White paper about embedded live stream surveillance

Live is Life

1 Introduction

Meanwhile embedded devices have a processing power and a variety of interfaces which make them interesting for industrial image processing. Embedded devices can even master applications in the area of Artificial Intelligence (AI) like Machine Learning (ML) or its sub-aspect Deep Learning (DL). For nearly 10 years MATRIX VISION supports ARM based devices with fitting drivers for the product families mvBlueFOX (USB 2.0), mvBlueFOX3 (USB 3.0) as well as for mvBlueCOUGAR (Gigabit Ethernet). The following table gives an overview of tested platforms using our cameras:

System	Architecture	Test results			Price	Performance	Suitable for	More information
		USB 2.0	USB 3.0	GigE				
NVIDIA Jetson AGX Xavier	NVIDIA Carmel ARMv8.2	✓	✓	✓	approx. 650-700$	++	Demanding applications	nvidia.com
NVIDIA Jetson Nano	ARM Cortex-A57	✓	✓	✓	90-100$	+	Mid-range applications	nvidia.com
NVIDIA Jetson TX2[1]	ARM Cortex-A57	✓	✓	✓	approx. 400$	++	Demanding applications	nvidia.com
Raspberry Pi 4	ARM Cortex-A72	✓	✓		35-75$	−	Price sensitive projects	raspberrypi.org

[1]*Not recommended for new projects.*

For price sensitive projects the Raspberry Pi is very popular, and you can find a lot of life hacks and smart applications in connection with the Raspberry Pi throughout the internet.

This document shows how you can implement a live stream surveillance application with the mvBlueFOX3 camera and a Raspberry Pi. The Raspberry Pi gets the images from the camera and distributes the image to a website, which can be accessed by the local network. With the appropriate hardware the live streaming application can be realized with the USB 2.0 camera mvBlueFOX or with the Gigabit Ethernet camera mvBlueCOUGAR.

Following topics are being addressed in this document:

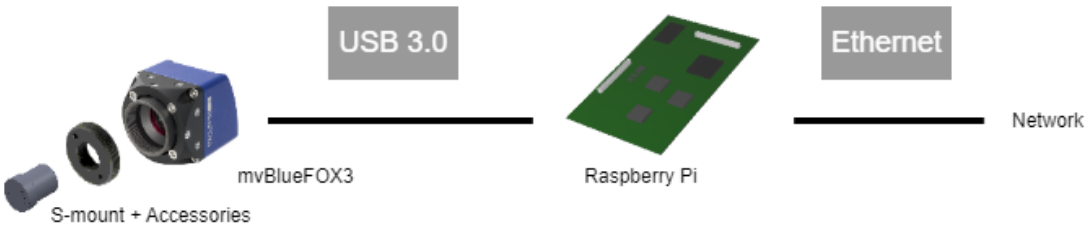
1. Which hardware is needed?
2. Which software components are needed on the Raspberry Pi?
3. How is the application implemented?
4. How can I set up an autostart?

2 Setup

Following components are used:

Part no.	Hardware	Quantity	Description
#12286	mvBlueFOX3-2016C-1112	1	USB3 camera with 1.6 Mpixel
#11571	E5M6018ND	1	S-mount lens 6mm
#07125	ADAPTER S-C AD01S	1	C to S-mount adapter
#07927	MV-LOCK RING SMOUNT	1	S-mount lock ring
	Raspberry Pi 4	1	ARM board as host machine
	USB3 cable	1	USB3 cable
	Ethernet cable	1	Ethernet cable
	Power supply for Raspberry Pi	1	Power supply for Raspberry Pi

Der Aufbau ist einfach:



3 Installation of the components

For the application, the following packages have to be built and installed on the Raspberry Pi (the building guidance for each package is embedded in the hyperlink):

- mvIMPACT Acquire Python API (camera driver and API for image acquisition)
- OpenCV (image manipulation and processing)
- numpy (image manipulation)
- Flask (Framework for a web interface)

4 Implementation

The program structure is as follows:

<Streaming>

| - *camera.py*

| - *main.py*

| - *templates*

    | - *index.html*

camera.py is a helper file which defines the **cam** class. This class manages the image acquisition whereas the camera is initiated (SoftwareTrigger, BGR8 as pixel format, etc.) and the function **singleFrame** is defined. The singleFrame function starts the acquisition of one image and dates it:

```
import sys
import ctypes
import string
import os
import cv2
import datetime
import numpy as np
from mvIMPACT import acquire

class cam(object):
    def __init__(self):
        # Open mvBlueFOX3
        self.pDev = None
        self.devMgr = acquire.DeviceManager()
        devCount = self.devMgr.deviceCount()
        for n in range(0, devCount-1):
            if self.devMgr[n].family.readS() == "mvBlueFOX3":
                self.pDev = self.devMgr[n]
        if self.pDev == None:
            print("No USB3 camera. Please plug in a mvBlueFOX3.")
            exit()
        self.pDev.acquisitionStartStopBehaviour.write(acquire.assbDefault)
        self.pDev.interfaceLayout.write(acquire.dilGenICam)
        self.pDev.open()
        print(self.pDev.product.readS() + " is open")
        # Configure the camera
        # Get handle to the needed objects
        self.fi = acquire.FunctionInterface(self.pDev)
        self.ifc = acquire.ImageFormatControl(self.pDev)
        self.imgDst = acquire.ImageDestination(self.pDev)
        self.aqc = acquire.AcquisitionControl(self.pDev)
        self.statistics = acquire.Statistics(self.pDev)
        # Set software trigger, exposure and binning.
        self.aqc.triggerSelector.writeS("FrameStart")
        self.aqc.triggerSource.writeS("Software")
        self.aqc.triggerMode.writeS("On")
        self.aqc.exposureAuto.writeS("Continuous")
        self.aqc.mvExposureAutoUpperLimit.write(8000)
        self.aqc.mvExposureAutoAOIMode.writeS("mvCenter")
        self.ifc.binningHorizontal.write(2)
        self.ifc.binningVertical.write(2)
        self.ifc.pixelFormat.writeS("BGR8")
        self.imgDst.pixelFormat.write(acquire.idpfRGB888Planar)
        # Prepare request objects: Push all empty requests in the request queue.
        while self.fi.imageRequestSingle() == acquire.DMR_NO_ERROR:
            print("Buffer queued.")
        self.pPreviousRequest = None

    def singleFrame(self):
        global frame
        # Capture a single frame.
        self.aqc.triggerSoftware.call()
        requestNr = self.fi.imageRequestWaitFor(500)
```

```
        if self.fi.isRequestNrValid(requestNr):
            self.pRequest = self.fi.getRequest(requestNr)
            if self.pRequest.isOK:
                imgWidth = self.pRequest.imageWidth.read()
                imgHeight = self.pRequest.imageHeight.read()
                frame_array = np.frombuffer((ctypes.c_char *
self.pRequest.imageSize.read()).from_address(int(self.pRequest.imageData.read())
), np.uint8)
                frame_ori = np.reshape(frame_array,
(self.pRequest.imageChannelCount.read(), imgHeight, imgWidth)) # [[R, G, B], H,
W]
                frame = frame_ori.transpose((1, 2, 0)) # [H, W, [R, G, B]]
                frame[:,:,[0, 1, 2]] = frame[:,:,[2, 1, 0]] # [H, W, [B, G, R]],
the correct format for MAT as an img.
                frame = np.array(frame, dtype=np.uint8).copy()
                timestamp = datetime.datetime.now()
                cv2.putText(frame, str(timestamp), (10,20),
cv2.FONT_HERSHEY_SIMPLEX, 0.48, (0,128,200),1)
            if self.pPreviousRequest != None:
                self.pPreviousRequest.unlock()
            self.pPreviousRequest = self.pRequest
            self.fi.imageRequestSingle()
        return frame
```

Code Block 1 camera.py

The actual Python main program is called **main.py**. Here the *cam* class is imported. In the main function a thread generates the frames (**genFrame**) using single capturing and *numpy* copies the data to outputFrame.

