Celoxica

# Platform Developer's Kit

## PixelStreams Manual

Authors: RG

Document number: 1

Customer Support at http://www.celoxica.com/support/

| | | |
|---|---|---|
| Celoxica in Europe | Celoxica in Japan | Celoxica in the Americas |
| T: +44 (0) 1235 863 656 | T: +81 (0) 45 331 0218 | T: +1 800 570 7004 |
| E: sales.emea@celoxica.com | E: sales.japan@celoxica.com | E: sales.america@celoxica.com |

# Contents

# Conventions

The following conventions are used in this document.

Warning Message. These messages warn you that actions may damage your hardware.

Handy Note. These messages draw your attention to crucial pieces of information.

Hexadecimal numbers will appear throughout this document. The convention used is that of prefixing the number with '0x' in common with standard C syntax.

Sections of code or commands that you must type are given in typewriter font like this:

```
void main();
```

Information about a type of object you must specify is given in italics like this:

copy ***SourceFileName DestinationFileName***

Optional elements are enclosed in square brackets like this:

`struct [type_Name]`

Curly brackets around an element show that it is optional but it may be repeated any number of times.

***string*** ::= "{***character***}"

# Assumptions & Omissions

This manual assumes that you:

- have used Handel-C or have the Handel-C Language Reference Manual
- are familiar with common programming terms (e.g. functions)
- are familiar with MS Windows
- are familiar with standard image processing and machine vision terminology

This manual does not include:

- instruction in VHDL or Verilog
- instruction in the use of place and route tools
- instruction in the use of DK Design Suite
- detailed description of image processing algorithms

# 1 Introduction

PixelStreams is a library of parameterisable IP for creating video processing systems. IP blocks (known as filters) are assembled into filter networks, connected by streams.

Filter networks may be assembled programmatically, in Handel-C, or graphically using the PixelStreams GUI. Although the GUI is useful as a training tool, we strongly recommend graduating to a programmatic style as this greatly increases the flexibility of the system.

PixelStreams supports a range of pixel formats, from traditional machine vision formats such as 1-bit and 8-bit monochrome, to display and broadcast formats such as 8-bit RGB and YCbCr, and a signed 16-bit high dynamic range format supporting image manipulation at increased precision. The library supports both progressive and interlaced scan types, as well as allowing for random access processing (where possible). All combinations of TV and VGA, input and output are supported.

The core PixelStreams library is architecture and platform independent. Each platform (board) supported by PixelStreams has an additional support library providing the specialized sinks (video outputs) and sources (video inputs) for that platform. Where external RAM support is needed (such as for framebuffers), the PAL APIs are used.

The PixelStreams architecture makes it easy to create custom filters for performing specialized image processing operations. The most difficult aspect of creating pipelined hardware is often handling flow control correctly: the PixelStreams architecture effectively eliminates these issues by providing re-usable flow control components.

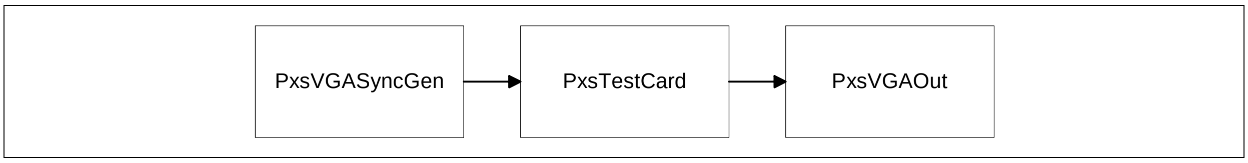
PixelStreams is designed primarily for dealing with high-speed video input processing and analysis, and high-speed back-end video generation and display. Many machine vision systems feature higher-level intermediate stages (such as object tracking), which typically handle much lower data rates but with more algorithmic processing. These intermediate stages are not currently addressed by PixelStreams, as they are typically better suited to embedded microprocessors. Finally, whilst PixelStreams can be used for still image processing, the high data rate, highly parallel nature of the generated hardware may be area inefficient for such tasks.

## 1.1 Theory of operation

Pixel values, and associated synchronization and coordinate (screen position) information, flow through streams from sources to sinks. A network operates synchronously, with each stream passing one datum per clock cycle (a datum may be flagged as invalid, to allow for rates of processing slower than one per clock cycle). A single datum can in theory contain multiple pixels, although this is not supported by the current library. Multiple networks running in different clock domains can be connected together to allow multi-rate processing.

## 1.2 Quick start

A simple filter network which displays a test card might be created as follows:



VGA compatible sync pulses and coordinates are generated by the source filter, `PxsVGASyncGen()`. `PxsTestCard()` then generates a test pattern stream by outputting an appropriate pixel color for each coordinate received. It also copies the sync pulses and coordinates from its input to its output. `PxsVGAOut()` then displays the resulting image on a VGA monitor.

The examples in this section are reproduced using the ***PixelStreams GUI*** (see page 32). This is a good way to get started quickly with PixelStreams.

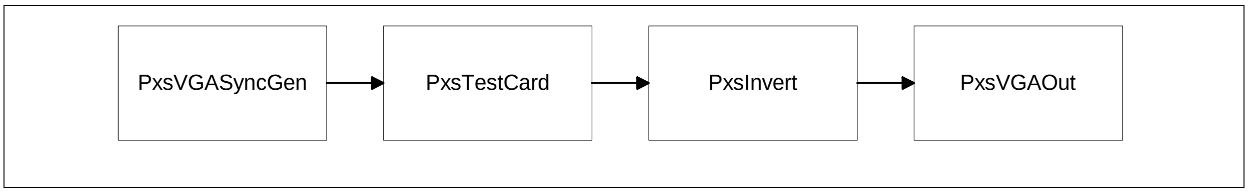
The source code for implementing the above network is provided in the "TestCard" example, and follows a standard declarative form for a simple filter network:

```
{ platform specifics }
#include "pxs.hch"
void main (void)
{
    { stream declarations }
    { other declarations }

    par
    {
        { filter instantiations }
        { non-filter processing }
    }
}
```

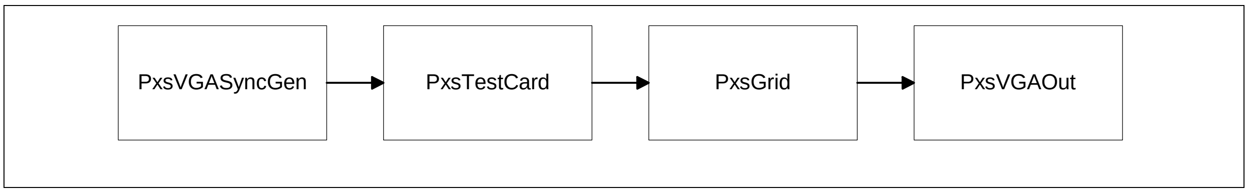
A typical stream declaration is `PXS_PV_S (Out, PXS_RGB_U8)`, which declares a stream called "Out", with progressive VGA sync type, synchronous coordinates, and unsigned 8-bit RGB pixels. ***Declaring streams*** (see page 18) covers the syntax of stream declarations. A typical filter instantiation is `PxsTestCard (&VGASync, &Out, Width, Height)`, where `VGASync` and `Out` are previously declared streams (which are always passed to filters as pointers). Input streams always come before output streams. Width and Height are compile-time parameterization of the filter. Parameter arguments always come after stream arguments.

The image can be altered by adding filters between the `PxsTestCard()` and `PxsVGAOut()` filters:



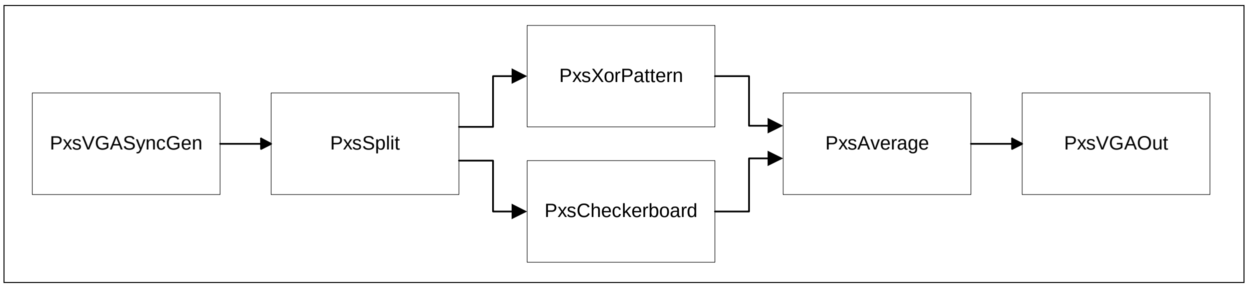
The image displayed on the monitor will be the inverse of the previous image. Note that the displayed image will remain correctly positioned on screen as the coordinates and sync pulses are an integral part of the stream.

Some filters can overlay images onto other streams, for example:



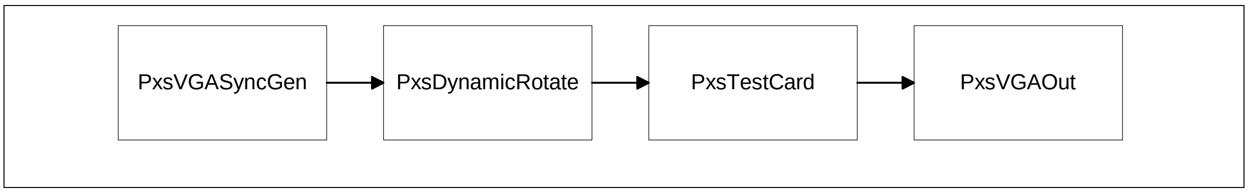
The image displayed on the monitor will be the test card, with a grid overlayed on it.

Streams can be split and then merged:



The image displayed on the monitor will be an equal blend of a checkerboard and XOR pattern. Note however that this technique will only work correctly if the latency between the two filters is identical (in this case, 1 cycle). If it is not, the two streams will become "skewed" with respect to each other. This can be corrected automatically by using the `PxsSynchronise()` filter.

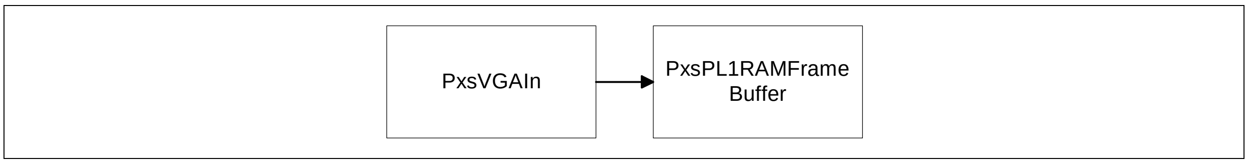
The coordinate component of streams can be manipulated independently of the pixel and sync components. For example:



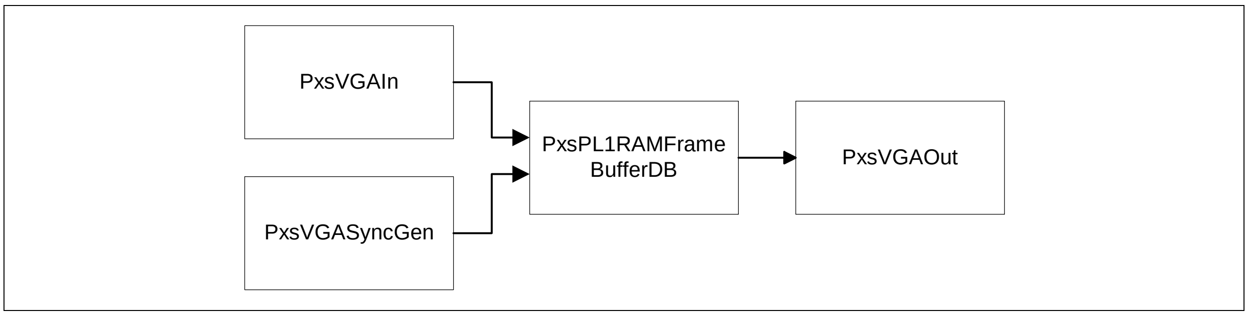
The image displayed on the monitor will be the test card, rotated by an amount specified by the "Angle" argument to `PxsDynamicRotate()`. Note that if the `PxsDynamicRotate()` filter appeared AFTER the `PxsTestCard()` then the image would not be rotated - VGA displays ignore the coordinate component and display in the usual raster scan order. This idea of pre-transforming coordinate streams before doing image lookups is key to geometric transforms within PixelStreams.

Not all filters can accept transformed ("asynchronous") coordinates - all filters check at compile time that they are compatible with the streams connected to their terminals. Untransformed coordinates can be re-created from sync pulses by the `PxsRegenerateCoord()` filter.

Video capture is also straightforward:

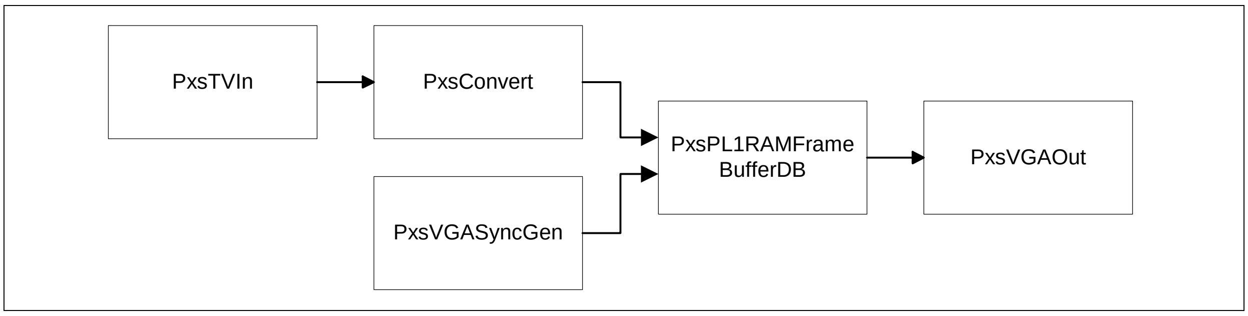


Will capture a VGA (or DVI as appropriate) image and store it in an external RAM framebuffer. However, in this example, no image will be displayed. To add display, the filter network will need to be:



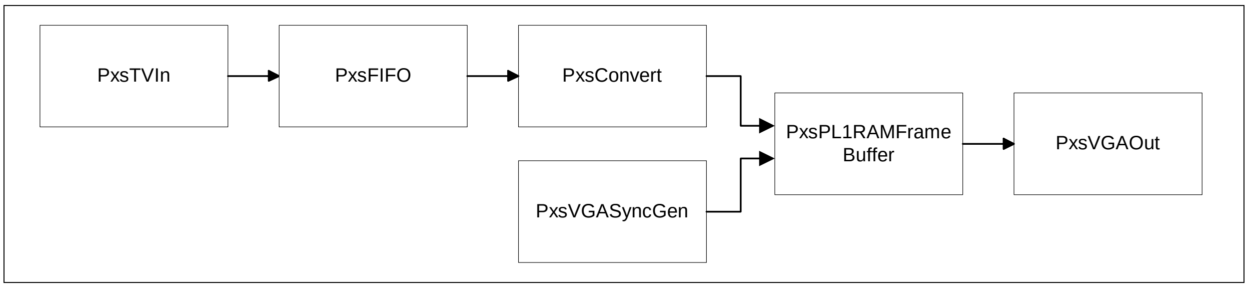
Note that we use a "DB" (double buffered) frame buffer here as both the video input and video output are competing for RAM bandwidth. The stream created by `PxsVGAIn()` is stored directly into one bank of RAM, whilst the coordinate stream created by `PxsVGASyncGen()` is used to fetch pixels from the framebuffer and pass them to the output.

TV input is handled similarly, except that it requires the introduction of a color-space converter:



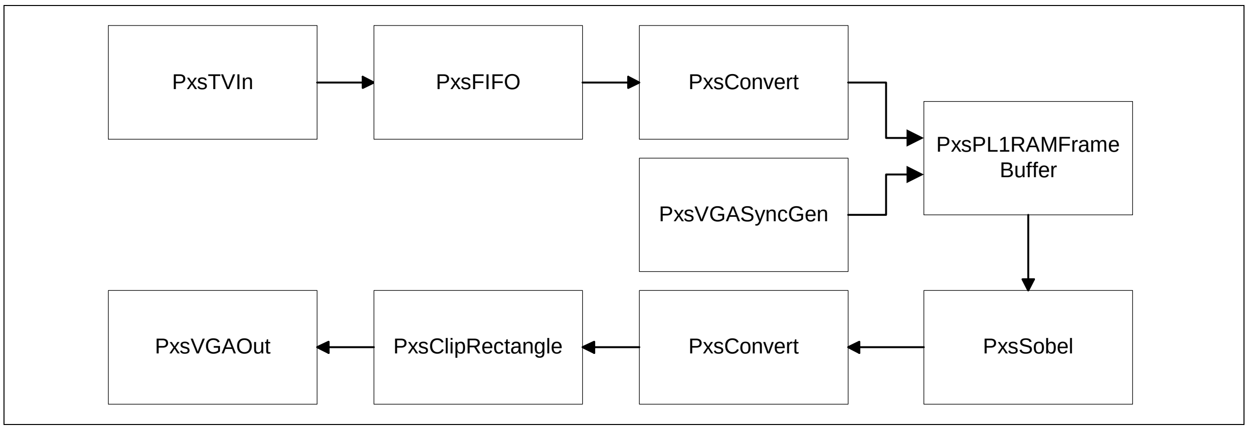
The TV input is in luminance-chrominance format (YCbCr), whereas VGA display is in RGB format. Here, we have chosen to do the color-space conversion before storing the image in the framebuffer, but this is not mandatory - the color-space converter could instead be inserted just before `PxsVGAOut()`. The framebuffers (like most PixelStreams filters) are polymorphic, which means they adapt automatically to the stream types they are connected to. For example, the `PxsConvert()` filter requires no parameters: it automatically deduces the conversion required from the types of the streams connected to its terminals.

The above example requires two RAM banks, which is somewhat inefficient. If a design is running at a sufficiently high clock rate, it is possible to share a single framebuffer between TV input and VGA display. This is because VGA display has a blanking period during each line in which no pixels need to be fetched. This provides sufficient time to store new pixels that have arrived from the TV input. In order to buffer up these new pixels, we need to introduce a `PxsFIFO()` filter. When the single-buffered framebuffer needs to fetch pixels for display, it "blocks" its other input. This stops the FIFO from outputting pixels to its output stream. Once the blanking period is reached, the input is unblocked, and the FIFO provides the pixels to be written to the framebuffer. The FIFO must be of sufficient length to hold all the pixels that can have arrived from the TV input during the visible period, otherwise pixels will be lost.



The image displayed will be of the TV input. To the right of and below the image will probably be some random data, as the VGA display resolution will not match the size of the input TV image, and so the framebuffer will fetch junk data from the RAM. This can be fixed by introducing a `PxsClipRectangle()` block to reduce the visible region-of-interest of the screen. This can be placed either directly before the VGA output (which will result in the same number of pixels being fetched, but some being then discarded), or between the sync generator and the framebuffer (which will result in fewer pixels being fetched).

This basic network provides the basis for simple video processing. For example, Sobel edge detection:



The image fetched from the framebuffer is sent through a Sobel edge-detector. This outputs a stream with monochrome pixels of type "signed 16". This type allows for the larger range of values created by some filters. For example, without scaling the maximum value from the Sobel edge detector is 2040, well outside what can fit into an 8-bit range. Using a signed 16-bit pixel type avoids the need for intermediate scaling or saturation. This type is then converted to a viewable form by a further `PxsConvert()` filter, clipped and then displayed. The image is that of the edge detected video input.

An enormous variety of video processing, analysis and generation is possible by composing filters into networks. PixelStreams is shipped with a large number of examples, covered in ***Description of examples*** (see page 28).

# 2 Streams

A stream is a container through which data is passed. It should have exactly one source filter and one sink filter attached to it. A stream consists of three top-level components:

- A "Valid" flag (flows downstream, from source to sink)
- One or more data components (flows downstream)
- A "Halt" flag (flows upstream, from sink to source)

On any given cycle, the Valid flag indicates whether the data components are valid or not. Valid data appears exactly once (and for exactly one clock cycle). The Halt flag indicates a request by a downstream filter that an upstream filter should stop producing valid data as soon as possible. A filter is described as "haltable" if it can respond to a halt request immediately, i.e. if on cycle x Halt is asserted, then Valid will be 0 on cycle x+1. No filter can respond more quickly than this as the Valid flag is registered.

The data component of a stream consists of:

- An "Active" flag (required)
- Pixels (optional)
- Coordinates (optional)
- Sync pulses (optional)

The Active flag is asserted in the current "region-of-interest" of the image. The coordinates component gives the (X, Y) location the corresponding pixel. The sync pulses component gives information about end-of-line and end-of-frame (and potentially the signals needed to drive a monitor).

A stream is said to be image forming if it contains both pixels and sync pulses (capable of aligning those pixels in raster scan format).

## 2.1 Pixel types

The pixels component consists of pixel shades/colors in one of a number of possible "pixel types". The possible pixel types are:

| | |
|---|---|
| `PXS_EMPTY` | No pixels |
| `PXS_MONO_U1` | Binary (black and white) |
| `PXS_MONO_U8` | Monochrome, unsigned 8-bit greyscale |
| `PXS_MONO_S16` | Monochrome, signed 16-bit greyscale |
| `PXS_RGB_U8` | RGB, unsigned 8-bit color (total 24-bits) |
| `PXS_YCbCr_U8` | YCbCr, unsigned 8-bit color (total 24-bits) |

`PXS_EMPTY` indicates that a stream does not contain any pixels. Such streams are created by filters such as sync generators (which create coordinates and sync pulses but no image).

