Applying Binarized Neural Networks on FPGAs to an Autonomous Driving Problem

Technical Report

Felix Müller (5017732)
Niklas Krekel (5016745)
Pablo Navarro (5016953)
Raffael Kaehn (5016769)
Prof. Dr. Jan Bredereke

HSB
Hochschule Bremen
City University of Applied Sciences

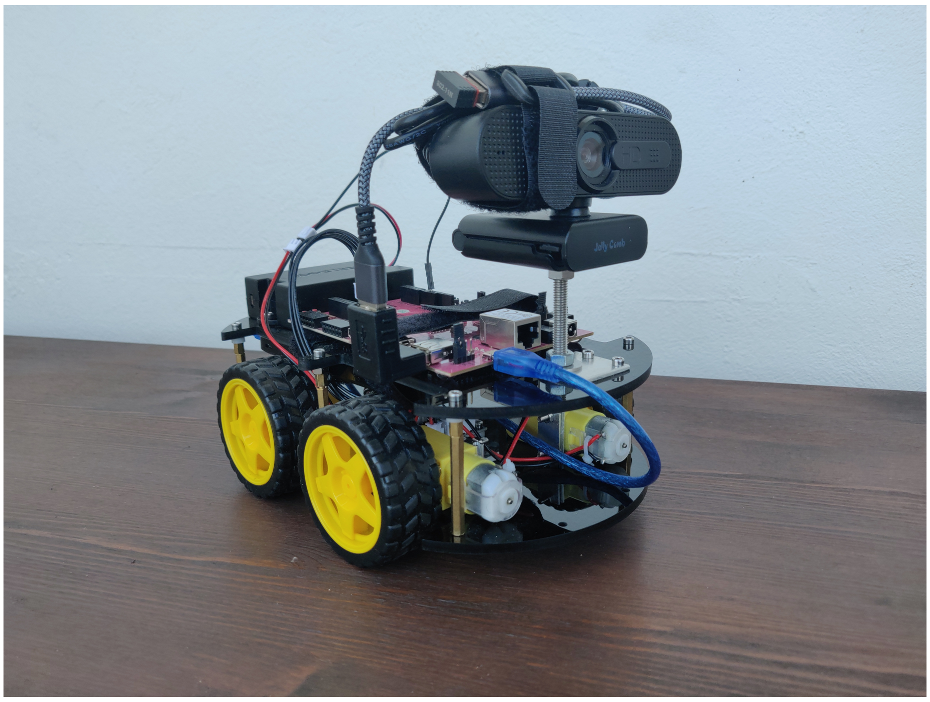
Bremen University of Applied Sciences
Faculty 4: Electrical Engineering and Computer Science
November 2, 2020 – March 31, 2021

Abstract

Autonomy of a moving vehicle requires continuous observations of the environment through a large number of sensors. This leads to a substantial amount of data that has to be processed as close to real time as possible. Artificial neural networks are a popular choice for problems like this, as they can find important patterns in data quickly and at the same time interpret these findings in the required context. Types of networks that use parameters with lower, fixed precision have been shown to require less computational resources and memory, while remaining accurate. Combined with specialised hardware like FPGAs, these networks promise reductions in latency and energy consumption.

Space exploration could benefit hugely from these improvements, as autonomy promises to reduce the effects of signal delays introduced by huge distances between vehicle and operator. More confidence could be put in a space probe's decisions, reducing situations in which time is wasted by waiting for control inputs from an operator. However, as space vehicles are among the most resource-constrained, harsh and expensive computing environments imaginable, it is impractical to add a lot of computing power. Quantized neural networks on FPGA hardware are therefore emerging as one solution to bring autonomy to more space vehicles.

This work presents a real life approximation of an autonomous roving scenario. It uses existing research results into binarized neural networks on FPGA hardware and combines it with actual low-power hardware that approximates what would currently be available to a space mission. The same hardware package also supplies the neural network with image data from a camera and translates its outputs into steering commands. Those are then used in a control scheme for the actual vehicle. The whole setup is able to find and identify a target object, work out its relative location and approach it, even if the target is not stationary. This demonstrates the potential of using FPGAs for complex deep learning inference tasks, reducing the requirements for computing power and energy consumption and therefore the applicability of this research to resource-constrained environments like space exploration.


The proxy vehicle used to simulate a roving scenario

Contents

1 Introduction

written by Jan Bredereke

We report on the practical application of a binarized neural network (BNN) implemented in a field programmable gate array (FPGA); it is used for real-time visual object recognition that supports the autonomous steering of a model car vehicle. This work was undertaken in the context of our research on fast digital circuits for artificial intelligence in space. On-board computers of space vehicles are restricted to comparatively slow hardware because of space radiation. An FPGA can boost the computing power of a neural network while still being radiation-tolerant. The project was part of the course "Embedded Systems" at the City University of Applied Sciences Bremen, supervised by Jan Bredereke. The participants were given the challenge to implement a neural network on an FPGA with computational capabilities that are comparable to an FPGA usable in space. Furthermore, the implementation should be as fast as achievable. They chose an application which also requires autonomous computing and an independent power supply. This application is the aforementioned real-time visual object recognition used for the autonomous steering of a model car vehicle.

In the remainder of this chapter, we describe more details of the research context, of the original research goals of this project and of the application selected. In Chap. 2, we present an overview on the relevant fundamentals of neural networks and their implementation on FPGAs. Chapter 3 describes our design decisions on breaking down the application into subsystems. These are in turn discussed in the following chapters. In Chap. 8, we evaluate the speed gain of our approach. Chapter 9 concludes this report and gives an outlook on future work.

1.1 Research Context: Fast Digital Circuits for Artificial Intelligence in Space

written by Jan Bredereke

On-board computers of space vehicles perform basic control as well as payload data processing. The basic control tasks are crucial for the safety of the craft. A satellite often costs hundreds of millions of Euros, and it must not be lost. Radiation not blocked by Earth's atmosphere would make current processor chips fail soon. This does not happen for special processors with structures at least 65 nm wide. These special processors are sufficiently robust; but their computation power is accordingly much smaller than that of processors manufactured at the current 16 nm or 10 nm scale. An extremely small number of these special processors is produced, only. Therefore, often they are not realized using application specific integrated circuits (ASICs), but by programmable standard hardware (FPGAs). Radiation-hardend versions of some FPGAs are available, featuring a suitable size of their structure (and other measures against radiation effects).

There is an increasing demand for on-board computing power. An example is on-board image processing, for, e.g., autonomous rovers on other celestial bodies or for swarms of small satellites with little bandwidth towards a ground station each. Neural networks are an approach that allows the autonomous classification of of visual objects. However, they require abundant computing power. Therefore, they are often run on particularly potent special hardware. This hardware is definitely unsuitable for the space environment.

Just executing an already trained neural network (i.e., "inference") requires orders of magnitude less computing power than to train the network. By doing the training on ground in advance, at least some image processing applications get into reach. Furthermore, the execution of a neural network inherently is a massively parallel task, and therefore highly suitable for an FPGA. Additionally, a neural network performing an image processing task can tolerate a certain rate of transient failures without damaging the result substantially. Space radiation causes such transient failures. In contrast, a CPU executing sequentially will often be completely thrown off its execution path by a single event upset. The disturbance in a neural network may well remain local and diminish the quality of the result only gradually.

Payload data processing does not put as high a demand on reliability as processing for basic control. Here, radiation-tolerant chips are used, too, instead of radiation-hardend chips. They suffer brief temporary failures, but they provide a comparably higher computing power.

1.2 Original Research Goals of This Project

written by Jan Bredereke

This section describes the original research goals of this project, as proposed by the supervisor. The participants' actual research work done, presented in the remainder of this report, delved into some of the aspects proposed much deeper and more successful than expected. But naturally it could not cover all of the interesting aspects identified before the project, thus leaving room for further work into many directions.

A goal of this project is to explore the feasibility of executing pre-trained neural networks on an FPGA; where the computing power of the FPGA is similar to that of a radiation-tolerant FPGA. The participants shall fathom the trade-off between the computing power necessary and the computing capacity available on a suitable FPGA. Many different approaches for optimization might help. Lowering the reliability of the individual neurons by reducing the structural size below the safe limit, as sketched above, is only one of them. Several further approaches exist for executing pre-trained neural networks on comparatively slow CPUs. This is because neural networks gain increasing importance in the embedded systems area, for example on smart phones.

A preceding project in this area took place in the winter term 2019/20, supervised by Jan Bredereke, too [9]. The participants successfully realized a tool chain for the hardware and software development for executing neural networks on an FPGA. They investigated many different tools which offered interesting approaches. But in the end, most were not sufficiently documented, not sufficiently mature, or unsuitable for the project. Using the tools finally chosen, the participants could demonstrate their tool chain in practice. They did this for the "hello world" of the neural networks, the xor function. They described the neural network in Python, trained it suitably, and then implemented the trained network on an FPGA. This implementation of the neural network turned out to be faster by a factor of 1000, compared to an implementation on a radiation-tolerant microcontroller using Python. A PYNQ-Z1 board serves as the hardware platform. It contains the system-on-chip (SoC) Zynq-7020 comprising a microcontroller and an FPGA Artix-7, coupled to the microcontroller. Suitable libraries facilitate to program the microcontroller in Python substantially. Tensorflow and Keras cover the software side.

The participants of the current project had the opportunity to choose from a range of ideas for their work. The coupling of the digital circuit in the FPGA to the microcontroller was rather simplistic. They could investigate and implement more powerful concepts for bulkier input data which do not fit on the SoC-internal AXI bus in a single clock cycle. Furthermore, the software/hardware interface between the software in Python and the internal AXI bus could be accelerated. Or they could dispose of the software in Python entirely, which pre-processed the input data. In parallel, they should identify and realize suitable more complex neural networks. Visual object recognition lent itself as an application here. As soon as more than the entire FPGA hardware is used for a computational cycle, the computation must be serialized in part. Pipelining should be applied here, in particular. The hardware usage per computational cycle could be optimized, too. This could be done by using only as many neurons as really required, or by reducing the width of the data types used to what is minimally necessary. For the latter, binarized neural networks were a promising approach, for example the framework BNN-PYNQ (FINN) [17].

Other, simpler tasks than classifying images could be approached, too. Good ideas for other autonomous classifying tasks or for autonomous control tasks on board of satellites or rovers would have their own merit as a project result.

A possible task with less input signal bandwidth would be the distinction of a few short audio signals. Potential applications of (substantially) extended versions of the audio signal detection would be on-board voice recognition [10] and the detection of critical changes in the mechanical structure of a satellite using a microphone. For example, these could be an imminent failure of a motor for pointing the solar array towards the sun, an imminent failure of a gyroscope (a spinning device for attitude control), the impact of a micro-meteorite, etc. However, there are less off-the-shelf solutions for neural networks for audio signal recognition than for visual object recognition.

1.3 The Selected Application: an Autonomous Vehicle

written by Niklas Krekel

The project team decided on building an autonomous model car in order to have a real-life proxy for an autonomous spacecraft. When sending probes away from Earth, autonomy in navigation becomes increasingly more important as the time delay to an earth-based 'decision maker' also increases. Intelligent steering of those spacecraft is therefore an area of ever increasing interest in research and innovation, while also being easily approximated on Earth.

For this project, the autonomy of the model vehicle was decided to be informed by object recognition. In comparison to other machine learning methods, computer vision is well established in research and can be applied to tasks with greatly varying complexities. This simplifies the selection of a realistic scenario that is useful to evaluate and at the same time aims to exhaust the performance of the FPGA in use. Object recognition is also trivial to test in a real environment while requiring relatively high data throughput, further exhausting hardware performance limits. The last point becomes especially important under real time conditions, which often apply to space exploration. Continuous on-board decision making in real time is another objective of this research undertaking. The faster any autonomous process can react to changes in the environment, the higher is the chance of avoiding adverse situations.

In summary, this project aims to approximate an autonomous spaceflight scenario by building a model vehicle that uses computer vision to detect a target object, translate the target's position into actionable maneuvering inputs and approach it, while continuously monitoring the environment and recalculating steering commands along the way.

With the overall goal in mind, the project's challenges were expected to be:

- Implementing or choosing a neural network model which capitalizes on the specialized FPGA hardware
- Setting up a processing pipeline from sensor to inference
- Interpreting the inference results and translating them to maneuvering inputs
- Building a vehicular platform and integrating it with sensor and processing hardware

2 Fundamentals

This section explains the fundamental principles of artificial neural networks, followed by the main differences of binarized neural networks. It also introduces the software framework and specific hardware used for their implementation.

2.1 Basic Principles of Neural Networks

written by Raffael Kaehn

To provide a base line for further explanations, this subchapter gives an overview of the elementary principles of neural networks, builds upon this to introduce specialized networks for image recognition and finishes with an explanation of the optimization process.

2.1.1 The Single-Layer Perceptron

Generally speaking, an artificial neural network can be understood as a mathematical function that receives an input vector, processes it in combination with internal parameters, and deterministically produces an output vector. Both the input and output vector are of finite, constant length and contain elements that are required to be within a predetermined range of values. To allow for a more intuitive and understandable representation, neural networks are often depicted as directed graphs, where each input vector element appears as a node in one layer, each output vector element appears as a node in another layer and relationships in calculations appear as edges between nodes of different layers:

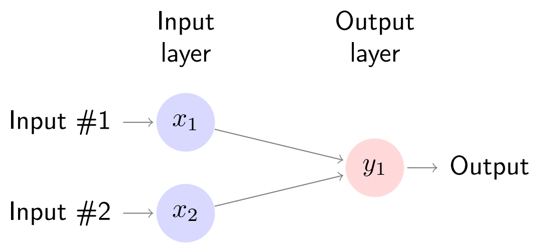


Figure 1: Basic single-layer perceptron

In the most basic neural network, called a perceptron, the value of an output node can be determined by first calculating the dot product between the values of the input nodes and those of the connecting edges. Intuitively, the latter can be thought of as a weight, or a measure of influence, of the previous nodes value in the calculation of the current nodes value. Since the result of the dot product is linearly dependent on the input values, the following problems arise:

- Depending on the respective weights, the distribution of the output values of two or more nodes of the same layer might be drastically different. This could become especially problematic when the output of one node is used as an input of another node.
- A linear function that takes the output of another linear function as an input can always be substituted for a single linear function. This property would make it impossible to approximate non-linear functions, even in multi-layer neural networks.

To solve the first problem, a bias is added to the dot product, allowing the mean of the distributions to be shifted in a positive or negative direction. To solve the second problem, the resulting sum of the dot product and the bias is given as an argument to an activation function. This is typically a non-linear function used to clamp or redistribute the values into a specific, predictable range. After the activation function has been applied, the calculated result as given by equation (1) is set as the value of the output node. The process is repeated for every node of the output layer.

$$y_j = g(a_j),\ a_j = \sum_i w_{ji} x_i + b \tag{1}$$

2.1.2 Multi-Layer Architectures

For most real-world applications, the described single-layer perceptron is not sophisticated enough to approximate complex dependencies in the processed data and therefore cannot produce reliable predictions for non-trivial input values. To combat this problem, it has proven useful to integrate additional layers between the input and output nodes of the network. These layers can be interpreted as solving smaller subproblems of an overarching classification or regression task. Since the intermediate results they produce are typically not presented to the user, they are referred to as *hidden layers*. The inner workings of these layers can range from identical to the principles explained above to completely unrelated, specialized functions, with the most relevant variants explained hereafter:

- **Fully connected layers** – These layers can be seen as a generalization of the output layer of the perceptron described above. In this type of layer, the value of each of its nodes is influenced by the weighted value of *every* node from the previous layer. In image recognition tasks, one or more of these layers are often located right before the output layer and fulfill the task of actually classifying data that was processed and normalized in previous layers of the following types.
- **Convolutional layers** – This layer type is easiest to understand by looking at its typical use case, as part of a neural network to recognize and classify objects in images. In this context, these layers can be thought of as applying *filters* to different regions of an image, where the dimensions of the filter are constant within a layer and the coordinates of the region are different and advancing for every node of that layer. This filtering mechanism is implemented by restricting the influence of nodes in the previous layer on the value of one node in the next layer to small and changing subsets with their respective weights in the calculation shared for every node in the next layer.
- **Pooling layers** – This layer type is most often found in direct succession to a convolutional layer and used to reduce the dimensionality of the previous layer. This dimensionality reduction can be achieved by separating the image into multiple non-overlapping square tiles with a side length greater than one and combining the color information of the fragments within that tile into a single figure, often either by calculating an average of the included values or by determining the maximum value.