The function **displayFrame** transforms the outputFrame data into a JPG, which will be dsitributed by the function **video_feed** to a HTML page:

```
# MATRIX VISION sample for streaming live images over a web page for multiple
clients.
# Company: MATRIX VISION GmbH
# Device Types: GenICam Cameras
# Description: Stream live images from a GenICam device over a web page for
multiple clients.

from camera import cam
import threading
import cv2
import numpy as np
from flask import Flask, render_template, Response

outputFrame = None
lock = threading.Lock()

# Set up flask app.
app = Flask(__name__)

# Render the web page.
@app.route('/')
def index():
    return render_template('index.html')

# Generate frames from the camera continuously.
def genFrame(dev):
    global outputFrame, lock
    while True:
        # Use lock to ensure thread-safe.
        with lock:
            outputFrame = np.copy(dev.singleFrame())

# Display frames.
def displayFrame():
    global outputFrame
    while True:
        # Check if the output frame is available, otherwise skip the iteration
of the loop.
        if outputFrame is None:
            continue
        # Encode the frame in JPEG format.
        (flag, img_jpg) = cv2.imencode(".jpg", outputFrame)
        # Ensure the frame was successfully encoded.
        if not flag:
            continue
        # Yield the output frame in the byte format.
        yield (b'--frame\r\n'
               b'Content-Type: image/jpeg\r\n\r\n' + img_jpg.tobytes() +
b'\r\n\r\n')

# Feed video to the web page.
@app.route('/video_feed')
def video_feed():
```

```
    return Response(displayFrame(),
                    mimetype='multipart/x-mixed-replace; boundary=frame')

if __name__ == '__main__':
    # Start a thread for continuous capture.
    t = threading.Thread(target=genFrame, args=(cam(),))
    t.start()
    # Defining server ip address and port.
    app.run(host='0.0.0.0',port='8000', debug=False)
```

Code Block 2 main.py

The corresponding index.html calls the video feed:

```
<html>
  <head>
    <title>MATRIX VISION Streaming</title>
  </head>
  <body>
    <center><img id="bg" style="-webkit-user-select:none;margin:auto;" src="{{
url_for('video_feed') }}"></center>
  </body>
</html>
```

Code Block 3 index.html

5 Autostart

To use the system as a plug and play device, the python program **main.py** needs to be automatically started with a certain delay at Raspberry Pi's boot. To do this, you need 3 files under */etc/systemd/system*:

- '*mvIA_env.conf*'
- '*run-script-with-delay.service*'
- '*run-script-with-delay.timer*' (same name as .service but with .timer as the suffix for the delay)

mvIA_env.conf: defines all the environment variables needed for the mvIMPACT Acquire. It is important, because **systemd** services simply ignore the environment variables defined under */etc/profile.d* which leads to mvIMPACT Acquire package not being able to be used when starting the python program in a systemd service.

```
MVIMPACT_ACQUIRE_DIR=/opt/mvIMPACT_Acquire
MVIMPACT_ACQUIRE_DATA_DIR=/opt/mvIMPACT_Acquire/data
GENICAM_ROOT=/opt/mvIMPACT_Acquire/runtime
GENICAM_ROOT_V3_1=/opt/mvIMPACT_Acquire/runtime
GENICAM_GENTL32_PATH=/opt/mvIMPACT_Acquire/lib/armhf
GENICAM_CACHE_V3_1=/opt/mvIMPACT_Acquire/runtime/cache/v3_1
GENICAM_LOG_CONFIG_V3_1=/opt/mvIMPACT_Acquire/runtime/log/config-
unix/DefaultLogging.properties
```

Code Block 4 mvIA_env.conf

run-script-with-delay.service: defines the service to be executed at boot. The full path of the python program can be configured at: '*ExecStart=*'.

```
[Unit]
Description="Run script at startup"
After=network.target

[Service]
Type=oneshot
PermissionsStartOnly=true
EnvironmentFile=/etc/systemd/system/mvIA_env.conf
ExecStart=/usr/bin/python3 /home/pi/PythonProjects/Streaming/main.py
User=pi
Group=pi
TimeoutStartSec=0

[Install]
WantedBy=multi-user.target
```

Code Block 5 run-script-with-delay.service

run-script-with-delay.timer: defines the time delay for starting the service at boot. The time delay for starting the service can be configured at '*OnBootSec=*'.

```
[Unit]
Description="Run script after 2min of boot"

[Timer]
OnBootSec=2min

[Install]
WantedBy=default.target
```

Code Block 6 run-script-with-delay.timer

Once everything is configured accordingly, calling this to refresh the **systemd**:

```
$sudo systemctl daemon-reload
```

You can check then if the service is loaded successfully by calling its status:

```
$sudo systemctl status run-script-with-delay.service
```

If successful, start the service to verify if the python program starts and runs without problem:

```
$sudo systemctl start run-script-with-delay.service
```

If everything runs without problems, disable the service, since we don't want to start the service right at boot:

```
$sudo systemctl disable run-script-with-delay.service
```

Enable the timer file, because we want to delay start the service:

```
$sudo systemctl enable run-script-with-delay.timer
```

Finally, reboot the system. After rebooting, the python program shall start automatically after the system is booted up.

```
$reboot
```