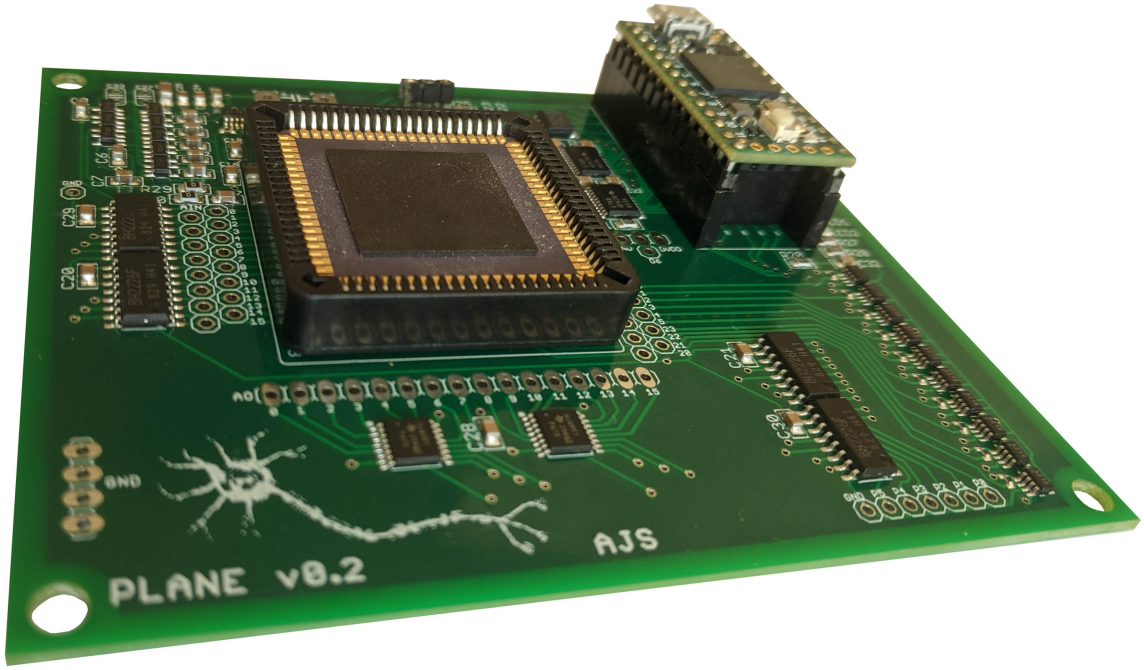
Interfacing Neuromorphic Hardware with a Microcontroller
NSC Master's Thesis

Alex Schwalb

August 9, 2020



University of Zurich[UZH] | ETH zürich

Supervisor ..... Date .....

Contents

i Abstract

Interfacing with neuromorphic devices is classically done with an FPGA due to the speed requirements. Although FPGAs are very costly in terms of power, development time and price. This became particularly problematic for a Neuromorphic Engineering class at ETH due to the COVID-19 pandemic, as each student now needed their own neuromorphic hardware to work from home. Previously a neuromorphic classchip named CoACH was developed, but buying dozens of FPGAs to interface with them would be very expensive and programming them would be very time consuming.

With the recent advances in microcontroller technologies allowing them to run at much higher clock frequencies, this might be a viable alternative. In this project we investigate the feasibility of using a microcontroller, namely the Teensy 4.0, to interface with neuromorphic devices while simultaneously communicating with a computer over USB.

We created a library for Teensy boards to interface with neuromorphic devices using AER. We successfully interfaced with a DVS, reaching an event rate of up to 979kEPS. This is less than what an FPGA is capable of, but it is more than enough for most use cases. In addition to just reading AER events, the Teensy is also capable of processing the events. We showed this by implementing two small tasks: one which tracks the center of events, and one which tracks the location and frequency of a blinking LED.

We also developed a PCB, named PLANE, containing the Teensy, CoACH and all the necessary circuits to interface between them. To control this PCB from a computer, we created a GUI. It also displays the readings of the classchip in the form of an oscilloscope or raster plot.

We conclude that microcontrollers are in fact a viable alternative, although not yet as fast as FPGAs. However, we do propose a method which could significantly increase the speed at which the microcontroller samples AER events.

ii Acknowledgements

I would like to thank Giacomo Indiveri for supervising this masters thesis and Lottie Walch for helping with the administrative things. I would also like to thank Germain Haessig for his help and guidance with numerous parts of this project. I would also like to thank Adrian Whatley for his advice with the software, and Mohammadali Sharifshazileh and German Köstinger for their advice with the hardware. I would also like to thank Elisa Donati for going to the institute just to unlock a door for me and help me find a classchip at the start of the lockdown. I would also like to thank Shih-Chii Liu for being my mentor in the NSC program. Finally I would like to thank my parents, George and Ronell Schwalb, for supporting me throughout my studies.

iii List of abbreviations

ACK: Acknowledge
ADC: Analog Digital Converter
AER: Address Event Representation
CPU: Central Processing Unit
DAC: Digital Analog Converter
DVS: Dynamic Vision Sensor
EPS: Events Per Second
FPGA: Field Programmable Gate Array
GUI: Graphical User Interface
IC: Integrated Circuit
I2C: Inter Integrated Circuit
ISR: Interrupt Service Routine
kEPS: kilo Events Per Second
LED: Light Emitting Diode
LSB: Least Significant Bit
PC: Personal Computer
PCB: Printed Circuit Board
PLANE: Platform for Learning About Neuromorphic Engineering
REQ: Request
SPI: Serial Peripheral Interface
USB: Universal Serial Bus

Chapter 1

Introduction

Neuromorphic engineering is the field that mimics biological neurons with electronic circuits[3]. There are two reasons why this is useful: it can help us learn about the brain, and it can provide novel circuits that perform well at certain tasks.

When compared to a conventional CPU simulating neurons, neuromorphic processors have been found to run two orders of magnitude faster and consume three orders of magnitude less power[7]. Some neuromorphic sensors also lead to a very sparse data representation, and some neuromorphic processors have online learning and real time computation capabilities[4][5].

However the asynchronous and often analog nature of neuromorphic devices require some intermediate system when interfacing with most digital systems, such as a desktop computer. This is classically done with Field Programmable Gate Arrays (FPGA), due to the requirement for very high speed sampling of asynchronous events. However FPGAs are very costly in terms of power, development time and price.

Recent developments in microcontrollers allow them to operate at very high clock frequencies, which make them a viable alternative to FPGAs. In this project we investigate this possibility with the Teensy 4.0, which operates at 600 MHz with an ARM Cortex-M7 processor[6]. The Teensy boards are also compatible with the Arduino platform, which makes developing with them very fast and easy.

In addition to lower cost and power consumption, and shorter development time, microcontrollers are also capable of doing some preprocessing on the data read from neuromorphic sensors. Although, strictly speaking FPGAs are also capable of this, the processors on microcontrollers allow them to do much more elaborate operations with much less effort from the programmer.

In this project, we first explain how to generically interface with neuromorphic devices using a Teensy in Chapter 2, then we provide two examples of devices we interfaced with.

In Chapter 3 we interface with a Dynamic Vision Sensor (DVS)[2]. This is a neuromorphic camera based on the biological eye. It has many photoreceptor circuits which detect changes in light intensity and outputs the location of these changes asynchronously as events. This has the benefit over frame based cameras of being a much more sparse data representation and it also does not suffer from motion blur. We use the Teensy to read these events and send them to a computer via USB.

In Chapter 4, we interface with the CoACH neuromorphic classchip. This is a chip that was developed for educational purposes to be used in a Neuromorphic Engineering class at ETH, Zurich. This chip contains 30 test circuits that are interfaced with using analog voltages and currents, as well as asynchronous digital events. We develop a PCB that contains the classchip and Teensy, as well as all the intermediate components required to interface them with each other.

Since this report contains many small tasks, we did not include a background section in the beginning and a results section in the end. Instead we provide the necessary background and obtained results for each

task in the sections describing that task itself. Also, since this project is more about development than research, the methods and results aren't completely separable. We try out different methods to solve certain problems, and modify the method based on the results.

Chapter 2

Teensy AER Interface

In this chapter we discuss how to interface with a Teensy and a neuromorphic device using AER. After introducing AER, we explain how to read events from a neuromorphic device, and then we explain how to best send these events to a computer over USB. Finally we explain how to use the library we made.

1 Address Event Representation

AER is a four phase hand-shaking protocol. The four phases are:

- When a master wants to send data to a slave, it sets all the data pins to their correct values. Once they are set, the master makes the request pin active.
- When the slave sees this request pin is active it reads the data pins. Once it has read everything, it makes the acknowledge pin active.
- Once the master sees the acknowledge pin is active, it is not obligated to drive the data pins with their correct values anymore. It then makes the request pin inactive again.
- When the slave notices this, it makes the acknowledge pin inactive.

The request and acknowledge pins were measured with an oscilloscope and are plotted in Figure 2.1. For this figure we used a DVS as the master and a Teensy as the slave.

2 Reading pins on Microcontroller

The first step in interfacing between a neuromorphic device and a Teensy is to get the Teensy to read the signals from it. Arduino has built in functions for reading pins, but they can be slow. They sacrifice speed for user friendliness. These functions include *digitalRead()* and *digitalReadFast()*. Both take the pin one wishes to read as an argument and return its state.

A faster way to read pins is through direct port manipulation. This means manually reading the internal register that is connected to the pin and masking it with the appropriate bit mask. The *digitalReadFast()* function does do this, but contains many conditionals in order to make the code reusable for different pins.

We measured the time taken to read eight pins and store their values in a byte by using *digitalRead()*, *digitalReadFast()* and direct port manipulation. We did this by setting a dedicated pin high just before

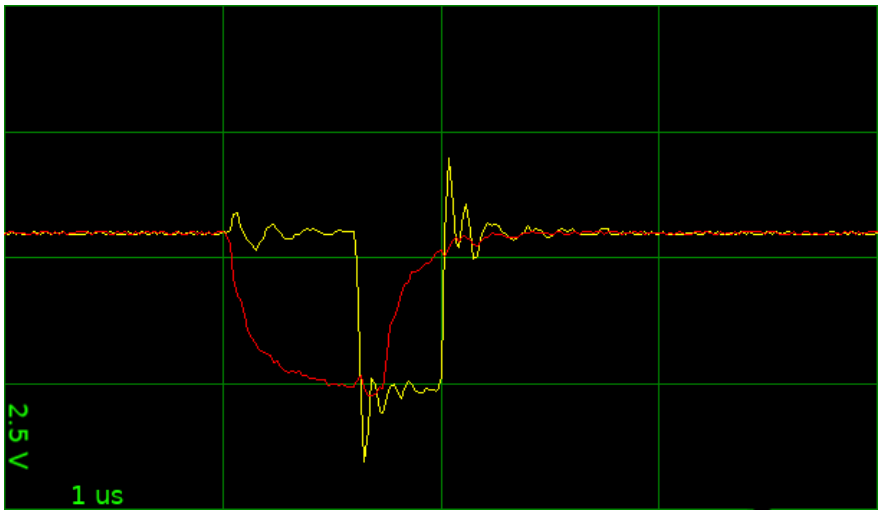


Figure 2.1: Example of AER handshake signals. The request signal is shown in red and the acknowledge signal is shown in yellow. Both signals are active low.

	digitalRead()	digitalReadFast()	direct port manipulation
time (ns)	350	140	135

Table 2.1: Time required to read eight pins using *digitalRead()*, *digitalReadFast()* and direct port manipulation.

starting the read, and setting it low again afterwards and measuring the pulse width with an oscilloscope. The results are shown in Table 2.1

Direct port manipulation is only slightly faster than *digitalReadFast()*, so it is generally recommended to rather use *digitalReadFast()* for code readability and reuseability, except when speed is critical. When interfacing with the DVS in Chapter 3 we use direct port manipulation to achieve a high event rate, but for the classchip in Chapter 4 we used *digitalReadFast()*.

3 Interrupt Service Routine

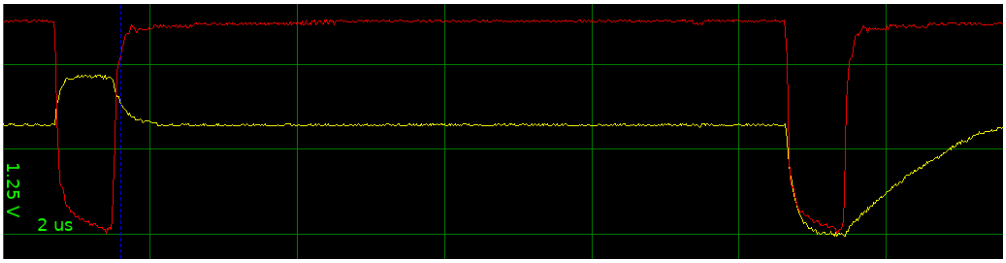
AER is an asynchronous protocol, so the neuromorphic device could try to send an event to the Teensy at any time. Therefore we require interrupts. Interrupts tell the processor some event occurred that needs immediate attention, so the processor pauses whatever it is busy with and runs the Interrupt Service Routine (ISR).