For image recognition tasks, the three above-mentioned layer types are often used in combination, where the image is first processed in a series of generalization stages, each consisting of a convolutional and pooling layer and meant to extract or emphasize certain elements in the data. These elements are then, layer by layer, combined into larger patterns. When the extraction of general elements and combination of elements into patters and the combinations of patters into objects has been completed, fully-connected layers are used for the actual classification. The exact configuration of these layers, called topology of the neural network, can and should vary depending on the problem to be solved, but the described principles and the reasoning behind them can be applied to problems in many domains.

2.1.3 Optimization

After the general architecture has been laid out, the network is able to accept input data, use that data and the internal parameters to calculate the values of the nodes in the next layer, and repeat this operation until the output layer is reached. The process of producing an output vector as a prediction for a given input vector is called *inference*.

Directly after initialization, during which the weights and biases are often set to random values drawn from a normal distribution, these predictions from the network will likely not be close to the expected results. To calculate a concrete numeric value for these deviations, the predicted and expected output vectors are processed by a so-called *loss function*. This function could be implemented as a simple sum of squared errors (SSE), consisting of the following steps:

1. Calculate the element-wise difference between the predicted and expected vector, this is referred to as the *error*.
2. Square each element, resulting in a value that is independent of the direction (positive/negative) in which the prediction diverged from the expected value.
3. Calculate the sum of all elements.

The resulting figure, called the *loss* of the network for a given input vector, is a direct measure of its performance, with lower values indicating more accurate predictions. When looking at a fixed pairs of input and expected output vectors, the loss is only dependent on the internal weights and biases and as such, the goal of the optimization process explained below is to find values for these parameters that *minimize* the loss across the data set.

The intuitive solution to this task would be to calculate the derivative of the loss function, set it equal to zero and solve for the weights to find the functions stationary points. These weights would then be inserted into the second derivative where, if the value of that function at the given input was greater than zero, a minimum of the loss function would have been found. Although this method works for very simple neural networks, solving equations with the high number of variables found in complex networks is too computationally intensive. For these cases, a method called *gradient descent* is used, where instead of trying to find a globally optimal set of weights to minimize the loss function, a gradual, step-by-step descent is being made towards a local minimum. Each gradient descent step consists of the following tasks:

1. Calculate the gradient of the loss function E with respect to the weights. The term gradient refers to a vector of the functions partial derivatives for each weight. For the single-layer perceptron described above, the elements of this vector would be calculated by applying the chain rule for differentiation as follows:
$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}} \tag{2}$$
After inserting equation (1), the factor on the far right can be solved like this:
$$\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_k w_{jk} x_k + b = \frac{\partial}{\partial w_{ji}} w_{ji} x_i = x_i \tag{3}$$
The factor in the middle describes the derivative of the activation function:
$$\frac{\partial y_j}{\partial a_j} = \frac{\partial}{\partial a_j} g(a_j) = g'(a_j) \tag{4}$$
When using the described SSE loss function and with $\hat{y_j}$ as the expected output of a neuron, the left factor becomes:
$$\frac{\partial E}{\partial y_j} = \frac{\partial}{\partial y_j} (y_j - \hat{y_j})^2 = 2(y_j - \hat{y_j}) \tag{5}$$
As such, the complete gradient can be written as:
$$\begin{bmatrix} \frac{\partial E}{\partial w_{11}} \\ \frac{\partial E}{\partial w_{12}} \\ \frac{\partial E}{\partial b} \end{bmatrix} = \begin{bmatrix} 2(y_j - \hat{y_j}) \cdot g'(a_j) \cdot x_1 \\ 2(y_j - \hat{y_j}) \cdot g'(a_j) \cdot x_2 \\ 2(y_j - \hat{y_j}) \cdot g'(a_j) \end{bmatrix} \tag{6}$$

2. The calculated gradient is then used as a delta by which the weights are modified:
$$\Delta w_{ji} = \frac{\partial E}{\partial w_{ji}} \tag{7}$$
To control how much the weights should be changed, a *learning rate* α is introduced. If this value is near its recommended upper bound of 1, the weights are adjusted in large steps towards their optimum value, but a chance of overshooting a local minimum is introduced. If the value is near its recommended lower bound of 0, the adjustments are made in small steps, potentially resulting in a slow optimization process.
$$w_{ji} = w_{ji} - \alpha \Delta w_{ji} \tag{8}$$

2.2 Binarized Neural Networks

written by Raffael Kaehn

Building upon an understanding of traditional neural networks, this subchapter introduces *binarized* neural networks, a specialized form that enables further optimization possibilities when implemented on FPGAs.

2.2.1 Motivation

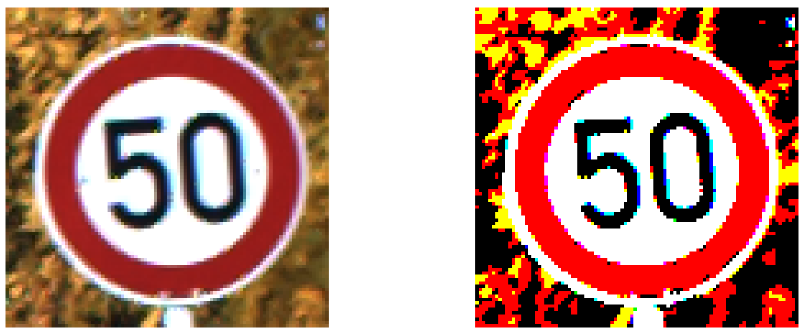
One potential problem with the previously introduced network topology for image recognition is their dependence on a relatively high number of layers and consequently many internal parameters to produce reliably accurate results. Typically, these parameters as well as all input values, intermediate results and output values are represented as floating point numbers, requiring significant amounts of memory and computational power. Recent research results have shown that similarly accurate results can be obtained with lower precision numbers, as long as this is compensated by a higher number of nodes per layer. This type of neural network, called *quantized neural network*, in which the input values, internal parameters and output values are represented by lower precision numbers, have the advantage that fewer processor cycles are needed to compute the values of the nodes and less memory is needed to store the intermediate results and parameters.
If the precision of these values is reduced even further to the point where each number is represented by only a single bit, the resulting network is called a *binarized neural network*. Building upon the advantages mentioned above, these networks show immense promise when implemented on FPGAs, since many of the calculations during inference can be implemented as simple binary operations and the lower memory requirements play nicely into the limited availability of such resources on FPGAs. [17]

2.2.2 Differences in the Inference Process

Since the precision of the values in the calculations has been reduced to a single bit, the range of values that can be represented is drastically reduced. In the literature, the alphabet $\{-1, 1\}$ is used for all calculations, in which a bit set to 0 corresponds to the value -1 and a bit set to 1 corresponds to the value 1 [3]. The conversion from a real number to a value from this alphabet is done by a quantization function, such as the following:

$$Q(x) = \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{9}$$

This quantization function may also be applied to the input values. Depending on the application, this could mean, for example, that the color values of a grayscale image would be binarized in a way where each pixel is either black or white, before then mapping these values to the alphabet $\{-1, 1\}$. For a color image, a division of each pixel into its subpixels (red, green, and blue) with subsequent binarization is conceivable. In this constellation the three primary colors, the three additive secondary colors, white, and black could be represented. An image converted using this method can be seen below:



(a) 256 possible values per subpixel [16] (b) 2 possible values per subpixel

Figure 2: A potential input image and its binarized counterpart

To show the complete inference process for these input values, it makes sense to revisit the single-layer perceptron from the beginning of this chapter. As can be seen in equation (10), calculating a_j as the dot product of the quantized input vector x and the weight vector w_j is no different from a conventional floating point network:

$$a_j = \sum_i w_{ji} x_i + b \tag{10}$$

However, since the result is an integer and thus not necessarily an element of the expected set of values $\{-1, 1\}$, the quantization function introduced above is then applied as an activation function, reestablishing the binarization conditions:

$$y_j = g(a_j),\ g(\cdot) = Q(\cdot) \tag{11}$$

2.2.3 Differences in the Optimization Process

The optimization process *gradient descent* is based on the principle that the weights are adjusted in small steps to reach a local minimum of the error function. However, since the weights of a binarized neural network can only have the values -1 and 1, it is not possible for them to be adjusted in small increments. The common solution to this problem is to store the weights of the network as real-valued floating point numbers, meaning they can have values excluded from the set $\{-1, 1\}$, during training. Only when these weights are used to calculate a prediction y are they binarized using the quantization function. This results in the following equation for calculating a prediction:

$$y_j = g(a_j),\ a_j = \sum_i Q(w_{ji}) x_i + b \tag{12}$$

This prediction would now be used to calculate the gradient of the error function with respect to the real-valued weights as such:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}} = 2(y_j - \hat{y_j}) \cdot g'(a_j) \cdot Q'(w_{ji}) x_i \tag{13}$$

Here, two new problems arise:

- The quantization function used in the binarization of the real-valued weights is not continuously differentiable. While it is at least semi-differentiable, its left or right derivatives are zero for every input, which would result in the elements of the gradient always being zero as well. To solve this problem, the so-called *straight-through estimator* is used, which sets the gradient of the error function with respect to the real-valued weights equal to its gradient with respect to the binarized weights:

  $$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial Q(w_{ji})} \tag{14}$$

  This results in the following partial derivative:

  $$\frac{\partial a_j}{\partial Q(w_{ji})} = x_i \tag{15}$$

- The quantization function is also used as the activation function $g(a)$, whose derivative is also part of the gradient. In this case, it is exchanged for another, continuously differentiable function. Due to the similar shape and bounds of the function values, the $tanh$ function is a suitable choice. This has the consequence that the output of a node is no longer limited to the alphabet $\{-1, 1\}$ with, in the case of the $tanh$ function, intermediate values also being possible.

With these two adjustments, the following complete equation for the calculation of the gradient of the error function is obtained:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial Q(w_{ji})} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial Q(w_{ji})} = 2(y_j - \hat{y_j}) \cdot g'(a_j) \cdot x_i \tag{16}$$

At the end of the training process, the calculated gradient is used as a delta to adjust the real-valued weights, similar to the process explained in the first subchapter. At the point where the training has been completed, the real-valued weights are permanently quantized and stored for future use during inference.

2.3 FINN-Framework

written by Niklas Krekel

FINN is a framework enabling the construction of quantized neural networks to be used specifically for inference on FPGA hardware. It was conceived by researchers at Xilinx Research Labs and is now an open source tool officially maintained by Xilinx, a leading company in the production of specialised computing hardware [6]. As explained in subsubsection 2.2.1, BNNs have been shown to be able to retain a high classification accuracy compared to unquantised networks, if the number of nodes per layer is increased. This opens up the possibility of capitalising on the high performance of FPGAs regarding binary operations. Due to the smaller memory requirements of binarized parameters, it is possible to keep them in on-chip memory, reducing off-chip memory access and therefore latency. Owing to these optimisations, BNNs on FPGA hardware are expected to offer a performance benefit over floating point networks without compromising on accuracy. [17]

To actually implement a binarized neural network in hardware, FINN does not require the user to be familiar with hardware description languages. Instead, models can be constructed with familiar and popular frameworks like PyTorch, and optimised for the problem at hand using quantization-aware training. The resulting layer model is then supplied to FINN, which streamlines and converts it into C++ code. This code can be synthesized and implemented into a bitfile for a specific target device and afterwards deployed on said hardware. [6, page 'End-to-End Flow']

In order to be able to convert a network model into synthesizable code, FINN organises it in a heterogeneous streaming architecture. For each layer, there is a separate compute engine tailored to the specific requirements of that layer. When a compute engine starts producing output, this is transmitted through on-chip data streams to the following compute engine, which can already start its computations while the one before it finishes up, effectively overlapping computation and communication and reducing latency. Since the BNN's parameters can also be stored on-chip, the amount of off-chip memory access is minimised, further reducing latency, i.e. the time spent classifying one input. [17]

2.4 Digilent Pynq-Z1

written by Felix Müller

The Digilent Pynq-Z1 is a development board for embedded systems with an ARM Cortex-A9 CPU (the „**P**rocessing **S**ystem“) and a FPGA equivalent to an Xilinx Artix-7 (the „**P**rogrammable **L**ogic“) inside one System-on-a-Chip (SoC) called Zynq. Both the CPU and FPGA have access to 512 MB of DDR3 memory through a memory controller with 8 DMA channels and 4 high performance AXI3 slave ports [4]. To lower the barrier of entry into embedded FPGA designs, the board is intended to be used with Pynq which is an open source tool chain that allows the use of Python to access high level abstractions of hardware interfaces. However, every layer of abstraction can be replaced by lower level implementations if higher performance and thus more control is required [14].

2.5 AXI4

written by Felix Müller

This project requires multiple peripherals to communicate with each other (e.g. the PS with the neural network on the PL). This is achieved by using an AXI4-bus to which all peripherals are connected. AXI belongs to the ARM AMBA family of micro controller buses and is officially supported by Xilinx in all it's tools (e.g. Vivado). The latest supported version is AXI4, which provides 3 types of interfaces [2]:

- **AXI4:** „for high-perfomance memory-mapped requirements.“
- **AXI4-Lite:** „for simple, low-throughput memory-mapped communication (for example, to and from control and status registers).“
- **AXI4-Stream:** „for high-speed streaming data.“

The fixed hardware inside the Zynq SoC only implements the AXI3 standard which is resolved by using a so called "Interconnect" to translate between the two versions of the standard.

3 Breakdown Into Subsystems

Multiple steps are required to get from a video signal to the automated steering of the vehicle, which are realized by separate subsystems that interface with each other. First an overview of all subsystems is given followed by the list of protocols for communications between the systems.

3.1 Subsystems

written by Felix Müller

This project consists of 4 subsystems in total, that are illustrated in Figure 3. Every rectangle represents one subsystem that is connected to at least one other subsystem (illustrated with an arrow). The direction of the arrow indicates the direction and the labelling the protocol of the data flow. The physical location of the subsystems is shown by the bigger boxes that group the subsystems. The group "Vehicle" contains all subsystems located on the actual vehicle that are required for operation. The group "Base Station" is situated at another (optional) physical location and runs on a any personal computer with network access.

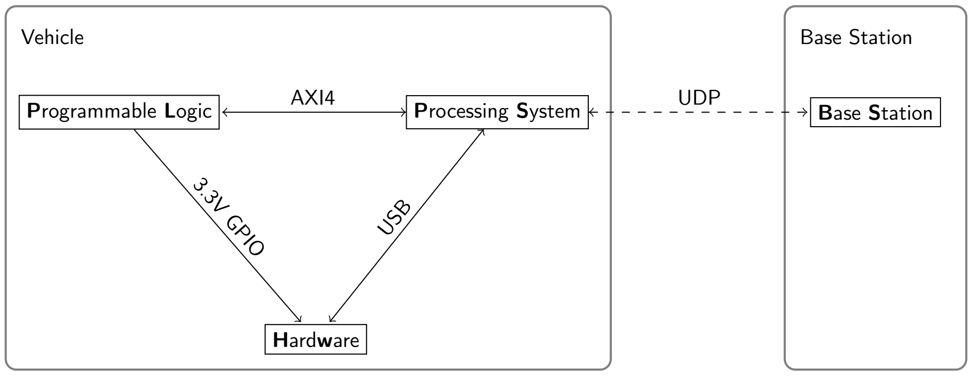


Figure 3: Combined System

The aforementioned subsystems are interconnected into a pipeline (illustrated in Figure 4) which takes the raw image data from the camera and processes it until it is reduced to mere control signals for the motors. The following descriptions should give a coarse overview of the pipeline whereas each individual step is explained in detail in the chapter of the matching subsystem (marked by the lower part of the nodes in the pipeline).

