TMG Library

Programmer's Manual

Active Silicon Limited

v4.0.1

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

"Apple", "Macintosh" and "MacOS" are trademarks of Apple Computer Inc. "AMCC" is a registered trademark of Applied Micro Circuits Corporation. "Dallas" is a registered trademark of Dallas Semiconductor Corporation. "Dell" is a registered trademark of Dell Computer Corporation. "Flash Graphics" and "X-32VM" are trademarks of Flashtek Limited. "IBM", "PC/AT", "PowerPC" and "VGA" are registered trademarks of International Business Machine Corporation. "MetroWerks" and "CodeWarrior" are registered trademarks of MetroWerks Inc. "Microsoft", "CodeView", "MS" and "MS-DOS", "Windows", "Windows NT", "Windows 95", "Windows 98", "Win32", "Visual C++" are trademarks or registered trademarks of Microsoft Corporation. "National Semiconductor" is a registered trademark of National Semiconductor Corporation. "Sun", "Ultra AX" and "Solaris" are registered trademarks of Sun Microsystems Inc. All "SPARC" trademarks are trademarks or registered trademarks of SPARC International Inc. "VxWorks" and "Tornado" are registered trademarks of Wind River Systems Inc. "Xilinx" is a registered trademark of Xilinx.
All other trademarks and registered trademarks are the property of their respective owners.

Part Information

Part Number: TMG-MAN-LIB

Version v4.0.1 September 1999

Printed in the United Kingdom.

Contact Details

Web www.active-silicon.com
Support support@active-silicon.com

Head Office:
Active Silicon Limited.
Brunel Science Park, Kingston Lane
Uxbridge, Middlesex, UB8 3PQ, UK

Tel +44 (0) 1895 234254
Fax +44 (0) 1895 230131

Table of Contents

Introduction

This manual describes the "TMG" image processing and display library. This library contains functions for reading, writing, displaying and manipulating images under a variety of operating systems.

Under Windows NT, Windows 95 and Windows 3.1, the library is available as a dynamic link library (DLL). Under MS-DOS, the library is available as a static 32 bit library for Symantec C++ (using DOSX/X-32VM) and Watcom C++ (using the 32 bit flat model). Under Solaris 2 it is available as a dynamic library (.so - shared object). Under MacOS it is available as a shared and static library. Under LynxOS and VxWorks, it is available as a static library. The API is identical across all supported operating systems, apart from some minor variations related to mainly display functions. The software development kit contains example code to illustrate how to use the library in real applications.

The following sections describe the concepts, structure and methodology behind the library, as well a section with detailed examples covering all the key areas. Finally each library function is described in detail. As this library is licensed predominantly with the "Snapper" image acquisition hardware, many examples refer to Snapper image acquisition functions, although alternative acquisition hardware could equally be used.

It is strongly recommended that all the introductory sections in this manual are read and the examples provided with the SDK are examined before using the library.

Concepts

OVERVIEW

All the TMG functions use a "handle" to represent the image, referred to as an "image handle". This handle is a 32 bit unsigned integer. Each TMG routine uses the handle(s) passed into it to reference a pointer (through an internal global array index) to an image structure, which contains all the image parameters and the image data itself (or a strip of the image). See the file "tmg.h" for details of the actual image structure (called *struct Timage*).

Each function is designed to operate on the whole image or a strip within the image. Strip processing allows several functions to be chained together using only a small amount of memory. This is necessary when processing large images or performing many processing operations in a chain such that the full images cannot be accommodated in memory. Another benefit with processing small amounts of memory (or strips of the image at a time) is the potential performance benefit from the use of the processor's cache. However with the large amounts of memory available in computers nowadays, it is often simpler and unnecessary to consider strip processing.

A SIMPLE EXAMPLE

The best way to gain an understanding of how to use the library is by example. A simple example in 'pseudo code' is shown below.

Generally speaking all image processing applications acquire an image, perhaps from a camera or read it from disk, perform some operation(s) on that image, then write it back to disk, display it or even discard it. This operation can be fairly memory intensive, if the images are very large. The TMG library copes with large images by processing images in strips as discussed above. The basic idea of strip processing is to read, process and write the image, N lines at a time.

For example, an image with dimensions is 256 x 256 could be processed 8 lines at a time. This would require 32 processing operations - i.e. 8 x 32 = 256. This is precisely how the TMG library operates. In 'pseudo code' the algorithm would be as follows:

Create the image structures *(TMG_image_create)*.

Read the image to find out its dimensions *(TMG_image_read)*.

Set the strip size to 8 lines per strip *(TMG_image_set_parameter)*.

Calculate the number of strips in the image *(TMG_image_calc_total_strips)*

For each strip:

> Read in a strip (*TMG_image_read*).
>
> Image Processing Operation Number 1 (e.g. *TMG_IP_crop*).
>
> Image Processing Operation Number 2.
>
> Image Processing Operation Number 3 etc...
>
> Write a strip back out (*TMG_image_write*).

Destroy the image structures (*TMG_image_destroy*).

Of course if the strip size was set to the height of the image, there would be no need for the strip loop - i.e. each function would only be called once and the code required much simpler.

For normal operation the final parameter on any strip processing function is set to *TMG_RUN*. If for any reason the strip processing operation was aborted before the whole image was processed, the processing functions should be called with *TMG_RESET* to reset internal statics that keep track of how much of the image has been processed. For example the operation may need to be aborted before it is finished. In practice *TMG_RESET* is rarely needed.

For the more "modern" 32 bit operating systems, such as Windows NT, Solaris 2 etc, it's easier to always process the image in one strip (i.e. the whole image at a time). There are however several exceptions - for example the function *TMG_JPEG_compress*, is fairly memory hungry and is best used with only 8 lines at a time.

Notice also that in the above pseudo code example it was necessary to read the image's height so that the number of strip iterations could be calculated. When processing in one strip, *TMG_AUTO_HEIGHT* can be used on reading images - this instructs the read function to automatically read the whole image.

MEMORY ALLOCATION

Each TMG function, that takes an input image and produces an output image, will free any memory associated with the output image and allocate new memory for it (as long as the memory is not locked). The newly allocated memory for the output image will be the correct amount for the size of the image that is being processed. This method is robust, but it is fairly wasteful in the amount of memory re-allocation it does - potentially slowing down the processing. The solution to this is to lock the memory (using *TMG_image_set_flags* with *TMG_LOCKED*). This means that once the memory is allocated - i.e. the first time the function is called, it will never be freed until it is unlocked (or the image destroyed using *TMG_image_destroy*). This has the benefit that memory is no longer re-allocated by every TMG process. However there are some potential pitfalls that require a little more care from the application; the main ones being:

(a) if a larger image is processed, the output image data area may not be large enough (i.e. it was allocated for a previous smaller image); and

(b) if the application allocates the memory and locks it, then it must unlock it and free it, because the TMG library may be using different memory allocation routines than the application.

For the case of the larger image, (a) above, the "root" image (the first one in the chain) would have its memory unlocked, which will result in all downstream images unlocking, freeing and re-allocating their image memory (the root image could then be re-locked). There are plenty of examples of this on the SDK release disks. The section on "Operating System Issues" explains the actual memory allocation routines used for each operating system.

IMAGE DATA VERSUS JPEG IMAGE DATA

Each image handle references an image structure through an internal global array. This image structure contains a pointer to image data. This image data is usually raster image data, whose amount is related other internal parameters such as image width and height. However it is also possible to regard this image data as pure data - the flag *TMG_DATA_STREAM* indicates that this is the case (in this instance, the "image" is regarded as having a height of one). Note also that the image data may actually contain a sequence of frames, as determined by the internal parameter *num_frames*. Usually however only one image is contained within one image handle, and it's often easier to have an array of image handles for sequence work (not always the case for JPEG data - see below).

Within the image structure is a pointer to another structure that may or may not exist depending whether *TMG_image_create* or *TMG_JPEG_image_create* was used to create the image. *TMG_JPEG_image_create* creates an additional JPEG structure containing all the JPEG parameters and a pointer to JPEG data as well. The JPEG data can represent multiple frames ("motion JPEG"). In this situation JPEG restart markers are inserted between frames of JPEG data to allow direct replay (and recording) from suitable JPEG hardware. There are a number of functions for the manipulation of frames within a JPEG sequence.

It is possible for the application to allocate and setup the JPEG data area in the same way as for image memory area (see *TMG_image_set_ptr*). This is the only route when recording a motion JPEG sequence in which the application knows how many frames and hence how much memory it should allocate. Of course when working with JPEG data, generally it is not possible to know in advance how much memory will be required. The TMG library allocates an excess (actually half the memory required for the raw image) and then optimises it later.

The image structures can be seen in the file "tmg.h" available on the SDK disks. The image structure is *struct Timage*, and the JPEG structure is *struct Tjpeg*.

ADDING CUSTOM FUNCTIONS

Each TMG function has the same straight forward structure, which makes it convenient for a user to add his own functions if required. The file "tmg_scl.c", available on the SDK disks, contains the function *TMG_IP_subsample* which has been written in such a way that it does not need to be compiled into the DLL (or static library) to run. There are also many helpful comments added. This function may be used as a template for written custom functions.

VIDEO FIELDS AND THE "*TMG_HALF_ASPECT*" FLAG

When used with video acquisition hardware and software (such as Snapper) it may be a requirement that single video fields are acquired, processed and displayed. The TMG library copes with this through the use of a flag and parameter associated with the image. The flag, *TMG_HALF_ASPECT*, indicates that the image is a half aspect one, i.e. a video field. The parameter, *TMG_FIELD_ID*, is used to indicate which field it is - i.e. first or second. There are only a few functions that need to use this information - one is *TMG_SPL_2fields_to_frame* which is used to reconstruct a full height image. The other ones that use this information are some of the display functions that re-interlace the fields whilst displaying to achieve real-time display rates. See *TMG_display_image* for more details.

For some further information on the flag and parameter, see the functions *TMG_image_set_flags* and *TMG_image_set_parameter*.

Library Structure

The TMG functions are split into seven main groups that are conveniently indexed by their name. The groups are:

- *TMG_CK_...* This group of functions performs operations related to chroma keying.
- *TMG_cmap_...* This group of functions relates to operations with colourmaps (or palettes). For example their optimum generation etc.
- *TMG_display_...* This group performs image display (including printing).
- *TMG_image_...* This group performs all the general purpose "housekeeping" type functions.
- *TMG_IP_...* This group perform image processing functions
- *TMG_JPEG_...* This group contains all the function relating to JPEG images.
- *TMG_LUT_...* This group is a set of functions for the generation and manipulation of look up tables (LUTs).
- *TMG_SPL_...* This group contains special functions that don't neatly fit into any of the other groups.

Some functions apply only to certain operating systems/environments. These functions have the operating environment in square brackets after the function name for easy reference. If there is none, then the function applies to all operating environments.

(Note: The term "operating environment" means the combined operating system and windowing system in use. For example the Solaris operating environment is Solaris 2 running the Common Desktop Environment, which itself is running on top of Motif and in turn, the X Windows system.)

Pixel Formats

INTERNAL IMAGE TYPES

Internally the image data can be stored in many types of pixel formats. There are quite a few different pixel formats, but they all have their uses. Some of them are pixel formats from acquisition hardware, some are pixel formats suitable for saving to standard file formats, and others are pixel formats that match that of display hardware, thus saving valuable time by allowing direct display. As well as different pixel formats, there are three basic types of image. Firstly, the 'standard' image that contains raw image data in one particular pixel format; secondly, JPEG images that contain JPEG compressed data; and thirdly, DIB (device independent bitmap) images that contain the image data in the DIB format suitable for display under certain specific operating systems (Windows NT/95/3.1).

The function *TMG_image_convert* allows conversion between all these different pixel formats.

The pixel formats are as follows:

TMG_BILEVEL The image is a black and white "line art" image, where each pixel is represented by one binary bit. A '1' represents white and a '0' represents black. The binary data is packed into bytes, such that the MSB is the left most pixel. There is limited support for this type of image in the TMG library.

TMG_Y8 The image is a grayscale image, with each pixel represented by one byte, thus allowing 256 gray levels.

TMG_Y16 The image is a grayscale image, with each pixel represented by up to 16 bits, thus allowing up to 65536 gray levels. An additional internal image parameter, *data_width,* gives the number of valid bits. The data is always LSB aligned.

TMG_PALETTED The image is a paletted (or colourmapped) image, where each pixel is represented by one byte. This byte is an index into the palette (or colourmap). Typically the palette will have 256 entries of 24 bit RGB colours.

TMG_RGB8 The image is a colour image, with each pixel represented by 8 bits, of the form RRRGGGBB (i.e. 3 bits for red and blue and 2 bits for green - RGB 3:3:2, with red at the most significant end). This format is similar to *TMG_PALETTED* and could easily be represented as a paletted image. For this reason there is limited support (and use) for this format.

TMG_RGB15 The image is a colour image, with each pixel represented by 15 bits, of the form RRRRRGGGGGBBBBB (i.e. 5 bits per colour, with red at the most significant end). A 16 bit word is used to store the data with the MSB unused.

TMG_RGB16 The image is a colour image, with each pixel represented by 16 bits, of the form RRRRRGGGGGGBBBBB (i.e. 5 bits for red and blue and 6 bits for green, with red at the most significant end).

TMG_RGB24 The image is a colour image, with each pixel represented by 24 bits, of the form RGB (i.e. 8 bits per colour). The arrangement in memory is such that red is the first byte in byte addressing.

TMG_RGBX32 The image is a colour image, with each pixel represented by 32 bits, of the form RGBX (i.e. 8 bits per colour). The arrangement in memory is such that red is the first byte in byte addressing. The X refers to an unused plane (although it could be used as an overlay plane). This format conveniently aligns the data to a 32 bit boundary.

TMG_BGRX32 This format is identical to *TMG_RGBX32* except that the RGB order is reversed. Blue is now the first byte in byte addressing. This is the native pixel format of most PC based 24 bit graphics cards (in 32 bit, 16.7 million colours mode).

TMG_XBGR32 This format is similar to *TMG_BGRX32* except that the RGB bytes are shift by one and the X byte comes first in byte addressing. This is the native pixel format of most X Windows based 24 bit graphics displays on Sun SPARCstations.

TMG_XRGB32	This format is similar to *TMG_RGBX32* except that the RGB bytes are shifted by one and the X byte comes last in byte addressing. This format is often used internally by Mac workstations. There is limited supported for this format.
TMG_YUV422	The image is a colour image in the form YUV 4:2:2. This is arranged such that is byte addressing the data appears as YUYV. This is a standard digital video format for colour encoded video signals.
TMG_HSI	The image is a colour image in the form hue, saturation and intensity. The byte ordering in memory is two bytes for hue, followed by one byte for saturation and one for intensity. There is limited supported for this format.
TMG_CMYK32	This is a 32 bit colour format, whereby the image is represented by the "complementary" colour cyan, magenta, yellow (and 'black'). There is limited support for this format.

Many of these pixel formats are also used to describe the colour organisation of the display. For example a Windows NT graphics mode using 65K colours nearly always uses the *TMG_RGB16* pixel format.

The data alignment is to 16 bit boundaries, in other words the number of bytes per line is always even. Some functions (especially when running on 32 bit operating systems) are slightly faster if the data is 32 bit aligned (i.e. if the number of bytes per line divides by four - this is true of all the 32 bit pixel formats listed above). Odd width images are not fully supported by all functions, so it is recommended that even width images are used (this is usually the case with all video related acquisition). Future releases of the TMG library will automatically align to 32 bits (in fact programmable alignment) and provide full support for odd width images.

ACCESSING THE IMAGE DATA

The data can be directly accessed using the *TMG_image_get_ptr* function. Individual pixels may then be randomly accessed by using the pointer to the image data, knowledge of the bits per pixel (given by the format) and the number bytes per line (returned *TMG_image_get_parameter* with *TMG_BYTES_PER_LINE*).

Error Returns

Almost all of the TMG library functions return a *Terr*. *Terr* is a 32 bit unsigned integer, with the bit positions defined as follows:

31 to 24 Library identifier (returned on error, otherwise 0 is returned). This is used to allow a top level calling function to determine the library in which the error occurred.

The identifier is 0x10 (#defined as *TMG_LIBRARY_ID*).

23 to 16 Error number, otherwise 0 if no error.

15 to 0 Function return value.

If the function call is successful, *ASL_OK* is returned (which is #defined as 0) or the requested parameter. If an error occurs, an error number is returned in bits 23 to 16 along with the library identifier in bits 31 to 24.

The following is a list of error codes used by the TMG library, and a description of each error.

BAD_XXX ERRORS

ASLERR_BAD_HANDLE	The handle has not been set up, or is corrupt.
ASLERR_BAD_IMAGE	The image structure has not been set up, or is corrupt.

'NOT POSSIBLE' ERRORS

ASLERR_NOT_SUPPORTED	The requested operation is not supported, and is unlikely to be supported in future.
ASLERR_NOT_IMPLEMENTED	The requested operation is not currently implemented, but may be implemented in future.
ASLERR_INCOMPATIBLE	The requested option is not compatible with existing Snapper settings.
ASLERR_NOT_RECOGNIZED	The requested option is not recognized.

FUNCTION PARAMETER ERRORS

ASLERR_BAD_PARAM	A parameter passed to a function has not been recognised.
ASLERR_OUT_OF_RANGE	A parameter passed to a function is invalid - typically too large or too small.
ASLERR_PARAM_CONFLICT	Two or more parameters passed to a function are mutually exclusive.

OPERATING SYSTEM ERRORS

ASLERR_OUT_OF_MEMORY	A system call to reserve memory has failed.
ASLERR_THREAD_ERROR	A system call to control a separate thread of execution has failed.
ASLERR_DRIVER_CALL_FAILED	A call to the Snapper device driver has failed. For operating systems with a console (e.g. Solaris) check the console for any error messages from the driver.
ASLERR_SYSTEM_CALL_FAILED	An operating system call (other than those listed above) failed. For MS-DOS and Windows 3.1 this is used for BIOS and graphics driver calls.

FILE AND RELATED ERRORS

ASLERR_OPEN_FAILED	Open failed.
ASLERR_CLOSE_FAILED	Close failed.
ASLERR_READ_FAILED	Read failed.
ASLERR_WRITE_FAILED	Write failed.
ASLERR_SEEK_FAILED	Seek operation failed.
ASLERR_CORRUPT	File or data stream is corrupt.

MISCELLANEOUS ERRORS

ASLERR_OUT_OF_HANDLES	No free handles were found.
ASLERR_INTERNAL_ERR	An internal error was detected in the libraries.
ASLERR_IN_PROGRESS	The requested operation is already in progress.
ASLERR_INVALID_STATE	The library has detected an invalid state in the software or hardware, but cannot determine a more specific cause of the problem.

Operating System Issues

The TMG library is designed to run on virtually any operating system (and internally uses the same source code). To allow this, certain types and functions have been defined in the header files that allow for differences between operating systems whilst still preserving common source code. Some of the differences are described below. Full details can be found in the header file "os_sys.h" available with the SDK.

DATA TYPES AND IMAGE DATA POINTERS

Sizes of integers vary between compilers and operating systems and are a potential source of portability errors. To overcome this the following types are used - that are constant across all compilers and operating systems:

ui8 8 bit unsigned integer (unsigned char)
ui16 16 bit unsigned integer
ui32 32 bit unsigned integer
i16 16 bit signed integer etc.

For pointers to image data the following types are used:

*IM_UI8** Pointer to an 8 bit unsigned integer
*IM_UI16** Pointer to a 16 bit unsigned integer
*IM_UI32** Pointer to a 32 bit unsigned integer.

These are actually the same as the basic data types above (i.e. *IM_UI8* = ui8**) under all operating systems apart from Windows 3.1. Under Window 3.1 these types include the *_huge* modifier that allows the pointer to auto-increment across a 64K memory boundary. Note that the *_huge* modifier only modifies the variable to its immediate right, so the following code will fail:

```
IM_UI8 *pData1, *pData2;  /* Only pData1 is modified to __huge */
```

The correct definition is as follows:

```
IM_UI8 *Pdata1;
IM_UI8 *Pdata2;
```

Under Windows 3.1 the large memory model should be used.

MEMORY ALLOCATION METHODS

The method used for memory allocation varies between the different operating systems. For example Solaris 2 uses *memalign*. The #defines *MALLOC* and *FREE* are used internally in the TMG library and are defined in the file "asl_gen.h" available with the SDK - please refer to this file for more details.

ENDIAN ISSUES

Different operating systems (usually dependent on the processor architecture) will have different byte ordering for 16 bit and 32 bit words. For example most Intel processors are little endian, that is byte 0 is stored in bit locations 0 to 7, where as on a big endian processor (such as the SPARC) byte 0 is stored in bit locations 24 to 31 in a 32 bit word.

Endian issues obviously do not effect the interpretation of 8 bit grayscale data, but will potentially have an effect on any 16 or 32 bit format (such as *TMG_Y16* or *TMG_RGB16* etc). To avoid potential pitfalls here, 16 bit data should always be accessed as a 16 bit word (apart from simple copies when 32 bit read/writes can be done). This is what the TMG library does internally. 32 bit formats, such as *TMG_RGBX32* are effectively endian independent, because here the format is defined to be the byte order in byte addressable memory, thus *TMG_RGBX32* means red in byte 0, green in byte 1 etc. This means that if read as a 32 bit word the bytes will appear in different locations within that word on different endian processors. The TMG library is written to take account of this, and any application program accessing the data directly needs to be aware of this.

CONVERSION LOOK UP TABLES

Some look up tables (LUTs) used within the library have different sizes dependent on the operating system. The basic rule is that the LUTs may be smaller under Windows 3.1.

In summary:

- The *TMG_LUT_* suite of functions draws no distinction in LUT size.
- The image conversion LUTs (see *TMG_image_conv_LUT_generate*) YUV 4:2:2 to RGB15/16 are 64K bytes under Windows 3.1 and 1M byte under all other operating systems. These definitions can be found in the file "tmg.h" under "YUV to RGB LUT Definitions".
- The UV to hue LUT used in the *TMG_CK_* functions is 64K bytes under Windows 3.1 and 128k bytes under all other operating systems. See *TMG_CK_generate_UV_to_hue_LUT*.

The smaller LUTs used by Windows 3.1 result in a slightly lower conversion quality, but this is not significant.

Image Display Functions and Examples

The same basic core of functions are used to display images under each supported operating system. There are some minor variations between each operating system due to the native API of the actual environment, but essentially the methodology is the same. Note that the use of TMG functions to display images does not preclude the use of other libraries and/or native calls in the operating environment. The following sub-sections describe in detail how to display images under each supported operating environment.

The term "operating environment" means the combined operating system and windowing (or display) system in use. For example the Solaris operating environment is Solaris 2.x running the Common Desktop Environment, which itself is running on top of Motif and in turn, the X Windows system.

The convention in this manual is that any commands that do not apply to all operating environments list the ones that they do apply to by listing them in square brackets after the function name. For example:

TMG_display_set_paint_hDC [Windows]

The Windows NT/95/3.1 examples are from real applications using the Microsoft Foundation Class (MFC) application framework.

In summary the different types of operating environments with respect to display are:

- "Windows" – This is for Windows NT and Windows 95 using DirectDraw. It also applies to Windows 3.1 and DCI (Display Control Interface – the predecessor of DirectDraw).
- "DOS" – This is for MS-DOS and other MS-DOS lookalike operating systems.
- "X Windows" – This covers the X Windows system that is used for Solaris, LynxOS, VxWorks
- "MAC" – This covers display to the MacOS GUI.

IMAGE DISPLAY UNDER WINDOWS (INCLUDES WINDOWS NT, WINDOWS 95 AND WINDOWS 3.1)

The TMG library uses the basic Windows calls as well as DirectDraw and some proprietary display methods. The TMG display functions are designed to make it as easy as possible to display images with a reasonable amount of flexibility, yet without the learning curve and complications of displaying using the standard Windows calls. Also the fast DirectDraw method is all handled automatically within the library. Of course experienced Windows programmers can still use the GDI (Graphical Device Interface) function calls directly.

The basic function groups are as follows:

- *TMG_display_create*, *TMG_display_init* and *TMG_display_set_paint_hDC [Windows]* are used to initialise the display. *TMG_display_create* would be used once at the start of the program to create a display handle. *TMG_display_init* would be used to associate a particular Windows 3.1 window with the display handle. *TMG_display_set_paint_hDC* would be used each time a different device context was provided - for example in an *OnDraw* function.
- *TMG_display_set_ROI* is used to set a region of interest to display to (within a window), and *TMG_display_image* actually displays the image.
- *TMG_display_get_parameter* can be used to read back certain information about the display such as colour depth etc.
- *TMG_display_print_DIB [Windows]* can be used to print a DIB image to the printer in the usual way under Windows.

The following example code shows how an image would be displayed. The code has been lifted from the example application "IMV" from the Snapper Windows NT SDK. Please refer to this for more details.

```
// This code resides in imv.cpp.
// Create the display handles.
IMV.m_hDisplay = TMG_display_create();
IMV.m_hPrinter = TMG_display_create();  // The printer is a display device.
```

```
// This code is from imvview.cpp - but simplified slightly.
// This is the standard OnDraw function.
void CImvView::OnDraw(CDC* pDC)
{
   static BOOL bFirstTime = TRUE;

   if (bFirstTime == TRUE)
   {
      bFirstTime = FALSE;
      if ASL_is_err(TMG_display_init(IMV.m_hDisplay, GetSafeHwnd()))
         ::MessageBox(0, "Unsupported Mode.", "FAILED", MB_OK);
   }

   if ((TMG_image_get_ptr(IMV.m_hDIBImage, TMG_IMAGE_DATA) != NULL) &&
       (TMG_image_get_ptr(IMV.m_hDDBImage, TMG_IMAGE_DATA) != NULL))
   {
      if (pDC->IsPrinting())
      {
         // Set up the dimensions, then print supplying a % scaling parameter.
         TMG_display_set_parameter(IMV.m_hPrinter, TMG_WIDTH,
                                   (ui16) pDC->GetDeviceCaps(HORZRES));
         TMG_display_set_parameter(IMV.m_hPrinter, TMG_HEIGHT,
                                   (ui16) pDC->GetDeviceCaps(VERTRES));
         // special device context for printer
         TMG_display_set_paint_hDC(IMV.m_hPrinter, pDC->GetSafeHdc());
         // Print at 85% full size print area:
         TMG_display_print_DIB(IMV.m_hPrinter, IMV.m_hDIBImage, 85, TMG_RUN);
         TMG_display_set_paint_hDC(IMV.m_hPrinter, 0);
      }
      else  /* Display */
      {
         switch (m_pDlg->m_nScalingOptions)
         {
            case DLG_NO_SCALING:
               TMG_display_set_flags(IMV.m_hDisplay, TMG_STRETCH, FALSE);
               TMG_display_set_flags(IMV.m_hDisplay, TMG_KEEP_ASPECT, FALSE);
               break;
            case DLG_SCALE_NO_ASPECT:
               TMG_display_set_flags(IMV.m_hDisplay, TMG_STRETCH, TRUE);
               TMG_display_set_flags(IMV.m_hDisplay, TMG_KEEP_ASPECT, FALSE);
               break;
            case DLG_SCALE_KEEP_ASPECT:
               TMG_display_set_flags(IMV.m_hDisplay, TMG_STRETCH, TRUE);
               TMG_display_set_flags(IMV.m_hDisplay, TMG_KEEP_ASPECT, TRUE);
               break;
         }

         // Only need hDC in OnDraw/OnPaint where pDC is passed in.
         TMG_display_set_paint_hDC(IMV.m_hDisplay, pDC->GetSafeHdc());

         switch (m_pDlg->m_nDisplayDIB)
         {
            case DLG_DIB_DISPLAY:
               TMG_display_image(IMV.m_hDisplay, IMV.m_hDIBImage, TMG_RUN);
               break;
            case DLG_DDB_DISPLAY:
               TMG_display_image(IMV.m_hDisplay, IMV.m_hDDBImage, TMG_RUN);
               break;
            case DLG_DIRECT_DISPLAY:
               if (TMG_display_get_parameter(IMV.m_hDisplay,
                                   TMG_DISPLAY_DIRECT_CAPS) == 0)
```

```
                {  // DirectDraw for WinNT/95, DCI for Win31
                   PrintToStatusBar("A DirectDraw/DCI driver is not present");
                   eResult = ~ASL_OK;   // Things are not OK.
                }
                else
                {
                   TMG_display_set_flags(IMV.m_hDisplay, TMG_DISPLAY_DIRECT,
                                         TRUE);
                   TMG_display_image(IMV.m_hDisplay, IMV.m_hDDBImage, TMG_RUN);
                }
                TMG_display_set_flags(IMV.m_hDisplay, TMG_DISPLAY_DIRECT,
                                         FALSE);  // Switch off to keep tidy.
                break;
             } /* End switch statement */
             TMG_display_set_paint_hDC(IMV.m_hDisplay, 0);   /* Set back */
} /* End else display */
   }
}
```

The difference between the DIB (device independent bitmap) and DDB (device dependent bitmap) is the format of the images *IMV.m_hDIBImage* and *IMV.m_hDDBImage*. The DIB is a 24 bit file, generated using *TMG_image_convert* to format *TMG_BGR24* with *TMG_IS_DIB* set. The DDB is pixel format that matches that of the display. For example if the display format (see *TMG_display_get_parameter*) is *TMG_RGB16*, then the format of the DDB will also be *TMG_RGB16*. For example, the DDB image may have been generated using *TMG_image_convert* to *TMG_RGB16*. The DDB display method is much faster than the DIB method but sometimes the final rendered quality is not as good. (This is because the display driver may dither the 24 bit DIB down to RGB16.) Using DirectDraw is essentially the same as the DDB method except the DirectDraw method is generally faster.

IMAGE DISPLAY UNDER DOS

To display images under MS-DOS, the Flash Graphics library, by Flashtek Inc. is required. This is a low cost yet comprehensive graphics library that is royalty free and can be purchased with the SDK. Please contact you local distributor for information if you do not possess a copy.

The TMG library is a layer above the Flash Graphics library and converts TMG API calls into Flash Graphics functions calls. Only a small proportion of Flash Graphics routines are available through TMG calls and anyone seriously programming graphics under DOS should refer to the Flash Graphics manual to see what else is available.

The basic function groups are as follows:

- *TMG_display_create* and *TMG_display_init* are used to initialise the display.
- *TMG_display_set_ROI* is used to set a region of interest to display to, and *TMG_display_image* actually displays the image.
- *TMG_display_get_parameter* can be used to read back certain information about the display such as colour depth etc.

