MangoX

User Manual

Release 1.3

Contact Information

Mango DSP Ltd.

Postal:	Har Hotzvim Industrial Park	Tel:	+972-2-588 5000
	P.O. Box 45116	Fax:	+972-2-532 8705
	Jerusalem, 91450 Israel		

Detailed Revision History

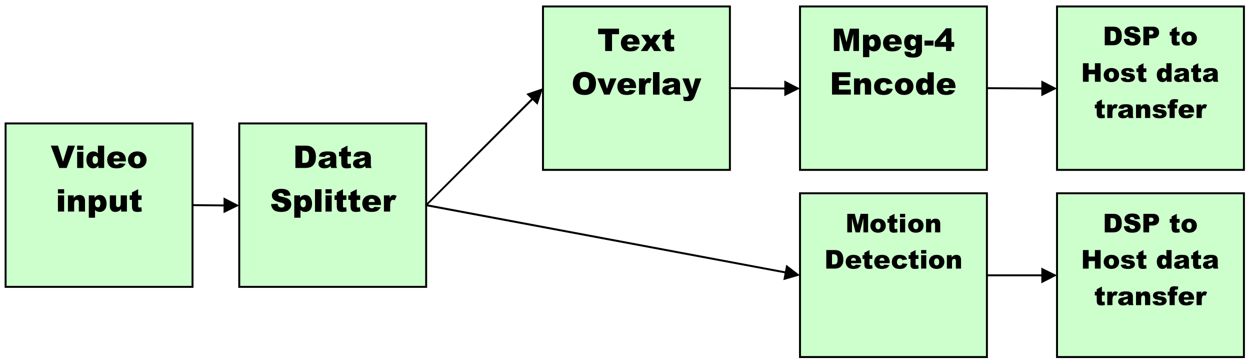
Rev.	Date	Author	Approved	Description
1	2.3.05	Itay Chamiel		Preliminary Release
2	19.7.05	Itay Chamiel		Updated to include filters
3	30.1.06	Itay Chamiel		Updates for v1.1
4	26.3.06	Itay Chamiel		Updates for v1.3
5	7.5.06	Itay Chamiel		New: Standalone Mode

Introduction

MangoX is a new multimedia framework for DSPs. It is intended to replace MangoAV and enables the user to perform any kind of video or audio processing with very little effort, as well as enabling the user to easily add their own functionality.

While MangoAV was based on Streams, MangoX is based on *Filters* that each perform a distinct function, and can be ordered in any desirable way to create a *Filter Graph*. This system is similar to Microsoft's DirectShow. Suppose, for example, a user would like to create a system that will receive video frames from a camera, perform some motion detection, overlay some text on the image, encode in Mpeg-4, add the motion detection data, and finally send all information to the host. Normally the user would have to write code that performs each of these tasks one after the other, taking care of issues such as multiple buffering and data integrity – the end result of which is long code that is hard to write, understand and maintain.

Using MangoX, on the other hand, the user would define a graph that looks something like this:



Each of these tasks will be performed by a self-contained module that only receives data, performs its function, and passes the output data on to the next filter in the chain.

MangoX is a C++ based architecture; each filter is defined in a class and an instantiation of each filter is an object. Although C++ is considered to be a more resource intensive language than C and therefore less suited for embedded applications, the instantiation of objects and the use of any polymorphism only occur at initialization time and not during the running of any algorithm. The algorithms themselves can and should be written in C and/or assembly. Therefore the actual performance overhead of the system is negligible and far outweighs the benefits of a clean, versatile framework.

MangoX has two very different modes of operation, each suited to different applications. One mode is named "PCI Mode". In this mode, it is a framework for creating PC-based applications that can run on most common operating systems, and employ the DSP (on a PCI board) as a slave for processor-intensive tasks such as video transcoding. In this mode, the DSP serves as a "black box" and is mostly untouched by the software developer, except perhaps for adding new filters for more functionality. The host PC is where the application code resides. The PC application defines the graphs used by the system, passes the information to the DSP via the PCI bus, and then receives and/or sends data to the DSP.

In the second mode, "Standalone", MangoX is a framework for developing multimedia DSP applications on stand-alone DSP systems such as the Mango Raven family of networked nodes. In this mode, the application is completely contained in the DSP. Graphs are defined, created and run within the DSP (an exception to this rule is the Raven-D board; See below for more details).

If you have received this package along with a Mango PCI board, read the sections titled "PCI Mode". If you have received it with a networked board, read the sections titled "Standalone Mode".

1. System Overview

PCI Mode

This mode is intended for PCI-based Mango boards, and therefore the delivered package consists of a host side and a DSP side. On the host, the user application must use the MangoX host library, which is responsible for technicalities such as loading and initializing the DSP and FPGA (if applicable), and passing information regarding creation and destruction of filter graphs. The application must also include the source files for the base classes as well as each filter that it intends to use.

The DSP side is comprised of a small main application that should remain mostly untouched by the user (unless filters are added or removed). In addition, it includes the source file for the base classes as well as each filter module that will be used by the system.

Most filter source files are shared between the host and DSP sides. Each filter has its own set of initialization parameters, and since the host creates and initializes the graph it must be fully aware of the class hierarchy and the parameters for the constructor of each class. Also, filters that transfer data between the host and DSP require functionality on the host side as well as the DSP side.

To separate DSP and host functionality a preprocessor directive "DSP" is defined on the DSP side. Most processing functions are defined within an #ifdef DSP block, so they are not compiled by the host compiler. Host functions are in the #else part.