PXS_MONO_U1 pixels are typically the result of some binary characterization and can be processed and stored using much less space than other types. Note however that the standard framebuffers provided by PixelStreams make no effort to pack the data efficiently (they are packed as one pixel to one location). Users should create custom framebuffers if they require an alternative behaviour.

PXS_MONO_U8 pixels are used by most machine vision applications as they provide a good trade-off between dynamic range and storage/computation requirements.

PXS_MONO_S16 pixels are used where greater dynamic range is required. They can be used as when dealing with 12-bit sources such as some CameraLink cameras. In addition, they can be used to avoid (or alleviate) loss of information when doing arithmetic manipulations on images. Being signed, they can represent abstract non-intensity values such as intensity gradients.

PXS_RGB_U8 pixels are used when displaying streams on monitors, or when taking images from non-camera sources such as PCs. They consists of three 8-bit channels, R, G and B for Red, Green and Blue respectively.

PXS_YCbCr_U8 pixels are used in broadcast applications, and are defined by the ITU-R BT.601 standard. They consist of three 8-bit channels, Y (Luminance, or brightness), Cb (Chrominance-Blue) and Cr (Chrominance-Red).

A stream will contain exactly one of these types. This list may be extended in future (for example, to support packing of multiple pixels into a single datum).

### *2.1.1 Ranges of pixel types*

The range of values for the various pixel types are as follows:

- PXS_MONO_U1 has only two values 0 (defined as black) and 1 (defined as white).
- PXS_MONO_U8 has 256 values, ranging from 0 (defined as black) to 255 (defined as white).
- PXS_MONO_S16 has 65,536 values, ranging from -32768 to 32767. Most of this range does not have a direct meaning in terms of color, as we continue to define 0 as black and 255 as white. This does not preclude applications from using their own local definition of black and white (for example 0 to 4095), but should be borne in mind when using the standard conversion filters.
- PXS_RGB_U8 has three independent channels (R, G and B) each of which has 256 values, ranging from 0 to 255. Hence there are 256^3 = 16,777,216 possible values. Black is defined as the triple (0, 0, 0), and white is defined as the triple (255, 255, 255).
- PXS_YCbCr_U8 has three independent channels (Y, Cb, Cr). Y has 220 possible values ranging from 16 to 235. Cb and Cr have 225 possible values, ranging from 16 to 240. Hence there are 220*225*225 = 11,137,500 possible values. Note however that some of these do not correspond to valid colors, and in addition values outside this range are possible as a result of operations on image data. Black is defined as the triple (16, 128, 128), and white is defined as the triple

(235, 128, 128). (Note that 128 is actually offset zero for the chrominance channels, and therefore means no color offset).

### *2.1.2 Conversions between pixel types*

Conversions between the standard types respect the definitions of black and white whilst attempting to preserve as much information as possible. For example, converting `PXS_MONO_S16` to any other type will first clamp its value to the range 0 to 255.

Converting from RGB to any MONO type takes into account the relative luminance of each color channels, using the standard factors (0.299, 0.587, 0.114).

Converting between RGB and YCbCr is a slightly tricky issue. The ranges for YCbCr specified above are clearly defined by BT.601. This document also defines the ranges for digital RGB as being from 16 (black) to 235 (white). This definition is at odds with the definition used in PCs and machine vision (and as adopted above). As a result, the conversions between YCbCr and RGB are NOT as defined by BT.601. Custom filters can easily be written to implement the BT.601 conversions if desired. The conversion from RGB to YCbCr (and vice versa) is lossy, and so such conversions should be avoided wherever possible.

Please note that many compression schemes (such as JPEG) which use luminance-chrominance color spaces will often use "full scale" types where each channel can range over the full 0 .. 255. Interchanging between these and YCbCr defined above without taking account of this difference will result in under- or over-saturated images.

## 2.2 Coordinate types

A stream can contain three types of coordinates:

- `PXS_COORD_NONE`
- `PXS_COORD_ASYNCHRONOUS`
- `PXS_COORD_SYNCHRONOUS`

`PXS_COORD_NONE` indicates that the stream doesn't contain any coordinates. This is sometimes done deliberately by the user to reduce the storage or buffering requirements of a stream (coordinates can be regenerated from sync pulses, if they exist). Coordinates can be discarded from a stream simply by changing its declaration appropriately.

`PXS_COORD_ASYNCHRONOUS` indicates that the stream contains asynchronous coordinates. This means that the coordinates do not relate directly to the sync pulses. These are typically the result of a coordinate transform.

`PXS_COORD_SYNCHRONOUS` indicates that the stream contains synchronous coordinates, that is to say, the coordinates are synchronized to the sync pulses. This means that the coordinates start at (0, 0) and proceed linearly (1, 0), (2, 0) etc until the end of the line, when they go to (0, 1) and so forth. Synchronous coordinates are produced by sync generators, and are useful in indexing operations.

In either the asynchronous or synchronous cases, the coordinates consist of a Coord.X and Coord.Y component, both signed 16-bit. The large size and signed nature of coordinates allows them to be transformed (translated, rotated and warped in other ways) without encountering overflow problems.

By common convention, (0, 0) is defined to be the top-left of an image formed by a stream, increasing positively in the X direction (rightwards) and the Y direction (downwards).

## 2.3 Sync types

A stream can contain sync pulses of the following types:

- `PXS_SYNC_NONE`
- `PXS_SYNC_INTERLACED`
- `PXS_SYNC_INTERLACED_TV`
- `PXS_SYNC_PROGRESSIVE`
- `PXS_SYNC_PROGRESSIVE_VGA`

`PXS_SYNC_NONE` indicates that the stream doesn't contain sync pulses. This is an unusual stream type as it does not represent any form of raster scanned image. A stream like this will therefore tend to contain a combination of asynchronous coordinates with pixels. This stream type is created by the special `PxsPlot()` filter.

`PXS_SYNC_INTERLACED` indicates that the stream contains an interlaced image. In an interlaced image, a complete image is made up of two sequential fields; an "even" field consisting of lines 0, 2, 4, 8, ..., followed by an "odd" field consisting of lines 1, 3, 5, 7, ... . Interlaced image types are traditionally used in a broadcast setting (such as PAL, SECAM, NTSC or HDTV 1080i). Interlaced images can be converted to progressive images (which are better suited to 2D image processing) using a framebuffer.

`PXS_SYNC_INTERLACED_TV` is a strict subset of `PXS_SYNC_INTERLACED`, in which the sync pulses are compatible with a TV standard mode, and therefore capable of driving a suitable television or other output device.

`PXS_SYNC_PROGRESSIVE` indicates that a stream contains a progressively scanned image. In a progressively scanned image, each line follows on immediately from the preceding line: 0, 1, 2, 3, ... . Progressive image types are used in computer generating images (VGA or DVI), for HDTV 720p, and for most forms of 2D image processing.

`PXS_SYNC_PROGRESSIVE_VGA` is a strict subset of `PXS_SYNC_PROGRESSIVE`, in which the sync pulses are compatible with a VGA standard mode, and therefore capable of driving a suitable VGA or DVI monitor (or other output device).

In all the non-empty cases, the sync component consists of four flags:

- Sync.HSync Horizontal sync (polarity undefined)
- Sync.VSync Vertical sync (polarity undefined)
- Sync.Blank Blanking (0 = Visible, 1 = Blanked)

- Sync.Field (0 = Even, 1 = Odd)

For progressive sync types, Sync.Field is always 0.

## 2.4 Stream compatibility

An important concept when dealing with different stream types is the idea of "compatibility". For many (but not all!) filters in the PixelStreams library, it is stated that the output stream type must be compatible with the input. This is a less strict requirement than the types being identical. For example, a `PxsDelay()` filter which takes in a stream of pixel type `PXS_RGB_U8`, might have its output directed to a stream of pixel type `PXS_EMPTY`. In this case, the user is deliberately discarding pixels. On the other hand, if the input type was `PXS_EMPTY` and the output type `PXS_RGB_U8` then a compile-time error would be produced, as this filter cannot create pixels where none previously existed.

In general, each filter checks that its input streams meet its *minimum* requirements, and checks that its output streams do not exceed the *maximum* it is capable of generating (which may be dependent on the type of its inputs).

So for an input stream "In" and an output stream "Out", compatibility is defined as follows:

- Out is compatible with In when all of its components (pixel type, coordinate type, sync type) are compatible.

This therefore requires definition of compatibility for each component, as follows.

### Pixel type

Out is compatible if its pixel type is `PXS_EMPTY`, or if its pixel type is identical to In. In other words, if the output pixels are thrown away (by being used in a `PXS_EMPTY` stream), then the type of the input is irrelevant. In all other cases, the output must be the same as the input.

### Coordinate type

Out is compatible if its coordinate type is "less than" or equal to that of In. We define `PXS_COORD_NONE` < `PXS_COORD_ASYNCHRONOUS` < `PXS_COORD_SYNCHRONOUS`. So synchronous coordinates can be used as asynchronous coordinates (but not the other way around), and both types of coordinates can be thrown away by using them in an output stream of coordinate type `PXS_COORD_NONE`.

### Sync type

Out is compatible if its sync type is "less than" or equal to that of In. In this setting:

```
PXS_SYNC_NONE < PXS_SYNC_INTERLACED < PXS_SYNC_INTERLACED_TV
```

and

```
PXS_SYNC_NONE < PXS_SYNC_PROGRESSIVE < PXS_SYNC_PROGRESSIVE_VGA
```

So a stream containing sync pulses capable of driving a TV or monitor can be passed into a stream containing sync pulses that cannot. All sync information can be thrown away by outputting into a stream containing sync pulses of type PXS_SYNC_NONE. Interlaced and progressive sync pulses are never compatible.

## 2.5 Declaring streams

Streams are declared explicitly, with the user required to specify exactly the type of coordinates, sync pulses and pixels contained by the stream. Although this is somewhat cumbersome, it has two advantages:

1. The stream type can be checked statically by the filters it is connected to, such that the filter can be sure that it is receiving all the components it requires at its input stream(s), and is creating all the components required by its output stream(s).
2. The filters can be polymorphic; they can recognize the type of their inputs, and (at compile time) modify their behaviour accordingly.

Streams can be declared with each component type specified explicitly as follows:

```
PXS_STREAM_DECLARE_STATIC (Name, PixelType, SyncType, CoordType);
```

Where Name is the identifier name to be declared, and PixelType, SyncType and CoordType are the type enumerations as previously defined. So a typical declaration might be:

```
PXS_STREAM_DECLARE_STATIC (MyStream, PXS_RGB_U8, PXS_PROGRESSIVE_VGA,
                           PXS_COORD_SYNCHRONOUS);
```

In addition PXS_STREAM_DECLARE_STATIC, there exists PXS_STREAM_DECLARE (which may only be used for global streams that are being exported to other files), and PXS_STREAM_DECLARE_EXTERN (for importing global streams exported from other files).

This form of declaration is rather cumbersome, and so a number of short-hands exist. These follow the form:

PXS_***SyncType_CoordType*** (***Name***, ***PixelType***);

Where ***SyncType*** is one of:

| | |
|---|---|
| N | PXS_SYNC_NONE |
| I | PXS_INTERLACED |
| IT | PXS_INTERLACED_TV |
| P | PXS_PROGRESSIVE |
| PV | PXS_PROGRESSIVE_VGA |

and ***CoordType*** is one of:

| | |
|---|---|
| N | `PXS_COORD_NONE` |
| A | `PXS_COORD_ASYNCHRONOUS` |
| S | `PXS_COORD_SYNCHRONOUS` |

So the example declaration above is reduced to:

```
PXS_PV_S (MyStream, PXS_RGB_U8);
```

In practice, relatively few stream types are commonly used. As a guide, these are:

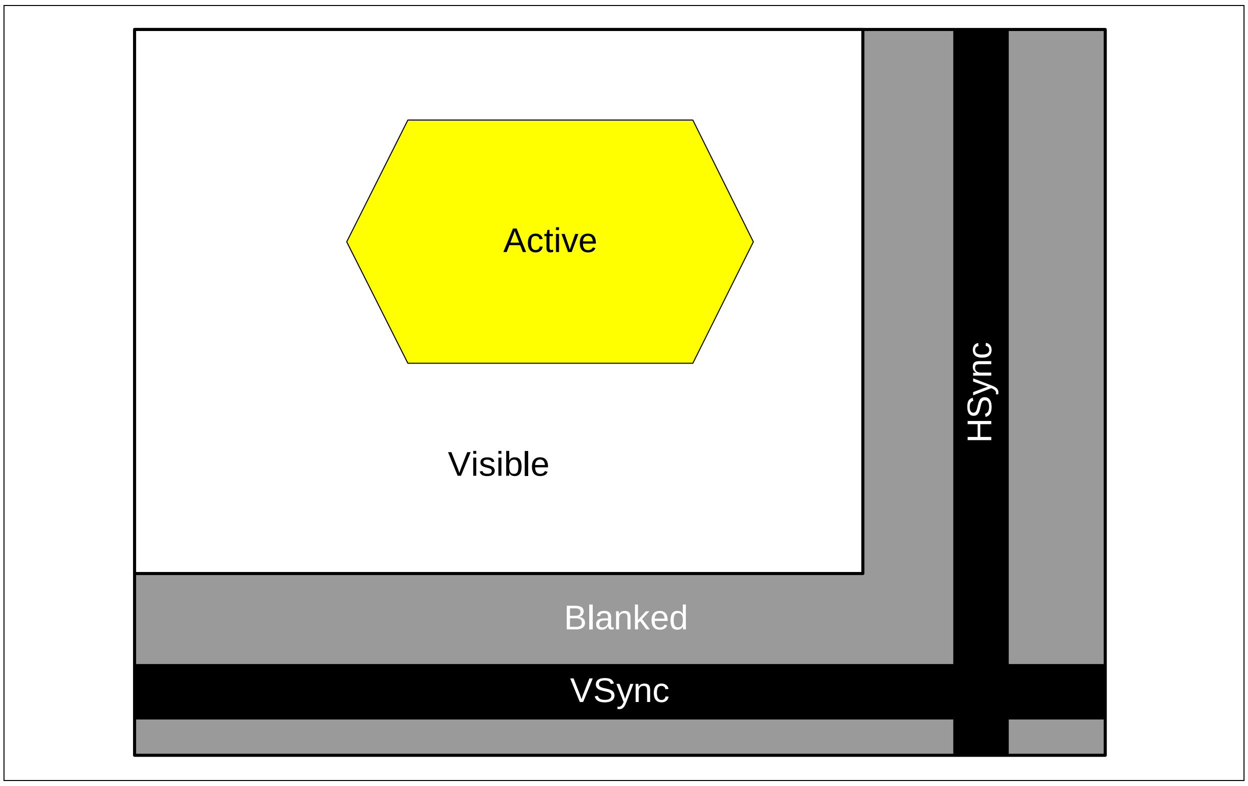
| | |
|---|---|
| `PXS_I_S` | Typically from TV Input |
| `PXS_P_S` | Typically from VGA Input |
| `PXS_IT_S` | Created by TV sync generator, suitable for TV output |
| `PXS_PV_S` | Created by VGA sync generator, suitable for VGA output |
| `PXS_PV_A` | VGA with transformed coordinates, also suitable for VGA output |
| `PXS_N_A` | Created by `PxsPlot()` filter |

In addition, two auxiliary macros exist for custom filter writers:

- `PXS_SAME (Name, Stream)` declares a new stream called "Name" with the same types as "Stream".
- `PXS_GENERIC (Name)` declares a new stream called "Name" with invalid properties - this is a shorthand that can be used for streams in a pipeline whose types will not be checked.

## 2.6 Image formation

In an image forming stream (that is, one which contains pixels and sync pulses), the total scanned area consists of a number of overlapping regions, as shown.



In all regions, valid datum are indicated by the "Valid" flag being 1.

The "active" region is the area to be processed (also known as a "Region-Of-Interest" or ROI). In an unclipped image, the active region is the same as the visible region. In this region, the Active flag is 1, and the Sync.Blank flag is 0. This region may be of any shape, and always lies within the visible region.

The "visible" region is the allowable area in which visible pixels may occur. In this region, the Sync.Blank flag is 0. (In other words, this is the exact inverse of the blanked region). This region is always rectangular in shape, stretching from synchronous coordinates (0, 0) to (VisibleWidth - 1, VisibleHeight - 1), where VisibleWidth and VisibleHeight are the size of rectangular image formed.

The "blanked" region is the area that continues horizontally beyond each line and vertically beyond each frame. Although coordinates do not need to be valid in this region (even for PXS_COORD_SYNCHRONOUS streams), they conceptually cover two overlapping regions: horizontal blanking, from (VisibleWidth, 0) to (TotalWidth - 1, TotalHeight - 1), and vertical blanking from (0, VisibleHeight) to (TotalWidth - 1, TotalHeight - 1). In this region, the Active flag is 0 and the Sync.Blank flag is 1. This region can be any shape, although it is typically rectangular.

The final two regions are those in which horizontal and vertical sync are asserted. Horizontal sync is asserted for a brief period (>= 1 datum) in the horizontal blanking region defined above. Vertical sync is asserted for a period (>= 1 line) in the vertical blanking region. Active data separated by horizontal sync are on different lines, whilst active data separated by vertical sync are on different frames (or fields, in the case of interlaced sync types). In both regions, the Active flag is 0 and the Sync.Blank flag is 1. The polarity of Sync.HSync and Sync.VSync is not defined, and so an end-of-line or end-of-frame condition can only be detected by looking for transitions in these flags.

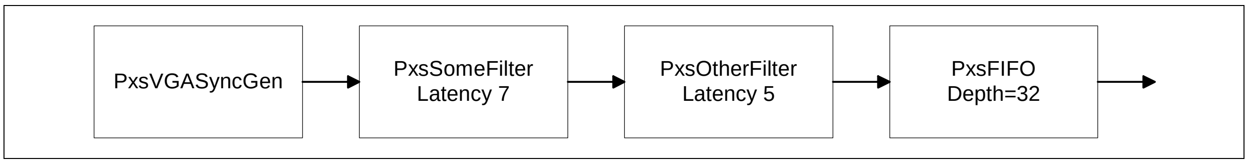
## 2.7 Ensuring pixels are not lost

In order to never lose data, a given filter (or network of filters) must satisfy several conditions:

1. It must always accept and process (or store) valid data at its inputs.
2. The source driving the filter (or network) must be haltable.
3. It must assert Halt "*n*" cycles before it will run out of space to store valid data, where *n* is the number of cycles of latency between its input and the haltable source.

Individually satisfying these conditions for every filter would be extremely cumbersome. Instead, most filters act as simple pipelines, passing the Valid flag through along with the (processed) data associated with it (with the same number of cycles of latency as the data), and passing the Halt flag upstream (with 0 cycles of latency).

Therefore, in order to build a lossless filter network, a sequence of filters is typically finished with a `PxsFIFO()` filter, who's depth is just over double of latency of the sequence. This is because the FIFO will assert "Halt" when it is more than half-full (and similarly, de-assert it when it is less than half-full). So a network such as:



Will never lose pixels, as it's behaviour will be:

- If the output of the FIFO is halted, it will stop producing valid data on the next cycle and start to fill up.
- Once it has 16 items in it, it will assert Halt, which will propagate immediately back up to the sync generator.
- The sync generator will stop producing valid data on the next cycle.
- The remaining (7+5)=12 items in the two filters will propagate through to the FIFO, with it eventually having (16+12)=28 elements in it. The network will then be fully halted.
- Once the output of the FIFO is un-halted (unblocked), it will start to output valid data on the next cycle.

- Once the FIFO has only 16 items in it, it will de-assert Halt, which will propagate immediately back up to the sync generator.
- The sync generator will start producing valid data, which will arrive at the FIFO 12 cycles later, before it is completely empty.

In this way, once the network has started up, its output is always ready and able to provide data at one per cycle, and never loses data.

# 3 Writing custom filters

Most filters follow a fairly standard pattern:

```
macro proc PxsMyFilter (In, Out, Parameter)
{
    // internal streams
    PXS_GENERIC(Stage0);
    PXS_GENERIC(Stage1); // ... more

    // other declarations
    // ...

    // assertions checking the types of input and output streams
    PXS_EXPECT_PIXEL_TYPE (In, PXS_RGB_U8);
    PXS_PROVIDE_COORD_ASYNCHRONOUS (Out);
    // ...

    par
    {
        // single-cycle (per-datum) processing pipeline
        // with 3 cycles of latency
        while (1)
        {
            PxsCopyVAPCSH (In, &Stage0);
            // processing for first stage of pipeline


            PxsCopyVAPCSH (&Stage0, &Stage1);
            // processing for second stage of pipeline


            PxsCopyVAPCSH (&Stage1, Out);
            // processing for last stage of pipeline
        }

        // multi-cycle processing, e.g. per frame
        while (1)
        {
            // ...
        }
    }
}
```

There are several conventions to note here:

- Streams are always passed to filters by reference (i.e. as pointers)
- In the declarations of filters, the argument lists are always in the order:
  - Input streams
  - Output streams
  - Parameters (constant or variable)

- If there is only one input stream, it is usually named `In`.
- If there is only one output stream, it is usually named `Out`.
- Filters with latency greater than one have internal streams that are declared with `PXS_GENERIC`.
- Input streams are checked to ensure that they provide the pixel, sync and coord types needed by the filter. Consult the pxs_private.hch header file for a list of such macros.
- Output streams are checked to ensure that they do not expect any component types not provided by the filter. Consult the pxs_private.hch header file for a list of such macros.
- The stream processing is all done with a single cycle pipeline of n stages.
- Each stage reads data only from the stream directly before it, and writes data only to the stream directly after it.
- `PxsCopy*()` macros can be used to copy stream components that are not being altered. There are a number of macros providing common subsets of the letters "VAPCSH", where "VAPCSH" corresponds to Valid, Active, Pixel, Coord, Sync and Halt. The first five components are copied from input to output, the last component is copied from output to input. So, a stage that alters pixel data but keeps all other components intact would use the `PxsCopyVACSH()` macro.
- Multi-cycle processing (such as per-frame updates) are handled in parallel with the stream processing, so that they do not affect the stream flow.