The following example code shows how an image would be displayed. The code has been lifted from the example application "s24dos" from the Snapper SDK. Please refer to this for more details.

```
Thandle Hdisplay;

Hdisplay = TMG_display_create();    /* Create a handle to the screen */

/* Initialise display to 800 by 600 by 65k colours */
if ASL_is_err(TMG_display_init(Hdisplay, TMG_800x600x16))
   printf("Failed to initialise display");

/* Convert to an RGB16 image and display */
TMG_image_convert(Hvid_image, Hdisplay_image, TMG_RGB16, 0, TMG_RUN);
TMG_display_image(Hdisplay, Hdisplay_image, TMG_RUN);
```

```
/* Switch back to the normal DOS prompt on exit */
TMG_display_init(Hdisplay, TMG_DOS_PROMPT);
```

The Flash Graphics library can be purchased from your local Snapper distributor.

IMAGE DISPLAY UNDER X WINDOWS

The X Window System, based on the X library, is the low level graphics interface used by most Unix type operating systems. This includes Solaris 2.x, LynxOS and VxWorks.

The TMG library is a layer above the X library and converts TMG API calls into Xlib functions calls. Only a small proportion of Xlib routines are available through TMG calls and anyone seriously programming graphics under X Windows should refer to the Xlib Programming Manual.

The basic function groups are as follows:

- *TMG_display_create* and *TMG_display_init* are used to initialise the display.
- *TMG_display_set_ROI* is used to set a region of interest to display to, and *TMG_display_image* actually displays the image.
- *TMG_display_set_Xid [X Windows]* is used with several different parameters (such as X Window ID) to set up the display.
- *TMG_display_get_parameter* can be used to read back certain information about the display such as colour depth, or number of reserved colours etc.

The following example code shows how an image would be displayed. The code has been lifted from the example application "Xtmg" from the Snapper LynxOS SDK. Please refer to this for more details.

```
/* This code would typically go in main() */
Thandle hDisplay;
Thandle hImage;
.
ASL_err_set_reporting(ASL_ERR_SET_HANDLER, ASL_err_display);
hImage   = TMG_image_create();
hDisplay = TMG_display_create();

/* Connect to X-server to obtain window and display information */
pdDisplay = XOpenDisplay(...);

/* Set the X Window ID before TMG_display_init */
TMG_display_set_Xid(hDisplay, TMG_XID_WINDOW, wWin);
/* Initialise the TMG display interface */
TMG_display_init(hDisplay, TMG_X_WINDOWS);
get_gc(wWin, &gcView, xfsFont);
XMapWindow( pdDisplay, wWin);

event_loop();

}  /* End main */
```

The X Windows programming convention uses an event handling loop which branches on user or system events. The 'update display' event would be used for display of the image on initial display and whenever the window is moved or re-sized:

```
void event_loop(void)
{
static int bFirst = TRUE;
```

```
while (TRUE)
{
   XNextEvent(pdDisplay, &xeReport);
   switch(xeReport.type)
   {
      case Expose:
         /*Don't redraw unless this is the last contiguous expose */
         if (bFirst == FALSE)
         {
            if (xeReport.xexpose.count != 0)
               break;
         }

         bFirst=FALSE;
         /* Draw something */
         dwStatus = TMG_display_image(hDisplay, hImage1, TMG_RUN);
         break;
      case ConfigureNotify:
        /* Window has been moved/re-sized, update any window size variables so
         * imminent redraw takes place correctly
         */
         break;
      case ButtonPress:
      case KeyPress:
         XUnloadFont(pdDisplay,xfsFont->fid);
         XFreeGC(pdDisplay,gcView);
         XCloseDisplay(pdDisplay);
         TMG_display_destroy(hDisplay);
         TMG_image_destroy(hImage1);
         exit(1);   /* In this example we use keypress to quit */
         break;
    default:
        /* All events selected by StructureNotifyMask except ConfigureNotify
       are
         * thrown away here since nothing is done with them.
         */
         break;
   } /* End switch     */
  }  /* End while      */
}    /* End Event Loop */
```

More examples can be found in the demonstration applications available with the Snapper LynxOS SDK.

The following example code shows how an image would be displayed under Solaris and OpenWindows. Although OpenWindows is no longer marketed by Sun, this example may still be useful.

```
/* This code would typically go in main() */
Thandle Hdisplay;
.
.
Hdisplay = TMG_display_create();

/* Set up the X Windows IDs before TMG_display_init */
TMG_display_set_Xid( Hdisplay, TMG_XID_FRAME,
       (Window) xv_get(View_Basewin->Basewin, XV_XID) );
TMG_display_set_Xid( Hdisplay, TMG_XID_CANVAS,
       (Window) xv_get(View_Basewin->BasewinCanvas, XV_XID) );

if ( TMG_display_init(Hdisplay, TMG_X_WINDOWS) == ASL_OK ) {
   if ( TMG_display_get_parameter(Hdisplay, TMG_DEPTH) == 8 ) {
```

```
        XReservedColours = (ui16) TMG_display_get_parameter(Hdisplay,
         TMG_RESERVED_COLOURS);
        fprintf(stderr, "\nDisplay initialised: %d free colours found.\n\n",
         (int) 256 - XReservedColours);

        if (XReservedColours > 16) {
           XReservedColours = 16;   /* we will use at least 240 colours */
           TMG_display_set_parameter(Hdisplay, TMG_RESERVED_COLOURS,
         XReservedColours);
      }
}
else
   fprintf(stderr, "\nDisplay initialised: 24 bit display.\n\n");
.
.
}  /* End main */
```

The X Window ID would typically be set from the repaint routine as shown below:

```
/*
 * Repaint callback function for `BasewinCanvas'.
 */
void BasewinRepaint(Canvas canvas, Xv_window paint_window, Display *display,
       Window xid, Xv_xrectlist *rects)
{

   TMG_display_set_Xid(Hdisplay, TMG_XID_WINDOW, xid);
.
.
}
```

More examples can be found in the demonstration applications available with the Snapper Solaris SDK.

IMAGE DISPLAY UNDER MACOS

The TMG display environment for MacOS uses the QuickDraw or Colour QuickDraw display manager to handle all display to the screen. Multiple screens are allowed as long as the QuickDraw manager supports it.

For in-depth details on the QuickDraw interface refer to the MacOS Toolbox reference manuals, available online from Apple or in hardback from bookshops.

The basic function groups are as follows:

- *TMG_display_create* and *TMG_display_init* are used to create and then initialise the display.
- *TMG_display_set_ROI* is used to set the location and clipping of an image on the display.
- *TMG_display_set_mask* is a MacOS-specific function used to set a mask region on the display for use in overlays.
- *TMG_display_get_parameter* can be used to read back certain information about the display such as colour or grey scale display, pixel depth and pixel format.
- The following example code shows how an image would be displayed. The code has been lifted from the example application "gui.c" from the Snapper SDK application example code. Please refer to this for more details.

```
struct GUI {……..} gui;
struct GUI* psGui;
```

```
/* This code would typically go in main() */
void main(void)
{
   Terr teStatus = ASL_OK;
   PixMapHandle hpmPlayThru;
   char TempString[MAX_FIELD_LENGTH];

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows(); /* Loads the window resource */
   InitMenus();   /* Loads the menu resource */
   TEInit();      /* Text-editor init - needed for window title display
       apparently */
   InitDialogs(nil); /* Needed for system alert message box & other stuff-
       apparently */
   InitCursor();

   /* Set up system memu entry for our window - 'QUIT' is the only entry. */
   psGui->hMenuBar = GetMenuBar( mTmgMBar );
   SetMenuBar( psGui->hMenuBar );
   DrawMenuBar();


  /* Make a new window for drawing in, and it must be a color window.
   * The display can be sized to the window size later.
   */
   strcpy( TempString, psGui->szWinName );
   psGui->pWin = NewCWindow( nil, &psGui->windRect, c2pstr(TempString), true,
       noGrowDocProc, (WindowPtr) -1, true, 0);

  /* set window to current graf port */
   SetPort( psGui->pWin );

   psGui->hDisplay = TMG_display_create( );
   if( psGui->hDisplay == TMG_INVALID_HANDLE )
      TMG_err_ret( ASL_ERROR, "Failed to acquire display ", 0 , szFnName );

   /* Obtain a valid PixMapWindow to display with */
   hpmPlayThru = GetWindowPort( psGui->pWin )->portPixMap;

   /* Initialise display with PixMapWindow */
   teStatus = TMG_display_init( psGui->hDisplay, hpmPlayThru );
   psGui->wDisplayFormat = TMG_display_get_parameter( psGui->hDisplay,
       TMG_PIXEL_FORMAT );
   psGui->wDisplayDepth = TMG_display_get_parameter( psGui->hDisplay,
       TMG_DEPTH );

   /* Set QuickDraw font size */
   TextSize( 8 );

   EventLoop();

}  /* End main */

/* The event loop consists of numerous events which must be detected and
       handled, the one of interest is the redraw event…*/
ui32 EventLoop ( void )
{
   Terr teStatus = ASL_OK;

   /* Mac variables */
   EventRecord evEvent;
   WindowPtr wWindow;
```

```
  /* Check for any user input - ie mouse or keystrokes.
   *  Set kSleep to 0 for immediate return if no message waiting – returns
   * FALSE & NULL event.
   * However value of 0 fails to allow the OS to do background processing -
   * VERY important for MacOS - nasty side effects otherwise!.
   */
   while ( (seCurrentEvent.dwCmd) != 'Q' &&
           WaitNextEvent( everyEvent, &evEvent, /*kSleep*/ 1, nil ) == true &&
           (!ASL_is_err( teStatus )) )
   {
      switch ( evEvent.what )
      {
         case nullEvent:/* No message found so break out should be caused by
                        * WaitNextEvent returning 'false' on nullEvent.
                        */
         case updateEvt:/* If the message is telling us that it is our window
                        * needing the update.
                        */
            if ( (psGui->pWin != NULL) &&
                 ((WindowPtr)evEvent.message == (WindowPtr )psGui->pWin) )
            {
               BeginUpdate( psGui->pWin);
               /* redisplay */
               TMG_image_display( psGui->hImage );
               EndUpdate( psGui->pWin);
            }
            break;
         default:
            break;
      }  /* End switch statement */
   }  /* End while */
return;
}
```

Sample Applications

This section contains either example applications or major code fragments that show how to use key areas of the TMG library.

All the examples shown have been extracted from real examples provided with the Snapper SDK. There are additional (more detailed) example applications in the SDK and it is strongly recommended that these are referred to before embarking on application development. The Windows NT/95/3.1 examples are from real applications using the Microsoft Foundation Class (MFC) application framework.

A SIMPLE TMG PROCESSING EXAMPLE

This example shows the basic operation of the TMG library. This program reads in a TIFF file (or in fact any supported file format), mirrors it and then writes it out as a TIFF file. The file contains a compiler pre-processor directive *_PROCESS_IN_1_STRIP* to determine whether to process the image in one strip or strips of 8 lines at a time. This example is from the file "process.c":

```
#include <asl_inc.h>

void main(ui16 argc, char** argv)
{
   Thandle Hin_image, Hout_image;
   ui16 strip, total_strips;
   ui32 lines_this_strip = 8;

   printf("\nTMG Image Processing Example - v3.0\n");
   printf("Usage : process <input_filename> <output_filename>\n");

   Hin_image = TMG_image_create();
   Hout_image = TMG_image_create();
   TMG_image_set_infilename(Hin_image, argv[1]);
   TMG_image_set_outfilename(Hin_image, argv[2]);

#ifdef _PROCESS_IN_1_STRIP
   /* Note generally only 32 bit applications can process the image in */
   /* 1 strip because of memory limitations / cache benefit */
   TMG_image_set_parameter(Hin_image, TMG_LINES_THIS_STRIP, TMG_AUTO_HEIGHT);
   total_strips = 1;
#else   /* Multiple strips */
   /* Check that input file exists, and determine its size */
   TMG_image_set_parameter(Hin_image, TMG_LINES_THIS_STRIP, 0);
   if ( TMG_image_read(Hin_image, NULL, TMG_RUN) != ASL_OK ) {
      printf("Failed to open file %s\n", argv[1]);
      exit(0);
   }
   TMG_image_read(Hin_image, NULL, TMG_RESET);
   TMG_image_set_parameter(Hin_image, TMG_LINES_THIS_STRIP, lines_this_strip);
   total_strips = (ui16) TMG_image_calc_total_strips(Hin_image);
#endif   /* multiple strips */

   for (strip = 0; strip < total_strips; strip++) {
      TMG_image_read(Hin_image, TMG_NULL, TMG_RUN);
      TMG_IP_mirror_image(Hin_image, Hout_image, TMG_RUN);
      TMG_image_write(Hout_image, TMG_NULL, TMG_TIFF, TMG_RUN);
   }
TMG_image_destroy(TMG_ALL_HANDLES);   /* Free the memory */
}
```

TEST PATTERN GENERATION

This example shows how to generate an image "from scratch" - in this case a grayscale ramp, and save it as a TIFF file. Note this example also illustrates how individual pixels may be accessed and modified, hence allowing image processing operations from a user application. This example is from the file "pgen.c":

```
#include <asl_inc.h>

void main(ui16 argc, char** argv)
{
   Thandle Himage;
   ui32 height, width;
   ui32 line, pixel;
   ui8  *Pdata;

   printf("\nTIFF File Pattern Generator - v3.0\n");

   /* Create an image */
   Himage = TMG_image_create();
   TMG_image_set_outfilename(Himage, "pattern.tif");

   /* OK lets make an image */
   width = 256;
   height = 256;
   TMG_image_set_parameter(Himage, TMG_WIDTH, width);
   TMG_image_set_parameter(Himage, TMG_HEIGHT, height);
   TMG_image_set_parameter(Himage, TMG_PIXEL_FORMAT, TMG_Y8);
   TMG_image_set_parameter(Himage, TMG_LINES_THIS_STRIP, height);

   if (TMG_image_check(Himage) != ASL_OK) /* Fills in bytes_per_line */
   {
      printf("PGEN: Corrupt image\n");
      exit(0);
   }

  /* This is an internal function that conveniently allocates a strip
   * of image data based on lines_this_strip and bytes_per_line. */
   if (TMG_image_malloc_a_strip(Himage) != ASL_OK)
   {
      printf("Failed to malloc sufficient memory\n");
      exit(0);
   }

   Pdata = (ui8*) TMG_image_get_ptr(Himage, TMG_IMAGE_DATA);

   /* Lets put a ramp pattern in the image */
   for (line = 0; line < height; line++)
      for (pixel = 0; pixel < width; pixel++)
      {
         *Pdata++ = pixel % 256;
      }

   TMG_image_write(Himage, NULL, TMG_TIFF, TMG_RUN);

   TMG_image_destroy(TMG_ALL_HANDLES);  /* Free the memory */
}
```

SOFTWARE JPEG DECOMPRESSION AND DISPLAY

This example shows how to use the TMG JPEG decompression functions to read in a JPEG file and display it. The input JPEG file is decompressed and displayed 8 lines at a time. This example is from the file "jview.c":

```
#include <asl_inc.h>

void main(ui16 argc, char** argv)
{
   Thandle Hjpeg_image, Htemp_image, Hdisp_image;
   Thandle Hdisplay;
   ui16 strip, total_strips;

   printf("\nJPEG Image Viewer - v3.0\n");

   /* Create image structures */
   Hdisplay = TMG_display_create();  /* create a handle to the screen */
   Hjpeg_image = TMG_JPEG_image_create();
   Htemp_image = TMG_image_create();
   Hdisp_image = TMG_image_create();
   TMG_image_set_infilename(Hjpeg_image, argv[1]]);

   if (TMG_JPEG_file_read(Hjpeg_image) != ASL_OK) {
      printf("Failed to read in JPEG file\n");
      exit(0);
   }

   if ( ASL_is_err(TMG_display_init(Hdisplay, TMG_800x600x16)) ) {
      printf("\nFailed to initialise VESA graphics card\n\n");
      exit(1);
   }

   TMG_image_set_parameter(Hjpeg_image, TMG_LINES_THIS_STRIP, 8);
   total_strips = (ui16) TMG_image_calc_total_strips(Hjpeg_image);

   for (strip = 0; strip < total_strips; strip++) {
      TMG_JPEG_decompress(Hjpeg_image, Htemp_image, TMG_RUN);
      TMG_image_convert(Htemp_image, Hdisp_image, TMG_RGB16, 0, TMG_RUN);
      TMG_display_image(Hdisplay, Hdisp_image, TMG_RUN);
   }

   /* return to the DOS prompt */
   TMG_display_init(Hdisplay, TMG_DOS_PROMPT);

   /* Free the memory used by the images */
   TMG_image_destroy(TMG_ALL_HANDLES);
}
```

SOFTWARE JPEG COMPRESSION

This example shows how to use the TMG JPEG functions to compress a TIFF file (or in fact any supported file format). The input file is read in (the whole image) then compressed 8 lines at a time to conserve memory, before being saved as a JPEG file. This example is from the file "compsw.c".

```
#include <asl_inc.h>

void main(ui16 argc, char** argv)
{
   Thandle Hin_image, Hjpeg_image;   /* input and output images */

   Hin_image = TMG_image_create();
   Hjpeg_image = TMG_JPEG_image_create();

   TMG_image_set_infilename(Hin_image, "in.tif");
   TMG_image_set_outfilename(Hin_image, "out.jpg");
```

```
   TMG_image_set_parameter(Hin_image, TMG_LINES_THIS_STRIP, 8);
   TMG_JPEG_set_Quality_factor(Hjpeg_image, 32);
   TMG_JPEG_compress_image_to_image(Hin_image, Hjpeg_image, TMG_FILE,
       TMG_FILE);

   TMG_image_destroy(TMG_ALL_HANDLES);
}
```

CONVERTING A 24 BIT COLOUR IMAGE TO A PALETTED IMAGE

This example shows how use the *TMG_cmap* functions to generate an optimum colour palette and use it to convert a 24 bit colour image to an 8 bit paletted one. (Note that if the image is processed in strips, as in this example, two passes are effectively needed). This example is from the file "cmap.c":

```
#include <asl_inc.h>

void main(ui16 argc, char** argv)
{
   Thandle Hin_image, Hout_image;
   ui16 strip, total_strips;
   ui32 lines_this_strip = 8;

   printf("\nTMG Colourmap/Palette Generation Example - v3.0\n");

   /* create the images */
   Hin_image = TMG_image_create();
   Hout_image = TMG_image_create();
   TMG_image_set_infilename(Hin_image, argv[1]);
   TMG_image_set_outfilename(Hin_image, argv[2]);

   /* work out the height and number of strips */
   TMG_image_set_parameter(Hin_image, TMG_LINES_THIS_STRIP, 0);
   if ( TMG_image_read(Hin_image, NULL, TMG_RUN) != ASL_OK ) {
      printf("Failed to open file %s\n", argv[1]);
      exit(0);
   }
   TMG_image_read(Hin_image, NULL, TMG_RESET);
   TMG_image_set_parameter(Hin_image, TMG_LINES_THIS_STRIP, lines_this_strip);
   total_strips = (ui16) TMG_image_calc_total_strips(Hin_image);

   printf("Generating palette");
   for (strip = 0; strip < total_strips; strip++)
   {
      TMG_image_read(Hin_image, TMG_NULL, TMG_RUN);
      TMG_cmap_generate(Hin_image, 256, TMG_RUN);
   }

   printf("\nMapping image to palette");
   for (strip = 0; strip < total_strips; strip++)
   {
      TMG_image_read(Hin_image, TMG_NULL, TMG_RUN);
      TMG_image_convert(Hin_image, Hout_image, TMG_PALETTED, 0, TMG_RUN);
      TMG_image_write(Hout_image, TMG_NULL, TMG_TIFF, TMG_RUN);
   }
   printf("complete\n");

   /* Free the memory */
   TMG_image_destroy(TMG_ALL_HANDLES);
}
```

DISPLAYING COLOUR AND GRAYSCALE IMAGES SIMULTANEOUSLY TO A PALETTED DISPLAY

This example code fragment shows how use the *TMG_cmap* functions to set up the palette so that a colour and grayscale image can be displayed simultaneously with reasonable quality. The trick here is to generate a palette that has a sufficient mix of grayscale tones and colours. The first example uses an equal spread of colours and the second example generates an optimum palette based on an acquired video image, having already reserved standard VGA colours and a selection of grayscales.

```
/* Set up the colourmap */
TMG_cmap_set_type(Hyuv_image, TMG_332_RGB);  /* even spread of colours */

/* add standard VGA colours */
TMG_cmap_set_type(Hyuv_image, TMG_VGA16);
TMG_image_set_parameter(Hyuv_image, TMG_CMAP_SIZE, 256);

/* now set some additional grayscales */
TMG_cmap_set_RGB_colour(Hyuv_image, 16, 32, 32, 32);
TMG_cmap_set_RGB_colour(Hyuv_image, 17, 48, 48, 48);
TMG_cmap_set_RGB_colour(Hyuv_image, 18, 64, 64, 64);
TMG_cmap_set_RGB_colour(Hyuv_image, 19, 80, 80, 80);
TMG_cmap_set_RGB_colour(Hyuv_image, 20, 96, 96, 96);
TMG_cmap_set_RGB_colour(Hyuv_image, 21, 112, 112, 112);
TMG_cmap_set_RGB_colour(Hyuv_image, 22, 128, 128, 128);
TMG_cmap_set_RGB_colour(Hyuv_image, 23, 144, 144, 144);
TMG_cmap_set_RGB_colour(Hyuv_image, 24, 160, 160, 160);
TMG_cmap_set_RGB_colour(Hyuv_image, 25, 176, 176, 176);
TMG_cmap_set_RGB_colour(Hyuv_image, 26, 192, 192, 192);
TMG_cmap_set_RGB_colour(Hyuv_image, 27, 208, 208, 208);
TMG_cmap_set_RGB_colour(Hyuv_image, 28, 224, 224, 224);
TMG_cmap_set_RGB_colour(Hyuv_image, 29, 240, 240, 240);

/* put the colourmap into the display palette */
TMG_display_cmap_install(Hdisplay1, Hyuv_image);

/* display the palette - just for interest */
TMG_display_cmap(Hdisplay1, Hyuv_image, TMG_RUN);

/* now generate our YUV to paletted LUT for displaying colour images
 * captured from Snapper-16.  Note this will take several seconds,
 * or we could load a previously saved one.
 */
#ifdef _SAVE_LUT
TMG_image_conv_LUT_generate(Hyuv_image, TMG_YUV422_TO_PALETTED_LUT);
TMG_image_conv_LUT_save(Hyuv_image, TMG_YUV422_TO_PALETTED_LUT,
      "convlut.bin");
#else
TMG_image_conv_LUT_load(Hyuv_image, TMG_YUV422_TO_PALETTED_LUT,
      "convlut.bin");
#endif

/* and the same for the grayscale images... */
TMG_cmap_copy(Hyuv_image, Hy8_image);  /* give Hy8 the same colourmap */
TMG_image_conv_LUT_generate(Hy8_image, TMG_Y8_TO_PALETTED_LUT);
```

This second example is basically the same, except an optimum palette is generated. When generating an optimum palette, the image used as the reference image to generate the palette must contain a good mix of all the colours that can be expected in the live application situation.

```
/* enter standard VGA colours */
TMG_cmap_set_type(Hyuv_image,  TMG_VGA16);
```

```
TMG_image_set_parameter(Hyuv_image, TMG_CMAP_SIZE, 30);

/* now set some additional grayscales */
TMG_cmap_set_RGB_colour(Hyuv_image, 16, 32, 32, 32);
TMG_cmap_set_RGB_colour(Hyuv_image, 17, 48, 48, 48);
TMG_cmap_set_RGB_colour(Hyuv_image, 18, 64, 64, 64);
TMG_cmap_set_RGB_colour(Hyuv_image, 19, 80, 80, 80);
TMG_cmap_set_RGB_colour(Hyuv_image, 20, 96, 96, 96);
TMG_cmap_set_RGB_colour(Hyuv_image, 21, 112, 112, 112);
TMG_cmap_set_RGB_colour(Hyuv_image, 22, 128, 128, 128);
TMG_cmap_set_RGB_colour(Hyuv_image, 23, 144, 144, 144);
TMG_cmap_set_RGB_colour(Hyuv_image, 24, 160, 160, 160);
TMG_cmap_set_RGB_colour(Hyuv_image, 25, 176, 176, 176);
TMG_cmap_set_RGB_colour(Hyuv_image, 26, 192, 192, 192);
TMG_cmap_set_RGB_colour(Hyuv_image, 27, 208, 208, 208);
TMG_cmap_set_RGB_colour(Hyuv_image, 28, 224, 224, 224);
TMG_cmap_set_RGB_colour(Hyuv_image, 29, 240, 240, 240);

/* grab a reference colour image */
SNP16_set_input_mode(Hsnp16, input_mode);
SNP16_set_format(Hsnp16, SNP16_FORMAT_YUV422, TMG_YUV422);

/* some extra captures to allow colour lock to settle */
SNP16_capture(Hsnp16); SNP16_capture(Hsnp16);
SNP16_capture(Hsnp16); SNP16_capture(Hsnp16);
SNP16_capture(Hsnp16); SNP16_capture(Hsnp16);

for (strip = 0; strip < total_strips; strip++) {
   SNP16_read_video_data(Hsnp16, Hyuv_image, TMG_RUN);
   TMG_image_convert(Hyuv_image, Himage1, TMG_RGB24, 0, TMG_RUN);

/* generate an optimum palette - not using the reserved
 * colours already in the colourmap.  The resultant
 * colourmap shall be 256 colours.
 */
   TMG_cmap_generate(Himage1, 256, TMG_RUN);  /* 256 colours in total */
}
TMG_cmap_copy(Himage1, Hyuv_image);
TMG_cmap_copy(Hyuv_image, Hy8_image);

/* Force display routines to regenerate new LUTs */
TMG_image_conv_LUT_destroy(TMG_YUV422_TO_PALETTED_LUT);
TMG_image_conv_LUT_destroy(TMG_Y8_TO_PALETTED_LUT);

/* now write the colourmap into the display hardware */
TMG_display_cmap_install(Hdisplay1, Hyuv_image);
TMG_display_cmap(Hdisplay1, Hyuv_image, TMG_RUN);  /* for interest */
```

LOOK UP TABLE EXAMPLES - USING TMG LUT FUNCTIONS

The first code fragment shows how the *TMG LUT* functions may be used within an application to vary the brightness, contrast, gamma or colour balance of a colour image prior to displaying it.

```
Thandle hLUT;
i16 Brightness = TMG_DEFAULT_BRIGHTNESS;
i16 Contrast = TMG_DEFAULT_CONTRAST;
i16 Gamma = TMG_DEFAULT_GAMMA;
i16 Ri = TMG_DEFAULT_INTENSITY;
i16 Gi = TMG_DEFAULT_INTENSITY;
i16 Bi = TMG_DEFAULT_INTENSITY;
.
hLUT = TMG_LUT_create();
```

```
.
/* set up default parameters in hLUT */
TMG_LUT_generate(hLUT, brightness, contrast, gamma, ri, gi, bi);
.
/* we may vary the LUT parameters here... */
.
/* apply the software LUT function */
TMG_LUT_apply(Himage1, Himage2, hLUT, TMG_RUN);

/* now display the result of the LUT operation */
TMG_display_image(Hdisplay, Himage2, TMG_RUN);
```

The following example is a code fragment that uses the *TMG LUT* functions to generate LUTs that are subsequently used to program hardware LUTs contained in image acquisition hardware:

```
// Set up the hardware LUTs - using TMG_LUT functions to generate them
// (The parameters have been set by sliders via the application GUI)
TMG_LUT_generate(S24.m_hLut, S24.m_pLutDlg->m_nBrightnessLevel,
                             S24.m_pLutDlg->m_nContrastLevel,
                             S24.m_pLutDlg->m_nGammaLevel,
                             S24.m_pLutDlg->m_nRedILevel,
                             S24.m_pLutDlg->m_nGreenILevel,
                             S24.m_pLutDlg->m_nBlueILevel);

// Now extract pointers to the actual data
pLutRed   = (ui8*) TMG_LUT_get_ptr(S24.m_hLut, TMG_RED);
pLutGreen = (ui8*) TMG_LUT_get_ptr(S24.m_hLut, TMG_GREEN);
pLutBlue  = (ui8*) TMG_LUT_get_ptr(S24.m_hLut, TMG_BLUE);

// Set the hardware LUTs in Snapper-24
SNP24_set_LUTs(S24.m_hSnapper, SNP24_LUT_SET, SNP24_252_RED, pLutRed);
SNP24_set_LUTs(S24.m_hSnapper, SNP24_LUT_SET, SNP24_252_GRN, pLutGreen);
SNP24_set_LUTs(S24.m_hSnapper, SNP24_LUT_SET, SNP24_252_BLU, pLutBlue);
```

CHROMA KEYING

This example shows how to use the chroma keying functions. It shows how to calibrate to a chroma screen and then key in a background colour. This example is from the file "chroma.c" and assumes the use of the Snapper-16 video acquisition module:

```
/* This code fragment shows how to calibrate the background (screen) */
SNP16_capture(Hsnp16);
for (strip = 0; strip < total_strips; strip++) {
   SNP16_read_video_data(Hsnp16, Hvid_image, TMG_RUN);
   TMG_CK_calibrate(Hvid_image, Hchroma_key, TMG_RUN);
}
key_colour = TMG_GREEN;  /* key to green */
```

This next code fragment is the inner part of the capture and display loop, using Snapper-16:

```
SNP16_capture(Hsnp16);
for (strip = 0; strip < total_strips; strip++) {
   SNP16_read_video_data(Hsnp16, Hvid_image, TMG_RUN);
   if ((key_to_ref_image == TRUE) && (chroma_keying == TRUE)) {
#ifdef _DOS16  /* must read from disk */
      TMG_image_set_parameter(Himage3, TMG_LINES_THIS_STRIP, lines_per_strip);
      TMG_image_set_infilename(Himage3, ref_image_filename);
```

```
      TMG_image_read(Himage3, TMG_NULL, TMG_RUN);
      TMG_image_convert_to_YUV422(Himage3, Himage4, TMG_YUV422, 0, TMG_RUN);
#else  /* 32 bit */
      TMG_image_set_parameter(Href_image, TMG_LINES_THIS_STRIP,
       lines_per_strip);
      TMG_image_read(Href_image, Himage4, TMG_RUN);
#endif
      TMG_CK_chroma_key(Hvid_image, Himage1, Himage4, key_colour, Hchroma_key,
       filter, TMG_RUN);
   }
   else if ((key_to_ref_image == FALSE) && (chroma_keying == TRUE))
      TMG_CK_chroma_key(Hvid_image, Himage1, TMG_NULL, key_colour,
       Hchroma_key, filter, TMG_RUN);
   else
      TMG_image_move(Hvid_image, Himage1);

   TMG_image_convert(Himage1, Himage2, TMG_RGB16, TMG_USE_LUT, TMG_RUN);
   TMG_display_image(Hdisplay, Himage2, TMG_RUN);
}
TMG_image_set_parameter(Hvid_image, TMG_LINES_THIS_STRIP, lines_per_strip);
```

Function List

This section groups the TMG functions logically. Each function described in detail alphabetically in the next section.

