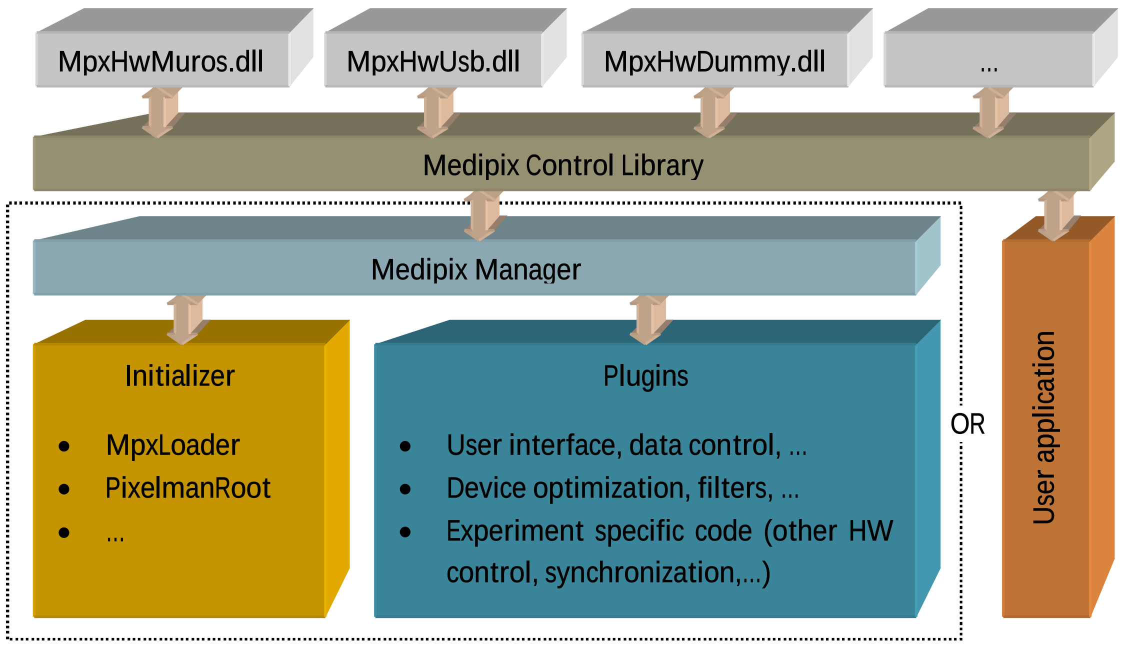
Pixelman architecture

Pixelman consists of several layers:

- HW libraries (MpxHwMuros.dll, MpxHwUsb.dll, MpxHwDummy.dll)
- Medipix Control library (MpxCtrl.dll)
- Medipix Manager (MpxManager.dll, MpxMgrUtils.dll)
- Plugins (MpxCtrlUI.dll, ThsEqualization.dll, DacPanel.dll, FrameBrowser.dll, FChEditor.dll, ...)

The architecture is depicted in following figure:



Introduction

Header files

There are following main “public” header files:

- common.h – definition of common macros, typedefs and structures which are used on all levels (HW libraries, plugins). This header file is included by other public header files, there is no need to include it directly.
- mpxhw.h – definition of interface between Medipix Control Library and HW libraries. You will need this header file if you want to implement HW library or if you want to use some HW library directly.
- mpxctrlapi.h – interface for access to Medipix Control Library. You will need this header file (with corresponding import library) only if you want to use Medipix Control Library directly.

- mpxmanagerapi.h – interface to control Pixelman/Medipix Manager by external application. You will use this header file (with corresponding import library) if you need to use your own initializer or control Pixelman from your own application.
- mpxpluginmgrapi.h – interface for Pixelman plugins.

Data types

The following typedefs are defined in common.h:

```
typedef char i8;
typedef unsigned char u8;
typedef short i16;
typedef unsigned short u16;
typedef long i32;
typedef unsigned long u32;
typedef int BOOL;
typedef unsigned char byte;
typedef unsigned short DACTYPE;          // type for DAC value
typedef INT_PTR INTPTR;                  // integral type, also safe for
storing pointer (32/64 bit platform)
```

Some functions or structures use general buffer (byte *) and buffer type is specified by following enum type:

```
typedef enum _Data_Types
{
    TYPE_BOOL   = 0,        // C bool value (int)
    TYPE_CHAR   = 1,        // signed char
    TYPE_UCHAR  = 2,        // unsigned char
    TYPE_BYTE   = 3,        // byte (unsigned char)
    TYPE_I16    = 4,        // signed short
    TYPE_U16    = 5,        // unsigned short
    TYPE_I32    = 6,        // int
    TYPE_U32    = 7,        // unsigned int
    TYPE_FLOAT  = 8,        // float
    TYPE_DOUBLE = 9,        // double
    TYPE_STRING = 10,       // zero terminated string
    TYPE_LAST   = 11,       // border
} Data_Types;
```

HW libraries

For each type of Medipix interface there is a single library (DLL) which provides access to devices connected via this interface. Pixelman uses ONLY this library to access particular Medipix interface. There are currently 3 HW libraries – MpxHwMuros.dll (National Instruments card + MUROS), MpxHwUSb.dll (USB interface 1.x) and MpxHwDummy.dll (dummy device – Medipix simulation).

Initialization

During initialization of Medipix Control library (on startup) hwlibs directory is searched and every DLL within this directory is loaded as HW library (run-time linking). The interface between HW library and Medipix Control library is defined in mpxhw.h. Each HW library has to export function named `getMandatoryFuncs()` of type `GetMandatoryFuncsType`. This function has to return pointer to valid `MpxHwFuncs` structure containing pointers to functions which form interface between Medipix

Control layer (MpxCtrl.dll) and HW library. Each function in this structure has to be implemented. MpxCtrl uses hwGetFirst()/hwGetNext() functions to obtain all devices which are connected via hwlib-type interfaces. Each device is identified via unique hwID (which is generated/selected by hwlib and provided via hwGetFirst()/hwGetNext()). HW library should perform its initialization/device searching on hwGetFirst() call. HW library should look for new devices (if hot plug is supported) each time hwGetFirst() is called (hwID of already connected devices should remain the same). For each detected device HW library should manage DevInfo structure:

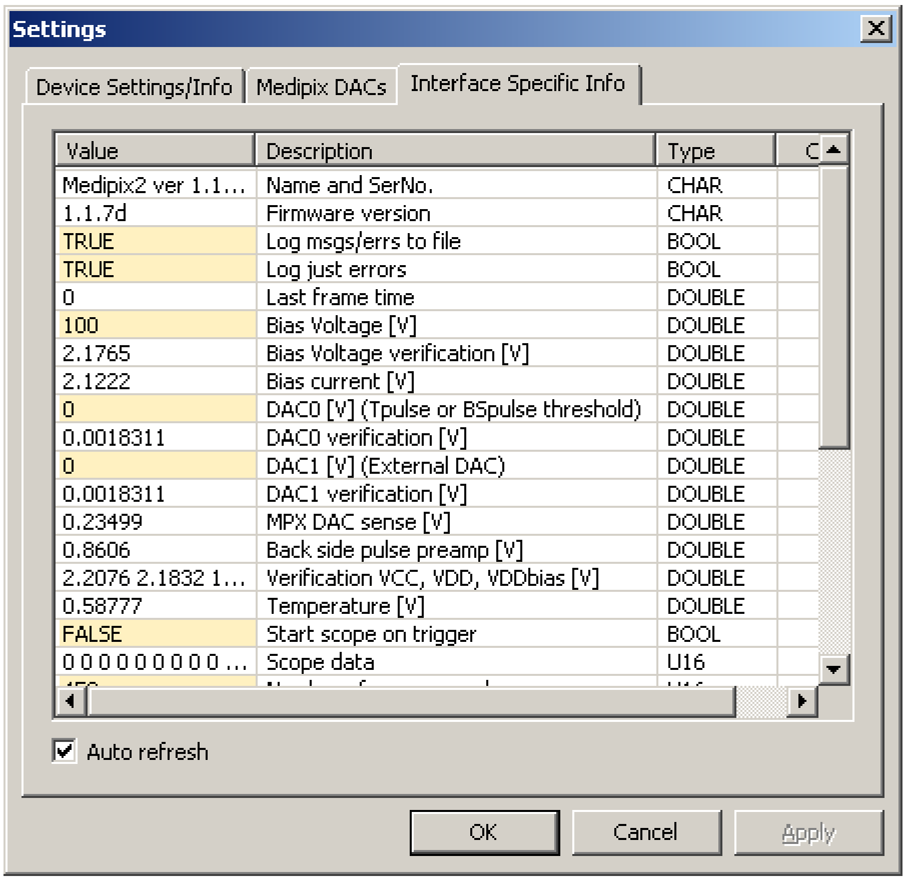
```
typedef struct _DevInfo
{
    int pixCount;                       // total number of pixels
    int rowLen;                         // length of row in pixels (e.g 256
for single chip, 512 for quad);
    int numberOfChips;                  // number of chips
    int numberOfRows;                   // number of rows in which chips
are aligned (e.g. quad has 4 chips, which are in 2 rows)
    int mpxType;                        // medipix type - MPX_ORIG,
MPX_MXR, MPX_TPX
    char chipboardID[MPX_MAX_CHBID];    // id of chip/chipboard
    const char *ifaceName;              // name of interface

    u32 suppAcqModes;                   // supported acq. mode bitwise
combinations of ACQMODE_xxx flags
    BOOL suppCallback;                  // callback is supported (acq. is
finished, triggers)

    double clockReadout;                // clock frequency [MHz] for
readout
    double clockTimepix;                // clock in frequency [MHz] for
Timepix information
    // hw timer capabilities
    double timerMinVal;                 // minimum value of hw timer [s]
    double timerMaxVal;                 // maximum value of hw timer [s]
    double timerStep;                   // step of hw timer [s]

    // test pulse capabilities
    u32 maxPulseCount;                  // maximum number of pulses that
can be sent
    double maxPulseHeight;              // max pulse height [V]
    double maxPulsePeriod;              // max period of pulses [s], length
of pulse should be period/2

    // ext DAC capabilities
    double extDacMinV;                  // minimum external DAC voltage
    double extDacMaxV;                  // maximum external DAC voltage
    double extDacStep;                  // ext. DAC step size
} DevInfo;
```

Interface specific settings

Interface-specific settings are handled by hwGetHwInfoCount(), hwGetHwInfoFlags(), hwGetHwInfo(), hwSetHwInfo() functions. HwInfoItem structure is used to make accessible interface-specific settings. In HwInfoItem structure is "complete" description of custom data attribute (data type, name, size, ...). With this information MpxCtrl layer and layers above can make accessible this information to user. The usage of these data can be controlled via MPX_HWINFO_XXX flags (bitwise ORed). MPX_HWINFO_CFGSAVE means that value will be stored in *.mcf file.

`MPX_HWINFO_CANCHANGE` means that this value can be set by user (it is not just informative value). `MPX_HWINFO_MPXFRAME` means that this value will be stored as an attribute (meta data) for each frame.

Interface specific setting is available for Pixelman user via MpxCtrlUI.dll plugin:



Value	Description	Type	C
Medipix2 ver 1.1...	Name and SerNo.	CHAR	
1.1.7d	Firmware version	CHAR	
TRUE	Log msgs/errs to file	BOOL	
TRUE	Log just errors	BOOL	
0	Last frame time	DOUBLE	
100	Bias Voltage [V]	DOUBLE	
2.1765	Bias Voltage verification [V]	DOUBLE	
2.1222	Bias current [V]	DOUBLE	
0	DAC0 [V] (Tpulse or BSpulse threshold)	DOUBLE	
0.0018311	DAC0 verification [V]	DOUBLE	
0	DAC1 [V] (External DAC)	DOUBLE	
0.0018311	DAC1 verification [V]	DOUBLE	
0.23499	MPX DAC sense [V]	DOUBLE	
0.8606	Back side pulse preamp [V]	DOUBLE	
2.2076 2.1832 1...	Verification VCC, VDD, VDDbias [V]	DOUBLE	
0.58777	Temperature [V]	DOUBLE	
FALSE	Start scope on trigger	BOOL	
0 0 0 0 0 0 0 0 0 0 ...	Scope data	U16	

USB interface page

Thread blocking scheme

All operations (except acquisition `hwStartAcquisition()`) has to be blocking (function should return after operation is finished). `hwStartAcquisition()` is non-blocking, it should return even if acq. is not started (e.g. if acq. is started by trigger). Acquisition thread waits in MpxCtrl layer - it measures time (if in manual PC timer mode), checks busy state (manual HW timer) or waits for callbacks from hwlib (trigger starts, stops acq, ...). All functions should return 0 (`MPXERR_NOERROR`) on success and negative value on error.

