Transport DLL Development Guide

Introduction

A transport is the link between a camera device and the Aptina software libraries. The purpose of a transport is to implement the camera control (read and write registers, set modes) and image acquisition functions needed by the higher level software. With a Transport DLL you can:

1. Support an Aptina sensor on a device other than the Demo System camera so DevWare or any other ApBase-based application will work with the device.
2. Create a software simulated camera and sensor, simulating either an existing sensor or new sensor, and be able to use DevWare as the user interface for the simulation.

Transport DLL Exports

The functions you need to implement and export from your Transport DLL are described below.
The exported functions should use the C calling convention. If you are using C or C++ this would normally be the default, so no special syntax is needed. If it is not, the keyword to force a function to the C calling convention is normally *__cdecl* in most C/C++ compilers. The function prototypes in this document assume C calling convention is the default.
To make your functions export from the DLL in C or C++ you would normally precede the function declaration with *__declspec(dllexport)*.
The symbols and data structures used by the Transport DLL interface are defined in the midlib2.h and midlib2_trans.h header files.
Several of these functions are required to be implemented and exported from your DLL. If they are not exported then the Aptina libraries will not load the DLL. The remaining functions are optional. If an optional function is not present then either a default handler will be provided to the application, of the feature will not be available. Whether a function is required or optional will be noted for each one below.

DLL Export SetHelpersDLL()

Definition:
__declspec(dllexport) mi_s32 SetHelpersDLL(mi_transport_helpers_t *pHelpers);
Summary:
This function is called when the DLL is loaded, before any other function. The parameter provides pointers to some functions that may be useful to a Transport DLL, but are not appropriate to be called by ordinary applications, so are not exported from the libraries like the main APIs.
Parameters:

pHelpers	Pointer to a structure containing pointers to helper functions.

Returns:

MI_CAMERA_SUCCESS	Initialization successful.

Optional:
This function is not necessary if you don't need the helper functions.
The following sections describe the helper functions in the pHelpers structure.

Helper Function getSensor()

Definition:
mi_s32 getSensor(mi_camera_t *pCamera, const char *dir_or_file);
Summary:
Performs the device probe to automatically find the correct sensor data file for the current sensor, and create the pCamera->sensor structure from

the data in the file. Pass in the pCamera pointer and the name of a file or directory. If the parameter is a sensor data file, the function will load the file. If the parameter is a directory, the function will search the directory for a sensor data file that matches the device and load it.
Parameters:

pCamera	Pointer to a camera structure.
dir_or_file	Name of a directory or sensor data file.

Returns:

MI_CAMERA_SUCCESS	Initialization successful.
MI_SENSOR_FILE_PARSE_ERROR	A syntax error was found in the sensor data file.
MI_SENSOR_DOES_NOT_MATCH	No sensor data file was found that matches the current sensor.
MI_CAMERA_ERROR	Various other error conditions.

Helper Function updateFrameSize()

Definition:
void updateFrameSize(mi_camera_t* pCamera , mi_u32 nWidth, mi_u32 nHeight, mi_s32 nBitsPerClock, mi_s32 nClocksPerPixel);
Summary:
Boilerplate code for UpdateFrameSizeDLL(). Sets pCamera->sensor->width, ~~->height, ->pixelBits, and ->pixelBytes according to the parameters. Also sets BITS_PER_CLOCK and CLOCKS_PER_PIXEL in the camera context structure (see setContext() below). Sets pCamera~~>sensor->buffe rSize to width * height * pixelBytes; you may need to fix up the bufferSize after calling this function. Call from UpdateFrameSizeDLL(). See sample code.
Parameters (same as UpdateFrameSizeDLL()):

pCamera	Pointer to a camera structure.
nWidth	New image width, or 0 to keep the current width.
nHeight	New image height, or 0 to keep the current height.
nBitsPerClock	New bits/pixel clock, or 0 to keep the current value.
nClocksPerPixel	New clocks/pixel, or 0 to keep the same value.

Returns:

void	

Helper Function updateImageType()

Definition:
void updateImageType(mi_camera_t* pCamera);
Summary:
Make sure the sensor->imageType field is consistent with the current bits per pixel. Call from UpdateFrameSizeDLL() before returning. See sample code.
Parameters:

pCamera	Pointer to a camera structure.

Returns:

void	

Helper Function errorLog()

Definition:

mi_s32 errorLog(mi_s32 errorCode, mi_s32 errorLevel, const char *logMsg, const char *szSource, const char *szFunc, mi_u32 nLine);
Summary:
Adds an error message to the log file. This is the error logging that is controlled on the DevWare Options dialog. Normally you call this function from the return statement. For example, instead of:
return MI_CAMERA_ERROR;
Use
return HelperFn->errorLog(MI_CAMERA_ERROR, -1, "Couldn't open driver",
*_FILE, __FUNCTION, __LINE_*);
Since the last three parameters are always the same, you can save typing by creating a preprocessor macro.
#define ERRORLOG(e,m) HelperFn->errorLog(e,-1,m,*_FILE,FUNCTION,LINE_*)
The example becomes:
return ERRORLOG(MI_CAMERA_ERROR, "Couldn't open driver");
Parameters:

errorCode	An error code defined in midlib2.h.
errorLevel	Severity of the error. Normally you can use -1 for this parameter as the severity is implied by the error code.
logMsg	Additional information to put in the log file.
szSource	Source code file name where error occurred. Normally *_FILE_*.
szFunc	Name of function where error occurred. Normally *_FUNCTION_* or "?" if your compiler doesn't support that symbol.
nLine	Source code line number where error occurred. Normally *_LINE_*.

Returns:

errorcode	Always returns the first parameter.

