X4EASY SDK Documentation

Release 1.0

X4EASY SDK

This X4EASY SDK has been created to allow developers to author their own applications to control the X4EASY compatible devices. The interface to the X4EASY SDK is simple to use; there are no time-critical calls which need to be made, nor are there calls which depend on other calls already having been performed. The documentation contained in this document should guide you towards a fully functional X4EASY application within a short space of time.

Overview

The SDK structure is quite simple; a C style library file (X4EASY.LIB) and the corresponding C Header file (X4EASY.H) is distributed in this package. Calls are made into this LIB file (using the function prototypes in the header file, and the function documentation found in this document). In its simplest form, the application needs only to use the X4Load(), X4Free() functions, then call the enumeration functions (perhaps also calling the X4EasyDeviceGetSerialNumber/ X4EasyDeviceGetFriendlyName functions to fetch user-friendly "names" for the device(s) connected to the PC, then proceed to interrogate for in-effect parameters in the device and subsequently call functions to alter the configuration state of the open device.

Having found the number of devices (and optionally their names), a single device can be opened using X4EasyDeviceOpen using the index number of the desired device. This will return a handle to the device which should be used in all subsequent calls to API functions. When the device has finished being manipulated, X4EasyDeviceClose should be called to finish with the device and close the handle. Unclosed devices will leak handles if the application closes without calling this function.

The SDK functions are named such that there are 'Device' class functions (X4EasyDeviceOpen and X4EasyDeviceClose being two examples already introduced of these), 'Input' class functions which affect the operation and report status of the current output parameters, 'Output' class functions which affect the operation and report status of the output parameters. There are also a few handy helper functions which remove complexity from the application and place it in the X4EASY SDK so that all applications can benefit from bug-fixes along the way.

Functions are provided to manipulate the input EDID, the cropping parameters for each channel, and the output video mode for each output.

Important note

The parameters of the device are stored in Flash memory. It is therefore imperative that regular calls to X4EasyDeviceUpdateFlash are not made – calls to this function should be made sparingly, only after the user wants settings in the box finalised. In the X4Control application, these save-to-flash operations happen after each dialog box is dismissed with the "Ok" button. Regular, automatic, calling of this function will cause the flash memory to wear out – as with all writes to any type of flash memory.

X4EASY SDK Function list

X4EasyLoad

```
X4ERROR X4API X4EasyLoad(PHX4DLL pHX4DLL);
```

This function loads the X4EASY API and provides a handle for subsequent calls into the SDK. Call X4EasyFree when the SDK is finished with.

X4EasyFree

```
X4ERROR X4API X4EasyFree(HX4DLL hX4DLL);
```

This function unloads the X4EASY API, it should be the last call made to the SDK before the application terminates; no further X4EASY SDK calls can be made once this has been called, they will return X4ERROR_UNSUPPORTED.

X4EasyDeviceGetCount

```
X4ERROR X4API X4EasyDeviceGetCount(HX4DLL hX4DLL, unsigned long * pCount);
```

Call this function with the DLL handle to retrieve the number of attached devices on the system. The result is passed back to the caller through *pCount; this parameter may not be NULL.

X4EasyDeviceGetSerialNumber

```
X4ERROR X4API X4EasyDeviceGetSerialNumberA(HX4DLL hX4DLL,
                                           unsigned long index,
                                           char *pSerialNumber);
X4ERROR X4API X4EasyDeviceGetSerialNumberW(HX4DLL hX4DLL,
                                           unsigned long index,
                                           wchar_t *pSerialNumber);
```

There are two forms of this function on platforms which differentiate Unicode and non-unicode character types. On Windows, if _UNICODE or UNICODE is defined by the pre-processor, then X4EasyDeviceGetSerialNumber will be of the latter type, taking a wchar_t * type as the final parameter. On Windows, without these pre-processor defines, or on systems which support UTF-8 character encoding, the function will be of the former type, taking a char * type as the final parameter. On successful completion of the function, the buffer pointed to by *pSerialNumber will be filled with the textual representation of the serial number of the device, using the indicated character encoding. This is a text representation as the serial number may start with a 'Z' character (indicating that the device is in Firmware Update mode) and in any case has { and } characters surrounding the serial number. This is by design. Note that a device whose serial number begins with 'Z' is in bootloader mode, and will not communicate with the functions contained within this SDK.