- Camera: The camera captures the image data with a sensor and delivers it to the PS over USB.
- Scaling: The native resolution of the camera is too high for the available processing power, therefore it is scaled down before any further computations.
- Tiling: Because the used neural network is limited to images with a resolution of 32x32 pixel, the captured camera frame needs to be cut up into several tiles.
- Inference: This is the heart of the entire pipeline. Every generated tile is inferred with a confidence for each of the available classes in the neural network.
- Analysis: The results of the individual classifications are reduced to the tiles which contain the object in question. Afterwards a position relative to the image center is generated based on the tiles that contained the object in question with the highest confidences.
- PID: Depending on the presence of the object, the PID controller either generates a signal to drive and steer towards this object or enters a search mode, where it rotates around it's own axis until it finds a matching object.
- Motor Driver: Generates the analog signals based on the values given by the PID controller.
- Motors: The physical motors that propell the vehicle forward according to the given signals by the motor driver.

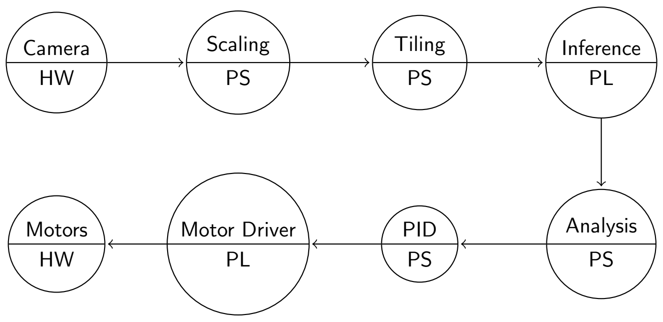


Figure 4: Data Pipeline

3.2 Protocols

This section will give an overview over the protocols that are used for communication between the subsystems and why they were selected. The actual implementation and usage of the protocol will be explained in the subsystem where it is first used.

3.2.1 Between Processing System and Programmable Logic

The AXI4 protocol was used between PS and PL because it provides the highest bandwidth and flexibility for the SoC on the Pynq-Z1 board and is the intended method by the chip manufacturer Xilinx. Two of the available interfaces provided by the AXI4 standard were used in this project:

- AXI4-Lite: For the configuration of the neural network (weights & thresholds) and for motor control (direction & speed).
- AXI4: For high speed memory mapped transfers of the image data to the neural network and the resulting inference results back to the CPU.

3.2.2 Between Processing System and Base Station

The BS has a different physical location than the vehicle (see Figure 3) therefore a tethered connection is not applicable because it would severely limit the vehicle in its movement.

To keep the implementation portable, the User Datagram Protocol (UDP) was used. This protocol was extended by putting so-called frames into the datagram that follow a very simple structure. The header of the frame contains only a single byte which is the identifier. The structure of the body is determined by each identifier.

3.2.3 Between Programmable Logic and Hardware

The communication between the PL and the HW is different to the other protocols since it is the only one-way data transfer. This is done by using the 3.3V logic level GPIO pins of the PL to control the driver circuitry of the motors. This circuitry provides no feedback which leaves the input functionality of the GPIO pins unused.

3.2.4 Between Processing System and Hardware

The camera of the hardware system is connected to the PS over USB using the video device class. Because the Linux kernel provides generic drivers for this use case, it does not need further explanation.

4 Design of "Hardware" Subsystem

This sections describe the components that make up the physical hardware of the vehicle. These components consist of a base model car, a camera and several miscellaneous peripherals.

4.1 Vehicle Platform

written by Niklas Krekel

In order to simulate a simplified roving scenario, a toy vehicle is used as the mobile platform for the system. It features four DC motors to which the wheels attach directly. Two motors on each side of the vehicle are controlled together, steering is possible by varying the speed of the motor groups independently. This control scheme is implemented through the included motor controller board. A battery compartment, including two 18650 Lithium-Ion batteries, a charging port and power switch, supplies the motors. Two acrylic plates stacked onto of each other with metal spacers make up the vehicle chassis. On the lower plate, the motors and the controller are mounted. The upper plate carries the battery box. A schematic representation of the base vehicle is included in Figure 5.

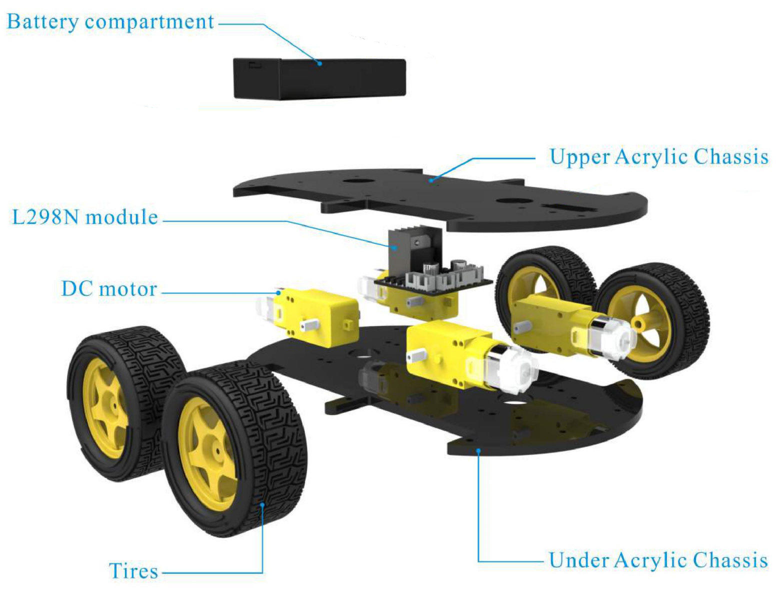


Figure 5: Vehicle platform schematic [5, file 'ELEGOO Smart Robot Car Kit Parts and Names.png']

4.2 Camera

written by Felix Müller

A USB webcam with a maximum resolution of 1920x1080 pixels and a frame rate of 30 fps was used. The only selection criteria were a tripod mount for easy attachment to the vehicle and a narrow viewing angle for maximum detail of objects at a given resolution.

4.3 Additional Peripherals

written by Niklas Krekel

The Pynq-Z1 board is connected to the camera mentioned in subsection 4.2 and these additional peripherals:

1. Power bank
   10400 mAh Lithium Polymer power bank with USB-A outputs capable of supplying up to 3A at 5V. Used to power the Pynq-Z1 board and its peripherals.

2. USB angle and two-way splitter cable

3. WiFi adapter
   USB2.0 WiFi adapter for connecting the Pynq-Z1 to a 2.4GHz wireless network through standard 802.11b/g/n.

4.4 Combined Vehicle

written by Niklas Krekel

The power bank is mounted to the base vehicle by extending the spacer rods between the acrylic plates with stand-off screws and strapping it to the bottom of the upper plate. A screw for mounting the camera is attached to the front of this plate, and the Pynq-Z1 board is strapped to it between the camera screw and the battery compartment. Figure 6 shows the complete vehicle.

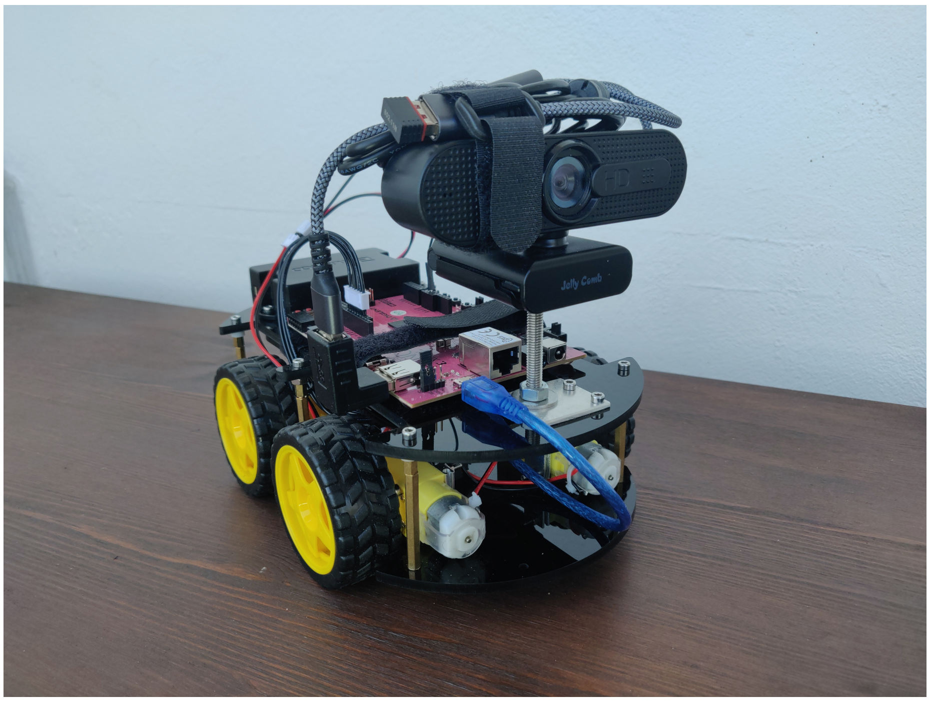


Figure 6: Fully assembled vehicle

5 Design of "Programmable Logic" Subsystem

written by Felix Müller

This section describes the design of the hardware running on the FPGA aka Programmable Logic of the whole vehicle system. There are multiple components in the design: the neural network (BNN-PYNQ), the motor controller and several miscellaneous components required to connect the components. A block design was used a as a visual way to connect the components which can be seen in Figure 14.

5.1 Neural Network (BNN-PYNQ)

The BNN-PYNQ was used as the hardware implementation of a neural network without any modifications. It was compiled from the sources in the Github repository with the following configuration parameters:

- Target Platform: Digilent Pynq-Z1
- Neural Network Topology: CNV (a convolutional network)
- Weight Bits: 1
- Activation Bits: 1

The evaluation by the original researches already measured the performance in their paper [17] for a standalone usage of the network. In the vehicle however it is not the only logic on the PL and bound to realtime constraints properly control the vehicle. Therefore a compromise between hardware usage and throughput needs to be found that is compatible with the other constraints of the vehicle. One such compromise are the chosen parameters as they achieve a respectable precision at a tolerable cost in resources according to the measurements in the paper. Further research might be necessary to evaluate if this is true in practice.

The result of the compilation is an IP-Core which can be dropped into a block design in Vivado. The block can then be treated as a black box with interfaces at its boundaries. The most notable interfaces are the AXI4 ports which are used for the configuration and data transfer of the image data and the classification results. The following subsections will give a coarse overview for the most most important aspects of these interfaces.

5.1.1 AXI4-Lite

This interface is used to configure the network, trigger one or multiple inferences or retrieve it's status (e.g. inference complete). Communication is achieved over registers that can be accessed at an offset from the base address of this peripheral. The most important registers are the control register (table B.2) and the input/output data address register (table B.3 & B.4). The registers to load the weights and thresholds were intentionally left out because they would require too much room in this report and are already documented by the original research group.

5.1.2 AXI4

This interface is used to retrieve image data to classify and to store the results. Because this interface implements a full AXI4 interface, the BNN-PYNQ has direct access to the DDR3 memory. The addresses of the data can be configured via AXI4-Lite as described in table B.3 and B.4. The expected format of the input data is Red-Green-Blue (RGB). Each color channel is described as a 8-bit fixed point number therefore one pixel requires three bytes. For the input resolution of 32x32 pixel this means that each image requires 3072 bytes (32 per row x 32 rows x 3 color channels per pixel). Several images can be stored consecutively in memory. The amount of images that the BNN-PYNQ needs to process is configured via AXI4-Lite as described in table B.5. The output format of the inference results are 16-bit integers that represent the confidence of a class. For each image 64 of these confidences are written independently of the available class from a trained parameter set. Therefore 128 bytes are required for the results of a classified image. A multiple of these blocks will be stored consecutively depending on the amount of images classified.

5.2 Motor Controller

The motor controller generates the signals to address the L298N motor driver and was implemented in pure VHDL. It receives the desired speed and direction via AXI4-Lite registers and generates the appropriate signals on the GPIO pins connected to the analog driver circuitry.

5.2.1 AXI4-Lite Interface

Only 24-bit are required to address the L298N motor driver, but because there is a minimum of 4 registers for AXI4-Lite, three separate registers were used to keep the addressing simple from the PS. The register width of 32-bit was chosen to stay consistent with the other used peripherals on the AXI4-bus and therefore require less interconnects. The structure of the implemented registers can be seen in Table B.1. The left/right speed settings control the speed speed of both motors on either side with a value from 0 to 1023. The state of the bits IN1 to IN4 determine the rotation direction of the motors. Table 1 gives an overview of the most important combinations.

IN4	IN3	IN2	IN1	Description
0	0	0	0	Stop all motors
0	1	1	0	Forward
1	0	0	1	Backward
1	0	1	0	Stationary Rotation (Left)
0	1	0	1	Stationary Rotation (Right)

Table 1: Motor Direction Control

5.2.2 GPIO Interface

The L298N motor driver is connected to 6 GPIO and the ground pin on the Pynq-Z1 board. The driver gets its power from an external battery and therefore only control signals are shared between Pynq-Z1 and the motor driver. The mapping of the GPIO pins is described in table 2. All GPIO pins are used digital signals with a logic level of 0V or 3.3V. The motor enable pins are additionally addressed by applying a pulse width modulation to the signal. The period of the PWM signal is 1.024 milliseconds or 976.6 Hz to keep the losses from power switching to a minimum. A disadvantage of the low frequency is that the switching is quite loud, because the frequency is still in the audible range thus omitting a high pitched hum from the motor driver. To reduce this noise, the frequency could be increased to 15 kHz, which is an appropriate value according to the application nodes from the manufacturer of the motor driver, but this would require a calculation if the switching time is still low enough based on the resistance of the connected motors [1]. The resolution of the PWM signal is 10-bit, where a duty cycle of 0% (0) is the slowest and 100% (1023) the highest speed. The actual slowest speed is at a duty cycle of around 40%. This value was determined by empirical testing with an observation at which duty cycle the motors start moving (without load). Higher values are possible based on the actual weight of the vehicle.

Pynq-Z1	L298N	Description
GND	GND	Ground
J3-IO39	ENA	Motor Enable A
J3-IO38	IN1	Motor In 1
J3-IO37	IN2	Motor In 2
J3-IO36	IN3	Motor In 3
J3-IO35	IN4	Motor In 4
J3-IO34	ENB	Motor Enable B

Table 2: Connections between Motor driver (L298N) and Pynq-Z1

6 Design of "Processing System" Subsystem

This sections describes the design of the software for the ARM-v9 CPU of the SoC. First thich reasons for the chosen toolchain are described which is followed by descriptions for the implemented algorithms (e.g. for tiling or the PID controller).

6.1 Toolchain

written by Felix Müller

For the processing of the inference results and data transmission to the base station on the PS several operating systems and toolchains were considered:

- Toolchain: Xilinx Vitis
  - OS: Standalone
  - OS: FreeRTOS
  - OS: PetaLinux
- Toolchain: GCC
  - OS: PynqLinux (Ubuntu 18.04, the default OS shipped with the board)

6.1.1 Xilinx Vitis

This toolchain is called a *unified platform* by Xilinx and was launched in 2019 [18]. It combines several previously separate tools into one. The important bit of this platform is their embedded software integrated software environment based on the Eclipse IDE called *Software Platform*, which contains several useful tools and conveniences, such as an debugger or syntax highlighting. There are also some additions tailored for embedded development, such as a serial monitor. Projects are split into platform and application projects. The platform contains the underlying operating system, information about the hardware on the PL (e.g. address mappings of peripherals on a AXI4-bus) and optionally a bitstream for the PL. The application is based on the platform and contains the actual running application [19]. This tight integration with the hardware design has the advantage that several operations are done automatically. This includes the automatic generation of header files for the addresses of peripherals on the AXI4-bus or direct compatibility with included libraries (e.g. for a lightweight TCP/IP stack or FAT32-filesystem support for SD cards).