Helper Function log()

Definition:
void log(mi_s32 logType, const char *logMsg, const char *szSource, const char *szFunc, mi_u32 nLine);
Summary:
Adds a message to the log file. This is the logging that is controlled on the DevWare Options dialog. As with errorLog(), you can save typing by creating a macro that fills in the last three parameters automatically. Note that this function has no equivalent to the errorLevel parameter of errorLog().
Parameters:

logType	A log message type defined in midlib2.h. The different log types can be turned on/off by the user on the DevWare Options dialog. MI_LOG_SHIP – A SHiP (I2C) operation. MI_LOG_USB – A USB command. MI_LOG_DEBUG – Debug info. MI_LOG – Any other kind of message.
logMsg	The text of the message.
szSource	Source code file name where this log() call is located. Normally *_FILE_*.
szFunc	Name of function where this log() call is located. Normally *_FUNCTION_*.
nLine	Source code line number of this log() call. Normally *_LINE_*.

Returns:

void	

Helper Function setMode()

Definition:
mi_s32 setMode(mi_camera_t *pCamera, mi_modes mode, mi_u32 val);
Summary:
This is the default setMode() handler. If you are implementing SetModeDLL() you should call this function from your SetModeDLL() so the pCamera structure stays consistent. If you don't implement SetModeDLL() in your DLL then the software automatically uses this function instead.

See SetModeDLL().
Parameters:

pCamera	Pointer to the camera structure.
mode	The mode to set.
val	The new mode value.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
MI_CAMERA_ERROR	An error occurred.

Helper Function getMode()

Definition:
mi_s32 getMode(mi_camera_t *pCamera, mi_modes mode, mi_u32 *val);
Summary:
Calls the default getMode() handler. If you are implementing GetModeDLL() you can call this function to handle modes that you don't need to override. If you don't implement GetModeDLL() in your DLL then the software automatically uses this function instead. See GetModeDLL().
Parameters:

pCamera	Pointer to the camera structure.
mode	The mode to get.
val	Pointer to a 32-bit variable to store the mode value.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
MI_CAMERA_ERROR	An error occurred.

Helper Function setContext()

Definition:
mi_s32 setContext(mi_camera_t *pCamera, mi_context_fields field, mi_intptr val);
Summary:
Set fields in the pCamera->context structure. The context structure holds internal data associated with the camera. The context structure definition can change frequently and is not published. This function gives access to the fields needed by a Transport DLL. See the sample code.
MI_CONTEXT_PRIVATE_DATA gives you a convenient way to associate your own data with the pCamera. The val parameter can be an integer or a pointer to a data structure.
If the field is MI_CONTEXT_DRIVER_INFO, the val parameter is interpreted as a const char *. The DRIVER_INFO is a string up to 255 characters long that will be included in bug reports, and can be viewed by the user in DevWare with the menu Help System Dump. You can put any information you want that you think would be useful.
Parameters:

pCamera	Pointer to the camera structure.
Field	The field to set. MI_CONTEXT_PRIVATE_DATA MI_CONTEXT_DRIVER_INFO MI_CONTEXT_BITS_PER_CLOCK MI_CONTEXT_CLOCKS_PER_PIXEL MI_CONTEXT_PIXCLK_POLARITY
Val	The new field value.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
MI_CAMERA_ERROR	An error occurred.

Field MI_CONTEXT_PRIVATE_DATA

Summary:
MI_CONTEXT_PRIVATE_DATA gives you a convenient way to associate your own per-camera device data with the pCamera pointer. The val parameter can be an integer or a pointer to a data structure.

Field MI_CONTEXT_DRIVER_INFO

Summary:
In this case the val parameter is interpreted as a const char *. The DRIVER_INFO is a string up to 255 characters long that will be included in bug reports, and can be viewed by the user in DevWare with the menu Help System Dump. You can put any information you want that you think would be useful.

Field MI_CONTEXT_BITS_PER_CLOCK

Summary:
Initialize the bits per clock value. Combined with the clocks per pixel, this defines the format of pixels.

Field MI_CONTEXT_CLOCKS_PER_PIXEL

Summary:
Initialize the clocks per pixel value. Combined with the bits per clock, this defines the format of pixels.

Field MI_CONTEXT_PIXCLK_POLARITY

Summary:
Initialize the pixel clock polarity for parallel data.

Helper Function getContext()

Definition:
mi_s32 getContext(mi_camera_t *pCamera, mi_context_fields field, mi_intptr *val);
Summary:
Get fields from the pCamera->context structure. See the sample code. If the field is MI_CONTEXT_DRIVER_INFO, the val parameter is interpreted as a char *, and the buffer needs to be at least 256 bytes.
Parameters:

pCamera	Pointer to the camera structure.
field	The field to get. MI_CONTEXT_BITS_PER_CLOCK MI_CONTEXT_CLOCKS_PER_PIXEL MI_CONTEXT_PIXCLK_POLARITY MI_CONTEXT_PRIVATE_DATA MI_CONTEXT_APP_HWND MI_CONTEXT_DRIVER_INFO
val	Pointer to a variable to hold the field value.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
MI_CAMERA_ERROR	An error occurred.

