Aptina Imaging Device Library

Revision: 4.1.0

Date: June 21, 2011

TABLE OF CONTENTS

1.0 OVERVIEW

The Aptina (formerly Micron) Imaging Device Library (midlib) provides a device independent API for seamless device control and transport. A transport is a protocol for moving data off and on an imaging sensor. The currently supported transports are USB2.0 bulk transport, video file and bitmap file. Additional transports will be added as needed. Users can add a new transport by creating a DLL that implements the camera structure (mi_camera_t) functions. See the MIDLib Transport DLL Development Guide document.

The library supports any number of imaging sensors through use of sensor data files. Known sensor files are included with the library and users may add additional sensor data files as needed. An application begins using the library by calling the routine mi_OpenCameras(). This function goes through all supported transport types and attempts to initialize all cameras. It returns an array of pointers to camera structures (mi_camera_t) for all successfully initialized cameras. This camera structure holds all of the pertinent information about a camera, including a transport specific context, a pointer to a sensor, and function pointers to routines used to interact with the transport mechanism. The most common device specific transport routines are:

startTransport	- Starts a transport stream
stopTransport	- Stops a transport stream
grabFrame	- Returns an image frame
updateFrameSize	- Allows the user to update the image sensor data size after initialization

Given the array of cameras, the application may then use any of the these cameras by calling the camera's startTransport() routine. Once the transport has been started, the application can grab image frames and read and write sensor registers. When a camera is no longer needed, its transport may be stopped using the stopTransport() routine. When the application finishes with all cameras, it must call mi_CloseCameras(). An application can start and stop the camera's transport as many times as needed.

The library will attempt to locate the sensor data file that corresponds to the attached sensor in the directory passed in by the user. Alternately, a specific sensor data file may be provided by the user. mi_updateFrameSize() should be called when the image size or format has changed due to changes to the sensor registers. If a sensor data file does not exist the application may call mi_OpenCameras() with a null string for the sensor data file. The user must call mi_initTransport() and mi_updateFrameSize() if they want to grab image data in this mode, please see initTransport() for more information about this functionality.

2.0 QUICK START

For those of you who just want to jump in, here are the steps required to get you running. If you need more details, refer to the example code in Section 5.0.

```
#include “C:\Aptina Imaging\include\midlib2.h”

mi_camera_t         *cameras[MI_MAX_CAMERAS]; //Array of cameras found
mi_s32              nNumCameras;        //Actual number of cameras found
mi_s32              retValue;           //Function error return value
mi_u8               *pCameraBuff;       //Buffer to store image data

//Open a camera, providing either a specific sensor data file or a directory
retValue = mi_OpenCameras(cameras,&nNumCameras,mi_SensorData());
if (nNumCameras == 0) exit();

//Allocate the buffer to store the data, you must use the size that is
//computed for the particular camera, it may be larger than width*height
pCameraBuff = (mi_u8*)malloc(cameras[0]->sensor->bufferSize);

//”Turn on” the camera
mi_startTransport(cameras[0]);

//Grab a frame of data, buffer size is provided for error checking only
retValue = mi_grabFrame(cameras[0], pCameraBuff,
        cameras[0]->sensor->bufferSize);

//Verify that we got a “good” frame before displaying data
while (retValue != MI_CAMERA_SUCCESS)
{
   retValue = mi_grabFrame(cameras[0], pCameraBuff,
           cameras[0]->sensor->bufferSize);
}
//[Insert display code here]

//”Turn off” the camera
mi_stopTransport(cameras[0]);
mi_CloseCameras();
```

add midlib2.lib to your project, for linking

That’s it you’re ready to go!

3.0 APTINA IMAGING DEVICE LIBRARY API

3.1 Camera Functions

The following API functions are used by the application to open and close cameras.

3.1.1 mi_OpenCameras()

Definition:
mi_s32 mi_OpenCameras(mi_camera_t *pCameras[MI_MAX_CAMERAS], mi_s32 *nNumCameras, const char *sensor_dir_file);

Summary:
This routine goes through all supported transport types and attempts to initialize each camera. It returns the number of successfully opened cameras and a pointer to each. The user may specify the full path of a particular sensor data file (.sdat or .xsdat) or image file or video file, or a directory name to search for a valid sensor data file. For this purpose, the function mi_SensorData() returns the path to the sensor data files installation directory. If the call is successful, the user must then start the transport they wish to connect to by calling mi_startTransport() on the desired camera. This is the first device library routine the application must call.

If you specify a directory to search, mi_OpenCameras() will try all .sdat and .xsdat files in the directory until it finds a match with the attached sensor. The operation may take some time, so you may want provide a Cancel button on your user interface. Call mi_CancelProbe() from another thread to cancel the device probe. If so cancelled, mi_OpenCameras() will return MI_CAMERA_ERROR.

If you specify a particular sensor data file, the device probe operation will be skipped, and the SHiP base address will be set to the first address listed in the sensor data file. If the sensor may be on a secondary base address then you may call mi_DetectSensorBaseAddr() to automatically select the correct base address.

If there is no sensor data file available, either because the hardware doesn't contain a sensor or because no sensor data file has been defined for this sensor; the user may pass a NULL string "" for the sensor_dir_file. No mi_sensor_t structure (mi_camera_t::sensor) will be created at this time. To create one the user must call mi_updateFrameSize. The user must have a mi_sensor_t structure in order to use mi_grabFrame().

The sensor data file may be an image file or video file. The library will create mi_camera_t and mi_sensor_t structures that can be used in any midlib API call. mi_grabFrame() will return the image data from the file. In this case no physical camera need be present.

Parameters:

mi_camera_t	*pCameras[MI_MAX_CAMERAS]	- list of cameras found
mi_s32	*nNumCameras	- number of cameras found

const char * sensor_dir_file - full directory path or filename of sensor data file or “” if there is no sensor data file

Returns:
Error code if no cameras are found otherwise MI_CAMERA_SUCCESS

See also:
mi_initTransport()
mi_updateFrameSize()

3.1.2 mi_OpenCameras2()

Definition:
mi_s32 mi_OpenCameras2(mi_camera_t *pCamera[MI_MAX_CAMERAS], mi_s32 *nNumCameras, const char *sensor_dir_file, mi_u32 transportType, const char *dllName);

Summary:
This function is an extended version of mi_OpenCameras() that lets you specify which transport types to use and optionally the filename of the DLL or directory name to search for a DLL-defined transport. See the MIDLib Transport DLL Development Guide document (MIDLib Transport DLL Development Guide.pdf) for details on how to write a transport DLL.

Parameters:
mi_camera_t *pCameras[MI_MAX_CAMERAS] - list of cameras found
mi_s32 *nNumCameras - number of cameras found
const char * sensor_dir_file - full directory path or filename of sensor data file or “” if there is no sensor data file
mi_u32 transportType - Logical OR of transport types defined in midlib2.h
const char * dllName - Optional filename for DLL or directory to search for DLLs.

Returns:
Error code if no cameras are found otherwise MI_CAMERA_SUCCESS

3.1.3 mi_CloseCameras()

Definition:
void mi_CloseCameras();

Summary:
This routine closes and frees all opened cameras. It must be called before exiting the application.

Parameters:
None

Returns:
None

3.1.4 mi_CloseCameras2()

Definition:
void mi_CloseCameras2(mi_camera_t *pCameras[], mi_s32 nNumCameras);

Summary:
This routine closes and frees the cameras specified in the array of camera pointers passed in. Reverses the effect of a call to mi_OpenCameras() or mi_OpenCameras2(). It is useful in particular when you have simultaneously opened both a physical camera and an image file pseudo-camera with two separate calls to mi_OpenCameras() and you want to close only one of them.

Parameters:

mi_camera_t	*pCameras[MI_MAX_CAMERAS]	- list of cameras
mi_s32	nNumCameras	- number of cameras in the list

Returns:
None

3.1.5 mi_DetectSensorBaseAddr()

Definition:
mi_s32 mi_DetectSensorBaseAddr(mi_camera_t *pCamera);

Summary:
This function searches the SHiP base addresses listed in the sensor data file to find out which base address the sensor is using, and then sets the sensor->shipAddr field of the pCamera structure. It works by attempting to read CHIP_VERSION_REG register using each base address and looking for a match. If you passed a specific sensor data file name to mi_OpenCameras() then you should call this function to find the sensor base address in use.

Parameters:

mi_camera_t	*pCamera	- camera structure being used

Returns:
MI_CAMERA_SUCCESS if the sensor base address was positively identified.
MI_CAMERA_ERROR otherwise. In this case the shipAddr field will remain unchanged.

3.1.6 mi_DetectPartNumber()

Definition:
mi_s32 mi_DetectPartNumber(mi_camera_t *pCamera);

Summary:

If the sensor data file had multiple part numbers declared, this function selects the correct part number. If you passed a specific sensor data file name to mi_OpenCameras() then you may call this function to correct the part number.

Parameters:
mi_camera_t *pCamera - camera structure being used

Returns:
MI_CAMERA_SUCCESS always.

3.1.7 mi_ProbeFarBus()

Definition:
mi_s32 mi_ProbeFarBus(mi_camera_t *pCamera);

Summary:
This function is for ICP-HD or similar devices. It will test all device addresses on the ICP-HD bus master and remember which ones ACK. Subsequently, any attempt to access an address that did not ACK during probe will immediately fail. This is necessary because the ICP-HD DMA function will halt the firmware if there is no ACK. This function must be called at the appropriate point in the startup sequence. It is unsafe to call after ICP-HD is up and running.

Parameters:
mi_camera_t *pCamera - camera structure being used

Returns:
MI_CAMERA_SUCCESS always.

3.1.8 mi_CancelProbe()

Definition:
void mi_CancelProbe();

Summary:
Cancels any device probe (the process of searching a directory for sensor data files that match the sensors attached to the computer) that may be in progress in another thread. mi_OpenCameras() will return immediately with MI_CAMERA_ERROR.

Parameters:
None

Returns:
None

3.2 Register Search Functions

The following are functions used to search the list of registers, defined in the sensor data file. These are not necessary for basic camera operation and are only provided for convenience.

3.2.1 mi_ExistsRegister()

Definition:
mi_s32 mi_ExistsRegister(mi_camera_t* pCamera, const char* pszRegisterName);

Summary:
This routine determines if the pszRegisterName (Unique identifier) exists in the sensor register list (pCamera->sensor->regs). Put another way, it returns whether mi_FindRegister() will return a valid pointer.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
const char*	pszRegisterName	- unique ID of register to look for

Returns:
True (1) if register exists or False (0) if the register cannot be found

3.2.2 mi_ExistsBitfield()

Definition:
mi_s32 mi_ExistsBitfield(mi_camera_t* pCamera, const char* pszRegisterName, const char* pszBitfieldName);

Summary:
This routine determines if the pszBitfieldName exists as a bitfield for the register of pszRegisterName (unique identifier) in the sensor register list (pCamera->sensor->regs). If the bitfield name is NULL or “” the function returns false. Put another way, it returns whether mi_FindBitfield() will return a valid pointer.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
const char*	pszRegisterName	- unique ID of register containing bitfield
const char*	pszBitfieldName	- name of bitfield to look for

Returns:
True (1) if register exists or False (0) if the bitfield cannot be found

3.2.3 mi_FindRegister()

Definition:

mi_reg_data_t* mi_FindRegister(mi_camera_t* pCamera, const char* pszRegisterName);

Summary:
This routine returns a pointer to the mi_reg_data_t structure with pszRegisterName (unique identifier) found in the sensor register list (pCamera->sensor->regs).

Parameters:

mi_camera_t*	pCamera	- camera structure being used
const char*	pszRegisterName	- unique ID of register to look for

Returns:
Pointer to the mi_reg_data_t structure with name “pszRegisterName”. Returns NULL if register not found.

3.2.4 mi_FindBitfield()

Definition:
mi_bitfield_t* mi_FindBitfield(mi_camera_t* pCamera, const char* pszRegisterName, const char* pszBitfieldName);

Summary:
This routine returns a pointer to the mi_bitfield_t structure with “pszBitfieldName” within the register of pszRegisterName (unique identifier) found in the sensor register list (pCamera->sensor->regs).

Parameters:

mi_camera_t*	pCamera	- camera structure being used
const char*	pszRegisterName	- unique ID of register to look for
const char*	pszBitfieldName	- name of bitfield in register to look for

Returns:
Pointer to the mi_bitfield_ structure with name “pszBitfieldName”. Returns NULL if register or bitfield is not found.

3.2.5 mi_ExistsRegister2()

Definition:
mi_s32 mi_ExistsRegister2(mi_camera_t* pCamera, const char* pszRegisterName);

Summary:
Similar to mi_ExistsRegister(), but searches the chip (non-sensor) registers also.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
const char*	pszRegisterName	- unique ID of register to look for

Returns:

True (1) if register exists or False (0) if the register cannot be found

3.2.6 mi_ExistsBitfield2()

Definition:
mi_s32 mi_ExistsBitfield2(mi_camera_t* pCamera, const char* pszRegisterName, const char* pszBitfieldName);

Summary:
Similar to mi_ExistsBitfield(), but searches the chip (non-sensor) registers also.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
const char*	pszRegisterName	- unique ID of register containing bitfield
const char*	pszBitfieldName	- name of bitfield to look for

Returns:
True (1) if register exists or False (0) if the bitfield cannot be found

3.2.7 mi_FindRegister2()

Definition:
mi_reg_data_t* mi_FindRegister(mi_camera_t* pCamera, const char* pszRegisterName);

Summary:
Similar to mi_FindRegister(), but searches the chip (non-sensor) registers also.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
const char*	pszRegisterName	- unique ID of register to look for

Returns:
Pointer to the mi_reg_data_t structure with name “pszRegisterName”. Returns NULL if register not found.

3.2.8 mi_FindBitfield2()

Definition:
mi_bitfield_t* mi_FindBitfield2(mi_camera_t* pCamera, const char* pszRegisterName, const char* pszBitfieldName);

Summary:
Similar to mi_FindBitfield(), but searches the chip (non-sensor) registers also.

Parameters:

mi_camera_t*	pCamera	- camera structure being used

const char*	pszRegisterName	- unique ID of register to look for
const char*	pszBitfieldName	- name of bitfield in register to look for

Returns:
Pointer to the mi_bitfield_ structure with name “pszBitfieldName”. Returns NULL if register or bitfield is not found.

3.2.9 mi_FindRegisterAddr()

Definition:
mi_reg_data_t* mi_FindRegisterAddr(mi_camera_t* pCamera, mi_u32 regAddr, mi_u32 addrSpace, mi_addr_type addrType);

Summary:
This routine returns a pointer to the mi_reg_data_t structure with a given register address, address space and address type found in the sensor register list (pCamera->sensor->regs)

Parameters:

mi_camera_t*	pCamera	- camera structure being used
mi_u32	regAddr	- register address
mi_u32	addrSpace	- address space or page being searched
mi_addr_type	addrType	- address type for this register

(i.e., MI_REG_ADDR, MI_MCU_ADDR)

Returns:
Pointer to the register structure of the matching register. Returns NULL if register is not found.