Depending on the selected operating system for the platform there are 3 common options to put the application on the board:

- JTAG (programmed over USB)
- Quad-SPI flash memory (programmed over USB)
- Boot Image (copied to a micro SD card which must be put in the slot on the board)

The first option (JTAG) is the fastest to execute but volatile (will be lost after power-cycle), whereas the last option requires the most manual work because the boot image has to be copied manually. The flash memory option survives power-cycles and can be programmed over USB but is limited to 16MB of memory [4].

Standalone

In this mode of operation the application runs almost bare-metal as there is only a very thin abstraction layer to the actual hardware. Therefore this approach would result in the most control and best utilization of processing power. Typically leaving out an operating system would require extra work to implement the functions that are normally provided by an OS but luckily almost all these functions can be replaced by low-level libraries provided by Xilinx that are included in a default Vitis installation.

FreeRTOS

This is a small real-time operating system for microcontrollers and therefore very lightweight. The biggest advantage over the standalone mode would be the included methods to deal with several threads and the synchronization between them (e.g. with semaphores). Additionally it provides schedulers to queue jobs based on prepared tasks [7].

Petalinux

PetaLinux is a SDK to create a Linux distribution that is customized for a particular hardware setup on embedded systems utilizing FPGAs for their programmable logic [13].

6.1.2 GCC

The GNU Compiler Collection is, as the name suggests, a collection of compilers for several programming languages (including C and C++), which is not nearly as full-featured as the Vitis toolchain (whre the compiler is just a small part in chain) but is far more portable and independent from Xilinx products because of this. For this reason it is possible to target almost any operating system, including the official Pynq distribution, even if it is not targeting specific hardware or event embedded systems [8].

PynqLinux (Ubuntu 18.04)

This Linux distribution is based on Ubuntu 18.04 and part of the Pynq ecosystem labeled *Python Productivity* by Xilinx because it provides interfaces to allow an easy entry into embedded design and specifically systems consisting of a a processor and a programmable logic. This is achieved by utilizing the programming language together with Jupyter notebooks for interactive programming and rapid prototyping. Still, the heart of this distribution is a fully featured Linux kernel with special kernel drivers for the included hardware of the supported boards (e.g. DMA) [14]. Because it is a complete kernel with many included drivers, it is also possible to make use of a USB WiFi adapter to remotely update and debug the vehicle while it is untethered. The disadvantage against a lightweight kernel is the higher power consumption, amplified by the additional running processes (e.g. for the webserver of the Jupyter notebook).

6.1.3 Selection

The GCC toolchain on PynqLinux was chosen, because it allowed the project members to work remotely and provides a complete network and USB stack which made the integration of peripherals such as a webcam or a WiFi dongle trivial. Furthermore a disadvantage of all options belonging to the Vitis platform is a requirement for special hardware to work remotely.

6.2 Programming Language

The main consideration for the programming language of the application running on the PS was speed to reduce the bottleneck compared to the hardware accelerated components of the system. The requirements for the software are simple enough to put all the functionality in one monolithic application, therefore it was decided that a single language should suffice for the final application. The natural approach would have been to use Python because this is the way that was intended by Xilinx and Digilent for the Pynq platform. This however was not viable because Python as a interpreted language is way to slow for realtime image processing on an 650 Mhz ARM chip. Thus we decided to stay as close to the hardware as possible and used C/C++ for the PS application.

6.3 Vehicle Configuration

written by Niklas Krekel

Vehicle configuration parameters are specified in the file "vehicle.conf", where the available parameters are also documented. This file is loaded through the simple library included in *dotenv.h*. While commented and empty lines are ignored, lines that have a key-value-pair in the form *key=value* are loaded as environment variables for the application. The value of a known key can be retrieved by *std::getenv("key")*, and is returned as string. In order to interpret an environment variable as a different data type, the wrapper functions *env::getenv_int, env::getenv_uint32* and *env::getenv_double* are provided. Further annotations are provided as comments in the configuration file. In the following sections all relevant parameters will be referenced and explained.

6.4 Positioning With Tiling

written by Pablo Navarro

The classifier infers an image containing $32 \times 32 \times 3$ values at a time, given by network architecture and denoted as $t_{res,w} \times t_{res,h} \times n_{channel}$ in the following. This resolution is sufficient for classifying which prominent object is present in an image. For targeting an object it is necessary to determine its presence and position in the frame. Therefore, the image is divided into several overlapping tiles of different dimensions which are then being scaled to a final size of $t_{res,w} \times t_{res,h}$ pixel. This size can be modified by parameters `RESIZED_TILE_W` and `RESIZED_TILE_H` e.g. if a different classifier is used. Two methods for generating tiles are being provided. Setting parameter `TILING_FN` to value 0 selects the method described in subsubsection 6.4.2. Choosing the value 1 selects the method described in subsubsection 6.4.3. Furthermore, all values fed into the neural network must be of 8-bit fixed point arithmetic. Therefore a conversion from 8-bit unsigned integer, as provided by the frame-grabber, to fixed point is required for all pixels and all color channels. A look up table of size $2^8 = 256$ and type 8-bit unsigned integer is precomputed at initialization time and stored in memory to save expensive computation time at each loop.

6.4.1 Tile and Frame Specification

A given frame of width f_w and height f_h containing $n_{channel}$ color channels is specified by parameters `INITIAL_RESOLUTION_WIDTH`, `INITIAL_RESOLUTION_HEIGHT` and `N_CHANNELS` respectively. Each pixel is therefore represented by $n_{channel}$ bytes, e.g. 3 bytes for RGB888 as requested by the neural network used in this project. Gray-scale images can be processes by setting this parameter to value 1. For frames in RGB565 color mode $n_{channel} = 2$ can be specified so one pixel is represented by 2 bytes. The frame is divided into overlapping tiles of different tile classes which are specified by parameter `TILE_SPEC` containing four parameters for each of n_{tclass} tile classes. Each tile class j is characterized by a tile width $t_{j,w}$, tile height $t_{j,h}$, relative horizontal stride $t_{j,swr}$ and relative vertical stride $t_{j,shr}$. Absolute stride values are given by equation 17, rounding to the next smallest integer. Index j is limited to range $0 \leq j < n_{tclass}$ in the following equations.

$$\begin{aligned} t_{j,swa} &= \lfloor t_{j,swr} \cdot t_{j,w} \rfloor \\ t_{j,sha} &= \lfloor t_{j,shr} \cdot t_{j,h} \rfloor \end{aligned} \tag{17}$$

The number of tiles for tile class j in horizontal and in vertical direction $n_{j,w}$ and $n_{j,h}$ is then given by equation 18. The total number of tiles for tile class j is denoted by n_j.

$$\begin{aligned} n_{j,w} &= \left\lfloor \frac{f_w - t_{j,w}}{t_{j,swa}} + 1 \right\rfloor \\ n_{j,h} &= \left\lfloor \frac{f_h - t_{j,h}}{t_{j,sha}} + 1 \right\rfloor \\ n_j &= n_{j,w} \cdot n_{j,h} \end{aligned} \tag{18}$$

The total number of tiles n is given by equation 19.

$$n = \sum_{j=0}^{n_{tclass}-1} n_j \tag{19}$$

Tile classes can specify tiles with arbitrary width and height. Both dimensions are clipped to frame size f_w and f_h before processing tiles. Furthermore all tiles must be scaled to a final size of $t_{res,w} \times t_{res,h}$ pixel to be classified by the neural network. Therefore scaling factors $t_{j,sx}$ and $t_{j,sy}$ are being computed at initialization time according to equation 20.

$$\begin{aligned} t_{j,sx} &= \frac{t_{j,w}}{t_{res,w}} \\ t_{j,sy} &= \frac{t_{j,h}}{t_{res,h}} \end{aligned} \tag{20}$$

All tiles are being appended to a buffer of type 8-bit unsigned integer in memory which then is passed to the classifier. The order of pixels is row-major while each pixel is represented as $n_{channel}$ channel values next to each other. The size of the buffer n_{buffer} to be allocated for the given neural network is computed in equation 21.

$$n_{\mathit{buffer}} = t_{res,w} \cdot t_{res,h} \cdot n_{channel} \cdot n = 3072 \cdot n, [n_{\mathit{buffer}}] = \text{byte} \tag{21}$$

Figure 7 shows an example of the tiling process with a frame of size $f_w \times f_h$ and $n_{channel} = 3$ (RGB888). $n_{tclass} = 2$ tile classes are given by specification as sketched in the upper left and bottom right corner of the frame. Some characteristic tile values discussed in the preceding equations are being shown alongside. As a result n_0 and n_1 tiles are generated for tile classes $j = 0$ and $j = 1$ respectively. A total amount of $n = n_0 + n_1 = 46$ tiles is being extracted and indexed by value $i, 0 \le i < n$. All tiles must be rescaled to size 32×32 and are then mapped to a buffer, sorted by index i. Buffer size is $n_{\mathit{buffer}} = 138$ kB as specified by equation 21.

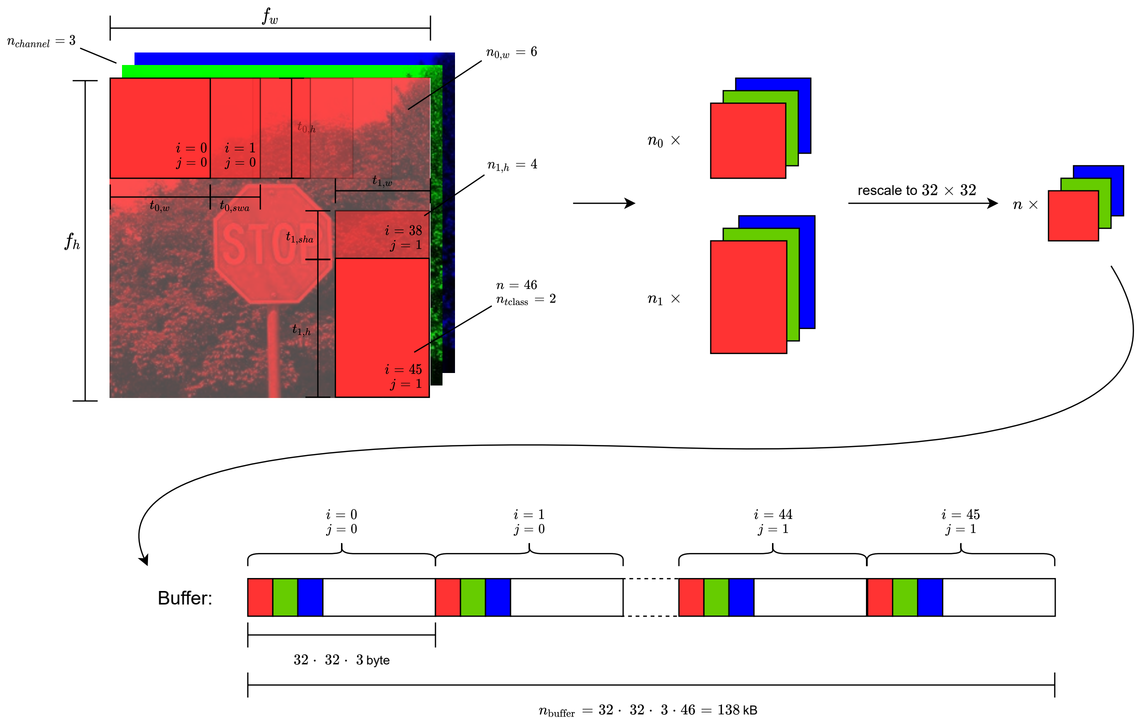


Figure 7: Tiling process example

Some considerations are made to avoid redundant computation on each loop and to maximise the frame rate. At initialisation time and when frame dimensions are modified (e.g. by a command from BS) all specified tile parameters are being verified and all relevant values are being computed as described above. The tile buffer can then be allocated once with a static size n_{buffer} and will be overwritten at each loop. Additionally the coordinates of tile centres are being precomputed using the complete frame as reference and can be then looked up by using tile index i. This enables target object location as described in subsubsection 6.4.4.

All tiles will be saved to file "tiles.csv" for each frame by setting parameter `TILING_DEBUG=1`. This enables exporting and inspecting all created tiles on a separate machine. The first two values in each line specify tile dimensions $t_{j,w}, t_{j,h}$ when saved before resizing and resized tile dimensions $t_{res,w}, t_{res,h}$ when saved after resizing tiles. The third value specifies number of channels (bytes per pixel) $n_{channel}$. The following values of each line contain pixel values in row-major order.

6.4.2 Tile Generation by ROI

The frame is provided as an OpenCV matrix. OpenCV is a framework for computer vision aimed at real-time applications. The matrix type allows efficient cropping to a ROI (region of interest) submatrix. A new matrix header with specified boundaries is created in an O(1) operation and no data is copied [12]. Firstly the provided frame is duplicated so data send to BS will be kept unchanged. All values are converted to precomputed fixed point values by `cv::Mat::forEach` method. Corresponding tiles are generated by iterating through each tile class j, all tiles in vertical direction $n_{j,h}$ and all tiles in horizontal direction $n_{j,w}$ so all combinations are processed. ROI width $r_{j,w}$, height $r_{j,h}$ and upper left coordinate $r_{jn,x}, r_{jm,y}$ is given by equation 22. Single tiles are then being rescaled to size $t_{res,w} \times t_{res,h}$ pixel and appended to a buffer in memory.

$$\begin{aligned} r_{j,w} &= t_{j,w} \\ r_{j,h} &= t_{j,h} \\ r_{jn,x} &= n \cdot t_{j,swa}, 0 \leq n < n_{j,w} \\ r_{jm,y} &= m \cdot t_{j,sha}, 0 \leq m < n_{j,h} \end{aligned} \tag{22}$$

OpenCV provides several interpolation methods for scaling, which can be specified by parameter `INTERPOLATION_FN` [12]. In the following `cv::INTER_AREA`, `cv::INTER_LINEAR`, `cv::INTER_NEAREST` and `cv::INTER_LANCZOS4` are being compared. Different tile dimensions $t_{0,w}, t_{0,h}$ are given for a frame of size $f_w = 640, f_h = 480, n_{channel} = 3$ and $n_{tclass} = 1$ tile classes with $t_{0,swr} = t_{0,shr} = 0.25$. As a result a varying amount of tiles n is being generated, which can be calculated by equation 19. Note that only downscaling is being considered in this project as the requested size of 32×32 pixel is tiny in terms of frame size. Table 3 shows computation time per tile $\frac{dt}{n}$ for all four interpolation methods.

$t_{0,w} \times t_{0,h}$	n	Time per tile $\frac{dt}{n}$ (approximation) in μs			
		`INTER_AREA`	`INTER_LINEAR`	`INTER_NEAREST`	`INTER_LANCZOS4`
64×64	999	60	60	44	906
96×96	391	345	243	64	964
100×100	352	1142	247	65	1105
128×128	204	525	270	93	1324
150×150	126	2095	312	119	1479
160×160	117	700	316	132	1516

Table 3: Comparison of OpenCV interpolation methods computation time

Figure 8 shows a graphical representation of the results. While lanczos interpolation (`INTER_LANCZOS4`) can produce smooth and high quality results it adds a significant amount of computation time per tile and is therefore discarded for further considerations. Bilinear (`INTER_LINEAR`) and nearest neighbor (`INTER_NEAREST`) interpolation provide a fast and efficient solution for rescaling tiles. Both methods seem to scale linearly by the tile size and by a small factor. The method `INTER_AREA` provides especially fast computation when tile dimensions $t_{j,w}, t_{j,h}$ are an integer multiple of destination dimension $t_{res,w}, t_{res,h}$, i.e. when scaling factors are integer $t_{j,sx}, t_{j,sy} \in \mathbb{N}$. Otherwise the computation time will scale strongly in terms of tile size.

