The Microscope Device Abstraction Layer of Micro-Manager

Nenad Amodaj[1], Henry Pinkard[2], Nick Anthony[3] and Nico Stuurman[4]

[1]Luminous Point, LLC

[2]Dept. Of Electrical Engineering and Computer Sciences, University of California Berkeley

[3]Dept. Of Biomedical Engineering Northwestern University, Evanston, IL

[4]Dept. of Cell. Pharm., University of California San Francisco/Howard Hughes Medical Institute

Introduction

Micro-Manager (also named μManager, *https://micro-manager.org*)[1, 2] is open-source software for operation of automated microscopes. The Micro-Manager application is used by thousands of laboratories world-wide, mainly for routine applications, but also for more creative, complicated approaches (e.g. [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], which became easier with the development of Pycro-Manager, a bridge to the Python computing environment [13]. The Micro-Manager application contains a GUI that communicates with microscope equipment through a device abstraction layer. This device abstraction layer can also be used outside of the Micro-Manager application and was designed about 15 years ago. Because of the great number of devices now supported by this software layer and the tantalizing potential to use this code under many more conditions, we will review here the original software design and identify possible improvements to enable its future use under a wider variety of circumstances.

The device abstraction layer was made so that anyone can create a "Device Adapter" to support any kind of microscopy related hardware device and make it available in the Micro-Manager application domain. We chose the term "Device Adapter" for Micro-Manager device modules and reserved "driver" for manufacturers' native device software components. A Micro-Manager Device Adapter performs normalization of various software commands specific to a set of devices (often from one manufacturer), so they conform to Micro-Manager's conceptual model of the microscope. Through the contributions of many people, in both academia and industry, hundreds of such Device Adapters now exist, operating a very large set of microscope hardware components. This rich resource can be used outside of the Micro-Manager GUI application and potentially facilitate operation of many custom-built microscopes.

To help ourselves and others think about the most efficient path towards easily reusable components for microscope software control, we discuss in detail the microscope model that implicitly emerges from the application programming interfaces (APIs): “MMDevice”, a particular hardware device, and “MMCore”, a single core component that facilitates interoperability between a collection of such devices and serves as the primary access point for device interaction. In designing these APIs we deliberately left some details unspecified to provide flexibility for future development and customization, but made the model specific enough for microscopy so that resulting scripts and applications end up being simple yet expressive.

The foundations of the Micro-Manager device interface were developed in 2005 and 2006, and reflect the paradigm of the "motorized microscope." Usually, this is a computer-controlled system consisting of a microscope stand (with built-in reflector changers, focus drive, etc..) equipped with cameras, stages, light sources, and various other peripherals attached. Both deficiencies and virtues of the Micro-Manager device interface originate from this tradeoff between

flexibility and domain efficiency.

In terms of practical use, the intent was to provide the following workflow for a programmer tasked to produce a Device Adapter for a new hardware device:

1. Install the device manufacturer's drivers (if required), software development kit (SDK), and any other software/hardware needed for the particular hardware device being worked with.
2. Take the Micro-Manager interface definition (two C++ header files) and a few utility classes (optional) and implement all functions required by the interface. The entire MMDevice package is small, lightweight, and located in a single directory (MMDevice).
3. Write code to control the new device and build a run-time module (a ".dll", ".dylib", or ".so" depending on the Operating System). The final result is a dynamically linked library file that conforms to Micro-Manager's API.
4. Install the Micro-Manager application from the Micro-Manager web site (*https://micro-manager.org*). Copy the new dynamic library file to the Micro-Manager installation directory and do final testing with the full software.
5. (Optional, but highly encouraged:) Contribute the Device Adapter source code to the Micro-Manager repository under an Open Source license so that it can be included in future binary distributions of the software.

Terminology

Module – A run-time software component located in a single file, usually an executable or dynamic library

Device Adapter – A C++ class implementing the MMDevice interface, that converts a device manufacturer's command set (or API) to Micro-Manager's device interface API

Driver – One or more software components provided by the device manufacturer to enable other software such as Micro-Manager to control the device (some devices do not require drivers)

Library – A dynamic library (e.g., .dll, .dylib or .so) conforming to Micro-Manager's module API and naming convention that contains one or more Device Adapters

Core Interface - The interface defining the core functionality of Micro-Manager. MMCore is the default (and only) implementation of the Core interface.

High-level Overview

Building blocks

Although MMCore is implemented in C++, its interface only uses C-compatible data types and can therefore be relatively easily used from other languages. To automate wrapping MMCore, we have been using the development tool "SWIG" (*http://www.swig.org/*), and export to Java (MMCoreJ_wrap), Python 2 (MMCorePy_wrap), and Python 3 (in the separate github-hosted project "pymmcore"). With some effort, wrapping to other languages using Swig can be accomplished.

The Micro-Manager user interface - being a plugin to ImageJ [14] written in Java - uses the Java interface to MMCore. Other Java compatible applications, such as Matlab, can use this same interface (and have access to the Micro-Manager Java API, which is not discussed further in this manuscript).

Initial Software Design Requirements

- To be cross-platform (Windows/Mac/Linux)
- A single layer (MMCore) to be used as an abstraction for a motorized microscope, i.e. strict separation of an API facing the user, and an API facing the devices