3.2.10 mi_FindRegisterWildcard()

Definition:
mi_reg_data_t* mi_FindRegisterWildcard(mi_camera_t* pCamera, const char* pszWildcard, int *pnPos);

Summary:
This routine returns a pointer to the first mi_reg_data_t structure with a unique identifier that matches the wildcard found in the sensor register list (pCamera->sensor->regs). The search starts at position *pnPos in the pCamera->sensor->regs list.

The wildcard works like filename wildcards where “*” matches 0 or more characters and “?” matches any single character. For example “AWB*” matches any register name beginning with “AWB”. On sensors with multiple pages of registers you can restrict the search to a single page by specifying the page name followed by a colon, for example “CORE:*GAIN”. A page name by itself matches all registers on the page. “*” matches all registers. Initialize *pnPos to 0 to start the search at the beginning of the sensor register list.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
const char*	pszWildcard	- unique ID wildcard string
int *	pnPos	- pointer to an integer to track the search

Returns:
Pointer to the mi_reg_data_t structure matching the wildcard, or NULL.

3.2.11 mi_CurrentAddrSpace()

Definition:
mi_addr_space_val_t* mi_CurrentAddrSpace(mi_camera_t* pCamera);

Summary:
This routine returns a pointer to the mi_addr_space_val_t for the current address space being used by the sensor. This is only relevant for sensors which have an address space (page) register.

Parameters:

mi_camera_t *	pCamera	- camera structure being used

Returns:
Pointer to the relevant mi_addr_space_val_t structure from the array pCamera->sensor->addr_space->addr_space_val.

If the sensor data file has no address spaces defined, then that array will not exist and this function returns NULL. This is typical of Bayer-only sensors with only one address space. In such a case, use 0 for the address space and MI_REG_ADDR for the address space type in midlib function calls which require those parameters.

3.3 Register Read and Write Functions

There are several read/write functions that are used to access “high level” sensor registers; they only vary in the parameters which select the register to access. “High Level” means that the register is not limited to hardware registers it also includes microprocessor variables and registers that cover multiple address spans. The mi_Read/WriteSensorReg versions are used when you have access to the mi_reg_data_t structure. The mi_Read/WriteRegAddr functions are used when you only have register address information for the register. The mi_Read/WriteSensorRegStr can be used when you know the unique ID for the register.

3.3.1 mi_ReadSensorReg()

Definition:
mi_s32 mi_ReadSensorReg(mi_camera_t* pCamera, mi_reg_data_t* pReg, mi_u32* val);

Summary:
This routine will read the sensor register pointed to by pReg. The register can be a hardware register, a microprocessor variable or a multi-span register.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
mi_reg_data_t*	pReg	- pointer to the structure for the register being read
mi_u32*	val	- value read

Returns:
MI_CAMERA_SUCCESS for successful register read or error code for failure

3.3.2 mi_WriteSensorReg()

Definition:
mi_s32 mi_WriteSensorReg(mi_camera_t *pCamera, mi_reg_data_t* pReg, mi_u32 val);

Summary:
This routine will write “val” to the sensor register pointed to by pReg. The register can be a hardware register, a microprocessor variable or a multi-span register.

Parameters:

mi_camera_t*	pCamera	- pointer to camera structure being used
mi_reg_data_t*	pReg	- pointer to the register structure of the register being written
mi_u32	val	- value to write

Returns:
MI_CAMERA_SUCCESS for successful register write or error code for failure

3.3.3 mi_ReadSensorRegAddr()

Definition:
mi_s32 mi_ReadSensorRegAddr(mi_camera_t* pCamera, mi_addr_type addrType, mi_u32 addrSpace, mi_u32 addr, mi_s32 is8, mi_u32 *value);

Summary:
This routine will read the sensor register defined by addrType, addrSpace and Addr. The register type can be a hardware register (MI_REG_ADDR), a microprocessor variable (MI_MCU_ADDR) or a multi-span register. This routine is only recommended if you do not have access to the register data structure or unique ID since you are required to pass in all of the information required to access the register. The is8 parameter has different meaning depending on addrType. If addrType is MI_MCU_ADDR or MI_SFR_ADDR then is8 is the width of the variable in bytes, with the special case that is8 == 0 means two bytes. If addrType is MI_REG_ADDR then is8 is the span, with the special case of is8 == 0 means span = 1.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
mi_addr_type	addrType	- address type for this register
mi_u32	addrSpace	- address space (or page) register is on

mi_u32	addr	- register address (or offset for mcu variables)
mi_s32	is8	- variable width or register span
mi_u32	*value	- value read

Returns:
MI_CAMERA_SUCCESS for successful register read or error code for failure

3.3.4 mi_WriteSensorRegAddr()

Definition:
mi_s32 mi_WriteSensorRegAddr(mi_camera_t* pCamera, mi_addr_type addrType, mi_u32 addrSpace, mi_u32 addr, mi_s32 is8, mi_u32 value);

Summary:
Like mi_ReadSensorRegAddr() except writes a register.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
mi_addr_type	addrType	- address type for this register
mi_u32	addrSpace	- address space (or page) register is on
mi_u32	addr	- register address (or offset for mcu variables)
mi_s32	is8	- variable width or register span
mi_u32	value	- value to write

Returns:
MI_CAMERA_SUCCESS for successful register write or error code for failure

3.3.5 mi_ReadSensorRegStr()

Definition:
mi_s32 mi_ReadSensorRegStr(mi_camera_t *pCamera, const char* pszRegisterName, const char* pszBitfieldName, mi_u32 *val);

Summary:
This routine will read the sensor register given the register name (unique ID from sensor data file). An optional bitfield name (from sensor data file) can be provided to read part of a register. The register can be any register or variable defined in the sensor data file.

Parameters:

mi_camera_t*	pCamera	- pointer to camera structure being used
const char*	pszRegisterName	- unique ID of register to read
const char*	pszBitfieldName	- name of bitfield in register read or “”
mi_u32	*val	- value read

Returns:
MI_CAMERA_SUCCESS for successful register read or error code for failure

3.3.6 mi_WriteSensorRegStr()

Definition:
mi_WriteSensorRegStr(mi_camera_t *pCamera, const char* pszRegisterName, const char* pszBitfieldName, mi_u32 val);

Summary:
This routine will write the sensor register given the register name (unique ID from sensor data file). An optional bitfield name (from sensor data file) can be provided to write part of a register (note that this will cause the entire register will be read first). The register can be any register or variable defined in the sensor data file.

Parameters:

mi_camera_t*	pCamera	- pointer to camera structure being used
const char*	pszRegisterName	- unique ID of register to write
const char*	pszBitfieldName	- name of bitfield in register write or “”
mi_u32	val	- value to write

Returns:
MI_CAMERA_SUCCESS for successful register write or error code for failure

3.4 Register Data Type Functions

Registers and bitfields may have a data type. For example, signed, unsigned, fixed point or floating point. The data types supported by midlib are enumerated in the mi_data_types definition. Midlib includes helper functions for converting the unsigned integer values used by the register read and write routines into double precision floating point and string representations.

For example if a register with bitmask 0xFFF (12 bits) and data type signed fixed point with five fraction bits has a value of 0xE13, the double representation is -15.40625, and the string representation is “-15.406”.

3.4.1 mi_Int2Str()

Definition:
mi_s32 mi_Int2Str(mi_data_types dataType, mi_u32 val, mi_u32 mask, char *pszStr, mi_s32 bufferLen);

Summary:
Convert the unsigned integer used by the register read/write routines into a string according to the given data type and mask. The pszStr parameter may be NULL in which case the function just returns the required buffer size. Conversion to string is always in decimal unless the data type MI_HEX is used.

Parameters:

mi_data_types	dataType	- data type
mi_u32	val	- register value data
mi_u32	mask	- right-aligned bitmask

char *	pszStr	- buffer to receive the string
mi_s32	bufferLen	- size of pszStr buffer

Returns:
The number of bytes written to the pszStr buffer, including the trailing NUL byte.

3.4.2 mi_Int2Double()

Definition:
double mi_Int2Double(mi_data_types dataType, mi_u32 val, mi_u32 mask);

Summary:
Convert the unsigned integer used by the register read/write routines into a C *double* according to the given data type and mask.

Parameters:

mi_data_types	dataType	- data type
mi_u32	val	- register value data
mi_u32	mask	- right-aligned bitmask

Returns:
The input value converted to a double according to the data type.

3.4.3 mi_Str2Int()

Definition:
mi_u32 mi_Str2Int(mi_data_types dataType, const char *pszStr, mi_u32 mask);

Summary:
Convert the string to the corresponding unsigned integer value used by the register read/write routines according to the given data type and mask.

Parameters:

mi_data_types	dataType	- data type
const char *	pszStr	- the string
mi_u32	mask	- right-aligned bitmask

Returns:
The unsigned integer value.

3.4.4 mi_Str2Double()

Definition:
double mi_Str2Double(mi_data_types dataType, const char *pszStr, mi_u32 mask);

Summary:

Convert the string to the corresponding double value according to the given data type and mask.

Parameters:

mi_data_types	dataType	- data type
const char *	pszStr	- the string
mi_u32	mask	- right-aligned bitmask

Returns:
The input string converted to a double according to the data type.

3.4.5 mi_Double2Int()

Definition:
mi_u32 mi_Double2Int(mi_data_types dataType, double d, mi_u32 mask);

Summary:
Convert the double value into the unsigned integer value used by the register read/write routines, according to the given data type and mask.

Parameters:

mi_data_types	dataType	- data type
double	d	- register value data
mi_u32	mask	- right-aligned bitmask

Returns:
The input value converted to an unsigned integer.

3.4.6 mi_Double2Str()

Definition:
mi_s32 mi_Double2Str(mi_data_types dataType, double d, mi_u32 mask, char *pszStr, mi_s32 bufferLen);

Summary:
Convert the double value into a string according to the given data type and mask. The pszStr parameter may be NULL in which case the function just returns the required buffer size. Conversion to string is always in decimal unless the data type MI_HEX is used.

Parameters:

mi_data_types	dataType	- data type
double	d	- register value data
mi_u32	mask	- right-aligned bitmask
char *	pszStr	- buffer to receive the string
mi_s32	bufferLen	- size of pszStr buffer

Returns:
The number of bytes written to the pszStr buffer, including the trailing NUL byte.

3.4.7 mi_GetDataTypeString()

Definition:
const char *mi_GetDataTypeString(mi_data_types dataType);

Summary:
Return a string representing the name of the data type. For example mi_GetDataTypeString (MI_FIXED5) returns “fixed5”.

Parameters:

mi_data_types	dataType	- data type

Returns:
Pointer to a string.

3.4.8 mi_DataTypeFromString()

Definition:
mi_data_types mi_DataTypeFromString(const char *pszStr);

Summary:
Return the mi_data_types value given the name of a data type. For example mi_DataTypeFromString(“fixed5”) returns MI_FIXED5.

Parameters:

const char *	pszStr	- name of a data type

Returns:
Data type.

3.4.9 mi_DataTypeMinimum()

Definition:
double mi_DataTypeMinimum(mi_data_types dataType, mi_u32 mask);

Summary:
Return the minimum value possible with the given data type and mask. For example, the minimum value for 12-bit signed fixed point with five fraction bits is -64.0. The minimum value is undefined for floating point types, and this function will return 0.0.

Parameters:

mi_data_types	dataType	- data type
mi_u32	mask	- right-aligned bitmask

Returns:

Minimum possible value.

3.4.10 mi_DataTypeMaximum()

Definition:
double mi_DataTypeMaximum(mi_data_types dataType, mi_u32 mask);

Summary:
Return the maximum value possible with the given data type and mask. For example, the maximum value for 12-bit signed fixed point with five fraction bits is 63.96875. The maximum value is undefined for floating point types, and this function will return 0.0.

Parameters:

mi_data_types	dataType	- data type
mi_u32	mask	- right-aligned bitmask

Returns:
Maximum possible value.

3.4.11 mi_DataTypeStep()

Definition:
double mi_DataTypeStep(mi_data_types dataType, mi_u32 mask);

Summary:
Return the difference between two successive values of the given data type. For example the step for fixed point with five fraction bits is 0.03125. The step is undefined for floating point types, and this function will return 0.0.

Parameters:

mi_data_types	dataType	- data type
mi_u32	mask	- right-aligned bitmask

Returns:
Step between successive values.

3.5 Xsdat/sdat, INI and Cdat File Functions

3.5.1 mi_ParseSensorFile()

Definition:
mi_s32 mi_ParseSensorFile(mi_camera_t *pCamera, const char* fileName, mi_sensor_t *sensor_data);

Summary:
This routine parses a sensor data (.sdat or .xsdat) file (fileName) into the mi_sensor_t structure and returns an error message if there are parse errors. For more detailed information as to which line failed, turn on error logging. You can parse a sensor data file into a memory-only mi_camera_t without accessing any physical device with the following code

```
mi_camera_t *pCamera = (mi_camera_t *)calloc(1, sizeof(mi_camera_t));
pCamera->sensor = (mi_sensor_t *)calloc(1, sizeof(mi_sensor_t));
mi_InitCameraContext(pCamera);
retVal = mi_ParseSensorFile(pCamera, fileName, pCamera->sensor);
```

The functions mi_FindRegister(), mi_ExistsRegister(), etc. can be used with the pCamera structure.
To clean up when finished with the memory-only mi_camera_t do the following:

```
mi_DestructSensor(pCamera->sensor);
mi_FreeCameraContext(pCamera);
free(pCamera->sensor);
free(pCamera);
```

Parameters:

mi_camera_t	*pCamera	- camera structure being used
const char*	filename	- fileName of sensor data file to parse
mi_sensor_t	*sensor_data	- structure to fill in with sensor data

Returns:

MI_PARSE_SUCCESS	Parsing was successful
MI_DUPLICATE_DESC_ERROR	Duplicate unique descriptor was found
MI_PARSE_FILE_ERROR	Unable to open sensor data file
MI_PARSE_REG_ERROR	Error parsing the register descriptor
MI_UKNOWN_SECTION_ERROR	Unknown Section found in sensor data file
MI_CHIP_DESC_ERROR	Error parsing the chip descriptor section
MI_PARSE_ADDR_SPACE_ERROR	Error parsing the address space section

See Also:
mi_OpenErrorLog()
mi_InitCameraContext()
mi_DestructSensor()

3.5.2 mi_ParseFarSensorFile()

Definition:
mi_s32 mi_ParseFarSensorFile (mi_camera_t *pCamera, const char *far_id, mi_addr_type far_type, mi_u32 far_base, const char *far_sdat, mi_sensor_t *sensor_data);

Summary:
This routine loads a sensor data file and adds the register data to the existing pCamera as FAR type registers. A Far sensor is one that is attached to a ‘host’ chip and accessed indirectly, for example a secondary sensor attached to a primary sensor. Up to two Far sensors may be declared, designated FAR1 and FAR2.