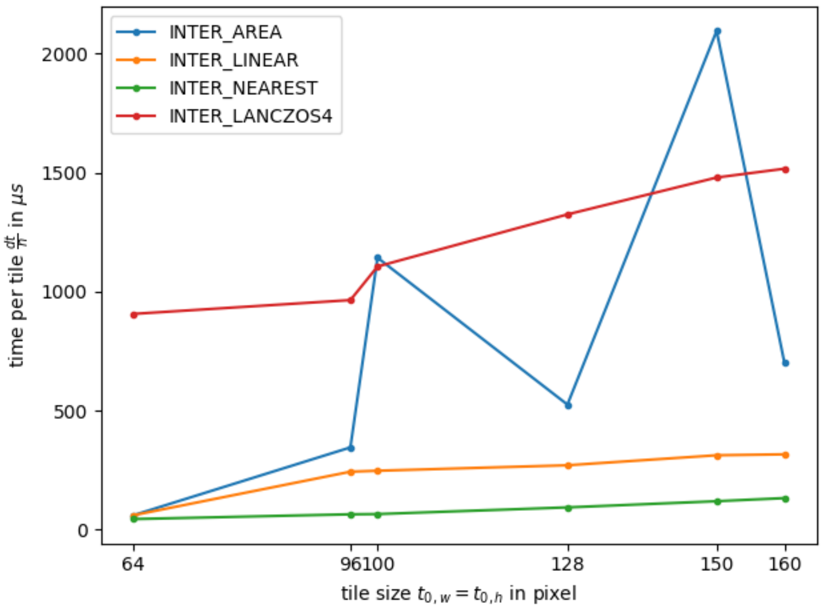


Figure 8: Comparison of OpenCV interpolation methods computation time (graphical)

In the following the quality of different interpolation methods is being investigated. Figure 9 shows tiles for an example frame. The upper tiles are being specified with dimensions $t_{0,w} = t_{0,h} = 100$ and then rescaled to $t_{res,w} = t_{res,h} = 32$, while the lower tiles are being specified with dimensions $t_{1,w} = t_{1,h} = 160$ and rescaled similarly. Note that latter are being scaled by an integer factor $t_{1,sx} = t_{1,sy} = \frac{160}{32} = 5$ and thus will be efficiently computed by `INTER_AREA` as shown in Figure 8. Quality differences are little perceptible for small tile dimensions while the quality for bigger tile dimensions differs strongly between bilinear or nearest neighbor interpolation and `INTER_AREA`. Latter interpolation method uses pixel area relation and gives "moire'-free results" as described in OpenCV documentation [12]. It is therefore useful to develop an algorithm to choose the optimal interpolation method based on tile class specification $t_{j,w}, t_{j,h}$, satisfying the tradeoff between computation time and tile quality. Small tiles can then be rescaled by bilinear or nearest neighbor interpolation to save computation time while `INTER_AREA` interpolation method should be preferred for increasing tile dimensions, which should be chosen as an integer multiple of $t_{res,w}, t_{res,h}$, so a high accuracy result can be achieved. Some tests could already verify that overall confidence values are grater when `INTER_AREA` interpolation method is used.

In some cases a tile dimension cannot be chosen to be an integer multiple of $t_{res,w}, t_{res,h}$ but a high quality rescaling is required. Especially for cases when the vehicle gets close to the target object a tile class with large tile dimensions must be specified for the target object to fit inside. The second method for tile generation, described in subsubsection 6.4.3, can be used as an alternative in such cases, filling the gap between efficient computation and flexible selection of tile specifications.

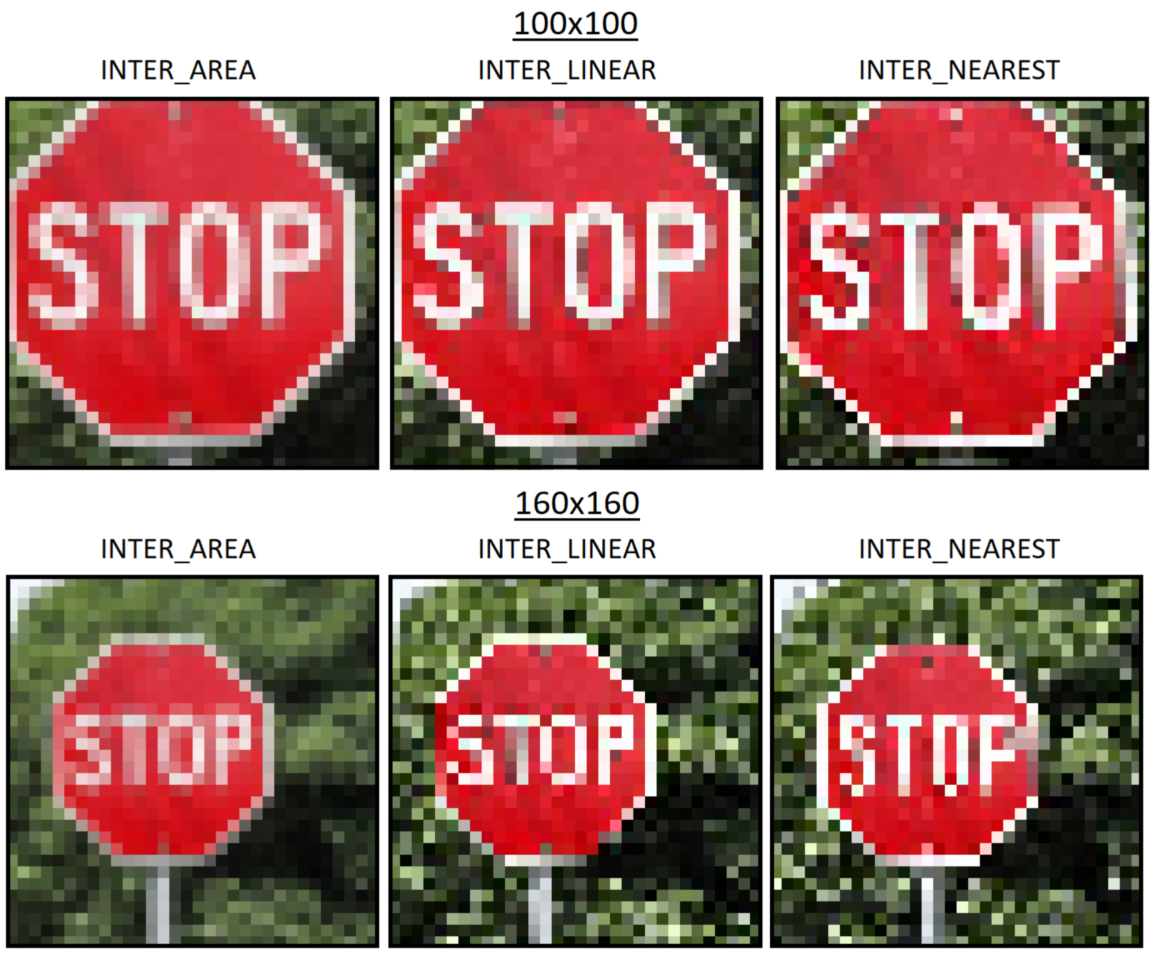


Figure 9: OpenCV interpolation methods quality for $t_{0,w}, t_{0,h} = 100$ (top) and $t_{1,w}, t_{1,h} = 160$ (bottom)

6.4.3 Tile Generation by Mapping

Tile generation by mapping uses a different approach then extracting submatrices from a given frame with dimensions f_w, f_h. At initialization time a matrix of size $f_w \times f_h \times 4$ is precomputed, which can be interpreted as a grid holding tile indices i for different areas of the frame. In a main loop all $f_w \cdot f_h$ pixels are processed once using OpenCV `cv::Mat::forEach` method. A given pixel with coordinates p_x, p_y belongs to all tiles in the range $[i_{j,min,x}, i_{j,max,x}] \wedge [i_{j,min,y}, i_{j,max,y}]$ according to equation 23. This four values are looked up inside the precomputed matrix for a given tile class j. This tile indices are then being transformed to absolute indices inside the tile buffer and a pixel is copied to associated positions. Furthermore a weight matrix for rescaling using bilinear interpolation is being constructed, holding weights for positions in the grid. A pixel value is then only partially copied to the tile buffer.

$$
\begin{aligned}
i_{j,min,x}(p_x) &= \left\{ \begin{array}{ll} 0, & \text{for } p_x < t_{j,w} \\ \left\lfloor \frac{p_x - t_{j,w}}{t_{j,swa}} + 1 \right\rfloor, & \text{else} \end{array} \right\} \\
i_{j,max,x}(p_x) &= \left\lfloor \min(n_{j,w}, \frac{p_x}{t_{j,swa}} + 1) \right\rfloor \\
i_{j,min,y}(p_y) &= \left\{ \begin{array}{ll} 0, & \text{for } p_y < t_{j,h} \\ \left\lfloor \frac{p_y - t_{j,h}}{t_{j,sha}} + 1 \right\rfloor, & \text{else} \end{array} \right\} \\
i_{j,max,y}(p_y) &= \left\lfloor \min(n_{j,h}, \frac{p_y}{t_{j,sha}} + 1) \right\rfloor
\end{aligned}
\qquad (23)
$$

Figure 10 shows an example of the simple case when $n_{tclass} = 1$ tile class is specified with relative strides $t_{0,swr} = t_{0,shr} = 0.5$. The number of tiles in horizontal and in vertical direction is $n_{0,w} = 4, n_{0,h} = 3$ and the total number of tiles is $n = 12$. Each cell in the shown grid is mapped to tiles with indices i specified inside the corresponding cell. Pixel coordinates must be converted into indices inside the tile buffer. Additionally each pixel must be weighted by a specific value $w_{i,xy}$ to enable scaling.

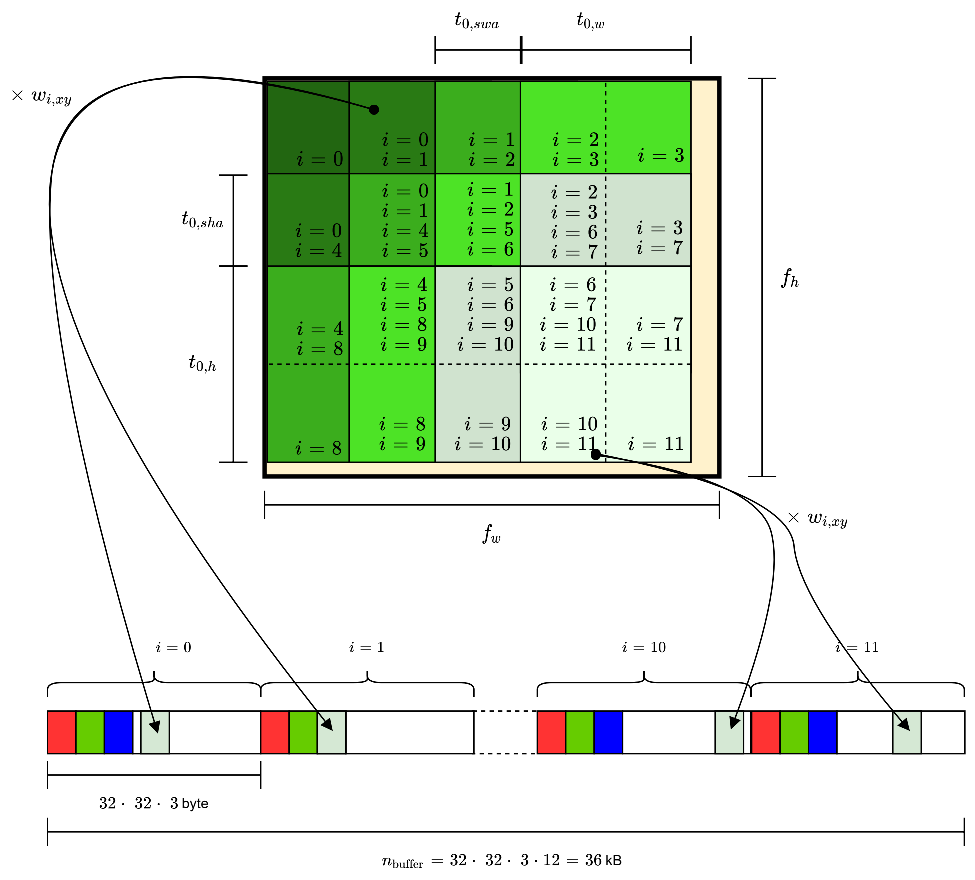


Figure 10: Example on tile generation by mapping

6.4.4 Target Object Location

For estimating the location of the target object inside a given frame a moving average and two threshold values are being used as described in the following. This process is used as a simple and efficient way to encode the classifier outputs into a single x- and y-axis-position.

The classifier returns confidence values for each tile and each object class. Therefore, when processing n tiles by a classifier with $n_{classes}$ object classes ($n_{classes} = 42$ for German Traffic Sign Recognition Benchmark), $n{\cdot}n_{classes}$ confidence values are returned. For targeting a specific object only the confidence value c_{ij} for tile i and object class j, given by parameter `SIGN_ID`, is of interest. A tile is considered valid only if the confidence value c_{ij} is greater then the value given by parameter `CONFIDENCE_THRESHOLD`. The coordinate t_{ix}, t_{iy}, describing the center of tile i inside the complete frame, is looked up from a precomputed array and added to a moving average ma_x, ma_y. Finally, the number of valid tiles n_{valid} must be greater than parameter `CORRECT_CLASS_THRESHOLD` for the complete frame to contain a valid target, i.e. for p_{valid} to be 1. The position of the target object is then given by $p_x = \frac{ma_x}{n_{valid}}, p_y = \frac{ma_y}{n_{valid}}$. Value p_x is normalized to range $[-1, 1]$ and fed into a PID controller as *system-output* Δx_{norm} as described in subsubsection 6.5.4. Equations 24, 25 and 26 describe this process in mathematical terms.

$$n_{valid} = \sum_{i=0}^{n-1} \left\{ \begin{array}{ll} 0, & \text{for } c_{ij} \leq \text{CONFIDENCE\_THRESHOLD} \\ 1, & \text{else} \end{array} \right\}, \text{where } j = \text{SIGN\_ID} \tag{24}$$

$$p_x = \frac{ma_x}{n_{valid}} = \frac{1}{n_{valid}} \sum_{i=0}^{n-1} \left\{ \begin{array}{ll} 0, & \text{for } c_{ij} \leq \text{CONFIDENCE\_THRESHOLD} \\ t_{ix}, & \text{else} \end{array} \right\}, \text{where } j = \text{SIGN\_ID} \tag{25}$$

$$p_{valid} = \begin{cases} 0, & \text{for } n_{valid} \leq \text{CORRECT\_CLASS\_THRESHOLD} \\ 1, & \text{else} \end{cases} \tag{26}$$

6.5 Vehicle Control

written by Pablo Navarro

The vehicle is controlled in two different modes. The first mode, called *Search*, is activated whenever no target object is located in the scene. The vehicle then turns by its longitudinal axis until a target object is located. Once an object is targeted the second mode, called *Approach*, is activated. The steering direction is then computed by subtracting the actual position of the object inside the scene from the center coordinates of the image stream at each time-step. A PID controller then estimates the optimal angular velocity $\dot{\varphi}$ of the vehicle for approaching the target object. This output, called *control-signal*, is then translated into rotational speed for all four motors. As the motors are controlled by a PWM signal, the rotational speed is translated to PWM duty cycle. The relation between all four motor speeds is calculated in such way, that the absolute velocity v of the vehicle is always equal to a constant parameter v_0.