The X4API.H header file will define a pre-processor macro to enable the source code to call X4EasyDeviceGetSerialNumber without the A or W suffix which redirects the compiler to call the correct text-representation of the function dependant on the state of the UNICODE or _UNICODE pre-processor defines.

X4EasyDeviceGetFriendlyName

```
X4ERROR X4API X4EasyDeviceGetFriendlyNameA(HX4DEVICE hDevice, char *pName);
X4ERROR X4API X4EasyDeviceGetFriendlyNameW(HX4DEVICE hDevice, wchar_t *pName);
```

This function allows the retrieval of the device's friendly name, which is set by the user. It is envisaged that these friendly names will give a terse description of the function/use of the device. For instance "Test R&D" or "Stage left wall".

There are two versions of this function, one wide character and one non-wide character. As with all character type dependant functions, a pre-processor macro defines the function call as X4EasyDeviceGetFriendlyName and points it at the correct form of the function's instantiation as indicated by the state of UNICODE and _UNICODE pre-processor defines. Systems which have no concept of wide characters (i.e. those platforms which use UTF-8) should leave UNICODE and _UNICODE undefined, and consequently will use the char * (-A) form of the function.

Caveat: Note that no translation is performed between character encodings; if an device is programmed using a wide-character system (i.e. Windows) and read back on a UTF-8 based system, the characters will appear corrupt on the UTF-8 system. It is assumed that a device will be connected to one type of system.

X4EasyDeviceSetFriendlyName

```
X4ERROR X4API X4EasyDeviceSetFriendlyNameA(HX4DEVICE hDevice, char *pName);
X4ERROR X4API X4EasyDeviceSetFriendlyNameW(HX4DEVICE hDevice, wchar_t *pName);
```

This function allows the setting of the device's friendly name, as outlined in X4EasyDeviceGetFriendlyName. The string handling aspects of this function are identical to those of the X4EasyDeviceGetFriendlyName, including the caveat above regarding character encoding between systems.

X4EasyDeviceGetCapabilities

```
X4ERROR X4API X4EasyDeviceGetCapabilities(HX4DEVICE hDevice, DWORD * pCapabilities);
```

This function retrieves a bitmask of capabilities that the device has. A certain amount of capabilities are implied. Currently the only capability defined is X4_CAP_NEWSCALINGALGORITHM which is abstracted away inside the SDK; no special coding is required to handle this. As more capabilities are added over time, this bitfield's definitions will grow and be described in this document.

X4EasyDeviceOpen

```
X4ERROR X4API X4EasyDeviceOpen(HX4DLL hX4DLL, unsigned long device, PHX4DEVICE phDevice);
```

This function opens a device connected to the system. "device" contains the zero-based index of the device which is to be opened. *phDevice on successful return will be populated with a handle to communicate with the device in question.

X4EasyDeviceClose

```
X4ERROR X4API X4EasyDeviceClose(HX4DEVICE hDevice);
```

Closes the device handle contained in hDevice. Once this call has been made, no further calls into the SDK using this handle are possible.

X4EasyDeviceGetSystemInfo

```
typedef struct _X4SYSTEMINFO
```

```
{
   unsigned long  Size;
   unsigned char  HardwareVersion;
   unsigned char  FPGAVersion;
   unsigned short FirmwareVersion;
}  X4SYSTEMINFO, *PX4SYSTEMINFO;

X4ERROR X4API X4EasyDeviceGetSystemInfo(HX4DEVICE hDevice, PX4SYSTEMINFO pSystemInfo);
```

For the given device handle in hDevice, get the information about the device. pSystemInfo points to a structure to hold the information to be retrieved from the device. Size should be filled in to indicate the size of the structure which is being passed into the function. On successful exit, the Size member will indicate how much of the structure's data is valid. The HardwareVersion field will contain the version number of the device's motherboard. FPGAVersion will contain the version of the flash file loaded into the device and FirmwareVersion will contain the version number of the firmware which is currently running on the device. These are all for information only, there are no SDK calls which can influence the state of these numbers.