Standalone Mode

This mode is intended for standalone, networked boards. In this mode a custom application is built for the DSP, with the capability to create processing graphs for tasks that are linear in nature. For example, the Mango AVS (Audio Video Server) application creates graphs for receiving and encoding incoming video, while the application handles RTSP, RTP streaming, and HTTP (Web) requests.

Application interfacing

Since data is transferred among filters, a problem can arise if you wish to transfer data between a graph and your application. This can either be data that is to be input to a graph, or the output data of a graph. There are two possible solutions to achieving this:

1. Set up your application in such a way that the graph is fully self-contained and performs all input and output necessary. For example, suppose your application needs to receive video, perform some processing, and send the resultant data to the network. This can be performed by a video input filter, a processing filter, and a third filter that sends data to the network.
2. Create custom *initiator* and/or *terminator* filters. These filters are a part of the graph (as an initial or final filter, respectively), but include a method to receive and/or send data to the application using whatever means you wish.

The AVS application uses the latter method. It includes custom Initiator and Terminator filters (which you may use for your own applications) that receive a pair of mailboxes as constructor parameters. When the application wishes to send or receive data from the graph, it simply accesses these mailboxes.

A Special Case: The Raven-D

On most standalone boards, all graphs are created and run on the DSP. An important exception to this rule is the Raven-D board. Although the board is a standalone unit, it contains two DSPs, only one of which is connected to the network. It is therefore designated the master, while the other DSP (the slave) can only communicate with the master (through an onboard hardware interface). In this case, the master and slave DSPs take on the role of the Host and DSP, respectively, as in PCI Mode. The master DSP runs the main application and some part of the data processing, while the slave DSP only creates and destroys graphs according to requests from the master.

For example, in the AVS application, each DSP can receive two video inputs. When a network request is received, the master DSP checks which camera number was requested. If it is 0 or 1, it creates the video encoding graph such that it runs on itself. If it is 2 or 3, it creates an encoding graph for the slave DSP to run, as well as a small graph, that runs on the master, which only receives data from the slave DSP and passes it to the application.

An interesting thing to note is that the code running on the master DSP that sends requests to the slave DSP looks virtually the same as code written for the Host PC that sends requests to the DSP.

MangoX Class Hierarchy

```
CMangoXPin
├ CMangoXOutPin
└ CMangoXInPin
```

`CMangoXFilter` (contains zero or more `CMangoXInPins` and zero or more `CMangoXOutPins`, but at least one pin in total.)
└ `CFilterFooBar` (user-defined filter classes go here.)

`CMangoXGraph` (contains two or more `CMangoXFilters`.)

Each single graph is represented by a `CMangoXGraph` object. This object contains all the filters, each of which is represented by a `CFilterFooBar` object (instead of FooBar, insert the actual functionality of the filter, such as H2D_PCI or Mpeg4VEnc). The filters are connected to each other through input and output pins; the output pin of one filter will be connected to the input pin of the next filter in the chain, and so on.

Each input pin is configured by the filter to only accept output pins of a certain type so, for example, an Mpeg-4 decoder won't accidentally receive data from an H.263 encoder. There are also pins that send arbitrary data (MT_ANY) and can connect to a pin of any type.

Since version 1.2 it is now possible for a pin to be **optional**. This means that the graph can function whether the pin is connected or not. For example, FilterVideoIn can output time tags for each frame separately from the video data itself, on a second output pin. This pin can remain unconnected if the application doesn't need timing information.

2. System Operation

PCI Mode

Example Project

The MangoX package includes an example project named UserApp, which demonstrates the operation of all of the following steps. Please refer to the UserApp.cpp source file while reading the following sections.

System Initialization

A Host MangoX application begins similarly to any MangoBIOS application – the process of opening device handles and card handles is not covered here. You may consult the MangoBIOS documentation for more information.

Once the card handle has been obtained, `MangoX_Open` is called to initialize the library. Finally, to initialize a card and its DSPs, call `MangoX_CardEnable`. This function loads all DSPs of a card and prepares them to receive graph information. Note that this function requires the path and filename of the .out (executable) file desired for each DSP (it is possible to load different .out files to different DSPs if you want different filters available on each DSP). Additionally, if your board has one or more FPGAs and requires them to be loaded for video functionality (for example, the Seagull PMC) you must specify the path and filename of the configuration data to load.

Graph initialization and operation

In the host application, the following operations are performed (the code is in UserApp.cpp, and the underlying code is in MangoX_Shared.cpp and in Host\MangoX.cpp):

1. Each of the filter objects are created and given initial parameters.
2. A new `CmangoXGraph` is created, with the number of filters as the constructor parameter.
3. The `CmangoXGraph` method `SetFilter` is called to bind each filter instance to this graph. You must bind exactly *n* filters to the graph and index them from 0 to *n*-1, where *n* is the number given in the previous step (number of filters in the graph).
4. The `CmangoXGraph` method `Attach` is called to connect the filters. It is called once for each connection. This function may return `false` if the output and input pins are not compatible. The function receives 4 parameters: SrcFiltIndex, DstFiltIndex, SrcPinIndex, DstPinIndex. The first and second values are the indices of the source and destination filters to connect. The third parameter is the pin number of the source pin, and the fourth parameter is the pin number of the destination pin. Pin numbers start from 0.
5. Finally, `MangoX_SubmitGraph` is called. This function serializes all information pertaining to the graph and passes it to the DSP. The function may return an error if the graph is not properly connected or the DSP encounters some error during graph construction, such as running out of memory.