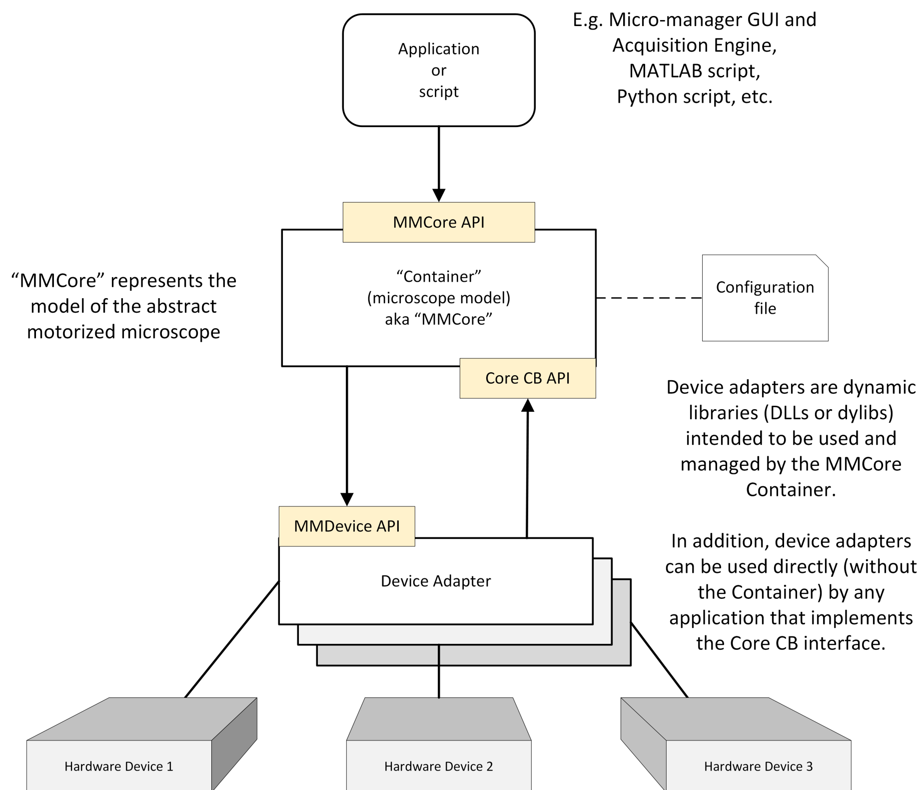

Figure 1: Micro-Manager software components
This diagram shows the relationship between Micro-Manager software components. At the very top sits the user application, the middle layer (MMCore) implements the model of the generic motorized microscope, through the "MMCore API". The microscope model layer communicates with devices through the "MMDevice API".

- Use of native C++ environment with C bindings for device interfacing to make all adapters and MMCore usable from any programming environment
- Allow drop-in run-time linking of all device adapters, i.e. copying a separately compiled device adapter binary into the correct location should work transparently
- All applications/scripts developed using MMCore API and developed on one microscope system should run without any changes on another microscope system (with different hardware components), as long as it has equivalent capabilities
- MMCore and devices should provide full run-time discovery of features and capabilities.
- It should be possible (and desirable) to write application code with absolutely zero *a priori* knowledge of the underlying hardware system

Threading model

To facilitate writing client applications, optimal operation of the microscope hardware should be possible using a single threaded application. This puts some of the burden of optimal operation to the Device Adapters. Most functions in Device Adapters (with one notable exception, the "snap" function of the camera that should block for the duration of the exposure) are expected to be non-blocking and return as fast as possible. Depending on the specific driver this may necessitate operating a separate thread for device communication within the adapter (even when communicating with a device through a serial port, round trip communication can easily take tens of milliseconds, which preferably should not be taking place on the calling thread).

Because of the complexities of writing multi-threaded code, this is not the case in many of the current device adapters, potentially slowing down the application software. Future improvements can include moving the complexities of multi-threaded coding into the core to simplify writing optimally performing Device Adapters.

Device Interface

MMDevice is an abstract interface representing a distinct conceptual hardware component. The relationship between actual physical devices and MMDevice interfaces is not necessarily one to one. A single piece of hardware can be represented in Micro-Manager as multiple MMDevice interfaces (for instance, a Nikon TI microscope will appear as a nosepiece, reflector turret, z-drive, shutter, etc.., and conversely, multiple hardware components can be aggregated into a single MMDevice (for instance, multiple shutters can be combined into a single "multi-shutter" device). The key is to choose the right set of available abstractions and *device types* that best represents each particular hardware component within the Micro-Manager environment.

Each specific device interface such as Camera, Shutter, etc.. is derived from the base MMDevice interface. MMDevice defines methods that all devices must implement. We will consider each group of API methods separately below.

The currently supported Device types which extend the functionality of a generic Device are listed below:

- CameraDevice: A device that produces image data. Images are assumed to be 2-dimensional but may be multi-component (e.g. RGB) as well as multi-channel
- ShutterDevice: A device which can be set open or closed, most likely to block a light path
- StateDevice: A device which may be set to any one setting from a finite set of possible states. Examples include filter wheels and objective turrets
- StageDevice: A device for which a position can be set along a single continuous axis.

The API assumes that units of the position are expressed in microns.

- XYStageDevice: A device similar to the StageDevice except with 2 independent axes (X and Y)
- SerialDevice: A device representing a computer communication port such as a serial port
- AutoFocusDevice: A device control hardware to automatically focus an image or evaluate image sharpness and provide a focus score
- ImageProcessorDevice: A software device that takes images in, performs an operation on the image and returns the result
- SignalIODevice: A device that can measure and/or output a signal, e.g. a DAC or ADC on a microcontroller
- MagnifierDevice: A device whose setting may result in changes to pixel size of a camera
- SLMDevice: Spatial Light Modulator Device, a device that can display an "image" (i.e. has a (2D) array of elements that can be set to differ)
- HubDevice: A device which acts as a central hub for multiple peripheral devices. This is useful with multiple devices share a resource such as a serial port
- GalvoDevice: A device that can point to a specific 2D location

The device interface is designed to use only primitive types and return integer error codes rather than use C++ exceptions in order to allow packaging into separate dll units and guarantee correct operation across dll boundaries on all operating systems and compilers. There are a few exceptions to this rule where pointers to classes pass through the API as "naked" pointers. If we pass a pointer to a non-primitive type we must rely on static casting, and not on run-time type identification (RTTI) since RTTI is a compiler-dependent feature.