X4EasyDeviceGetSystemHealth

```
typedef struct _X4SYSTEMHEALTH
{
   unsigned long  Size;
   unsigned short Temperature;
   unsigned short Volts50;
   unsigned short Volts33;
   unsigned short Volts25;
   unsigned short Volts18;
   unsigned short Volts12;
} X4SYSTEMHEALTH, *PX4SYSTEMHEALTH;

X4ERROR X4API X4EasyDeviceGetSystemHealth(HX4DEVICE hDevice, PX4SYSTEMHEALTH pSystemHealth);
```

For the given device handle, hDevice, fetch the state of the health of the system. pSystemHealth points to a a structure which holds the information returned. Prior to the call, Size should be filled in with the size of the structure. On successful exit, the Size member will indicate how much of the structure's data is valid.

Temperature will contain the temperature in degrees C as measured inside the device's case multiplied by 1000, to give a fractional reading.

The other fields all contain the voltages measured on the device's motherboard multiplied by 1000. The pair of digits after Volts indicates the ideal value for this parameter, some deviation is expected. This information is not for diagnostic purposes; the values contained herein are not to be used as indicators of faults.

X4EasyDeviceUpdateFlash

```
X4ERROR X4API X4EasyDeviceUpdateFlash(HX4DEVICE hDevice);
```

This function commits the currently in-effect parameters to flash memory. This function must be used sparingly; as with all flash memory technology, there are only a limited number of writes able to be tolerated by a flash memory cell, and so the flash memory has a limited number of writes lifetime. As a consequence, saving – for example – every 2 seconds would wear out the flash memory in a matter of a couple of weeks if it was left running.

X4EasyDeviceGetGenlockStatus

```
typedef enum _X4GENLOCK
```

```
{
   X4GENLOCK_NONE = 0,
   X4GENLOCK_MONITOR_MASTER,
   X4GENLOCK_INPUT_MASTER,
   X4GENLOCK_UNKNOWN
}  X4GENLOCK, *PX4GENLOCK;
X4ERROR X4API X4EasyDeviceGetGenlockStatus(HX4DEVICE hDevice, PX4GENLOCK pGenlock);
```

This function retrieves the status of the framelock feature; the function is misnamed for historical reasons. The enumeration values mean, respectively:

X4GENLOCK_NONE: No framelocking is in effect, the outputs are all operating asynchronously from each other, and likewise no framelocking is happening with the input signal.

X4GENLOCK_MONITOR_MASTER: All four outputs have been framelocked; they are generating the VSync signals at the same time as each other. However, input framelocking is not in force.

X4GENLOCK_INPUT_MASTER: All four outputs have been framelocked; they are generating the VSync signal at the same time as each other. Additionally, the output screen VSync signals are generated at the same time that the input signal has the VSync asserted.

X4GENLOCK_UNKNOWN: This is an invalid value, and will always be the last value in the enumeration. It will never be returned from the function.

X4EasyDeviceSetSmoothness

```
typedef enum _X4SMOOTHNESS {
   X4SMOOTHNESS_LEVEL0 = 0,
   X4SMOOTHNESS_LEVEL1,
   X4SMOOTHNESS_LEVEL2,
   X4SMOOTHNESS_LEVEL3,
   X4SMOOTHNESS_LEVEL4,
} X4SMOOTHNESS, *PX4SMOOTHNESS;
X4ERROR X4API X4EasyDeviceSetSmoothness(HX4DEVICE hDevice, X4SMOOTHNESS smoothness);
```

This function sets the smoothness of the scaler. Only two levels are currently recognised, X4SMOOTHNESS_LEVEL0 and any other value. X4SMOOTHNESS_LEVEL0 will set up the scaler to optimise the scaler output for text. Any other value will optimise for video. In practice these two values are for minimum smoothness and maximum smoothness respectively. There are several levels defined to allow for future expansion and allow degrees of smoothness.