Parameters:

mi_camera_t	*pCamera	- camera structure being used
const char *	far_id	- symbol for the registers, can be anything but is normally either “FAR1” or “FAR2”
mi_addr_type	far_type	- MI_FAR1_REG_ADDR or MI_FAR2_REG_ADDR
mi_u32	far_base	- SHIP base address of the Far sensor
const char *	far_sdat	- full path name of sensor data file to load
mi_sensor_t *	sensor_data	- data structure to accept the file data, may be NULL

Returns:

MI_PARSE_SUCCESS	Parsing was successful
MI_DUPLICATE_DESC_ERROR	Duplicate unique descriptor was found
MI_PARSE_FILE_ERROR	Unable to open sensor data file
MI_PARSE_REG_ERROR	Error parsing the register descriptor
MI_UKNOWN_SECTION_ERROR	Unknown Section found in sensor data file
MI_CHIP_DESC_ERROR	Error parsing the chip descriptor section
MI_PARSE_ADDR_SPACE_ERROR	Error parsing the address space section

See Also:
mi_ParseFarSensorFile()

3.5.3 mi_ParseChipFile()

Definition:

```
mi_s32  mi_ParseChipFile(mi_camera_t *pCamera, const char* fileName, mi_chip_t
*chip_data);
```

Summary:

This routine parses a chip data (.cdat) file (fileName) into the mi_chip_t structure and returns an error message if there are parse errors. For more detailed information as to which line failed, turn on error logging.

Parameters:

mi_camera_t	*pCamera	- camera structure being used
const char*	fileName	- fileName of chip data file to parse
mi_chip_t	*chip_data	- structure to fill in with chip data

Returns:

MI_PARSE_SUCCESS	Parsing was successful
MI_DUPLICATE_DESC_ERROR	Duplicate unique descriptor was found
MI_PARSE_FILE_ERROR	Unable to open chip data file
MI_PARSE_REG_ERROR	Error parsing the register descriptor
MI_UKNOWN_SECTION_ERROR	Unknown Section found in chip data file
MI_CHIP_DESC_ERROR	Error parsing the chip descriptor section

See Also:
mi_OpenErrorLog()

3.5.4 mi_DestructSensor()

Definition:
void mi_FreeCameraContext(mi_sensor_t *pSensor);

Summary:
Clean up and free the sensor data structure. If you called mi_ParseSensorFile(), this function will free the memory allocated by mi_ParseSensorFile().

Parameters:

mi_sensor_t	*pSensor	- sensor structure being used

Returns:
None.

See Also:
mi_ParseSensorFile()

3.5.5 mi_LoadINIPreset()

Definition:
mi_s32 mi_LoadINIPreset(mi_camera_t* pCamera, const char* szIniFileName, const char* szPresetName)

Summary:
Load a preset section from an INI file. For information on the INI file syntax see the DevWare inifile User Guide (DevWare inifile User Guide.pdf). General error messages are provided for errors. For more detailed information as to which line failed turn on error logging.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
const char*	szIniFileName	- full path name of .ini file
const char*	szPresetName	- preset name in .ini file to load

If the parameter szIniFileName is NULL or an empty string, the function will use the default INI file for the sensor.

If the parameter szPresetName is NULL or an empty string, the function will use "`Demo Initialization`" as the default preset name.

Returns:

MI_INI_SUCCESS	INI preset is loaded successfully
MI_INI_KEY_NOT_SUPPORTED	key is not supported – will be ignored
MI_INI_LOAD_ERROR	Error loading INI preset
MI_INI_POLLREG_TIMEOUT	time out in POLLREG command

See Also:
mi_OpenErrorLog()

3.5.6 mi_LoadPromColorCorrection()

Definition:
mi_s32 mi_LoadPromColorCorrection(mi_camera_t* pCamera, mi_u32 nBase, const char * szLabel)

Summary:
Load a color correction preset from EEPROM. If the sensor headboard has an EEPROM with a color correction setting loaded into it, then you can use this function to load the color correction data into the sensor.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
mi_u32	nBase	- base address of the EEPROM, normally 0xA8
const char*	szLabel	- preset name to load, may be NULL

Returns:

MI_CAMERA_SUCCESS	loaded successfully
MI_CAMERA_ERROR	error accessing EEPROM or sensor

See Also:

3.5.7 mi_LoadPromLensCalibration()

Definition:
mi_s32 mi_LoadPromLensCalibration(mi_camera_t* pCamera, mi_u32 nBase, const char * szLabel)

Summary:
Load a lens calibration preset from EEPROM. If the sensor headboard has an EEPROM with a lens calibration setting loaded into it, then you can use this function to load the lens calibration data into the sensor.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
mi_u32	nBase	- base address of the EEPROM, normally 0xA8
const char*	szLabel	- preset name to load, may be NULL

Returns:

MI_CAMERA_SUCCESS	loaded successfully
MI_CAMERA_ERROR	error accessing EEPROM or sensor

See Also:

3.5.8 mi_LoadPromScratchpad()

Definition:
mi_s32 mi_LoadPromScratchpad(mi_camera_t* pCamera, mi_u32 nBase, const char * szLabel)

Summary:
Load an ICP-HD scratchpad data block from EEPROM. If the sensor headboard has an EEPROM with scratchpad data loaded into it, then you can use this function to load the data into the device.

Parameters:

mi_camera_t*	pCamera	- camera structure being used
mi_u32	nBase	- base address of the EEPROM, normally 0xA8
const char*	szLabel	- preset name to load, may be NULL

Returns:

MI_CAMERA_SUCCESS	loaded successfully
MI_CAMERA_ERROR	error accessing EEPROM or sensor

See Also:

3.6 Miscellaneous Functions

3.6.1 mi_Home()

Definition:
const char * mi_Home();

Summary:
This function returns the installation directory for the DevSuite software, normally from the MI_HOME environment variable.

Parameters:
None

Returns:
Installation directory path

3.6.2 mi_SensorData()

Definition:
const char * mi_SensorData();

Summary:
This function returns the directory for the sensor data files. The main purpose is to pass it as the third parameter of mi_OpenCameras() or mi_OpenCameras2().

Parameters:
None

Returns:
Sensor data files directory path

3.6.3 mi_GetImageTypeStr()

Definition:
void mi_GetImageTypeStr(mi_image_types image_type, mi_string pImageStr);

Summary:
This routine returns the string describing a given image type, for example MI_YCBCR returns "YCbCr".

Parameters:

mi_image_types	image_type	- image type to get string for
mi_string	pImageStr	- string returned which describes image type

Returns:
None.

3.6.4 mi_InvalidateRegCache()

Definition:
void mi_InvalidateRegCache (mi_camera_t *pCamera);

Summary:
This routine is used to invalidate the internal sensor register cache. This causes the next sensor register reads to read from the sensor and update the cache. After doing a software reset on the sensor or similar operation that will change the contents of many registers, you should invalidate the cache. This routine is only relevant if the mode MI_USE_REG_CACHE has been set. See mi_setMode() for more information on MI_USE_REG_CACHE.

Parameters:

mi_camera_t	*pCamera	- camera structure being used

Returns:
None.

3.6.5 mi_InvalidateRegCacheReg()

Definition:
void mi_InvalidateRegCache (mi_camera_t *pCamera , mi_reg_data_t *pReg);

Summary:
Invalidate cached value of the specified register. The next read to this register will go the device.

Parameters:

mi_camera_t	*pCamera	- camera structure being used
mi_reg_data_t	*pReg	- register pointer into sensor->regs

Returns:
None.

3.6.6 mi_InvalidateRegCacheRegName()

Definition:
void mi_InvalidateRegCache (mi_camera_t *pCamera , const char *pszRegName);

Summary:
Invalidate cached value of the specified register. The next read to this register will go the device.

Parameters:

mi_camera_t	*pCamera	- camera structure being used
const char	*pszRegName	- name of register

Returns:
None.

3.6.7 mi_GetRegCacheValueReg()

Definition:
mi_s32 mi_GetRegCacheValueReg(mi_camera_t *pCamera , mi_reg_data_t *pReg, mi_u32 *val);

Summary:
Retrieve the cached value of the specified register.

Parameters:

mi_camera_t	*pCamera	- camera structure being used
mi_reg_data_t	*pReg	- register pointer into sensor->regs
mi_u32	*val	- pointer to receive the data

Returns:

MI_CAMERA_ERROR if there is no value in the cache, MI_CAMERA_SUCCESS otherwise.

3.6.8 mi_SetVolatileRegStr()

Definition:
void mi_SetVolatileRegStr(mi_camera_t *pCamera , const char *pszRegisterName);

Summary:
Set the named sensor register as volatile. The register value will never be cached, and reads of the register will always read from the device.

Parameters:

mi_camera_t	*pCamera	- camera structure being used
const char	*pszRegName	- name of register

Returns:
None.

3.6.9 mi_IsBayer()

Definition:
mi_u8 mi_IsBayer(mi_camera_t *pCamera);

Summary:
This routine checks to see if the imageType of the sensor is Bayer (e.g. MI-BAYER_8, MI_BAYER_10, MI_BAYER_8_ZOOM2, etc) or not.

Parameters:

mi_camera_t	*pCamera	- camera structure being used

Returns:
This routine returns TRUE (1) if the pCamera->sensor->imageType is Bayer data otherwise returns FALSE (0).

3.6.10 mi_IsSOC()

Definition:
mi_u8 mi_IsSOC(mi_camera_t *pCamera);

Summary:
This routine checks to see whether the sensor for this camera (pCamera->sensor->sensorType) is an SOC (e.g. A-1310SOC, A-360SOC, etc.)

Parameters:

mi_camera_t	*pCamera	- camera structure being used

Returns:
This routine returns True (1) if the sensor is considered an SOC sensor and returns 0 otherwise.

3.6.11 mi_IsBayerImageType()

Definition:
mi_s32 mi_IsBayerImageType(mi_image_types imageType);

Summary:
This routine checks to see if the imageType parameter is Bayer (e.g. MI-BAYER_8, MI_BAYER_10, MI_BAYER_8_ZOOM2, etc) or not.

Parameters:

mi_image_types	imageType	- image type to test

Returns:
This routine returns TRUE (1) if imageType is Bayer otherwise returns FALSE (0).

3.6.12 mi_GetAddrIncrement()

Definition:
int mi_GetAddrIncrement(mi_camera_t *pCamera, mi_addr_type addrType);

Summary:
On some sensors only even numbered register addresses are allowed (consecutive registers are at every other address), other sensor use even and odd numbered addresses. This function returns 2 if only even numbered addresses are allowed, 1 otherwise.

Parameters:

mi_camera_t	*pCamera	- camera structure being used
mi_addr_type	addrType	- register type being queried

Returns:
2 if only even numbered register addresses are allowed, 1 if even and odd numbered addresses are allowed.

3.7 Error Log Functions

3.7.1 mi_OpenErrorLog()

Definition:

void mi_OpenErrorLog(mi_s32 error_log_level, const char* baseFileName);

Summary:
This routine opens an error log file to log error messages. The error_log_level can be

MI_NO_ERROR_LOG	- Error logging is turned off
MI_ERROR_SEVERE	- Log Severe errors
MI ERROR_MINOR	- Log Minor errors
MI_ALL_ERRORS	- Logs all error messages (Severe and Minor)
MI_LOG	- Logs General logging message
MI_LOG_SHIP	- Log Serial I/O messages (SHIP)
MI_LOG_USB	- Log USB transactions
MI_LOG_DEBUG	- Log Debug messages

The baseFileName is the base name for the log file. The actual file name will be the base name plus a digit 0-4. The baseFileName may be NULL in which case the default “error.txt” (in the current directory) is used. So in the default case the filename will cycle from error0.txt to error4.txt. The name of the current error log is obtained with mi_GetErrorLogFileName()

Parameters:

mi_s32	error_log_level	- the types of errors to log
const char*	baseFileName	- the base name for the log file

Returns:
None

3.7.2 mi_CloseErrorLog()

Definition:
void mi_CloseErrorLog ();

Summary:
This routine closes the error log that was created with mi_OpenErrorLog().

Parameters:
None

Returns:
None

3.7.3 mi_GetErrorLogFileName()

Definition:
void mi_GetErrorLogFileName(mi_string fileName);

Summary:
This routine gets the name of the current error log. The actual file name will be the base name supplied in mi_OpenErrorLog() or the default name “error.txt” (in the current directory) plus a digit 0-4. So in the default case the filename will cycle from error0.txt to error4.txt. This routine returns the current filename being used.

Parameters:
mi_string fileName - The current filename returned

Returns:
None

3.8 Device Arrival/Removal Notification

3.8.1 mi_SetDeviceChangeCallback()

Definition:
mi_s32 mi_SetDeviceChangeCallback(HWND hwnd, MIDEVCALLBACK lpDCB);

Summary:
This function enables or disables application notification for removal of a plug-and-play camera. If the camera device is removed while the application is running, midlib stops the device transport, and calls the user-defined callback routine. To disable notification pass NULL for both parameters.

Once enabled, the window handle cannot be changed, but the callback function can be. To change the window handle you must disable notification, then enable it again on the new handle. Be sure to disable notification before the window is destroyed, otherwise midlib will be left registered to an invalid window handle.

You can call mi_SetDeviceChangeCallback() either before or after mi_OpenCameras(). Also you can disable notification before or after mi_CloseCameras(). Therefore, you just enable when your main window is created, and disable when it’s closed.

Device removal notification has the following requirements:
Not all devices can support this function, usually just USB devices.
Device notification only works with applications that have a Windows message queue; console applications will not work.
Device arrival is currently not supported.