All the Teensy pins are interrupt enabled. We trigger an interrupt with the request pins of the AER channel coming into the Teensy. In the ISR, we read the data pins and either record them and their timestamps into a buffer, or we increment the rate corresponding to the address of the event.

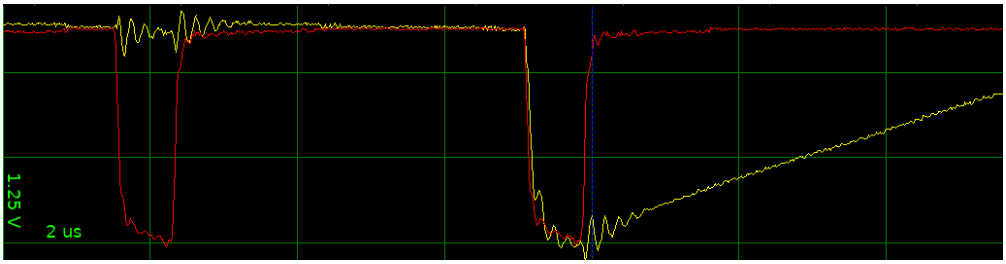
4 Pin 13

On the Teensy, pin 13 is connected to an LED. This causes that pin to draw more current when driven high. This could be problematic for some neuromorphic devices if they cannot provide sufficient current. The DVS

discussed in Chapter 3 is not capable of supplying the necessary current to drive that pin all the way up to $3.3V$, as shown in Figure 2.2. Fortunately the DVS still drives it high enough to be read as a logic 1. For this project it is actually quite useful for debugging, as one could easily see if there is communication between the Teensy and DVS by just looking at the activity of the LED. For the classchip in Chapter 4 pin 13 is an output, so the current to the LED is provided by the Teensy.



(a) Pin 13



(b) Pin 14

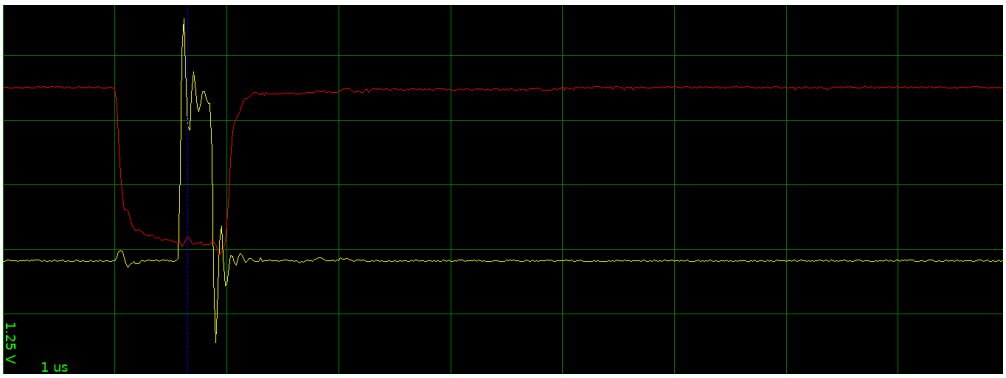
Figure 2.2: Voltage signals on pins 13 and 14 are shown in yellow for a logic high event followed by a logic low event. The request signal is shown in red. The discrepancy between the pins is due to pin 13 being connected to an LED, which causes it to draw more current.

5 Packets

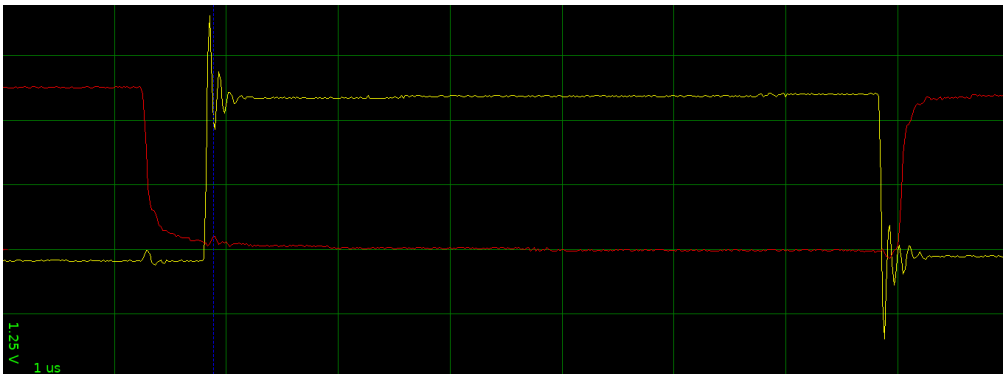
Instead of sending the events to a computer as they come, the events are stored into a buffer and then the whole buffer is transmitted once it is full. To show speedup of sending 64 bytes together, we measured the required time of calling *usb_serial_write()* 64 times with one byte, or once with 64 bytes. The oscilloscope reading is shown in Figure 2.3. Sending a single byte 64 times requires $6\mu s$ of processor time, but sending one chunk of 64 bytes requires only $0.3\mu s$, yielding an 20x increase in speed. Note that these values were measured when there were no programs running on the computer's side to read the packets.

The Teensy is fastest when transmitting packets that are a multiple of 64 bytes. We tested the transmission speed for different packet sizes and plotted this in Figure 2.4. Sending one packet of size 64 bytes requires $0.3\mu s$, and then another $0.2\mu s$ is required for any additional 64 bytes sent. So it is assumed that there is an overhead of $0.1\mu s$, and each 64 bytes require $0.2\mu s$.

Therefore, the more bytes are sent per packet, the less the relative cost of the overhead. However greater packet sizes also cause longer waiting times between packets and the bigger the packet size, the smaller the returns of increasing it even more.



(a) Sending one chunk of 64 bytes



(b) Sending 64 single bytes

Figure 2.3: The yellow signal goes high while transmitting the bytes over USB. The request signal is shown in red. Sending a single byte 64 times takes $6\mu s$, while sending one chunk of 64 bytes takes only $0.3\mu s$.

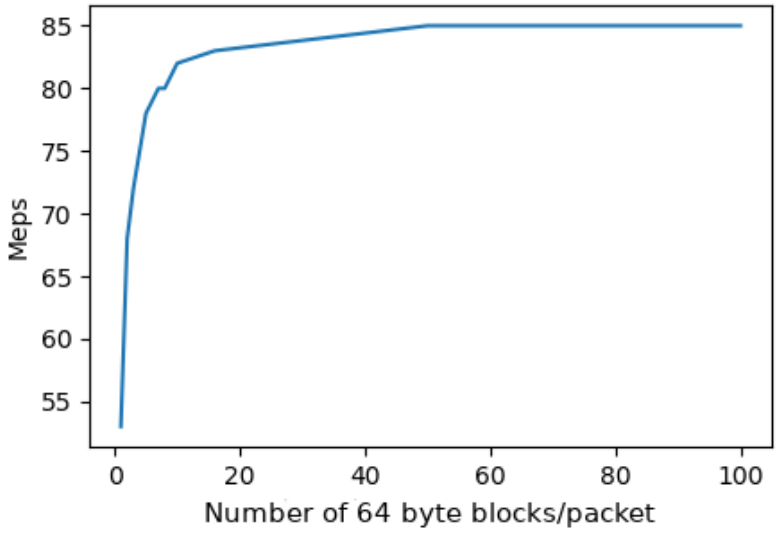


Figure 2.4: The maximum event rate that can be sent over USB is plotted against the number of 64 byte blocks per packet. It saturates at around 16 blocks, which is equivalent to 1024 bytes. This is the packet size we choose in Chapter 3.

6 Teensy AER Library

We created a library consisting of two files, *aer.h* and *aer.cpp*, that can be used to add AER input and output channels to a Teensy. It contains two classes, *AER_in* and *AER_out*, which are used for input and output channels respectively.

This library uses the *digitalWriteFast()* function to drive pins high or low. If direct port manipulation is to be used in order to achieve a higher event rate, the developer would have to manually modify the library and hardcode the ports. If this library is to be used with a board other than any of the Teensy boards, the *digitalWriteFast()* function would likely not work and should be replaced with *digitalWrite()*. Likewise for *digitalReadFast()*.

6.1 AER_in class

The *AER_in* class constructor has nine parameters listed below:

- **reqPin:** Integer indicating request pin.
- **ackPin:** Integer indicating acknowledge pin.
- **dataPins:** Array of integers indicating data pins.
- **numDataPins:** Integer indicating how many data pins there are.
- **buff:** Pointer to an array of bytes used to store events before transmitting them over USB. Defaults to NULL.

- **numTimestampBytes:** Integer indicating the number of bytes required to store the event timestamps. Defaults to two.
- **d:** Integer indicating delay in microseconds after changing acknowledge pin. Defaults to zero.
- **timestampShift:** Integer indicating the number of right bit shifts to perform on the timestamps. Each bit shift halves the resolution but doubles the time until overflow of the timestamps. Defaults to zero, which sets the resolution to one microsecond.
- **activeLow:** Boolean to indicate if the AER channel is active low. Defaults to true.

The public functions of *AER_in* include:

- **reqRead():** Returns the state of the request pin.
- **ackWrite(bool val):** Sets the acknowledge pin to *val*.
- **dataRead():** Returns the data pins as an unsigned integer.
- **dataRead_handshake():** Returns the data pins as an unsigned integer while taking care of handshaking.
- **record_event():** Records the current state of the data pins as well as the current time as an event into a buffer.
- **record_event_handshake():** Records the current state of the data pins as well as the current time as an event into a buffer while taking care of handshaking.
- **send_packet():** Transmits the contents of the buffer over USB.
- **get_index():** Returns the index indicating how full the buffer is.
- **set_index(int x)):** Sets the index indicating how full the buffer is to *x*.

6.2 AER_out Class

The *AER_out* class constructor has six parameters listed below:

- **reqPin:** Integer indicating request pin.
- **ackPin:** Integer indicating acknowledge pin.
- **dataPins:** Array of integers indicating data pins.
- **numDataPins:** Integer indicating how many data pins there are.
- **d:** Integer indicating delay in microseconds after changing request pin. Defaults to zero.
- **activeLow:** Boolean to indicate if the AER channel is active low. Defaults to true.

The public functions of *AER_out* include:

- **ackRead():** Returns the state of the acknowledge pin.
- **reqWrite(bool val):** Sets the request pin to *val*.
- **dataWrite(unsigned int data):** Writes *data* to the data pins.
- **dataWrite_handshake(unsigned int data):** Writes *data* to the data pins while taking care of handshaking.

Chapter 3

Dynamic Vision Sensor

In this chapter we discuss our implementation of an interface between a DVS and a Teensy. It is not a direct interface, as the DVS has an internal FPGA between the camera and AER output. However it is still useful in exploring the possibility of using microcontrollers to read AER events. We first cover the methods we used to design the system as well as the resulting performance. We then demonstrate the benefit of using a microcontroller to interface with the DVS by implementing two small tasks in which the microcontroller does some preprocessing on the DVS data.

1 Reverse Engineering the DVS Pinout

We knew that the data pins of the DVS consisted of seven bit x and y coordinates, a polarity pin and one extra pin, but we did not know exactly which physical pins correspond to each of these. We measured the activity on each pin for a known stimulus to reverse engineer the pinout.

The stimulus was a horizontal or vertical line sliding across the field of vision with a stationary background. We simply did this by moving a pen in front of the camera by hand. When a vertical line moves horizontally, only the x coordinates show the position of the line while the y coordinates are spread over the entire range of the line, and vice versa when a horizontal line moves vertically.

The most significant bit changes when the line crosses the centre of the screen, the second most significant bit changes every quarter and so on. So to identify which pins encode which bits we averaged every batch of 500 consecutive events and sent that value to the computer to plot in real time as shown in Figure 3.1. We counted the oscillations for each pin as the line moved across the screen once. This was difficult for some of the less significant bits, but since the location of the other pins made a pattern we could extrapolate the rest of the pins from that. Figure 3.2 shows what we suspect the DVS pinout to be. The numbers outside the squares show which pins of the Teensy we connected them to.

