RPi-DeviceAdapters Documentation

Kyle M. Douglass

Feb 16, 2019

Contents:

Micro-Manager device adapters for the Raspberry Pi

CHAPTER 1

User Tutorial

This tutorial is an introduction to using the RPi-DeviceAdapters Docker-based application. The application is built around the Micro-Manager Python wrapper and will show you how to run a simple script that communicates with the Micro-Manager core.

The steps in this tutorial should be executed either in a terminal running directly on a Raspberry Pi or through ssh. It is assumed that the Raspberry Pi is running a recent version of the Raspbian operating system, though the steps listed here may work on other Linux-based operating systems as well.

1.1 Prerequisites

Begin by opening a terminal window (also known as a shell).

If you do not already have Docker installed on your Raspberry Pi, you may install it by running the command:

```
$ curl -sSL https://get.docker.com | sh
```

Follow the steps described in the installation script. After the installation has finished, you may optionally add your user to the Docker group so that you do not need to enter *sudo* before running Docker commands.

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
```

Log out and log back in for the changes to take effect.

You will also likely want to grab the venv package for Python from the Raspbian package manager.

```
$ sudo apt-get install python3-venv
```

Create a new virtual environment for the application to isolate it from the rest of your system:

```
$ python3 -m venv ~/venvs/mm
```

~ corresponds to your home folder; you may instead replace *~/venvs/mm* with any directory that you wish. Next, activate the virtual environment:

```
$ source ~/venvs/mm/bin/activate
```

You should see the name of the venv (in this case, *mm*) at the start of the command line. Whenever you want to stop working on the project, type *deactivate*. To reactivate the venv, simply rerun the command above.

1.2 Installation

To install RPi-DeviceAdapters into the venv, run the following command:

```
$ pip install tacpho.adapters
```

It will be assumed throughout the rest of this tutorial that you have added your user to the Docker group. (See the previous section for details.)

Next, download the latest Docker image of the application:

```
$ mm.py pull
```

This command may take several minutes before it completes as it downloads the application from DockerHub. *mm.py* is a convenience script for interacting with RPi-DeviceAdapters' Docker resources. To see its help message, type

```
$ mm.py --help
```

1.3 Execute a script

Open your text editor and enter the following code:

```
import MMCorePy

mmc = MMCorePy.CMMCore()
mmc.loadDevice('tutorial', 'RPiTutorial', 'RPiTutorial')
mmc.initializeAllDevices()

print(mmc.getProperty('tutorial', 'Switch On/Off'))

mmc.setProperty('tutorial', 'Switch On/Off', 'Off')
print(mmc.getProperty('tutorial', 'Switch On/Off'))
```

Save the text to a file named *tutorial.py*. This is just a short Python script that uses the Micro-Manager Python API to load the tutorial device adapter. It will report the value of a "switch", flip its value, and then print the new value.

To run the tutorial, enter the following command from the same folder that contains the script you just saved (be sure that the virtual environment in which you installed tacpho.adapters is active).

```
$ mm.py run tutorial.py
```

You should see the output from the script appear in your console.

1.4 Next steps

Example scripts for other device adapters may be found in the examples folder of the RPi-DeviceAdapters root directory. Check out the Micro-Manager documentation on its Python interface for more information about interacting with the Micro-Manager core.

Do not forget to update the RPi-DeviceAdapters application when new versions and device adapters become available by running *mm.py pull*.

CHAPTER 2

Developer Tutorial

This tutorial will demonstrate how to use RPi-DeviceAdapters to write, build, and deploy a simple Micro-Manager device adapter for the Raspberry Pi on a laptop or desktop. RPi-DeviceAdapters provides these capabilities to make development easier; you do not need to develop new device adapters directly on the Raspberry Pi.

2.1 Linux

2.1.1 Requirements

- Git
- Subversion (for the Micro-Manager dependencies)
- Docker
- Docker Compose
- Make
- QEMU

QEMU installation

The QEMU emulator is used to emulate a ARM processor architecture on a x86_64 system. It is setup as follows:

On Ubuntu, install the emulation packages with the commands:

```
$ sudo apt update
$ sudo install qemu qemu-user-static qemu-user binfmt-support
```

If you are not using Ubuntu, search for and install these packages in your system's respective package manager. Next, register QEMU in the build agent:

```
$ docker run --rm --privileged multiarch/qemu-user-static:register --reset
```

2.1.2 Setup

Begin by opening a shell, cloning the RPi-DeviceAdapters repository, and navigating inside the root directory of the cloned repository.

