HUNT ENGINEERING
Chestnut Court, Burton Row,
Brent Knoll, Somerset, TA9 4BP, UK
Tel: (+44) (0)1278 760188,
Fax: (+44) (0)1278 760199,
Email: sales@hunteng.demon.co.uk
http://www.hunteng.co.uk
http://www.hunt-dsp.com

HUNT ENGINEERING

HEL_ILib

USER MANUAL

An optimised xDAIS-compliant image processing library for HERON-based DSP boards. HEL_ILib is intended for use with the TMS320C6000 based processor modules (eg HERON1-C6701, HERON3-C6211) and supports both cache and non-cache versions of the processor.

The first release of the library is for image processing on integer images – no floating point support is provided at this stage. Later versions will include additional functionality.

By providing feedback to Hunt Engineering, users can influence the functionality that will be added

Document Rev C
R. Weir 22/2/2001

COPYRIGHT

WARRANTIES LIABILITY and INDEMNITIES

HUNT ENGINEERING warrants the hardware to be free from defects in the material and workmanship for 12 months from the date of purchase. Product returned under the terms of the warranty must be returned carriage paid to the main offices of HUNT ENGINEERING situated at BRENT KNOLL Somerset UK, the product will be repaired or replaced at the discretion of HUNT ENGINEERING.

Exclusions - If HUNT ENGINEERING decides that there is any evidence of electrical or mechanical abuse to the hardware, then the customer shall have no recourse to HUNT ENGINEERING or its agents. In such circumstances HUNT ENGINEERING may at its discretion offer to repair the hardware and charge for that repair.

Limitations of Liability - HUNT ENGINEERING makes no warranty as to the fitness of the product for any particular purpose. In no event shall HUNT ENGINEERING'S liability related to the product exceed the purchase fee actually paid by you for the product. Neither HUNT ENGINEERING nor its suppliers shall in any event be liable for any indirect, consequential or financial damages caused by the delivery, use or performance of this product.

Because some states do not allow the exclusion or limitation of incidental or consequential damages or limitation on how long an implied warranty lasts, the above limitations may not apply to you.

TECHNICAL SUPPORT

Technical support for HUNT ENGINEERING products should first be obtained from the comprehensive Support section www.hunteng.co.uk/support/index.htm on the HUNT ENGINEERING web site. This includes FAQs, latest product, software and documentation updates etc. Or contact your local supplier - if you are unsure of details please refer to www.hunteng.co.uk for the list of current re-sellers.

HUNT ENGINEERING technical support can be contacted by emailing support@hunteng.demon.co.uk, calling the direct support telephone number +44 (0)1278 760775, or by calling the general number +44 (0)1278 760188 and choosing the technical support option.

N.B. Technical support for this Library (HEL_Ilib) will be provided for users of HUNT ENGINEERING hardware ONLY.

TABLE OF CONTENTS

HE_ILib is a library of image processing functions intended to make the task of building imaging systems quicker and easier. It is designed to be extendible; source code is available to users of Hunt Engineering boards, allowing functions to be modified if necessary.

The functions provided are heavily optimised. In most cases the functions are written in hand-coded assembler, but where this would provide no performance benefit, they are in optimised C.

The library is intended for use within the ExpressDSP environment. It is fully XDAIS Level 1 compliant.

No memory allocation is performed within the library. Similarly, no I/O functions are implemented – it is assumed that these are implemented by the hosting framework, such as HERON-API.

HEL_ILib is ideal for image processing applications, providing optimised functions for many common operations in imaging. It is an extendible library, and is designed for ease of use.

The user interface to the library is at a high level. Images are described by C-level structures, containing information about the image – size, resolution and address. It is these structures that are passed to the library's functions; the library will automatically run the optimal code for that image.

The library supports image tiling and Region of Interest. In many cases, these two approaches to splitting an image into fragments allows the processing to be performed on-chip, greatly enhancing performance.

Whether used at frame-level or tile-level, the library internally calls hand-optimised assembly routines to perform the image processing. These functions are not accessible to the user.

Function Classes

All functions are non-destructive, allowing source and destination images to be specified separately. Most can be used in-place (i.e. the destination image is the same as the source). Where this is not the case, a note is made in the documentation.

There are two classes of function in the library:

- MONADIC functions take a single input image to create a single output. They typically apply a constant operator to the image – for example, "AND with constant".
- DIADIC functions take two input images to create a single output – for example, "ADD two images".

Functions may use one or more constants in their operation. For example, a monadic operator might "AND image with constant". In this case, the constant is supplied as a parameter to the function.

Function Naming Conventions

Function names are built up from a prefix, logical function name and suffix. For the imaging functions, the prefix is always "HEL_Image".

Functions using a constant have a "K" as the suffix, immediately after the operator name.

Hence, the AND function would be named:

HEL_ImageAND	AND two images to a third. Performs resolution detection automatically from the image buffer passed to it. Data is not moved before processing.
HEL_ImageANDK	AND image with a constant. As the AND function, but the second image is replaced by a constant operator passed in the function's parameters.

Data Formats

This release of the library supports three integer data formats:

- 8-bit unsigned integer
- 16-bit signed integer
- 32-bit signed integer

Every operation performs saturation appropriate for each data type. Functions exist to convert images from one type to another.

Not all operations support all pixel depths. Where an operator is called for an unsupported image type, it will return an error code. The documentation notes the image resolutions & pixel depths supported by each operator.

Header Files

The library needs HEL_ImgLib.h included in each C source file. This includes the structure and function definitions required.

The C6000 processor's "branch" instruction takes 5 cycles to execute. As the processor can issue 8 cycles per clock, a branch can be accompanied by up to 39 instructions doing other useful work.

During the five cycles of a branch, the C6000 ignores interrupts. In many image processing operations, this could mean that interrupts are disabled for the entire duration of an operation – perhaps several milliseconds. This prevents the processor from switching to higher priority tasks, responding to external events, or servicing real-world requests. For many systems this is unacceptable.

This library has been coded to ensure that all loops take a minimum of 6 cycles. By taking this approach we can guarantee that the processor is always interruptible.

To ensure efficient operation, each loop processes multiple pixels so that all 6 cycles are utilised for useful work. Typically this means that up to 32 bytes are processed per loop.

Processing 32 bytes at a time means that the image lines have to be a multiple of 32 bytes long. This is not generally restrictive. Most common image sizes already meet these criteria. Others can be padded to make the line length a multiple of 32 bytes.

Many applications of image processing use images which are non-contiguous in memory. This can be a result of many things:

Frame buffer architecture. It is common for frame buffers to align all lines of an image to a memory page, so the address increment from the first pixel of the first line to the first pixel of the second line may well be larger than the line length.