The contents of the processing at each stage are entirely dependent on the processing that is being performed. As PixelStreams is provided with source of the complete library of standard filters, we recommend using this as a guide for implementing custom filters. A good grounding in Handel-C programming is essential; please consult the DK manuals and consider attending one of Celoxica's Handel-C training courses.

# 4 Platform specifics

The only platform specific filters within the PixelStreams libraries are those concerned with getting video streams in and out of the design. In all cases, designs should be linked with the pxs.hcl core library, and the appropriate platform specific library given below.

## 4.1 RC200 / RC200E / RC203 / RC203E

### Libraries

Select the appropriate one of pxs_rc200.hcl, pxs_rc200e.hcl, pxs_rc203.hcl, pxs_rc203e.hcl

### PxsVGAIn

Platform does not support VGA input.

### PxsTVIn

Platform has a single TV input (0). This input has three configurations:

| | |
|---|---|
| 0 | Camera input |
| 1 | Composite (CVBS) input |
| 2 | S-Video (YC) input |

### PxsVGAOut

Platform has a single VGA output (0). This output has a single configuration (0). When the clock rate is 25.175MHz, Expert (E) type boards will have the TFT LCD screen driven in parallel with the VGA DAC. At this clock rate, only the standard 640 x 480 @ 60Hz (VGA) mode should be used.

### PxsTVOut

Platform has a single TV output (0). This output has two configurations: 0 for NTSC, and 1 for PAL.

Please note that VGA output and TV output share a DAC, and so only one may be driven at any given time.

If your application processes TV input, and you wish to display output on the RC200E / RC203E TFT LCD, please see the "MultiDomain" example, which shows efficient use of RAM bandwidth using multiple clock domains.

## 4.2 RC300 / RC300E

### Libraries

Select the appropriate one of pxs_rc300.hcl or pxs_rc300e.hcl

### PxsVGAIn

Platform has two identical DVI inputs (0 and 1), each with a single configuration (0). When run with PxsVGAIn, an EDID slave is also instantiated allowing a host to query the board parameters. Note that these inputs only handle true DVI-D digital data, not DVI-I analogue VGA or VGA input via an adaptor.

### PxsTVIn

Platform has two TV inputs (0 and 1), each with two configurations: 0 for decoding composite (CVBS) inputs, and 1 for decoding S-Video (YC) inputs.

### PxsVGAOut

RC300 has two video outputs (0 and 1), driving outputs 0 and 1 respectively. Both VGA and DVI are driven in parallel. RC300E (Expert) has three video outputs. Output 0 drives the TFT LCD, and should only be run at a clock rate of 65MHz with the standard 1024 x 768 @ 60Hz (XGA) mode. Outputs 1 and 2 drive VGA/DVI outputs 0 and 1 respectively. For both boards, each output has exactly one configuration.

### PxsTVOut

Platform has two identical TV outputs (0 and 1), each with two configurations: 0 for NTSC, and 1 for PAL.

## 4.3 Simulation

The simulation platform re-uses the PAL Sim virtual platform to provide video input and output. At this time, only static images can be fed into the video input.

### Library

pxs_sim.hcl

### PxsVGAIn

Platform simulates four identical VGA inputs, each with 5 configurations, offering a range of resolutions:

| | |
|---|---|
| 0 | 640x480 |
| 1 | 800x600 |
| 2 | 1024x768 |
| 3 | 1280x1024 |
| 4 | 1600x1200 |

### PxsTVIn

Platform simulates four identical TV inputs, each with 3 configurations, offering a range of resolutions:

| | |
|---|---|
| 0 | 720x480 (NTSC SDTV) |
| 1 | 720x576 (PAL SDTV) |
| 2 | 1920x1080 (1080i HDTV) |

### PxsVGAOut

Platform simulates four identical VGA outputs, each capable of handling resolutions up to 1920x1200.

### PxsTVOut

Platform simulates four identical TV outputs, each capable of handling resolutions up to 1920x1200.

Tip: When simulating filter networks that include framebuffers, it is useful to pre-load the framebuffer with a frame of information. Use the simulated PL1RAM "dump now" option to dump the contents of a (filled) framebuffer to a file, then use the "load at startup" option to pre-load the framebuffer with that image for subsequent simulation runs.

# 5 Description of examples

PixelStreams comes with a large number of simple examples demonstrating the use of most of the included filters. The examples workspace can be accessed by going to Start>Programs>Celoxica>Platform Developer's Kit>PixelStreams>Examples Workspace [DK].

A description of these examples is given below.

| | |
|---|---|
| Affine | Arbitrary affine coordinate transforms. Requires RC300. Coefficents are entered using FTU3 "RegisterMap" functionality, in signed 9.8-bit fixed point. For example, coefficients { 256, 0, 0, 0, 256, 0 } is the identity transform. |
| Analysis | Display the results of the `PxsAnaylse()` filter. A histogram of pixel intensity (value) is overlayed, alongside a console display of average pixel value, etc. In addition, two overlayed cross hairs show the location of (one of) the minimal and maximal valued pixels. |
| Blend | Show a series of different blends. The two images being blended are the framebuffered input TV 0, and the standard test card. Also demonstrates the multiplexing and resynchronization of multiple streams. |
| Clip | Demonstrate clipping of pixels before and after framebuffering. A clipping circle bounces around the screen, only allowing pixels within its radius to be updated. The image from the framebuffer is then clipped to a rectangle before display. |
| Console | Demonstrate the `PxsConsole()` filter and its associated utility macros. This is useful for overlaying status information onto a video stream. |
| Convolution | 3x3 convolution with arbitrary filter coefficients. Requires RC300. Coefficents (plus shift and scale factors) are entered using FTU3 "RegisterMap" functionality. See the convolution.hcc source code for some sample coefficients to try (initial screen will be black!). |
| CustomCoord | An example of a custom coordinate transform. Coordinates are sinusoidally warped in both horizontal and vertical directions before being fed into the standard test card generator. |
| Dither | Dither a test card to a four shades of red, eight of green and two of blue (a total of 64 colors). Shows the use of ordered dither. |
| DitherVideo | Dither a TV video input to two shades each of red, green and blue (a total of 8 colors). Uses a fixed look-up-table to achieve approximate gamma correction (factor 2.0). |
| EdgeDetect | Sobel edge detection on TV input. No scaling or thresholding of the results is performed, and so the results may vary from text book examples. |
| FrameBuffer | Simple framebuffering of TV input, and display on VGA output. |
| FrameDifference | Demonstrate the use of two framebuffers for inter-frame analysis. The first framebuffer looks up from the same coordinate as the input stream. The two pixels are differenced, and the result stored in a second (display) framebuffer. |
| GUI | Demonstrate the use of the cursor (mouse pointer) overlay, and integration with the PalMouse core to provide interactive image operations. Clicking and dragging inverts an area of the test card. Using the mouse wheel selects a different mouse pointer. |

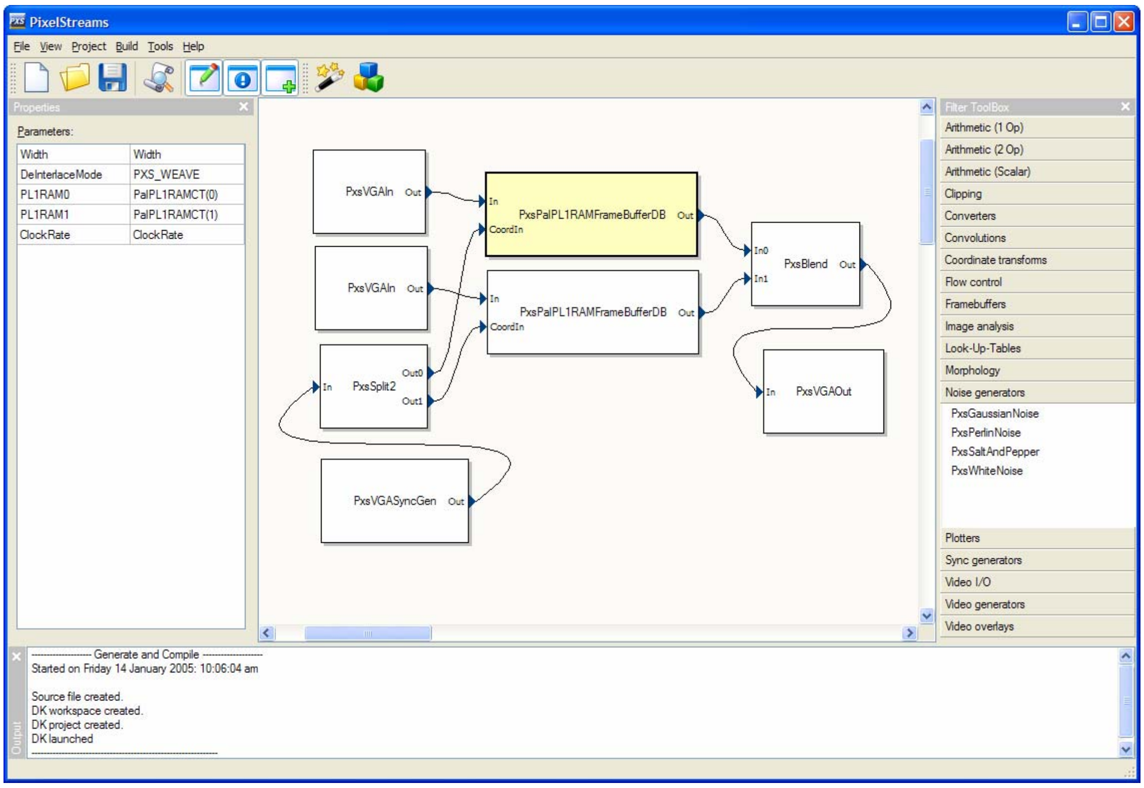
| | |
|---|---|
| HistogramEq | Dynamic histogram equalization of TV input. The input video stream is analysed to determine its histogram. The framebuffered output is then remapped using a dynamic look-up-table to achieve a approximately flat histogram of intensity (equivalently, a linear cumulative histogram). So an image with intensities concentrated in the middle of the range (one with poor contrast), will tend to have the contrast stretched. A light or dark image will be darkened or lightened respectively. This technique is a simple way of automatically compensating for poor brightness or contrast. |
| Join | Demonstrate the prioritized merging of two streams. Video input is simultaneously framebuffered and displayed. At the same time, using remaining RAM bandwidth, a simple pattern is plotted into the framebuffer. |
| LUT | Demonstrate use of static look-up-tables (LUTs) to transform color values. The color channels of a TV image are independently transformed, resulting in an unusually colorful display. |
| LabelBlobs | Label connected dark objects in the image, and plot green squares around their bounding boxes. Works best on images such as text and logos. |
| Laplacian5x5 | Perform a 5x5 Laplacian (high pass) convolution on a TV input. Demonstrates the use of generic 5x5 convolvers. |
| MedianFilter | Overlay salt-and-pepper (impulse) noise onto a TV input, and then attempt to remove it using linear filtering (Gaussian blur) and non-linear filtering (median filtering). The probability of noise varies with time. At low levels of noise the median filter is clearly superior to the Gaussian filter in terms of image degradation. |
| Morphology | Demonstrate simple greyscale morphology operators (erosion and dilation) on a TV input. The output image alternates between the two. Erosion tends to make bright objects smaller and dark ones larger, whilst dilation does the opposite. |
| MotionBlur | Demonstrates video feedback and blending to achieve frame averaging. The last frame is blended with the new one to create a new image which is stored in a second framebuffer. The effect is to average out noise and transient effects in the image. |
| MultiDomain | Passing of streams across clock domains for multi-rate processing. TV input is framebuffered in a 65MHz clock domain is transmitted to a display in a 25.175MHz clock domain. Flow control correctly manages the transmission of data and the blocking of the sending process. |
| Noise | Show a variety of different noise sources. The display alternates between white noise and Gaussian noise, dynamic and fixed pattern. The type of noise is overlayed using a scaled console, and the R, G, B histograms of the noise are overlayed at the bottom of the screen. |
| PerlinRipple | Framebuffer TV input, but displace lookups using a Perlin noise function, resulting in a slow rippling effect. |

| | |
|---|---|
| Plot | Plot a simple pattern, using the `PxsPlot()` filter attached to a framebuffer. |
| Pong | Simple bat-and-ball game using video overlay filters. |
| Reader | Pre-load an external RAM with an image drawn using ordinary PalPL1RAM accesses, then display it using a PixelStreams filter network. |
| Rotate | Display a rotating test card. Demonstrates simple coordinate transformations, translation and rotation. |
| RotateVGA | Framebuffer and dynamically rotate VGA input. |
| Scale | Demonstrate coordinate scaling, upsampling a TV input to VGA display size. |
| SelectLUT | Cycle through a number of different look-up-table color value transformations, displaying the function at each stage. |
| Sharpen | Sharpen the TV image and display on the VGA output. This uses the built-in `PxsSharpen3x3()` filter, which as a typical linear filter is quite prone to amplifying noise. |
| Stereo | Framebuffer two TV inputs, and combine them into a single output, with input 0 being turned into purple and input 1 being turned into green. |
| TVOut | Simple test card on TV output. |
| TestCard | Simple test card on VGA output. |
| VGAIn | Framebuffer and display the VGA input. Uses double buffered framebuffer to provide sufficient RAM bandwidth. Clock rate needs to be higher than the dot clock of the input, otherwise pixels may be dropped. If the clock rates are extremely close (for example, XGA input at 65MHz), pixels may be dropped in periodic ways, giving rise to unusual "wipe" effects. |
| VGAtoTV | Framebuffer the VGA input, and display it on the TV output. Demonstrates multi-rate design techniques. |
| VideoGen | Demonstrate a sequence of different video generators and overlays, changing between an XOR pattern, a checkerboard, a grid and a bouncing ball. |

# 6 PixelStreams GUI

The PixelStreams GUI is supplied to enable users to quickly and efficiently create hardware applications which use the PixelStreams library.

The GUI provides an environment in which filters and streams from the PixelStreams library can be linked together create a system. This system can then be built to create a Handel-C source file and a DK project and workspace.



| | |
|---|---|
| Width | Width |
| DeInterlaceMode | PXS_WEAVE |
| PL1RAM0 | PalPL1RAMCT(0) |
| PL1RAM1 | PalPL1RAMCT(1) |
| ClockRate | ClockRate |

```
------------------ Generate and Compile ------------------
Started on Friday 14 January 2005: 10:06:04 am

Source file created.
DK workspace created.
DK project created.
DK launched
-----------------------------------------------------------
```

The main features are:

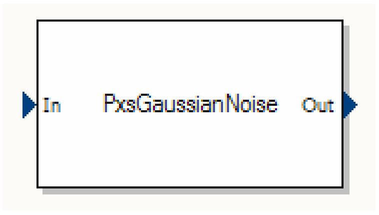
- Easy to use graphical display.
- Generates Handel-C source code and project for Celoxica's DK Design Suite.
- Easy to upgrade with extra features to the PixelStreams library.
- Targets Celoxica RC200, RC203 and RC300 series boards, and simulation libraries.

To run the PixelStreams GUI you will need

- Celoxica's PDK (version 3.1 or later).
- Celoxica's DK Design Suite (version 3.1 or later).

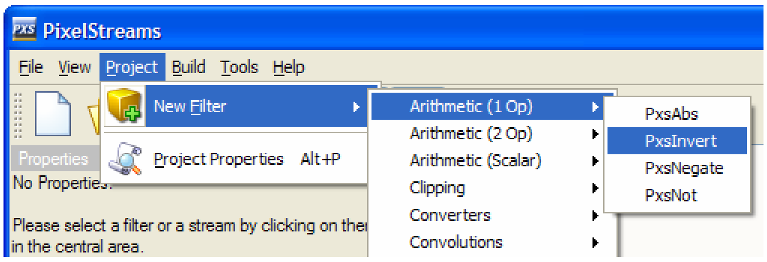
## 6.1 Adding a filter

Filters are represented in the GUI as blocks with a number of inputs and outputs:

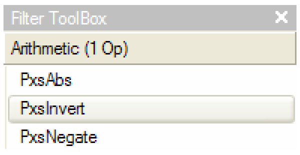


Filters can be added in three ways:

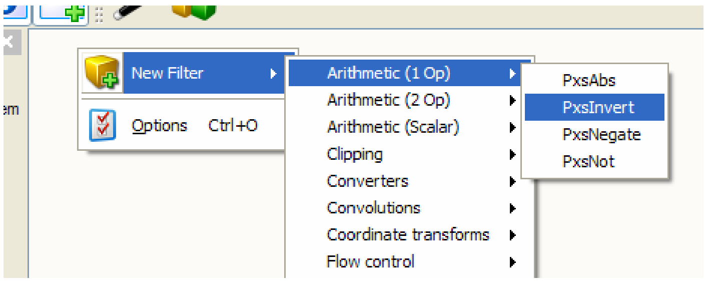
- By selecting them from the Project>New filter menu:



- By selecting them from the "Filter ToolBox" that appears on the right of the GUI at startup. This can be shown or hidden from the View menu or toolbar icons:



- By selecting them from a context menu on the canvas (right click):

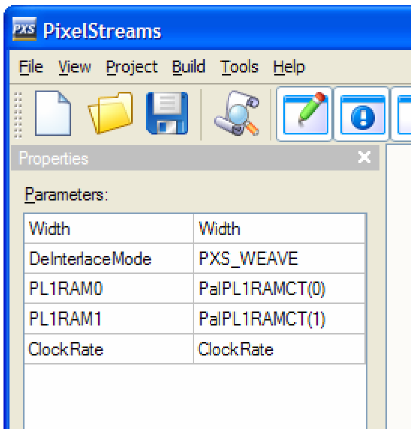


In all cases, the filter is added to the top-left of the current canvas and can be left-clicked and dragged into position.

To remove a filter from your design, open the filter context menu by right-clicking over the required filter, and select Remove Filter.

### Properties

Once the filter is created, its name and properties can be set via the Filter Properties window, which is by default docked on the left hand side. If the properties dock window is open, clicking on the filter will select its properties for viewing. If the properties dock window is not open, double clicking will open it and select the desired properties for viewing.

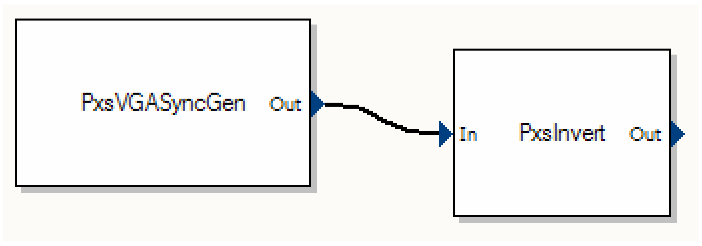


| | |
|---|---|
| Width | Width |
| DeInterlaceMode | PXS_WEAVE |
| PL1RAM0 | PalPL1RAMCT(0) |
| PL1RAM1 | PalPL1RAMCT(1) |
| ClockRate | ClockRate |

This window allows users to set the parameters for this filter by typing in the table.

## 6.2 Creating a stream

Streams connect filters together, allowing the pixel data or other data to flow between the filters. To create a stream an output port must be clicked and the stream is dragged to rest over an input port. If the stream is not dragged to rest over an input port it will be automatically deleted.

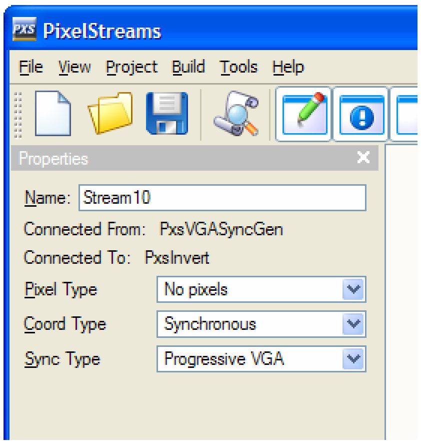


If a stream has been joined to the wrong filter or input port, it can be dragged to another. Select the stream by clicking on the input port area, and drag it to the correct port.

To remove a stream from your design, open the stream context menu by right-clicking on the required stream, and select Remove Stream.