```
$ # For HTTPS, use https://github.com/kmdouglass/RPi-DeviceAdapters.git
$ git clone git@github.com:kmdouglass/RPi-DeviceAdapters.git
$ cd RPi-DeviceAdapters
```

Inside you will find a folder named *ci* (for continuous integration). This folder contains all the tools necessary for developing a new device adapter.

Next, we use the ci/prebuild.sh script to checkout the Micro-Manager source code and 3rdpartypublic dependencies. These will be placed into a directory named */opt/rpi-micromanager*. It is required by the build tool's docker-compose.yml file to place the development files here; let's first create it and set its ownership:

```
$ sudo mkdir -p /opt/rpi-micromanager
$ sudo chown $USER:$USER /opt/rpi-micromanager
```

If you do not want to place the source code in this directory, then you can either:

1. create a symlink at */opt/rpi-micromanager* that points to your alternative directory, or
2. modify docker-compose.yml to point towards your alternative directory.

The build container uses ccache to decrease the build time. ccache requires that there be a directory in your *$HOME* folder named *.ccache* to store the cached artifacts; it will automatically be created for you if it does not already exist when you run the prebuild script.

Let's run the prebuild script now:

```
$ ci/prebuild.sh /opt/rpi-micromanager
```

This step usually takes a few minutes due to the large size of the 3rdpartypublic repository. After it has completed, you should find the following inside */opt/rpi-micromanager*:

```
$ tree -L 1 /opt/rpi-micromanager
/opt/rpi-micromanager
├── 3rdpartypublic
└── micro-manager
```

2.1.3 Writing device adapters

Writing a general purpose Micro-Manager device adapter is outside the scope of this tutorial; help may be found on the Micro-Manager website and the mailing list. Here we discuss how to build a simple device adapter for the Raspberry Pi. The device adapter will have a single property that can be switched between two states: *on* and *off*.

Navigate to the device adapters folder inside the RPi-DeviceAdapters folder.

```
$ cd src/DeviceAdapters
```

For the sake of this tutorial, delete the folder named *RPiTutorial*. We will recreate it and its contents next.

```
$ rm -rf RPiTutorial

# Recreate the (empty) folder
$ mkdir RPiTutorial
```

Next, create three empty files named *RPiTutorial.h*, *RPiTutorial.cpp*, and *Makefile.am* inside this folder.

```
$ cd RPiTutorial
$ touch RPiTutorial.h RPiTutorial.cpp Makefile.am
```

With your text editor, open the file named **RPiTutorial.h**, and enter the following code:

```
/**
 * Kyle M. Douglass, 2018
 * kyle.m.douglass@gmail.com
 *
 * Tutorial Micro-Manager device adapter for the Raspberry Pi.
 */

#ifndef _RASPBERRYPI_H_
#define _RASPBERRYPI_H_

#include "DeviceBase.h"

class RPiTutorial : public CGenericBase<RPiTutorial>
{
public:
  RPiTutorial();
  ~RPiTutorial();

  // MMDevice API
  int Initialize();
  int Shutdown();

  void GetName(char* name) const;
  bool Busy() {return false;};

  // Settable Properties
  // -------------------
  int OnSwitchOnOff(MM::PropertyBase* pProp, MM::ActionType eAct);

private:
  bool initialized_;
  bool switch_;
};

#endif //_RASPBERRYPI_H_
```

The most important method defined in this header file is *OnSwitchOnOff(MM::PropertyBase* pProp, MM::ActionType eAct)*, which is the callback method that is called whenever the switch is flipped. The internal state of the switch is stored in the private variable *switch_*. All other methods are required by the *CGenericBase* API.

Now let's implement the switch. Open the file **RPiTutorial.cpp** and enter the following lines:

```
/**
 * Kyle M. Douglass, 2018
 * kyle.m.douglass@gmail.com
```

(continues on next page)

(continued from previous page)

```
 *
 * Tutorial Micro-Manager device adapter for the Raspberry Pi.
 */

#include "RPiTutorial.h"
#include "ModuleInterface.h"

using namespace std;

const char* g_DeviceName = "RPiTutorial";