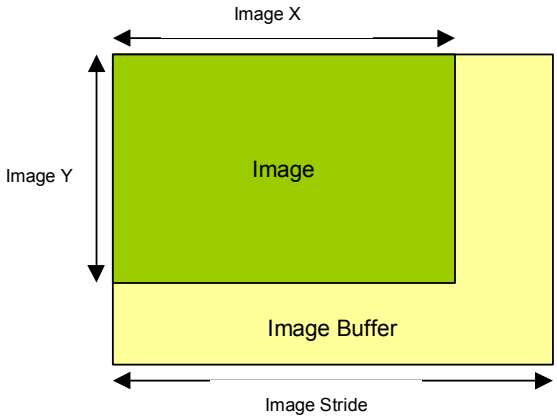
Region of Interest Selection. Here, an area of the image is identified for further processing. Typically, the processing is coplex – processing the whole image would be a waste of time, so we select a small "patch" or "region" to analyse.

In each case, copying the image to make the pixels contiguous in memory is wasteful of resources. The library implements ROI support to ensure this is never required.

This is done through the "stride" of the image. When defining an image, we set three parameters:

x	(image width)
y	(image height)
stride	(line-line address increment)

These are shown below:



When creating an image, specify the stride as being the same as X for contiguous memory, or greater than X for a Region of Interest.

The Image Copy routines will perform two types of copy:

- Source and destination have the same stride. In this case, both images can be non-contiguous.
- Images where one image is contiguous.

Note that the copy routine does not support copying images where the source and destination are non-contiguous with different strides. Do this through an intermediate buffer.

Introduction

The library processes images held in buffers. Typically these buffers are large – a standard 8-bit image at 512x512 resolution requires 256Kbytes, while a 1K*1K RGB image would require 4Mbytes.

For small images, it may be possible to hold these images in on-chip RAM. However, it is far more common for the image to be stored in external memory – typically SDRAM.

Off-chip memory is usually very slow in comparison with on-chip. The table below gives some representative times, taken on the TMS320C6201 processor:

Operation	Transfer Rate
CPU Read from on-chip RAM	0.125 cycles/byte
DMA Read from SDRAM (burst rate)	0.5 cycles/byte
CPU Read from SDRAM	4 cycles/byte

As can be seen, there are huge penalties in processing the off-chip images directly. Another point is that for many simple operations, the CPU can process images far faster than external memory can provide them:

Operation	Transfer Rate
AND image with constant	8-bit: 4 pixels / cycle 16-bit: 2 pixels / cycle 32-bit: 1 pixels / cycle
Internal memory read / write cycle	8-bit: 4 pixels / cycle 16-bit: 2 pixels / cycle 32-bit: 1 pixels / cycle
External SDRAM CPU read/write	8-bit: 0.25 pixels / cycle 16-bit: 0.125 pixels / cycle 32-bit: 0.0625 pixels / cycle

Plainly simple operations can be performed very quickly, but if the processor is accessing the image in external memory, a bottleneck is created.

Image Tiling

To overcome this, HEL_ILib offers the "Tiled" mode of operation. To achieve this, the image is split into tiles. Each tile is loaded into on-chip memory or the cache. The CPU then performs MULTIPLE operations on that tile, before saving it back and proceeding with the next tile.

To understand this fully, consider the following example processing pipeline. This is purely an example – it is not meant to be part of a real application. Image is 1K*1K*8 bits, stored in external SDRAM:

1. NOT Every pixel in the image is "inverted" using the NOT function
2. OR 0x80 Top bit of every pixel is set
3. AND 0xfe And the last bit of every pixel is cleared…

Implementing this sequence directly:

Operation	Description	Actions	Time
NOT	Load 1K*1K pixels, invert all bits, store back to SDRAM	Read: 1K*1K*16 cycles Write: 1K*1K*16 cycles CPU: Insignificant	32M
OR	Load 1K*1K pixels, OR with 0x80, store back to SDRAM	Read: 1K*1K*16 cycles Write: 1K*1K*16 cycles CPU: Insignificant	32M
AND	Load 1K*1K pixels, AND with 0xfe, store back to SDRAM	Read: 1K*1K*16 cycles Write: 1K*1K*16 cycles CPU: Insignificant	32M
Total			96M cycles

In contrast, the tiled approach splits the image into perhaps 100 tiles. In this case, we would repeat the following code 100 times:

Operation	Description	Actions	Time
Load Tile N	Load 1K*1K/100 pixels using DMA	Read: 1K*1K*2 / 100 cycles	20K
NOT Tile	invert all bits	CPU: (1K*1K/100) / 4 cycles	2.5K
OR Tile	OR tile with 0x80	CPU: (1K*1K/100) / 4 cycles	2.5K
AND Tile	AND tile with 0xfe	CPU: (1K*1K/100) / 4 cycles	2.5K
Save Tile N	Save 1K*1K/100 pixels using DMA	Write: 1K*1K*2 / 100 cycles	20K
Total (for Tile)			47.5K cycles
Total (for Image)			4.75M cycles

So, the tile approach is over an order of magnitude faster. Further savings are gained by overlapping the load and store of the images with the processing, using the DMA. In this

case, the load/store take 40K cycles – the processing would almost disappear.

Using the tiled approach gives a performance improvement of approximately 20x. Other benchmarks will give different results, but tiling will almost always be faster than the direct approach.

How Tiling is implemented

Overview

In HEL_DSPLib a 1-dimensional tiling system is used. This is the simplest possible form of tiling; 2-dimensional tiling would introduce overhead in processing and is unnecessary for most applications.

The image is divided into horizontal "bands", running across the image. Each band contains one or more "lines" of the image. A tile always contains an integer number of lines. The size of the tile is determined when the image buffer is created.

While the tiling system was originally conceived for non-cache processors, it is also extremely effective with the cache-based devices. In this case, the tiling system is used to reduce the frequency of L2 misses.

In the Version 1.0 library tiling must be implemented manually, using the DMA and ROI routines to copy data to and from the on-chip memories.

Using Tiling

In V1.0 of the library, tiling is implemented through the ROI system. Tiles are loaded from main memory as if they were "Regions of Interest", processed, and returned to external memory.

Data dependency of Tiles

We must divide the image logically into Tiles, where each tile can be processed independently of all others. There must be no data dependency between one tile and any other; for this reason, many algorithms will use a source tile that is larger than the image area being processed.

For example, to filter 25 lines of an image with a 3x3 convolution, we would load a tile of 27 lines – the extra line before and after the image allows the filter kernel to process out to the borders of the image. Loading this border is an overhead of the tiling approach; and generally, the fewer tiles there are, the lower the overhead.

Note that when we save the filtered data back to memory, we will save 25 lines – so in this case the ROI used to load the tile is different to that used to save it.

Dividing the Image Into Tiles

To decide how large the tiles can be, start by determining the largest overlap your algorithm requires. Because we operate 1D tiling, you only need to worry about lines. Many functions require no overlap – for example, per-pixel functions. Others will require several extra lines of data – for example, a convolution will always require extra lines. You can get the overlap figures from the function list in this manual. If no overlap is quoted, none is required.