2 Time Resolution

Each event contains 16 bits encoding the time of that event. It is a binary number that increments every $1024\mu s$ and overflows about every 67 seconds.

This value is found by calling the *micros()* function whenever an event is read from the DVS and performing ten right bit shifts on the return value. The reason we shifted this value is to increase the time before overflow. The reason we called *micros()* instead of *millis()* was because *millis()* uses interrupts and cannot be called within another ISR.

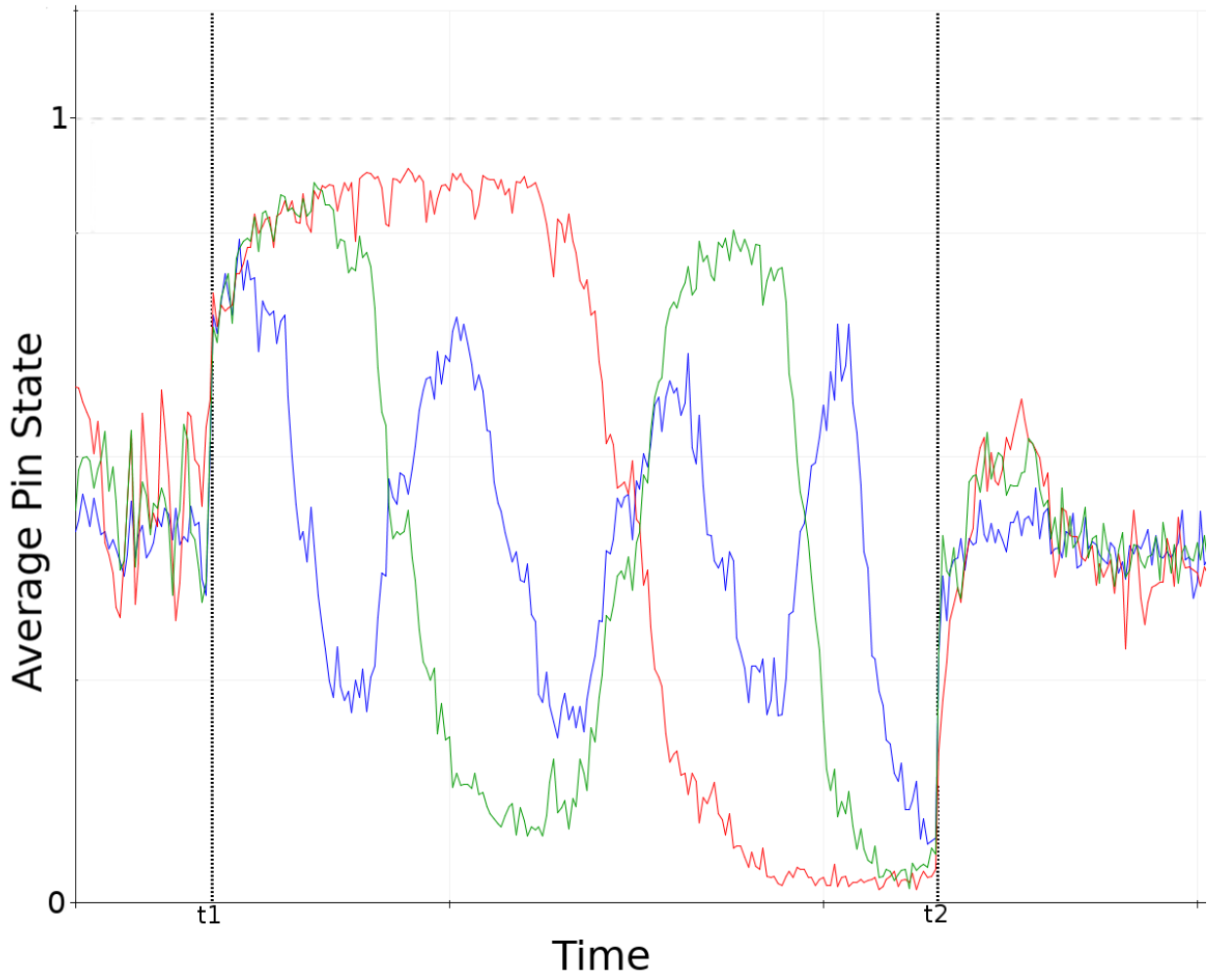


Figure 3.1: Activity of three most significant pins of X. From t_1 to t_2 a vertical bar is moving from left to right on the screen. The red signal has one wavelength, so it is the most significant bit. The green signal has two wavelengths, so it is the second most significant bit. The blue signal has four wavelengths, so it is the third most significant bit.

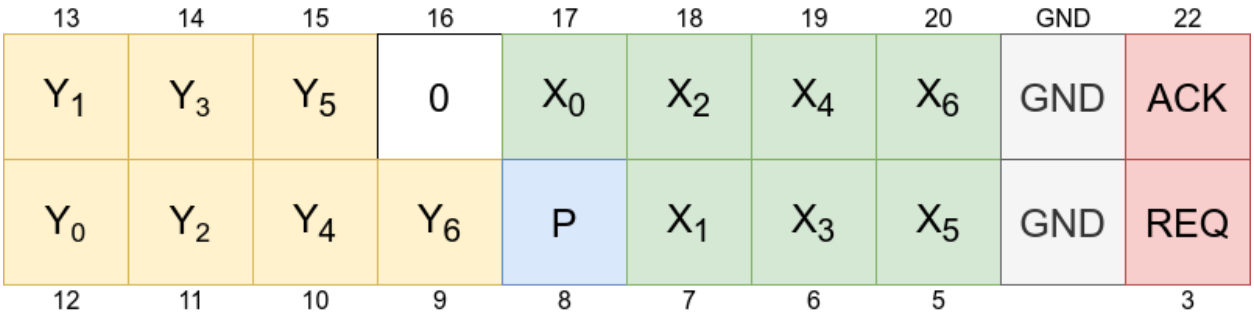


Figure 3.2: Pinout of the DVS. The subscript of 0 represents the least significant bit and the subscript 6 represents the most significant bit. P is the pin indicating the polarity and the 0 pin just outputs $0V$ whenever the data is valid. This is to be differentiated with GND, which is $0V$ all the time. ACK and REQ are acknowledge and request respectively.

3 Throughput Bandwidth

Each event requires four bytes: one for each of the x and y coordinates and two for the timestamp. Based on Figure 2.4 we decided that a packet size of 1024 bytes, or 256 events, seem reasonable. For every $256th$ event, we transmit one packet within the ISR reading that event. The ISR lasts about $1\mu s$ when only reading the events, and the transmission of one packet takes $5\mu s$(when there is a program on the computer's side listening). So now the ISR usually last $1\mu s$, except every 256th time it is called it will last $6\mu s$. This makes the average time per event $\frac{(255\times1)+(1\times6)}{256} = 1.02\mu s$. This results in a maximum expected event rate of 981 kEPS.

We measured the bandwidth on the computer's side with the program described in Section 4 that displays the DVS video. It is calculated by multiplying the number of events in each $20ms$ time window by 50 to convert it to EPS. The maximum bandwidth we measured was 979 kEPS, which is very close to the expected maximum of 981 kEPS. With the FPGA interface the DVS can achieve a higher event rate. We measured about 1.5 kEPS, but it could possibly go much higher with enough stimulus. Although under normal conditions 981 kEPS is more than enough.

One of the main bottlenecks limiting the bandwidth is to individually read each of the data pins. If a higher bandwidth is required, it is recommended to use a microcontroller in which the pins are mapped adjacent to each other on the same internal register. This would allow a single read operation to read all those pins simultaneously. Reading the DVS data pins would thus require one read operation instead of 15, which would significantly increase the speed.

4 Display

A program was written in C++ to visualise the data on the computer. It reads the data from the Teensy over USB and then displays it in a window using OpenCV. Some screenshots are shown in Figure 3.3 and 3.4. On events are displayed in red and off events in green on a black background. On the top left it displays the current event rate and on the top right it displays the maximum event rate achieved during the current session. It updates every 20 milliseconds, displaying all events that happened within the previous 20 milliseconds. Video demonstrations can be seen at `https://youtu.be/8xltupYo0MU`.

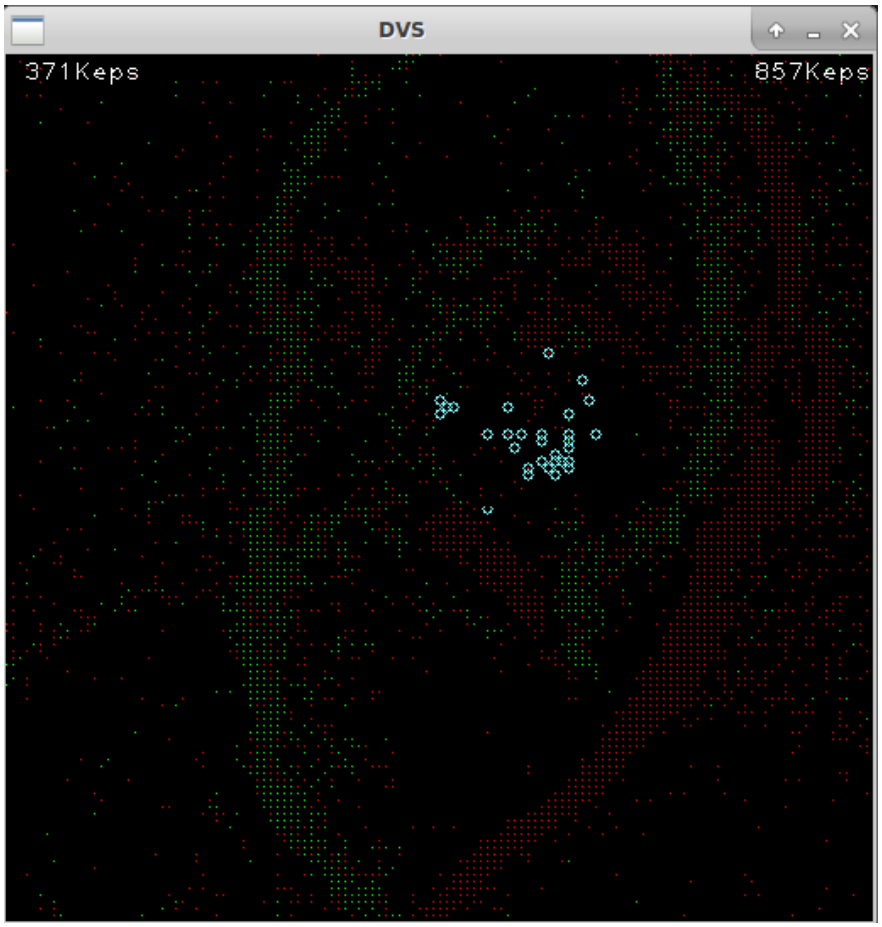


Figure 3.3: A screenshot of the DVS display while tracking the center of events. The on events are displayed in red and the off events in green. The center of events is shown by the circles. On the top left the current event rate is shown and on the top right the maximum event rate achieved in the current session is shown.

5 Center of Events

One of the benefits of using a microcontroller to interface with the DVS is that the microcontroller can already do some processing on the data. We demonstrated this by having it calculate the center of events. Within every packet, the Teensy sends one special event indicating the center of all the events in that packet. This center is calculated by averaging the x and y coordinates of all the other events in the packet.

To signify which event is the special event, we set a specific bit to high for the special event and low for all other events. We used the eighth bit of the byte indicating the y coordinate for this, since the coordinates only require seven bits to encode. In the display, these special events are shown as circles, as seen in Figure 3.3. There are multiple circles because there are multiple packets within each frame.

