ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Hardware Implementation of Optical Flow Algorithm

Semester Project

Submitted by:
Andreas Steiner

Supervisors:
Dr. Shih-Chii Liu
Prof. Tobi Delbrück

June 15, 2011

Contents

1 Introduction

1.1 Optical Flow, Flying Robots

Different groups are currently working on micro aerial vehicles that are capable of autonomous navigation (e.g. [19]). In contrast to flying robots that are controlled by a computer that is situated on the ground and controls these vehicles via an airborne connection, these autonomous systems have extreme weight and performance restrictions that have to be matched by a novel approach.

Insects, such as the fruitfly, on the other hand, show an extraordinary ability for flight control [4, 16], even in difficult environments, such as narrow canyons with strong turbulences. The visual input is believed to be the main sensory source for adapting the flight maneuvers to the changing environment.

The neurocomputational basis of visual sensory input processing of the fly has been studied in detail and different bio-inspired designs for flight control and autonomous navigation using visual sensors have emerged. Particularly, the computation of of the *optical flow* (i.e. the pattern of apparent motion of the optical scene caused by the relative motion between the observer and the surroundings) can be used to detect different important events, such as the deviation from the desired flight path, the approach of an obstacle or the presence nearby stationary or moving objects.

Because the implementation of the signal processing, such as it has been studied in the nervous system of insects, is computationally demanding, the discovered principles need to be adapted to the technical possibilities, especially with the weight and performance constraints that have to be met for a successful integration in a flying platform. There exists a wide variety of custom analog very large scale integration (aVLSI) implementations that perform optical flow computation in hardware [2]. At the Institute of Neuroinformatics (INI), several motion detection chips have been developed that are capable of performing various degrees of processing of visual data, ranging from simple filtering and amplification to calculation of the actual motion vector, completely parallelized and continuous in time.

In a previous semester project [17], the output of a two-dimensional variant of such a motion detection chip (the `MDC2D`, see section 1.2) was used to evaluate different algorithms that are commonly used for the determination of the optical flow. Among others, an algorithm based on linear interpolation of consecutive frames [15] was examined. This algorithm was chosen for this project because it can be performed with fewer calculations than traditional optical flow algorithms (such as gradient based optical flow determination) and is therefore suitable for implementation on a microcontroller.

The objective of this semester project is the efficient implementation of this system (estimation of global optical flow by interpolation performed on the output of the MDC2D) on a yet to be determined microcontroller architecture. For achieving this, a printed circuit board and computer-based test framework have first to be developed.

1.2 Motion Detection Chip 2D

In the past decades, a wide variety of *Motion Detection Chips* has emerged. These chips contain light sensitive elements such as photodiodes to extract information from the visual scene and process this data to a varying degree. They extract information from the visual scene either through a simple *Active Pixel Sensor* (APS, such as it can be found in commercial cameras) or through a more elaborate *pixel* that contains circuitry to process the raw light intensity signal, like adaptation elements or filter banks. Most of these motion detection chips perform some kind of analog computation that extracts information about the *optical flow* and this signal can then be read out from the chip. Different algorithms for estimating this optical flow are currently in use; for a more detailed description, see [2] *chapter 8, p101ff.*

The chip used in this project is called *Motion Detection Chip 2D* (`MDC2D`) and was designed by Shih-Chii Liu at INI. For converting infalling light into an electrical signal, a simplified version of the circuit decribed in [6] is used. The output of this first stage of processing is a voltage that relates logarithmically with the infalling light intensity and can be read out via an analog pad. Encoding the signal logarithmically has the advantage that the it becomes invariant to absolute light intensity (because the *reflectance*, i.e. the percentage of light reflected, is an object property that remains constant under all illumination conditions). A second stage amplifies this signal while also acting as a high-pass filter [9]. This circuit is inspired by the laminar monopolar cells in the fly visual system and aims at producing an output with increased contrast for further processing. See figure 1 for the circuit diagrams.

Contrary to most other motion detection chips, the `MDC2D` does not perform any analog computations on-chip and can therefore not directly output the values of the optical flow. In the design described in this report, the analog signal from the LMC circuit is sequentially read out and digitized, before the algorithm described in section 1.3 is applied for computing the actual optical flow. This sequential read-out has the disadvantage that it is prone to temporal aliasing (unlike chips that perform continuous analog computation of motion values), although this can be avoided with a sufficiently high readout rate. On the other side, it has the advantage that it reduces mismatch inherent to analog computation and allows the sharing of centralized resources (e.g. a microchip that performs optical flow computation among other tasks).

Additionally to the 20×20 pixels containing each the above-mentioned circuitery, the `MDC2D` also includes a scanner that multiplexes the output

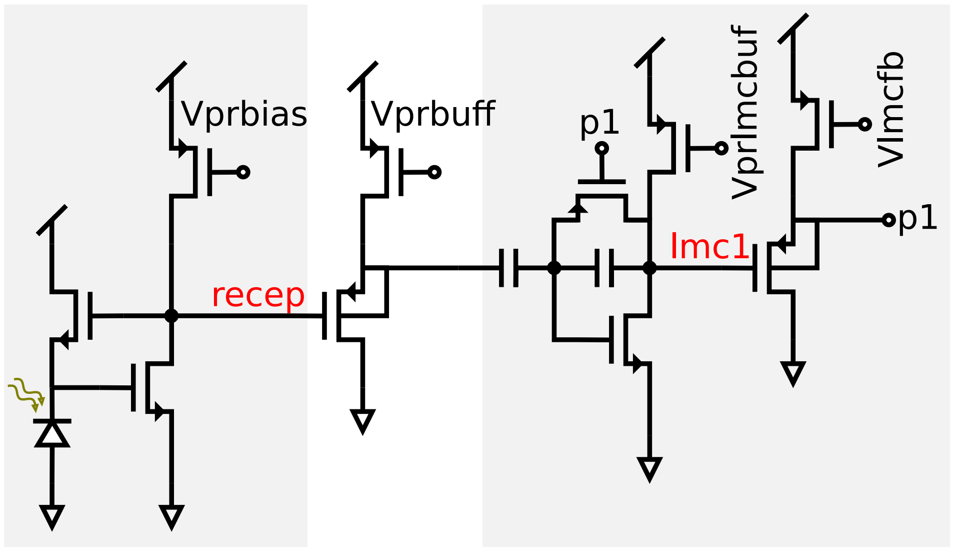

Figure 1: The in-pixel circuit of the `MDC2D`: the two analog signals that can be read-out via the pads are labeled in red (the output of the photoreceptor circuit `vrecep` and the output of the LMC stage `lmc1`). Note that the nodes labeled with `p1` are connected together.

of the 400 pixels to a shared analog pad, a bias generator that reads in a digital configuration and sets the operating point of the analog circuits and an analog-to-digital converter (which was not used in this project). Please refer to [17] for a more detailed description of the chip (such as pin-out, biases and their optimal value and timing-diagrams for the scanner/ADC).