Basic

Implementation of these methods is required for all implementations of the MMDevice interface. They constitute the minimum functionality required in order to interface with MMCore.

int Initialize()

int Shutdown()

DeviceType GetType()

void GetName(char* name)

void SetCallback(Core* callback)

bool GetErrorText(int errorCode, char* errMessage)

void SetLabel(const char* label)

void GetLabel(char* name)

Initialize() and **Shutdown()** are deliberately separated from the constructor and destructor of the class so that actual connection and any hardware initialization can be controlled more effectively by the calling program. A Device object is expected to be able to handle multiple cycles of Initialize/Shutdown in the same instance, i.e. without having to destroy and create a new object. This helps with automatic configuration and interactive configuration building.

The **GetType()** function returns an enumeration representing the actual type of the device (Camera, Shutter, Stage, etc.) which can map into an integer constant in C/C++. This is needed in order to treat devices polymorphically enabling classification of the device into a more specific device subgroup which may be subject to special handling in a given application.

The **GetName()** function returns a string to be used to refer to the Device Adapter, something like a “class name”. This is not to be confused with “label” which refers to a specific instance, and is used by the parent application (and human user) to keep track of which device is which.

MMCore is the default container for devices, but any other application that implements the Core interface could take that role. MMCore passes its handle (pointer) using the **SetCallback()** method, providing a way for a Device Adapter to communicate with the Core. It is possible for a container to pass "null" or not call this method at all, in which case Devices should be able to work normally or degrade gracefully (but face a significant loss in capabilities).

See more about the MMCore Callback interface in the *dedicated section* below. The MMCore Callback API is exclusively for devices to call back into MMCore, whereas MMCore API is a top-level interface for applications and scripts.

Property

unsigned GetNumberOfProperties()

int GetProperty(const char* name, char* value)

int SetProperty(const char* name, const char* value)

bool HasProperty(const char* name)

bool GetPropertyName(unsigned idx, char* name)

int GetPropertyReadOnly(const char* name, bool& readOnly)

int GetPropertyInitStatus(const char* name, bool& preInit)

int HasPropertyLimits(const char* name, bool& hasLimits)

int GetPropertyLowerLimit(const char* name, double& lowLimit)

int GetPropertyUpperLimit(const char* name, double& hiLimit)

int GetPropertyType(const char* name, MM::PropertyType& pt)

unsigned GetNumberOfPropertyValues(const char* propertyName)

bool GetPropertyValueAt(const char* propertyName, unsigned index, char* value)

The Property part of the API is intended for dynamic discovery and manipulation of an array of name-value pairs ("properties"). Each device adapter defines the set of properties it exposes to the container, outside code can not add or remove properties.

We can discover how many properties there are at any time, their names, value types, value limits, and allowed values (if it is a discrete range). In theory, a device can choose to dynamically add and remove properties, there is no guarantee that a set of properties discovered at any time will remain in effect indefinitely. The container code, therefore, must be structured defensively. However, it is considered bad practice to change properties ad-hoc during normal operation.

A typical use case for dynamic property management is during the **Initialize()** routine described in the *previous section*. For example, let's say we instantiate a Camera device object. At that point the adapter did not establish communication with the hardware yet, so we don't know the actual capabilities of the specific camera model. Then we call **Initialize()** and that's where the actual connection happens. Inside this routine, we query the camera and discover its capabilities, e.g. which readout rates, resolutions and pixel types it supports, and so on. At that same place, we dynamically create properties to accommodate camera capabilities. Once the **Initialize()** routine is finished we will have a fully functional device exposing a list of properties based on the actual camera model that is currently connected to the software.

Generally, it is considered good practice to not change the list of available properties after the **Initialize()** routine is completed. There are some exceptions to this rule, e.g. when setting specific properties to specific values might change the range of allowed values for other properties. Several device interfaces do not fit neatly with this concept (for instance, several camera APIs expose different sets of properties depending on the setting of another property). Also, it is sometimes needed that properties are set in a certain order, whereas the Micro-Manage API requires that prop-

erties can be set in any order. We do believe that careful coding of a device adapter can work around such issues, but are open to ideas for improvement.

The interface could have been made much simpler if we followed the object-oriented paradigm and created a Property class. However, we are forced to keep the interface "primitive" as discussed at the beginning of the *Device Interface* section, therefore the verbosity of the property API.

One interesting aspect of the importance of **Initialize()** routine is that it becomes important to know which properties can be used before the device is initialized. Why do we want to call any functions or set any properties on a device that is not initialized? Because often there are some parameters that must be set before we can even attempt to initialize the device. For example, we may have to assign a serial (COM) port before a device that connects through a serial port can be initialized, and so on. **GetPropertyInitStatus()** tells us whether a specific property has to be set before attempting to **Initialize()**.

Properties can be "sequenced", i.e. a sequence of property states can be uploaded to the device, and once the sequence is started, the device will transition through these states driven by a trigger (which in practice, is most often a TTL signal on the device input). For more information, see the "Sequencing and Hardware synchronization" section below.

Busy

bool Busy()

double GetDelayMs()

void SetDelayMs(double delay)

bool UsesDelay()

Devices use the "busy" flag to signal that they are doing something significant. By testing this flag, client programs can discover whether some critical operation is in progress. For example, the stage would set a busy flag to true when it is moving and false when it is at rest. The client program can poll this flag to find out when the stage stopped moving, in order to proceed to the next operation such as capturing images.

Whenever possible, time consuming operations (such as moving a stage) should not be blocking the execution thread. When the client is operating multiple devices simultaneously (for instance setting the positions of two filter wheels, a reflector changer, and a Z drive), it should be possible to send the commands to start moving from a single thread, then poll using the "busy" flag, and continue once all devices report to be no longer busy. Depending on the device interface, it may be optimal for the device adapter to use a second thread to communicate with the device in order to avoid blocking the client thread.

