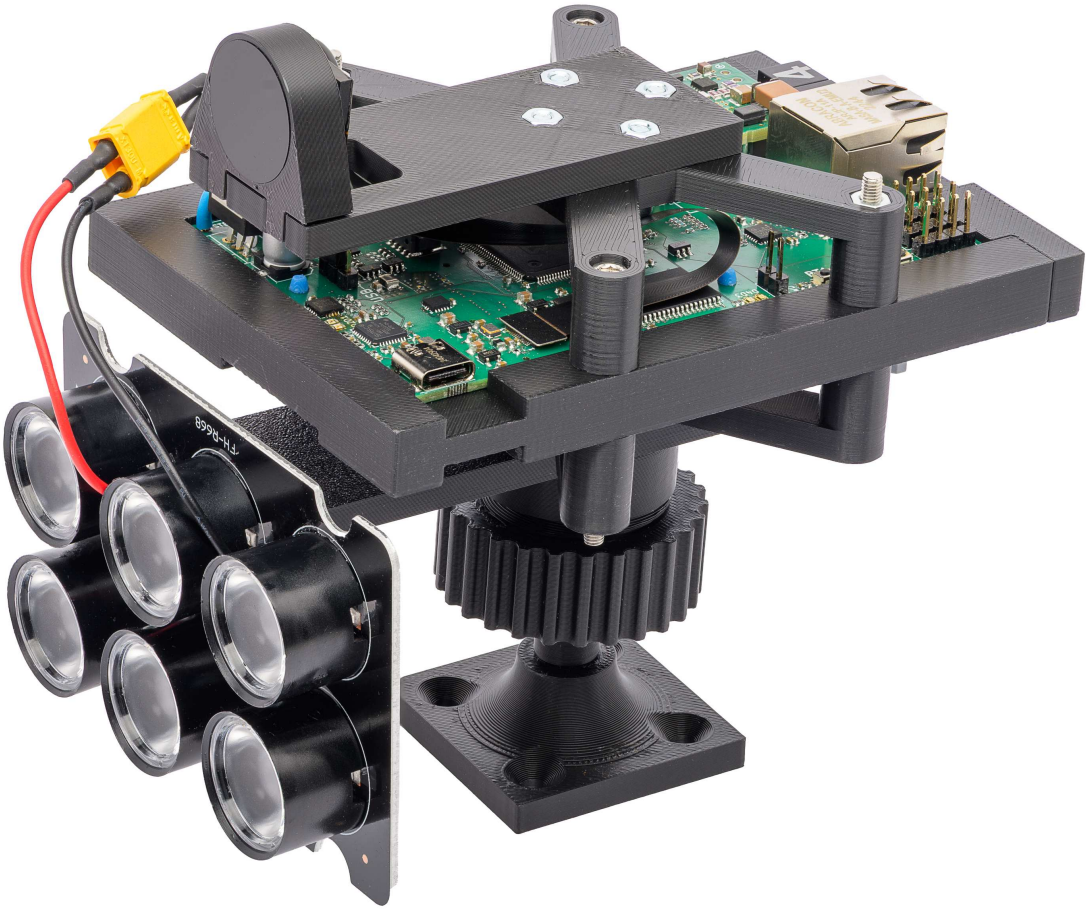


Low-Cost Motion Capture

Master Thesis: Building an FPGA-based Infrared Motion Capture System

Course of study	Master of Science in Engineering
Author	Tobias Jäggi
Advisor	Prof. Dr. Theo Kluter
Co-advisor	Prof. Marcus Hudritsch

Version 1.0 of July 20, 2025

Abstract

Infrared-based motion capture systems (MoCap) work by using multiple cameras to observe reflective markers, which are worn by the tracking subject. An infrared (IR) light source illuminates the markers, which then appear as bright spots in the image feed. An image processing algorithm detects the two-dimensional (2D) positions of all markers in each camera's frame. A central system then collects the 2D positions of all detected markers and triangulates their 3D position in physical space.

Although infrared-based motion capture technology is mature, commercial systems are expensive and often suffer from vendor lock-in. There is a lack of affordable, open-source, high-precision, and low-latency infrared-based solutions. Most openly developed systems rely on machine learning rather than physical trackers to capture motion data, often at the expense of tracking accuracy, flexibility, robustness, and sampling rate. An alternative to the commercial offerings was to be developed in this thesis.

To accelerate processing, an image-processing pipeline based on previous work done in [1] and [2] was implemented on a field-programmable gate array (FPGA). The pipeline receives a high frame-rate video feed from a global shutter camera, its output is a list of all detected two-dimensional marker positions for each frame. A custom PCB, powered by a Power over Ethernet (PoE) module, was developed. The PCB drives an infrared light source, which illuminates the reflective markers. Additionally, the PCB hosts a microcontroller that receives the markers detected by the FPGA (transmitted over SPI) and forwards them over UDP to a central system in real-time. This system consisting of a camera, infrared light source, microcontroller, and FPGA, is referred to as the Blob Detector.

Python software was developed, which receives the 2D marker positions over UDP and triangulates its 3D position in space. The marker's 3D position is rendered in real-time in a 3D scene. Additionally, a GUI was designed and implemented, which is used to configure and calibrate the Blob Detector.

Contents

1 Introduction

1.1 Starting Point

MoCap applications range from entertainment to medical research. There are different ways to capture 3D movement in space. Common solutions include passive optical trackers (IR reflectors used with multiple IR cameras), active trackers based on inertial measurement units (IMUs), and machine-learning based approaches that process video feeds from one or more cameras. When needing to achieve high spatial accuracy and high capture rate, IR based systems have become the industry standard. Vendors such as OptiTrack and Vicon have specialized in this segment. OptiTrack's prices start at CHF 500.- and go up to CHF 10000.- per camera [3]. Their software licence starts at CHF 800.- and can cost up to CHF 8000.- [3]. Thus, building a capable MoCap system using commercial solutions can cost tens to hundreds of thousands of CHF, only to end up with a vendor locked system. This is especially troublesome for academic research programs, where such a price can be out of reach and vendor lock-in is often not desirable.

Nonetheless, there do exist some notable low-cost commercial MoCap solutions like the Vive Tracker platform or the Rokoko Smartsuit Pro II. However, research [4] shows that Vive's platform is not suitable for academic research. Smartsuit Pro II can be used for medical studies [5], but is yet to be proven for high accuracy applications. Both commercial systems also suffer from vendor lock-in.

Several open-source, machine learning based MoCap systems, such as Freemocap [6] and EasyMocap [7] are available. However, these solutions only support tracking the specific subjects the machine learning model was trained on.

The lack of a low-cost, open-source IR based MoCap solution spawned Mimikry [8]. This is a proof of concept MoCap system that shows that real time IR based tracking can be done using cheap off the shelf hardware. Mimikry is limited in refresh rate, tracking volume and precision.

In Low Cost Motion Capture - Project I: Image Processing Pipeline [1], an architecture to do IR-based MoCap in hardware was proposed and parts of it implemented. In Low Cost Motion Capture - Project II: Marker detection unit [2], the architecture proposed in Project I, is fully implemented and further improved on an open-source platform. The result was a proof of concept marker detection unit, capable of detecting multiple markers in a camera feed and plotting their 2D coordinates on a central system in real time.

1.2 Objective

This project builds on top of both Project I and Project II and is part of a greater project with the aim of improving the refresh rate of the Mimikry system by moving the software-based image processing done on a Raspberry Pi 3 to a hardware architecture realized on a FPGA. The proof of concept marker detection unit of Project II is further improved with a custom PCB enabling the use of an improved camera compared to [2]. A full tracking system comprised of six marker detection units is built and validated.

2 Methods

2.1 Overview

In [1] an architecture for a hardware accelerated optical marker detection unit of an infrared (IR)-based motion capture (MoCap) system was discussed. The core of the system is the LinkRunCCA [9] algorithm capable of determining the bounding box of a blob in a binary image. The reference implementation of the LinkRunCCA algorithm was successfully validated using an Intel Cyclone V System on Chip (SoC) FPGA.

In the follow-up project [2], a proof of concept marker detection unit using LinkRunCCA was built. The proprietary Intel platform used in [1] was replaced with an open-source alternative. The system was able to detect markers in an image feed of 640 x 480 pixels at 30 frames per second. Their coordinates were sent to a central system over Wi-Fi, where a plot showed their position in real time.

In this project, a custom PCB, referred to as the Vision Add-On, is designed to replace the perifAddOn used in [2]. The motivation for the redesign is outlined in 2.2.8. The system comprised of the GECKO5Education [10] (see 2.2.1) and Vision Add-On is referred to as the Blob Detector. The function of the Blob Detector is to detect reflective markers, which are worn by the tracking subject, in an image feed and send their 2D coordinates to a central system. Next, embedded C++ software running on the Vision Add-On's microcontroller is developed. Additionally, a Python based graphical user interface (GUI) to configure and calibrate the Blob Detector is designed and implemented. Finally, Python software is developed which receives the data published by the Blob Detectors to triangulate and render the 3D position of a single marker in real time.

This thesis builds on top of both [1] and [2] as many of the components implemented on the GECKO5Education remain the same. The reader does not need to reference [1] or [2], all relevant information regarding the final system can be found in this document.

This document's instructions assume Linux is used as an operating system, Debian 12 was used by the author.

2.2 Hardware

2.2.1 GECKO5Education Platform

Overview

GECKO5Education [10] is an open-source and open hardware educational platform.

The GECKO5Education base board shown in figure 2.1 is powered by a Lattice FPGA, which is supported by an open-source toolchain presented in 2.2.2. GECKO5Education hosts a Lattice FPGA, an HDMI output (6 in 2.1), some buttons and LEDs, as well as plenty of general-purpose input/output (GPIO) pins. 32 MB of SDRAM (5 in 2.1) are connected to the FPGA. GECKO5Education is designed to be used with the Olimex CAMERA-OV7670. It can be connected to 2 in 2.1. The Olimex CAMERA-OV7670 was used in [2] but due to its limited resolution and frame rate (640 x 480 at 30 fps) as well as the fact that it is not sensitive to IR light without replacing the stock lens, a different camera will be used instead, see 2.2.8. 32 GPIO pins are connected to an expansion header, which allows the FPGA to interface with add-on boards. In [2] the perifAddOn was used. It features an Espressif ESP32-C3-MINI-1 [11] SoC, and SD card reader and a USB hub. A custom add-on board is designed to replace the perifAddOn, see 2.2.8

Programmable Logic

The FPGA on the GECKO5Education board is a Lattice LFE5U-85F-6BG381C FPGA (part of the ECP5 family) with 84k programmable lookup tables (LUTs). The Cyclone V used in [1] has 85k programmable LUTs as well as a Hard Processor System (HPS). Since the FPGA on the GECKO5Education has no HPS, a soft-core will be used instead to still allow the use of software on the FPGA.

2.2.2 Toolchain and Hardware Description Language

Synthesis

The OSS CAD Suite [12] is a collection of open-source tools used to design and verify digital logic. Yosys [13] is the main tool in OSS CAD Suite. It is a register-transfer level (RTL) synthesis tool, which supports Verilog 2005 and some select features of the SystemVerilog language. Using the GHDL Yosys Plugin [14], Yosys also supports VHDL. Since the reference project for the GECKO5Education was written in Verilog, any further components are also developed in Verilog. When synthesizing a Verilog design using Yosys, the tool outputs a netlist as a JSON file, which is used by other tools for further processing. Listing 2 shows a script declaring the steps that Yosys takes to synthesize the Verilog module `simpleLogic` shown in listing 1. The script can be executed by running `yosys -s simpleLogic.script` in a command shell. Line one of listing 2 reads the Verilog source file, line two sets the Simple Logic module as the top level component, declares the JSON output file name and

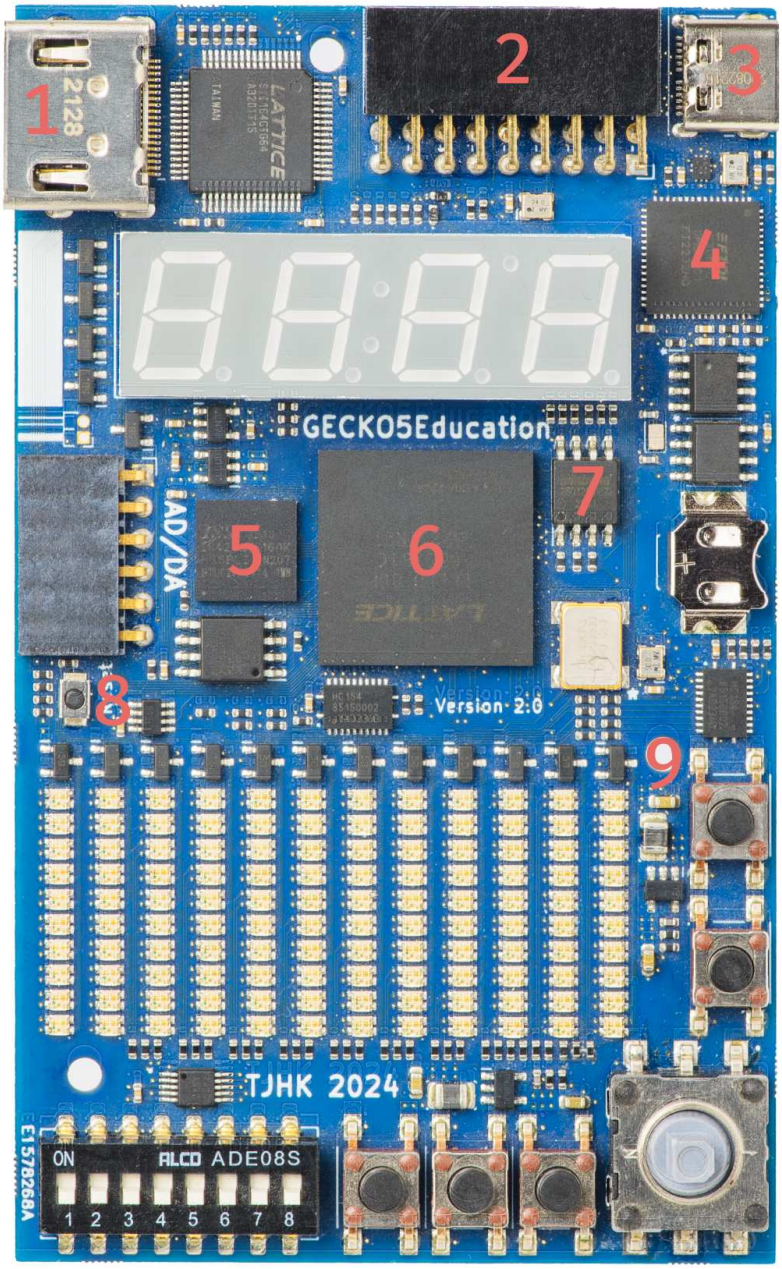


Figure 2.1: Front view of GECKO5Education

1: HDMI output
2: Camera Connector
3: USB-C Connector
4: USB to UART Bridge
5: SDRAM
6: FPGA
7: Flash
8: Reset Button
9: Buttons

starts the synthesis. As a bonus, line 3 creates a graphical representation of the synthesized netlist shown in figure 2.2.

The Lattice LFE5U-85F-6BG381C FPGA uses LUTs with 4 inputs and 1 output [15]. Note that in listing 2 synth_ecp5 [16] is used to run the synthesis. It is aware of the ECP5 family of FPGAs. As a result, the synthesized netlist seen in figure 2.2 uses LUTs with 4 inputs and 1 output.

```
1  module simpleLogic (
2      input  wire clk,
3      input  wire reset,
4      input  wire a,
5      input  wire b,
6      output reg  c
7  );
8
9    always @(posedge clk or posedge reset) begin
10     if (reset) c <= 1'b0;
11     else c <= a & b;
12   end
13
14 endmodule
```

Listing 1: Simple Logic Verilog module

```
1 read -sv simpleLogic.v
2 synth_ecp5 -top simpleLogic -json simpleLogic.json
3 show -colors 3
```

Listing 2: Yosys script to synthesize simpleLogic.v

Listing 3 shows the build script used in this project. Firstly, it calls Yosys in the same fashion as the Simple Logic example above. Next, the JSON output generated by Yosys is processed by nextpnr [17]. Nextpnr performs place and route algorithms as well as timing checks to fit the netlist into the Lattice LFE5U-85F-6BG381C FPGA. It also maps logic signals to physical I/O pins of the FPGA. The I/O mapping is defined in the file gecko5_or_1420.lpf, which is passed to nextpnr as an input argument. The output of nextpnr is converted to a bitstream using ecppack [18] Finally, openFPGALoader [19] is used to program the bitstream to the FPGA. The argument `-f` is used to write the bitstream to flash. If `-f` is omitted, the design is lost on power loss.

`<hdl>` denotes the root of the GECKO5Education Hardware Description Language (HDL) project.

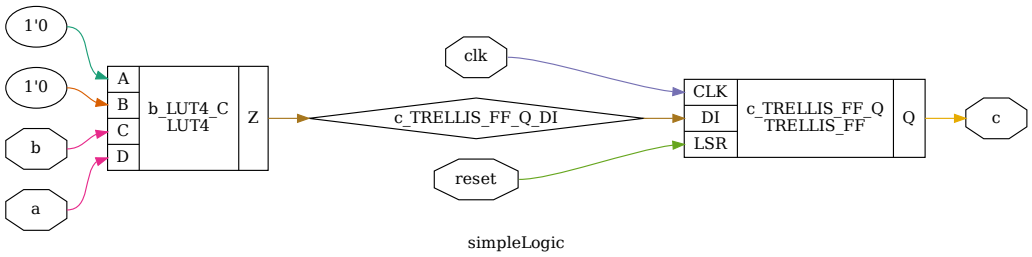


Figure 2.2: Netlist of simpleLogic.v synthesized by Yosys

```
#!/bin/bash
rm or1420SingleCore.*
yosys -D GECKO5Education -D fakeCamera -s ../scripts/yosysOr1420.script
  ↪  -l or1420SingleCore.yosys.log
nextpnr-ecp5 --timing-allow-fail --85k --package CABGA381 --json
  ↪  or1420SingleCore.json --lpf ../scripts/gecko5_or1420.lpf --textcfg
  ↪  or1420SingleCore.config
ecppack --compress --freq 62.0 --input or1420SingleCore.config --bit
  ↪  or1420SingleCore.bit
openFPGALoader or1420SingleCore.bit -f
```

Listing 3: Bash script defining the build process from synthesis to programming the FPGA

To synthesize the Verilog HDL of this project, navigate to the folder `<hdl>/systems/singleCore/sandbox` and run `../scripts/synthesizeOr1420.sh`.

The GECKO5Education's USB-C port is disabled if the Vision Add-On board is connected. Use the Vision Add-On's USB-C port to establish a connection between the PC and the GECKO5Education

When adding a new Verilog source file to this project, make sure to add it to the file `<hdl>/systems/singleCore/scripts/gecko5\_or1420.lpf`.

Simulation and Testing

OSS CAD Suite provides two notable tools to simulate and test Verilog HDL: Verilator [20] and ICARUS Verilog (iverilog) [21].

Verilator is a Verilog and SystemVerilog simulator. The tool compiles Verilog into C++ or SystemC, depending on the users choice. Testbenches are developed in the selected output language. Verilator offers very high simulation performance compared to other Verilog simulators.

Verilog defines four different states to represent a logic level:

- `0` - logical zero
- `1` - logical one
- `x` - unknown state
- `z` - high impedance

Verilator only supports two logic levels, `0` and `1`. Listing 4 shows a Verilog module which produces all four logic levels at its outputs. By analysing the Verilog HDL one can determine that the output `o` is undefined until it assumes the state of the input `i` once input `s` is high. The input `h` is set to high impedance. Figure 2.3 shows the result of simulating listing 4 using Verilator. The signals `o` and `h` do not show their expected behaviour since Verilator does not support the logic states they assume.

By switching to a simulator that supports unknown logic states, design errors due to undefined behaviour are apparent in the simulation results and thus caught much quicker.

Iverilog is a fully compliant Verilog HDL compiler. Therefore, it inherently also supports all four logic levels. Vvp is the runtime engine of iverilog and is capable of executing the compiled Verilog HDL. This document will not make the distinction between the compiler and the runtime engine, it will refer to the whole system as iverilog. Since iverilog is capable of compiling and executing Verilog HDL, it can be used to simulate and test a

Verilog module. When using iverilog without any additional tools, the user has to write the testbenches in Verilog. Figure 2.4 shows the result of simulating listing 4 using iverilog. All the signals show their expected behaviour.

At first, Verilator was used to develop and run C++ testbenches while developing Verilog modules. However, since iverilog produces more accurate simulation results, testbenches were later developed in Verilog and simulated using iverilog. The trade-off for more simulation accuracy are much slower testbenches.

Note that cocotb [22] can be used to develop and run Verilog (or VHDL) testbenches in Python. It supports most popular simulation backends, Verilator and iverilog are both supported. By using cocotb, the same test bench can be executed with different simulators. This is an interesting use case where the user benefits from the strengths of the different simulators but avoids their weaknesses by cross-referencing the simulation results.

Both Verilator and iverilog output value change dump (vcd) files which can be viewed using GTKWave.

```
1  module simulatorComparison (
2      input  wire i,
3      input  wire s,
4      output reg  o,
5      output wire h
6  );
7
8    always @(posedge s) begin
9      o <= i;
10   end
11
12   assign h = 1'bz;
13
14 endmodule
```

Listing 4: Simulator Comparison Verilog module

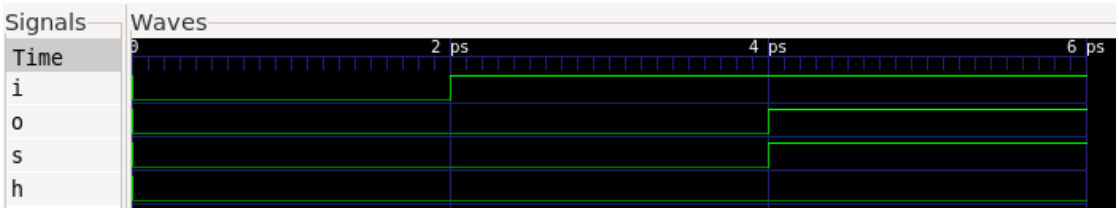


Figure 2.3: Screenshot of GTKWave rendering the vcd file produced by simulating listing 4 using Verilator

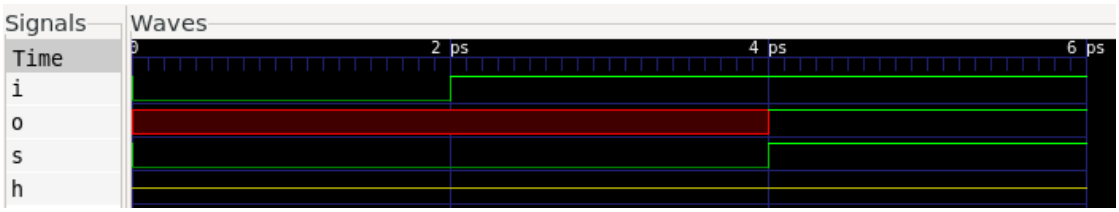


Figure 2.4: Screenshot of GTKWave rendering the vcd file produced by simulating listing 4 using iverilog

A module's testbench is located under `<hdl>/modules/<moduleName>/test`. To execute the testbench, run `make vvp` inside the modules test folder. Some modules feature a GTKWave project named <moduleName>.gtkw within the test folder, where the relevant signals are already selected. If no GTKWave project is present, use the waveform.vcd file to review the simulation results instead.

2.2.3 Blob Detector

Overview Figure 2.5 shows an overview of the Blob Detector. The design spans over the GECKO5Education as well as the Vision Add-On. This chapter will cover how the components on the GECKO5Education and the Vision Add-On interact during the different operation modes of the Blob Detector.

Figure 2.6 shows an overview of the programmable logic design implemented on the FPGA. The design is based on a project template for the GECKO5Education which already implements the following components seen on 2.6:

- OR 1420 CPU with custom instruction support
- Bus arbiter and system bus
- BIOS
- I2C controller
- HDMI controller
- SPI Flash controller
- SDRAM controller
- UART controller

Both the pre-existing components and the all custom components are documented in this chapter.

OR1420 soft-core system The Blob Detector uses a modified soft-core implementation of the open-source OpenRISC 1000 (OR1k) [23] architecture. A soft-core is a processor fully

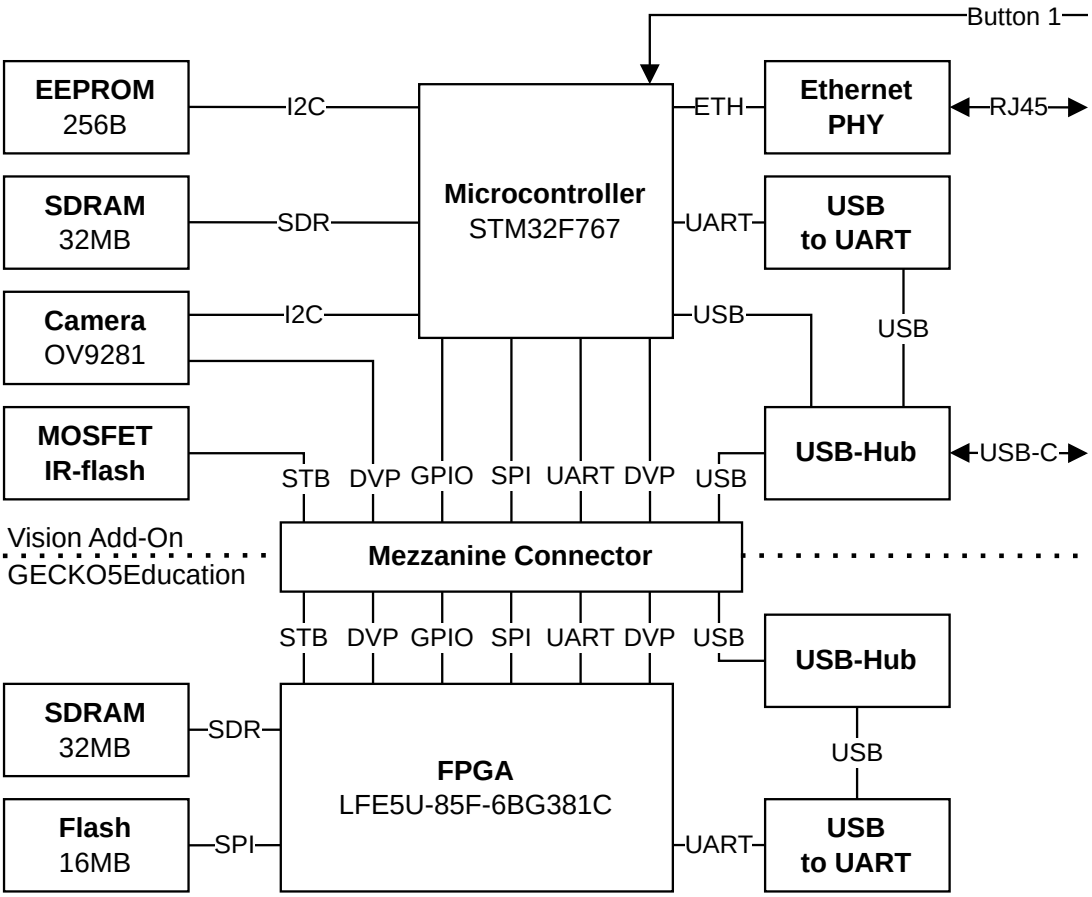


Figure 2.5: Blob Detector system overview

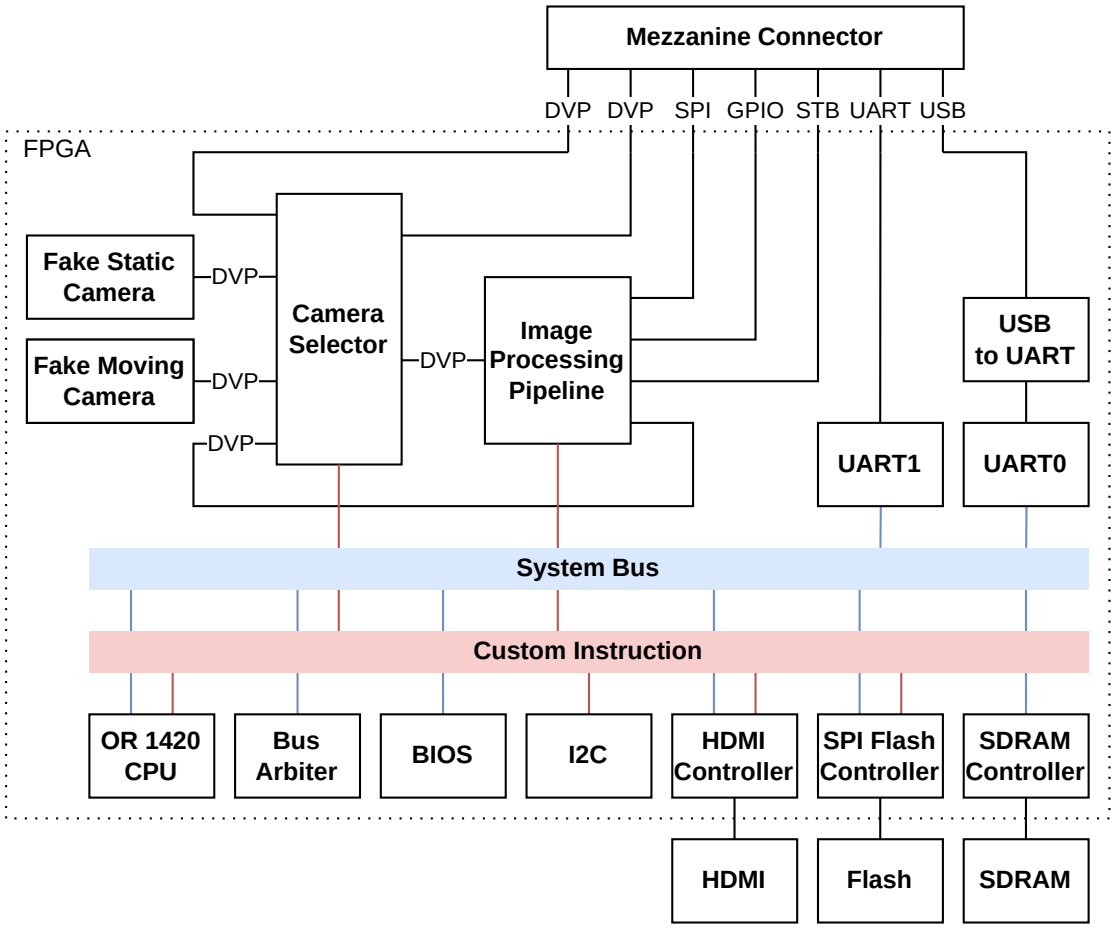


Figure 2.6: GECKO5 system overview

implemented in FPGA fabric. The OR1420 implements NIOS II compatible custom instructions (CIs) as well as a simple proprietary bus protocol. This allows the user to add custom system bus peripherals as well as CIs, both of which are used in the Blob Detectors design. The CI interface is covered in 2.2.5. The bus protocol is not covered in this document since it is not used by any module develop for the Blob Detector. It is only used by modules which were already part of the project template. The bus protocol was used in previous version of the Blob Detector and is documented in chapter 2.2.6 of [1].

The soft-core is a big-endian system.

SDRAM and Controller The SDRAM controller is connected to the system bus. The SDRAM chip used is the IS42VM16160k-75BLI with 4 banks with 4M of 16-bit words (results to 32MB) [24].

Flash and Controller The flash controller uses both the system bus and CI. The flash chip itself (7 in 2.1) is the 16MB [25] W25Q128JVS, which is connected to the Flash Controller using SPI.

BIOS The BIOS can load a program stored in flash into RAM form where the soft-core can boot. It also offers some user commands over UART0. They can be used to flash a new program.

I2C The I2C controller, accessed using CI, is intended to be used to configure the OV7670 camera, which was used in [2]. Since the OV7670 is replaced with a different camera located on the Vision Add-On, the I2C controller is no longer used.

UART Both UART controllers are accessed using the system bus. UART0 is accessible via an adapter chip (4 in 2.1) over USB (3 in 2.1) and UART1 is connected to the Vision Add-On.

Frame grabber The frame grabber takes a video stream as an input and stores each frame in the SDRAM using direct memory access (DMA). From SDRAM, the frame is accessible to the HDMI output module. To configure the frame grabber as well as start and stop image acquisition, the module implements the CI interface.

HDMI output The HDMI controller provides a split-screen 720p HDMI output (1 in 2.1). On the first half of the screen, text can be rendered. This functionality is used to provide a user interface to the BIOS module and the software running on the OR1420. The second half can

be used to display the contents of a camera frame buffer. It is located in the SDRAM. Direct memory access (DMA) is used to transfer the data from the SDRAM to the HDMI controller. This functionality was used in [2] but is no longer used in the present design since it is incompatible with the new camera resolution.

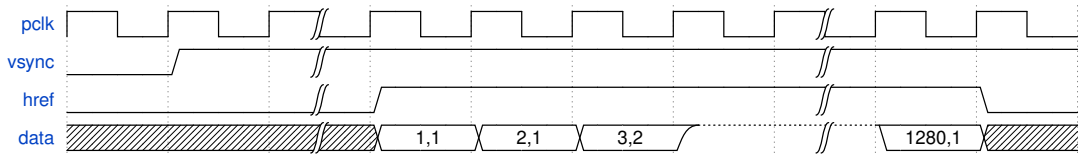
2.2.4 Fake Cameras

Motivation In order to test the image processing pipeline in a testbench, a component that generates synthetic images is required. For integration tests on the real hardware, a reproducible image is also beneficial. Thus, two modules, which generate a synthetic camera feed are present in the Blob Detector's design. The first module generates a static image (Fake Static Camera in figure 2.6) and the second one creates a moving image (Fake Moving Camera in figure 2.6). Both of them implement the same interface as the real camera located on the Vision Add-On, such that the image processing pipeline can not tell a difference between the real and the synthetic images.

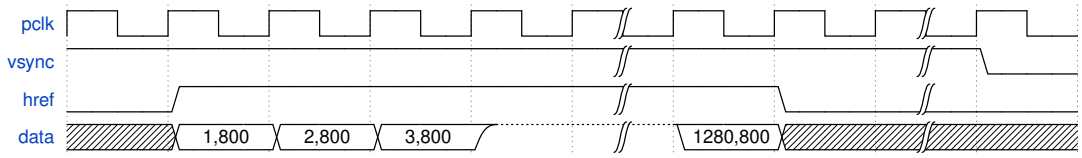
Interface The camera used by the Vision Add-On, is the KLT-E4MPF-OV9281 V4.2 NIR, manufactured by the Kai Lap Technologies Group. 2.2.8 will cover why this camera was chosen. The current chapter only focuses on the camera's interface, which is implemented by the fake cameras. Omnivision is the manufacturer of the camera's sensor, called OV9281. From this point on, the whole camera is referred to by its sensor's name. The camera output format used in this project is called DVP. It consists of the following signals:

- pixel clock (`pclk`)
- vertical sync (`vsync`)
- horizontal reference (`href`)
- 8-bit data (`data[7:0]`)

The camera transmits each frame line by line, from the top of the screen to the bottom. The pixels on each line are sent from left to right. The OV9281 camera is configured to output a greyscale video signal at a resolution of 1280x800 pixels. As a result, 800 lines are transmitted, where each line consists of 1280 pixels. `data[7:0]` represents the value of a single 8-bit greyscale pixel. The `data` signal is shown in figure 2.7. The numbers overlayed onto the signal represent the X,Y pixel coordinates of the current pixel being transmitted. `href` (high active) is active whenever a line is being transmitted as shown in figure 2.7. `vsync` (low active) denotes the end of the current frame (shown in figure 2.7b) and thus the start of a new frame (shown in figure 2.7a). Note that the OV9281 Camera actually outputs a `vsync` signal which is high activate. However, this signal is simply inverted on the FPGA. As a result, `data` is valid on the rising edge of `pclk` whenever both `href` and `vsync` are high.