At this point, the DSP performs the following actions (the code is in DSP\MangoX.cpp):

1. Receives all information pertaining to the new graph.
2. Creates a new graph and calls `SetConstructFunc` to give graph a translation function from class name to constructed filter instance. This is needed because the DSP receives the textual

name for each filter and wouldn't otherwise be able to construct objects of correct type. See function ConstructFilter in MangoX_Custom.cpp.

3. Calls the `Serialize` function to parse information received from the Host, create and configure all filters belonging to this graph.
4. Calls the `Create` function to connect and initialize all filters (this calls each filters' `Init` function).
5. Calls the `Start` function to start the graph's operation (this calls each filters' `Start` function).
6. Acknowledges to host that graph is ready.

From this point the graph is ready to use. Data is transferred to and from the DSP using member functions of the `CFilterH2D_PCI` and `CfilterD2H_PCI` objects. See the documentation for these filters for more information.

To stop a graph's operation and free all resources, perform the following in the Host application:

1. Call `MangoX_DeleteGraph` using the graph number received from `MangoX_SubmitGraph`. This will free all resources used on the DSP.
2. Use `delete` to free the host's graph object. This will automatically free all filter and pin objects, as well as free all resources such as PCI streams.

Error Handling

Occasionally the DSP will encounter an error condition that cannot easily be reported to the host, such as an encoder buffer overflow or a decoder error. Such error messages are sent to the host through a special error PCI stream that exists for each DSP. The host library automatically creates threads that poll on each of these streams and call a user supplied callback function if an error is reported. This function can alert the user by displaying the reported error to the standard output. Please see the API documentation for `MANGOX_ERR_CB` and `MangoX_CardEnable`, and see the example program for a usage example.

Standalone Mode

Example Project

The Mango AVS application serves as an example for employing MangoX in a DSP project.

Graph initialization and operation

These operations are performed when creating a new graph:

1. Each of the filter objects are created and given initial parameters.
2. A new `CmangoXGraph` is created, with the number of filters as the constructor parameter.
3. The `CmangoXGraph` method `SetFilter` is called to bind each filter instance to this graph. You must bind exactly *n* filters to the graph and index them from 0 to *n*-1, where *n* is the number given in the previous step (number of filters in the graph).
4. The `CmangoXGraph` method `Attach` is called to connect the filters. It is called once for each connection. This function may return `false` if the output and input pins are not compatible. The function receives 4 parameters: SrcFiltIndex, DstFiltIndex, SrcPinIndex, DstPinIndex. The first and second values are the indices of the source and destination filters to connect. The third

parameter is the pin number of the source pin, and the fourth parameter is the pin number of the destination pin. Pin numbers start from 0.

5. The `CmangoXGraph` method `Create` is called. This calls the `Init()` method of each filter, and this is where each filter should allocate its resources. This is the call that most commonly fails due to insufficient memory required by some filter; in this case it is safe to simply `delete` the graph object and issue some kind of error message. Check your error log (see Error Handling, below); it is likely that the offending filter has written the reason for the failure.
6. Finally, the `CmangoXGraph` method `Start` is called. This calls the `Start()` function of each filter, setting the graph in motion and beginning processing.

To stop a graph's operation and free all resources, simply `delete` the graph object. This will first call the `Stop()` function of each filter, and then call their destructors.

Error Handling

While error handling is mostly the application's responsibility, the filters will report most errors to an external `report_error` function. You may modify this function to create an error log as you wish.

Raven-D

Applications written for the Raven-D act differently when creating a graph that will run on the slave DSP. Under "PCI Mode" above, see the section "Graph initialization and operation". The Master DSP serves as the host, and the Slave DSP acts as the DSP.

An important addition for the Raven-D is the capability to transfer data between the DSPs. Two custom filters (which you should copy to your application) are added for this purpose, FilterD2DSender and FilterD2DReceiver.

See the AVS application for an example of using this capability. Pay special attention to all sections enclosed within the statement: `#if (BOARD_ID == RAVEN_D_BOARD_ID).`

3. Creating a custom filter

We have supplied a simple, empty filter that can be modified to suit your needs.

1. Under the MangoX/Filters directory, copy the FilterCustom directory and rename the copy to the name of your desired filter. It is recommended to maintain the directory structure and naming conventions.
2. Rename the header and cpp file to the name of your filter.
3. In both files, find and replace all instances of the text FilterCustom with the new name.
4. Add your new .cpp file to both host and DSP projects.
5. `#include` the new .h file in your Host application.
6. In the DSP file `MangoX_Custom.cpp`, `#include` your new .h file and add a relevant line in the `ConstructFilter` function.
7. Insert your functionality into the filter by modifying the source files. This process should be self-explanatory – see the comments.

Tips for creating filters

- Once your algorithm code is functional and debugged, optimize it as well as possible. Intrinsics, `restrict`, QDMA to access SDRAM, the -o3 compiler option, minimizing bank conflicts, hand coded pipelined assembly (for the brave) – use as many of these as possible to get as much speed as you can. There are whole books as well as courses given by TI on the subject.
- Follow the cache guidelines as described in the "Cache Coherency Conventions" section below.
- Buffers are passed between filters using mailboxes. Once your code has posted a buffer to a mailbox, it is considered as belonging to the next filter and must not be accessed anymore.
- Whenever possible, avoid copying data. Examples:
  - If your filter is only supposed to do something like overlay text on an image, it is probably unnecessary to copy the image to a new buffer before applying the overlay. The sample source file includes instructions on how to create a "pass-through" filter that does not create output buffers.
  - If your filter is a codec that requires the current and previous raw frames, it may sound intuitive to always copy the current frame to a static buffer so that it serves as a "previous" frame for the next iteration. However, this can be eliminated with some clever planning; see the Mpeg-4 decoder filter (in "fast mode") for an example.
- If any of your algorithms require space in ISRAM for data that is *not* guaranteed to remain between calls, you can use the scratch memory space (see "Memory Considerations" below) by using the `shared_isram` pointer, which you should define as `extern void*`. See the source of FilterVideoIn/Out for a method to ensure that this memory is allocated. You may modify `SHARED_MEM_SIZE` (decreasing is at your risk, of course). Access to this memory is protected by the semaphore `shared_scratch_SEM`, which you should pend on before running your algorithm and post when done.