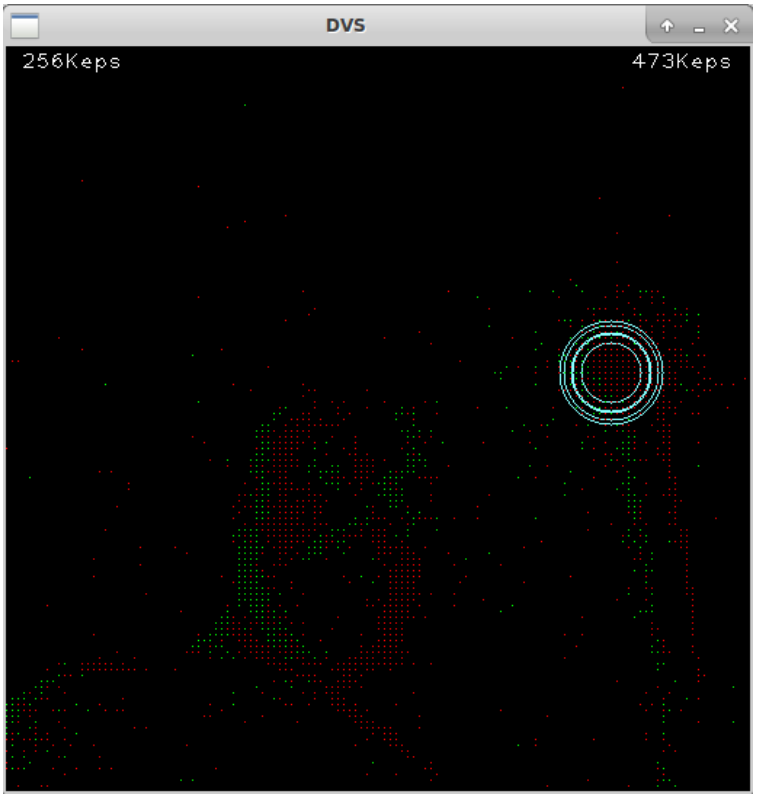


Figure 3.4: A screenshot of the DVS display while tracking the location of an LED. The circles show the location of the LED, and their sizes qualitatively show the frequency.

6 LED Tracker

Another benefit of using a microcontroller to interface with neuromorphic devices is the fast development time. We were inspired previous work[1] which used a DVS camera to track blinking LEDs and it could even distinguish their frequencies. Our interfaced microcontroller allowed us to implement something very similar with very little effort using only the processing power of the Teensy. Figure 3.4 is a screenshot of the display while tracking a blinking LED on a stick. The circles show the location of the LED and their sizes qualitatively show the blinking frequency. If accurate detection of the frequency is required, our system would have to be refined though.

In order to detect the location and frequency of LEDs, the Teensy keeps track of the event rates for each pixel in a 2D array. This is done by incrementing the rate corresponding to each event as it occurs, and resetting the rate whenever a packet is sent. Pixels that represent the LED would have very high event rates, so to detect the LED the Teensy simply looks for pixels with event rates above a certain threshold. The polarity of events are discarded, so the event rates would correspond to double the blinking frequency.

Packets are sent at a fixed interval containing the addresses of each pixel that likely represents part of an LED. The event rates are encoded into the bytes that would usually contain the timestamp. These events are encoded as special events as explained in Section 5. Their rates can be converted to EPS by multiplying with the rate at which the packets are sent.

Chapter 4

Classchip

A chip was developed for educational use in a Neuromorphic Engineering class offered by ETH. Due to the COVID-19 pandemic, a low cost interface for this chip was urgently needed. This could then be produced in large quantities to allow students to work individually, and the PCB could include all the necessary parts to allow students to work from home if required.

We first interfaced with the chip using a protoboard and readily available components to test some of the basic functionality. We then developed a PCB and purchased the components in order to have a more robust, user friendly and higher performance device.

In this chapter we first explain the communication protocols used between different elements on the PCB. Next we explain the various circuits required to interface with the classchip, both using the protoboard and PCB, and we compare the two approaches taken. Then we discuss the USB communication between the Teensy and the computer. Next we describe a GUI we created to control the PCB from a computer and finally we discuss the layout of the PCB.

1 Communication Protocols

Here we discuss the various communication protocols between the different ICs on the PCB. For details about the USB communication between the Teensy and a computer, refer to Sections 3.1 and 3.2. The Protocols used between the different ICs on the PCB are AER, I2C and SPI.

1.1 AER

There are three AER channels:

- **II:** This channel is an input to the classchip used to set biases, configure the chip and send pulses to the neurons. It uses the IIREQ and IIACK pins for handshaking and IID0 to IID10 for data.
- **C2F:** This channel is an output from the classchip. It converts currents from the internal test circuits to firing rates. It uses the C2FREQ and C2FACK pins for handshaking and C2FO0 to C2FO3 for data.
- **AERO:** This is another output channel. It uses the AERACK and AERREQ pins for handshaking and AERO0 to AERO2 for data.

All the pins mentioned above are shown in Appendix 1.

1.2 I2C

I2C is used to communicate with the expander and current sensors explained in Sections 2.3 and 2.5 respectively. In I2C only two pins are required, SCL and SDA, and both require pull up resistors. SCL is the clock and it is driven by a master. SDA is the data and can be driven by either a slave or a master.
There are various steps in a single I2C transaction. They are outlined in Table 4.1 and explained in further detail here:

- A transaction is initiated when a master generates a start condition. A start condition is indicated by a falling edge on SDA while SCL is high.
- The master sends a byte consisting of a seven bit address and a R/!W bit. When R/!W is low, it means the master wants to send data to the slave. When it is high, it means the master requests data from the slave.
- If a slave with that address exists on the bus it responds with an active low acknowledge signal. This means it drives SDA low for one clock cycle directly after the initial byte from the master.
- The data bytes are now transmitted either by the master or the slave depending on R/!W.
- After all the necessary data has been transmitted, a stop condition is caused to indicate the end of the transaction. A stop condition is indicated by a rising edge on SDA while SCL is high.

START	A_6	A_5	A_4	A_3	A_2	A_1	A_0	R/!W	ACK	data sequence	STOP

Table 4.1: I2C transaction. START and STOP are the start and stop conditions. A_0 to A_6 are the address bits and R/!W selects read or write. ACK is the acknowledge from the slave. *Data sequence* is the data that is either sent from the master to the slave or from the slave to the master. All the entries in the table represent single bits, except *data sequence* can be a variable number of bytes.

1.3 SPI

SPI is used to communicate with the DAC explained in Section 2.4. An SPI channel requires at least four pins, and for every additional slave it requires an additional pin. The MISO and MOSI pins are *master in slave out* and *master out slave in* respectively and are used to transmit the data. The SCK pin is the clock signal generated by the master. A separate CS pin is required for every slave device. This is an active-low *chip select* signal that the master uses to select which slave to communicate with.

2 Circuits

2.1 Power Supply

The classchip has eight positive power supply inputs requiring $1.8V$. This is to provide separate power to the analog, digital and pad circuitry for better noise performance, as well as to power each neuron circuit individually so that their power consumption can be measured. The power supply pins are DVDD, AVDD, ALLVDD and all the NCVDD pins as labeled in Appendix 1.

The Teensy does not have any $1.8V$ outputs, but it does have $3.3V$ and $5V$ outputs. The $5V$ is connected directly to the USB power line, and the $3.3V$ comes from a regulator on the Teensy.

On the protoboard, we built our own regulator as shown in Figure 4.1. It is essentially just a buffer with external output transistor to provide more current. The voltage supplied to the load follows the voltage at the positive input of the op amp. The $1.8V$ is obtained using a simple resistor divider. We powered everything using only one regulator, because for the purposes of the protoboard having low noise was not that important.

On the PCB we used eight store bought complete regulators. We chose the TPS77018DBVT regulators, because they are low cost and low power. They can only output 50 mA each, which is high enough to power everything it needs to, but also low enough to provide safety in case anything draws enough current to damage itself. This could happen if the classchip is configured incorrectly, if there is a short circuit in certain places, or if an incorrect analog voltage is applied to certain pins.

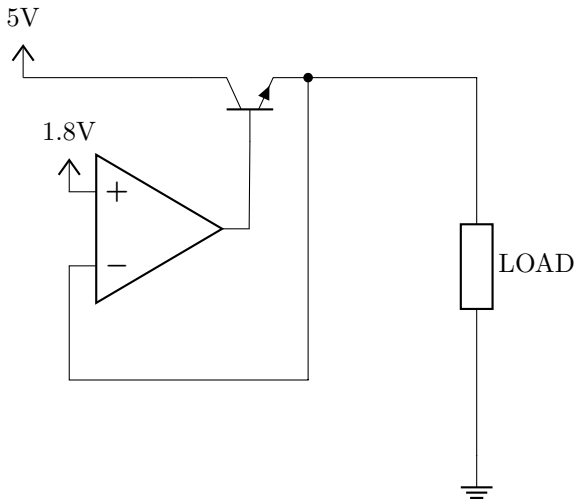


Figure 4.1: Regulator made from op amp and transistor.

The various power supplies on the classchip also have their own grounds so that the circuits can be completely isolated. These are the DGND, AGND and ALLGND pins as seen in Appendix 1. We connected them all to the same ground, sacrificing some noise performance for having a simpler PCB.

2.2 Level shifter

Since the classchip operates at $1.8V$ and the Teensy at $3.3V$, the voltage levels have to be shifted. On the protoboard, resistor dividers are used on the signals going from the Teensy to the classchip. Nothing is done to the signals going from the classchip to the Teensy, since $1.8V$ is high enough to be read as logic high by the Teensy. This does however decrease the error margin.

Using resistor dividers to shift the levels increases the time constant of toggling the pins. We used the resistor values shown in Figure 4.2 and measured a time constant of $0.5\mu s$. The time constant without the resistor dividers is too fast for the oscilloscope to measure.

This increase in time constant introduces a few problems. The first and most obvious problem is that it decreases the bandwidth. Another problem is that after the Teensy drives a pin to a certain logic level, it goes on running the next instructions assuming the pin is already at that level. So the Teensy might, for example, assume it is sending a logic high to the classchip while in fact the voltage is still low enough to be read as a logic low. To solve this, adequate delays are required in the code after every instruction that drives a level shifted pin high or low.

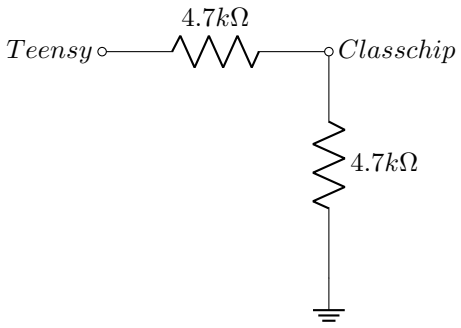


Figure 4.2: Resistor divider level shifter.

Another problem we observed from using a resistor divider is demonstrated in Figure 4.3, which shows the handshake signals of the C2F channel. The request signal is shown in yellow and the acknowledge signal is shown in red. Only the acknowledge signal is level shifted, as it is driven by the Teensy, whereas the request signal is driven by the classchip. The resistor divider causes an impedance between the level shifted acknowledge signal and the pin driving it, which makes it more susceptible to noise.

A source of noise which is particularly problematic here is the crosstalk between the request and acknowledge signals. In Figure 4.3, as soon as the voltage on the acknowledge signal reaches a level which the classchip reads as high the request signal is pulled down to indicate a new event. The crosstalk causes the acknowledge signal to momentarily drop enough for the classchip to read as low, so the classchip assumes the event is read and pulls the request signal high again. The Teensy never actually read this event, so it is lost.

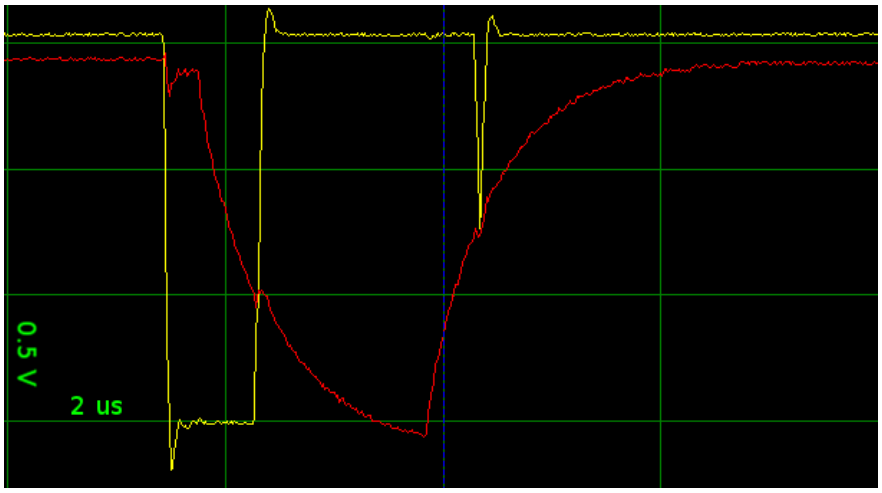


Figure 4.3: C2F handshake signals when using a resistor divider as a level shifter. The yellow signal corresponds to the request from the classchip and the red signal corresponds to the level shifted acknowledge from the Teensy. The crosstalk between the signals causes the acknowledge to be read incorrectly by the classchip which causes events to be lost.