X4EasyDeviceGetSmoothness

```
typedef enum _X4SMOOTHNESS {
   X4SMOOTHNESS_LEVEL0 = 0,
   X4SMOOTHNESS_LEVEL1,
   X4SMOOTHNESS_LEVEL2,
   X4SMOOTHNESS_LEVEL3,
   X4SMOOTHNESS_LEVEL4,
} X4SMOOTHNESS, *PX4SMOOTHNESS;
X4ERROR X4API X4EasyDeviceGetSmoothness(HX4DEVICE hDevice, X4SMOOTHNESS smoothness);
```

This function retrieves the value last set for the scaler smoothness parameter. See the documentation for X4EasyDeviceSetSmoothness for an explanation of the different values.

X4EasyInputGetPreferredTimings

```
typedef struct tagVDIFA
{
   unsigned long  HorFrequency;      /* Line rate in Hz. */
   unsigned long  VerFrequency;      /* Refresh rate in Hz*1000. */
   unsigned long  PixelClock;        /* Dot clock in Hz. */

   unsigned short Flags;             /* Bitwise OR of VDIF_FLAG_.*. */

   /* The following values are in pixels. */
   unsigned long  HorAddrTime;       /* Amount of active video (resolution). */
   unsigned long  HorRightBorder;
   unsigned long  HorFrontPorch;
   unsigned long  HorSyncTime;
   unsigned long  HorBackPorch;
   unsigned long  HorLeftBorder;

   /* The following values are in lines. */
   unsigned long  VerAddrTime;       /* Amount of active video (resolution). */
   unsigned long  VerBottomBorder;
   unsigned long  VerFrontPorch;
   unsigned long  VerSyncTime;
   unsigned long  VerBackPorch;
   unsigned long  VerTopBorder;

   /* TODO: Could make this [1] or a pointer. */
   char           Description[128];
}  VDIFA, *PVDIFA, *LPVDIFA;
X4ERROR X4API X4EasyInputGetPreferredTimings(HX4DEVICE hDevice,
                                             PVDIF pVDIF,
                                             unsigned long *pTiming);
```

This function retrieves a VDIF structure from the device which represents the preferred mode contained within the EDID contained within the device. The VDIF structure contains all the information (and more) needed to fully describe a video mode. The pTiming parameter is a return value which indicates the timing source of the mode as set by X4InputSetPreferredTimings.

X4EasyInputSetPreferredTimings

```
X4ERROR X4API X4EasyInputGetPreferredTimings(HX4DEVICE hDevice,
                                             PVDIF pVDIF,
                                             unsigned long timing);
```

This function sets the preferred mode in the device's EDID to match the timings given in the supplied VDIF structure. Also, the timing parameter is stored with the EDID so that the algorithm which made the mode (as determined by the application author) can be retrieved as a hint to X4InputGetPreferredTimings. The timings values currently used for this value are: 0 - Auto, 1 - CVT, 2 – CVT Reduced Blanking, 3 - GTF, 4 - SMPTE, 5 – Custom. Any other values are undefined. Auto will auto-choose an algorithm; it is a synonym for 2 – CVT Reduced Blanking. CVT and GTF are algorithms which will, given suitable X and Y dimensions for the output mode will generate all the porches and sync times for that mode. SMPTE means that the mode was derived from the SMPTE standard TV modes, so will have known characteristics which can be deduced from the active size and refresh rate of the mode. Discussion of these algorithms is beyond the scope of this document.

Note that although most parameters in this structure are straightforward and named for what they do, VerFrequency has slightly non-intuitive units – milli-Hertz. This allows for accuracy down to three decimal places in the Vertical frequency; 59.940Hz is represented as 59940. HorFrequency is absolutely straightforward with units of Hz, so 63.480kHz will be represented in HorFrequency as 63480.