Cache coherency conventions

This section is important only if you wish to create your own filters. We assume that you are familiar with the L2 cache and the commands defined in <csl_cache.h>. For more information, read the chapters relevant to L2 in SPRU610, and read the source code of the various included filters to see how the cache is manipulated.
In the flow of a graph on the DSP, data buffers are transferred among the filters. Since buffers are typically located in SDRAM, and some filters access data directly while others may use CPU (cache) accesses, there are potential issues with cache coherency. To prevent these issues and to avoid unnecessary cache cleaning, each filter must adhere to these simple guidelines:

1. Assume that any incoming buffer is fully coherent and not (fully or partially) in L2 cache.
2. Any buffer passed on to another filter must be fully cleaned out of the L2 cache.
3. All input and output buffers must be aligned on a cache line size boundary, and their size must be rounded up to a multiple of the cache line size. See the custom filter for an example.

Note that XDAIS compliant algorithms will generally access SDRAM buffers using DMA, and bypass the cache entirely (see TI document SPRU352, sec.6.14). Therefore, a filter that calls such an algorithm (such as Mpeg-4, H.263, etc) will implicitly comply with guideline 1. However, in order to comply with guideline 2 the filter must invalidate its output buffer from the cache, because it is possible that a filter further down the stream will use CPU access to read the buffer, potentially reading old data from the cache instead of new data from SDRAM. Another issue is that many encoders use a bitstream output routine that writes directly to memory, using the cache mechanism. Therefore, to keep with these conventions, it is recommended that after calling an encoder you writeback and invalidate the cache for the output buffer. After a decoder it is usually sufficient to invalidate, unless practice proves otherwise (distortion appears in the output data).

Memory Considerations

Please read this section only if you wish to add filters, or if you are using a DSP with 256KB memory (for example, the DM642 found on the Phoenix boards) and wish to tweak the memory configuration for highest performance of one specific codec.
The default memory configuration, in the DSP-side project, uses SDRAM for nearly all memory allocations including most of the program text, runtime stacks and DSP/BIOS objects. On-chip memory (known as ISRAM) is used for the following purposes:

- A reset vector at address 0 (required to allow the DSP to run from PCI).
- The L2 cache, set to 64KB for chips with 256KB memory, or 256KB for chips with 1MB.
- Any algorithm code or data sections which are required, according to the codec designer, to reside within ISRAM for performance reasons.
- A heap, from which codecs may allocate memory according to their specification. A large part of this heap is dedicated to algorithm *scratch memory*.

Scratch memory is memory that is non-exclusively allocated to an algorithm. This means that two algorithms use the same memory buffer for their temporary calculations. This is fine, as long as the two algorithms do not do so simultaneously; a protection semaphore exists to prevent this. The size of this scratch memory is normally defined as the total scratch memory required by an algorithm. Since there is more than one algorithm in the system, we look for the algorithm demanding the most scratch memory under its highest load, and use that value. This approach works well for DSPs with plenty of internal memory (1MB), but where memory is limited (256KB DSPs) it starts to become difficult to satisfy all algorithms in a single configuration. For example, the H.263 codec requires roughly 0x16000 bytes of scratch memory, but relatively little persistent memory. On the other hand, the MPEG-2 codec requires only about 0x7800 bytes of scratch but much larger persistent buffers. If the dedicated scratch memory is sized to fit all algorithms (that is, 0x16000 for the H.263) but MPEG-2 is used, there won't be enough memory left in the heap for MPEG-2's persistent buffers. If, in attempting to initialize an algorithm either the shared or persistent memory will run out, a diagnostic error will be sent to the Host. Use the information in this error message to adjust the size of the shared space accordingly. See file DSP\alg_malloc.cpp (look for a line starting with `#define SHARED_MEM_SIZE`) for the definition of the size of the scratch memory.

Another issue is the placement of code sections. Some codecs work best when some of its sections are placed (at link time) in internal memory. See file MangoX_link.cmd for a minimal configuration which satisfies the requirements of all included codecs. The problem is that this configuration cannot be used on 256KB DSPs, because there isn't enough on-chip memory.

The file MangoX_link_256KB.cmd is an example of a linker command file that gives lower preference to the H.263 and H.264 codecs in favor of MPEG-4. You may modify it to suit your preferences. For example, if you don't use MPEG-4, or you do not mind if it suffers a slight performance hit, you could send all its related sections into external memory (SDRAM), thus allowing you to link more sections relevant to the codec you do use into ISRAM. Note that you may even be able to increase the size of the ISRAM heap at this point – see the linker map file to see if there is spare memory to do so.

4. Filter Documentation

This section describes the filters created by Mango and supplied with MangoX.

General notes:

- Not all I/O filters are available for all hardware configurations; for example, the Seagull PCI has neither video nor audio capability.
- Memory requirements only include major buffers, not minor objects and data structures created by the filters.
- Memory requirements are only on DSP SDRAM unless noted otherwise.