### Properties

Once the stream is created, its name and properties can be set via the Stream Properties window. If the properties dock window is open, clicking on the stream will select its properties for viewing. If the properties dock window is not open, double clicking will open it and select the desired properties for viewing.



| | |
|---|---|
| Name: | Stream10 |
| Connected From: | PxsVGASyncGen |
| Connected To: | PxsInvert |
| Pixel Type | No pixels |
| Coord Type | Synchronous |
| Sync Type | Progressive VGA |

This window allows the user to set the name of the stream, and the type of data it will carry. The name entered will be the declaration name of the stream in the generated Handel-C code. The pixel type, coordinate type and the sync type of the data stream can also be set using the respective drop down boxes.

The PixelStreams GUI uses heuristics to insert the most likely types of stream required in designs, and to propagate those types throughout the design. The user can override these types at any time. The GUI does not check types before generating the Handel-C design, and so it is possible to create designs which fail to compile in DK.

## 6.3 Generating and compiling the design

The PixelStreams GUI can generate any design to create Handel-C source code, with its corresponding project and workspace.

### Generating Handel-C

The Handel-C source can be generated by clicking on the above icon on the main toolbar or selecting Generate Code from the Build menu. Its shortcut key is F7.

### Generating projects and launching DK

The Handel-C source plus a DK project and workspace can be generated by clicking on the compile button, or selecting Compile from the Build menu (shortcut key F5). This also launches DK, from where an appropriate configuration can be selected and final compilation of the design can be started.

The project is created with seven pre-set configurations targeted at specific boards: RC200, RC200E, RC203, RC203E, RC300, RC300E and Sim.

## 6.4 File menu

The file menu is accessible from the toolbar and it has the following options:

- New - Closes the current project and creates a new blank project with the specified name.
- Open - Opens a previously saved project file and loads the design.
- Close - Closes the current project.
- Save - Saves the project to the previously specified location.
- Save As - Saves the project to a new file.
- Exit - Exits the PixelStreams GUI application.

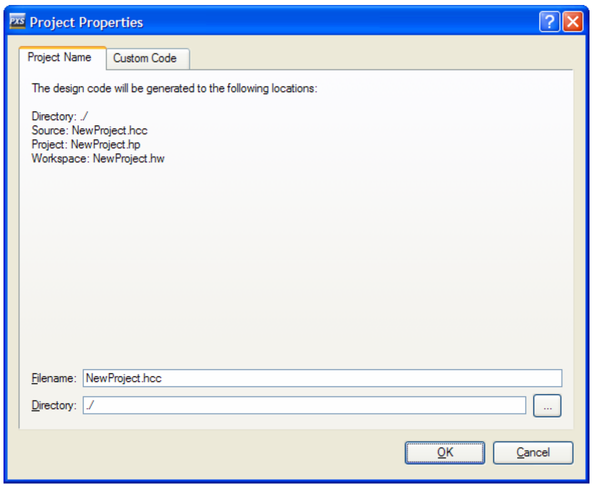
## 6.5 Project properties

The project properties dialog allows the user to set certain values and options for the current design. It is accessible by selecting Project > Project Properties from the menu, or clicking this icon:

The Project Properties dialog has four tabs:

- Name - Specifies the name of the design and the output directory for generation.
- Custom Code - Allows the user to modify and add new custom code to be inserted into the generated design.

## The Name tab



Project Properties

Project Name | Custom Code

The design code will be generated to the following locations:

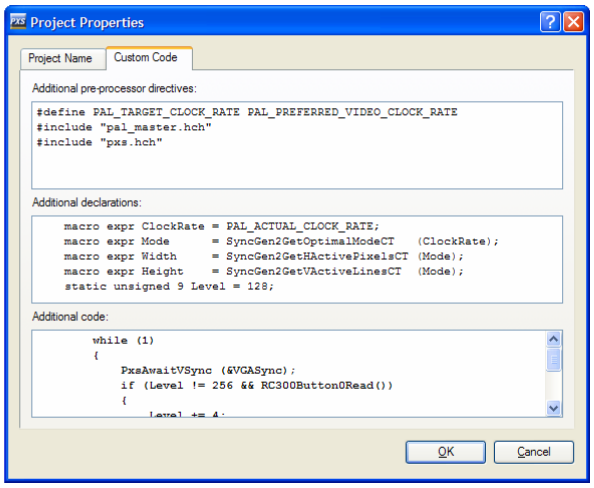
Directory: ./
Source: NewProject.hcc
Project: NewProject.hp
Workspace: NewProject.hw

Filename: NewProject.hcc

Directory: ./

The name tab allows you to change the name of the design and the directory in which the output files will be generated. It also shows the names of the DK project and workspace that will be generated.

## The Custom Code tab



Project Properties

Project Name | Custom Code

Additional pre-processor directives:

```
#define PAL_TARGET_CLOCK_RATE PAL_PREFERRED_VIDEO_CLOCK_RATE
#include "pal_master.hch"
#include "pxs.hch"
```

Additional declarations:

```
    macro expr ClockRate = PAL_ACTUAL_CLOCK_RATE;
    macro expr Mode      = SyncGen2GetOptimalModeCT   (ClockRate);
    macro expr Width     = SyncGen2GetHActivePixelsCT (Mode);
    macro expr Height    = SyncGen2GetVActiveLinesCT  (Mode);
    static unsigned 9 Level = 128;
```

Additional code:

```
        while (1)
        {
            PxsAwaitVSync (&VGASync);
            if (Level != 256 && RC300Button0Read())
            {
                Level += 4;
```

This page allows you to modify the boilerplate code that will be inserted into the generated Handel-C.

Code entered into the "Additional pre-processor directives" box will be inserted at the top of the generated file. This can be used to modify the clock rate or include other header files.

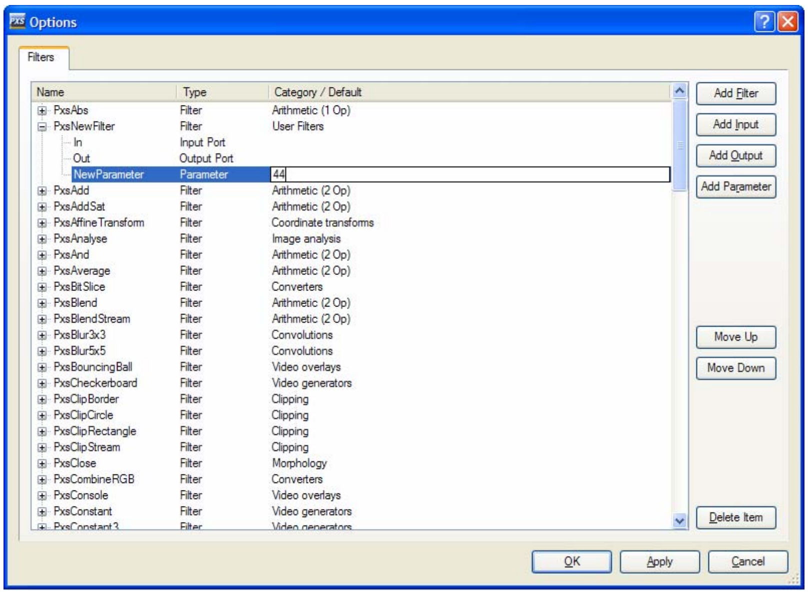
Code entered into the "Additional declarations" box will be inserted in the declarations section of the `main()` function. This can be used to declare additional variables, which can then be provided as parameters to filters.

Code entered into the "Additional code" box will inserted after the generated filter instantiations. This can be used to add per-frame behaviour, or other non-stream processing.

## 6.6 Options

The PixelStreams options dialog, accessible under Tools > Options, sets global values and settings for the PixelStreams GUI. The dialog has the one tab, which allows the library of filters to be modified and added to.

### Filters tab



| Name | Type | Category / Default |
|---|---|---|
| PxsAbs | Filter | Arithmetic (1 Op) |
| PxsNewFilter | Filter | User Filters |
| In | Input Port | |
| Out | Output Port | |
| NewParameter | Parameter | 44 |
| PxsAdd | Filter | Arithmetic (2 Op) |
| PxsAddSat | Filter | Arithmetic (2 Op) |
| PxsAffineTransform | Filter | Coordinate transforms |
| PxsAnalyse | Filter | Image analysis |
| PxsAnd | Filter | Arithmetic (2 Op) |
| PxsAverage | Filter | Arithmetic (2 Op) |
| PxsBitSlice | Filter | Converters |
| PxsBlend | Filter | Arithmetic (2 Op) |
| PxsBlendStream | Filter | Arithmetic (2 Op) |
| PxsBlur3x3 | Filter | Convolutions |
| PxsBlur5x5 | Filter | Convolutions |
| PxsBouncingBall | Filter | Video overlays |
| PxsCheckerboard | Filter | Video generators |
| PxsClipBorder | Filter | Clipping |
| PxsClipCircle | Filter | Clipping |
| PxsClipRectangle | Filter | Clipping |
| PxsClipStream | Filter | Clipping |
| PxsClose | Filter | Morphology |
| PxsCombineRGB | Filter | Converters |
| PxsConsole | Filter | Video overlays |
| PxsConstant | Filter | Video generators |
| PxsConstant3 | Filter | Video generators |

If you have added or changed functionality in the PixelStreams library, is it possible to add or change the available filter types within the PixelStreams GUI.

Filter, stream and parameter names, as well as filter categories and parameter defaults can all be edited in place by first selecting the appropriate line and then clicking a second time on the text.

New filters can be added with the "Add Filter" button, input and output streams and parameters are added to the filters similarly. Stream and parameter ordering can be changed with the "Move Up" and "Move Down" buttons. All selectable items can be removed with the "Delete Item" button.

Care must be taken when changing values on the filters tab. The filters refer to macros that are supplied with the PixelStreams library. The addition of filters or ports must only be done if the PixelStreams library has also been altered to reflect these changes.

At this time, the GUI does not support adding or removing heuristics from filter blocks.

The complete library is saved in a file named pxs_filters_user.xml in the startup directory if the GUI. If this becomes corrupted, it may be deleted, in which case the original pxs_filters.xml will be used. However, all changes to the filter library will be lost.

## 6.7 GUI examples

The GUI is supplied with a number of examples, in the subdirectories under ***`PDKInstallDir`***`\Examples\PixelStream\PixelStreamsGUI`. These are as follows:

- HelloWorld: shows use of custom code and the PxsConsole() overlay.

And the following examples based on the introductory tutorial (in order):

- TestCard
- InvertedTestCard
- Overlay
- SplitMerge
- Rotate
- VGAIn
- TVIn
- TVInFIFO
- Sobel

# 7 Standard filters

The PixelStreams library contains a large number of standard filters, in the following categories:

- Arithmetic (1-op)
- Arithmetic (2-op)
- Arithmetic (Scalar)
- Clipping
- Converters
- Convolutions
- Coordinate transforms
- Flow control
- Framebuffers
- Image analysis
- Look-Up-Tables (LUTs)
- Morphology
- Noise generators
- Plotters
- Sync generators
- Video I/O
- Video generators
- Video overlays

## 7.1 Arithmetic (1-op)

1-op arithmetic filters are those that transform the pixel values of a single input stream into a single output stream, according to some transfer function.

- PxsAbs: absolute value
- PxsInvert: inversion
- PxsNegate: negation
- PxsNot: bitwise not

### 7.1.1 PxsAbs: absolute value

```
#include "pxs.hch"
macro proc PxsAbs (In, Out);
```

#### Input Streams

***In:*** Must contain pixels of type `PXS_MONO_S16`.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Outputs the absolute value of the input pixel values.

#### Example

Used internally by the `PxsSobel()` filter.

### 7.1.2 PxsInvert: inversion

```
#include "pxs.hch"
macro proc PxsInvert (In, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Inverts the stream with respect to the white value, i.e. output value is (255 - input value). This is identical to PxsNot() for every pixel type except PXS_MONO_S16.

#### Example

See the "GUI" example.

### *7.1.3 PxsNegate: negation*

```
#include "pxs.hch"
macro proc PxsNegate (In, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Negates the stream, i.e. output value is -(input value).

### 7.1.4 PxsNot: bitwise not

```
#include "pxs.hch"
macro proc PxsNot (In, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Performs a bitwise NOT on the pixel values.

## 7.2 Arithmetic (2-op)

2-op arithmetic filters are those that transform the pixel values of two synchronized input streams into a single output stream, according to some transfer function.

- PxsAdd/PxsAddSat: image addition
- PxsAnd: bitwise and
- PxsAverage: image average
- PxsBlend: image blending
- PxsMax: image maximum
- PxsMin: image minimum
- PxsMul: image multiplication
- PxsOr: bitwise or
- PxsSub/PxsSubSat: image subtraction
- PxsXor: bitwise xor

### *7.2.1 PxsAdd/PxsAddSat: image addition*

```
#include "pxs.hch"
macro proc PxsAdd    (In0, In1, Out);
macro proc PxsAddSat (In0, In1, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Adds the pixel values of two streams together.

Subtracts the pixel values of stream ***In1*** from the pixel values of stream ***In0***.

The "Sat" variant include saturation to pixel value limits, avoiding overflow.

The output stream contains exactly the same sync and coordinate information as In0. In0 and In1 should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a PxsSynchronise() filter. If the output stream is halted, both input streams are halted.

#### Example

See the "Blend" example.

### 7.2.2 PxsAnd: bitwise and

```
#include "pxs.hch"
macro proc PxsAnd (In0, In1, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Perform bitwise AND on the pixel values of the two streams.

The output stream contains exactly the same sync and coordinate information as In0. In0 and In1 should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a `PxsSynchronise()` filter. If the output stream is halted, both input streams are halted.

#### Example

See the "VideoGen" example.

## 7.2.3 PxsAverage: image average

```
#include "pxs.hch"
macro proc PxsAverage (In0, In1, Out);
```

### Input Streams

***In:*** Must contain pixels.

### Output Streams

***Out:*** Must be compatible with ***In***.

### Latency

1 cycle

### Haltable

No

### Description

Generate the average pixel value of two streams (rounding down).

The output stream contains exactly the same sync and coordinate information as In0. In0 and In1 should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a `PxsSynchronise()` filter. If the output stream is halted, both input streams are halted.

### *7.2.4 PxsBlend: image blending*

```
#include "pxs.hch"
macro proc PxsBlend (In0, In1, Out, Level);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Level:*** Constant or variable, of type unsigned 9

#### Latency

3 cycles

#### Haltable

No

#### Description

Blend two streams together to create a single output stream. The mix of the two streams is determined by the value of ***Level***, with 0 corresponding to the exactly ***In0***, 256 corresponding to exactly ***In1***, and a linear combination for values within this range.

The output stream contains exactly the same sync and coordinate information as ***In0***. ***In0*** and ***In1*** should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a PxsSynchronise() filter. If the output stream is halted, both input streams are halted.

#### Example

See the "Blend" example.

### *7.2.5 PxsMax: image maximum*

```
#include "pxs.hch"
macro proc PxsMax (In0, In1, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Select the maximum of the pixel values of the two streams.

The output stream contains exactly the same sync and coordinate information as In0. In0 and In1 should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a `PxsSynchronise()` filter. If the output stream is halted, both input streams are halted.

#### Example

See the "Blend" example.

### 7.2.6 PxsMin: image minimum

```
#include "pxs.hch"
macro proc PxsMin (In0, In1, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Select the minimum of the pixel values of the two streams.

The output stream contains exactly the same sync and coordinate information as In0. In0 and In1 should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a `PxsSynchronise()` filter. If the output stream is halted, both input streams are halted.

#### Example

See the "Blend" example.

### 7.2.7 PxsMul: image multiplication

```
#include "pxs.hch"
macro proc PxsMul (In0, In1, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Multiply the pixel values of two streams.

The output stream contains exactly the same sync and coordinate information as In0. In0 and In1 should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a `PxsSynchronise()` filter. If the output stream is halted, both input streams are halted.

### *7.2.8 PxsOr: bitwise or*

```
#include "pxs.hch"
macro proc PxsOr (In0, In1, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Perform bitwise OR on the pixel values of the two streams.

The output stream contains exactly the same sync and coordinate information as In0. In0 and In1 should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a `PxsSynchronise()` filter. If the output stream is halted, both input streams are halted.

### 7.2.9 *PxsSub/PxsSubSat: image subtraction*

```
#include "pxs.hch"
macro proc PxsSub    (In0, In1, Out);
macro proc PxsSubSat (In0, In1, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Subtracts the pixel values of stream ***In1*** from the pixel values of stream ***In0***.

The "Sat" variant includes saturation to pixel value limits, avoiding overflow.

The output stream contains exactly the same sync and coordinate information as In0. In0 and In1 should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a PxsSynchronise() filter. If the output stream is halted, both input streams are halted.

#### Example

See the "Blend" example.

### 7.2.10 PxsXor: bitwise xor

```
#include "pxs.hch"
macro proc PxsXor (In0, In1, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Perform bitwise XOR on the pixel values of the two streams.

The output stream contains exactly the same sync and coordinate information as In0. In0 and In1 should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a `PxsSynchronise()` filter. If the output stream is halted, both input streams are halted.

## 7.3 Arithmetic (Scalar)

Scalar arithmetic filters are those that transform the pixel values of a single input stream into a single output stream, according to some transfer function involving a scalar parameter (which may be a variable or a constant).

- PxsSaturate: saturate levels
- PxsScalar*: scalar arithmetic

### *7.3.1 PxsSaturate: saturate levels*

```
#include "pxs.hch"
macro proc PxsSaturate (In, Out, Lower, Upper);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Lower/Upper:*** Constant or variable, of the same type as the pixel component of ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Limits the pixel values to the range Lower .. Upper (inclusive), clamping values outside this range to the appropriate limit.

### 7.3.2 PxsScalar*: scalar arithmetic

```
#include "pxs.hch"
macro proc PxsScalarAdd        (In, Out, Scalar);
macro proc PxsScalarAdd3       (In, Out, Scalar0, Scalar1, Scalar2);
macro proc PxsScalarAddSat     (In, Out, Scalar);
macro proc PxsScalarAddSat3    (In, Out, Scalar0, Scalar1, Scalar2);
macro proc PxsScalarSub        (In, Out, Scalar);
macro proc PxsScalarSub3       (In, Out, Scalar0, Scalar1, Scalar2);
macro proc PxsScalarSubSat     (In, Out, Scalar);
macro proc PxsScalarSubSat3    (In, Out, Scalar0, Scalar1, Scalar2);
macro proc PxsScalarMul        (In, Out, Scalar);
macro proc PxsScalarDiv        (In, Out, Scalar);
macro proc PxsScalarLeftShift  (In, Out, ShiftBits);
macro proc PxsScalarRightShift (In, Out, ShiftBits);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Scalar/Scalar0/Scalar1/Scalar2:*** Constant or variable, of the same type as the pixel component of ***In***.

***ShiftBits:*** Constant or variable, type dependent on pixel type of In: unsigned 1 (PXS_MONO_U1), unsigned 4 (PXS_MONO_U8, PXS_RGB_U8, PXS_YCbCr_U8), unsigned 5 (PXS_MONO_S16).

#### Latency

1 cycle

#### Haltable

No

#### Description

Perform scalar arithmetic on the input stream.

PxsScalarAdd*() adds to the pixel values of the input stream.

PxsScalarSub*() subtracts from the pixel values of the input stream.

The "3" variants have independent scalar values for each of the three channels. These apply only to PXS_RGB_U8 and PXS_YCbCr_U8.

The "Sat" variants include saturation to pixel value limits, avoiding overflow.

`PxsScalarMul()` multiplies the pixel values of the input stream.

`PxsScalarDiv()` divides the pixel values of the input stream.

`PxsScalarLeftShift()` shifts the pixel values of the input stream left by a number of bits (i.e. multiplies them by `2^`***`ShiftBits`***).

`PxsScalarRightShift()` shifts the pixel values of the input stream right by a number of bits (i.e. divides them by `2^`***`ShiftBits`***).

### Example

See the "Convolution" example.

## 7.4 Clipping

Clipping filters are those that affect the active region of an image, either reducing it (clipping) or increasing it (unclipping).

- PxsClipBorder: clip to remove a border
- PxsClipCircle: clip to a circle
- PxsClipStream: clip to a binary stream
- PxsUnclip: reset active region
- PxsUnclipAndBlank: blank out clipped regions

### *7.4.1 PxsClipBorder: clip to remove a border*

```
#include "pxs.hch"
macro proc PxsClipBorder (In, Out, Width, Height, Border);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Width/Height/Border:*** Constant or variable, of type signed 16

#### Latency

2 cycles

#### Haltable

No

#### Description

Clip the active region of a stream to a rectangle. The rectangle is the area (***Border***, ***Border***) to ((***Width*** - 1) - ***Border***, (***Height*** - 1) - ***Border***) inclusive. The output active region is the intersection of the rectangle with the active region of the input stream.

#### Example

See the "Clip" example.

### *7.4.2 PxsClipCircle: clip to a circle*

```
#include "pxs.hch"
macro proc PxsClipCircle (In, Out, X, Y, Radius);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***X/Y/Radius:*** Constant or variable, of type signed 16

#### Latency

4 cycles

#### Haltable

No

#### Description

Clip the active region of a stream to a circle. The circle is centered on (***X***, ***Y***) and has a radius of ***Radius*** pixels. The output active region is the intersection of the circle with the active region of the input stream.

#### Example

See the "Clip" example.