1.3 Image Interpolation for Global Flow Determination

There are several ways to computationally extract the global optical flow from an image stream. One approach is to extract features from the visual scene and track their spatial evolution (*token based* optical flow calculation). Another approach is to use gradient-based schemes that compute the local velocity based on the gradient in intensity (with some additional smoothness constraint) and combine these local values to compute the global flow. Srinivasan described yet another possibility to compute the global motion [15, 14] by using a simple single-staged procedure of image interpolation. In this project, the simplified version of this algorithm is used, which only computes the translation along two dimensions, ignoring the rotational component about the axis perpendicular to these two dimensions.

In the following paragraphs, a rigid textured plane is considered that translates “en bloc” in the fronto-parallel plane, without any rotational component. The basic assumption of this algorithm is that the image, as seen by the stationary camera, deforms linearly between two snapshots of the trajectory. Let $f(x, y)$ be the intensity function of an image captured by this camera, $f_0(x, y)$ denoting a preceding snapshot taken Δt earlier. Further, the true (and unknown) displacement between the two snapshots will be called $\Delta x, \Delta y$.

A simple one-dimensional example will illustrate the assumption of linear deformation between two snapshots : Figure 2 shows a arbitrary one-dimensional intensity function $f_0(x)$ as well as two versions shifted by Δx_r (the *reference amount*). The linear deformation can be expressed as

$$\hat{f}(x) = f_0(x) + \frac{1}{2}\frac{\widehat{\Delta x}}{\Delta x_r}\left(\underbrace{f_0(x + \Delta x_r)}_{=f_1(x)} - \underbrace{f_0(x - \Delta x_r)}_{=f_2(x)}\right) \tag{1}$$

where $\hat{f}(x)$ is the interpolated image based on the original image and the two shifted images. Figure 3 shows the *difference* $f(x) - f_0(x)$ between the original intensity function and the intensity function moved along the axis by Δx as well as the difference $\hat{f}(x) - f_0(x)$ between the original intensity function and the interpolated estimate.

The error between the moved intensity function its interpolated counterpart is defined as

$$E = \int \left(f(x) - \hat{f}(x)\right)^2 \; dx \tag{2}$$

and depends on the presumed shifted amount $\widehat{\Delta x}$. Minimizing this error by setting $\frac{\partial E}{\partial \widehat{\Delta x}} \overset{!}{=} 0$ and solving for $\widehat{\Delta x}$ yields

$$\widehat{\Delta x} = 2\Delta x_r \frac{\int \left(f(x) - f_0(x)\right)\left(f_1(x) - f_2(x)\right) \; dx}{\int \left(f_1(x) - f_2(x)\right)^2 \; dx} \tag{3}$$

As can be seen in figure 3, the algorithm yields good estimates for small Δx but under-estimates the moved amount when $\Delta x > \Delta x_r$. This observations was also made in two dimensions [15].

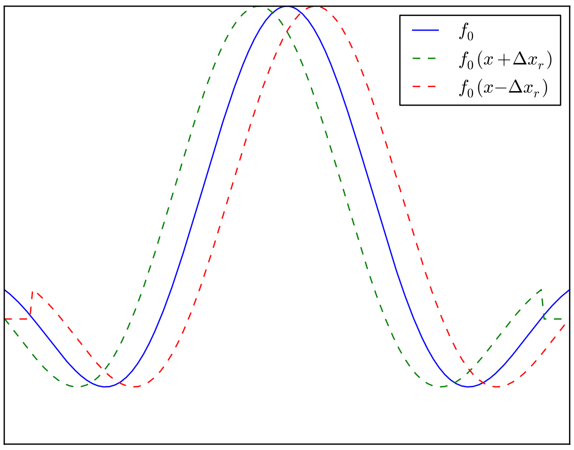

Figure 2: An arbitrary one-dimensional intensity function and two shifted versions; note that the shift is not a shift in the parameter of the function, but simply a shift of the values, filled up with zeroes at the borders.

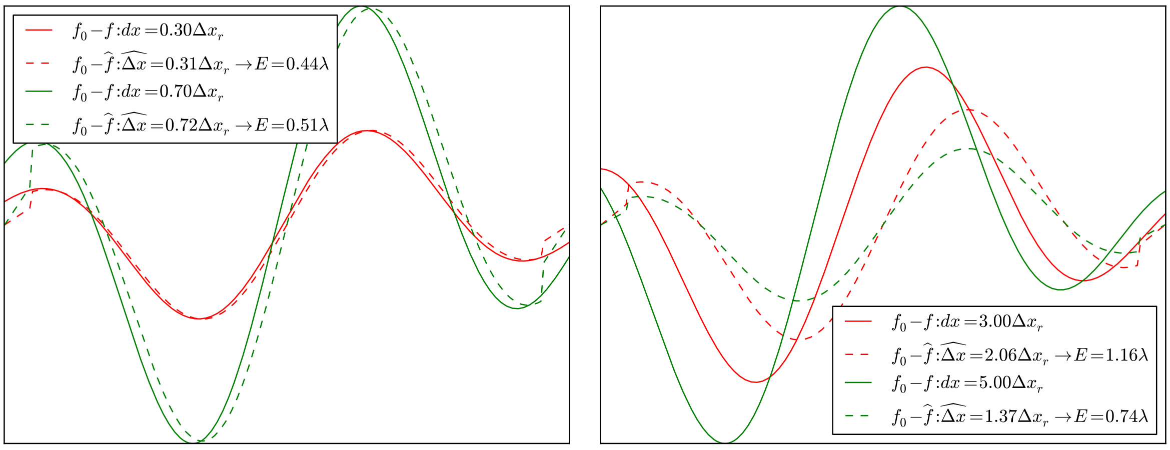

Figure 3: Difference of the *moved* intensity function and the original intensity function (solid line) as well as between the *interpolated* intensity and the original intensity functions (dashed line) – the amount of the interpolation was calculated using formula (3); the error is in arbitrary units.