1. I/O Filters

FilterH2D_PCI

Purpose: Streams data from the host to the DSP via PCI, using the PCI Stream library.
Input pins: none.
Output pins: 1 (MT_ANY).
Constructor arguments:

- int nNumBufs – highest number of buffers that can be used at once by the stream; defines buffering mechanism (1 for single-buffering, 2 for double-buffering, etc.).
- int nMaxXferSize – maximal size of buffer that will be transferred through this stream. The size of each transferred frame of data can range from 0 to this value, inclusive.
- int nTimeout – timeout when waiting for a buffer. Can be MANGOBIOS_FOREVER (to block until a buffer is received), MANGOBIOS_POLL (to return immediately) or any value, in milliseconds.
- h2d_pci_device_t * stPciDev – PCI Stream device for a particular DSP, given by MangoX_CardEnable. This handle must belong to the same DSP that will run this graph.

Host member functions:

- Connect – performs Host side initializations to this stream and verifies connection with DSP. In a working system this function will return immediately.
- GetEmptyBuffer – returns an empty buffer, of size nMaxXferSize, into which data for the DSP can be filled. Note that this is not a regular pointer; you should use "buf.local".
- SubmitBuffer – sends a buffer previously received using GetEmptyBuffer to the DSP.

Memory requirements: nMaxXferSize * nNumBufs (in DSP SDRAM and host shared memory).
Performance issues: Due to a limitation in the PCI architecture, if multiple DSPs attempt to simultaneously read from the Host, this will degrade overall throughput. In case of a multithreaded application it is recommended to place the transfer of data to the DSP in a critical section. If your application is single threaded, separate the transfers to each DSP so that they don't occur simultaneously. In both of these cases you will have to use single-buffer mode (nNumBufs=1), as only in this mode does the application have control over when exactly the data transfer takes place.
Notes:

- Host constructor will throw an exception if shared memory allocation fails. Destructor will throw an exception if there is any problem closing the stream or freeing shared memory.

FilterD2H_PCI

Purpose: Streams data from the DSP to the Host via PCI, using the PCI Stream library.
Input pins: 1 (MT_ANY).
Output pins: none.
Constructor arguments: identical to FilterH2DPCI.
Host member functions:

- Connect – see FilterH2DPCI.
- GetFullBuffer – waits for and receives data from the DSP. As with FilterH2DPCI, the received pointer is not a regular pointer but a struct of type PCI_STREAM_ptr_t; the "local" member of this struct will point to the data.
- FreeBuffer – returns the buffer to the PCI stream mechanism and indicate to DSP that it can be reused.

Memory requirements: nMaxXferSize * nNumBufs (in DSP SDRAM and host shared memory).
Notes:

- Host constructor will throw an exception if shared memory allocation fails. Destructor will throw an exception if there is any problem closing the stream or freeing shared memory.
- The performance issue applicable to FilterH2D_PCI does not apply to this filter.

FilterVideoIn

Purpose: Receives video from an external video source.
Input pins: none.
Output pins: 2 (MT_V_PLANAR_420, **optional** MT_TIME_TAG).
Constructor arguments:

- int nNumBufs – number of output buffers allocated by the filter.
- int nCamNum – camera number to read.
- enImageSize eImageSize – size of image to receive.
- enVideoStandard eVideoStandard – video standard (NTSC or PAL) of the analog video source.
- enVideoAnalogConnection eVideoConn – type of analog connection, composite or S-Video.
- int nCamBufs – number of input buffers for video input driver. Set between 2-10, recommended 3.

Memory requirements: Approx. 800KB * nCamBufs + nNumBufs * YUVsize (where YUVsize is width*height*3/2 of requested size).
Notes:

- Depending on your hardware configuration, the host may be required to call MangoX_ConfigVideo before creating the graph. Please see the example programs supplied with the library, or inquire with Mango customer support.
- Pin 1, if connected, will output a single buffer containing timing information along with each frame output from pin 0. The data is of type Time_tag_T. See the API for MangoAV_HW_get_img_from_cam for more information.

FilterVideoOut

Purpose: Outputs video to an analog video monitor.
Input pins: 1 (MT_V_PLANAR_420).
Output pins: none.
Constructor arguments:

- enImageSize eImageSize – size of image to expect to receive.
- enVideoStandard eVideoStandard – video standard (NTSC or PAL) to output.
- int nVideoBufs – number of buffers for video output driver. Set between 2-10, recommended 3.
- bool bDropFrames – If set to 0 (recommended), the maximal amount of frames that this filter will receive per second will be equal to the frame rate of the analog video output (25 for PAL, 29.97 for NTSC). If set to 1, frames received faster than this rate will be dropped.

Memory requirements: Approx. 800KB * nVideoBufs.
Notes:

- Depending on your hardware configuration, the host may be required to call MangoX_ConfigVideo before creating the graph. Please see the example programs supplied with the library, or inquire with Mango customer support.

FilterAudioIn

Purpose: Receives audio from an external audio source.
Input pins: none.
Output pins: 2 (MT_A_PCM, **optional** MT_TIME_TAG).
Constructor arguments:

- int nNumBufs – number of output buffers allocated by the filter.
- int nMicNum – audio input number to read.
- int nIsStereo – whether input is stereo.
- int nSampleRateDiv – sample rate divider; the value 96000 will be divided by this to obtain the desired sample rate. Recommended value is 2, for a rate of 48 kHz.