DACs and pixel configuration bits

DACs values are provided by array of `DACTYPE` values. The order of DACs in this array is defined by following enums (for each chip-type):

```
// order of the Medipix DACs in array DACTYPE[] for mpxCtrlSetDACs
typedef enum _DACS_ORDER
{
    DELAYN = 0,
    DISC = 1,
    PREAMP = 2,
    SETDISC = 3,
    THS = 4,
    IKRUM = 5,
    ABUFFER = 6,
    VTHH = 7,
    VTHL = 8,
    VFBK = 9,
    VGND = 10,
    BIASLVDSTX = 11,
    REFLVDSTX = 12,
    IKRUMHALF = 13,
} DACS_ORDER;

// order of the Medipix MXR DACs in array DACTYPE[] for mpxCtrlSetDACs
typedef enum _DACS_ORDER_MXR
{
    MXR_IKRUM = 0,
    MXR_DISC = 1,
    MXR_PREAMP = 2,
    MXR_BUFFA = 3,
    MXR_BUFFB = 4,
    MXR_DELAYN = 5,
    MXR_THLFINE = 6,
    MXR_THLCOARSE = 7,
    MXR_THHFINE = 8,
    MXR_THHCOARSE = 9,
    MXR_FBK = 10,
    MXR_GND = 11,
    MXR_THS = 12,
    MXR_BIASLVDS = 13,
    MXR_REFLVDS = 14,
} DACS_ORDER_MXR;

// order of the Medipix TPX DACs in array DACTYPE[] for mpxCtrlSetDACs
typedef enum _DACS_ORDER_TPX
{
    TPX_IKRUM = 0,
    TPX_DISC = 1,
    TPX_PREAMP = 2,
    TPX_BUFFA = 3,
    TPX_BUFFB = 4,
    TPX_HIST = 5,
    TPX_THLFINE = 6,
    TPX_THLCOARSE = 7,
    TPX_VCAS = 8,
    TPX_FBK = 9,
    TPX_GND = 10,
    TPX_THS = 11,
    TPX_BIASLVDS = 12,
    TPX_REFLVDS = 13,
} DACS_ORDER_TPX;
```

The order of pixel configuration bits (for each pixel 1 byte) is defined by following structeres:

```
// structure describing pixel configuration for Mpx 2.1 and MXR
typedef struct _PixelCfg
{
    byte maskBit: 1;         // mask bit (1 bit, low (0) is ACTIVE)
    byte testBit: 1;         // test bit (1 bit, low (0) is ACTIVE)
    byte lowTh: 3;           // low threshold (3 bits, low (0) is ACTIVE)
    byte highTh: 3;         // high threshold (3 bits, low (0) is ACTIVE)
} PixelCfg;

// structure describing pixel configuration for TimePix
typedef struct _TpxPixCfg
{
    byte maskBit: 1;        // mask bit (1 bit, low (0) is ACTIVE)
    byte testBit: 1;        // test bit (1 bit, low (0) is ACTIVE)
    byte thlAdj: 4;         // threshold adjustment
    byte mode: 2;           // pixel mode = {p0, p1}; mode: 0 - medipix, 1
- TOT, 2 - Timepix 1-hit, 3 - Timepix
} TpxPixCfg;
```

Functions description (from mpxhw.h):

```
// for runtime linking of HW library call function getMandatoryFuncs() of
type GetMandatoryFuncsType
// to obtain table of functions which every HW library has to provide
typedef MpxHwFuncs* (*GetMandatoryFuncsType)();

// hw enumarations, return 0 if hw was found, > 0 if no more devices, < 0
if error
// [out] hwID - hwID of first "found" device
typedef int (*HwGetFirstType)(int *hwID);

// hw enumarations, return 0 if hw was found, > 0 if no more devices, < 0
if error
// [in/out] hwID - input contains hwID of previous call of
HwInitFirstType/HwInitNextType, ouput - hwID of next found device
typedef int (*HwGetNextType)(int *hwID);

// initialize device associated with hwID
// [in] hwID - device indentification
typedef int (*HwInitType)(int hwID);

// registers callback
// [in] cb - callback
typedef int (*HwSetCbType)(HwCallback cb);

// sets "callback tag" for device
// [in] hwID - hw identification
// [in] tag - parameter for callback function, with this parameter callback
has to be called
typedef int (*HwSetCbTagType)(int hwID, TAGTYPE tag);

// returns number of HwInfo items for hwlibrary
typedef int (*HwGetHwInfoCountType)();

// getting HwInfo item flag
// [in] hwID - hw identification
// [in] index - index of HwItem (0..HwGetHwInfoCountType()-1)
// [out] flags - flag of item
typedef int (*HwGetHwInfoFlagsType)(int hwID, int index, u32 *flags);

// getting HwInfo item
// [in] hwID - hw identification
// [in] index - index of HwItem (0..HwGetHwInfoCountType()-1)
// [out] hwInfo - output structure of hw item
// [in/out] dataSize - size of allocated hwInfo->data buffer, if buffer is
not big enough dataSize will contain required size
typedef int (*HwGetHwInfoType)(int hwID, int index, HwInfoItem *hwInfo, int
*dataSize);

// setting data of HwInfo item
// [in] hwID - hw identification
// [in] index - index of HwItem (0..HwGetHwInfoCountType()-1)
// [in] data - data for specified HwItem
// [in] dataSize - size of data
typedef int (*HwSetHwInfoType)(int hwID, int index, void *data, int
dataSize);

// fills DevInfo structure (informations about mpx and interface
capabilities)
// [in] hwID - hw identification
// [out] devInfo - "device info" structure that should be filled
typedef int (*HwGetDevInfoType)(int hwID, DevInfo *devInfo);

// gets time of last acquisition
// [in] hwID - hw identification
// [out] time - acq. time (time has to be measured even for "manual" mode)
typedef int (*HwGetAcqTimeType)(int hwID, double *time);

// sets acq. parameters
// [in] hwID - hw identification
// [in] pars - acquisition parameters/settings
typedef int (*HwSetAcqParsType)(int hwID, AcqParams *pars);

// starts aquisition
// [in] hwID - hw identification
typedef int (*HwStartAcquisitionType)(int hwID);

// stops aquisition
// [in] hwID - hw identification
typedef int (*HwStopAcquisitionType)(int hwID);

// [in] hwID - hw identification
// [in] pulseHeight - pulse height [V]
// [in] period - period of pulses [s] (distance between pulses is same as
length i.e. period/2)
// [in] pulseCount - number of pulses to send
typedef int (*HwTestPulsesType)(int hwID, double pulseHeight, double
period, u32 pulseCount);

// setting MPX DACs
// [in] hwID - hw identification
// [in] dacVals - array of DAC values (number of chips)*(number of DACs),
order of DACs is given by enum type DACS_ORDER
// [in] size - size of dacVals array (number of items for size check)
// [in] senseChip - 8bit mask for enabling/disabling preparation of ADC
conversion of DAC selected in codes array
// [in] extDacChip - 8bit mask for enabling/disabling usage of external DAC
for DAC selected in codes array
// [in] codes - is [number of chip] array which contains DAC codes for
individual chips for ADC conversion/external DAC selection
// [in] tpReg - setting test pulse register in MXR, ignored for original
chip
typedef int (*HwSetMpxDACsType)(int hwID, DACTYPE dacVals[], int size, byte
senseChip, byte extDacChip, int codes[], u32 tpReg);
```

```
// reading (sensing) analog value of DAC, DAC which is read is set by DAC
code in codes array in HwSetMpxDACsType
// [in] hwID - hw identification
// [in] chipNumber - chip that should be sensed
// [out] value - sensed analog value
typedef int (*HwGetMpxDacValType)(int hwID, int chipNumber, double *value);

// [in] hwID - hw identification
// [in] value - value in Volts
typedef int (*HwSetExtDacValType)(int hwID, double value);

// set pixels mask
// [in] hwID - hw identification
// [in] cfgs - array [number of chips * MATRIX_SIZE] of pix. configurations
bits
// [in] size - size of cfgs array (number of items for size check)
typedef int (*HwSetPixelsCfgType)(int hwID, byte cfgs[], u32 size);

// reset matrix
// [in] hwID - hw identification
typedef int (*HwResetMatrixType)(int hwID);

// matrix readout (deserialized and converted from pseudo)
// [in] hwID - hw identification
// [out] buff - output buffer, size of buffer has to be at least
(MATRIX_SIZE*numberOfChips)
// [in] buffSize - size of buff (number of items for size check)
typedef int (*HwReadMatrixType)(int hwID, i16 *buff, u32 buffSize);

// write matrix to medipix (for each chip MATRIX_SIZE matrix)
// [in] hwID - hw identification
// [in] buff - matrix that should be written, size of buffer has to be at
least (MATRIX_SIZE*numberOfChips)
// [in] buffSize - size of buff (number of items for size check)
typedef int (*HwWriteMatrixType)(int hwID, i16 *buff, u32 buffSize);

// resets interface and medipix to default state
// [in] hwID - hw identification
typedef int (*HwGeneralResetType)(int hwID);

// return string describing last error
typedef const char* (*HwGetLastErrorGeneralType)();

// return string describing last error for specified HW
// [in] hwID - hw identification
typedef const char* (*HwGetLastErrorType)(int hwID);

// check if device is busy
// [in] hwID - hw identification
// [out] busy - set TRUE if busy
typedef int (*HwCheckBusyType)(int hwID, BOOL *busy);

// close device (for cleanup)
// [in] hwID - hw identification
typedef int (*HwCloseDeviceType)(int hwID);

typedef struct _MpxHwFuncs
{
    int size;
    HwGetFirstType hwGetFirst;
    HwGetNextType hwGetNext;
    HwInitType hwInit;
    HwSetCbType hwSetCb;
    HwSetCbTagType hwSetCbTag;
    HwGetHwInfoCountType hwGetHwInfoCount;
    HwGetHwInfoFlagsType hwGetHwInfoFlags;
    HwGetHwInfoType hwGetHwInfo;
    HwSetHwInfoType hwSetHwInfo;
    HwGetDevInfoType hwGetDevInfo;
    HwGetAcqTimeType hwGetAcqTime;
    HwSetAcqParsType hwSetAcqPars;
    HwStartAcquisitionType hwStartAcquisition;
    HwStopAcquisitionType hwStopAcquisition;
    HwTestPulsesType hwTestPulses;
    HwSetMpxDACsType hwSetMpxDACs;
    HwGetMpxDacValType hwGetMpxDacVal;
    HwSetExtDacValType hwSetExtDacVal;
    HwSetPixelsCfgType hwSetPixelsCfg;
    HwResetMatrixType hwResetMatrix;
    HwReadMatrixType hwReadMatrix;
    HwWriteMatrixType hwWriteMatrix;
    HwGeneralResetType hwGeneralReset;
    HwGetLastErrorGeneralType hwGetLastErrorGeneral;
    HwGetLastErrorType hwGetLastError;
    HwCheckBusyType hwCheckBusy;
    HwCloseDeviceType hwCloseDevice;
} MpxHwFuncs;
```

Medipix Control Library (MpxCtrl.dll)

Medipix Control Library (MCL) provides high level functions for different operations on Medipix devices. MCL manages configurations of all MPX devices, access synchronization, frame buffer allocation and provides callbacks for events related to MPX device.

Initialization

MCL is initialized automatically by Medipix Manager. If you want to use MCL directly it is necessary to call `initMpxCtrl()`for initialization and `exitMpxCtrl()` before unloading of library. If you want to use callbacks from MCL you should also register your callbacks by calling `setCbTable()` (before initialization). For direct usage you should contact us (medipix@utef.cvut.cz) to receive corresponding header files and import library for MCL. During MCL initialization all HW libraries in ".\hwlibs" are loaded and all MPX devices are initialized.

MPX configuration files

After first connection of MPX device the device is initialized with default configuration (according to chip type). After exit (`exitMpxCtrl()`) current configuration for each MPX is saved to default configuration files. Next time during initialization (`initMpxCtrl()`) these configuration are automatically loaded and used. Each MPX configuration is stored in *.mcf (medipix configuration file). The information stored includes DACs settings, polarity, acq. mode, timer settings, acq. mode, and interface specific settings containing flag `MPX_HWINFO_CFGSAVE` (e.g. bias settings for USB interface). The name of *.mcf consists of interface name and chip ID, e.g. USB_H10-W0015.mcf. If it is used Medipix 2.1, which does not contain ID, the ID of USB is used as replacement (e.g. USB_0069.mcf). If the interface does not support ID and Medipix 2.1 is used then predefined ID is used (e.g. dummy_2000.mcf or muros_1001.mcf) Pixel configuration bits are stored in binary format in *.bpc (binary pixel configuration) file, 1 byte for each pixel. The name for automatically stored pixel configuration is exactly the same as for *.mcf file (e.g. USB_H10-W0015.bpc). The position of individual configuration bits is defined in common.h in `PixelCfg` (Medipix 2.1 and MXR) and `TpxPixCfg` (Timepix) structures. User can also load/save individual fields in pixel configuration (in ASCII mode) via `loadPixelsCfgAscii()/savePixelsCfgAscii()`.

Access synchronization

MCL provides access synchronization for individual MPX devices. When operation (like e.g. settings DAC) is running the device is „locked“ to prevent interference of other operations (running in other threads). If another thread tries to perform some operation which may collide with currently running operation this thread waits until running operation is finished. It is possible to check if device is „locked“ or „lock“ device in advance by `tryLockDevice()` where the maximum time the caller wants to wait for device lock can be specified.

Callbacks

The callbacks available are defined in `MpxCbTable` structure. If MCL is used directly (without MpxManager) this `MpxCbTable` structure should be supplied during initialization by `setCbTable()`. Plugins and/or MpxManager user can use relevant callbacks via name (`MPXCTRL_CB_XXX`) in mpxpluginmgrapi.h.

Medipix Manager (MpxManager.dll)

Medipix manager (MM) is the core of Pixelman. It handles plugin management. It is responsible for initialization and communication of MCL and plugins. It maintains public registers of frames, functions (offered by plugins), events, filters and filter chains.

Initialization

MM is initialized simply by calling `mgrInitManager()`. When this function is called MCL is initialized and plugins according to second parameter are loaded and initialized. You can either specify list of plugins in string that should be loaded or supply 0 to let load all plugins from mpxmanager.ini. You can also specify whether tray menu should be visible and if termination of Pixelman via tray menu is possible.

```
// MM initialization
// [in] flags - flags controlling tray menu via bitwise OR-ed
MGRINIT_TRAYMENUXXX flags
// [in] pluginNames - string of \0 separated path names of plugins (e.g.
"plugins\\mpxctrlui.dll\0plugins\\thsequalization.dll\0") that should be
loaded (both absolute and relative paths are possible), 0 if plugins should
be loaded from ini file
int mgrInitManager(u32 flags, const char *pluginNames);
```

MM is load-time linked to (depends on) MCL and MpxMgrUtils.dll. MpxMgrUtils.dll contains user interface (uses MFC) used in MM – tray menu, logging window, browser of managed items.