6.5.1 Vehicle Dynamics

The vehicle is driven by two DC motors at each size. Both left sided motors as well as both right sided motors are controlled together, so vehicle left velocity $v_l(t)$ and vehicle right velocity $v_r(t)$ can be adjusted. The resulting vehicle absolute velocity $v(t)$ is given by the average of both side velocities and can be computed by equation 27.

$$v(t) = \frac{v_l(t) + v_r(t)}{2} \tag{27}$$

Rotational speed of the motors $n_l(t), n_r(t)$ is related to vehicle velocities $v_l(t), v_r(t)$ by equation 28 as both left motors drive at same rotational speed and both right motors do similarly. Tire radius `TIRES_RADIUS` is denoted as r in m and must be measured for the given vehicle. Note that a maximal coefficient of ground-friction $\mu = 1$ is assumed.

$$\begin{aligned} v_l(t) &= 2\pi \cdot r \cdot n_l(t) \\ v_r(t) &= 2\pi \cdot r \cdot n_r(t) \end{aligned} \tag{28}$$

The vehicle direction can be described in terms of an angle φ. Its angular velocity along the z-axis is given by $\dot{\varphi}(t)$ and can be calculated by equation 29. The length `TIRES_DISTANCE`, denoted as a, describes the distance from tire to vehicle symmetry line in m and is also being measured in advance.

$$\dot{\varphi}(t) = \frac{v_l(t) - v_r(t)}{a} \tag{29}$$

The vehicle angular velocity $\dot{\varphi}(t)$ can also be expressed by vehicle rotational speed $n_V(t)$ by equation 30.

$$\dot{\varphi}(t) = 2\pi n_V(t) \tag{30}$$

Figure 11 illustrates the described parameters and vehicle dynamics.

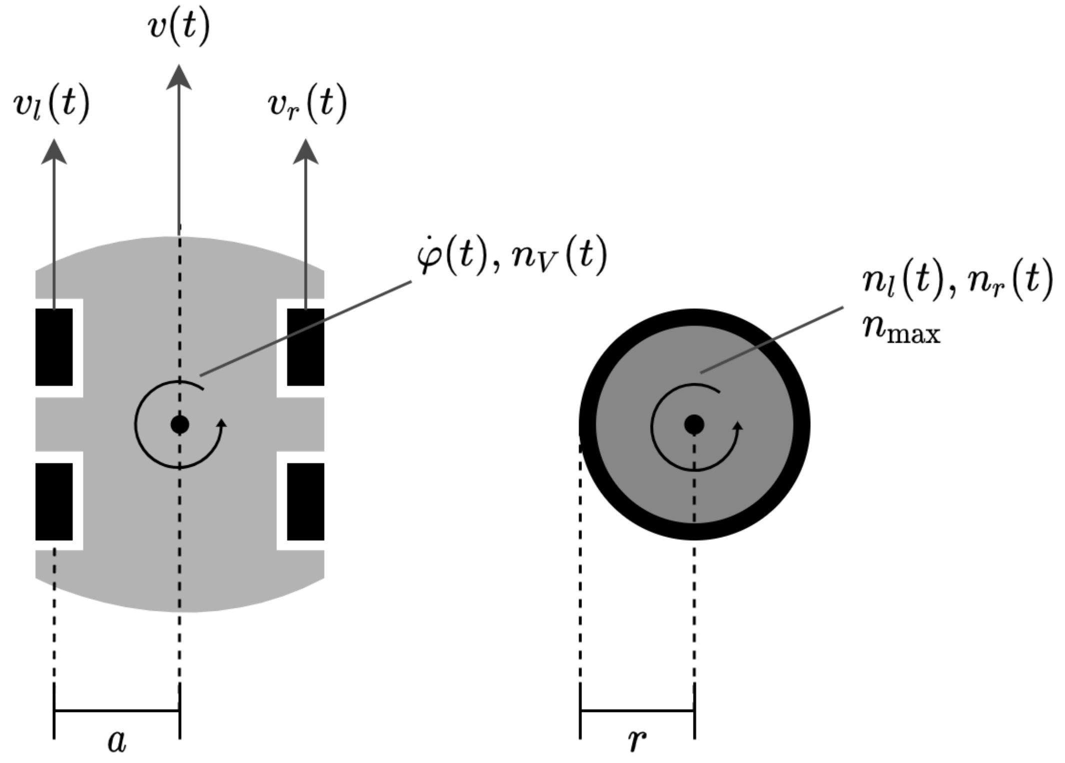


Figure 11: Vehicle dynamics (left: vehicle top view; right: vehicle tire front view)

6.5.2 Control Modes

The vehicle switches between two modes, *Search* and *Approach*, as mentioned in subsection 6.5. As described in subsubsection 5.2.2 the four motors are controlled by L298N motor H-Bridge by six pins which must be set accordingly. To prevent frequent mode switches and therefore instability in vehicle control two parameters can be adjusted. The transition from *Approach*-Mode to *Search*-Mode only occurs after `MIN_SUCCESSIVE_FALSE` successive frames are not satisfying the validity condition specified in subsubsection 6.4.4. Anyways, as soon as a frame is considered invalid, the vehicle instantly halts while still remaining in *Approach*-Mode until the above condition is fulfilled. Similarly, the transition from *Search*-Mode to *Approach*-Mode is performed after `MIN_SUCCESSIVE_TRUE` successive frames are satisfying the validity condition specified in subsubsection 6.4.4.

***Search*-Mode**

In *Search*-Mode the vehicle turns by its z-axis in intervals of length `VEHICLE_TURN_TIME` ms and pauses of length `VEHICLE_TURN_SLEEP` ms in between. This ensures that frames are grabbed while the vehicle is in rest and therefore tiles with best possible quality are fed into the classifier. The initial turn direction is defined by parameter `VEHICLE_TURN_DIRECTION_INITIAL` and is either clockwise (0) or counter-clockwise (1). Afterwards, at every transition from *Approach*-Mode to *Search*-Mode, a turn direction, denoted as $turn\_dir$, is determined by the last known position of the target which can be either on the left or right frame half as described by equation 31. In some cases this procedure can lead to faster recovery of the target. Δx_{norm} describes the horizontal deviation from vehicle direction to target object direction and is normalized in the range $-1 \leq \Delta x_{norm} \leq 1$.

$$turn\_dir = \left\{ \begin{array}{ll} 0, & \text{for } \Delta x_{norm} \geq 0 \\ 1, & \text{for } \Delta x_{norm} < 0 \end{array} \right\}, \text{evaluated when transitioning Approach-Mode} \rightarrow \text{Search-Mode} \tag{31}$$

While turning, the absolute velocity should be $v = 0$, i.e. the vehicle should not move away from its position. Therefore both left and right sided motors are set to opposite directions using pins IN1, IN2, IN3 and IN4. Pin ENA controls motor speed of left sided motors and pin ENB of right sided motors respectively and are set depending on parameter `VEHICLE_TURN_SPEED`, denoted as $n_{VSearch}$, which defines the constant rotational speed of the vehicle when turning in $\frac{1}{s}$. As both pins require a PWM signal its specific PWM duty cycle d_{Search} in μs must be computed. Using equations 28, 29 and 30 and setting $n_V(t) = n_{VSearch}$ follows equation 32. Note that both left and right side motors rotational speed $n_l(t), n_r(t)$ are equal magnitude but opposite direction and get substituted by $n_{lSearch} = -n_{rSearch} = n_{Search}$.

$$2\pi n_{VSearch} = \frac{2\pi \cdot r \cdot (n_{lSearch} - n_{rSearch})}{a} = \frac{4\pi \cdot r \cdot n_{Search}}{a} \tag{32}$$

Translating motor rotational speed $n_l(t), n_r(t)$ in $\frac{1}{s}$ to PWM signal $d_l(t), d_r(t)$ in μs requires maximum PWM duty cyle `PWM_MAX`, denoted as d_{max} in μs. Additionally, the maximal rotational speed of motors `MOTORS_ROT_SPEED_MAX`, denoted as n_{max} in $\frac{1}{s}$ must be measured for the given vehicle or derived from technical information about the motors. Equation (33) shows the relationship between both variables. Note that, for simplicity, the correlation between PWM duty cycle (and therefore motor supply voltage) and motor rotational speed is assumed to be linear.

$$\begin{aligned} d_l(t) &= \left| \frac{d_{max}}{n_{max}} \cdot n_l(t) \right| \\ d_r(t) &= \left| \frac{d_{max}}{n_{max}} \cdot n_r(t) \right| \end{aligned} \tag{33}$$

The PWM duty cycle d_{Search}, which is equal for both sides, can then be computed by equation (34) using equations (32) and (33).

$$d_{Search} = d_{lSearch} = d_{rSearch} = \left| \frac{d_{max}}{n_{max}} \cdot n_{Search} \right| = \left| \frac{d_{max}}{n_{max}} \cdot \frac{a}{2r} \cdot n_{VSearch} \right| \tag{34}$$

The constant rotational speed $n_{VSearch}$ of the vehicle when turning is constrained by a maximum value $n_{VSearch,max}$ by setting $n_{Search} = n_{max}$ in equation 32 as given in equation 35.

$$n_{VSearch,max} = \frac{2r \cdot n_{max}}{a} \tag{35}$$

***Approach*-Mode**

In *Approach*-Mode both sides of the vehicle move forward, so L298N pins IN1 and IN3 are set to low (0) and pins IN2 and IN4 are set to high (1). Therefore the condition $v_l(t), v_r(t) \geq 0$ holds. Vehicle angular velocity $\dot{\varphi}(t)$ is given as the *control-signal* in $\frac{rad}{s}$. Vehicle absolute constant velocity v_0 is given as a parameter `VEHICLE_ABS_VEL` in $\frac{m}{s}$. It is constrained to a maximum value $v_{0,max}$ given by equation 36 which can be achieved even if a maximum vehicle angular velocity is required, i.e. when either $v_l(t) = 0$ and $v_r(t)$ is maximum or $v_r(t) = 0$ and $v_l(t)$ is maximum.

$$v_{0,max} = v(t)|_{\substack{v_l(t)=0 \\ v_r(t)=2\pi r n_{max}}} = v(t)|_{\substack{v_l(t)=2\pi r n_{max} \\ v_r(t)=0}} = \pi r n_{max} \tag{36}$$

To translate vehicle angular velocity $\dot{\varphi}(t)$ into specific PWM duty cycles $d_l(t), d_r(t)$ for addressing both left and both right sided motors the equation 37 is derived from equations 27 and 29.

$$\begin{aligned} v(t) = v_0 &= \frac{v_l(t) + v_r(t)}{2} = \frac{a\dot{\varphi}(t) + 2v_r(t)}{2} \\ \Rightarrow v_r(t) &= v_0 - \frac{a}{2}\dot{\varphi}(t) \end{aligned} \tag{37}$$

From equations 28 and 37 then equation 38 can be derived.

$$n_r(t) = \frac{v_r(t)}{2\pi r} = \frac{1}{2\pi r} \cdot (v_0 - \frac{a}{2}\dot{\varphi}(t)) \tag{38}$$

Finally, the PWM duty cycle $d_r(t)$ is given by equation 39 according to equations 33 and 38. The same process is repeated for PWM duty cycle $d_l(t)$.

$$\begin{aligned} d_l(t) &= \left| \frac{d_{max}}{n_{max}} \cdot \frac{1}{2\pi r} \cdot (v_0 + \frac{a}{2} \cdot \dot{\varphi}(t)) \right| \\ d_r(t) &= \left| \frac{d_{max}}{n_{max}} \cdot \frac{1}{2\pi r} \cdot (v_0 - \frac{a}{2} \cdot \dot{\varphi}(t)) \right| \end{aligned} \tag{39}$$

Figure 12 illustrates the described parameters and vehicle dynamics in both modes.

Table 4 gives an overview of how all six L298N pins are mapped during both modes.

L298N-Pin	*Search*-Mode	*Approach*-Mode
IN1	$\neg turn\_dir$	0
IN2	$turn\_dir$	1
IN3	$turn\_dir$	0
IN4	$\neg turn\_dir$	1
ENA	d_{Search} (PWM)	$d_l(t)$ (PWM)
ENB	d_{Search} (PWM)	$d_r(t)$ (PWM)

Table 4: L298N pinout for *Search*- and *Approach*-Mode

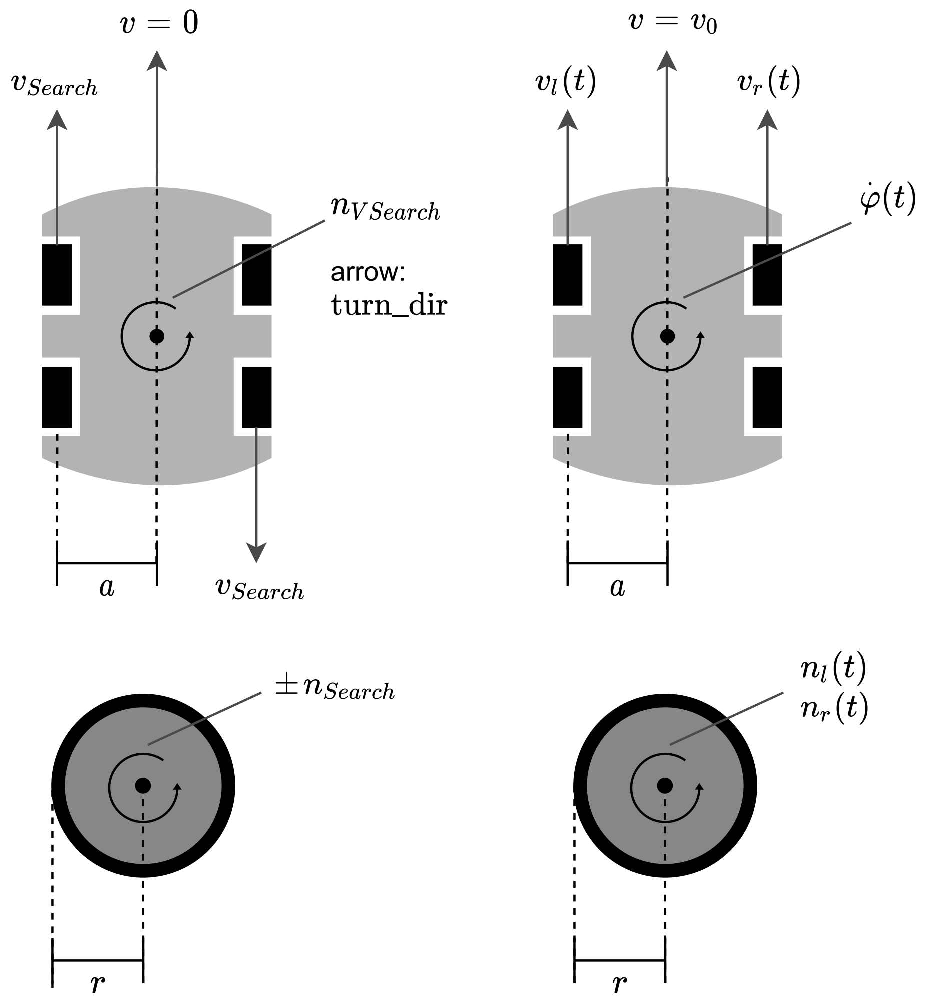


Figure 12: Vehicle dynamics in Search-Mode (left) and in Approach-Mode (right)

6.5.3 Control Parameters

Table 5 summarizes all relevant parameters discussed in subsubsection 6.5.2. Constant parameters have to be measured for a specific vehicle or specified while variable parameters can be easily adapted and tuned. Column Value shows the parameters used in this project.