Helper Function unswizzleBuffer()

Definition:
void unswizzleBuffer(mi_camera_t *pCamera, mi_u8 *pInBuffer);
Summary:
Unswizzles the data in the buffer. Call from your GrabFrameDLL() function as needed to unswizzle Bayer data. Unswizzling refers to putting out-of-order bits into the correct order. For example, 10-bit unswizzling does the following:
x x x x x x 1 0 9 8 7 6 5 4 3 2 x x x x x x 9 8 7 6 5 4 3 2 1 0
Parameters:

pCamera	Pointer to the camera structure.
pInBuffer	Pointer to the raw data

Returns:

void	

Helper Function readReg()

Definition:
mi_u32 readReg(mi_camera_t *pCamera, const char *szRegister, const char *szBitfield, mi_s32 bCached);
Summary:
Reads a named sensor register or register bitfield, optionally using the cached value kept by the library. If the szBitfield parameter is NULL or "" it reads the whole register. This function doesn't report errors, just returns 0 if an error occurs. Indirectly calls one of the register read functions, if needed.
Parameters:

pCamera	Pointer to the camera structure.
szRegister	Name of a sensor register defined in the sensor data file.
szBitfield	Name of a bitfield of the register or NULL.
bCached	If non-zero then get the register value from the cache instead of reading the hardware.

Returns:

Register or bitfield value	

Helper Function getBoardConfig()

Definition:
mi_s32 getBoardConfig(mi_camera_t *pCamera, const char *dir_or_file);
Summary:
If the camera is an Aptina Demo camera such as Demo2X or Demo3, this function fills in the chip[] array of the pCamera structure by loading .cdat files that correspond to FPGAs or peripherals on the camera system I2C bus.
Parameters:

pCamera	Pointer to a camera structure.
dir_or_file	Path to sensor data file (same as used for getSensor()).

Returns:

MI_CAMERA_SUCCESS	Initialization successful (even if no matching cdat files were found).
MI_PARSE_FILE_ERROR	Error opening the config file.

DLL Export OpenCameraDLL()

Definition:
__declspec(dllexport) mi_s32 OpenCameraDLL(mi_camera_t *pCamera, const char *sensor_dir_file, int deviceIndex);
Summary:
This function is called when the DLL is loaded, repeatedly, once for each camera. The deviceIndex parameter will = 0 on the first call, 1 on the second call, and so on. For each camera, do any internal initialization you may need. Fill in pCamera fields including pCamera->sensor, but do not touch pCamera->context, which is already initialized and is used internally by the Aptina libraries. You can use the setContext() helper function to associate the deviceIndex and/or any other private per-camera data to the pCamera object. The sensor_dir_file parameter is the name of the sensor data file or directory passed by the application to ap_DeviceProbe(). You can quickly initialize your pCamera->sensor with the helper function getSensor().
If all cameras have already been initialized, return MI_CAMERA_ERROR to end the init process. For example, if there are two cameras, return MI_CAMERA_SUCCESS on deviceIndex = 0 and deviceIndex = 1, and return MI_CAMERA_ERROR on deviceIndex = 2.
Parameters:

pCamera	Pointer to the camera structure.
sensor_dir_file	Name of sensor data file or directory to search for sensor data files.

Returns:

MI_CAMERA_SUCCESS	Initialization successful.
Other error code	In case of an error, or when all cameras have been initialized. If this is deviceIndex = 0 then the error code will be passed back to the application.

Required:
If this function is not exported, the Transport DLL will be ignored.

DLL Export CloseCameraDLL()

Definition:
__declspec(dllexport) mi_s32 CloseCameraDLL(mi_camera_t *pCamera);
Summary:
This is the last function called before the DLL is unloaded. Free all memory and resources. Do not free pCamera or pCamera->context. You may, but do not need to, free pCamera->sensor.
Parameters:

pCamera	Pointer to the camera structure.

Returns:
The return value is ignored.
Required:
If this function is not exported, it could cause memory leaks.

DLL Export StartTransportDLL()

Definition:
__declspec(dllexport) mi_s32 StartTransportDLL(mi_camera_t *pCamera);
Summary:
Implements the mi_camera_t::startTransport() function. This function will be called before grabFrame(), readSensorRegister(), etc. Get ready to do I/O functions on the camera. Open the device driver, etc.
Parameters:

pCamera	Pointer to the camera structure.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Required:
If this function is not exported, the DLL will be ignored.

DLL Export StopTransportDLL()

Definition:
__declspec(dllexport) mi_s32 StopTransportDLL(mi_camera_t *pCamera);
Summary:
Implements the mi_camera_t::stopTransport() function. The application will not do any more I/O to the device. Close driver, etc. Usually this is called right before closeCamera().
Parameters:

pCamera	Pointer to the camera structure.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Required:
If this function is not exported, the DLL will be ignored.

DLL Export ReadSystemRegistersDLL()

Definition:
__declspec(dllexport) mi_s32 ReadSystemRegistersDLL(mi_camera_t *pCamera, mi_addr_type addrType, mi_u32 addrSpace, mi_u32 startAddr, mi_u32 dataSize, mi_u32 numRegs, mi_u32 vals[]);
Summary:
Read sensor, SOC or ISP registers, variables or RAM according to the parameters.
Parameters:

pCamera	Pointer to the camera structure.
addrType	The type of register or memory location, for example MI_REG_ADDR, MI_MCU_ADDR, MI_SFR_ADDR, etc.
addrSpace	Address space or memory region.
startAddr	Address of first register to read.
dataSize	Size of the data items, 8, 16 or 32 bits.
numRegs	Number of registers to read.
Vals	Array for register values. Put one register value in each array element even if the registers are less than 32 bits wide.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported, lower-level register accesses will be used instead if possible.

DLL Export WriteSystemRegistersDLL()

Definition:
__declspec(dllexport) mi_s32 WriteSystemRegistersDLL(mi_camera_t *pCamera, mi_addr_type addrType, mi_u32 addrSpace, mi_u32 startAddr, mi_u32 dataSize, mi_u32 numRegs, mi_u32 vals[]);

Summary:
Write sensor, SOC or ISP registers, variables or RAM according to the parameters.
Parameters:

pCamera	Pointer to the camera structure.
addrType	The type of register or memory location, for example MI_REG_ADDR, MI_MCU_ADDR, MI_SFR_ADDR, etc.
addrSpace	Address space or memory region.
startAddr	Address of first register to write.
dataSize	Size of the data items, 8, 16 or 32 bits.
numRegs	Number of registers to write.
vals	Array for register values. Put one register value in each array element even if the registers are less than 32 bits wide.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported, lower-level register accesses will be used instead if possible.