The user-defined callback function has the following prototype:

typedef mi_u32 (*MIDEVCALLBACK)(HWND hwnd, struct mi_camera_t *pCamera, mi_u32 Flags);

Callback parameters:
HWND hwnd - handle of the window that received the removal notice

mi_camera_t	*pCamera	- camera structure being monitored for removal
mi_u32	Flags	- defined below

Flags can be one or more of the following definitions:
MI_DEVEVENT_REMOVAL - A device is being removed

Parameters:

HWND	hwnd	- handle of the window that will receive removal notification
MIDEVCALLBACK	lpDCB	- pointer to a routine to call when a device is removed

Returns:
MI_CAMERA_SUCCESS for success,
MI_CAMERA_NOT_SUPPORTED if the current transport does not support the device notification
MI_CAMERA_ERROR for invalid parameters or if an older version of MIUSB2.SYS is used

3.9 Types

3.9.1 mi_camera_t

```
typedef struct
{
   mi_product_ids  productID;
   mi_u32          productVersion;
   mi_string       productName;
   mi_u32          firmwareVersion;
   mi_string       transportName;
   mi_u32          transportType;
   void*           context;
   mi_sensor_t*    sensor;
   mi_s32          num_chips;
   mi_chip_t*      chip;
   void*           int_dev_functions;
   mi_s32          (* startTransport)(mi_camera_t *pCamera);
   mi_s32          (* stopTransport)(mi_camera_t *pCamera);
   mi_s32          (* readSensorRegisters)( mi_camera_t *pCamera, mi_u32 addrSpace, mi_u32
                   start_addr, mi_u32 num_regs, mi_u32 vals[]);
  mi_s32           (* writeSensorRegisters)(mi_camera_t *pCamera, mi_u32 addrSpace,  mi_u32
                   start_addr, mi_u32 num_regs, mi_u32 vals[]);
  mi_s32           (* readSensorRegList)(mi_camera_t *pCamera, mi_u32 numRegs, mi_u32
                   addrSpaces[], mi_u8 regAddrs[],  mi_u32 vals[]);
  mi_s32           (* readRegister)(mi_camera_t *pCamera, mi_u8 shipAddr, mi_u32 reg_addr,
                   mi_u32 *val);
  mi_s32           (* writeRegister)(mi_camera_t *pCamera, mi_u8 shipAddr, mi_u32 reg_addr,
                   mi_u32 val);
```

```
    mi_s32          (* readRegisters)( mi_camera_t *pCamera, mi_u8 shipAddr, mi_u32
                    start_addr, mi_u32 num_regs, mi_u32 vals[]);
    mi_s32          (* writeRegisters)(mi_camera_t *pCamera, mi_u8 shipAddr,  mi_u32
                    start_addr, mi_u32 num_regs, mi_u32 vals[]);
    mi_s32          (* grabFrame)(mi_camera_t *pCamera, mi_u8 *pInBuffer, mi_u32 bufferSize);
    mi_s32          (* getFrameData)(mi_camera_t *pCamera, mi_frame_data_t *frame_data);
    mi_s32          (* updateFrameSize)(mi_camera_t *pCamera, mi_u32 width, mi_u32 height,
                    mi_u8 nBitsPerClock , mi_u8 nClocksPerPixel);
    mi_s32          (* updateBufferSize)(mi_camera_t* pCamera, mi_u32 rawBufferSize);
    mi_s32          (* setMode) (mi_camera_t *pCamera,  mi_modes mode, mi_u32 val);
    mi_s32          (* getMode) (mi_camera_t *pCamera,  mi_modes mode, mi_u32* val);
    mi_s32          (* initTransport)( mi_camera_t *pCamera, mi_u8 bitsPerClock, mi_u8
                    clocksPerPixel, mi_u8 polarity, mi_u8 pixelOffset, mi_u8 noFWCalls);
} mi_camera_t;
```

The following data is part of the camera structure:

productID	Product ID for the Camera (see mi_product_ids for more information)
productVersion	Version number of the product (camera)
productName	Name of product
firmwareVersion	Version of the firmware on board
transportName	Name of the transport
transportType	Type of transport camera is using (see Camera Transport defines for values)
context	This is the internal camera specific context
sensor	Sensor that is attached to the device (see mi_sensor_t for more information)
num_chips	Number of additional chips on board
chip	The chip data (see mi_chip_t for more information)
int_dev_functions	Internal development functions are very low level functions for reading and writing different signals on the USB2 controller. For more information regarding these functions please email svedovato@aptina.com

The following are transport specific function pointers. More information on these functions can be found in Section 3.10 this document. Normally you wouldn't call these functions directly, but instead use mi_startTransport(), etc.

startTransport	Starts a transport
stopTransport	Stops a transport
readSensorRegisters	Read a sequence of sensor registers
writeSensorRegisters	Write a sequence of sensor register values
readSensorRegList	Read a non-sequential list of sensor registers
readRegister	Reads a single register (use readSensorRegisters for sensor registers)
writeRegister	Writes to a register (use writeSensorRegisters for sensor registers)
readRegisters	Read a sequence of registers (use readSensorRegisters for sensor registers)
writeRegisters	Write a sequence of register values (use writeSensorRegisters for sensor registers)
grabFrame	Returns an image frame
getFrameData	Returns additional information about the last frame grabbed
updateFrameSize	Set frame data size given new width, height, bits per clock and clocks per pixel
updateBufferSize	Set frame data size given new rawBufferSize
setMode	Sets one of the mi_modes
getMode	Gets the value of one of the mi_modes
initTransport	Initialize the transport if no sensor data file was used in mi_OpenCameras

3.9.2 mi_sensor_t

This structure holds all of the pertinent information about a supported sensor.

```
typedef struct {
   mi_string          sensorName;     // name of the sensor
   mi_sensor_types    sensorType;     // the sensor type
   mi_u32             fullWidth;      // full image width of sensor
   mi_u32             fullHeight;     // full image height of sensor
   mi_u32             width;          // current image width
   mi_u32             height;         // current image height
   mi_u32             zoomFactor;     // current zoom factor (default 1)
   mi_u32             pixelBytes;     // number of bytes per pixel
   mi_u32             pixelBits;      // number of bits per pixel
   mi_u32             bufferSize;     // minimum size of image buffer
   mi_image_types     imageType;      // the raw image type
   mi_u32             shipAddr;       // base SHIP address for sensor
   mi_s32             reg_addr_size;  // register address size (8/16)
   mi_s32             reg_data_size;  // register data size (8/16)
   mi_s32             num_regs;       // number of sensor registers
   mi_reg_data_t*     regs;           // array of sensor registers
   mi_addr_space_t*   addr_space;     // holds addr space info or NULL
   mi_string          sensorFileName;// filename of sensor data file
   mi_u32             sensorVersion;  // version number of the sensor
   mi_string          partNumber;     // Aptina/Micron MT9 part number
   mi_string          versionName;    // sensor version name (Rev0,ES1)
} mi_sensor_t;
```

3.9.3 mi_reg_data_t

This structure holds all of the register information stored in the sensor data files. This structure describes information about a specific register and its associated bitfields within the register.

```
typedef struct {
   mi_string        unique_desc;  // unique register descriptor name
   mi_u32           reg_addr;     // register address
   mi_u32           reg_space;    // superseded by addr_space
   mi_u32           bitmask;      // bitmask describing valid bits
   mi_u32           default_val;  // default value for register
   mi_s32           rw;           // 1 if read/write; 0 if read-only; 2 if write-only
   mi_string        reg_desc;     // short description of register
   mi_string        detail;       // additional details describing field
   mi_s32           num_bitfields; // number of bitfields in this register
   mi_bitfield_t*   bitfield;     // pointer to array of bitfields
   mi_s32           addr_span;    // number of addresses register covers
   mi_addr_space_val_t* addr_space;// pointer to address space structure
   mi_data_types datatype;        // data type of this register
   mi_u32           minimum;      // minimum recommended value
   mi_u32           maximum;      // maximum recommended value
} mi_reg_data_t;
```

3.9.4 mi_bitfield_t

This structure is used to hold bit field information for a particular bitfield. This data is parsed from the sensor data files.

```
typedef struct
{
   mi_string      id;              // bitfield descriptor name (unique within register)
   mi_u32         bitmask;         // bitmask describing valid bits of bitfield
   mi_s32         rw;              // 1 if read/write; 0 if read-only; 2 if write-only
   mi_string      desc;            // short description of bitfield
   mi_string      detail;          // longer detailed description
   mi_data_types datatype;         // data type of this bitfield
   mi_u32           minimum;       // minimum recommended value
   mi_u32           maximum;       // maximum recommended value
} mi_bitfield_t;
```

3.9.5 mi_addr_space_t

This structure is used to describe the address space register for this sensor. It is filled in when a sensor data file is parsed only for a sensor which has an address space selector register. This data is only used for sensors with address space registers (ie, the SOC sensors).

```
typedef struct {
   mi_u32                  reg_addr;      // register address for the sensor's address space
                                          // selector (page register)
   mi_s32                  num_vals;      // number of possible address spaces available
   mi_addr_space_val_t*    addr_space_val; // array of possible address space values
   mi_u32                  far1_reg_addr; // 1st far sensor's page register
   mi_u32                  far2_reg_addr; // 2nd far sensor's page register
} mi_addr_space_t;
```

Note that reg_addr will be 0 if there is no address space selector register

3.9.6 mi_addr_space_val_t

This structure is used to describe a value for the address space selector. This information is filled in when a sensor data file is parsed

```
typedef struct
{
   mi_string       ID;          // unique ID used in sensor data file – example "CORE"
   mi_string       name;        // name of the address space
   mi_u32          val;         // value to set the address space selector to
   mi_addr_type    type;        // register address type
```

```
    mi_u32          far_base;     // base address of sensor on far bus
    mi_u32          far_addr_size; // register address width on far sensor
    mi_u32          far_data_size; // register width on far sensor
} mi_addr_space_t;
```

3.9.7 mi_chip_t

This structure holds all of the pertinent information about a supported companion chip on a camera.

```
typedef struct {
    mi_string       chipName;           //Name of the companion chip
    mi_u32          baseAddr;           //Base I2C address for chip
    mi_s32          serial_addr_size;   //8/16 bit register address size
    mi_s32          serial_data_size;   //8/16 bit register data size
    mi_s32          num_regs;           //Number of registers on chip
    mi_reg_data_t*  regs;               //Array of registers on chip
} mi_chip_t;
```

3.9.8 mi_frame_data_t

This structure is used to hold all of the information about a frame grabbed with the last grabFrame().

```
typedef struct
{
    mi_u32      frameNumber;     //Frame number of last frame grabbed
    mi_u32      bytesRequested;  //Number bytes requested for last frame
    mi_u32      bytesReturned;   //Number bytes returned for last frame
    mi_u32      numRegsReturned;    //no longer used
    mi_u32      regValsReturned[MI_MAX_REGS]; //no longer used
    mi_u32      imageBytesReturned; // The number of bytes of image data for last frame grabbed
} mi_frame_data_t;
```

3.9.9 Data Types

The following data types are used for portability between compilers, for more information on portability, see Section 4.0.

```
typedef unsigned char   mi_u8;      //unsigned 8 bit data type
typedef unsigned short  mi_u16;     //unsigned 16 bit data type
typedef unsigned int    mi_u32;     //unsigned 32 bit data type
typedef signed char     mi_s8;      //signed 8 bit data type
```

```
typedef signed short      mi_s16;         //signed 16 bit data type
typedef signed int        mi_s32;         //signed 32 bit data type
typedef __w64 int         mi_intptr;      //type that can be cast to an int or a pointer
typedef char mi_string[MI_MAX_STRING];  //char array of size MI_MAX_STRING
```

3.10 Camera transport functions

The following functions are the camera specific functions defined in the camera structure mi_camera_t. Each camera will have its own set of function pointers.

3.10.1 mi_startTransport()

Definition:
mi_s32 mi_startTransport(mi_camera_t *pCamera)

Summary:
This routine starts the transport, it must be called before any of the other camera specific routines can be called.

Parameters:
mi_camera_t *pCamera - pointer to camera to start

Returns:
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:
mi_stopTransport()

3.10.2 mi_stopTransport()

Definition:
mi_s32 mi_stopTransport(mi_camera_t *pCamera)

Summary:
This routine stops the previously opened camera.

Parameters:
mi_camera_t *pCamera - pointer to the camera to stop

Returns:
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:
mi_startTransport()

3.10.3 mi_readSensorRegisters()

Definition:
mi_s32 mi_readSensorRegisters(mi_camera_t* pCamera, mi_u32 addrSpace, mi_u32 startAddr mi_u32 numRegs, mi_u32 vals[])

Summary:
This function reads a set of consecutive "numRegs" **sensor** registers starting at startAddr and returns the array of values in "vals". This call is only used when the camera contains a **sensor** and will use the ship address from the sensor structure to do the register reads. The addrSpace is only used with sensors that have multiple address spaces (SOC sensors for example), otherwise it is ignored. The consecutive registers must occur on a single page (address space) and the address space register will be set if necessary before the registers are read.
If the mode MI_USE_REG_CACHE has been set then the register value will be read from the cache if possible. If register caching is being used then you must use mi_writeSensorRegisters() to write sensor registers.
This function can only read registers of type MI_REG_ADDR, that is, simple registers, not MCU variables or other kinds of registers accessed indirectly.

Parameters:

mi_camera_t*	pCamera	- pointer to the camera structure
mi_u32	addrSpace	- Address space of this register if multiple address spaces
mi_u32	startAddr	- first sensor register address to read from
mi_u32	numRegs	- number of consecutive sensor registers to read (1 or more)
mi_u32	val[]	- array of numRegs values read

Returns:
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:
mi_setMode()
mi_InvalidateRegCache()
mi_writeSensorRegisters()
mi_ReadSensorReg()

3.10.4 mi_writeSensorRegisters()

Definition:
mi_s32 mi_writeSensorRegisters(mi_camera_t *pCamera, , mi_u32 addrSpace, mi_u32 startAddr, mi_u32 numRegs, mi_u32 vals[])

Summary:
This function writes a set of **sensor** register values to a consecutive "numRegs" of **sensor** registers starting at startAddr. This call is only used when the camera contains a **sensor** and will use the ship address from the sensor structure to do the register writes. The addrSpace is only used with sensors that have multiple address spaces (SOC sensors for example), otherwise it is ignored. The consecutive registers must occur on a single page (address space) and the address space register will be set if necessary before the registers are written.

If register caching is being used to read sensor registers, then this routine must be used to write sensor registers.

This function can only write registers of type MI_REG_ADDR, that is, simple registers, not MCU variables or other kinds of registers accessed indirectly.

Parameters:

mi_camera_t*	pCamera	- pointer to the camera structure
mi_u32	addrSpace	- Address space of this register if multiple address spaces
mi_u32	startAddr	- first sensor register address to write to
mi_u32	numRegs	- number of consecutive sensor registers to write (1 or more)
mi_u32	val[]	- array of numRegs values to write

Returns:

MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:

mi_setMode()
mi_InvalidateRegCache()
mi_readSensorRegisters()

3.10.5 mi_readRegister()

Definition:

mi_s32 mi_readRegister(mi_camera_t* pCamera, mi_u32 shipAddr, mi_u32 regAddr, mi_u32* val)

Summary:

When reading registers from a sensor the preferred method is the mi_readSensorRegisters() which allows for register caching and automatic setting of the address space register. This function reads a value from a general register, given the register's address and the SHIP base address. Use mi_setMode() to set the register address width and register data width.

Parameters:

mi_camera_t*	pCamera	- pointer to the camera structure
mi_u32	shipAddr	- SHIP base address to read from
mi_u32	regAddr	- register address to read from
mi_u32*	val	- value read

Returns:

MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:

mi_readSensorRegisters()
mi_readRegisters()
mi_setMode()

3.10.6 mi_writeRegister()

Definition:
mi_s32 mi_writeRegister(mi_camera_t* pCamera, mi_u32 shipAddr, mi_u32 regAddr, mi_u32 val)

Summary:
When writing registers to a sensor the preferred method is the mi_writeSensorRegisters() which allows for register caching and automatic setting of the address space register. This function writes a value to a given register address and the SHIP base address. Use mi_setMode() to set the register address width and register data width.

Parameters:

mi_camera_t*	pCamera	- pointer to the camera structure
mi_u32	shipAddr	- SHIP base address to write to
mi_u32	regAddr	- register address to write to
mi_u32	val	- value to write

Returns:
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:
mi_writeSensorRegisters()
mi_writeRegisters()
mi_setMode()

3.10.7 mi_readRegisters()

Definition:
mi_s32 mi_readRegisters(mi_camera_t* pCamera, mi_u32 shipAddr, mi_u32 startAddr mi_u32 numRegs, mi_u32 vals[])

Summary:
This function reads a set of consecutive “numRegs” registers starting at startAddr and returns the array of values in “vals”. When reading registers from a sensor the preferred method is mi_readSensorRegisters() which allows for register caching and automatic setting of the address space register. Note that the USB firmware can currently only do 64 sequential reads, if more than 64 are requested then the call will be broken down into single read requests. Use mi_setMode() to set the register address width and register data width.