GENERAL PURPOSE FUNCTIONS

TMG_image_create
TMG_image_destroy

TMG_image_copy
TMG_image_move
TMG_image_is_colour

TMG_image_check
TMG_image_calc_total_strips
TMG_image_find_file_format

TMG_image_malloc_a_strip
TMG_image_free_data

TMG_image_get_flags
TMG_image_get_parameter
TMG_image_get_ptr
TMG_image_get_infilename,
TMG_image_get_outfilename

TMG_image_set_flags
TMG_image_set_parameter
TMG_image_set_ptr
TMG_image_set_infilename,
TMG_image_set_outfilename

PIXEL FORMAT CONVERSION FUNCTIONS (AND RELATED)

TMG_image_convert
TMG_image_conv_LUT_generate
TMG_image_conv_LUT_destroy
TMG_image_conv_LUT_load
TMG_image_conv_LUT_save

IMAGE READING AND WRITING FUNCTIONS

TMG_image_read
TMG_image_write

COLOURMAP/PALETTE RELATED FUNCTIONS

TMG_cmap_copy
TMG_cmap_find_closest_colour
TMG_cmap_generate
TMG_cmap_get_occurrences
TMG_cmap_get_RGB_colour
TMG_cmap_is_grayscale
TMG_cmap_set_colour
TMG_cmap_set_RGB_colour
TMG_cmap_set_type

IMAGE PROCESSING FUNCTIONS

SPECIAL PROCESSING FUNCTIONS

JPEG RELATED FUNCTIONS

CHROMA KEYING AND RELATED FUNCTIONS

LOOK UP TABLE (LUT) FUNCTIONS

GENERIC DISPLAY FUNCTIONS

WINDOWS NT, 95 & 3.1 SPECIFIC DISPLAY (AND PRINTING) FUNCTIONS:

DOS SPECIFIC DISPLAY FUNCTIONS:

X WINDOWS SPECIFIC DISPLAY FUNCTIONS:

MacOS SPECIFIC DISPLAY FUNCTIONS:

The functions are described in alphabetical order in the following pages.

THIS PAGE IS INTENTIONALLY BLANK

TMG_CK_calibrate

USAGE

Terr TMG_CK_calibrate(Thandle Hin_image, Thandle Hchroma_key, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hchroma_key	Handle to a chroma keying structure.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function generates suitable values for the chroma keying structure, referenced by *Hchroma_key*. It finds the average hue and luminance of the image and assigns it to the chroma keying structure.

To use this function, remove all subject material from in front of the chroma screen and capture an image. It is very important that the chroma screen fills the whole image and that nothing else is in the view of the camera. Also suitable lighting must be used - for example tungsten studio lights or diffused daylight. Fluorescent lights **will not** work.

The hue tolerance is set to 20 degrees and the luminance tolerance set to 64. (This means ± 20 and ± 64 respectively.)

Note that this function may well be useful for applications simply requiring the average luminance or hue in an image (and not needing any chroma keying functionality). *TMG_CK_get_parameter* can be used to read the actual settings generated.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following example shows how to find the average brightness and hue in a colour image:

```
Thandle hChromaKey;
Thandle hImage, hYUVImage;

hChromaKey = TMG_CK_create();
hImage = TMG_image_create();
hYUVImage = TMG_image_create();

/* sky.tif is a 24 bit RGB colour image */
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_image_convert(hImage, hYUVImage, TMG_YUV422, 0, TMG_RUN);
TMG_CK_calibrate(hYUVImage, hChromaKey, TMG_RUN);

/* now we can read the average brightness and hue (if we want)*/
AverageLuma = (ui16) TMG_CK_get_parameter(hChromaKey, TMG_LUMA);
AverageHue = (ui16) TMG_CK_get_parameter(hChromaKey, TMG_HUE);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

Hin_image must be a 16 bit YUV 4:2:2 image *(TMG_YUV422)*.

SEE ALSO

TMG_CK_create, *TMG_CK_get_parameter*.

TMG_CK_chroma_key

USAGE

Terr TMG_CK_chroma_key(Thandle Hin_image, Thandle Hout_image, Thandle Href_image, ui16 colour, Thandle Hchroma_key, ui16 filter, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
Href_image	Handle to an image to key to (or *TMG_NULL*)
colour	The colour to key to.
Hchroma_key	Handle to a chroma keying structure.
filter	Either *TRUE* or *FALSE* - selects horizontal filtering.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function performs chroma keying on the input image, *Hin_image*, and generates a chroma keyed output image, *Hout_image*. If *Href_image* is *TMG_NULL*, then the function will key to the colour defined by *colour*. *colour* can be one of *TMG_RED, TMG_GREEN* etc - see *TMG_cmap_set_colour* for a complete list of available colours.

The chroma keying structure, reference by *Hchroma_key*, contains information about the hue (angle), hue tolerance, luminance and luminance tolerance. For each pixel in the input image, if its hue and luminance are within the limits defined by *Hchroma_key*, the key colour (or reference image if defined) will be used, otherwise the input image will be used to generate the output image.

The image type for the input image and optional reference image must be *TMG_YUV422*. Also the reference must be the same size (in terms of width and height) as the input image. *TMG_image_convert* (to *TMG_YUV422*) can be used to generate a reference image from a 24 bit RGB image if required.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

Hin_image and *Href_image* (if used) must both be a 16 bit YUV 4:2:2 image *(TMG_YUV422)*.

SEE ALSO

TMG_CK_create, *TMG_CK_calibrate*, *TMG_CK_set_parameter*.

TMG_CK_create

USAGE

Terr TMG_CK_create()

ARGUMENTS

None.

DESCRIPTION

This function creates a *Tchroma_key* structure by the use of malloc, and returns a handle to the structure. The contents of the chroma keying structure is shown below along with their default initialization values. (Refer to the file "tmg.h" for the actual structure definition.)

```
ui16 hue = 340;          /* typical blue chroma screen hue */
ui16 hue_tol = 20;       /* +/- 20 degrees */
ui16 luma = 120;         /* suitable luminance value and tolerance */
ui16 luma_tol = 64;
```

The handle to this structure is used by the chroma keying functions.

RETURNS

On success a valid handle is returned in the lower 16 bits of the return value (the upper 16 bits will be 0). On failure an error code will be returned in the upper 16 bits as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code creates a chroma keying structure and gets a handle to it:

```
Thandle hChromaKey;

if ( ASL_is_err(hChromaKey = TMG_CK_create() )
   printf("Failed to create LUT");
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_CK_destroy, *TMG_CK_chroma_key*, *TMG_CK_set_parameter*.

TMG_CK_destroy

USAGE

Terr TMG_CK_destroy(Thandle Hchroma_key)

ARGUMENTS

Hchroma_key Handle to a chroma keying structure or *TMG_ALL_HANDLES*.

DESCRIPTION

This function destroys a chroma keying structure by freeing all the memory associated with that structure.

If the parameter *TMG_ALL_HANDLES* is used, all previously created chroma keying structures are destroyed and their associated handles freed.

TMG_image_destroy(*TMG_ALL_HANDLES*) will destroy all TMG chroma keying structures by calling *TMG_CK_destroy* for all chroma key handles. This is a convenient way of destroying everything with just one function call.

RETURNS

ASL_OK.

EXAMPLES

The following code destroys a previously created chroma keying structure:

```
Thandle hChromaKey;
.
/* destroy the chroma keying structure */
TMG_CK_destroy(hChromaKey);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_CK_create, *TMG_image_destroy*.

TMG_CK_destroy_UV_to_hue_LUT

USAGE

Terr TMG_CK_destroy_UV_to_hue_LUT()

ARGUMENTS

None

DESCRIPTION

This function destroys the UV to hue LUT previously generated using the function *TMG_CK_generate_UV_to_hue_LUT*

TMG_image_destroy(*TMG_ALL_HANDLES*) destroys all TMG structures including this UV to hue LUT and may therefore be a more convenient way of destroying this LUT.

Note that this LUT is not related to the *TMG_LUT* suite of functions or the TMG image conversion LUTs.

RETURNS

ASL_OK.

EXAMPLES

The following code destroys the UV to LUT structure:

```
/* destroy the UV to hue LUT */
TMG_CK_destroy_UV_to_hue_LUT();
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_CK_generate_UV_to_hue_LUT, *TMG_image_destroy*.

TMG_CK_generate_UV_to_hue_LUT

USAGE

Terr TMG_CK_generate_UV_to_hue_LUT()

ARGUMENTS

None

DESCRIPTION

This function generates a LUT to convert from UV (i.e. the colour components of a YUV 4:2:2 image) to hue (i.e. the angle representing the colour). This LUT is used by the functions *TMG_CK_chroma_key* and *TMG_CK_calibrate*. If the LUT is not generated when these functions are called, it is automatically generated. This function is sometimes useful to generate the LUT in advance of actually using it (so as to save time).

The memory used by the LUT is dynamically allocated when the LUT is generated.

Note that this LUT is not related to the *TMG_LUT* suite of functions or the TMG image conversion LUTs.

TMG_CK_destroy_UV_to_hue_LUT can be used to destroy this LUT (i.e. free the allocated memory), but *TMG_image_destroy(TMG_ALL_HANDLES)* will automatically destroy all TMG structures including this LUT.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment generates the UV to hue conversion LUT:

```
/* Generate the LUT */
if ( ASL_is_err(TMG_CK_generate_UV_to_hue_LUT()) )
   printf("Failed to generate LUT");
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

The LUT size is 64K words (16 bit words) under all operating systems, except that under Windows 3.1, the LUT size is 64K bytes, which limits the hue resolution of the output to two degrees instead of one.

There are no known bugs.

SEE ALSO

TMG_image_convert, *TMG_image_conv_LUT_destroy*, *TMG_image_conv_LUT_save*, *TMG_image_conv_LUT_load*.

TMG_CK_get_component

USAGE

Terr TMG_CK_get_component(ui16 colour, ui16 component)

ARGUMENTS

colour	Colour (e.g. *TMG_RED, TMG_GREEN* etc).
component	Parameter to select - one of Y, U or V components

DESCRIPTION

This function returns the Y, U or V component value of a particular colour. See *TMG_cmap_set_colour* for a complete list of available colours.

The Y output has a range of 16..255. The U and V outputs have a range of 0..255. This YUV format is identical to the output of *Snapper-16* (composite/S-Video acquisition module). See *TMG_image_convert* for a description of the conversion formula.

This function calls *TMG_CK_get_YUV_values* internally (which in turn calls *TMG_CK_get_YUV_values_RGB).*

RETURNS

The Y, U or V component as the lower 8 bits of the 32 bit return value, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following example shows how to find the luminance component of the VGA defined colour blue.

```
YComp = TMG_CK_get_component(TMG_BLUE, TMG_Y_COMPONENT);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_CK_get_YUV_values,
TMG_CK_get_YUV_values_RGB.

TMG_CK_get_parameter

USAGE

Terr TMG_CK_get_parameter(Thandle Hchroma_key, ui16 parameter)

ARGUMENTS

Hchroma_key	Handle to a chroma keying structure.
parameter	The parameter type to be read.

DESCRIPTION

This function returns selected parameters from the chroma keying structure referenced by *Hchroma_key*. Each parameter is described below:

TMG_HUE	Return the current hue.
TMG_HUE_TOL	Return the current hue tolerance.
TMG_LUMA	Return the current luminance.
TMG_LUMA_TOL	Return the current luminance tolerance.

RETURNS

The selected parameter in the lower 16 bits of the return value, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code reads the current hue setting (without error checking):

```
Hue = ASL_get_ret( TMG_CK_set_parameter(hChromaKey, TMG_HUE) );
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_CK_create, *TMG_CK_set_parameter*.

TMG_CK_get_YUV_values, TMG_CK_get_YUV_values_RGB

USAGE

ui32 TMG_CK_get_YUV_values(ui16 colour)

ui32 TMG_CK_get_YUV_values_RGB(ui8 red, ui8 green, ui8 blue)

ARGUMENTS

colour	Colour (e.g. *TMG_RED, TMG_GREEN etc*).
red	Red value 0..255.
green	Green value 0..255.
blue	Blue value 0..255.

DESCRIPTION

TMG_CK_get_YUV_values returns the YUV values of a particular colour defined by for example *TMG_RED*. See *TMG_cmap_set_colour* for a complete list of available colours.

TMG_CK_get_YUV_values_RGB returns the YUV values for a particular colour defined by individual red, green and blue intensities.

The Y output has a range of 16..255. The U and V outputs have a range of 0..255. This YUV format is identical to the output of *Snapper-16* (composite/S-Video acquisition module). See *TMG_image_convert* for a description of the conversion formula.

RETURNS

A 32 bit unsigned integer is returned with Y in the lower 8 bits, U in bit positions 8 to 15, and V in bit positions 16 to 23.

EXAMPLES

The following example shows how to find the luminance component of the colour defined by the following intensities: red 240, green 100, blue 220:

```
YUVComp = TMG_CK_get_YUV_values_RGB(240, 100, 220);
YComp = (ui8) YUVComp;
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_CK_get_component.

TMG_CK_set_parameter

USAGE

Terr TMG_CK_set_parameter(Thandle Hchroma_key, ui16 parameter, ui16 value)

ARGUMENTS

Hchroma_key	Handle to a chroma keying structure.
parameter	The parameter type to be set.
value	The actual required value.

DESCRIPTION

This function sets parameters in the chroma keying structure referenced by *Hchroma_key*. Each parameter is described below:

TMG_HUE	Set the desired hue in degrees from 0 to 359.
TMG_HUE_TOL	Set the hue tolerance - a typical value for a reasonable quality chroma screen would be 20.
TMG_LUMA	Set the desired luminance or brightness from 0 to 255.
TMG_LUMA_TOL	Set the luminance tolerance - a typical value would be 64.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment sets values for a typical blue chroma keying screen:

```
/* Set values for a typical blue chroma screen */
TMG_CK_set_parameter(hChromaKey, TMG_HUE, 340);
TMG_CK_set_parameter(hChromaKey, TMG_HUE_TOL, 20);
TMG_CK_set_parameter(hChromaKey, TMG_LUMA, 120);
TMG_CK_set_parameter(hChromaKey, TMG_LUMA_TOL, 64);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_CK_create, *TMG_CK_get_parameter*.

TMG_cmap_copy

USAGE

Terr TMG_cmap_copy(Thandle Hin_image, Thandle Hout_image)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.

DESCRIPTION

This function copies the colourmap from the input image, *Hin_image*, to the output image, *Hout_image*. This function is sometimes useful when using *TMG_cmap_generate*.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment copies the colourmap after generating an optimum colourmap:

```
TMG_cmap_generate(hImage, 256, TMG_RUN);
/* hSrcImage is the first function used in a chain elsewhere */
TMG_cmap_copy(hImage, hSrcImage);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_cmap_generate.

TMG_cmap_generate

USAGE

Terr TMG_cmap_generate(Thandle Himage, ui16 num_colours, ui16 TMG_action)

ARGUMENTS

Himage	Handle to an image.
num_colours	The number of final colours (including any reserved colours).
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function generates an optimum colourmap for *Himage*. *Himage* must be a 24 bit colour image (type *TMG_RGB24*) or an 8 bit grayscale image (type *TMG_Y8*). The function analyses the number and type of colours/gray levels using a proprietary histogram technique and generates a new colourmap for *Himage*.

The function uses the input parameter, *num_colours*, to decide how many colours entries the resulting colourmap will have. The size and contents of *Himage*'s colourmap on entry to the function determines how many colours to reserve. For example, 16 colours may be set in colourmap locations 0..15 and the colourmap size set to 16. Then *TMG_cmap_generate* would be called with 256 colours as a parameter, which would mean 240 colours optimised to the image would be stored in the remainder of the locations.

Note that the whole image must be processed before the colourmap can be used by other functions, such as *TMG_image_convert*. In other words if the image is being processed in strips, the strip loop generating the colourmap must complete before the (separate) strip loop that uses the colourmap starts.

A colourmap can be saved and re-loaded by simply saving the image as a paletted image - this will force the colourmap to be saved with the image (and re-loaded on read).

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_conv_LUT_generate, *TMG_image_convert*, *TMG_cmap_set_type*, *TMG_cmap_get_occurrences*.

TMG_cmap_get_occurrences

USAGE

ui32 TMG_cmap_get_occurrences(Thandle Himage, ui16 index)

ARGUMENTS

Himage	Handle to an image.
index	Index into the *Himage's* colourmap (0..255).

DESCRIPTION

This function returns the number of occurrences of a particular colour (or gray level) referenced by *index*, where *index* is the location of the colour in *Himage's* colourmap. Note that for grayscale images, generating a colourmap and then using this function is a convenient way of returning histogram information about the image. (For grayscale images the index value is the same as the actual gray level intensity as long as the colourmap is allowed to have 256 entries.)

This function can only be used after *TMG_cmap_generate* has been used on *Himage*.

RETURNS

The number of occurrences of a particular colour as a 32 bit word.

EXAMPLES

The following example shows how to generate a colourmap for an image and then print out a histogram of the colour index versus the number of occurrences:

```
ui16 n;
ui32 num;

TMG_image_set_parameter(Himage, TMG_CMAP_SIZE, 0);  /* none reserved */
TMG_cmap_generate(Himage, 256, TMG_RUN);

printf("Histogram of 256 most popular colours:\n");
for (n = 0; n < 256; n++) {
   num = TMG_cmap_get_occurrences(Himage, n);
   printf("Index %d occurred %ld times\n", (int) n, (long) num);
}
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_cmap_generate, *TMG_cmap_get_RGB_colour*, *TMG_cmap_find_closest_colour*.

TMG_cmap_get_RGB_colour

USAGE

ui32 TMG_cmap_get_RGB_colour(Thandle Himage, ui16 index)

ARGUMENTS

Himage	Handle to an image.
index	Index into the *Himage's* colourmap (0..255).

DESCRIPTION

This function returns the RGB value of the colour at position *index* in *Himage's* colourmap. The red, green and blue component are packed into a 32 bit return value such that red occupies bits 16..23, green 9..15 and blue 0..7.

RETURNS

A 32 bit unsigned integer is returned with blue in the lower 8 bits, green in bit positions 8 to 15, and red in bit positions 16 to 23.

EXAMPLES

The following example shows how to read a colour from an image's colourmap:

```
ui32 PackedColour;
ui8 red, grn, blu;

PackedColour = TMG_cmap_get_RGB_colour(Himage, 0);
red = (ui8) (PackedColour >> 16);
grn = (ui8) (PackedColour >> 8);
blu = (ui8) PackedColour;
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_cmap_set_RGB_colour.

TMG_cmap_find_closest_colour

USAGE

ui8 TMG_cmap_find_closest_colour(Thandle Himage, ui8 red, ui8 green, ui8 blue)

ARGUMENTS

Himage	Handle to an image.
red	Red intensity of input colour.
green	Green intensity of input colour.
blue	Blue intensity of input colour.

DESCRIPTION

This function returns the index of the closest colour in *Himage's* colourmap to the input colour defined by the intensities of *red*, *green* and *blue*. The least squares algorithm is used (it uses a LUT to do the multiplication so its relatively fast).

RETURNS

The index of the closest colour as an 8 bit unsigned integer.

EXAMPLES

The following example shows how to find the nearest colour to a fully saturated red:

```
ui8 index;
ui32 PackedColour;
ui8 red, grn, blu;

index = TMG_cmap_find_closest_colour(Himage, 255, 0, 0);
PackedColour = TMG_cmap_get_RGB_colour(Himage, (ui16) index);
red = (ui8) (PackedColour >> 16);
grn = (ui8) (PackedColour >> 8);
blu = (ui8) PackedColour;
printf("Closest colour is R: %d, G: %d, B: %d", (int) red, (int) grn, (int)
       blu);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_cmap_get_RGB_colour, *TMG_cmap_generate*.

TMG_cmap_is_grayscale

USAGE

Tboolean TMG_cmap_is_grayscale(Thandle Himage)

ARGUMENTS

Himage Handle to an image.

DESCRIPTION

Returns *TRUE* if *Himage's* colourmap contains only gray levels.

RETURNS

Returns *TRUE* or *FALSE*.

EXAMPLES

The following code fragment reads an image and determines if it is colour or not (this is actually a slightly simplified version of *TMG_image_is_colour*):

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
if (TMG_image_get_parameter(hImage, TMG_PIXEL_FORMAT) == TMG_PALETTED)
   if (TMG_cmap_is_grayscale(hImage) == TRUE)
      printf("We have a grayscale paletted image");
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_is_colour.

TMG_cmap_set_colour

USAGE

Terr TMG_cmap_set_colour(Thandle Himage, ui16 index, ui16 colour)

ARGUMENTS

Himage	Handle to an image.
index	The colourmap entry.
colour	The colour, which can be one of the following:

TMG_BLACK	*TMG_GRAY*
TMG_BLUE	*TMG_LIGHT_BLUE*
TMG_GREEN	*TMG_LIGHT_GREEN*
TMG_CYAN	*TMG_LIGHT_CYAN*
TMG_RED	*TMG_LIGHT_RED*
TMG_MAGENTA	*TMG_LIGHT_MAGENTA*
TMG_YELLOW	*TMG_LIGHT_YELLOW*
TMG_WHITE	*TMG_LIGHT_WHITE*

DESCRIPTION

This function sets a colour in *Himage's* colourmap using one of the above colours. The non-light colours have individual colour intensities of 152 and the light colours have intensities of 255.

Internally this functions calls *TMG_cmap_set_RGB_colour*.

This function is generally used in conjunction with *TMG_cmap_generate*.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment sets three colours:

```
TMG_cmap_set_colour(hImage, 0, TMG_LIGHT_RED);
TMG_cmap_set_colour(hImage, 1, TMG_LIGHT_WHITE);
TMG_cmap_set_colour(hImage, 2, TMG_LIGHT_BLUE);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_cmap_set_RGB_colour, *TMG_cmap_set_type*.

TMG_cmap_set_RGB_colour

USAGE

Terr TMG_cmap_set_RGB_colour(Thandle Himage, ui16 index, ui8 red, ui8 green, ui8 blue)

ARGUMENTS

Himage	Handle to an image.
index	The colourmap entry.
red	The red intensity (0..255).
green	The green intensity (0..255).
blue	The blue intensity (0..255).

DESCRIPTION

This function writes an individual colour into *Himage's* colourmap. It is generally used in conjunction with *TMG_cmap_generate*.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment sets three colours:

```
TMG_cmap_set_RGB_colour(hImage, 0, 255, 0, 0);
TMG_cmap_set_RGB_colour(hImage, 1, 255, 255, 255);
TMG_cmap_set_RGB_colour(hImage, 2, 0, 0, 255);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_cmap_set_colour, *TMG_cmap_set_type*, *TMG_cmap_generate*.

TMG_cmap_set_type

USAGE

Terr TMG_cmap_set_type(Thandle Himage, ui16 type)

ARGUMENTS

Himage	Handle to an image.
type	The type of colourmap/palette required. This can be one of the following: *TMG_VGA16, TMG_GRAYSCALE_RAMP, TMG_332_RGB, TMG_BLACK.*

DESCRIPTION

This function writes a colourmap into *Himage's* colourmap. The colourmap size is automatically set to 16 for *TMG_VGA16*, and 256 for *TMG_GRAYSCALE_RAMP, TMG_332_RGB* and *TMG_BLACK*. The colourmap types are as follows:

TMG_VGA16 — The palette is set to a size 16, consisting of a standard DOS or Windows 3.1 VGA palette. That is the entries 0 to 15 are as follows:

0	Black	8	Gray ("Light Black")
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Yellow	14	Light Yellow
7	White	15	Light White

TMG_332_RGB — A 256 entry colourmap, based on RGB 332. That is, it is a direct colourmap, such that each entry is represented by three bits of red, three bits of green and two bits of blue, with red at the most significant end of the byte.

TMG_GRAYSCALE_RAMP — A 256 entry colourmap, where each entry represents a grayscale from 0 to 255, with entry 0 having value 0, linearly increasing to entry 255 having value 255.

TMG_BLACK — This simply clears down all the entries to black and automatically sets the colourmap size to 256.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment sets the first 16 colourmap entries to the standard VGA set of colours:

```
TMG_cmap_set_type(hImage, TMG_VGA16);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_cmap_set_RGB_colour, *TMG_cmap_generate*.

TMG_display_box_fill [DOS]

USAGE

*Terr TMG_display_box_fill(Thandle Hdisplay, ui16 colour, i16 *roi)*

ARGUMENTS

Hdisplay	Handle to a display.	
colour	Colour of the box.	
roi	"ROI" array with four elements, with #defined element names:	
	ASL_ROI_X_START	Horizontal start position (0 = left of screen).
	ASL_ROI_Y_START	Vertical start position (0 = bottom of screen).
	ASL_ROI_X_LENGTH	Horizontal width of box.
	ASL_ROI_Y_LENGTH	Vertical height of box.

DESCRIPTION

This function draws a box on the screen in one of the following solid colours (the Flash Graphics definitions are used):

FG_BLACK	*FG_GRAY*
FG_BLUE	*FG_LIGHT_BLUE*
FG_GREEN	*FG_LIGHT_GREEN*
FG_CYAN	*FG_LIGHT_CYAN*
FG_RED	*FG_LIGHT_RED*
FG_MAGENTA	*FG_LIGHT_MAGENTA*
FG_YELLOW	*FG_BROWN*
FG_WHITE	*FG_LIGHT_WHITE*

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the example for *TMG_display_draw_text [DOS]*

BUGS / NOTES

This function is only supported under DOS and requires the Flash Graphics library. See the section on "Image Display Functions and Examples" at the start of this manual.

If this function is used in paletted display modes, the first 16 colours need to be reserved for the standard set of VGA colours (see *TMG_cmap_set_type*), otherwise the colour of the drawn boxes is unlikely to be as expected!

There are no known bugs.

SEE ALSO

TMG_display_clear [X Windows, DOS].

TMG_display_clear [X Windows, DOS]

USAGE

Terr TMG_display_clear(Thandle Hdisplay)

ARGUMENTS

Hdisplay Handle to a display.

DESCRIPTION

This function clears the display to black.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the example for *TMG_display_draw_text [DOS]*.

BUGS / NOTES

This function is only supported under DOS and X Windows. Under DOS the Flash Graphics library is required. See the section on "Image Display Functions and Examples" at the start of this manual.

Under DOS, if this function is used in paletted display modes, the first 16 colours need to be reserved for the standard set of VGA colours (see *TMG_cmap_set_type*), otherwise the display may not be cleared to black.

There are no known bugs.

SEE ALSO

TMG_display_box_fill [DOS].

TMG_display_cmap [DOS]

USAGE

Terr TMG_display_cmap(Thandle Hdisplay, Thandle Himage, ui16 TMG_action)

ARGUMENTS

Hdisplay	Handle to a display.
Himage	Handle to an image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function displays *Himage's* colourmap. Often it can be useful to see the colourmap during code development.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code sets up a colourmap, installs it in the display hardware and displays it:

```
/* Setup the colourmap - note we must reset the size to 256 as TMG_VGA16
 * will set it to a size of 16.
 */
TMG_cmap_set_type(hImage, TMG_332_RGB);
TMG_cmap_set_type(hImage, TMG_VGA16);
TMG_image_set_parameter(hImage, TMG_CMAP_SIZE, 256);

/* initialise the display and write the colourmap to the hardware */
TMG_display_init(hDisplay, TMG_800x600x8_RGB);   /* paletted mode */
TMG_display_cmap_install(hDisplay, hImage);

/* finally display the colourmap */
TMG_display_cmap(hDisplay, hImage, TMG_RUN);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

This function is only supported under DOS and requires the Flash Graphics library. See the section on "Image Display Functions and Examples" at the start of this manual.

There are no known bugs.

SEE ALSO

TMG_display_cmap_install, TMG_cmap_generate.

TMG_display_cmap_install [X Windows, DOS]

USAGE

Terr TMG_display_cmap_install(Thandle Hdisplay, Thandle Himage)

ARGUMENTS

Hdisplay	Handle to a display.
Himage	Handle to the image containing the desired colourmap.

DESCRIPTION

This function writes *Himage's* colourmap into the display's hardware colourmap (or palette). The display must be in a paletted mode for the function to work.

Under DOS, the colourmap will be immediately applied.

Under X Windows, the colourmap is applied when the window with which its associated with receives input focus (i.e. when the mouse pointer is moved into it).

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the example code for *TMG_display_cmap [DOS]*.

BUGS / NOTES

Under DOS, this function requires the Flash Graphics library. See the section on "Image Display Functions and Examples" at the start of this manual.

There are no known bugs.

SEE ALSO

TMG_display_init, *TMG_display_cmap [DOS]*, *TMG_cmap_generate*.

TMG_display_create

USAGE

Terr *TMG_display_create()*

ARGUMENTS

None.

DESCRIPTION

This function creates an internal display structure and returns a handle as a reference to it. The display structure referenced by the display handle stores information such as the screen dimensions, colour depth, associated window etc. A new display handle would be created for each display or window within a display. For multiple paint areas within one window, again a new display handle would be created and initialised.

RETURNS

On success a valid handle is returned in the lower 16 bits of the return value (the upper 16 bits will be 0). On failure an error code will be returned in the upper 16 bits as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code creates a display structure and gets a handle to it:

```
Thandle hDisplay;    /* Handle to display structure */

if ( ASL_is_err(hDisplay = TMG_display_create() )
   printf("Failed to create display handle");
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

This function is supported under all supported operating systems. Under DOS this function requires the Flash Graphics library and a VESA compatible graphics card.

There are no known bugs.

SEE ALSO

TMG_display_destroy, *TMG_display_init*, *TMG_display_set_ROI*, *TMG_display_set_parameter*, *TMG_display_set_flags*.

TMG_display_destroy

USAGE

T*handle TMG_display_destroy(Thandle Hdisplay)*

ARGUMENTS

Hdisplay Handle to a display structure or *TMG_ALL_HANDLES*.

DESCRIPTION

This function destroys a display structure, by freeing all the memory associated with that structure, and frees the display handle.

If the parameter *TMG_ALL_HANDLES* is used, all previously created display structures are destroyed and their associated handles freed.