### 7.4.3 PxsClipRectangle: clip to a rectangle

```
#include "pxs.hch"
macro proc PxsClipRectangle (In, Out, X0, Y0, X1, Y1);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***X0/Y0/X1/Y1:*** Constant or variable, of type signed 16

#### Latency

2 cycles

#### Haltable

No

#### Description

Clip the active region of a stream to a rectangle. The rectangle is the area (***X0***, ***Y0***) to (***X1***, ***Y1***) inclusive. (***X0***, ***Y0***) should be the top-left coordinates, and (***X1***, ***Y1***) the bottom-right coordinates. The output active region is the intersection of the rectangle with the active region of the input stream.

#### Example

See the "FrameBuffer" example.

### 7.4.4 PxsClipStream: clip to a binary stream

```
#include "pxs.hch"
macro proc PxsClipStream (In, Out, Control);
```

#### Input Streams

***In:*** All stream Types.

***Control:*** Must contain pixels of type PXS_MONO_U1.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Clip the active region of a stream to the pixel value of another stream. The output active region is the intersection of area where the ***Control*** stream has a pixel value of 1 and the active region of the input stream.

***In*** and ***Control*** should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a PxsSynchronise() filter. If the output stream is halted, both input streams are halted.

#### Example

See the "Clip" example.

### 7.4.5 PxsUnclip: reset active region

```
#include "pxs.hch"
macro proc PxsUnclip (In, Out);
```

#### Input Streams

***In:*** Must contain sync pulses.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Set the active region of a stream to the visible area of the stream. Pixels in the clipped region may have been corrupted by upstream filters, these will now become visible. To set these pixels to black, use `PxsUnclipAndBlank()`.

### *7.4.6 PxsUnclipAndBlank: blank out clipped regions*

```
#include "pxs.hch"
macro proc PxsUnclipAndBlank (In, Out);
```

#### Input Streams

***In:*** Must contain sync pulses.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Set the active region of a stream to the visible area of the stream. Pixels in the previously clipped region are set to black.

#### Example

See the "SelectLUT" example.

## 7.5 Converters

Conversion filters are those that transform streams of one pixel type to streams of another (or perform some similar operation such as dithering).

- PxsBitSlice: bit slice extraction
- PxsCombineRGB: create an RGB stream
- PxsConvert: color-space conversion
- PxsExtractRGB: split an RGB stream
- PxsOrderedDither/PxsOrderedDither3: ordered dithering
- PxsThreshold: binary threshold

### *7.5.1 PxsBitSlice: bit slice extraction*

```
#include "pxs.hch"
macro proc PxsBitSlice (In, Out, Bit);
```

#### Input Streams

***In:*** Must contain pixels of type PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Must contain pixels of type PXS_MONO_U1.

#### Parameters

***Bit:*** Constant or variable, of type unsigned 3 (for inputs of type PXS_MONO_U8), or unsigned 4 (for inputs of type PXS_MONO_S16).

#### Latency

1 cycle

#### Haltable

No

#### Description

Extract the specified bit from the two's-complement representation of the input stream and generate a binary stream from it.

## *7.5.2 PxsCombineRGB: create an RGB stream*

```
#include "pxs.hch"
macro proc PxsCombineRGB (InR, InG, InB, Out);
```

### Input Streams

***InR/InG/InB:*** Must contain pixels of type PXS_MONO_U8.

### Output Streams

***Out:*** Sync and coord types must be compatible with ***InR***. Must contain pixels of type PXS_RGB_U8.

### Latency

1 cycle

### Haltable

No

### Description

Combine three independent R, G, B channel streams into a single RGB stream. The output stream contains exactly the same sync and coordinate information as ***InR***. ***InR***, ***InG*** and ***InB*** should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using PxsSynchronise() filters. If the output stream is halted, all input streams are halted.

### Example

See the "Stereo" example.

### 7.5.3 PxsConvert: color-space conversion

```
#include "pxs.hch"
macro proc PxsConvert (In, Out);
```

#### Input Streams

***In:*** Must contain pixels.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Must contain pixels, and these must be a different pixel type to ***In***.

#### Latency

between 1 and 4, depending on the conversion.

#### Haltable

No

#### Description

Convert one pixel type to another. All non-empty pixel types can be converted to all others, however most conversions lose some information.

***Ranges of pixel types*** (see page 14) and ***Conversions between pixel types*** (see page 15) provide more information regarding the conversions between different color spaces.

#### Example

See the "FrameBuffer" example.

### *7.5.4 PxsExtractRGB: split an RGB stream*

```
#include "pxs.hch"
macro proc PxsExtractRGB (In, OutR, OutG, OutB);
```

#### Input Streams

***In:*** Must contain pixels of type PXS_RGB_U8.

#### Output Streams

***OutR/OutG/OutB:*** Sync and coord types must be compatible with ***In***. Must contain pixels of type PXS_MONO_U8.

#### Latency

1 cycle

#### Haltable

No

#### Description

Split an RGB stream into independent R, G and B channel streams. The output streams contain exactly the same sync and coordinate information as the input streams. If any output stream is halted, the input stream is halted.

#### Example

See the "Noise" example.

## *7.5.5 PxsOrderedDither/PxsOrderedDither3: ordered dithering*

```
#include "pxs.hch"
macro proc PxsOrderedDither  (In, Out, Bits);
macro proc PxsOrderedDither3 (In, Out, BitsR, BitsG, BitsB);
```

### Input Streams

***In:*** Must contain synchronous coordinates. Must contain pixels of type `PXS_MONO_U8` (`PxsOrderedDither` only), `PXS_RGB_U8` or `PXS_YCbCr_U8`.

### Output Streams

***Out:*** Must be compatible with ***In***.

### Parameters

***Bits/BitsR/BitsG/BitsB:*** Constant ranging from 1 to 8 inclusive.

### Latency

1 cycle

### Haltable

No

### Description

Dither the input video to a given number of significant bits. Standard ordered dithering is used. The output values are padded such that the full output range is used. For example, settings Bits to 1 will dither a `PXS_MONO_U8` source to pure black and white. Setting bits to 6 will dither a `PXS_RGB_U8` source to a format suitable for driving an 18-bit display (such as some types of LCD).

Note that when dithering natural images, it is useful to apply gamma correction before dithering (since a 50% black / 50% white image will typically display much brighter than a 50% grey image). This is demonstrated by the "DitherVideo" example.

### Example

See the "Dither" and "DitherVideo" examples.

### *7.5.6 PxsThreshold: binary threshold*

```
#include "pxs.hch"
macro proc PxsThreshold (In, Out, Lower, Upper);
```

#### Input Streams

***`In:`*** Must contain pixels of type `PXS_MONO_U8` or `PXS_MONO_S16`.

#### Output Streams

***`Out:`*** Sync and coord types must be compatible with ***`In`***. Must contain pixels of type `PXS_MONO_U1`.

#### Parameters

***`Lower/Upper:`*** Constant or variable, of the same type as the pixel component of ***`In`***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Threshold the input stream. Pixel values within the range ***`Lower`*** .. ***`Upper`*** (inclusive) are transformed to white, all other values are transformed to black.

#### Example

See the "LabelBlobs" example.

## 7.6 Convolutions

Convolutions are filters where the output pixel value is dependent on a function of the region around the input pixel.

- PxsBlur3x3: blurring
- PxsBlur5x5: heavy blurring
- PxsConvolution3x3, PxsConvolutionDual3x3: arbitrary 3x3 convolutions
- PxsLaplacian3x3: high-pass filter
- PxsMedianFilter: median filtering
- PxsSharpen3x3: sharpening
- PxsSobel: edge detection

### 7.6.1 PxsBlur3x3: blurring

```
#include "pxs.hch"
macro proc PxsBlur3x3 (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

1 line and 8/9 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 3x3 Gaussian blur on the input stream. ***Width*** is the maximum active width of the input stream in pixels. The stream is convolved with the coefficients:

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 4 | 2 |
| 1 | 2 | 1 |

The result is then divided by 16.

A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

### 7.6.2 PxsBlur5x5: heavy blurring

```
#include "pxs.hch"
macro proc PxsBlur5x5 (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

1 line and 10/11 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 5x5 Gaussian blur on the input stream. ***Width*** is the maximum active width of the input stream in pixels. The stream is convolved with the coefficients:

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 6 | 4 | 1 |
| 4 | 16 | 24 | 16 | 4 |
| 6 | 24 | 36 | 24 | 6 |
| 4 | 16 | 24 | 16 | 4 |
| 1 | 4 | 6 | 4 | 1 |

The result is then divided by 256.

A two pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

## 7.6.3 PxsConvolution3x3, PxsConvolutionDual3x3: arbitrary 3x3 convolutions

```
#include "pxs.hch"
macro proc PxsConvolution3x3     (In, Out, Width, A);
macro proc PxsConvolutionDual3x3 (In, OutA, OutB, Width, A, B);
```

### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

### Output Streams

***Out/OutA/OutB:*** Sync and coord types must be compatible with ***In***. Must contain pixels of type PXS_MONO_S16.

### Parameters

***Width:*** Constant, greater than or equal to 1.

***A/B:*** Constant list-of-lists, or variable array (see description).

### Latency

1 line and 7 cycles (subject to change)

### Haltable

No

### Description

Perform an arbitrary 3x3 convolution on the input stream. ***Width*** is the maximum active width of the input stream in pixels. A is a 3x3 matrix of coefficients. For constants, these should be passed as a "list-of-lists" (see the "Laplacian5x5" example). If the coefficients are to be dynamically changed, A should be an array of 3x3 signed 16-bit integers (see the "Convolution" example). The multiplication and summation stages are done at signed 16-bit precision, so care should be taken to avoid overflow. No truncation or rounding is performed. A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

PxsConvolutionDual3x3() performs two convolutions in parallel on the same input stream. This filter is more efficient than two independent convolutions as it can share line buffer logic (and in some case tile RAMs more efficiently).

### Example

See the "Convolution" example.

### 7.6.4 PxsConvolution5x5: arbitrary 5x5 convolution

```
#include "pxs.hch"
macro proc PxsConvolution5x5 (In, Out, Width, A);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Must contain pixels of type PXS_MONO_S16.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

2 lines and 9 cycles (subject to change)

#### Haltable

No

#### Description

Perform an arbitrary 5x5 convolution on the input stream. ***Width*** is the maximum active width of the input stream in pixels. A is a 5x5 matrix of coefficients. For constants, these should be passed as a "list-of-lists" (see the "Laplacian5x5" example). If the coefficients are to be dynamically changed, A should be an array of 5x5 signed 16-bit integers (see the "Convolution" example). The multiplication and summation stages are done at signed 16-bit precision, so care should be taken to avoid overflow. No truncation or rounding is performed. A two pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

#### Example

See the "Laplacian5x5" example.

### *7.6.5 PxsLaplacian3x3: high-pass filter*

```
#include "pxs.hch"
macro proc PxsLaplacian3x3 (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be PXS_MONO_S16.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

1 line and 7/8 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 3x3 Laplacian (high pass filter) on the input stream. ***Width*** is the maximum active width of the input stream in pixels. The stream is convolved with the coefficients:

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | -8 | 1 |
| 1 | 1 | 1 |

The output value varies either side of 0, it can be offset by 128 (using PxsScalarAddSat()) for visualization purposes.

A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

### *7.6.6 PxsMedianFilter: median filtering*

```
#include "pxs.hch"
macro proc PxsMedianFilter (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

1 line and 12 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 3x3 median filter on the input stream. ***Width*** is the maximum active width of the input stream in pixels. The output value is the median of the 9 input values in the 3x3 window around the central pixel. This filter is suitable for removing impulse noise.

A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

#### Example

See the "MedianFilter" example.

### 7.6.7 PxsSharpen3x3: sharpening

```
#include "pxs.hch"
macro proc PxsSharpen3x3 (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

1 line and 7/8 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 3x3 sharpen (unsharp mask) on the input stream. ***Width*** is the maximum active width of the input stream in pixels. The stream is convolved with the coefficients:

| | | |
|---|---|---|
| -1 | -1 | -1 |
| -1 | 9 | -1 |
| -1 | -1 | -1 |

A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

Note that this filter typically produces a very noisy output image, more advanced adaptive techniques are typically recommended.

### *7.6.8 PxsSobel: edge detection*

```
#include "pxs.hch"
macro proc PxsSobel (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

1 line and 9/10 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 3x3 Sobel edge detection on the input stream. ***Width*** is the maximum active width of the input stream in pixels. The stream is convolved with the coefficients:

| | | |
|---|---|---|
| -1 | -2 | -1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

to compute the vertical gradient, and coefficients

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

to compute the horizontal gradient. The result is the sum of the absolute value of these two gradients, and is proportional to the smoothed edge intensity at a given point.

A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

#### Example

See the "EdgeDetect" example.

## 7.7 Coordinate transforms

Coordinate transform filters are those that affect the coordinate component of a stream in some way.

- PxsAffineTransform: affine transformation
- PxsDisplace: coordinate displacement
- PxsDynamicRotate: coordinate rotation
- PxsRegenerateCoord: recreate synchronous coordinates
- PxsScale: coordinate scaling
- PxsScalePower2: simple coordinate scaling
- PxsTranslate: coordinate shifting

### 7.7.1 PxsAffineTransform: affine transformation

```
#include "pxs.hch"
macro proc PxsAffineTransform (In, Out, A, IntBits, FracBits);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and pixel type must be compatible with ***In***. Coord type must be PXS_COORD_ASYNCHRONOUS.

#### Parameters

***A:*** Constant list-of-lists, or variable array.

***IntBits:*** Constant, greater than or equal to zero.

***FracBits:*** Constant, greater than or equal to zero.

#### Latency

3 cycles

#### Haltable

No

#### Description

Perform an affine transformation on the input coordinates. The parameter A must be either an 3x2 array or a list of constants of the same dimension. The transformation performed is:

*X' = (X . A[0][0]) + (Y . A[0][1]) + A[0][2];*
*Y' = (X . A[1][0]) + (Y . A[1][1]) + A[1][2];*

The values in A are in signed fixed point, arranged as (***IntBits***.***FracBits***). For purely integer transformations, ***FracBits*** should therefore be zero.

#### Example

See the "Affine" example.

### 7.7.2 PxsDisplace: coordinate displacement

```
#include "pxs.hch"
macro proc PxsDisplace (In, DeltaXIn, DeltaYIn, Out);
```

#### Input Streams

***In:*** Must contain coordinates.

***DeltaXIn:*** Must contain pixels of type PXS_MONO_S16.

***DeltaYIn:*** Must contain pixels of type PXS_MONO_S16.

#### Output Streams

***Out:*** Sync and pixel type must be compatible with ***In***. Coord type must be PXS_COORD_ASYNCHRONOUS.

#### Latency

1 cycle

#### Haltable

No

#### Description

Displace (translate) the coordinates of a stream according to the pixel values of two further streams. This is typically used to shift a coordinate stream before using it to lookup from a framebuffer.

***In***, ***DeltaXIn*** and ***DeltaYIn*** should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a PxsSynchronise() filter. If the output stream is halted, all three input streams are halted.

#### Example

See the "PerlinRipple" example.

### 7.7.3 PxsDynamicRotate: coordinate rotation

```
#include "pxs.hch"
macro proc PxsDynamicRotate (In, Out, Angle);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and pixel type must be compatible with ***In***. Coord type must be PXS_COORD_ASYNCHRONOUS.

#### Parameters

***Angle:*** Constant or variable, of type unsigned 11

#### Latency

3 cycles

#### Haltable

No

#### Description

Rotate the coordinates of a stream about (0, 0). The coordinates are rotated clockwise by an angle of *((pi * **Angle**) / 1024)* radians.

#### Example

See the "Rotate" example.

### 7.7.4 PxsRegenerateCoord: recreate synchronous coordinates

```
#include "pxs.hch"
```

```
macro proc PxsRegenerateCoord (In, Out);
```

#### Input Streams

***`In:`*** Must contain sync.

#### Output Streams

***`Out:`*** Sync type must be compatible with ***`In`***. Pixel type must be same as ***`In`***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Regenerate synchronous coordinates from sync pulses. This is typically useful in two circumstances. Firstly, if the coordinate stream has been transformed (for example, to do a rotated look up in a framebuffer), but the user then wishes to add an untransformed overlay (such as a console). Secondly, when the coordinate stream has been deliberately discarded to reduce the size of preceding filters (such as FIFOs), but is needed for some later process.

#### Example

See the "DitherVideo" example.

### 7.7.5 PxsScale: coordinate scaling

```
#include "pxs.hch"
macro proc PxsScale (In, Out, InWidth, InHeight,  OutWidth, OutHeight);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and pixel type must be compatible with ***In***. Coord type must be PXS_COORD_ASYNCHRONOUS.

#### Parameters

***InWidth/InHeight/OutWidth/OutHeight:*** Constant or variable, of type signed 16

#### Latency

3 cycles

#### Haltable

No

#### Description

Perform a simple scaling of coordinates. This is typically used to scale a coordinate stream before using it to lookup from a framebuffer. The scaling is such that:

- (0, 0) at the input maps to (0, 0) at the output.
- (***OutWidth***, ***OutHeight***) at the input maps to (***InWidth***, ***InHeight***) at the output.

This is the reciprocal of what might be imagined, but is designed for framebuffer lookups where (***OutWidth***, ***OutHeight***) is the size of the output screen and (***InWidth***, ***InHeight***) is the size of the input image.

If ***OutWidth*** and ***OutHeight*** are not constants, the hardware will be both large and slow.

#### Example

See the "Scale" example.

### *7.7.6 PxsScalePower2: simple coordinate scaling*

```
#include "pxs.hch"
macro proc PxsScalePower2 (In, Out, Power);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and pixel type must be compatible with ***In***. Coord type must be `PXS_COORD_ASYNCHRONOUS`.

#### Parameters

***Power:*** ***C***onstant or variable, of type unsigned 5

#### Latency

1 cycle

#### Haltable

No

#### Description

Perform simple power-of-two scaling of coordinates. This is typically used to scale a coordinate stream before using it to lookup from a framebuffer. The coordinate values are both divided by 2^***Power***.

#### Example

See the "Noise" example.

### 7.7.7 PxsTranslate: coordinate shifting

```
#include "pxs.hch"
macro proc PxsTranslate (In, Out, DeltaX, DeltaY);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and pixel type must be compatible with ***In***. Coord type must be PXS_COORD_ASYNCHRONOUS.

#### Parameters

***DeltaX/DeltaY:*** Constant or variable, of type signed 16

#### Latency

1 cycle

#### Haltable

No

#### Description

Perform horizontal and vertical translation of coordinates. This is typically used to shift a coordinate stream before using it to lookup from a framebuffer. The coordinate values are both shifted by (***DeltaX***, ***DeltaY***) pixels. Positive values of ***DeltaX*** and ***DeltaY*** will result in a leftwards and upwards translation of a looked up image.

#### Example

See the "Scale" example.

## 7.8 Flow control

Flow control filters are those that affect only the flow of data through streams, and not their contents.

- PxsDelay: delaying streams
- PxsFIFO: first-in-first-out buffering of streams
- PxsJoin: join two intermittent streams
- PxsLineBuffer/PxsDualLineBuffer: line buffering
- PxsMux*: multiplexing streams
- PxsMux2Stream: multiplexing controlled by streams
- PxsNonReturnValve: inhibiting flow control
- PxsRateLimiter: limit stream data rate
- PxsSend/PxsReceive: passing streams over channels
- PxsSplit*: splitting streams
- PxsSynchronise: synchronize skewed streams
- PxsValve: controlling streams

### *7.8.1 PxsDelay: delaying streams*

```
#include "pxs.hch"
macro proc PxsDelay (In, Out);
```

#### Input Streams

***In:*** All stream types

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Delay the input stream by a single cycle. Useful in balancing latency when combining previously split streams.

#### Example

See the "GUI" example.

### 7.8.2 PxsFIFO: first-in-first-out buffering of streams

```
#include "pxs.hch"
macro proc PxsFIFO (In, Out, Length);
```

#### Input Streams

***In:*** All stream types.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Length:*** Constant, maximum number of elements in the FIFO + 1

#### Latency

Variable

#### Haltable

Yes

#### Description

Implement a simple FIFO (first-in-first-out) of valid pixels. If the output of the FIFO is halted, the next cycle of output is guaranteed not to contain a valid pixel.

The input stream is halted when the FIFO becomes at least half full. Therefore, to halt a chain of filters of total latency N without losing data, ***Length*** should be at least (2 * N) + 1. This assumes that the ultimate source of the chain is itself haltable.

Pixels will be lost if the FIFO overflows.

The FIFO will buffer all components in a stream, which may in some cases use an undesirable amount of RAM. For example, buffering a stream that contains coordinates will require (***Length*** * 32) bits of RAM over the same stream without coordinates. To avoid this overhead, it is sometimes possible to strip the coordinates from a synchronous stream (by declaring it as PXS_*_N rather than PXS_*_S), and regenerate them afterwards using PxsRegenerateCoord().

#### Example

See the "FrameBuffer" example.

## *7.8.3 PxsJoin: join two intermittent streams*

```
#include "pxs.hch"
macro proc PxsJoin (In0, In1, Out);
```

### Input Streams

***In0/In1:*** All stream types.

### Output Streams

***Out:*** Must be compatible with ***In0*** and ***In1***.

### Latency

1 or 2 cycles (see below)

### Haltable

No

### Description

Join two intermittent streams together to a single output stream. Valid pixels on ***In0*** are passed to ***Out*** with two cycles of latency, and stream ***In1*** is halted. If on any given cycle, ***In0*** does not contain valid pixels, then pixels on input ***In1*** are passed to ***Out*** with one cycle of latency.

This filter is useful for combining two slow-running streams (for example, two TV inputs can be combined in this way if the clock rate is above 27MHz). Because input ***In1*** can be frequently halted, a FIFO is typically used on this input.

### Example

See the "Join" example.

### *7.8.4 PxsLineBuffer/PxsDualLineBuffer: line buffering*