For the PCB, we bought ICs to do the level shifting. We chose the TXB0108, as it is bidirectional. This simplifies the layout, and allows for fewer overall types of ICs. Upon first testing it worked just as expected.

2.3 Expander

The Teensy does not have enough pins to simultaneously interface with everything on the classchip. The Teensy has 34 accessible pins in total excluding the power supply and ground pins, while the classchip has 71.

Our solution for this on the protoboard was to use jumper cables which would then be moved around based on what is being interfaced with. This method is error prone, requires a lot of effort, the jumper cables pick up a lot of noise, and leaving classchip pins floating can affect the behaviour in unexpected ways. A photo of this setup is shown in Figure 4.4.

For the PCB we bought a GPIO Expander. We used the MCP23017T, which works with I2C. The I2C packets are structured as follows: After the first byte is sent to select the slave device, as explained is Section 1, a second byte is sent to select which register on the MCP23017T is to be modified, and finally a third byte with the new contents of that register.

For our purposes, all the pins are used as output pins. So the only registers we are interested in are the ones used the set the mode of the pins to output, and the ones to set the values of the output pins. The modes are set with the IODIRA and IODIRB registers at addresses 0x00 and 0x01 respectively. The values are set with the GPIOA and GPIOB registers at adresses 0x12 and 0x13 respectively.

The fastest way to set the modes and values of the pins is to simultaneously set all the pins on the same register to the desired value. However in the code it is much more readable and modifiable to set them one by one using the *Adafruit_MCP23017.h* library. We went for the latter option.

Setting the pins on the expander is inherently slower than setting the pins on the Teensy itself. We measured it to take $0.71ms$ to toggle a pin on the expander, whereas it only takes about $100ns$ to toggle a pin on the Teensy itself. Therefore we only put the signals that can afford to be slow on the expander. These include the PRST, SRST, IIDLY0, IIDLY1 and all the IID pins as seen in Appendix 1. The IID pins are the data pins for the II AER channel. This channel is also sometimes used to generate pulses, which might have to be done fast. For this reason we put the handshake signals on the Teensy itself so that they can be fast. To continuously generate pulses, the IID signals stay constant while only IIREQ and IIACK change.

Instead of level shifting all the GPIO pins on the expander going to the classchip, we decided to power the expander with $1.8V$ and only level shift the I2C pins communicating with the Teensy. We used the TCA9517 for this, which is a level shifter specifically made for I2C channels. It worked just as expected on the first try.

2.4 Analog Voltage In

The classchip has some analog voltage input pins. These are the AIN0 to AIN15, GO0 to GO5 and GO20 to GO23 pins as seen in Appendix 1. On the protoboard we simply supplied the analog input voltages with potentiometers and jumper cables. This method works, but it is a lot of effort to set many voltages. Since potentiometers are also big, there is a trade off between being able to set many different voltages and having a very large board.

On the PCB we used a DAC instead. We used the BH2226, which is an eight channel, eight bit DAC that is controlled via SPI. Each SPI message consists of a four bit channel setting followed by an eight bit data setting. The channel settings relevant to this project are shown in Table 4.2.

To set up the DAC, first a message with the *Power Down Release* channel setting is sent as instructed by the datasheet. The data setting of this message does not matter. Next all the pins have to be set to digital analog conversion mode. This is done by sending a message with the *I/O D/A Select* channel setting and a data setting of 0xFF. The reason for this is that each of the eight bits of the data setting sets one output and when a bit is high, the corresponding output is in digital analog conversion mode. To set the analog value of any of the pins, use the channel setting corresponding to that pin in Table 4.2 and put the data setting equal to the analog value.

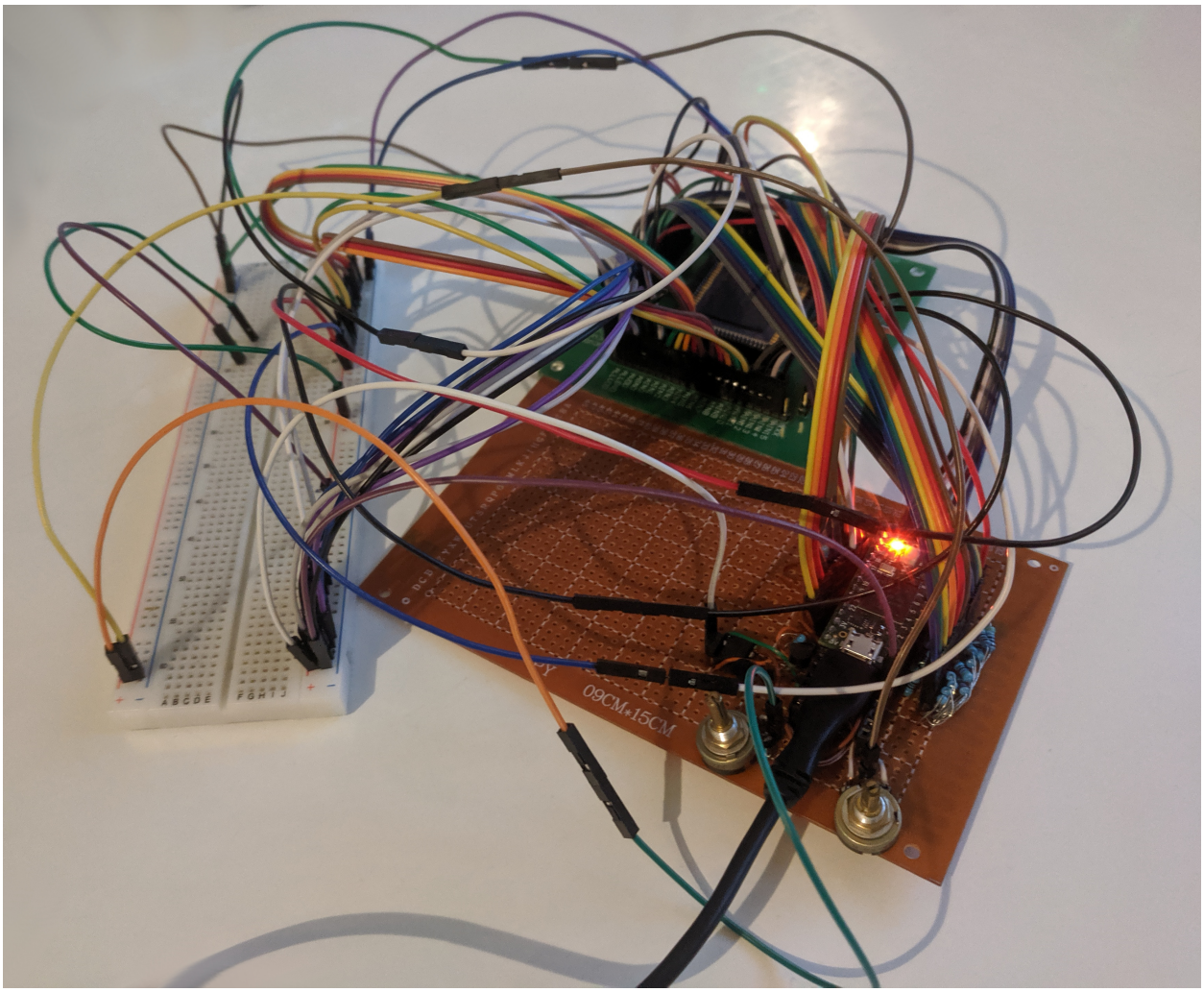


Figure 4.4: Protoboard setup with jumpers. On the left is a breadboard used to connect multiple pins to the same voltage. On the top right is the classchip within a breakout board. On the bottom right is the protoboard containing the Teensy, regulator, potentiometers and resistor divider level shifters. As can be seen, this setup requires many jumper cables. This is problematic, as it is error prone and noisy.

DA1	DA2	DA3	DA4	DA5	DA6	DA7	DA8	I/O D/A Select	I/O status	Power Down Release
0x1	0xE	0x6	0xA	0x2	0xC	0x4	0x8	0x3	0xF	0x9

Table 4.2: Channel settings for the DAC.

A potential problem with this DAC is that it is only eight bits and works with a reference voltage of $3.3V$. Therefore it can only set voltages in steps of about $13mv$. Fortunately, if a source with a higher resolution (or better performance in any other way) is required, it can be connected with jumper cables while the corresponding DAC pin is set to high impedance. To set a pin as high impedance, it first has to be set to I/O mode using a message with the *I/O D/A Select* channel setting and a data setting where the bit corresponding to that pin is low. Then it has to be set as an input using the *I/O status* channel setting and a data setting where the bit corresponding to that pin is also low.

The classchip has 26 analog voltage inputs, and the DAC has eight outputs. Therefore four DACs are required, leaving six extra DAC outputs. These are connected to the pins labeled P0 to P5 on the PCB and they can be connected with jumper cables to anything that is later found to require an analog voltage.

An example where these extra DAC pins might be useful, is when a higher resolution is required on any of the classchip analog inputs. In such a case the DAC output directly connected to that classchip input can be set to high impedance, while two or more of the extra DAC outputs are then connected to that classchip input through resistors. The resulting analog input voltage would then be a weighted average of the connected DAC outputs, where the contribution of each DAC output is inversely proportional to the resistor it is connected with.

2.5 Current Sensors

Most of the currents that students would be interested in measuring on the classchip are converted to spike rates and output via the C2F AER channel. In order to do that, the current is mirrored and that mirrored current is used to generate the spikes. Some of the currents however cannot be measured in this way, because in addition to measuring the current at those nodes, the voltage has to be set. These nodes are connected to pins GO0 to GO5 and GO20 to GO23 as seen in Appendix 1. The current consumption of the neuron circuits also needs to be measured. This is done by measuring the current going into the NCVDD pins as seen in Appendix 1.

On the protoboard, we did not measure any currents. On the PCB, to get the current, we measured the voltage drop across a series resistor with an INA219. The INA219 converts this measured voltage to a twelve bit digital value and sends it to the Teensy using I2C. The INA219 is bidirectional, so it can measure currents that are sinked our sourced to the classchip.

The maximum current we are expecting to measure is $10\mu A$, and the maximum shunt voltage of the INA219 in the mode we are using it is $40mV$. We chose a series resistor of $1k\Omega$. This would lead to a shunt voltage of $10mV$ for a current of $10\mu A$, meaning we are effectively only using ten bits of the twelve bit ADC. We designed for a shunt voltage of $10mV$ instead of $40mV$, because we are not 100% certain yet that the maximum current draw would be $10\mu A$ on every pin.

On startup, each INA219 has to be configured using I2C. The I2C message to write to the INA219 requires three data bytes. The first byte sets the register pointer, while the second and third contain the data that should be written to that register. So to configure the INA219, the first byte contains a pointer to the configuration register(0x00), and the second and third bytes contain the data shown in Table 4.3. The structure of the configuration register is also shown in Table 4.3.

MODE selects whether the device runs continuously or only when triggered, and also whether the shunt voltage, bus voltage or both are being read. We set it to run continuously while only reading the shunt

voltage. SADC and BADC selects the bit depth of the ADC reading the shunt and bus voltage respectively. We set both to be twelve bit. PG programs the gain of the amplifier, which in turn sets the shunt voltage range. We set it to have a shunt voltage range of $40mV$. BNRG sets the bus voltage range, which is irrelevant for our purpose, so we just set it to the minimum of $16V$. RST resets the chip when it is high, so we just set it low.

bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
name	MODE1	MODE2	MODE3	SADC1	SADC2	SADC3	SADC4	BADC1	BADC2	BADC3	BADC4	PG0	PG1	BRNG	-	RST
value	1	0	1	1	1	0	0	1	1	0	0	0	0	0	0	0

Table 4.3: Configuration register of INA219.

The INA219 is capable of converting the shunt voltage reading to a current value with an on-board multiplier and a programmable calibration value. Below are the equations from the datasheet to calculate the calibration value, or *cal*, and if we plug in our values, we get a calibration value of 33554.432. Unfortunately the calibration register is only 15 bits, which is too small to contain that number. Therefore we just simply work with the shunt voltage ADC reading and only convert to ampere when displaying the value to the end user.

$$Current\_LSB = \frac{MaxExpectedCurrent}{2^{15}}$$

$$cal = trunc(\frac{0.04096}{Current\_LSB \times R_{SHUNT}})$$

In order to read from the INA219, one first has to set the register pointer on it to point to the register one wants to read from, which is the shunt voltage register(0x01) in our case. This pointer is always equal to the last register that was written to with an I2C message. So to set it to the shunt voltage register, simply send the I2C message to try write to that register. The shunt voltage register is a read only register. It won't actually be written to, so the contents of the second and third data bytes in the I2C message don't matter.