////////////////////////////////////////////////////////////////////////////
// Exported MMDevice API
////////////////////////////////////////////////////////////////////////////

/**
 * List all supported hardware devices here
 */
MODULE_API void InitializeModuleData()
{
  RegisterDevice(
    g_DeviceName,
    MM::GenericDevice,
    "Control of the Raspberry Pi GPIO pins."
  );
}

MODULE_API MM::Device* CreateDevice(const char* deviceName)
{
  if (deviceName == 0)
     return 0;

  // decide which device class to create based on the deviceName parameter
  if (strcmp(deviceName, g_DeviceName) == 0)
  {
     // create the test device
     return new RPiTutorial();
  }
  // ...supplied name not recognized
  return 0;
}

MODULE_API void DeleteDevice(MM::Device* pDevice)
{
  delete pDevice;
}

////////////////////////////////////////////////////////////////////////////
// RPiTutorial implementation
// ~~~~~~~~~~~~~~~~~~~~~~~~~~

/**
 * RPiTutorial constructor.
 *
 * Setup default all variables and create device properties required to exist before
 * intialization. In this case, no such properties were required. All properties will
↪be created in
```

(continues on next page)

(continued from previous page)

```
 * the Initialize() method.
 *
 * As a general guideline Micro-Manager devices do not access hardware in the the constructor. We
 * should do as little as possible in the constructor and perform most of the initialization in the
 * Initialize() method.
 */
RPiTutorial::RPiTutorial() :
  initialized_ (false),
  switch_ (true)
{
  // call the base class method to set-up default error codes/messages
  InitializeDefaultErrorMessages();
}

/**
 * RPiTutorial destructor.
 *
 * If this device used as intended within the Micro-Manager system, Shutdown() will be always
 * called before the destructor. But in any case we need to make sure that all resources are
 * properly released even if Shutdown() was not called.
 */
RPiTutorial::~RPiTutorial()
{
  if (initialized_)
     Shutdown();
}

/**
 * Obtains device name.  Required by the MM::Device API.
 */
void RPiTutorial::GetName(char* name) const
{
  // We just return the name we use for referring to this device adapter.
  CDeviceUtils::CopyLimitedString(name, g_DeviceName);
}

/**
 * Intializes the hardware.
 *
 * Typically we access and initialize hardware at this point. Device properties are typically
 * created here as well. Required by the MM::Device API.
 */
int RPiTutorial::Initialize()
{
  if (initialized_)
    return DEVICE_OK;

  // set property list
  // -----------------
  // Name
  int ret = CreateStringProperty(MM::g_Keyword_Name, "RPiTutorial device adapter", true);
```

(continues on next page)

(continued from previous page)

```
111   assert(ret == DEVICE_OK);
112
113   // Description property
114   ret = CreateStringProperty(MM::g_Keyword_Description, "A test device adapter",␣
      ↪true);
115   assert(ret == DEVICE_OK);
116
117   // On/Off switch
118   CPropertyAction* pAct = new CPropertyAction (this, &RPiTutorial::OnSwitchOnOff);
119   CreateProperty("Switch On/Off", "Off", MM::String, false, pAct);
120   std::vector<std::string> commands;
121   commands.push_back("Off");
122   commands.push_back("On");
123   SetAllowedValues("Switch On/Off", commands);
124
125   // synchronize all properties
126   // --------------------------
127   ret = UpdateStatus();
128   if (ret != DEVICE_OK)
129     return ret;
130
131   initialized_ = true;
132   return DEVICE_OK;
133 }
134
135 /**
136  * Shuts down (unloads) the device.
137  *
138  * Ideally this method will completely unload the device and release
139  * all resources. Shutdown() may be called multiple times in a row.
140  * Required by the MM::Device API.
141  */
142 int RPiTutorial::Shutdown()
143 {
144   initialized_ = false;
145   return DEVICE_OK;
146 }
147
148 /**
149  * Callback function for on/off switch.
150  */
151 int RPiTutorial::OnSwitchOnOff(MM::PropertyBase* pProp, MM::ActionType eAct)
152 {
153   std::string state;
154   if (eAct == MM::BeforeGet) {
155     if (switch_) { pProp->Set("On"); }
156     else { pProp->Set("Off"); }
157   } else if (eAct == MM::AfterSet) {
158       pProp->Get(state);
159       if (state == "Off") { switch_ = false; }
160       else if (state == "On") { switch_ = true; }
161       else { return DEVICE_ERR; }
162   }
163
164   return DEVICE_OK;
165 }
```

Most of this code is boilerplate, i.e. code that is required by the MMDevice API but that does not directly affect the functionality that the user sees. The property that implements the On/Off switch is created here:

```
  CPropertyAction* pAct = new CPropertyAction (this, &RPiTutorial::OnSwitchOnOff);
  CreateProperty("Switch On/Off", "Off", MM::String, false, pAct);
  std::vector<std::string> commands;
  commands.push_back("Off");
  commands.push_back("On");
  SetAllowedValues("Switch On/Off", commands);
```

Its switching behavior is defined here:

```
/**
 * Callback function for on/off switch.
 */
int RPiTutorial::OnSwitchOnOff(MM::PropertyBase* pProp, MM::ActionType eAct)
{
  std::string state;
  if (eAct == MM::BeforeGet) {
    if (switch_) { pProp->Set("On"); }
    else { pProp->Set("Off"); }
  } else if (eAct == MM::AfterSet) {
     pProp->Get(state);
     if (state == "Off") { switch_ = false; }
     else if (state == "On") { switch_ = true; }
     else { return DEVICE_ERR; }
  }

  return DEVICE_OK;
}
```

Now, open *Makefile.am* and add the following lines:

```
AM_CXXFLAGS = $(MMDEVAPI_CXXFLAGS)
deviceadapter_LTLIBRARIES = libmmgr_dal_RPiTutorial.la
libmmgr_dal_RPiTutorial_la_SOURCES = RPiTutorial.cpp RPiTutorial.h \
   ../../MMDevice/MMDevice.h ../../MMDevice/DeviceBase.h
libmmgr_dal_RPiTutorial_la_LIBADD = $(MMDEVAPI_LIBADD)
libmmgr_dal_RPiTutorial_la_LDFLAGS = $(MMDEVAPI_LDFLAGS)
```

This file instructs Autotools how to create the Makefile when the code is compiled.

2.1.4 Building the libraries

To build the Micro-Manager core and device adapter that we just wrote, we first need to add RPiTutorial to the list of device adapters in *src/DeviceAdapters/Makefile.am* and *src/DeviceAdapters/configure.ac*. Here is what *Makefile.am* looks like:

```

AUTOMAKE_OPTIONS = foreign
ACLOCAL_AMFLAGS = -I ../m4

# Please keep these ASCII-lexically sorted (pass through sort(1)).
SUBDIRS = \
        DemoCamera \
        RPiGPIO \
```

(continues on next page)

(continued from previous page)

```
        RPiTutorial \
        Video4Linux
```

And here is an excerpt of the relevant part of *configure.ac* that should be modified. In both files, the list of DeviceAdapters should be in alphabetical order.

```
# Please keep the list of device adapter directories in ASCII-lexical order,
# with an indent of 3 spaces (no tabs)! (Just pass through sort(1).)
# This is the list of subdirectories containing a Makefile.am.
m4_define([device_adapter_dirs], [m4_strip([
   DemoCamera
   RPiGPIO
   RPiTutorial
   Video4Linux
```

Now that we have written our device adapter and updated the Autotools files, we need to merge our code with the Micro-Manager source code. This is easily performed with the ci/merge.sh utility script:

```
$ ci/merge.sh /opt/rpi-micromanager
```

Each time you change the code you can run this script and it will copy only the changed files into the appropriate directories of */opt/rpi-micromanager/* (or whatever directory you pass as an argument).

The final step is to compile Micro-Manager and the libraries for our device adapter. If this is the first time you are doing this, it may take a long time (around half an hour). Subsequent compilations should be three or four times faster because the compiler cache will have been built.

To begin compilation, change into the *ci/build* directory.

```
$ cd ../../../ci/build
```

The build will be performed inside a Docker container that contains the build dependencies and QEMU, the emulator for the ARM processor architecture. Having a Docker image that is already configured for compilation ensures that you will have the proper dependencies without having to manually configure your environment. To download the Docker image from Dockerhub, run the following command.