```
#include "pxs.hch"
macro proc PxsLineBuffer (In, Out, Width);
macro proc PxsDualLineBuffer (In0, In1, Out0, Out1, Width);
```

#### Input Streams

***`In/In0/In1:`*** Must contain synchronous coordinates and progressive sync signals.

#### Output Streams

***`Out/Out0/Out1:`*** Must be compatible with corresponding input.

#### Parameters

***`Width:`*** Constant, equal to the horizontal width in pixels of the visible area.

#### Latency

1 line + 1 cycle

#### Haltable

No

#### Description

Delay a stream by a single line. Only the pixel and active data are buffered, the coordinates and sync signals are reconstructed, to minimize on-chip RAM usage.

`PxsDualLineBuffer()` allows two lines to be buffered at a time. This is typically used to construct 3x3 (or greater) convolutions. Packing two lines into a single RAM optimizes RAM usage.

### *7.8.5 PxsMux*: multiplexing streams*

```
#include "pxs.hch"
typedef enum
{
    PXS_NONBLOCK,
    PXS_BLOCK
}
PxsBlockingMode;
macro proc PxsMux2 (In0, In1, Out, Control, BlockingMode);
macro proc PxsMux3 (In0, In1, In2, Out, Control, BlockingMode);
macro proc PxsMux4 (In0, In1, In2, In3, Out, Control, BlockingMode);
```

#### Input Streams

***In0/In1/In2/In3:*** All stream types.

#### Output Streams

***Out:*** Must be compatible with all inputs.

#### Parameters

| | |
|---|---|
| ***Control*** | An unsigned int, of width 1 (PxsMux2) or 2 (PxsMux3 or PxsMux4). |
| ***BlockingMode*** | Constant or variable, of type PxsBlockingMode. |

#### Latency

1 cycle

#### Haltable

No

#### Description

Dynamically multiplex two, three or four input streams to a single output stream. The variable ***Control*** selects which input is active at any given time.

In blocking mode (***BlockingMode*** set to PXS_BLOCK), when any given input is selected, all other inputs are halted. If the input streams are the result of an upstream PxsSplit filter then the original source will be halted, which will in turn cause the active input to be halted.

In non-blocking mode (***BlockingMode*** set to PXS_NONBLOCK), inputs are only halted when the output is halted.

#### Example

See the "Blend" example.

### 7.8.6 PxsMux2Stream: multiplexing controlled by streams

```
#include "pxs.hch"
macro proc PxsMux2Stream (In0, In1, ControlStream, Out);
```

#### Input Streams

***In0/In1:*** All stream types.

***ControlStream:*** A stream of pixel type PXS_MONO_U1.

#### Output Streams

***Out:*** Must be compatible with ***In0*** and ***In1***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Dynamically multiple two input streams to a single output stream, based on the input pixel value of a third stream (***ControlStream***). This is useful for multiplexing on a pixel-by-pixel basis. When the pixel value of ***ControlStream*** is 0, data at ***In0*** are copied to ***Out***, otherwise data at ***In1*** are copied to ***Out***.

In blocking mode (***BlockingMode*** set to PXS_BLOCK), when any given input is selected, all other inputs are halted. If the input streams are the result of an upstream PxsSplit filter then the original source will be halted, which will in turn cause the active input to be halted.

In non-blocking mode (***BlockingMode*** set to PXS_NONBLOCK), inputs are only halted when the output is halted.

### *7.8.7 PxsNonReturnValve: inhibiting flow control*

```
#include "pxs.hch"
macro proc PxsNonReturnValve (In, Out);
```

#### Input Streams

***`In:`*** All stream types.

#### Output Streams

***`Out:`*** Must be compatible with ***`In`***.

#### Latency

1 cycle

#### Haltable

No

#### Description

The non-return valve prevents the upstream halt flag from being propagated from stream ***`Out`*** to stream ***`In`***. This can be used when splitting streams to prevent one branch of the split from blocking the source.

### *7.8.8 PxsRateLimiter: limit stream data rate*

```
#include "pxs.hch"
macro proc PxsRateLimiter (In, Out, Cycles);
```

#### Input Streams

***In:*** All stream types.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Cycles***: Constant, greater than one.

#### Latency

1 cycle

#### Haltable

No

#### Description

Implement rate limiting on the input stream by controlling the halt flag. The input stream is halted for all but one cycle in every ***Cycles*** cycles. Assuming the source filter can be halted, the stream will therefore contain a maximum ratio of 1 valid pixel every ***Cycles*** pixels.

#### Example

See the "Join" example.

## 7.8.9 PxsSend/PxsReceive: passing streams over channels

```
#include "pxs.hch"
macro proc PxsSend    (In,  ChanPtr);
macro proc PxsReceive (Out, ChanPtr);
```

### Input Streams

***In:*** All stream types.

### Output Streams

***Out:*** Must be identical to ***In***.

### Parameters

***ChanPtr*** Pointer to a channel of type "unsigned", sufficiently wide to hold all the components of the input stream.

### Latency

N/A

### Haltable

No

### Description

`PxsSend()` serialises the contents of a stream, and sends it down a channel. If the channel blocks, the input stream is halted. If the input stream is invalid, the channel is not sent to.

`PxsReceive()` receives from the channel and de-serializes it into the stream. If the output stream is halted, the channel will block. If the channel is not received from, the output stream is set as invalid.

These two macros should always be used as a pair. Since the only type checking available is the width of the channel, the user must ensure that streams ***In*** and ***Out*** are of identical (and not just compatible) types.

Typically these macros are used to pass streams across clock domains. In this situation, it is important to use the "`with { fifolength = ... }`" specifier on the channel declaration to achieve one pixel per clock cycle performance (in the slowest domain).

### Example

See the "MultiDomain" example.

### *7.8.10 PxsSplit*: splitting streams*

```
#include "pxs.hch"
macro proc PxsSplit2 (In, Out0, Out1);
macro proc PxsSplit3 (In, Out0, Out1, Out2);
macro proc PxsSplit4 (In, Out0, Out1, Out2, Out3);
```

#### Input Streams

***In:*** All stream types.

#### Output Streams

***Out0/Out1/Out2/Out3***: Must be compatible with ***In***.

#### Latency

1 cycle

#### Haltable

No

#### Description

Split an input stream two, three or four ways. The input stream is duplicated to all output streams. In any output stream is halted, the input stream is halted. Use a PxsNonReturnValve() to override this behaviour on any given output.

#### Example

See the "VideoGen" example.

## *7.8.11 PxsSynchronise: synchronize skewed streams*

```
#include "pxs.hch"
macro proc PxsSynchronise (EarlyIn, LateIn, EarlyOut, LateOut, MaxSkew);
```

### Input Streams

***EarlyIn:*** Must contain synchronous coordinates.

***LateIn:*** Must contain synchronous coordinates.

### Output Streams

***EarlyOut:*** Must be compatible with ***EarlyIn***.

***LateOut:*** Must be compatible with ***LateIn***.

### Parameters

***MaxSkew:*** Constant, greater than or equal to 1.

### Latency

1 cycle from ***LateIn*** to ***LateOut***, variable from ***EarlyIn*** to ***EarlyOut***

### Haltable

No

### Description

Synchronize two streams with differing latencies. This can be used when synchronizing a pair of split streams which have been processed through different chains of filters, typically as a prelude to re-combining them in some way. The stream ***LateIn*** is copied directly to ***LateOut.*** The stream ***EarlyIn*** is delayed by a sufficient number of cycles such that it is horizontally synchronized to ***LateOut***, and is output as ***EarlyOut***.

This filter obviates the need to calculate the exact latency of each side, only requiring the user to estimate the maximum skew between them (setting ***MaxSkew*** too high will at worst cause excess RAM usage). Because the coordinate stream is used to do the resynchronization, both input streams must have synchronous coordinates.

### Example

See the "Blend" example.

### *7.8.12 PxsValve: controlling streams*

```
#include "pxs.hch"
macro proc PxsValve (In, Out, Control);
```

#### Input Streams

***`In:`*** All stream types.

#### Output Streams

***`Out:`*** Must be compatible with ***`In`***.

#### Parameters

***`Control:`*** An unsigned int, of width 1.

#### Latency

1 cycle

#### Haltable

No

#### Description

Turn a stream on and off. When ***`Control`*** is set to 1, the stream flows normally. When ***`Control`*** is set to 0, the input stream is halted, and no valid pixels are passed to the output stream.

#### Example

See the "Reader" example.

## 7.9 Framebuffers

Framebuffer filters are those capable of storing and reading out entire images from RAM (normally off chip RAM).

- PxsPalPL1RAMReader: read a static image
- PxsPalPL1RAMFrameBufferDB: double-buffered framebuffer
- PxsPalPL1RAMFrameBuffer: single-buffered framebuffer

### *7.9.1 PxsPalPL1RAMReader: read a static image*

```
#include "pxs.hch"
macro proc PxsPalPL1RAMReader (In, Out, Width, PL1RAM, ClockRate);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

***PL1RAM:*** Handle to a PAL PL1RAM.

***ClockRate:*** Constant, equal to the local clock rate.

#### Latency

4 cycles

#### Haltable

No

#### Description

Read a static image from an external PAL PL1RAM. The output stream contains pixels looked up according to the coordinate component of the input stream. Width is the width of the image in pixels, as stored in RAM. One pixel is stored per location in RAM, packed as the concatenation of the pixel channels (for example, R @ G @ B). Any remaining data bits are ignored. Since the macro calls PalPL1RAMRun( ) on the ***PL1RAM*** argument, any other accesses to the RAM must run in parallel with this macro.

#### Example

See the "Reader" example.

### 7.9.2 PxsPalPL1RAMFrameBufferDB: double-buffered framebuffer

```
#include "pxs.hch"
typedef enum
{
    PXS_BOB,
    PXS_WEAVE
}
PxsDeInterlaceMode;
macro proc PxsPalPL1RAMFrameBufferDB (In, CoordIn, Out, Width, DeInterlaceMode,
PL1RAM0, PL1RAM1, ClockRate);
```

#### Input Streams

***In:*** Must contain coordinates.

***CoordIn:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***CoordIn***. Pixel type must be the same as ***In***.

#### Parameters

***DeInterlaceMode:*** Constant or variable, of type `PxsDeInterlaceMode`.

***Width:*** Constant, greater than or equal to 1.

***PL1RAM0/PL1RAM1:*** Handles to two independent PAL PL1RAMs.

***ClockRate:*** Constant, equal to the local clock rate.

#### Latency

4 cycles

#### Haltable

No

#### Description

Framebuffer an input stream into a pair of external PAL PL1RAMs. The output stream contains pixels looked up according to the coordinate component of the ***CoordIn*** stream.

***Width*** is the width of the image in pixels, as stored in RAM. One pixel is stored per location in RAM, packed as the concatenation of the pixel channels (for example, R @ G @ B). Any remaining data bits are ignored. Since the macro calls `PalPL1RAMRun()` on the ***PL1RAM*** arguments, any other accesses to these RAMs must run in parallel with this macro.

At present, only PXS_WEAVE mode is implemented by this framebuffer. The RAM banks are swapped at the beginning of the even field of a new input frame (asynchronous to the output). As reading and writing use independent banks of RAM, the maximum bandwidth of the both (in pixels per second) is equal to ***ClockRate.***

### Example

See the "VGAIn" example.

### *7.9.3 PxsPalPL1RAMFrameBuffer: single-buffered framebuffer*

```
#include "pxs.hch"
typedef enum
{
    PXS_BOB,
    PXS_WEAVE
}
PxsDeInterlaceMode;
macro proc PxsPalPL1RAMFrameBuffer (In, CoordIn, Out, Width, DeInterlaceMode,
PL1RAM, ClockRate);
```

#### Input Streams

***In:*** Must contain coordinates.

***CoordIn:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***CoordIn***. Pixel type must be the same as ***In***.

#### Parameters

***DeInterlaceMode:*** Constant or variable, of type PxsDeInterlaceMode.

***Width:*** Constant, greater than or equal to 1.

***PL1RAM:*** Handle to a PAL PL1RAM.

***ClockRate:*** Constant, equal to the local clock rate.

#### Latency

6 cycles

#### Haltable

No

#### Description

Framebuffer an input stream in an external PAL PL1RAM. The output stream contains pixels looked up according to the coordinate component of the ***CoordIn*** stream. When the ***CoordIn*** stream is either invalid or is outside the active region, pixels are taken from the ***In*** stream and stored in the framebuffer. At all other times, the ***In*** stream is halted (see note below regarding bandwidth).

***Width*** is the width of the image in pixels, as stored in RAM. One pixel is stored per location in RAM, packed as the concatenation of the pixel channels (for example, R @ G @ B). Any remaining data bits are ignored. Since the macro calls `PalPL1RAMRun()` on the ***PL1RAM*** argument, any other accesses to the RAM must run in parallel with this macro.

Interlaced video inputs are de-interlaced according to the value of ***DeInterlaceMode***. `PXS_BOB` mode performs simple line doubling, which results in minimal interlacing artifacts but reduced vertical resolution. This is preferred for fast moving images. `PXS_WEAVE` mode interleaves the lines of each field, which gives the best vertical resolution, but generates comb-like artifacts on moving images.

The maximum bandwidth of the framebuffer (in pixels per second) is equal to ***ClockRate***, as PAL PL1RAMs support exactly one read or write per cycle. As reading is given priority, the "In" stream should normally be fed from a FIFO of suitable depth. Users should ensure there is sufficient bandwidth available, as it is shared between reading and writing. For example, displaying 1024*768*60Hz requires 47.2 MPixels/s of bandwidth. Writing from a CCIR-601 PAL TV input requires 720*576*(50/2) = 10.4 MPixels/s, giving a total RAM bandwidth of 57.6 MPixel/s. So at 65MHz, there is sufficient bandwidth to read a full TV stream and display on an XGA monitor. On the other hand, displaying 640*480*60Hz requires 18.4 MPixel/s, giving a total of 28.8 MPixel/s. So at 25.175MHz, there is insufficient bandwidth read a full TV stream and display on a VGA monitor.

In situations where this framebuffer does not have sufficient bandwidth, you should use a `PxsPALPL1RAMFrameBufferDB()`, or alternatively implement a framebuffer that packs multiple pixels into single words.

### Example

See the "FrameBuffer" example.

## 7.10 Image analysis

Image analysis filters are those concerned with analysing the contents of streams on a per-frame basis.

- PxsAnalyse: analyse pixel values
- PxsLabelBlobs: connected component labelling

### 7.10.1 PxsAnalyse: analyse pixel values

```
typedef struct PxsAnalyseHandle;
#include "pxs.hch"
macro proc PxsAnalyse                      (In, AnalyseHandlePtr);
macro proc PxsAnalyseAwaitUpdate           (AnalyseHandlePtr);
macro expr PxsAnalyseGetNumActive          (AnalyseHandlePtr);
macro expr PxsAnalyseGetSumValues          (AnalyseHandlePtr);
macro expr PxsAnalyseGetMinValue           (AnalyseHandlePtr);
macro expr PxsAnalyseGetMaxValue           (AnalyseHandlePtr);
macro expr PxsAnalyseGetModeValue          (AnalyseHandlePtr);
macro expr PxsAnalyseGetMinX               (AnalyseHandlePtr);
macro expr PxsAnalyseGetMinY               (AnalyseHandlePtr);
macro expr PxsAnalyseGetMaxX               (AnalyseHandlePtr);
macro expr PxsAnalyseGetMaxY               (AnalyseHandlePtr);
macro expr PxsAnalyseGetFrequency          (AnalyseHandlePtr, Value);
macro expr PxsAnalyseGetCumulativeFrequency (AnalyseHandlePtr, Value);
```

#### Input Streams

***In:*** Must contain coordinates and sync pulses. Must have pixel type PXS_MONO_U8.

#### Parameters

***AnalyseHandlePtr:*** Pointer to a PxsAnalyseHandle structure

***Value:*** Constant or variable of type unsigned 8

#### Latency

N/A

#### Haltable

N/A

#### Description

Perform simple value analysis of a stream. For each frame, the PxsAnalyse() filter gathers information about pixels in the active region of the input stream, and stores them in a PxsAnalyseHandle structure declared by the user. At the end of a frame, these values are made available to the through the macros below. The user can pause processing until an update happens by calling PxsAnalyseAwaitUpdate().

PxsAnalyseGetNumActive() returns the number of active pixels in a frame.

PxsAnalyseGetSumValues() returns the accumulated total of pixel values in a frame. Dividing this number by PxsAnalyseGetNumActive() therefore yields a mean (average) value.

`PxsAnalyseGetMinValue()` returns the minimum pixel value in a frame. `PxsAnalyseGetMinX()` and `PxsAnalyseGetMinY()` return the X and Y coordinate of one such pixel (there may be many within a frame).

`PxsAnalyseGetMaxValue()` returns the maximum pixel value in a frame. `PxsAnalyseGetMaxX()` and `PxsAnalyseGetMaxY()` return the X and Y coordinate of one such pixel (there may be many within a frame).

`PxsAnalyseGetModeValue()` returns the most frequent (mode) value in a frame.

`PxsAnalyseGetFrequency()` returns the frequency of a given pixel value, that is to say, the number of pixels with that value.

`PxsAnalyseGetCumulativeFrequency()` returns the cumulative frequency of a given pixel value, that is to say, the number of pixels with that exact value or lower.

### Example

See the "Analysis" example.

### 7.10.2 PxsLabelBlobs: connected component labelling

```
typedef struct PxsBlobList;
macro proc PxsLabelBlobs            (In, Out, Width, BlobListPtr, PL1RAM,
ClockRate);
macro proc PxsBlobListLock          (BlobListPtr);
macro proc PxsBlobListUnlock        (BlobListPtr);
macro expr PxsBlobListNumBlobs      (BlobListPtr);
macro proc PxsBlobListGetArea       (BlobListPtr, Blob, AreaPtr);
macro proc PxsBlobListGetSumXY      (BlobListPtr, Blob, SumXPtr, SumYPtr);
macro proc PxsBlobListGetBoundingBox (BlobListPtr, Blob, X0Ptr, Y0Ptr, X1Ptr,
Y1Ptr);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be PXS_MONO_U1.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be PXS_MONO_S16.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

***BlobListPtr***: Pointer to a PxsBlobList structure.

***PL1RAM:*** Handle to a PAL PL1RAM.

***ClockRate:*** Constant, equal to the local clock rate.

***Blob***: Constant or variable index into blob list.

***AreaPtr***: Pointer to a variable of type unsigned 32.

***SumXPtr***/***SumYPtr***: Pointer to a variable of type signed 32.

***X0Ptr***/***X1Ptr***/***Y0Ptr***/***Y1Ptr***: Pointer to a variable of type signed 16.

#### Latency

1 frame + 3 cycles

#### Haltable

No

#### Description

Perform connected-component (blob) labelling and analysis. Connected components are clusters of white pixels that are connected together (connectivity is four way: up, down, left and right).

NOTE: `PxsLabelBlobs()` is currently in BETA, and is not recommended for production use. The API is also subject to change.

`PxsLabelBlobs()` labels the binary input stream and outputs a new stream one frame later in which each blob is given a unique pixel value, starting from 1. ***`Width`*** is the maximum width of the active image region. ***`PL1RAM`*** is a constant handle to the PalPL1RAM structure used as a buffer. ***`ClockRate`*** is the maximum clock rate.

`PxsBlobListLock()` and `PxsBlobListUnlock()` are used to lock and unlock the blob list structure respectively. The structure becomes available for locking shortly after a VSync transition, and must be unlocked before the beginning of the next frame.

`PxsBlobListNumBlobs()` gives the number of blobs labelled.

`PxsBlobListGetArea()` gives the area of a specific blob.

`PxsBlobListGetSumXY()` gives the sum of the X and Y coordinates of the blob. Dividing these numbers by the area yields the centroid of the blob.

`PxsBlobListGetBoundingBox()` gives the coordinates of the bounding box of a blob.

### Example

See the "LabelBlobs" example.

## 7.11 Look-Up-Tables (LUTs)

Look-up-tables transform pixel values according to arbitrary functions stored in small RAMs or ROMs.

- PxsDynamicLUT: dynamic value transforms
- PxsDynamicLUT3: dynamic 3-channel value transforms
- PxsHistogramEqualize: histogram equalization
- PxsSelectLUT: selectable value transforms
- PxsStaticLUT: static value transforms
- PxsStaticLUT3: static 3-channel value transforms
- PxsLUT8*: standard LUT initializers

### 7.11.1 PxsDynamicLUT: dynamic value transforms

```
#include "pxs.hch"

typedef mpram
{
    rom unsigned 8 W[256];
    rom unsigned 8 R[256];
}
PxsLUT8;

macro proc PxsDynamicLUT (In, Out, LUTPtr);
```

#### Input Streams

***In:*** Must contain pixels of type PXS_MONO_U8.

#### Output Streams

***Out:*** Must be compatible with both ***In***.

#### Parameters

***LUTPtr:*** Pointer to a PxsLUT8

#### Latency

1 cycle

#### Haltable

No

#### Description

Transform the pixel values of the input stream according to a dynamically variable look-up-table (LUT) declared by the user. The PxsLUT8 should be treated as a normal Handel-C on-chip RAM: it can be declared as either distributed or block RAM, and it may be optionally initialized. The contents of the LUT can be altered by writing to the "W" port of the RAM. The "R" port is reserved for the LUT filter itself.

#### Example

See the "DynamicLUT" example.

### *7.11.2 PxsDynamicLUT3: dynamic 3-channel value transforms*