During normal operation, we will never want to read from any other registers, so we just set the register pointer once at startup, after configuration. Then to read the current, the master does a read request and the slave replies with the contents of the shunt voltage register.

Setting the address of each slave device is done by connecting the address pins to either GND, VDD, SCL or SDA. There are two address pins, and each can be connected to one of four signals, so this allows for 16 unique addresses. We have 15 devices, so one I2C bus is enough. The three most significant bits of the seven bit address are already hard coded as {1, 0, 0}, so the addresses will be between 64 and 79. The Addresses are shown in Table 4.4.

2.6 Analog Voltage Out

The classchip has 16 analog voltage outputs. These are pins AO0 to AO15 as seen in Appendix 1. On both the protoboard and the PCB we used the ADC on the Teensy to read these. On the Protoboard, we used four ADC pins on the Teensy and connected them to different classchip pins with jumper cables.

On the PCB however we only used two ADC pins on the Teensy and looped through all the classchip pins with an analog multiplexer. The Teensy itself has two onboard ADCs(connected to the various pins through its own multiplexers), so we made sure to connect our multiplexers to pins connected to the different internal ADCs. This way they can run in parallel, which requires only half the time that running them sequentially would.

The Teensy ADCs have a resolution of ten bits, but a higher resolution can be achieved by dithering. This is the technique of averaging consecutive samples in the presence of noise. For each doubling of the

Pin Name	I2C Address	USB Address
NCVDD1	68	0
NCVDD2	67	1
NCVDD3	66	2
NCVDD4	65	3
NCVDD5	64	4
GO0	69	5
GO1	70	6
GO2	71	7
GO3	72	8
GO4	73	9
GO5	74	10
GO23	78	11
GO22	77	12
GO21	76	13
GO20	75	14

Table 4.4: Addresses of current sensors.

number of samples used, one bit of resolution is added. We chose to average four samples for every reading, which gave a resolution of twelve bits.

Taking four samples requires $20\mu s$. Since we are sampling 16 channels with two parallel ADCs, reading all channels require $8 \times 20\mu s = 160\mu s$. This results in a sampling rate of $6.25kHz$. Of course this sampling rate can only be achieved if the Teensy is not spending any processing time on other tasks, but that is not a problem, since the sampling rate is already much faster than would be required for our purposes. We do not know exactly what sampling rate is required for the classchip, but it should be below $1kHz$.

Just in case an external measurement device with higher specifications is to be used, we added headers to the analog voltage pins. Since the voltages at these headers are also read by the Teensy ADC, they can be used to measure the voltage at any other node on the PCB. One just has to make sure that the classchip is not driving the node that the header is on, and that the voltage one wants to measure will not damage the classchip.

3 USB Communication

3.1 PC to Teensy

The computer always sends three byte commands which can be sent back to back in the same packets. The content of those commands are shown in Figure 4.5. The red indicates fixed bits that tell the Teensy what type of message it is. The a's and b's indicate the variables that are sent. The x's indicate don't cares. There are six types of messages implemented at the time of writing:

- **Reset:** This causes the classchip, DACs and current sensors to reset. It also toggles the LEDs, which is useful to test whether USB communication is working.
- **Update Sample Rate:** The Teensy periodically sends packets to the computer. The exact structure of these packets will be described in Section 3.2, but for each packet all the analog voltages are sampled and sent. The *update sample rate* message sets the frequency at which these packets are sent. The

second and third bytes in this message together make a 16 bit integer value that represents the sample rate in Hertz. Setting it to zero causes the Teensy to stop sending these packets.

- **Set Voltage:** This is used to set the DAC outputs. The second byte is the address of the pin you want to set as shown on the fourth column of Table 4.5. The third byte is the value you want to set. The LSB of this value corresponds to a voltage of $\frac{3.3V}{2^8} = 12.9mV$.
- **Read Current:** This message requests that a certain current is read, after which the Teensy would respond with the value of that current. This response will be discussed in Section 3.2. The second byte in this message selects which current to read according to Table 4.4.
- **Set DAC High Impedance:** This message is used to set the DAC outputs to High Impedance. The second byte is the DAC number as seen in the first column of Table 4.5. The bits in the third byte should be low for every pin that has to be high impedance, and high for the rest.
- **Configure CoACH:** This message sends the 22 bit AER message of the II channel. The contents of this II message are shown in Appendix 2.

	Byte 0								Byte 1								Byte 2							
bit nr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
reset	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
update sample rate	0	0	0	0	0	0	0	1	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
set voltage	0	0	0	0	0	0	1	0	a	a	a	a	a	a	a	a	b	b	b	b	b	b	b	b
read current	0	0	0	0	0	1	0	0	a	a	a	a	a	a	a	a	x	x	x	x	x	x	x	x
set DAC high impedance	0	0	0	0	0	1	0	1	a	a	a	a	a	a	a	a	b	b	b	b	b	b	b	b
configure CoACH	1	1	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

Figure 4.5: Structure of messages sent from the computer to the Teensy.

3.2 Teensy to PC

The Teensy has two types of packets which it can send to the computer and they are distinguished by their length. The one type is always exactly two bytes, and the other type is variable, but will always be at least 66 bytes.

Packet Type 1

The first type of packet is the reply to the computer after it requested a current read. The least significant four bits of the first byte together with all eight bits of the second byte make up the twelve bit value of the current reading. The LSB of this value corresponds to a current of $\frac{40\mu A}{2^{12}} = 9.8nA$. The most significant four bits of the first byte have to all be low for this message to be registered as a current reading. These bits are reserved for when we want to send more types of messages in the future, for example to indicate when buffers are full or other errors.

Packet Type 2

The next type of packet is at least 66 bytes and it is sent periodically. Each two consecutive bytes make up a 16 bit value. There are four types of values:

- Bytes 0 to 31 make up the 16 voltage reading values. Note that even though these are only twelve bit values, we dedicate two full bytes to each to simplify encoding and decoding. The LSB in this value corresponds to a voltage of $\frac{3.3V}{2^{12}} = 0.81mV$.

DAC number	Pin on DAC	Pin Name	USB Address
0	DA1	AIN0	0
	DA2	AIN1	1
	DA3	AIN2	2
	DA4	AIN3	3
	DA5	AIN4	4
	DA6	AIN	5
	DA7	AIN6	6
	DA8	AIN7	7
1	DA1	AIN8	8
	DA2	AIN9	9
	DA3	AIN10	10
	DA4	AIN11	11
	DA5	AIN12	12
	DA6	AIN13	13
	DA7	AIN14	14
	DA8	AIN15	15
2	DA1	GO0	16
	DA2	GO1	17
	DA3	GO2	18
	DA4	GO3	19
	DA5	GO4	20
	DA6	GO5	21
	DA7	GO23	22
	DA8	GO22	23
3	DA1	GO21	24
	DA2	GO20	25
	DA3	P0	26
	DA4	P1	27
	DA5	P2	28
	DA6	P3	29
	DA7	P4	30
	DA8	P5	31

Table 4.5: Addresses of DAC outputs.

- Bytes 32 to 63 are the 16 rates for the C2F AER channel. These rates are simply the number of events since the previous packet has been sent, so to convert these rates to EPS, they still have to be multiplied by the rate at which the packets are being sent. This rate is the same as the configurable sample rate as described in Section 3.1.
- Bytes 64 and 65 make up a 16 bit value indicating the number of events from the AERO channel. Even though one byte should be enough for this, we use two bytes in order to preserve the alignment for the bytes to follow.
- The rest of the bytes contain the events from the AERO channel. Each event comes in a pair of two bytes, the first containing the timestamp and the second the address. The timestamp is in steps of $1024\mu s$, for the reasons described in Section 2 of Chapter 3. Once again, even though these addresses only require three bits, we use a full byte for each to simplify encoding and decoding. However if we later find that we need the timestamps to run longer before overflowing, we will use the unused bits of the address byte to add more bits to the timestamp.

4 Graphical User Interface

We created a GUI that runs on Linux computers that can be used to interface with the PCB. It is written in C++ and uses *cvui*, which is a user interface library that is built on top of OpenCV. The GUI has two main functions. Firstly it is an easy to use front end for sending the USB messages described in Section 3.1. Secondly it visualises the data read from the classchip described in Section 3.2.

4.1 Header

On start up, the only thing shown in the GUI is the header shown in Figure 4.6. The checkboxes are used to enable or disable the other GUI elements. The *PULSE* button sends the *configure CoACH* USB message used to send a pulse to the neuron circuits. The *RESET* button sends the *reset* USB message.

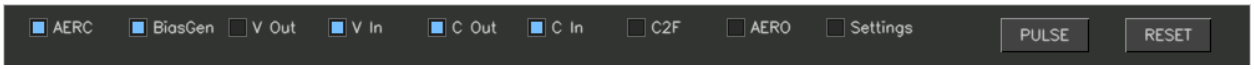


Figure 4.6: Header that is displayed at the top of the GUI.

4.2 AERC

The *AERC* GUI element shown in Figure 4.7 is used to configure the classchip. The counters set the select lines of the multiplexers and demultiplexers on the classchip. The checkboxes enable or disable certain circuits or behaviours of certain circuits. The *SEND AERC* button sends the *configure CoACH* USB message to configure the classchip with the current settings.

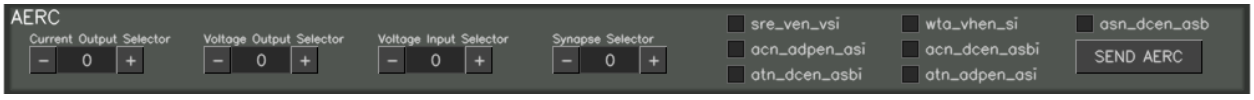


Figure 4.7: GUI element to configure the AERC message.

4.3 BiasGen

The *BiasGen* GUI element shown in Figure 4.8 is used to set the biases on the classchip. The coloured checkboxes are used to select which bias one wants to set. The checkboxes at the bottom are used to set the master current, while the slider sets the fine current. The *SET BIAS* button sends the *configure CoACH* USB message that sets the selected bias to the selected current.

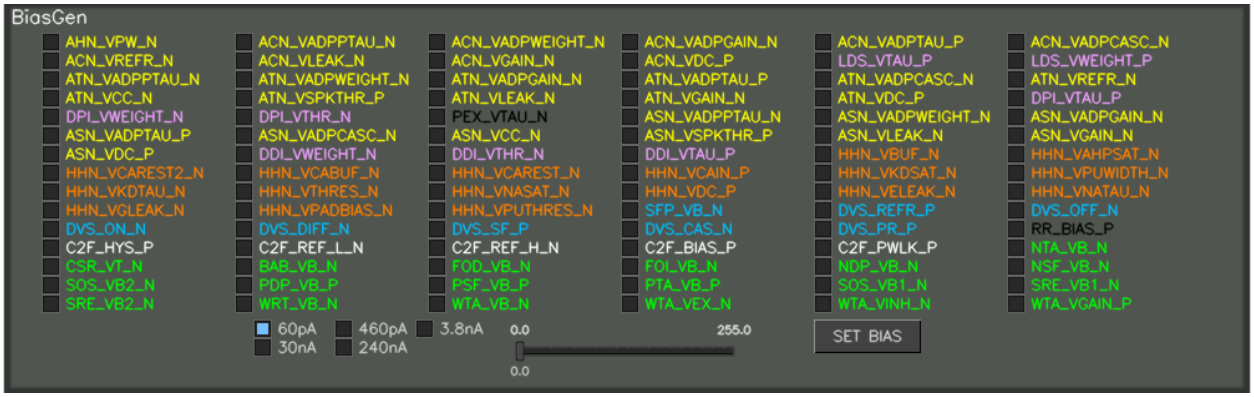


Figure 4.8: GUI element to configure the AER message used for generating biases.

4.4 V In

The *V In* GUI element shown in Figure 4.9 is used to set the DAC output voltages. The checkboxes select which output to set, and the slider sets the value. The *SET* button sends the *set voltage* USB message to set the selected output to the selected value. The *HighZ* button sends the *set dac high impedance* USB message that sets the selected output as high impedance. At the moment only one output can be set as high impedance at a time using the GUI.

Figure 4.9: GUI element for setting DAC outputs.

4.5 C Out

The *C Out* GUI element shown in Figure 4.10 is used to request a current reading. The checkboxes select which current one wants to read, and the *READ* button sends the *read current* USB message. When the Teensy replies with the current reading, the value is displayed in the *Last reading* text area.