Extending the algorithm into the second dimension is simple : equation (1) needs to be rewritten for an interpolation with two variables Δx and Δy. The error (2) is integrated over two dimensions. Then, again by setting $\frac{\partial E}{\partial \widehat{\Delta x}} \overset{!}{=} 0$, $\frac{\partial E}{\partial \widehat{\Delta y}} \overset{!}{=} 0$ and by using the following abbreviations

$$
\begin{aligned}
f_1(x,y) &= f_0(x+\Delta x_r, y) && (4)\\
f_2(x,y) &= f_0(x-\Delta x_r, y) && (5)\\
f_3(x,y) &= f_0(x, y+\Delta y_r) && (6)\\
f_4(x,y) &= f_0(x, y-\Delta y_r) && (7)\\
a &= \iint \left(f_2(x,y) - f_1(x,y)\right)^2 \, dx \cdot dy && (8)\\
b &= \iint \left(f_2(x,y) - f_1(x,y)\right)\left(f_4(x,y) - f_3(x,y)\right) \, dx \cdot dy && (9)\\
c &= \iint \left(f_4(x,y) - f_3(x,y)\right)^2 \, dx \cdot dy && (10)\\
d &= \iint \left(f_2(x,y) - f_1(x,y)\right)\left(f(x,y) - f_0(x,y)\right) \, dx \cdot dy && (11)\\
e &= \iint \left(f_4(x,y) - f_3(x,y)\right)\left(f(x,y) - f_0(x,y)\right) \, dx \cdot dy && (12)\\
& && (13)
\end{aligned}
$$

we get a simple linear system of equations with two unknowns.

$$
\begin{pmatrix} a & b \\ b & c \end{pmatrix} \cdot \begin{pmatrix} \widehat{\Delta x}/\Delta x_r \\ \widehat{\Delta y}/\Delta y_r \end{pmatrix} = \begin{pmatrix} 2d \\ 2e \end{pmatrix} \tag{14}
$$

which can easily be solved explicitly, e.g.

$$
\widehat{\Delta y}/\Delta y_r = \frac{2e - \frac{2db}{a}}{c - \frac{b^2}{a}} \quad , \quad \widehat{\Delta x}/\Delta x_r = \frac{2d - by}{a} \tag{15}
$$

As mentioned earlier, the advantage of this algorithm is its relative low computational cost: the reference images f_1, f_2, f_3, f_4 can be calculated by simply subtracting shifted versions of the same image; the calculation of a, b, c, d, e reduces to a looped multiply-and-accumulate and the global motion estimates in (15) can then be calculated with some few algebraic operations.

2 Development

2.1 Choice of an Adequate Microcontroller

The algorithm described in section 1.3, together with the target application of the motion calculation (i.e. micro aerial vehicles with relatievly high optical flow) defined the performance criteria that lead to the choice of an appropriate microcontroller.

In particular, performing a linear interpolation over a 20×20 pixel array results in a total of 2006 multiplications, 2404 additions and 8 divisions (as calculated in [17]) – the number of algebraic operations was further reduced in the adaption of the algorithm to the microcontroller (see section 2.5). In order to perform the whole calculation in less than a millisecond, the microprocessor should support single-cycle multiplications and run at a speed of 10 MHz or higher. The speed of the division instruction is less important because it is used only a few times for the whole calculation.

This considerations led to the choice of the `dsPIC33FJ128MC804` [13] (`dsPIC33F`). This 16 bit processor is designed for the use in digital signal processing, runs at 40 MIPS and has single-cycled multiply-accumulate instructions (see section 2.5). The divide instruction is also implemented in hardware, but takes 19 instruction cycles. The processor's relatively large amount of memory (16 kB) grants further optimization of the motion algorithm, while it's powerful ADC module allows fully automated acquisition of analog signal values. Furthermore, this processor features a fast UART interface and Direct Memory Access for an efficient streaming of data to the computer.

The part chosen in this project is packaged in a 44 lead plastic thin quad flatpack, because of its ease of assembly and the number of available i/o pins for debugging purposes, but the same chip is also available in a lighter 28 lead plastic quad flat for use in very weight restricted applications.

2.2 Printed Circuit Board Design

For testing this new microcontroller with the `MDC2D`, a new printed circuit board (PCB) had to be designed. The new PCB was loosely based on a old design featuring a `C8051F320` microcontroller from SiLabs with an integrated High Speed USB stack [8] (this board was designed by Markus Bernet [1]; the mistakes mentioned in [17] were corrected).

The purpose of this PCB is to provide a testing platform for the interaction of the `dsPIC33F` and the `MDC2D`, as well as the other parts described below. It should facilitate development and debugging of the firmware and be a basis for a smaller version that will be developed later (see section 4.1).

