data across the PCI bus to result in re-constructed full frame video in memory. (Default FALSE.)

nImageWidth — Set to the desired output image width. Must tie in with the scaling factor (*fsHorzScaling*) and the horizontal start of image readout (*nHorzStart*). (Default 384.)

nImageHeight — Set to the desired output image height. Must tie in with the scaling factor (*fsVertScaling*) and the vertical start of image readout (*nVertStart*). (Default 288.)

dwFieldsToAcquire — Set to one of *LFG_FIELDS_F1*, *LFG_FIELDS_F2* or *LFG_FIELDS_BOTH* to determine whether to acquire fields 1, 2 or both. (Default Field 1 only.)

The following settings are independent of the Video Mode setting (*dwVideoMode*):

dwPixelFormat	Set the desired pixel format to be one of the following:

`LFG_PIXEL_FORMAT_AUTO`	Automatically select between Y8 or YUV422 depending on whether colour/mono video (default).
`LFG_PIXEL_FORMAT_Y8`	8 bits per pixel monochrome data.
`LFG_PIXEL_FORMAT_YUV422`	YUV 4:2:2 interleaved YCbCr data with byte ordering YUYV.
`LFG_PIXEL_FORMAT_RGB8_D`	RGB 3:3:2, 8 bits per pixel, dithered.
`LFG_PIXEL_FORMAT_RGB15`	RGB 5:5:5, 2 bytes per pixel.
`LFG_PIXEL_FORMAT_RGB15_D`	RGB 5:5:5, 2 bytes per pixel, dithered.
`LFG_PIXEL_FORMAT_RGB16`	RGB 5:6:5, 2 bytes per pixel.
`LFG_PIXEL_FORMAT_RGB16_D`	RGB 5:6:5, 2 bytes per pixel, dithered.
`LFG_PIXEL_FORMAT_RGB32`	32 bit RGB data with byte ordering (on Intel based PC) BGRX.

fColorBars	When set to *TRUE*, switches the input to hardware generated colour bars. Default is *FALSE*.
fLumaNotchOn	Set to *TRUE* to enable the luma notch filter when acquiring colour. The notch filter is automatically disabled when acquiring from an S-Video source. When acquiring from a monochrome source, it should be set to *FALSE* (but not when a monochrome picture is desired from a colour video source). Default is *TRUE*.
fRemoveGamma	Set to *TRUE* to apply an inverse gamma function to the video. This can be useful to remove gamma correction if unable to do so at the video source. Default is *FALSE*.
fFullRange	Set to *TRUE* to provide full dynamic range (0..255) luminance data, rather than the usual 16..253 CCIR range. The Cb, Cr range is always 2..253 with 128 representing zero colour information. Default is *FALSE*.
dwCoring	Coring option, set to one of the following. Coring can improve subjective image quality by mapping luminance pixels below a set threshold to black. Note the coring level is above black level, thus with *fFullRange* set *FALSE* (16 is black level) a coring level of 8 is actually 24.

`LFG_CORING_0`	No coring (default).
`LFG_CORING_8`	Pixel luminance level of 8 and below mapped to black.
`LFG_CORING_16`	Pixel luminance level of 16 and below mapped to black.
`LFG_CORING_32`	Pixel luminance level of 32 and below mapped to black.

bBrightness	Vary the image brightness using a number between 0 and 255 (128 represents no change and is the default setting).
bContrast	Vary the image contrast using a number between 0 and 255 (128 represents no change and is the default setting).
bColor	Vary the image colour level using a number between 0 and 255 (128 represents no change and is the default setting).
bHueShift	Only application to NTSC, this setting varies the hue. Default 128.
nNumImages	The number of images in the DMA buffer. The default is 2, which allows for the basic requirement of acquisition into one image, whilst displaying (or processing) from the other.
dwTriggerMode	Set the mode for the trigger input. Note that when using level sensitive interrupts, the interrupt handler will be continually called whilst the interrupt signal is at the interrupting level (i.e. high or low depending on the setting).

	LFG_TRIGGER_RISING_EDGE	A trigger event is signalled on a TTL rising edge.
	LFG_TRIGGER_FALLING_EDGE	A trigger event is signalled on a TTL falling edge (default).
	LFG_TRIGGER_LEVEL_HIGH	A trigger event is signalled on a TTL high.
	LFG_TRIGGER_LEVEL_LOW	A trigger event is signalled on a TTL low.
dwEndian	This setting allows byte swapping and 16 bit word swapping of 32 bit data in hardware.	
	LFG_ENDIAN_NO_SWAP	No swapping of the data (default).
	LFG_ENDIAN_BYTE_SWAP	Swap bytes 0 and 1, and 2 and 3.
	LFG_ENDIAN_WORD_SWAP	Swap the lower and upper 16 bit words.
	LFG_ENDIAN_BW_SWAP	Perform both a byte and word swap.
fLed1_On	A boolean setting to turn on or off the LED viewable through the end panel (default on - *TRUE*).	
fI2cExtEnable	Enable the I2C bus to drive externally (default disabled - *FALSE*).	
dwIo0, dwIo1, dwIo2	Configure each of the three TTL I/O lines to one of the following:	
	LFG_IO_INPUT	As a TTL input (default for all I/Os).
	LFG_IO_OUT_HI	As an output and drive to a TTL high level.
	LFG_IO_OUT_LO	As an output and drive to a TTL low level.
nOneEveryNImages	Temporal sub-sampling: Set to a number, N, which will result in an image being acquired every N images (default 1). Useful for time-lapsed photography / recording.	