The "Delay" functions historically were intended to be an alternative mechanism to "discover" when a device was no longer busy. However, in practice, these functions are exclusively used by devices that either do not know by themselves when their action is complete (for instance, for a filter wheel device that can only send a command to move to a certain position, but has no sensors to read-out its actual position to know when it is done moving), or by devices that themselves report to be no longer busy prematurely (for instance, an XY stage that jiggles around its target even though its controller reports that the destination has been reached), and setting a delay allows the user to overcome this problem. Thus, the "Delay" itself needs to be completely implemented by the Device Adapter code by noting a time stamp when the action starts (or in case an extra delay is needed, by noting the time stamp when the controller thinks the device is no longer Busy), and then in the Busy function call using this time stamp, the current time and the delay to determine whether the Busy function should return true or false.

Device Discovery

bool SupportsDeviceDetection(void)

MM::DeviceDetectionStatus DetectDevice(void)

Device detection is used to find out if the device is actually connected without running a full initial-

ization. It is most often used to detect the serial port to which the device is connected. If device detection is supported, a pre-initialization property (named g_Keyword_Port, defined as "Port" in MMDevice/MMDeviceConstants.h) containing the name of a communication port will be set, and the DetectDevice function will be executed for each available port. The Device adapter can use this function to set the appropriate properties of the port (for instance, baud rate of a serial port). Although this feature is helpful to the user (it can greatly facilitate hardware setup), it can also lead to problems by sending commands to unrelated devices that may stop communicating altogether.

Device Hierarchy

void SetParentID(const char* parentId)

void GetParentID(char* parentID)

Often, a single physical device contains multiple logical devices. For instance, a physical microscope may contain a shutter (Shutter Device), reflector changer (State Device), focus drive (Stage Device), and objective changer (State Device), all behind the same communication port. In such cases, it is desired to create a logical "Hub Device". Hub Devices can discover attached devices. The relation between the hub and detected (child) devices is maintained by the child devices (who maintain knowledge of the parent Hub device) and the Core container (GetParentHub()).

Sequencing and Hardware Synchronization

Due to latencies in software command execution, and communication with external devices, it is very difficult or impossible to achieve microsecond time-scale synchronization between devices in software alone. Such synchronization is possible when devices can respond (i.e. change position or state, or start an action) to TTL signals provided by, for instance, the camera (to signal sensor exposure), or an external clock (such as a National Instruments DAQ board or an Arduino microcontroller). Micro-Manager facilitates such work-flows using the concept of "Sequences".

A sequence is a finite, discrete sequence of states or actions on a particular device, each of which can optionally take arguments and data and produce data. For example, a sequence on a camera will result in the sequential exposure and readout of multiple images, and will place those images into a buffer. A Z-stage will take a position as an argument for each state in the sequence. A spatial light modulator will take a buffer (usually an image) to describe the pattern it will display at each step of the sequence

An important distinction is whether sequences will be triggered internally or externally, a distinction that is equivalent to "leader" vs. "follower" devices. For example, cameras can be asked to deliver a sequence of images, which will most often be accomplished by putting the camera into "internal trigger mode" so that it uses an internal clock to guarantee exact exposure times and minimal dead-time in between exposures. Most cameras in this mode will also produce TTL pulses at defined times relative to the start of exposure, so that follower devices can be synchronized to this clock. Alternatively, the camera can be set in "external trigger mode", in which it waits for an external trigger to start exposure of a new image.

The sequencing API contains functions checking whether a device is sequenceable, loading states of the sequence into the device, and starting the sequence. For instance, a (Z) stage can signal that it can respond to TTL signals using its "int IsStageSequenceable(bool& isSequenceable)" function. A sequence of positions is uploaded using the "AddToStageSequence(double position)" and "SendStageSequence()" functions. The Stage will start cycling through these positions, driven by the external TTL signal after the "StartStageSequence()" is called. Likewise, XY stages, DAs (analog output devices), and SLMs (spatial light modulator or projectors) have similar sequencing interfaces. Moreover, any property of any device can declare that it too can be sequenced. The interface is similar (a list of property values is created, sent to the device, and the sequence is started).

This sequencing interface enables multiple hardware configurations (camera as "leader", or external device as "leader"), and the code executing image acquisition

can query the hardware and figure out how to most optimally execute an acquisition sequence. However, it only works for devices that respond more or less instantaneously. Providing a mechanism for the device to provide feedback how long a transition will take would be a useful addition to the interface. Currently, there is no abstract notion of a device that generates the hardware timing pulses (TTLs). Such an abstraction of a hardware clock could also be beneficial.

Library Interface

void InitializeModuleData()

MM::Device* CreateDevice(const char* name)

void DeleteDevice(MM::Device* pDevice)

long GetModuleVersion()

long GetDeviceInterfaceVersion()

unsigned GetNumberOfDevices()

bool GetDeviceName(unsigned deviceIndex, char* name, unsigned bufferLength)

bool GetDeviceType(const char* deviceName, int* type)

bool GetDeviceDescription(const char* deviceName, char* name, unsigned bufferLength)

void RegisterDevice(const char* deviceName, MM::DeviceType deviceType, const char* description)

Each dynamically loaded library recognized by MMCore (i.e. files whose name starts with "mmgr_dal_" and with the extension ".dll", ".dylib", or ".so") needs to implement this interface. Most of these functions are implemented by boiler-plate code that, however, the device adapter must implement **InitializeModuleData()** in which it should call **RegisterDevice(deviceName, deviceType, description)** for all devices it supports, which lets MMCore auto-discover the available devices. In addition, the functions **CreateDevice(const char* name)** and **DeleteDevice(MM::Device* pDevice)** have to be implemented.

Device Types

The enumeration type declaration in MMDeviceConstants.h lists all currently available device types: CameraDevice, ShutterDevice, StateDevice, StageDevice, XYStageDevice, SerialDevice, GenericDevice, AutoFocusDevice, ImageProcessorDevice, SignalIODevice, MagnifierDevice, SLMDevice, HubDevice, and GalvoDevice.

Each device type provides a functional API - in addition to the basic MMDevice API - that consists of two parts:

1. Essential API, representing an abstraction for the common functionality associated with a particular device type. For example, a camera must implement a **SnapImage()** method while a translation stage must implement a **SetPositionUm(double position)** method.
2. Properties API, i.e any number of property-value pairs. This is how any particular device can extend its functionality beyond the Essential API. There are no mandatory properties and there is no minimum number of properties, 0 properties are OK as well as 1000 properties.