(a) Start of frame



(b) End of frame

Figure 2.7: DVP timing diagram

Implementation Fake Camera refers to both Fake Static Camera and Fake Moving Camera shown in 2.6. The Fake Camera module implements the same signals show in 2.7.

Based on the following constants, Fake Camera generates the output signals:

- number of visible pixels per line $NVP_{Line} = 1280$
- number of pixels per horizontal front porch $NP_{HFP} = 19$
- number of pixels per horizontal sync pulse $NP_{HS} = 80$
- number of pixels per horizontal back porch $NP_{HBP} = 45$
- number of pixel-clock clock cycles per pixel $Npclk_{pix} = 1$
- number of visible lines per frame $NVL_{frame} = 800$
- number of lines per vertical front porch $NL_{VFP} = 10$
- number of lines per vertical sync pulse $NL_{VS} = 3$
- number of lines per vertical back porch $NL_{VBP} = 17$

Note that the Fake Camera can also output an RGB565 video signal by changing $Npclk_{pix}$ to 2, so that two clock cycles are needed per pixel. This is a leftover from [2] and no longer used in this project.

Using the constants above, the number of pixel-clock cycles per line $Npclk_{line}$ can be calculated with 2.1.

$$Npclk_{line} = (NVP_{Line} + NP_{HFP} + NP_{HS} + NP_{HBP}) * Npclk_{pix} \tag{2.1}$$

And the number of lines per frame NL_{frame} can be calculated with 2.2.

$$NL_{frame} = NVL_{frame} + NL_{VFP} + NL_{VS} + NL_{VBP} \tag{2.2}$$

As a result, one gets $Npclk_{line} = 1424$ and $NL_{frame} = 830$.

Fake Camera uses two looping counters, `counterX` and `counterY`, to generate it's output signals. The maximum value of `counterX` is $Npclk_{line}$ and the maximum value of `counterY` is NL_{frame}. `CounterX` is incremented with each rising edge of `pclk`. `CounterY` is incremented whenever `counterX` reaches $Npclk_{line}$. Both counters loop back to 0 once they have reached their maximum value. By comparing the counter values against some constants, the following signals are generated:

The signal `href` is active whenever condition 2.3 is met.

$$(\texttt{counterX} \in [0, NVL_{frame}[) \wedge (\texttt{counterY} \in \left[0, NVP_{Line} * Npclk_{pix}\right[) \tag{2.3}$$

The signal `hsync` is high whenever condition 2.4 is met (the signal is low active):

$$\texttt{counterX} \notin \left[(NVP_{Line} + NP_{HFP}) * Npclk_{pix}, (NVP_{Line} + NP_{HFP} + NP_{HS}) * Npclk_{pix}\right[\tag{2.4}$$

The signal `vsync` is high whenever condition 2.5 is met (the signal is low active):

$$\texttt{counterY} \notin [NVL_{frame} + NL_{VFP}, NVL_{frame} + NL_{VFP} + NL_{VS}[\tag{2.5}$$

Whenever condition 2.5 is met, the signal `data` is drawn to the screen.

$$(\texttt{counterX} \in \left[0, NVP_{Line} * Npclk_{pix}\right[) \wedge (\texttt{counterY} \in [0, NVL_{frame}[) \tag{2.6}$$

Listing 5 shows how multiple squares can be drawn to the screen by using logical or-gates. Setting `data` to either `0x00` or `0xFF` results in white squares being drawn onto a black background.

Compared to the Fake Static Camera module, the Fake Moving Camera increments the origin (both X and Y axis) of the square after each frame as seen in 6. As a result, the square slides over the screen from left to right, top to bottom. The stride parameters are used to change the speed of the movement.

Even though this project only uses the horizontal reference signal, a horizontal sync signal is also generated by Fake Camera. As a result, the component can be used as a generic VGA video source by simply adjusting the timing parameters.

2.2.5 Camera Selector

The Camera Selector module, shown in figure 2.8, is capable of switching between different image sources. All switched video signals consist of horizontal reference, vertical sync and 8-bit data signals. Note that the pixel clock is not listed since the clock signal is not switched by the multiplexers. Passing a clock signal through a multiplexer results in a

```
localparam PCLK_PER_PIXEL = 1;
wire s_square0 = ((s_counterX > (100 * PCLK_PER_PIXEL)) && (s_counterX < (200 * PCLK_PER_PIXEL)) && (s_counterY > 80) && (s_counterY < 380));
wire s_square1 = ((s_counterX > (275 * PCLK_PER_PIXEL)) && (s_counterX < (550 * PCLK_PER_PIXEL)) && (s_counterY > 5) && (s_counterY < 90));
wire s_square2 = ((s_counterX > (450 * PCLK_PER_PIXEL)) && (s_counterX < (500 * PCLK_PER_PIXEL)) && (s_counterY > 200) && (s_counterY < 320));
wire s_square3 = ((s_counterX > (150 * PCLK_PER_PIXEL)) && (s_counterX < (400 * PCLK_PER_PIXEL)) && (s_counterY > 390) && (s_counterY < 440));
wire data[7:0] = (s_square0 || s_square1 || s_square2 || s_square3) ? 8'hFF : 8'h00;
```

Listing 5: Drawing multiple static square

```
// 1 extra bit more to prevent immediate overrun if stride results in origin which is out of bounds
reg [$clog2(COUNTER_X_MAX):0] squareOriginX;
reg [$clog2(COUNTER_Y_MAX):0] squareOriginY;
localparam STRIDE_X = 'd10;
localparam STRIDE_Y = 'd8;
localparam PCLK_PER_PIXEL = 1;

always @(posedge pclk or posedge reset) begin
  if (reset) begin
    squareOriginX <= 0;
    squareOriginY <= 0;
  end else begin
    if (vsyncPosEdge == 'b1) begin
      squareOriginX <= (squareOriginX >= WIDTH) ? 'd0 : (squareOriginX + STRIDE_X);
      squareOriginY <= (squareOriginY >= HEIGHT) ? 'd0 : (squareOriginX >= WIDTH) ? (squareOriginY + STRIDE_Y) : squareOriginY;
    end
  end
end
assing s_square = ((s_counterX > (squareOriginX * PCLK_PER_PIXEL)) && (s_counterX < ((squareOriginX + SQUARE_WIDTH) * PCLK_PER_PIXEL)) && (s_counterY > squareOriginY) && (s_counterY < (squareOriginY + SQUARE_HEIGTH)));
```

Listing 6: Drawing a moving square

delayed clock signal, which is not desirable. Instead, all image sources use the same pixel clock, `pclk` provided by the real camera. To control which video source is active, the Camera Selector module implements a NIOS II compatible CI interface. The operation of a CI is shown in figure 2.9. Signals labelled with `uC` are driven by the soft-core and act as the input of the CI implementation. Signals labelled with `CI` are driven by the CI implementation and are read back by the soft-core.

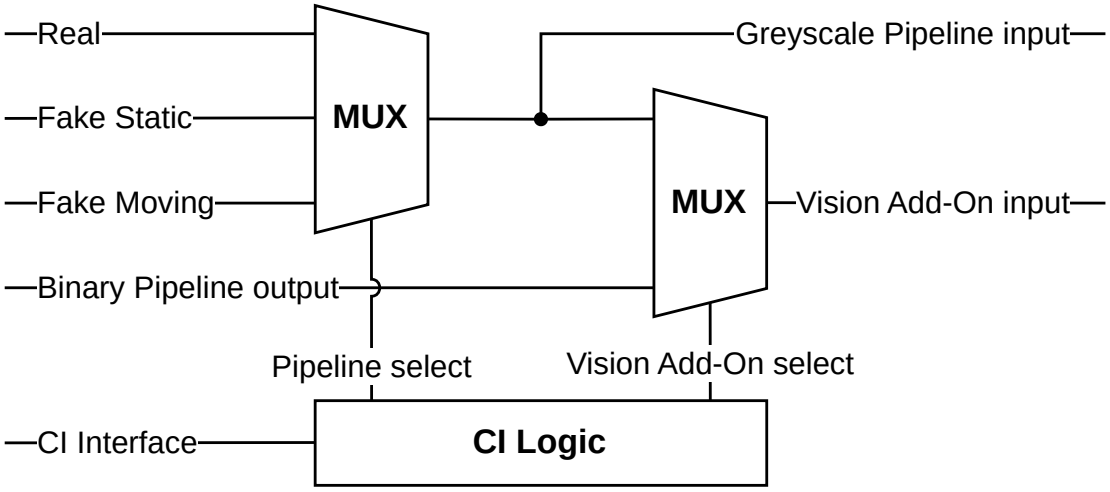


Figure 2.8: Camera Selector module

At the start of the CI sequence, the soft-core modifies the following signals in the same clock cycle:

- `ciStart` is set to high to indicate the start of the CI
- `ciN` is set to the unique identifier of the CI implementation which is addressed
- `ciDataA` is set to the desired value of the input A of the CI implementation
- `ciDataB` is set to the desired value of the input B of the CI implementation

Whenever `ciStart` is high, the CI implementation checks if `ciN` matches with its unique identifier. If there is a match, `ciResult` is set to the result of its custom operation based on `ciDataA` and `ciDataB`. `ciDone` is set to high in the same clock cycle as `ciResult` is valid. The CI implementation can either present its result in the same clock cycle as it is triggered, or take an arbitrary amount of cycles. This behaviour is labelled as *single cycle* and *multi cycle* respectively in figure 2.9. If `ciN` does not match with the CI's unique identifier, the instruction is ignored (`incorrect id` in figure 2.9) and the CI's outputs set to 0.

Figure 2.10 shows a system, which uses more than one CI. All the CI instances share the same input signals, the outputs are combined using OR-gates.

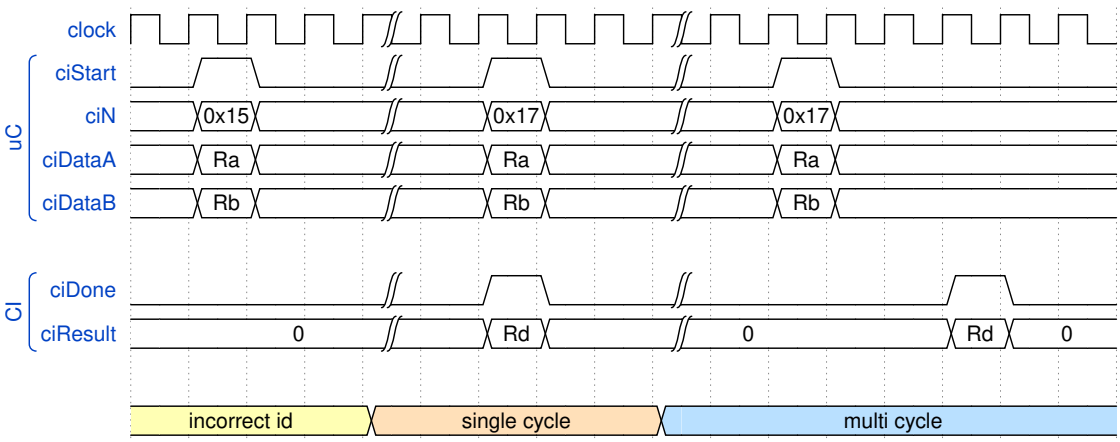


Figure 2.9: Custom instruction timing diagram [26]

Since the outputs of all CIs are connected using OR-gates, it is imperative that a CI's outputs remain at 0 until it is addressed directly.

As seen in figure 2.8, it is possible to select between three different image sources (Real, Fake Static, Fake Moving) to serve as the input of the image processing pipeline. The Vision Add-On's input can either be set to the greyscale image resulting in the path illustrated in figure 2.11, or the processed binary image, resulting in the path illustrated in figure 2.12. Only the real camera's signal path is shown in both figures, but the two Fake Cameras can also be selected to be sent to the Vision Add-On. Table 2.1 lists the implemented CI of the Camera Selector. Interfacing with the CI using assembly statements is covered in 2.3.1.

A[31:0]	B[31:0]	RES[31:0]	Description
0	0		set output mode to greyscale image
	1		set output mode to binary image
1	0		set input mode to real camera
	1		set input mode to fake static camera
	2		set input mode to fake moving camera

Table 2.1: Camera selector CI description

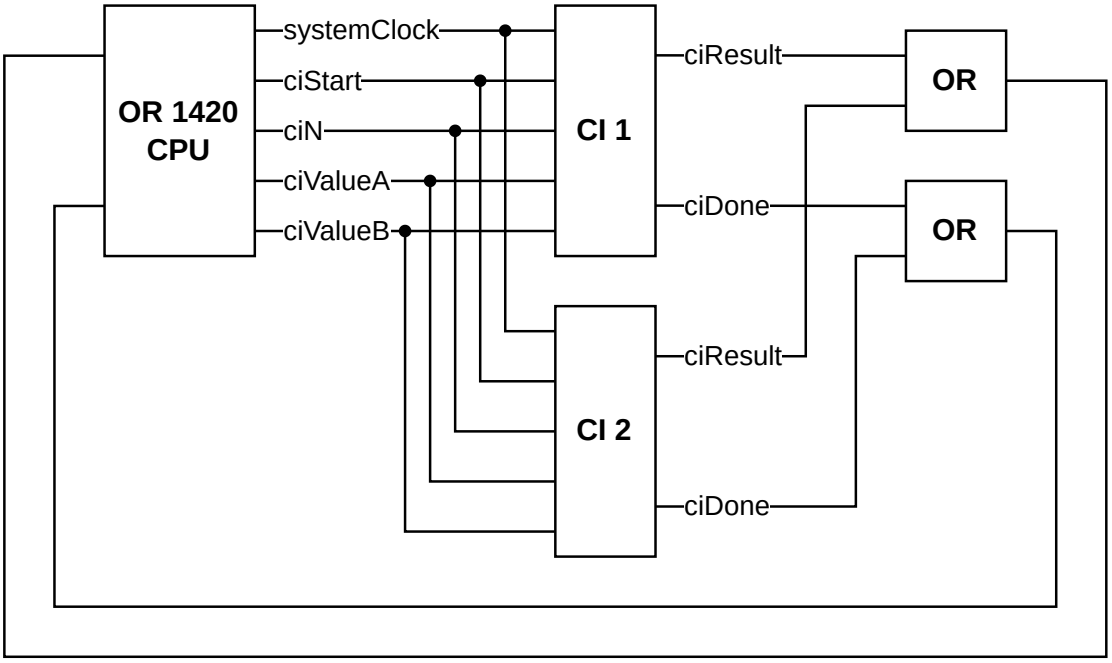


Figure 2.10: System using multiple custom instructions

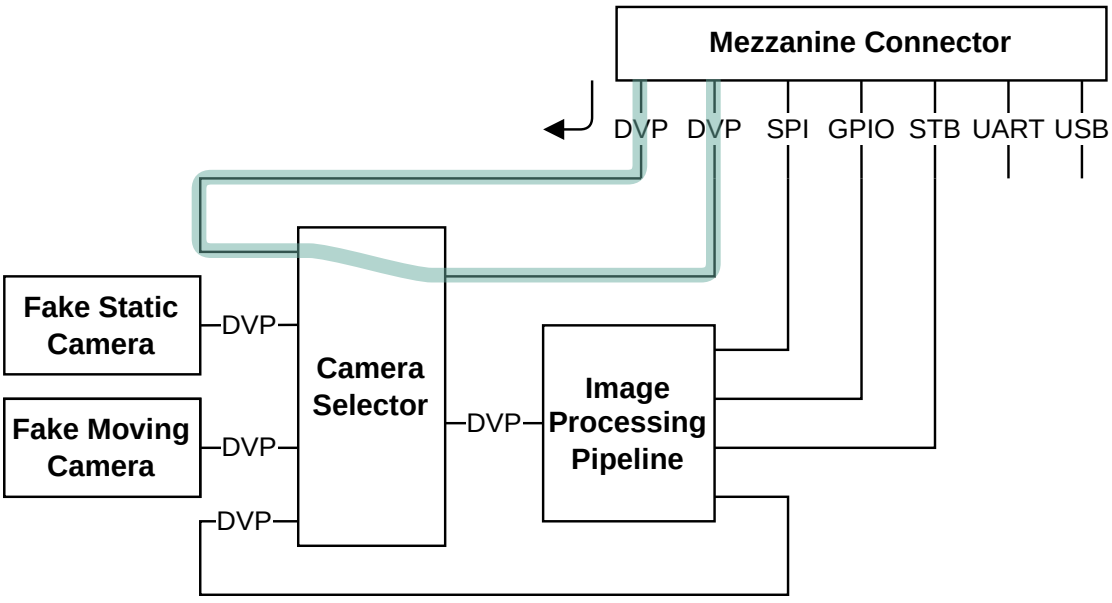


Figure 2.11: Unprocessed greyscale video path on the GECKO5Education

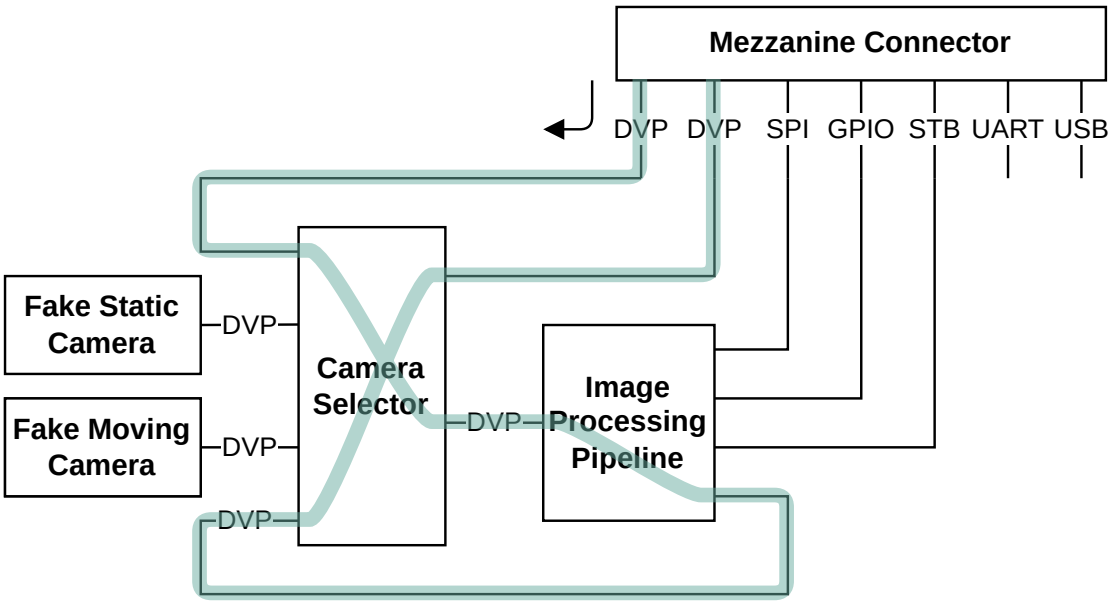


Figure 2.12: Processed binary video path on the GECKO5Education

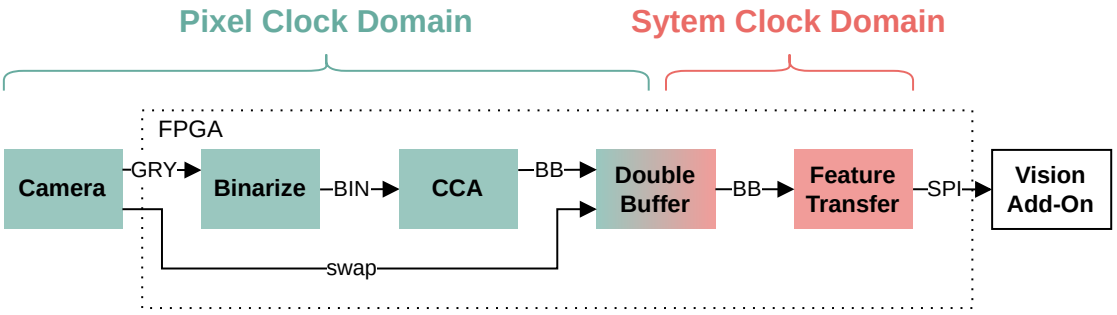
2.2.6 Image Processing Pipeline

Overview

The goal of the image processing pipeline is to detect markers in a video feed received from the Vision Add-On and forward their position back to the Vision Add-On. The image processing pipeline's input is the Camera Selector module as seen in 2.6. Figure 2.13 shows the image processing pipeline together with its relevant peripherals (Camera Selector is omitted). First, the camera feed is binarized. Next, markers are detected, and their coordinates stored in a Double Buffer. Finally, the coordinates are transferred to the Vision Add-On using SPI. The resulting signal path on the GECKO5Education is illustrated in figure 2.14.

Binarize

The first stage of the pipeline the greyscale video feed received from the Vision Add-On into a binary video feed. Inputs of the module are a DVP video signal as described in 2.2.4 and the CI interface as described in 2.2.5. The module outputs the binary video signal using the same DVP interface as the input. This is done so that the binarized video can be sent to the Vision Add-On same as the greyscale video. As show in listing 7, the 8-bit `data` pixel value is compared to a threshold, which can be set using a CI (see 2.2) from the soft-core. If the pixel value is greater or equal to the threshold, the 8-bit `binarized` pixel value is set to white (`0xFF`), otherwise it is set to black (`0x00`). Together with the horizontal reference signal and vertical sync pulse, the resulting value is clocked to the modules output on



GRY: greyscale DVP signal
BIN: binary DVP signal
BB: bounding box

Figure 2.13: Overview of image processing pipeline

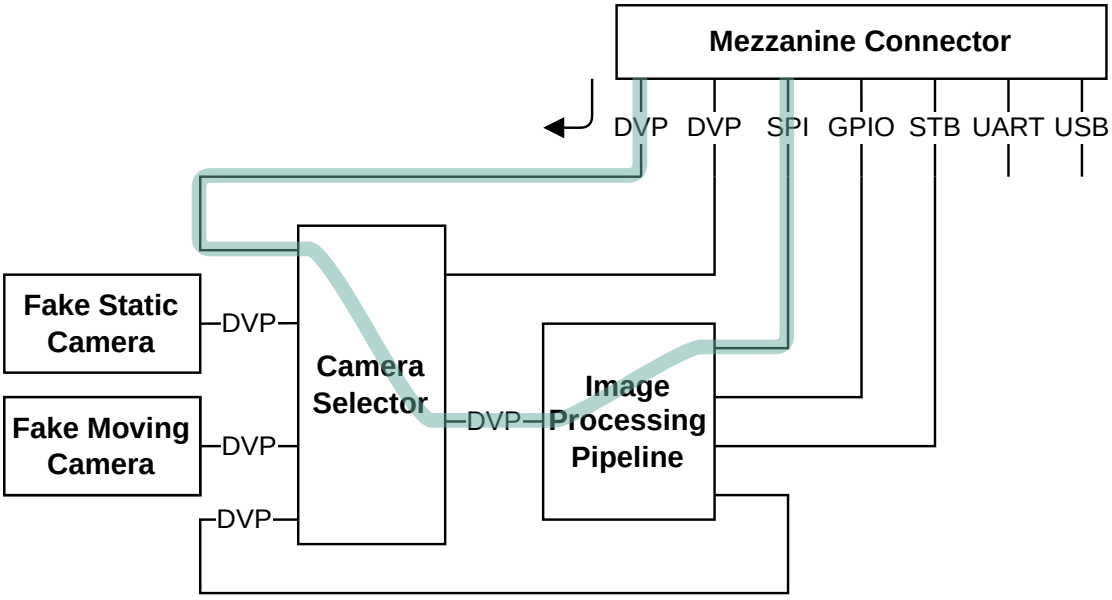


Figure 2.14: Marker detection path on the GECKO5Education

the rising edge of the pixel clock. The result is a video signal which only contains black and white pixels. It serves as both the input of the Connected Component Analysis (CCA) module and the Camera Selector module, which in turn can send the video signal back to the Vision Add-On.

```
1  wire [7:0] binarized = (data >= threshold) ? 'hFF : 'h00;
2
3  always @(posedge pclk) begin
4      hrefBin <= href;
5      vsyncBin <= vsync;
6      dataBin <= binarized;
7  end
```

Listing 7: Greyscale to Binary DVP conversion

A[31:0]	B[31:0]	RES[31:0]	Description
0		threshold	read threshold value
1	threshold		write threshold value

Table 2.2: Binarize CI description

CCA

Overview The CCA component is responsible for detecting blobs in the binarized image and determining their position within the frame. A blob is the region formed by a group of pixels which are connected. A connection is established if any neighbouring pixel shares the same value. In the case of a binarized image, pixels with the value `0` form the background, pixels with the value `1` form blobs.

Algorithm In [1] LinkRunCCA [9] was determined to be a suitable algorithm to detect the bounding boxes of blobs in a binary steamed image. The bounding box is the smallest rectangle (perpendicular to the image axis), which fits the blob. Figure 2.15 shows the bounding box of a circle. LinkRunCCA can produce bounding boxes at the same rate that the video is streamed into the component. No additional dead times are required at the end of a frame. As soon as a blob is complete, its bounding box is available at the components `bb` port shown in figure 2.16. The `valid` port indicates if the `bb` is valid. LinkRunCCA offers no drawbacks as long as the video is steamed sequentially. This makes LinkRunCCA a suitable choice for a DVP based video signals.

Integration Figure 2.16 shows how the output of the Binarize module serves as the input of the LinkRunCCA module. Since LinkRunCCA only needs a binary pixel value, only the

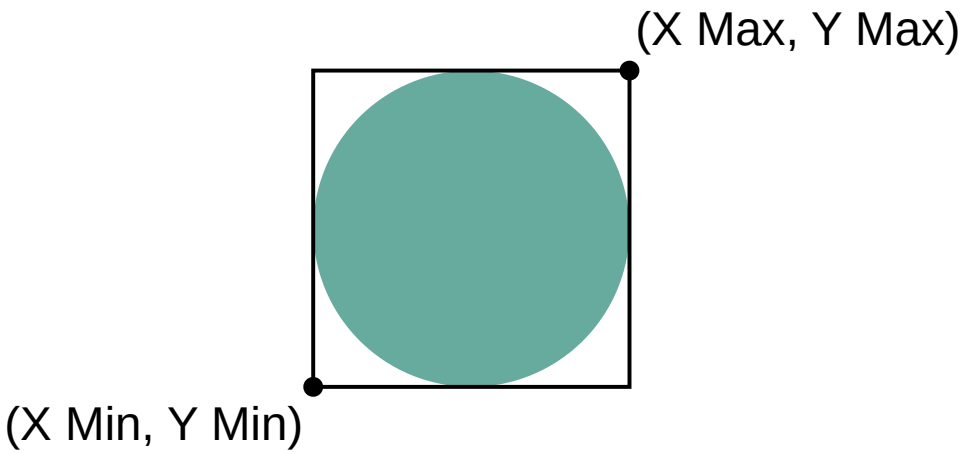


Figure 2.15: Bounding box of a circle

least significant bit (LSB) of the `data` signal serves as the binary input for the LinkRunCCA module. The `valid` signal indicates if the current pixel is valid, which is the case whenever `href` and `vsync` (low active) are high. `pclk` generated by the camera is the LinkRunCCA's clock source. LinkRunCCA does not use any line or frame sync pulses, to stay in sync with the video stream. The module relies solely on the width and height of the video signal which is configured in HDL. Instead, `reset` is used to synchronize LinkRunCCA with the incoming video stream. `reset` is triggered in-between each frame using the falling edge of `vsync`. This way, it is ensured that the LinkRunCCA module never goes out of sync.

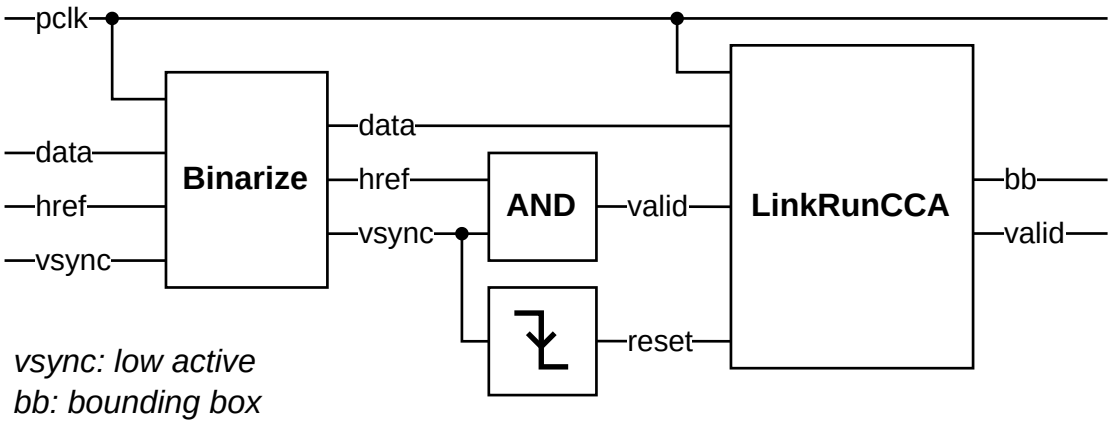


Figure 2.16: LinkRunCCA input logic

Feature Transfer

Overview The Feature Transfer module makes the output of the `CCA` module, the detected features on a frame, and makes them available to the Vision Add-On using SPI.

Double Buffer Since LinkRunCCA operates in the *pixel clock* clock domain and the Feature Transfer module in the *system clock* clock domain (also referred to as the CPU clock domain), a component is required to cross between the two clock domains. The two different clock domains are shown in both 2.13 and 2.19 using two different colours. Not handling clock domain crossing properly can lead to metastability [27]. Metastability occurs whenever a flip-flop samples a signal which is not in its steady state (still rising or falling). The bounding box data is transferred using the Double Buffer shown in figure 2.13 as part of the pipeline and in greater detail in figure 2.19. The Double Buffer hosts two dual ported SRAM modules. Note that Double Buffer must use dual ported SRAM, since the read and write clocks differ. The SRAM is configured to write data on the rising edge and read data on the falling edge of the respective clock. Using multiplexers and de-multiplexers, the buffer management logic can select which SRAM is connected to the write input of Double Buffer (`dataIn`) and which is connected to the modules read output (`dataOut`). One buffer is populated with new data in the producer's clock domain, while data from the other buffer is read in the consumer's clock domain. Once the producer has completed its write access, the two SRAMs can be swapped using the `switch` signal. While the consumer now has access to the new data, the producer can populate the new buffer.

Double Buffer does not prevent the producer from swapping the buffers while the consumer is still reading. This would result in loss of data. Since this project swaps the buffers at a constant rate and the consumption rate of the data is at a known, higher rate than the production, the buffers can be safely swapped by the producer without feedback from the consumer.

Figure 2.17 shows how the input of the Double Buffer is connected to its upstream peripherals. For ease of use, the Double Buffer's write input was designed to be a FIFO. The data producer does not have to worry about incrementing the address whenever new data is written to the buffer. The `dataIn` port is connected to the LinkRunCCA's `bb` output. With some additional logic, the `valid` output of LinkRunCCA is connected to the `writeEnable` port of Double Buffer. Any bounding boxes which are detected after the buffer is full are lost. The Double Buffer supports configurable buffer depth and width. The width must be set to the width of the bounding box output by the LinkRunCCA module, which can be calculated using 2.7 and the following constants:

- image width $w_{image} = 1280$
- image height $h_{image} = 800$

$$w_{buffer} = 2 * (ceil(log2(w_{image})) + ceil(log2(h_{image}))) \tag{2.7}$$

The resulting buffer width w_{buffer} is 42 bits. The buffer depth can be tuned to match the system's throughput. The higher the depth, the more blobs can be tracked per frame. In the current implementation, the depth is set to 7-bit, resulting in space for $2^7 - 2 = 126$ bounding boxes to be buffered. The Double Buffer's `switch` port is connected to the camera's `vsync` port via falling edge detection. As a result, the buffers are switched in-between each frame.

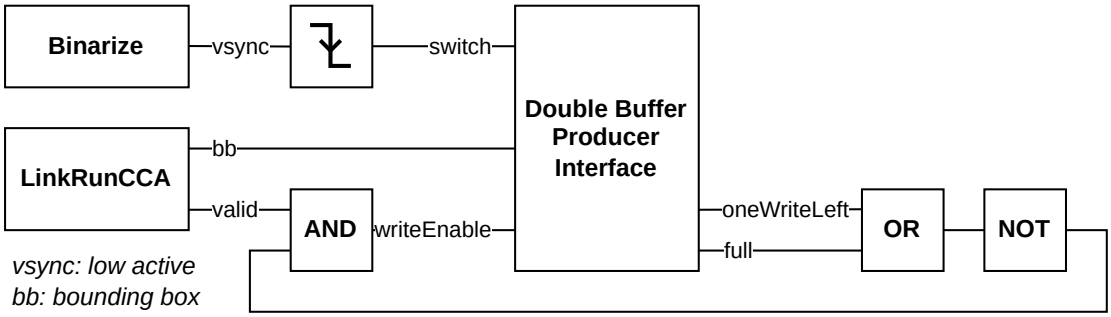


Figure 2.17: Double Buffer producer side

The only signal that needs to be protected against metastability is the `switch` signal. This is done by using the Synchronous Flop module shown in figure 2.18, which was already part of the project template. The Synchronous Flop module ensures that the signal is stable when it reaches the consumer clock domain. The signal acts as a synchronization flag for any other data which crosses the clock domains.

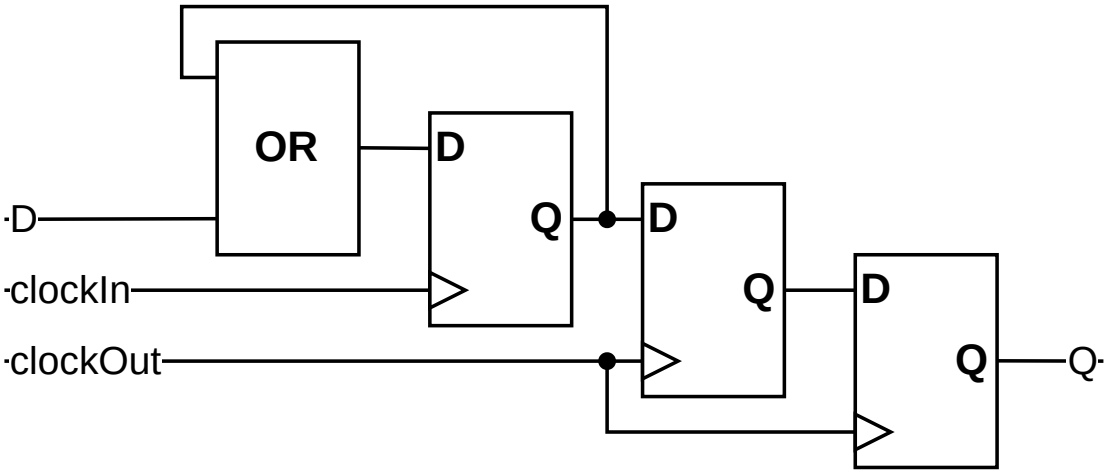


Figure 2.18: Synchronous Flop module

The buffer's consumer interface is discussed in 2.2.6, the buffer management logic will not be discussed in this document.