Examples of initializations

1) Plugins are loaded from ini file, tray menu is visible and it is allowed to exit via tray menu Exit command:

```
// loading of plugins from ini
mgrInitManager(MGRINIT_TRAYMENUEXIT | MGRINIT_TRAYMENUSHOW, 0);
// we can wait until Pixelman is finished (exited via tray menu)
while (mgrIsRunning())
    Sleep(100);
// Pixelman finished…we can end this thread…
```

2) Only explicitly specified plugins are loaded, tray menu is not available:

```
// loading of specified plugins, no tray menu visible
mgrInitManager(0,
"plugins\\mpxctrlui.dll\0plugins\\thsequalization.dll\0");
// … other operations
// we have finished and we want to quit Pixelman
mgrExitManager();
```

Frames

Frame consists from data buffer which contains data matrix ("picture") and attributes (metadata). The typical source of frames is Medipix device via MCL. If acquisition series is started specified number of frame is created one by one as they are measured. Once frame is created it is available to plugins (or user application). When MCL creates frame it notifies MM (or user application if MM is not used) that frame was created and MM put it to public register of frames and generates callback event that frame was created. Of course frames can be created also by plugins (or user application). The life of frame in Pixelman is controlled by reference counter. When a frame is created by plugin (by `mgrCreateFrame()`/`mgrLoadFrame()` functions or corresponding functions in `FuncTableType` structure) its reference counter is set to 1. When certain plugin (or user application) wants to use some frame (and be sure that frame will not be released) it should "open" it by calling `mgrOpenFrame()`. Each call of `mgrOpenFrame()` increment reference counter by one. Once a creator or user of the frame does not need it anymore it should close it by calling `mgrCloseFrame()`. Each call of `mgrCloseFrame()` decrease the reference counter by one. Once the reference counter reaches 0 the frame is released and corresponding resources are freed.

The lifetime of frame created by MCL (during acquisition series) is guaranteed to be at least until next acquisition series is started on the same device. Once new acquisition series is started the frames are released if no one opened them.

The data type of frame matrix can be `i16`, `u32` or `double`. Frame can be stored (saved) by Pixelman in various formats:

- ASCII full matrix (`FSAVE_ASCII`) – full matrix with dimension width*height
- ASCII [X, C] (`FSAVE_ASCII | FSAVE_SPARSEX`) – only non-zero pixels are stored, one pixel on single line in format [x-coordinate, value]. X-coordinate can be in interval <0, width*height-1>

- ASCII [X, Y, C] (FSAVE_ASCII | FSAVE_SPARSEXY) – only non-zero pixels are stored, one pixel on single line in format [x-coordinate, y-coordinate, value]. X-coordinate can be in interval <0, width-1>, Y-coordinate can be in interval <0, height-1>
- Binary full matrix (FSAVE_BINARY) – full matrix is saved in binary. The size of matrix element depends on frame type (2 bytes for i16, 4 bytes for u32, 8 bytes for double).
- Binary [X, C] (FSAVE_BINARY | FSAVE_SPARSEX) – only non-zero pixels are stored, X-coordinate is saved as u32 (4 bytes), value of the pixel depends on frame type(2 bytes for i16, 4 bytes for u32, 8 bytes for double).
- Binary [X, Y, C] (FSAVE_BINARY | FSAVE_SPARSEXY) – only non-zero pixels are stored, X and Y coordinates are saved as u32 (4 bytes and 4 bytes), value of the pixel depends on frame type(2 bytes for i16, 4 bytes for u32, 8 bytes for double).

The format of the existing frame can be changed (mgrSetFrameType()) or output format can be forced (FSAVE_I16, FSAVE_U32, FSAVE_DOUBLE).

There are two files created when a frame is saved. First one is data file (e.g. test_frame.txt) and second one is description file (e.g. test_frame.dsc). Description file contains information about frame type and format of data file. It also contains information about all attributes (frame metadata) – their names, types and values.

The attributes which are associated with certain frame can be added by any plugin. If frame is created by MCL (measured with device) some predefined attributes which describes the condition under which measurement was performed are automatically added. Here is the list of predefined frame attributes:

```
#define FAN_MPXTYPE                    "Mpx type"
#define FAN_CHIPBOARDID                "ChipboardID"
#define FAN_INTERFACE                  "Interface"
#define FAN_MPXCLOCK                   "Mpx clock"
#define FAN_ACQTIME                    "Acq time"
#define FAN_POLARITY                   "Polarity"
#define FAN_ACQMODE                    "Acq mode"
#define FAN_HWTIMER                    "Hw timer"
#define FAN_DACS                       "DACs"
#define FAN_STARTTIME                  "Start time"
#define FAN_STARTTIMESTR               "Start time (string)"
#define FAN_APPLIEDFILTERS             "Applied filters"
```

Important types

There are few types defined and used in various function:

- DEVID – “Medipix device handle” type
- FRAMEID – handle to frame
- ITEMID – handle to various objects (functions, callbacks, filters…)
- CBPARAM – callback function parameter

Callbacks

MM maintain register of events and corresponding callbacks. Registered callback functions (of type CallbackFuncType) are called when certain event occurs. The source of such events can be MCL (callback events named MPXCTRL_CB_XXX e.g. when acquisition is started or finished), MM (callback events named MPXMGR_CB_XXX e.g. when new frame is created or deleted) or plugin/user application itself (there are functions for registration of new callback events). Any plugin (or user application) can register the callback function for any predefined (MPXCTRL_CB_XXX, MPXMGR_CB_XXX) or registered event. Once certain event occurs all callback functions registered for that event are called in the same order as they were registered.

There are following predefined events (fom mpxpluginmgrapi.h):

```
#define MPXMGR_CB_START         "Start"             // callback event name
for start event (every plugin was loaded)
#define MPXMGR_CB_EXIT          "Exit"              // callback event name
for exit event (application is shutting down, but everything is still
functional (message processing, mpxCtrl is library still alive,...)
                                                    // good time for clean
up if cleanup requires win message processing

// frame callbacks - as the parameter of callback function plugin receives
FRAMEID of corresponding frame
#define MPXMGR_CB_FRAME_NEW     "FrameCreated"      // new frame was
created
#define MPXMGR_CB_FRAME_DEL     "FrameDelete"       // frame will be
deleted (still exist during callback)
#define MPXMGR_CB_FRAME_DCHNG   "FrameDataChanged"  // frame data were
changed
#define MPXMGR_CB_FRAME_ACHNG   "FrameAttrChanged"  // frame attributes
were changed

#define MPXCTRL_NAME            "mpxctrl"           // "plugin name" for
access callback events registered by MpxCtrl library
#define MPXCTRL_CB_ACQSTART     "AcqStarted"        // callback event name
for notification of acquisition start (for triggered acq)
#define MPXCTRL_CB_ACQCOMPL     "AcqCompleted"      // callback event name
for acquisition completition
#define MPXCTRL_CB_ACQSERCOMPL  "AcqSerCompleted"   // callback event name
for acquisition series completition
#define MPXCTRL_CB_INFOMSG      "InfoMsg"           // callback event name
for info/error messages
#define MPXCTRL_CB_DACSSET      "DACs set"          // DACs were set
(changed)
#define MPXCTRL_CB_PIXCFG       "PixCfg"            // PixCfg changed
#define MPXCTRL_CB_MPXNEW       "MpxDevNew"         // new mpx device was
detected
#define MPXCTRL_CB_MPXCHANGED   "MpxDevChanged"     // mpx device was
"reconnected" (e.g. some mpx device was switched on one of interfaces)
#define MPXCTRL_CB_EXTDACSET    "ExtDACSet"         // external DAC was
changed (value and/or it replaces another DAC)
```

Structures for objects registration

As it was said there is possibility to offer (or to use) various objects via MM. Plugins or user application can register (offer) functions, events, custom attributes ... Here is the description of structures used in “registration” functions:

```
// structure describing one parameter of plugin function
typedef struct _ExtParamInfo
{
    Data_Types type;                    // parameter type
    int count;                          // count (for array of type)
    char description[DESC_LENGTH];      // description string
} ExtParamInfo;

// function flags (distinguish certain func. types)
#define FUNCFLAG_FILTER     0x01        // function is a filter, (function
parameters - two input parameters - frameID and instance id (for adjustable
filters, 0 for simple filters), so the type should be: int func(INTPTR,
&{FRAMEID, ITEMID}, 0))
#define FUNCFLAG_ADJFILTER  0x02        // function is adjustable filter
(userData has to contained pointer to filled AdjFilterPars structure)

// structure describing plugin function registered in Manager
typedef struct _ExtFunctionInfo
{
    PluginFuncType func;                    // pointer to registered
function
    int paramInCount;                       // number of input parameters
    int paramOutCount;                      // number of output parameters
    ExtParamInfo paramsInfoIn[MAX_PARAM];   // input parameters infos
    ExtParamInfo paramsInfoOut[MAX_PARAM];  // output parameters infos
    char pluginName[NAME_LENGTH];           // plugin name
    char functionName[NAME_LENGTH];         // function name
    char description[DESC_LENGTH];          // function description
    INTPTR userData;                        // custom parameter that will
be used for func call
    u32 flags;                              // combination of FUNCFLAG_XXX
    ITEMID funcID;                          // function ID (filled by
MpxManager)
} ExtFunctionInfo;

// structure describing plugin callback event registered in Manager
typedef struct _ExtCBEventInfo
{
    char pluginName[NAME_LENGTH];           // plugin name
    char cbEventName[NAME_LENGTH];          // callback event name
    char description[DESC_LENGTH];          // callback event description
} ExtCBEventInfo;

// structure to get information about certain FilterChain
typedef struct _ExtFilterChainInfo
{
    char pluginName[NAME_LENGTH];       // plugin which created filter
chain
    char filterChainName[NAME_LENGTH];  // filter chain name
    ITEMID *filterIDs;                  // pointer to buffer (provided by
user) to obtain IDs of filters which are in chain, can be NULL if caller do
not want this info
    u32 filterIDsSize;                  // size of filterIDs buffer
    char *filterNames;                  // pointer to buffer (provided by
user) to obtain names of filters which are in chain (separated by '|'), can
be NULL if caller do not want this info
    u32 filterNamesSize;                // size of filterNames buffer
} ExtFilterChainInfo;

// structure describing custom frame attribute provided by plugin (template
for attribute)
typedef struct _ExtFrameAttribInfo
{
    char pluginName[NAME_LENGTH];   // plugin which provides attribute
value
    char attribName[NAME_LENGTH];   // name of attribute
    char description[DESC_LENGTH];  // description
    Data_Types type;                // type of attribute
    INTPTR userData;                // userData for GetFrameAttribType
function
    GetFrameAttribType func;        // get attribute function
} ExtFrameAttribInfo;

// structure describing custom frame attribute provided by plugin
typedef struct _ExtFrameAttrib
{
    char name[NAME_LENGTH];         // name of attribute
    char desc[DESC_LENGTH];         // description of attribute
    Data_Types type;                // type of attribute
    int size;                       // size of data in bytes
} ExtFrameAttrib;

// codes of frame attribute change
#define FATTR_CHANGE_ADD        1       // attribute was added
#define FATTR_CHANGE_DEL        2       // attribute was deleted
#define FATTR_CHANGE_DELALL     3       // all attributes were removed
#define FATTR_CHANGE_SET        4       // attribute's value was changed

// structure for callback MPXMGR_CB_FRAME_ACHNG
typedef struct _FrameAttribChange
{
    FRAMEID frameID;            // ID of frame which attribute was changed
    char name[NAME_LENGTH];     // name of attribute
    int size;                   // size of attribute
    int changeCode;             // code of frame attribute change
} FrameAttribChange;
```

Filters and filters chains

Filter is a registered function of certain type (`int func(INTPTR, &{FRAMEID, ITEMID}, 0))` which was registered with flags parameter set to `FUNCFLAG_FILTER` (simple filter) or `FUNCFLAG_ADJFILTER` (adjustable filter). Filter takes a frame and performs some operation on that frame (which may change the frame). Simple filter has no adjustable parameters, it behaves always in the same way and therefore there is just one instance of such filter. Example of such filter can be filter which changes the frame type to `double` or filter which performs `log10` on frame. Adjustable filter has several parameters which can be changed by user. Therefore there is a need to have possibility to have any number of instances of certain filter. Example of adjustable filter is *Crop* filter which selects rectangular region of interest from frame. The parameters of such filter can be e.g. left and bottom coordinates of selection rectangle and its width and height. While for simple filter the user (plugin/user application) has to provide just filter function for adjustable filter the user has to provide also function for instance creation, instance deletion and for setting and getting parameters. This is done via `AdjFilterPars` structure:

```
// if plugin wants to register "adjustable filter" it has to fill
AdjFilterPars structure and pass it
// to manager in userData parameter in ExtFunctionInfo. UserData parameter
for filter (function)
// is passed via AdjFilterPars. Manager will then provide access to these
function via
```