Now, determine the pixel resolution you will process the image at, and the line length of the image. This will tell you how many bytes of on-chip memory an image line will require.

Finally, consider how the dataflow will work in your algorithm. In many cases, we would use four buffers:

DMA_In	Buffer to load tiles into
Process_1	Buffer to process image in
Process_2	Buffer to place results in
DMA_Out	Storage buffer

In a situation like this, we would use the DMA to load the DMA_In buffer with image data. When ready, we would convert that buffer to the on-chip format – perhaps converting 8-bit pixels to 16-bit – and place the result in the Process_1 buffer.

The image tile, now at 16-bit resolution, is processed. Typically, each processing step takes the image from one processing buffer to the other. However, it may be possible to use a single buffer with some algorithms; or it may be that the algorithm requires two buffers, as in the case of convolution – in which case Process_2 would be used.

In either case, we would take the output buffer and convert it back to DMA_Out. Note that we do not need to convert the overlap – and we MUST NOT store the overlap pixels into the results image.

Maximum Image Size

The library supports a maximum resolution of 64Kbytes (X-dimension) by 64K pixels (Y-dimension).

Where tiling is used, the maximum tile size is dictated by the amount of on-chip memory available to place the tile in. More complex functions require several lines of the image to be present in the tile – for example, a 9x9 convolution requires at least 9 lines. Thus, the limit for tiling is set by the amount of on-chip memory.

Image Alignment & Granularity

The library uses highly-optimised C and assembler code for performance. To achieve this, some trade-offs have been made; for example, many routines process several pixels at a time. Examples of this are the logical operators, which can often operate on 4 8-bit pixels using a single processor instruction.

This limits the minimum line length of the image; it also affects the granularity of the line length. There are some restrictions on the line lengths and image alignments as a result.

- Images must be aligned on a 64-bit boundary. The library makes use of known data alignment to optimise memory accesses, resulting in a very low rate of memory collisions. Violating this will invalidate all the timing information given in this document.
- Images must have a multiple of 32 bytes per line, regardless of pixel depth. If the image is not a multiple of 32 bytes, operation is undefined.
- The minimum line length is 64 bytes, regardless of pixel depth. If this restriction is not met, operation is undefined.

While the restrictions sound severe, all common image sizes are supported. The restrictions were introduced to increase performance with no loss of flexibility in use.

Introduction

The library operates on images. To simplify the use of the library, all details of an image are stored in the HEL_ImageObj structure. To apply an operator to an image, we pass the HEL_ImageObj structure.

As an example, consider the following library call:

```
HEL_ImageObj Image;
….
HEL_ImageFILLK(&Image,0x0000);
```

HEL_ImageFILLK is used to fill an image with a constant value. In this case, we've passed the address "&Image" as the full description of the image to be filled. Apart from this, the only other information required is the constant for the fill operation.

Creating the HEL_ImageObj Structure

The HEL_ImageCreate function initialises the structure. This is passed all the parameters of an image. It does not allocate memory for the structure, nor does it allocate memory for the image.

There is no explicit need to destroy the structure. It is up to the controlling program to allocate the memory, and when the structure is no longer required, it can de-allocate the memory.

Creating Regions of Interest (ROI)

The library supports ROI operations directly. Regions of Interest are described by HEL_ImageObj structures; these can be initialised by a call to HEL_CreateROI.

This function takes coefficients of the part of the image to be processed, plus pointers to two HEL_ImageObj structures – one for the image of interest, the other for an empty HEL_ImageObj structure which will be initialised.

As with HEL_CreateImage there is no need to specifically destroy the ROI when finished with it; the controlling program may de-allocate the memory as required.

The essence of using the library in standard mode can be seen from this example. Here, we capture an image, AND with a constant, filter it, subtract a constant and save the result to memory. In the first example, this is performed using the standard library; in the second, tiling is used to enhance performance.

RASW Note – this is just pseudo code to show concepts – later it'll be real!

Standard Example

```
/* set up an image handle */
HELImageObj  OurImage;

/* Initialise HELImageObj structure. This sets it up for an image of
640*480, in a buffer with 1k long lines      */
/* Pixel depth (resolution) is 8-bits, while the format is Mono (Grey
Scale)             */
Image1 =
HEL_ImageCreateImage(&OurImage,&buffer,640,480,1024,8,HEL_MonoImage);

/* create an endless loop capturing & processing images */
for(;;)
{
/* capture image. This is beyond the scope of the library */
….
/* And image with constant, result overwrites original */
HEL_ImageANDK(OurImage,OurImage,0x7f);

/* OR image with constant, result overwrites input */
HEL_ImageORK(Image2,0x80, Image2);

/* Display image – again beyond the scope of this library */
….
}
```

Image Management

ImageCreate	Create an image object	
ImageConvert	Convert image from one pixel depth to another	
ImageCopy	Copy an image using a DMA	
CreateROI	Create an ROI object from an existing image.	
CreateTile		*Not implemented in this release*
ScaleImage	Scale image from one resolution to another	*Not implemented in this release*

Arithmetic Operators

ABS	Absolute value of image	
ADD	Add two images	
MPY	Multiply two images	
MPYK	Multiply image by constant	
MPYKSCALE	Multiply image by constant, shift result	
MPYSCALE	Multiply two images and shift result	
SQUARE	Square the pixels in an image	
SUB	Subtract one image from another	
SUBK	Subtract a constant from the image	

Logical Operators

AND	Bitwise AND two images	
ANDK	Bitwise AND image with constant	
SHLK	Left-shift image	
SHRK	Right-shift image	
NOT	Bitwise NOT of image	
OR	Bitwise OR two images	
ORK	Bitwise OR image with constant	
XOR	Bitwise XOR two images	
XORK	Bitwise XOR image with constant	
FILLK	Fill image with constant	
FILLRAMP	Fill image with ramp	*Not optimised in this release*

Filtering Functions

CONV	Performs N*M convolution	
MEDIAN	Performs median filtering	*Not implemented in this release*
MAX	Performs maximum filtering	*Not implemented in this release*
MIN	Performs minimum filtering	*Not implemented in this release*

Overview

Following is a description of each function, with any special requirements. The following notes are applicable to the whole library:

Special Requirements:

Images must meet the size constraints outlined earlier. If there are any further special requirements these are outlined in the notes section for each function.

Implementation Notes:

For each function, code is implemented as an unrolled loop, processing 8 32-bit words per pass. This is independent of pixel depth, so could be 32 pixels (8-bit) 16 pixels (16-bit) or 8 pixels (32-bit). This ensures that loops are interruptible.

The loop is guaranteed to execute at least twice.

Data Checking:

All functions perform saturation arithmetic unless otherwise specified.

Bank Conflicts:

All code is written to eliminate bank conflicts.

Endian:

All code is little-endian.

Interrupts:

All code is fully interruptible (and interrupt tolerant). Maximum latency ~6 cycles.