The layout of the printed circuit board can be seen in figure 4; it features the following components

- `MDC2D`: The motion detection chip described in section 1.2
- `dsPIC33F`: The microcontroller described in section 2.1 that reads out the pixel values from the `MDC2D`, performs the motion calculation and interacts with the host-sided software.
- `FT232RL`: This microchip [11] is an UART [18] to USB [10] interface. This part is necessary because the `dsPIC33F` has no integrated USB stack, but the host-sided software (see section 2.4) communicates via the USB port. It basically tunnels an asynchronous serial communication through the USB protocol to the computer where its driver simulates a virtual serial communication port.
- `AD5391`: This high precision digital to analog converter [3] is not actually necessary for normal function of the board because the `MDC2D` has an integrated bias-generator that has been proven to work correctly. It has been included nevertheless for developing firmware code handling the `AD5391` that can be used in further work featuring the same microcontroller. In the final design this chip is only used to provide the bias **`Vrefminbias`**, because the corresponding node is internally hooked up to a wrong connection, making it impossible to bias the gate of the attached nFET transistor with a subthreshold value.
- `TPS79333`: Two low-noise linear voltage regulators [7] were used for providing a separate analog and digital power supply.

The PCB consists of a top layer, a bottom layer and two inner layers; it was designed using *Altium Designer (Winter 09 Edition)*, manufactured by PCB Pool[1] and soldered by hand, without using a reflow oven.

The routing of the new connections (the left half of figure 4) was done by hand and the analog signals (`scanvrecep`, `scanvlmc`) were routed in a way to avoid crossing of digital signals. Care was also taken that the high frequency digital signals (especially the connection between the `dsPIC33F` and the `FT232RL`) were kept as short and straight as possible. The two inner layers of the PCB make up for the different power planes and are separated in digital and analog regions. This was hindered by the fact that the `AD5391`, the `dsPIC33F` and the `MDC2D` all feature analog as well as digital pins and the other considerations constrained the free placement of these components. The `FT232RL` and the `TPS79333` were kept next to the USB port that also constitutes the board's only power source. All bypass capacitors were placed as close to the respective pins as possible and the resistance of the main power connections was minimized by using copper pours and multiple vias wherever possible.

In further use of this design, the following points should be considered

[1]http://www.pcb-pool.com

- It is already mentioned in [17] that the MDC2D is not packaged correctly (silicon die out-of-center) and therefore the lens has to be mounted horizontally and adjusted to an off-center position by hand to get as much light onto the chip as possible.
- When soldering the parts of the board, TPS79328 were used instead of the originally planned TPS79333, resulting in a power supply of $V_{dd} = 3.28\ V$ instead of $V_{dd} = 3.33\ V$. This difference of 50 mV does not affect the digital part and most of the analog, but it results in a reduced gate-source voltage of the pFETs when their operating point is set by the AD5391(refer to [17] for the optimal voltage biases). The current biases for the internal bias generator need not be adapted because they produce themselves voltages relative to V_{dd}.
- As mentioned above, the AD5391 is not needed in a new design; the bias Vrefminbias does not need to be set very accurately and could be provided by a resistive voltage divider.
- The readout of the pixel values is quite noisy (see section 3.1). This could probably be ameliorated by re-routing some of the wires, reducing the number of vias and keeping delicate signal lines short and further away from digital signals.
- The connector used for programming the dsPIC33F is too close to the USB connector. In the current PCB, the connectors emerge on the backside to fix this problem.

2.3 Firmware Design

The main purpose of the program running on the dsPIC33F is to read out the pixel values from the MDC2D and calculate the global motion vector. For developing this, a more general framework had first to be built that could interact with the computer as well as with the MDC2D so that the read-out from the chip could be visually verified and the calculated motion vectors could be compared to motion vectors calculated on the computer. Besides fulfilling this task, the firmware should also be reusable for further projects using the same setup.

When creating this new firmware from scratch, a major task was to develop a bidirectional channel of communication with the computer (see section 2.4). The dsPIC33F has an UART (universal asynchronous receiver/transceiver) interface that allows bi-directional exchange of asynchronous data over two serial lines with another chip. On the PCB developed for this project, the FT232RL tunnels this asynchronous data over a USB interface to the computer, where the appropriate driver (that is included in most operating systems, because this chip is so wide-spread) emulates a

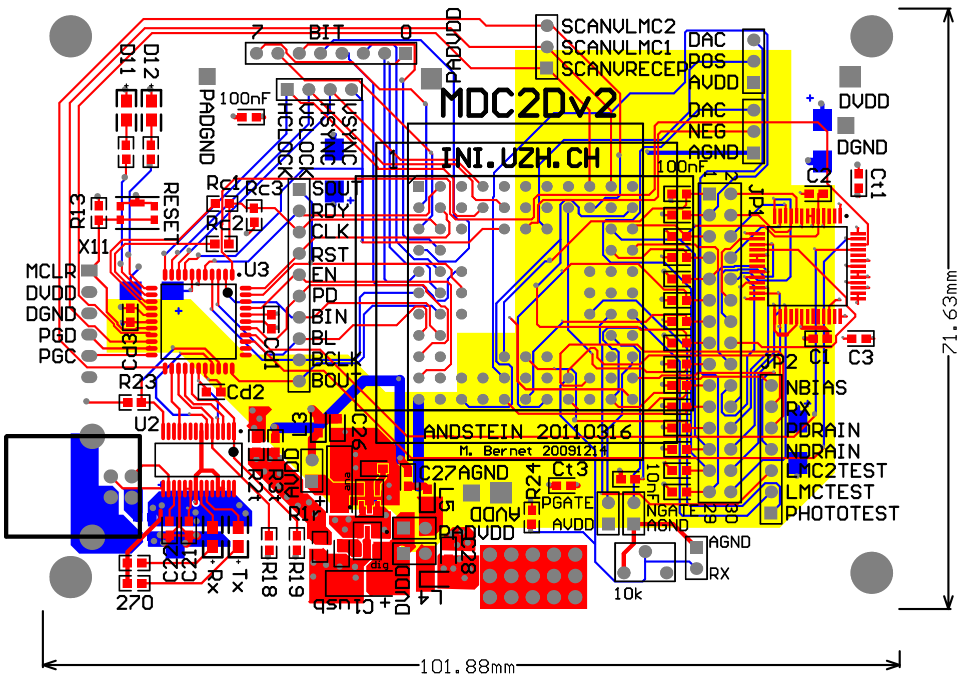

Figure 4: Layout of the PCB, showing connections on top (red), connections on bottom (blue) and overlay (black); the two inner power planes are not shown for the sake of clarity; the yellow polygon indicates the part of the power planes that is dedicated to the analog power supply

`RS232` interface, over which data can be exchanged with the microchip; see figure 5 for an illustration.