X4EasyInputIsEqualisationSupported

```
X4ERROR X4API X4EasyInputIsEqualisationSupported(HX4DEVICE hDevice,
                                                 signed long *pbSupported);
```

Returns in the signed long memory pointed at by *pbSupported whether the device connected has got the equalisation chips fitted to it (and therefore whether X4EasyInputGetEqualisation will return defined values.

X4EasyInputGetEqualisation

```
typedef enum _X4EQUALISATION
{
   X4EQUALISATION_LEVEL0 = 0,
   X4EQUALISATION_LEVEL1,
   X4EQUALISATION_LEVEL2,
   X4EQUALISATION_LEVEL3,
   X4EQUALISATION_LEVEL4,
   X4EQUALISATION_LEVEL5,
   X4EQUALISATION_LEVEL6,
   X4EQUALISATION_LEVEL7,
}  X4EQUALISATION, *PX4EQUALISATION;
X4ERROR X4API X4EasyInputGetEqualisation(HX4DEVICE hDevice, PX4EQUALISATION pValue);
```

This function gets the current equalisation setting on the device's input. This equalisation is the boost applied to the incoming signal, and can compensate for very long or poor quality cable runs by boosting the received signal. This is not foolproof, and on a good signal equalisation other than 0 (off) will actually introduce problems. There will be a sweet spot in the range of values which is "best". This value will be surrounded either side by less good values. There is no guarantee that any of the values will work for any given set of input cables.

X4EasyInputSetEqualisation

```
typedef enum _X4EQUALISATION
{
   X4EQUALISATION_LEVEL0 = 0,
   X4EQUALISATION_LEVEL1,
   X4EQUALISATION_LEVEL2,
   X4EQUALISATION_LEVEL3,
   X4EQUALISATION_LEVEL4,
   X4EQUALISATION_LEVEL5,
   X4EQUALISATION_LEVEL6,
   X4EQUALISATION_LEVEL7,
}  X4EQUALISATION, *PX4EQUALISATION;
X4ERROR X4API X4EasyInputSetEqualisation(HX4DEVICE hDevice, X4EQUALISATION value);
```

This function sets the equalisation value on the device's input. Please see X4EasyInputGetEqualisation for explanation on the values sent into this function.

X4EasyInputGetEdid

```
X4ERROR X4API X4EasyInputGetEdid(HX4DEVICE hDevice, char *pEdid);
```

This function retrieves the EDID from the device which the device will present to video sources that ask for it. Note that this EDID contains the encoded preferred mode which can be read/written with X4EasyInput[Get|Set]PreferredTimings(). The memory pointed at by pEdid should be at least 256 bytes long; whilst the EDID shipped in the device is currently a 128 byte EDID, 256 byte EDIDs are supported, and may be read back.

X4EasyInputSetEdid

```
X4ERROR X4API X4EasyInputSetEdid(HX4DEVICE hDevice, char * pEdid);
```

This function sets the EDID in the device specified. Note that no error checking is performed on the EDID; the bytes pointed to by pEdid will be written into the device as-is.

X4EasyInputGetCurrentTimings

```
X4ERROR X4API X4EasyInputGetCurrentTimings(HX4DEVICE hDevice,
                                           PVDIF pVDIF,
                                           unsigned long *pTiming);
```

This function retrieves the current input mode's timings for the device specified. Please see X4EasyInputGetPreferredTimings for the definition of and information about a VDIF structure. The pTiming pointer should point to an unsigned long which receives the timing standard associated with the timings. See X4EasyInputGetPreferredTimings for a list of supported timings values.

X4EasyInputGetDualLinkStatus

```
X4ERROR X4API X4EasyInputGetDualLinkStatus(HX4DEVICE hDevice,
                                           int *pbDualLink,
                                           void * pReserved);
```

This function retrieves the dual link state for the current input mode. pbDualLink will be set to 0 if the current input mode is single link (or is a dual link mode carried over a single link cable). It will be not 0 if the current video mode on the input is dual link. pReserved is reserved for future use, and should be set to NULL.

X4EasyOutputGetCount

```
X4ERROR X4API X4EasyOutputGetCount(HX4DEVICE hDevice, unsigned long *pCount);
```

This function is used to retrieve the number of outputs on the device.