Image Management Functions

HEL_ImageCreate

HEL_ImageObj *HEL_ImageCreate (HEL_ImageObj *Image, int *Memory, short x, short y, short stride, short res, short format);

Parameters:	ImageObj:	Image object to be initialised.
	Memory:	Array of ints used to store the image.
	X:	X-dimension of the image, in pixels.
	Y:	Y-dimension of the image, in pixels.
	Stride:	Size of the image buffer in pixels. Note that this might be larger than the image – for example, image buffers from the HEGD6 always have an additional 4-byte boundary at the end of each line.
	Res:	Pixel depth of the image. This can be 8, 16 or 32, indicating 8/16/32-bit pixels.
	Format:	Indicates the format of the pixel. Currently only HEL_MonoImage supported (single pixel plane, integer).
Return:		Pointer to object on success, HEL_ERR otherwise

Description:

HEL_CreateImage initialises an image object. It allocates no memory, and must be passed pointers to:

a) The image object

b) The image array

Other parameters passed set the size, pixel depth and stride of the image. Note that the image must meet the requirements for image size specified elsewhere.

The image structure can be manipulated directly; however this function provides easy access to it.

HEL_ROICreate

HEL_ImageObj *HEL_ROICreate (HEL_ImageObj *Image,
HEL_ImageObj *ROI,
short x,
short y,
short line_length,
short lines);

Parameters:	Image:	Image from which we will create ROI.
	ROI:	Image object to be initialised with ROI
	X:	X-coefficient of start of ROI, in pixels.
	Y:	Y-coefficient of start of ROI, in pixels.
	line_length	X-dimension of ROI, in pixels
	lines:	Y-dimension of the ROI, in pixels.
Return:	Pointer to object on success, HEL_ERR otherwise	

Description:

HEL_ROICreate creates an ROI object. It is passed a source image, an empty image object, and parameters for creating the ROI.

The ROI is created by setting up pointers to the ROI. No memory is copied or allocated. This is an efficient way of handling ROI processing.

Note that the ROI must meet the requirements for image size specified elsewhere. These specify the alignment of the image, and restrictions on the number of pixels in a line.

The image structure used is identical to that for an image. ROIs are then processed in exactly the same way as images.

As no memory is allocated it is not necessary to explicitly destroy the ROI; the image object can be de-allocated or re-used as required.

HEL_TileCreate

Not implemented in this release

The HEL_TileCreate initialises the control object for the tiling system. It will be implemented in a forthcoming release.

HEL_ImageCONVERT

HEL_ImageObj *HEL_ImageConvert (HEL_ImageObj *InImage, HEL_ImageObj *OutImage);

Parameters:	InImage:	Source Image object
	OutImage:	Destination Image object.
Return:	Pointer to object on success, HEL_ERR otherwise	

Description:

HEL_ConvertImage takes an image and converts it to a different pixel depth. The conversion performed is controlled by the two image objects passed.

As an example, if InImage describes an 8-bit image and OutImage describes a 16-bit image, the function will copy the image from one to the other, converting the 8-bit source pixels to 16-bit.

When the pixel depth is increased, no checking of pixels is required. Pixels are placed in the low-order part of the output pixel. Sign extension is performed.

When reducing the pixel depth, no saturation is performed. Note that the 32-bit and 16-bit formats are signed, while the 8-bit format is unsigned

The only parameter read from the OutImage structure is the required pixel depth. All other parameters are written with the parameters of the converted image.

Notes:

The Convert function cannot operate in-place when the output pixels are larger than the input pixels. In this case, InImage and OutImage must not be the same.

Where Convert is called to convert between two images of the same pixel depth, the second image object is initialised to point to the first. No copy is performed.

HEL_ImageCOPY

HEL_ImageObj *HEL_ImageCOPY (HEL_ImageObj *InImage, HEL_ImageObj *OutImage, int DMA_Channel);

Parameters: InImage: Source Image object

OutImage: Destination Image object.

DMA_Channel: DMA controller to be used

Return: Pointer to object on success, HEL_ERR otherwise

Description:

HEL_ImageCOPY uses the DMA to replicate an image, ROI or tile. No translation of the image pixel data is performed; the source and destination images are identical in appearance.

XXXX????

The image stride is set by the destination image. This allows ROI regions to be copied efficiently to on-chip memory.

The function is called asynchronously; that is, calling the function starts the DMA, but the function may return before the DMA completes. This allows the processor to do other useful work while the DMA is in progress.

To check if the copy is complete or not, call the HEL_ImageCOPYCHK function.

The DMA_Channel passed to the function should be available for immediate use. It is up to the user to ensure this DMA is reserved for the duration of the copy; this can be performed within HERON-API using the DMA claim functions.

Note that if copying an image between locations in on-chip memory, the DMA is not the fastest approach. Instead, use a simple operator with no translation; for example, OR(image1, image2, 0x00) will copy the image in image1's framebuffer to image2's framebuffer without modifying the pixel data.

HEL_ImageSCALE

Not implemented in this release

This function will scale an input image to the resolution of the output image. All other parameters of the output image are ignored, and overwritten with the parameters of the scaled image.

Arithmetic Operators

HEL_ImageABS

int HEL_ImageABS (HEL_ImageObj *InImage,
HEL_ImageObj *OutImage);

Parameters:	InImage:	Source Image object
	OutImage:	Destination Image object.
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Takes the absolute value of an image. Supports all resolutions.

Destination may be the same as source.

HEL_ImageADD

int HEL_ImageADD(HEL_ImageObj *InImage1,
HEL_ImageObj *InImage2,
HEL_ImageObj *OutImage);

Parameters:	InImage1:	Source Image object
	InImage2:	Second source image
	OutImage:	Destination Image object.
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Calculate the sum of two images on a pixel by pixel basis. Supports all pixel depths. Images must be the same resolution and pixel depth.

Destination may be the same as source.

HEL_ImageADDK

int HEL_ImageADDK(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage,
int constant);

Parameters:	InImage1:	Source Image object
	OutImage:	Destination Image object
	Constant:	Constant to be added
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Adds a constant to an image on a pixel by pixel basis. Supports all pixel depths.

Destination may be the same as source.

HEL_ImageMPY

int HEL_ImageMPY(HEL_ImageObj *InImage1,
HEL_ImageObj *InImage2,
HEL_ImageObj *OutImage);

Parameters:	InImage1:	Source Image object
	InImage2:	Second source image
	OutImage:	Destination Image object.
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Calculate the product of two images on a pixel by pixel basis. Supports all pixel depths. Images must be the same resolution and pixel depth.

Destination may be the same as source.

HEL_ImageMPYK

int HEL_ImageMPYK(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage,
int constant);

Parameters:	InImage1:	Source Image object
	OutImage:	Destination Image object
	Constant:	Constant to be multiplied
	Shift:	Shift value.
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Forms the product of an image with a constant on a pixel by pixel basis, effectively scaling the pixel values of the image. Supports all pixel depths.

Destination may be the same as source.