Parameter	Name in Code	Unit	Range	Value
a	TIRES_DISTANCE	m	$\mathbb{R}^+$	0.065
r	TIRES_RADIUS	m	$\mathbb{R}^+$	0.0325
n_{max}	MOTORS_ROT_SPEED_MAX	$\frac{1}{s}$	$\mathbb{R}^+$	4
d_{max}	PWM_MAX	μs	$\mathbb{N}$	1023
$turn\_dir$	VEHICLE_TURN_DIRECTION_INITIAL	-	$\{0,1\}$	0
$n_{VSearch}$	VEHICLE_TURN_SPEED	$\frac{1}{s}$	$[0, \frac{2r \cdot n_{max}}{a}]$	1.3
-	VEHICLE_TURN_TIME	ms	$\mathbb{N}$	250
-	VEHICLE_TURN_SLEEP	ms	$\mathbb{N}$	4000
v_0	VEHICLE_ABS_VEL	$\frac{m}{s}$	$[0, \pi r n_{max}]$	0.2

Table 5: Constant parameters (top) and variable parameters (bottom)

6.5.4 Control Loop

The control loop is applied only in *Approach* mode and consists of multiple blocks chained together, where a *setpoint* w is given as input and a *system output* Δx_{norm} is controlled and returned to the input as a *feedback signal*. This is referred to as a *closed-loop-* or *feedback-control* in control theory. The *error signal* e is calculated as $e = w - \Delta x_{norm}$ and is passed to a *controller*, which then outputs a *control signal* $\dot{\varphi}$. An *actuator* computes the *actuator signal* φ_R which is influenced by *disturbances* z from the surrounding environment. Finally a *system input* φ is passed into the *process* which then results in the returned *system output*. [11]

The control target is to approximate the *system output* to the chosen *setpoint*, that is $\Delta x_{norm} \approx w$. Δx_{norm} describes the horizontal deviation from vehicle direction to target object direction and is normalized in the range $-1 \leq \Delta x_{norm} \leq 1$. Therefore the *setpoint* w, defined by parameter PID_SETPOINT, is chosen to be 0 as the target object should be approached in a straight line.

The motor control block inside the *actuator* translates an angular velocity $\dot{\varphi}$ to PWM duty cycle d_l, d_r (see (39)) for left and right motors, which are then transmitted to the motor H-Bridge L298N described in subsection 5.2. The vehicle then is rotated resulting in a change of its absolute angle φ_R. Influences by the surrounding environment like uneven ground or unequal behaviour of motors could then affect this angle. The resulting angle φ is perceived by the vehicle camera, Processing System (PS) and Programmable Logic (PL) and is then translated into a deviation to the target object. Figure 13 illustrates the control loop.

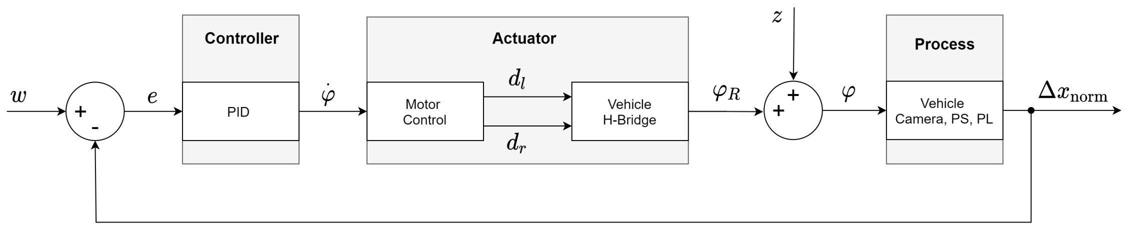


Figure 13: Control loop

PID Controller Block

A proportional–integral–derivative (PID) controller is chosen as controller block with input e and output $\dot{\varphi}$. The time domain characterisation is given by equation 40. Note that the term is being simplified as the *setpoint* w is constant throughout the entire process.

$$\begin{aligned}\dot{\varphi}(t) &= K_P \cdot e(t) + K_I \cdot \int\limits_0^t e(\tau)d\tau + K_D \frac{de(t)}{dt} \\ &= K_P \cdot e(t) + K_I \cdot \int\limits_0^t e(\tau)d\tau + K_D \frac{d\Delta x_{norm}(t)}{dt}\end{aligned} \tag{40}$$

The values K_P, K_I and K_D characterise how the response to an input reflects on the output and can be set by parameters `PID_KP`, `PID_KI` and `PID_KD` respectively. The PID evaluation function is not called in fixed time-steps, as it depends on the competition of preceding tasks like tiling and inference. This can possibly lead to inconsistent behaviour, especially due to peaks in the derivative term. Therefore a *sample time* t_s is specified by parameter `PID_SAMPLE_TIME`, ensuring that a fixed time interval passes between PID evaluation loops. Therefore parameters K_I and K_D can be multiplied or divided by t_s respectively once at initialisation time, which saves two operations per loop. The parameter values chosen in this project are summarised in Table 6. The benefit of integral and derivative gain values K_D and K_I is limited by maximum FPS and vehicle velocity achieved in this process. Therefore only the proportional factor K_P is being considered to keep the process simple.

Parameter	Value
K_P	2.2
K_I	0
K_D	0
t_s	$50\,ms$
w	0

Table 6: PID controller parameter values

The *control signal* or vehicle angular velocity $\dot{\varphi}(t)$ in $\frac{rad}{s}$ is constrained to a minimum and maximum value to fulfil the requirement of constant absolute velocity v_0 as described in subsubsection 6.5.2. By setting $d_l(t) = d_{max}$ and $d_r(t) = d_{max}$ in equation 39 the values $\dot{\varphi}_{max}$ and $\dot{\varphi}_{min}$ can be computed as described in equation 41.

$$\begin{aligned}\dot{\varphi}_{max} &= \frac{2}{a} \cdot (2\pi r n_{max} - v_0) \\ \dot{\varphi}_{min} &= -\frac{2}{a} \cdot (2\pi r n_{max} - v_0) = -\dot{\varphi}_{max}\end{aligned} \tag{41}$$

7 Design of “Base Station” Subsystem

The “Base Station” subsystem is the only subsystem at another physical location than the vehicle. It's purpose is to give insights and control over the vehicle while in operation but is not required for the vehicle to function. Instead it is intended as a debugging tool.

7.1 Toolchain

written by Felix Müller

To be consistent with the PS, the same toolchain (C++ compiled with GCC) is used. Additonally a graphics library is required to show a preview of the image seen by vehicle and other debug information (e.g. the predicted position). For this purpose SDL2 was selected, which is a library that provides low level access to different system components for input and output (e.g. keyboard, mouse or audio) but most importantly for this project are the interfaces to access the graphics hardware of the host system [15].

7.2 Components

The software is split into three important components. The "Network" component communicates with the vehicle. The "Video" component displays information to the user. The "Input" component takes user input and forwards it to the vehicle over the network.

written by Felix Müller

7.2.1 Network

For the communication between PS and BS the User Datagram Protocol (UDP) is used, which is accomplished by simple sockets in the standard library of the GCC. The packet size must be decided beforehand and set to the same value as in the PS application. This size has to be selected based on the reliability of the used network. Whereas packages of around 1000 Bytes or even jumbo frames above 1500 Bytes might work in a wired Ethernet environment this is certainly not the case in a wireless network (e.g. WiFi or radio transmission via a satellite). For now a size of 512 Bytes was selected, because an analysis of transmission reliability is not the focus of this project. This aspect could be explored in the future.

One received datagram contains one so-called frame of data which is identified by an identifier as the first byte. Currently there are two types of frames sent from the vehicle to the base station: "Control" and "(Image) Data". The control frame contains the state of the vehicle. This is accomplished by packing several bytes of information into one frame as described in Table C.1. The image frames contain a video feed that is split into several frames which are identified by a 24-bit ID at the beginning of the frame. The format of the remaining data is specified by the selected color mode (e.g. RGB565 is 2 bytes per pixel in the format of 5-bit red, 6-bit green and 5-bit blue). Double-buffering is used to minimize flickering or artifacts in the preview. This method uses two buffers where one will be filled with new incoming data and the other one will be displayed to the user by the video component. Once an entire frame was received, the two buffers get swapped, therefore the image does not change while it is being displayed.

It is also possible to send data frames to the vehicle. Five such frames were implemented to set the resolution (Table C.2), set the color mode (Table C.3), enable/disable the video feed (Table C.4), change the displayed tile (Table C.5) or control the motors (Table C.6).

7.2.2 Video

The visible buffer that was received over the network gets converted into a texture and send to an appropriate device (e.g. a graphics card) which displays it to the user. Other debug information is converted into human readable strings and then further converted into textures that are overlayed onto the image preview. To preserve resources, these text textures are only regenerated if the data has changed.

7.2.3 Input

The provided event loop by SDL2 is used to process keyboard input by users. All inputs are forwarded to the vehicle according to the pressed key. The key functions are described in Table 7.

Key	Functionality
R	Cycle through image resolutions
V	Toggle video transmission
M	Toggle mode (manual or autonomous driving)
Space	Emergency stop (activates manual mode and disables all motors)
Up or W	Move the vehicle forward (only in manual mode)
Down or S	Move the vehicle backward (only in manual mode)
Left or A	Steer the vehicle left (only in manual mode)
Right or D	Steer the vehicle right (only in manual mode)

Table 7: Base Station Shortcuts

8 Evaluation

This section firstly shows how the complete system was able to accomplish different scenarios in multiple integration tests which is then backed up by numeric measurements showing the improvements in computation time, compared to a reference Python implementation, that made it possible to carry out the accomplished scenarios.

8.1 Integration Tests

written by Raffael Kaehn

After the implementation had concluded, the vehicle was subjected to a final evaluation under real-world conditions. All tests were conducted in the open air on a public parking space to set a realistic scene without disrupting road traffic. A decommissioned stop sign was utilised as a prop to be recognised by the neural network. After the vehicle had been set on the ground, there was no human intervention involved – all movements were initiated solely by the on-board control system.
The evaluation was split into three parts to showcase and test different aspects of the vehicle controller with each one explained in short hereafter:

- **Scenario 1: Approaching a static target** – For this test, the vehicle is positioned roughly five metres away from the stop sign, facing away from it at an angle of at least 45 degrees. At first, the vehicle is not able to see the target and remains in *search mode*, where it slowly turns around until a target is found. At that point, the vehicle transitions into *approach mode* where it moves forward in the direction of the target until it is no longer in the frame. During the whole test, the stop sign remains stationary.

- **Scenario 2: Tracking a dynamic target** – The vehicle is again placed roughly five metres away from the stop sign, but this time directly facing it. When the target is found and the vehicle moving, the sign is carried away from its initial position, but still remains visible in the vehicles camera frame. Here, the control systems ability to adjust the steering angle depending on the targets position relative to the vehicle can be observed: If the calculated center of the object is on the right side of the camera frames center, the vehicle moves to the right, if it is on the left, the vehicle moves left, with the magnitude respectively proportional to the distance from the target to the center line.

- **Scenario 3: Recovering from a target loss** – After the vehicle is placed on the ground, the stop sign is moved into the vehicles camera frame. Subsequent to its recognition, the vehicle moves in the correct direction to follow it. From there, the stop sign temporarily moves out the vehicles field of view, causing it to correctly transition back into *search mode*. Here it can be seen that this time, since the target is last seen on the left side before it is lost, the vehicle turns counterclockwise. As soon as it becomes visible again, the vehicle resumes its approach of the target.

The integration test has shown that the vehicle has the ability to fulfil all specified tasks of basic autonomous control, as it is capable of searching for and approaching a stationary target, following a target as it is moving, and rediscovering a target after it has been lost.

8.2 Measurements

written by Pablo Navarro

In the following the computation time for tiling and inference of a frame with size $f_w = 640, f_h = 480$ and $n_{channel} = 3$ between a given python example by BNN-PYNQ project and the solution developed in this project is being compared. Tile specification is given by equation 42.

$$\begin{gathered} t_{res,w} = t_{res,h} = 32 \\ n_{tclass} = 2 \\ t_{0,w} = t_{0,h} = 64, t_{0,swr} = t_{0,shr} = 0.25 \\ t_{1,w} = t_{1,h} = 96, t_{1,swr} = t_{1,shr} = 0.5 \\ \Rightarrow n = n_0 + n_1 = 37 \cdot 27 + 12 \cdot 9 = 1107 \end{gathered} \tag{42}$$

The process to be measured consists of creating and resizing tiles, transferring tiles to PL, inferring on PL and transferring confidence values back to PS. The first measurement is made using the algorithms discussed in subsubsection 6.4.2 and interpolation method `cv::INTER_AREA`. The second measurement is made by executing a python script inside a Jupyter Notebook environment. The results are shown in Table 8. Measurement times are broken down into three categories as detailed as possible for the given implementations.

	Time in ms	
	New Implementation	Python Example
Tiling	9.62	282.24
Inference	364.64	364.64
Data Transfer + Rescaling	128.27	3123.09
Total	502.53	3769.97

Table 8: Comparison of computing time between new implementation and BNN-PYNQ example

8.3 Analysis

written by Felix Müller

The new implementation turned out to be over over seven times faster than the original Python implementation and is able to process two images per second instead of one image every four seconds. This can be attributed to multiple factors. First off the new implementation is compiled specifically for the ARM-v9 architecture and therefore much faster in comparison to an implementation in an interpreted language (e.g. Python) when executing comparable algorithms. Secondly a combination of optimisation techniques were used that are more efficient than the algorithms used in the Python implementation to severely reduce the computation time of the image tiling and scaling. Finally the data preparation and transfer of the image data to the neural network on the FPGA was improved. Any improvement to the total processing time was achieved by these adaptions because exactly the same implementation for the inference on the FPGA was used which results in the same times for the inference.

9 Conclusion & Outlook

written by Felix Müller, Pablo Navarro, Niklas Krekel

This project established a basic hardware and software platform to research artificial neural networks on FPGAs. A realisation of an example use case demonstrated that the multiple subsystems (including the neural network) can work together to perform basic scenarios.

As the measurements have shown, this project already achieves a throughput of multiple frames per second at a rate of several hundreds of tiles per frame. But this does not nearly makes use of the full potential of the BNN-PYNQ which has shown to be able to process around twelve thousand 32x32 images per second. This measurement was taken on a FPGA that has around four times the resources of the chip used in this project yet even when taking a rough estimate of four times less throughput (around 3k images/second) into account, there still appears to be much more potential with this network. Some areas of improvement could be optimizations to the tiling of images, the interfacing with the PYNQ-BNN between PS and PL or adjustments to the PYNQ-BNN. Adjustments could be made in the form of reducing the size of the network by lowering the amount of classes and training a network for a more realistic object to track. The object used in this project was sufficient for some first tests but a more plastic object would allow evaluations closer to a real scenario.

The hardware platform has shown to be good enough for basic manoeuvring of the vehicle. Still the precision of the motor control turned out be restricting, especially in cramped areas. Therefore more work on the hardware is required to unleash the full potential of the software. Additionally the attachment of the camera needs to be improved because the current mount relays any vibrations from the drive train to the camera thus severely distorting the captured image.

Following is a list of additional ideas for possible improvements:

- Onboard compass to reliably determine vehicle rotation or swap motors for stepper motors to reliably get the current (relative) rotation angle.
- Optimize workflow by pulling the latest Git commit on boot, compile it and run it. With this it would be possible to use the vehicle completely headless. Of course there needs to be a fail safe for pulling when there is no WiFi connection. A RGB led on the board could be used to display the current state (e.g. yellow: pulling, blue: building, red: build failed, green: running). The build log should be captured to a log file in case of a failed build. With this workflow project members would be able to develop locally with a mere text editor and git. To deploy to a vehicle, the local state just needs to be pushed and the vehicle restarted. This could be extended further by using a git branch for each vehicle or triggering an update from the base station.
- Forward console output to base station to have debug information in one place, removing the need for a SSH connection.
- Increase the motor supply voltage by adding more cells in series. This would possibly grant more movement accuracy.
- Switch between vehicle modes more intelligently and add new modes to get more distinct control over the vehicle.
- Implement dynamic tiling, where at first, big tiles are used to find objects and smaller tiles are only created to get more precise location information. Tile sizes and locations can be carried over to the next frame, as it is likely to find the object again in the vicinity.
- Update PynqLinux and all libraries, this can aid system stability and maybe even speed.
- Look for a faster way to transfer data to the PL, right now *copyBufferHostToAccel* takes some time.