DLL Export ReadSensorRegistersDLL()

Definition:
__declspec(dllexport) mi_s32 ReadSensorRegistersDLL(mi_camera_t *pCamera, mi_u32 addrSpace, mi_u32 startAddr, mi_u32 numRegs, mi_u32 vals[]);
Summary:
Implements the mi_camera_t::readSensorRegisters() function. Write addrSpace to the address space register before doing the read. The address space register is given by pCamera->sensor->addr_space->reg_addr. If it is 0 or if pCamera->sensor->addr_space is NULL then ignore addrSpace.
Parameters:

pCamera	Pointer to the camera structure.
addrSpace	On sensors with paged register addressing, this is the value to write into the address space register
startAddr	Address of first register to read. The register address can be 8 or 16 bits wide determined by pCamera->sensor->reg_addr_size.
numRegs	Number of registers to read
vals	Array for register values. The registers can be 8, 16 or 32 bits wide determined by pCamera->sensor->reg_data_size. Put one register value in each array element even if the registers are less than 32 bits wide.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported, lower-level register access functions will be used instead.

DLL Export WriteSensorRegistersDLL()

Definition:
__declspec(dllexport) mi_s32 WriteSensorRegistersDLL(mi_camera_t *pCamera, mi_u32 addrSpace, mi_u32 startAddr, mi_u32 numRegs, mi_u32 vals[]);
Summary:
Implements the mi_camera_t::writeSensorRegisters() function. Write addrSpace to the address space register before doing the read. The address

space register is given by pCamera->sensor->addr_space->reg_addr. If it is 0 or if pCamera->sensor->addr_space is NULL then ignore addrSpace.
Parameters:

pCamera	Pointer to the camera structure.
addrSpace	On sensors with paged register addressing, this is the value to write into the address space register
startAddr	Address of first register to write. The register address can be 8 or 16 bits wide determined by pCamera->sensor->reg_addr_size.
numRegs	Number of registers to write
vals	Array of register values. The registers can be 8, 16 or 32 bits wide determined by pCamera->sensor->reg_data_size. There is one register value in each array element even if the registers are less than 32 bits wide.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported, lower-level register access functions will be used instead.

DLL Export ReadRegisterDLL()

Definition:
__declspec(dllexport) mi_s32 ReadRegisterDLL(mi_camera_t *pCamera, mi_u32 shipAddr, mi_u32 regAddr, mi_u32 *val);
Summary:
Implements the mi_camera_t::readRegister() function. Read one register.
Parameters:

pCamera	Pointer to the camera structure.
shipAddr	Base address
regAddr	Register address. The register address may be 0, 8 or 16 bits, determined by the state of MI_REG_ADDR_SIZE mode.
val	Pointer to where to put the register value. The register value may be 8, 16 or 32 bits, determined by the MI_REG_DATA_SIZE mode.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported, ReadRegistersDLL will be used. Either this function or ReadRegistersDLL must be implemented to support register read capability.

DLL Export WriteRegisterDLL()

Definition:
__declspec(dllexport) mi_s32 WriteRegisterDLL(mi_camera_t *pCamera, mi_u32 shipAddr, mi_u32 regAddr, mi_u32 val);
Summary:
Implements the mi_camera_t::writeRegister() function. Write one register.
Parameters:

pCamera	Pointer to the camera structure.
shipAddr	Base address
regAddr	Register address. The register address may be 0, 8 or 16 bits, determined by the state of MI_REG_ADDR_SIZE mode.
val	The register value to write. The register value may be 8, 16 or 32 bits, determined by the MI_REG_DATA_SIZE mode.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Required:
If this function is not exported, WriteRegistersDLL will be used. Either this function or WriteRegistersDLL must be implemented to support register write capability.

DLL Export ReadRegistersDLL()

Definition:
__declspec(dllexport) mi_s32 ReadRegistersDLL(mi_camera_t *pCamera, mi_u32 shipAddr, mi_u32 startAddr, mi_u32 numRegs, mi_u32 vals[]);

Summary:
Implements the mi_camera_t::readRegisters() function. Read multiple consecutive registers.
Parameters:

pCamera	Pointer to the camera structure.
shipAddr	Base address
regAddr	Register address. The register address may be 0, 8 or 16 bits, determined by the state of MI_REG_ADDR_SIZE mode.
numRegs	Number of registers to read
vals	Array to hold the register values. The register values may be 8, 16 or 32 bits, determined by the MI_REG_DATA_SIZE mode. Put one value in each array element even if they are less than 32 bits each.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported, ReadRegisterDLL will be used. Either this function or ReadRegisterDLL must be implemented to support register read capability.

DLL Export WriteRegistersDLL()

Definition:
__declspec(dllexport) mi_s32 WriteRegistersDLL(mi_camera_t *pCamera, mi_u32 shipAddr, mi_u32 startAddr, mi_u32 numRegs, mi_u32 vals[]);
Summary:
Implements the mi_camera_t::writeRegisters() function. Write multiple consecutive registers.
Parameters:

pCamera	Pointer to the camera structure.
shipAddr	Base address
regAddr	Register address. The register address may be 0, 8 or 16 bits, determined by the state of MI_REG_ADDR_SIZE mode.
numRegs	Number of registers to write
vals	Array of register values. The register values may be 8, 16 or 32 bits, determined by the MI_REG_DATA_SIZE mode. One register value per array element, even if the data width is less than 32 bits.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Required:
If this function is not exported, WriteRegisterDLL will be used. Either this function or WriteRegisterDLL must be implemented to support register write capability.