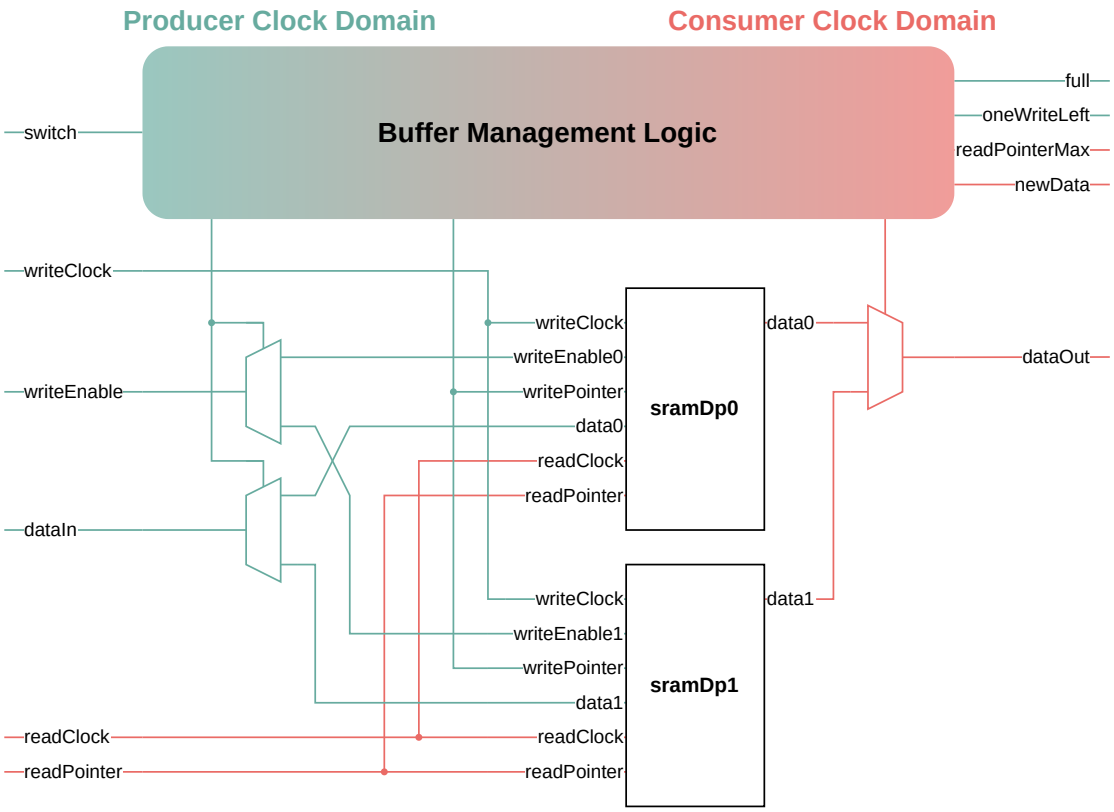


Figure 2.19: Double Buffer module

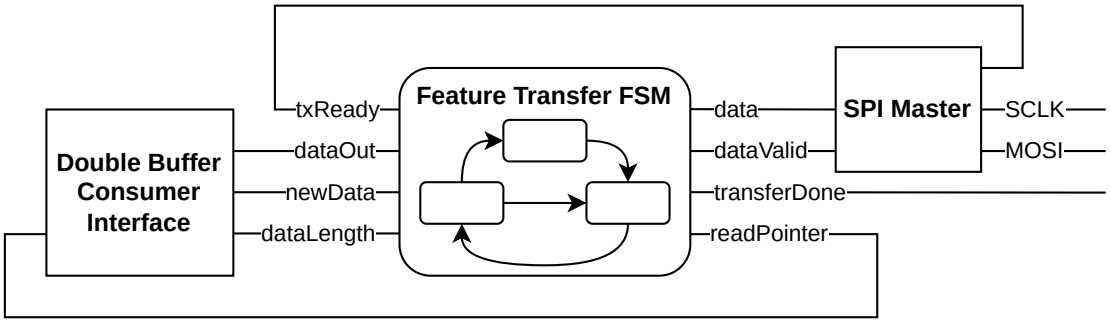


Figure 2.20: Double Buffer consumer side

SPI Figure 2.20 shows the consumer side of the Double Buffer. The finite state machine (FSM) shown in 2.22 controls the readout logic of the Double Buffer and forwards the data to the SPI Master module developed by nandland [28].

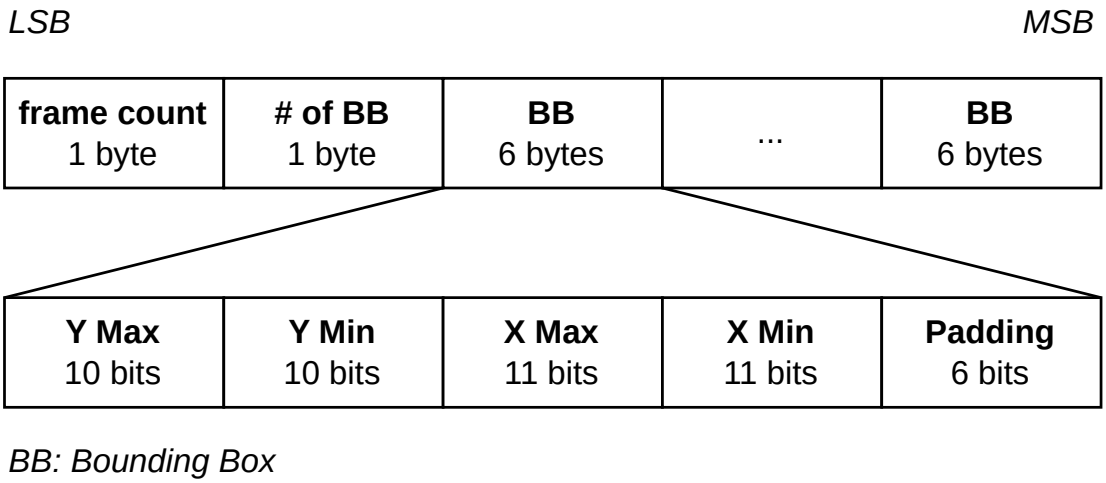


Figure 2.21: Feature Transfer packet

Figure 2.21 shows the structure of the data sent over SPI. The first byte transmitted is the current frame count, which is the output of an 8-bit counter. In increments by one with each rising edge of `vsync`. The second byte indicates the number of bounding boxes. Next, the detected bounding boxes are transmitted. Each bounding box entry is made up by two points, `(X Min, Y Min)` and `(X Max, Y Max)` (see 2.15) and some padding up to the next byte. If no bounding boxes were detected in the previous frame, only the frame count and the number of bounding boxes (0) are transmitted.

The SPI Master module show in figure 2.20 reads a single byte (`data`) when `dataValid` is set to 1. It is only allowed to set `dataValid` to 1 if the SPI Master module is ready to transmit, indicated by `dataValid`. Whenever a byte has been read by the SPI Master module, it is sent out over SPI by driving the `SCLK` and `MOSI` signals in accordance to the SPI specification. During SPI transmission, `txReady` is set to 0 by the SPI Master module.

The FSM shown in 2.22 drives the SPI Master module. In the *Idle* state, the FSM waits for new data to read from the Double Buffer. The `newData` pulse generated by the Double Buffer module initiates a new SPI transfer, which leads the FSM to transitions to the *Send Frame Count Wait Ready* state. Once the SPI Master module signals that it is ready to transmit a byte by raising the `txReady` signal, the FSM transitions to the *Send Frame Count Pulse Valid* state. There, the FSM sets the `data` signal to the current frame count, pulses the SPI Master's `dataValid` input signal and then transitions to the *Send Length Wait Ready* state. While the SPI Master module transmits the frame count, the FSM transitions to the *Send Length Wait Ready* state, where it waits for the `txReady` signal. Once `txReady` is 1, the FSM transitions to the *Send Length Pulse Valid* state. In the *Send Length Pulse Valid*

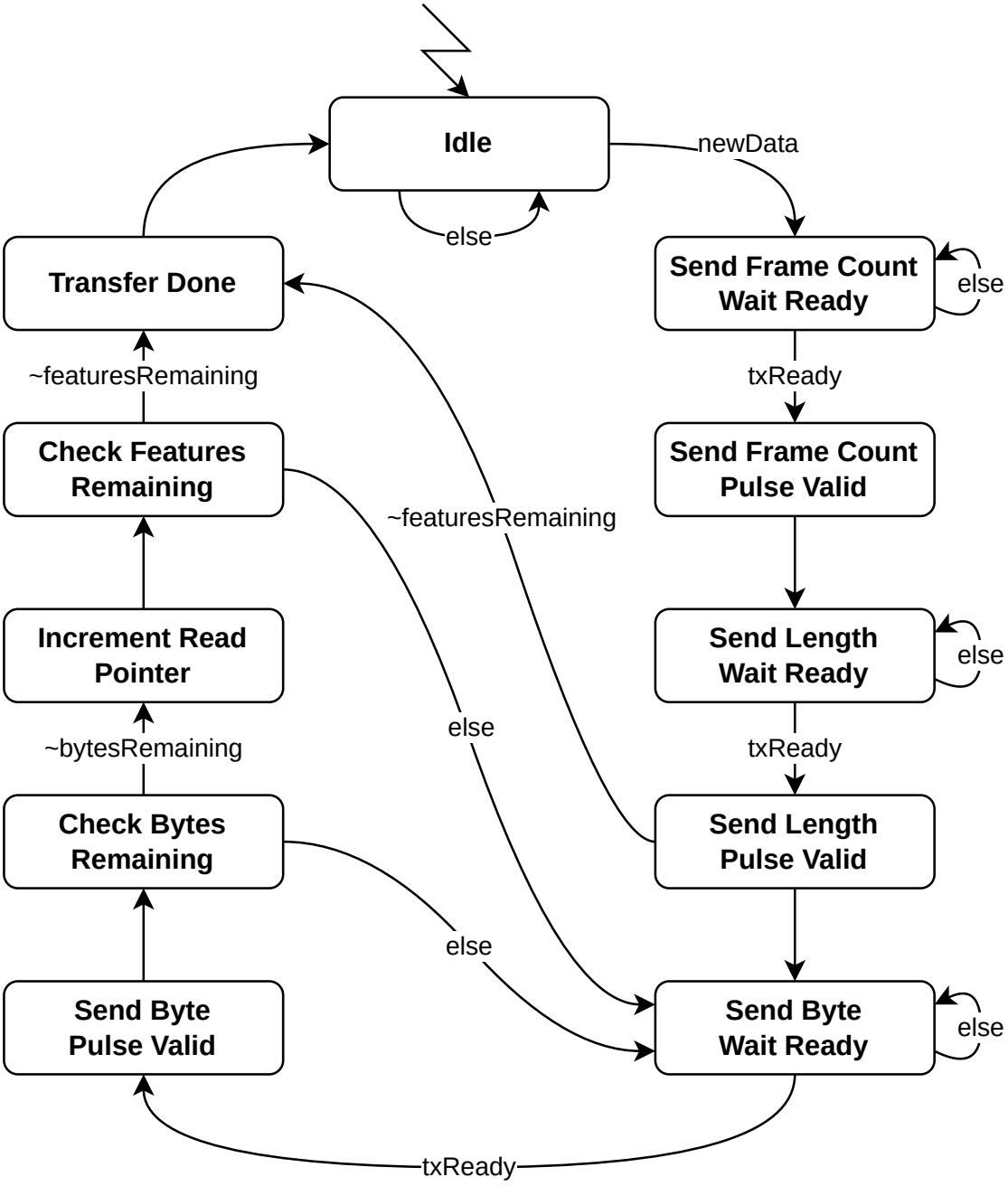
state, the data signal is set to the dataLength signal output by the Double Buffer module. The dataLength signal indicates how many bounding boxes are currently stored in Double Buffer and thus are to be transferred over SPI. If this value is zero, no bounding boxes were detected in the previous frame. In that case, the featuresRemaining signal is low and the FSM transitions to the *Transfer Done* state. Otherwise, the FSM pulses the dataValid signal so that the SPI Master module sends out the number of bounding boxes over SPI. The FSM continues to the *Send Byte Wait Ready* state and once the SPI Master module is ready further continues to the *Send Byte Pulse Valid* state. This is the start of the bounding box transfer, where the first byte of the first bounding box is transferred by setting SPI Master data input to the first byte of the Double Buffers dataOut output. The FSM moves on to the state where it performs a check if all bytes of the current bounding box have been transferred. If there are still bytes remaining, the FSM returns to the *Send Byte Wait Ready* state transmit the next byte of the Double Buffers dataOut output. Since one bounding box consists of six bytes and the SPI Master module only reads one byte at a time, six send cycles are required per bounding box. Once all 6 bytes of the bounding box have been sent to the SPI Master, the FSM transitions to the *Increment Read Pointer* state The Double Buffer's readPointer is used to select which buffered bounding box is accessible over the dataOut output. Since the next bounding box is to be transferred, the read pointer is incremented by one. Next, the FSM moves on to the *Check Features Remaining* state, where the read pointer is compared to the number of bounding boxes to be transmitted. If the read pointer is smaller than the number of bounding boxes to be transmitted, the FSM once again returns to the *Send Byte Wait Ready* state. However, this time the next bounding box will be transmitted since the Double Buffer's read pointer was incremented. Back in the *Check Features Remaining* state, if the read pointer is equal to the number of bounding boxes to be transmitted, the FSM enters the *Transfer Done* state, where the transferDone signal is pulsed. This signal connected to one of the Vision Add-On's GPIO pins and acts as an interrupt to signal that the SPI transfer has completed and the data can be processed. Finally, the FSM returns to the *Idle* state, where it awaits new data.

The FSM also has an *Error* state, which is not shown in figure 2.22. All states except the *Idle* state transition to the *Error* state if newData is true. The *Error* resets the internal logic of the FSM and transitions to the *Idle* state.

All the logic on the consumer side of the Double Buffer is driven by the system clock and not the pixel clock. As a result, the SPI transfer speed remains independent of how fast the camera's pixel clock is running.

Strobe Control

The Strobe Control module generates a pulse, the STB signal seen on 2.6, in sync with the camera frame rate. This pulse is used by the Vision Add-On to drive an IR flash. The vsync signal is used as the source of the STB signal. The STB is pulsed on the rising edge of vsync. Using a CI, the pulse width and delay relative to vsync signal can be adjusted. There is also an option to constantly enable the STB. Table 2.3 lists the implemented CI of the Strobe



Error state not shown

Figure 2.22: Feature Transfer Finite State Machine

Control module.

A[31:0]	B[31:0]	RES[31:0]	Description
0	enable		enable strobe control
1	delay		write the pulse delay in pixel clock cycles
2	hold time		write the pulse hold time in pixel clock cycles
3		delay	read the pulse delay in pixel clock cycles
4		hold time	read the pulse hold time in pixel clock cycles
5	enable		enable constant output

Table 2.3: Strobe Control CI description

2.2.7 Top Level Model

The top level model (`<hdl>/systems/singleCore/verilog/or1420SingleCore.v`) is what instantiates all the different modules shown in the dotted area in figure 2.6. The CIs and their IDs used are listed in table 2.4. The gaps in the CI IDs are due to the fact that some modules were removed from both the initial project template and previous versions of the project. A memory map of the system is documented in figure 2.23.

2.2.8 Vision Add-On

Motivation and System Requirements

In [2], the perifAddOn was used to transmit the detected marker coordinates over Wi-Fi to a central system. The perifAddOn is an add-on board for the GECKO5Education, which connects to the FPGA over the mezzanine connector located on the GECKO5Education. The add-on board hosts an ESP32-C2-MINI-1 SoC, which has a built-in Wi-Fi radio. However, the transmission over Wi-Fi proofed to be unreliable and highly dependent on how many other devices operated in the same frequency range as the perifAddOn. Next to the unreliable Wi-Fi connection, the perifAddOn is limited in other regards, which lead to the decision replace the perifAddOn with a custom add-on board.

CI ID	CI Name
0	Text Controller
1	Swap Bytes
2	SPI Flash
3	*not used*
4	CPU Frequency
5	I2C
6	Delay
7	*not used*
8	*not used*
9	*not used*
10	Bus Error Counter
11	Binarize
12	Camera Selector
13	*not used*
14	Strobe Control

Table 2.4: CI IDs of the blob detector's FPGA design

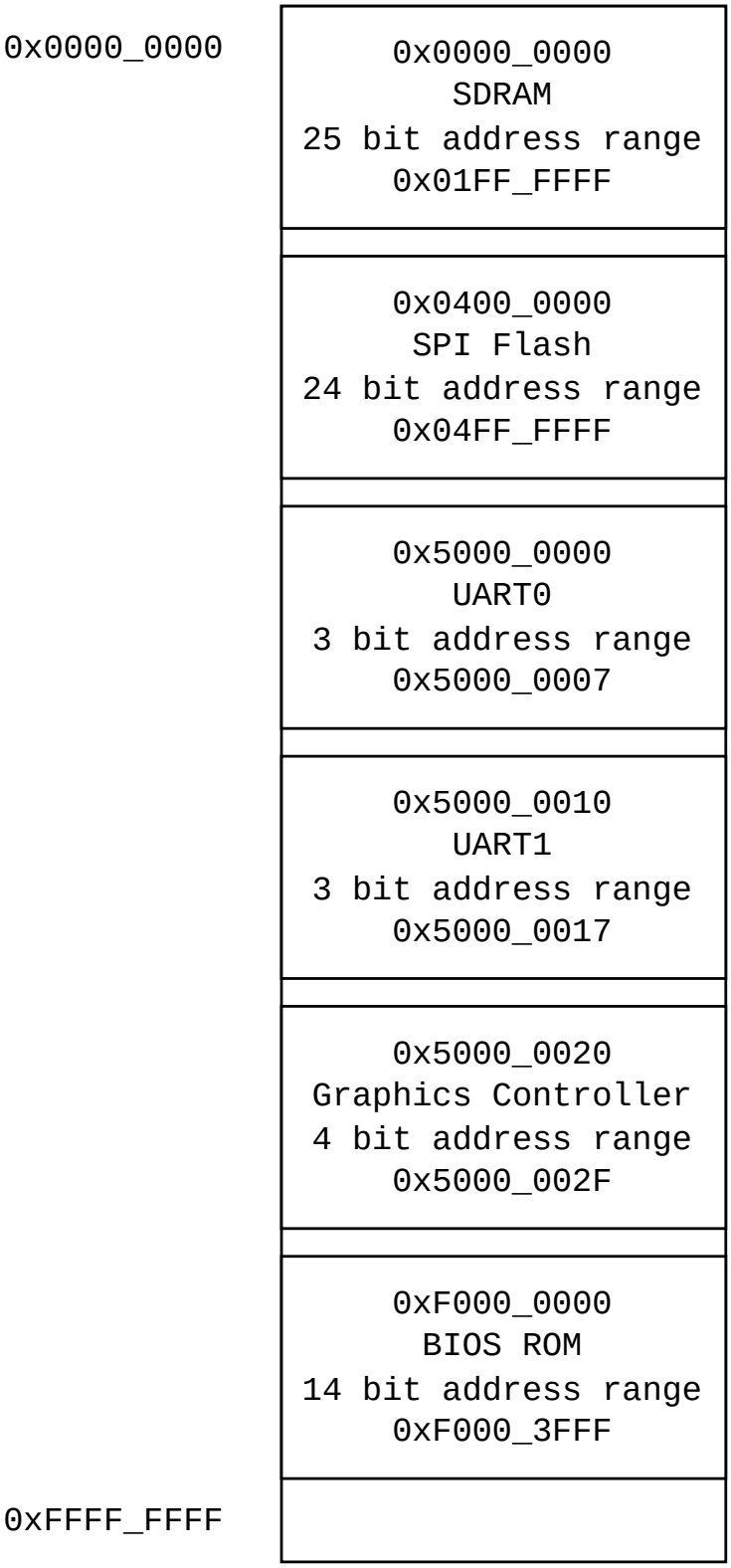


Figure 2.23: Memory map of the Blob Detector's FPGA design

The following requirements need to be met by the new add-on board, the Vision:

- enable the use of a high frame rate IR camera
- receive marker coordinates from the GECKO5Education and forward them over Ethernet
- capture a frame of the camera feed
- send the captured image frame over Ethernet
- powered by Power over Ethernet PoE
- store configuration parameters in non-volatile memory
- drive an IR flash

Figure 2.24 shows the two operation modes the Blob Detector is intended to operate in. The main mode is shown in 2.24a where the camera feed sent from the Vision Add-On to the GECKO5Education to detect the position of markers. The marker positions are then sent back to the Vision Add-On, where they are received my the microcontroller and published over Ethernet.

In the second mode shown in 2.24b, the camera feed is relayed back to the Vision Add-On, where a single frame is captured and published over Ethernet. This operation is further described in 2.2.8.

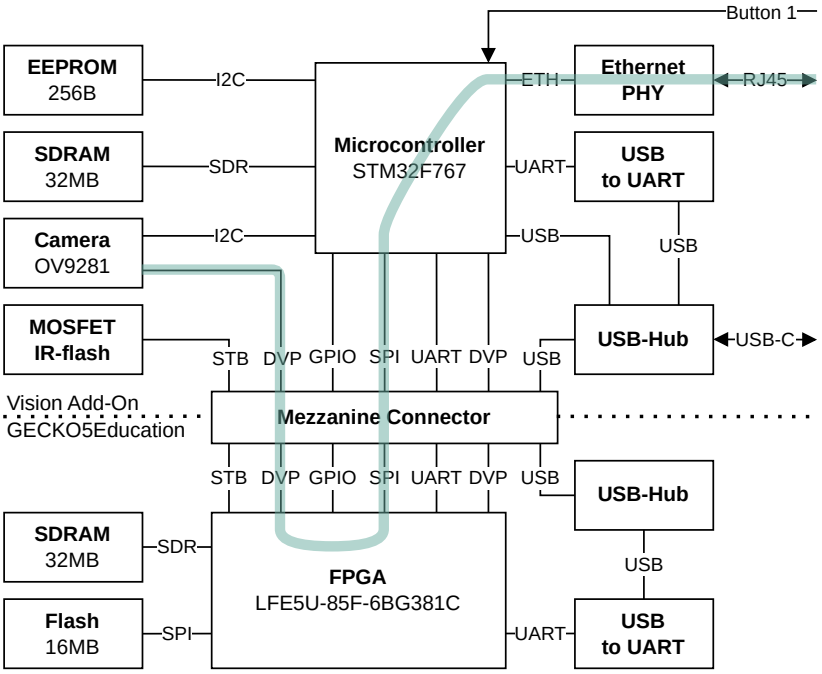
PCB Design Software and physical Stackup

KiCad [29] is an open-source PCB design software. Since the GECKO5Education was designed using KiCad, and the project offers a KiCad based add-on template [30], the Vision Add-On was also designed using KiCad version 8.0. The board's physical stackup differs from the template. Figure 2.25 shows the six layer stackup used. Inner layer 1 and 4 are used for high speed signals. Inner layer 2 is the ground plane Inner layer 3 is the 3.3V plane.

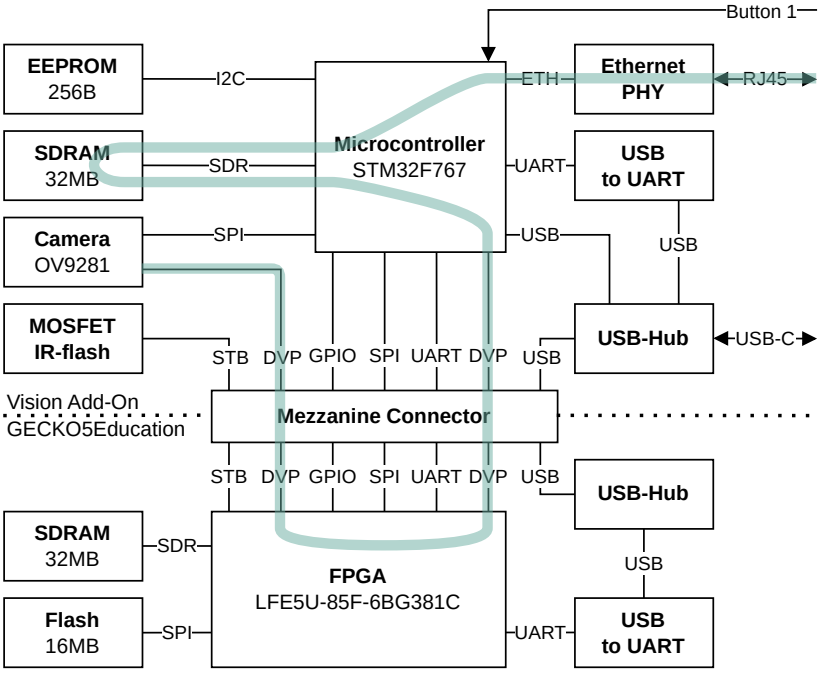
Camera

Requirements As discussed in the conclusion on [2], the OV7670 camera is not suited for an IR-based MoCap system without further modification. The camera is limited in resolution and frame rate (640x480 at 30 fps) and its optics have an IR filter installed. A faster, higher resolution camera, which is sensitive to the non-visible Near Infrared NIR light spectrum is required. The NIR spectrum starts at 750nm, which marks the end of the visible spectrum.

MIPI Camera Serial Interface 2 (CSI-2) [31] is the industry standard when it comes to high speed camera interfaces. Many camera sensor manufacturers offer sensors, which imple-



(a) Blob Detector marker path



(b) Blob Detector camera snapshot path

Figure 2.24: Blob Detector operation modes

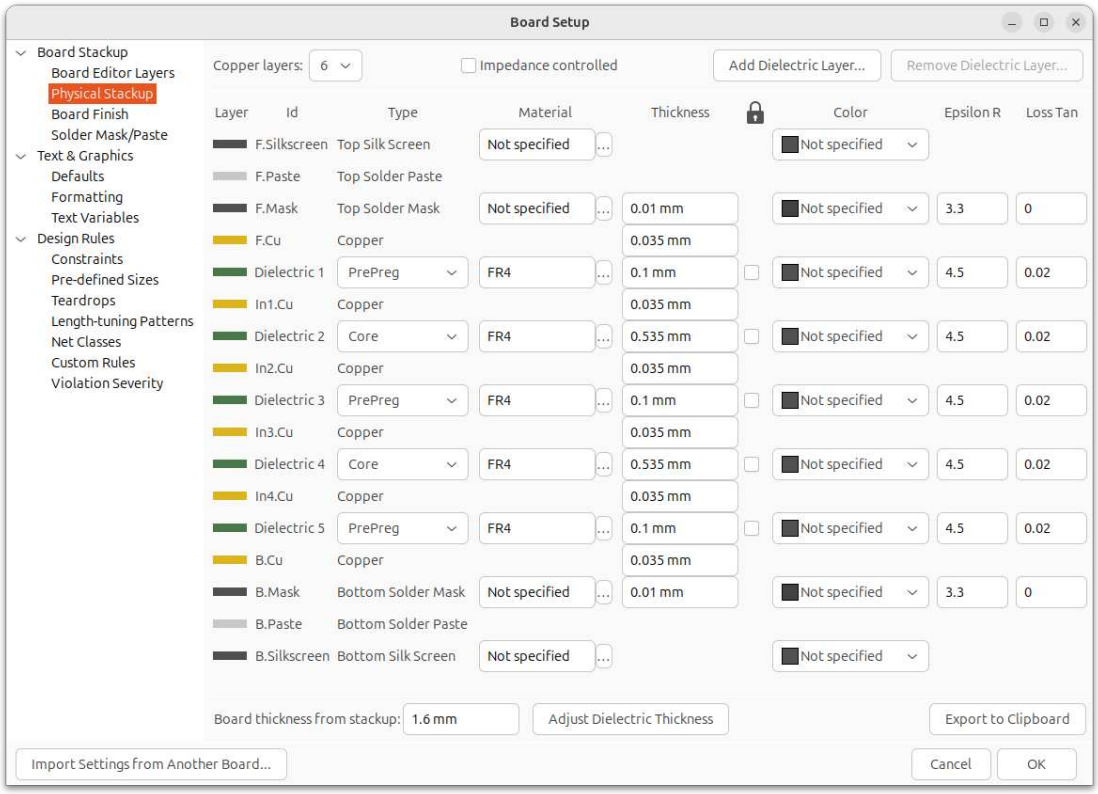


Figure 2.25: Vision Add-On physical stackup

ment the MIPI CSI-2 interface. However, the MIPI CSI-2 specification is not publicly available. Accessing the specification requires payment and signing a Non Disclosure Agreement (NDA). Implementing the specification is out of scope for this project.

There are different vendors, which sell IP-Cores which implement the MIPI CSI-2 specification for use on an FPGA. However, these IP-Core usually require its licensee to sign a NDA, which is against the open-source spirit of this project.

There exists an open-source implementation of the MIPI CSI-2 interface based on leaked specifications. The implementation is part of a MIPI CSI-2 to USB-C camera, the project sources can be found at [32] and a blog post about the project at [33]. Extracting the MIPI CSI-2 implementation from the project and integrating it into the GECKO5Education was deemed too risky to pursue in this project.

In [2], the use of the TC358748XBG MIPI adapter chip [34] was proposed. This chip takes a MIPI CSI-2 camera signal and converts it to a DVP video signal (DVP is explained in 2.2.4). Both [1] and [2] already worked with a DVP signal. Using the adapter chip would open up the use of any MIPI CSI-2 compliant camera without knowledge of the interface. However, this adapter chip adds additional complexity to the system. Ideally, the camera itself outputs a DVP signal.

To sum up, the camera needs to meet the following requirements:

- high frame rate - the higher, the better
- high resolution - the higher, the better
- sensitive to NIR spectrum
- DVP output

Omnivision OV9281 camera sensor The Omnivision OV9281 camera sensor, which outputs both MIPI CSI-2 and DVP signals [35]. The sensor is a high fps, global shutter image sensor. It outputs a 10-bit greyscale image. The maximum resolution of the sensor is 1280x800 pixels. At this resolution, the sensor can output 120 fps. The spectrum response of the OV9281 is shown in 2.26. The graph was taken from the OV9281 data sheet [35]. Even though the Quantum Efficiency (QE) seen on the graphs Y-Axis does not have any units or numerical values assigned to it, it is apparent that the sensor is most sensitive in the visible light spectrum between 500nm and 700nm. However, the sensor is also sensitive above 750nm just at a lower rate. By using an optical filter that blocks light bellow 750nm and raising the exposure and gain of the sensor, it seems plausible that the sensor can be used in and IR-based MoCap system. To test how sensitive the OV9281 sensor is to the NIR spectrum, the Waveshare OV9281-110 Mono Camera [36] was used. The Waveshare camera uses the OV9281 sensor but outputs the signal over MIPI CSI-2. A Raspberry Pi 4 Model B is capable of decoding the camera signal and offer a live preview of what the sensor sees. With this camera setup and an IR LED, it was confirmed that the OV9281 sensor

is indeed suited for IR-based MoCap applications before incorporating the sensor into the Vision Add-On's design.

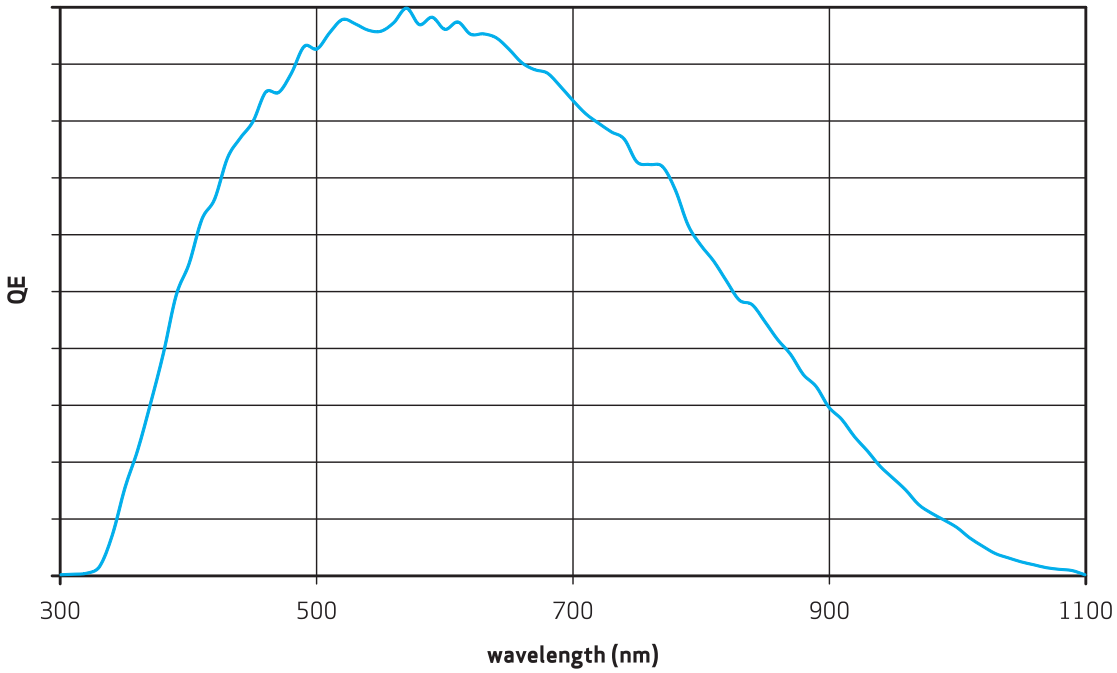


Figure 2.26: Omnivision OV9281 spectrum response [35]

One big drawback is that the OV9281's data sheet is not publicly available. Omnivision support was contacted multiple times to request access to the sensor's data sheet. However, no reply was ever received. Luckily, anyone trying to sell you a camera based on the OV9281 sensor will gladly provide you with the data sheet and application notes.

KLT-E4MPF-OV9281 V4.2 NIR The KLT-E4MPF-OV9281 V4.2 NIR, manufactured by Kai Lap Technologies, is a fixed focus camera that uses the OV9281 sensor. The in the `NIR` version, the camera does not have an IR filter, which makes it suitable for IR-based MoCap applications. The camera is shown in 2.27. It is designed to be used with the DF30FC-24DS-0.4V connector (number 6 in 2.29) manufactured by Hirose Electric. The camera can be used in both MIPI CSI-2 or DVP mode. When used in DVP mode, only eight of the ten data bits are available. The two least significant bits (LSB) of the sensor's parallel data port are not connected to the camera's connector. This results in a slight loss in resolution but is not significant for this project.

Schematics Within the KiCad project, the camera schematics are located on page 2, labelled camera. The design is based on Antmicro's OV9281 Dual Camera Board [37], the

Figure 2.27: KLT-E4MPF-OV9281 V4.2 NIR camera

schematics of which are openly available. The Dual Camera Board hosts two OV9281 sensors, multiple Low-dropdout regulators (LDOs) to generate the voltages that the sensor requires, a level shifter to convert the 1.2V I2C interface of the sensor to 3.3V, a clock source and some additional components.