GenericDevice

This is a device with no essential API and it provides useful functionality only through its list of properties.

CameraDevice

Cameras are sensors that provide the computer with a 2-dimensional array of values representing light intensity, with a specified number of bytes (e.g. 1 or 2) per each pixel. Cameras can operate as a photo-camera, taking single snapshots, or as a movie camera, generating uninterrupted streams of images. The latter

operation mode is always much faster when sequences of images are desired.

Synchronization of camera exposure with other events (such as opening and closing of the shutter controlling the light source illuminating the sample) is extremely important. To this end, the camera device adapter implements the function **SnapImage()** that should start the exposure as soon as possible, block for the duration of the exposure, and then return as soon as possible. Often, read-out and transfer of data to the computer takes a significant amount of time, hence this process should not take place in the **SnapImage()** call, but in **GetImageBuffer()** (issues with synchronization regularly arise when this is not implemented correctly in the Device Adapter code). The **GetImageBuffer()** function returns a pointer to the image data. The size of this buffer will be returned in the function **GetImageBufferSize()**, which in turn should be consistent with values returned by **GetImageWidth()**, **GetImageHeight(),** and **GetImageBytesPerPixel()**. The camera should never change the size of the pixel buffer on its own. In other words, the buffer size can change only when camera features change (such as binning, pixel type, region of interest etc.). Color (RGB) cameras with 8 bits per color (16 bit per color is currently not fully supported) will return 4 for **GetImageBytesPerPixel().** They also will need to report to have 4 components in the function **GetNumberOfComponents().** Color images on little endian platforms (most of the current computers) should be organized as BGRA8888 (see: *https://en.wikipedia.org/wiki/RGBA_color_model*). Some special devices can deliver multiple images simultaneously. These are called "Channels" and their number can be found through the function **GetNumberOfChannels()** (1 for most cameras). Image data for each of the channels are retrieved with the function **GetImageBuffer(unsigned channelNr).**

The function **StartSequenceAcquisition (double intervalMs)** starts a sequence of images from the camera (i.e., the "movie camera" mode). The parameter "**intervalMs**" has not been implemented by any of the cameras supported by Micro-Manager and should be ignored. In most cases, the camera will switch to an internal trigger mode, and the interval between images will be highly reproducible (somewhat higher than the exposure time, depending on how fast the sensor is ready for the next exposure). The default implementation is to call **StartSequenceAcquisition(long numImages, double interval_ms, bool stopOnOverflow)** with **numImages** set to the maximum possible number, and **stopOnOverflow** to false. Images coming from the camera driver are to be copied by the Device Adapter code into the Core's circular buffer using the Core Callback function **InsertImage.** When this function indicates that the circular buffer overflowed, it is the Device Adapter's responsibility to either stop the sequence (when **stopOnOverflow** is true), or to clear the circular buffer and continue. Metadata can optionally be attached to the images before insertion in the circular buffer. Cameras that have the capability to use different exposures times during a sequence, can use the **ExposureSequence** functions to do so.

Cameras can start exposure triggered either by a signal from the computer ("software trigger"), an electrical signal coming from another device ("external trigger"), or an internal clock ("internal trigger"). Exposure time can be set in software (and controlled by an internal clock), or be determined by the duration of the external signal. A sequence can be started by an external trigger, or each image in a sequence can be triggered by an external device. Currently, no API exists for camera triggering. Trigger modes can be set through properties, however, changes in trigger behavior when switching between "Snap" and "Sequence" mode are not prescribed but left to the device adapter code, leading to opportunities for unexpected differences in behavior between cameras. Formalizing trigger modes in the API will be a worthwhile extension.

In practice, camera device adapters are the most complex device adapters to code, making the large number of supported cameras (currently 57 camera device adapters) even more remarkable.

ShutterDevice

A shutter device is a device with two states: open and closed. It is mainly used for illumination control, but could be anywhere in the microscope's light path. Its interface is simple: **SetOpen(bool open)**, and **GetOpen(bool& open)**. The interface function **Fire(double deltaT)**, is implemented by so few devices that it probably should be removed.

StateDevice

A state device is a device that at any point in time is in a single state out of a list of possible states, like a filter wheel, an objective turret, etc.. The interface contains functions to get and set the state (**GetPosition(long& pos)**, **SetPosition(long pos)**), to give states human readable labels (**SetPositionLabel(long pos, const char* label)**, **GetPositionLabel(long pos, char* label)**), and functions to make it possible to treat the state device as a shutter (**SetGateOpen(bool open)**, **GetGateOpen(bool& open)**).

StageDevice

The stage device interface has functions to get and set the position of the stage, both in microns (**SetPositionUm(double pos)**, **GetPositionUm(double& pos)**) and in steps. The idea here is that the stage has a "native" coordinate system that it should translate to movement in microns. In addition, functions are provided to start moving at a certain velocity (**Move(double velocity)**) and to stop (**Stop()**). A given position can be set to be the zero position for the device adapter (**SetAdapterOriginUm(double d)**) or for the device itself (**SetOrigin()**). There is no expectation whether movement is towards or away from the sample, but the stage can report directionality (**GetFocusDirection(FocusDirection& direction)**). Functions are provided to upload sequences of positions to the stage for fast, externally triggered transitions between positions.

XYStageDevice

An XY stage is more or less the 2-dimensional version of a stage device. However, the microscope system expects a certain directionality of the stage. When a sample is placed on the stage, it is expected that the lowest numeric position is in the top left corner. Since such directionality can not be expected to be implemented natively in the stage or stage driver, "translation" functions are provided in the DeviceBase.h file that translate the commands to set and get positions in microns to positions in steps, given three standardized properties signaling directionality. Although this system works, it is difficult to understand, places system-level responsibility in the device adapter layer, and is incomplete, as it does not correct for all possible transformations, and therefore should be re-considered.