DLL Export SendCommandDLL()

Definition:
__declspec(dllexport) mi_s32 SendCommandDLL(mi_camera_t *pCamera, mi_u32 command, mi_u32 paramSize, mi_u8 *paramBuffer, mi_u32 *statusCode, mi_u32 resultSize, mi_u8 *resultBuffer, mi_u32 *bytesReturned);
Summary:
Some SOCs and ISPs have a host command protocol separate from register reads and writes, for example implemented over a serial port or Ethernet port. Implement this function if such a protocol is available.
Parameters:

pCamera	Pointer to the camera structure.
command	Command number.
paramSize	The size of the command parameter data.
paramBuffer	Buffer containing command parameters.
statusCode	Pointer to receive command status code from the device.
resultSize	Size in bytes of the resultBuffer.
resultBuffer	Buffer to hold command result data.
bytesReturned	Number of bytes of data placed in resultBuffer by the function.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
MI_CAMERA_TIMEOUT	Timeout waiting for the device to finish processing the command.
Other error code	Appropriate for the error that occurred.

Optional:
This function is not needed if the underlying transport mechanism doesn't support host commands.

DLL Export ReadFarRegistersDLL()

Definition:
__declspec(dllexport) mi_s32 ReadFarRegistersDLL(mi_camera_t *pCamera, mi_u32 shipAddr, mi_u32 startAddr, mi_u32 addrSize, mi_u32 dataSize, mi_u32 numRegs, mi_u32 vals[]);
Summary:
Like ReadRegistersDLL(), but uses a second, 'far', bus master. On some systems, the main sensor or ISP may have its own I2C bus master, and a sensor or other devices connected to it. This function is to do a read transaction on that bus.
Parameters:

pCamera	Pointer to the camera structure.
shipAddr	Base address
regAddr	Register address. The register address may be 0, 8 or 16 bits, determined by the addrSize parameter.
addrSize	Register address size, in bits. May be 0, 8 or 16.
dataSize	Register data size, in bits. May be 8, 16, or 32.
numRegs	Number of registers to read
Vals	Array to hold the register values. The register values may be 8, 16 or 32 bits, determined by the dataSize parameter. Put one value in each array element even if they are less than 32 bits.

Returns:

MI_CAMERA_SUCCESS	Function was successful.

Other error code	Appropriate for the error that occurred.

Optional:
This function is not needed if there is no bus master on the main device, or if the main device is an Aptina product.

DLL Export WriteFarRegistersDLL()

Definition:
__declspec(dllexport) mi_s32 WriteFarRegistersDLL(mi_camera_t *pCamera, mi_u32 shipAddr, mi_u32 startAddr, mi_u32 addrSize, mi_u32 dataSize, mi_u32 numRegs, mi_u32 vals[]);
Summary:
Like WriteRegistersDLL(), but uses a second, 'far', bus master. On some systems, the main sensor or ISP may have its own I2C bus master, and a sensor or other devices connected to it. This function is to do a write transaction on that bus.
Parameters:

pCamera	Pointer to the camera structure.
shipAddr	Base address
regAddr	Register address. The register address may be 0, 8 or 16 bits, determined by the addrSize parameter.
addrSize	Register address size, in bits. May be 0, 8 or 16.
dataSize	Register data size, in bits. May be 8, 16, or 32.
numRegs	Number of registers to read
Vals	Array of the register values. The register values may be 8, 16 or 32 bits, determined by the dataSize parameter. Each array element will hold one value, even if they are less than 32 bits.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
This function is not needed if there is no bus master on the main device, or if the main device is an Aptina product.

DLL Export GrabFrameDLL()

Definition:
__declspec(dllexport) mi_s32 GrabFrameDLL(mi_camera_t *pCamera, mi_u8 *pInBuffer, mi_u32 bufferSize);
Summary:
Implements the mi_camera_t::grabFrame() function.
Parameters:

pCamera	Pointer to the camera structure.
pInBuffer	Pointer to data buffer
bufferSize	Size of the data buffer being passed in, for error checking only

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported, MI_GRAB_FRAME_ERROR will be returned to the application for any grab frame request.

DLL Export UpdateFrameSizeDLL()

Definition:
__declspec(dllexport) mi_s32 UpdateFrameSizeDLL(mi_camera_t *pCamera, mi_u32 width, mi_u32 height, mi_s32 nBitsPerClock, mi_s32 nClocksPerPixel);
Summary:
Implements the mi_camera_t::updateFrameSize() function. Update pCamera->sensor->width, height, pixelBits, pixelBytes, bufferSize, and imageType. You should also update the internal bitsPerClock and clocksPerPixel values using the setContext() helper function. You can use the updateImageType() helper function to handle imageType. See the sample code.
Parameters:

pCamera	Pointer to the camera structure.
width	New image width, or 0 to keep the current image width
height	New image height, or 0 to keep the current image height
nBitsPerClock	New bits/clock or 0 to keep the current value
nClocksPerPixel	New clocks/pixel or 0 to keep the current value

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported, the default handler will perform the calculation based on image dimensions and bits per pixel.

DLL Export UpdateBufferSizeDLL()

Definition:
__declspec(dllexport) mi_s32 UpdateBufferSizeDLL(mi_camera_t *pCamera, mi_u32 bufferSize);
Summary:
Implements the mi_camera_t::updateBufferSize() function. Update pCamera->sensor->bufferSize. In some cases the bufferSize computed by UpdateFrameSizeDLL() may be too small. In such a case this function will be called with the minimum required buffer size.
Parameters:

pCamera	Pointer to the camera structure.
bufferSize	Minimum buffer size.