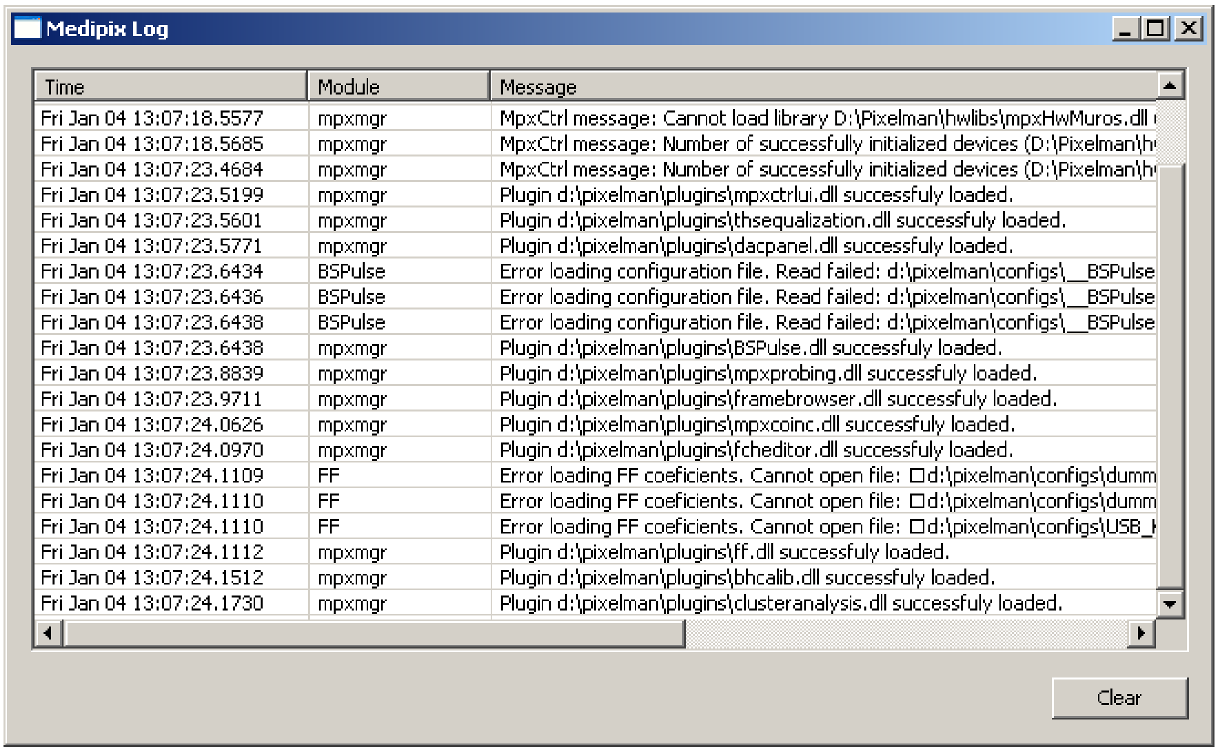
```
// MgrCreateFilterInstType, MgrDeleteFilterInstType, MgrSetFilterParType,
MgrGetFilterParType functions
// NOTE: instance ID of filter returned by MgrCreateFilterInstType is
globally unique across all plugins
// while instance ID returned by AdjFiltCreateInstType is unique just for
plugin => these IDs are not same

// Function type for creating new instance of "Adjustable filter"
// [in] userData - custom parameter that will be passed by manager to
plugin
// [out] instID - ID of filter instance (it is generated by plugin, it has
to be unique for instances of certain filter, it can be recycled (if
instance with certain id is deleted, this id can be used for new instance
again))
typedef int (*AdjFiltCreateInstType)(INTPTR userData, ITEMID *instID);

// Function type for deleting instance of "Adjustable filter"
// [in] userData - custom parameter that will be passed by manager to
plugin
// [in] instID - ID of filter instance to delete
typedef int (*AdjFiltDeleteInstType)(INTPTR userData, ITEMID instID);

// Function type for setting parameter of "Adjustable filter"
// [in] userData - custom parameter that will be passed by manager to
plugin
// [in] instID - instance ID of filter
// [in] idx - zero-based index of parameter
// [in] data - data for parameter
// [in] dataSize - size of data (in bytes)
typedef int (*AdjFiltSetParType)(INTPTR userData, ITEMID instID, int idx,
byte *data, int dataSize);

// Function type for getting informations about parameter and/or its value
of "Adjustable filter"
// [in] userData - custom parameter that will be passed by manager to
plugin
// [in] instID - instance ID of filter
// [in] idx - zero-based index of parameter or -1, in case of -1, *dataSize
will contain number of parameters of adjustable filters, (info will be
unused so it can be NULL)
// [in/out] info - data for parameter, info->data can contain allocated
buffer for param. value, if info->data is NULL info structure will be
filled without data
// [in/out] dataSize - dataSize should contain size of allocated buffer in
info->data (in bytes), after return it will contain required size of data
buffer for parameter
typedef int (*AdjFiltGetParType)(INTPTR userData, ITEMID instID, int idx,
ItemInfo *info, int *dataSize);


struct AdjFilterPars
{
    INTPTR userData;                    // user data of filter function
(userData in ExtFunctionInfo contains pointer to AdjFilterPars struct)
    INTPTR createInstUserData;          // user data parameter for provided
createInst function
    INTPTR deleteInstUserData;          // user data parameter for provided
deleteInst function
    INTPTR setParUserData;              // user data parameter for provided
setPar function
```

```
    INTPTR getParUserData;              // user data parameter for provided
getPar function
    AdjFiltCreateInstType createInst;   // function of adj. filter instance
creation
    AdjFiltDeleteInstType deleteInst;   // function of adj. filter instance
deletetion
    AdjFiltSetParType setPar;           // function for setting parameter
of adj. filter
    AdjFiltGetParType getPar;           // function for getting parameter
of adj. filter
};
```

Filter chain (FCh) is another type of object which is managed by MM. FCh is a simple way which allows convenient processing of frames. FCh is formed by series of filters. It can be created/modified/deleted by plugin/user application. Once FCh is created any plugin/user application can directly obtain output of this processing chain by calling `mgrGetFilteredFrame()`. If FCh is applied on some frame, the frame remains untouched and caller receives copy of this frame which went through all filters in FCh one by one.

Logging

MM manages system log to which any plugin/user application can write anything. The logged messages are written to file log.txt and are available in log window:

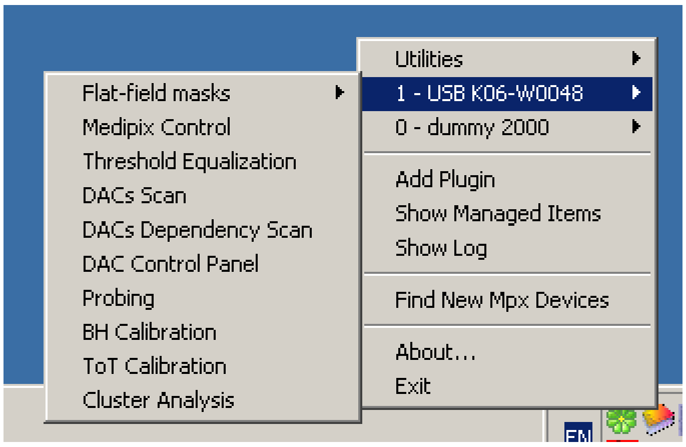


Time	Module	Message
Fri Jan 04 13:07:18.5577	mpxmgr	MpxCtrl message: Cannot load library D:\Pixelman\hwlibs\mpxHwMuros.dll
Fri Jan 04 13:07:18.5685	mpxmgr	MpxCtrl message: Number of successfully initialized devices (D:\Pixelman\h
Fri Jan 04 13:07:23.4684	mpxmgr	MpxCtrl message: Number of successfully initialized devices (D:\Pixelman\h
Fri Jan 04 13:07:23.5199	mpxmgr	Plugin d:\pixelman\plugins\mpxctrlui.dll successfuly loaded.
Fri Jan 04 13:07:23.5601	mpxmgr	Plugin d:\pixelman\plugins\thsequalization.dll successfuly loaded.
Fri Jan 04 13:07:23.5771	mpxmgr	Plugin d:\pixelman\plugins\dacpanel.dll successfuly loaded.
Fri Jan 04 13:07:23.6434	BSPulse	Error loading configuration file. Read failed: d:\pixelman\configs\__BSPulse
Fri Jan 04 13:07:23.6436	BSPulse	Error loading configuration file. Read failed: d:\pixelman\configs\__BSPulse
Fri Jan 04 13:07:23.6438	BSPulse	Error loading configuration file. Read failed: d:\pixelman\configs\__BSPulse
Fri Jan 04 13:07:23.6438	mpxmgr	Plugin d:\pixelman\plugins\BSPulse.dll successfuly loaded.
Fri Jan 04 13:07:23.8839	mpxmgr	Plugin d:\pixelman\plugins\mpxprobing.dll successfuly loaded.
Fri Jan 04 13:07:23.9711	mpxmgr	Plugin d:\pixelman\plugins\framebrowser.dll successfuly loaded.
Fri Jan 04 13:07:24.0626	mpxmgr	Plugin d:\pixelman\plugins\mpxcoinc.dll successfuly loaded.
Fri Jan 04 13:07:24.0970	mpxmgr	Plugin d:\pixelman\plugins\fcheditor.dll successfuly loaded.
Fri Jan 04 13:07:24.1109	FF	Error loading FF coeficients. Cannot open file: □d:\pixelman\configs\dumm
Fri Jan 04 13:07:24.1110	FF	Error loading FF coeficients. Cannot open file: □d:\pixelman\configs\dumm
Fri Jan 04 13:07:24.1110	FF	Error loading FF coeficients. Cannot open file: □d:\pixelman\configs\USB_
Fri Jan 04 13:07:24.1112	mpxmgr	Plugin d:\pixelman\plugins\ff.dll successfuly loaded.
Fri Jan 04 13:07:24.1512	mpxmgr	Plugin d:\pixelman\plugins\bhcalib.dll successfuly loaded.
Fri Jan 04 13:07:24.1730	mpxmgr	Plugin d:\pixelman\plugins\clusteranalysis.dll successfuly loaded.

Each logged message contains timestamp and module name which was specified in logging function (`mgrLogMsg()` or `mgrLogMsgV()`). These logging functions allows (via flag variable) to optionally display the message in modal or modeless message box and optionally append the string with system error (string created from Win API function `GetLastError()`). Error and info messages that are sent by MCL via corresponding callback are automatically logged by MM.

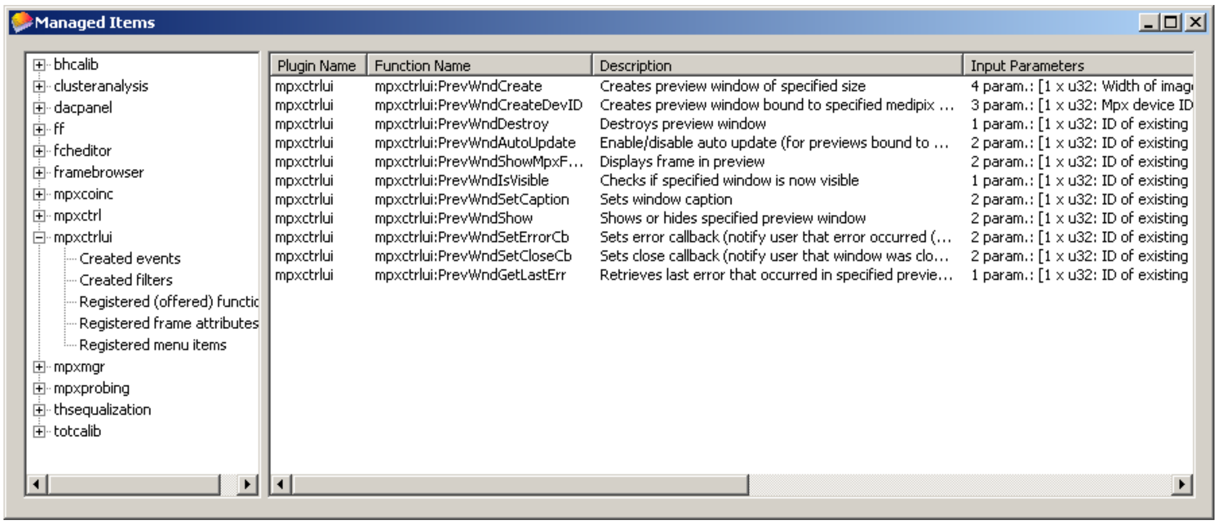
Tray menu

Tray menu contains commands for operation like adding new plugin (*Add Plugin*), showing log window (*Show Log*) etc. and menu items added by plugins/user application by calling `mgrAddMenuItem()` (adding to main menu) or `mgrAddMpxMenuItem()` (adding to submenu of device with specified device ID).



Managed objects

There is an inspector of registered objects (events, filter chains, functions, frame attributes, menu items). This allows to see e.g. names and types of all input/output parameters of functions offered by other plugins, so one can use them even in the case when no header file or function specification is available.



Plugin Name	Function Name	Description	Input Parameters
mpxctrlui	mpxctrlui:PrevWndCreate	Creates preview window of specified size	4 param.: [1 x u32: Width of imag
mpxctrlui	mpxctrlui:PrevWndCreateDevID	Creates preview window bound to specified medipix ...	3 param.: [1 x u32: Mpx device ID
mpxctrlui	mpxctrlui:PrevWndDestroy	Destroys preview window	1 param.: [1 x u32: ID of existing
mpxctrlui	mpxctrlui:PrevWndAutoUpdate	Enable/disable auto update (for previews bound to ...	2 param.: [1 x u32: ID of existing
mpxctrlui	mpxctrlui:PrevWndShowMpxF...	Displays frame in preview	2 param.: [1 x u32: ID of existing
mpxctrlui	mpxctrlui:PrevWndIsVisible	Checks if specified window is now visible	1 param.: [1 x u32: ID of existing
mpxctrlui	mpxctrlui:PrevWndSetCaption	Sets window caption	2 param.: [1 x u32: ID of existing
mpxctrlui	mpxctrlui:PrevWndShow	Shows or hides specified preview window	2 param.: [1 x u32: ID of existing
mpxctrlui	mpxctrlui:PrevWndSetErrorCb	Sets error callback (notify user that error occurred (...	2 param.: [1 x u32: ID of existing
mpxctrlui	mpxctrlui:PrevWndSetCloseCb	Sets close callback (notify user that window was clo...	2 param.: [1 x u32: ID of existing
mpxctrlui	mpxctrlui:PrevWndGetLastErr	Retrieves last error that occurred in specified previe...	1 param.: [1 x u32: ID of existing

Difference in API for user application and plugins

The functions available for plugins are subset of all functions exported by MM. There are several functions which are useful only to “user” of MM like function for MM initialization. All functions exported from MM are exported without decoration (`extern "C"`) so they can be easily accessed and used even from e.g. MATLAB. The naming convention of exported functions is following –

functions provided by MCL (via MM) have the prefix mpxCtrl (e.g. `mpxCtrlSetDACs()`), MM functions have the prefix mgr (e.g. `mgrCreateFrame()`).