This bidirectional interface can be used to send/receive date to/from the computer. In this project, the UART communication speed was set to 618984 baud (the available baud-rates are a function of the microcontroller's clock speed). Because this is a relatively slow speed compared to the 40 MIPS at which the `dsPIC33F` runs, it would be very inefficient to stream data via blocking input/output routines. In this project, the microcontroller's direct memory access (DMA) capabilities were fully utilized and the streaming of large chunks of data – such as frames – is done by the DMA module while the microcontroller's core can perform other tasks.

As in any communication, transmission errors can occur and because the UART interface implements no means of error correction, it can happen that single bytes of the data stream get lost and a system for resynchronization must therefore be implemented in software. In this project, all data is streamed in simple messages (see figure 6) that allow verification of the integrity of the message frame and a fast recovery in case some characters are unexpectedly lost.

The computer controls the `dsPIC33F` by sending commands over this serial interface and gets answers in return, indicating success or failure of the requested operation. The original implementation of the RS232 standard includes two special lines (RTS/CTS) that can be used for hardware flow control (i.e. indicating when new data can be sent/received). The current design does not need any flow control, because it is not critical when some few messages are lost due to a buffer overrun, and uses one of these lines (RTS, see figure 5) in a different way: the computer signals that it will send a command by asserting the RTS line. The firmware then immediately executes a interrupt service routine (ISR) that parses the incoming command and reacts appropriately. Asserting a special signal line when sending a command has two advantages: First, the firmware reacts to the command even if it is stuck in some unexpected state, because the ISR is executed independently. Second, because the command data is sent while asserting a special line, it can be seen as "out-of-band" data and it would be possible to stream data from the computer to firmware and at the same time transmitting commands, although such a streaming was not used in the current project.

For details on the interaction between the microcontroller and the `MDC2D` as well as the `AD5391` see **??** or refer to the documentation included in the firmware. The calculation of the global motion vector is explained in section 2.5. One interesting implementation detail can be seen in figure 7. Instead of looping through the 20×20 pixel array in the main loop, digitizing the analog values and writing them into the RAM, the `dsPIC33F` only sets up a precisely timed ISR that in turn triggers an automated ADC conversion (resulting in a switch being opened, disconnecting the ADC's sampling ca-

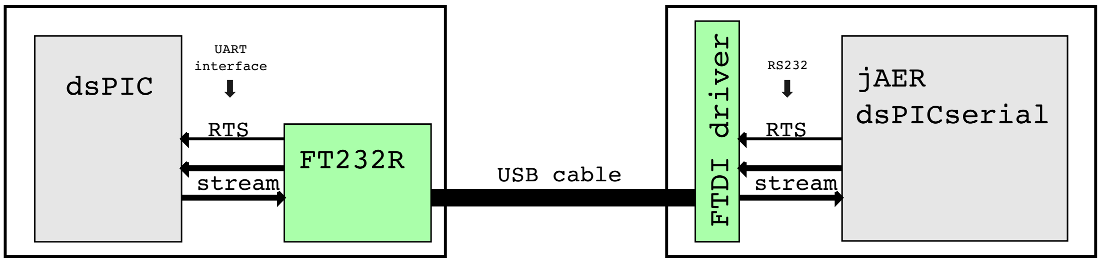

Figure 5: The left side of this figures represents the PCB with the `dsPIC33F` that communicates over its UART interface with the `FT232RL` that in turn tunnels the serial data over a USB interface to the computer, where a driver emulates a serial line from which the stream can be read. The RTS signal can be asserted by the host-side software and results in a change of potential on a digital pin on the microcontroller's side.

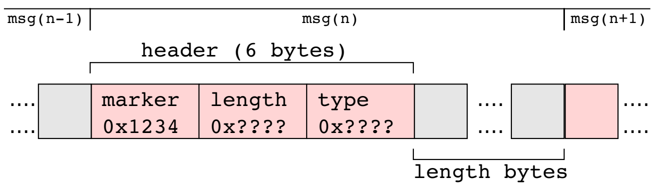

Figure 6: This figure illustrates a short extract of the message stream. A message consists of a minimal header (in red), 6 bytes in total (i.e. three 16 bit little endian words), followed by a variable message content (green), whose length is indicated by the `length` field. The `marker` can be used to identify the beginning of the message header in case the synchronization is lost.

pacitor from the input pin), before moving the `MDC2D`'s scanner to the next pixel. The converted value is automatically written to the RAM by DMA. This results in a precisely timed sampling of the analog values (the sampling interval is set by the timer calling the ISR) and a minimal workload on the `dsPIC33F`'s core that can perform other computation during the frame acquisition.

2.4 Host-Side Integration

The firmware described in section 2.3 streams live images and global motion vectors to the computer (the "host"). The computer on the other side has to instruct the `dsPIC33F` how the `MDC2D` should be read out (e.g. what channel to use, how fast the scanner should be moved, etc) and displays the captured data for visual control, while comparing the computed motion vector values

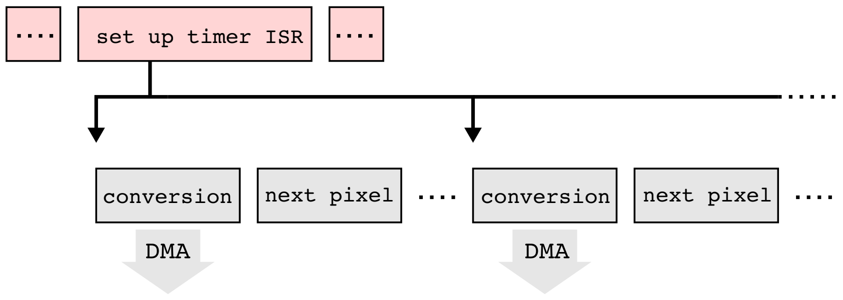

Figure 7: The code executed in the main loop is highlighted in red; after setting up the timer ISR, the main code continues normal execution, while a periodically executed timer interrupt triggers the conversion of the analog signal to a digital value that is then stored into RAM by the DMA module, while the `MDC2D`'s scanner is moved to the next pixel to be converted (by the next timer interrupt).

with the more precise values obtained by performing the algorithm with floating point numbers.