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported, it may not be possible to capture JPEG format images.

DLL Export GetFrameDataDLL()

Definition:
__declspec(dllexport) mi_s32 GetFrameDataDLL(mi_camera_t *pCamera, mi_frame_data_t *pFrameData);
Summary:
Implements the mi_camera_t::getFrameData() function. Fill in the pFrameData structure based on the most recent call to GrabFrameDLL(). The imageBytesReturned field is needed for compressed image capture and should be set to the number of bytes of actual image data that were in the last frame grabbed, with byte accuracy.
Parameters:

pCamera	Pointer to the camera structure.

pFrameData	Pointer to the frame data structure

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported it will not be possible to capture compressed image formats.

DLL Export SetModeDLL()

Definition:
__declspec(dllexport) mi_s32 SetModeDLL(mi_camera_t *pCamera, mi_modes mode, mi_u32 val);
Summary:
Implements the mi_camera_t::setMode() function. Normally, you would not need to implement this function unless you want to be notified of a mode set or override the default function. You should call HelperFn->setMode() to keep the pCamera structure consistent.
Parameters:

pCamera	Pointer to the camera structure.
mode	The mode to set. See midlib2.h.
val	New value of the mode

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported from the DLL, a standard setMode function will be used.

DLL Export GetModeDLL()

Definition:
__declspec(dllexport) mi_s32 GetModeDLL(mi_camera_t **pCamera, mi_modes mode, mi_u32** val);
Summary:
Implements the mi_camera_t::getMode() function. Normally, you would not need to implement this function unless you want to override the default function of a mode get. You can call HelperFn->getMode() to perform the default get mode function for modes you do not intend to override.
Parameters:

pCamera	Pointer to the camera structure.
mode	The mode to get. See midlib2.h.
val	Pointer to store the value of the mode

Returns:

MI_CAMERA_SUCCESS	Function was successful.
Other error code	Appropriate for the error that occurred.

Optional:
If this function is not exported from the DLL, a standard getMode function will be used.

DLL Export GetDeviceNameDLL()

Definition:
__declspec(dllexport) const char * GetDeviceNameDLL(mi_camera_t *pCamera);
Summary:
This function should return a string that is unique to the device. That is, if there are two of your devices connected then this function should return a different string for each. Further, if the user creates a second process (for example two instances of DevWare at the same time), this function should return the same name for the same physical device.
If it is certain that there can only be one device connected to the computer at a time, then the name can be a string constant.
The name is used for inter-process synchronization. With this name all processes accessing a device will use the same mutex (mutual exclusion) object. So, for example, two processes will never call WriteSensorRegistersDLL() at the same time on the same device. Also the register cache and certain other sensor state will be shared among processes.
Parameters:

pCamera	Pointer to the camera structure.

Returns:

device name string	

Optional:
This function is optional. However, if it's not implemented and the user opens two applications at the same time, the processes won't be synchronized, possibly causing unexpected results.

DLL Export EnableDeviceNotificationDLL()

Definition:
__declspec(dllexport) mi_s32 EnableDeviceNotificationDLL(mi_camera_t *pCamera, mi_intptr hWnd);
Summary:
This function will be called after a successful camera open if the application has requested to be notified of device removal events. If your device type supports hot plugging (such as USB) then this function should register the given window handle for device notification messages (call RegisterDeviceNotification()).
If there is more than one camera attached, this function will be called for each one. The hWnd is global (not per-camera) so normally this function will have to keep track of whether the window has already been registered and not do it twice.
Parameters:

pCamera	Pointer to the camera structure.
hWnd	Window handle application's main window

Returns:

MI_CAMERA_SUCCESS	Registered for notifications (or already registered)
MI_CAMERA_NOT_SUPPORTED	Not supported by the device
MI_CAMERA_ERROR	Any other error

Optional:
This function is optional. If not implemented, the application will not recieve device removal messages from this device.

DLL Export MatchDevChangeMsgDLL()

Definition:
__declspec(dllexport) mi_s32 MatchDevChangeMsgDLL(mi_camera_t *pCamera, DWORD evtype, void *pDBH);
Summary:
This function will be called when a WM_DEVICECHANGE message is sent to the previously registered window handle. This function will be called for all WM_DEVICECHANGE messages, regardless of whether they are for this camera or not. Examine the structure pointed to by pDBH and determine if the message is for this device (see the Windows header file dbt.h). Return 1 if it is or 0 if it is not.
If the event type is DBT_DEVICEREMOVECOMPLETE and MatchDevChangeMsgDLL() returns 1 then StopTransportDLL() will be called. If you want to do any other processing, or handle any other event types, do so in this function.
Parameters:

pCamera	Pointer to the camera structure.

evtype	Event type (wParam from the WM_DEVICECHANGE message)
pDBH	Pointer to a DEV_BROADCAST_xxxx structure. Depends on event type. (lParam from the WM_DEVICECHANGE message)

Returns:

1	The message is for this device
0	The message is not for this device

Optional:
This function is optional. If not implemented, the application will not recieve device removal messages from this device.

DLL Export DisableDeviceNotificationDLL()

Definition:
__declspec(dllexport) mi_s32 DisableDeviceNotificationDLL(mi_camera_t *pCamera);
Summary:
This function will be called before closing a camera, or if the application sets the window handle to NULL with mi_SetDeviceChangeCallback(). Undo the enable operation.
If there is more than one camera attached, this function will be called for each one, but the unregister should probably be done only once when the last camera is closed.
Parameters:

pCamera	Pointer to the camera structure.