HEL_ImageMPYKSCALE

int HEL_ImageMPYKSCALE(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage,
int constant,
int shift);

Parameters:	InImage1:	Source Image object
	OutImage:	Destination Image object
	Constant:	Constant to be multiplied
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Multiplies each pixel in an image by a constant, then scales the result by shifting left or right. Allows implementation of fixed-point calculations.

Positive values of the "Shift" constant shift left; negative values shift right.

Supports all pixel depths.

Destination may be the same as source.

HEL_ImageMPYSCALE

int HEL_ImageMPYSCALE(HEL_ImageObj *InImage1,
HEL_ImageObj *InImage2,
HEL_ImageObj *OutImage,
int Shift);

Parameters:	InImage1:	Source Image object
	InImage2:	Second source image
	OutImage:	Destination Image object.
	Shift:	Shift constant
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Multiplies the pixels of two images together, then scales the result by shifting left or right. Allows implementation of fixed-point calculations.

Positive values of the "Shift" constant shift left; negative values shift right.

Supports all pixel depths.

Destination may be the same as source.

HEL_ImageSQR

int HEL_ImageSQR (HEL_ImageObj *InImage,
HEL_ImageObj *OutImage);

Parameters:	InImage:	Source Image object
	OutImage:	Destination Image object.
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Squares all the pixels of an image. Supports all pixel depths. Images must be the same resolution and pixel depth.

Destination may be the same as source.

Note that this function is based on **HEL_ImageMPY**; if a shift is required, the most efficient approach is to use **HEL_ImageMPYSCALE.**

HEL_ImageSUB

int HEL_ImageSUB(HEL_ImageObj *InImage1,
HEL_ImageObj *InImage2,
HEL_ImageObj *OutImage);

Parameters:

InImage1: Source Image object

InImage2: Second source image (to be subtracted)

OutImage: Destination Image object.

Return: HEL_OK on success, HEL_ERR otherwise

Description:

Subtracts one image from another on a pixel by pixel basis. Supports all pixel depths. Images must be the same resolution and pixel depth.

Destination may be the same as source.

HEL_ImageSUBK

int HEL_ImageSUBK(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage,
int constant);

Parameters:

InImage1: Source Image object

OutImage: Destination Image object

Constant: Constant to be subtracted

Return: HEL_OK on success, HEL_ERR otherwise

Description:

Subtracts a constant to an image on a pixel by pixel basis. Supports all pixel depths.

Destination may be the same as source.

Logical Operators

HEL_ImageAND

int HEL_ImageAND (HEL_ImageObj *InImage1,
HEL_ImageObj *InImage2,
HEL_ImageObj *OutImage);

Parameters:	InImage1:	Source Image object
	InImage2:	Second source image
	OutImage:	Destination Image object.
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

ANDs one image with another on a pixel by pixel basis. Supports all pixel depths. Images must be the same resolution and pixel depth.

Destination may be the same as source.

HEL_ImageANDK

int HEL_ImageANDK(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage,
int constant);

Parameters:	InImage1:	Source Image object
	OutImage:	Destination Image object
	Constant:	Constant for AND
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

ANDS a constant with an image on a pixel by pixel basis. Supports all pixel depths.

Destination may be the same as source.

HEL_ImageSHLK

int HEL_ImageSHLK(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage,
int constant);

Parameters:	InImage1:	Source Image object
	OutImage:	Destination Image object
	Constant:	Number of bits to shift left
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Logically left-shifts an image by a constant value, on a pixel by pixel basis. Supports all pixel depths.

Destination may be the same as source.

HEL_ImageSHRK

int HEL_ImageSHRK(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage,
int constant);

Parameters:	InImage1:	Source Image object
	OutImage:	Destination Image object
	Constant:	Number of bits to shift right
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Logically right-shifts an image by a constant value, on a pixel by pixel basis. Supports all pixel depths.

Destination may be the same as source.

HEL_ImageNOT

int HEL_ImageNOT(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage);

Parameters:	InImage1:	Source Image object
	OutImage:	Destination Image object
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Forms the logical NOT of an image. Supports all pixel depths.

Destination may be the same as source.

Note that this operator is functionally equivalent to XORK(image, image, 0xffffffff); if XORK is used elsewhere in your code you may be able to reduce program size through using XORK rather than NOT.

HEL_ImageOR

int HEL_ImageOR (HEL_ImageObj *InImage1,
HEL_ImageObj *InImage2,
HEL_ImageObj *OutImage);

Parameters:	InImage1:	Source Image object
	InImage2:	Second source image
	OutImage:	Destination Image object.
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

ORs one image with another on a pixel by pixel basis. Supports all pixel depths. Images must be the same resolution and pixel depth.

Destination may be the same as source.

HEL_ImageORK

int HEL_ImageORK(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage,
int constant);

Parameters:

InImage1: Source Image object

OutImage: Destination Image object

Constant: Constant for OR

Return: HEL_OK on success, HEL_ERR otherwise

Description:

ORs a constant with an image on a pixel by pixel basis. Supports all pixel depths.

Destination may be the same as source.

HEL_ImageXOR

int HEL_ImageXOR(HEL_ImageObj *InImage1,
HEL_ImageObj *InImage2,
HEL_ImageObj *OutImage);

Parameters:

InImage1: Source Image object

InImage2: Second source image

OutImage: Destination Image object.

Return: HEL_OK on success, HEL_ERR otherwise

Description:

XORs one image with another on a pixel by pixel basis. Supports all pixel depths. Images must be the same resolution and pixel depth.

Destination may be the same as source.

HEL_ImageXORK

int HEL_ImageXORK(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage,
int constant);

Parameters:	InImage1:	Source Image object
	OutImage:	Destination Image object
	Constant:	Constant for XOR
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

XORs a constant with an image on a pixel by pixel basis. Supports all pixel depths.

Destination may be the same as source.

HEL_ImageFILLK

int HEL_ImageFILLK(HEL_ImageObj *InImage,
HEL_ImageObj *OutImage,
int constant);

Parameters:	InImage1:	Source Image object
	OutImage:	Destination Image object
	Constant:	Constant for fill
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Fills an image with a constant. Supports all pixel depths.

Destination may be the same as source.

HEL_ImageFILLRAMP

int HEL_ImageFILLRAMP(HEL_ImageObj *InImage,
int start_value,
int x_ramp,
int y_ramp);

Parameters:	InImage1:	Source Image object
	Start_Value:	Value for first pixel in image
	X_Ramp:	Increment for horizontal ramp
	Y_Ramp:	Increment for vertical ramp
Return:	HEL_OK on success, HEL_ERR otherwise	

Description:

Fills an image with a 1D or 2D ramp. Supports all pixel depths.

The first pixel in the image ([0,0] – top left hand corner) is filled with Start_Value.

Working along the first horizontal line of the image, each pixel is incremented by X_Ramp. The start value of subsequent lines is incremented by Y_Ramp.