The naming of functions In `FuncTableType` structure (which is give to plugin via `InitMpxPlugin()`) is following – functions provided by MCL have mpxCtrl prefix and functions provided by MM are without prefix. So the usage by plugins is following:

```
extern FuncTableType *mgr;       // pointer to function table initialized in
InitMpxPlugin()
mgr->mpxCtrlSetDACsDefault(0);  // calling of functions from MCL
mgr->createFrame(&frameID,  256,  256,  TYPE_DOUBLE,  "test  plugin");     //
calling function from MM
```

API functions description (from mpxpluginmgrapi.h):

```
// gets path to configs directory where all configuration files should be
stored
// [out] path - output buffer for path (path end with (back)slash)
typedef int (*MgrGetCfgsPathType)(char path[MPX_MAX_PATH]);


// flags for functions MgrLogMsgType
#define MGRLOG_GET_SYSERROR     0x1       // get last error and makes it
part of logged error string (to simplify error reporting when WINAPI calls
fail)
#define MGRLOG_SHOW_MSGBOX      0x2       // show messagebox with error
#define MGRLOG_SHOW_MSGBOX_UM   0x4       // show message window which is
not modal


// error logging function type
// [in] pluginName - name of plugin which request loging
// [in] flags - bitwise MGRLOG_XXX flags
// [in] format - "printf-like" format string
// [in] ... - ellipsis (printf-like)
typedef int (*MgrLogMsgType)(const char *pluginName, const char *format,
UINT flags, ...);

// error login function type
// [in] pluginName - name of plugin which request loging
// [in] flags - bitwise MGRLOG_XXX flags
// [in] format - "printf-like" format string
// [in] argptr - variable argument list (vprintf-like)
typedef int (*MgrLogMsgVType)(const char *pluginName, const char *format,
UINT flags, va_list argptr);

// shows/hides log window error login functions types
// [in] show - TRUE to show log window, FALSE to hide it
typedef int (*MgrLogShowWindowType)(BOOL show);

// shows/hides windows with list of items managed by manager
// [in] show - TRUE to show window, FALSE to hide it
typedef int (*MgrManagedItemsShowWindowType)(BOOL show);

// function for registering plugin in Manager
// [in] fileName - fileNamePath to DLL
typedef int (*MgrAddPluginType)(const char *fileName);

// function type for adding item in Loader menu
// [in] pluginName - name of plugin which adds item to menu
// [in] itemName - name of item in menu
// [in] menuPath - sequence of submenus where menuitem will be placed,
submenus are separated by pipe (|) e.g. "Special Plugins|Filteres"
// [in] func - pointer to function that will be called when user clicks on
menuitem
// [in] param - parameter which will be given to function 'func'
// [out] menuItemID - pointer to ID which will be assigned to this
menuitem, this can be 0 (NULL) if caller is not interested in this value
(this ID is used for removing menuitem from menu)
// [in] flags - reserved flags, use 0
typedef int (*MgrAddMenuItemType)(const char *pluginName, const char
*itemName, const char *menuPath, MenuFuncType func, MENUFUNCPAR param,
ITEMID *menuItemID, u32 flags);

// function type for adding item in Loader menu to submenu corresponding to
Medipix device identified by devID
// [in] devID - device ID of medipix
// [in] pluginName - name of plugin which adds item to menu
// [in] itemName - name of item in menu
// [in] menuPath - sequence of submenus where menuitem will be placed,
submenus are separated by pipe (|) e.g. "Special Plugins|Filteres", these
submenus will be placed in submenu correspnding to spedified mpx device
// [in] func - pointer to function that will be called when user clicks on
menuitem
// [in] param - parameter which will be given to function 'func'
// [out] menuItemID - pointer to ID which will be assigned to this
menuitem, this can be 0 (NULL) if caller is not interested in this value
(this ID is used for removing menuitem from menu)
// [in] flags - reserved flags, use 0
typedef int (*MgrAddMpxMenuItemType)(DEVID devID, const char *pluginName,
const char *itemName, const char *menuPath, MenuFuncType func, MENUFUNCPAR
param, ITEMID *menuItemID, u32 flags);

// function type for removing item in Loader menu
// [in] menuItemID - ID of item that will be removed
typedef int (*MgrRemoveMenuItemType)(ITEMID menuItemID);

// [in] devID - device ID of medipix
// [out] menuPath - path to "menu folder" of mpx device
typedef int (*MgrGetMpxMenuPathType)(DEVID devID, char
menuPath[MENU_LENGTH]);

// types of function for browsing of items in tray menu
// to enumerate all menu items MgrGetFirstMenuItem has to be called first,
// return value 0 means that "name" contains path name of first menu item
(e.g. path1|path2|someitem)
// use the itemID in subsequent calls of MgrGetNextMenuItem, call it
repeatedly until return value != 0
// [in/out] itemID - pointer to "browse handle", do not change its value
// [out] name - path name of menu item, used for call in
MgrCallMenuItemByName()
typedef int (*MgrGetFirstMenuItemType)(ITEMID *itemID, char
name[MENU_LENGTH]);
typedef int (*MgrGetNextMenuItemType)(ITEMID *itemID, char
name[MENU_LENGTH]);
```

```
// type of function for calling menu item
// [in] name - path name of menu item
typedef int (*MgrCallMenuItemByNameType)(const char *name);

// type of function for calling menu item
// [in] itemID - ID of menu item
typedef int (*MgrCallMenuItemByIDType)(ITEMID itemID);


// type of function for adding own handler for win messages (pretranslation
of win msgs, support for MFC plugins)
// [in] handler - message handler function
typedef void (*MgrAddMsgHandlerType)(MsgHandlerType handler);

// type of function for removing (unregistering) handler
// [in] handler - message handler function that should be unregistered
typedef void (*MgrRemoveHandlerType)(MsgHandlerType handler);

// function type for registering plugin function in Manager
// [in] funcInfo - this structure has to be filled to give manager all
informations about function
typedef int (*MgrAddFuncItemType)(ExtFunctionInfo *funcInfo);

// function type for unregistering plugin function in Manager
// [in] pluginName - name of plugin containing function that should be
removed
// [in] functionName - name of function that should be unregistered
typedef int (*MgrRemoveFuncItemType)(const char *pluginName, const char
*functionName);

// types of function for browsing of registered functions
// to enumerate all registered functions mgrGetRegFirstFunc has to be
called first,
// return value 0 means that funcInfo contains information about registered
function,
// use the funcID in subsequent calls of mgrGetRegNextFunc, call it
repeatedly until return value != 0
// [in/out] funcID - pointer to "browse handle", do not change its value
// [out] funcInfo - this structure will be filled by manager to provide all
informations about registered function
typedef int (*MgrGetRegFirstFuncType)(ITEMID *funcID, ExtFunctionInfo
*funcInfo);
typedef int (*MgrGetRegNextFuncType)(ITEMID *funcID, ExtFunctionInfo
*funcInfo);

// type of function for getting function info via plugin.functionName
// [in] funcID - id of function
// [out] extFuncInfo - this structure will be filled by manager to provide
all informations about registered function
typedef int (*MgrGetFuncItemIDType)(ITEMID funcID, ExtFunctionInfo
*extFuncInfo);

// type of function for getting function info via plugin.functionName
// [in] pluginName - name of plugin where function resides
// [in] functionName - name of function we are interested in
// [out] extFuncInfo - this structure will be filled by manager to provide
all informations about registered function
typedef int (*MgrGetFuncItemType)(const char *pluginName, const char
*functionName, ExtFunctionInfo *extFuncInfo);

// type of function for calling function info via plugin.functionName
// [in] pluginName - name of plugin where function resides
// [in] functionName - name of function we are interested in
// [in] in - input parameters for specified function
// [out] out - output parameters for specified function
typedef int (*MgrCallFuncItemType)(const char *pluginName, const char
*functionName, byte *in, byte *out);


// function type for registering callback event (by plugin) in Manager
// [in] cbEventInfo - structure describing callback event which will be
registered
// [out] eventID - assigned ID, this value should be used for setting cb
event
typedef int (*MgrAddCBEventType)(ExtCBEventInfo *cbEventInfo, ITEMID
*eventID);

// function type for unregistering callback event (by plugin) in Manager
// [in] pluginName - name of plugin where function resides
// [in] cbEventName - name of callback event that shoud be unregistered
typedef int (*MgrRemoveCBEventType)(const char *pluginName, const char
*cbEventName);

// types of function for browsing of registered callback events
// to enumerate all registered events mgrGetRegFirstCBEvent has to be
called first,
// return value 0 means that funcInfo contains information about registered
function,
// use the cbEventID in subsequent calls of mgrGetRegNextCBEvent, call it
repeatedly until return value != 0
// [in/out] cbEventID - pointer to "browse handle", do not change its value
// [out] extCBEventInfo - this structure will be filled by manager to
provide all informations about registered event
typedef int (*MgrGetRegFirstCBEventType)(ITEMID *cbEventID, ExtCBEventInfo
*extCBEventInfo);
typedef int (*MgrGetRegNextCBEventType)(ITEMID *cbEventID, ExtCBEventInfo
*extCBEventInfo);

// type of function for setting the registered callback event (Manager then
calls registered callback function for this "callback event"
// [in] ID - event ID that should be set
// [in] par - parameter that will be given to callback function registered
for this event
typedef int (*MgrSetCBEventType)(ITEMID ID, CBPARAM par);

// type of function for registering to receive notifications about callback
event
// [in] pluginName - name of plugin which offers required cb event
// [in] cbEventName - name of event for which we register
// [in] callback - pointer to call back that should be called when event
occures
typedef int (*MgrRegisterCallbackType)(const char *pluginName, const char
*cbEventName, CallbackFuncType callback);

// type of function for registering to receive notifications about callback
event
// [in] pluginName - name of plugin which offers the event
// [in] cbEventName - name of event for which should be registration
changed
// [in] callback - callback function which we want remove from "call list"
when event occurres
```

```
typedef int (*MgrUnregisterCallbackType)(const char *pluginName, const char
*cbEventName, CallbackFuncType callback);

// function type for registering custom frame attribute template
// [in] attribInfo - this structure has to be filled to give manager
required informations
typedef int (*MgrAddFrameAttribTemplType)(ExtFrameAttribInfo *attribInfo);

// function type for unregistering custom frame attribute template
// [in] attribName - name of attribName that should be unregistered
typedef int (*MgrRemoveFrameAttribTemplType)(const char *attribName);


// type of function for creating chain of filters
// [in] pluginName - name of plugin which is creating chain of filters
// [in] filterChainName - name of filter chain
// [out] filterChainID - ID of created filter chain
typedef int (*MgrCreateFilterChainType)(const char *pluginName, const char
*filterChainName, ITEMID *filterChainID);

// removes filter chain
// [in] filterChainID - ID of filter chain that should be removed
typedef int (*MgrRemoveFilterChainType)(ITEMID filterChainID);

// type of function for adding filter to chain of filters
// [in] filterChainID - ID of filter chain to which the filter will be
added
// [in] filtInstID - instance ID of filter (obtained via
MgrCreateFilterInstType)
// [in] order - zero based index of filter in chain (0 add to begining,
MAXDWORD - add to end)
typedef int (*MgrAddToFilterChainType)(ITEMID filterChainID, ITEMID
filtInstID, u32 order);

// type of function for removing filter from chain of filters
// [in] filterChainID - ID of filter chain from which the filter will be
removed
// [in] order - order of filter (zero based)
typedef int (*MgrRemoveFromFilterChainType)(ITEMID filterChainID, u32
order);

// type of function for getting frame processed by filter chain
// [in] filterChainID - ID of filter chain which will be used
// [in] inID - ID of input frame
// [out] outID - ID of created output frame
typedef int (*MgrGetFilteredFrameType)(ITEMID filterChainID, FRAMEID inID,
FRAMEID *outID);

// types of function for browsing created filter chains
// to enumerate all filter chains MgrGetFirstFilterChainType has to be
called first,
// return value 0 means that filterChainInfo contains information about
registered function,
// use the filterChainID in subsequent calls of MgrGetNextFilterChainType,
call it repeatedly until return value != 0
// [in/out] filterChainID - pointer to "browse handle", do not change its
value
// [out] filterChainInfo - this structure will be filled by manager to
provide all informations about filter chain, caller must supply
filterChainInfo->filterNames and filterChainInfo->filterIDs buffer and
their size if he wants fill informations about filter names and/or filter
```

```
IDs, filterChainInfo->filterNamesSize and filterChainInfo->filterIDsSize
will contain required size of buffers
typedef int (*MgrGetFirstFilterChainType)(ITEMID *filterChainID,
ExtFilterChainInfo *filterChainInfo);
typedef int (*MgrGetNextFilterChainType)(ITEMID *filterChainID,
ExtFilterChainInfo *filterChainInfo);