```
$ docker-compose pull
```

Finally, begin the compilation by running

```
$ docker-compose up
```

If all goes well, then you will find the build artifacts in */opt/rpi-micromanager/build* at the end of the compilation:

```
$ tree /opt/rpi-micromanager/build
/opt/rpi-micromanager/build/
├── lib
│   └── micro-manager
│       ├── libmmgr_dal_DemoCamera.la
│       ├── libmmgr_dal_DemoCamera.so.0
│       ├── libmmgr_dal_RPiGPIO.la
│       ├── libmmgr_dal_RPiGPIO.so.0
│       ├── libmmgr_dal_RPiTutorial.la
│       ├── libmmgr_dal_RPiTutorial.so.0
│       ├── _MMCorePy.la
│       ├── MMCorePy.py
```

(continues on next page)

(continued from previous page)

```
│       └── _MMCorePy.so
└── share
    └── micro-manager
        └── MMConfig_demo.cfg
```

(The contents of your build directory may be different depending on what device adapters were built.) The libraries for the RPiTutorial device adapter are the files *libmmgr_dal_RPiTutorial.la* and *libmmgr_dal_RPiTutorial.so.0*. In addition, RPi-DeviceAdapters builds the Micro-Manager Python wrapper. The relevant files for the wrapper are *_MMCorePy.** and *MMCorePy.py*. The Python wrapper may be imported into a Python script to gain access to the methods in the Micro-Manager core.

Whenever you make changes to your code during development, you will need to run the *ci/merge.sh* script to copy the changes into */opt/rpi-micromanager* before recompiling. It would also be good to occassionally pull any updates to the build container by running *docker-compose pull*, but this should rarely be necessary.

2.1.5 Deploying the app

At this point, you may transfer the compiled librariers to your Raspberry Pi for use. However, manual transfers of the libraries can be cumbersome. Furthermore, it can be diffcult for others to benefit from your work if they have to recompile your source code on their own. For these reasons, RPi-DeviceAdapters provides tools to create a Docker-based app that can easily be uploaded and downloaded from Docker Hub for on-demand use.

To create the app, first navigate to the *ci/app* folder.

```
$ cd ../app
```

Next, use the Makefile to build the Docker image that contains the app.

```
$ make build
```

While creating the image, the Makefile will copy the contents of */opt/rpi-micromanager/build* into the image and configure the Python environment. (You will need to edit the Makefile and change the location of the build artifacts if you are using a directory other than */opt/rpi-micromanager*.)

Let's verify that the image has been built. Though your output will differ slightly, you should see something similar to the output found below.

```
$ docker image ls
REPOSITORY                          TAG                                                 ␣
↪IMAGE ID            CREATED              SIZE
kmdouglass/rpi-micromanager         build                                               ␣
↪24d67a46e281        5 days ago           745MB
```

At this point, you will need to create a Docker Hub account if you do not already have one and login via the command line.

```
$ docker login
```

The final step before uploading the image is to retag it so that it points to your Docker Hub repository and not the default one.

```
$ docker tag kmdouglass/rpi-micromanager USERNAME/rpi-micromanager
```

Here, *USERNAME* is your Docker Hub username. Finally, we can upload the image:

```
$ docker push USERNAME/rpi-micromanager
```

and, from the Raspberry Pi, download the image:

```
$ docker pull USERNAME/rpi-micromanager
```

You will need Docker installed on your Raspberry Pi to pull the image.

CHAPTER 3

Troubleshooting

3.1 "Failed to open /dev/video0" when using the Video4Linux device adapter

If you do not see a device file named *video0* inside the */dev* folder of your Raspberry Pi, then you may need to load the bcm2835-v4l2 kernel module.

```
$ sudo rpi-update

# Restart the Pi, then run the following command
$ sudo modprobe bcm2835-v4l2
```

Verify that *video0* exists by looking for the */dev/video0* output from the following command. (No output means that the file is not present.)

```
$ ls /dev | grep video
```

To ensure that the bcm2835-v4l2 kernel module is loaded at startup, add it to modprobe.d.

CHAPTER 4

Related pages

- Zulip Chat: Help and discussion about the project
- RPi-DeviceAdapters : Main project page and source code
- rpi-micromanager : The project's Docker images
- Micro-Manager : Open-source microscopy software