Parameters:

mi_camera_t*	pCamera	- pointer to the camera structure
mi_u32	shipAddr	- SHIP base address to read from
mi_u32	startAddr	- first register address to read from
mi_u32	numRegs	- number of consecutive registers to read
mi_u32	val[]	- array of numReg values read

Returns:
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:
mi_readRegister()
mi_readSensorRegisters()
mi_setMode()

3.10.8 mi_writeRegisters()

Definition:
mi_s32 mi_writeRegisters(mi_camera_t *pCamera, mi_u32 shipAddr, mi_u32 startAddr, mi_u32 numRegs, mi_u32 vals[])

Summary:
This function writes a set of consecutive "numRegs" register values starting at startAddr. When writing registers to a sensor the preferred method is mi_writeSensorRegisters() which allows for register caching and automatic setting of the address space register. Note that the USB firmware can currently only do 64 sequential writes, if more than 64 are requested then the call will be broken down into single write requests. Use mi_setMode() to set the register address width and register data width.

Parameters:

mi_camera_t*	pCamera	- pointer to the camera structure
mi_u32	shipAddr	- SHIP base address
mi_u32	startAddr	- first register address to write to
mi_u32	numRegs	- number of consecutive registers to write
mi_u32	val[]	- array of numReg values to write

Returns:
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:
mi_writeRegister()
mi_writeSensorRegisters()
mi_setMode()

3.10.9 mi_grabFrame()

Definition:
mi_s32 mi_grabFrame(mi_camera_t *pCamera, mi_u8 *pInBuffer, mi_u32 bufferSize)

Summary:
Reads an entire image and returns it in pInBuffer. The bufferSize is the size of the allocated buffer (pInBuffer) and should be at least the size of the sensor->bufferSize. Note that only a single frame is read. If there is a failure, it is the responsibility of the application to retry and grab the frame.

For Bayer images, mi_grabFrame() will grab either 8-bit, 10-bit data 12 bit data depending on the particular sensor and whether mi_updateFrameSize() has been called to change the default bits per pixel.

For 12bit images the data will be stored as

```
Byte 1      Byte 0      (Final 12 bit data ordering)
xxxxBA98 76543210
```

where bit B is the most significant bit. This allows the user to read the data in as 16 bit data.
For 10bit images the data will be stored as

```
Byte 1      Byte 0      (Final 10 bit data ordering)
xxxxxx98 76543210
```

where bit 9 is the most significant bit. This allows the user to read the data in as 16 bit data.
For 8 bit Bayer data we use the 8 most significant bits:

```
Byte 0
98765432
```

The way the data is wired from the sensor to the camera is such that the 10bit data is originally ordered as

```
Byte 1      Byte 0
xxxxxx10 98765432
```

and 12-bit data originally ordered as

```
Byte 1      Byte 0
xxxx1032 BA987654
```

On a Demo2 board the hardware will reorder the data so that it is in the “Final data ordering”. On Demo1 and Demo1A board this data reordering is done in software. By default mi_grabFrame() will reorder (unswizzle) the data, unless you turn off the unswizzling option using mi_setMode(pCamera, MI_UNSWIZZLE_MODE, 0) and then the application is responsible for unswizzling the data into the “Final data order”. Please see section 8.2 for more information about data swizzling and why the application may want to do this.

If mi_OpenCameras() was called with a “” for the sensor_dir_file parameter, then the user must call mi_updateFrameSize() before calling mi_grabFrame() in order to initialize the mi_sensor_t structure. If not, the user will get the MI_SENSOR_NOT_INITIALIZED error message.

If mi_OpenCameras() was called with an image file or video file then mi_grabFrame() will return the data from the file. The sensor width, height and imageType will normally be set to the appropriate values automatically based on the type of image in the file, but if the image file is raw data with no header then it may be necessary to call mi_updateFrameSize() and set pCamera->sensor->imageType. If the file is compressed (like JPEG or PNG) then mi_grabFrame() will return the compressed data and the application must decompress it. mi_grabFrame() can turn RGB data into Bayer, and can do simple RGB->RGB conversion like convert 24bpp RGB to 32bpp RGB.

Please refer to the function mi_getFrameData() which is used in conjunction with mi_grabFrame() to get additional frameData.

Parameters:

mi_camera_t	*pCamera	- pointer to the camera structure
mi_u8	*pInBuffer	- buffer to return image
mi_u32	bufferSize	- size of the buffer being used, used to verify buffer size

Returns:
MI_CAMERA_SUCCESS for success
MI_CAMERA_ERROR, MI_GRAB_FRAME_ERROR, MI_NOT_ENOUGH_DATA_ERROR, MI_EOF_MARKER_ERROR or MI_SENSOR_NOT_INITIALIZED for failure

See also:
mi_getFrameData() mi_setMode()
mi_updateFrameSize() mi_getMode()
mi_grabFrame() returns error code 4 (Section 8.2) mi_OpenCameras()

3.10.10 mi_getFrameData()

Definition:
mi_s32 mi_getFrameData(mi_camera_t* pCamera, mi_frame_data_t *frameData)

Summary:
This function returns the frame data associated with the last frame grabbed. The frame data currently consists of the frame number, the number of bytes of data requested and the number of bytes returned in mi_grabFrame(). This structure (mi_frame_data_t) is expected to expand over time and may contain different information depending on the hardware. It is mostly used for debugging purposes and will probably not be used by most applications. Not all transports implement this function.

Parameters:

mi_camera_t*	pCamera	- pointer to the camera structure
mi_frame_data_t*	frameData	- frame data returned from last frame grabbed

Returns:
MI_CAMERA_SUCCESS for success
MI_CAMERA_ERROR for failure

See also:
mi_writeRegister()
mi_writeSensorRegisters()
mi_setMode()
mi_grabFrame()
mi_frame_data_t

3.10.11 mi_updateFrameSize()

Definition:
mi_s32 mi_updateFrameSize(mi_camera_t *pCamera, mi_u32 width, mi_u32 height, mi_s32 nBitsPerClock, mi_s32 nClocksPerPixel)

Summary:
This routine allows the user to update the image data size and format after initialization. A new width and height may be supplied as well as the number of bits per clock and/or clocks per pixel. Changing the width and height may be necessary if the rowsize or colsize registers are

modified. The nBitsPerClock can be changed if the user wishes to only use 8bits instead of 10bits per clock to improve performance. The number of clocks per pixel might change if an SOC is running in bypass IP mode. This routine will update the sensor pixelBits, pixelBytes and bufferSize if necessary. Note that if the bufferSize has increased, then the buffer for mi_grabFrame() may need to be reallocated. If there is no new value for one of the parameters a value of zero may be entered such that the value stays the same. This may be useful if the user doesn't know the value of nClocksPerPixel, for example.

This routine is also used to initialize the pCamera->sensor structure if the user did not supply a sensor data file when calling mi_OpenCameras().

Parameters:

mi_camera_t	*pCamera	- pointer to the camera structure
mi_u32	width	- new width value
mi_u32	height	- new height value
mi_s32	nBitsPerClock	- Number of bits to send per clock
mi_s32	nClocksPerPixel	- number of clocks per pixel

Returns:

This function will alter the value of pCamera->sensor->bufferSize.
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:

mi_writeRegister()
mi_writeSensorRegisters()
mi_setMode()
mi_grabFrame()
mi_OpenCameras()
mi_updateBufferSize()

3.10.12 mi_updateBufferSize()

Definition:

mi_s32 mi_updateBufferSize(mi_camera_t *pCamera, mi_u32 rawBufferSize)

Summary:

This routine is similar to mi_updateFrameSize(). It allows the user to update the image data size (sensor->bufferSize) by providing the rawBufferSize. This can be used when the rawBufferSize is not a simple width*height*bpp calculation. JPEG frames are an example of this. The rawBufferSize may not be the same as sensor->bufferSize because the transport might require a particular memory alignment and/or padding. In most cases the application should use mi_updateFrameSize() instead of this routine. Not all transports implement this function.

Parameters:

mi_camera_t	*pCamera	- pointer to the camera structure
mi_u32	rawBufferSize	- The bufferSize required to hold the image data

Returns:

This function will alter the value of pCamera->sensor->bufferSize.
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:
mi_writeRegister()
mi_writeSensorRegisters()
mi_setMode()
mi_grabFrame()
mi_updateFrameSize()

3.10.13 mi_setMode()

Definition:
mi_s32 mi_setMode (mi_camera_t *pCamera, mi_modes mode, mi_u32 val);

Summary:
This routine is used to set several internal modes. The current modes supported are:

MI_ERROR_CHECK_MODE
Turn off or on marker error checking (MI_EOF_MARKER_ERROR) in mi_grabFrame() routine.
Possible values:

0	Do not perform marker error checking
1	(default) Perform marker error checking

MI_REG_ADDR_SIZE
Set the register address size for mi_readRegister(), mi_writeRegister(), mi_readRegisters() and mi_writeRegisters().
Possible values:

0	do not send register addresses
8	(default) send 8 bit register addresses
16	send 16 bit register address
32	send 32-bit register addresses

MI_REG_DATA_SIZE
Set the register size for mi_readRegister(), mi_writeRegister(), mi_readRegisters() and mi_writeRegisters()
Possible values:

8	perform 8 bit register reads/writes
16	(default) perform 16 bit register reads/writes
32	perform 32-bit register reads/writes

MI_USE_REG_CACHE
Allow mi_readSensorRegisters() to use cached sensor register values instead of re-reading the value from the sensor. This is a performance optimization and the application must determine the volatility of the register being requested for correct behavior.
Possible values:

0	(default) Do not perform register caching, always do hardware read
1	Perform register caching

MI_SIMUL_REG_FRAMEGRAB
(Read only.) Returns whether or not the driver can support a pair of threads simultaneously accessing registers and grabbing images. If the value is 0 and your application is multi-threaded,

then your application must ensure that a frame grab (mi_grabFrame()) is not initiated while another thread is accessing registers, or else you may experience I/O errors or stalls.

MI_SW_UNSWIZZLE_MODE
It is used to set whether midlib should unswizzle the 10-bit, 12-bit, 14-bit or 16-bit data in software.

See mi_setMode()
mi_grabFrame() for more information about data format. Data unswizzling is only necessary for boards that send data in the wrong order (MI_DEMO_1 and MI_DEMO1_A). This mode was added because doing the unswizzle in mi_grabFrame() was causing frames to be dropped, allowing the application to unswizzle the data solved this performance issue.

Possible values:

0	mi_grabFrame() should not unswizzle data
1	(default for Demo1/Demo1A) mi_grabFrame should unswizzle data

MI_UNSWIZZLE_MODE
This is a more general method for setting the unswizzle mode and supercedes MI_SW_UNSWIZZLE_MODE. It gives the user more control by allowing them to unswizzle data that is greater than 8 bits in hardware (MI_HW_UNSWIZZLE) or in software (MI_SW_UNSWIZZLE). For more information on data swizzling see mi_setMode() mi_grabFrame().

Possible values:

MI_NO_UNSWIZZLE	turn off any unswizzling
MI_ANY_UNSWIZZLE	do HW unswizzling if available, otherwise do SW unswizzle
MI_HW_UNSWIZZLE	do HW unswizzle if available, otherwise to no unswizzle
MI_SW_UNSWIZZLE	do SW unswizzle regardless of whether HW can do it

MI_SPOOF_SIZE
Tell the transport what the JPEG spoof image dimensions are, for the JPEG spoof mode on SOC sensors supporting JPEG. Some transports need the spoof dimensions in order to program the DMA on the device. The value holds the width and height like so: val = (height << 16) + width.

MI_HW_BUFFERING
Sets the number of frame buffers on the camera, for camera devices that support frame buffering such as DEMO2.

MI_OUTPUT_CLOCK_FREQ
Set the frequency of the XMCLK signal on the camera board. Normally this clock is, or can be, jumpered to the external clock input on the sensor. Set the value in Hz.

MI_PIXCLK_POLARITY
Sets whether the camera samples data from the sensor on the rising or falling edge of PIXCLK. This setting will override the setting the sensor data file.

Possible values:

0	Falling edge
1	Rising edge

MI_PIXCLK_FREQ

(Read only.) Measure the input PIXCLK frequency, if the device supports this measurement. DEMO2 and DEMO2X support it. Returns the value in Hz.

MI_DETECT_FRAME_SIZE
(Read only.) Measure the incoming image width and height in pixels, if the device supports this measurement. DEMO2 and DEMO2X support it. For the width it really measures the number of pixel clocks and divides by the current clocksPerPixel. The return value has the image width in the low 16 bits and the height in the high 16 bits.

MI_ALLOW_FAR_ACCESS
For companion chips or SOC sensors that can have external sensors connected to a secondary SHiP bus, this setting allows the host to access registers on the secondary ("far") device. Normally, accessing registers on the far device will interfere with the firmware on the main device, but if the firmware is disabled, such access will work.

Possible values:

0	(default) Midlib will ignore attempts to read or write far registers. Reads will return 0, writes will do nothing.
1	Accesses to far registers will go through.

MI_SENSOR_POWER
Turns the power to the sensor on and off, if the camera mainboard supports this. Supported on DEMO2X but not DEMO2.

MI_SENSOR_RESET
Sets the state of the sensor RESET pin. 1 is asserted.

MI_SENSOR_SHUTDOWN
Sets the state of the sensor RESET pin. 1 is asserted.

MI_SHIP_SPEED
Sets the bit rate of the SHiP bus in kHz. Possible values are 100 and 400.

MI_DIRECT_VAR_ACCESS
Recent SOC sensors can map the firmware variables into the register address space starting at 0x8000. This mode sets whether midlib should access variables by this direct addressing scheme, or indirectly through the XDMA registers.

MI_XDMA_LOGICAL
On SOC sensors with variable direct access, this tracks the current state of the device's XDMA logical or physical addressing mode.

MI_XDMA_PHY_A15
On SOC sensors with variable direct access, this tracks the current state of the device's XDMA physical address bit 15.

MI_XDMA_PHY_REGION
On SOC sensors with variable direct access, this tracks which memory region is currently mapped to the register address space.

MI_XDMA_ADV_BASE

On ICP1, this tracks the current state of the ADVANCED_BASE register which is used for addressing RAM in this device.

MI_HW_FRAME_COUNT
Reads the frame count register on the camera mainboard on cameras that support it.

MI_HIDY
Set by the application when the device is in high dynamic range mode. Midlib will ensure that the imageType is one of the high dynamic range types.

MI_COMPRESSED_LENGTH
If the sensor data file is really an image file in compressed format such as JPEG or PNG, this mode returns the length of the compressed image data.

MI_BITS_PER_CLOCK
Read-only, the current number of bits captured per pixel clock on the main camera board parallel input. Set through mi_updateFrameSize().

MI_CLOCKS_PER_PIXEL,
Read-only, the current number of pixel clocks per pixel on the main camera board parallel input. Set through mi_updateFrameSize().

MI_RX_TYPE
Set the type of image data interface being used by the sensor. Programs the serial receiver, if any, for the type of interface.
Possible values:
MI_RX_DISABLED
MI_RX_PARALLEL
MI_RX_CCP
MI_RX_MIPI
MI_RX_HISPI

MI_RX_LANES (MIPI and HiSPi only)
MI_RX_BIT_DEPTH (CCP and HiSPi only)
MI_RX_MODE (HiSPi only)
MI_RX_CLASS (CCP only)
MI_RX_SYNC_CODE (HiSPi only)
MI_RX_EMBEDDED_DATA
MI_RX_VIRTUAL_CHANNEL (MIPI only)
MI_RX_MSB_FIRST (HiSPi only)
These are for setting the corresponding modes for the serial receiver board, for devices using CCP, MIPI or HiSPi.