X4EasyOutputGetPreferredTimings

```
X4ERROR X4API X4EasyOutputGetPreferredTimings(HX4DEVICE hDevice,
                                              unsigned long output,
                                              PVDIF pVDIF,
                                              unsigned long *pTiming);
```

This function is used to retrieve the preferred mode contained in the EDID inside the monitor connected to the specified output channel. For a definition of and details about the VDIF structure and the pTiming parameter, please see X4EasyInputGetPreferredTimings() above.

X4EasyOutputGetCurrentTimings

```
X4ERROR X4API X4EasyOutputGetCurrentTimings(HX4DEVICE hDevice,
                                            unsigned long output,
                                            PVDIF pFVDIF,
                                            unsigned long *pTiming);
```

This function is used to retrieve the video mode which is currently being output by the specified output channel on the specified device. For a definition of and details about the VDIF structure and the pTiming parameter, please see X4EasyInputGetPreferredTimings() above.

X4EasyOutputGetDefaultTimings

```
X4ERROR X4API X4EasyOutputGetDefaultTimings(HX4DEVICE hDevice,
                                            unsigned long output,
```

```
                                        PVDIF pFVDIF,
                                        unsigned long *pTiming);
```

This function is used to retrieve the video mode which will be used as the default mode on the devices's output when no EDID information is retrieved from the monitor. For a definition of and details about the VDIF structure and the pTiming parameter, please see X4EasyInputGetPreferredTimings() above.

X4EasyOutputSetDefaultTimings

```
X4ERROR X4API X4EasyOutputSetDefaultTimings(HX4DEVICE hDevice,
                                        unsigned long output,
                                        PVDIF pFVDIF,
                                        unsigned long *pTiming);
```

This function is used to set the default mode – the mode which will be used by the device as its output mode if no EDID was read on that output. For a definition of and details about the VDIF structure and the pTiming parameter, please see X4EasyInputGetPreferredTimings() above.

Note that the outputs of the device are single link only, therefore the output mode's pixel clock must be less than or equal to 165MHz.

X4EasyOutputSetTimingSource

```
typedef enum _X4TIMINGSOURCE
{
   MONITOR,
   DEFAULT,
}  X4TIMINGSOURCE, *PX4TIMINGSOURCE;
X4ERROR X4API X4EasyOutputSetTimingSource(HX4DEVICE hDevice,
                                        unsigned long output,
                                        X4TIMINGSOURCE Source,
                                        unsigned long refOutput);
```

This call is used to set the behaviour of the output, specifically which video mode it will generate given the current conditions.

An output can generate either the preferred mode contained in the EDID of a connected monitor, or the default mode. An output will, if Source == MONITOR, output the mode contained in the EDID, if it's present. If an EDID is not present, or Source == DEFAULT, then the output will generate the mode contained in the default mode (see X4EasyOutputSetDefaultTimings above).

In normal operation, refOutput's value should equal the value contained in output; this parameter specifies the output channel from which the specified channel will draw its EDID from. For instance, if all outputs had refOutput set to 3, then the EDID in the monitor connected to output 3 would determine the mode generated by this output; all outputs will then operate in framelock mode.

X4EasyOutputGetTimingSource

```
X4ERROR X4API X4EasyOutputGetTimingSource(HX4DEVICE hDevice,
                                        unsigned long output,
                                        PX4TIMINGSOURCE pSource,
                                        unsigned long *refOutput);
```

This call will return in *pSource the timing source (MONITOR or DEFAULT) currently in effect for the given output. *refOutput will contain the output number from which this output will derive its EDID. See X4EasyOutputSetTimingSource() for more information about this parameter.

X4EasyOutputSetCropping

```
X4ERROR X4API X4EasyOutputSetCropping(HX4DEVICE hDevice,
```

```
                                    unsigned long output,
                                    unsigned long top,
                                    unsigned long left,
                                    unsigned long right,
                                    unsigned long bottom);
```

This is the major configuration of the device from a user perspective. It sets the crop area of the specified output, i.e. the pixels from the input which will be displayed on the output. The parameters are all in input mode independent units; use the provided function X4EasyCoordinateConvertInputToMI() to move from input units to these mode-independent units.