Figure 4.10: GUI element to request a certain current reading and then display the result.

4.6 Oscilloscope

The *V Out* GUI element shown in Figure 4.11 displays the analog voltage readings in the form of an oscilloscope. There is also a *C2F* GUI element that shows the event rates of the C2F AER channel in the form of an oscilloscope, but it is omitted from this report as both oscilloscope GUI elements are functionally equivalent. It is a very basic oscilloscope that does not yet have the capability to freeze frame or use cursors to measure. Also, at the moment the first two samples on the oscilloscope have to be discarded, as they are not readings. They are used to set the minimum and maximum value of the oscilloscope. The checkboxes on the right are used to select which channels one wants to see. Next to them the values of the most recent samples are shown.

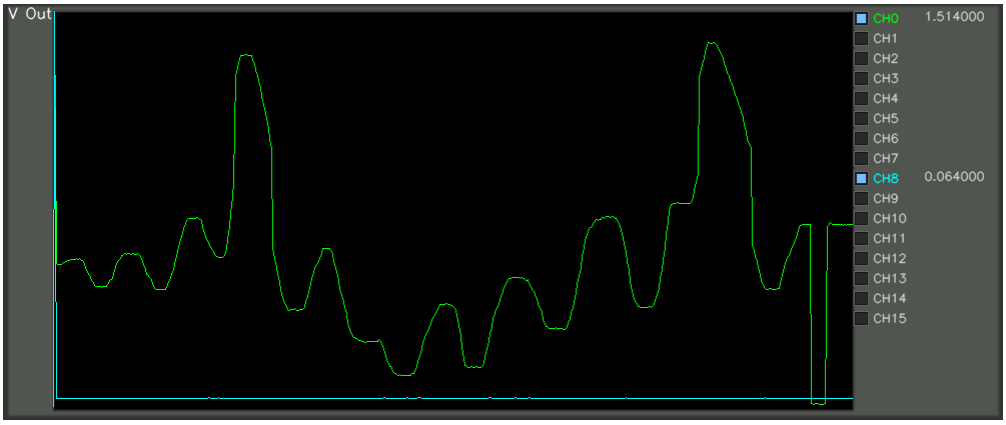


Figure 4.11: GUI element that displays analog readings in the form of an oscilloscope. In this example channel zero is reading an analog voltage of a potentiometer that is manually turned. Channel eight is a floating input to the ADC.

4.7 Raster Plot

The *AERO* GUI element shown in Figure 4.12 displays the AERO events in the form of a raster plot. There are eight channels stacked in the y axis, and the x axis is time. The x axis continuously rolls to the left as new events appear on the right. At the moment, the raster plot only works when the sample rate is set to $4Hz$. This sample rate refers to the rate at which USB packets containing the events are sent from the Teensy. In Figure 4.12, the AER events are randomly generated by the Teensy, as we have not yet configured the classchip on the PCB. It was however confirmed on the protoboard that the Teensy is able to read AERO events.

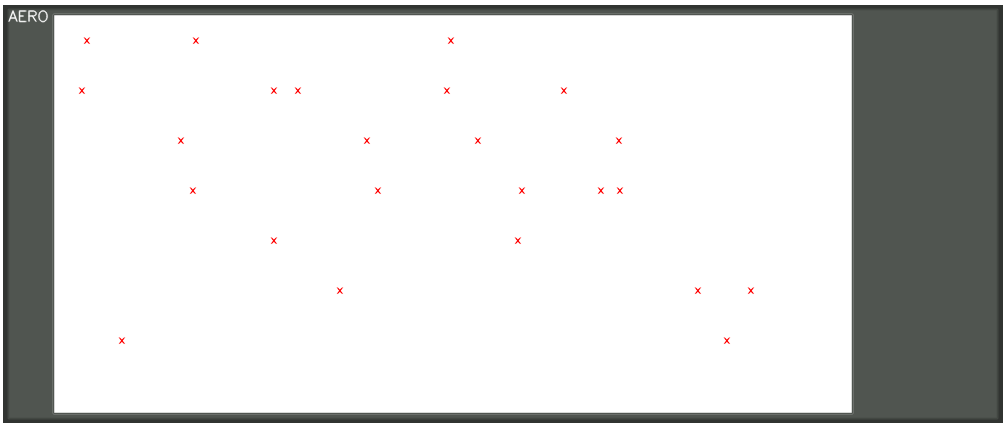


Figure 4.12: GUI element that displays AERO events in the form a raster plot.

4.8 Settings

The *Settings* GUI element shown in Figure 4.13 configures the GUI itself, as well as set the sample rate. The *X range* slider sets how many samples are shown on the oscilloscopes. The *VO max* and *C2F max* sliders set the maximum values on their respective oscilloscopes. The *Sample Rate* is used to set the rate at which the Teensy samples the analog voltages and sends the USB packets. The *SET* button applies the selected settings to the GUI and sends the *update sample rate* USB message.

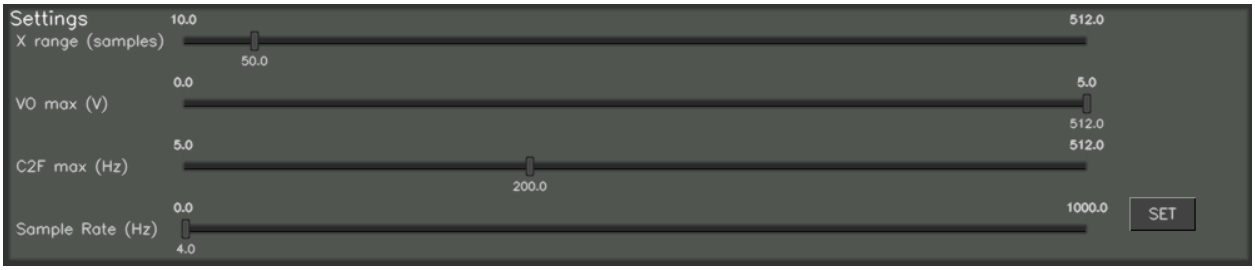


Figure 4.13: GUI element that configures the GUI itself, as well as set the sample rate.

4.9 Multithreading

The reading of USB messages take up a lot of processing time just waiting for the messages to arrive. This may introduce lags in the GUI which have a very negative impact on user experience. Therefore we use multithreading. Multithreading is the ability of a processor to run different threads in parallel. We use two threads.

The first takes care of reading the USB packets and copying their contents to relevant places. It also does a bit of processing on the data, such as clipping the voltage readings that are higher than the max setting of the oscilloscope for example.

The second thread continuously updates the GUI, reads the user input and displays the data read in the first thread. If the user input requires the sending of USB packets, that is also taken care of by this thread.

This does not result in the same lag that receiving USB packets do, as the main cause of this lag is waiting for a packet to arrive at an unknown time.

After receiving the USB packets, the first thread needs to write the data some place where the second thread can read it. Since both threads are accessing the same data, this might lead to errors. If the second thread is reading it while the first thread is writing to it, the second thread would read inconsistent data.

The solution is to use locking. This is a method to prevent certain parts of different threads from running at the same time. For each of these parts it locks before running and then releases the lock afterwards. If thread A tries to lock while thread B is already locked, thread A just waits until the lock is released. So in our case, we just lock each thread sometime before it accesses the shared data, and release the lock again afterwards.

5 Layout

The PCB was designed using Eagle CAD and the layout is shown in Figure 4.14. The red traces are on the top and the blue traces are on the bottom of the board. Since the classchip is called CoACH, we named the PCB PLANE, which stands for *Platform for Learning About Neuromorphic Engineering*.

The Teensy is located on the North Eastern side of the PCB in such a way that it is easy to plug in the USB cable. The pinout we allocated to the Teensy and the expander are shown in Appendix 4. Some pins were fixed due to the capabilities of the Teensy itself, such as the I2C and SPI channels. The rest we mostly allocated in a way to make layout simpler. The extra pins were used for LEDs.

5.1 Ground Layers

Since this is a very dense board and contains analog signals, we added two dedicated GND layers. We added two because the manufacturing process of PCBs add layers in multiples of two. Multiple GND layers also increase the same benefits that having only one would provide. This makes the PCB four layers in total.

The first benefit of GND layers is that they ensure that all the ICs have the same reference GND by minimizing the impedance between their GND pins as much as possible. This reduces noise caused by fluctuations in return currents.

Another benefit of GND layers is that they reduce crosstalk between traces. Crosstalk occurs due to the electromagnetic coupling between traces that are physically close to each other. A ground layer between two traces that are on opposite sides of the board reduces crosstalk by physically adding a conductor in between, stopping the electromagnetic fields from passing through.

A GND layer also reduces crosstalk between parallel traces on the same side of the board. This is because it ensures that the optimal path for return current is always available. This path would more or less be underneath the trace itself, as this encloses the least area, reducing inductance. The traces together with their return paths create transmission lines, reducing the electromagnetic radiation from the traces and therefore crosstalk.

5.2 Passive components

All the capacitors on the PCB are used for decoupling. They filter out noise on the power supply lines. On the North Western corner of the PCB there are many parallel capacitors of different values and sizes. This is because different capacitors have different characteristics at different frequencies, so to get good filtration on a wide range of frequencies, a variety of capacitors are required. These are connected to the inputs of the regulators.

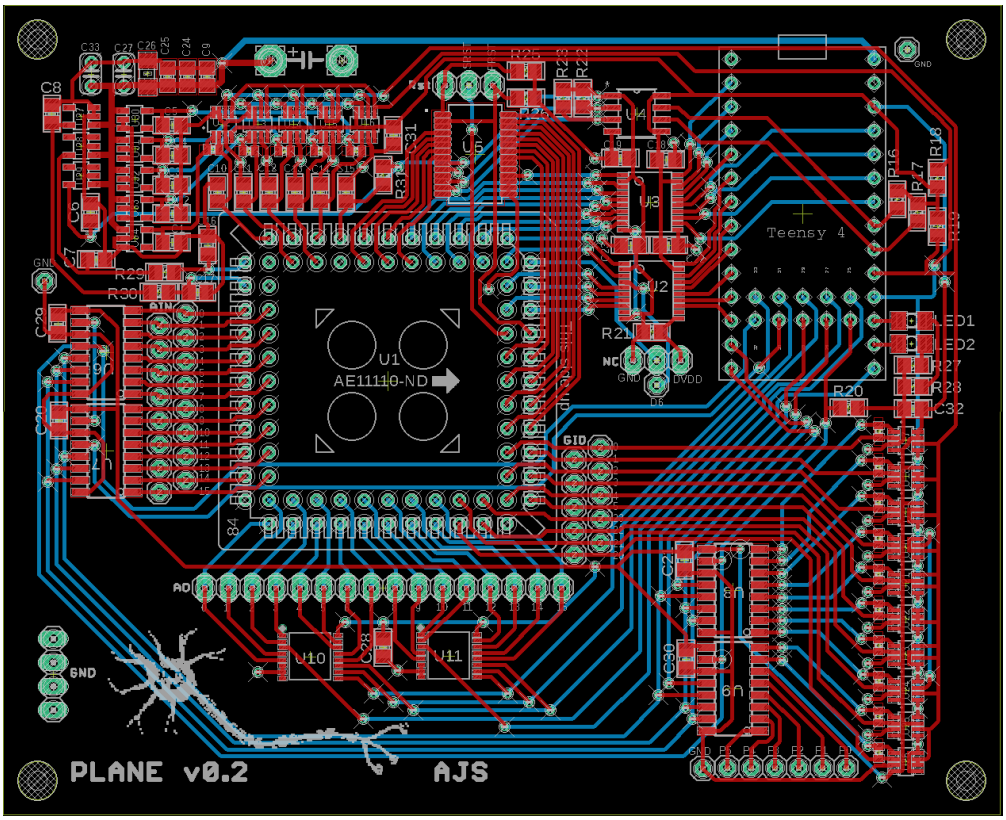


Figure 4.14: PCB board layout made with Eagle CAD.

A significant source of high frequency noise on the power supply lines is ICs that vary the amount of current they pull. For this reason, capacitors are situated very close to the power supply pins of most ICs. These capacitors have small values, generally only $0.1\mu F$, so that they have good high frequency performance.

Most of the resistors are used as pull up or pull down resistors. There are also some series 0Ω resistors on the DVDD, AVDD and ALLVDD lines. If something is not working, these resistors can be removed so that a series current sensor can be used for debugging.

Chapter 5

Conclusion

We showed that microcontrollers are a viable alternative to FPGAs for interfacing with neuromorphic devices. While interfacing with the DVS, we used some example programs to show that some major benefits of using a microcontroller include the fast development time and on board processing capabilities. We also designed a PCB that is to be used in the Neuromorphic Engineering class offered at ETH.