TMG_image_destroy(*TMG_ALL_HANDLES*) will destroy all TMG display structures by calling *TMG_display_destroy* for all display handles. This is a convenient way of destroying everything with just one function call.

RETURNS

ASL_OK.

EXAMPLES

The following code destroys a previously created display structure:

```
Thandle  hDisplay;

hDisplay = TMG_create_display();
.
/* Destroy the display structure */
TMG_display_destroy(hDisplay);
```

In practice it is generally easier and more convenient to use *TMG_image_destroy*(*TMG_ALL_HANDLES*) to destroy all TMG structures on exit from the application.

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

This function is supported under all supported operating systems. Under DOS this function requires the Flash Graphics library and a VESA compatible graphics card.

There are no known bugs.

SEE ALSO

TMG_display_create, TMG_image_destroy.

TMG_display_direct_w31 [Windows 3.1]

USAGE

Terr TMG_display_direct_w31(Thandle Hbase, Thandle Himage, ui32 phys_addr, ui16 x_start, ui16 y_start, ui16 display_width, ui16 display_depth, ui16 operation)

ARGUMENTS

Hbase	Handle to a Bus Interface Board.
Thandle	Handle to the image to be displayed.
phys_addr	The physical address of the PCI display card's frame memory.
x_start	The X coordinate of the top left of the image's target position.
y_start	The Y coordinate of the top left of the image's target position.
display_width	The width of the display in pixels.
display_depth	The depth of the display in bytes.
operation	The pixel operation, one or more of *TMG_COPY, TMG_LAT_INV, TMG_VERT_INV, TMG_ROTATE180, TMG_CHECKER0, TMG_CHECKER1, TMG_MASKBL.* *TMG_RESET* is a special case described below.

DESCRIPTION

NOTE: This function is called internally by *TMG_display_image* when the flag *TMG_DISPLAY_DIRECT* is set and DCI is not present. It is documented here for the purposes of describing the raster operations defined by *operation*. In general this function should not be called directly.

This function provides the facility to write directly from PC host memory to certain PCI VGA cards at very high speed. It uses the same methodology as DCI (which if available should be used instead - as it supports full window clipping etc). The command bypasses the usual Windows GDI (graphical device interface) and the display card driver, so therefore it will always write the image on top (i.e. there can be no overlapping windows). It is up to the application to determine whereabouts to write the image. The following paragraphs explain each of the parameters:

Hbase is the usual handle to a Snapper Bus Interface Board (see the Bus Interface Library Programmer's Manual).

phys_address is the physical address of start of frame memory for the display card. Typically, PCI VGA cards are addressed near the top of the PC's address space. For example at 0xf0000000. Many PCI display cards use the first 16 Mbytes for register and control access and the next 16 Mbytes for frame memory. Therefore a typical start address of frame memory would be 0xf0800000. The example code fragment below shows how to extract the base address using two Snapper driver functions (which internally make calls to the PCI BIOS).

x_start and *y_start* are the coordinates of the top left of the target area for the image. The origin is at the top left of the display.

screen_width is the width of the display in pixels (e.g. 800, 1024 etc).

screen_depth is the depth in bytes of the display. Valid depths are 2 and 4 only. This represents 15 and 16 bit colour modes (2 bytes deep) and the 32 bit (24 bits per pixel, but 32 bit word aligned) colour mode respectively.

operation specifies the pixel operation to do whilst displaying the image. The options listed below can be ORed together to provide multiple operations at the same time (without any loss of speed). *TMG_RESET* is not a pixel operation like the other operations but a method of freeing the internal memory selectors used. See the description below for more details.

TMG_COPY	Simple copy operation.

TMG_LAT_INV	The image is laterally inverted.
TMG_VERT_INV	The image is vertically inverted.
TMG_ROTATE180	The image is rotated 180 degrees. This is actually equivalent to *(TMG_LAT_INV | TMG_VERT_INV)*.
TMG_CHECKER0	A checkerboard mask is applied, the block size of which is defined by *TMG_CHECKER0* | <size>, where <size> is a 16 bit unsigned integer. *TMG_CHECKER0* will update the top left square.
TMG_CHECKER1	Same as *TMG_CHECKER0* except it will now not update the top left square.
TMG_MASKBL	The same as *TMG_COPY* except the bottom left quadrant of the image will not get updated. This option is not available with the checkerboarding modes, but works with all other modes.
TMG_RESET	When the function *TMG_display_direct_w31 [Windows 3.1]* is called for the first time, memory selectors are used internally to reference the display memory. These selectors are freed and re-allocated whenever the size of the displayed image (or the display mode) is changed. To free these selectors the function should be called with *operation* set to *TMG_RESET*. Typically this would be done on program exit.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code shows how to read the physical address of the PCI graphics card and use the write direct function:

```
ui8 bus_num, dev_func_num;
ui32 phys_addr;

/* setup physical address */
PCI_DRV_find_class_code( (ui32) 0x030000, 0, &bus_num, &dev_func_num);
phys_addr = PCI_DRV_read_cs_ui32(bus_num, dev_func_num, (ui16) 0x10);
phys_addr += 0x800000;   /* display hardware dependent, but typical */

/* optimised capture and display loop */
SNP24_capture(Hsnp24, SNP24_START_AND_RETURN);

while (live == TRUE)
{
   while ( SNP24_is_capture_finished(Hsnp24) == FALSE )
      ;
   SNP24_read_video_data(Hsnp24, Himage, TMG_RUN);
   SNP24_capture(Hsnp24, SNP24_START_AND_RETURN);  /* start next capture */
   TMG_display_direct_w31(Hbase, Himage, phys_addr, 4, 50, 1024, 2,
       TMG_LAT_INV );
}
```

BUGS / NOTES

This is a low level function only applicable under Windows 3.1. It is called by *TMG_display_image* and writes directly to display hardware and therefore cannot be guaranteed to work on all PCI display cards. However it is known to work on several popular PCI graphics cards. It is strongly recommended that DCI is used in preference to this function under Windows 3.1. In fact an even better solution is to use Windows NT and DirectDraw.

SEE ALSO

PCI_DRV_DMA_to_display_win31 (described in the Snapper Bus Interface Board Library Programmer's Manual)

TMG_display_draw_text [DOS]

USAGE

*Terr TMG_display_draw_text(Thandle Hdisplay, char *text, ui16 x, ui16 y)*

ARGUMENTS

Hdisplay	Handle to a display.
text	Text string to write to the display.
x	X position of the text (origin = bottom left).
y	Y position of the text (origin = bottom left).

DESCRIPTION

This function writes text in the previously initialised font size (using *TMG_display_set_font [DOS]*) to the display at screen coordinates *x*, *y* (the origin is at the bottom left). The text is written as light white (*TMG_LIGHT_WHITE*).

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code sets a font type and draws the text in the top left of the screen, over a green box:

```
ui16 ScreenHeight;
i16 Roi[ASL_SIZE_2D_ROI];

TMG_display_clear(hDisplay);
ScreenHeight = (ui16) TMG_display_get_parameter(hDisplay, TMG_HEIGHT);

Roi[ASL_ROI_X_START] = 0;
Roi[ASL_ROI_Y_START] = ScreenHeight - 110;
Roi[ASL_ROI_X_LENGTH] = 274;
Roi[ASL_ROI_Y_LENGTH] = 100;
TMG_display_box_fill(hDisplay, FG_GREEN, Roi);
TMG_display_set_font(hDisplay, TMG_FG_15X19);
TMG_display_draw_text(hDisplay, "TMG - FG Demo", 10, ScreenHeight - 40);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

This function is only supported under DOS and requires the Flash Graphics library. See the section on "Image Display Functions and Examples" at the start of this manual.

If this function is used in paletted display modes, the first 16 colours need to be reserved for the standard set of VGA colours (see *TMG_cmap_set_type*), otherwise the colour of the drawn boxes is unlikely to be as expected!

Greater control of over the text (such as alternative colours) is provided by the Flash Graphics library and direct calls can be made from applications using the TMG library. For further details see the Flash Graphics manual.

SEE ALSO

TMG_display_set_font [DOS].

TMG_display_get_flags

USAGE

Tboolean TMG_display_get_flags(Thandle Hdisplay, ui16 type)

ARGUMENTS

Hdisplay	Handle to a display structure.
type	Flag type.

DESCRIPTION

This function returns the boolean state of the flag, selected by *type*, in *Hdisplay*.

The flag types are described in *TMG_display_set_flags*.

RETURNS

Returns *TRUE* or *FALSE*.

EXAMPLES

The following code determines if the display is colour:

```
if ( TMG_display_get_flags(hDisplay, TMG_DISPLAY_IS_COLOUR) == TRUE )
   printf("We have a colour display");
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_display_set_flags, *TMG_display_get_parameter*, *TMG_display_init*.

TMG_display_get_hWnd [Windows]

USAGE

HWND TMG_display_get_hWnd(Thandle Hdisplay)

ARGUMENTS

Hdisplay Handle to a display.

DESCRIPTION

This function returns *Hdisplay's* internal window handle. If *Hdisplay* is not valid, 0 is returned.

RETURNS

The handle to the window that *Hdisplay* references on success, otherwise 0.

EXAMPLES

The following code fragment shows a sub-routine used to repaint a window:

```
/* Repaint image */
void S24Repaint()
{
   RECT rc;

   ::GetClientRect(TMG_display_get_hWnd(S24.m_hDisplay), &rc);
   ::InvalidateRect(TMG_display_get_hWnd(S24.m_hDisplay), &rc, TRUE);
}
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_display_init, *TMG_display_set_hWnd [Windows]*.

TMG_display_get_paint_hDC [Windows]

USAGE

HDC TMG_display_get_paint_hDC(Thandle Hdisplay)

ARGUMENTS

Hdisplay Handle to a display.

DESCRIPTION

This function returns *Hdisplay's* internal handle to a device context. If *Hdisplay* is not valid, 0 is returned. Generally the device context will be 0 (i.e. no device context) because it is released on exit from *TMG_display_image*. The only time it isn't 0 is when the application has set it (before *TMG_display_image* or *TMG_display_print_DIB [Windows]* is called). This function is rarely needed and only documented for completeness.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

```
HDC hDC;

hDC = TMG_display_get_paint_hDC(hDisplay);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_display_print_DIB [Windows], *TMG_display_set_paint_hDC [Windows]*.

TMG_display_get_parameter

USAGE

ui32 TMG_display_get_parameter(Thandle Hdisplay, ui16 parameter)

ARGUMENTS

Hdisplay	Handle to a display structure.
parameter	Parameter type.

DESCRIPTION

This function returns the value of an internal parameter from *Hdisplay* selected by *parameter*. The parameter is always returned as a 32 unsigned integer although some of the parameters are stored as 16 unsigned integers internally in the display structure.

The parameter types are described in *TMG_image_set_parameter*.

RETURNS

The parameter selected by *parameter* as an unsigned 32 bit integer (*ui32*).

EXAMPLES

The following code fragment reads back the screen resolution:

```
ui32 width;
ui32 height;

width = TMG_display_get_parameter(hDisplay, TMG_WIDTH);
height = TMG_display_get_parameter(hDisplay, TMG_HEIGHT);
printf("Display size = %ld x %ld", (long) width, (long) height);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_display_set_parameter, *TMG_display_set_flags*.

TMG_display_get_ROI

USAGE

*Terr TMG_display_get_ROI(Thandle Hdisplay, i16 *roi)*

ARGUMENTS

Hdisplay	Handle to a display structure.	
roi	ROI array with four elements, with #defined element names:	
	ASL_ROI_X_START	Horizontal start position of ROI (0 = left of region).
	ASL_ROI_Y_START	Vertical start position of ROI (0 = top of region).
	ASL_ROI_X_LENGTH	Horizontal width of ROI.
	ASL_ROI_Y_LENGTH	Vertical height of ROI.

DESCRIPTION

This function copies the current ROI (Region of Interest) into the array *roi*.

The top left corner of the region is defined with the *ASL_ROI_X_START* and *ASL_ROI_Y_START* coordinates and the region size defined with the *ASL_ROI_X_LENGTH* and *ASL_ROI_Y_LENGTH* values.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code reads back the display ROI:

```
i16 Roi[ASL_SIZE_2D_ROI];   /* a 4 element array */
i16 ROIWidth;
i16 ROIHeight;

TMG_display_get_ROI(hDisplay, Roi);
ROIWidth = Roi[ASL_ROI_X_LENGTH];
ROIHeight = Roi[ASL_ROI_Y_LENGTH];
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_display_get_ROI, *TMG_display_image*.

TMG_display_image

USAGE

Terr TMG_display_image(Thandle Hdisplay, Thandle Himage, ui16 TMG_action)

ARGUMENTS

Hdisplay	Handle to a display.
Himage	Handle to an image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function displays the image *Himage,* to the display (or window) referenced by *Hdisplay.*

DOS

Under DOS, strip processing is fully supported and the function can be used in a strip processing loop in the same way as other TMG functions, although with modern PCs there is almost always enough memory not to worry about strip processing.

The image pixel format must always be the same as that of the display. For example if the display is initialised as *TMG_800x600x16* (i.e. 16 bit colour), then the image to be displayed must be type *TMG_RGB16*.

WINDOWS

Under Windows NT, 95 and 3.1 the TMG display API is the same, but with Windows 3.1 having an extra specialist function (*TMG_display_direct_w31 [Windows 3.1]*). For all these operating systems, the image must be displayed in one strip - i.e. the whole image at a time.

There are three methods by which images can be displayed under Windows - these are DIB, DDB and DirectDraw (or DCI as it's called on Windows 3.1). The algorithm to decide which method to use is as follows: If the *TMG_IS_DIB* flag is set, then the image will be displayed as a DIB. If it is not set and the *TMG_DISPLAY_DIRECT* flag is set then DirectDraw will be used (if available, or else a proprietary direct display method), otherwise DDB will be used. Each method is described in detail below:

Firstly the image can be displayed as a (24 bit) DIB. A DIB is a device independent bitmap which will get displayed under any graphics mode by the graphics driver (to the best of its ability). To generate a DIB image, *TMG_image_convert* is be used with the output format set to *TMG_BGR24* and the *TMG_IS_DIB* flag set. This method is usually not the fastest, but is generally very reliable - i.e. it should always work. The quality of the rendered image will vary according to the screen mode and graphics card driver. It is strongly recommended to use 32k colours or more. To achieve full 8 bit dynamic range on each primary colour (and/or gray levels), a 32 bit display mode will need to be used (i.e. 16.7 million colours).

The second method is to convert the image into the required pixel format prior to display, so that the graphics card driver does not need to convert the pixel format itself. This method is referred to as the DDB method (Device Dependent Bitmap). This time, *TMG_image_convert* is used to convert the pixel format of the image to that of the display (or it may be acquired from video acquisition hardware already in that pixel format). The display's pixel format can be determined using the function *TMG_display_get_parameter*. This method is generally faster than the DIB method but has been known to fail on some graphics cards - usually because graphics card driver does not support "BitBlt of greater than 64k bytes" at a time. (This DDB method only really applies to Windows 3.1 – for Windows NT and 95, DirectDraw, as described below, is the preferred method.)

The third method, and certainly for Windows NT and 95 is using a direct access (and hence fast) method – DirectDraw for Windows NT/95, or DCI (Display Control Interface). To use the direct method, the flag

TMG_DISPLAY_DIRECT is set (see *TMG_display_set_flags*). DirectDraw (or DCI) requires a driver from the graphics card vendor that supports this direct API (and for Windows 3.1, the Video for Windows runtime libraries). (Video for Windows runtime should be available from your graphics card vendor and is often provided with the vendor's Windows 3.1 drivers.)

Display under DirectDraw/DCI can also make use of the *TMG_HALF_ASPECT* flag and *TMG_FIELD_ID* parameter to re-interlace fields whilst displaying to re-construct full size correct aspect ratio images at high speed.

X WINDOWS

Under X Windows the image must always be displayed in one strip - i.e. the whole image at a time.

The image pixel format must always be the same as that of the display - that is either paletted or 32 bit (i.e. *TMG_PALETTED* or *TMG_XBGR32*). For detailed examples, see the application examples supplied with the Solaris or LynxOS SDKs.

MacOS

Under MacOS the image must always be displayed in one strip - i.e. the whole image at a time. The action parameter is ignored.

The image must be supplied in one of the following formats*: TMG_Y8, TMG_Y16, TMG_RGB15* or *TMG_XBGR32*.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the examples in the "Image Display Functions and Examples" section at the start of this manual.

BUGS / NOTES

Under DOS this function requires the Flash Graphics library. See the section on "Image Display Functions and Examples" at the start of this manual.

Under all operating systems, apart from DOS, the image must be displayed in one strip - i.e. the whole image at a time.

There are no known bugs.

SEE ALSO

TMG_display_init, TMG_display_direct_w31 [Windows 3.1].

TMG_display_init

USAGE

Terr TMG_display_init(Thandle Hdisplay, HWND hWnd) [Windows]

Terr TMG_display_init(Thandle Hdisplay, ui16 mode) [DOS, X Windows]

Terr TMG_display_init(Thandle Hdisplay, PixMapHandle hPmap) [MAC]

ARGUMENTS

Hdisplay	Handle to a display.
hWnd	Handle to a window [For Windows, MAC].
mode	For DOS this is set to the graphics mode - see below. For X Windows this parameter must be set to *TMG_X_WINDOWS*.

DESCRIPTION

WINDOWS

This function initialises the internal structure referenced by *Hdisplay* using the handle to a valid window, *hWnd*. The display organisation (i.e. depth, colour format etc) is derived and stored within the structure. TMG_display_get_parameter can be used to examine these fields. If DirectDraw (or DCI - Display Control Interface for Windows 3.1) is present, it will be initialised and its presence can be tested using *TMG_display_get_parameter* with the parameter *TMG_DISPLAY_DIRECT_CAPS*. If the return value is non-zero, DirectDraw (or DCI) is present.

The following display types are supported at all resolutions:

- 32K colours with colour organisation RRRRRGGGGGBBBBB (*TMG_RGB15*).
- 65K colours with colour organisation RRRRRGGGGGGBBBBB (*TMG_RGB16*) (Preferred mode).
- 16.7 million colours with colour organisation BGR24 (*TMG_BGR24*).
- 16.7 million colours with colour organisation BGRX32 (*TMG_BGRX32*) (Preferred mode).

The 16.7 million colours mode using colour organisation BGR24 (sometimes known as "packed pixel") has limited support for fast display update. This is because each pixel is not aligned to 32 bits, thus certain raster operations are not so efficient. It is strongly recommended that the alternative modes are used.

The 32K and 64K colour modes give good performance and good colour quality, although some degradation will be noticed on gradually changing tones. BGRX32 provides the ultimate quality and is only slightly slower than the 16 bit modes.

When displaying to multiple windows, a display handle should be created and initialised for each window.

DOS

This function initialises the graphics card to one of the following screen modes:

TMG_DOS_PROMPT	This puts the display into the usual DOS text mode. It is used to switch the display back to text mode from one of the graphics modes described below.
TMG_640x480x8_GRAYSCALE	This is an 8 bit grayscale paletted mode, with a screen resolution of 640 x 480, where the palette is written with grayscales from 0 to 255. For example, writing a pixel value of 255 results in white.
TMG_640x480x8_RGB	This is an 8 bit colour paletted mode, with a screen resolution of 640 x 480, where the palette is written to contain a RGB 3:3:2 direct colourmap. For example, writing a pixel value of 0xFF results in

	white; 0xE0 results in a fully saturated red; 0x03 in blue etc.
TMG_640x480x16	This is a 16 bit colour mode, with a screen resolution of 640 x 480, where the colours are represented by RGB 5:6:5. For example a pixel value of 0xF800 results in a saturated red; and 0x001F in a saturated blue etc.
TMG_640x480x24	This is a 24 bit colour mode, with a screen resolution of 640 x 480, where the colours are represented by RGB 8:8:8.
TMG_800x600x8_GRAYSCALE	This is an 8 bit grayscale paletted mode, with a screen resolution of 800 x 600. The palette is the same as *TMG_640x480x8_GRAYSCALE*.
TMG_800x600x8_RGB	This is an 8 bit colour paletted mode, with a screen resolution of 800 x 480. The palette is the same as *TMG_640x480x8_RGB*.
TMG_800x600x15	This is a 15 bit colour mode, with a screen resolution of 800 x 600, where the colours are represented by RGB 5:5:5.
TMG_800x600x16	This is a 16 bit colour mode, with a screen resolution of 800 x 600. Pixel values are mapped into colours in the same way as *TMG_640x480x16*.
TMG_800x600x24	This is a 24 bit colour mode, with a screen resolution of 800 x 600, where the colours are represented by RGB 8:8:8.
TMG_1024x768x8_RGB	This is an 8 bit colour paletted mode, with a screen resolution of 1024 x 768. The palette is the same as *TMG_640x480x8_RGB*.
TMG_1280x1024x8_RGB	This is an 8 bit colour paletted mode, with a screen resolution of 1280 x 1024. The palette is the same as *TMG_640x480x8*_RGB.

X WINDOWS

This function initialises the internal structure referenced by *Hdisplay*. The parameter *mode*, must be set to *TMG_X_WINDOWS*. The X display is opened using the DISPLAY environment variable (i.e. *XOpenDisplay(NULL)* is called). Various parameters are set internally in the display structure including the width, height, depth and if applicable, number of free colours, of the display. These may be examined using the function *TMG_display_get_parameter*.

Only paletted and 24 bit displays are supported. However this covers the majority of SPARCstation based Solaris 2 environments. The pixel formats in detail are:

- Paletted: 8 bit writeable palette (or colourmap) (*TMG_PALETTED*).
- 16.7 million colours with colour organisation XBGR32 (*TMG_XBGR32*). This mode is basically 24 bits per pixel, but with each pixel aligned to a 32 bit word boundary. The display depth and pixel format returned by *TMG_display_get_parameter* are 24 and *TMG_RGB24* respectively, rather than the expected 32 and *TMG_XBGR32*. This is because internal X functions, such as *XCreateImage* expect a depth of 24 even though the display is really 32 bits deep.

The 24 bit display mode is much better quality and is often faster in terms of overall performance (as no colourmapping operations are needed).

When displaying to multiple windows, a display handle should be created and initialised for each window.

MacOS

This function initialises the internal structure referenced by *Hdisplay* using the handle to a valid window pixmap, *hPmap,* attached to the window area you want to display in. The display organisation (i.e. depth, colour format etc) is derived and stored within the structure. *TMG_display_get_parameter* can be used to examine these fields.

The default settings are for image stretching to fit the window size. This can be over-ridden by a subsequent call to *TMG_display_set_parameter* with the argument of *TMG_STRETCH* set to zero.

The MAC display modes supported are 256-level greyscale, 64K greyscale, "thousands of colours" and "millions of colours":

- 256 greyscale (*TMG_Y8*).
- 64K greyscale (*TMG_Y16*).
- 32K colours with colour organisation RRRRRGGGGGBBBBB (*TMG_RGB15*).
- 16.7 million colours with colour organisation BGRX32 (*TMG_BGR24*) (Preferred mode).

The 32K colour mode gives good performance and good colour quality, although some degradation will be noticed on gradually changing tones. BGRX32 provides the ultimate quality.

When displaying to multiple windows, a display handle should be created and initialized for each window.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

WINDOWS

The following initialises the TMG display structure in the view class the first time that *OnDraw* is called:

```
void CS24View::OnDraw(CDC* pDC)
{
   static BOOL bFirstTime = TRUE;

   if (bFirstTime == TRUE)
   {
      TMG_display_init(S24.m_hDisplay, GetSafeHwnd());
      TRACE("DirectDraw Caps = %08lx\n",
       TMG_display_get_parameter(S24.m_hDisplay, TMG_DISPLAY_DIRECT_CAPS) );
      bFirstTime = FALSE;
   }
.
}
```

See also the extended examples in the "Sample Applications" section.

DOS

The following initialises the display to a resolution of 800 by 600 in 65K colours:

```
Thandle hDisplay;

hDisplay = TMG_display_create();
TMG_display_init(hDisplay, TMG_800x600x16);
getch();  /* Press any key to exit */
TMG_display_init(hDisplay, TMG_DOS_PROMPT);
exit(0);
```

See also the extended examples in the "Sample Applications" section.

X WINDOWS

See the section "Image Display Functions and Examples".

MacOS

The following code fragment obtains a PixMap handle from a valid grafport window and initialises the display with it as the destination:

```
PixMapWindow hpmPlayThru;

hDisplay = TMG_display_create();

/* Obtain a valid PixMapWindow to display with */
hpmPlayThru = GetWindowPort(pwWindow)->portPixMap;

/* Initialise display with PixMapWindow */
TMG_display_init(hDisplay, hpmPlayThru);
```

See also the section "Image Display Functions and Examples".

BUGS / NOTES

Under Windows, display modes of 256 colours or less are not supported by the TMG library. However the Windows API may be programmed directly if 256 colours or less have to be used.

Under DOS this function requires the Flash Graphics library and a graphics card capable of VESA display modes. See the section on "Image Display Functions and Examples" at the start of this manual.

In the Flash Graphics library there are more supported graphics mode which are readily programmed. Direct calls can be made from applications using the TMG library direct to the Flash Graphics library. For further details see the Flash Graphics manual.

There are no known bugs.

SEE ALSO

TMG_display_create.

TMG_display_print_DIB [Windows]

USAGE

Terr TMG_display_print_DIB(Thandle Hprinter, Thandle Himage, i16 percentage, ui16 TMG_action)

ARGUMENTS

Hprinter	Handle to a printer (this is identical to a display handle).
Himage	Handle to an image.
percentage	Percentage scaling factor (100% = full print area).
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function is a convenient way of printing DIB images under Windows. *Hprinter* is created in the same way as a display handle, but the device context and internal device dimensions are modified (see example below).

percentage is used to control the size of the printed image. It represents the percentage of the maximum print area whilst still preserving the image's aspect ratio.

The image must always be displayed in one strip - i.e. the whole image at a time.

The function name includes "*_display*" in it because it is a sub-set of the display function group.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following example shows how the device context is tested upon and then the image is either printed or displayed as appropriate:

```
void CS24View::OnDraw(CDC* pDC)
{
.
   if ( TMG_image_get_ptr(S24.m_hDIBImage, TMG_IMAGE_DATA) != NULL )
      if (pDC->IsPrinting()) {
         // Set up the printer dimensions, then print
         TMG_display_set_parameter(S24.m_hPrinter, TMG_WIDTH,
                                   (ui16) pDC->GetDeviceCaps(HORZRES));
         TMG_display_set_parameter(S24.m_hPrinter, TMG_HEIGHT,
                                   (ui16) pDC->GetDeviceCaps(VERTRES));
         TMG_display_set_paint_hDC(S24.m_hPrinter, pDC->GetSafeHdc());
         TMG_display_print_DIB(S24.m_hPrinter, S24.m_hDIBImage, 85, TMG_RUN);
         TMG_display_set_paint_hDC(S24.m_hPrinter, 0);
      }
      else
      { /* display */
         TMG_display_set_paint_hDC(S24.m_hDisplay, pDC->GetSafeHdc());
         TMG_display_image(S24.m_hDisplay, S24.m_hDIBImage, TMG_RUN);
         TMG_display_set_paint_hDC(S24.m_hDisplay, 0);  /* Set back */
      }
   }
}
```

S24.m_hPrinter is created as a display device in the usual way (a printer is a form of display):

```
S24.m_hPrinter = TMG_display_create();
```

See also the examples in the "Image Display Functions and Examples" section at the start of this manual.

BUGS / NOTES

The image must always be printed in one strip - i.e. the whole image at a time.

There are no known bugs.

SEE ALSO

TMG_display_image, *TMG_display_set_paint_hDC [Windows]*.

TMG_display_set_flags

USAGE

Terr TMG_display_set_flags(Thandle Hdisplay, ui16 type, Tboolean state)

ARGUMENTS

Hdisplay	Handle to a display structure or *TMG_ALL_HANDLES*.
type	Flag type.
state	Either *TRUE* or *FALSE*.

DESCRIPTION

This function sets flags in *Hdisplay* which are then subsequently tested on by various TMG display functions (in particular *TMG_display_image*).

The flags are as follows:

TMG_DISPLAY_IS_COLOUR	This flag is set automatically as appropriate by *TMG_display_init* and should only be read by a user application. It indicates whether the display is colour or not.
TMG_DISPLAY_DIRECT	This flag, set by a user application, indicates that *TMG_display_image* should use the best method available to it that displays directly to the display surface (i.e. directly to screen memory). For example, under Windows, DirectDraw will be used (DCI for Windows 3.1).
TMG_STRETCH	This flag, set by a user application, indicates that the image should be stretched (or scaled) to fit the display window. This is not supported under all display environments.
TMG_KEEP_ASPECT	This flag, when used in conjunction with *TMG_STRETCH*, indicates that the image should be stretched (or scaled) to fit the display window, but that the original aspect ratio should be preserved. This is not supported under all display environments.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code sets the display window referenced by *hDisplay* to display images as large as possible within the window, but still preserving the aspect ratio.

```
TMG_display_set_flags(hDisplay, TMG_STRETCH | TMG_KEEP_ASPECT);
```

BUGS / NOTES

The *TMG_STRETCH* and *TMG_KEEP_ASPECT* flags are only supported under Windows.

The *TMG_STRETCH* flag is the default setting under MacOS. Turn it off to improve the image display rate for images where image size and display area size are not equal.

SEE ALSO

TMG_display_get_flags, *TMG_display_set_parameter*, *TMG_display_image*.

TMG_display_set_font [DOS]

USAGE

Terr TMG_display_set_font(Thandle Hdisplay, ui16 font)

ARGUMENTS

Hdisplay	Handle to a display.
font	Font type - one of the following: *TMG_FG_6X7* *TMG_FG_8X8* *TMG_FG_8X14* *TMG_FG_8X16* *TMG_FG_15X19*

DESCRIPTION

This function sets the size of the font in preparation for using *TMG_display_draw_text [DOS]*. For any complex font display application, is it recommended that the Flash Graphics library is called directly.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the example for *TMG_display_draw_text [DOS]*.

BUGS / NOTES

This function requires the Flash Graphics library and is only supported under DOS with a VESA compatible graphics driver. Please refer to the Flash Graphics manual for more information.

Greater control of over the fonts (such as custom fonts) is provided by the Flash Graphics library and direct calls can be made from applications using the TMG library. For further details see the Flash Graphics manual.

There are no known bugs.

SEE ALSO

TMG_display_draw_text [DOS].

TMG_display_set_hWnd [Windows]

USAGE

Terr TMG_display_set_hWnd(Thandle Hdisplay, HWND hWnd)

ARGUMENTS

Hdisplay	Handle to a display.
hWnd	Handle to a window.