Status information read back from the card and represented by the following logical members in the LFG structure:

fVideoPresent	A boolean that indicates video is present on the selected input.
fHLock	A boolean that indicates whether the acquisition phase locked loop has locked to the selected video source. Note that this status flag may not work for certain types of VCR (video cassette recorder) or other video devices that have varying line lengths. However acquisition will still function correctly.
fIo0_InHi, *fIo1_InHi*, *fIo2_InHi*	Boolean flags representing the status of the TTL I/O lines.

RETURNS

LFG_OK on success or an error code as defined in the programmer's manual "LFG Error Handling".

EXAMPLES

The following code sets up the card to use 50Hz PAL video at a resolution of 384 x 288 acquiring both fields at real time rates - hence at 50 fields per second.

```
psLFG->sLog.dwVideoMode = LFG_50_PAL_384x288_F12;
LFG_SetAndGet(hCard, psLFG, LFG_WRITE);
```

The following code switches on the hardware colour bars generator (perhaps in order to test an application when a real video source is not available):

```
psLFG->sLog.fColorBars = TRUE;
LFG_SetAndGet(hCard, psLFG, LFG_WRITE);
```

The following code sets a custom video mode for the readout of a region of interest that is 300 pixels by 300 lines without any scaling:

```
psLFG->sLog.dwVideoMode = LFG_VMODE_USER;
psLFG->sLog.f50Hz = TRUE;
psLFG->sLog.dwVideoDecodeStd = LFG_VID_STD_PAL;
psLFG->sLog.nHorzStart = LFG_50HZ_HORZ_START+100;
psLFG->sLog.nVertStart = LFG_50HZ_VERT_START+50;
psLFG->sLog.fsHorzScaling = 1.0;
psLFG->sLog.fsVertScaling = 1.0;
psLFG->sLog.fVertFieldAlign = FALSE;
psLFG->sLog.fReInterlace = TRUE;
psLFG->sLog.nImageWidth  = 300;
psLFG->sLog.nImageHeight = 300;
psLFG->sLog.dwFieldsToAcquire = LFG_FIELDS_BOTH;
LFG_SetAndGet(hCard, psLFG, LFG_WRITE );
```

The following code sets a custom video mode for the readout of a region of interest that is 120 pixels by 120 lines with a 60% scaling reduction:

```
psLFG->sLog.dwVideoMode = LFG_VMODE_USER;
psLFG->sLog.f50Hz = TRUE;
psLFG->sLog.dwVideoDecodeStd = LFG_VID_STD_PAL;
psLFG->sLog.nHorzStart = LFG_50HZ_HORZ_START+100;
psLFG->sLog.nVertStart = LFG_50HZ_VERT_START+50;
psLFG->sLog.fsHorzScaling = 0.4;

/* Note: As soon as we want less than half vertical size, we
 * set fReInterlace FALSE and scale from one field (that already
 * has a factor of 2 reduction), hence to achieve 0.4 we set
 * 0.8 for fsVertScaling.
 */
psLFG->sLog.fsVertScaling = 0.8;
psLFG->sLog.fReInterlace = FALSE;

psLFG->sLog.nImageWidth  = 120;
psLFG->sLog.nImageHeight = 120;
psLFG->sLog.dwFieldsToAcquire = LFG_FIELDS_F1;
psLFG->sLog.fVertFieldAlign = TRUE;
LFG_SetAndGet(hCard, psLFG, LFG_WRITE );
```

See also the application source for the demonstration programs installed as part of the LFG SDK for comprehensive examples.

BUGS / NOTES

None.

SEE ALSO

-

LFG_TMG_ImageCreate

USAGE

*ui32 LFG_TMG_ImageCreate(ui32 *phImage)*

ARGUMENTS

phImage	Address of a 32 bit unsigned integer to be used as a handle to a TMG image.

DESCRIPTION

This function creates a TMG image and returns a handle to reference the image in the 32 bit unsigned integer referenced by *phImage*.

The description for the function *LFG_TMG_ImageSet* explains in detail exactly how the TMG images are used by the LFG library.