The device has some restrictions on which areas can be cropped out. The firmware will silently correct for these conditions, but the restrictions are on the horizontal dimension. The start pixel must be even (0 being even for the purposes of this discussion), and the number of pixels to be cropped out horizontally must be even. As a caveat, when rotations of 90 or 270 degrees are applied, the vertical numbers must obey these two conditions, vertical becomes horizontal after these rotations.

Should the parameters specify, or the input mode change to force, a cropping width or height which is larger than the output mode width or height, then the firmware will automatically crop-within-the-crop such that the centre pixel which would be displayed is kept at the centre of the crop, but the crop is shrunk so that the resultant image is the size of the output mode.

X4EasyOutputGetCropping

```
X4ERROR X4API X4EasyOutputGetCropping(HX4DEVICE hDevice,
                                      unsigned long output,
                                      unsigned long * top,
                                      unsigned long * left,
                                      unsigned long * right,
                                      unsigned long * bottom);
```

This function returns the cropping in effect from the specified channel. On successful return, *top, *left, *right and *bottom will contain the resolution independent values which are being used to crop that output's area from the input. Please pass these values into X4EasyCoordinateConvertMIToInput() to get the absolute input-mode-specific pixel values which are currently being used.

X4EasyOutputSetTransformation

```
typedef enum _X4TRANSFORM
{
   ROTATION_NONE = 0,
   ROTATION_90,
   ROTATION_180,
   ROTATION_270,
   FLIP_HORZ,
   FLIP_VERT,
}  X4TRANSFORM, *PX4TRANSFORM;
X4ERROR X4API X4EasyOutputSetTransformation(HX4DEVICE hDevice,
                                            unsigned long output,
                                            X4TRANSFORM transform);
```

This function sets the rotation for a specific output which will display the cropped source image "correctly" in a monitor which has been rotated this many degrees clockwise.

So, for an output displaying to a monitor which has been rotated 90 degrees anti-clockwise, use the ROTATION_270. The FLIP_HORZ and FLIP_VERT transforms are useful for projectors; they raster the horizontal or vertical in the opposite direction. Only one transform can be applied at once.

X4EasyOutputGetTransformation

```
X4ERROR X4API X4EasyOutputGetTransformation(HX4DEVICE hDevice,
                                            unsigned long output,
                                            PX4TRANSFORM pTransform);
```

This function retrieves the rotation currently in effect for the specified output and places it in *pTransform. Please see X4OutputSetTranformation for an explanation of the values in the X4TRANSFORM enumeration.

X4EasyOutputEnable

```
X4ERROR X4API X4EasyOutputEnable(HX4DEVICE hDevice, unsigned long output, BOOL bEnable);
```

This call is designed to provide the ability to turn off unused outputs, or to turn on outputs which should be enabled.

This function is unimplemented as of this release and will return X4ERROR_INVALID_METHOD for all calls; instead, it is recommended to not connect monitors to unused outputs. Whilst this function returns X4ERROR_INVALID_METHOD, the output should be considered to be in its "Enabled" state.

X4EasyOutputIsEnabled

```
X4ERROR X4API X4EasyOutputIsEnabled(HX4DEVICE hDevice,
                                    unsigned long output,
                                    unsigned long *pbEnable);
```

This call will fetch the output's enabled state and place it into *pbEnable. Whilst the X4EasyOutputEnable method returns X4ERROR_INVALID_METHOD – i.e. an application cannot influence the enabled state of an output – this function will return a value of TRUE in *pbEnable.

X4EasyCalculateVideoTimings

```
X4ERROR X4API X4EasyCalculateVideoTimings(PVDIF pVDIF, unsigned l`ong timing);
```

This function is provided for your convenience; it will, when the VDIF structure's HorAddrTime, VerAddrTime and VerFrequency are filled in with the desired values for the generated mode, and passed a valid timing enumeration, calculate the video mode components for the desired mode and return the values in *pVDIF.