All host-sided software communicates over a RS232 interface with the `dsPIC33F` (see figure 5). This serial interface is emulated by a driver called VCP (for virtual COM port) on Windows and a loadable kernel module called `ftdi_sio` (for Future Technologies Devices International serial input/output) on Linux. This implementation was chosen for its apparent simplicity. Standard serial ports have been supported in all operating operating systems for a long time and there are many tools that can be used to communicate with a device over a serial line using a text-based protocol, such as the one used in this project. Refer to figure 8 for an overview of the different software layers that are used to communicate with the `dsPIC33F`.

Because the `dsPIC33F` communicates over a UART interface with the `FT232RL`, there is no possibility to receive any data apart from sending it through the serial line or by asserting one of the two hardware flow-control lines (RTS/CTS). Furthermore, the serial communication settings (baudrate, parity, stop-bits) have to match on both sides. If these settings are not adjusted correctly, no data can be sent/received over the serial line.

All host-side software from this project is included in jAER [5]. The package `ch.unizh.ini.jaer.projects.dspic.serial` implements the asynchronous communication with the `dsPIC33F`. After choosing a serial communication port and setting the baudrate, Java programs can use this package to send commands to the microcontroller and they are asynchronously notified by any incoming messages, answers to previously sent commands or unexpected events (such as a loss in synchronization or a time-out of an expected answer). This package also contains an example application that can be used to control the firmware via a command-line, while dumping the

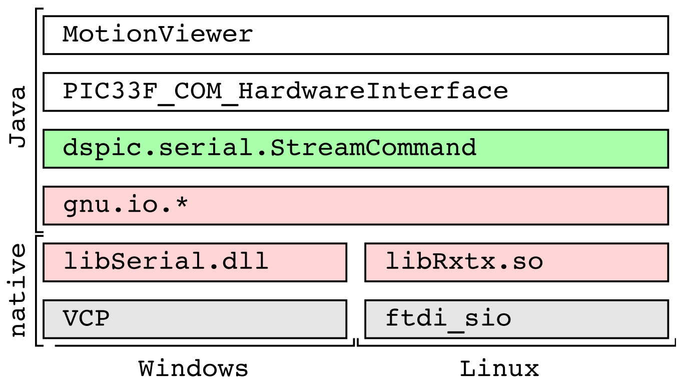

Figure 8: This figure shows the different layers that are used in the host-side software to enable a Java application to interact with the `dsPIC33F`. The class colored in green is the main interface between the actual application (in this case the motion viewer from the jAER package, but this could be any other Java application) and the underlying serial interface/operating system. In this case, the `rxtx` package was used (colored in red), but this part might be exchanged by FTDI's `D2XX` driver in the future for better Linux support.

received answers and providing a simple display for the streamed frames and the global motion vector.

Tobi Delbrück developed a Java framework for other motion detection chips that is capable of displaying streamed data and interactively changing bias configurations, called `MotionViewer` (also included in jAER [5]). Reto Thürer extended this software so it is capable of performing different motion algorithms on the streamed data (among others the one presented in section 1.3 and used in this project). Therefore, a new hardware interface called `dsPIC33F_COM_OpticalFlowHardwareInterface` was integrated into jAER that enables the framework to use the new hardware described in section 2.2 into the existing application. Also, some new features, such as recording and a command-line interface, were added specifically for the new hardware (in a new GUI element called the *hardware interface control panel*).

A drawback of the presented implementation is that some operating systems (e.g. Linux) do not support arbitrary baudrates and therefore cannot communicate with the `dsPIC33F` at 618984 baud. It is still possible to use the framework and stream data at a standard 115.2 kbaud, but this is not enough to stream frames at 60 fps. This problem could be fixed by using FTDI's `D2XX` driver that is available for Linux, Windows and Mac OS X. Because all host-side software strictly adheres to a layered communication stack (see figure 8), it would be sufficient to change some code in the `dspic.serial.StreamCommand` class when the driver was to be switched. All the other parts would continue to work as before.

2.5 Optical Flow Algorithm Implementation

The equations (8) to (12) from section 1.3 must first be reformulated for discrete values; in the remainder of this report, the *reference amount* is set to 1 pixel (i.e. $\Delta x_r = \Delta y_r = 1$) and the following abbreviations are introduced:

$$\begin{aligned}
f_\Delta x(x,y) &= f_2(x,y) - f_1(x,y) && (16)\\
f_\Delta y(x,y) &= f_4(x,y) - f_3(x,y) && (17)\\
\Delta f(x,y) &= f(x,y) - f_0(x,y) && (18)\\
a &= \sum_x \sum_y (f_\Delta x)^2 && (19)\\
b &= \sum_x \sum_y f_\Delta x \cdot f_\Delta y && (20)\\
c &= \sum_x \sum_y (f_\Delta y)^2 && (21)\\
d &= \sum_x \sum_y \Delta f \cdot f_\Delta x && (22)\\
e &= \sum_x \sum_y \Delta f \cdot f_\Delta y && (23)
\end{aligned}$$

As the `dsPIC33F` supports additions, subtractions, multiplications and divisions, equations (19) to (23) and finally (15) could all be solved by using its arithmetic logic unit (ALU), which supports signed/unsigned calculations with integer as well as fixed point fractional numbers. In this project, a special approach has been chosen to exploit some of the microcontroller's features:

1. The digital signal processing (DSP) unit has a powerful *multiply accumulate* (MAC) instruction that is capable of fetching data from within two different memory locations, performing a 16×16 bit multiplication, adding the sign-extended result to a 40 bit register as well as updating two registers holding memory locations, *all in one instruction cycle.*

2. The microcontroller is equipped with 16 kB of RAM, allowing generous storage of intermediate results.

3. By performing all calculations with integers, the inaccuracy that is inherent to the microcontroller's limited 16 bit wide number representation is minimized.