The schematics of this project are more or less half of the Antmicro board with the OV9281 sensor replaced with the DF30FC-24DS-0.4V connector to be able to connect the KLT-E4MPF-OV9281 V4.2 NIR camera. The camera's I2C interface used to configure the sensor is connected to the Vision Add-On's microcontroller (peripheral I2C1, address 0xC0) as shown in 2.5. The DVP output is connected to the mezzanine connector and forwarded to the GECKO5Education.

Microcontroller and External SDRAM

The driving forces for choosing the STM32F767ZIT6 (number 11 in 2.29) as the microcontroller of the Vision Add-On were its Digital Camera Interface (DCMI) and the Flexible Memory Controller (FMC) and how they can operate together. The DCMI receive a DVP camera signal such as the one generated by the OV9281 and buffers it in either internal or external RAM using the microcontroller's Direct Memory Access (DMA). This functionality is used to capture a frame from the camera feed, which is needed to calibrate the camera. Camera calibration is covered in 2.4.8.

Since the STM32F767ZIT6 does not have enough internal SRAM (512kB) to buffer a whole

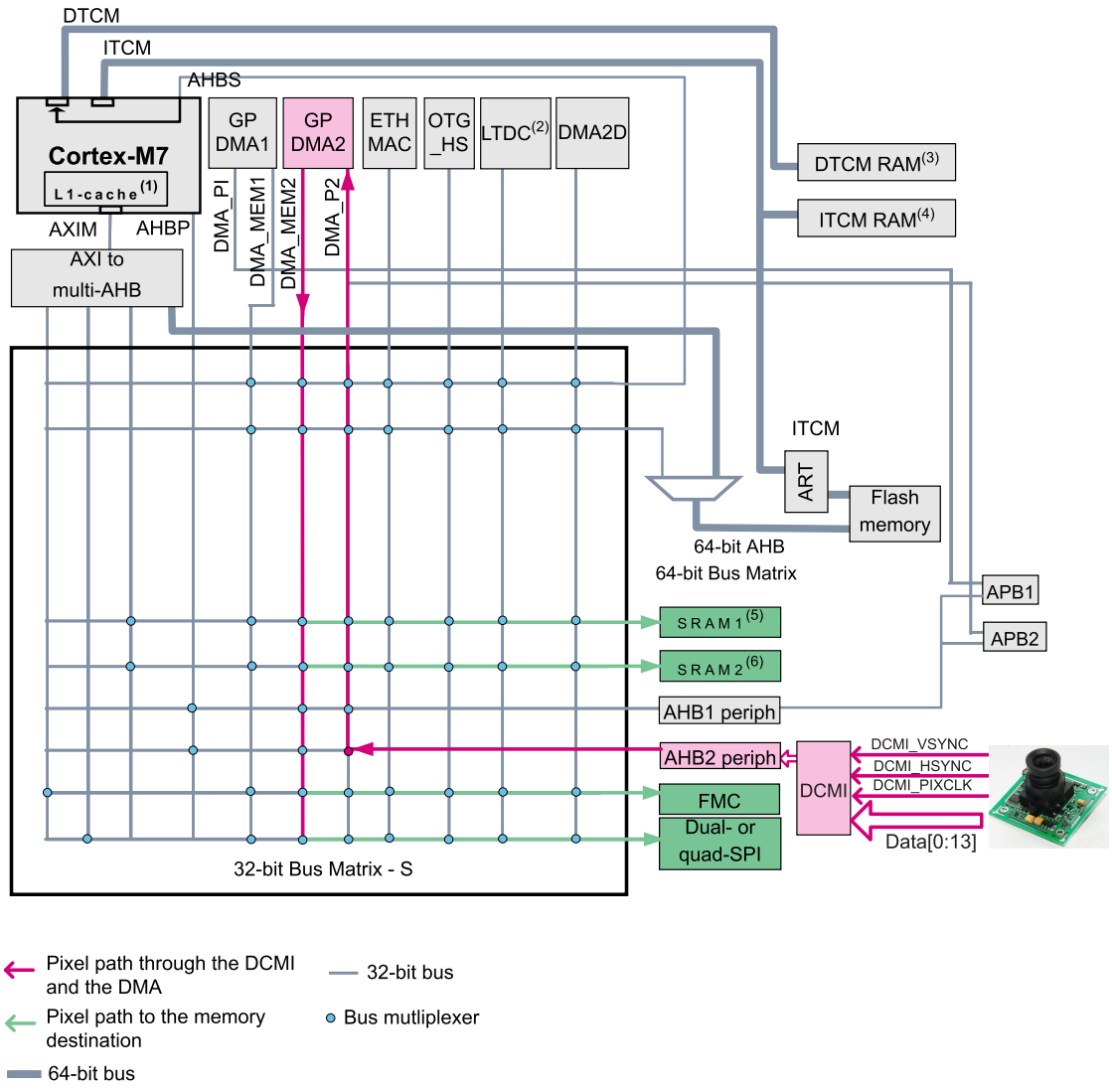


Figure 2.28: DCMI slave AHB2 peripheral in STM32F7 [38]

frame ($1280 * 800 * 1B \approx 1MB$), the Vision Add-On needs to host SDRAM connected to the microcontroller using its FMC. Figure 2.28 taken from [38] shows that the STM32F7 supports using DMA2 to transfer data from DCMI to FMC and thus external SDRAM.

Since the Vision Add-On's camera is not directly connected to the microcontroller but to the GECKO5Education instead, the FPGA needs to relay the DVP signal back to the microcontroller, where it is then captured and transferred to the SDRAM. This path is shown in 2.24b. Relaying data back from the FPGA to the microcontroller also allows the Vision Add-On to capture the binarized image output by the Image Processing Pipeline or the output of the Fake Cameras.

The SDRAM (32MB) (number 10 in 2.29) used on the Vision Add-On is the same as on the Leguan development board [39]. This was done in part to be able to copy the schematic, but also to copy the SDRAM timing configuration used in the Leguan's board support package (BSP). All traces running from the SDRAM to the microcontroller were length matched using KiCad's meandering feature.

Ethernet

Ethernet is used for both communication and power delivery. The Murata MYBSP0055AABFT (number 15 in 2.29) is an IEEE 802.3at Class 4 PoE PD module than can supply up to 25.5W at 5V. This is more power than the Vision Add-On can use in its current configuration. Murata also offers the MYBSP00502ABF IEEE 802.3af Class 0 PoE PD module, which only supplies 10W at 5V. This is a more reasonable choice. However, the 10W module was out of stock during the design phase of the Vision Add-On, so the 25.5W version is used instead. The Murata PoE module is connected to the RJ45 jack (number 17 in 2.29), which provides it with 48V. The transformer's 5V output is buffered using large capacitors.

The LAN8742A (number 14 in 2.29) is an Ethernet Physical Layer (PHY). It is connected to the RJ45 jack and the microcontroller, enabling Ethernet based communication for the microcontroller. It is the same PHY that is used on the STMicroelectronics NUCLEO-F767ZI evaluation board, the schematics of which are publicly available. The NUCLEO-F767ZI's Ethernet design was adapted to the Vision Add-On. The differential pair traces running between the RJ45 jack and the Ethernet PHY were drawn using KiCad's differential pair tool, their length was kept at a minimum. To minimized signal reflections, the differential pair traces should be impedance matched to 100Ohm. KiCad does not offer an impedance matching tool to directly draw impedance matched traces. Instead, KiCad's Calculator Tools were used to calculate the impedance manually.

EEPROM

To be able to store configuration values such as a network configuration in non-volatile memory, a 256B EEPROM (number 8 in 2.29) is connected to the microcontroller using I2C (peripheral I2C4, read address 0xAE, write address 0xAF).

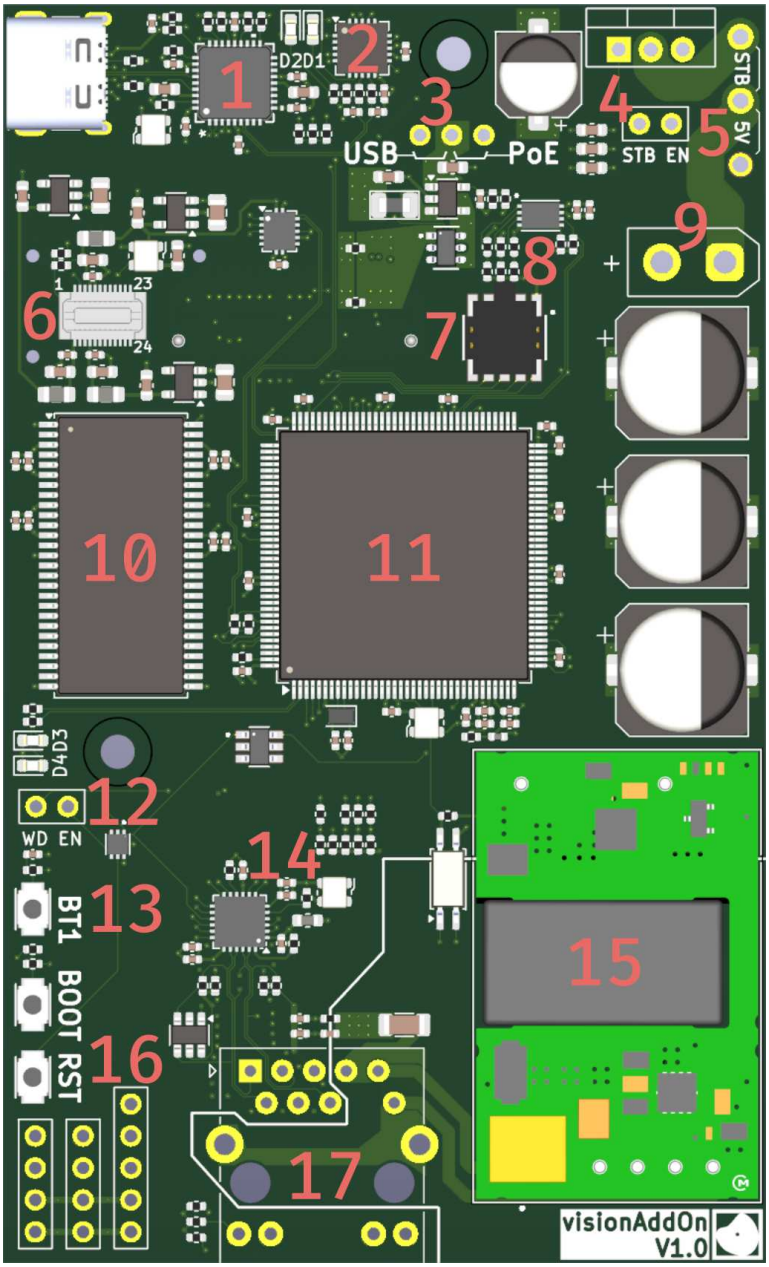


Figure 2.29: KiCad render of Vision Add-On PCB

1: USB Hub
2: USB to UART Bridge
3: Vision Add-On Supply Select
4: Strobe Enable
5: Flash Supply Select
6: Camera Connector
7: Programmer Pins
8: EEPROM
9: Flash Connector
10: SDRAM
11: Microcontroller
12: Voltage Supervisor
13: Buttons
14: Ethernet PHY
15: PoE Module
16: IO Expansion
17: RJ45 jack

USB

The USB2513B USB hub (number 1 in 2.29) is connected to a USB-C connector, a USB to UART bridge (FT231XQ) (number 2 in 2.29), the microcontroller and the mezzanine connector. It is the same USB hub that is used on the GECKO5Education. The design was copied for the Vision Add-On. If the Vision Add-On is connected to the GECKO5Education, the GECKO5Education's USB-C port is disabled and the Vision Add-On's is used instead. The serial port of the USB to UART bridge is connected to the microcontroller (UART1). The Vision Add-On can be supplied by either 5V USB power or by the PoE module. A jumper (number 3 in 2.29) is used to select between the two options.

All USB data traces were drawn using KiCad's differential pair tool and kept as short as possible.

IR Flash

The Vision Add-On provides a 5V output (number 9 in 2.29) via an XT30 connector. An N-channel MOSFET is used to switch the 5V output of the PoE module. The gate of the MOSFET is connected to the strobe signal coming from the FPGA via the mezzanine connector (4 in 2.29). As a result, the GECKO5Education can control the IR flash from the FPGA. If the Vision Add-On is powered over USB, the IR flash is not powered The MOSFET can also be bypassed using a jumper (number 5 in 2.29) so that the XT30 connector is directly connected to the 5V PoE output. This feature can be used if the IR flash uses a current source chip such as the Texas Instrument (TI) LM3644 Camera Flash LED Driver.

The IR LEDs used in this project are the FH-R668 IR Floodlight (sold by [40]), a 2x3 IR LED matrix driven by 5V.

Reset logic

The MIC826 voltage supervisor chip (12 in 2.29) is used to generate a reset or enable signal for all the chips which require one. The supervisor chip monitors the 5 V supply voltage. If the voltage drops bellow a critical threshold, the reset signal is asserted and held. Only once the voltage rises back over the threshold, is the reset signal released. This results in a clean power-on behaviour of the whole Vision Add-On.

The voltage supervisor chip can also act as a watchdog timer that needs to be kicked in regular intervals by the microcontroller. Failure to kick the watchdog results in a controlled reset of the whole board. The watchdog feature can be enabled using a jumper (12 in 2.29). Pressing the Vision Add-On's reset button (12 in 2.29) also triggers the watchdog's reset function.

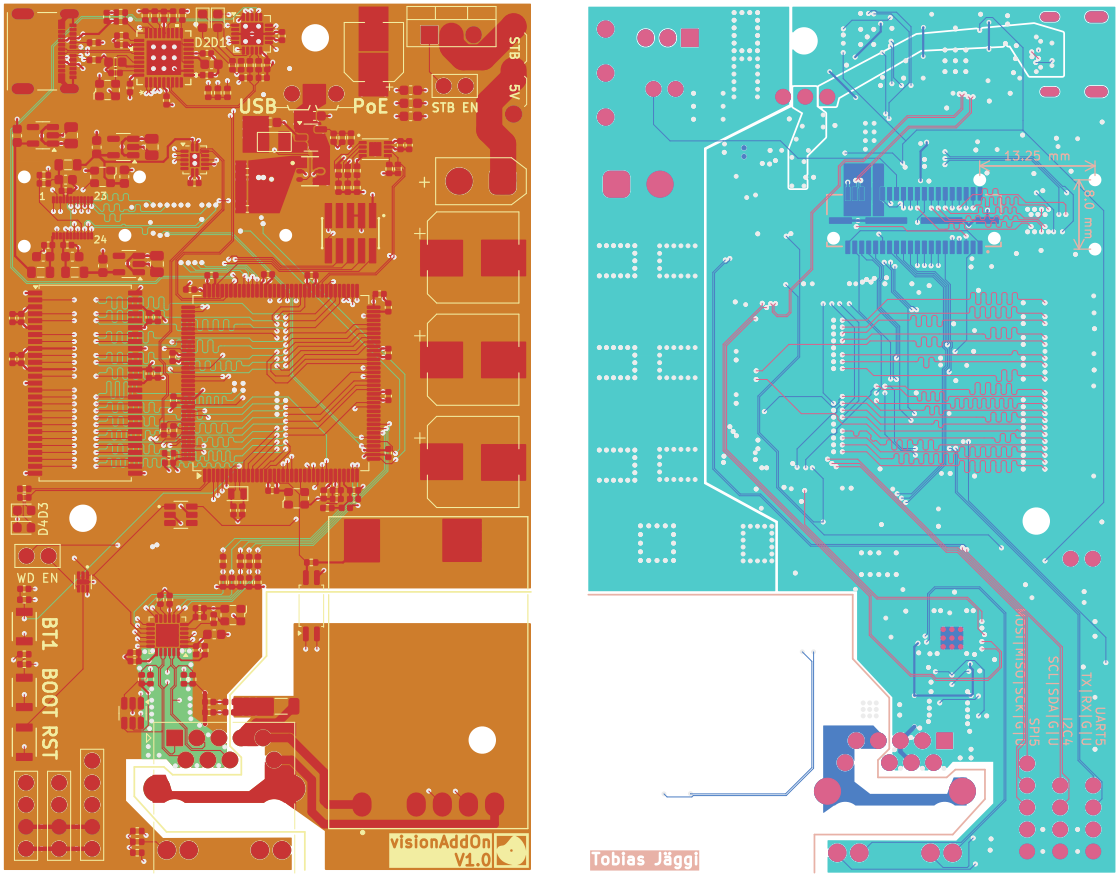
User IO

Pin (16 in 2.29) expose additional IO from the microcontroller (UART5, I2C4, SPI5) Additionally, one GPIO pin of the microcontroller is connected to a button (13 in 2.29). Two LEDs are also driven by the microcontroller (12 in 2.29).

2.2.9 Manufacturing

Figure 2.30 shows all six layers of the completed PCB layout.

Each layer is represented by a colour.



(a) Front layers (b) Back layers

Front outer layer: Inner layer 1: Inner layer 2: Inner layer 3: Inner layer 4: Back outer layer:

Figure 2.30: Vision Add-On PCB layout

The design files were sent to Eurocircuits for manufacturing. Six units were ordered. The components listed on the bill of material (BOM) were sources from Mouser and Digikey.

Assembly of the PCBs was done by the electronics apprenticeship program at the Bern University of Applied Sciences (BFH).

2.2.10 Mechanical Design

Onshape is a cloud based CAD software. Its use is free for non-commercial purposes. Onshape was used to design a frame to mount the PCBs of the Blob Detector alongside the camera, an optical filter, the IR-flash and a ball joint. Figure 2.31 shows a render of the finished CAD design. An optical band filter, which is transparent to wavelength from 700 nm to 1600 nm (sold by [41]) is installed on the camera mount (1 in 2.31). The ball joint (3 in 2.31) allows the frame angle to be adjusted after it is mounted. By tightening the nut, the joint is locked and holds its position. All parts of the frame were printed on a Fused Filament Fabrication (FDM) 3D printer. Polylactide (PLA) filament was used for all parts except the pins which bond the Vision Add-On and the GECKO5Education. Thermoplastic Polyurethane (TPU) was used for the pins for its elastic properly.

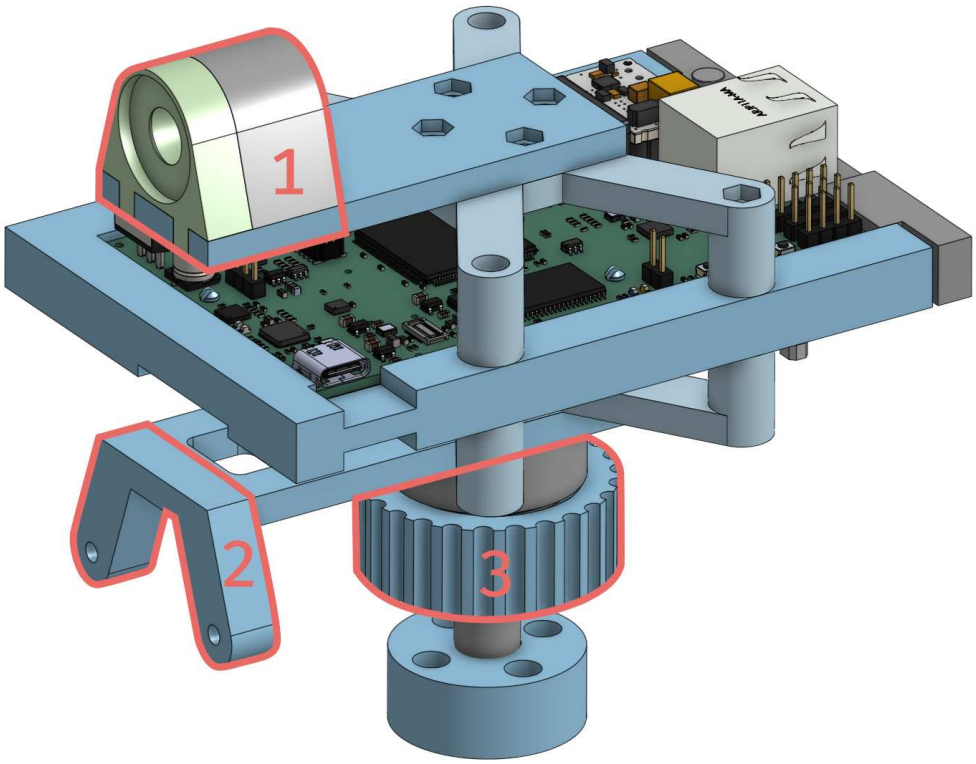


Figure 2.31: onshape render of frame

1: Camera and Optical Filter Mount
2: IR Flash Mount
3: Ball Joint

2.3 Firmware

2.3.1 OR1420 soft-core

Toolchain

Building The or1k_toolchain is a GCC based toolchain, tailored to the OR1420 architecture.

A zip file containing the toolchain is located at `<hdl>/tools/or1k/or1ktoolchain.zip`. The BIOS, which can load a new program into SDRAM or Flash requires a different file format (*.cmem*), from what is output by the or1k_toolchain (*.elf*). The convert_or32 utility takes care of that translation step. A precompiled version of the tool is located at `<hdl>/tools/or1k/convert_o32/convert_o32`. To install both the toolchain and the conversion utility, follow the installation instructions documented in the README at `<hdl>/tools/or1k/README.md`.

> Compiling the convert_or32 sources located at `<hdl>/modules/bios/util` and using the resulting binary to convert the *.elf* files does not work. The soft-core does not boot when the *.cmem* file output by the utility is loaded. Instead, use the precompiled binary mentioned above.

To compile the program used in this project, open a shell at `<hdl>/programs/blobDetector` and run the `make` command. The `make` command executes the build steps specified in the makefile. The build binary which can be flashed to the GECKO5Education is located at `<hdl>/programs/blobDetector/build-release/blobDetector.cmem` after the build process is completed.

> The toolchain does not include the C standard library.

Flashing Connect the Blob Detector to the host PC using USB-C. The GECKO5Education's USB port is disabled if the Vision Add-On is connected to the board. Use the Vision Add-On's USB port instead. Use a serial terminal to establish a connection between the host PC and the Blob Detector. This document's instructions are based on CuteCom [42], a graphical serial terminal shown in figure 2.33. In CuteCom, select the device in the *Device:* dropdown list and open the connection by pressing the *Open* button. Make sure the line feed option *LF* is selected in the dropdown list next to the *Input:* field. In the *Settings* ▾ dropdown menu, confirm that the settings match the ones shown in figure 2.32. To flash the binary to the GECKO5Education, first enter the BIOS by pressing the reset button (8 in 2.1) while holding the top button on the right edge (9 in 2.1) of the GECKO5Education. Once the device enters the BIOS, it outputs a help screen on both the HDMI output and over the serial connection as seen in figure 2.33. In CuteCom, input `*p` in the *Input:* field and press return to send the command to the Blob Detector. The Blob Detector is now in programming

mode, where a new program can be loaded into the device's SDRAM. In CuteCom, click the *Send file...* button and select the *.cmem* file (located in `<hdl>/programs/blobDetector/build-release`) in the file explorer pop-up. After the selection is confirmed, the program is sent to the Blob Detector. The program can either be persisted in the SPI-Flash by sending the `*f` command or directly started with the `$` command. After a reset, the Blob Detector automatically loads and starts a program located SPI-Flash. If no program is present, the device remains in the BIOS.

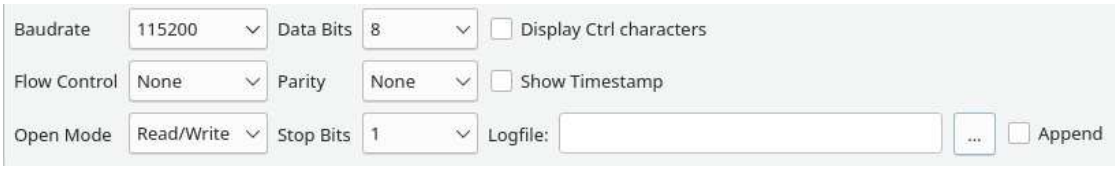


Figure 2.32: CuteCom Settings ▾ dropdown menu

Overview

The image processing and feature transfer happen completely independantly of the soft-core. The C program running on the soft-core only acts as a bridge to make the CIs available to the Vision Add-On (UART1) and a user via USB (UART0). The BSP is located at `<hdl>/programs/blobDetector/support`, the main entry point of the program running on the soft-core is located at `<hdl>/programs/blobDetector/support/src/blobDetector.c`. If a hardware peripheral is connected to the system bus, it is referred to as a memory-mapped peripheral. Refer to the memory map in figure 2.23 for an overview of the addresses of the memory-mapped peripherals.

Hardware initialization

After booting, the soft-core initializes both UART0 and UART1 and then enters an infinite loop. The baud rate is set to 115200 bit/s, 8 data bits, 1 stop bit and no flow control are used. The UART hardware is exposed to the soft-core as a memory-mapped peripheral. To apply the configuration to the UART hardware, the BSP's `uart_init` function is used. The function requires a pointer to the memory-mapped peripheral (see 2.23). Within the infinite loop, the program handles any incoming messages received on UART0 and UART1.

Custom Instructions

This section describes how to call a CI based on the Binarize module. The same methodology applies to all other CIs of the Blob Detector.

As documented in table 2.2, the binarization threshold can be read using the Binarize module's CI interface. To address the Binarize module, the signal `ciN` shown in figure 2.9 needs

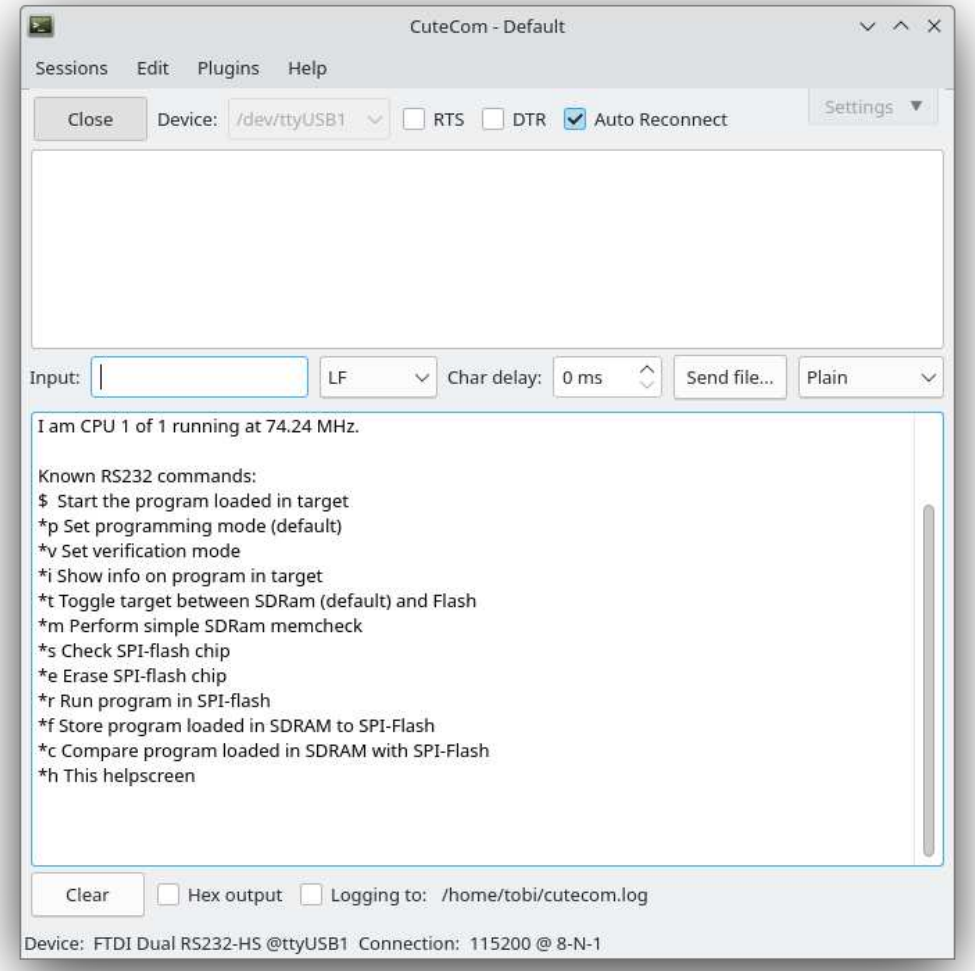


Figure 2.33: CuteCom showing the BIOS start message of the Blob Detector

to be set to the module's CI ID. Table 2.4 lists all IDs. In the case of the Binarize module, the ID is 11 (0xB). To read the threshold, ciDataA in figure 2.9 is set to 0, ciDataB in figure 2.9 is ignored. To set the required hardware signals form the program running on the soft-core, assembly statements as seen in listing 8 are used.

```
1  #include "binarize.h"
2
3  // binarization ci
4  static const int CI_BIN_A_READ_THRESHOLD = 0;
5  static const int CI_BIN_A_WRITE_THRESHOLD = 1;
6
7  bool binarizeSetThreshold(uint32_t threshold){
8    static const uint32_t THRESHOLD_MAX = 0xff;
9    if (threshold > THRESHOLD_MAX) {
10     return false;
11   }
12   asm volatile ("l.nios_rrr r0, %[ra], %[rb],
     ↪  0xB"::[ra]"r"(CI_BIN_A_WRITE_THRESHOLD), [rb]"r"(threshold));
13   return true;
14 }
15
16 uint32_t binarizeGetThreshold(){
17     uint32_t threshold;
18     asm volatile ("l.nios_rrr %[res], %[ra], r0,
       ↪  0xB":[res]"=r"(threshold):[ra]"r"(CI_BIN_A_READ_THRESHOLD));
19     return threshold;
20 }
```

Listing 8: Binarize BSP

Line 18 shows the assembly statement that reads the threshold value. The statement `l.nios_rrr` tells the compiler that the assembly statement is a NIOS-II instruction. `0xB` is the ID of the CI. The value of `CI_BIN_A_READ_THRESHOLD` is the argument A of the CI. The signal `ciDataA` is set to this value. The value returned by the CI is `res` (signal `ciResult` in figure 2.9). It is stored in the variable `threshold`. The signal `ciDataB` is not used and thus set to `r0`. To write the threshold, the argument A of the CI is set to the value of `CI_BIN_A_WRITE_THRESHOLD`, argument B is set to the desired threshold. The return value can be ignored (set to `r0`). Writing the binarization threshold is shown on line 12 in 8.

Command Handler

To allow both the user and the Vision Add-On to configure settings such as the binarization threshold or camera selection at runtime, a command handling mechanism was implemented. Commands can be input using UART0 (user via USB) or UART1 (from Vision

Add-On). Listing 9 shows the Command Handler's help text documenting its supported commands.

```
help text - 'h'
  h - print help text
input selector - 'i'
  io - ov9281 camera as input
  is - fake static image as input
  im - fake moving image as input
output selector - 'o'
  or - raw image as output
  ob - binary image as output
bin threshold - 't'
  t<value> - set bin threshold to <value>
  <value>: [0-2^16-1]
strobe control - 's'
  strobe trigger: camera vsync delayed for "delay for" then held for
  ↪ "hold for"
  se<enable> - enable/disable strobe output
  sd<value> - set "delay for" value to <value>
  sh<value> - set "hold for" value to <value>
  sc<enable> - enable/disable constant output
  <enable>: [0-1], 0=disable, 1=enable
  <value>: [0-2^32-1]
```

Listing 9: Help text for the serial commands implemented by the soft-core

Commands such as changing the binarization threshold require the ability to input an integer value over the serial interface. Since the toolchain does not provide a C standard library implementation, a custom string to integer function was implemented based on [43]. To apply the user's configuration, the CI interface of the respective module is used. The user command's help text also shows up on the HDMI output, since all `printf` statements are both sent to UART0 and the text render engine, which is part of the HDMI controller module in figure 2.6.

2.3.2 Vision Add-On

Toolchain

Rust At first, the goal was to develop the firmware for the Vision Add-On using the Rust Programming Language [44]. Rust's promises about memory and thread safety with no compromise to its performance make it an interesting alternative to C and C++ for embedded systems programming. Bern RTOS [45] is a real time operating system (RTOS) developed in Rust. It was developed by a former student of BFH during his master's degree.

The RTOS is compatible with the STM32F7 family of ST's microcontrollers and thus the microcontroller on the Vision Add-On.

The Vision Add-On was successfully running Bern RTOS together with the smoltcp [46] network stack. However, kernel instability of the RTOS ultimately lead to the decision to switch from Rust to C++ and a different RTOS. This was accounted for in the risk analysis [47] which was done for the Vision Add-On.

C++

STM32CubeMX The Vision Add-On's firmware project is set up using STM32CubeMX [48] version 6.13.0. All hardware peripherals and the microcontroller's pinout are configured using STM32CubeMX. The program generates a project with a hardware abstraction layer (HAL) of the selected microcontroller and initialization code based on the selected configuration. The generated code is in C. As the build system, CMake is used in this project. STM32CubeMX is able to generate the required CMake files. Next to hardware configuration, STM32CubeMX also includes support for third party libraries. FreeRTOS is a C-based open-source RTOS supported by STM32CubeMX. LwIP is a C-based open-source network stack supported by STM32CubeMX. Both FreeRTOS and lwIP are enabled in the STM32CubeMX project. The lwIP stack runs in its own FreeRTOS task running at a period of 100 ms. `<firmware>` denotes the root of the Vision Add-On firmware project. The STM32CubeMX source file is `<firmware>/visionAddOn.ioc`.

The firmware project is set up to use C++ instead of C. However, STM32CubeMX generates a main.c file instead of a main.cpp file. Before generating any new code using STM32CubeMX, rename the file `<firmware>/Core/Src/main.cpp` to `<firmware>/Core/Src/main.c`. After the code generation is complete, revert the file's name back to `<firmware>/Core/Src/main.cpp`. Additionally, manually update the CMake project by replacing the string *../../Core/Src/main.c* with *../../Core/Src/main.cpp* within the file `<firmware>/cmake/CMakeList.txt`.

FreeRTOS uses a cooperative scheduler. If a task never yields, all other tasks starve.

Building and Flashing This document's instructions are based on VS Code. To build the firmware, open the firmware project folder (`<firmware>`) in VS Code. Install the project's recommended plugins when prompted by VS Code. Next, open the VS Code Command Palette by pressing *Ctrl + Alt + P* and then input `CMake: build target` and press return. To flash the firmware to the Vision Add-On, connect it to the PC using a compatible debug probe to the board's JTAG connector (7 in 2.29). The STLINK-V3MINIE debugger was used

by the author. Either supply power via the Vision Add-On's USB port or the RJ45 jack. Remember to set the supply select jumper accordingly (3 in 2.29). In VS Code, press F5 to flash the Vision Add-On with the compiled firmware and start a debugging session.