There are however some improvements that could be made on our work. Events could be sampled significantly faster if we find or design a microcontroller in which the pins are mapped adjacent to each other on the internal register. Our algorithm for tracking the location and frequency of a blinking LED also has to be improved for a greater accuracy. The PCB would benefit from having DACs with a higher resolution than eight bits. The oscilloscope and raster plot in the GUI are still very basic. They would be much more useful if they had the ability to freeze frame and take measurements with cursors.

A very useful application area for our work would be robotics. Many autonomous robots use microcontrollers to do all their processing, and the low power consumption and sparse data representation make neuromorphic sensors very attractive to these robots. A drone, for example, fitted with just one programmable microcontroller and a DVS would be a very versatile system capable of some intelligent behaviour.

The system that tracks LED blinking frequency could also have some interesting applications. A blinking LED could send a message to a DVS in the form of a binary message using two frequencies and the microcontroller could decode that message. Or the LED could be used to transmit a frequency modulated analog signal, such as audio for example.

Next we will work on implementing an API running on the computer's side for interfacing with the classchip. We will also add more functionality, such as the ability to sweep certain voltages and currents while reading different voltages and currents.

Appendices

1 CoACH Pad Frame

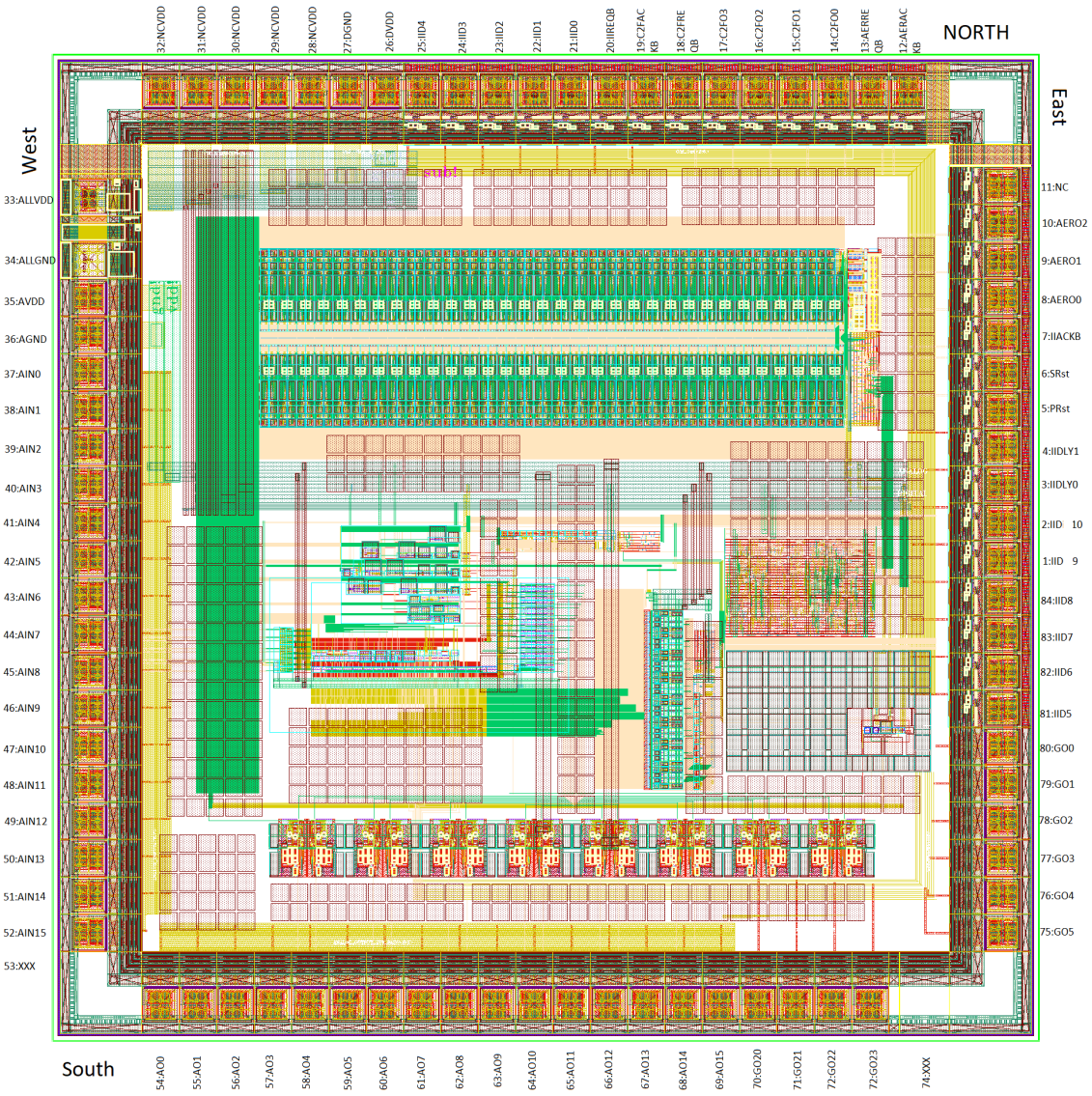


Figure 1: CoACH pad frame

2 Input Interface Message Structure

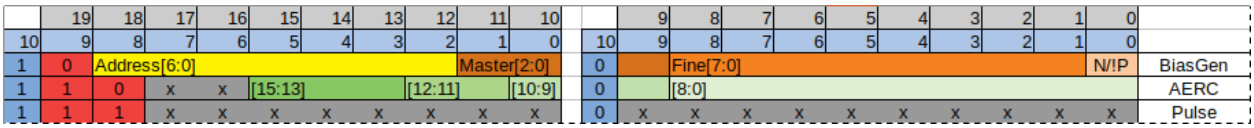


Figure 2: Input Interface message structure. Two ten bit messages are sent consecutively to make up the full twenty bit message.

3 PCB Schematic

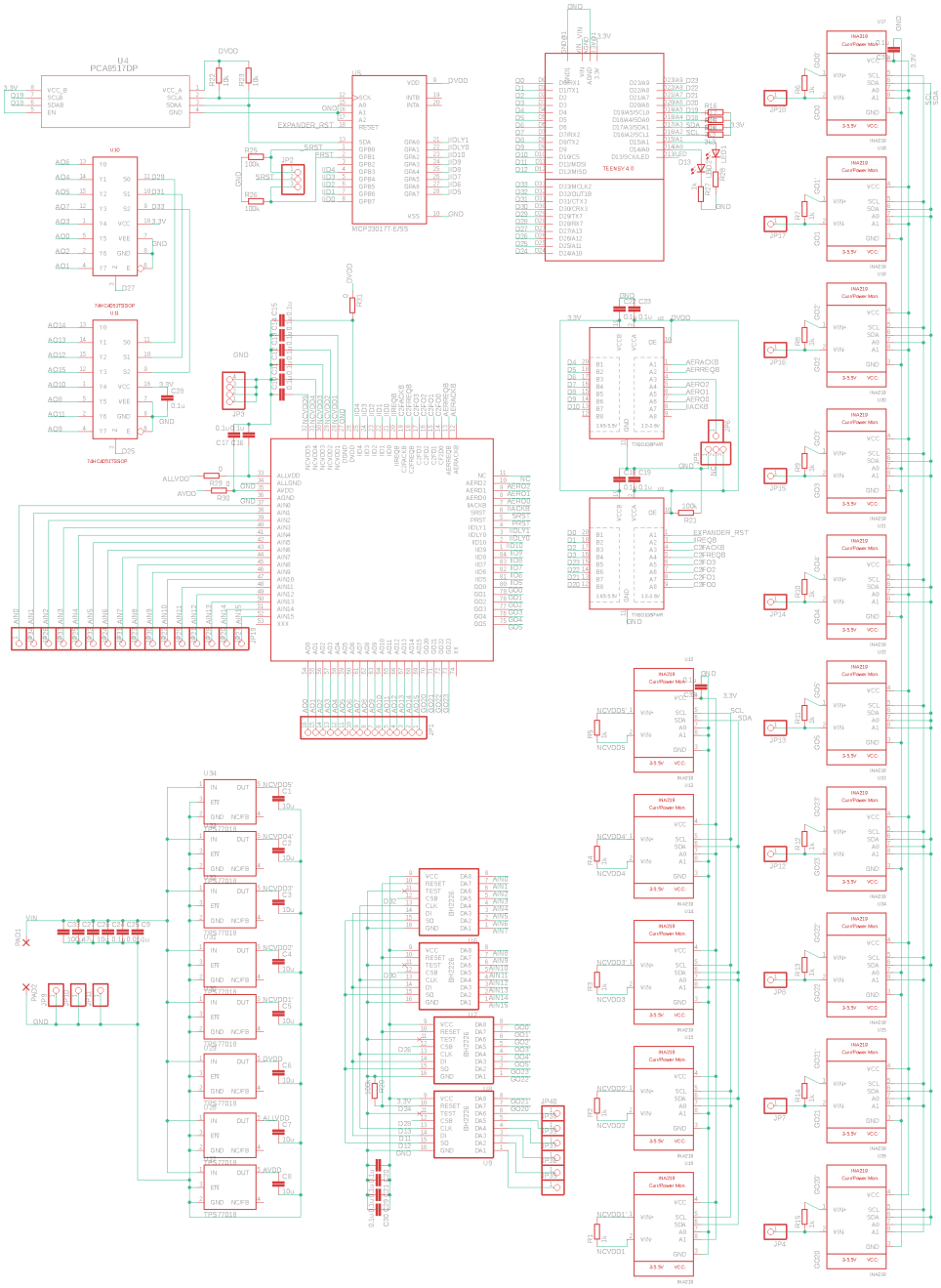


Figure 3: PCB Schematic made with Eagle CAD.

4 Teensy 4 Pin Allocation in PLANE

	Teensy pins										Expander pins		
	GND						5V			SRST	B0	A0	IID5
expander reset	0						GND			PRST	B1	A1	IID6
IIREQB	1						3.3V				B2	A2	IID7
C2FACKB	2						23	C2FO3		IID4	B3	A3	IID8
C2FREQB	3						22	C2FO2		IID3	B4	A4	IID9
AERACKB	4						21	C2FO1		IID2	B5	A5	IID10
AERREQB	5						20	C2FO0		IID1	B6	A6	IIDLY1
NC	6						19	I2C expander		IID0	B7	A7	IIDLY0
AERO2	7						18	I2C expander			INTB	INTA	
AERO1	8						17	I2C current					
AERO0	9						16	I2C current					
IIACKB	10	mux s2	mux s1	mux s0	adc	adc	15	LED		SCK			A2
SPI MOSI	11	33	31	29	27	25	14	LED		SDA			A1
SPI MISO	12	32	30	28	26	24	13	SPI SCK					A0
		SPI CS	SPI CS	SPI CS	SPI CS	SPI reset							

Figure 4: Teensy and expander pinout. The grey signals are level shifted to $1.8V$.

Bibliography

[1] A. Censi et al. "Low-latency localization by Active LED Markers tracking using a Dynamic Vision Sensor". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems* (2013).

[2] P. Lichtsteiner, C. Posch, and T. Delbruck. "A 128× 128 120 dB 15 μs Latency Asynchronous Temporal Contrast Vision Sensor". In: *IEEE Journal of Solid-State Circuits* 43.2 (2008), pp. 566–576.

[3] C. Mead. "Neuromorphic electronic systems". In: *Proceedings of the IEEE* 78.10 (1990), pp. 1629–1636.

[4] Ning Qiao et al. "A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses". In: *Frontiers in Neuroscience* 9 (2015), p. 141. ISSN: 1662-453X. DOI: 10.3389/fnins.2015.00141. URL: https://www.frontiersin.org/article/10.3389/fnins.2015.00141.

[5] Oliver Rhodes et al. "Real-time cortical simulation on neuromorphic hardware". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.2164 (Dec. 2019), p. 20190160. DOI: 10.1098/rsta.2019.0160. URL: https://doi.org/10.1098/rsta.2019.0160.

[6] Author Paul Stoffregen. *Teensy 4.0*. Aug. 2019. URL: https://www.pjrc.com/teensy-4-0/.

[7] Timo Wunderlich et al. "Demonstrating Advantages of Neuromorphic Computation: A Pilot Study". In: *Frontiers in Neuroscience* 13 (2019), p. 260. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00260. URL: https://www.frontiersin.org/article/10.3389/fnins.2019.00260.