Each pixel can then be calculated by:

Pixel Value = Start_Value + X * X_RAMP + Y * Y_RAMP;

Destination may be the same as source. Note that this function is not fully optimised in this release.

Filtering Functions

HEL_ImageCONV

int HEL_ImageCONV (HEL_ImageObj *SourceImage,
HEL_ImageObj *DestImage,
int KernelX,
int KernelX,
int *Kernel);

Parameters:

SourceImage:	Source Image object
DestImage:	Result Image object
KernelX:	X-dimension of kernel
KernelY:	Y-dimension of kernel
Kernel:	Kernel data, as a packed array of integers

Return: HEL_OK on success, HEL_ERR otherwise

Description:

Convolves the image with the given kernel. Supports 16-bit pixels only in this release.

Applies a convolution kernel of N*M, where N can be any odd-valued integer from 1-9. M can be any value greater than 1.

The kernel is an array of integers – for example, a 3x3 kernel would be declared as:

int Kernel33 [3][3] = {1,2,3,4,5,6,7,8,9};

This operation cannot be performed in-place. It is essential that source and destination buffers are different.

Standard Filter Configurations

The library includes some standard filter profiles. These are defined in HEL_ImageFilter.h. This file only needs to be included in the C file calling the convolution function.

The standard profiles are:

Kernel Name	Size	Gain	Notes
Kern33LPF Kern33Gaussian	3x3	16x	Gaussian Low Pass Filter. Kern33LPF is an alias for ease of use.
Kern33HPF Kern33Laplacian	3x3	1x	Laplacian High Pass Filter. Kern33HPF is an alias for ease of use.
Kern33PrewittV	3x3	1x	Vertical Prewitt operator / gradient filter
Kern33PrewittH	3x3	1x	Horizonal Prewitt operator / gradient filter
Kern33SobelV	3x3	1x	Vertical Sobel operator / gradient filter
Kern33SobelH	3x3	1x	Horizontal Sobel operator / gradient filter
Kern33Sharpen	3x3	1x	Sharpening filter
Kern33APF	3x3	1x	All pass filter – use for testing. Applies kernels to image but should not affect pixels.
Kern55LPF Kern55Gaussian	5x5	571x	Gaussian Low Pass Filter. Kern55LPF is an alias for ease of use.
Kern55HPF Kern55Laplacian	5x5	1x	Laplacian High Pass Filter. Kern55HPF is an alias for ease of use.

Standard kernels are applied in the same way as custom kernels:

```
HEL_CONV(SourceImage, DestImage,KernelX, KernelY, &Kernel);
```

For example, to apply a high pass filter with a 3x3 kernel:

```
HEL_CONV(SourceImage, DestImage, 3, 3, &Kern33HPF);
```

To apply a low pass filter with a 3x3 kernel, then scale the result for unity gain:

```
HEL_CONV(SourceImage, DestImage, 3, 3, &Kern33LPF);
HEL_ASHRK(DestImage,DestImage, 4);
```

Modifying the standard configurations

HEL_ImageFilter.h creates all the standard configurations as global variables. Once included, they can be accessed from any file in the project; however, this is not a good use of resources.

Instead, we recommend that the user copy the profiles he needs out of HEL_ImageFilter.h and includes them within his code – creating them as global or local variables as appropriate.

The following tables give the expected performance of the library when applied to an M*N image, based on code and data being in internal/cache memory.

These figures are based on the standard build of the library using the TI v4 compiler. Options include setting the interrupt latency to 100, no bad aliases and optimisation / inlining enabled. Other build options may generate different code sizes and performance figures!

8-bit Functions

Function	Description	Cycles/ Pixel
ADD	Add 2 images	248 + M * (149 + N * 0.4)
ADDK	Adds constant to image	588 + M * (28 + N * 1.1)
AND	AND 2 images	328 + M * (21 + N * 0.3)
ANDK	AND image with constant	248 + M * (13 + N * 0.3)
ASHLK	Arithmetic Left-shift image by constant	600 + M * (7 + N * 1.5)
ASHRK	Arithmetic Right-shift the image by constant	612 + M * (6 + N * 0.9)
FILLK	Fill image with constant	172 + M * (0 + N * 0.2)
FILLRAMP	Fill image with ramp	224 + M * (5 + N * 1.5)
MPY	Multiply two images	364 + M * (14 + N * 1.5)
MPYK	Multiply image by a constant	364 + M * (91 + N * 1.0)
MPYKSCALE	Multiply image by a constant and scale the result	572 + M * (29 + N * 1.2)
MPYSCALE	Multiply two images & right-shift result	336 + M * (136 + N * 0.9)
NOT	Performs NOT on pixels of image	296 + M * (31 + N * 0.2)
OR	OR 2 images	368 + M * (26 + N * 0.3)
ORK	Ors the image with a constant	248 + M * (13 + N * 0.3)
SHLK	Left-shift image by constant	412 + M * (15 + N * 0.3)
SHRK	Right-shift image by constant	336 + M * (35 + N * 0.1)
SQUARE	Squares all pixel values in the image	348 + M * (144 + N * 0.5)
SUB	Subtract one image from another	344 + M * (121 + N * 0.6)
SUBK	Subtract constant from image	616 + M * (18 + N * 1.1)
XOR	XOR 2 images	296 + M * (31 + N * 0.2)
XORK	XOR image with constant	248 + M * (13 + N * 0.3)

16-bit Functions

ABS	Absolute value of image	384 + M * (6 + N * 1.0)
ADD	Add 2 images	252 + M * (124 + N * 0.6)
ADDK	Adds constant to image	512 + M * (3 + N * 1.0)
AND	AND 2 images	76 + M * (8 + N * 0.8)
ANDK	AND image with constant	312 + M * (12 + N * 0.5)
ASHLK	Arithmetic Left-shift image by constant	512 + M * (10 + N * 1.2)
ASHRK	Arithmetic Right-shift the image by constant	544 + M * (11 + N * 1.2)
FILLK	Fill image with constant	76 + M * (-2 + N * 0.4)
FILLRAMP	Fill image with ramp	304 + M * (-5 + N * 3.0)
MPY	Multiply two images	268 + M * (13 + N * 0.9)
MPYK	Multiply image by a constant	440 + M * (12 + N * 0.9)
MPYKSCALE	Multiply image by a constant and scale	272 + M * (22 + N * 0.8)
MPYSCALE	Multiply two images & right-shift result	104 + M * (13 + N * 0.9)
NOT	Performs NOT on pixels of image	268 + M * (13 + N * 0.5)
OR	OR 2 images	56 + M * (8 + N * 0.8)
ORK	Ors the image with a constant	160 + M * (14 + N * 0.5)
SHLK	Left-shift image by constant	36 + M * (20 + N * 0.5)
SHRK	Right-shift image by constant	120 + M * (6 + N * 0.7)
SQUARE	Squares all pixel values in the image	556 + M * (21 + N * 0.9)
SUB	Subtract one image from another	76 + M * (109 + N * 0.7)
SUBK	Subtract constant from image	632 + M * (3 + N * 1.0)
XOR	XOR 2 images	80 + M * (8 + N * 0.8)
CONV3x1	3x1 Convolution	144 + M * (182 + N * 2.0)
CONV3x3	3x3 Convolution	436 + M * (17 + N * 6.0)
CONV3x5	3x5 Convolution	276 + M * (216 + N * 14.8)
CONV3x7	3x7 Convolution	292 + M * (254 + N * 21.0)
CONV3x9	3x9 Convolution	348 + M * (322 + N * 26.9)
CONV5x1	5x1 Convolution	176 + M * (141 + N * 2.4)
CONV5x3	5x3 Convolution	268 + M * (168 + N * 9.7)
CONV5x5	5x5 Convolution	316 + M * (192 + N * 17.0)
CONV5x7	5x7 Convolution	416 + M * (258 + N * 24.0)
CONV5x9	5x9 Convolution	488 + M * (324 + N * 31.0)
CONV7x1	7x1 Convolution	164 + M * (152 + N * 2.8)
CONV7x3	7x3 Convolution	292 + M * (188 + N * 11.1)
CONV7x5	7x5 Convolution	396 + M * (211 + N * 19.5)
CONV7x7	7x7 Convolution	468 + M * (300 + N * 27.4)