Feature Transfer

Figure 2.34 shows how the SPI data sent from the GECKO5Education is processed by the Vision Add-On. At the start of the sequence, the SPI peripheral is already configured to receive incoming data using DMA. After the GECKO5Education is done transmitting data over SPI, it pulses the `transferDone` signal. This triggers the Vision Add-On's External Interrupt Handler, which is registered to the GPIO pin that receives the `transferDone` signal. The External Interrupt Handler stops the DMA-based SPI data reception and publishes the start address of the receive buffer as well as the number of bytes received to a message queue. Next, it acquires a new buffer from the Buffer Pool with which the DMA-based SPI data reception is re-started. The Blob Receiver periodically checks the message queue to which the External Interrupt Handler published its data for incoming messages. If a new message is present, the Blob Receiver consumes it. The Blob Receiver now knows the buffer address containing the received SPI data as well as how many bytes were received. The buffer address and the number of bytes are forwarded to a lwip UDP socket, which sends out the data contained in the buffer. From its reception over SPI to its transmission via UDP, the data is never copied. After publishing the data over UDP, the Blob Receiver continues checking for new data in the message queue. Notice that the acquired buffer is never released back to the pool. Internally, the Buffer Pool is a circular buffer, which means that there is no guarantee of ownership after acquiring a buffer from the pool. For the Vision Add-On, this is no issue since it can keep up with the incoming data rate. The Blob Receiver runs in its own FreeRTOS task at a period of 1 ms.

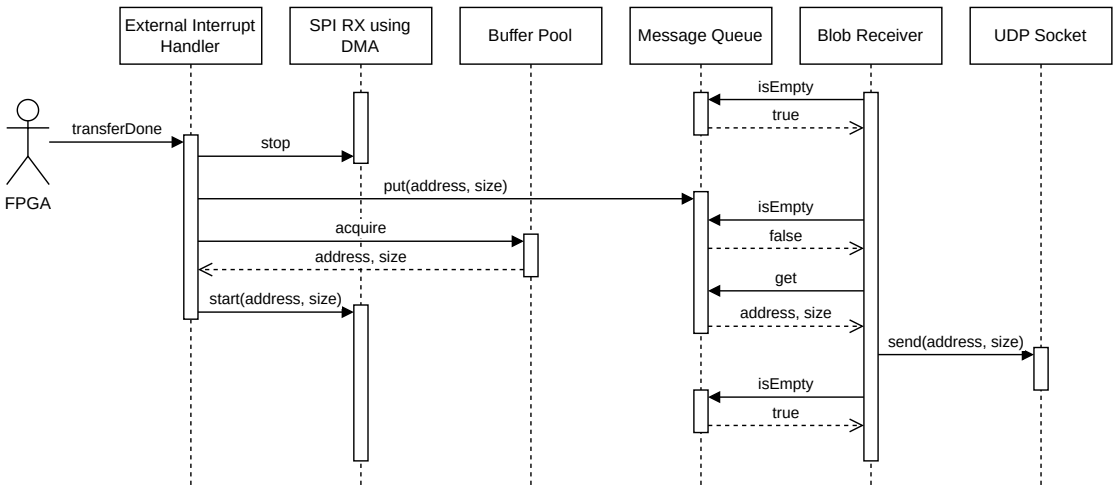


Figure 2.34: Sequence diagram showing how new SPI data is processed on the Vision Add-On

The Blob Receiver publishes the received SPI without further processing over UDP. See 2.21 for the packet structure.

Command Handler

To be able to control the Blob Receiver remotely, a command handling service was implemented. The Command Handler accepts incoming TCP connections on port 80. Figure 2.35 shows the packet structure of both the command request packet and its response. The `request id` field, interpreted as an unsigned 8-bit integer, is used match a response packet to its request. The `command id` field, interpreted as an unsigned 8-bit integer, is used to identify the command type and determines the formatting of the payload. The payload does not have a fixed size. It is defined by the `size` field, interpreted as an unsigned 8-bit integer. Its value represents the payload size in bytes. A command's success is indicated by the `complete` field in the response packet. It is set to 0x00 if the command failed, 0x01 if it was successful. Table 2.5 lists all the commands, which are implemented by the Command Handler.

The byte order of the command packets is little-endian.

LSB **Request** *MSB*

request id 1 byte	**command id** 1 byte	**reserved** 1 byte	**size** 1 byte	**payload** command id based formatting for size bytes

Response

request id 1 byte	**command id** 1 byte	**complete** 1 byte	**size** 1 byte	**payload** command id based formatting for size bytes

Figure 2.35: Command request and response packets

Figure 2.36 shows how to read the network configuration of the Vision Add-On using the `network_get_config` command. The reply sent by the Vision Add-On either contains the network configuration in the success case (`complete` field 0x01) or no payload at all in the failed case (`complete` field 0x00). It is also shown how to set a network configuration using the `network_set_config` command and the possible replies sent by the Vision Add-On. The complete list of all command packets and their payload structures implemented by the Command Handler can be found in the Markdown file ‹firmware›/App/command/-commands.md, an HTML export of which is handed in with this document [49]. A hosted version is found at [50]. The Command Handler runs in its own FreeRTOS task at a period of 100 ms.

command ID	command name
0x10	log_set_level
0x20	camera_request_capture
0x21	camera_request_transfer
0x22	camera_set_whitebalance
0x23	camera_set_exposure
0x24	camera_set_gain
0x25	camera_set_fps
0x30	network_get_config
0x31	network_set_config
0x32	network_persist_config
0x40	calibration_load_camera_matrix
0x41	calibration_store_camera_matrix
0x42	calibration_load_distortion_coefficients
0x43	calibration_store_distortion_coefficients
0x44	calibration_load_rotation_matrix
0x45	calibration_store_rotation_matrix
0x46	calibration_load_translation_vector
0x47	calibration_store_translation_vector
0x50	pipeline_set_input
0x51	pipeline_set_output
0x52	pipeline_set_binarization_threshold
0x60	strobe_enable_pulse
0x61	strobe_set_on_delay
0x62	strobe_set_hold_time
0x63	strobe_enable_constant

Table 2.5: List of commands implemented by the Vision Add-On

LSB *MSB*

Request network_get_config

rid	cid	rsvd	size	payload
42	30	00	00	

Response network_get_config **success**

rid	cid	cmpt	size	payload															
				MAC Address						IP Address				Netmask				Gateway			
42	30	01	12	00	80	E1	00	00	00	0A	00	00	0A	FF	FF	FF	00	0A	00	00	01

Response network_get_config **failed**

rid	cid	cmpt	size	payload
42	30	00	00	

Request network_set_config

rid	cid	rsvd	size	payload															
				MAC Address						IP Address				Netmask				Gateway			
42	31	00	12	00	80	E1	00	00	00	0A	00	00	0A	FF	FF	FF	00	0A	00	00	01

Response network_set_config **success**

rid	cid	cmpt	size	payload
42	30	01	00	

Response network_set_config **failed**

rid	cid	cmpt	size	payload
42	30	00	00	

rid: request id
cid: command id
rsvd: reserved
cmpt: complete

Figure 2.36: Command packet example showing `network_get_config` and `network_set_config` requests and their replies

Camera and Frame Transfer

The OV9281 camera is initialized over I2C by the Vision Add-On. The I2C initialization sequence is based on the Omnivision application notes for the OV9282 camera sensor [51]. However, the I2C sequence is compatible with the OV9281 sensor. The camera is initialized to output a 1280x800 image feed at 72 fps over the DVP interface. It is unclear if the camera supports the full 120 fps when running in DVP mode. The application notes only provide the I2C sequence to run the camera at 72 fps The data sheet only lists the supported resolution and frame rates when using the MIPI interface.

The max pixel clock frequency that the STM32F767ZIT6's DCMI peripheral can handle is 54 MHz [38]. When the OV9281 sensor is running at 72 fps, its pixel clock frequency is 96 Hz, which is too fast for the microcontroller. To capture the camera feed on the microcontroller, the pixel clock frequency first needs to be lowered. The Command Handler implements the command `camera_set_fps`, with which the cameras frame rate can be set to 13 fps by lowering the pixel clock. This is done by adjusting the OV9281 Phase-Locked Loop (PLL) configuration. As a result, the DCMI peripheral is able to process the incoming data. Note that the H7 family of microcontrollers by STMicroelectronics would be able to handle the 96 MHz pixel clock.

The command `camera_request_capture` configures the DCMI peripheral to buffer the next incoming frame in the external SDRAM using DMA. The Command Handler sends out the response packet once the frame has been transferred to the SDRAM or a timeout has occurred.

The SDRAM is directly mapped into the address range of the microcontroller via the FMC peripheral. The FMC peripheral needs to be initialized with the timing parameters of the SDRAM. This initialization was copied from the Leguan BSP.

The SDRAM memory is unmanaged. The Frame Transfer service is hardcoded to save the incoming frame to the SDRAM in the address range 0xC0000000 - 0xc2000000. Stay clear of this range when using memory from the SDRAM.

To read the stored frame, the command `camera_request_transfer` is used. The Command Handler calls the Frame Transfer service which reads the frame from the SDRAM in chunks. Each chunk is transmitted to a host IP using UDP (see 2.3.2). There is no further formatting applied to the data, the SDRAM data is sent out over UDP as is. Once the transfer is complete, the Command Handler send the command's reply packet.

FPGA Commander

The FPGA Commander service links the Vision Add-On's Command Handler to the Command Handler running on the GECKO5Education's soft-core. The Vision Add-On's Command Handler maps incoming commands intended for the GECKO5Education to functions implemented by the FPGA Commander. The FPGA Commander implements functions to

select the Image Processing Pipeline's input, select the Image Processing Pipeline's output, set the Image Processing Pipeline's binarization threshold and configure the Strobe Control module. To do so, it sends the corresponding string (see 9) to the GECKO5Education using the UART connection.

Static Logger

A logging service was implemented. It publishes the log messages both to a host IP over UDP (see 2.3.2) and over Serial Wire Viewer (SWO) if a compatible debug probe is connected. The log service is implemented as a static C++ class. As a result, its logging functions can be called from anywhere in the codebase without the need of a reference to the logger instance. The logger offers five methods to create log messages at different log levels, `trace`, `debug`, `info`, `warning` and `error`, as seen in listing 10.

```
// C++ interface
Log::trace("log message with level trace");
Log::debug("log message with level debug");
Log::info("log message with level info");
Log::warning("log message with level warning");
Log::error("log message with level error");

// get and set the log level
Log::info("the current log level is %u",
 ↪  static_cast<uint8_t>(Log::level()));
Log::level(Level::LOG_WARNING);
Log::info("this message won't be seen because its level is too low");

// C interface
log_trace("log message with level trace")
log_debug("log message with level debug")
log_info("log message with level info")
log_warning("log message with level warning")
log_error("log message with level error")
```

Listing 10: Static Logger calls

The `trace` level is the lowest log level, whereas `error` is the highest. The logger can be configured to only publish log messages above a certain level using the `level` method. The static logger can also be called from C source files using its C-style interface as shown in 10. ANSI escape codes are used to change to message formatting based on the log level. Be aware that log recipient needs to support this formatting and apply it when displaying the message.

The VS Code project is configured to open a SWO connection when debugging. The log messages appear in the VS Code terminal named `SWO:ITM`, which has support for the ANSI formatting.

Calling the logger from within an interrupt call can result in a mangled log message.

EEPROM

Table 2.6 shows the parameters which are stored in the non-volatile EEPROM memory of the Vision Add-On.

Address	Length	Name
0x10	0x06	MAC Address
0x18	0x04	IP Address
0x20	0x04	Netmask
0x28	0x04	Gateway
0x40	0x24	Camera Matrix
0x70	0x14	Distortion Coefficients
0x90	0x24	Rotation Matrix
0xc0	0x0c	Translation Vector

Table 2.6: Parameters stored in the Vision Add-On's EEPROM

Network Configuration

The Vision Add-On's IP address, netmask, gateway and MAC address can be configured using the Command Handler as shown in figure 2.36 using the command `network_set_config`. The configuration can be persisted on the EEPROM using the command `network_persist_config`. DHCP is not enabled, lwIP however does support it.

The Command Handler listens for incoming traffic on TCP port 80. Outgoing UDP traffic is hardcoded to be transmitted to the host IP address 10.0.0.2. The Frame Transfer service sends the image data to UPD port 1055, the Blob Receiver sends the bounding box data to UDP port 1056 and the Log is published to UDP port 1057.

The host IP address, TCP and UDP ports are defined as compile time constants in the file `<firmware>/App/constants.h`.

Pressing and holding button *BT1* (13 in 2.29) when the Vision Add-On boots, restores the default network configuration.

Locked Heap and Assert Handler

Since the STM32F767ZIT6 does not have a memory management unit and exception handling is disabled in the compiler's configuration, it is not advised to use allocate and deallocate heap a runtime. A Memory fragmentation can lead to failed heap allocation and thus undefined behaviour. The Vision Add-On's firmware uses GCC's `--warp` linker flag to wrap the function `malloc`. As a result, GCC creates two functions. One named `__real_malloc`, another named `__wrap_malloc`. Any calls to `malloc` result in a call to `__wrap_malloc`. Listing 11 uses this mechanism to lock the use of `malloc` and thus the heap after the function `lock_heap` was called. Prior to that point, calls are forwarded to `__real_malloc`.

```
1  static bool heap_locked = false;
2  void lock_heap() { heap_locked = true; }
3
4  void *__real_malloc(size_t size);
5
6  void *__wrap_malloc(size_t size) {
7      if(heap_locked){
8          log_error("[__wrap_malloc] someone is tring to use the heap
           ↪  after is was locked!");
9          ASSERT(false);
10     };
11     void *ptr = __real_malloc(size);
12     return ptr;
13 }
```

Listing 11: Lock Heap implementation for `malloc`

The same is done for the functions `cmalloc`, `realloc` and `free`, all of which access the heap. In the Vision Add-On's firmware, all heap usage is pre-allocated during the system initialization phase. After the system is up and running, `lock_heap` is called to prevent any further heap usage. This is especially useful since the C++ standard library is used, which heavily relies on dynamic heap usage. Using a locked heap exposes any of the standard libraries functions who might use the heap during development.

In the implementation shown in 11, a custom `ASSERT` macro is called. Its implementation is showed in 12. If `ASSERT` is called with an expression which evaluates to `false`, all interrupts are disabled, the scheduler stopped and a final log message which exposes the callee is sent out (only over SWO since the network is no longer running) before finally stopping the code execution using `abort`.

```
#define ASSERT(expr) \
    if (!(expr)) \
        assert_handler(__FILE__, __LINE__)

void assert_handler(const char *file, int line) {
    taskDISABLE_INTERRUPTS();
    vTaskSuspendAll();
    log_error("[ASSERT] assertion failed in file: %s, line: %d", file,
    ↪ line);
    abort();
}
```

Listing 12: Custom `ASSERT` implementation

Whenever a function receives a raw pointer, `ASSERT` is used to make sure it is not a null-pointer. As a result, uninitialized code can be caught early during development.

2.4 Software

2.4.1 Overview

Two main applications were developed. A GUI used to configure the Blob Detector and receive captures frames, and a script which triangulates the tree dimensional position of a detected marker in real time.

`<software>` denotes the root of the software project.

2.4.2 Toolchain

All software to interact with the Blob Detector is written in Python. Uv [52] is used as the package and project manager. It automatically installs all the required Python package and uses the correct Python version. Use `uv run <script>` to execute a Python script with uv. A VS Code project with useful extensions for Python development was set up. The project requires the following tools to be installed on the host PC.

- Python version 3.13 or higher [53]
- uv version 0.5.4 [54]

2.4.3 Command Handling

CommandPacket class

The CommandPacket class implements the command packet structure shown in 2.35. It offers getters and setters for the `request id`, `command id`, `complete`, `size` and `data` fields. The object can be converted to a byte representation matching the command packet structure by calling its `serialize` method. The `deserialize` method is used to reverse the process. Internally, the class uses Python's struct module [55] to do the to and from bytes conversion.

CommandSender class

The CommandSender sends command packets to the Vision Add-On's Command Handler. Listing 13 shows the `log_level` method, which implements the `log_set_level` command listed in 2.5. The method creates an instance of `CommandPacket`, populates its members and passes it to the `CommandSender`'s `_send` private method. The `_send` method converts the command packet to bytes using the `serialize` method and finally sends it to the Vision Add-On using a TCP socket (see 2.3.2).

The `CommandSender` class implements methods for all the commands listed in 2.5.

```
def log_level(
    self,
    level: LogLevel,
    request_id: int = 1,
    blocking: bool = True,
    timeout_s: int = 1,
) -> bool:
    c = CommandPacket(
        request_id=request_id,
        command_id=CommandIds.LOG_SET_LEVEL.value,
        data=bytearray([level.value]),
    )
    return self._send(c, blocking, timeout_s) is not None
```

Listing 13: `CommandSender` member `log_level`

2.4.4 Log Receiver

The `LogReceiver` class opens a UDP socket (see 2.3.2) and listens for incoming log messages. The received messages are pushed to a thread safe message queue.

2.4.5 Camera Frame Transfer

The `FrameReceiver` class implements the client side of the Vision Add-On's camera frame transfer. The sequence diagram in figure 2.37 shows how the `FrameReceiver` works. The method `receiveSnapshot` both triggers the capture of a frame on the Vision Add-On and its transfer from the Vision Add-On to the host PC. It triggers the frame capture using the `CommandSender`'s `capture` method, which implements the `camera_request_capture` command listed in 2.5. After the Vision Add-On successfully captures a single frame, the `FrameReceiver` launches a receiver thread, which opens a UDP socket (see 2.3.2) and listens for incoming image data. The Vision Add-On starts sending chunks of the image data once it received the camera_request_tranfer command listed in 2.5, which is implemented by the `CommandSender`'s `transfer` method. The method is called by the `FrameReceiver`. After triggering the transfer, the `FrameReceiver` awaits the successful completion of the camera_request_tranfer command and the termination of the receiver thread. The receiver thread writes any incoming data to a file until he received the expected amount of bytes. After that, the thread terminates itself, upon which the `FrameReceiver` converts the received binary data to a greyscale image. Finally, the `receiveSnapshot` method returns with the path to the received image. Python's asyncio [56] framework is used to implement the asynchronous thread and event handling of the `receiveSnapshot` method.

Asyncio uses cooperative multitasking. If a task never yields, all other tasks starve.

If used correctly, asyncio can be more performant than Python's threading [57] module since it does not rely on the operating system (OS) scheduler. It is especially suited for network bound application. Asyncio message queues are not thread safe, they are not suited for communication between OS scheduled threads.

The completed image is stored on the host PC's file system in the folder `/tmp/frameReceiver/snapshots`.

2.4.6 Blob Receiver

The `BlobReceiver` class implements the client side of the Vision Add-On's feature transfer. The class opens a UDP socket (see 2.3.2) and listens for incoming data. Incoming data has the structure shown in 2.21. The `BlobReceiver` parses the data and publishes the resulting bounding box coordinates together with the IP address of the sender to an asyncio message queue. When parsing the packet, the `BlobReceiver` class keeps track of the frame count field of the last packet sent by each device. If the frame index is incremented by more than one, the class outputs a log message warning about the missed frames. Next to publishing the coordinates to a message queue, the `BlobReceiver` also offers the option to log the timestamped data to a file.

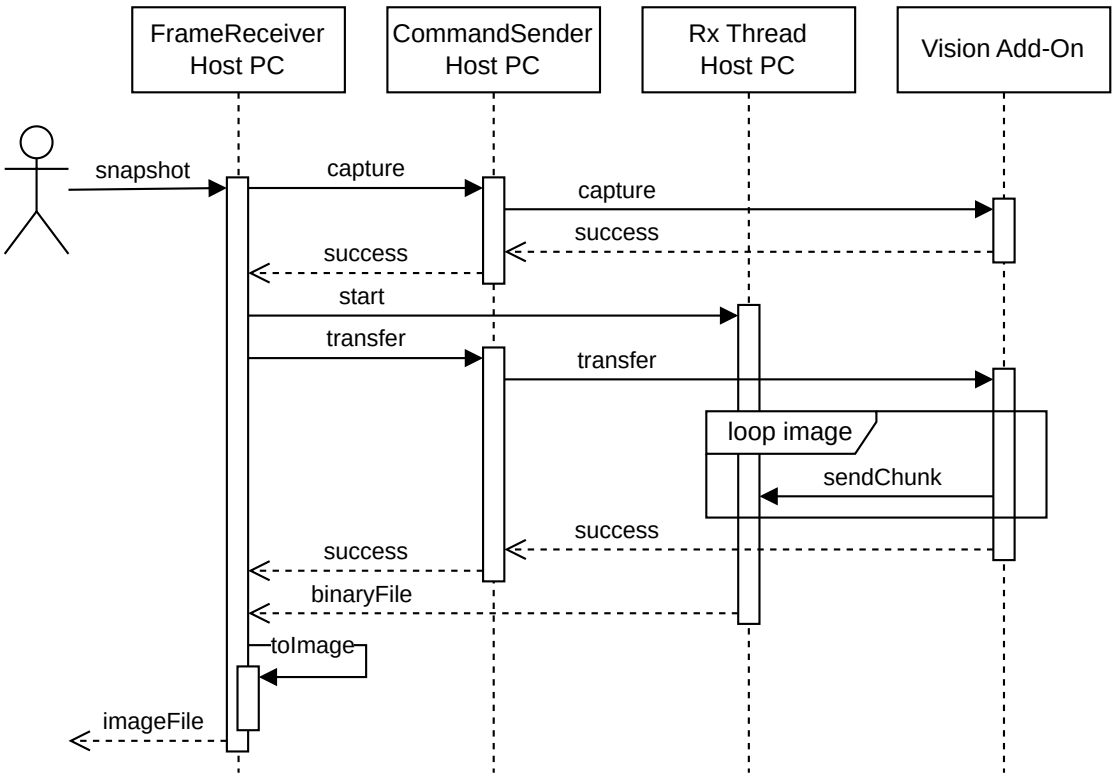


Figure 2.37: Sequence diagram showing how the `FrameReceiver` receives a captured image from the Vision Add-On

> The data published to the message queue has the following structure: `Tuple[IP Address, List[Tuple[xmin, xmax, ymin, ymax]]]`.

The `BlobReceiver` heavily relies on the asyncio framework. Reception is started using the asynchronous (Python `async` handle) `server` function. The `serve` function needs to be called from an asyncio asynchronous context. For instances where this is not possible or cumbersome, or a thread safe message queue is needed, the `BlobReceiverThreadWrapper` offers a wrapper, which uses Python's threading module. The wrapper is launched using its `serve` method. An OS thread is created which then runs the asyncio native variant of the `BlobReceiver` class. The thread can be stopped using the wrapper's `stop` method.

A stand-alone Blob Receiver service which prints the message queue contents to the command line was implemented. Listing 14 shows its usage.

> To record the 2D marker positions detected by any device, launch the blobReceiver.py script with `uv run blobReceiver.py -r boundingBoxes.log`.

```
Usage: blobReceiver.py [OPTIONS]

Options:
  -m, --mode [NATIVE|WRAPPER]  Select between asyncio native or thread
                               wrapper execution
  -r, --record-to PATH         Optional path to a log file which records
                               all the received coordinates
  --help                       Show this message and exit.
```

Listing 14: blobReceiver.py usage

2.4.7 Commander GUI

The Commander GUI to configure the Blob Detector and receive captures frames was developed. The GUI framework used is Dear PyGui. Dear PyGui is a Python wrapper around Dear ImGui. All the functionality of the `CommandSender`, `LogReceiver`, `BlobReceiver` and `FrameReceiver` classes can be accessed through the Commander GUI. The Dear PyGui framework is blocking and thus would starve the asyncio native implementation of the Blob Receiver. Instead, the `BlobReceiverThreadWrapper` class is used.

> Launch the commander GUI with `uv run commander.py`.

> The Commander GUI can draw the real-time bounding boxes on top of a captured frame. This is useful when tuning camera and pipeline parameters.

2.4.8 Camera Calibration

The goal of the camera calibration is to describe how a camera projects a 3D scene into a 2D image. OpenCV [58] is an open-source, cross-platform computer vision library. It offers several functions which can be used to calibrate a camera. To describe a camera, OpenCV uses the following parameters:

The distortion coefficients 2.8 that are applied to a distortion model [59], which models how the camera lens distorts an image.

$$distortion\ coefficients = (k_1, k_2, p_1, p_3, k_3) \tag{2.8}$$

The intrinsic camera matrix 2.9, which describes the camera focal point f_x, f_y and principal point c_x, x_y.

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

The Python package opencv-python offers Python bindings to OpenCV. It is used by the `CameraCalibration` class. The class is based on OpenCV's camera calibration tutorial [60]. The inputs of the calibration routine are multiple images of the same chessboard captured at different angles with the camera that is to be calibrated. Additionally, the chessboard geometry is specified. Since the chessboard has a known geometry, the camera matrix and distortion coefficients can be calculated. The parameters are computed by the `calibrateCamera` function. Its inputs are the set of images, 3D object points and their 2D projections on the camera sensor. The 3D object points are 3D coordinates of the corners of the chessboard. The position and orientation of the board in space does not matter since the board serves as the reference frame of the coordinate. On the captured images, the same points are detected using the `findChessboardCorners` and `cornerSubPix` OpenCV functions. The resulting 2D coordinates (camera sensor is the reference frame of the coordinate system) are the 2D projections of the 3D chessboard points. Next to the camera matrix and distortion coefficients, the `calibrateCamera` function also returns the rotation and translation vectors, which describe the camera's orientation in space relative to the chessboard (one set of vectors per image). They are used to compute the reprojection error, a metric quantifying how good the camera calibration is. It is based on the distance between a 3D point projected to the camera plane using the computed model, and its measured position. The rotation and translation vectors are also referred to as the extrinsic camera parameters. All computed parameters are stored in a JSON file together with some metadata.

The result of the camera calibration are the intrinsic camera matrix and the distortion coefficients (left side of figure 2.38).

The camera calibration functionality is integrated in the commander GUI under the Calibrate Optics tab. It also implements the calibration_store_camera_matrix and calibration_store_distortion_coefficients commands listed in 2.5. These commands can be used to persist the camera calibration data to the Vision Add-On.

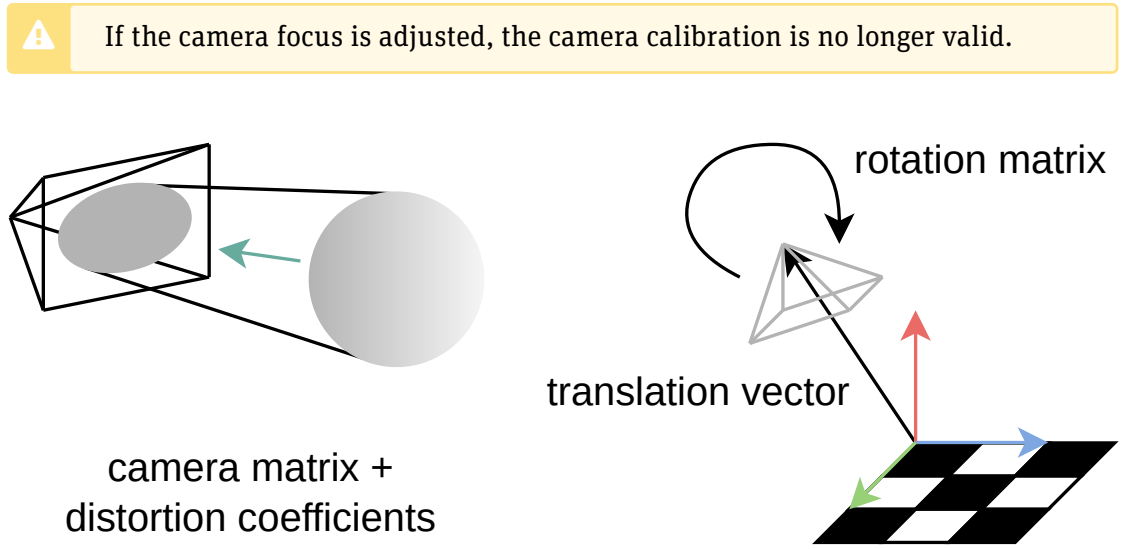


Figure 2.38: Projection into camera (left) and camera position relative to chessboard (right)

2.4.9 Position Calibration

Position calibration is used to determine the position and orientation of all cameras relative to a common coordinate system. The `PositionCalibration` class is based on OpenCV's pose estimation tutorial [61]. Once again, a chessboard is used. It serves as the common coordinate reference frame for all cameras. Using the OpenCV function `solvePnP`, a camera's rotation and translation vectors relative to the chessboard are computed. This is done for each camera. The function's inputs are a single image of the chessboard, the camera matrix and the distortion coefficients computed in 2.4.8. The chessboard needs to remain in the same position for all images taken by the cameras whose position is to be calibrated. The rotation vector computed by `solvePnP` is converted to a rotation matrix using the `Rodrigues` function. The translation vector defines the offset of the camera to the origin. The rotation matrix defines the orientation of the camera. Again, all the computed parameters are stored in a JSON file together with some metadata.

The result of the position calibration are the translation vector and rotation matrix (right side of figure 2.38).

The position calibration functionality is integrated in the commander GUI under the Calibrate Position tab. It also implements the calibration_store_rotation_matrix and calibra-

tion_store_translaton_vector commands listed in 2.5. These commands can be used to persist the position calibration data to the Vision Add-On.

If the camera is moved, the position calibration is no longer valid.

2.4.10 Triangulation

2.4.11 Overview

The `Triangulate` class receives the bounding boxes of a detected marker and triangulates its position in 3D space.

The sequence diagram shown in figure 2.39 shows how the `Triangulate` class interacts with the `Command Sender` and `Blob Receiver` classes. For each Blob Detector, the calibration data is fetched from the Vision Add-On's EEPROM using the `Command Sender` class. Detected marker coordinates published by the `Blob Receiver` class are used to triangulate the 3D position. The triangulated position is published to both the render message queue and the record message queue.

2.4.12 Undistort points

The `Triangulate` class awaits incoming coordinates from the `Blob Receiver`'s message queue. Upon reception, the centre of each transmitted bounding box is computed. This point represents the 2D position of the detected marker. Using OpenCV's `undistortPoints` function, the lens distortion is compensated. The function takes image points, the camera matrix and the distortion coefficients as its input. The function's output is the image points corrected for the camera's lens distortion. In the case of the `Triangulate` class, the image points are the centres of the Bounding Boxes. The class stores the undistorted coordinates in a map, with the Blob Receiver IP address as the key, referred to as the IP to coordinate map. The Blob Detector transmits a packet (see 2.21) for each frame processed. Even if no marker was detected, a packet (containing no markers) is still sent out and received by the `Blob Receiver` class. In such a case, the `Triangulate` class invalidates the previous coordinate entry from the IP to coordinates map. This destinction is depicted in the *alt* block in figure 2.39.

2.4.13 Midpoint triangulate

The `midpoint_triangulate` method of the `Triangulate` class is based on the code snippet in the answer by user *Nirmal* on the Stack Overflow forum post *Generalisation of the "mid-point" method for triangulation to n points* [62]. The function implements the method described in section 3.3 of [63]. From the origin of each camera, a ray is projected through the detected marker as shown in figure 2.40. The distance to the marker is unknown.

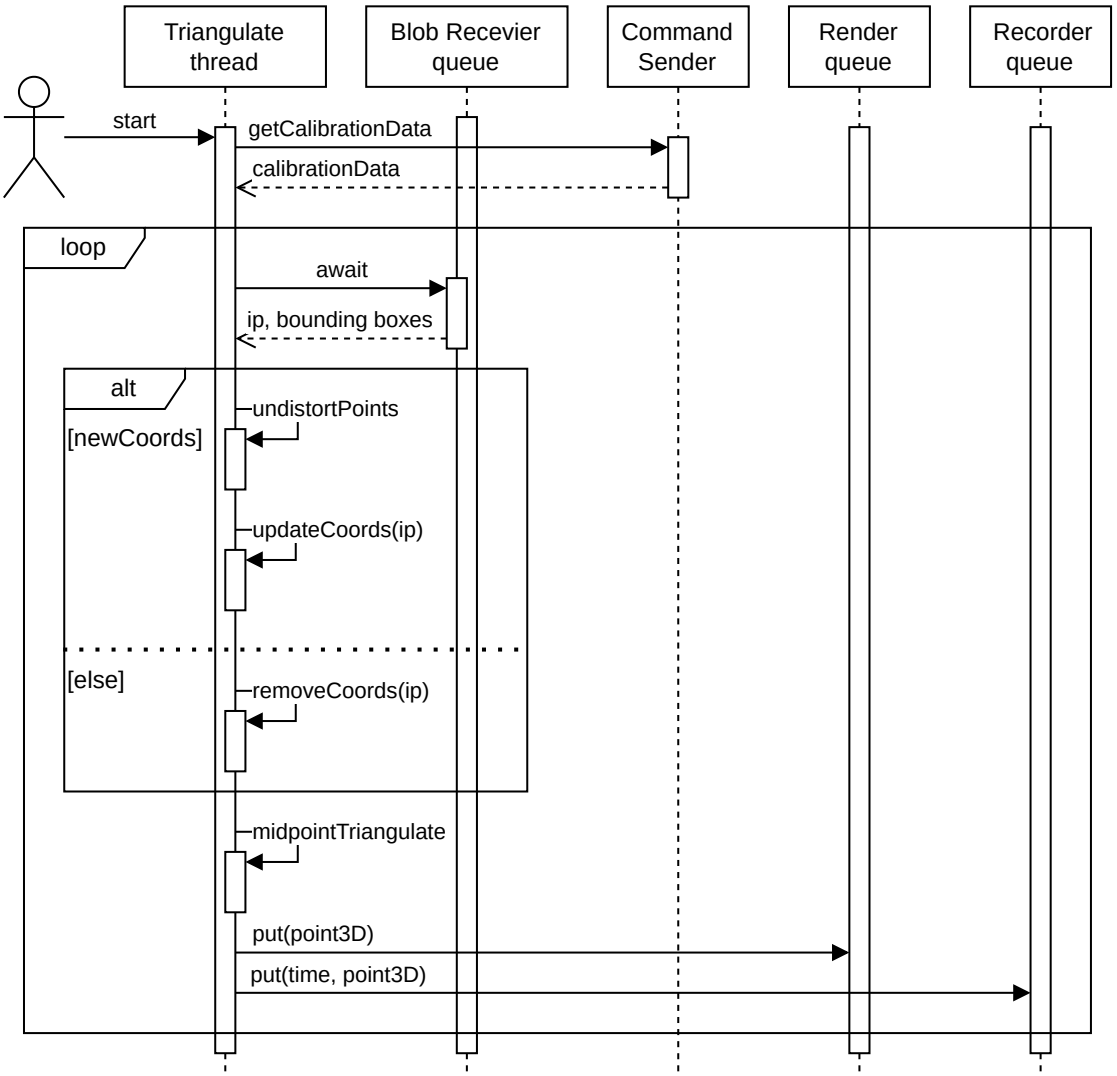


Figure 2.39: Sequence diagram showing `Triangulate` class interactions

Ideally, all rays should intersect at the true 3D point. In reality, they do not. Instead, the closest point to all rays is calculated by minimizing the sum of the squared distances. This is referred to as the midpoint triangulation method. It is the geometric solution to the problem. All vectors are weighted equally.

