Whitepaper über Embedded Live-Stream-Überwachung

Live is Life

1 Einführung

Embedded-Geräte weisen mittlerweile eine Rechenleistung und Schnittstellenvielfalt auf, welche sie äußerst interessant für den Einsatz in der industriellen Bildverarbeitung machen. Anwendungen im Bereich der Künstliche Intelligenz (KI) wie bspw. Machine Learning (ML) oder dessen Teilaspekt Deep Learning (DL) können Embedded-Geräte meistern. MATRIX VISION unterstützt ARM-basierte Geräte seit fast zehn Jahren mit passenden Treibern sowohl für die Produktfamilien mvBlueFOX (USB 2.0), mvBlueFOX3 (USB 3.0) als auch für mvBlueCOUGAR (Gigabit Ethernet). Die folgende Tabelle listet eine Übersicht der von uns mit unseren Kameras getesteten Plattformen auf:

System	Architektur	Test-Ergebnisse			Preis	Leistung	Geeignet für	Weitere Informationen
		USB 2.0	USB 3.0	GigE				
NVIDIA Jetson AGX Xavier	NVIDIA Carmel ARMv8.2	✓	✓	✓	ca. 650-700$	⊕⊕	Anspruchsvolle Anwendungen	nvidia.com
NVIDIA Jetson Nano	ARM Cortex-A57	✓	✓	✓	90-100$	⊕	Mid-Range Anwendungen	nvidia.com
NVIDIA Jetson TX2[1]	ARM Cortex-A57	✓	✓	✓	ca. 400$	⊕⊕	Anspruchsvolle Anwendungen	nvidia.com
Raspberry Pi 4	ARM Cortex-A72	✓	✓		35-75$	⊖	Preissensitive Projekte	raspberrypi.org

[1] *Nicht empfohlen für neue Projekte.*

Für preissensitive Anwendungen ist das Raspberry Pi sehr beliebt und das Internet bietet viele Life Hacks und smarte Anwendungen in Verbindung mit dem Raspberry Pi.

Dieses Dokument zeigt, wie Sie beispielsweise mit einer mvBlueFOX3 Industriekamera in Verbindung mit dem Raspberry Pi eine Live-Stream-Überwachung implementieren können. Hierbei bekommt das Raspberry Pi die Bilder von der Kamera geliefert und bettet diese in eine Website ein, die im lokalen Netzwerk aufgerufen werden kann. Mit der entsprechenden Hardware kann Live-Streaming-Demo auch mit der USB 2.0 Kamera mvBlueFOX oder mit der Gigabit Ethernet Kamera mvBlueCOUGAR realisiert werden.

Folgende Punkte werden in diesem Dokument behandelt:

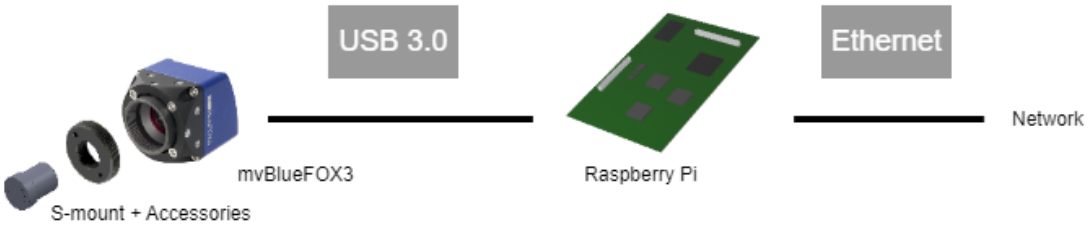
1. Welche Hardware wird benötigt?
2. Welche Software-Komponenten werden auf dem Raspberry Pi benötigt?
3. Wie wird die Anwendung konkret implementiert?
4. Wie kann ich ein Autostart einrichten?