By using a TMG image, all the standard functions in the TMG library are available such as image file save, load, optimised JPEG compression and decompression, and image display. See the TMG Imaging Library Manual, provided with the LFG SDK for further details. See also the example applications installed as part of the LFG SDK. Note the TMG library may only be used in a PC with a LFG capture card fitted. Standalone use requires a separate licence - please contact Active Silicon Ltd for details in required.

This function is not compiled into the LFG library, but provided as source code ("lfg_tmg.c" or "lfg_tmg.cpp") so that the TMG library may be used by simply compiling the file lfg_tmg.c with the application. This allows the flexibility of using the TMG library if required but also shows a methodology for interfacing to typical image processing and display libraries, should it be required to interface to another library.

RETURNS

LFG_OK on success or an error code as defined in the programmer's manual "LFG Error Handling".

EXAMPLES

The following code creates two images and then later on, before program exit, destroys them.

```
ui32 hImage1;
ui32 hImage2;

LFG_TMG_ImageCreate( &hImage1) );
LFG_TMG_ImageCreate( &hImage2) );

.
.  /* Main program … */
.

LFG_TMG_ImageDestroy( hImage1 );
LFG_TMG_ImageDestroy( hImage2 );
```

BUGS / NOTES

None.

SEE ALSO

LFG_TMG_ImageDestroy, *LFG_TMG_ImageSet*.

LFG_TMG_ImageDestroy

USAGE

ui32 LFG_TMG_ImageDestroy(ui32 hImage)

ARGUMENTS

hImage	Handle to a TMG image.

DESCRIPTION

This function destroys a previously created TMG image.

This function is not compiled into the LFG library, but provided with source code ("lfg_tmg.c") so that the TMG library may be used by simply compiling the file "lfg_tmg.c" with the application (or "lfg_tmg.cpp" for MFC applications). This allows the flexibility of using the TMG library if required but also shows a methodology for interfacing to typical image processing and display libraries, should it be required to interface to another library.

RETURNS

LFG_OK on success or an error code as defined in the programmer's manual "LFG Error Handling".

EXAMPLES

See the example code for *LFG_TMG_ImageCreate*.

BUGS / NOTES

None.

SEE ALSO

LFG_TMG_ImageCreate, *LFG_TMG_ImageDestroy*.

LFG_TMG_ImageSet

USAGE

*ui32 LFG_TMG_ImageSet(ui32 hCard, struct tLFG *psLFG, ui32 hImage, i32 nImageNum)*

ARGUMENTS

hCard	Handle to a LFG capture card.
psLFG	Pointer to the user defined LFG structure.
hImage	Handle to a TMG image.
nImageNum	A number that represents the Nth image created. This number is used by the function to associate the TMG image, *hImage*, to the correct image in the DMA image buffer.

DESCRIPTION

This function configures the TMG image, referenced by *hImage*, to reference the *N*th image in the sequence of *N* DMA image buffers, where *N* is the parameter *nImageNum*. The number of image buffers created is determined by the logical LFG member *nNumImages*, set up prior to a call to *LFG_SetAndGet*. See below for an example of 4 image buffers setup along with four TMG images that reference each of these buffers.

As well as associating a TMG image with an image buffer, the function also sets up various other parameters required to define the image, such as image width, height and pixel format.

This function is not compiled into the LFG library, but provided with source code so that the TMG library may be used by simply compiling the file "lfg_tmg.c" with the application (or "lfg_tmg.cpp" with MFC applications). This allows the flexibility of using the TMG library if required but also shows a methodology for interfacing to typical image processing and display libraries, should it be required to interface to another library.

RETURNS

LFG_OK on success or an error code as defined in the programmer's manual "LFG Error Handling".

EXAMPLES

The following code sets up the capture card to acquire full frame 50Hz video at a rate of 25 frames per second (50 fields per second) into a circular image buffer of 4 frames (8 fields). The images are re-interlaced during DMA to the target (host) memory to provide full frame, 768 x 576 pixel resolution images. *hImage1* references the first image, *hImage2* the second and so on. The capture card will continually acquire in real time sequential frames to image 1, 2, 3 and 4 and then back to 1 again.

```
psLFG->sLog.dwVideoMode = LFG_50_PAL_768x576_F12;
psLFG->sLog.nNumImages  = 4;
LFG_SetAndGet(hCard, psLFG, LFG_WRITE);

LFG_TMG_ImageSet( hCard, psLFG, hImage1, 1);
LFG_TMG_ImageSet( hCard, psLFG, hImage2, 2);
LFG_TMG_ImageSet( hCard, psLFG, hImage2, 3);
LFG_TMG_ImageSet( hCard, psLFG, hImage2, 4);

LFG_AcquisitionStart(hCard);
```

BUGS / NOTES

None.

SEE ALSO

LFG_SetAndGet, *LFG_TMG_ImageCreate*, *LFG_TMG_ImageDestroy*.