MI_HDMI_MODE
Sets the mode for the HDMI output on the HDMI Demo board.

MI_EIS
Enables electronic image stabilization feature on the Demo board , if supported.

MI_MEM_CAPTURE

For use on demo boards with memory that can be used to capture short video clips. The parameter is the number of frames to capture. When this mode is set to a non-zero value, the board will begin accumulating the number of frames specified. After setting it, use mi_getMode() to read it back. The result may be less than the number of frames requested depending on the amount of available memory. Use mi_grabFrame() to retrieve the frames. If the parameter is set to zero the capture is cancelled. It will return to zero automatically after the last frame is read out through mi_grabFrame().

MI_MEM_CAPTURE_PROGRESS
Read-only. While a video clip capture is accumulating frames, this will return the number of frames captured so far. When the capture is complete it will be equal to MI_MEM_CAPTURE. It will return to 0 after the last frame is read out through mi_grabFrame().

MI_MEM_CAPTURE_MB
Read-only. How many megabytes (2^{20} bytes) of memory is available for video clip capture.

Parameters:

mi_camera_t	*pCamera	- pointer to the camera structure
mi_modes	mode	- mode to set
mi_u32	val	- new value of mode

Returns:
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:
mi_getMode()
mi_modes
mi_unswizzle_modes

3.10.14 mi_getMode()

Definition:
mi_s32 getMode (mi_camera_t *pCamera, mi_modes mode, mi_u32* val);

Summary:
This routine is used to get the internal mode values. See mi_setMode() for more information about the modes.

Parameters:

mi_camera_t	*pCamera	- pointer to the camera structure
mi_modes	mode	- mode to set
mi_u32*	val	- current value of mode

Returns:
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:

mi_setMode()
mi_unswizzle_modes
mi_modes

3.10.15 mi_initTransport()

Definition:
mi_s32 mi_initTransport(mi_camera_t *pCamera, mi_u8 bitsPerClock, mi_u8 clocksPerPixel, mi_u8 polarity, mi_u8 pixelOffset, mi_u8 noFWCalls);

Summary:
This routine should only be used to initialize the transport information when a sensor data file was not used during the call to mi_OpenCameras(). In normal operation (using a sensor data file) this information is filled in automatically. This routine cannot be called until after mi_startTransport() is called and is only needed if the user will be calling mi_grabFrame(). Not all transports implement this function.

Parameters:

mi_camera_t	*pCamera	- pointer to the camera structure
mi_s32	nBitsPerClock	- (default 8) number of bits per clock
mi_s32	nClocksPerPixel	- (default 1)number of clocks per pixel
mi_s32	nPolarity	- (default 1)pixel clock polarity; which pixclk edge to grab data, this is only relevant for Demo1 and Demo1A
mi_s32	nPixelOffset	- (default 0) buffer offset (used for Demo1 workaround)
mi_s32	noFWCalls	- 0=Allows firmware calls, 1=disables firmware calls, set to 1 for debugging purposes only

Returns:
MI_CAMERA_SUCCESS for success or MI_CAMERA_ERROR if failure

See also:
mi_OpenCameras()
mi_startTransport()

3.11 Enums

3.11.1 mi_error_code

Definition:

```
typedef enum
  {
    //Generic return codes
    MI_CAMERA_SUCCESS              = 0x00,
    MI_CAMERA_ERROR                = 0x01,
    //Grabframe return codes
    MI_GRAB_FRAME_ERROR            = 0x03,
    MI_NOT_ENOUGH_DATA_ERROR       = 0x04,
```

```
    MI_EOF_MARKER_ERROR             = 0x05,
    MI_BUFFER_SIZE_ERROR            = 0x06,
    //mi_OpenCameras return codes
    MI_SENSOR_FILE_PARSE_ERROR      = 0x07,
    MI_SENSOR_DOES_NOT_MATCH        = 0x08,
    MI_SENSOR_NOT_INITIALIZED       = 0x09,
    MI_SENSOR_NOT_SUPPORTED         = 0x0A,
    //I2C return codes
    MI_I2C_BIT_ERROR                = 0x0B,
    MI_I2C_NACK_ERROR               = 0x0C,
    MI_I2C_TIMEOUT                  = 0x0D,
    MI_CAMERA_TIMEOUT               = 0x0E,

    MI_CAMERA_NOT_SUPPORTED         = 0x10,
    //return codes for parsing sensor data files
    MI_PARSE_SUCCESS                = 0x20,
    MI_DUPLICATE_DESC_ERROR         = 0x21,
    MI_PARSE_FILE_ERROR             = 0x22,
    MI_PARSE_REG_ERROR              = 0x23,
    MI_UKNOWN_SECTION_ERROR         = 0x24,
    MI_CHIP_DESC_ERROR              = 0x25,
    MI_PARSE_ADDR_SPACE_ERROR       = 0x26,
   //Error codes for loading INI presets
    MI_INI_SUCCESS                  = 0x100,
    MI_INI_KEY_NOT_SUPPORTED        = 0x101,
    MI_INI_LOAD_ERROR               = 0x102,
    MI_INI_POLLREG_TIMEOUT          = 0x103,
  } mi_error_code;
```

Summary:

The following are all of the possible error/return codes for all of the midlib routines.

```
    //Generic return codes
    MI_CAMERA_SUCCESS               success value for midlib routines
    MI_CAMERA_ERROR                 general failure for midlib routines
    MI_CAMERA_NOT_SUPPORTED         The function call is not supported

    //Grabframe return codes
    MI_GRAB_FRAME_ERROR             general failure for grab frame routine
    MI_NOT_ENOUGH_DATA_ERROR        grab frame failed to return enough data
    MI_EOF_MARKER_ERROR             EOF packet not found in grab frame data
    MI_BUFFER_SIZE_ERROR            GrabFrame buffer is too small

    //mi_OpenCameras return codes
    MI_SENSOR_FILE_PARSE_ERROR      There was an error parsing the sensor data file
    MI_SENSOR_DOES_NOT_MATCH        Cannot find sensor data file which matches sensor
    MI_SENSOR_NOT_INITIALIZED       The sensor structure has not been initialized
                                                              (call updateFrame)
    MI_SENSOR_NOT_SUPPORTED         The sensor is no longer supported
```

```
//I2C return codes
MI_I2C_BIT_ERROR                I2C bit error
MI_I2C_NACK_ERROR               I2C NAC error
MI_I2C_TIMEOUT                  I2C time out error
MI_CAMERA_TIMEOUT

//return codes for parsing sensor data files
MI_PARSE_SUCCESS                Parsing was successful
MI_DUPLICATE_DESC_ERROR         Duplicate unique descriptor was found
MI_PARSE_FILE_ERROR             Unable to open sensor data file
MI_PARSE_REG_ERROR              Error parsing the register descriptors
MI_UKNOWN_SECTION_ERROR         Unknown Section found in sensor data file
MI_CHIP_DESC_ERROR              Error parsing the chip descriptor section
MI_PARSE_ADDR_SPACE_ERROR       Error parsing the address space section

//Error codes for loading INI presets
MI_INI_SUCCESS                  INI Preset is loaded successfully
MI_INI_KEY_NOT_SUPPORTED        Key is not supported - will be ignored
MI_INI_LOAD_ERROR               Error loading INI preset
MI_INI_POLLREG_TIMEOUT          Timeout in POLLREG command
```

3.11.2 mi_image_types

Definition:

typedef enum mi_image_types { ... };

Summary:

These are the supported image types.

```
MI_UNKNOWN_IMAGE_TYPE  - used for an unknown image type
MI_BAYER_8             - used to describe an 8 bit BAYER image
MI_BAYER_10            - used to describe a 10 bit BAYER image
MI_BAYER_8_ZOOM        - used to describe an 8 bit BAYER image in zoom mode
MI_BAYER_10_ZOOM       - used to describe an 10 bit BAYER image in zoom mode
MI_YCBCR               - a YCBCR image as used by MI SOC Sensors
MI_RGB565              - this is an output format for an MI SOC Sensor
MI_RGB555              - this is an output format for an MI SOC Sensor
MI_RGB444X             - this is an output format for an MI SOC Sensor
MI_RGBX444             - this is an output format for an MI SOC Sensor
MI_RGB24               - this is an output display image format
MI_RGB32               - this is an output display image format
MI_BAYER_12            - used to describe a 12 bit BAYER image
MI_BAYER_S12           - internal format for Signed 12bit BAYER
MI_RGB48               - this is an output display image format
MI_JPEG                - this is an output format for an MI SOC Sensor
MI_BAYER_STEREO        - this is an image type used by stereo demo board
MI_PNG                 - PNG compressed data
MI_BGRG                - Like YCbCr but with Y->G Cb->B and Cr->R
MI_YUV420              - Like YCbCr, but even (numbering from 0) rows are Y-only
```

MI_BAYER_14 - used to describe a 14 bit BAYER image
MI_BAYER_12_HDR - compressed HiDy on A-1000ERS
MI_BAYER_14_HDR - compressed HiDy on A-1000ERS
MI_BAYER_20 - used to describe a 20 bit BAYER image
MI_RGB332 - this is an output format for an MI SOC Sensor
MI_M420 - Like YUV420, Ys and Cb/Cr are not interleaved
MI_BAYER_10_IHDR - Interlaced HiDy on A-8150, etc.
MI_JPEG_SPEEDTAGS - JPEG with Scalado SpeedTags embedded

3.11.3 mi_product_ids

Definition:

```
typedef enum mi_product_ids {
        MI_UNKNOWN_PRODUCT,
        MI_DEMO_1    = 0x1003,
        MI_DEMO_1A   = 0x1004,
        MI_DEMO_2    = 0x1007,
        MI_DEMO_2X   = 0x100D};
```

Summary:

These are the product identifiers (PIDs) for all supported products.

MI_UNKNOWN_PRODUCT - used to describe an unknown product
MI_DEMO_1 - Demo_1 Product ID (USB2 camera)
MI_DEMO_1A - Demo_1A Product ID (USB2 camera w/ timing modification)
MI_DEMO_2 - Demo_2 Product ID (USB2 camera w/ 16MB Memory)
MI_DEMO_2X - Demo_2X Product ID (USB2 camera w/ 64MB Memory)
MI_PCCAM - Miniature PC demo camera Product ID

3.11.4 mi_sensor_types

Definition:
typedef enum mi_sensor_types { ... };

MI_UNKNOWN_SENSOR
MI_0360
MI_0366
MI_1300
MI_1310_SOC
MI_2000
MI_0360_SOC
MI_1310
MI_3100
MI_0350
MI_0366_SOC
MI_2010
MI_2010_SOCJ
MI_0370
MI_1320_SOC
MI_1320
MI_5100
MI_2020_SOC
MI_0354_SOC
MI_3120
MI_2020
MI_0350_ST
MI_8130
MI_1325
MI_BITMAP_SENSOR
MI_0380_SOC
MI_1330_SOC
MI_1600
MI_0380
MI_3130
MI_5130
MI_RAINBOW2
MI_3130_SOC
MI_2030_SOC
MI_3125
MI_0351
MI_9130
MI_5135
MI_2025
MI_0351_ST
MI_0356_SOC
MI_5140_SOC
MI_3132_SOC
MI_10030

MI_5131
MI_8131
MI_1000GS
MI_1000ERS

Summary:
These are the currently supported sensors; additional types included are for older sensors or experimental parts. MI_BITMAP_SENSOR is used when the “sensor” is really an image file or video file. Pre-production or new production sensors may not be listed here. If you don’t see the sensor you are using then contact your Aptina application engineer for the correct symbol.

3.11.5 mi_addr_type

Definition:
typedef enum mi_addr_type {
…
};

Summary:
These are possible address space or page types. Note that these are not supported on all sensors.

MI_REG_ADDR	Register address
MI_MCU_ADDR	MCU variable (MCU logical addressing)
MI_SFR_ADDR	Special Function Register (MCU absolute addressing)
MI_IND_ADDR	Indirect address
MI_FAR1_REG_ADDR	Registers on 1st sensor on far bus
MI_FAR1_MCU_ADDR	MCU driver variable on 1st sensor on far bus
MI_FAR1_SFR_ADDR	SFR on 1st sensor on far bus
MI_FAR2_REG_ADDR	Registers on 2nd sensor on far bus
MI_FAR2_MCU_ADDR	MCU driver variable on 2nd sensor on far bus
MI_FAR2_SFR_ADDR	SFR on 2nd sensor on far bus

3.11.6 mi_data_types

Definition:
typedef enum mi_data_types { … };

Summary:
These are the register or bitfield data types used by the data type conversion functions.

```
MI_UNKNOWN_DATA_TYPE = -1,
MI_HEX          //  Unsigned integer with conversion to string in hex
MI_UNSIGNED     //  Unsigned integer with conversion to string in decimal
MI_UFIXED1      //  Unsigned fixed point with 1 fraction bit
MI_UFIXED2      //  Unsigned fixed point with 2 fraction bits
…               //  etc.
```

```
MI_UFIXED32     //  Unsigned fixed point with 32 fraction bits
MI_SIGNED       //  2's complement signed integer
MI_FIXED1       //  2's complement signed fixed point with 1 fraction bit
MI_FIXED2       //  2's complement signed fixed point with 2 fraction bits
…               //  etc.
MI_FIXED32      //  2's complement signed fixed point with 32 fraction bits
MI_SIGNMAG      //  sign-magnitude signed integer, MSB is sign
MI_FLOAT        //  32-bit IEEE-754 floating point
MI_FLOAT16      //  half-precision floating point with exponent in low bits
```

3.11.7 mi_modes

Definition:

```
typedef enum mi_modes {
    MI_ERROR_CHECK_MODE,
    MI_ REG_ADDR_SIZE,
    MI_ REG_DATA_SIZE,
    MI_USE_REG_CACHE,

    MI_SW_UNSWIZZLE_MODE,
    MI_UNSWIZZLE_MODE,
    MI_SW_UNSWIZZLE_DEFAULT,
    MI_DATA_IS_SWIZZLED,

    MI_ERRORLOG_TYPE,
    MI_SPOOF_SIZE,
    MI_HW_BUFFERING,
    MI_OUTPUT_CLOCK_FREQ,
    MI_ALLOW_FAR_ACCESS,
    MI_PIXCLK_POLARITY,
    MI_SENSOR_POWER,
    MI_SENSOR_RESET,
    MI_SHIP_SPEED,

    MI_DIRECT_VAR_ACCESS,
    MI_XDMA_LOGICAL,
    MI_XDMA_PHY_A15,
    MI_XDMA_PHY_REGION,

    MI_HW_FRAME_COUNT,

    MI_HIDY,
    MI_COMPRESSED_LENGTH,
    MI_SENSOR_SHUTDOWN,

    MI_XDMA_ADV_BASE,
    MI_PIXCLK_FREQ,
    MI_SIMUL_REG_FRAMEGRAB,
    MI_DETECT_FRAME_SIZE,
```

```
    MI_BITS_PER_CLOCK,
    MI_CLOCKS_PER_PIXEL,

    MI_RX_TYPE,
    MI_RX_LANES,
    MI_RX_BIT_DEPTH,
    MI_RX_MODE,
    MI_RX_CLASS,
    MI_RX_SYNC_CODE,
    MI_RX_EMBEDDED_DATA,
    MI_RX_VIRTUAL_CHANNEL,

    MI_HDMI_MODE,
    MI_EIS,

    MI_MEM_CAPTURE,
    MI_MEM_CAPTURE_PROGRESS,
    MI_MEM_CAPTURE_MB,
};
```

Summary:
These are the modes that can be used with the mi_setMode() and mi_getMode() routines. See mi_setMode() and mi_getMode() for more details.