```
#include "pxs.hch"

typedef mpram
{
    wom unsigned 8 W[256];
    rom unsigned 8 R[256];
}
PxsLUT8;

macro proc PxsDynamicLUT3 (In, Out, LUT0Ptr, LUT1Ptr, LUT2Ptr);
```

#### Input Streams

***In:*** Must contain pixels of type PXS_RGB_U8 or PXS_YCbCr_U8.

#### Output Streams

***Out:*** Must be compatible with both ***In***.

#### Parameters

***LUTPtr:*** Pointer to a PxsLUT8

#### Latency

1 cycle

#### Haltable

No

#### Description

Transform the pixel values of the input stream according to three dynamically variable look-up-tables (LUTs) declared by the user. The PxsLUT8s should be treated as a normal Handel-C on-chip RAMs: they can be declared as either distributed or block RAM, and can optionally be initialized. The contents of each LUT can be altered by writing to the "W" port of the RAM. The "R" port is reserved for the LUT filter itself.

#### Example

See the "DynamicLUT" example.

### *7.11.3 PxsHistogramEqualize: histogram equalization*

```
#include "pxs.hch"
```

```
macro proc PxsHistogramEqualize (In, Out, AnalysisPtr);
```

#### Input Streams

***In:*** Must contain pixels of type `PXS_MONO_U8`.

#### Output Streams

***Out:*** Must be compatible with both ***In***.

#### Parameters

***AnalysisPtr:*** Pointer to a `PxsAnalysis` structure

#### Latency

1 cycle

#### Haltable

No

#### Description

Perform histogram equalization of the input stream. This redistributes the spread of pixel values in order to give an approximately flat distribution (and therefore an even balance of shades). `AnalysisPtr` should point to a structure that is being updated by a corresponding `PxsAnalyse()` filter.

#### Example

See the "HistogramEq" example.

### 7.11.4 PxsSelectLUT: selectable value transforms

```
#include "pxs.hch"
macro proc PxsSelectLUT  (In, Out, LUTCount, LUTInit, LUTSelect);
```

#### Input Streams

***In:*** Must contain pixels of type `PXS_MONO_U8`, `PXS_RGB_U8` or `PXS_YCbCr_U8`.

#### Output Streams

***Out:*** Must be compatible with both ***In***.

#### Parameters

***LUTCount:*** Constant, greater than or equal to 1.

***LUTInit:*** List of (***LUTCount*** * 256) constants in the range 0..255 inclusive.

***LUTSelect:***Variable, of type unsigned (log2ceil (***LUTCount*** - 1))

#### Latency

1 cycle

#### Haltable

No

#### Description

Transform the pixel values of the input stream according to a number of fixed look-up-tables (LUTs). Each pixel channel value is looked up in a ROM initialized by the LUTInit argument. Several different tables can be concatenated together and dynamically chosen at runtime by varying the value of LUTSelect. On Xilinx Virtex-II and later devices, 8 single-channel LUTs can fit within one 18-KBit BlockRAM.

Some standard LUTs are provided (see the `PxsLUT8*` macros).

#### Example

See the "SelectLUT" example.

### 7.11.5 PxsStaticLUT: static value transforms

```
#include "pxs.hch"
macro proc PxsStaticLUT  (In, Out, LUTInit);
```

#### Input Streams

***In:*** Must contain pixels of type `PXS_MONO_U8`, `PXS_RGB_U8` or `PXS_YCbCr_U8`.

#### Output Streams

***Out:*** Must be compatible with both ***In***.

#### Parameters

***LUTInit:*** List of 256 constants in the range 0..255 inclusive.

#### Latency

1 cycle

#### Haltable

No

#### Description

Transform the pixel values of the input stream according to a fixed look-up-table (LUT). Each pixel channel value is looked up in a ROM initialized by the LUTInit argument.

Some standard LUTs are provided (see the `PxsLUT8*` macros).

#### Example

See the "LUT" example.

### 7.11.6 PxsStaticLUT3: static 3-channel value transforms

```
#include "pxs.hch"
macro proc PxsStaticLUT3 (In, Out, LUT0Init, LUT1Init, LUT2Init);
```

#### Input Streams

***In:*** Must contain pixels of type PXS_RGB_U8 or PXS_YCbCr_U8.

#### Output Streams

***Out:*** Must be compatible with both ***In***.

#### Parameters

***LUTInit:*** List of 256 constants in the range 0..255 inclusive.

#### Latency

1 cycle

#### Haltable

No

#### Description

Transform the pixel values of the input stream according to a fixed look-up-table (LUT). Each pixel channel value is looked up in one ROM per channel, initialized by the LUT*Init arguments.

Some standard LUTs are provided (see the PxsLUT8* macros).

#### Example

See the "LUT" example.

### *7.11.7 PxsLUT8*: standard LUT initializers*

```
macro expr PxsLUT8Linear;
macro expr PxsLUT8Inverse;
macro expr PxsLUT8Square;
macro expr PxsLUT8SquareRoot;
macro expr PxsLUT8SCurve;
macro expr PxsLUT8HalfSine;
macro expr PxsLUT8HalfCosine;
macro expr PxsLUT8Sine;
macro expr PxsLUT8Cosine;
macro expr PxsLUT8DoubleSine;
macro expr PxsLUT8DoubleCosine;
macro expr PxsLUT8InverseHalfSine;
macro expr PxsLUT8InverseHalfCosine;
macro expr PxsLUT8InverseSine;
macro expr PxsLUT8InverseCosine;
macro expr PxsLUT8InverseDoubleSine;
macro expr PxsLUT8InverseDoubleCosine;
macro expr PxsLUT8Logarithm;
```

#### Description

A range of standard LUT initializer expressions. If *i* is the input value, scaled to the range 0..1, *angle* is *(i * 2 * pi)*, and *j* is the output value, then the LUT transformations are as follows:

`PxsLUT8Linear` is the identity function, *j = i*

`PxsLUT8Inverse` inverts the pixel values, *j = 1 - i*

`PxsLUT8Square` computes the square of pixel values, *j = i*i*

`PxsLUT8SquareRoot` computes the square root values, *j = sqrt (i)*

`PxsLUT8SCurve` computes a contrast enhancing S-Curve, *j = f0*f0*f0*(f0 * (f0 * 6 - 15) + 10)*

`PxsLUT8HalfSine` computes a half sine-wave transform, *j = sin (angle/2)*

`PxsLUT8HalfCosine` computes a half cosine-wave transform, *j = cos (angle/2)*

`PxsLUT8Sine` computes a sine-wave transform, *j = sin (angle)*

`PxsLUT8Cosine` computes a cosine-wave transform, *j = cos (angle)*

`PxsLUT8DoubleSine` computes a double sine-wave transform, *j = sin (2 * angle)*

`PxsLUT8DoubleCosine` computes a double cosine-wave transform, *j = cos (2 * angle)*

`PxsLUT8InverseHalfSine`, `PxsLUT8InverseHalfCosine`, `PxsLUT8InverseSine`, `PxsLUT8InverseCosine`, `PxsLUT8InverseDoubleSine`, `PxsLUT8InverseDoubleCosine` are the inverse of the above functions.

`PxsLUT8Logarithm` computes a logarithm transform (with a baseline offset by 1/256).

### Example

See the "LUT" and "SelectLUT" examples.

## 7.12 Morphology

Morphological filters apply shape-based transforms on greyscale or binary images.

- PxsClose: morphological closing
- PxsDilate: morphological dilation
- PxsErode: morphological erosion
- PxsOpen: morphological opening

### 7.12.1 PxsClose: morphological closing

```
#include "pxs.hch"
macro proc PxsClose (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be PXS_MONO_U1, PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

2 lines and 10 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 3x3 closing operation on the input stream. ***Width*** is the maximum active width of the input stream in pixels. Closing is dilation followed by erosion, and tends to increase the connectivity of the image (whilst also retaining approximately the same shape and area).

A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

### *7.12.2 PxsDilate: morphological dilation*

```
#include "pxs.hch"
macro proc PxsDilate (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be PXS_MONO_U1, PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

1 line and 5 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 3x3 dilation filter on the input stream. ***Width*** is the maximum active width of the input stream in pixels. The output value is the maximum of the 9 input values in the 3x3 window around the central pixel. This filter is useful for growing bright regions (often to increase connectivity).

A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

#### Example

See the "Morphology" example.

### 7.12.3 PxsErode: morphological erosion

```
#include "pxs.hch"
macro proc PxsErode  (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be PXS_MONO_U1, PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

1 line and 5 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 3x3 erosion or dilation filter on the input stream. ***Width*** is the maximum active width of the input stream in pixels. The output value is the minimum of the 9 input values in the 3x3 window around the central pixel. This filter is useful for shrinking bright regions (typically to decrease connectivity).

A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

#### Example

See the "Morphology" example.

### 7.12.4 PxsOpen: morphological opening

```
#include "pxs.hch"
macro proc PxsOpen  (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be PXS_MONO_U1, PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

2 lines and 10 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 3x3 opening operation on the input stream. ***Width*** is the maximum active width of the input stream in pixels. Opening is erosion followed by dilation, and tends to reduce the connectivity of the image (whilst retaining approximately the same shape and area).

A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

### 7.12.5 *PxsNonMaximaSuppressLine: clip pixels that are not a 2D maxima*

```
#include "pxs.hch"
macro proc PxsNonMaximaSuppressLine (In, Out, Width);
```

#### Input Streams

***In:*** Must have synchronous coordinates. Must be progressively scanned. Pixel type must be either PXS_MONO_U8 or PXS_MONO_S16.

#### Output Streams

***Out:*** Must be compatible with ***In***.

#### Parameters

***Width:*** Constant, greater than or equal to 1.

#### Latency

1 line and 9/10 cycles (subject to change)

#### Haltable

No

#### Description

Perform a 3x3 Sobel edge detection on the input stream. ***Width*** is the maximum active width of the input stream in pixels. The stream is convolved with the coefficients:

| | | |
|---|---|---|
| -1 | -2 | -1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

to compute the vertical gradient, and coefficients

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

to compute the horizontal gradient. The result is the sum of the absolute value of these two gradients, and is proportional to the smoothed edge intensity at a given point.

A single pixel layer bordering the active region will contain invalid results (since the calculation will involve pixels outside the active region).

#### Example

See the "EdgeDetect" example.

## 7.13 Noise generators

Noise generators insert pseudo-random values into streams, typically for simulating sensor degradation.

- PxsGaussianNoise: Gaussian noise generator
- PxsPerlinNoise: Perlin noise generator
- PxsSaltAndPepper: impulse noise overlay
- PxsWhiteNoise: white noise generator

### *7.13.1 PxsGaussianNoise: Gaussian noise generator*

```
#include "pxs.hch"
macro proc PxsGaussianNoise (In, Out, Seed, FixedPattern);
```

#### Input Streams

***In:*** Must contain sync pulses.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***.

#### Parameters

***Seed:*** Constant of type unsigned 32

***FixedPattern:*** Constant or variable of type unsigned 1

#### Latency

1 cycle

#### Haltable

No

#### Description

Generate independent (approximate) Gaussian noise on all channels of the output. The pixel values are spread over the complete range allowed by each channel. Seed can be any number (0 is acceptable), and is typically used to create several independent channels of noise. The noise can be forced to be "fixed pattern" by setting the value of ***FixedPattern*** to 1. This produces noise which is identical on each frame (synchronized to sync pulses).

Note: whilst the noise has a distribution that is roughly Gaussian, it is only an approximation achieved by application of the central limit theorem. In addition, the caveat regarding randomness described in `PxsWhiteNoise()` also applies.

#### Example

See the "Noise" example.

## 7.13.2 PxsPerlinNoise: Perlin noise generator

```
#include "pxs.hch"
macro proc PxsPerlinNoise (In, Out, Bits);
```

### Input Streams

***In:*** Must contain coordinates, and have pixel type PXS_MONO_S16.

### Output Streams

***Out:*** Must be compatible with ***In***.

### Parameters

***Bits:*** Constant, from 0 to 9 inclusive.

### Latency

16 cycles

### Haltable

No

### Description

Generate Perlin noise. Perlin noise is an approximation to filtered Gaussian noise that supports random access without requiring pre-computation. This is often used to produce hypertextures. The noise generated by PxsPerlinNoise() is 3D, indexed by the (X, Y) coordinates of the input stream, and the value of the pixel input (Z). The parameter ***Bits*** sets the scale of the noise, with a value of 0 being the smallest scale up (and progressing in powers of two).

Note that the current implementation takes a significant amount of time and memory to compile.

### Example

See the "PerlinRipple" example.

### *7.13.3 PxsSaltAndPepper: impulse noise overlay*

```
#include "pxs.hch"
macro proc PxsSaltAndPepper (In, Out, Seed, FixedPattern, Frequency);
```

#### Input Streams

***In:*** Must contain sync pulses.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be the same as ***In***, unless ***In*** is PXS_EMPTY.

#### Parameters

***Seed:*** Constant of type unsigned 32

***FixedPattern:*** Constant or variable of type unsigned 1

***Frequency:*** Constant or variable of type unsigned 9

#### Latency

1 cycle

#### Haltable

No

#### Description

Generate independent impulse (salt-and-pepper) noise on all channels of the output. The input stream is copied to the output stream, except that the output color is sometimes forced to black or white (chosen at random) with a given frequency. Seed can be any number (0 is acceptable), and is typically used to create several independent channels of noise. The noise can be forced to be "fixed pattern" by setting the value of ***FixedPattern*** to 1. This produces noise which is identical on each frame (synchronized to sync pulses). Frequency sets the probability that a given pixel will be corrupted, and ranges from 0 (no corruption) to 256 (every pixel corrupted).

> Note: the caveat regarding randomness described in `PxsWhiteNoise()` also applies.

#### Example

See the "MedianFilter" example.

### 7.13.4 PxsWhiteNoise: white noise generator

```
#include "pxs.hch"
macro proc PxsWhiteNoise (In, Out, Seed, FixedPattern);
```

#### Input Streams

***In:*** Must contain sync pulses.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***.

#### Parameters

***Seed:*** Constant of type unsigned 32

***FixedPattern:*** Constant or variable of type unsigned 1

#### Latency

1 cycle

#### Haltable

No

#### Description

Generate independent white noise on all channels of the output. The pixel values are spread over the complete range allowed by each channel. Seed can be any number (0 is acceptable), and is typically used to create several independent channels of noise. The noise can be forced to be "fixed pattern" by setting the value of ***FixedPattern*** to 1. This produces noise which is identical on each frame (synchronized to sync pulses).

Note: whilst the noise is visually uncorrelated and has a flat distribution, it is produced by a relatively crude method which would not satisfy standard measures of randomness.

#### Example

See the "Noise" example.

## 7.14 Plotters

Plotters are filters that generate (Pixel value, Coordinate) pairs in arbitrary orders.

- PxsPlot: plotted graphics

### 7.14.1 PxsPlot: plotted graphics

```
#include "pxs.hch"
typedef struct PxsPlotHandle;
macro proc PxsPlot                (Out, PlotHandlePtr);
macro proc PxsPlotSetPen_MONO_U1  (PlotHandlePtr, Shade);
macro proc PxsPlotSetPen_MONO_U8  (PlotHandlePtr, Shade);
macro proc PxsPlotSetPen_MONO_S16 (PlotHandlePtr, Shade);
macro proc PxsPlotSetPen_RGB_U8   (PlotHandlePtr, R, G, B);
macro proc PxsPlotSetPen_YCbCr_U8 (PlotHandlePtr, Y, Cb, Cr);
macro proc PxsPlotPixel           (PlotHandlePtr, X, Y);
macro proc PxsPlotLine            (PlotHandlePtr, X0, Y0, X1, Y1);
macro proc PxsPlotRectangle       (PlotHandlePtr, X0, Y0, X1, Y1);
```

#### Output Streams

***Out:*** Must not expect sync pulses (declare with PXS_N_A).

#### Parameters

***PlotHandlePtr:*** Pointer to a PxsPlotHandle structure

***Shade***/***R***/***G***/***B***/***Y***/***Cb***/***Cr***: Value of the appropriate type for the pixel channel

***X***/***Y***/***X0***/***Y0***/***X1***/***Y1***: Value of type signed 16

#### Latency

N/A

#### Haltable

Yes

#### Description

A general purpose plotting engine capable of drawing pixels, lines and filled rectangles. The output of the engine is an intermittent stream of pixels and coordinates, but no sync pulses. As a result, this stream cannot be used directly to drive a display, and must be fed into a framebuffer to be visualized. Access to the plot engine is via a PxsPlotHandle structure which should be declared by the user.

PxsPlot() is the filter itself.

PxsPlotSetPen_*() sets the current pen color for each of the possible pixel types. Note that the correct macro must be used to match the output pixel type of stream "Out". No compile-time checking is performed.

PxsPlotPixel() plots a single pixel in the current pen color. If you have a substantial number of pixels to plot (such as rendering a bitmap), this method is not recommended. Instead, consider creating a custom filter.

`PxsPlotLine()` plots a line from (***X0***, ***Y0***) to (***X1***, ***Y1***) in the current pen color, using Bresenham's algorithm.

`PxsPlotRectangle()` plots a filled rectangle (***X0***, ***Y0***) to (***X1***, ***Y1***) inclusive, the current pen color.

The capabilities of this engine may be extended in future.

The plot access macros may be called from a different clock domain to the filter itself. This is convenient for multirate designs and those including microprocessor interfaces.

### Example

See the "Plot", "Join", and "LabelBlobs" examples.

## 7.15 Sync generators

Sync generators create streams of sync pulses and coordinates capable of driving output devices.

- PxsTVSyncGen: TV sync generator
- PxsVGASyncGen: VGA sync generator

### *7.15.1 PxsTVSyncGen: TV sync generator*

```
#include "pxs.hch"
macro proc PxsTVSyncGen (Out, Mode);
```

#### Output Streams

***Out:*** Sync type must be PXS_INTERLACED_TV (or a subset of). Pixel type must be PXS_EMPTY.

#### Parameters

***Mode:*** Constant, of type unsigned 1.

#### Latency

N/A

#### Haltable

Yes

#### Description

Generate TV compatible sync pulses with synchronous coordinates. The output of this source filter is typically passed through a series of video generating blocks (for example, a Console or FrameBuffer) and filters, before being passed to a PxsTVOut() sink block. Mode is either NTSC (0) or PAL (1).

#### Example

See the "TVOut" example.

### 7.15.2 PxsVGASyncGen: VGA sync generator

```
#include "pxs.hch"
macro proc PxsVGASyncGen (Out, Mode);
```

#### Output Streams

***Out:*** Sync type must be PXS_PROGRESSIVE_VGA (or a subset of). Pixel type must be PXS_EMPTY.

#### Parameters

***Mode:*** Constant, of type SyncGen2Mode.

#### Latency

N/A

#### Haltable

Yes

#### Description

Generate VGA compatible sync pulses with synchronous coordinates. The output of this source filter is typically passed through a series of video generating blocks (for example, a Console or FrameBuffer) and filters, before being passed to a PxsVGAOut() sink block. Mode must be one of:

| Mode | Horizontal Resolution, Pixels | Vertical Resolution, Lines | Refresh Rate, Hz | Clock rate, MHz |
|---|---|---|---|---|
| SYNCGEN_MODE_640_480_60HZ | 640 | 480 | 60 | 25.175 |
| SYNCGEN_MODE_640_480_72HZ | 640 | 480 | 72 | 31.500 |
| SYNCGEN_MODE_640_480_75HZ | 640 | 480 | 75 | 31.500 |
| SYNCGEN_MODE_640_480_85HZ | 640 | 600 | 85 | 36.000 |
| SYNCGEN_MODE_800_600_56HZ | 800 | 600 | 56 | 38.100 |
| SYNCGEN_MODE_800_600_60HZ | 800 | 600 | 60 | 40.000 |
| SYNCGEN_MODE_800_600_72HZ | 800 | 600 | 72 | 50.000 |
| SYNCGEN_MODE_800_600_75HZ | 800 | 600 | 75 | 49.500 |
| SYNCGEN_MODE_800_600_85HZ | 800 | 600 | 85 | 56.250 |
| SYNCGEN_MODE_1024_768_60HZ | 1024 | 768 | 60 | 65.000 |
| SYNCGEN_MODE_1024_768_70HZ | 1024 | 768 | 70 | 75.000 |
| SYNCGEN_MODE_1024_768_75HZ | 1024 | 768 | 75 | 78.750 |
| SYNCGEN_MODE_1024_768_85HZ | 1024 | 768 | 85 | 94.500 |
| SYNCGEN_MODE_1152_864_75HZ | 1152 | 864 | 75 | 108.000 |
| SYNCGEN_MODE_1152_864_85HZ | 1152 | 864 | 85 | 128.940 |
| SYNCGEN_MODE_1152_882_70HZ | 1152 | 882 | 70 | 94.500 |
| SYNCGEN_MODE_1152_882_85HZ | 1152 | 882 | 85 | 121.500 |
| SYNCGEN_MODE_1280_1024_60HZ | 1280 | 1024 | 60 | 108.000 |
| SYNCGEN_MODE_1280_1024_75HZ | 1280 | 1024 | 75 | 135.000 |
| SYNCGEN_MODE_1280_1024_85HZ | 1280 | 1024 | 85 | 157.500 |
| SYNCGEN_MODE_1600_1200_60HZ | 1600 | 1200 | 60 | 162.000 |
| SYNCGEN_MODE_1600_1200_75HZ | 1600 | 1200 | 75 | 202.500 |
| SYNCGEN_MODE_1600_1200_80HZ | 1600 | 1200 | 80 | 216.000 |
| SYNCGEN_MODE_1600_1200_85HZ | 1600 | 1200 | 85 | 229.500 |

### Example

See the "TestCard" example.

### 7.15.3 PxsVGASyncGenDynamic: adjustable VGA sync generator