SerialDevice

Probably more aptly named "CommunicationDevice", but Serial Devices were the first to be implemented and the architecture is geared towards serial communication ports. Functions are provided to send (ASCII) characters to the port (**SetCommand(const char* command, const char* term)**), to receive characters (**GetAnswer(char* txt, unsigned maxChars, const char* term))**, to send binary (**Write(const unsigned char* buf, unsigned long bufLen)**) and to receive binary (**Read(unsigned char* buf, unsigned long bufLen, unsigned long& charsRead)**). Implementations are available for serial ports, certain USB devices with libusb drivers, certain hidapi devices, and TCP/IP ports. Port settings are communication through standardized properties.

AutofocusDevice

Devices that automatically can find and/or maintain optimal focus of the sample in the optical system. These can either be devices that maintain focus continuously (often by measuring light invisible to the detector and reflected near the sample), or devices that can do a "one shot" focus. Autofocus devices

have become more and more widely used through the last decade, and a clear generic interface to such devices is difficult to design. The current interface contains functions for continuously focussing devices (**SetContinuousFocusing(bool state)**, **GetContinuousFocusing(bool& state)**, **IsContinuousFocusLocked()**), and functions for "one shot" devices (**FullFocus()**, **IncrementalFocus()**) that can be implemented by continuously focussing devices by switching continuous focus on until a lock is achieved. In addition, there are functions to get focus scores and to get and set offsets, that are inspired by certain autofocus devices, but that may better be abstracted in other ways (for instance, the Nikon Perfect Focus Offset is implemented as a Stage Device - even though its "positions" can not be related to microns- that is easier to work with in the Micro-Manager user interface).

Re-evaluation of the autofocus device interface will be worthwhile.

ImageProcessorDevice

A software only device that can change image data. It sole unique function is **Process(unsigned char* buffer, unsigned width, unsigned height, unsigned byteDepth)** that allows the device to change the (pixel) data in the provided buffer. The idea is that such ImageProcessorDevices can modify pixel data with the C++ layer at high efficiency, before sending them on to the upper software layers. The concept never received much traction and few if any such devices exist, and this device can probably be removed or replaced by a better version.

SignalIODevice

Probably better named AnalogOutDevice, this device can set (**SetSignal(double volts)**) and get (**GetSignal(double& volts)**) analog signals, where it is unclear if the latter is supposed to provide the output voltage that is actually applied, or is a way to read input voltages. Like a StateDevice, the output signal can be used as a shutter using the **SetGateOpen(bool open)** and **GetGateOPen(bool& open)** functions. In addition, output signals can be sequenced, i.e. a series of values can be uploaded to the device, and transition from one to the next can be triggered with a TTL pulse.

We are planning to deprecate the SignalIODevice and replace it with a dedicated device for analog output, and a dedicated device for analog input. Both will need careful thought and planning, for instance, the analog output device interface may need to support arbitrary waveforms, and the analog input device may need to support reading data streams, which could include the necessity to buffer such data streams.

MagnifierDevice

A magnifier device brings magnification into the optical system. Examples are motorized zoom lenses in low mag microscopes. The amount of magnification can be queried using the **double GetMagnification()** function and can be used by a higher software layer to calculate the current magnification/physical pixel size.

SLMDevice

A Spatial Light Modulator (SLM) is essentially treated as a display, i.e. a pixelated device onto which an image can be written. Sequences of images can be loaded into the device, and transitions between triggered by an external signal. SLMs (displays) with an internal light source can have an "exposure time", i.e. display of an image will only happen for the duration of the exposure time.

GalvoDevice

The interface to a Galvo device is highly targeted to the use of a galvo as a pointing device in Fluorescence Recovery After Photobleaching (FRAP) experiments. The main functions "point" in a 2D space (**SetPosition(double x, double y)**, **GetPosition(double& x, double& y)**). Integration with a light source controller is (perhaps regretfully) assumed, hence functions like **PointAndFire(double x, double y, double time_us)**, **SetSpotInterval(double pulseInterval_us)**, and **SetIllumina-**

tionState(bool on). A number of functions to add polygons are present. In general, this interface seems highly targeted to one or a few specific implementations of a FRAP device, but does not offer all functionality associated with Galvo devices, such as the ability to apply waveforms, and synchronize movement with other signals. Careful re-design of the interface to Galvo devices (or perhaps even only to a device for analog voltage outputs) seems prudent.

HubDevice

A HubDevice is a container for several other devices sharing certain infrastructure. For instance, an automated microscope, connected through a single USB cable to the computer and containing filter turrets, a Z-driver, objective turrets and other devices is an example of a Hub device.

MMCore Callback interface

Communication from a device back to higher software layers happens through the Core Callback object. A pointer to the Core Callback object is provided immediately after loading the Device Adapter. We discuss the current implementation of this Core Callback object in the Micro-Manage code base, but want to stress that other implementations are possible, and that Device Adapters should be able to operate without this callback object. With the Core object, a Device Adapter can:

- Log messages. The message will be added to the applications logging output, maintained by a logging facility implemented in MMCore
- Get pointers to other devices. These can be used to send commands to other devices, and are the basis for all the “Utilities” devices that “translate” one type of device into another type (such as the “StateDeviceShutter” device), or that combine multiple devices into one (such as the “MultiCamera” device). Even though this functionality was clearly needed, there is risk in direct inter-device communication, and it may be worthwhile to brainstorm about other approaches.
- Send and receive commands through a (serial) port device.
- Signal to upper software layers that something in the device (such as a property, state, stage position, etc..) changed. These callbacks are used in Micro-Manager to update a cache of the state of the microscope, and sent to the User Interface so that it can reflect these changes. Often, these callbacks run on their own thread (for instance, the user pushes a button on the microscope that results in a thread in the Device Adapter receiving the state change, which then calls back to the Core object), which needs to be taken into account when coding the upper layers of an application using such callbacks.
- Signal that a sequence acquisition is starting or is finished. Since a (camera) sequence takes place in its own thread, the calling code (MMCore) does not know when the camera stops acquiring. The callback **AcqFinished(const Device* caller, int statusCode)** allows the core to take needed actions (such as closing a shutter) when the camera sequence ends. Likewise, the function **PrepareForAcq(const Device* caller)** is used to synchronice shutter opening with sequence acquisition start.
- Manage and copy data into an image buffer (relevant to camera devices only). The device can initialize and clear an image buffer, as well as copy data into that buffer. In the current implementation, this is a single buffer, shared by all cameras. The MMDevice API interface definition does not preclude each camera to have its own data buffer.
- Interact with groups of properties maintained in the MMCore object (see below). The device adapter can apply a certain con-

figuration, and enquire about the current configuration.