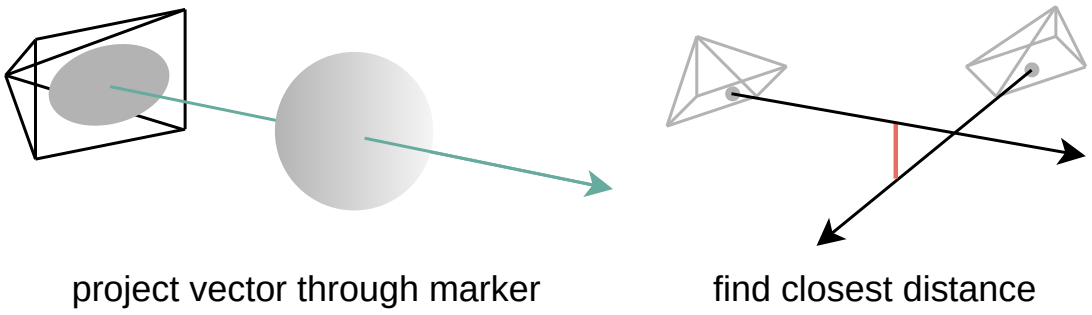


Figure 2.40: Projecting vector through detected marker (left) and finding vector intersection (right)

The function takes n 2D image points taken from n cameras. For each camera, the calibration data computed in 2.4.8 and 2.4.9 is required by the function. The output of the function is the 3D point closest to all projected rays.

The `Triangulate` class performs triangulation whenever it receives a new packet from any Blob Detector. Triangulation is performed using all coordinates stored in the IP to coordinates map.

The midpoint triangulate method only works for a single image point. If multiple markers are observed per image, multiple image points per camera are input to the function. The output is the closest 3D point to all observed markers. The image point would have to be grouped to the corresponding marker (called solving marker correspondence) before calling `midpoint_triangulate` for each group. This was not done in this thesis.

The `Triangulate` class only works if a single marker is in view of the cameras.

Rendering and recording

The `Triangulate` class publishes the triangulated 3D marker position to a render queue. The Python package VisPy [64] is used to render the detected parker in real-time in a 3D scene. VisPy's backend is OpenGL. As a result, the render code of the `Triangulate` class is GPU accelerated.

Additionally, the triangulated 3D marker position can also be published to a message queue whose contents are logged to a file.

To triangulate the marker position detected by the devices 10.0.0.10, 10.0.0.11 and 10.0.0.12, launch the triangulate.py script with `uv run triangulate.py 10.0.0.10 10.0.0.11 10.0.0.12`.

```
Usage: triangulate.py [OPTIONS] IP...

  Triangulate the marker position

  IP: one or multiple device IPs

Options:
  --record-2d PATH             Optional path to a log file which records
                               all the received 2d marker coordinates
  --record-3d PATH             Optional path to a log file which records
                               all the triangulated 3d marker
                               ↪ coordinates
  -p, --play-from PATH         Optional path to a log file to play back
                               recorded 2d marker coordinates
  -c, --calibration-data PATH  Optional path to a json file containing
                               camera calibration data
  --help                       Show this message and exit.
```

Listing 15: triangulate.py usage

2.4.14 Other Supporting Tools

configSender.py

The script configSender.py can apply a configuration, read from a JSON file, to the Blob Detector. Listing 16 shows an example config file with setting to apply to the Blob Receivers with the IP address 10.0.0.10 and 10.0.0.11. The functionality of the configSender.py script is integrated in the commander GUI under the Configuration File tab.

blobPlotter.py

The script blobPlotter.py creates an instance of the `BlobReceiver` class and plots all incoming coordinates (the centre of the bounding boxes) in real time using pygame [65].

plotTrinagulatedData.py

The script plotTrinagulatedData.py plots the recorded 3D data, output by the `Triangulate` class in a 3D scatter plot, from different viewing angles.

```
{
    "10.0.0.10": {
        "fps": 72,
        "exposure": 90,
        "gain": 31,
        "threshold": 30,
        "light": true
    },
    "10.0.0.11": {
        "fps": 13,
        "exposure": 200,
        "gain": 31,
        "threshold": 100,
        "light": false
    },
}
```

Listing 16: Example JSON configuration file which can be applied using configSender.py

Use the file created when running triangulate.py with the `--record-3d` option as the input for the plotTrinagulatedData.py script.

fetchCalibrationData.py

The script plotTrinagulatedData.py downloads the camera and position calibration data from the specified Blob Detectors and stores it to a JSON file. This functionality combined with the recording capability of the `BlobReceiver` class can be used to re-run the triangulate.py script with recorded data.

Use the file created by fetchCalibrationData.py as the argument passed to the `--calibration-data` option of the triangulate.py script.

numpyArrayEncoder.py

The script numpyArrayEncoder.py contains a custom JSON encoder for handling NumPy data types, that are not natively serializable by Python's built-in JSON module. This is used whenever calibration data is stored to a JSON file.

3 Results and Discussion

3.1 FPGA utilization

Table 3.1 shows the resource usage of the Lattice LFE5U-85F-6BG381C for the design presented in this project. It was generated by nextpnr-ecp5 using the `--report` argument. Less than 13% of the available LUTs (`TRELLIS_COMB`) are used. Flip-flop (`TRELLIS_FF`) usage is even lower at less than 8%.

3.2 Verilog test bench results

3.2.1 Double Buffer

Listings 17, 18 and 19 all show different sections of the test bench used to verify the Double Buffer implementation. The task shown in listing 17 was used to write a test pattern to the Double Buffer. With each cycle of `writeClock`, a new value is written to the Double Buffer. Multiple such tasks were defined, each with a different pattern, some with fewer data than fits into the buffer, others with a toggling `writeEnable` signal. Listing 18 shows the logic used to swap the buffers and wait until the newly available data is read by the consumer. Listing 19 shows the consumer's read logic. As long as `readAddress` is less than `readAddressMax`, `readAddress` is incremented with each cycle of `readClock`. Finally, the actual test sequence is simply altering between calls to the different `writePattern` tasks and the `swapBlocking` task. Note that none of the test benches use assert statements, verification has to be done by manually analysing the simulation results.

To execute the Double Buffer's test bench and load the GTKWave project to render the simulation output, run `make vvp && gtkwave doubleBuffer.gtkw >/dev/null 2>&1 &` in the directory `<gecko5>/modules/doubleBuffer/test`.

Figure 3.1 shows a segment of the Double Buffer's test bench simulation result. Data is written to the buffer by the `writePattern` task shown in listing 17. The signal `writeData` shows the data being written to the buffer. As soon as the last element is being written, the signal `oneWriteLeft` is set to high (label 2 in 3.1), in the next cycle, the `full` signal is set to high (label 1 in 3.1). Next, the task `swapBlocking` shown in listing 18 is called. As a result, the signal `switchBuffer` is set to high for one clock cycle of the producer (label 4 in 3.1). After a delay caused by the Synchrous Flop (see chapter 2.2.6) and other buffer

resource	available	used	utilization
ALU54B	78	0	0.00%
CLKDIVF	4	0	0.00%
DCCA	56	6	10.71%
DCSC	2	0	0.00%
DCUA	2	0	0.00%
DDRDLL	4	0	0.00%
DLLDELD	8	0	0.00%
DP16KD	208	27	12.98%
DQSBUFM	14	0	0.00%
DTR	1	0	0.00%
ECLKBRIDGECS	2	0	0.00%
ECLKSYNCB	10	0	0.00%
EHXPLLL	4	1	25.00%
EXTREFB	2	0	0.00%
GSR	1	0	0.00%
IOLOGIC	224	0	0.00%
JTAGG	1	0	0.00%
MULT18X18D	156	5	3.21%
OSCG	1	0	0.00%
PCSCLKDIV	2	0	0.00%
SEDGA	1	0	0.00%
SIOLOGIC	141	0	0.00%
TRELLIS_COMB	83640	10703	12.80%
TRELLIS_ECLKBUF	8	0	0.00%
TRELLIS_FF	83640	6465	7.73%
TRELLIS_IO	365	101	27.67%
TRELLIS_RAMW	10455	164	1.57%
USRMCLK	1	0	0.00%

Table 3.1: Utilization report

```
task writePattern;
  begin
    @(posedge writeClock);
    #0 writeData = 4'h8;
    #0 writeEnable = 1;
    @(posedge writeClock);
    #0 writeData = 4'h9;
    @(posedge writeClock);
    #0 writeData = 4'ha;
    @(posedge writeClock);
    #0 writeEnable = 0;
  end
endtask
```

Listing 17: Double Buffer test bench write task

```
task swapBlocking;
  begin
    @(posedge writeClock);
    #0 switchBuffer = 1;
    @(posedge writeClock);
    #0 switchBuffer = 0;
    wait(readAddress == readAddressMax) // wait for read to complete
    @(negedge writeClock);
  end
endtask
```

Listing 18: Double Buffer test bench swap task

```
reg [ADDRESS_WIDTH-1:0] maxAddr = 0;
always begin
  @(posedge newData);
  maxAddr = readAddressMax;
  @(posedge readClock);
  readAddress = 0;
  @(negedge readClock);
  while (readAddress < maxAddr) begin
    @(posedge readClock);
    readAddress = readAddress + 1;
    @(negedge readClock);
  end
end
```

Listing 19: Double Buffer test bench read process

management logic, the signal `newData` is set to high (label 5 in 3.1). With the positive edge of `newData`, the read process shown in listing 19 starts to read the data from the buffer. The signal `dataOut` shows the data being read from the buffer (label 6 in 3.1). One can observe that the written data (label 3 in 3.1) matches the data being read.

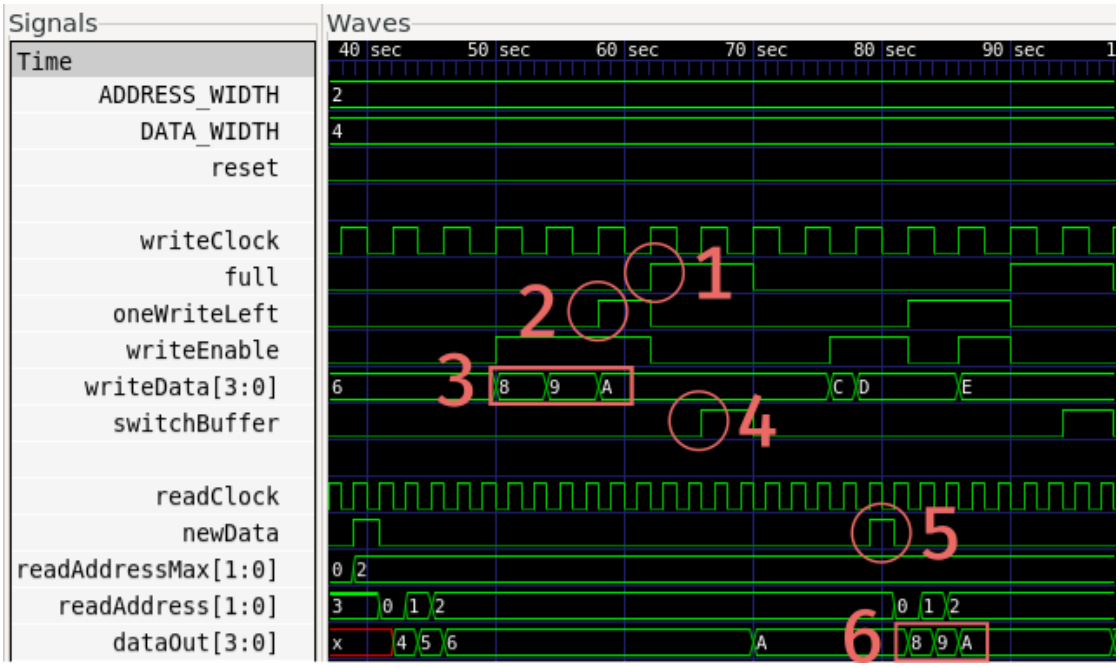


Figure 3.1: Double Buffer simulation results

1: buffer is full
2: one write left
3: data written to buffer
4: trigger buffer switch
5: buffer switch complete
6: data read from buffer

3.2.2 Feature Transfer

Just like the Double Buffer, the Feature Transfer module was tested using a test bench. Listing 20 shows a selection of the tasks that were declared within the test bench. The `writeFeature` task pulses the `featureValid` signal for one clock cycle. The `sync` task does the same for the `vSync` signal, which initiates a buffer swap. The `waitForTransfer` task only returns once the SPI transfer is complete. Listing 21 shows these tasks in use to feature send vectors to the Feature Transfer module. This mimics the output of the CCA module.

Figure 3.2 shows the simulation results of the Feature Transfer testbench. The signal `paddedFeatureVector` is the data at the output of the Double Buffer module. Labels 2 and 3 in figure 3.2 show the feature vectors which are seen in the test sequence in listing 21. Labels 5 and 6 in figure 3.2 show the first feature vector being received over SPI. La-

bel 4 in figure 3.2 shows the number of features. This follows the feature transfer packet structure shown in figure 2.21. Label 1 in figure 3.2 shows the SPI transfer done pulse, which the `waitForTransfer` task shown in listing 20 waits for, to synchronize the flow of the test sequence.

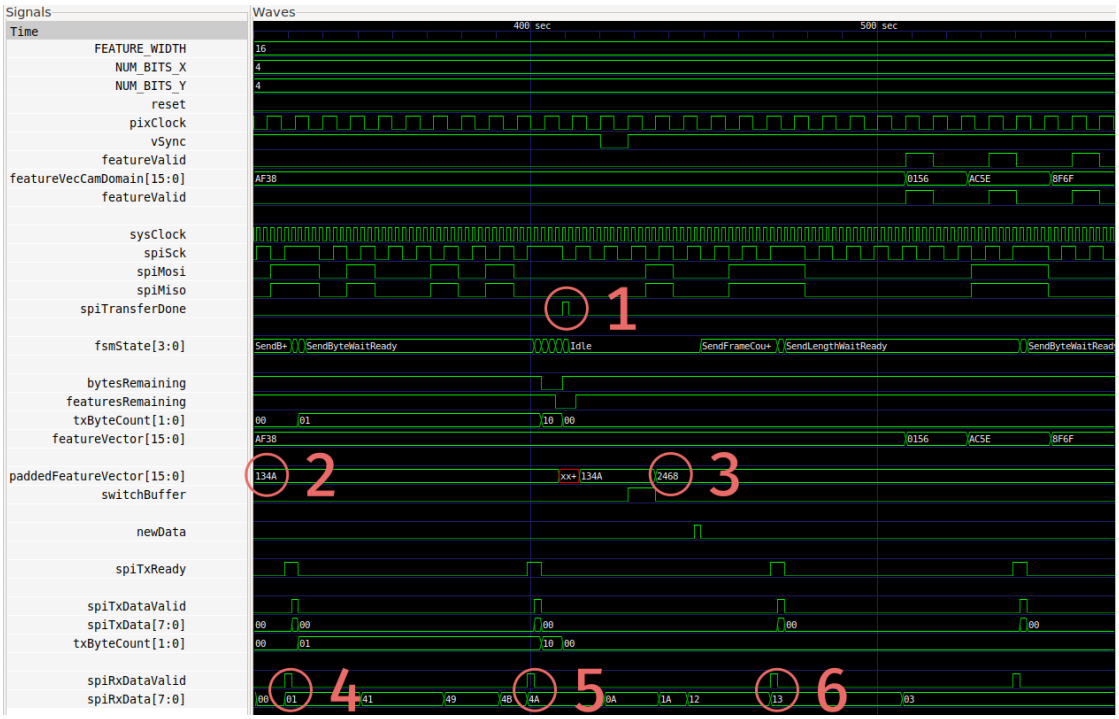


Figure 3.2: Feature Transfer simulation results

1: SPI transfer done pulse
2: previous feature vector
3: new feature vector
4: number of features
5: 1st byte of feature vector
6: 2nd byte of feature vector

3.3 Triangulate using Blender

To not rely on the Blob Detector whilst developing the triangulation method, a virtual scene was created in the 3D modelling software Blender [66]. Figure 3.3 shows a screenshot of the Blender scene. A chessboard for camera calibration and a single marker is placed in the scene. The scene is rendered from different perspectives. The resulting images are used for both the camera calibration and triangulation. Using a Python script, the marker's pixel coordinates are extracted. These coordinate serve as the input to the triangulate function.

Figure 3.4a shows the Blender scene rendered from one camera view. The resolution of all rendered images is 1920x1080 pixels. In the Blender test scene, the furthest camera is placed at a distance of 11 m, most cameras are within 8 m or closer. The origin of the

```
task writeFeature;
  begin
    @(posedge pixClock) #0 featureValid = 1;
    @(posedge pixClock) #0 featureValid = 0;
  end
endtask

task sync;
  begin
    @(posedge pixClock) #0 vSync = 0;
    @(posedge pixClock) #0 vSync = 1;
  end
endtask

task waitForTransfer;
  begin
    @(posedge spiTransferDone);
  end
endtask
```

Listing 20: Feature Transfer test bench tasks

```

#10 featureVecCamDomain = 'h134A;
writeFeature();
waitForTransfer();
#10 sync();

#80 featureVecCamDomain = 'h2468;
writeFeature();
#10 featureVecCamDomain = 'hAF38;
writeFeature();
waitForTransfer();
#10 sync();
```

Listing 21: Feature Transfer test sequence

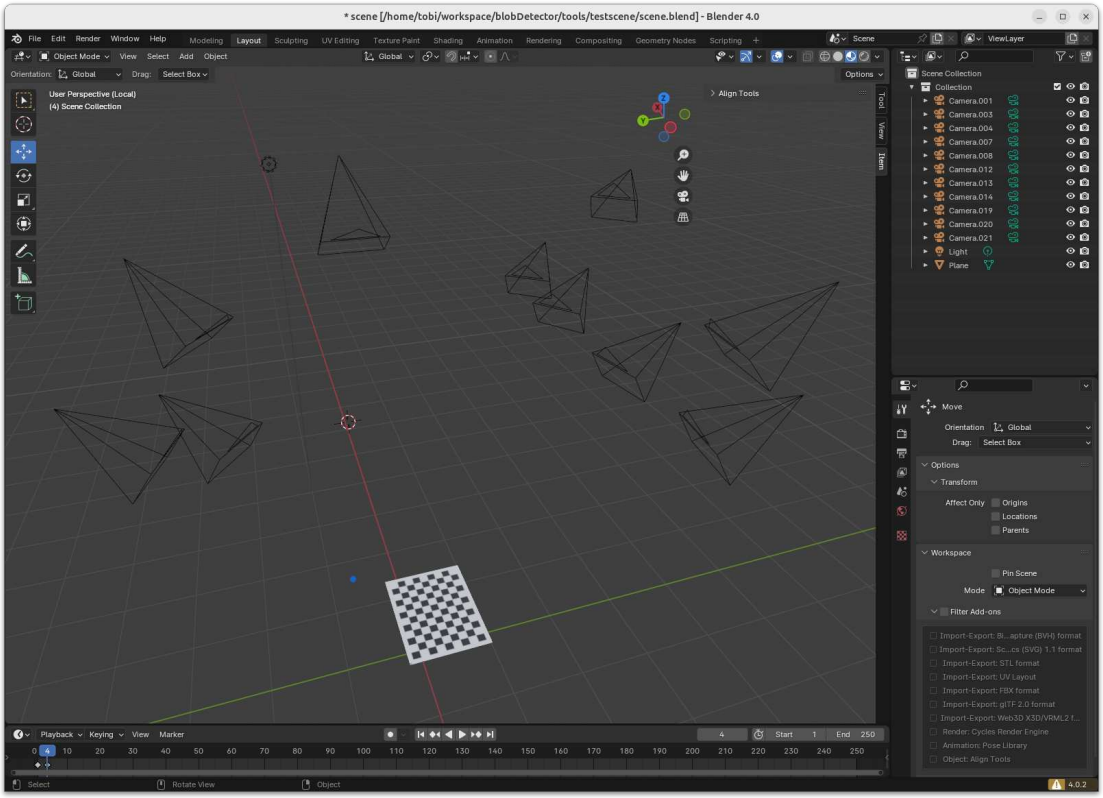


Figure 3.3: Triangulation test scene created in blender

coordinate system is visualized on the chessboard. Figure 3.4a shows a ray being projected from the camera (x) through marker (o). The marker's 3D position within the Blender scene is used to draw it on the plot. These coordinates are not used to project the vector, only its 2D projection onto the camera is used. The distance from the camera to the 3D marker is not known at this stage, only its direction. Figure 3.4c shows all rays projected from all cameras. The closest point to all rays is computed using the midpoint triangulate method described in 2.4.13.

The marker coordinates ((x, y, z) offset from origin) within the Blender scene are (0.12, 0.8, 1.5) m. The triangulated position is (0.119895, 0.800638, 1.50066) m. The distance between the two positions is 0.92409 mm.

This is a greater error than expected. Commercial solutions claim accuracies of 0.10 mm at a room scale tracking setup when running at an image resolution of 4096x3072 pixels [67].

To even the odds, the test was rerun with a render resolution of 4096x3072 pixels. The triangulated marker position is (0.119997, 0.800036, 1.500043) m. The distance between the exact and the computed marker positions is 0.05637 mm.

Higher image resolution clearly improves the triangulation accuracy.

3.4 Vision Add-On PCB

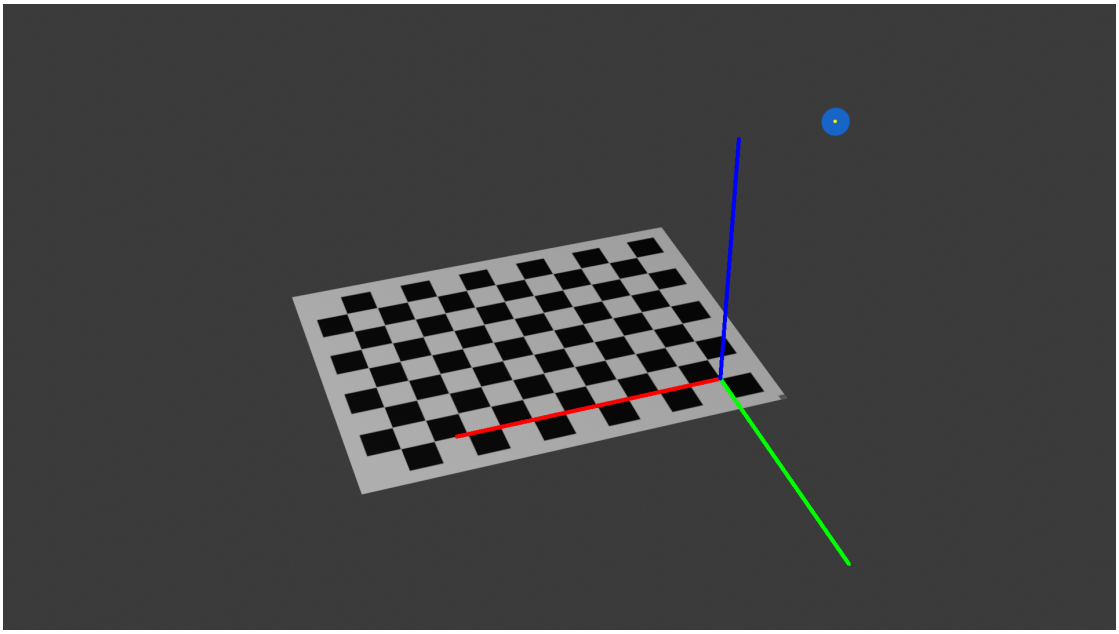
Figure 3.5 shows an assembled Vision Add-On board To make the communication between the microcontroller and the Ethernet PHY (LAN87) work, a minor hardware patch was required.

Pin 2 of the LAN87 needs 10kOhm pull-down resistor and the 10kOhm pull-down on Pin 3 needs to be removed.

The additional pull-down resistor can be seen at the pins of the RJ45 connector in 3.5.

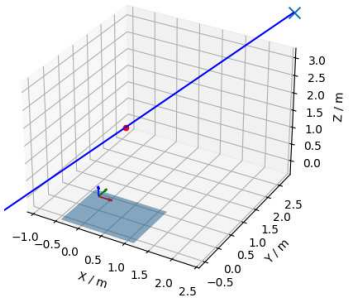
On some Blob Detector, the USB connection between the Vision Add-On and the GECKO5Education did not work. This was due to the mezzanine connector not being soldered on properly. All units which exhibited the behaviour were fixed except the unit labelled 13. There, the root cause of the issue was not discovered. Replacing the mezzanine connector did not help. As a result, the GECKO5Education's USB connection is not forwarded to the Vision Add-On's USB port.

The Blob Detector unit 13 has a faulty USB connection. The GECKO5Education needs to be disconnected from the Vision Add-On in order to upload synthesized HDL or to access the soft-core's command handler over USB.

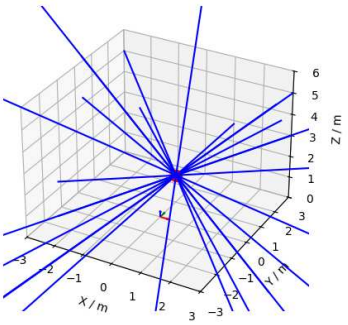


(a) Blender scene rendered from one camera view

The coordinate system axis and the yellow dot (detected marker position) the are overlaid in Python



(b) Ray projected from camera origin (x) through marker (o)



(c) All rays intersecting

Figure 3.4: Blender scene triangulate results

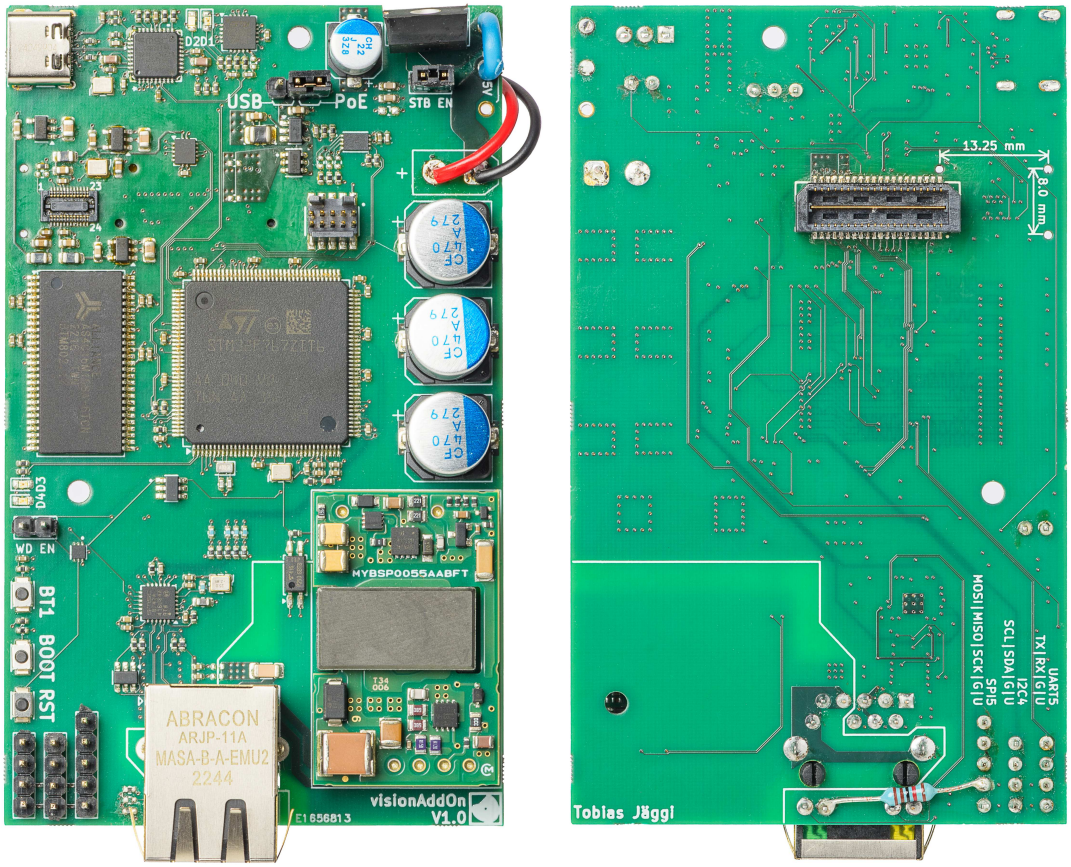


Figure 3.5: Vision Add-On front and back view

3.5 OV9281 camera logic level

The logic level of the OV9281 is 1.2V. The logic level of the GECKO5Education's GPIO bank connected to the mezzanine connector (GPIO bank U6G) is configured to use the same logic level as the add-on board, which connects to the mezzanine connector. To achieve this, pin 39 of the mezzanine connector, shown in figure 3.6, is connected to the VCCIO7 pin of GPIO bank U6G. The voltage, supplied by the add-on board an thus applied to this pin, determines the bank's logic level. If no add-on board is connect, the GPIO bank's logic level is 1.2V.

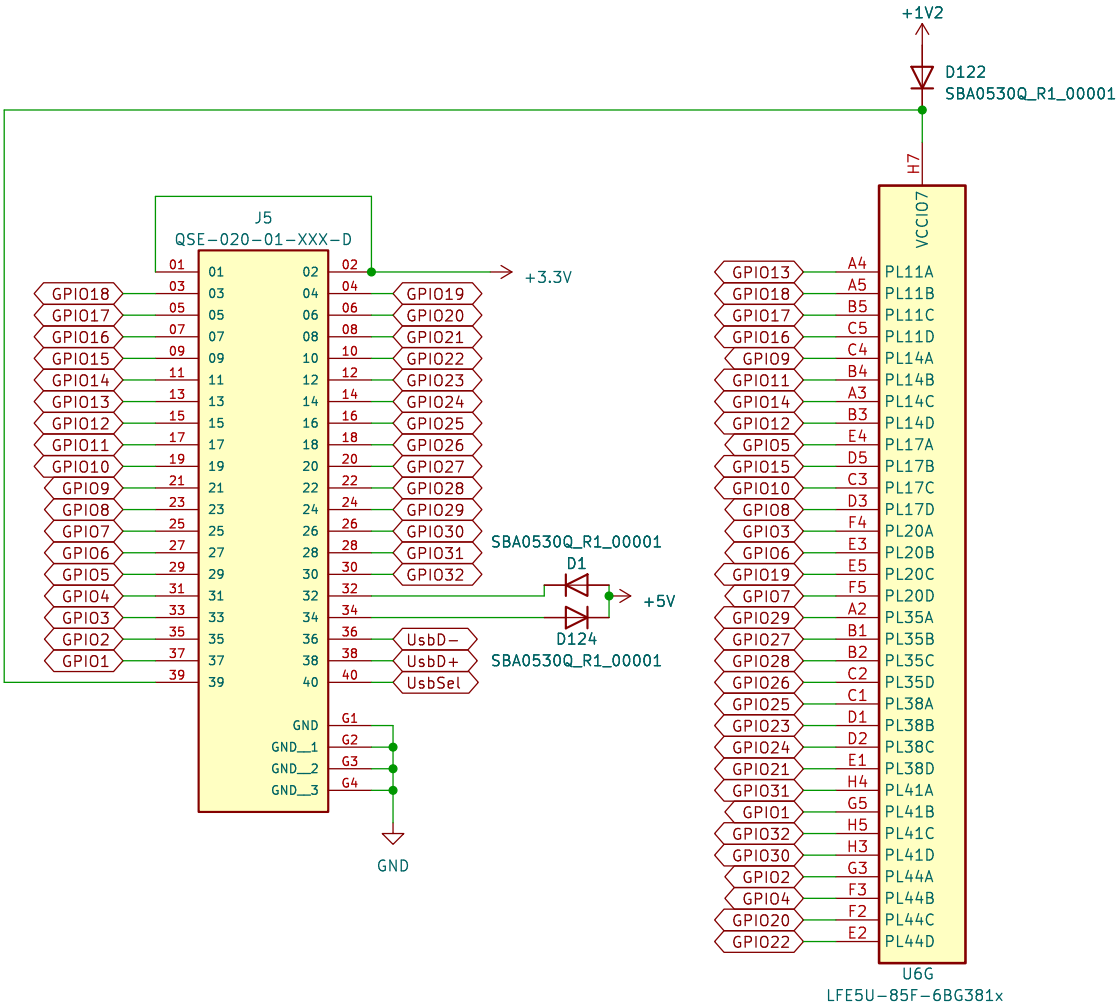


Figure 3.6: GECKO5Education mezzanine connector schematics

The microcontroller on the Vision Add-On board uses 3.3V logic level, pin 39 of the mezzanine connector is supplied with 3.3V. As a result, the GPIO bank runs at 3.3V logic level, opposed to the 1.2V of the OV9281 camera sensor. This was an oversight during the hardware design phase and only discovered after the Vision Add-On was produced. The Lattice

LFE5U-85F-6BG381C supports mixed voltage levels on a single bank. Not all logic level are supported as shown in figure 3.7 taken from [68]. Luckily, a GPIO bank with 3.3V supplied to its VCCIO pin can still support 1.2V by setting the pin type to type LVCMOS12. By changing the pin type from LVCMOS33 to LVCMOS12 for all GPIO pins which receive the OV9281 camera feed, the GECKO5Education works properly with the Vision Add-On.

For Top/Bottom Banks										
V_{CCIO} (V)	Input sysI/O Standards					Output sysI/O Standards				
	1.2 V	1.5 V	1.8 V	2.5 V	3.3 V	1.2 V	1.5 V	1.8 V	2.5 V	3.3 V
1.2 V	Yes	—	—	Yes	Yes	Yes	—	—	—	—
1.35 V	Yes	—	—	Yes	Yes	—	—	—	—	—
1.5 V	Yes	Yes	—	Yes	Yes	—	Yes	—	—	—
1.8 V	Yes	—	Yes	Yes	Yes	—	—	Yes	—	—
2.5 V	Yes	—	—	Yes	Yes	—	—	—	Yes	—
3.3 V	Yes	—	—	Yes	Yes	—	—	—	—	Yes
For Left/Right Banks										
V_{CCIO} (V)	Input Signal					Output sysI/O Standards				
	1.2 V	1.5 V	1.8 V	2.5 V	3.3 V	1.2 V	1.5 V	1.8 V	2.5 V	3.3 V
1.2 V	Yes	—	—	—	—	Yes	—	—	—	—
1.35 V	Yes	—	—	—	—	—	—	—	—	—
1.5 V	Yes	Yes	—	—	—	—	Yes	—	—	—
1.8 V	Yes	—	Yes	—	—	—	—	Yes	—	—
2.5 V	Yes	—	—	Yes	—	—	—	—	Yes	—
3.3 V	Yes	—	—	Yes	Yes	—	—	—	—	Yes

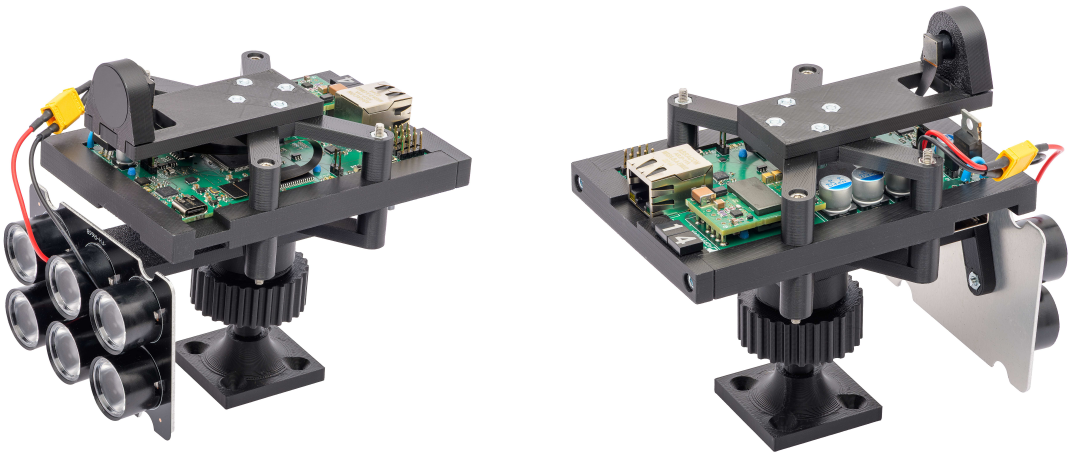
Figure 3.7: Lattice LFE5U-85F-6BG381C mixed voltage I/O support [68]

Before realizing that the Lattice LFE5U-85F-6BG381C has support for different logic levels on the same GPIO bank, the Waveshare OV5640 camera [69] was used as a fallback camera. This was accounted for in the risk analysis [47] which was done for the Vision Add-On. The fallback camera plugs directly into the GECKO5Education (2 in 2.1). Its resolution is 640x480 pixels with a frame rate of 90 fps. The camera's output format is the same as the OV7670 camera, which was used in [2]. This meant that there was minimal effort required to make the image processing pipeline work with the fallback camera. Development was able to continue uninterrupted using the fallback camera. Once the OV9281 camera was running, the image processing design was updated to work with the new camera. The pipeline no longer supports the OV5640 camera.