2 Aufbau

Folgende Komponenten kamen zum Einsatz:

Sachnummer	Hardware	Anzahl	Beschreibung
#12286	mvBlueFOX3-2016C-1112	1	USB3 Industriekamera mit 1,6 Mpixel
#11571	E5M6018ND	1	S-mount Objektiv 6mm
#07125	ADAPTER S-C AD01S	1	C zu S-Mount Adapter
#07927	MV-LOCK RING SMOUNT	1	S-mount Verschlussring
	Raspberry Pi 4	1	ARM-Gerät als Host
	USB3 cable	1	USB3 Kabel
	Ethernet cable	1	Ethernet Kabel
	Power supply for Raspberry Pi	1	Netzteil für Raspberry Pi

Der Aufbau ist einfach:



3 Installation der Komponenten

Für die Anwendung müssen folgende Paket auf dem Raspberry Pi gebaut bzw. installiert werden (die Links verweisen auf die jeweilige Installations- und Einrichtungsanleitung):

- mvIMPACT Acquire Python API (Kameratreiber und API für den Bildeinzug)
- OpenCV (Bildmanipulation und -verarbeitung)
- numpy (Bildmanipulation)
- Flask (Framework für eine Web-Schnittstelle)

4 Implementierung

Die Programm-Struktur sieht wie folgt aus:

<Streaming>

| - *camera.py*

| - *main.py*

| - *templates*

    | - *index.html*

camera.py ist die Helfer-Datei, in welcher die **cam**-Klasse vorgeben wird. Diese bewerkstelligt die Bildakquise, wobei die Kamera initialisiert (SoftwareTrigger, BGR8 als Pixelformat, usw.) und die Funktion **singleFrame** definiert wird. Die singleFrame Funktion startet die Aufnahme eines Bildes und versieht das Bild mit einem Zeitstempel:

```
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

Das eigentlich Python-Hauptprogramm heißt **main.py**. Hier wird die *cam* Klasse importiert. In der Main-Funktion wird in einem Thread das Erzeugen eines Frames (**genFrame**) mittels Einzelbildaufnahmen gestartet und via *numpy* in einen outputFrame kopiert.

Die Funktion **displayFrame** wandelt die outputFrame Daten dann in ein JPG um, welche dann über die Funktion **video_feed** in einer HTML ausgespielt werden:

```
# 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

Die dazugehörende index.html ruft den Video-Feed auf:

```
<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

Damit das System als Plug-&-Play-Gerät verwendet werden kann, muss das Python-Programm **main.py** am Ende des Boot-Prozesses automatisch gestartet werden. Dazu werden drei Dateien unter */etc/systemd/system* benötigt:

- '*mvIA_env.conf*'
- '*run-script-with-delay.service*'
- '*run-script-with-delay.timer*' (mit dem gleichen Namen wie .service, nur mit .timer als Endung für die Verzögerung)

mvIA_env.conf: Legt die benötigten Umgebungsvariablen für mvIMPACT Acquire an. Dies ist notwendig, da **systemd** die Umgebungsvariablen unter */etc/profile.d* ignoriert, was dazu führen würde, dass die mvIMPACT Acquire Pakete aus dem systemd-Service nicht verwendet werden können.

```
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: Legt den Service an, der beim Booten gestartet wird. Der Pfad zum Python-Programm wird hier angeben: '*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: Gibt die Zeit vor, wie lange der Start des Services nach dem Booten verzögert werden soll. Die Verzögerung kann hier angeben werden: '*OnBootSec=*'.

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

[Timer]
OnBootSec=2min

[Install]
WantedBy=default.target
```

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

Sobald alles eingerichtet ist, kann **systemd** wie folgt neu geladen werden:

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

Der Status kann über diesen Befehl geprüft werden:

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

Falls das erfolgreich war, kann der Service gestartet werden, um zu prüfen, ob auch Python-Programm gestartet ist und ohne Probleme läuft.

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

Falls alles läuft, kann der Service deaktiviert werden, denn der Service soll verzögert nach dem Booten gestartet werden:

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

Hierzu muss die Konfiguration mit der Verzögerung aktiviert werden:

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

Abschließend muss neu gestartet werden. Nach dem Reboot sollte das Python-Programm automatisch starten.

```
$reboot
```