CONV7x9	7x9 Convolution	620 + M * (360 + N * 35.5)
CONV9x1	9x1 Convolution	220 + M * (157 + N * 3.8)
CONV9x3	9x3 Convolution	352 + M * (202 + N * 20.0)
CONV9x5	9x5 Convolution	424 + M * (223 + N * 36.4)
CONV9x7	9x7 Convolution	588 + M * (278 + N * 52.5)
CONV9x9	9x9 Convolution	744 + M * (347 + N * 68.5)
XORK	XOR image with constant	252 + M * (13 + N * 0.5)

32-bit Functions

ABS	Absolute value of image	204 + M * (23 + N * 0.9)
ADD	Add 2 images	552 + M * (7 + N * 1.5)
ADDK	Adds constant to image	144 + M * (65 + N * 0.6)
AND	AND 2 images	56 + M * (8 + N * 1.5)
ANDK	AND image with constant	312 + M * (12 + N * 1.0)
ASHLK	Arithmetic Left-shift image by constant	284 + M * (52 + N * 0.7)
ASHRK	Arithmetic Right-shift the image by constant	292 + M * (52 + N * 0.7)
FILLK	Fill image with constant	76 + M * (0 + N * 0.8)
FILLRAMP	Fill image with ramp	236 + M * (2 + N * 6.0)
MPY	Multiply two images	232 + M * (35 + N * 2.5)
MPYK	Multiply image by a constant	80 + M * (154 + N * 1.1)
MPYKSCALE	Multiply image by a constant and scale the result	284 + M * (167 + N * 1.1)
MPYSCALE	Multiply two images & right-shift result	56 + M * (209 + N * 1.4)
NOT	Performs NOT on pixels of image	240 + M * (13 + N * 1.0)
OR	OR 2 images	48 + M * (8 + N * 1.5)
ORK	Ors the image with a constant	168 + M * (12 + N * 1.1)
SHLK	Left-shift image by constant	28 + M * (29 + N * 0.9)
SHRK	Right-shift image by constant	92 + M * (6 + N * 1.3)
SQUARE	??? Squares all pixel values in the image - why???	224 + M * (193 + N * 1.3)
SUB	Subtract one image from another	344 + M * (7 + N * 1.5)
SUBK	Subtract constant from image	360 + M * (27 + N * 0.9)
XOR	XOR 2 images	52 + M * (9 + N * 1.5)
XORK	XOR image with constant	228 + M * (12 + N * 1.0)

Introduction

Several examples are included with the library. These demonstrate the way the library is constructed, how it deals with images, and gradually introduce more complex concepts.

The examples are intended for use on a HUNT ENGINEERING system, running Server/Loader. There is no requirement for a video interface – images are read to & from disk.

Using the Examples

Assuming you are using HUNT ENGINEERING hardware, build the examples as follows:

1) Create a new project in the examples directory, using the Tools/Hunt Engineering/Create New Project tool. The project name should match the example you want to use – eg "Example1.mak" for the first example.
2) Add file "HrnFile.c" to the project. This should be in the examples directory alongside the other source files.
3) Set the stack and heap sizes as follows:

   Example 1 – default settings

   Example 2 – system stack size to 0xA000, IDRAM heap size to 0x2000.

   Example 3 – system stack size to 0x2000, IDRAM heap size to 0xA000.
4) Add the imaging library to the project (HEL_ImgLib.lib).
5) Build the project and run it. In the event you receive any errors, check the source images are accessible – Code Composer Studio may be performing disk I/O to a different directory to what you expect!
6) The most common error message is "Cannot Open File". This is caused by either CCS accessing the wrong directory, or insufficient space being available on the stack. To change directory with CCS, use File>Open and open a file in the directory the source image is in.

This will build the example using the standard Code Composer Studio I/O system – using JTAG to perform all data transfers. This is easy to do, but very slow – it can take minutes to transfer each image. To switch to using the Server/Loader:

1) Use the "Create Project" tool to create a new project, again named "ExampleX.mak" – eg Example1.mak for Example 1. This time, enable Server/Loader support. It will issue several warnings about files already existing – don't worry, you are creating new versions!
2) For Example 1 & Example 2, the default stack settings should work. For Example 3, the System Stack size should be set to 0x1800, while the IDRAM heap size should be set to 0xA000.
3) As before, add the library and HrnFile.c to your project. You may not need to do this if you started with the JTAG-based version above.

4) Open file "Example.h" in your editor. Find the line:

   #define HESL 0

   Edit this to set HESL to 1. This will enable server/loader support within the code. Rebuild the application.

5) Launch the Server/Loader plug-in in Code Composer. Select file "NetworkX" as the network file (eg Network1 for Example 1). Note that the network file as supplied is for a single HERON module on an HEPC8 – you may need to edit this to match your configuration.

6) Click "Start S/L".

7) When Server/Loader has completed loading the code Code Composer should display the first line of C – debug as normal from here. This is several orders of magnitude faster than JTAG!

Using the Examples without the "Create New Project" Tool

It is possible to use the examples without using the "Create New Project" Tool. To do this, you will need to:

1. Create a new BIOS project using Code Composer Studio's File>New>New DSP/BIOS Configuration option. Ensure you select the correct base for your hardware, and that you set global characteristics like clock speed, processor type etc.
2. Create the correct memory map for your hardware. Ensure that the on-chip RAM segment is named IDRAM and the bulk store is named SDRAM.
3. Set the SDRAM heap size to be as large as possible. Also, set the IDRAM heap to the sizes discussed for use with JTAG.
4. Set the system stack size as outlined earlier. Also set the TSK0 and TSK_IDLE stacks to 0x400.
5. Ensure all interrupt handlers are initialised properly for your hardware.
6. Add ExampleX.c to that project
7. Add HrnFile.c to that project
8. Add the imaging library to the project
9. Open the "Example.h" file. Modify "#define HESL 1" to read:
10. #define HESL 0
11. Add any other required libraries for your configuration
12. Build the project and run it. In the event you receive any errors, check the source images are accessible – Code Composer Studio may be performing disk I/O to a different directory to what you expect!