DESCRIPTION

This function sets *Hdisplay's* internal window handle, that is subsequently used by *TMG_display_image* to know into which window to display images.

This function is rarely needed, because *TMG_display_init* takes the appropriate window handle as parameter.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

```
HWND hWnd1;
.
TMG_display_set_hWnd(hDisplay, hWnd1);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_display_init, *TMG_display_set_hWnd [Windows]*.

TMG_display_set_mask [MAC]

USAGE

Terr TMG_display_set_mask(Thandle Hdisplay, RgnHandle hRegion)

ARGUMENTS

Hdisplay	Handle to a display.
hRegion	Region Handle containing a single bit mask.

DESCRIPTION

This function allows a pass-through mask to be set for the display. Only at the pixels where the mask is set to 1 will the video be displayed to the Mac Display. This offers users the ability to set overlays or non-rectangular windows through which the video can be displayed.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

SEE ALSO

TMG_display_create.

TMG_display_set_paint_hDC [Windows]

USAGE

Terr TMG_display_set_paint_hDC(Thandle Hdisplay, HDC hDC)

ARGUMENTS

Hdisplay	Handle to a display.
hDC	Handle to a device context.

DESCRIPTION

This function sets the device context in preparation for using *TMG_display_image* or *TMG_display_print_DIB [Windows]*.

The device context is normally derived internally to *TMG_display_image* (and released on exit), but occasionally it is necessary to set a specific device context - for example, when a pointer to a specific device context is passed into the *OnDraw* function in the *view* class (using Microsoft Visual C++). When setting the device context, it is important to "release it" from the TMG library when finished displaying (or printing) by calling *TMG_display_set_paint_hDC* with an *hDC* of *0*.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the example for *TMG_display_print_DIB [Windows]*.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_display_print_DIB [Windows], *TMG_display_get_paint_hDC [Windows]*.

TMG_display_set_parameter

USAGE

Terr TMG_display_set_parameter(Thandle Hdisplay, ui16 parameter, ui32 value)

ARGUMENTS

Hdisplay	Handle to a display structure.
parameter	Parameter type.
value	The actual value passed in as a 32 bit unsigned integer (although some parameters will only be 16 bit unsigned integers).

DESCRIPTION

This function sets internal parameters in the display structure referenced by *Hdisplay*.

The parameters are as follows:

TMG_PIXEL_FORMAT	This parameter is set automatically by *TMG_display_init* and should only be read by a user application. It represents the pixel format of the display and has the same values as the image pixel formats, for example *TMG_RGB16*. Pixel formats are type *ui16*. For a complete list of pixel formats see the section "Pixel Formats and Return Types" at the start of this manual.
TMG_WIDTH	This parameter is set automatically by *TMG_display_init* and should only be read by a user application. It represents the width in pixels of the display. Also known as horizontal resolution. This parameter is type *ui16*.
TMG_HEIGHT	This parameter is set automatically by *TMG_display_init* and should only be read by a user application. It represents the height in pixels of the display. Also known as vertical resolution. This parameter is type *ui16*.
TMG_DEPTH	This parameter is set automatically by *TMG_display_init* and should only be read by a user application. It represents the depth in bits of the display - in other words the number of bits per pixel. For example the pixel format *TMG_RGB15* has depth 15. This parameter is type *ui16*.
TMG_FRAME_MEMORY_OFFSET	This parameter is set from a user application and represents the memory offset from a graphics card's base address to the actual start of image memory. This is only supported under Windows 3.1 with the *TMG_DISPLAY_DIRECT* flag and with no DCI present. Often the memory offset is 16Mbytes (0x800000L). This parameter is type *ui32*.
TMG_RASTER_OP	This parameter is set from a user application and represents the raster operation that may be performed at the same time as displaying the image. This is only supported under Windows 3.1 with a suitable PCI graphics card, the *TMG_DISPLAY_DIRECT* flag set and with no DCI present. Raster operations include lateral/vertical inversions etc - see *TMG_display_direct_w31 [Windows 3.1]*. This parameter is type *ui32*.

TMG_DISPLAY_DIRECT_CAPS This parameter is set automatically by *TMG_display_init* and should only be read by a user application. It represents the DirectDraw capability flags as defined in the DirectDraw specification (DCI for Windows 3.1). This is only supported under Windows. This parameter is type *ui32*.

Note that the width, height and depth represent the dimensions of the overall display size and not of any particular (child) window. Under X Windows, the dimensions represent the root window.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how to determine the current graphics display depth:

```
TMG_display_init(hDisplay, GetSafeHwnd());
TRACE("Screen Depth = %08lx\n", TMG_display_get_parameter(hDisplay,
       TMG_DEPTH));
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_display_get_parameter, *TMG_display_set_flags*.

TMG_display_set_ROI

USAGE

*Terr TMG_display_set_ROI(Thandle Hdisplay, Tparam mode, i16 *roi)*

ARGUMENTS

Hdisplay	Handle to a display structure.	
mode	Required mode - *TMG_ROI_INIT* or *TMG_ROI_SET*.	
roi	ROI array with four elements, with #defined element names:	
	ASL_ROI_X_START	Horizontal start position of ROI (0 = left of region).
	ASL_ROI_Y_START	Vertical start position of ROI (0 = top of region).
	ASL_ROI_X_LENGTH	Horizontal width of ROI.
	ASL_ROI_Y_LENGTH	Vertical height of ROI.

DESCRIPTION

This function defines a ROI (Region of Interest) for the display referenced by *Hdisplay*. A region of interest represents an area within the total display surface or window referenced by *Hdisplay*. Note that there can be multiple display handles referencing the same display, thus making it easy to have multiple ROIs without having to re-call the function *TMG_display_set_ROI*.

The top left corner of the region is defined with the *ASL_ROI_X_START* and *ASL_ROI_Y_START* coordinates and the region size defined with the *ASL_ROI_X_LENGTH* and *ASL_ROI_Y_LENGTH* values.

MODE PARAMETER LIST

TMG_ROI_INIT	This is the default option set by *TMG_display_init*. In this mode the ROI is set to the whole window or display area. Thus *ASL_ROI_X_START* and *ASL_ROI_Y_START* are set to zero, and *ASL_ROI_X_LENGTH* and *ASL_ROI_Y_LENGTH* are set to *TMG_AUTO_WIDTH* and *TMG_AUTO_HEIGHT* respectively. The "*AUTO*" means that the full width and height of the image will be displayed subject to the clipping restraints of the window.
TMG_ROI_SET	The *roi* passed in is set.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

A typical application is to display a sub-sampled image in the foreground - say in the bottom left of the display:

```
i16 Roi[ASL_SIZE_2D_ROI];   /* a 4 element array */

/* The display size is 800 x 600 - we will set a ROI of 128 x 128 */
Roi[ASL_ROI_X_START] = 462;
Roi[ASL_ROI_Y_START] = 10;
Roi[ASL_ROI_X_LENGTH] = 128;
Roi[ASL_ROI_Y_LENGTH] = 128;
TMG_display_set_ROI(hDisplay, TMG_ROI_SET, Roi);
```

This next example shows how the whole image can be displayed, but starting at a different origin:

```
i16 Roi[ASL_SIZE_2D_ROI];   /* a 4 element array */
```

```
Roi[ASL_ROI_X_START] = 462;
Roi[ASL_ROI_Y_START] = 10;
Roi[ASL_ROI_X_LENGTH] = TMG_AUTO_WIDTH;
Roi[ASL_ROI_Y_LENGTH] = TMG_AUTO_HEIGHT;
TMG_display_set_ROI(hDisplay, TMG_ROI_SET, Roi);
```

BUGS / NOTES

Under DOS, initialisation using *TMG_ROI_INIT* actually sets the *ASL_ROI_X_START* and *ASL_ROI_Y_START* parameters to *TMG_AUTO_CENTRE,* which automatically centres the image on the display. This is only supported under DOS.

Also the origin is the bottom left under DOS and not the top left as it is in other operating environments.

SEE ALSO

TMG_display_get_ROI, *TMG_display_image*.

TMG_display_set_Xid [X Windows]

USAGE

Terr TMG_display_set_Xid(Thandle Hdisplay, ui32 type, Window xid)

ARGUMENTS

Hdisplay	Handle to a display.
type	X Window type.
xid	The X Window ID.

DESCRIPTION

This function sets various X Window IDs in the structure referenced by *Hdisplay*.

The X Window types are as follows:

TMG_XID_FRAME	This is the X Window ID of the applications root frame.
TMG_XID_CANVAS	This is the X Window ID of the drawable canvas associated with the frame.
TMG_XID_WINDOW	This is the X Window ID of the display window.

The X Window IDs for the frame and canvas need to be set before *TMG_display_init* is called, so that the correct colourmap association can be set up.

The IDs *TMG_XID_CANVAS* and *TMG_XID_FRAME* are only needed when using Solaris 2 and Sun's OpenWindows and "Devguide" toolkit (no longer actively supported bu Sun).

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the section "Image Display Functions and Examples".

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_display_init.

TMG_image_calc_total_strips

USAGE

Terr TMG_image_calc_total_strips(Thandle Himage)

ARGUMENTS

Himage Handle to an image.

DESCRIPTION

Returns the total number of strips in the image, based on the internal image parameters *height* and *lines_this_strip*. The image *height* is divided by the number of *lines_this_strip* and rounded up to the next whole number if necessary.

For example if an image has a *height* of 128 lines and *lines_this_strip* is set to 8, the number of strips returned would be 16. If the height was 127, the number of strips would still be 16, but the TMG functions processing the image would automatically change the internal parameter *lines_this_strip* to 7 for the last strip. Note therefore that this would need to be reset if repeatedly using a strip processing loop.

TMG_image_set_parameter is used to set the *lines_this_strip* parameter. The *height* of the image would normally be automatically calculated from loading the image.

RETURNS

The total number of strips in the lower 16 bits of the 32 bit return value, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

This example reads in a TIFF file in strips saves it as its mirror image. Note that a "dummy" read is needed first to determine the height of the image so that the number of strips can be calculated:

```
TMG_image_set_infilename(hImage, "sky.tif");
/* The outfilename parameter gets transferred to hOutImage */
TMG_image_set_outfilename(hImage, "sky_mirror.tif");

/* open the file to read the image height */
TMG_image_set_parameter(hImage, TMG_HEIGHT, 0);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_image_read(hImage, TMG_NULL, TMG_RESET);

/* Now set up the strip processing loop and proceed, 8 lines at time */
TMG_image_set_parameter(hImage, TMG_LINES_THIS_STRIP, 8);
TotalStrips = TMG_image_calc_total_strips(hImage);

for (Strip = 0; Strip < TotalStrips; Strip++) {
   TMG_image_read(hImage, TMG_NULL, TMG_RUN);
   TMG_IP_mirror_image(hImage, hOutImage, TMG_RUN);
   TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
}
TMG_image_set_parameter(hImage, TMG_LINES_THIS_STRIP, 8); /* just in case */
```

Note that if the image height of "sky.tif" did not divide exactly by 8, the *TMG_LINES_THIS_STRIP* parameter of *hImage* will no longer contain 8 and may in applications different to this example need setting back to 8 - hence the "just in case" comment at the end.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_set_parameter.

TMG_image_check

USAGE

Terr TMG_image_check(Thandle Himage)

ARGUMENTS

Himage Handle to an image.

DESCRIPTION

This function performs a simple check on the image - checking that the image format and depth are compatible. For example a *TMG_RGB16* image should have a depth of 16. It also calculates *Himage's* internal parameter *bytes_per_line*. The *bytes_per_line* parameter can be read using *TMG_image_get_parameter* with *TMG_BYTES_PER_LINE*. This can be useful for accessing the image data directly from an application.

This function is rarely needed in an application. but it is sometimes useful as a confidence check.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

No known bugs.

SEE ALSO

TMG_image_get_parameter.

TMG_image_conv_LUT_destroy

USAGE

Terr TMG_image_conv_LUT_destroy(ui16 LUT_type)

ARGUMENTS

LUT_type A look up table type as defined below used by *TMG_image_convert*.

DESCRIPTION

This function destroys a conversion LUT selected by one of the parameters from the list below.

LUT_type can be one of the following:

TMG_Y8_TO_PALETTED_LUT	Used to convert an 8 bit grayscale image to an 8 bit paletted image.
TMG_Y16_TO_PALETTED_LUT	Used to convert a 16 bit grayscale image to an 8 bit paletted image.
TMG_RGB16_TO_PALETTED_LUT	Used to convert a 16 bit RGB image (*TMG_RGB16*) to an 8 bit paletted image.
TMG_YUV422_TO_PALETTED_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to an 8 bit paletted image.
TMG_YUV422_TO_RGB15_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to a 15 bit RGB image (*TMG_RGB15*).
TMG_YUV422_TO_RGB16_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to a 16 bit RGB image (*TMG_RGB16*).

The memory used by the LUTs is freed when the LUT is destroyed.

Note that the actual LUTs are internal globals within the TMG library and not related to any other structure.

TMG_image_destroy(TMG_ALL_HANDLES) will automatically destroy all TMG structures including these LUTs.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment destroys a previously created conversion LUT:

```
/* destroy the conversion LUT */
TMG_image_conv_LUT_destroy(TMG_YUV422_TO_RGB16_LUT);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_conv_LUT_generate, *TMG_image_destroy*.

TMG_image_conv_LUT_generate

USAGE

Terr TMG_image_conv_LUT_generate(Thandle Himage, ui16 LUT_type)

ARGUMENTS

Himage	Handle to an image.
LUT_type	A look up table type as defined below and used by *TMG_image_convert*.

DESCRIPTION

This function generates the appropriate LUT for use by the function *TMG_image_convert* when it is used with the flag *TMG_USE_LUT*. If the relevant LUT has not been generated, the first call to *TMG_image_convert* will automatically generate the LUT. The function *TMG_image_conv_LUT_generate* is sometimes useful to generate the LUT in advance of actually performing the conversion.

LUT_type can be one of the following:

TMG_Y8_TO_PALETTED_LUT	Used to convert an 8 bit grayscale image to an 8 bit paletted image.
TMG_Y16_TO_PALETTED_LUT	Used to convert a 16 bit grayscale image to an 8 bit paletted image.
TMG_RGB16_TO_PALETTED_LUT	Used to convert a 16 bit RGB image (*TMG_RGB16*) to an 8 bit paletted image.
TMG_YUV422_TO_PALETTED_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to an 8 bit paletted image.
TMG_YUV422_TO_RGB15_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to a 15 bit RGB image (*TMG_RGB15*).
TMG_YUV422_TO_RGB16_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to a 16 bit RGB image (*TMG_RGB16*).

The memory used by the LUTs is dynamically allocated when the LUT is generated.

Under Windows 3.1, the 15 and 16 bit YUV 4:2:2 LUTs are each 64K in size. Under true 32 bit memory models (all other operating systems), the 15 and 16 bit output LUTs are 1 Mbyte in size, resulting in better quality colour conversion. The YUV 4:2:2 to paletted LUT is always 32K in size under all memory models.

When generating a LUT for conversion to a paletted image, *Himage* must contain the desired palette. For non-paletted conversion LUTs, *Himage* is not used (but it still needs to be a valid image). See *TMG_cmap_generate* for details on creating palettes (or colourmaps)

The LUTs used by the conversion function, *TMG_image_convert*, are not related to the "*TMG_LUT*" suite of functions. The LUTs are internal globals within the TMG library and not related to any other structure. Note also that all of the above LUTs can be used in parallel - they are all independent of each other.

TMG_image_conv_LUT_destroy can be used to destroy any specific conversion LUT(s) that have be generated (i.e. to free the allocated memory - or to force their regeneration).
TMG_image_destroy(TMG_ALL_HANDLES) will automatically destroy all TMG structures including these LUTs.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment generates the conversion LUT for converting YUV 4:2:2 data to the 16 bit RGB format (*TMG_RGB16*) in advance of using the function *TMG_image_convert* (for example when the application is first started):

```
/* Generate the LUT */
if (ASL_is_err(TMG_image_conv_LUT_generate(Himage, TMG_YUV422_TO_RGB16_LUT)))
   printf("Failed to generate LUT");
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_convert, *TMG_image_conv_LUT_destroy*, *TMG_image_conv_LUT_save*, *TMG_image_conv_LUT_load*.

TMG_image_conv_LUT_load

USAGE

*Terr TMG_image_conv_LUT_load(Thandle Himage, ui16 LUT_type, char *filename)*

ARGUMENTS

Himage	Handle to an image.
LUT_type	A look up table type as defined below and used by *TMG_image_convert*.
filename	Pointer to a *NUL* terminated ASCII text string.

DESCRIPTION

This function loads a previously generated and saved LUT from a file called *filename*. *Himage* is only used when loading a *TMG_Y16_TO_PALETTED_LUT*. *Himage* contains the actual data width of the grayscale data - for example, although *TMG_Y16* is the pixel format, there may only be 10 bits of valid grayscale data, resulting in a corresponding LUT size of 1024.

Note that the actual LUTs are internal globals within the TMG library and not related to any other structure.

The LUT type is determined by *LUT_type* and can be one of the following:

TMG_Y8_TO_PALETTED_LUT	Used to convert an 8 bit grayscale image to an 8 bit paletted image.
TMG_Y16_TO_PALETTED_LUT	Used to convert a 16 bit grayscale image to an 8 bit paletted image.
TMG_RGB16_TO_PALETTED_LUT	Used to convert a 16 bit RGB image (*TMG_RGB16*) to an 8 bit paletted image.
TMG_YUV422_TO_PALETTED_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to an 8 bit paletted image.
TMG_YUV422_TO_RGB15_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to a 15 bit RGB image (*TMG_RGB15*).
TMG_YUV422_TO_RGB16_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to a 16 bit RGB image (*TMG_RGB16*).

The use of this function can save time in the regeneration of a LUT. (Some LUTs are quite slow to generate - up to 20 seconds depending on the type of machine). It is also useful if an optimum colourmap (and LUT) have been generated using a test image that may not always be available. Note the colourmap can be saved and re-loaded with an image if a paletted file format is used (*TMG_PALETTED*).

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment generates, saves and then re-loads a colourmap and a LUT:

```
/* Generate an optimum colourmap */
TMG_cmap_generate(hImage, 256, TMG_RUN);
TMG_image_conv_LUT_generate(hImage, TMG_YUV422_TO_PALETTED_LUT);

/* now save our optimum colourmap and LUT */
TMG_image_convert(hImage, hPalImage, TMG_PALETTED, 0, TMG_RUN);
TMG_image_set_outfilename(hPalImage, "palette.tif");
```

```
TMG_image_write(hPalImage, TMG_NULL, TMG_TIFF, TMG_RUN);
TMG_image_conv_LUT_save(hPalImage, TMG_YUV422_TO_PALETTED_LUT, "yuv2p.lut");
.
.
/* load our previously saved colourmap and LUT */
TMG_image_set_infilename(hPalImage, "palette.tif");
/* we only need to read the palette - not the whole image */
TMG_image_set_parameter(hPalImage, TMG_HEIGHT, 0);
TMG_image_read(hPalImage, TMG_NULL, TMG_RUN);
TMG_image_conv_LUT_load(hPalImage, TMG_YUV422_TO_PALETTED_LUT, "yuv2p.lut");
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_conv_LUT_save.

TMG_image_conv_LUT_save

USAGE

*Terr TMG_image_conv_LUT_save(Thandle Himage, ui16 LUT_type, char *filename)*

ARGUMENTS

Himage	Handle to an image.
LUT_type	A look up table type as defined below and used by *TMG_image_convert*.
filename	Pointer to a *NUL* terminated ASCII text string.

DESCRIPTION

This function saves an already generated LUT to a file called *filename*. *Himage* is only used when saving a *TMG_Y16_TO_PALETTED_LUT*. *Himage* contains the actual data width of the grayscale data - for example although *TMG_Y16* is the pixel format, there may only be 10 bits of valid grayscale data, resulting in a corresponding LUT size of 1024.

Note that the actual LUTs are internal globals within the TMG library and not related to any other structure.

The LUT type is determined by *LUT_type* and can be one of the following:

TMG_Y8_TO_PALETTED_LUT	Used to convert an 8 bit grayscale image to an 8 bit paletted image.
TMG_Y16_TO_PALETTED_LUT	Used to convert a 16 bit grayscale image to an 8 bit paletted image.
TMG_RGB16_TO_PALETTED_LUT	Used to convert a 16 bit RGB image (*TMG_RGB16*) to an 8 bit paletted image.
TMG_YUV422_TO_PALETTED_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to an 8 bit paletted image.
TMG_YUV422_TO_RGB15_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to a 15 bit RGB image (*TMG_RGB15*).
TMG_YUV422_TO_RGB16_LUT	Used to convert a 16 bit YUV 4:2:2 image (*TMG_YUV422*) to a 16 bit RGB image (*TMG_RGB16*).

The use of this function can save time in the regeneration of a LUT. (Some LUTs are quite slow to generate - up to 20 seconds depending on the type of machine). It is also useful if an optimum colourmap (and LUT) have been generated using a test image that may not always be available. Note the colourmap can be saved and re-loaded with an image if a paletted file format is used (*TMG_PALETTED*).

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See *TMG_image_conv_LUT_load* for an example piece of code.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_conv_LUT_load.

TMG_image_convert

USAGE

Terr TMG_image_convert(Thandle Hin_image, Thandle Hout_image, ui16 out_format, ui32 flags, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
out_format	The required output pixel format.
flags	Flags - such as *TMG_USE_LUT*, *TMG_IS_DIB*.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function converts the input image, *Hin_image*, to the output image *Hout_image*, such that the output image's pixel format is defined by *out_format*. *flags* is used to control the type of conversion used and/or the type of output image generated.

This function would be used to convert the pixel format of an image to a suitable format to allow it to be saved, displayed or processed.

Acceptable output pixel formats are listed below with a brief description. For a more detailed description of pixel formats, see "Pixel Formats and Return Values" at the start of this manual.

TMG_Y8	8 bit grayscale.
TMG_Y16	9 to 16 bits of grayscale. Actual number of grayscale bits given by the internal parameter *data_width* (see *TMG_DATA_WIDTH* in *TMG_image_set_parameter*).
TMG_YUV422	YUV 4:2:2 colour image format.
TMG_PALETTED	8 bit paletted data.
TMG_RGB8	8 bit colour defined as RRRGGGBB.
TMG_RGB15	15 bit colour defined as RRRRRGGGGGBBBBB.
TMG_RGB16	16 bit colour defined as RRRRRGGGGGGBBBBB
TMG_RGB24	24 bit colour with byte ordering RGB.
TMG_BGR24	24 bit colour with byte ordering BGR.
TMG_RGBX32	32 bit colour with byte ordering RGBX.
TMG_BGRX32	32 bit colour with byte ordering BGRX.
TMG_XBGR32	32 bit colour with byte ordering XBGR.
TMG_XRGB32	32 bit colour with byte ordering XRGB.
TMG_HSI	Hue, saturation and intensity representation.
TMG_Y8_OR_RGB24	This is a special "format" used to indicate that a paletted image should converted to either *TMG_Y8* or *TMG_RGB24* - depending on whether the paletted image is grayscale or colour. See below for details.

Acceptable *flags* are listed and described below:

TMG_USE_LUT	This flag is only valid for colourspace conversion between *TMG_YUV422*, RGB and Y8 to paletted formats. It indicates that a LUT should be used instead of matrix multiplication. A LUT is faster but the quality of conversion is not as good.
TMG_IS_DIB	This flag indicates that image data in the output image should be in the DIB format based on the selected pixel format, *out_format*. This is used when the output image is to be displayed an a DIB image (i.e. under Windows NT/95/3.1).
0	Anything else, i.e. no special flag required.

The various types of conversion are listed below:

SIMPLE PIXEL FORMAT CONVERSION

This is the simplest group of conversions and refers to conversion between pixel formats without colourspace conversion or any other flags used. For example conversion from *TMG_Y8* to *TMG_RGB16* to suit a particular graphics card, or from *TMG_RGBX32* to *TMG_RGB24* so that the image can be saved as a TIFF file. See "Simple Pixel Format Conversion" in the examples section below.

The formula used for conversion between colour RGB formats and grayscale is:

$Y = 0.299R + 0.587G + 0.114B$

The formula used for conversion between colour RGB formats and CMYK is:

$C = 255 - R$
$M = 255 - G$
$Y = 255 - B$
$K = 0$

COLOURSPACE CONVERSION

This refers to any conversion either to or from *TMG_YUV422*. This format is a different colourspace (see Glossary for definition) and requires the use of either multiplication to achieve full resolution or a LUT to achieve reduced resolution. This type of conversion is usually needed from *TMG_YUV422* to an RGB format - for example acquisition from a colour frame grabber or JPEG decompression hardware. Conversion is rarely needed from RGB colourspace to YUV colourspace. Hence any conversion from RGB (or *TMG_Y8*) colourspace to *TMG_YUV422* will always use the (slow but accurate) matrix multiplication method. See "Conversion to YUV 4:2:2" in the examples section below. The conversion from *TMG_YUV422* to RGB colourspace has the option of using matrix multiplication or a software LUT. To do the conversion using a LUT, *flags* is set to *TMG_USE_LUT*. If the conversion LUT is not already generated it will be automatically generated the first time the function is called (see *TMG_image_conv_LUT_generate*). The size of this LUT (and hence the quality) vary slightly between operating systems (see "Operating System Issues" at the start of this manual).

When converting from *TMG_YUV422* to a paletted image, the required palette must be set up in advance of the LUT generation. This is because the LUT generation function needs to know the target colourmap (or palette) in advance, so it knows what colours (actually indexes into the colourmap) that the input YUV 4:2:2 data should be mapped to. See "Conversion from *TMG_YUV422* to Paletted" in the examples section below.

For a detailed example of YUV 4:2:2 to paletted conversion, see the extended examples in the "Sample Applications" section for more details. For conversion from YUV 4:2:2 to RGB formats, see "Conversion from YUV422" in the examples section below.

The formula used for conversion from RGB formats to YUV 4:2:2 is as follows:

Y = 0.299R + 0.587G + 0.114B
U = -0.169R - 0.331G + 0.500B
V = 0.500R - 0.419G - 0.081B

The formula used for conversion from YUV 4:2:2 and RGB formats is as follows:

R = Y + 0U + 1.402V
G = Y - 0.344U - 0.714V
B = Y + 1.772U + 0V

In both of the above formulas, R, G, B and Y all have the range 0..255, whilst U and V have the range -128 to +127. (The U and V components are level shifted by adding 128 to allow them to be stored as an 8 bit unsigned number.)

There is also limited support for HSI colourspace. See BUGS / NOTES below. The formulas used to convert from YUV 4:2:2 to HSI are:

H = tan-1(B-Y)/(R-Y) (Implemented using the Chroma keying *UV_to_hue_LUT*.)

Note: Small values (noise) are trapped in the LUT and set to 180 degrees.

S = 255 - MIN(R,G,B) (Simple approximation.)

I = Y

See also *TMG_SPL_HSI_to_RGB_pseudo_colour*.

PALETTED (COLOURMAPPED) CONVERSION

This refers to any conversion either to or from *TMG_PALETTED*. For conversion from a paletted image, the output image format should be *TMG_Y8_OR_RGB24*. This is a special format, that isn't really a pixel format, but a way of instructing the conversion function that it should convert to either *TMG_Y8* or *TMG_RGB24* depending on whether the paletted image is grayscale or colour. (It determines this automatically during conversion.) See "Conversion from Paletted" in the examples section below.

To convert to a paletted image involves the generation of a colourmap (or palette) first of all. This can be done several ways using the TMG colourmap ("*TMG_cmap*") functions (see *TMG_cmap_generate* and related).

The conversion from *TMG_YUV422* to paletted always uses a LUT and is described in the "Colourspace Conversion" section above. See "Conversion from *TMG_YUV422* to Paletted" in the examples section below.

A LUT can also be used to convert directly from *TMG_Y8*, *TMG_Y16* or *TMG_RGB16* to *TMG_PALETTED*. This is useful for fast display to paletted displays. This methodology is used extensively in the Snapper Solaris SDK. See "Conversion from *TMG_Y8* to Paletted - using a LUT" in the examples section below. Note that given suitable acquisition hardware, the grayscale to paletted conversion LUT could be loaded into a hardware LUT thus saving time in paletted conversion.

There is an alternative method to using a LUT, suitable for *TMG_Y8* and *TMG_RGB24*. The algorithm is based upon generating an optimum palette, then mapping each input pixel to its closest colour in the palette. For *TMG_Y8* the quality is the same as the LUT method but the conversion takes marginally longer. For *TMG_RGB24* this is the only method of directly generating a paletted image. See "Conversion from *TMG_RGB24* to Paletted" in the examples section below.

CONVERSION TO DIB

If the flag *TMG_IS_DIB* is used during image conversion, the output image format will be a DIB (device independent bitmap). This is a one way process - there is currently no function to convert from a DIB back to an ordinary TMG image. This conversion is designed to be used for display and printing under the appropriate operating systems (Windows NT/95/3.1). See "Conversion to DIB" in the examples section below.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

SIMPLE PIXEL FORMAT CONVERSION

This example shows how to convert between simple pixel formats in the same (RGB) colourspace:

```
/* hSrcImage is a TMG_RGBX32 image, we will convert it to TMG_RGB16,
 * ready for display (the display is format RGB16).
 * Note we are processing in 1 strip (i.e. whole image at once).
 */
TMG_image_convert(hSrcImage, hDispImage, TMG_RGB16, 0, TMG_RUN);
```

CONVERSION TO YUV 4:2:2

This example shows how to convert from RGB colourspace to YUV colourspace:

```
TMG_image_convert(hSrcImage, hYUVImage, TMG_YUV422, 0, TMG_RUN);
```

For a more detailed example, see the chroma keying examples in the "Sample Applications" section.

CONVERSION FROM YUV 4:2:2 TO RGB16 - WITHOUT A LUT

```
/* Slow but accurate */
TMG_image_convert(hYUVImage, hRGBImage, TMG_RGB16, 0, TMG_RUN);
```

CONVERSION FROM YUV 4:2:2 TO RGB16 - USING A LUT

```
/* Fast but less accurate */
TMG_image_convert(hYUVImage, hRGBImage, TMG_RGB16, TMG_USE_LUT, TMG_RUN);
```

CONVERSION FROM YUV 4:2:2 TO PALETTED

This example shows how to convert from YUV 4:2:2 to a paletted image using, such that the palette is made up of 3 bits of red, 3 bits of green and 2 bits of blue:

```
/* map to an equal mix of red, green. blue */
TMG_cmap_set_type(hYUVImage, TMG_332_RGB);
/* force the generation of a new LUT within TMG_image_convert */
TMG_image_conv_LUT_destroy(TMG_YUV422_TO_PALETTED_LUT);

/* the first call will be slower - as its generating the LUT */
TMG_image_convert(hYUVImage, hPalImage, TMG_PALETTED, TMG_USE_LUT, TMG_RUN);
```

For a more detailed example, in which certain colours are reserved and optimum colourmap generation, see the extended examples in the "Sample Applications" section.

CONVERSION FROM PALETTED

This example shows how to convert from a paletted image to either an RGB or grayscale one:

```
TMG_image_set_infilename(hSrcImage, "sky.tif");
TMG_image_set_parameter(hSrcImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hSrcImage, TMG_NULL, TMG_RUN);
if (TMG_image_get_parameter(hSrcImage, TMG_PIXEL_FORMAT) == TMG_PALETTED) {
   printf("\nConverting from paletted...");
   TMG_image_convert(hSrcImage, hImage, TMG_Y8_OR_RGB24, 0, TMG_RUN);
}
else
   TMG_image_move(hSrcImage, hImage);
```

```
PixelFormat = (ui16) TMG_image_get_parameter(hImage, TMG_PIXEL_FORMAT);
if (PixelFormat == TMG_RGB24)
   printf("\nWe have a 24 bit colour image");
else
   printf("\nWe have a grayscale image");
```

CONVERSION FROM GRAYSCALE TO PALETTED - USING A LUT

This example shows how to convert from *TMG_Y8* to *TMG_PALETTED*:

```
/* set colourmap to a grayscale ramp */
TMG_cmap_set_type(hYImage, TMG_GRAYSCALE_RAMP);
/* force the generation of a new LUT within TMG_image_convert */
TMG_image_conv_LUT_destroy(TMG_Y8_TO_PALETTED_LUT);

/* the first call will be slower - as its generating the LUT */
TMG_image_convert(hYImage, hPalImage, TMG_PALETTED, TMG_USE_LUT, TMG_RUN);
```

CONVERSION FROM RGB24 TO PALETTED

This example shows how to convert from *TMG_RGB24* to a paletted image, such that the palette is optimised to the colours contained in the source image:

```
/* generate the optimum colourmap */
TMG_cmap_generate(hRGBImage, 256, TMG_RUN);
TMG_image_convert(hRGBImage, hPalImage, TMG_PALETTED, 0, TMG_RUN);
```

CONVERSION TO DIB

This example shows how to convert to a DIB suitable for display:

```
/* 24 bit colour DIBs use the BGR24 pixel format */
TMG_image_convert(hRGBImage, hDIBImage, TMG_BGR24, TMG_IS_DIB, TMG_RUN);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

Not all combinations of image format conversions are supported. If you come across an unsupported conversion option which you require please fax in the Bug Report Form (in the Appendices to this manual) and it will be fixed in the next release (or sooner via the Bulletin Board System).