MMCore API and implementation

The "MMCore" object is the implementation of the container for device adapters provided in Micro-Manager. Other implementations are possible (although we are not aware of any), here we discuss the features provided by this particular implementation.

The MMCore object makes the MMCore API available to higher level application code. The MMCore API provides the following facilities:

Device management

MMCore can list all available device modules in multiple given locations, as well as the devices contained in these modules. Devices can be loaded and initialized, shut-down and unloaded. Although loading and unloading of individual Device Adapters can be executed through MMCore API calls, device loading is most often specified in a configuration file that is read-in and parsed by MMCore (see below).

"Pass-through" operations

Many functions pass-through directly to the corresponding function in a Device Adapter. For instance, MMCore function **deviceBusy(const char* label)** looks up the device with name "label", and calls the function **Busy()** in the corresponding Device Adapter. This intermediate software layer is a safety feature (in principle only the core has pointers to the actual instances of the device adapters), and makes it possible to implement some level of device and thread synchronization. Currently, per-device thread locks are implemented.

In some cases, optional software synchronization is offered by MMCore. When the "AutoShutter" capability is enabled, the default shutter device will be opened before camera exposure, and closed thereafter.

Core properties and default roles

MMCore defines a number of its own properties, most of which concern themselves with default roles for devices. For instance, when multiple camera device adapters are loaded, the property "Core-Camera" determines which of these available cameras will be used when the **snapImage()** function is called. Thus to snap an image with another camera, the application code first needs to set the Core-Camera property to the value of that other camera before calling **snapImage()**. Some devices that follow the "default" paradigm also can be called directly (for instance one can set "Core-Shutter" and call **setShutterOpen(bool state)**, or call **setShutterOpen(const char* shutterLabel, bool state)**), but others (most notably "Camera") can not (this would be trivial to add if so desired).

Another important Core property is "TimeoutMs", that determines how long MMCore will wait for a device to stop being "Busy", before returning a timeout error.

Grouping of properties

Gettings and setting individual properties quickly becomes cumbersome. For a user it often makes sense to assemble properties in a group. For instance, many microscopes switch "channels" by changing the position of multiple filter wheels and reflector turrets. The MMCore API names a group of device properties a "Config", and a group of configs a "ConfigGroup" (where all Configs in a ConfigGroup usually contain the same set of Properties albeit with different Property values). These ConfigGroups map directly to the "Configuration settings" "Group" and "Preset" table in the main window of the Micro-Manager UI. ConfigGroups and Configs can be stored in a configuration file.

Pixel sizes and affine transforms

A special type of ConfigGroup concerns itself with pixel size (the size of the current's camera pixel in the sample plane), as well as the relationship between XY stage movement and the camera (as de-

fined by an affine transform). This information allows the MMCore API to provide the current pixel size. When asked (**double getPixelSizeUm()**) the MMCore API implementation will check if the current state matches any of the PixelSize Config groups, and provide the pixel size corrected for potential binning of the camera and - when present - for magnification by Magnifier devices in the system.

Caching

Asking Device Adapters repeatedly about the current state of all their properties can induce undesirable time-consuming communication between the computer and device. MMCore therefore maintains a cache with the last known state of all properties it is aware of. The client application determines whether to use information from the cache or to force re-enquiring with the device (for instance: **getPixelSizeUm(bool cached)**, **getCurrentConfigFromCache(const char* groupName)**, **getPropertyFromCache(const char* deviceLabel, const char* propName)**).

In some sense, the caching mechanism is redundant, since certain Device Adapters provide caching themselves (for instance, the Zeiss, Nikon Ti and Ti2, and Leica device adapters that maintain internal models of the state of the microscope), so having yet a second layer of caching is not optimal. This could potentially be avoided by giving Device Adapters the capability to signal whether they maintain their own cache of the device's state.

Logging

MMCore implements an asynchronous logging facility that is available to MMCore itself, scripts/applications using MMCore, and to the Device Adapters through the Core Callback interface. Log output is written to an application-specified file location. This logging works very well, but it can be imagined that applications already implementing their own logging mechanism would want MMCore logging output to be available through a callback function rather than directly written to the file system.

Configurations

Microscope "configurations" are stored in a human-readable configuration file. When MMCore is instructed to parse such a file, it treats it as instructions to change its state. For instance, a line starting with the word "Device" will be treated as an instruction to load the specified device from the specified Device Adapter ("Device,Dichroic,DemoCamera,DWheel" loads the device DWheel from the Device Adapter "DemoCamera" and makes it available under the name "Dichroic"). Lines are executed in order, so the line "Property,Core,Initialize,1" leads to initialization of all devices that were loaded so far (a step that needs to take place before properties from the device can be used).

The keyword "Parent" is used to make devices that are part of a "Hub" device aware of their "Hub" parent. Labels can be assigned to the individual states of State Devices ("Label,Dichroic,1,Q505LP" specifies that position/state 1 of the State Device "Dichroic" is labeled Q505LP"). ConfigGroups and Presets are stored in the Configuration file (under keyword "ConfigGroup") as are pixel size configurations, pixel size, and affine transforms relating camera orientation to stage movement.

All actions taking place when MMCore parses a configuration file could be executed through API function calls, but it is more convenient to execute all tasks needed to get the software ready to operate on a given microscope system by parsing a single file.