It should be noted that this approach is very slow on many platforms. The JTAG path used in the emulator offers Kbyte/second bandwidth, rather than the Mbyte/second performance of the PCI bus.

Example 1 – Creating & Loading Images

Example1 loads an image from disk, negates all the pixels, and stores the image back to disk. In doing so, it shows the basics of using the library – creating image objects and simple processing of them.

The procedure followed is:

Operation	Description
HEL_ImageCreate	Initialise an image object. The memory for the structure was allocated by the HEL_ImageObj declaration earlier.
HEL_ImageReadBMPInfo	Read the header of the bitmap. We pass this a pointer to the image object we've just created. After the call, the image object describes the image on disk.
HEL_ImageCreate	We re-initialise the image object to reflect the image on disk. As part of this call, we allocate memory using the memalign() function.
HEL_ImageReadBMP	Read the bitmap into the image object we've created and the memory we allocated. Note that the read will add a border to the image if it does not meet the library's size restrictions, so the image in memory may not be the same size as the original.
HEL_ImageNOT	Invert all the pixels in the image through the NOT operator. This is done in-place – the destination image is the same as the source.
HEL_ImageWriteBMP	Save the processed image to disk. This image may be slightly larger than the original – the ReadBMP operator may enlarge the image to make sure it has valid line lengths.
free	Frees the image buffer we allocated using memalign. Not strictly necessary here as we're about to quit, but nice for tidiness!

Running under Server/Loader, this process is almost instantaneous; using Code Composer Studio's stdio routines it may take several minutes. This is to be expected when doing image I/O through CCS.

The example is best accessed as a CCS project – Example1.mak, located in the Examples directory.

Example 2 – Region of Interest Processing and Copying Images

In this example we introduce the concepts of Region of Interest and copying.

Region of Interest (ROI) selection allows us to pick an area of a large image for processing. By selecting a small area, we can implement high performance algorithms, without wasting time on pixels we don't care about. Copying the image is important too – we can bring images on-chip for fast access.

In the example, we select a Region of Interest (ROI) and copy it into frame buffer on-chip for processing. Once processed, we put it back into the image, but as a "picture in picture" in the top right corner.

The procedure followed is:

Operation	Description
DMA_claim	Claim a DMA from the framework. In these examples, this is stubbed out; in any real application, it will be necessary to request the DMA resource from a framework before use.
HEL_ImageCreate	As Example 1
HEL_ImageReadBMPInfo	As Example 1
HEL_ImageCreate	As Example 1
HEL_ImageReadBMP	As Example 1
HEL_ROICreate	Initialise an image object to describe the ROI (Region of Interest). The underlying image object is exactly the same as an image, but the function allows easier set-up of the object.
HEL_ImageCreate	Initialise an image object on-chip. This gives us access to on-chip RAM. We'll copy the ROI into this later for processing. There's no need to copy the ROI – it could be processed in place. However, this is faster than processing in SDRAM…
HEL_ImageCOPY	Use the DMA to copy the ROI into the on-chip buffer we declared. The DMA was claimed from the framework earlier.
HEL_ImageCOPYCHK	Wait for the DMA to complete.
HEL_ImageANDK	Now we have the image on-chip we can perform an AND function. We AND the image with 0x80…
HEL_ROICreate	Initialise the ROI to somewhere else in the image. When we copy the ROI data back to the image, it will have moved…
HEL_ImageCOPY	Copy the ROI data back to the original framebuffer. As before, we wait for the DMA to complete.
HEL_ImageWriteBMP	As Example 1
Free	As Example 1

As before, this is accessed through the CCS project Example2.mak.

Example 3 – Convolution

Here we use ROI and copying to load an image, select an area, and filter it using some pre-defined convolution filter kernels. This example also shows how the ROI and Image objects need only be initialised once, typically at start-up – they do not need to be initialised for every frame processed.

The convolution function only operates on 16-bit data, so the function also converts from 8 -> 16-bit pixels and back again.

The procedure followed is:

Operation	Description
DMA_claim	As Example 2
HEL_ImageCreate	As Example 1
HEL_ImageReadBMPInfo	As Example 1
HEL_ImageCreate	As Example 1
HEL_ImageReadBMP	As Example 1
HEL_ROICreate	As Example 2
HEL_ROICreate	The second ROICreate initialises a second ROI object for the destination for the processed data. We do this earleir as we're doing it statically – all initialisation can be done before we enter the frame-handling loop.
HEL_ImageCreate	(Repeated 4 times) The first two calls create 8-bit buffers for the ImageCOPY functions. The second paiur create two 16-bit buffers. Note that the memory is actually re-used – an 8-bit buffer can exist in the first half of each 16-bit buffer.
LOOP STARTS	
Grab_Image	Fake function – the start of the per-frame processing. Take an image from the "camera".
HEL_ImageCOPY HEL_ImageCOPYCHK	Copy ROI on-chip
HEL_ImageCONVERT	Convert the image from the fromat of the off-chip image to the format for processing – in this case from 8-bit to 16-bit.
HEL_ImageCONV	Convolve the image using the 3x3 kernel Kern33HPF – defined in the header HEL_ImageFilter.h…
HEL_ImageCONVERT	Convert the filtered data back to 8-bit
HEL_ImageCOPY	Copy the filtered data back to the original framebuffer. As before, we wait for the DMA to complete.
HEL_ImageWriteBMP	As Example 1
Free	As Example 1

As before, this is accessed through the CCS project Example3.mak.

22nd February 2001: BIOS support added to examples.

14th February 2001: Updated with performance figures for v4 compiler

28th February 2001: Updated examples with BIOS support and fixed some v4 compiler issues.

Technical support for HUNT ENGINEERING products should first be obtained from the comprehensive Support section www.hunteng.co.uk/support/index.htm on the HUNT ENGINEERING web site. This includes FAQs, latest product, software and documentation updates etc. Or contact your local supplier - if you are unsure of details please refer to www.hunteng.co.uk for the list of current re-sellers.

HUNT ENGINEERING technical support can be contacted by emailing support@hunteng.demon.co.uk, calling the direct support telephone number +44 (0)1278 760775, or by calling the general number +44 (0)1278 760188 and choosing the technical support option.

If you are in North America, South America or Canada, contact our strategic partner Traquair Data Systems at www.traquair.com/company/support.html for support information and contact details.

N.B. Technical support for this Library (HEL_Ilib) will be provided for users of HUNT ENGINEERING hardware ONLY.