Passing in pVDIF->HorAddrTime=1920, pVDIF->VerAddrTime=1080, pVDIF->VerFrequency=59940 and timings=SMPTE_TIMING_MODE to this function will pass back a filled in VDIF which when presented to X4OutputSetDefaultTimings will make the device's output generate a signal which fully conforms to the SMPTE 1080p timings at 59.94Hz.

X4EasyOutputSetSlicing

```
typedef struct _X4OUTPUT
{
   unsigned long  Size;
   unsigned long  top;
   unsigned long  left;
   unsigned long  right;
   unsigned long  bottom;
   unsigned long  transform;
}  X4OUTPUT, *PX4OUTPUT;
X4ERROR X4API X4EasyOutputSetSlicing(HX4DEVICE hDevice, PX4OUTPUT pOutput);
```

This function is a convenience function supplied as a sibling of X4OutputSetCropping and X4OutputSetTransform. This function wraps up those two functions and allows setting of all the cropping and transform properties of all four outputs at once. The pOutput parameter should point to an array of X4OUTPUT structures, and have at least as many elements in the array as X4OutputGetCount() returns; there is one structure in the array per output – entry 0 corresponds to output 0, entry 1 to output 1 and so on.

Each of the top, left, right and bottom parameters corresponds to the identically named parameter of X4EasyOutputSetCropping – and the absolute pixel values should, therefore, be derived from a call to X4EasyCoordinateConvertInputToMI(). The transform parameter likewise corresponds to the same parameter in X4EasyOutputSetTransform.

X4EasyOutputGetSlicing

```
X4ERROR X4API X4EasyOutputSetSlicing(HX4DEVICE hDevice, PX4OUTPUT pOutput);
```

This function is the mirror of X4EasyOutputSetSlicing; it will retrieve all four outputs' cropping and transform properties in one call. As with X4EasyOutputSetSlicing, pOutput points to an array of structures of type X4OUTPUT, where the number of array elements should not be smaller than the number of outputs returned by X4OuputGetCount(). On successful return, the structures in the array correspond to the output sequenced by array index, and the values within the structures are identical to the values returned by X4EasyOutputGetCropping() and X4EasyOutputGetTransform(). For the cropping parameters, therefore, the values passed back should be in turn passed to X4EasyCoordinateConvertMIToInput() to get absolute pixel values.

X4EasyCoordinateConvertMIToInput

```
X4ERROR X4API X4EasyDeviceConvertMIToInput(HX4DEVICE hDevice,
                                           PX4OUTPUT pMI,
                                           PX4OUTPUT pInput);
```

This function will convert mode independent cropping parameters into absolute pixel values, as used in the device for the current input mode if present, or EDID mode if not. This abstraction is intended to increase the accuracy of the mode-independance of the cropping values from the input mode. This means that a default quarter (for instance) set up against a low-resolution input video mode will be quartered accurately when a high-resolution mode is input.

The Device Independent figures should be treated as bit patterns to be passed into X4EASY API functions. If no video mode is present on the input at the time of this call, then the mode described in the EDID of the device will be used instead as the target video mode. Note that the transform member of the X4OUTPUT structure is not used in this call.

X4EasyCoordinateConvertInputToMI

```
X4ERROR X4API X4EasyDeviceConvertInputToMI(HX4DEVICE hDevice,
                                           PX4OUTPUT pInput,
                                           PX4OUTPUT pMI);
```

This function will take the pixel-specified cropping values (against the current input mode, if one is present; EDID mode otherwise) in *pInput and turn them into mode independent values suitable for passing into the X4EASY API functions which require them. The values are returned in *pMI. The returned values will be the pixel values that would be used in the cropping engine of the device when presented with the mode independent values as supplied to this function, given the current input video mode. If there is no input video mode, then the mode described in the EDID preferred mode will be used instead as the source mode. Note that the transform member of the X4OUTPUT structure is not used in this call.

Contents