Using the MAC instruction conditions an important constraint on the memory organization of the data (see figure 9) : because two words of data can only be fetched at the same time as performing a multiply accumulate instruction if they are located on two different memory buses, all different combinations of data words used in the discrete integrations of equations (19) to (23) must reside in these different memory regions (called X and Y). This was achieved by storing Δf in the X data region as well as in the Y data region.

Using integers for the intermediate results is problematic, because different overflows can occur that all must be handled differently. The current implementation handles the following overflows

- $\widehat{\Delta x}$, $\widehat{\Delta y}$: by using $d/2$ instead of $2d$ and $e/2$ instead of $2e$ in equation (15), the algorithm actually calculates $\widehat{\Delta x}/4$ and $\widehat{\Delta y}/4$ internally. When adding the sign bit before returning the results as signed fractionals, these values are shifted by one bit to the left and therefore $|\widehat{\Delta x}/2|$ as well as $|\widehat{\Delta y}/2|$ must be smaller than one. This restriction poses no problem in the application of the algorithm, since values of $|\widehat{\Delta x}|, |\widehat{\Delta y}|$ that are bigger than 2 become very imprecise due to the algorithms limitations (see section 1.3).

- a from (19) : if a should overflow, it is shifted until it fits into one word and all the values divided by a (when resolving (15)) are simply shifted by the same amount of bits before performing the division.

- denominator and nominator of $\widehat{\Delta y}$ in (15) : should one of these values overflow, they are simply both shifted in parallel prior to the division until they fit into one word.

- c from (21) : is only stored in the MAC register and can therefore be up to 40 bits long.

Some other overflows, such as of b in (20) simply result in an error. This also happens for other rare conditions, such as $a = 0$ or $c - \frac{b^2}{a} = 0$. For all the data gathered during this project, this conditions were very rare and the work to prevent them was therefore not deemed necessary (see section 3.2).

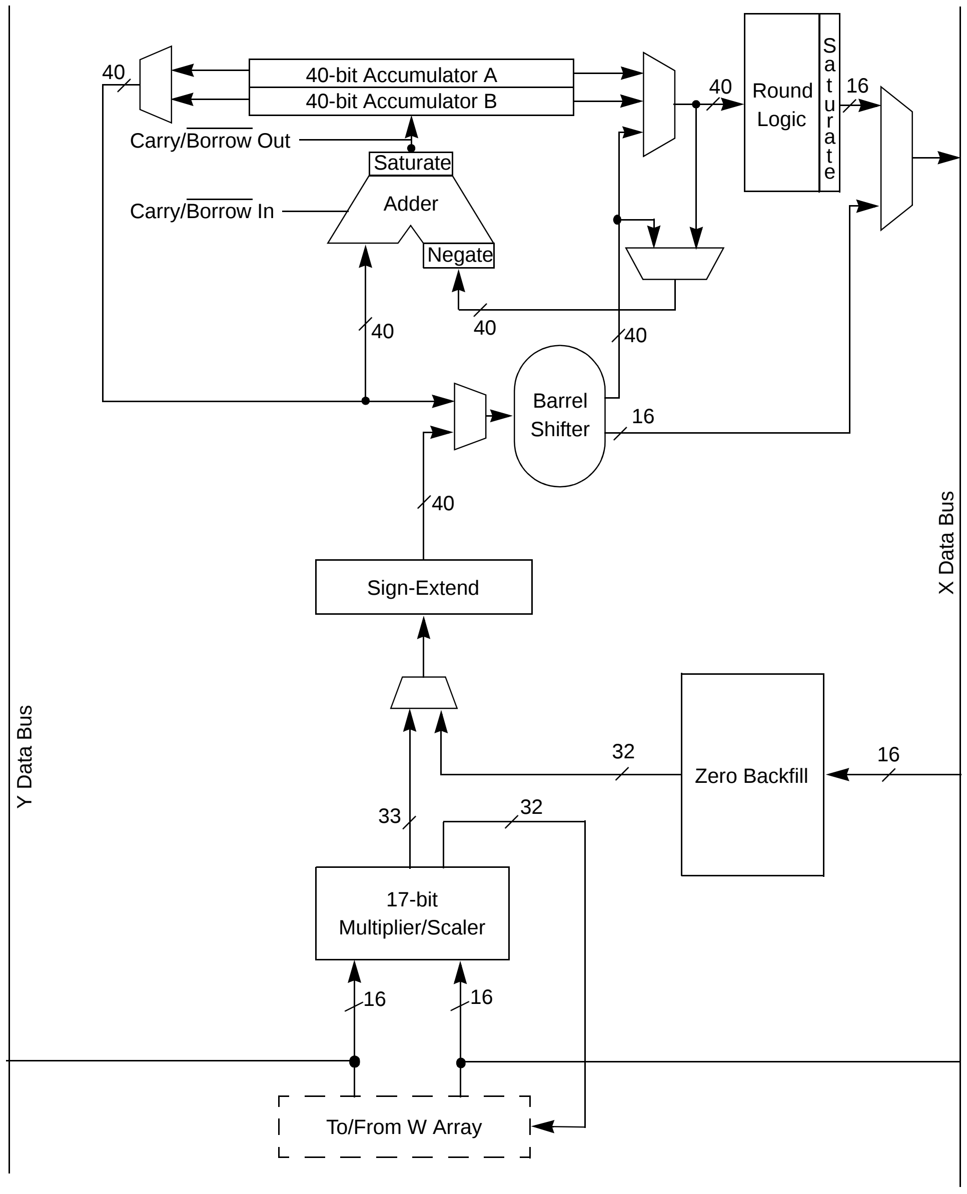

Figure 9: Block Diagram of DSP Unit (taken from [13])