Returns:

MI_CAMERA_SUCCESS	Unregistered for notifications (or already unregistered)
MI_CAMERA_ERROR	Any other error

Optional:
This function is optional, but should be implemented if EnableDeviceNotificationDLL() was implemented.

DLL Export RegControlDLL()

Definition:
__declspec(dllexport) mi_s32 RegControlDLL(mi_camera_t *pCamera, mi_reg_data_t *pRegData, mi_u32 curVal, mi_u32 newVal, mi_u32 bitMask, mi_s32 test_only);
Summary:
Determine what the side-effects of writing a register will be, and update the pCamera accordingly. This function embodies the semantics of the sensor registers. RegControlDLL() has two functions: to return flags indicating semantic errors (like value out of legal range) or side-effects (like output image size changed), and to update the pCamera structure. If the test_only flag is set then the function should only return status flags and should not change pCamera structure.
The application will call this function with test_only = 1 before writing a register. Depending on the return value and the user's response the application may not write the register. If the application does write the register it will call RegControlDLL() again with test_only = 0, and the function should update the pCamera structure.
Parameters:

pCamera	Pointer to the camera structure.
pRegData	Pointer the register structure corresponding to the register being written
curVal	Current value of the register
newVal	New value, or proposed new value, of the register
bitMask	Bits of interest within the register
test_only	1: Do not change the pCamera structure, just return the flags. 0: Update pCamera as appropriate and return the flags.

Returns:

Bit-wise OR of the following flags, all that apply.

0	OK. No significant side-effects
1	Reallocate. Buffer size is changing, application needs to reallocate buffers.
2	Halt. The camera will stop producing images
4	Resume. The camera *may* resume producing images. Application will call PlayCapableDLL() to find out.
8	Not supported. Register change will activate a sensor feature that the camera cannot support.
16	Illegal combination. The new register value is in conflict with the current values of other registers.
32	Illegal value. The new register value is out of bounds for this register.
64	Reset. The register write will reset many or all other register values.
128	Clock. The register write will change the master clock frequency (writing a PLL register).

Optional:
The application will first call its built-in function, then the DLL function if it's exported.

DLL Export PlayCapableDLL()

Definition:
__declspec(dllexport) mi_s32 PlayCapableDLL(mi_camera_t *pCamera);
Summary:
Return whether the camera is producing images. In other words, whether it's OK to call grabFrame(). (Enables/disables the Play button on the DevWare toolbar.)
Normally, you only need to export this function if you are simulating a new type of sensor that is unknown to the application.
Parameters:

pCamera	Pointer to the camera structure.

Returns:

1	Camera or sensor can fulfill GrabFrameDLL() calls
0	Camera or sensor is in a disabled state

Optional:
If this function is not exported from the DLL, the application will use its built-in function.

DLL Export IsViewfinderDLL()

Definition:
__declspec(dllexport) mi_s32 IsViewfinderDLL(mi_camera_t *pCamera);
Summary:
Return whether the camera is in viewfinder mode or not. (Determines the state of the preview button the DevWare toolbar.)
Normally, you only need to export this function if you are simulating a new type of sensor that is unknown to the application.
Parameters:

pCamera	Pointer to the camera structure.

Returns:

1	Camera is in viewfinder mode
0	Camera is not in viewfinder mode

Optional:
If this function is not exported from the DLL, the application will use its built-in function.

DLL Export SetNotifyCallbackDLL()

Definition:
__declspec(dllexport) mi_u32 SetNotifyCallbackDLL(mi_camera_t *pCamera, MI_NOTIFYCALLBACK pNotifyCallback);
Summary:
The application will call this function to pass down the pointer to its notification message handler. The notification messages let you pause and resume the application, and update the image size safely.
When the application calls this function, save the value of pNotifyCallback in your own variable to be used later as needed.
You only need to use this mechanism when changes to the registers or image format are being driven by your code instead of by the application—so the application otherwise does not know about the changes.
Parameters:

pCamera	Pointer to the camera structure.
pNotifyCallback	Pointer to the notify message handler in the application.

Returns:

MI_CAMERA_SUCCESS	Always

Optional:
If this function is not exported from the DLL, there is no consequence.

Application Callback AppNotifyCallback()

Definition:
mi_u32 AppNotifyCallback(mi_camera_t *pCamera, mi_notify_codes notifyCode, mi_intptr lParam);
Summary:
A pointer to this function is passed from the application to the SetNotifyCallbackDLL() function. The name *AppNotifyCallback* is only an example name, you can call it what you like.
Call this function to send notification messages to the application when something is happening.
Parameters:

pCamera	Pointer to the camera structure.
notifyCode	One of the MI_NOTIFY_... symbols, described below.
lParam	Notify message parameter. Meaning depends on notifyCode. When passing a pointer cast it to mi_intptr type.

Returns:

MI_CAMERA_SUCCESS	Message was received and accepted.
MI_CAMERA_ERROR	pCamera pointer does not match the application's pCamera.notifyCode value is not a recognized notification code.lParam value is illegal.Application is not ready.Illegal calling thread.

MI_NOTIFY_PAUSE

lParam: 0 (ignored)
Action:
Stop the application from accessing the device. The application will shut down its worker threads. The function does not return until the application is stopped and it is safe to change width, height, pixel type and buffer size fields of pCamera.
Calling Thread:
In general, you can't call this from GrabFrameDLL() or register access functions since the application my call these functions from the very threads that need to be shut down.

MI_NOTIFY_RESUME

lParam: 0 (ignored)
Action: Reallocate buffers and undo MI_NOTIFY_PAUSE. Function does not return until the application is resumed.
Calling Thread: Call from any thread.