// [in] filterChainID - ID of filter chain
// [out] filterChainInfo - this structure will be filled by manager to
provide all informations about filter chain, caller must supply
filterChainInfo->filterNames and filterChainInfo->filterIDs buffer and
their size if he wants fill informations about filter names and/or filter
IDs, filterChainInfo->filterNamesSize and filterChainInfo->filterIDsSize
will contain required size of buffers
typedef int (*MgrGetFilterChainInfoType)(ITEMID filterChainID,
ExtFilterChainInfo *filterChainInfo);

// [in] filterID - ID of filter
// [in] name - name of instance (no need to be unique)
// [out] instID - ID of filter instance
typedef int (*MgrCreateFilterInstType)(ITEMID filterID, const char *name,
ITEMID *instID);

// [in] instID - ID of filter instance
typedef int (*MgrDeleteFilterInstType)(ITEMID instID);

// types of function for browsing created filter instances
// to enumerate all instances of given filter MgrGetFirstFilterInstType has
to be called first,
// return value 0 means that instID contains valid ID of existing filter
instance
// use returned instID in subsequent calls of MgrGetNextFilterInstType,
call it repeatedly until return value != 0
// [in] filterID - ID of existing filter (function info ID)
// [in/out] instID - instance ID of filter instance
// [out] name - name of filter instance (can be NULL)
typedef int (*MgrGetFirstFilterInstType)(ITEMID filterID, ITEMID *instID,
char name[NAME_LENGTH]);
typedef int (*MgrGetNextFilterInstType)(ITEMID filterID, ITEMID *instID,
char name[NAME_LENGTH]);


// [in] instID - instance ID of filter
// [in] idx - zero-based index of parameter
// [in] data - data for parameter
// [in] dataSize - size of data
typedef int (*MgrSetFilterParType)(ITEMID instID, int idx, byte *data, int
dataSize);


// [in] instID - instance ID of filter
// [in] idx - zero-based index of parameter or -1, in case of -1, *dataSize
will contain number of parameters of adjustable filters, (info will be
unused so it can be NULL)
// [in/out] info - data for parameter, info->data can contain allocated
buffer for param. value, if info->data is NULL info structure will be
filled without data
// [in/out] dataSize - dataSize should contain size of allocated buffer in
info->data (in bytes), after return it will contain required size of data
buffer for parameter
```

```
typedef int (*MgrGetFilterParType)(ITEMID instID, int idx, ItemInfo *info,
int *dataSize);

// frame control
//

#define FCREATE_ZERO        0x0100      // created frame will be filled
with zeros

// creates frame
// [out] frameID - ID of new created frame
// [in] width - width of frame
// [in] height - height of frame
// [in] flags - combination of type of frame (TYPE_I16, TYPE_U32,
TYPE_DOUBLE) and FCREATE_XXX flags
typedef int (*MgrCreateFrameType)(FRAMEID *frameID, u32 width, u32 height,
int flags, const char *creatorName);

// load frame from file
// [out] frameID - ID of new loaded frame
// [in] fileName - name of description or data file
typedef int (*MgrLoadFrameType)(FRAMEID *frameID, const char *fileName);

// save frame to file
// [in] frameID - ID of new loaded frame
// [in] fileName - name of data file
// [in] flags - bitwise combination of FSAVE_* flags see common.h for
details
typedef int (*MgrSaveFrameType)(FRAMEID frameID, const char *fileName, u32
flags);

// makes copy of frame
// [in] origID - ID of original frame
// [out] newID - ID of new created copy
typedef int (*MgrDuplicateFrameType)(FRAMEID origID, FRAMEID *newID);

// frame browsing, return val 0 OK, > 0 no more frames, < 0 error
// [out] frameID - if return value is 0, frameID contains ID of first
frame, use this ID for subsequent call of MgrGetNextFrameType
typedef int (*MgrGetFirstFrameType)(FRAMEID *frameID);

// frame browsing, return val 0 OK, > 0 no more frames, < 0 error
// [in/out] frameID - if return value is 0, frameID contains ID of browsed
frame
typedef int (*MgrGetNextFrameType)(FRAMEID *frameID);

// open specified frame (increase reference count, if reference count drops
to zero, frame is automatically deleted)
// [in] frameID - frame to open
// [out] refCount - actual reference count
typedef int (*MgrOpenFrameType)(FRAMEID frameID, u32 *refCount);

// close specified frame (decrease reference count, if reference count
drops to zero, frame is automatically deleted)
// [in] frameID - frame to lock
// [out] refCount - actual reference count
typedef int (*MgrCloseFrameType)(FRAMEID frameID, u32 *refCount);

// locks frame for exclusive access
// [in] frameID - frame to lock
// [in] timeout - how long wait (in ms) for lock in ms (if lock will not be
successfull in timeout milliseconds, function will return
MPXERR_LOCK_TIMEOUT)
typedef int (*MgrLockFrameType)(FRAMEID frameID, u32 timeout);

// unlocks frame (this function has to be called after successful
MgrLockFrameType or MgrGetLockedFrameBuffType)
// [in] frameID - frame to unlock
typedef int (*MgrUnlockFrameType)(FRAMEID frameID);

// retrieves type of selected frame
// [in] frameID - frame identification
// [out] type - type of frame (TYPE_I16, TYPE_U32, TYPE_DOUBLE)
typedef int (*MgrGetFrameTypeType)(FRAMEID frameID, Data_Types *type);

// sets type of selected frame (converts frame to specified format)
// [in] frameID - frame identification
// [in] type - type of frame (TYPE_I16, TYPE_U32, TYPE_DOUBLE)
typedef int (*MgrSetFrameTypeType)(FRAMEID frameID, Data_Types type);

// retrieves information about frame size
// [in] frameID - frame identification
// [out] width - width of frame
// [out] height - height of frame
typedef int (*MgrGetFrameSizeType)(FRAMEID frameID, u32 *width, u32
*height);

// sets frame size
// [in] frameID - frame identification
// [in] width - width of frame
// [in] height - height of frame
typedef int (*MgrSetFrameSizeType)(FRAMEID frameID, u32 width, u32 height);

// copy supplied buffer to frame internal data buffer
// [in] frameID - frame identification
// [in] buffer - buffer to copy
// [in] size - size of buffer (in bytes)
typedef int (*MgrSetFrameDataType)(FRAMEID frameID, byte *buffer, u32
size);

// retrieves data (copy) from frame in specified format
// [in] frameID - frame identification
// [out] buffer - output buffer for data
// [in/out] size - size of supplied buffer (in bytes)
// [in] type - type of output format (TYPE_I16, TYPE_U32, TYPE_DOUBLE)
typedef int (*MgrGetFrameDataType)(FRAMEID frameID, byte *buffer, u32
*size, Data_Types type);

// locks frame and retrieves pointer to internal data buffer
// [in] frameID - frame identification
// [out] buffer - pointer to internal data buffer
// [out] size - size (in bytes) of internal data buffer
// [in] timeout - how long wait (in ms) for lock in ms (if lock will not be
successfull in timeout milliseconds, function will return
MPXERR_LOCK_TIMEOUT)
typedef int (*MgrGetLockedFrameBuffType)(FRAMEID frameID, byte **buffer,
u32 *size, u32 timeout);

// loads data file to existing frame
// [in] frameID - frame identification
// [in] fileName - name of data file
// [in] flags - flags describing data file format (bitwise combination of
following flags: FSAVE_BINARY, FSAVE_ASCII, FSAVE_I16, FSAVE_U32,
FSAVE_DOUBLE, FSAVE_NODESCFILE)
// flags FSAVE_BINARY/FSAVE_ASCII and flags
FSAVE_I16/FSAVE_U32/FSAVE_DOUBLE are mutually exclusive
typedef int (*MgrLoadFrameDataType)(FRAMEID frameID, const char *fileName,
u32 flags);


// sets frame name
// [in] frameID - frame identification
// [in] name - frame name
typedef int (*MgrSetFrameNameType)(FRAMEID frameID, const char *name);

// gets frame name
// [in] frameID - frame identification
// [out] name - frame name
typedef int (*MgrGetFrameNameType)(FRAMEID frameID, char
name[MPX_MAX_PATH]);

// sets creator name (it should be plugin name that created frame)
// [in] frameID - frame identification
// [in] name - creator name
typedef int (*MgrSetFrameCreatorNameType)(FRAMEID frameID, const char
*name);

// gets creator name (it should be plugin name that created frame)
// [in] frameID - frame identification
// [out] name - creator name
typedef int (*MgrGetFrameCreatorNameType)(FRAMEID frameID, char
name[NAME_LENGTH]);

// sets "logical" path of frame ("logical" path defines context of frame)
// [in] frameID - frame identification
// [in] path - logical path (folders are separated by '|', e.g.
"myplugin|filtered|device1"
typedef int (*MgrSetFrameLogPathType)(FRAMEID frameID, const char *path);

// gets "logical" path of frame ("logical" path defines context of frame)
// [in] frameID - frame identification
// [out] path - logical path (folders are separated by '|', e.g.
"myplugin|filtered|device1"
typedef int (*MgrGetFrameLogPathType)(FRAMEID frameID, char
path[MPX_MAX_PATH]);

// adds attribute to frame
// [in] frameID - frame identification
// [in] name - name of attribute
// [in] desc - description of attribute
// [in] type - type of data of attribute
// [in] data - data of attribute ('count' of items of type 'type')
// [in] count - number of items of type 'type' in 'data' buffer
typedef int (*MgrAddFrameAttribType)(FRAMEID frameID, const char *name,
const char *desc, Data_Types type, const byte *data, u32 count);

// sets value of selected attribute of selected frame
// [in] frameID - frame identification
// [in] name - name of attribute to set
// [in] data - data (value) of attribute
// [in] size - size of 'data' buffer in bytes
typedef int (*MgrSetFrameAttribType)(FRAMEID frameID, const char *name,
byte *data, u32 size);

// gets value of selected attribute of selected frame
// [in] frameID - frame identification
// [in] name - name of attribute to get
// [in] data - buffer for attribute value
// [in/out] size - size of 'data' buffer in bytes, if 'data' buffer is not
big enough MPXERR_BUFFER_SMALL is returned and size contains required size
of buffer)
// [out] type - type of attribute (may be 0 if caller do not want this
information)
typedef int (*MgrGetFrameAttribType)(FRAMEID frameID, const char *name,
byte *data, u32 *size, Data_Types *type);

// removes selected attribute from frame
// [in] frameID - frame identification
// [in] name - name of attribute to remove
typedef int (*MgrRemoveAttribType)(FRAMEID frameID, const char *name);

// removes all attributes from frame
// [in] frameID - frame identification
typedef int (*MgrRemoveAllAttribsType)(FRAMEID frameID);

// browsing of frame attributes
// [in] frameID - frame identification
// [out] itemID - "browse handle", use it in subsequent calls of
MgrGetFrameNextAttribType
// [out] attrib - ouput structure containing information about attribute
typedef int (*MgrGetFrameFirstAttribType)(FRAMEID frameID, ITEMID *itemID,
ExtFrameAttrib *attrib);

// browsing of frame attributes
// [in] frameID - frame identification
// [in/out] itemID - "browse handle"
// [out] attrib - ouput structure containing information about attribute
typedef int (*MgrGetFrameNextAttribType)(FRAMEID frameID, ITEMID *itemID,
ExtFrameAttrib *attrib);


#define SRVFRAME_MASKBITS   1       // mask bits
#define SRVFRAME_TESTBITS   2       // test bits
#define SRVFRAME_THLADJ     3       // THL adj. bits
#define SRVFRAME_THHADJ     4       // THH adj. bits (does not exist for
Timepix)
#define SRVFRAME_MODE       5       // mode bits (Timepix)
// gets service frame ID of selected type for particular device
// [in] devID - device ID of medipix
// [in] type - type of service frame (SRVFRAME_xxx)
// [out] frameID - id of service frame (it is constant for certain FRAMEID
during program run)
typedef int (*MgrGetServiceFrameType)(DEVID devID, u32 type, FRAMEID
*frameID);


// MPXCTRL FUNCTION TYPES
//

// types of functions for enumerating found Medipix devices
// to enumerate all devices mpxCtrlGetFirstMpx has to be called first
// return value 0 means that *devID is valid ID of successfuly initialized
Medipix
// use the devID in subsequent calls of mpxCtrlGetNextMpx, call it
repeatedly until return value != 0
// [in/out] devID - this device ID uniquely identifies particular Medipix
in MpxCtrl functions
// [out] count - after call will contain number of found devices; can be
NULL
typedef int (*MpxCtrlGetFirstMpxType)(DEVID *devID, int *count);
typedef int (*MpxCtrlGetNextMpxType)(DEVID *devID);

// tries to find new devices (it will notify via MPXCTRL_CB_NEWMPX
callback)
typedef int (*MpxCtrlFindNewMpxsType)();

// it will "reconnect" (reinitialize) the interface and connected medipix
(for hot swap of medipix on interface)
// [in] devID - medipix device identification
typedef int (*MpxCtrlReconnectMpxType)(DEVID devID);

// type of function for getting number of "hardware info items" for
particular Medipix
// "hardware info items" are used to access interface specific variables to
obtain or set interface specifice information (e.g. muros clock frequency)
// [in] devID - medipix device identification
// [out] count - number of hw info items of this device
typedef int (*MpxCtrlGetHwInfoCountType)(DEVID devID, int *count);

// obtains selected HwInfoItem structure
// [in] devID - medipix device identification
// [in] index - zero-based index of "hardware info item", should be in <0,
count-1> (count is obtained via mpxCtrlGetHwInfoCount)
// [out] infoItem - structure that will contain informations about selected
item
// [in/out] dataSize - size of infoItem->data for "raw item data", if the
dataSize is not big enough for item data,
// the function returns error (!=0) and *dataSize will contain required
size (to allow caller allocate big enough data buffer)
typedef int (*MpxCtrlGetHwInfoItemType)(DEVID devID, int index, HwInfoItem