References

[1] *Applications of monolithic bridge drivers*. AN240/1288. SGS-Thomson Microelectronics. 1995.

[2] *AXI Reference Guide*. UG761. Xilinx. 2011.

[3] Matthieu Courbariaux et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. arXiv: 1602.02830 [cs.LG].

[4] *Digilent Pynq-Z1 Product Page*. URL: https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/ (visited on 12/23/2020).

[5] *Elegoo Smart Robot Car Kit V3.0.2019.10.12.zip*. https://www.elegoo.com/pages/arduino-kits-support-files or https://drive.google.com/file/d/1nSlkYJ7oCfMkG1p-KDfVHdLQt3B4Nmo5/view?usp=sharing. (Visited on 03/30/2021).

[6] *FINN - Fast, Scalable Quantized Neural Network Inference on FPGAs (GitHub)*. URL: https://github.com/Xilinx/finn (visited on 03/30/2021).

[7] *FreeRTOS*. URL: https://freertos.org/ (visited on 12/23/2020).

[8] *GCC*. URL: https://gcc.gnu.org/ (visited on 12/23/2020).

[9] Colin von Huth et al. *Bericht zum Projekt ‚Neuronale Netze auf strahlungstoleranten FPGAs für die Raumfahrt'*. German. Bremen, Feb. 14, 2020. 88 pp. URL: http://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/neuronale-netze-fpgas-projekt-1920.html (visited on 02/15/2021).

[10] Jasminka Matevska et al. "CRUISE – Evaluating Enhanced Crew Autonomy Concepts On-Board the ISS as a Preparation for Future Long Term Crewed Space Missions". In: *Proc. of DASIA 2014 Data Systems in Aerospace* (Warsaw, Poland, June 3–5, 2014). Ed. by L. Ouwehand. ESA Special Publication ESA SP-725. id.56. Aug. 2014. ISBN: 978-92-9221-289-6. URL: https://ui.adsabs.harvard.edu/link_gateway/2014ESASP.725E..56M/ADS_PDF (visited on 02/15/2021).

[11] Gerd-J. Menken. *Mechatronik 1 - Teil III. Regelung dynamischer Systeme*. Vorlesungsskript, Hochschule Bremen, SoSe 2020, 2020.

[12] *OpenCV documentation*. URL: https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html#ga5bb5a1fea74ea38e1a5445ca803ff121 (visited on 03/29/2021).

[13] *PetaLinux Tools Documentation (UG1144)*. Xilinx, 2020.

[14] *Pynq*. URL: http://www.pynq.io/ (visited on 12/23/2020).

[15] *SDL2 - About SDL*. URL: https://www.libsdl.org/index.php (visited on 01/05/2021).

[16] J. Stallkamp et al. "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition". In: *Neural Networks* 32 (2012). Selected Papers from IJCNN 2011, pp. 323–332. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2012.02.016. URL: https://www.sciencedirect.com/science/article/pii/S0893608012000457.

[17] Yaman Umuroglu et al. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. ACM, 2017, pp. 65–74.

[18] *Xilinx Vitis announcement*. URL: https://www.xilinx.com/news/press/2019/xilinx-announces-vitis--a-unified-software-platform-unlocking-a-new-design-experience-for-all-developers.html (visited on 12/23/2020).

[19] *Xilinx Vitis Software Platform*. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html (visited on 12/23/2020).

A Illustrations

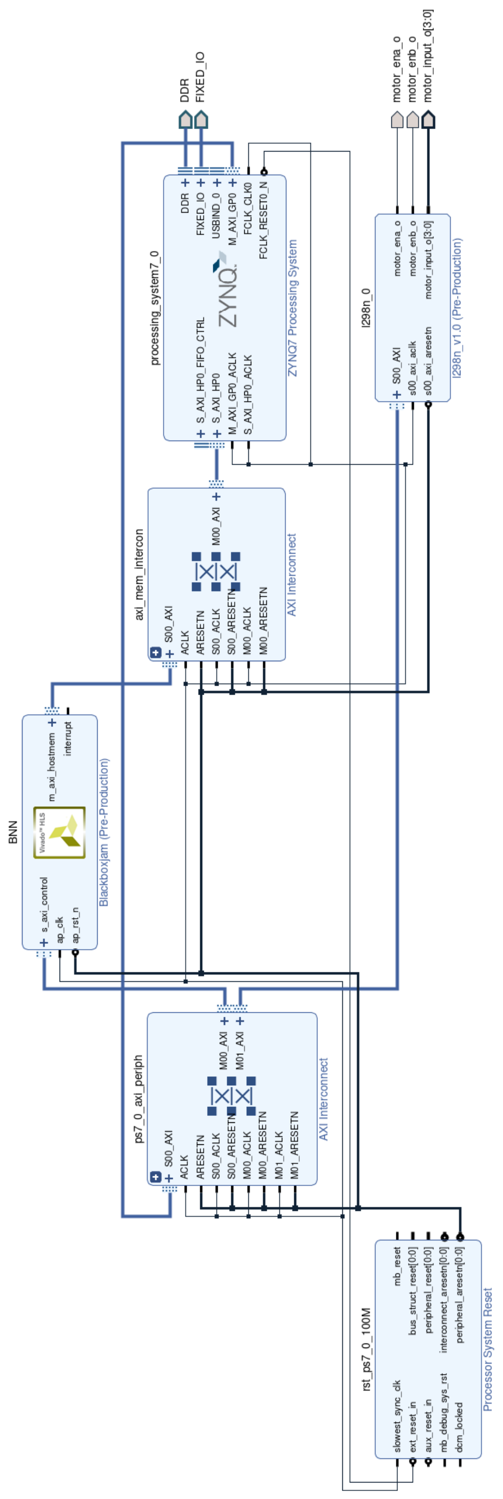


Figure 14: Block design of the "Programmable Logic" subsystem

B Hardware Registers

Legend:

- W: Writable Value
- R: Readable Value

Register B.1: Motor Control Registers (0x43C20000)

Reserved	IN4	IN3	IN2	IN1	
31 … 4	3	2	1	0	
-	W	W	W	W	Offset 00h

Reserved	Speed Left	
31 … 10	9 … 0	
-	W	Offset 04h

Reserved	Speed Right	
31 … 10	9 … 0	
-	W	Offset 08h

Reserved	
31 … 0	
-	Offset 0Ch

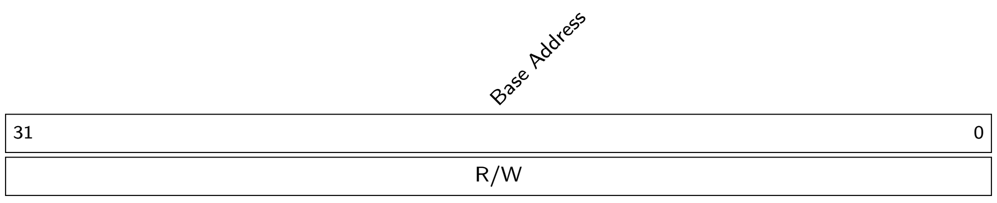
Register B.2: BNN-Pynq Control Signals Register (0x43C00000)

Reserved	Auto Restart	Reserved	Ready	Idle	Done	Start
31 … 8	7	6 … 4	3	2	1	0
-	R/W	-	R	R	R	R/W

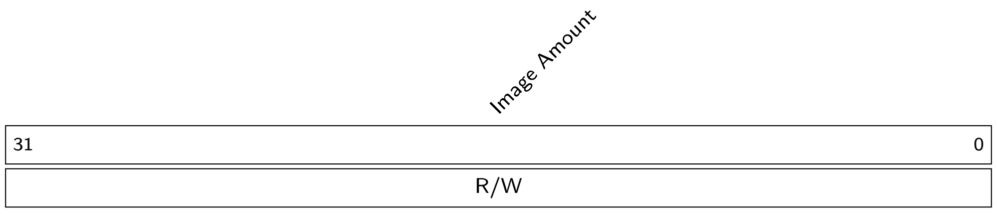
Register B.3: BNN-Pynq Input Data Address Register (0x43C00010)

Base Address
31 … 0
R/W

Register B.4: BNN-Pynq Output Data Address Register (0x43C0001C)



Register B.5: BNN-Pynq Number of Repetitions Register (0x43C0005C)



C Network Dataframes

C.1 Vehicle to Base Station

Dataframe C.1: Control Dataframe (Vehicle to Base Station)

Inferred Y Position (High Byte)	Inferred Y Position (Low Byte)	Inferred X Position (High Byte)	Inferred X Position (Low Byte)	Inferred Class	Frame Processing Time (High Byte)	Frame Processing Time (Byte 2)	Frame Processing Time (Byte 1)	Frame Processing Time (Low Byte)	Motor Right Speed (High Byte)	Motor Right Speed (Low Byte)	Motor Left Speed (High Byte)	Motor Left Speed (Low Byte)	Motor Direction	Motor Control Mode	Transmitted Tile	Tile Count	Video Feed Enable	Color Mode	Image Height (High Byte)	Image Height (Low Byte)	Image Width (High Byte)	Image Width (Low Byte)	Frame ID
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																							0

C.2 Base Station to Vehicle

Dataframe C.2: Set Resolution Dataframe (Base Station to Vehicle)

Height (High Byte)	Height (Low Byte)	Width (High Byte)	Width (Low Byte)	Frame ID
4	3	2	1	0
				0

Dataframe C.3: Set Color Mode Dataframe (Base Station to Vehicle)

Color Mode	Frame ID
1	0
	1

Dataframe C.4: Set Video Feed Enable Dataframe (Base Station to Vehicle)

Video Feed Enable	Frame ID
1	0
	2

Dataframe C.5: Set Displayed Tile Dataframe (Base Station to Vehicle)

Tile Index (High Byte)	Tile Index (Low Byte)	Frame ID
2	1	0
		3

Dataframe C.6: Set Motors Dataframe (Base Station to Vehicle)

Speed Right (High Byte)	Speed Right (Low Byte)	Speed Left (High Byte)	Sped Left (Low Byte)	Direction	Manual Mode	Frame ID
6	5	4	3	2	1	0
						4

D Building and Running the Subystems

written by Felix Müller

You always need to install the PS because this is the software that controls the vehicle. The PL project only needs to be touched if you want to make changes to the logic on the FPGA. The compiled Bitstream is included in the PS repository and will be loaded onto the FPGA on startup of the PS. If you want to debug the state of the PS while its running (beyond the console output) you have to install the BS project.

D.1 Make

Both the PS and BS projects use Makefiles for their build scripts, which detects changes in the source files and only executes compilation when files have changed.

D.1.1 Targets

Make allows the definition of multiple targets that have some output file(s) and can depend on other targets. The syntax run a target ist the following:

make [Target]

The PS and BS projects contain (at least) the following targets:

- **all:** Creates all binaries to run the application
- **run:** Runs the application (depends on all, therefore if source files changed, the application will be automatically recompiled before startup)
- **clean:** Deletes all generated files

Both projects place their generated files in a directory called *build* in the same directory as the Makefile.

D.2 PS

Prerequisites:

- Pynq-Z1 board with PynqLinux installed
- Access to the repositories or a snapshot

All build dependencies are provided with PynqLinux.

Installation:

1. Clone the repository or use a snapshot
2. Clone the submodules (which are all in public repositories maintained by Xilinx):

   git submodule update –init –recursive

3. Build the application:

   make all

4. Run the application:

   make run

D.3 PL

Prerequisites:

- Vivado (at least version 2020.2)

Installation:

1. Clone the repository or use a snapshot
2. Open *src/vehicle.xpr* in Vivado
    - When using Linux, the following env-variables should be set to avoid localization errors:
        - **LC_NUMERIC**=en_US.UTF-8
        - **LC_ALL**=en_US.UTF-8
3. Run the target *Generate Bitstream* in the left sidebar
4. Wait for the task to complete
5. Copy the generated files to the PS project:
    - **PL**/src/vehicle.runs/impl_1/vehicle_wrapper.bit ⟶ **PS**/hw/vehicle.bit
    - **PL**/src/vehicle.gen/sources_1/bd/vehicle/hw_handoff/vehicle.hwh ⟶ **PS**/hw/vehicle.hwh
6. Update ***PS****/hw/peripherals.h* if you changed the address of some AXI peripheral
    - An address list can be found in the address editor in Vivado
    - The address of the BNN (BlackboxJam) **cannot** be changed because it is embedded in the library by Xilinx

D.4 BS

The base station application was developed for Linux. Some changes may be required to run it on another OS.

Dependencies:

- make
- gcc
- sdl2

Installation:

1. Clone the repository or use a snapshot
2. Build the application:

    make all

3. Run the application:

    make run

E Vehicle Network Configuration

The Pynq-Z1 boards have an Ethernet connector and should be connected to a USB WiFi adapter when mounted on a model vehicle. A connection through Ethernet offers more stability, especially when using the included Samba share to access and edit files. WiFi on the other hand should be used when the vehicle is driving.

E.1 Ethernet

The Ethernet interface is configured with DHCP by default, it therefore should be assigned an IPv4 address automatically. In a fairly standard network configuration, connecting to the board via SSH can be done by using the host name 'pynq':

ssh xilinx@pynq

If this does not work, the host name should be replaced with the IPv4 address. This can be found using any of the following methods:

- Running *arp -a*
- Using *nmap* command
- Looking up connected clients on the standard gateway/router

E.2 WiFi

Since the Pynq-Z1 is preferably run headless, command line tools *ifup, ifdown* and *wpa_supplicant* can be used. To confirm that the WiFi adapter has been detected, any of these commands can be run:

iwconfig
ifconfig -a

This should show the interface *wlan0*. To add a new WiFi network, its SSID needs to be known:

sudo iwlist wlan0 scan

With the SSID and passphrase, this command will return the pre-shared key (*psk*) for the network:

wpa_passphrase <SSID> <PASSPHRASE>

Copy the string after *psk* and open the interface setup file:

sudo nano /etc/network/interfaces.d/wlan0

This file should look like the following, where <SSID> and <PSK> are replaced with the concrete values determined before. Including *'auto wlan0'* ensures that the system will try to reconnect to the network when booting.

```
auto wlan0
iface wlan0 inet dhcp
 wpa-ssid <SSID>
 wpa-psk <PSK>
```

The shortcuts *Ctrl+S* and *Ctrl+X* can be used to save and exit the file. After restarting the network service by running

sudo /etc/init.d/networking restart

the interface *wlan0* should be in state UP and have an IPv4 address assigned, both can be checked by running *ifconfig wlan0*. The LEDs on the WiFi adapter should also show activity.

F Working Together Remotely

One Pynq-Z1 board was made accessible through a VPN, to enable the project members to work on-chip remotely. While this already simplified software development, debugging was made possible by also publishing the base station on the local network. A Raspberry Pi 4 was used as host for the VPN Docker container and the base station.

These steps were followed to achieve a suitable network setup:

1. Acquire Dynamic DNS service (domain pointing to public IP)
2. Configure edge router for dynamic DNS (updates DDNS with its public IP)
3. Set up VPN Docker container on a host in same network as Pynq-Z1 (guides can be found online)
4. Forward the ports from the router to Docker host and from there to container
5. Install the base station as documented in subsection D.4, configure vehicle with the host's IP
6. Configure ssh server on base station host to allow X11 forwarding

Project members can, with the right credentials, connect to the VPN. It should then be possible to access the Pynq-Z1 and base station through SSH the same way as directly from within the local network. The Pynq-Z1's Samba share is now also available for easy file access, as well as graphical output and keyboard input for the base station. It is necessary to use one of the command line options *'-X'* or *'-Y'* when opening an SSH connection to the base station host, to accept X11 forwarding.