There is limited support for *TMG_Y16*.

The conversion to DIB only generates DIBs with pixel format *TMG_BGR24*.

There's limited support for *TMG_HSI*. The only conversion option is to *TMG_HSI* from *TMG_YUV422*.

Dithering is currently not supported as a conversion flag option.

SEE ALSO

TMG_image_conv_LUT_generate, *TMG_image_conv_LUT_destroy*, *TMG_cmap_generate*, *TMG_SPL_HSI_to_RGB_pseudo_colour*.

TMG_image_copy

USAGE

Terr TMG_image_copy(Thandle Hin_image, Thandle Hout_image)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.

DESCRIPTION

This function copies the image, consisting of various parameters and the image data itself, from *Hin_image* to *Hout_image*. If *Hout_image* has any image data associated with it and it is not locked (see *TMG_image_set_flags*), it is freed and new memory is allocated for the image. If the image memory in *Hout_image* is locked, then it will be preserved and the image data from *Hin_image* is copied to this area in *Hout_image*. Note that there must be sufficient memory already allocated in *Hout_image* if it is locked, otherwise a general protection fault or similar will occur.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

This example reads in a TIFF file in strips, copies it and saves it. Note that a "dummy" read is needed first to determine the height of the image so that the number of strips can be calculated:

```
TMG_image_set_infilename(hImage, "sky.tif");
/* The outfilename parameter gets transferred to hOutImage */
TMG_image_set_outfilename(hImage, "sky_copy.tif");

/* open the file to read the image height */
TMG_image_set_parameter(hImage, TMG_HEIGHT, 0);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_image_read(hImage, TMG_NULL, TMG_RESET);

/* Now set up the strip processing loop and proceed, 8 lines at time */
TMG_image_set_parameter(hImage, TMG_LINES_THIS_STRIP, 8);
TotalStrips = TMG_image_calc_total_strips(hImage);

for (Strip = 0; Strip < TotalStrips; Strip++) {
   TMG_image_read(hImage, TMG_NULL, TMG_RUN);
   TMG_image_copy(hImage, hOutImage);
   TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
}
TMG_image_set_parameter(hImage, TMG_LINES_THIS_STRIP, 8); /* just in case */
```

Note that if the image height of "sky.tif" did not divide exactly by 8, the *TMG_LINES_THIS_STRIP* parameter of *hImage* will no longer contain 8 and may in applications different to this example need setting back to 8 - hence the "just in case" comment at the end.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_move.

TMG_image_create

USAGE

Terr TMG_image_create()

ARGUMENTS

None.

DESCRIPTION

This function creates a *Timage* structure by the use of malloc, and returns a handle to that *Timage* structure. (see the file "tmg.h" for the actual structure definition). It also performs some initialization - that is characters strings are set to '\0' and the image data pointer set to *NULL*. The structure variable *lines_this_strip* is set to 8. Note that no memory is created for the image itself - this is performed by TMG functions when loading or processing an image.

RETURNS

On success a valid handle is returned in the lower 16 bits of the return value (the upper 16 bits will be 0). On failure an error code will be returned in the upper 16 bits as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code creates an image and gets a handle to it:

```
Thandle hImage;   /* Handle to an image structure */

if ( ASL_is_err(hImage = TMG_image_create() )
   printf("Failed to create an image");
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_destroy, *TMG_JPEG_image_create*.

TMG_image_destroy

USAGE

Terr TMG_image_destroy(Thandle Himage)

ARGUMENTS

Himage Handle to an image or *TMG_ALL_HANDLES*.

DESCRIPTION

This function destroys an image structure (including JPEG images) by freeing all the memory associated with that structure.

If the parameter *TMG_ALL_HANDLES* is used, all TMG structures are destroyed and their associated handles freed. Note that not only are all images destroyed but all other TMG structures too - that is LUTs, chroma keying structures etc. This is a convenient way of destroying everything with just one function call - usually on program exit.

Any images with locked memory (*TMG_LOCKED*) will have that memory automatically freed prior to the image being destroyed.

Care must be taken if another application is using the TMG library in a multi-threaded environment. For example, it may be using some TMG LUT structures, and the use of *TMG_ALL_HANDLES* would destroy these without the other application knowing about it. In this type of environment, each application would destroy only its own image handles and then call *TMG_image_destroy* with the parameter *TMG_ALL_DATA_STRUCTURES*. This would destroy the data structures only if there were no image handles in use (and then return *ASL_OK*), otherwise, if there were image handles in use, it would return *ASL_ERR_IN_PROGRESS*.

Care must also be taken with memory that has been allocated by the application and not the TMG library. If memory has been allocated by the application and used by the TMG library (in an image structure), it must be freed at the application level and the internal image pointer set to *NULL* before calling *TMG_image_destroy*. One of the examples below shows how this is done.

RETURNS

ASL_OK or *ASL_ERR_IN_PROGRESS* (see above).

EXAMPLES

The following code destroys a previously created image:

```
Thandle hImage;
.
/* Destroy the image structure */
TMG_image_destroy(hImage);
.
/* Destroy all TMG structures */
TMG_image_destroy(TMG_ALL_HANDLES);
```

This next example shows how image memory is allocated and destroyed at the application level and not in the TMG library:

```
Thandle hImage;
IM_UI8 *pImageMemory;  /* "my" image memory */

hImage = TMG_image_create();
pImageMemory = malloc(100000);
TMG_image_set_ptr(hImage, TMG_IMAGE_DATA, pImageMemory);
TMG_image_set_flags(hImage, TMG_LOCKED, TRUE);
```

```
.
/* We now have hImage using our memory that the TMG library will not touch */
.
/* To destroy it we must free it and set the pointer to NULL */
free(pImageMemory);
TMG_image_set_ptr(hImage, TMG_IMAGE_DATA, NULL);
TMG_image_set_flags(hImage, TMG_LOCKED, FALSE);
TMG_image_destroy(hImage);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_create, *TMG_image_set_flags*.

TMG_image_find_file_format

USAGE

*Terr TMG_image_find_file_format(char *filename)*

ARGUMENTS

filename Name of an image file.

DESCRIPTION

This function attempts to find the file format of an image file called *filename*. This function is used internally by *TMG_image_read*.

RETURNS

On success one of the following file types is returned as a #define: *TMG_TIFF, TMG_TARGA, TMG_JPEG, TMG_EPS, TMG_BMP;* otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment attempts to find the image file format:

```
ui32 dwResult;
ui16 wFileFormat = 0;

Result = TMG_image_find_file_format("sky.tif");
if ( ASL_is_err(dwResult) )
   printf("Failed to recognise the file type");
else
   wFileFormat = ASL_get_ret(dwResult);

if (wFileFormat == TMG_TIFF)
   printf("The file is a TIFF file");
```

BUGS / NOTES

This function is not guaranteed to work on image files that have not been generated by the TMG library (but in general it should work).

SEE ALSO

TMG_image_read.

TMG_image_free_data

USAGE

Terr TMG_image_free_data(Thandle Himage)

ARGUMENTS

Himage Handle to an image.

DESCRIPTION

This function frees the image data used internally by *Himage*. This may include raw image data or JPEG compressed data. If the memory is locked (i.e. the *TMG_LOCKED* flag is set to true), freeing the data will have no effect (but *ASL_OK* will still be returned as this is not regarded as an error).

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment, lifted from the Windows 3.1 application D16, frees the device dependent bitmap to save memory:

```
if (D16.m_bModeChange == TRUE)
{
   /* Free the image memory associated with the DDB image to save memory */
   TMG_image_set_flags(D16.m_hDDBImage, TMG_LOCKED, FALSE);
   TMG_image_free_data(D16.m_hDDBImage);
}
```

BUGS / NOTES

This function is rarely needed is a user application.

SEE ALSO

TMG_image_malloc_a_strip, *TMG_image_set_flags*.

TMG_image_get_flags

USAGE

Tboolean TMG_image_get_flags(Thandle Himage, ui16 type)

ARGUMENTS

Himage	Handle to an image.
type	Flag type.

DESCRIPTION

This function returns the state (*TRUE* or *FALSE*) of the flag, selected by *type*, in *Himage*.

The flag types are described in *TMG_image_set_flags*.

RETURNS

TRUE or *FALSE* reflecting the flag status.

EXAMPLES

The following code shows how a file is read then tested to see if its a JPEG file:

```
TMG_image_set_parameter(hSrcImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
if (TMG_image_read(hSrcImage, NULL, TMG_RUN) != ASL_OK) {
   printf("Failed to read file");
}
if (TMG_image_get_flags(hSrcImage, TMG_IS_JPEG) == TRUE)
   /* decompress the image */
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_set_flags, *TMG_image_convert*, *TMG_image_set_parameter*.

TMG_image_get_infilename, TMG_image_get_outfilename

USAGE

*char *TMG_image_get_infilename(Thandle Himage)*

*char *TMG_image_get_outfilename(Thandle Himage)*

ARGUMENTS

Himage Handle to an image.

DESCRIPTION

These functions return a pointer to the input/output file name associated with *Himage*. The pointer returned may point to relocatable memory in the image structure, therefore the contents should be copied to memory within the application.

The file name may also contain a full path to the file - for example "c:\snapsdk\apps\imv\test.tif".

RETURNS

A pointer to a *NUL* terminated string if *Himage* is valid, otherwise it returns *NULL*.

EXAMPLES

```
char szFileName[256];

TMG_image_set_infilename(hImage, "sky.tif");
.
strcpy(szFileName, TMG_image_get_infilename(hImage));
printf("File = %s", szFileName);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_set_infilename,
TMG_image_set_outfilename.

TMG_image_get_parameter

USAGE

ui32 TMG_image_get_parameter(Thandle Himage, ui16 parameter)

ARGUMENTS

Himage	Handle to an image.
parameter	Parameter type.

DESCRIPTION

This function returns the value of an internal parameter from *Himage* selected by *parameter*. The parameter is always returned as a 32 bit unsigned integer although some of the parameters are stored as 16 bit unsigned integers internally.

The parameter types are described in *TMG_image_set_parameter*, apart from the read only parameter *TMG_LIBRARY_REV_LEVEL*. This parameter returns the revision level of the library as a 5 digit number which represents major.minor.sub-minor. For example 32002 means version 3.2 rev. 2. This is useful to check that the correct revision level of DLL is present.

RETURNS

The parameter selected by *parameter* as an unsigned 32 bit integer (*ui32*).

EXAMPLES

The following code fragment reads a file and detects whether its a 24 bit colour image:

```
ui16 PixelFormat;

TMG_image_set_infilename(hImage, "car.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
PixelFormat = (ui16) TMG_image_get_parameter(hImage, TMG_PIXEL_FORMAT);
if (PixelFormat == TMG_RGB24)
   printf("Its a 24 bit colour image");
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_set_parameter, *TMG_image_set_flags*.

TMG_image_get_ptr

USAGE

*void *TMG_image_get_ptr(Thandle Himage, ui16 type)*

ARGUMENTS

Himage	Handle to an image.
type	The pointer type (see list below).

DESCRIPTION

This function returns the internal pointer selected by *type*, in the image structure referenced by *Himage*. The return value must be cast to the required pointer type (shown below).

TMG_IMAGE_DATA and *TMG_JPEG_DATA* are the most likely ones to be used. The remainder are implemented for internal library use (and completeness).

Getting access to the image data via a pointer can be a useful way of reading and setting individual pixels. See the "Sample Application" sections in this manual.

The possible pointer types are as follows:

TMG_IMAGE_DATA	Returns the pointer to the raw image data. The return value should be cast to *IM_UI8**, *IM_UI16** or *IM_UI32**.
TMG_JPEG_DATA	Returns the JPEG compressed data pointer. The return value should be cast to *IM_UI8**.
TMG_JPEG_CURRENT_PTR	Returns the current pointer to JPEG data. (This is used for the replay of motion JPEG sequence files.) The return value should be cast to *IM_UI8**.
TMG_CMAP_STRUCT	Returns the pointer to the colourmap structure (*struct Tcmap**). Under Windows 3.1 the structure definition is *struct Tcmap far**. (There is a type defined as *CMAP_PTR** that automatically includes the *far* keyword when appropriate). The return value should be cast to *CMAP_PTR**.
TMG_PIMAGE_STRUCT	Returns the pointer to the image structure. The return value should be cast to *struct Timage**.
TMG_PIMAGE_PJPEG_STRUCT	Returns the pointer to the JPEG structure. The return value should be cast to *struct Tjpeg**.

See the section on "Operating System Issues" for a fuller description of defined types such as *IM_UI8*. See also the file "tmg.h" for the actual structure definitions mentioned above.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code gets a pointer to the actual image data in a TMG image:

```
Thandle hImage;
IM_UI8 *pImageMemory;  /* "my" image memory */

hImage = TMG_image_create();
...  /* Read in an image from disk for example */
pImageMemory = (IM_UI8*) TMG_image_get_ptr(hImage, TMG_IMAGE_DATA);
```

```
    /* We now have pImageMemory pointing at the start of our image data, so we can
          manipulate it directly */
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_set_flags, *TMG_image_set_ptr*.

TMG_image_is_colour

USAGE

Tboolean TMG_image_is_colour(Thandle Himage)

ARGUMENTS

Himage Handle to an image.

DESCRIPTION

Returns *TRUE* if the image is colour and *FALSE* if it is grayscale or bilevel (i.e. line art - 1 bit per pixel). Note that this function will return *FALSE* if the image is paletted and the colourmap (or palette) consists only of grayscales.

RETURNS

Returns *TRUE* or *FALSE*.

EXAMPLES

The following code fragment reads an image and determines if it is colour or not:

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
if (TMG_image_is_colour(hImage) == TRUE)
   printf("We have a colour image");
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_cmap_is_grayscale, *TMG_image_get_parameter*.

TMG_image_malloc_a_strip

USAGE

Terr TMG_image_malloc_a_strip(Thandle Himage)

ARGUMENTS

Himage Handle to an image.

DESCRIPTION

This function allocates sufficient memory for one strip of the image currently being processed. The amount of memory allocated is determined by two parameters internal to the image structure - *bytes_per_line* and *lines_this_strip*. If *lines_this_strip* is set the height of the image, then the image will be processed in one strip. The function *TMG_image_check*calculates and fills in *bytes_per_line* from the pixel format and width of the image.

The method used for memory allocation varies between the different operating systems. For example, Solaris 2 uses *memalign*. The #defines *MALLOC* and *FREE* are used internally in the TMG library and are defined in the file "asl_gen.h" available with the SDK - please refer to this file for more details.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the "Test Pattern Generation" example in the extended examples section in the "Sample Applications" section.

BUGS / NOTES

This function is rarely needed is a user application.

SEE ALSO

TMG_image_free_data, *TMG_image_set_parameter*, *TMG_image_set_flags*.

TMG_image_move

USAGE

Terr TMG_image_move(Thandle Hin_image, Thandle Hout_image)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.

DESCRIPTION

This function copies the image parameters and moves the data (i.e. the pointer is copied) from *Hin_image* to *Hout_image*. The internal data pointer in *Hin_image* is set to *NULL*.

If either the input or output image has locked memory, then this function will fail - as locked memory cannot be freed or moved. Generally the application should be written not to use this function with locked memory (in fact not to use this function at all if possible - but sometimes it is useful). Alternatively *TMG_image_copy* can be used to copy from locked memory to locked/unlocked memory. However in speed critical applications it wastes time to copy image data around and the application should be re-structured so as not to have to do this.

Obviously this function is much faster than *TMG_image_copy* as the data is simply moved - in fact the time taken to perform the move is insignificant in an application, but the time taken to copy generally is significant.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

This example reads in a TIFF file in strips, moves it and saves it. Note that a "dummy" read is needed first to determine the height of the image so that the number of strips can be calculated:

```
TMG_image_set_infilename(hImage, "sky.tif");
/* The outfilename parameter gets transferred to hOutImage */
TMG_image_set_outfilename(hImage, "sky_copy.tif");

/* open the file to read the image height */
TMG_image_set_parameter(hImage, TMG_HEIGHT, 0);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_image_read(hImage, TMG_NULL, TMG_RESET);

/* Now set up the strip processing loop and proceed, 8 lines at time */
TMG_image_set_parameter(hImage, TMG_LINES_THIS_STRIP, 8);
TotalStrips = TMG_image_calc_total_strips(hImage);

for (Strip = 0; Strip < TotalStrips; Strip++)
{
   TMG_image_read(hImage, TMG_NULL, TMG_RUN);
   TMG_image_move(hImage, hOutImage);
   TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
}
TMG_image_set_parameter(hImage, TMG_LINES_THIS_STRIP, 8); /* just in case */
```

Note that if the image height of "sky.tif" did not divide exactly by 8, the *TMG_LINES_THIS_STRIP* parameter of *hImage* will no longer contain 8 and may in applications different to this example need setting back to 8 - hence the "just in case" comment at the end.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_copy, *TMG_image_set_flags*.

TMG_image_read

USAGE

Terr TMG_image_read(Thandle Hin_image, Thandle Hout_image, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to an optional output image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function reads an image from disk or memory. If reading from disk, the function is called with *Hout_image* set to *TMG_NULL*. *TMG_image_read* uses *TMG_image_find_file_format* internally to determine the image file format. The supported file formats are TIFF, Windows Bitmap, Targa, Encapsulated PostScript and JPEG/JFIF. (Note that JPEG/JFIF files are only read in, *TMG_JPEG_decompress* is required to decompress them.)

This function will read an image in strips or as one whole strip (i.e. the whole image). When reading the whole image at once, the height of the image should be set to *TMG_AUTO_HEIGHT*. When reading in strips, the internal image parameter *lines_this_strip* is set as usual. (See example below.)

The purpose of reading image data from memory - i.e. from *Hin_image* to *Hout_image*, would be to read strips of *Hin_image* at a time (say 8 lines per strip) to conserve memory for a chain of image processing functions.

The concept of strips does not directly apply to reading JPEG data. Therefore if the input file is a JPEG file, the complete image will always be read, unless the *TMG_LINES_THIS_STRIP* parameter set in the input image (see *TMG_image_set_parameter*) is zero - in which case the JPEG file will be opened (using *TMG_JPEG_file_open*) and then closed after the image dimensions etc have been be read. This will all happen internally to *TMG_image_read*.

When reading non-JPEG files, the only three possible pixel formats in the resulting image are *TMG_Y8* for grayscale, *TMG_RGB24* for colour and *TMG_PALETTED* for a paletted image. (Note that the paletted image may actually be grayscale - see *TMG_cmap_is_grayscale* and *TMG_image_convert*.)

Internally *TMG_image_read* calls the following file read functions:

TMG_read_from_memory(Thandle Hin_image, ui16 TMG_action)
TMG_read_TIFF_file(Thandle Hin_image, ui16 TMG_action)
TMG_read_EPS_file(Thandle Hin_image, ui16 TMG_action)
TMG_read_TGA_file(Thandle Hin_image, ui16 TMG_action)
TMG_read_BMP_file(Thandle Hin_image, ui16 TMG_action)
TMG_JPEG_file_read(Thandle Hin_image)

It is recommended that the *TMG_image_read* function is used, as it provides a simple common interface. However applications linked with static libraries may prefer to use the individual function calls, to reduce the size of the resulting executable.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads a TIFF file (the whole image in one go) and displays it to a 16 bit colour display:

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_image_convert(hImage, hDispImage, TMG_RGB16, 0, TMG_RUN);
TMG_display_image(hDisplay, hDispImage, TMG_RUN);
```

This next example reads in a TIFF file in strips and saves it as its mirror image. This strip processing approach would be needed for very large images. Note that a "dummy" read is needed first to determine the height of the image so that the number of strips can be calculated:

```
TMG_image_set_infilename(hImage, "sky.tif");
/* The outfilename parameter gets transferred to hOutImage */
TMG_image_set_outfilename(hImage, "sky_mirror.tif");

/* open the file to read the image height */
TMG_image_set_parameter(hImage, TMG_HEIGHT, 0);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_image_read(hImage, TMG_NULL, TMG_RESET);

/* Now set up the strip processing loop and proceed, 8 lines at time */
TMG_image_set_parameter(hImage, TMG_LINES_THIS_STRIP, 8);
TotalStrips = TMG_image_calc_total_strips(hImage);

for (Strip = 0; Strip < TotalStrips; Strip++) {
   TMG_image_read(hImage, TMG_NULL, TMG_RUN);
   TMG_IP_mirror_image(hImage, hOutImage, TMG_RUN);
   TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
}
```

This final example assumes the complete image is in memory in *hFullImage*, and again it is necessary to write the mirror image of the image to file. Because there may not be enough memory to hold the full mirrored image, it is necessary to write it in strips:

```
/* The outfilename parameter gets transferred to hOutImage */
TMG_image_set_outfilename(hFullImage, "sky_mirror.tif");

/* Now set up the strip processing loop and proceed, 8 lines at time */
TMG_image_set_parameter(hFullImage, TMG_LINES_THIS_STRIP, 8);
TotalStrips = TMG_image_calc_total_strips(hFullImage);

for (Strip = 0; Strip < TotalStrips; Strip++) {
   TMG_image_read(hFullImage, hStripImage, TMG_RUN);
   TMG_IP_mirror_image(hStripImage, hOutImage, TMG_RUN);
   TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
}
FullHeight = TMG_image_get_parameter(hFullImage, TMG_HEIGHT);
TMG_image_set_parameter(hFullImage, TMG_LINES_THIS_STRIP, FullHeight);
```

BUGS / NOTES

Of the supported TIFF and BMP formats, only uncompressed image data is supported.

Only EPS files written using *TMG_image_write* can be read. The TMG library does not have any PostScript interpreting ability.

SEE ALSO

TMG_image_set_infilename, *TMG_image_set_outfilename*, *TMG_image_write*, *TMG_JPEG_file_read*.

TMG_image_set_flags

USAGE

Terr TMG_image_set_flags(Thandle Himage, ui16 type, Tboolean state)

ARGUMENTS

Himage	Handle to an image or *TMG_ALL_HANDLES* to select all images.
type	Flag type.
state	Either *TRUE* or *FALSE*.

DESCRIPTION

This function sets flags in *Himage* which are then subsequently tested on by various TMG functions - in particular *TMG_image_convert*.

The flags are as follows:

TMG_LOCKED	This indicates that once the image memory (JPEG or raw) has been allocated it will not be freed (or re-allocated) unless the image is destroyed.
TMG_IS_JPEG	This is an internal flag used to indicate that the image contains JPEG data.
TMG_IS_DIB	This indicates that the image is a DIB. This is the standard Windows DIB structure stored in the image memory area in the image. It is also used by *TMG_image_convert* to indicate the output image should be a DIB.
TMG_DIB_NON_INVERTED	This flag indicates that the DIB is not inverted. In the original DIB format, the image was inverted (vertically), but now increasingly DIBs may contain the image the other (correct) way up.
TMG_USE_LUT	This flag is used by *TMG_image_convert* to indicate that the colourspace conversion from YUV 4:2:2 data to RGB data should use a LUT.
TMG_HALF_ASPECT	This flag indicates that the image has half the usual aspect ratio - in other words if displayed normally the image will appear squashed vertically. This flag is usually used to indicate that the image is a single field of video. Other routines, such as display routines, may examine this flag to determine how to display the image.
TMG_DATA_STREAM	This flag indicates that the data in the image structure is not formatted in the usual way with a fixed image width and height. For example it may simply represent a continuous stream of data - not necessarily image data. This can be useful when used in conjunction with the Snapper acquisition libraries for acquiring non-standard data formats.

Himage can be *TMG_ALL_HANDLES* which is sets the flag as requested in all images. This is particularly useful to unlock all image memory prior to destroying all images.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code shows how to lock memory in an image and also how to unlock all images:

```
/* Lock memory to save re-allocation time */
TMG_image_set_flags(hImage, TMG_LOCKED, TRUE);
.
/* Destroy all images */
TMG_image_set_flags(TMG_ALL_HANDLES, TMG_LOCKED, FALSE);
TMG_image_destroy(TMG_ALL_HANDLES);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_get_flags, *TMG_image_convert*, *TMG_image_set_parameter*.

TMG_image_set_infilename, TMG_image_set_outfilename

USAGE

*Terr TMG_image_set_infilename(Thandle Himage, char *filename)*

*Terr TMG_image_set_outfilename(Thandle Himage, char *filename)*

ARGUMENTS

Himage	Handle to an image.
filename	Pointer to a NUL terminated ASCII text string.

DESCRIPTION

These functions set the input/output file name of the image referenced by *Himage*. This input file name is used by any file reading functions, such as *TMG_image_read*, and the output file name is used by file writing functions, such as *TMG_image_write*.

The input file name and the output file name are stored in separate fields within the image structure and any TMG processing function will propagate these parameters to "downstream" functions.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads, sub-samples, then writes a TIFF file (the whole image in one go):

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_outfilename(hImage, "sky_x2.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_IP_subsample(hImage, hOutImage, 2, TMG_RUN);
TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_get_infilename,
TMG_image_get_outfilename, TMG_image_read.

TMG_image_set_parameter

USAGE

Terr TMG_image_set_parameter(Thandle Himage, ui16 parameter, ui32 value)

ARGUMENTS

Himage	Handle to an image.
parameter	Parameter type.
value	The actual value passed in as a 32 bit unsigned integer (although some parameters will only be 16 bit unsigned integers).

DESCRIPTION

This function sets internal parameters in the image structure referenced by *Himage*.

The parameters are as follows:

TMG_PIXEL_FORMAT	This sets the pixel format of the image, *Himage*. As well as setting the pixel format it will also automatically set the internal parameter *depth*. Pixel formats are type *ui16*. For a complete list of pixel formats see the section "Pixel Formats and Return Types" at the start of this manual. This option is rarely needed in a user application.
TMG_WIDTH	This parameter sets the width of an image. This would not normally be needed in a user application.
TMG_HEIGHT	This parameter sets the height of an image. This would usually be used with *value* set to *TMG_AUTO_HEIGHT* for reading an image into memory from file - see *TMG_image_read*.
TMG_DEPTH	This parameter sets the depth of the image. Depth refers to the number of bits per pixels - for example, pixel format *TMG_RGB16* would have a depth of 16. This would not normally be needed in a user application.
TMG_LINES_THIS_STRIP	This sets the number of lines to read/process/write per iteration. If used, a typical value would be 8. See the section "Concepts" at the start of this manual.
TMG_BYTES_PER_LINE	This represents the number bytes from pixel 1 on line N to pixel 1 on line N+1. It is automatically set by the function *TMG_image_check* (used internally) and is used as an "accelerator" for processing an image.
TMG_FIELD_ID	This is used to indicate which field is present if the image contains only a single field of video data (see also the *TMG_HALF_ASPECT* flag under *TMG_image_set_flags*). Valid settings are *TMG_1ST_FIELD*, *TMG_2ND_FIELD*, *TMG_FRAME*, *TMG_FIELD_1_OR_2*, *TMG_FRAME_FIELDS_12* or *TMG_FRAME_FIELDS_21*.
TMG_NUM_BYTES_DATA	This represents the total number of bytes of image data. It is only valid when used with the image flag *TMG_DATA_STREAM*.
TMG_JPEG_NUM_BYTES_DATA	This represents the total number of bytes of JPEG data. It is only valid when the image contains JPEG data. When multiple JPEG images are contained within one image handle, this represents the total amount of data, including the restart markers between frames.