Notes:

- Frames created by this filter will be of 1152 samples each. A sample is a signed 16-bit (short) value, or two in case of a stereo input.
- Pin 1, if connected, will output a single buffer containing timing information along with each frame output from pin 0. The data is of type Time_tag_T. See the API for MangoAV_HW_get_audio_frame_from_mic for more information.

FilterAudioOut

Purpose: Sends audio to an analog audio output.
Input pins: 1 (MT_A_PCM).
Output pins: none.
Constructor arguments:

- int nSpkNum – audio output number to use.
- int nIsStereo – whether output is stereo.
- int nSampleRateDiv – sample rate divider; the value 96000 will be divided by this to obtain the desired sample rate. Recommended value is 2, for a rate of 48 kHz.

Notes:

- It is recommended to use the same format as that generated by FilterAudioIn.

2. Video Codecs

All video codecs convert a 4:2:0 Planar YCbCr image to a compressed bitstream, or vice versa.

FilterH263VEnc

Codec format: An H.263 implementation that satisfies the minimal requirement defined in the ITU-T H.263 specification. None of the annexes have been implemented.
Codec supplier: TI.
Input pins: 1 (MT_V_PLANAR_420).
Output pins: 1 (MT_V_H263).
Constructor arguments:

- stH263V_h2dparams& params – struct containing the following arguments:
    - int nNumBufs - number of output buffers allocated by the filter.
    - int nMaxOutputSize – size of output buffers allocated.
    - enVideoStandard eVideoStandard – video standard used; only PAL is supported.
    - enImageSize eImageSize – image size; only SQCIF, QCIF and CIF are supported.
    - int nBitrate – target bitrate (bits/second).
    - int nFramerate – target framerate, from 1 to 30 (frames/second).
    - int nNumGOPFrames – size of GOP, distance between consecutive I-Frames. 1 for I-Frames only.
    - int nQmax – maximum quantization value for encoding.
    - int nQmin – minimum quantization value for encoding.
    - int nQI – fixed quantization value for all I-frames.

Notes:

- If the output from this codec is to be decoded by a PC, the endianness of the generated bitstream must be reversed. Code for doing this is included (see comments in the filter code).

FilterH263VDec

Input pins: 1 (MT_V_H263).
Output pins: 1 (MT_V_PLANAR_420).
Constructor arguments:

- int nNumBufs - number of output buffers allocated by the filter.

Notes:

- Output buffers will be allocated when first encoded frame is received and size of output image is known.

FilterH264VEnc

Codec format: H.264 Baseline Video Encoder.
Codec supplier: W&W Communications, Inc.
Input pins: 2 (MT_V_PLANAR_420, **optional** MT_H264V_CMD).
Output pins: 1 (MT_V_H264).
Constructor arguments:

- stH264_params& params – struct containing the following arguments:
    - int nNumBufs - number of output buffers allocated by the filter.
    - int nMaxOutputSize – size of output buffers allocated.
    - enVideoStandard eVideoStandard – video standard used.
    - enImageSize eImageSize – image size.
    - int nBitrate – target bitrate (bits/second).
    - int nFramerate – target framerate (frames/second).

  - int nNumGOPFrames – size of GOP, distance between consecutive I-Frames. 1 for I-Frames only.
  - int nQp0 – Initial quantization factor for I-Frames (1-51, lower is higher quality).
  - int nQpN – Initial quantization factor for P-Frames (1-51, lower is higher quality).
  - int nCustomWidth - If image size is set to MX_CUSTOMSIZE, set custom width here. It accepts any multiple of 16 up to 720.
  - int nCustomHeight – If image size is set to MX_CUSTOMSIZE, set custom height here. It accepts any multiple of 16 up to 576.

Notes:

- If input pin 1 is connected, it can receive commands during stream runtime. The command shall be of type stH264VEncCmd, and can specify a reopening (with different parameters) or forcing of an I-Frame. See the filter header file for details.

FilterJPEGVEnc

Codec format: A JPEG implementation compliant with ITU-T Recommendation T.81.
Codec supplier: Mecoso Technology Inc.
Input pins: 2 (MT_V_PLANAR_420, **optional** MT_JPEG_CMD).
Output pins: 1 (MT_V_JPEG).
Constructor arguments:

- stJpegV_h2dparams& params – struct containing the following arguments:
  - int nNumBufs - number of output buffers allocated by the filter.
  - int nMaxOutputSize – size of output buffers allocated.
  - int width – width of input images. Recommended to use multiples of 8.
  - int height – height of input images.
  - int pitch – Normally set to 0 or the same as width. If encoding a small sub-image within a larger image, set this to the width of the large image.
  - int jfif – Set to 1 to generate JFIF information in the header, 0 otherwise.
  - int quality – Quality, from 1 (lowest) to 100 (highest).

Notes:

- If input pin 1 is connected, it can receive commands during stream runtime. The command shall be of type stJpegVEncCmd, and will force a reopening (with specified parameters) of the encoder. See the filter header file for details.

FilterJPEGVDec

Input pins: 1 (MT_V_JPEG).
Output pins: 1 (MT_V_PLANAR_420).
Constructor arguments:

- int nNumBufs - number of output buffers allocated by the filter.
- int pitch – Normally set to 0. If image to decode lies within a larger image, set to width of larger image.

Notes:

- Output buffers are allocated when JPEG headers are parsed and image size is known. Since each received image may be of a different size, output buffers may be reallocated if a currently decoded image is larger than a previous one.