```
#include "pxs.hch"
macro proc PxsVGASyncGenDynamic (Out, TimingPtr);
```

#### Output Streams

***`Out:`*** Sync type must be `PXS_PROGRESSIVE_VGA` (or a subset of). Pixel type must be `PXS_EMPTY`.

#### Parameters

***`TimingPtr:`*** Pointer to a SyncGen2Timing structure.

#### Latency

N/A

#### Haltable

Yes

#### Description

Generate VGA compatible sync pulses with synchronous coordinates. The output of this source filter is typically passed through a series of video generating blocks (for example, a Console or FrameBuffer) and filters, before being passed to a `PxsVGAOut()` sink block. The timing parameters of the mode are specified by the structure pointer to by ***`TimingPtr`***. These values can be statically initialised, and modified at run time.

The SyncGen2Timing structure contains the following members:

| | |
|---|---|
| `DotClock` | Dot clock, in Hz |
| `RefreshRate` | Refresh rate, in Hz |
| `HActivePixels` | X resolution (visible), in pixels |
| `HFrontPorchPixels` | Pixels before horizontal sync |
| `HSyncPixels` | Pixels of horizontal sync |
| `HBackPorchPixels` | Pixels after horizontal sync |
| `HTotalPixels` | Total horizontal pixels |
| `VActiveLines` | Y resolution (visible) |
| `VFrontPorchLines` | Lines before vertical sync |
| `VSyncLines` | Lines of vertical sync |
| `VBackPorchLines` | Lines after vertical sync |
| `VTotalLines` | Total vertical lines |
| `HSyncPolarity` | Polarity of HSync, 1 = Positive |
| `VSyncPolarity` | Polarity of VSync, 1 = Positive |

**Example**

See the "SyncGenDynamic" example.

## 7.16 Video I/O

Video input filters read video data from external devices such as cameras. Video output filters write streams to output devices such as monitors.

- PxsTVIn: TV input
- PxsTVOut: TV output
- PxsVGAIn: VGA input
- PxsVGAOut: VGA output

### 7.16.1 PxsTVIn: TV input

```
#include "pxs.hch"
macro proc PxsTVIn            (Out, InputIndex, Config, ClockRate);
macro expr PxsTVInInputCount  ();
macro expr PxsTVInConfigCount (InputIndex);
```

#### Output Streams

***Out:*** Must expect pixels of type PXS_YCbCr_U8. Must expect interlaced sync type.

#### Parameters

***InputIndex:*** Constant, greater than or equal to 0.

***Config:*** Constant or variable, greater than or equal to 0.

***ClockRate:*** Constant, equal to the local clock rate.

#### Latency

N/A

#### Haltable

No

#### Description

Read pixels from a TV input. ***InputIndex*** selects a particular physical input (each physical input may be used only once). ***Config*** selects a particular configuration of that input (for example, when multiplexing multiple inputs at the board level). On some platforms, ***Config*** may be variable at runtime. The total number of inputs can be queried by evaluating PxsTVInInputCount(). The number of configurations for a given input can be queried by evaluating PxsTVInConfigCount(). The meaning of the input number and configuration number are entirely platform dependent, however we recommend that (Input 0, Config 0) be the typical configuration.

In order to avoid dropping pixels, ClockRate should be greater than the dot clock of the input signal (typically 13.5MHz for SDTV).

#### Example

See the "FrameBuffer" example.

### *7.16.2 PxsTVOut: TV output*

```
#include "pxs.hch"
macro proc PxsTVOut            (In, OutputIndex, Config, ClockRate);
macro expr PxsTVOutOutputCount ();
macro expr PxsTVOutConfigCount (OutputIndex);
```

#### Input Streams

***In:*** Sync type must be interlaced TV. Pixel type must be PXS_RGB_U8.

#### Parameters

***OutputIndex:*** Constant, greater than or equal to 0.

***Config:*** Constant or variable, greater than or equal to 0.

***ClockRate:*** Constant, equal to the local clock rate.

#### Latency

N/A

#### Haltable

No

#### Description

Display a stream on a TV output. ***OutputIndex*** selects a particular physical output (each physical output may be used only once). ***Config*** selects a particular configuration of that output. On some platforms, ***Config*** may be variable at runtime. The total number of outputs can be queried by evaluating `PxsTVOutOutputCount()`. The number of configurations for a given output can be queried by evaluating `PxsTVOutConfigCount()`. The meaning of the output number and configuration number are entirely platform dependent, however we recommend that (Output 0, Config 0) be the typical configuration.

***ClockRate*** must exactly match the pixel rate of the stream to be displayed. If you need to display at a lower pixel rate than you are processing, you must use multiple clock domains (see the "VGAtoTV" example).

#### Example

See the "TVOut" example.

### 7.16.3 PxsVGAIn: VGA input

```
#include "pxs.hch"
macro proc PxsVGAIn            (Out, InputIndex, Config, ClockRate);
macro expr PxsVGAInInputCount  ();
macro expr PxsVGAInConfigCount (InputIndex);
```

#### Output Streams

***`Out:`*** Must expect pixels of type `MONO_RGB_U8`. Must expect progressive sync type.

#### Parameters

***`InputIndex:`*** Constant, greater than or equal to 0.

***`Config:`*** Constant or variable, greater than or equal to 0.

***`ClockRate:`*** Constant, equal to the local clock rate.

#### Latency

N/A

#### Haltable

No

#### Description

Read pixels from a VGA (or DVI) input. ***`InputIndex`*** selects a particular physical input (each physical input may be used only once). ***`Config`*** selects a particular configuration of that input (for example, when multiplexing multiple inputs at the board level). On some platforms, ***`Config`*** may be variable at runtime. The total number of inputs can be queried by evaluating `PxsVGAInInputCount()`. The number of configurations for a given input can be queried by evaluating `PxsVGAInConfigCount()`. The meaning of the input number and configuration number are entirely platform dependent, however we recommend that (Input 0, Config 0) be the typical configuration.

Note that VGA inputs typically produce much greater pixel rates than TV inputs. As current pixel types allow for at most one pixel per clock cycle, the pixel rate of the input must be less than ***`ClockRate`***, otherwise pixels may be dropped. For example, to capture 1024 x 768 @ 60Hz (XGA) at a dot-clock of 65MHz, it is recommended to run the design at a speed just slightly greater than this (for example, 66MHz) to allow for PLL tolerances.

**Example**

See the "VGAIn" example.

## *7.16.4 PxsVGAOut: VGA output*

```
#include "pxs.hch"
macro proc PxsVGAOut            (In, OutputIndex, Config, ClockRate);
macro expr PxsVGAOutOutputCount ();
macro expr PxsVGAOutConfigCount (OutputIndex);
```

### Input Streams

***In:*** Sync type must be progressive VGA. Pixel type must be PXS_RGB_U8.

### Parameters

***OutputIndex:*** Constant, greater than or equal to 0.

***Config:*** Constant or variable, greater than or equal to 0.

***ClockRate:*** Constant, equal to the local clock rate.

### Latency

N/A

### Haltable

No

### Description

Display a stream on a VGA (or DVI) output. ***OutputIndex*** selects a particular physical output (each physical output may be used only once). ***Config*** selects a particular configuration of that output. On some platforms, ***Config*** may be variable at runtime. The total number of outputs can be queried by evaluating `PxsVGAOutOutputCount()`. The number of configurations for a given output can be queried by evaluating `PxsVGAOutConfigCount()`. The meaning of the output number and configuration number are entirely platform dependent, however we recommend that (Output 0, Config 0) be the typical configuration.

***ClockRate*** must exactly match the pixel rate of the stream to be displayed. If you need to display at a lower pixel rate than you are processing, you must use multiple clock domains (see the "MultiDomain" example).

### Example

See the "TestCard" example.

## 7.17 Video generators

Video generators insert pixel values into streams based on sync and coordinate information.

- PxsCheckerboard: checkerboard pattern generator
- PxsConstant/PxsConstant3: constant color generator
- PxsConstant/PxsConstant3: constant color generator
- PxsTestCard: test card generator
- PxsXorPattern: XOR pattern generator

### *7.17.1 PxsCheckerboard: checkerboard pattern generator*

```
#include "pxs.hch"
macro proc PxsCheckerboard (In, Out, CheckSize);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***.

#### Parameters

***CheckSize:*** Constant, greater than or equal to 1.

#### Latency

1 cycle

#### Haltable

No

#### Description

Generate a black and white checker board pattern.

***CheckSize*** can be any constant, but it is significantly more efficient (in terms of area and delay) to choose a power of two.

#### Example

See the "VideoGen" example.

### *7.17.2 PxsConstant/PxsConstant3: constant color generator*

```
#include "pxs.hch"
macro proc PxsConstant  (In, Out, Value);
macro proc PxsConstant3 (In, Out, Value0, Value1, Value2);
```

#### Input Streams

***In:*** All stream types.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***.

#### Parameters

***Value/Value0/Value1/Value2:*** Constant or variable, of the same type as the pixel channels of the output stream

#### Latency

1 cycle

#### Haltable

No

#### Description

Generate a constant shade on the output. PxsConstant() generates the same value on all channels of a multi-channel stream (e.g. PXS_RGB_U8). PxsConstant3() can generate different values on each of the channels of a multi-channel stream.

#### Example

See the "VideoGen" example.

### 7.17.3 PxsTestCard: test card generator

```
#include "pxs.hch"
macro proc PxsTestCard (In, Out, Width, Height);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be either PXS_MONO_U1, PXS_MONO_U8, PXS_MONO_S16 or PXS_RGB_U8.

#### Parameters

***Width:*** Constant, width of visible area in pixels

***Height:*** Constant, height of visible area in pixels

#### Latency

3 cycles

#### Haltable

No

#### Description

Generate a test card stream. The image consists of a dark grey background, framed with a white line of single pixel thickness. Inside the grey area are animated vertical and horizontal color bars. If the area is large enough, a diagonal XOR pattern and a series of stripes/checkerboards are also generated. If the area is larger still, a central animated target pattern is also generated.

#### Example

See the "TestCard" example.

### *7.17.4 PxsXorPattern: XOR pattern generator*

```
#include "pxs.hch"
macro proc PxsXorPattern (In, Out);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with In. Pixel type must be either PXS_MONO_U1, PXS_MONO_U8, PXS_MONO_S16 or PXS_RGB_U8.

#### Latency

1 cycle

#### Haltable

No

#### Description

Generate a traditional XOR pattern by combining X and Y coordinates to create the color channels. This test pattern is particularly cheap to generate (in terms of logic elements).

#### Example

See the "VideoGen" example.

## 7.18 Video overlays

Video overlay filters create output streams in which some areas are "see through".

- PxsBouncingBall: bouncing ball overlay
- PxsConsole: text console overlay
- PxsCursor: pointer overlay
- PxsGrid: grid overlay
- PxsHistogramDisplay: generate a histogram overlay
- PxsOverlay: generic overlays
- PxsRectangle: rectangle overlay

### *7.18.1 PxsBouncingBall: bouncing ball overlay*

```
#include "pxs.hch"
macro proc PxsBouncingBall (In, Out, Width, Height, InitX, InitY, Size, Color1);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be the same as ***In***, unless ***In*** is PXS_EMPTY.

#### Parameters

***Width:*** Constant, width of visible area in pixels

***Height:*** Constant, height of visible area in pixels

***InitX:*** Constant, initial X position of the "ball"

***InitY:*** Constant, initial Y position of the "ball"

***Size:*** Constant, size in pixels of the "ball"

***Color1:*** Constant or variable of type unsigned 1

#### Latency

3 cycles

#### Haltable

No

#### Description

Overlay a simple "bouncing ball" onto the input stream. The initial position, size and color of the ball are all set at compile time. The ball moves one pixel per frame and always stays within the area (0, 0) - (***Width*** - 1, ***Height*** - 1).

#### Example

See the "VideoGen" example.

### 7.18.2 PxsConsole: text console overlay

```
#include "pxs.hch"
typedef struct PxsConsoleHandle;
macro proc PxsConsole           (In, Out, ConsolePtr, Width, Height);
macro proc PxsConsoleSetCursor  (ConsolePtr, CursorOn);
macro proc PxsConsoleMoveCursor (ConsolePtr, X, Y);
macro proc PxsConsolePutChar    (ConsolePtr, Char);
macro proc PxsConsolePutString  (ConsolePtr, String);
macro proc PxsConsoleClear      (ConsolePtr);
macro proc PxsConsolePutHex32   (ConsolePtr, Value);
macro proc PxsConsolePutUInt32  (ConsolePtr, Value);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be the same as ***In***, unless ***In*** is PXS_EMPTY.

#### Parameters

***ConsolePtr:*** Pointer to a PxsConsoleHandle structure

***Width:*** Constant, width of visible area in pixels

***Height:*** Constant, height of visible area in pixels

***CursorOn:*** Constant or variable, of type unsigned 1

***X/Y:*** Constant or variable, of type unsigned 8

***Char:*** Constant or variable, of type unsigned 8

***String:*** Array or RAM/ROM, of type unsigned 8[]

***Value:*** Constant or variable, of type unsigned 32

#### Latency

5 cycles

#### Haltable

No

## Description

Overlay a text console onto the input stream. The console is drawn with a standard 16x8 bitmap font. Access to the console is via a `PxsConsoleHandle` structure which should be declared by the user.

`PxsConsole()` is the filter itself.

`PxsConsoleSetCursor()` makes the current point visible (1) or invisible (0). By default, the cursor is visible.

`PxsConsoleMoveCursor()` moves the current point (the location where new text will be written).

`PxsConsolePutChar()` writes a single character at the current point. Any existing character at the current location is replaced. The insertion point wraps at the end of a line. Writing a character at the last location on the screen (bottom right) causes the screen to scroll by one line, inserting a blank line at the bottom of the screen. The first 128 characters as defined by ASCII are supported, characters 128-255 are drawn as identical to the first 128 but inverted. The control characters '\n' and '\r' are interpreted as carriage returns, and '\t' is interpreted as tab to next 8-character stop.

`PxsConsolePutString()` writes a string of characters from either a RAM, ROM or array. The string is terminated either by a NULL character (0), or by index overflow. The characters in the string are interpreted exactly as for `PxsConsolePutChar()`.

`PxsConsoleClear()` clears the screen and moves the cursor to position (0, 0).

`PxsConsolePutHex32()` writes a 32-bit unsigned integer as a hex value, prefixed by "0x".

`PxsConsolePutUInt32()` writes a 32-bit unsigned integer as decimal.

The console access macros may be called from a different clock domain to the filter itself. This is convenient for multirate designs and those including microprocessor interfaces.

## Example

See the "Console" example.

### 7.18.3 PxsCursor: pointer overlay

```
#include "pxs.hch"
typedef enum
{
    PXS_CURSOR_POINTER,
    PXS_CURSOR_CROSS,
    PXS_CURSOR_BUSY,
    PXS_CURSOR_MOVE,
    PXS_CURSOR_SIZE1,
    PXS_CURSOR_SIZE2,
    PXS_CURSOR_SIZE3,
    PXS_CURSOR_SIZE4
}
PxsCursorShape;
macro proc PxsCursor (In, Out, X, Y, CursorShape);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be the same as ***In***, unless ***In*** is PXS_EMPTY.

#### Parameters

***X:*** Variable of type signed 16, X position of cursor in pixels

***Y:*** Variable of type signed 16, Y position of cursor in pixels

***CursorShape:*** Constant or variable, of type PxsCursorShape

#### Latency

3 cycles

#### Haltable

No

#### Description

Overlay a mouse pointer onto the input stream. The cursor is positioned by changing the values of X and Y in parallel with the filter. The pointer has eight dynamically selectable shapes:

| | |
|---|---|
| `PXS_CURSOR_POINTER` | Classical mouse pointer |
| `PXS_CURSOR_CROSS` | Crosshairs (XOR'd with underlying image) |
| `PXS_CURSOR_BUSY` | Hourglass |
| `PXS_CURSOR_MOVE` | Four-headed arrow |
| `PXS_CURSOR_SIZE1` | Diagonal (top-right to bottom-left) two-headed arrow |
| `PXS_CURSOR_SIZE2` | Diagonal (top-left to bottom-right) two-headed arrow |
| `PXS_CURSOR_SIZE3` | Horizontal two-headed arrow |
| `PXS_CURSOR_SIZE4` | Vertical two-headed arrow |

Hot-spot adjustment is handled automatically (i.e. the tip of the pointer, or center of the cross hairs, will always be at the position given in X and Y, regardless of the pointer shape).

### Example

See the "GUI" example.

### *7.18.4 PxsGrid: grid overlay*

```
#include "pxs.hch"
macro proc PxsGrid (In, Out, GridSizeX, GridSizeY, OffsetX, OffsetY, Color1);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be the same as ***In***, unless In is PXS_EMPTY.

#### Parameters

***GridSizeX:*** Constant greater than one

***GridSizeY:*** Constant greater than one

***OffsetX:*** Constant or variable between 0 and ***GridSizeX*** - 1

***OffsetY:*** Constant or variable between 0 and ***GridSizeY*** - 1

***Color1:*** Constant or variable of type unsigned 1

#### Latency

1

#### Haltable

No

#### Description

Overlay a grid onto the input stream. The grid is spaced at intervals of ***GridSizeX*** and ***GridSizeY*** in the horizontal and vertical directions respectively. The grid is offset by (***OffsetX***, ***OffsetY***) pixels. The grid is either black (0) or white (1) depending on the value of ***Color1***.

The active area of the stream is extended to include the grid.

#### Example

See the "VideoGen" example.

### *7.18.5 PxsHistogramDisplay: generate a histogram overlay*

```
#include "pxs.hch"
macro proc PxsHistogramDisplay (In, Out, AnalyseHandlePtr, OriginX, OriginY,
Cumulative, ShiftFactor);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be the same as ***In***, unless ***In*** is PXS_EMPTY.

#### Parameters

***AnalyseHandlePtr:*** Pointer to a PxsAnalyseHandle structure

***OriginX/OriginY:*** Constant or variable of type signed 16

***Cumulative:*** Constant or variable of type unsigned 1

***ShiftFactor:*** Constant or variable of type unsigned 6

#### Latency

3 cycles

#### Haltable

No

#### Description

Overlay a histogram of pixel values onto a stream. ***AnaylseHandlePtr*** is a pointer to a AnalyseHandle structure declared by the user. This should be being updated by a PxsAnalyse() filter. (***OriginX***, ***OriginY***) form the coordinates of the origin of the histogram overlay. Cumulative selects between a normal histogram (0) and a cumulative histogram (1). ***ShiftFactor*** is the degree to which the frequency values are shifted in order to display them (i.e. a ***ShiftFactor*** of 5 will divide the frequencies by 2^5, resulting in frequency bars 1/32th of the length).

#### Example

See the "Analysis" example.

### *7.18.6 PxsOverlay: generic overlays*

```
#include "pxs.hch"
macro proc PxsOverlay (Upper, Lower, Out);
```

#### Input Streams

***`Upper:`*** All stream types.

***`Lower:`*** All stream types.

#### Output Streams

***`Out:`*** Must be compatible with both ***`Upper`*** and ***`Lower`***.

#### Latency

1 cycles

#### Haltable

No

#### Description

Overlay one stream onto another. On a given cycle, if stream ***`Upper`*** contains an active pixel this is copied to stream ***`Out`***. Otherwise, the pixel at stream ***`Lower`*** is copied to stream ***`Out`***. The active regions of streams can be controlled using the `PxsClip*()` filters.

The output stream contains exactly the same sync and coordinate information as Upper. Upper and Lower should be synchronized, i.e. they should originate from the same source and have the same latency. This can be achieved using a `PxsSynchronise()` filter. If the output stream is halted, both input streams are halted.

#### Example

See the "VideoGen" example.

### 7.18.7 PxsRectangle: rectangle overlay

```
#include "pxs.hch"
macro proc PxsRectangle (In, Out, X0, Y0, X1, Y1, Color1);
```

#### Input Streams

***In:*** Must contain coordinates.

#### Output Streams

***Out:*** Sync and coord types must be compatible with ***In***. Pixel type must be the same as ***In***, unless In is PXS_EMPTY.

#### Parameters

***X0/Y0/X1/Y1:*** Constant or variable of type signed 16

***Color1:*** Constant or variable of type unsigned 1

#### Latency

1 cycle

#### Haltable

No

#### Description

Overlay a rectangle onto the input stream. The rectangle stretches from (***X0***, ***Y0***) to (***X1***, ***Y1***) inclusive. The rectangle is either black (0) or white (1) depending on the value of ***Color1***.

#### Example

See the "Pong" example.

# 8 Index