See also:
mi_setMode()
mi_getMode()
mi_unswizzle_modes

3.11.8 mi_unswizzle_modes

Definition:

```
typedef enum mi_unswizzle_modes {
    MI_NO_UNSWIZZLE,
    MI_HW_UNSWIZZLE,
    MI_SW_UNSWIZZLE,
    MI_ANY_UNSWIZZLE};
```

Summary:
Possible unswizzle modes when using MI_UNSWIZZLE_MODE

MI_NO_UNSWIZZLE	turn off any unswizzling
MI_ANY_UNSWIZZLE	do HW unswizzling if available, otherwise do SW unswizzle
MI_HW_UNSWIZZLE	do HW unswizzle
MI_SW_UNSWIZZLE	do SW unswizzle regardless of whether HW can do it

See also:
mi_setMode()

mi_getMode()
mi_modes

3.11.9 mi_rx_types

Definition:
typedef enum { MI_RX_UNKNOWN = -1,
MI_RX_DISABLED = 0,
MI_RX_PARALLEL,
MI_RX_CCP,
MI_RX_MIPI,
MI_RX_HISPI, };

Summary:
Possible values of mode MI_RX_TYPE.

MI_RX_UNKNOWN	midlib did not detect a serial receiver board
MI_RX_DISABLED	the serial receiver board is disabled
MI_RX_PARALLEL	the serial receiver board is set to parallel input
MI_RX_CCP	the serial receiver board is set to CPP
MI_RX_MIPI	the serial receiver board is set to MIPI
MI_RX_HISPI	the serial receiver board is set to HiSPi

3.12 Defines

3.12.1 General Constants

MI_MAX_CAMERAS	(20) maximum number of cameras supported
MI_MAX_REGS	(400) maximum number of registers supported in mi_frame_data_t structure
MI_APTINA_VID	(0x20FB) Aptina Imaging Vendor ID
MI_MICRON_VID	(0x0634) Micron Imaging Vendor ID
MI_MAX_STRING	(256) Maximum Length of an mi_string type

3.12.2 Error and Log Types

The following are used to determine what type of messages the user wants to log. This value is passed into mi_OpenErrorLog()

MI_NO_ERROR_LOG	Error logging is turned off
MI_ERROR_SEVERE	Log Severe errors
MI ERROR_MINOR	Log Minor errors
MI_ALL_ERRORS	Logs all error messages (Severe and Minor)
MI_LOG	Logs General logging message
MI_LOG_SHIP	Log Serial I/O messages (SHIP)
MI_LOG_USB	Log USB transactions
MI_LOG_DEBUG	Log Debug messages
MI_ALL_LOGS	Log everything but errors
MI_ERROR_LOG_ALL	Log everything

3.12.3 Camera Transport types

The following types are defined, but are not all fully supported at this time. For the transportType parameter of mi_OpenCameras2().

MI_USB_BULK_TRANSPORT	Aptina USB devices. DEMO2, etc.
MI_CARDBUS_TRANSPORT	
MI_DLL_TRANSPORT	A plug-in transport
MI_CL_TRANSPORT	
MI_BMP_TRANSPORT	An image file
MI_AVI_TRANSPORT	An AVI video file
MI_RAW_TRANSPORT	A RAW video file
MI_NUL_TRANSPORT	A memory-only camera structure. Provides a way to parse a sensor data file without a physical device.
MI_ALL_TRANSPORTS	Only used as a parameter to mi_OpenCameras2().

After initialization the pCamera->transportType field will be initialized to one of these values.

3.12.4 Device Removal/Arrival Notification Flags

The following flags are returned by the callback function to alert the user that a device was removed or added. See FAQ
Which register read/write routines should I use?
In older versions of midlib the only register read/write routines were in the pCamera structure. For example pCamera->readSensorRegisters. With the addition of new address space types (MI_MCU_ADDR, MI_SFR_ADDR) with more recent sensors, higher level API calls were added.

The transport read/write register routines:
mi_readRegister and **mi_writeRegister** read and write one register at a time. This is the most basic request. It can be used for any type of SHIP or I2C register access, not just sensor registers.
mi_readRegisters and **mi_writeRegisters** can read and write to multiple sequential SHIP or I2C registers. If possible it will make the request in a single transaction in order to improve I/O performance.
mi_readSensorRegisters and **mi_writeSensorRegisters** are meant to be used for sensor registers only. Like mi_readRegisters and mi_writeRegisters, they will perform multiple sequential reads and writes. The address space is passed into this routine so that the address space register can be set to this value prior to reading or writing the register. These routines can only be used with MI_REG_ADDR sensor registers.
mi_readSensorRegisterList allows the user to read a non-sequential list of registers. It only has limited support and is not recommended for general use.

The “high level” read/write register functions:
mi_ReadSensorReg and **mi_WriteSensorReg** will read a sensor register given a pointer to the mi_reg_data_t structure. This can be used to read or write any type of sensor register (MCU variable, SFR register, multi-span register) as long it is defined in the sensor data file.

mi_ReadSensorRegStr and **mi_WriteSensorRegStr** are similar to the previous two function except that the Register name and bitfield name are passed into the function. This function can read any register defined in the sensor data file. Internally the routine will call the find functions to get the mi_reg_data_t structure and then call the above functions.
mi_ReadSensorRegAddr and **mi_WriteSensorRegAddr** are similar to the previous functions except that they require the user to pass in all of the relevant information, such as addrType and width. This function is only recommended to be used when the register is not defined in the sensor data file.

3.13 When to call the function mi_updateFrameSize(...)

mi_updateFrameSize() should be called whenever the output format of the sensor changes with respect to one or more of the following parameters: image width, height, bits per pixel, or number of clocks per pixel.

Midlib does not maintain the state of the sensor internally. This means that Midlib does not know about sensor output format changes that might occur because of a change that the host application has requested through a register settings operation.

It is the responsibility of the host application to maintain the state of the sensor, and to update Midlib whenever the frame output format changes.

If the amount of data that Midlib receives when mi_grabFrame() is called is smaller than the expected amount of data based these parameters the call will return MI_NOT_ENOUGH_DATA_ERROR.

Refer to the section mi_updateFrameSize() for more details about the use of this function.

3.14 Why does the field frameCounter equal 0?

The field frameNumber is part of the structure mi_frame_data_t which is returned by the call to mi_getFrameData().

The validity of this field depends on which camera hardware you are using. DEMO1 has a frame counter and would transmit the count in an additional USB packet after the image pixels. DEMO2 and DEMO2X do not have this capability, so frameNumber is always 0. DevWare's frame counter is not from the hardware; it is a variable incremented after grabFrame().

```
pCamera->grabFrame();
++frameCount;
```

Sensors supporting SMIA embedded data can embed some register data including the frame counter register into the first two rows of image data. Embedded data must be enabled on the sensor, usually register 0x3064 bit 8.

Example Code and mi_EnableDeviceNotification() and mi_DisableDeviceNotification

MI_DEVEVENT_REMOVAL	A device is being removed
MI_DEVEVENT_PRIMARY	A primary device (i.e. opened device) is being removed
MI_DEVEVENT_OTHER	A non-opened device is being removed
MI_DEVEVENT_ARRIVAL	currently not supported

4.0 PORTABILITY

The Aptina (Micron) Imaging Device Library (midlib) can be used by most programming languages and compilers. To use the library with C/C++ you must include the midlib2.h file. For Borland Delphi Pascal, use the midlib2.pas file. A library (midlib2.lib) module is included which can be used with MSVC projects and a Borland C++ Builder library (midlib2_omf.lib) is also included for Builder projects.

For step-by-step instructions on how to create a Visual Studio or Visual C++ project which uses midlib2 please refer to the document **Visual Studio Project Settings Guide.pdf**.

The midlib2.dll file is included for dynamic linking of the library. It is important to make sure that the midlib2.dll file which is loaded, matches the version of the library that the application has been built with. You should place the midlib2.dll file in the same directory as your application to make sure the correct version is found. You can also statically link the midlib library to ensure there are no mismatches. The SensorDemo and DevWare applications are statically linked.

To use Link-time dynamic loading (default) do the following:
Include the library module (midlib2.lib or midlib2_omf.lib) as part of the link stage in your project.
Call the functions as described in section 3.1.

In order for the structures to be aligned properly make sure your application project uses 8-byte alignment. This option can usually be found in the compiler options section for your integrated development environment. Also make sure that enums are treated as integers.

NOTE: midlib2.h should compile in your application without generating errors or warning messages. You may receive an error message regarding the Windows handle (HWND) in your console application or DLL. (A console application uses the DOS command line for input, it does not have a window in the Graphical User Interface). The error will be similar to the following and followed by several other errors:

> Error C2065: 'HWND' : undeclared identifier

If you receive this error message, please do the following:
You will not be able to use device removal/arrival notification in your application. Remove this from your application if you have enabled it (for information on device removal/arrival notification, see section 3.8 above).
In your project's properties/settings dialog box, remove the '_WINDOWS' entry from the C/C++, Preprocessor Definitions entry.
Recompile your application.

5.0 FAQ

5.1 Which register read/write routines should I use?

In older versions of midlib the only register read/write routines were in the pCamera structure. For example pCamera->readSensorRegisters. With the addition of new address space types (MI_MCU_ADDR, MI_SFR_ADDR) with more recent sensors, higher level API calls were added.

The transport read/write register routines:
mi_readRegister and **mi_writeRegister** read and write one register at a time. This is the most basic request. It can be used for any type of SHIP or I2C register access, not just sensor registers.
mi_readRegisters and **mi_writeRegisters** can read and write to multiple sequential SHIP or I2C registers. If possible it will make the request in a single transaction in order to improve I/O performance.
mi_readSensorRegisters and **mi_writeSensorRegisters** are meant to be used for sensor registers only. Like mi_readRegisters and mi_writeRegisters, they will perform multiple sequential reads and writes. The address space is passed into this routine so that the address space register can be set to this value prior to reading or writing the register. These routines can only be used with MI_REG_ADDR sensor registers.
mi_readSensorRegisterList allows the user to read a non-sequential list of registers. It only has limited support and is not recommended for general use.

The “high level” read/write register functions:
mi_ReadSensorReg and **mi_WriteSensorReg** will read a sensor register given a pointer to the mi_reg_data_t structure. This can be used to read or write any type of sensor register (MCU variable, SFR register, multi-span register) as long it is defined in the sensor data file.
mi_ReadSensorRegStr and **mi_WriteSensorRegStr** are similar to the previous two function except that the Register name and bitfield name are passed into the function. This function can read any register defined in the sensor data file. Internally the routine will call the find functions to get the mi_reg_data_t structure and then call the above functions.
mi_ReadSensorRegAddr and **mi_WriteSensorRegAddr** are similar to the previous functions except that they require the user to pass in all of the relevant information, such as addrType and width. This function is only recommended to be used when the register is not defined in the sensor data file.

5.2 When to call the function mi_updateFrameSize(...)

mi_updateFrameSize() should be called whenever the output format of the sensor changes with respect to one or more of the following parameters: image width, height, bits per pixel, or number of clocks per pixel.

Midlib does not maintain the state of the sensor internally. This means that Midlib does not know about sensor output format changes that might occur because of a change that the host application has requested through a register settings operation.

It is the responsibility of the host application to maintain the state of the sensor, and to update Midlib whenever the frame output format changes.

If the amount of data that Midlib receives when mi_grabFrame() is called is smaller than the expected amount of data based these parameters the call will return MI_NOT_ENOUGH_DATA_ERROR.

Refer to the section mi_updateFrameSize() for more details about the use of this function.

5.3 Why does the field frameCounter equal 0?

The field frameNumber is part of the structure mi_frame_data_t which is returned by the call to mi_getFrameData().

The validity of this field depends on which camera hardware you are using. DEMO1 has a frame counter and would transmit the count in an additional USB packet after the image pixels. DEMO2 and DEMO2X do not have this capability, so frameNumber is always 0. DevWare's frame counter is not from the hardware; it is a variable incremented after grabFrame().

```
pCamera->grabFrame();
++frameCount;
```

Sensors supporting SMIA embedded data can embed some register data including the frame counter register into the first two rows of image data. Embedded data must be enabled on the sensor, usually register 0x3064 bit 8.

6.0 EXAMPLE CODE

The following example demonstrates how to initialize the midlib interface. After initialization, it shows how to read registers from the attached camera. A second example shows how to enable USB device removal notification.
Please note that there are additional source code examples in the “C:\Aptina Imaging\samples” directory.

```
include "c:\aptina imaging\include\midlib2.h"

mi_camera_t     *cameras[MI_MAX_CAMERAS];//cameras found
mi_s32          num_cameras;            //number of cameras found
mi_camera_t     *camera= NULL;          //current camera
int             chipId;                 //register number for chipID
mi_reg_data_t   *chip_id_reg;           //pointer to the chipid reg
mi_reg_data_t   *reg;                   //pointer to a register
mi_u8           *pCameraBuff = NULL;    //camera image buffer
mi_u32          val;                    //register value returned
int             new_val;                //register value to write
int             error;                  //error return value
mi_s32          retVal;                 //Return value

//mi_OpenCameras() is the first midlib routine which must be called
//The third parameter is the directory containing the sensor data files
//The environment variable MI_HOME is set at installation, the default
//location is C:\Aptina Imaging

retVal = mi_OpenCameras(cameras,&num_cameras, mi_SensorData());

//we need at least one camera to use the sensor
if (num_cameras == 0) {
        if (retVal == MI_SENSOR_ERROR)
                Error("Could not find sensor data file.");
                //Could prompt user for a particular filename and try again
        else
                Error("Could not connect to camera.");
        mi_CloseCameras();
        exit(0);
}

//Let the user choose the camera to use, we will use the
//first camera found in this example
camera = cameras[0];

//Allocate a buffer to store the images, use the size that is in
//sensor->bufferSize
pCameraBuff  = (mi_u8*)malloc(camera->sensor->bufferSize);
```

```
//We must start the camera before we can grab an image
mi_startTransport(camera);

//An example of reading a register by name
mi_ReadSensorRegStr(camera, “CHIP_VERSION_REG”, NULL, &val);
//Examples of reading and writing registers by address. In this case we are
// reading from register address 0 and writing to register address 0x2B
mi_ReadSensorRegAddr(camera, MI_REG_ADDR, 0, 0, 1, &val);
mi_WriteSensorRegAddr(camera, MI_REG_ADDR, 0, 0x2B, 1, 0x10);

//Continue to grabFrames until user prompts to stop
while (!done) {
        //Only draw the “good” frames
        if (mi_grabFrame(camera, pCameraBuff, camera->sensor->bufferSize) ==
        MI_CAMERA_SUCCESS)
        draw_image(pCameraBuffer);
}

//stop the camera and clean up
mi_stopTransport(camera);
mi_CloseCameras();
free(pCameraBuff);


// The following code demonstrates USB device removal notification. First,
// define the function that will be called when the USB device is removed.
mi_u32 miDevCallBack (HWND hwnd, _mi_camera_t *pCamera, mi_u32 Flags) {
        if (Flags & MI_DEVEVENT_REMOVAL) {
                MessageBox(hwnd, TEXT("Device Removed"), TEXT("Device Status"), MB_OK);
        }
        return MI_CAMERA_SUCCESS;
}


// Next, enable USB device removal notification. The call back function
// is called on removal. NOTE: MIUSB2.SYS driver version 5.1.0.3505
// or higher must be used.
retVal = mi_EnableDeviceNotification(myhWnd, pCamera, &miDevCallBack);
if (retVal == MI_CAMERA_SUCCESS) {
   // Notification initialization was successfully enabled!
}

 . . .

// When no longer needed or when shutting down the camera, disable
// device notification.
retVal = mi_DisableDeviceNotification(myhWnd, pCamera);
```

7.0 SENSOR DATA FILE DEFINITION

Each sensor will have an associated sensor data file. The sensor data file is meant to be the "spec" for the sensor. These files were taken directly from the specifications provided for each sensor.