FilterMpeg2VEnc

Codec format: ITU-T H.262, MPEG-2 encoder compliant to main profile main level (MP@ML) specification of International Standard Organisation ISO/IEC 13818-2:1995.
Codec supplier: TI.
Input pins: 1 (MT_V_PLANAR_420).
Output pins: 1 (MT_V_MPEG2).
Constructor arguments:

- stMp2V_h2dparams& params – struct containing the following arguments:
    - int nNumBufs - number of output buffers allocated by the filter.
    - int nMaxOutputSize - size of output buffers allocated.
    - enVideoStandard eVideoStandard – video standard, PAL or NTSC. Affects image size and frame rate.
    - enImageSize eImageSize – size of incoming video.
    - int nNumBFrames – number of B-frames, can be 0, 1 or 2. If set to 0 there will be no B-frames (IPPP…). If set to 1, there will be 1 B-frame between P-frames: IBPBP… If set to 2, the output will be IBBPBBP…
    - int nNumGOPFrames - Size of GOP. Recommended value is 15, but must be a whole multiple of (nNumBFrames+1).
    - int nConstantBitrate - target bitrate (bits/second).

Notes:

- When encoding with B-frames, the output from the second (and third, if nNumBFrames is 2) encoded frames will be 4-byte fillers.

FilterMpeg2VDec

Input pins: 1 (MT_V_MPEG2).
Output pins: 1 (MT_V_PLANAR_420).
Constructor arguments:

- int nNumBufs - number of output buffers allocated by the filter.
- enImageSize eMaxImageSize – maximal image size that this decoder should expect to decode. Needed for allocation of output image buffers.

Notes:

- Although the codec allocates its output buffer at init time, the filter does not. It only allocates its output buffers when the first encoded frame is received and the exact size of the output image is known.

FilterMpeg4VEnc

Codec format: MPEG-4 Simple Profile Visual Encoder.
Codec supplier: Prodys.
Input pins: 2 (MT_V_PLANAR_420, **optional** MT_MP4V_CMD).
Output pins: 1 (MT_V_MPEG4).
Constructor arguments:

- stMp4V_h2dparams& params – struct containing the following arguments:
    - int nNumBufs - number of output buffers allocated by the filter.
    - int nMaxOutputSize - size of output buffers allocated.

  - enVideoStandard eVideoStandard – video standard, PAL or NTSC. Affects image size and frame rate.
  - enImageSize eImageSize – size of incoming video.
  - nFramerateDivisor – value by which the standard frame rate (25 for PAL, 30 for NTSC) should be divided to obtain the requested frame rate. Use 1 for full frame rate. Values allowed for PAL are: 1, 2, 4, 5, 8, 10, 16, 20 and 25. For NTSC: 1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 16, 20, 24, 25 and 30.
  - nNumGOPFrames – size of GOP, distance between consecutive I-Frames. 1 for I-Frames only.
  - enRCMode eRC_mode – rate control mode. RC_CBR for constant bit rate, RC_CBR_NON_DROP for CBR with more aggressive bitrate control (does not request dropping frames), RC_VBR for variable bitrate, RC_VBR_NON_DROP for VBR in non-drop mode, or RC_CONSTANT_Q to disable rate control and encode at constant quality.
  - nConstantBitrate – (CBR only) bitrate.
  - nAvgBitrate – (VBR only) average bitrate.
  - nMaxBitrate – (VBR only) peak bitrate.
  - nQinitial – initial quantization value to use for first frame. In CONSTANT_Q mode this value will be used for all frames.
  - nQmax – (CBR only) maximum quantization value.
  - nQmin – (CBR, VBR) – minimum quantization value.
  - nCodecBufsize – (CBR only) buffer size in bits; see Prodys' documentation for details.
  - nCustomWidth – If image size is set to MX_CUSTOMSIZE, set custom width here. Values allowed are: 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 256, 272, 288, 320, 336, 352, 384, 400, 416, 432, 448, 480, 512, 528, 544, 560, 576, 624, 640, 672, 704, 720
  - nCustomHeight - If image size is set to MX_CUSTOMSIZE, set custom height here. Values allowed are: 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 256, 272, 288, 304, 320, 336, 352, 368, 384, 400, 416, 432, 448, 464, 480, 496, 512, 528, 544, 560, 576

Notes:

- In CBR and VBR mode, the codec may occasionally request that the filter drop a frame in order to maintain the requested bitrate. In this case the filter will output a buffer of size 0. If this is unacceptable, you can either use the NON_DROP modes, or define MP4ENC_IGNORE_FRAMESKIP in the filter header file, which will cause the filter to ignore these requests (at the expense of an output bitrate that may be too high).
- If input pin 1 is connected, it can receive commands during stream runtime. The command shall be of type stMpeg4VEncCmd, and can specify a reopening (with different parameters) or forcing of an I-Frame. See the filter header file for details.

FilterMpeg4VDec

Input pins: 1 (MT_V_MPEG4).
Output pins: 1 (MT_V_PLANAR_420).
Constructor arguments:

- int nNumBufs - number of output buffers allocated by the filter.

- bool bFastMode – fast mode. If false, codec's output and reference buffers will be allocated separately from the filter's output buffers. If true, the codec will use the filter's output buffers directly. This requires two conditions: (1) there are at least 2 output buffers; (2) these buffers are not modified by any filter down the stream, such as an OSD filter. Fast mode saves on memory and prevents a large data copy after each decoding, and is therefore highly recommended.

Notes:

- Output buffers will be allocated when first encoded frame is received and size of output image is known.