*infoItem, int *dataSize);

// set selected "hardware info item" data
// [in] devID - medipix device identification
// [in] index - zero-based index of "hardware info item", should be in <0,
count-1> (count is obtained via mpxCtrlGetHwInfoCount)
// [in] data - pointer to data
// [in] dataSize - size of data (to allow sizecheck of data for certain hw
info item)
typedef int (*MpxCtrlSetHwInfoItemType)(DEVID devID, int index, byte *data,
int dataSize);

// sets custom name for mpx device
// [in] devID - medipix device identification
// [in] name - custom name for mpx device
typedef int (*MpxCtrlSetCustomNameType)(DEVID devID, const char *name);

// gets custom name for mpx device
// [in] devID - medipix device identification
// [out] name - custom name for mpx device
typedef int (*MpxCtrlGetCustomNameType)(DEVID devID, char
name[NAME_LENGTH]);
```

```
// try to locks medipix device for exclusive access
// [in] devID - medipix device identification
// [out] success - TRUE if locking was successful
// [in] timeout - how long wait (in ms) for lock in ms (if lock will not be
successfull in timeout milliseconds, function will return
MPXERR_LOCK_TIMEOUT)
typedef int (*MpxCtrlTryLockDeviceType)(DEVID devID, BOOL *success, u32
timeout);

// release locked device, MUST be called after each successful call to
MpxCtrlTryLockDevice
// [in] devID - medipix device identification
typedef int (*MpxCtrlReleaseDeviceType)(DEVID devID);

// [in] devID - medipix device identification
// [out] id - id of thread that currently owns medipix device lock
typedef int (*MpxCtrlGetLockOwnerIDType)(DEVID devID, DWORD *id);

// tries to revive selected device (e.g. after short disconnection)
// difference between MpxCtrlReviveMpxDeviceType and
MpxCtrlInitMpxDeviceType is that init uses the same config as during start
up ("safe values") while revive preserves current config (dacs, pix cfg,
etc.)
// [in] devID - medipix device identification
typedef int (*MpxCtrlReviveMpxDeviceType)(DEVID devID);

// initialize particular Medipix device with values from medipix
configuration file
// [in] devID - medipix device identification
// [in] fileName - filepathname of medipix configuration file, can be NULL,
if it is NULL default fileName (based on chipboard ID is used)
typedef int (*MpxCtrlInitMpxDeviceType)(DEVID devID, const char *fileName);

// saves current configuration of particular Medipix device to selected
medipix configuration file
// [in] devID - medipix device identification
// [in] fileName - filepathname of medipix configuration file, can be NULL,
if it is NULL default fileName (based on chipboard ID is used)
typedef int (*MpxCtrlSaveMpxCfgType)(DEVID devID, const char *fileName);

// loads configuration from selected medipix configuration file to internal
structures of particular Medipix device
// [in] devID - medipix device identification
// [in] fileName - filepathname of medipix configuration file
typedef int (*MpxCtrlLoadMpxCfgType)(DEVID devID, const char *fileName);

// saves current configuration as default one (for new devices which do not
have their own configurations)
// [in] devID - medipix device identification
typedef int (*MpxCtrlSaveMpxCfgAsDefaultType)(DEVID devID);


// SET/GET ACQUISITION SETTING
//

// sets polarity of selected Medipix device
// [in] devID - medipix device identification
// [in] positive - TRUE for positive polarity, FALSE for negative
typedef int (*MpxCtrlSetPolarityType)(DEVID devID, BOOL positive);
```

```
// gets polarity setting for selected medipix device
// [in] devID - medipix device identification
// [out] positive - TRUE for positive polarity, FALSE for negative
typedef int (*MpxCtrlGetPolarityType)(DEVID devID, BOOL *positive);

// sets charge sharing test for selected Medipix device
// [in] devID - medipix device identification
// [in] enable - TRUE to enable, FALSE to disable
typedef int (*MpxCtrlSetCSTType)(DEVID devID, BOOL enable);

// gets charge sharing test setting for selected medipix device
// [in] devID - medipix device identification
// [out] enabled - TRUE if enabled, FALSE if disabled
typedef int (*MpxCtrlGetCSTType)(DEVID devID, BOOL *enabled);

// sets acquisition mode for selected Medipix device
// [in] devID - medipix device identification
// [in] mode - acq. mode to set (check ACQMODE_MANUAL, ACQMODE_xxxx)
typedef int (*MpxCtrlSetAcqModeType)(DEVID devID, int mode);

// gets currently selected acquisition mode for selected Medipix device
// [in] devID - medipix device identification
// [out] mode - current acq mode (check ACQMODE_MANUAL, ACQMODE_xxxx)
typedef int (*MpxCtrlGetAcqModeType)(DEVID devID, int *mode);

#define MPXCTRL_HWTIMER_ENABLE      0       // HW timer enabled
#define MPXCTRL_HWTIMER_DISABLE     1       // HW timer disabled (PC timer
used)
#define MPXCTRL_HWTIMER_AUTO        2       // MpxCtrl library decides
which timer will be used according to time of measurement and capabilities
of device
// selects which timer is used for selected Medipix device
// [in] devID - medipix device identification
// [in] state - MPXCTRL_HWTIMER_ENABLE, MPXCTRL_HWTIMER_DISABLE,
MPXCTRL_HWTIMER_AUTO
typedef int (*MpxCtrlSetHwTimerType)(DEVID devID, int state);

// gets HW timer setting
// [in] devID - medipix device identification
// [out] state - MPXCTRL_HWTIMER_ENABLE, MPXCTRL_HWTIMER_DISABLE,
MPXCTRL_HWTIMER_AUTO (see definitions)
typedef int (*MpxCtrlGetHwTimerType)(DEVID devID, int *state);

// sets all DACs of all chips to default (typical for usage) values,
depends on currently set polarity
// [in] devID - medipix device identification
typedef int (*MpxCtrlSetDACsDefaultType)(DEVID devID);


#define REFDAC_THL      0x01
#define REFDAC_THH      0x02
// sets DACs of particular chip or all chips of certain device
// [in] devID - medipix device identification
// [in] dacVals - array of DAC values
// [in] size - size of array dacVals (should be number of DACs, 13
original, 14 MXR or numberOfDacs*number of chips if chipNumber == ALLCHIPS)
// [in] chipNumber - zero-based index of chip or ALLCHIPS (all chips values
are set)
// [in] refChip - zero-based index of chip which will be used for reference
value for refDacs (chipNumber has to be ALLCHIPS - dacs values for all chip
are supplied), to disable this feature set refChip to invalid chipNumber or
refDacs to 0
// [in] refDacs - bitwise ORed REFDAC_xxx flags, if nonzero and device is
multichip (quad) THL/THH DACs values for chips are computed from values of
refChip (the dependency parameters are set by MpxCtrlSetDACRefCoefType)
typedef int (*MpxCtrlSetDACsType)(DEVID devID, DACTYPE *dacVals, int size,
int chipNumber, int refChip, u32 refDacs);

// gets DAC values for one chip of selected medipix device
// [in] devID - medipix device identification
// [out] dacVals - output array for DAC values
// [in] size - size of array dacVals (should be number of DACs, 13
original, 14 MXR or numberOfDacs*number of chips if chipNumber == ALLCHIPS)
// [in] chipNumber - zero-based index of chip or ALLCHIPS (all chips values
are gset)
typedef int (*MpxCtrlGetDACsType)(DEVID devID, DACTYPE *dacVals, int size,
int chipNumber);

// gets single DAC value of single chip or all chips of selected medipix
device
// [in] devID - medipix device identification
// [out] dacVals - output for DAC analog values, size of array should be 1
for single value or numberOfChips to read from all chips (if its 2 or
2*numberOfChips, the second part will contain std. deviation of reading (if
senseCount > 1))
// [in] size - size of dacVals array (for size check)
// [in] dacNumber - number from DACS_ORDER specifying which DAC analog
value should be read
// [in] chipNumber - zero-based index of chip or ALLCHIPS if dac value
should be read from all chips
// [in] senseCount - number of sensing (to receive mean of senseCount
values)
typedef int (*MpxCtrlGetSingleDACAnalogType)(DEVID devID, double *dacVals,
int size, int dacNumber, int chipNumber, int senseCount);

// gets all DACs values from specified chip or from all chips  of selected
medipix device
// [in] devID - medipix device identification
// [out] dacVals - output array for DAC values, size has to be
double[number of dacs] for single chip or le[numberOfChips*number of dacs]
for all chips
// [in] size - size of dacVals array (for size check)
// [in] chipNumber - zero-based index of chip or ALLCHIPS if all the dacs
values should be read from all chips
typedef int (*MpxCtrlGetDACsAnalogType)(DEVID devID, double *dacVals, int
size, int chipNumber);

// sends last set DAC values to all chips of selected medipix device
// [in] devID - medipix device identification
typedef int (*MpxCtrlRefreshDACsType)(DEVID devID);

// sends last set DAC values to all chips of selected medipix device
// [in] devID - medipix device identification
// [out] names - *names is const array of names (const char *) of DACs of
medipix chip
// [out] precisions - *precisions is const array of precisions of
individual DACs in bits
// [out] size - *size of *names array, (e.g. 13 for normal mpx chip, 12 for
MXR)
typedef int (*MpxCtrlGetDACsNamesType)(DEVID devID, const char * const
**names, const int **precisions, int *size);
```

```
// sets dependence of THL/THH for multichip device
// [in] devID - medipix device identification
// [in] coef - array of coefficients for computing THL/THH of chip1,
chip2,... from THL/THH of chip0, array should have the size
[(numberOfChips-1)*2], for each chip chip1, chip2,... there are two coef.
for linear dependency on values of chip0 (i.e. coef[0]*thlOfChip0+coef[1]
for chip1, coef[2]*thlOfChip0+coef[3] for chip2,...)
// [in] size - size of coef array (for size check)
// [in] refDac - REFDAC_THL or REFDAC_THH (THL or THH dependency parameters
are set)
typedef int (*MpxCtrlSetDACRefCoefType)(DEVID devID, double *coef, int
size, int refDac);

// gets currently set dependence parameters for THL/THH (for multichip
device)
// [in] devID - medipix device identification
// [out] coef - array of coefficients
// [in] size - size of coef array (for size check)
// [in] refDac - REFDAC_THL or REFDAC_THH (THL or THH dependency parameters
are get)
typedef int (*MpxCtrlGetDACRefCoefType)(DEVID devID, double *coef, int
size, int refDac);

// sets external DAC value
// [in] devID - medipix device identification
// [in] dacNumber - medipix DAC index that should be replaced by external
DAC, -1 means that ext DAC is not used
// [in] value - value in V, the valid range and step can be obtained from
DevInfo structure
typedef int (*MpxCtrlSetExtDACType)(DEVID devID, int dacNumber, double
value);

// gets external DAC value
// [in] devID - medipix device identification
// [out] dacNumber - medipix DAC index that is currently overloaded by
external DAC, -1 means ext DAC is not used
// [out] value - value in V
typedef int (*MpxCtrlGetExtDACType)(DEVID devID, int *dacNumber, double
*value);

// sets pixels configurations of one selected chip or all chips of selected
medipix device
// [in] devID - medipix device identification
// [in] pixCfgs - array of pixels configurations, PixelCfg for MPX_ORIG and
MPX_MXR and TpxPixCfg for MPX_TPX, size should be MATRIX_SIZE for one
selected chip or numberOfChips*MATRIX_SIZE for all chips mode
// [in] size - size of pixCfgs array (for size check)
// [in] chipNumber - zero-based number of selected chip or ALLCHIPS for
"all chips mode"
typedef int (*MpxCtrlSetPixelsCfgType)(DEVID devID, byte *pixCfgs, int
size, int chipNumber);

// gets pixels configurations of one selected chip or all chips of selected
medipix device
// [in] devID - medipix device identification
// [out] pixCfgs - array for pixels configurations, PixelCfg for MPX_ORIG
and MPX_MXR and TpxPixCfg for MPX_TPX, size should be MATRIX_SIZE for one
selected chip or numberOfChips*MATRIX_SIZE for all chips mode
// [in] size - size of pixCfgs array (for size check)
// [in] chipNumber - zero-based number of selected chip or ALLCHIPS for
"all chips mode"
typedef int (*MpxCtrlGetPixelsCfgType)(DEVID devID, byte *pixCfgs, int
size, int chipNumber);

// resets pixels configurations (nothing masked, no test bits, adj bits 7)
of one selected chip or all chips of selected medipix device
// [in] devID - medipix device identification
// [in] chipNumber - zero-based number of selected chip or ALLCHIPS  for
"all chips mode"
typedef int (*MpxCtrlResetPixelsCfgType)(DEVID devID, int chipNumber);

// sets pixels configurations of all chips of selected medipix device from
one "super matrix" (e.g. for quad 512x512)
// [in] devID - medipix device identification
// [in] pixCfgs - buffer of pix. cfgs
// [in] size - size of buffer
typedef int (*MpxCtrlSetSuperMatrixPixCfgType)(DEVID devID, byte *pixCfgs,
int size);

// gets pixels configurations of all chips of selected medipix device in
one "super matrix"
// [in] devID - medipix device identification
// [out] pixCfgs - output buffer for pix. cfgs
// [in] size - size of buffer for size-check
typedef int (*MpxCtrlGetSuperMatrixPixCfgType)(DEVID devID, byte *pixCfgs,
int size);

// sends last set pixels cfgs to all chips of selected medipix device
// [in] devID - medipix device identification
typedef int (*MpxCtrlRefreshPixelsCfgType)(DEVID devID);

// sets acquisition spacing for making acquisition in several steps, in
each steps part of pixels are active
// the distance is given by spacing parameter (min 1, max 256)
// [in] devID - medipix device identification
// [in] spacing - spacing values
typedef int (*MpxCtrlSetAcqSpacingType)(DEVID devID, int spacing);

// gets acquisition spacing