3.6 Blob Detector assembly

Figure 3.8 shows the final assembly of the Blob Detector. Each unit has an ID assigned to it and a network configuration was created based on that ID. The ID is the last octet of the IP address 10.0.0.<ID> as well as the least significant byte of the MAC address 00:80:e1:00:00:<ID>. The unit shown in figure 3.8b has the ID 14, so its IP address is 10.0.0.14 and its MAC address is 00:80:e1:00:00:14.

The optics of the LED module were later removed as they focus the light too much. A more evenly spread light source proved to be better suited. When using the Strobe Control module to pulse the LED's supply voltage, the LEDs were unable to reach their peak brightness in less than one frame period. Instead of flashing in sync with the camera refresh rate, the LEDs supply voltage was configured to be a constant 5 V for all tests. This maximized the LED's light output. The 25.5 W PoE module can easily sustain the 4.5W (900mA at 5V) power draw of the LEDs. However, at constant power, the aluminium heat sink gets warm enough to soften the PLA mount. The warping was small enough to not affect any tests. It is advised to print future flash mounts in a more heat-resistant material other than PLA.



(a) Blob Detector front view

(b) Blob Detector back view

Figure 3.8: Blob Detector fully assembled

3.7 Image processing pipeline

3.7.1 Camera

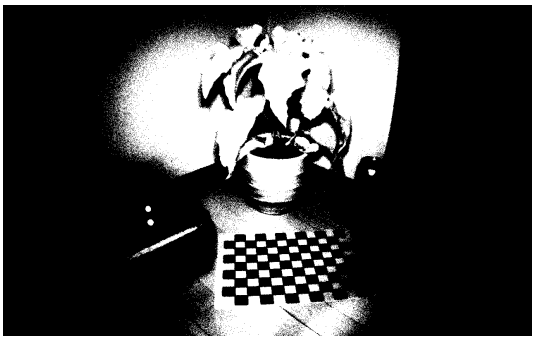
The collection of camera frames in figure 3.9 shows the same scene captured with (3.9b) and without (3.9a) the IR LED and using different binarization thresholds (3.9c and 3.9d). All frames were captured using the Vision Add-On's Frame Transfer service.

3.7.2 CCA

Using the Fake Camera module, the CCA was tested. Figure 3.10 shows the test pattern output by the Fake Camera module, captured using the Commander GUI. The *show markers* option (label6 in 3.11) is enabled to plot the received bounding boxes on top of the test pattern. The CCA module functions as expected, all bounding boxes are correctly determined.

(a) Unprocessed camera frame taken with led IR LEDs off

(b) Unprocessed camera frame taken with led IR LEDs on



(c) Binary (threshold 100) camera frame taken with led IR LEDs on

(d) Binary (threshold 200) camera frame taken with led IR LEDs on

Figure 3.9: Camera and image processing pipeline results

The optical filter was installed for all images

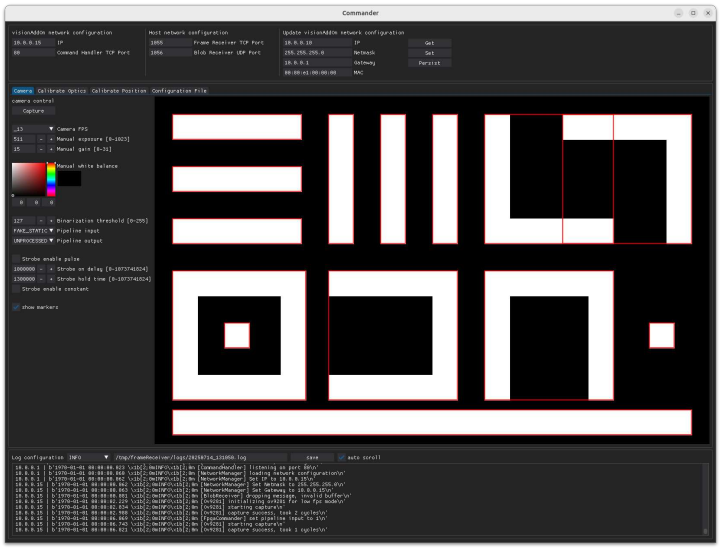


Figure 3.10: Fake Static camera frame with bounding boxes overlaid by the Commander GUI

3.8 Commander GUI

3.8.1 Main view

Figure 3.11 shows the *Camera* tab of the Commander GUI. It is used to request a control camera parameters and request a snapshot (label 3 in figure 3.11), control pipeline settings (label 4 in figure 3.11), and control IR LED settings (label 5 in figure 3.11). The captured camera frame is also displayed in this tab. When enabled, the *show markers* option (label 6 in figure 3.11) draws the received bounding boxes on top of the captured frame. This is done in real time. The log window (label 7 in figure 3.11) shows the Vision Add-On log. It also offers controls to configure the log level, store the log to a file and disable auto scroll. The top bar of the GUI is the network configuration window. It is used to select the IP of the Vision Add-On (label 1 in 3.11). All commands are sent to this IP. The same window also offers controls to update and persist the Vision Add-On's network configuration.

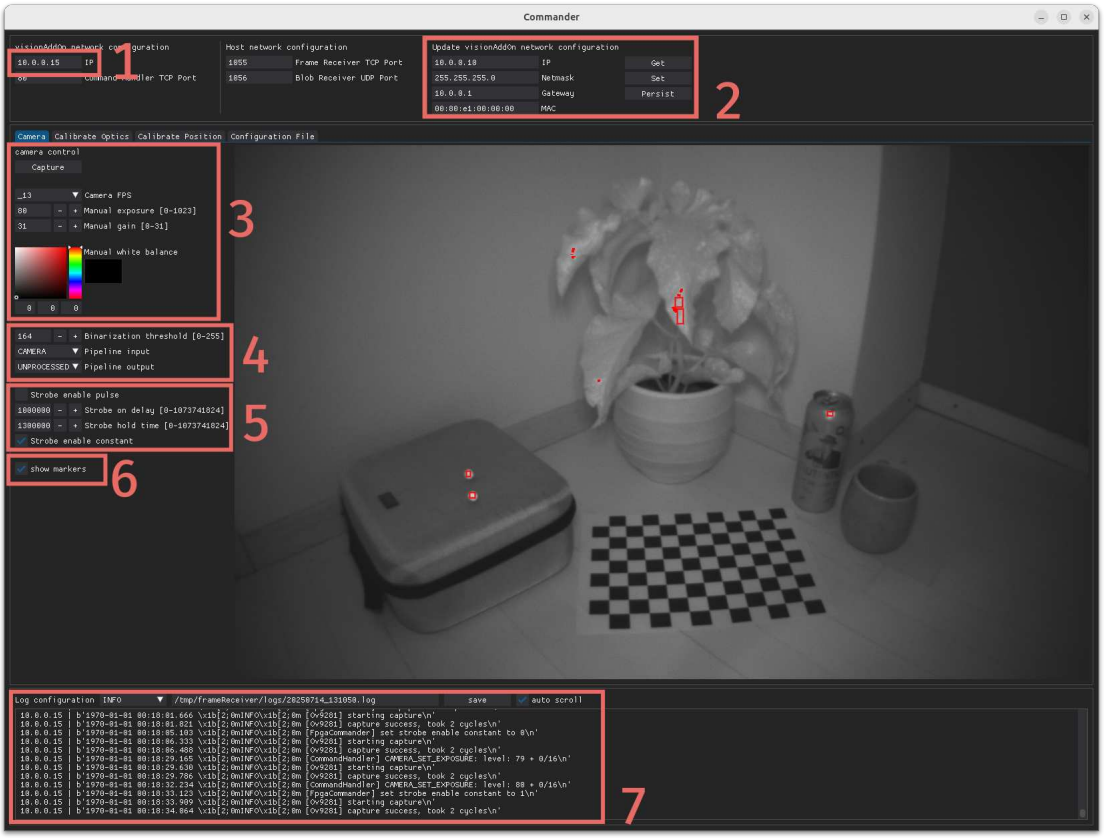


Figure 3.11: Commander GUI *Camera* tab

1: Vision Add-On IP selection
2: Update Vision Add-On network configuration
3: Camera controls
4: Pipeline controls
5: IR LED controls
6: Show bounding boxes
7: Vision Add-On log

3.8.2 Camera Calibration

Figure 3.11 shows the *Camera Calibration* tab of the Commander GUI. It was used to calibrate each camera individually. Ref 3.13 shows a collection of pictures taken for this procedure. Between 45 and 70 images were taken with each device. The pictures were taken from different positions and angles. Care was taken to capture different images where the chessboard is located in different position of the image to ideally cover the whole sensor with the calibration sequence. In the Commander GUI, the path to the directory containing the images for one device can be selected (label 1 in 3.13). The chessboard grid size can also be configured. After computing the camera calibration parameters, they are displayed in the GUI and can be persisted on the Vision Add-On (label 2 in 3.13). The computed parameters are stored to a JSON file. This file will be reused for the position calibration. Additionally, the camera parameters stored on a Blob Detector can be downloaded using the `Load from device` button (label 3 in 3.13).

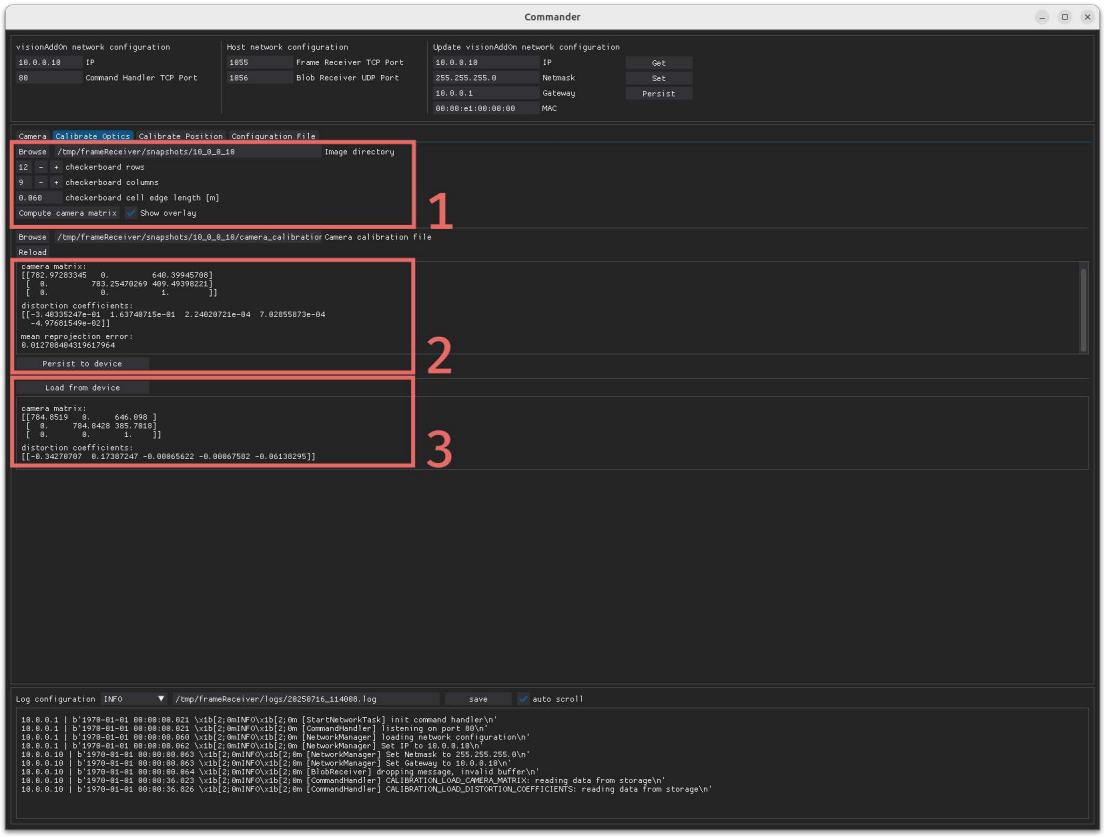


Figure 3.12: Commander GUI *Camera Calibration* tab

1: Compute camera position
2: Persist camera position to Vision Add-On
3: Load camera position from Vision Add-On

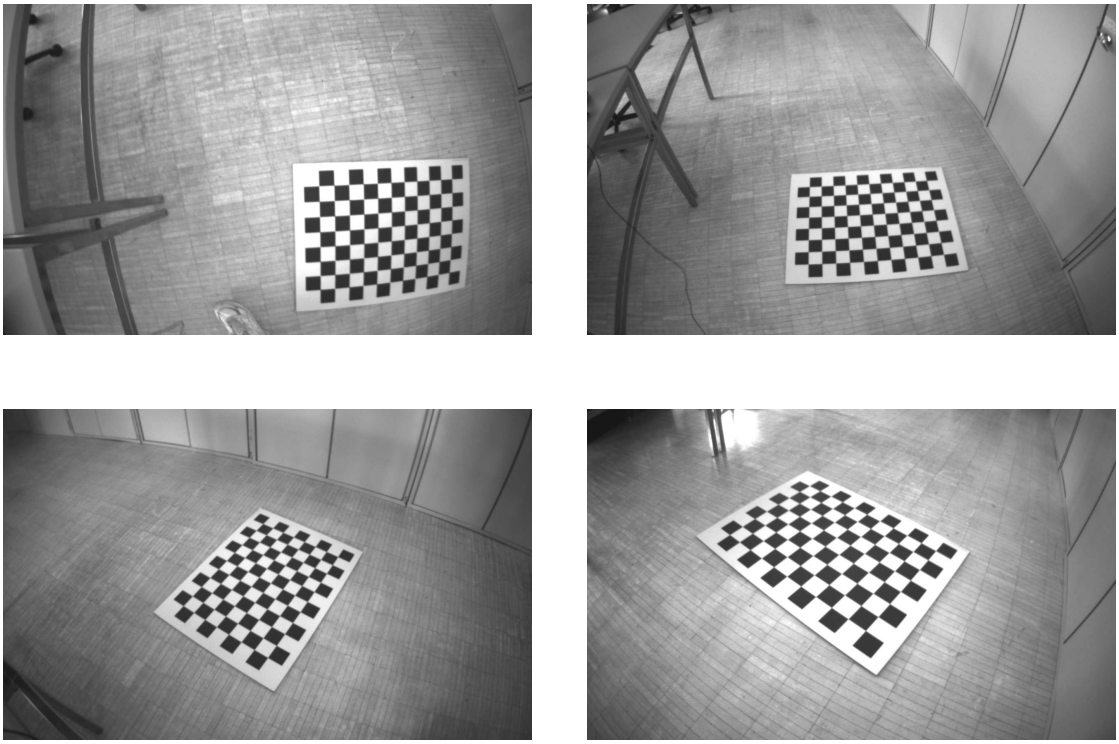


Figure 3.13: Images used for camera calibration

The optical filter was installed for all images

3.8.3 Position Calibration

To perform the position calibration, the chessboard was placed in the scene. Each Blob Detector needs to have a clear line of sight of the chessboard. Figure 3.14 shows the *Position Calibration* tab of the Commander GUI. The image containing the chessboard can be selected (label 1 in 3.14). Again, the chessboard size can be configured. To compute the position calibration parameters, the camera calibration parameters are required. They can be loaded from the JSON file created during the position calibration sequence. After computing the position calibration parameters, they are displayed in the GUI and can be persisted on the Vision Add-On (label 2 in 3.14). Additionally, the position calibration parameters stored on a Blob Detector can be downloaded using the `Load from device` button (label 3 in 3.14).

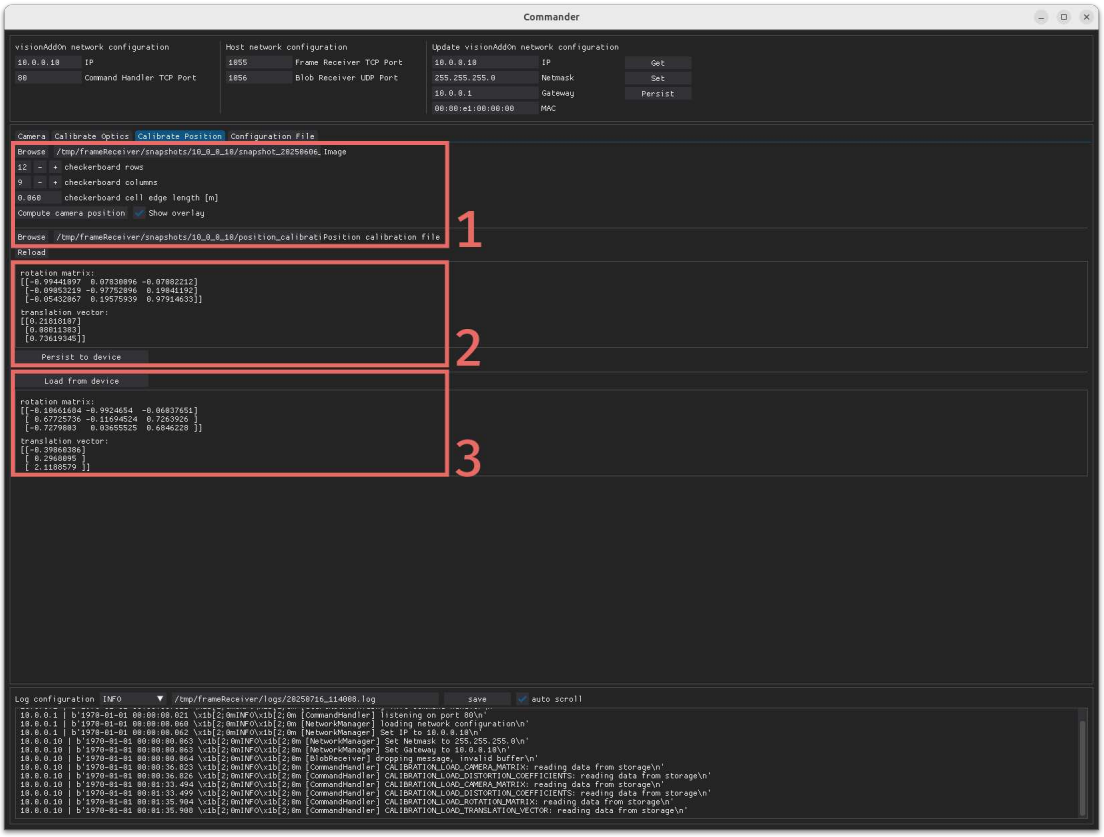


Figure 3.14: Commander GUI *Position Calibration* tab

1: Compute camera parameters
2: Persist camera parameters to Vision Add-On
3: Load camera parameters from Vision Add-On

3.9 Lens distortion

As an early test, linear movement was tracked. A single reflective marker as shown in figure 3.15 was mounted on a linear slider. The slider automatically moved from one end to the other. Figure 3.16a shows early measurement results. It is clearly visible, that the tracked movement does not follow a straight line. This is due to a bug in the code which distorts the received coordinates to account for the lens distortion. Since all receive coordinates were logged, the same data set can be reused with the lens distortion code corrected. Figure 3.16b shows the results. The tracked movement is now a straight line.

Figure 3.15: Reflective marker used in all tracking tests

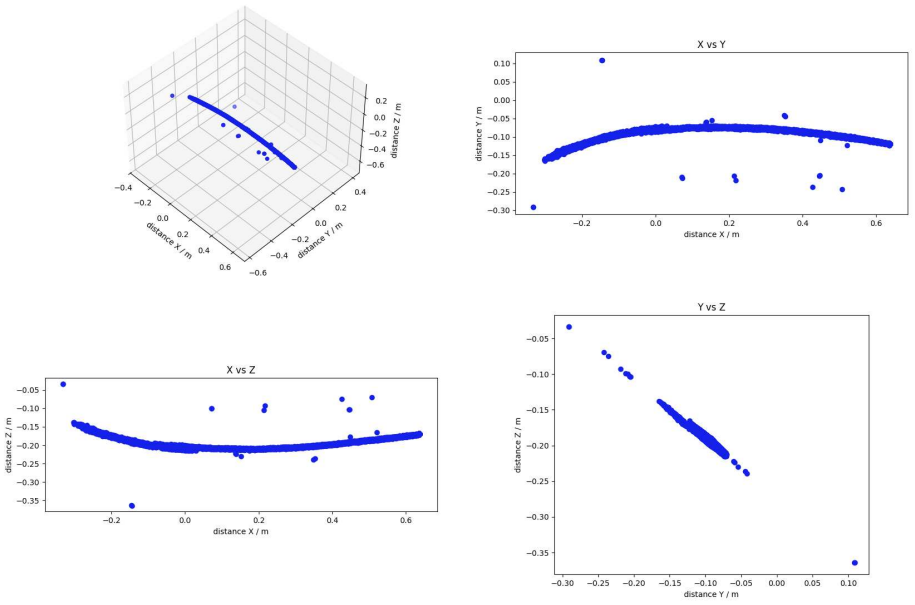
The marker diameter is 12 mm

The stray points seen in 3.16 are unwanted reflections picked up by the Blob Detectors.

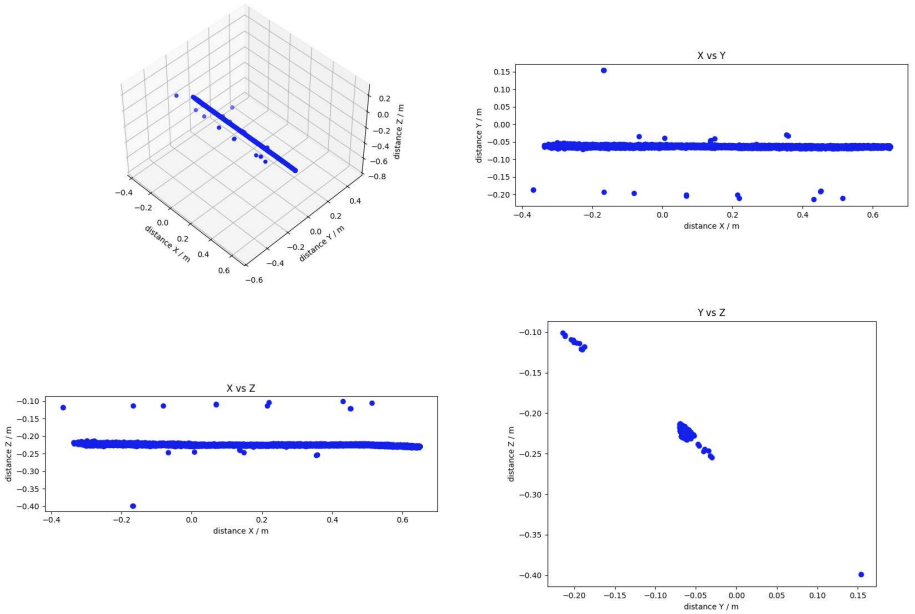
3.10 Tracking accuracy

To measure the absolute tracking precision of the system, a marker was placed on a two axis XY stage shown in figure 3.17. The XY stage was placed on the optical table show in 3.19a. The table is 0.6 m high. The Blob Detectors are mounted at a height of 2.5 m as shown in figure 3.19a. Five units were used for the tests. The Blob Detectors span a floor space of 2 m * 2.5 m. A curtain is used to block outside light from entering the test scene. The XY stage can be adjusted in both X and Y direction with greater than 0.1 mm precision. Using the XY stage, the marker was positioned at nine different position for nine different measurements. The positions are arranged in a 3 x 3 grid, evenly spaced 10 mm apart in both the X-axis and Y-axis. The triangulate.py script was used to measure each point individually. The movement in between points was not measured. Figure 3.18 shows the test setup recorded using the Blob Detector. The reflective marker, illuminated by the IR LEDs, shows up as a bright spot.

Figure 3.20 shows the combined measurement results. The red coordinates and red dots with a black edege are the mean of each cluster. The coordinates in the figure are the mean position of each cluster, To align the clusters with the position of the XY stage, all coordinates were offset. The offset is the mean of the cluster measured with the XY stage



(a) Linear movement triangulated without lens distortion correction



(b) Linear movement triangulated with lens distortion correction

Figure 3.16: The effects of lens distortion

deflection at X=0mm any Y=0mm. This results in an offset of (-152.08mm, 34.33mm, -170.88mm) Table 3.2 list the computed mean, spread and standard deviation of each cluster. The mean of the spread is (1.96 mm 1.76 mm, 2.85 mm) The mean standard deviation is (0.32 mm, 0.27 mm, 0.40 mm).

This test shows that in similar conditions a tracking accuracy of +/- 3mm can be expected.

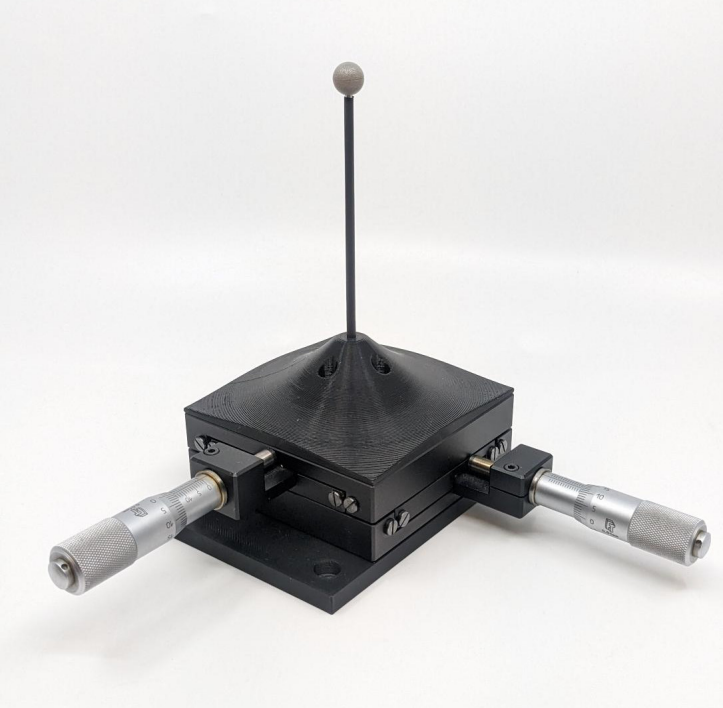


Figure 3.17: XY stage with a reflective marker mounted on top

3.11 Tracking moving objects

A robot arm was used to move a reflective marker along a predetermined path. The robot arm used it the Franka Emika Panda [70]. A reflective marker is placed on the end effector of the robot arm. Figure 3.19b shows how it was mounted. Figure 3.19b also shows some blue painters tape. This was an effort to reduce unwanted reflections. The tape was later replaced by black paper which is less reflective in the IR spectrum than the blue tape. The video [71] shows a screen recording of the Commander GUI where reflections are falsely picked up as markers.

The robot arm was programmed to trace half of the edges of a 120mm cube, figure 3.22 shows the end effector path. During the measurement, the robot arm repeated the same movement three times. The robot arm's advertised repetition accuracy is +/- 0.1 mm. No claims about the absolute position accuracy of the end effector are made. Figure 3.23

Figure 3.18: XY stage from the view of the Blob Detector

deflection / mm	mean / mm	spread / mm	standard deviation / mm
(0, 0)	(-0.0, 0.0, -0.0)	(2.1, 2.7, 4.3)	(0.3, 0.4, 0.4)
(0, 10)	(-0.5, -10.2, 1.1)	(1.9, 1.2, 2.6)	(0.4, 0.2, 0.4)
(0, 20)	(-0.4, -19.5, 0.2)	(1.5, 1.2, 2.5)	(0.2, 0.2, 0.3)
(10, 0)	(-10.6, -0.0, -0.1)	(2.1, 2.2, 2.8)	(0.4, 0.3, 0.4)
(10, 10)	(-10.2, -9.6, 1.3)	(1.2, 1.7, 1.7)	(0.2, 0.1, 0.2)
(10, 20)	(-10.7, -20.2, -0.2)	(1.8, 1.5, 2.0)	(0.2, 0.3, 0.4)
(20, 0)	(-20.5, 0.1, -0.0)	(2.7, 1.7, 3.6)	(0.4, 0.3, 0.5)
(20, 10)	(-20.2, -9.8, 0.8)	(2.0, 1.9, 2.3)	(0.3, 0.3, 0.2)
(20, 20)	(-19.2, -20.1, -0.2)	(2.4, 1.7, 3.8)	(0.4, 0.4, 0.8)

Table 3.2: Position accuracy measurement results at different XY stage deflections

(a) Five Blob Detectors mounted at a height of 2.5 m

(b) A reflective marker mounted on the robot arm

Figure 3.19: Blob Detector test setup

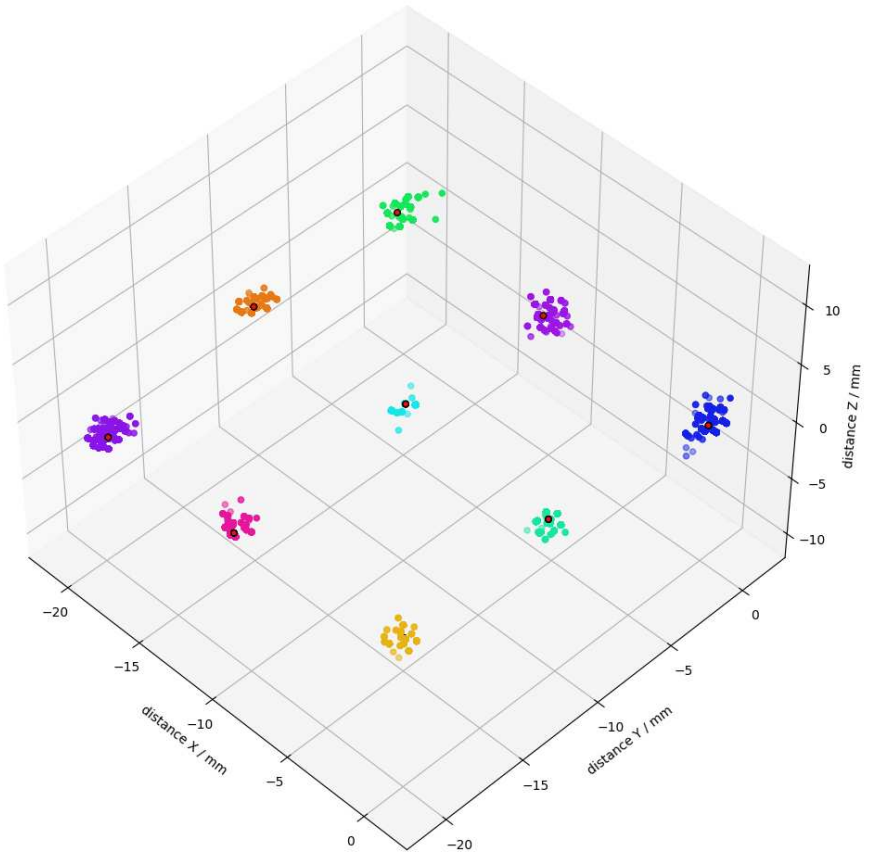
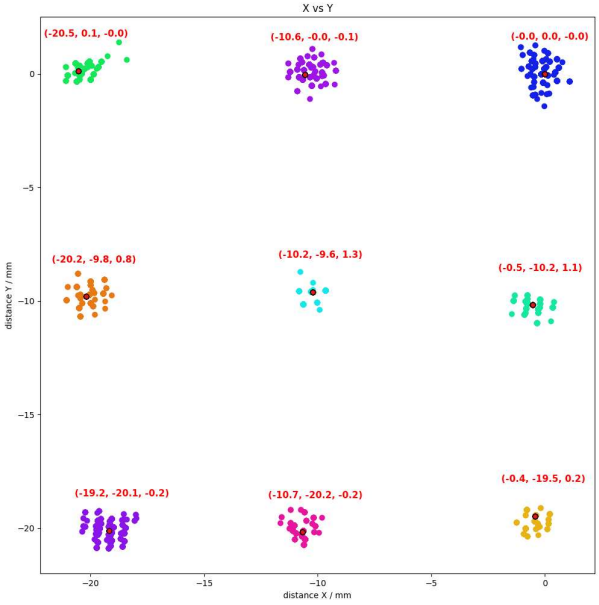


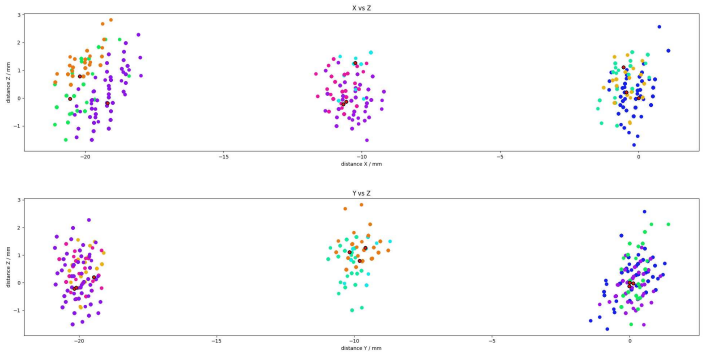
Figure 3.20: 3D view of position accuracy measurement results

deflection (0, 0) mm:
deflection (0, 10) mm:
deflection (0, 20) mm:
cluster mean:
deflection (10, 0) mm:
deflection (10, 10) mm:
deflection (10, 20) mm:
deflection (20, 0) mm:
deflection (20, 10) mm:
deflection (20, 20) mm:



(a) Top-down view of the position accuracy measurement results

The red coordinates are the mean of each cluster



(b) Side profile of the position accuracy measurement results

Figure 3.21: 2D views of the position accuracy measurement results

deflection (0, 0) mm:
deflection (0, 10) mm:
deflection (0, 20) mm:
cluster mean:
deflection (10, 0) mm:
deflection (10, 10) mm:
deflection (10, 20) mm:
deflection (20, 0) mm:
deflection (20, 10) mm:
deflection (20, 20) mm:

shows the triangulated marker positions. The spread of a cube (with no rotation) is its edge length. The spread of the measured points is (124, 128.45, 127) mm. Note that the measured cube is rotated along the Z axis. This can be seen in the slight tilt of the square shown in the *X vs Y* plot shown in figure 3.23. The rotation is due to the fact that the X and Y axis of the coordinate system used to measure the marker positions (determined by chessboard placement during position calibration) was not perfectly aligned with the coordinate system of the robot arm tracing the cube. This means that the spread can not be taken as the measured edge length of the cube. The Z axis of the traced cube is parallel to the Z axis of the robot arm, so the spread is the maximum measured edge length. The spread on the Z axis is 7 mm greater than expected. Since the absolute position accuracy of the robot arm is unknown, it is not clear how much of the measured inaccuracy comes from the tracking system versus the robot arm. However, the position accuracy test performed in 3.10 showed that the tracking accuracy along the Z axis is the worst.