3. Audio Codecs

FilterMp3AEnc

Codec format: MPEG-1 or MPEG-2 Layer 3 Audio.
Codec supplier: Ittiam.
Input pins: 1 (MT_A_PCM).
Output pins: 1 (MT_A_MPEG12L3).
Constructor arguments:

- stMP3A_h2dparams& params – struct containing the following arguments:
  - int nNumBufs - number of output buffers allocated by the filter.
  - int nMaxOutputSize - size of output buffers allocated.
  - int nBitrate – bitrate per channel, in kbit (ex: 128, 256).
  - int nSampleFreq – sampling frequency (ex: 44100).
  - int nIsStereo – set to 1 for stereo, 0 for mono.

Notes:

- Each frame sent to this filter must contain AUDIO_SAMPLES_PER_FRAME (1152) samples of audio data in 16-bit signed format. In case of stereo input, the left and right channel data should be interleaved.
- There is a delay of two frames between the insertion of a frame and the output of the corresponding compressed data. Therefore, when you are done sending your data send two additional frames containing zeroes.

FilterMp3ADec

Input pins: 1 (MT_A_MPEG12L3).
Output pins: 1 (MT_A_PCM).
Constructor arguments:

- stMP3A_h2dparams& params – struct containing the following arguments:
  - int nNumBufs - number of output buffers allocated by the filter.
  - int nMaxOutputSize - size of output buffers allocated.

Notes:

- This filter can accept data from standard .mp3 files.
- The output is in 16-bit signed PCM format. Stereo data will be interleaved.
- The filter has an internal buffering mechanism that allows the user to send data packets of any size. If there is not enough input data to decode, the filter will emit a 0-byte buffer. In this case, continue sending data until there is enough (do not resend any data). On the other hand, if there

is more than one encoded frame in the buffer, they will all be decoded at once. Make sure your output buffer is at least 10 times larger than the largest buffer you intend to send to this filter, or it may overflow. If you run into any problems, such as hangups or buffer overflow errors, try reducing the size of the input buffer and increasing the size of the output buffer.

4. Processing Codecs

FilterOSD

Purpose: Overlays text and images over a video frame.
Input pins: 2 (MT_V_PLANAR_420, MT_OSD_CMD). The first pin is for input images; the second pin is for commands.
Output pins: 1 (MT_V_PLANAR_420) for the output image.
Constructor arguments:

- enVideoStandard eVideoStandard – video standard, PAL or NTSC.
- enImageSize eImageSize – size of incoming images.

Using the command stream:
The filter can display up to MAX_OSD_ENTRIES different overlays simultaneously on the image. The contents of each of these entries can be a text string or a bitmap. In the default state all entries are empty and therefore the filter does nothing to images that pass through it.
Sending commands to the stream is performed by filling a struct of type stOSDData with information pertaining to this command, and sending it, along with an optional text string or bitmap, as a single buffer to the command stream. The text or bitmap shall be appended immediately after the struct. The source of the data for the command pin can be the host (using a separate FilterH2D_PCI filter for this purpose) or a user-created filter in the DSP.
The contents of the stODSData struct are:

- enOSDCommand cmd – this is the command issed to the filter. Can be one of:
  - MX_OSD_ADD_ENTRY – Adds a new entry of the requested parameters to the overlay database. The entry must be empty.
  - MX_OSD_MODIFY_ENTRY – Modifies all modifiable parameters of this entry (see below). Data (text string or bitmap) is not changed, and should not be sent along with the struct in this case.
  - MX_OSD_MODIFY_DATA – Modifies the data (text or bitmap) of this entry. The rest of the struct (except entryNum) is ignored.
  - MX_OSD_DELETE_ENTRY – Deletes this entry and frees its resources. It can be reused by calling ADD_ENTRY again.
- int entryNum – the entry number to modify, must be lower than MAX_OSD_ENTRIES.
- enOSDType type – (ADD_ENTRY only) type of overlay, MX_OSD_TEXT or MX_OSD_BMP.
- enOSDTextSize textSize – (ADD_ENTRY only) size of text, MX_OSD_TEXT_SIZE_SMALL (16x16 pixel characters), MX_OSD_TEXT_SIZE_MEDIUM (32x32 pixels) or MX_OSD_TEXT_SIZE_LARGE (64x64 pixels).
- int bmp_width, bmp_height - (ADD_ENTRY only) for bitmap, its size.

The following parameters are modifiable with MODIFY_ENTRY.

- int isVisible – whether this entry will be displayed on the screen. Text can be flashed by toggling this value.

- int isTransparent – whether transparency is enabled. If true, all pixels with the value (255, 255, 255) shall be transparent. Text will be rendered with a black border around each character.
- int isYOnly – if true, only the Y (brightness) of the image will be affected, with the original colors showing through the overlay.
- int fore_Y, fore_Cb, fore_Cr – (for text only) color of the text.
- int back_Y, back_Cb, back_Cr – (for text only) background color. Set to 255, 255, 255 when enabling transparency.
- int x, y – position of upper left corner of overlay. Both values must be even.

Notes:

- When sending a text sting, it must be terminated with a null character. Set the size of the buffer being sent to: sizeof(stOSDData) + strlen(text_string) + 1. This will ensure that the null character is sent. Do not send a longer buffer than necessary.
- Bitmaps must be sent in 8-bit YCbCr, 4:2:0 format, with the Y, Cb and Cr buffers separate. The size of the buffer must be exactly sizeof(stOSDData) + width*height*3/2 or else the DSP will fail and return an error.
- Since text rendering is an expensive process, it is only done as a result of a command, with the result stored in a bitmap buffer. This buffer is then overlaid on each incoming image.
- This filter modifies the data in the incoming buffer rather than copying it to an output buffer; care should be used when using in conjunction with filters whose output must not be modified, such as FilterMpeg4VDec in Fast mode.