TMG_DATA_WIDTH	This represents the actual number of valid bits of grayscale data. For example, 12 bit grayscale data (acquired from say a 12 bit digital camera) would be stored in an image with pixel format *TMG_Y16*. The data width would be set to 12 to inform TMG functions that image data is represented by the least significant 12 bits of each 16 bit word.
TMG_NUM_FRAMES	This represents the number of frames of image data or JPEG image data contained in *Himage*. It is mainly used by the motion JPEG functions for sequence acquisition and replay.
TMG_CURRENT_FRAME	This indicates the frame that the current data pointer is pointing to in a sequence of frames contained in *Himage*. It is mainly used by the motion JPEG functions for sequence acquisition and replay.
TMG_CMAP_SIZE	This is used to set the colourmap (or palette) size. By default the colourmap size 256 - that is 256 colour entries - each defined by 24 bits of RGB. See the colourmap examples in the "Sample Applications" section at the start of this manual.
TMG_NUM_PLANES	This indicates the number of planes of data in the image. This is calculated as follows: *TMG_RGB8*, *TMG_RGB15*, *TMG_RGB16* and *TMG_YUV422* are return a value of 3; *TMG_Y16* returns a value of 1; all other formats return the image depth divided by 8.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads, sub-samples, then writes a TIFF file. It performs each step on the whole image as determined by setting *TMG_HEIGHT* to the special parameter *TMG_AUTO_HEIGHT*:

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_outfilename(hImage, "sky_x2.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_IP_subsample(hImage, hOutImage, 2, TMG_RUN);
TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_get_parameter, *TMG_image_set_flags*.

TMG_image_set_ptr

USAGE

*Terr TMG_image_set_ptr(Thandle Himage, ui16 type, void *ptr)*

ARGUMENTS

Himage	Handle to an image.
type	The pointer type (see list below).
ptr	The pointer itself.

DESCRIPTION

This function sets internal pointers in the structure referenced by *Himage*. It is most commonly used when the application program wishes to allocate memory (for JPEG or raw image data) instead of letting the TMG library do it. For example, the application program may want to force the TMG library to use a particular area or type of memory (perhaps shared with another program in a driver application).

The possible pointer types are as follows:

TMG_IMAGE_DATA	Sets the raw image data pointer.
TMG_JPEG_DATA	Sets the JPEG compressed data pointer.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code allocates memory for image data and then destroys it in exit:

```
Thandle hImage;
IM_UI8 *pImageMemory;  /* "my" image memory */

hImage = TMG_image_create();
pImageMemory = malloc(100000);
TMG_image_set_ptr(hImage, TMG_IMAGE_DATA, pImageMemory);
TMG_image_set_flags(hImage, TMG_LOCKED, TRUE);
.
/* We now have hImage using our memory that the TMG library will not touch */
.
/* To destroy it we must free it and set the pointer to NULL */
free(pImageMemory);
TMG_image_set_ptr(hImage, TMG_IMAGE_DATA, NULL);
TMG_image_destroy(hImage);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_set_flags, *TMG_image_get_ptr*.

TMG_image_write

USAGE

Terr TMG_image_write(Thandle Hin_image, Thandle Hout_image, ui16 format, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to an optional output image.
format	The desired file format - one of: *TMG_MEMORY*, *TMG_TIFF*, *TMG_TARGA*, *TMG_JPEG*, *TMG_EPS*, *TMG_BMP*, *TMG_RAW*.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function writes an image to disk or memory. If writing to disk, the function is called with *Hout_image* set to *TMG_NULL* and *format* set to the required file format.

This function is essentially the inverse of *TMG_image_read*.

The *TMG_RAW* format listed above is simply a binary dump of the image data as a byte stream. 16 bit image data is always written in Intel order - that is least significant byte first).

The concept of strips does not directly apply to writing JPEG data to file (but more so than to reading JPEG data). In the situation of writing JPEG data in "strips", each strip of processed image will produce a certain amount of JPEG data. The number of bytes of JPEG data generated each strip by the compression function (e.g. *TMG_JPEG_compress*) will be stored internally in the image structure. This will be used by the writing function to know how many bytes of JPEG data to write each strip. The JPEG end of data marker will automatically be added at the end of the last strip. See *TMG_JPEG_compress* for an example. When it is necessary to write "strips" of JPEG data to memory use the function *TMG_JPEG_build_image*.

Internally this function calls the following file read functions:

TMG_write_to_memory(Thandle Hin_image, Thandle Hout_image, ui16 TMG_action)
TMG_write_TIFF_file(Thandle Hin_image, ui16 TMG_action)
TMG_write_EPS_file(Thandle Hin_image, ui16 TMG_action)
TMG_write_TGA_file(Thandle Hin_image, ui16 TMG_action)
TMG_write_BMP_file(Thandle Hin_image, ui16 TMG_action)
TMG_write_RAW_data_file(Thandle Hin_image, ui16 TMG_action)
TMG_JPEG_file_write(Thandle Hin_image)

It is recommended that the *TMG_image_write* function is used, as it provides a simple common interface. However applications linked with static libraries may prefer to use the individual function calls, to reduce the size of the resulting executable.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads, sub-samples, then writes a TIFF file (the whole image in one go):

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_outfilename(hImage, "sky_x2.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
```

```
TMG_IP_subsample(hImage, hOutImage, 2, TMG_RUN);
TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
```

This next example reads a TIFF file in strips and saves it as its mirror image. This strip processing approach would be needed for very large images. Note that a "dummy" read is needed first to determine the height of the image so that the number of strips can be calculated:

```
TMG_image_set_infilename(hImage, "sky.tif");
/* The outfilename parameter gets transferred to hOutImage */
TMG_image_set_outfilename(hImage, "sky_mirror.tif");

/* open the file to read the image height */
TMG_image_set_parameter(hImage, TMG_HEIGHT, 0);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_image_read(hImage, TMG_NULL, TMG_RESET);

/* Now set up the strip processing loop and proceed, 8 lines at time */
TMG_image_set_parameter(hImage, TMG_LINES_THIS_STRIP, 8);
TotalStrips = TMG_image_calc_total_strips(hImage);

for (Strip = 0; Strip < TotalStrips; Strip++) {
   TMG_image_read(hImage, TMG_NULL, TMG_RUN);
   TMG_IP_mirror_image(hImage, hOutImage, TMG_RUN);
   TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
}
```

This final example assumes that a complete image is required in memory in *hFullImage*, after reading and mirroring the image in strips to conserve memory:

```
TMG_image_set_infilename(hImage, "sky.tif");

/* open the file to read the image height */
TMG_image_set_parameter(hImage, TMG_HEIGHT, 0);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_image_read(hImage, TMG_NULL, TMG_RESET);

/* Now set up the strip processing loop and proceed, 8 lines at time */
TMG_image_set_parameter(hFullImage, TMG_LINES_THIS_STRIP, 8);
TotalStrips = TMG_image_calc_total_strips(hFullImage);

for (Strip = 0; Strip < TotalStrips; Strip++) {
   TMG_image_read(hStripImage, TMG_NULL, TMG_RUN);
   TMG_IP_mirror_image(hStripImage, hOutImage, TMG_RUN);
   TMG_image_write(hOutImage, hFullImage, TMG_MEMORY, TMG_RUN);
}
```

BUGS / NOTES

Of the TIFF and BMP formats, only uncompressed image data is supported.

The supported pixel formats are:

TIFF: *TMG_RGB24, TMG_Y8, TMG_Y16, TMG_PALETTED.*
EPS: *TMG_RGB24, TMG_Y8, TMG_PALETTED.*
TGA: *TMG_RGB24, TMG_RGB15, TMG_Y8.*
BMP: *TMG_RGB24, TMG_Y8, TMG_PALETTED.*
RAW: All formats.

SEE ALSO

TMG_image_read, TMG_image_get_infilename,
TMG_image_get_outfilename, TMG_JPEG_file_write.

TMG_IP_crop

USAGE

*Terr TMG_IP_crop(Thandle Hin_image, Thandle Hout_image, i16 *roi, ui16 TMG_action)*

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
roi	"ROI" array with four elements, with #defined element names:
	ASL_ROI_X_START Horizontal start position (0..N).
	ASL_ROI_Y_START Vertical start position (0..N).
	ASL_ROI_X_LENGTH Horizontal width of box.
	ASL_ROI_Y_LENGTH Vertical height of box.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function takes the input image, *Hin_image*, and crops it to the dimensions of the *roi*, to generate the output image *Hout_image*.

When processing in strips, it may well be that one or more of the output strips contains no image data (i.e. they are part of the cropped region). Any downstream TMG function cope with this.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads a TIFF file and crops it to be a multiple of 8 in both dimensions:

```
i16 Roi[ASL_SIZE_2D_ROI];

Roi[ASL_ROI_X_START] = 0;
Roi[ASL_ROI_Y_START] = 0;

TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_outfilename(hImage, "sky_crop.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
Roi[ASL_ROI_X_LENGTH] = (i16) (TMG_image_get_parameter(hImage,
       TMG_WIDTH)/8)*8;
Roi[ASL_ROI_Y_LENGTH] = (i16) (TMG_image_get_parameter(hImage,
       TMG_HEIGHT))/8)*8;
TMG_IP_crop(hImage, hOutImage, Roi, TMG_RUN);
TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_IP_pixel_rep, *TMG_IP_subsample*, *TMG_IP_extract_region*.

TMG_IP_extract_region

USAGE

Terr TMG_IP_extract_region(Thandle Hin_image, Thandle Hout_image, ui32 dwRegionType, ui32 dwX, ui32 dwY, ui32 dwRadius, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
dwRegionType	Type of region to extract. (Currently only support a circle - type 0.)
dwX	Origin of region centre (x).
dwY	Origin of region centre (y).
dwRadius	Radius of circle.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function takes the input image, *Hin_image*, extracts a region from it and places the output image data into *Hout_image*.

The coordinate system has the origin at the top left of the image. The output "image" has its width set to 1 and its width set to the number of pixels extracted.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads a TIFF file and extracts a circular region of data in preparation for some image processing:

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
/* Extract a circular region from the image, 0 = region type circle */
TMG_IP_extract_region(hImage, hImageData, 0, dwTargetOriginX, dwTargetOriginY,
       dwTargetRadius, TMG_RUN);
… /* Image data processing as required */
```

BUGS / NOTES

The only supported region is a circle.

SEE ALSO

TMG_IP_crop.

TMG_IP_filter_3x3

USAGE

*Terr TMG_IP_filter_3x3(Thandle Hin_image, Thandle Hout_image, i16 *array, ui16 TMG_action)*

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
array	3 x 3 filter mask coefficients.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function performs a filtering operation on the image, *Hin_image*, to produce an output image, *Hout_image*. The filter is a 3 x 3 mask function of the following form:

c1 c2 c3
c4 c5 c6
c7 c8 c9, where *array* points to the first element c1.

Output pixels are generated by multiplying each coefficient by the pixel in the same orthogonal position as the coefficient, and then dividing by the sum of the coefficients. Typical applications include smoothing or sharpening an image. Example coefficients are as follows:

1 1 1 1 1 1 1 1 1	Smoothing functioning.
-1 -1 -1 -1 12 -1 -1 -1 -1	Strong edge sharpening.
-1 -1 -1 -1 20 -1 -1 -1 -1	Medium edge sharpening.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how to edge enhance an image:

```
i16 Array[9];
.
Array[0] = -1; Array[1] = -1; Array[2] = -1;
Array[3] = -1; Array[4] = 20; Array[5] = -1;
Array[6] = -1; Array[7] = -1; Array[8] = -1;

TMG_IP_filter_3x3(hInImage, hOutImage, Array. TMG_RUN);
```

BUGS / NOTES

This function only works on grayscale and 24 bit RGB images. (i.e. *TMG_Y8* and *TMG_RGB24*).

SEE ALSO

-

TMG_IP_generate_averages

USAGE

*Terr TMG_IP_generate_averages(Thandle Hin_image, struct tTMG_Averages *psAverages, ui16 TMG_action)*

ARGUMENTS

Hin_image	Handle to the input image.
psAverages	Pointer to a TMG "Averages" structure (see source include file "tmg.h" for full details).
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function takes the input image, *Hin_image*, and calculates the average value for each plane in the image. The results are put into the TMG Averages structure pointed to by *psAverages*. The structure is defined in the source include file "tmg.h".

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how to calculate the average hue, saturation and intensity levels in an image.

```
struct tTMG_Averages sAverages;
…
TMG_image_convert(hYuvImage, hHsiImage, TMG_HSI, 0, TMG_RUN);
TMG_IP_generate_averages(hHsiImage, &sAverages, TMG_RUN);
printf("Average hue = %d\n", (int) sAverages.dwPlane1);
printf("Average sat = %d\n", (int) sAverages.dwPlane2);
printf("Average int = %d\n", (int) sAverages.dwPlane3);
```

BUGS / NOTES

There is only support for TMG image type *TMG_HSI*.

SEE ALSO

TMG_IP_histogram_generate.

TMG_IP_histogram_clear

USAGE

*Terr TMG_IP_histogram_clear(struct tTMG_Histogram *psHistogram)*

ARGUMENTS

psHistogram Pointer to a TMG "Histogram" structure (see source include file "tmg.h" for full details).

DESCRIPTION

This function takes the histogram structure, *psHistogram*, and clears down all structure elements to 0.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment clears down the histogram structure, *psHistogram*:

```
struct tTMG_Histogram *psHistogram;
…
TMG_IP_histogram_clear(psHistogram);
```

BUGS / NOTES

-

SEE ALSO

TMG_IP_histogram_generate, *TMG_IP_histogram_match*, *TMG_IP_histogram_filter*.

TMG_IP_histogram_filter

USAGE

*Terr TMG_IP_histogram_filter(struct tTMG_Histogram *psHistogram, i32 nFilterOrder)*

ARGUMENTS

psHistogram	Pointer to a TMG "Histogram" structure (see source include file "tmg.h" for full details).
nFilterOrder	Filter type. *e.g. TMG_LP_FILTER_ORDER_0.*

DESCRIPTION

This function takes the histogram structure, *psHistogram*, and filters the histogram(s) using the filter specified by *nFilterType*.

TMG_LP_FILTER_ORDER_0 is the only supported option for *nFilterOrder*. This applies the following digital filter algorithm:

Y(n) = (X(n+1) + 2.X(n) + X(n-1)) / 4.

In order to apply higher order filters, simply call the function repeatedly as required.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment generates an HSI image, then generates and filters the histogram for each plane:

```
TMG_image_convert(hYuvImage, hHsiImage, TMG_HSI, 0, TMG_RUN);
TMG_IP_histogram_generate(hHsiImage, psHistogram, TMG_RUN);
TMG_IP_histogram_filter(psHistogram, TMG_FILTER_ORDER_0);
```

BUGS / NOTES

There is only support for TMG image type *TMG_HSI*.

TMG_LP_FILTER_ORDER_0 is the only supported option for *nFilterOrder*.

SEE ALSO

TMG_IP_histogram_generate, *TMG_IP_histogram_match*, *TMG_IP_histogram_clear*.

TMG_IP_histogram_generate

USAGE

*Terr TMG_IP_histogram_generate(Thandle Hin_image, struct tTMG_Histogram *psHistogram, ui16 TMG_action)*

ARGUMENTS

Hin_image	Handle to the input image.
psHistogram	Pointer to a TMG "Histogram" structure (see source include file "tmg.h" for full details).
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function takes the input image, *Hin_image*, and generates a histogram for each plane in the image. The results are put into the TMG Histogram structure pointed to by *psHistogram*. The structure is defined in the source include file "tmg.h".

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment generates an HSI image and then generates a histogram for each plane:

```
struct tTMG_Histogram *psHistogram;
…
TMG_image_convert(hYuvImage, hHsiImage, TMG_HSI, 0, TMG_RUN);
TMG_IP_histogram_generate(hHsiImage, psHistogram, TMG_RUN);
```

BUGS / NOTES

There is only support for TMG image type *TMG_HSI*.

SEE ALSO

TMG_IP_histogram_filter, *TMG_IP_histogram_match*, *TMG_IP_histogram_clear*, *TMG_IP_generate_averages*.

TMG_IP_histogram_match

USAGE

*Terr TMG_IP_histogram_match(struct tTMG_Histogram *psRefHistogram, struct tTMG_Histogram *psInHistogram, i32 nPlane, ui32 *pdwResult)*

ARGUMENTS

psRefHistogram	Pointer to the reference TMG "Histogram" structure.
psInHistogram	Pointer to the input TMG "Histogram" structure.
nPlane	References plane 1, 2 or 3, representing HSI, RGB or YUV 4:2:2 planes.
pdwResult	Pointer to 32 bit unsigned integer which is filled in by the function with the result.

DESCRIPTION

This function takes the input histogram structure, *psInHistogram*, and compares it to the reference histogram, *psRefHistogram*, and gives a percentage match in *pdwResult*. Only one histogram plane, selected by *nPlane*, is compared at a time.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment generates an HSI image, then performs a histogram match of the hue component:

```
TMG_image_convert(hYuvImage, hHsiImage, TMG_HSI, 0, TMG_RUN);
TMG_IP_histogram_generate(hHsiImage, psHistogram, TMG_RUN);
/* Hue match - plane 1 (selects H/S/I) */
TMG_IP_histogram_match(psRefHistogram, psHistogram, 1, pnHueMatch);
printf("Spectrum match = %d\n", *pnHueMatch);
```

BUGS / NOTES

-

SEE ALSO

TMG_IP_histogram_generate, *TMG_IP_histogram_filter*, *TMG_IP_histogram_clear*.

TMG_IP_mirror_image

USAGE

Terr TMG_IP_mirror_image(Thandle Hin_image, Thandle Hout_image, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function takes the input image, *Hin_image,* and mirrors it (lateral inversion), to generate the output image *Hout_image.*

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads a TIFF file, mirrors it and writes it back:

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_outfilename(hImage, "sky_mirrored.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_IP_mirror_image(hImage, hOutImage, TMG_RUN);
TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

-

TMG_IP_pixel_rep

USAGE

Terr TMG_IP_pixel_rep(Thandle Hin_image, Thandle Hout_image, ui16 rep_factor, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
rep_factor	The pixel replication factor.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function takes the input image, *Hin_image*, and replicates the each pixel using the simple integer scaling factor, *rep_factor*, to generate the output image *Hout_image*.

This function can be used to "zoom" an image for display purpose as long as there is sufficient memory.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads a TIFF file, expands it by two and writes it back:

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_outfilename(hImage, "sky_x2.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_IP_pixel_rep(hImage, hOutImage, 2, TMG_RUN);
TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
```

BUGS / NOTES

This function only supports simple binary scaling, that is *rep_factor* must be 1, 2, 4, 8 etc.

SEE ALSO

TMG_IP_subsample.

TMG_IP_rotate_image

USAGE

Terr TMG_IP_rotate_image(Thandle Hin_image, Thandle Hout_image, ui32 dwDegrees)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
dwDegrees	Degrees by which to rotate the image. Must be one of 0, 90, 180 or 270.

DESCRIPTION

This function takes the input image, *Hin_image*, and rotates it by the angle *dwDegrees*. This function requires the whole image to be present in *Hin_image* (i.e. it cannot operate in strips).

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads a TIFF file, rotates it and writes it back out as a TIFF file.

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_outfilename(hImage, "sky_r90.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_IP_rotate_image(hImage, hOutImage, 90);
TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

-

TMG_IP_subsample

USAGE

Terr TMG_IP_subsample(Thandle Hin_image, Thandle Hout_image, ui16 sub_factor, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
sub_factor	The sub-sample factor.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function takes the input image, *Hin_image*, and sub-samples it using a simple integer scaling factor, *sub_factor*, to generate the output image *Hout_image.*

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads a TIFF file, sub-samples it by two and writes it back:

```
TMG_image_set_infilename(hImage, "sky.tif");
TMG_image_set_outfilename(hImage, "sky_x2.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_IP_subsample(hImage, hOutImage, 2, TMG_RUN);
TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
```

BUGS / NOTES

This function only supports simple binary sub-sampling, that is *sub_factor* must be 1, 2, 4, 8 etc. Also the input image must be exactly divisible by the sub-sampling factor.

SEE ALSO

TMG_IP_pixel_rep, *TMG_IP_crop*.

TMG_IP_threshold_grayscale

USAGE

Terr TMG_IP_threshold_grayscale(Thandle Hin_image, Thandle Hout_image, ui8 white_level, ui8 black_level, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to an input image.
Hout_image	Handle to an output image.
white_level	White level threshold.
black_level	Black level threshold.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This processing function accepts a grayscale image (of type *TMG_Y8*) and performs a white level and black level threshold operation on it. All pixels in the image with a value greater than or equal to *white_level* are set to white (value 255), and all pixels with values less than *black_level* are set to black (value 0). The output image is a grayscale image. This function can be useful for mapping background gray levels to white or black and could be regarded as a simple type of luma keying.

For a single threshold, resulting in an output image with only black (0) or white (255) pixels, *white_level* and *black_level* would be set equal to one and other.

Applications include pre-processing an image prior to JPEG compression (to improve the compression ratio), or prior to printing to obtain a better looking image.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment reads a TIFF file, thresholds it and writes it back:

```
TMG_image_set_infilename(hImage, "car.tif");
TMG_image_set_outfilename(hImage, "car_out.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
/* All gray levels less than 20 will become 0 and all */
/* white levels greater than 200 will become 255.     */
TMG_IP_threshold_grayscale(hImage, hOutImage, 20, 200, TMG_RUN);
TMG_image_write(hOutImage, TMG_NULL, TMG_TIFF, TMG_RUN);
```

BUGS / NOTES

This function only works for grayscale images.

There are no known bugs.

SEE ALSO

-

TMG_JPEG_buffer_read

USAGE

*Terr TMG_JPEG_buffer_read(Thandle Hjpeg_image, ui8 *pbData, ui32 dwBytesData)*

ARGUMENTS

Hjpeg_image	A handle to a JPEG image.
pbData	Pointer to the buffer containing data in JPEG interchange format.
dwBytesData	The amount of data in the buffer.

DESCRIPTION

This function reads a full JPEG image from the buffer, *pbData*, into *Hjpeg_image* in one go (i.e. not in strips). The internal image parameter *lines_this_strip* is set to height to indicate that the whole image is present (as compressed JPEG data). The amount of memory allocated for the compressed data itself is set to *dwBytesData* (this is a convenient number to use, although typically around 700 bytes more than is necessary. Using this number, to allocate memory, saves a memory to memory copy that would be required if memory usage was to be optimised.)

This function is useful when sending JPEG "files" over network links – see the example below.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how the function could be used to receive a JPEG image over a network:

```
/* Read the data from a remote machine using "SCA_Recv" (comms read fn) */
dwStatus = SCA_Recv(hConnection, &dwCommand, &pbData, &dwSize, 5000);
if ( (dwStatus == 0) && (pbData != NULL) )  /* Received data OK? */
{
   TMG_JPEG_buffer_read(hJPEGImage, pbData, dwSize);
   /* Decompress to RGB (as opposed to YUV422) */
   TMG_image_set_parameter(hSrcImage, TMG_PIXEL_FORMAT, TMG_RGB24);
   /* If grayscale this will automatically change to Y8 in the decompress
       function. */
   TMG_JPEG_decompress(hJPEGImage, hSrcImage, TMG_RUN);
   /* Convert to DIB */
   TMG_image_convert(hSrcImage, hDIBImage, TMG_BGR24, TMG_IS_DIB, TMG_RUN);
   TMG_display_image(hDisplay, hDIBImage, TMG_RUN);
}
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_buffer_write, *TMG_JPEG_file_read*.

TMG_JPEG_buffer_write

USAGE

*Terr TMG_JPEG_buffer_write(Thandle Hjpeg_image, ui8 *pbData, ui32 *pdwCount, ui16 TMG_action)*

ARGUMENTS

Hjpeg_image	A handle to a JPEG image.
pbData	Pointer to target buffer (user allocated).
pdwCount	Pointer to a 32 bit unsigned word, filled in by the function, indicating the total amount of data written.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* to abort.

DESCRIPTION

This function writes the JPEG image, *Hjpeg_image*, to a buffer in JPEG/JFIF "file" format. It may be called as part of a strip processing loop, in which case it will automatically write the amount of compressed data produced so far by the compression process. In this instance, the internal image parameter *lines_this_strip*, would be used as an indicator of when the last strip of compressed data is written so that an EOI (End of Image) marker can be appended to the end of the data stream.

If the whole compressed image is written as one strip, *lines_this_strip* should be set to the image height, in which case the EOI marker will again be automatically appended. Alternatively, if *TMG_JPEG_buffer_write* is called with *TMG_RESET*, the EOI marker is immediately written to the file.

This function is useful when sending JPEG images over network connections.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how the function could be used to send a JPEG image over a network:

```
TMG_JPEG_compress(hImage, hJpegImage, TMG_RUN);
TMG_JPEG_buffer_write(hJpegImage, pbData, &dwSize, TMG_RUN);
SCA_Send(hConnection, CC_NET_IMAGE, pbData, dwSize, TIMEOUT_5SECS);
/* "SCA_Send" is an example network send command */
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_buffer_read, *TMG_JPEG_file_write*.

TMG_JPEG_build_image

USAGE

Terr TMG_JPEG_build_image(Thandle Hin_image, Thandle Hout_image, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input JPEG image.
Hout_image	Handle to the output JPEG image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function is designed to be used as the last function in a strip processing loop to build a full JPEG image in memory. For example the source raw image may be compressed 8 lines at a time and be 64 lines high - in this case *TMG_JPEG_build_image* would be called 8 times (in the strip processing loop) and the resulting complete JPEG image contained in *Hout_image*.

This function will also optimise the memory usage of the JPEG image using one of the features of the function *TMG_JPEG_sequence_build*.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment JPEG compresses the image *Hin_image*, which contains is a full image in memory (and has its internal parameter *lines_this_strip* set to 8).

```
TMG_image_set_parameter(Hin_image, TMG_LINES_THIS_STRIP, 8);
total_strips = (ui16) TMG_image_calc_total_strips(Hin_image);
for (strip = 0; strip < total_strips; strip += 1) {
   TMG_image_read(Hin_image, Hstripped_image, TMG_RUN);
   TMG_JPEG_compress(Hstripped_image, Htemp_image, TMG_RUN);
   TMG_JPEG_build_image(Htemp_image, Hjpeg_image, TMG_RUN);
}
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_file_write, *TMG_JPEG_file_open*, *TMG_JPEG_file_read*, *TMG_JPEG_sequence_build*.

TMG_JPEG_compress

USAGE

Terr TMG_JPEG_compress(Thandle Himage, Thandle Hjpeg_image, ui16 TMG_action)

ARGUMENTS

Himage	Handle to a raw (uncompressed) image.
Hjpeg_image	Handle to a JPEG image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* to abort.

DESCRIPTION

This function compresses an image (or strip) using the JPEG baseline compression algorithm. When the function is called with *TMG_action* set to *TMG_RUN*, raw image data is read from *Himage*, and compressed JPEG data written to *Hjpeg_image*. The strip size is determined by the *lines_this_strip* parameter of *Himage* (set using *TMG_image_set_parameter*). If the function is called with *TMG_action* set to *TMG_RESET* the compression process is aborted and local static (internal) variables are reset. *TMG_RESET* is rarely needed.

Its recommended that images are compressed a strip at a time (set *lines_this_strip* to 8), because this function uses several intermediary images internally.

This function is called by *TMG_JPEG_compress_image_to_image*.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

This function is slightly faster if the image to be compressed has a width which is exactly divisible by 16 for colour images and 8 for grayscale images, and a height that is exactly divisible by 8.

The JPEG images are generated using the "default" Huffman tables as suggested in the JPEG specification.

SEE ALSO

TMG_JPEG_decompress, *TMG_JPEG_compress_image_to_image*, *TMG_JPEG_set_Quality_factor*.

TMG_JPEG_compress_image_to_image

USAGE

Terr TMG_JPEG_compress_image_to_image(Thandle Himage, Thandle Hjpeg_image, ui16 in_format, ui16 out_format)

ARGUMENTS

Himage	Handle to raw image.
Hjpeg_image	Handle to JPEG image.
in_format	*TMG_MEMORY* or *TMG_FILE*.
out_format	*TMG_MEMORY* or *TMG_FILE*.

DESCRIPTION

This is a convenient wrapper function for *TMG_JPEG_compress* that compresses a complete image using the JPEG baseline compression algorithm. Raw image data is read from *Himage*, compressed and written to *Hjpeg_image*. If the input format, *in_format*, is set to *TMG_MEMORY*, raw image data is read from memory (from *Himage*). If *in_format* is set to *TMG_FILE*, it is read from the file associated with *Himage* (i.e. set using *TMG_image_set_infilename*). Similarly, if the output format, *out_format*, is set to *TMG_MEMORY*, compressed data is written to memory (in *Hjpeg_image*). If *out_format* is set to *TMG_FILE* it is written directly to the JPEG file referenced by *Hjpeg_image* (i.e. set using *TMG_image_set_outfilename*). If the *lines_this_strip* parameter of *Himage* is less than the total image height, then compression is performed in strips. The *lines_this_strip* parameter is set using *TMG_image_set_parameter*.

Its recommended that images are compressed a strip at a time (set *lines_this_strip* to 8), because this function uses several intermediary images internally.

This function is a convenient way of compressing from file to file with just one call.

The output memory used by *Hjpeg_image* is optimized to its exact requirements.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

This function is slightly faster if the image to be compressed has a width which is exactly divisible by 16 for colour images and 8 for grayscale images, and a height that is exactly divisible by 8.

The JPEG images are generated using the "default" Huffman tables as suggested in the JPEG specification.

SEE ALSO

TMG_JPEG_decompress_image_to_image, *TMG_JPEG_compress*, *TMG_JPEG_set_Quality_factor*.

TMG_JPEG_decompress

USAGE

Terr TMG_JPEG_decompress(Thandle Hjpeg_image, Thandle Himage, ui16 TMG_action)

ARGUMENTS

Hjpeg_image	Handle to compressed JPEG image.
Himage	Handle to raw (uncompressed) image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* to abort.

DESCRIPTION

This function decompresses a single image strip using the JPEG baseline decompression algorithm. If the function is called with *TMG_action* set to *TMG_RUN*, compressed image data is read from *Hjpeg_image*, and raw data written to *Himage*. The strip size is determined by the *lines_this_strip* parameter of *Hjpeg_image*. If the function is called with *TMG_action* set to *TMG_RESET*, the decompression is aborted and internal static variables are reset.

For colour images, the JPEG image may be decompressed to YUV 4:2:2 data or RGB data depending on the pixel format set in *Himage* (see *TMG_image_set_parameter* with *TMG_PIXEL_FORMAT*). By default the decompressed image will have the pixel format *TMG_RGB24*, but if the pixel format is set to *TMG_YUV422* prior to calling this function, the pixel format of the decompressed image will be *TMG_YUV422*.

Its recommended that images are compressed a strip at a time (set *lines_this_strip* to 8 using *TMG_image_set_parameter*), because this function uses several intermediary images internally.

This function is called by *TMG_JPEG_decompress_image_to_image*.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

Only JPEG images with the "default" Huffman tables as suggested in the JPEG specification are supported.