Discussion

The large number of Device Adapters available in the Micro-Manager ecosphere provides a unique opportunity to create a truly universal software interface to microscope components, greatly facilitating development of software needed for new types of microscopes. By reviewing the existing situation we have identified several shortcomings of the current MMDevice API as well as its implementation that could be readily addressed. We also recognized that the MMCore layer is

strongly targeted to the 2005 paradigm of a motorized microscope. One interesting path forward could be to modularize the MMCore layer further, splitting it up into independent units, one each for device control (i.e. a pure device abstraction layer), device synchronization, data management and saving, configuration storage, logging, metadata handling, and perhaps others. This would allow an application developer to pick and choose the relevant parts of MMCore, make future development more manageable, yet maintain compatibility with existing client applications that use MMCore. We will continue discussion of these ideas elsewhere.

References

[1] Arthur Edelstein, Nenad Amodaj, Karl Hoover, Ron Vale, and Nico Stuurman. Computer control of microscopes using μmanager. *Curr Protoc Mol Biol*, Chapter 14:Unit14.20, 2010.

[2] Arthur D Edelstein, Mark A Tsuchida, Nenad Amodaj, Henry Pinkard, Ronald D Vale, and Nico Stuurman. Advanced methods of microscope control using μmanager software. *Journal of Biological Methods*, 1(2):10, 2014.

[3] Catherine B. Carbone, Ronald D. Vale, and Nico Stuurman. An acquisition and analysis pipeline for scanning angle interference microscopy. *Nature Methods*, 13(11):897–898, 2016.

[4] Friedrich Walter Schenk, Nicolai Brill, Ulrich Marx, Daniel Hardt, Niels König, and Robert Schmitt. High-speed microscopy of continuously moving cell culture vessels. *Scientific Reports*, 6:34038, 2016.

[5] Iliya Sigal, Margaret M. Koletar, Dene Ringuette, Raanan Gad, Melanie Jeffrey, Peter L. Carlen, Bojana Stefanovic, and Ofer Levi. Imaging brain activity during seizures in freely behaving rats using a miniature multi-modal imaging system. *Biomedical Optics Express*, 7(9):3596, 2016.

[6] Zhicheng Long, Eileen Nugent, Avelino Javer, Pietro Cicuta, Bianca Sclavi, Marco Cosentino Lagomarsino, and Kevin D. Dorfman. Microfluidic chemostat for measuring single cell dynamics in bacteria. *Lab on a Chip*, 13(5):947, 2013.

[7] Zhengyi Yang, Peter Haslehurst, Suzanne Scott, Nigel Emptage, and Kishan Dholakia. A compact light-sheet microscope for the study of the mammalian central nervous system. *Scientific Reports*, 6:26317, 2016.

[8] Xiaowei Yan, Nico Stuurman, Susana A. Ribeiro, Marvin E. Tanenbaum, Max A. Horlbeck, Christina R. Liem, Marco Jost, Jonathan S. Weissman, and Ronald D. Vale. High-content imaging-based pooled CRISPR screens in mammalian cells. *Journal of Cell Biology*, 220, 2021.

[9] Bin Yang, Merlin Lange, Alfred Millett-Sikking, Ahmet Can Solak, Shruthi Vijay Kumar, Wanpeng Wang, Hirofumi Kobayashi, Matthew N. McCarroll, Lachlan W. Whitehead, Reto P. Fiolka, Thomas B. Kornberg, Andrew G. York, and Loic A. Royer. High-resolution, large imaging volume, and multi-view single objective light-sheet microscopy. *bioRxiv*, page 2020.09.22.309229, 2020. Publisher: Cold Spring Harbor Laboratory Section: New Results.

[10] Bin Yang, Xingye Chen, Yina Wang, Siyu Feng, Veronica Pessino, Nico Stuurman, Nathan H. Cho, Karen W. Cheng, Samuel J. Lord, Linfeng Xu, Dan Xie, R. Dyche Mullins, Manuel D. Leonetti, and Bo Huang. Epi-illumination SPIM for volumetric imaging with high spatial-temporal resolution. *Nature Methods*, 16(6):501–504, 2019. Number: 6 Publisher: Nature Publishing Group.

[11] Zachary R. Fox, Steven Fletcher, Achille Fraisse, Chetan Aditya, Sebastián Sosa-Carrillo, Sébastien Gilles, François Bertaux, Jakob Ruess, and Gregory Batt. MicroMator: Open and flexible software for reactive microscopy. *bioRxiv*, page 2021.03.12.435206, 2021. Publisher: Cold Spring Harbor Laboratory Section: New Results.

[12] Peter T. Brown, Rory Kruithoff, Gregory J. Seedorf, and Douglas P. Shepherd. Multicolor

structured illumination microscopy and quantitative control of polychromatic coherent light with a digital micromirror device. *bioRxiv*, page 2020.07.27.223941, 2020. Publisher: Cold Spring Harbor Laboratory Section: New Results.

[13] Henry Pinkard, Nico Stuurman, Ivan E. Ivanov, Nicholas M. Anthony, Wei Ouyang, Bin Li, Bin Yang, Mark A. Tsuchida, Bryant Chhun, Grace Zhang, Ryan Mei, Michael Anderson, Douglas P. Shepherd, Ian Hunt-Isaak, Raymond L. Dunn, Wiebke Jahr, Saul Kato, Loïc A. Royer, Jay R. Thiagarajah, Kevin W. Eliceiri, Emma Lundberg, Shalin B. Mehta, and Laura Waller. Pycromanager: open-source software for customized and reproducible microscope control. *Nature Methods*, 18(3):226–228, 2021. Number: 3 Publisher: Nature Publishing Group.

[14] Caroline A. Schneider, Wayne S. Rasband, and Kevin W. Eliceiri. NIH image to ImageJ: 25 years of image analysis. *Nature Methods*, 9(7):671–675, 2012. Number: 7 Publisher: Nature Publishing Group.