// [in] devID - medipix device identification
// [out] spacing - *spacing contains currently set spacing
typedef int (*MpxCtrlGetAcqSpacingType)(DEVID devID, int *spacing);

// sets auto conversion of frame to "super matrix" for quad, enabled by
default, useful to disable this for per chip operation
// [in] devID - medipix device identification
// [in] enable - if TRUE each frame is automatically converted to "super
matrix" (512*512 matrix for quad) at the end (no effect for single)
typedef int (*MpxCtrlSetAutoConvToSMType)(DEVID devID, BOOL enable);

// gets settings of auto conversion of frame to "super matrix" for quad
// [in] devID - medipix device identification
// [out] enable - if TRUE each frame is automatically converted to "super
matrix" (512*512 matrix for quad) at the end (no effect for single)
typedef int (*MpxCtrlGetAutoConvToSMType)(DEVID devID, BOOL *enable);

// function for conversion of "chip by chip" frame to "super matrix" frame
(for quad)
// [in] devID - medipix device identification
// [in] chipByChip - input buffer containing frame where data are in chip
by chip order
// [out] superMatrix - output buffer for super matrix frame (frame contains
proper image)
// [in] buffSize - size of buffer in bytes (for size check)
// [in] buffType - buffer types (allowed values - TYPE_I16, TYPE_I32,
TYPE_DOUBLE)
typedef int (*MpxCtrlConvToSuperMatrixType)(DEVID devID, const byte
*chipByChip, byte *superMatrix, u32 buffSize, Data_Types buffType);

// function for conversion of "super matrix" frame to "chip by chip" frame
(for quad)
// [in] devID - medipix device identification
// [in] superMatrix - input buffer with super matrix frame (frame contains
proper image)
// [out] chipByChip - output buffer containing frame in chip by chip order
// [in] buffSize - size of buffer in bytes (for size check)
// [in] buffType - buffer types (allowed values - TYPE_I16, TYPE_I32,
TYPE_DOUBLE)
typedef int (*MpxCtrlConvFromSuperMatrixType)(DEVID devID, const byte
*superMatrix, byte *chipByChip, u32 buffSize, Data_Types buffType);


// loads pixels configurations (mask bit, test bit, and 3-bit threshold
adjustements) from "binary pixels configuration" file
// [in] devID - medipix device identification
// [in] fileName - filenamepath of binary file, if it is NULL default
filename based on chipboard ID is used
// [in] loadDacs - if TRUE, DACs are loaded from "$fileName.dacs"
typedef int (*MpxCtrlLoadPixelsCfgType)(DEVID devID, const char *fileName,
BOOL loadDacs);

// saves pixels configurations (mask bit, test bit, and 3-bit threshold
adjustements) to "binary pixels configuration" file
// [in] devID - medipix device identification
// [in] fileName - filenamepath of binary file
// [in] saveDacs - if TRUE, current DACs are saved to file "$fileName.dacs"
typedef int (*MpxCtrlSavePixelsCfgType)(DEVID devID, const char *fileName,
BOOL saveDacs);

// loads pixels configurations (mask bit, test bit, and 3-bit threshold
adjustements) from several ASCII files
// [in] devID - medipix device identification
// [in] maskBitFile - filenamepath of ASCII file containing mask bit for
each pixel (0 masked)
// [in] testBitFile - filenamepath of ASCII file containing test bit for
each pixel (0 test bit active)
// [in] thlFile - filenamepath of ASCII file containing values
corresponding to THL adjust bits for each pixel
// [in] thhOrModeFile - filenamepath of ASCII file containing values
corresponding to THH adj. bits for each pixel (for Medipix) or mode mask
(for Timepix)
// [in] loadDacs - if TRUE, DACs are loaded from "$fileName.dacs"
typedef int (*MpxCtrlLoadPixelsCfgAsciiType)(DEVID devID, const char
*maskBitFile, const char *testBitFile, const char *thlFile, const char
*thhOrModeFile, BOOL loadDacs);

// saves pixels configurations (mask bit, test bit, and 3-bit threshold
adjustements) to several ASCII files
// [in] devID - medipix device identification
// [in] maskBitFile - name of file to which mask bit for each pixel (0
masked) will be written
// [in] testBitFile - name of file to which test bit for each pixel (0 test
bit active) will be written
// [in] thlFile - name of file to which values corresponding to THL adjust
bits for each pixel will be written
// [in] thhOrModeFile - name of file to which values corresponding to THH
adj. bits (Medipix) or mode mask (Timepix) for each pixel will be written
// [in] saveDacs - if TRUE, current DACs are saved to file "$fileName.dacs"
typedef int (*MpxCtrlSavePixelsCfgAsciiType)(DEVID devID, const char
*maskBitFile, const char *testBitFile, const char *thlFile, const char
*thhOrModeFile, BOOL saveDacs);


// ACQUISITION CONTROL
//

// perform acquisition in frame mode
// [in] devID - medipix device identification
// [in] numberOfFrames - acquisition count
// [in] timeOfEachAcq - time of each acquistion in seconds
// [in] fileFlags - combinations of format flags (use bitwise or "|") for
saving (FSAVE_BINARY, FSAVE_ASCII, FSAVE_I16, FSAVE_U32, FSAVE_DOUBLE,
FSAVE_NODESCFILE), if fileFlags=0 frames are held in memory
// [in] fileName - basename of filename (if fileFlags is set properly),
resulting file name is created by appending acquisition number to basename
// flags FSAVE_BINARY/FSAVE_ASCII and flags
FSAVE_I16/FSAVE_U32/FSAVE_DOUBLE are mutually exclusive
typedef int (*MpxCtrlPerformFrameAcqType)(DEVID devID, int numberOfFrames,
double timeOfEachAcq, u32 fileFlags, const char *fileName);


// perform acquisition in integral mode
// [in] devID - medipix device identification
// [in] numberOfAcq - acquisition count
// [in] timeOfEachAcq - time of each acquistion in seconds
// [in] fileFlags - combinations of format flags (use bitwise or "|") for
saving (FSAVE_BINARY, FSAVE_ASCII, FSAVE_I16, FSAVE_U32, FSAVE_DOUBLE,
FSAVE_NODESCFILE)
// [in] fileName - filename (if fileFlags is set properly) to which
integral frame will be automatically saved
typedef int (*MpxCtrlPerformIntegralAcqType)(DEVID devID, int numberOfAcq,
double timeOfEachAcq, u32 fileFlags, const char *fileName);

// perform test pulse acquisition
// [in] devID - medipix device identification
// [in] pulseHeight - height of pulses [V]
// [in] period - period of pulses [s] (distance between pulses is same as
length i.e. period/2)
// [in] pulseCount - number of pulses
// [in] manual - controls manual/automatic handling of spacing/ctpr, 0 =
automatic, nonzero manual (for MXR for manual control this variable has to
contain column test pulse register)
typedef int (*MpxCtrlPerformTestPulseAcqType)(DEVID devID, double
pulseHeight, double period, u32 pulseCount, u32 manual);

// perform full digital test (write random matrix, read it back and counts
number of pixels where values agree)
// [in] devID - medipix device identification
// [out] goodPixels - number fo good pixels (custom, may be NULL)
// [out] frameID - if frameID is specified (custom, may be NULL), frame is
created and its ID is stored in *frameID, caller is then responsible for
closing frame
// [in] delay - delay between writing and reading matrix (in s)
typedef int (*MpxCtrlPerformDigitalTestType)(DEVID devID, u32 *goodPixels,
FRAMEID *frameID, BOOL show, double delay);

// abort operation in progress
// [in] devID - medipix device identification
typedef int (*MpxCtrlAbortOperationType)(DEVID devID);

// SW trigger for acq. start/stop; can be also used to prematurely
terminate acq. in other modes than SW triggered
// [in] devID - medipix device identification
// [in] trigger - trigger type (TRIGGER_ACQSTART/TRIGGER_ACQSTOP)
typedef int (*MpxCtrlTriggerType)(DEVID devID, int trigger);


// BUFFER CONTROL
//

// close opened frames
// [in] devID - medipix device identification
typedef int (*MpxCtrlCloseFramesType)(DEVID devID);

// copy frame from last acquisition to supplied i16[] buffer, this
functionality is availaible for all acquisition modes (even for acquisition
to files)
// this function can be used for asynchronous access to buffer (e.g. during
acquisition, so buffer for error message should be provided
// [in] devID - medipix device identification
// [out] buffer - supplied buffer for frame
// [in] size - size of buffer (for size-check)
// [in] frameNumber - zero-based index of frame
typedef int (*MpxCtrlGetFrame16Type)(DEVID devID, i16 *buffer, u32 size,
u32 frameNumber);

// copy frame from last acquisition to supplied u32[] buffer, this
functionality is availaible for all acquisition modes (even for acquisition
to files)
// this function can be used for asynchronous access to buffer (e.g. during
acquisition, so buffer for error message should be provided
// [in] devID - medipix device identification
// [out] buffer - supplied buffer for frame
// [in] size - size of buffer (for size-check)
// [in] frameNumber - zero-based index of frame
typedef int (*MpxCtrlGetFrame32Type)(DEVID devID, u32 *buffer, u32 size,
u32 frameNumber);

// copy frame from last acquisition to supplied double[] buffer, this
functionality is availaible for all acquisition modes (even for acquisition
to files)
// this function can be used for asynchronous access to buffer (e.g. during
acquisition, so buffer for error message should be provided
// [in] devID - medipix device identification
// [out] buffer - supplied buffer for frame
// [in] size - size of buffer (for size-check)
// [in] frameNumber - zero-based index of frame
typedef int (*MpxCtrlGetFrameDoubleType)(DEVID devID, double *buffer, u32
size, u32 frameNumber);
```

```
// saves selected frame to file in various formats
// [in] devID - medipix device identification
// [in] fileName - filepathname of file, if output is set to multiple files
(see flags parameter) this filename is a base name (final filename contains
index suffix)
// [in] frameNumber - zero-based frame index, if frameNumber is ALLFRAMES
all frames are saved
// [in] flags - bitwise combination of following flags: FSAVE_BINARY,
FSAVE_ASCII, FSAVE_I16, FSAVE_U32, FSAVE_DOUBLE, FSAVE_NODESCFILE
// flags FSAVE_BINARY/FSAVE_ASCII and flags
FSAVE_I16/FSAVE_U32/FSAVE_DOUBLE are mutually exclusive
typedef int (*MpxCtrlSaveFrameType)(DEVID devID, const char *fileName, int
frameNumber, u32 flags);

// retrieve value of attribute from selected frame
// [in] devID - medipix device identification
// [in] frameNumber - zero-based index of frame from last series of
acquisition
// [in] name - name of attribute
// [out] value - output buffer for attribute
// [in/out] size - size of output buffer, if buffer is too small for data
of selected attribute MPXERR_BUFFER_SMALL is returned and "size" contains
required buffer size
// [out] type - type of attribute (may be 0 if caller do not want this
information)
typedef int (*MpxCtrlGetFrameAttribType)(DEVID devID, int frameNumber,
const char *name, byte *value, u32 *size, Data_Types *type);

// converts frameNumber to frameID
// if frame acquisition was performed to file and therefore there is not
corresponding frame registered in manager the new frame is created and
registered
// [in] devID - medipix device identification
// [in] frameNumber - zero-based index of frame from last series of
acquisition
// [out] frameID - id of corresponding frame
typedef int (*MpxCtrlGetFrameIDType)(DEVID devID, int frameNumber, FRAMEID
*frameID);

// INFO FUNCTIONS
//

// gets info about device (interface+medipix)
// [in] devID - medipix device identification
// [out] devInfo - info structure describing capabilities of
interface+medipix
typedef int (*MpxCtrlGetDevInfoType)(DEVID devID, DevInfo *devInfo);

// gives basic information about selected medipix device
// [in] devID - medipix device identification
// [out] numberOfChips - number of chips
// [out] numberOfRows - number of rows to which is chips are placed in
// [out] chipBoardID - chipboard ID
// [out] ifaceName - name of hardware interface (muros, usb,... ) which is
used
typedef int (*MpxCtrlGetMedipixInfoType)(DEVID devID, int *numberOfChips,
int *numberOfRows, char chipBoardID[MPX_MAX_CHBID], char
ifaceName[MPX_MAX_IFACENAME]);
```

```
// provides information about acquisition (in progress or completed),
(acqNumber, ...) pointers can be NULL
// [in] devID - medipix device identification
// [out] acqNumber - zero-based acqusition number
// [out] acqTotalCount - total number of acquisition which was ordered
// [out] acqType - type of acqusition (possible output values -
MPXCTRL_ACQTYPE_FRAME, MPXCTRL_ACQTYPE_FRAME_FILE,
MPXCTRL_ACQTYPE_INTEGRAL, MPXCTRL_ACQTYPE_INTEGRAL_FILE)
// [out] frameFilled - number of frames that are available from last serie
(in integral mode - number of integrated frames)
typedef int (*MpxCtrlGetAcqInfoType)(DEVID devID, int *acqNumber, int
*acqTotalCount, int *acqType, u32 *frameFilled);
#define MPXCTRL_ACQTYPE_FRAME              0       // frames to memory
#define MPXCTRL_ACQTYPE_FRAME_FILE         1       // frames to files
#define MPXCTRL_ACQTYPE_INTEGRAL           2       // frames integrated to
single frame in memory
#define MPXCTRL_ACQTYPE_INTEGRAL_FILE      3       // frames integrated to
single file

// gets info/error message for selected medipix device
// MpxCtrl library registeres several callback events (see MPXCTRL_CB_XXX)
- one of this is MPXCTRL_CB_INFOMSG
// when important/error condition occurred this event is set and plugins
registered for this event can call in callback function mpxCtrlGetInfoMsg
function to obtain this info/error message
// [in] devID - medipix device identification
// [out] msgType - info/error code
// [out] msg - pointer to info/error message
typedef int (*MpxCtrlGetInfoMsgType)(DEVID devID, int *msgType, const char
**msg);
```