MI_NOTIFY_UPDATEFRAMESIZE

lParam: Pointer to an mi_notify_updateframesize_t object, cast to mi_intptr type.
Action:
Tell the application that the image format has changed. The application will: Pause itself; call pCamera->updateFrameSize() with the parameters from the mi_notify_updateframesize_t structure and set pCamera->sensor->imageType to the value in the mi_notify_updateframesize_t::imageType field; and resume. If any field of mi_notify_updateframesize_t is 0 that indicates to keep the current value. Function may return before the action is completed. **Calling Thread:** Call from any thread.

MI_NOTIFY_REFRESH

lParam: 0 (ignored)
Action:
Refresh dialogs. Use this message to resync the user interface to the register values if you change a lot of register values unbeknownst to the application. Function may return before the action is completed.
Calling Thread: Call from any thread.

Employing the DLL

To use your transport DLL, just copy it to the Plugins folder in the Demo Software installation location, typically C:\Aptina Imaging\Plugins. Alternatively, the application can pass the filename of the DLL to ApBase using the ap_DeviceProbeDll() function. If the application passes in a DLL filename then the Plugins folder is not searched.
DevWare-specific: You can use a command line parameter to specify the DLL. DevWare will pass the given DLL path to the Aptina libraries.
DevWare.exe /DLL="C:\path\to\your\transport\yourdev.dll"

Example Code

Here is a minimal, skeleton implementation of the required APIs. There is a more comprehensive example in the samples\TransportDLL subdirectory of the Aptina Imaging installation directory.

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#include "midlib2.h"
#include "midlib2_trans.h"
mi_transport_helpers_t *HelperFn;
#define ERRORLOG(e,m) HelperFn->errorLog(e,-1,m,_FILE_,\
_FUNCTION,LINE_)
#define MSGLOG(t,m) HelperFn->log(t,m,_FILE,FUNCTION,LINE_)
__declspec(dllexport) mi_u32 SetHelpersDLL(
mi_transport_helpers_t *pHelpers)
{
HelperFn = pHelpers;
return MI_CAMERA_SUCCESS;
}
__declspec(dllexport) mi_s32
OpenCameraDLL(mi_camera_t *pCamera, const char *sensor_dir_file,
mi_s32 deviceIndex)
{
mi_s32 retVal;
if (deviceIndex > 0)// one camera
return MI_CAMERA_ERROR;
pCamera->productID = MI_UNKNOWN_PRODUCT;
pCamera->productVersion = 0;
pCamera->firmwareVersion = 0;
pCamera->num_chips = 0;
strcpy(pCamera->productName, "My Camera");
strcpy(pCamera->transportName, "mydriver");
// - open the hardware device driver here
HelperFn->getBoardConfig(pCamera, sensor_dir_file);
retVal = HelperFn->getSensor(pCamera, sensor_dir_file);
// - close the hardware device driver here
if (retVal != MI_CAMERA_SUCCESS)
```

```
return ERRORLOG(retVal, "No matching sdat");
return MI_CAMERA_SUCCESS;
}
__declspec(dllexport) mi_s32 CloseCameraDLL(mi_camera_t *pCamera)
{
return MI_CAMERA_SUCCESS;
}
__declspec(dllexport) mi_s32 StartTransportDLL(mi_camera_t *pCamera)
{
// open the hardware device
return MI_CAMERA_SUCCESS;
}
__declspec(dllexport) mi_s32 StopTransportDLL(mi_camera_t *pCamera)
{
// close the hardware device
return MI_CAMERA_SUCCESS;
}
__declspec(dllexport) mi_s32
GrabFrameDLL(mi_camera_t *pCamera, mi_u8 *pInBuffer, mi_u32 bufferSize)
{
// get image data from device or create simulated image
return MI_CAMERA_SUCCESS;
}
__declspec(dllexport) mi_s32
ReadRegistersDLL(mi_camera_t *pCamera, mi_u32 shipAddr,
mi_u32 startAddr, mi_u32 numRegs, mi_u32 vals[])
{
mi_u32 addr_size;
mi_u32 data_size;
int i;
HelperFn->getMode(pCamera, MI_REG_ADDR_SIZE, &addr_size);
HelperFn->getMode(pCamera, MI_REG_DATA_SIZE, &data_size);
for (i = 0; i < numRegs; ++i)
{
vals[i] = 0;// do I2C reads
}
return MI_CAMERA_SUCCESS;
}
__declspec(dllexport) mi_s32
WriteRegistersDLL(mi_camera_t *pCamera, mi_u32 shipAddr,
mi_u32 startAddr, mi_u32 numRegs, mi_u32 vals[])
{
mi_u32 addr_size;
mi_u32 data_size;
HelperFn->getMode(pCamera, MI_REG_ADDR_SIZE, &addr_size);
HelperFn->getMode(pCamera, MI_REG_DATA_SIZE, &data_size);
// do I2C writes
return MI_CAMERA_SUCCESS;
}
__declspec(dllexport) mi_s32
UpdateBufferSizeDLL(mi_camera_t *pCamera, mi_u32 bufferSize)
{
pCamera->sensor->bufferSize = bufferSize;
return MI_CAMERA_SUCCESS;
}
__declspec(dllexport) mi_s32
UpdateFrameSizeDLL(mi_camera_t *pCamera, mi_u32 nWidth, mi_u32 nHeight,
mi_s32 nBitsPerClock, mi_s32 nClocksPerPixel)
{
HelperFn->updateFrameSize(pCamera, nWidth, nHeight, nBitsPerClock,
nClocksPerPixel);
UpdateBufferSizeDLL(pCamera, pCamera->sensor->width
```

- pCamera->sensor->height
- pCamera->sensor->pixelBytes);
  HelperFn->updateImageType(pCamera);
  return MI_CAMERA_SUCCESS;
  }