The sensor data file should adhere to the following format. Note that all of the fields in the chip descriptor section are required. The register section contains all registers for the sensor. Some registers are used internally in our supported applications and should not be removed or renamed or the application will be unable to function properly.

All numerical constants can be expressed in decimal or hex with 0x prefix, ex. 0x7B.

Some reserved register names are:
CHIP_VERSION_REG
RESET_REG
GREEN1_GAIN_REG
BLUE_GAIN_REG
RED_GAIN_REG
GREEN2_GAIN_REG
HORZ_BLANK_REG
INTEG_TIME_REG
VREF_REG

//Comments may be placed outside of sections
//This is the chip descriptor section, where each line has a "field = value"

[CHIP_DESCRIPTOR]
SENSORNAME = "string"
WIDTH = integer
HEIGHT = integer
IMAGE_TYPE = BAYER
BITS_PER_CLOCK = integer
CLOCKS_PER_PIXEL = integer
PIXEL_CLOCK_POLARITY = 0/1
SHIP_BASE_ADDRESS = integer *or* (integer integer …)
[END]

//The [ADDR_SPACE] section is required for sensors which have multiple
//address spaces (ie A-1310SOC). The ID "ID1" will be used in the register
//section for all of the registers that are part of the ID1 address
//space. The IDs can be any string, although common ones have been "SENSOR",
//"IP", "COLPIPE", etc. The type can be REG, MCU or SFR. If the type is not
//used, the default is REG. The integer is the value of the address space
//for that page.

[ADDR_SPACE]
ID1 = {type, integer, string}

```
ID2 = {type, integer, string}
ID3 = {type, integer, string}
[END]

//These are optional settings.
//PART_NUMBER is the Micron/Aptina MT9 number for the sensor
//EXTRA_DESC_REG is a workaround for sensors without a unique chip id
//SENSOR_VERSION specifies the REV version of the sensor
//VERSION_NAME overrides using REV as the version name, so for example ES1
//FULL_WIDTH and FULL_HEIGHT are used if sensors default width is not the
//    full image size
//SDAT_VERSION is used to describe the version number of the sensor data file –
//    current value is 2
//REG_ADDR_SIZE is the size of the register address (default is 8)
//REG_DATA_SIZE is the size of the register data (default is 16)

[OPTIONAL]
PART_NUMBER = string
EXTRA_DESC_REG = register
SENSOR_VERSION = integer
VERSION_NAME = string
FULL_WIDTH = integer
FULL_HEIGHT = integer
SDAT_VERSION = integer
REG_ADDR_SIZE = integer
REG_DATA_SIZE = integer
[END]

//This is the register section
//The register definitions supply the following information, the
//bitfield information is optional

//REGDEF1 = {ADDR, TYPE, SPAN, MASK, RW, DEFAULT, DESC, DETAIL}
//  {BITDEF1,              MASK, RW,        DESC, DETAIL}
//  {BITDEF2,              MASK, RW,        DESC, DETAIL}
//  ...
//  {BITDEF3,              MASK, RW,        DESC, DETAIL}

[REGISTERS]
   REGDEF1 = {integer, CORE/IP, integer, integer, RW/RO, integer, "string", "string", "string"}
     {BITDEF1, integer, RW/RO, "string", "string"}
     {BITDEF2, integer, RW/RO, "string", "string"}
    ...
     {BITDEF3, integer, RW/RO, "string", "string"}
   REGDEF2 = {integer, CORE/IP, integer, RW/RO, integer, "string", "string", "string"}
 [END]
```

7.1 Sample Sensor Data File

This is an example of a partial sensor data file. Note that the register section includes register descriptions (i.e., CHIP_VERSION_REG ROW_WINDOW_START_REG) as well as bitfield descriptions (APERTURE_GAIN_VALUE, APERTURE_GAIN_AUTO) for the APERTURE_GAIN register.
The registers cover all three address spaces listed: SENSOR, COLPIPE and CAMCTRL.

```
[CHIP_DESCRIPTOR]
SENSORNAME = "A-1310SOC"
WIDTH = 640
HEIGHT = 512
IMAGE_TYPE = YCBCR
BITS_PER_CLOCK = 8
CLOCKS_PER_PIXEL = 2
PIXEL_CLOCK_POLARITY = 1
SHIP_BASE_ADDRESS = 0xBA
[END]

[ADDR_SPACE]
SENSOR  = {0, "0: Sensor Core"}
COLPIPE = {1, "1: Color Pipe"}
CAMCTRL = {2, "2: Camera Control"}
[END]

[OPTIONAL]
PART_NUMBER = "MT9M111"
SENSOR_VERSION = 1
VERSION_NAME = "REV1""
FULL_WIDTH = 1280
FULL_HEIGHT = 1024
[END]

[REGISTERS]
CHIP_VERSION_REG =     {0x00, SENSOR, 0xFFFF, RO, 0x1419, "Chip Version", ""}
ROW_WINDOW_START_REG = {0x01, SENSOR, 0x07FF, RW, 0x000C, "Row Start", ""}
COL_WINDOW_START_REG = {0x02, SENSOR, 0x07FF, RW, 0x001C, "Column Start", ""}
APERTURE_GAIN =        {0x05, COLPIPE,0x000F, RW, 0x0003, "Aperture", ""}
   {APERTURE_GAIN_VALUE,            0x0007, RW,"0-2: Sharpening", ""}
   {APERTURE_GAIN_AUTO,             0x0008, RW,"3: Auto Sharpening", ""}
BASE_MATRIX_SIGNS =    {0x02, CAMCTRL,0x01FF, RW, 0x006E, "Signs", ""}
[END]
```

8.0 APPLICATION TROUBLESHOOTING

8.1 Application is using the wrong midlib2.dll

Make sure that you copy the midlib2.dll (found in C:\Aptina Imaging) into the same directory as your executable. A mismatch between the midlib2.lib you link with and the midlib2.dll can show up in many different ways. Also make sure that you recompile your application anytime you receive a new software installation and that the midlib2.h and the midlib2.lib you use also come from the same software release.

8.2 mi_grabFrame() returns error code 4

Error code 4 is MI_NOT_ENOUGH_DATA_ERROR. There are several reasons this can be happening. As a sanity check, make sure that what you are trying to do works with DevWare. There are some board/sensor/PC combinations that just cannot handle the amount of data being sent. For example this may happen with a Demo1 or Demo1A board when trying to run 10-bit data for a 2MPixel sensor. If DevWare can handle it, then your application should be able to handle it as well.

If you are changing the size of the image by writing to the sensor registers, make sure that you are supplying the correct width and height to mi_updateFrameSize(). Remember mi_updateFrameSize() does not affect the amount of data being sent. It just tells the library how much data to expect.

If you are capturing JPEG data, this return value is normal since the compressed image will be smaller than the buffer.

Another reason you may not be able to capture an image is due to data swizzling. This is only relevant to Demo1/Demo1A boards when outputting 10bit Bayer data. The data coming from the sensor head has been wired such that the 8 most significant bits are in the low order byte and the 2 least significant bits are in the high order byte. This is done so that we can run in 8bit mode. When running in 10bit mode the data has to be reordered (swizzled) so that all of the bits are in the correct order. This is done in hardware on the Demo2 boards, however on the Demo1/Demo1A boards this can either be done by the library or the application. In order to prevent the user from having to swizzle the data, we provide this mechanism in the library. However due to timing considerations, if the library has to grab a frame and then swizzle the data, the processor may not be able to keep up and will start dropping frames. To prevent this, we have a swizzle mode which can be disabled so that the library doesn't do the swizzling and the Application does it instead. This works because we usually grab frames in one thread and another thread processes the frames. There are examples of the data swizzling in the SimpleCapture application.

8.3 Midlib2.h causes compilation errors

Please see the NOTE in the Portability section (4.0) above.

9.0 ACCESSING APTINA IMAGING DEVICES WITH PERL

This application note describes how Aptina imaging devices such as demonstration cameras and emulation boards can be accessed using the Perl programming language on a Windows platform. It also introduces the Aptina (Micron) Perl wrapper MIDLib.pm which provides convenient access to mode setting commands which are not normally available.

9.1 Prerequisites

ActiveState Perl 5.8 (Win32::OLE module required)
Aptina Imaging software
Aptina Perl MIDLib module (not essential, but provides convenient access to mode set/get options)
An understanding of Object Oriented Perl is useful, but not essential.

9.2 Win32 Perl Tips for Unix Programmers

For non-Windows Perl programmers, the first step is getting the correct 'hash bang' line. Like a Unix system this points to your Perl binary, but must also include the Windows drive designator.

It might look something like this -
#! c:/Program\ Files/Perl/bin/perl

9.2.1 Command line access

The native Windows command prompt is a rather impoverished place to work. It is recommended that the Cygwin environment is installed and Perl run from there.

9.2.2 Running Perl Programs

Note that some installations of ActiveState Perl refuse to recognize the path to your Perl binary in your code and may have to be run like this -

%perl my_program.pl

rather than -

```
%my_program.pl
```

9.2.3 Finding Out About Available Perl Methods

The OLE-Browser available in the ActiveState Perl menu is an excellent tool for locating methods available for Perl to use. Scroll down the list of libraries to the one labelled *MidLIBCom Type Library*, clicking on this will show all the method calls available. It is guaranteed to be more up to date than any documentation.

9.2.4 Setting OLE Warnings Level

The OLE class can be set to different levels of warning sensitivity, for debugging code use level 2 or 3. Anything lower risks masking useful messages. The example (Listing 1) shows how this is done.

```
0       Ignore error, return undef
1       Carp::carp if $^W is set (-w option)
2       always Carp::carp
3       Carp::croak
```

9.3 Perl access to Aptina Imaging Devices

To access data from an Aptina Imaging device several steps are needed.
Create new COM wrapper object which points to the Micron Imaging Library

Get list of all cameras connected to PC (usually just one device)

Open connection to camera

Start transport stream

Listing 1. How these steps are performed in Perl -
#! c:/Program\ Files/Perl/bin/perl

```
use strict;
use Win32::OLE;
Win32::OLE->Option( Warn => 3 );

# [1] set up COM connection to MIDLib
my $Com = Win32::OLE->new('MIDLibCom.MIDLib');
die "Error - couldn't get COM object\n" unless $Com;

# [2] open connections to all available cameras
# If you have a sensor data file (.sdat or .xsdat) pass it's path as an
argument.
```

```
# If you do not have an sdat file a null string must be passed (not
nothing).
my $Cameras = $Com->OpenCameras('');

# [3] $Cameras is an arrayref. Each array item is a Camera object.
my $Camera = $Cameras->[0];
die "Error - couldn't get CAMERA object\n" unless $Camera;

# [4] need to start transport stream before we can do anything
$Camera->startTransport;
```

Once processing has finished the camera stream can be stopped and the COM connection closed.

```
$Camera->stopTransport;
$Com->closeCameras;
```

9.4 Taking a Photo

Here's an example program that takes a photo and save it to a file in raw format. The file can be viewed using the Aptina 'PlayBack' tool.

```
#! c:/Program\ Files/Perl/bin/perl
use strict;

use Win32::OLE;
Win32::OLE->Option( Warn => 3 );

# photo is stored here
my $capture   = 'photo.raw';

# change sensor data file to suit your imaging device
my $sdat = 'C:\Aptina Imaging\sensor_data\A-3100-ES3-DEV.xsdat';

my $Com = Win32::OLE->new('MIDLibCom.MIDLib');

die "Error - couldn't get COM object ($Com)\n" unless $Com;

# open connections to all available cameras
my $Cameras = $Com->OpenCameras($sdat);

# $Cameras is an arrayref. Each array item is a Camera object.
my $Camera = $Cameras->[0];

die "Error - couldn't get CAMERA object\n" unless $Cameras;

# need to start transport stream before we can do anything
$Camera->startTransport;

# $Sensor is an object inside Camera object containing sensor data
my $Sensor = $Camera->sensor;

# get buffer size
my $width  = $Sensor->width;
my $height = $Sensor->height;

print "Grabbing frame ($width, $height)...\n";
my @frame = $Camera->grabFrame();

print "Stopping transport stream\n";
$Camera->stopTransport();
$Com->closeCameras;
```

```
print "Saving photo to $capture\n";

die "Couldn't open $capture for writing\n" unless (open RAW, ">
$capture");

binmode RAW;

for (my $y = 0; $y <= $height-1; $y++) {
      for (my $x = 0; $x <= $width-1; $x++) {
            my $data = $frame[0][$x][$y];
            my $fdata = pack("v", $data);
            print RAW $fdata;
      }
}
close RAW;
```

9.5 Aptina MIDLib Perl Module

This is a wrapper around the Win32::OLE class that provides convenient access to the Aptina mode setting functions, which are not easy to perform normally. It would be used in a similar way to the Win32::OLE approach as shown below.
The MIDLib module is available here: C:\Aptina Imaging\lib\MIDLib.pm

```
#! c:/Program\ Files/Perl/bin/perl
use strict;

use MIDLib;

# set up MIDLib
my $Com = MIDLib->new('MIDLibCom.MIDLib');
MIDLib->Option( Warn => 2 );

die "Error - couldn't get COM object\n" unless $Com;

# open connections to all available cameras
my $Cameras = $Com->OpenCameras('');

# $Camera is an arrayref. Each array item is a Camera object.
my $Camera = $Cameras->[0];
die "Error - couldn't get CAMERA object\n" unless $Camera;

# need to start transport stream before we can do anything
$Camera->startTransport;
```

However, the advantage of the MIDLib module is that it provides convenient access to the setMode command with mode names.
For example -

```
$Camera->setMode('MI_REG_ADDR_SIZE', 16);       # sets mi_reg_addr_size
to 16 bits
$Camera->setMode('MI_REG_DATA_SIZE', 8);        # sets mi_reg_data_size
to 16 bits
```

And the same with the getMode command -

```
my $data_bits = $Camera->getMode('MI_REG_DATA_SIZE');
```