SEE ALSO

TMG_JPEG_compress, *TMG_JPEG_decompress_image_to_image*.

TMG_JPEG_decompress_image_to_image

USAGE

Terr TMG_JPEG_decompress_image_to_image(Thandle Hjpeg_image, Thandle Himage, ui16 in_format, ui16 out_format)

ARGUMENTS

Hjpeg_image	Handle to compressed image.
Himage	Handle to raw image.
in_format	*TMG_MEMORY* or *TMG_FILE.*
out_format	*TMG_MEMORY* or a file type: *TMG_TIFF*, *TMG_TGA*, *TMG_EPS*, *TMG_BMP* etc.

DESCRIPTION

This function decompresses a complete image using the JPEG baseline compression algorithm. Compressed image data is read from *Hjpeg_image*, and raw image data written to *Himage.* If the parameter, *in_format*, is set to *TMG_MEMORY*, compressed image data is read directly from memory (referenced by *Hjpeg_image*). If *in_format* is set to *TMG_FILE,* the JPEG data is read from the JPEG/JFIF file associated with *Hjpeg_image*. Similarly, if the parameter, *out_format*, is set to *TMG_MEMORY*, raw image data is written to memory (in *Himage*). If *out_format* is set to a file type, for example *TMG_TIFF*, it is written directly to the file referenced by *Himage* in that format. If the *lines_this_strip* parameter of *Hjpeg_image* is less than the total image height, the decompression is performed in strips.

When decompressing colour images to memory, the output pixel format may be YUV 4:2:2 or RGB depending on the pixel format set in *Himage* (see *TMG_image_set_parameter* with *TMG_PIXEL_FORMAT*). By default the decompressed image will have the pixel format *TMG_RGB24*, but if the pixel format is set to *TMG_YUV422* prior to calling this function, the pixel format of the decompressed image will be *TMG_YUV422*.

Its recommended that images are compressed a strip at a time (set *lines_this_strip* to 8 using *TMG_image_set_parameter*), because this function uses several intermediary images internally.

This function is a convenient way of decompressing from file to file with just one call.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

Only JPEG images with the "default" Huffman tables as suggested in the JPEG specification are supported.

SEE ALSO

TMG_JPEG_decompress, *TMG_JPEG_compress_image_to_image*.

TMG_JPEG_file_close

USAGE

Terr TMG_JPEG_file_close(Thandle Himage)

ARGUMENTS

Himage A handle to a JPEG image.

DESCRIPTION

This function is used to close a JPEG file that has previously been opened using *TMG_JPEG_file_open*. It is rarely needed because the JPEG file will be closed by the functions that access it. However if the (decompression) process is aborted it will be necessary for the application program to close the file itself.

The example below shows a situation in which the JPEG image is not decompressed, but simply examined, which requires the use of this function.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment uses *TMG_JPEG_file_open* to see what type of image it is. It then needs to use *TMG_JPEG_file_close* to close the file. *TMG_JPEG_file_read* could be used but that would be slower and have to allocated memory for the JPEG data.

```
ui16 PixelFormat;

TMG_image_set_infilename(hJPEGImage, "car.jpg");
TMG_JPEG_file_open(hJPEGImage);
PixelFormat = (ui16) TMG_image_get_parameter(hJPEGImage, TMG_PIXEL_FORMAT);
if ( (PixelFormat == TMG_YUV422) || (PixelFormat == TMG_RGB24) )
   printf("Its a colour image");
else if (PixelFormat == TMG_Y8)
   printf("Its a grayscale image");

TMG_JPEG_file_close(hJPEGImage);
```

BUGS / NOTES

For use with decompression directly from file, this function is only supported when using Crunch (hardware) JPEG decompression. It is not supported when using TMG software JPEG decompression - in this mode the JPEG file must be read into memory first.

The function *CRUNCH_decompress* or *CRUNCH_decompress_image_to_image* will close the file automatically after it has been decompressed. However if it is not decompressed (perhaps because the operation was aborted), the application program should call *TMG_JPEG_file_close* directly.

The Huffman tables are currently not read in and "default" Huffman tables as suggested in the JPEG specification are always used.

SEE ALSO

TMG_JPEG_file_open, TMG_image_get_infilename, TMG_image_get_outfilename, TMG_JPEG_file_read.

TMG_JPEG_file_open

USAGE

Terr TMG_JPEG_file_open(Thandle Himage)

ARGUMENTS

Himage A handle to a JPEG image.

DESCRIPTION

This function reads all the JPEG header information from a JPEG file, referenced by *Himage,* but does not read the JPEG data itself. It leaves an internal file pointer (internal to *Himage)* pointing at the JPEG data for use by other processing functions. This can be useful when memory is limited.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section. See also the example for *TMG_JPEG_file_close*.

BUGS / NOTES

This function is only supported when using Crunch (hardware) JPEG decompression. It is not supported when using TMG software JPEG decompression - in this mode the JPEG file must be read into memory first.

The function *CRUNCH_decompress* or *CRUNCH_decompress_image_to_image* will close the file automatically after it has been decompressed. However if it is not decompressed, the application program should call *TMG_JPEG_file_close* directly.

The Huffman tables are currently not read in and "default" Huffman tables as suggested in the JPEG specification are always used.

SEE ALSO

TMG_image_get_infilename, TMG_image_get_outfilename, TMG_JPEG_file_read, TMG_JPEG_file_close, TMG_JPEG_sequence_calc_length

TMG_JPEG_file_read

USAGE

Terr TMG_JPEG_file_read(Thandle Hjpeg_image)

ARGUMENTS

Hjpeg_image A handle to a JPEG image.

DESCRIPTION

This function reads a full JPEG image from file into *Hjpeg_image* in one go (i.e. not in strips). The internal image parameter *lines_this_strip* is set to height to indicate that the whole image is present (as compressed JPEG data). Note that this function does not optimise the amount of memory it uses for the JPEG data – it actually allocates the same amount of memory as would be used in a raw image. This is a speed optimisation at the detriment of memory efficiency (the amount of JPEG data is not known in advance). For memory optimisation, either the memory can be allocated and locked in advance by the application (see *TMG_image_set_ptr*), or more simply, the function *TMG_JPEG_sequence_build* can be used.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_buffer_read, TMG_image_read, TMG_JPEG_file_write, TMG_image_get_infilename, TMG_image_get_outfilename.

TMG_JPEG_file_write

USAGE

Terr TMG_JPEG_file_write(Thandle Hjpeg_image, ui16 TMG_action)

ARGUMENTS

Himage	A handle to a JPEG image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* to abort.

DESCRIPTION

This function writes the JPEG image, *Hjpeg_image*, to a in JPEG/JFIF file. It may be called as part of a strip processing loop, in which case it will automatically write the amount of compressed data produced so far by the compression process. In this instance, the internal image parameter *lines_this_strip*, would be used as an indicator of when the last strip of compressed data is written so that an EOI (End of Image) marker can be appended to the end of the data stream.

If the whole compressed image is written as one strip, *lines_this_strip* should be set to the image height, in which case the EOI marker will again be automatically appended. Alternatively, if *TMG_JPEG_file_write* is called with *TMG_RESET*, the EOI marker is immediately written to the file.

Note that *TMG_JPEG_file_read* sets *lines_this_strip* to the image height.

Normally this function would not be used, but the wrapper function *TMG_image_write* used instead.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_buffer_write, TMG_image_write, TMG_JPEG_file_read, TMG_image_get_infilename, TMG_image_get_outfilename.

TMG_JPEG_image_create

USAGE

Terr TMG_JPEG_image_create()

ARGUMENTS

None.

DESCRIPTION

This function creates a *Timage* structure and *Tjpeg* structure which is pointed to from the *Timage* structure, and returns a handle to that *Timage* structure. It also performs some initialization - that is characters strings are set to '\0' and the data pointers set to *NULL*. The variable *lines_this_strip* is set to 8. Note that no memory is created for the JPEG data itself - this is performed by TMG functions. The *Tjpeg* structure can hold single or multiple (motion) JPEG encoded image(s) or a strip of a single image.

Note that a JPEG image is a superset of an ordinary image. Note that a JPEG image can hold either JPEG data or raw image data (but not both unless it is the same image). See the file "tmg.h" for the actual structure definitions.

RETURNS

On success a valid handle is returned in the lower 16 bits of the return value (the upper 16 bits will be 0). On failure an error code will be returned in the upper 16 bits as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code creates an image and gets a handle to it:

```
Thandle hJPEGImage; /* Handle to a JPEG image structure */

if ( ASL_is_err(hImage = TMG_JPEG_image_create() )
   printf("Failed to create a JPEG image");
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_image_destroy, *TMG_image_create*.

TMG_JPEG_sequence_build

USAGE

Terr TMG_JPEG_sequence_build(Thandle Hin_image, Thandle Hout_image)

ARGUMENTS

Hin_image	Handle to the input image or *TMG_NULL*.
Hout_image	Handle to the output image.

DESCRIPTION

This function is builds a JPEG sequence of images. The input images are sequentially added to the JPEG data stream in *Hout_image*. Restart markers are inserted between each frame (or scan in JPEG terminology) in the data stream. Note that it is assumed that each successive image is of the same type and has the same width and height.

This function can also be used to optimise the memory usage by the compressed data. When memory is allocated for a JPEG image, an excess is allocated, because the precise requirements cannot be predicted in advance. If *Hin_image* is set to *TMG_NULL*, the memory allocation for *Hout_image* will be re-done to match precisely its requirements. Obviously this can only be done when *Hout_image* contains a valid JPEG image.

For a motion JPEG sequence acquisition of tens of frames, note that several megabytes may typically be used.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the Crunch Library Programmer's Manual for example code and the file "seq.c" available with the Snapper SDK.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_sequence_calc_length.

TMG_JPEG_sequence_calc_length

USAGE

Terr TMG_JPEG_sequence_calc_length(Thandle Hjpeg_image)

ARGUMENTS

Hjpeg_image Handle to JPEG compressed image sequence.

DESCRIPTION

This function calculates the sequence length of a motion JPEG file after the function *TMG_JPEG_file_open* has been called. It is designed to be used in conjunction with the Crunch JPEG hardware replay functions (*CRUNCH_sequence_replay*).

The number of frames is calculated by scanning the file for the number of restart markers in the JPEG data. (Restart markers are used to signify the end of a frame in a sequence of JPEG frames when stored in a single JPEG/JFIF file.)

If *TMG_JPEG_file_read* is used (instead of *TMG_JPEG_file_open*), then the number of frames of JPEG data is automatically calculated and stored in *Hjpeg_image,* as part of the operation of *TMG_JPEG_file_read*.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment uses open a (motion) JPEG file and calculates the sequence length ready for replay:

```
TMG_image_set_infilename(hJPEGImage, "sequence.jpg");
TMG_JPEG_file_open(hJPEGImage);
/* Now fill in the internal TMG image parameter "num_frames" */
TMG_JPEG_sequence_calc_length(hJPEGImage);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_file_read, *TMG_JPEG_file_open*, *TMG_JPEG_sequence_set_start_frame*.

TMG_JPEG_sequence_extract_frame

USAGE

Terr TMG_JPEG_sequence_extract_frame(Thandle Hin_image, Thandle Hout_image, ui32 frame)

ARGUMENTS

Hin_image	Handle to JPEG compressed image sequence.
Hout_image	Handle to output (single frame) JPEG image.
frame	The number of the frame to extract (from 1..N).

DESCRIPTION

This function copies a single JPEG image from a JPEG sequence in *Hin_image* to *Hout_image*. The frame number to copy is given by *frame_num*.

Hout_image must be a JPEG image (i.e. created using *TMG_JPEG_image_create*). Any JPEG image memory in *Hout_image* will be destroyed and new memory allocated to precisely match the size of the extracted frame (unless the memory is locked).

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment extracts the 8th frame of a 16 frame motion JPEG sequence and saves it as an individual file:

```
TMG_image_set_infilename(hJPEGImage, "sequence.jpg");
TMG_JPEG_file_read(hJPEGImage);
.
hOutImage = TMG_JPEG_image_create();
TMG_JPEG_sequence_extract_frame(hJPEGImage, hOutImage, 8);
TMG_image_set_outfilename(hOutImage, "frame08.jpg");
TMG_JPEG_file_write(hOutImage, TMG_RUN);
```

BUGS / NOTES

The starting frame (set using *TMG_JPEG_sequence_set_start_frame*) gets changed as a side effect of using this function.

This function only works on JPEG sequence files stored in memory - i.e. the JPEG file must have been read in using *TMG_JPEG_file_read*.

SEE ALSO

TMG_JPEG_file_read, *TMG_JPEG_file_open*, *TMG_JPEG_sequence_set_start_frame*.

TMG_JPEG_sequence_set_start_frame

USAGE

Terr TMG_JPEG_sequence_set_start_frame(Thandle Hjpeg_image, ui32 start_frame)

ARGUMENTS

Hjpeg_image	Handle to JPEG compressed image sequence.
start_frame	Desired starting frame.

DESCRIPTION

This function sets the starting frame ready for replaying a sequence using the function *CRUNCH_sequence_replay* (JPEG hardware acceleration function). It will work on *Hjpeg_image* whether the image is in memory or on file. It works by scanning the file looking to the inter-frame (restart) markers.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the Crunch Library Programmer's Manual for example code and the file "seq.c" available with the Snapper SDK.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_sequence_calc_length.

TMG_JPEG_set_image

USAGE

Terr TMG_JPEG_set_image(Thandle Himage, Thandle Hjpeg_image)

ARGUMENTS

Himage	Handle to the input image.
Hjpeg_image	Handle to the output JPEG image.

DESCRIPTION

This function is sets up the JPEG parameters in *Hjpeg_image* based on the raw image parameters in *Himage*. This function is rarely needed in a user application, but a novel use is given in the example below.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment is an example of how to "reconstruct" a JPEG file ready for decompression from only the JPEG data, the image width, height and (original) Q factor:

```
/* The image is a colour 256 x 256 image */
/* NumBytes contains the amount of the JPEG data */
/* pJPEGData points to the JPEG data */
hJPEGImage = TMG_JPEG_image_create();
TMG_image_set_parameter(hJPEGImage, TMG_WIDTH, 256);
TMG_image_set_parameter(hJPEGImage, TMG_HEIGHT, 256);
TMG_image_set_parameter(hJPEGImage, TMG_PIXEL_FORMAT, TMG_RGB24);
TMG_JPEG_set_image(hJPEGImage, hJPEGImage);
TMG_JPEG_set_Quality_factor(hJPEGImage, 32);
TMG_JPEG_make_Q_tables(hJPEGImage);
TMG_JPEG_set_default_H_tables(hJPEGImage);
TMG_image_set_ptr(hJPEGImage, TMG_JPEG_DATA, pJPEGData);
TMG_image_set_parameter(hJPEGImage, TMG_JPEG_NUM_BYTES_DATA, NumBytes);
/* hJPEGImage is now a valid JPEG image */
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_image_create.

TMG_JPEG_set_Quality_factor

USAGE

Terr TMG_JPEG_set_Quality_factor(Thandle Hjpeg_image, ui16 Q_factor)

ARGUMENTS

Hjpeg_image	Handle to JPEG image.
Q_factor	An integer representing image quality after compression from 1 to 100.

DESCRIPTION

Sets the JPEG quality factor for compression. This parameter is only used in compression. It represents the quality of the compressed image and is therefore related to compression ratio, i.e. if a high quality factor is set, the compressed image quality will be high, and the compression ratio not as high as it would be if a lower quality factor was used. The default quality factor is 32, which results in the coefficients in the standard Q (Quantization) tables, as defined in the JPEG specification, being halved. i.e. the Quality factor is normalised to 16. This is common practice in JPEG software packages. The range of the quality factor is from 1 to 400, although numbers above 100 will give very little improved quality, and low (typically 6:1) compression ratios. Using the default Quality factor of 32 results in an image which is "usually nearly indistinguishable from the original" (quote from the JPEG Specification).

The default quality factor of 32 is used if this function (or *TMG_JPEG_set_Quantization_factor*) is not called. Note that a default quality factor of 32 is equivalent to a default quantization factor of 50.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code sets the quantization factor:

```
TMG_JPEG_set_Quality_factor(Himage, 20);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_set_Quantization_factor.

TMG_JPEG_set_Quantization_factor

USAGE

Terr TMG_JPEG_set_Quantization_factor(Thandle Hjpeg_image, ui16 Q_factor)

ARGUMENTS

Hjpeg_image	Handle to JPEG image.
Q_factor	JPEG quantization factor.

DESCRIPTION

Sets the JPEG quantization factor for compression. This parameter is only used in compression. It is used to generate the quantization table which defines the number of quantization levels at which the luminance and chrominance frequencies are quantized to. In simple terms, a higher quantization factor means lower image quality and vice-versa. The quantization factor is normalised to 50, which is consistent with other JPEG compression systems. This means that when set to 50, the luminance and chrominance quantization tables are identical to those in the JPEG Specification.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code sets the quantization factor:

```
TMG_JPEG_set_Quantization_factor(Hbase, 100);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_JPEG_set_Quality_factor.

TMG_LUT_apply

USAGE

Terr TMG_LUT_apply(Thandle Hin_image, Thandle Hout_image, Thandle hLUT, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
hLUT	Handle to a *TMG LUT* structure.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function performs a LUT operation on *Hin_image* to produce an output image *Hout_image*. This function can be used to enhance an image through the use of brightness, contrast, gamma and individual colour controls (for colour balancing). The function *TMG_LUT_generate* is used to generate the LUT before this function is called (and *TMG_LUT_create* to create it prior to generation).

The functions accepts the following image types: *TMG_YUV422, TMG_Y8, TMG_RGB24, TMG_RGB16, TMG_RGB15* and *TMG_RGB8*. If the individual colour intensities have been changed from their default values, the function operates slower with a YUV image than an RGB image. This is because it must convert from YUV colour space to RGB colour space in order to use the LUTs. Therefore it is usually better to operate on the RGB image - i.e. convert to (or use) a RGB image before using *TMG_LUT_apply*.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_LUT_create, *TMG_LUT_generate*.

TMG_LUT_create

USAGE

Terr TMG_LUT_create(ui32 num_elements, ui16 element_size)

ARGUMENTS

num_elements	The size of the look up table in terms of the number of elements.
element_size	The size of each element in bytes - must be either 1 or 2.

DESCRIPTION

This function creates a *Ttmg_LUT* structure by the use of malloc, and returns a handle to that structure. The LUT structure contains four independent LUTs - one for luma (grayscale data) and one for red, green and blue image data. Separate LUTs for red, green and blue allow colour balancing as well as overall brightness/contrast variation.

The size of the LUT is determined by the two parameters *num_elements* and *element_size*. *num_elements* must be less than or equal to 256 if *element_size* is 1, or less than or equal to 65536 if *element_size* is 2.

The handle to this structure is used by the LUT function *TMG_LUT_apply*.

RETURNS

On success a valid handle is returned in the lower 16 bits of the return value (the upper 16 bits will be 0). On failure an error code will be returned in the upper 16 bits as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code creates a LUT and gets a handle to it:

```
Thandle hLUT;        /* Handle to LUT */

/* Create a LUT of 256 elements (8 bits in, 8 bits out) */
if ( ASL_is_err(hLUT = TMG_LUT_create(256, 1) )
   printf("Failed to create LUT");
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_LUT_destroy, *TMG_LUT_apply*, *TMG_LUT_generate*, *TMG_LUT_get_ptr*.

TMG_LUT_destroy

USAGE

Terr TMG_LUT_destroy(Thandle hLUT)

ARGUMENTS

hLUT Handle to a TMG LUT structure or *TMG_ALL_HANDLES*.

DESCRIPTION

Destroys a *Ttmg_LUT* structure by freeing all the memory associated with that structure.

If the parameter *TMG_ALL_HANDLES* is used, all previously created LUT structures are destroyed and their associated handles freed.

TMG_image_destroy(*TMG_ALL_HANDLES*) will destroy all TMG LUT structures by calling *TMG_LUT_destroy* for all LUT handles. This is a convenient way of destroying everything with just one function call.

RETURNS

ASL_OK.

EXAMPLES

The following code destroys a previously created LUT:

```
Thandle hLUT;      /* Handle to LUT */
.
TMG_LUT_destroy(hLUT);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_LUT_create, *TMG_image_destroy*.

TMG_LUT_generate

USAGE

Terr TMG_LUT_generate(Thandle hLUT, i16 brightness, i16 contrast, i16 gamma, i16 ri, i16 gi, i16 bi)

ARGUMENTS

brightness	Desired brightness setting (from 0 to 200, default 100).
contrast	Desired contrast setting (from 0 to 200, default 100).
gamma	Desired gamma setting (from 0 to 400, default 100).
ri	Desired red intensity (from 0 to 200, default 100).
gi	Desired green intensity (from 0 to 200, default 100).
bi	Desired blue intensity (from 0 to 200, default 100).

DESCRIPTION

This function generates the actual LUT data in the LUT structure referenced by *hLUT*. This function must be called before the function *TMG_LUT_apply* is used to perform a LUT operation. If called with the default values, the resulting LUTs will contain straight lines and therefore have no effect on the image. If the LUT function is going to operate on a grayscale image, then *ri*, *gi* and *bi* have no effect and should be set to zero.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

See the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_LUT_create, *TMG_LUT_apply*, *TMG_LUT_get_ptr*.

TMG_LUT_get_ptr

USAGE

*void *TMG_LUT_get_ptr(Thandle hLUT, ui16 colour)*

ARGUMENTS

hLUT	Handle to a TMG LUT structure
colour	Colour plane of the LUT required. One of *TMG_GRAY, TMG_RED, TMG_GREEN* or *TMG_BLUE*.

DESCRIPTION

This function returns the pointer to the actual look up table data for that particular colour. This can be useful, if for example, the *TMG_LUT_generate* function was being used to generate data for another function - for example programming hardware LUTs.

The returned pointer type must be cast the pointer type that reflects the size of the data in the LUT. If the LUT has an element size of 1 (see *TMG_LUT_create*), then the result from *TMG_LUT_get_ptr* should be cast to *ui8**. If the element size is 2, then the result should be cast to *IM_UI16**. (For all operating systems apart from Windows 3.1, *IM_UI16** is equivalent to *ui16**.)

RETURNS

A pointer to the LUT data on success, otherwise *NULL*.

EXAMPLES

The following code gets the pointer from a LUT structure:

```
ui8* pLUT;        /* Pointer to 256 element LUT. */
.
hLUT = TMG_LUT_create(256, 1);
.
pLUT = (ui8*) TMG_LUT_get_ptr(hLUT);
```

See also the extended examples in the "Sample Applications" section.

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_LUT_create, *TMG_LUT_generate*.

TMG_SPL_2fields_to_frame

USAGE

Terr TMG_SPL_2fields_to_frame(Thandle Himage1, Thandle Himage2, Thandle Hout_image, ui16 TMG_action)

ARGUMENTS

Himage1	Handle to an input image - field 1
Himage2	Handle to an input image - field 2
Hout_image	Handle to the output image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function is similar to the function *TMG_SPL_field_to_frame*, but takes two input images which represent field 1 and field 2 of a complete frame. These images are interlaced to provide a complete frame, *Hout_image*. This function is likely to be used in conjunction with the Snapper sequence acquisition mode, in which fields are acquired in a real-time sequence to separate images. This function can be used to re-construct full frames.

If the internal image flag *TMG_HALF_ASPECT* is set, the function uses the parameter *TMG_FIELD_ID* to determine which field is which, so it can correctly interlace them. If this flag is not set, then *Himage1* will be assumed to be field 1 and written to lines 1, 3, 5 etc in the output image, and *Himage2* will be written to lines 2, 4, 6 etc.

For parameter and flag information, see *TMG_image_set_parameter* and *TMG_image_set_flags* respectively.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how to reconstruct a frame sequence from a sequence of fields:

```
/* hImages is an array of fields captured in a sequence from
 * Snapper-24.  This code re-interlaces them and displays the
 * resulting frames.
 */
for (n = 0; n < SequenceLength; n += 2) {
   TMG_SPL_2fields_to_frame(hImages[n], hImages[n+1], hTempImage, TMG_RUN);
   TMG_convert_to_RGB16(hTempImage, hOutImage, TMG_RUN);
   TMG_display_image(hDisplay, hOutImage, TMG_RUN);
}
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_SPL_field_to_frame.

TMG_SPL_Data32_to_Y8

USAGE

Terr TMG_SPL_Data32_to_Y8(Thandle Hin_image, Thandle Hout_image, ui16 wShiftRight, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
wShiftRight	The number of bitwise right shift operations to apply to each 32 bit pixel in *Hin_image*.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function takes a 32 bit input image, which typically contains data from a digital camera, and facilitates the conversion to 8 bit grayscale. This is useful when, for example, the image data from the camera is 12 bit – this function allows it to be conveniently converted to an 8 bit format suitable for display etc. Care must be taken with endian issues and the source data format.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how to convert 12 bit data to an 8 bit grayscale image suitable for display:

```
/* hSrcImage contains a 12 bit grayscale image – we'll convert the
 * image data to 8 bit so that we can view it.
 */
TMG_SPL_Data32_to_Y8(hSrcImage, hY8Image, 4, TMG_RUN);
TMG_display_image(hDisplay, hY8Image, TMG_RUN);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_SPL_XXXX32_to_Y8.

TMG_SPL_field_to_frame

USAGE

Terr TMG_SPL_field_to_frame(Thandle Hin_image, Thandle Hout_image, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function takes the input image, treats it as if it is one video field of an interlaced frame, and duplicates the other field resulting in a complete frame. In other words it doubles the vertical size of the image by repeating lines as follows:

Line 1 of the input image is duplicated to form lines 1 and 2 in the output image;
Line 2 of the input image is duplicated to form lines 3 and 4 in the output image etc.

This function might be needed in an application that acquires single video fields, which subsequently have to be viewed as a normal image.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how to use the function:

```
/* carpark.tif is a single field of video grabbed from a remote
 * security camera - we use TMG_SPL_field_to_frame to view it.
 */
TMG_image_set_infilename(hImage, "carpark.tif");
TMG_image_set_parameter(hImage, TMG_HEIGHT, TMG_AUTO_HEIGHT);
TMG_image_read(hImage, TMG_NULL, TMG_RUN);
TMG_SPL_field_to_frame(hImage, hOutImage, TMG_RUN);
TMG_display_image(hDisplay, hOutImage, TMG_RUN);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_SPL_2fields_to_frame, *TMG_IP_pixel_rep*, *TMG_IP_subsample*.

TMG_SPL_HSI_to_RGB_pseudo_colour

USAGE

Terr TMG_SPL_HSI_to_RGB_pseudo_colour(Thandle Hin_image, Thandle Hout_image, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input *TMG_HSI* image.
Hout_image	Handle to the output *TMG_RGB24* image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function simply maps the HSI planes to RGB planes. That is, Hue is mapped directly to Red, Saturation to Green and Intensity to Blue. This can be useful when examining HSI images, as it allows an HSI image to be put into a representation that standard image viewing packages can recognise and hence view – for example to compute and display histograms for each HSI component.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how to convert an image to HSI format and then to a pseudo colour RGB format suitable for saving as a TIFF file:

```
TMG_image_read(hInImage, TMG_NULL, TMG_RUN);
TMG_image_convert(hInImage, hYuvImage, TMG_YUV422, 0, TMG_RUN);
TMG_image_convert(hYuvImage, hHsiImage, TMG_HSI, 0, TMG_RUN);
TMG_SPL_HSI_to_RGB_pseudo_colour(hHsiImage, hPseudoRgbImage, TMG_RUN);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_SPL_YUV422_to_RGB_pseudo_colour, *TMG_image_convert*.

TMG_SPL_YUV422_to_RGB_pseudo_colour

USAGE

Terr TMG_SPL_YUV422_to_RGB_pseudo_colour(Thandle Hin_image, Thandle Hout_image, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input *TMG_YUV422* image.
Hout_image	Handle to the output *TMG_RGB24* image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function simply maps the YUV 4:2:2 planes to RGB planes. That is, luminance (Y) is mapped directly to Red, the U component to Green and the V component to Blue. This can be useful when examining YUV 4:2:2 images, as it allows a YUV 4:2:2 image to be put into a representation that standard image viewing packages can recognise and hence view – for example to compute and display histograms for each YUV 4:2:2 component.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how to convert an image to YUV 4:2:2 format and then to a pseudo colour RGB format suitable for saving as a TIFF file:

```
TMG_image_read(hInImage, TMG_NULL, TMG_RUN);
TMG_image_convert(hInImage, hYuvImage, TMG_YUV422, 0, TMG_RUN);
TMG_SPL_YUV422_to_RGB_pseudo_colour(hYuvImage, hPseudoRgbImage, TMG_RUN);
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_SPL_HSI_to_RGB_pseudo_colour, *TMG_image_convert*.

TMG_SPL_XXXX32_to_Y8

USAGE

Terr TMG_SPL_XXXX32_to_Y8(Thandle Hin_image, Thandle Hout_image, ui16 plane, ui16 TMG_action)

ARGUMENTS

Hin_image	Handle to the input image.
Hout_image	Handle to the output image.
plane	1, 2, 3 or 4 representing the plane to strip out of the input image.
TMG_action	Either *TMG_RUN* for normal operation or *TMG_RESET* if the operation needs to be aborted.

DESCRIPTION

This function takes a multi-planar input image - for example *TMG_RGBX32*, and strips out one component, for example the red component, and stores it as a grayscale image in *Hout_image*.

plane refers to the byte position in the RGB colour format (see the example below).

This function can be used if, for example, a colour image is acquired from colour acquisition hardware (three channel RGB) connected to three monochrome cameras and the grayscale image from one camera is required. See the example below.

RETURNS

ASL_OK on success, otherwise an error return as defined in the Error Returns section at the start of this manual.

EXAMPLES

The following code fragment shows how to use the function:

```
/* hSrcImage contains a colour image in the form RGBX32 which actually
 * represents three grayscale images acquired from colour acquisition
 * hardware connected to three synchronized monochrome cameras.
 */
switch (ColourPlane) {
   case RED:  /* read out correct mono plane from red channel */
      TMG_SPL_XXXX32_to_Y8(hSrcImage, hGrayImage, 1, TMG_RUN);
      break;
   case GREEN:
      TMG_SPL_XXXX32_to_Y8(hSrcImage, hGrayImage, 2, TMG_RUN);
      break;
   case BLUE:
      TMG_SPL_XXXX32_to_Y8(hSrcImage, hGrayImage, 3, TMG_RUN);
      break;
}
```

BUGS / NOTES

There are no known bugs.

SEE ALSO

TMG_SPL_Data32_to_Y8.