Figure 3.24a shows the triangulated marker positions rendered in real-time using VisPy. The video [72] shows two screen recordings of the Commander GUI from different Blob Detectors. The GUI was configured to overlay the marker position on top of a captured frame. Both a recording of the robot arm and the real-time render are synchronized with the Commander GUI perspectives. The video shows that the system can keep up with the fast movements of the robot arm. A screenshot of the video is shown in 3.24b.

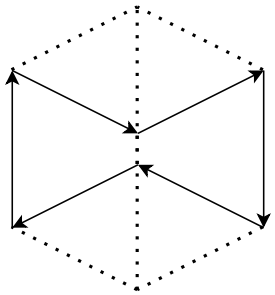


Figure 3.22: Programmed end effector path

This test shows that the full system is capable of tracking a single marker at 72 fps per camera and plot the triangulated data in real time.

3.12 Bottlenecks

3.12.1 KLT-E4MPF-OV9281 V4.2 NIR

The frame rate of the camera determines the tracking rate of the Blob Detector. The KLT-E4MPF-OV9281 V4.2 NIR is currently only running at 72 fps using the DVP interface. It is unclear if the camera supports the advertised 120 fps over the DVP interface, or if the highest frame rate is only available over the MIPI CSI-2 interface.

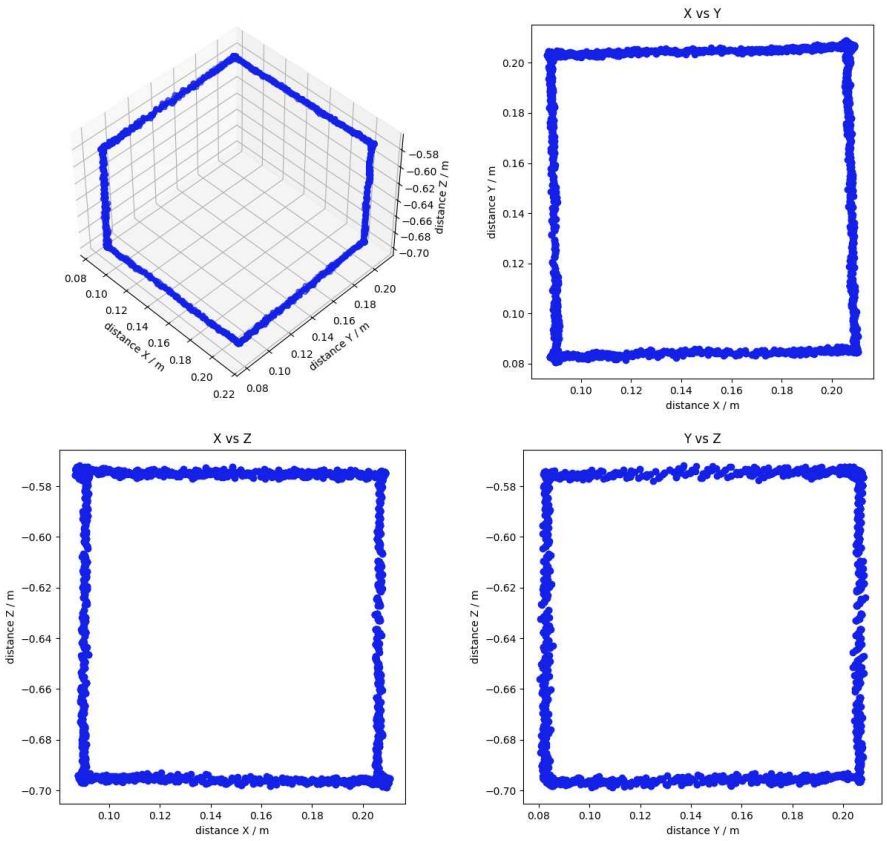
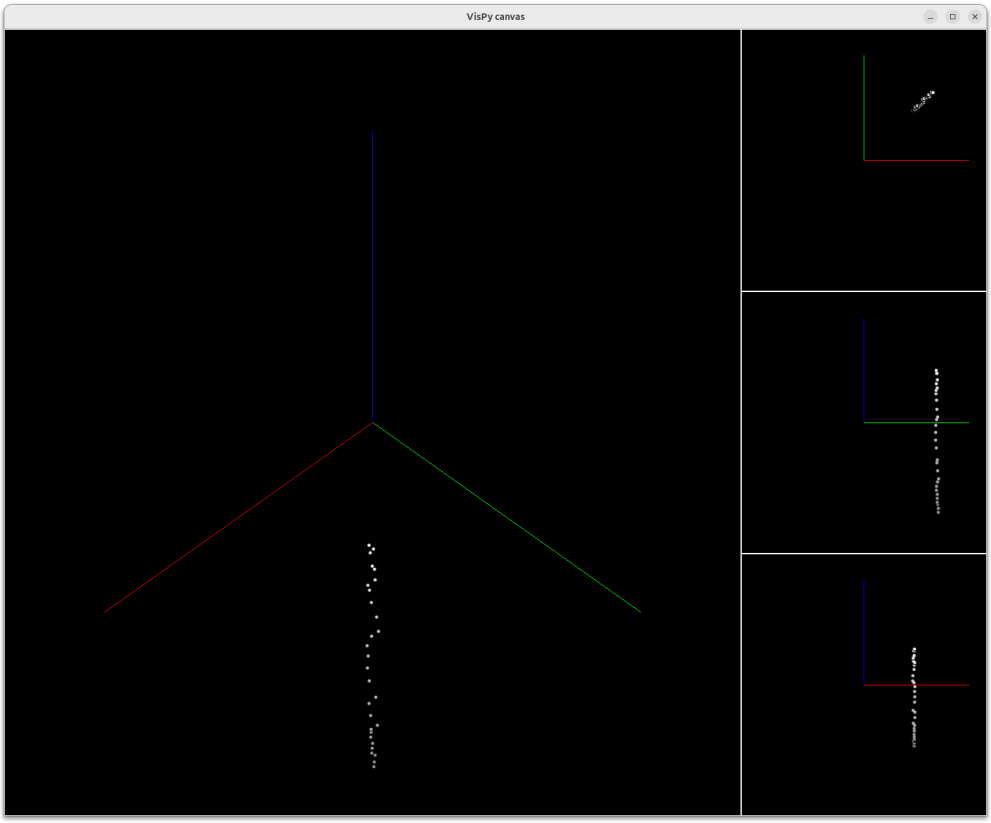
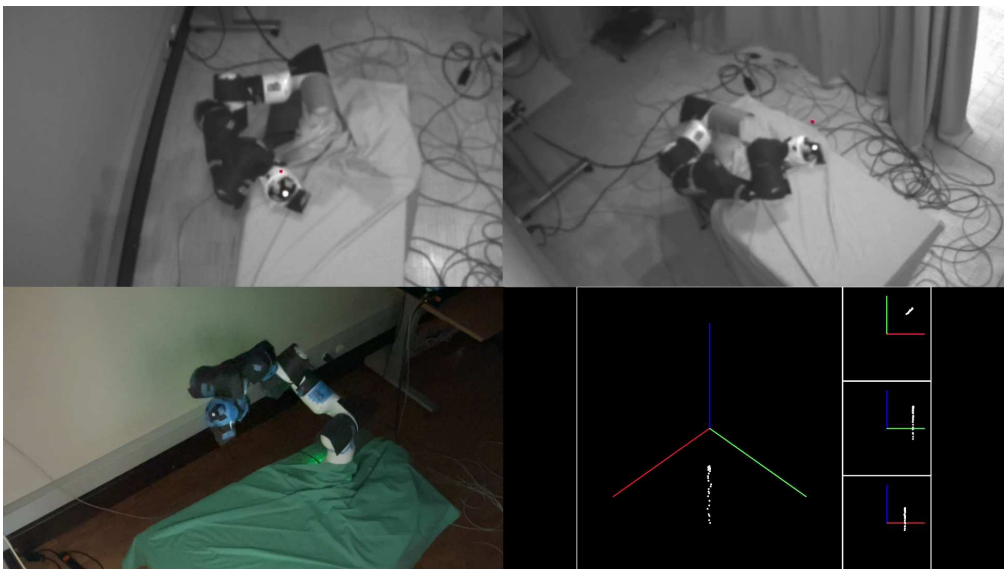


Figure 3.23: 120 mm cube measurement results



(a) Tracked markers rendered in real-time using VisPy



(b) Screenshot of video [72]

Figure 3.24: Real-time render results

3.12.2 CCA

As long as the image data is provided one pixel at a time and the FPGA fabric can handle the clock rate of the image source, the CCA algorithm can keep up.

3.12.3 Double Buffer and SPI

The Double Buffer's write rate is determined by the pixel clock. As long as the FPGA fabric can handle the pixel clock, the Double Buffer's producer side can keep up. The consumer side needs to be clocked fast enough to read the full buffer depth within the time of one frame. Using the following constants, equation 3.1 describes this relationship:

- f_{pclk} frequency of the pixel clock
- T_{swap} time between the producer's `switch` rising edge and the consumer's `newData` rising edge
- $T_{readFeature}$ time to read one feature
- n maximum number of features stored in buffer

$$1/f_{pclk} >= T_{swap} - (T_{readFeature} * n) \tag{3.1}$$

The SPI transfer needs to be able to transmit all detected bounding boxes within the period of one frame. Using a logic analyser, the SPI transfer sequence was measured. When transferring 254 bounding boxes, the SPI clock is active for 720us. At a frame rate of 72 Hz, the SPI transfer is active for 52% of the frame. During the transfer, the measured SPI clock frequency is 18.519 MHz. The current system configuration only supports half the amount of bounding boxes (126 instead of the measured 254), so the current SPI load is approximately at 25% of its total throughput.

If the double buffer or the SPI transfer ever becomes a bottleneck, increasing the system clock could be a solution. If the system clock can not be increased, the number of features stored in the Double Buffer could be reduced.

3.12.4 UDP data transfer

UDP does not guarantee that the packets are received in the same order that they were sent in. For all tests, the Blob Detectors and the Host PC used their own Ethernet network with no other traffic. Out of order packet reception was never observed, and no packets were dropped.

Figure 3.25 shows a timing analysing based on the Blob Receiver reception time stamps. The packets sent by the Blob Detector 10.0.0.10 sent during the test documented in 3.11

are used. The measured mean in between received packets is 13.7 ms. This is less than the ideal at $1/(72\text{Hz}) = 13.9\text{ms}$. The cause is unknown and hard to diagnose with just the recorded data. The camera pixel clock, the SPI transfer, the lwip stack timing, and finally Python's timing all have an impact on the measured timestamp. Adding time stamps to the data on the GECKO5Education before sending it to the Vision Add-On, as well as adding a timestamp to the UDP packet on the Vision Add-On would help perform a better timing analysis. Note that the GECKO5Education actually has a real time clock which could be synchronized using Network Time Protocol (SNTP) via the Vision Add-On. This timing data could also be used to improve triangulation. Data which is too old or out of sync compared to other packets could be ignored.

The spike shown in 3.25 is not caused by the Blob Detector. The packet timing of all Blob Detectors used during the test show the same spike. It is either caused by the network or the host PC.

The timing jitter is not noticeable, an improvement over the noticeable lag of the perifAddOn used in [2].

3.13 Cost Breakdown

A budget of CHF 2000.- was provided for this thesis. This limit was not reached.

Table 3.3 lists the estimated cost of manufacturing the Blob Detector. The GECKO5Education units used were produced outside the scope of this project in a batch of 100 units for a cost of CHF 238.- per unit. The cameras were also not paid for with the project budget. Ethernet patch cables and 3D printer filament were available at BFH. The cost of these items is still included in the cost breakdown.

The Vision Add-On has a unit cost of CHF 224.90. Including the price for the GECKO5Education, each Blob Detector costs CHF 462.90 to build. Additionally, CHF 200.- are accounted for RJ45 patch cables, a PoE switch, and 1 kg of PLA filament for 3D printing all mechanical parts. This puts the real cost of this project at CHF 2977.40.

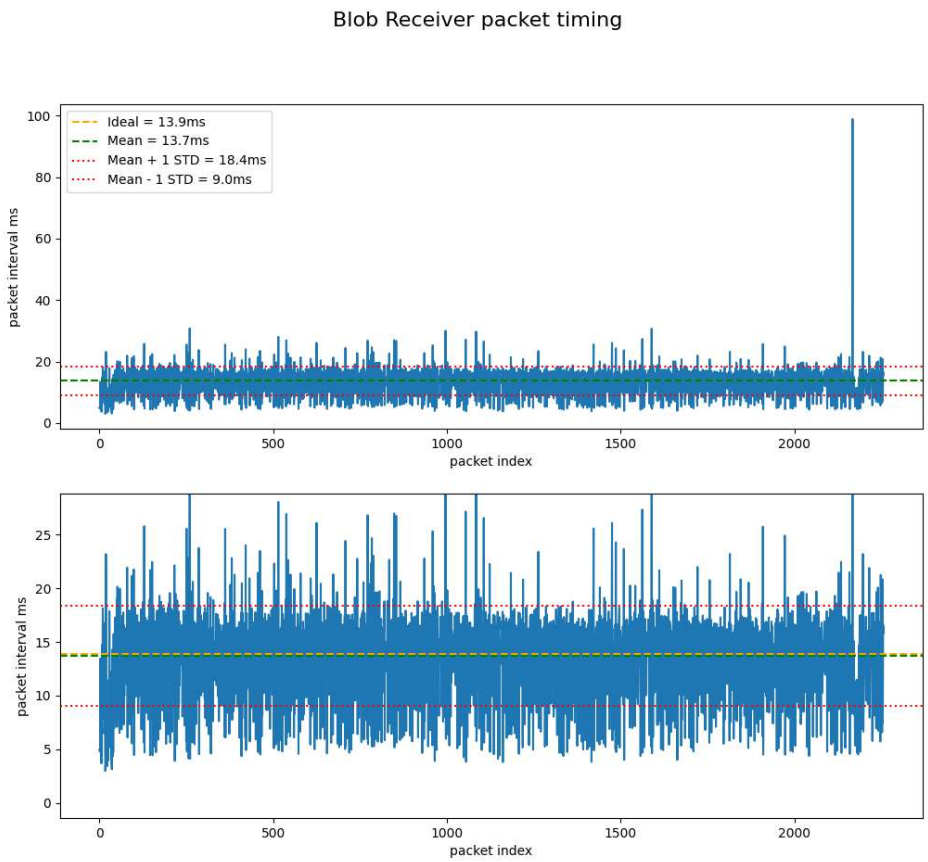


Figure 3.25: Blob Receiver timing in between received packets

Name	units	total Cost / CHF	cost normalized to 1 unit
GECKO5Education	100	23800	238
BOM Vision Add-On	6	580	96.70
PCB Vision Add-On	6	645	107.50
KLT-E4MPF-OV9281 V4.2 NIR camera	45	402	8.90
FH-R668 IR LED	6	60	10
700 nm - 1600 nm optical filter	10	18	1.80
total			224.90

Table 3.3: Cost breakdown

4 Conclusion

KLT-E4MPF-OV9281 V4.2 NIR camera The KLT-E4MPF-OV9281 V4.2 NIR camera is a suitable camera for an IR-based MoCap system when paired with an optical filter to block out visible light. The camera resolution is 1280x800 pixels at a frame rate of 72 fps. At the time of writing, the OV9281 sensor offers the best resolution and frame rate of any sensor using the DVP interface. For higher resolutions cameras with higher frame rates, the MIPI CSI-2 interface is required.

Image processing pipeline The pipeline has no issues keeping up with the 72 fps 1280x800 DVP signal provided by the OV9281 camera sensor. The Fake Camera modules developed as part of this project proved very useful for verification in both test benches and integration tests.

Vision Add-On All requirements outlined in chapter 2.2.8 were met.

- The custom PCB allowes the use of the KLT-E4MPF-OV9281 V4.2 NIR camera.
- The Wi-Fi issues of the perifAddOn observed in [2] are solved with the wired Ethernet connection.
- A camera frame can be captured and published over Ethernet
- The Blob Detector uses PoE
- The EEPROM storage is used to persist configuration parameters and calibration data
- The IR-LED can strobe in sync with the camera frame rate

Even though the IR-LED can operate as a strobe, it turned out that it is better to used it as a constant light source since the light output is higher.

Triangulation The tracking accuracy of approximately +/- 3mm leaves room for improvement. A single marker can be tracked at a rate of 72Hz. The biggest draw back of the system is that it can not handle more than one marker. This is not a hardware limitation. An update to the triangulation software can solve this limitation.

Low-Cost This thesis had a budged of CHF 2000.- However, the GECKO5Education boards were not part of the project budget and the PCB assembly was done for free by an apprentice. The cameras were also not paid for using the project budget.

It is not likely that this project can be reproduced with the target budget of CHF 2000.-.

Open-source All files required to replicate the project are hosted at [73].

Outlook The current design of the Vision Add-On would allow the use of the MIPI CSI-2 interface of the KLT-E4MPF-OV9281 V4.2 NIR camera. Integrating the open-source Circuit Valley MIPI IP-core [32] into the GECKO5Education would allow the camera to run at 120 fps.

A redesign combining the Vision Add-On and the GECKO5Education could reduce cost. Less than 13% of the FPGA's available LUTs are used. A less capable FPGA could be used to save cost. At the time of writing, the STM32F7 family are the cheapest microcontrollers produced by STMicroelectronics which allows the use of both external SDRAM and the DCMI interface. Both are required to capture a camera frame. The PoE module could also be replaced by one which can deliver less power to further lower cost.

Creating a brighter IR light source using a current source LED driver could improve the physical tracking volume of the system.

Updating the triangulation code with the ability to track multiple markers would turn this project into a usable MoCap system. To do so, marker correspondence needs to be solved. This is typically done using epipolar geometry. Epipolar geometry is the geometric relationship between two camera views of the same scene, which constrains where the projection of a 3D point in one image must lie in the other image. Additionally, the triangulation method could be replaced by one which does not weigh all perspectives equally. A straight on view from a close distance should contribute more to the solution than a view from far away. For markers of the same size, the bounding box is smaller if the marker is further away. This fact could also be used to determine the weight of different views.

Declaration of Authorship

I hereby declare that I have written this thesis independently and have not used any sources or aids other than those acknowledged.

All statements taken from other writings, either literally or in essence, have been marked as such.

I hereby agree that the present work may be reviewed in electronic form using appropriate software.

July 20, 2025

T. Jäggi

Bibliography

[1] Tobias Jäggi. Low cost motion capture - project i: Image processing pipeline. Technical report, Bern University of Applied Sciences, 2023.

[2] Tobias Jäggi. Low cost motion capture - project ii: Marker detection unit. Technical report, Bern University of Applied Sciences, 2024.

[3] OptiTrack. Optitrack. `https://optitrack.com`. Accessed 18.07.2025.

[4] Diederick C Niehorster, Li Li, and Markus Lappe. The accuracy and precision of position and orientation tracking in the HTC vive virtual reality system for scientific research. *Iperception*, 8, 2017.

[5] Samet Ciklacandir, Sultan Ozkan, and Yalcin Isler. A comparison of the performances of video-based and imu sensor-based motion capture systems on joint angles. In *2022 Innovations in Intelligent Systems and Applications Conference (ASYU)*, pages 1–5, 2022.

[6] FreeMoCap. The freemocap project. `https://freemocap.org/`. Accessed 18.07.2025.

[7] Zhejiang University. Easymocap. `https://github.com/zju3dv/EasyMocap`. Accessed 18.07.2025.

[8] Jan Dellsperger and David Sheppard. Mimikry - ein low-cost motion-tracking system. Technical report, Bern University of Applied Sciences, 2019.

[9] Jia Wei Tang, Nasir Shaikh-Husin, Usman Ullah Sheikh, and M. N. Marsono. A linked list run-length-based single-pass connected component analysis for real-time embedded hardware. *Journal of Real-Time Image Processing*, 15(1):197–215, Jun 2018.

[10] Theo Kluter. Gecko5education. `https://github.com/logisim-evolution/GECKO5Education`, 2024. Accessed 18.07.2025.

[11] Espressif. Esp32-c3-mini-1. `https://www.espressif.com/sites/default/files/documentation/esp32-c3-mini-1_datasheet_en.pdf`. Accessed 18.07.2025.

[12] YosysHq. Oss cad suite. `https://github.com/YosysHQ/oss-cad-suite-build`. Accessed 18.07.2025.

[13] YosysHQ. Yosys source. `https://github.com/YosysHQ/yosys`. Accessed 18.07.2025.

[14] GHDL. Ghdl yosys plugin. https://github.com/ghdl/ghdl-yosys-plugin. Accessed 18.07.2025.

[15] Lattice. *ECP5 and ECP5-5G Family Data Sheet*. Lattice, 2022.

[16] YosysHQ. Yosys documentation. https://yosys.readthedocs.io/_/downloads/en/latest/pdf/. Accessed 18.07.2025.

[17] YosysHQ. nextpnr. https://github.com/YosysHQ/nextpnr. Accessed 18.07.2025.

[18] YosysHQ. Project trellis. https://github.com/YosysHQ/prjtrellis. Accessed 18.07.2025.

[19] Gwenhael Goavec-Merou. openfpgaloader. https://github.com/trabucayre/openFPGALoader. Accessed 18.07.2025.

[20] VERIPOOL. Verilator. https://github.com/verilator/verilator. Accessed 18.07.2025.

[21] Stephen Williams. Icarus verilog. https://github.com/steveicarus/iverilog. Accessed 18.07.2025.

[22] cocotb. cocotb. https://github.com/cocotb/cocotb. Accessed 18.07.2025.

[23] openrisc. openrisc. https://openrisc.io/. Accessed 18.07.2025.

[24] ISSI. *IS42/45SM/RM/VM16160K SDRAM Data Sheet*. ISSI, 2015.

[25] Winbond. *W25Q128JVS Data Sheet*. Winbond, 2024.

[26] Theo Kluter. Epfl embedded system design lecture 3 - custom instructions.

[27] Saurabh Verma and Ashima Dabare. Understandin clock domain crossing issues. https://www.eetimes.com/understanding-clock-domain-crossing-issues/. Accessed 18.07.2025.

[28] nadland. Spi master. https://github.com/nandland/spi-master. Accessed 18.07.2025.

[29] KiCad. Kicad. https://www.kicad.org/. Accessed 18.07.2025.

[30] Theo Kluter. Gecko5education add-on template. https://github.com/logisim-evolution/GECKO5Education/tree/main/kicad/addonTemplate, 2025. Accessed 18.07.2025.

[31] MIPI Alliance. Mipi csi-2. https://www.mipi.org. Accessed 18.07.2025.

[32] Circuit Valley. Usb c industrial camera fpga usb3 sources. https://github.com/circuitvalley/USB_C_Industrial_Camera_FPGA_USB3. Accessed 18.07.2025.

[33] Circuit Valley. Usb c industrial camera fpga usb3 blog. https://www.circuitvalley.com/2022/06/

pensource-usb-c-industrial-camera-c-mount-fpga-imx-mipi-usb-3-crosslinknx.html. Accessed 18.07.2025.

[34] Toshiba. Tc358748xbg datasheet. https://toshiba.semicon-storage.com/info/TC358746AXBG_datasheet_en_20201214.pdf?did=30612&prodName=TC358746AXBG, 2020. Accessed 18.07.2025.

[35] Omnivision. *OV9281 Datasheet Version 1.53*. Omnivision, 2019.

[36] Waveshare. Ov9281-110 mono camera for raspberry pi. https://www.waveshare.com/ov9281-110-camera.htm. Accessed 18.07.2025.

[37] Antmicro. Ov9281 dual camera board. https://github.com/antmicro/ov9281-camera-board. Accessed 18.07.2025.

[38] STMicroelectronics. *Introduction to digital camera interface (DCMI) for STM32 MCUs Application Notes*. STMicroelectronics, 2023.

[39] BFH. Leguan. https://leguan.ti.bfh.ch/. Accessed 18.07.2025.

[40] Aliexpress; amto store. Store operated by Shenzehn Yi Chang Wang Electronics Co., Ltd.

[41] Aliexpress; tangsinuo store. Store operated by Shijiazhuang Tang Snow Photoelectric Technology Co., Ltd.

[42] CuteCom. Cutecom. https://gitlab.com/cutecom/cutecom/. Accessed 18.07.2025.

[43] geeksforgeeks. Write your own atoi(). https://www.geeksforgeeks.org/write-your-own-atoi/. Accessed 18.07.2025.

[44] rust. rust. https://www.rust-lang.org/. Accessed 18.07.2025.

[45] bernRtos. bernrtos. https://bern-rtos.org/. Accessed 18.07.2025.

[46] smoltcp. smoltcp. https://crates.io/crates/smoltcp. Accessed 18.07.2025.

[47] Tobias Jäggi. Vision add-on risks. Submitted as supplementary material alongside the main thesis, 2025.

[48] STMicroelectronics. Stm32cube. https://www.st.com/en/development-tools/stm32cubemx.html. Accessed 18.07.2025.

[49] Tobias Jäggi. Vision add-on commands. Submitted as supplementary material alongside the main thesis, 2025.

[50] Tobias Jäggi. Vision add-on commands hosted. https://github.com/TobiasJaeggi/fpga-mocap/blob/main/visionAddOn/firmware/App/command/commands.md. Accessed 10.07.2025.

[51] Omnivision. *OV9282 Camera Module Application Note Version 1.00*. Omnivision, 2017.

[52] Astral Software Inc. uv. https://docs.astral.sh/uv/. Accessed 18.07.2025.

[53] Python Software Foundation. Python installation instructions. https://wiki.python.org/moin/BeginnersGuide/Download. Accessed 18.07.2025.

[54] Astral Software Inc. uv installation instructions. https://docs.astral.sh/uv/getting-started/installation/. Accessed 18.07.2025.

[55] Python Software Foundation. Python struct. https://docs.python.org/3/library/struct.html. Accessed 18.07.2025.

[56] Python Software Foundation. asyncio. https://docs.python.org/3/library/asyncio.html. Accessed 18.07.2025.

[57] Python Software Foundation. Python threading. https://docs.python.org/3/library/threading.html. Accessed 18.07.2025.

[58] OpenCV. Opencv. https://opencv.org/. Accessed 18.07.2025.

[59] OpenCV. Opencv distortion model. https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html#ga7dfb72c9cf9780a347fbe3d1c47e5d5a. Accessed 18.07.2025.

[60] OpenCV. Opencv camera calibration tutorial. https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html. Accessed 18.07.2025.

[61] OpenCV. Opencv pose estimation tutorial. https://docs.opencv.org/4.x/d7/d53/tutorial_py_pose.html. Accessed 18.07.2025.

[62] Stack Overflow user Nirmal. Answer to stack overflow post - generalisation of the “mid-point” method for triangulation to n points. https://stackoverflow.com/questions/28779763/generalisation-of-the-mid-point-method-for-triangulation-to-n-points. Accessed 18.07.2025.

[63] Peter Sturm Srikumar Ramalingam, Suresh K. Lodha. A generic structure-from-motion framework. In *Computer Vision and Image Understanding*, volume 103, pages 218–228, 2006.

[64] VisPy. Vispy. https://vispy.org/. Accessed 18.07.2025.

[65] Pygame. Pygame. https://www.pygame.org. Accessed 18.07.2025.

[66] Blender. Blender. https://www.blender.org/. Accessed 18.07.2025.

[67] OptiTrack. Optitrack versa x 120. https://www.optitrack.com/cameras/versax-120/. Accessed 18.07.2025.

[68] Lattice. *ECP5 and ECP5-5G sysI/O Usage Guide*. Lattice, 2023.

[69] Waveshare. Ov5640 camera board (b). https://www.waveshare.com/ov5640-camera-board-b.htm. Accessed 18.07.2025.

[70] Franka robotics. https://franka.de/. Accessed 18.07.2025.

[71] Tobias Jäggi. Open source fpga based mocap - test 5. https://youtu.be/8qy8fF-Y2Gw. Accessed 18.07.2025.

[72] Tobias Jäggi. Open source fpga based mocap - test 4. https://youtu.be/6Va6dF9vp88. Accessed 18.07.2025.

[73] Tobias Jäggi. Low-cost fpga-based motion capture. https://github.com/TobiasJaeggi/fpga-mocap. Accessed 18.07.2025.

[74] Omar Cornut. Dear imgui. https://github.com/ocornut/imgui. Accessed 18.07.2025.

[75] Jonathan Hoffstadt. Dear pygui. https://github.com/hoffstadt/DearPyGui. Accessed 18.07.2025.

[76] OpenCV. opencv-python. https://github.com/opencv/opencv-python. Accessed 18.07.2025.

[77] PlatformIO. Platformio. https://platformio.org/. Accessed 18.07.2025.

List of Figures

List of Tables

List of Listings

Glossary

bitstream **bitstream** The binary configuration that implements a hardware design on a specific FPGA.

BSP Board Support Package

CCA **Connected Component Analysis (CCA)** An image processing technique used to identify and label clusters of connected pixels.

CI custom instruction

cocotb **cocotb** A Python library for writing for digital designs described in Verilog and VHDL.

CuteCom **CuteCom [42]** A graphical serial terminal.

DCMI **Digital Camera Interface (DCMI)** A hardware peripheral available on some STMicroelectronics microcontrollers capable of capturing VGA-like camera feeds.

Dear ImGui **Dear ImGui [74]** A C++ user interface library.

Dear PyGui **Dear PyGui [75]** A Python wrapper for Dear ImGui.

DMA Direct Memory Access

ECP5 **ECP5** A family of Lattice FPGAs.

ecppack **ecppack** An open-source tool within the OSS CAD Suite used for generating bitstreams for ECP5 FPGAs.

ESP32-C3-MINI-1 **ESP32-C3-MINI-1** An SoC by Espressif featuring a single-core 32-bit RISC-V microprocessor, Wi-Fi and Bluetooth connectivity.

FPGA **Field-Programmable Gate Array (FPGA)** A semiconductor device with a matrix of configurable logic blocks and programmable interconnects that allows it to be reprogrammed after manufacturing, enabling flexibility and adaptability in meeting diverse application or functionality requirements.

GECKO5Education **GECKO5Education** An open-source and open-hardware educational platform featuring a Lattice ECP5 FPGA.

GHDL Yosys Plugin **GHDL Yosys Plugin** Yosys plugin which enables the synthesis of VHDL using Yosys.

GTKWave **GTKWave** value change dump (vcd) file viewer.

HDL hardware description language

HPS **Hard Processor System (HPS)** A dedicated computing system implemented in hardware instead of FPGA fabric.

IMU **Inertial Measurement Unit (IMU)** An sensor device which measures data related to an object's motion and orientation in three-dimensional space.

IR **Infrared (IR)** A spectrum of light not visible to the human eye.

iverilog **ICARUS Verilog (iverilog)** An open-source Verilog simulation and synthesis tool commonly used in digital design and verification.

Lattice LFE5U-85F-6BG381C **Lattice LFE5U-85F-6BG381C** A Lattice ECP5 FPGA with 85k programmable LUTs.

LinkRunCCA **Linked List Run-Length-Based Single-Pass Connected Component Analysis (LinkRunCCA)** LinkRunCCA is a real-time single-pass connected component analysis/labelling (CCA/CCL) hardware implementation in Verilog HDL [9].

LUT **Lookup Table (LUT)** A data structure used to map input values to output values.

MoCap **Motion Capture (MoCap)** Technology which tracks a body's movement in space and ouputs data based on the movement.

netlist **netlist** A structured representation of an electronic design.

nextpnr **nextpnr** Vendor neutral FPGA place and route tool.

NIOS II **NIOS II** soft-core designed by Altera.

NIR Near Infrared

OpenCV **OpenCV [58]** An open-source, cross-platform computer vision library.

opencv-python **opencv-python [76]** Python package offering Python bindings to OpenCV

openFPGALoader **openFPGALoader** Universal utility for programming FPGAs.

OR1420 OpenRISC 1240

OR1k OpenRISC 1000

OSS CAD Suite **OSS CAD Suite** An open-source toolchain for for FPGA and ASIC design, integrating various tools for tasks like synthesis, simulation, place-and-route, and verification.

perifAddOn **perifAddOn** An open-hardware expansion board for GECKO5Education featuring an ESP32-C3-MINI and SD card reader.

PoE Power over Ethernet

QE Quantum Efficiency

RTL Register-transfer level

SoC **System-on-Chip (SoC)** An integrated circuit which features multiple electronic or computing subsystems on the same chip.

soft-core **soft-core** A computing system implemented in FPGA fabric instead of dedicated hardware.

STM32CubeMX **STM32CubeMX [48]** A code generation program to configure STMicroelectronics microcontrollers.

STMicroelectronics **STMicroelectronics** A semiconductor company specializing in the design and manufacture of a wide range of integrated circuits and electronic components.

SystemC **SystemC** A C++ library and simulation kernel used for modeling and simulating electronic systems at various levels of abstraction.

SystemVerilog **SystemVerilog** An extention to Verilog HDL that includes features for design, verification, and synthesis of digital systems.

uv **uv [52]** A Python package and project manager.

vcd **value change dump (vcd)** Text-based file format to record changes in signal values over time.

Verilator **Verilator** An open-source tool that converts Verilog HDL designs into C++ or SystemC code. Known for its high performance and use in verifying large-scale digital circuits.

VS Code **Visual Studio Code (VS Code) [77]** An open-source code editor developed by Microsoft.

vvp **vvp** ICARUS Verilog runtime engine.

Yosys **Yosys** An open-source framework for Verilog RTL synthesis.

.1 Items delivered

The following items were delivered alongside this document:

- `blobDetector.tar.gz`: GECKO5Education HDL project sources, referred to as `<gecko5>`
- `cameraCalibration.tar.xz`: Images used to calibrate the Blob Detectors camera
- `commandsVisionAddOn.html`: Packet structure of all commands implement by the Vision Add-On command handler ([49])
- `firmware.tar.gz`: Vision Add-On firmware project sources, referred to as `<firmware>`
- `hardware.tar.gz`: Vision Add-On PCB KiCad project sources
- `OpenSourceFPGABasedMoCap-Test4.mkv`: Copy of demo video [72]
- `OpenSourceFPGABasedMoCap-Test5.mkv`: Copy of demo video [71]
- `or1k.tar.gz` Precompiled amd64 OpenRISC GCC toolchain
- `OV9281_v1.53.pdf`: Omnivision OV9281 camera sensor data sheet
- `OV9282CameraModuleApplicationNotesR1.00.53.pdf`: Omnivision OV9282 camera sensor app notes
- `pinmapping.html` Vision Add-On mezzanine connector pin mapping
- `risksVisionAddOn.html` Vision Add-On risk analysis ([47])
- `software.tar.gz`: Python project sources, referred to as `<firmware>`
- `stl.tar.gz`: STL exports of all mechanical parts
- `visionAddOn.pdf`: Vision Add-On schematics