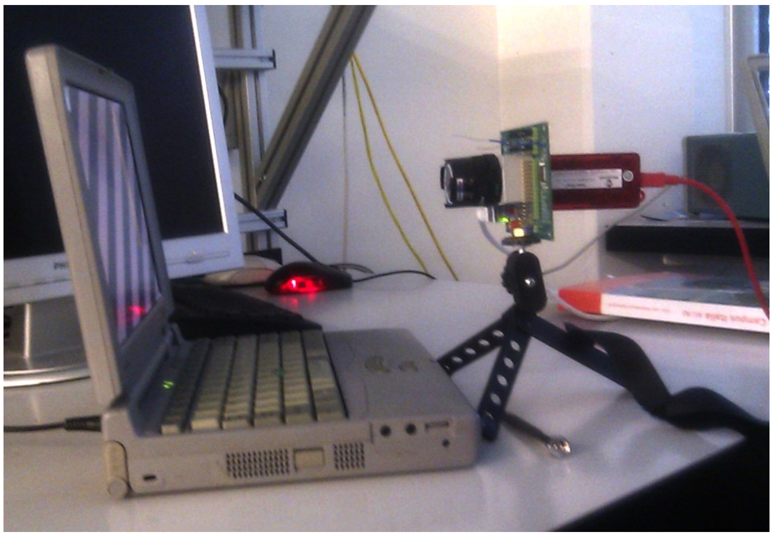

Figure 10: Setup for quantitative assessment of the global motion calculation.

3 Results

3.1 Global Optical Flow Estimation

The quantitative recordings were performed using an old Toshiba 610CT notebook with a monochrome passive LCD screen, running a program called STIMULUS that was previously developed at INI for testing motion detection chips. The visual stimulus consisted of high contrast bar gratings with a sinusoidal intensity gradient moving in all four directions at varying speeds. The chip was mounted with 4 mm lenses from Computar (1 : 1.2, 1/3) and the focus plane was adjusted to the screen. The setup can be seen in figure 10. The biases used for the recordings are stored in the file `MDC2D_dsPIC.xml` and can be found in the jAER [5] project's bias directory.

The speed scale used in the plots is arbitrary, but linear: a speed of 1 corresponds to 7.25 cm/s on the LCD. The plotted $\Delta x, \Delta y$ values are the output from the algorithm (i.e. from solving the matrix in equation (14)). Because the same $\Delta t = 20\ ms$ is used in all plots (apart from figure 12 the left-hand plot in figure 11), these values scale directly with the actual global motion. In order to reduce a recording of motion vector values to a single point, a normally distributed probability distribution was *assumed* and the center of this distribution as well as the putative standard deviation are plotted. Unless noted otherwise, this calculations were done on the raw data of the flow computation on values read from `scanvlmc`; in some cases (such as in figure 14 and the right side of figure 16), the data was also averaged in time with a FIR filter with the coefficients [0.2 0.2 0.2 0.2 0.2].

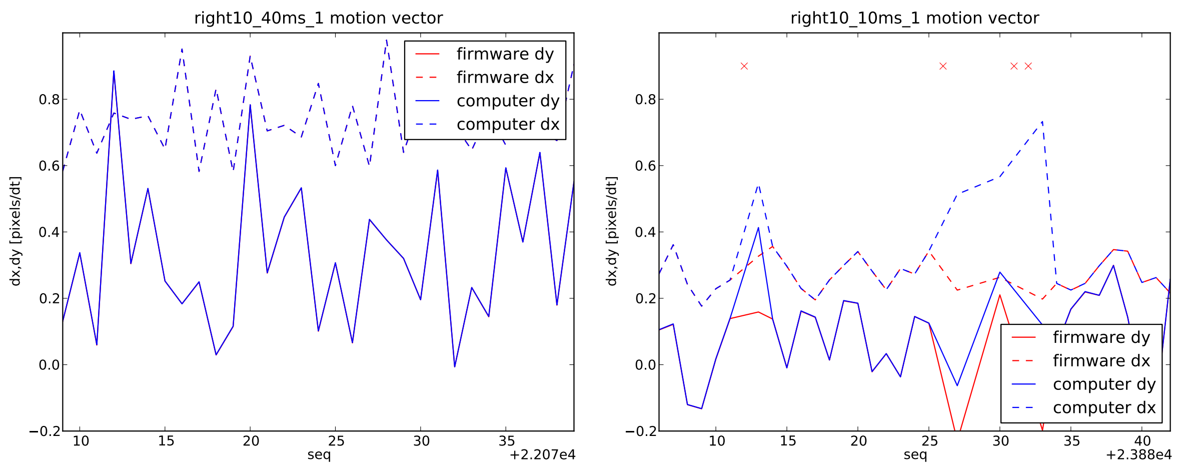

Figure 11: An extract of a recording comparing global motion calculations performed on the computer [17] and on the `dsPIC33F`. At 25 fps (left plot), the recordings are perfectly equal, because the host sees the same pixel data as the firmware and the calculations therefore match. On the right side, the framerate was increased fourfold. The computer now misses some frames (marked with a red x at the top of the plot) due to the jAER polling at only 60 fps. This results in a *overestimation* of the movement by the computer. These two recordings also illustrate how the Δx and Δy decrease as Δt is decreased while the same stimulus is shown.

In a first step, the global flow results of the global flow computation from the firmware were compared to the values calculated on the computer. Because they are identical (as can be seen in figure 11), all results mentioned in [17] can in principle also be applied to the new hardware.

The algorithm is expected to give a precise estimate of the global motion only as long as the $\Delta x, \Delta y$ are below 1 pixel/Δt [15]. This finding is illustrated in figure 12. To assess the linearity of the global motion estimation, recordings over some seconds[2] were done for all four directions over a range of different speeds. The results of these measurements can be seen in figures 13 and 14.

The same data that was used to plot figures 13 and 14 can be seen in figure 15, where not only the global motion vector's length, but also its direction was plotted. Despite the linearity of the length shown in the previous figures, there is apparent non-linearity regarding the global motion vector's direction. A possible explanation for this aberrations could be the chip's misalignment (see section 2.2) and a resulting blurring and/or distortion in some parts of the optical sensor (see figure 18). These non-linearities might also be the source of the different slopes of the linear fits in figure 13.

[2]longer than 4 seconds for every measurement with a mean of 10 seconds

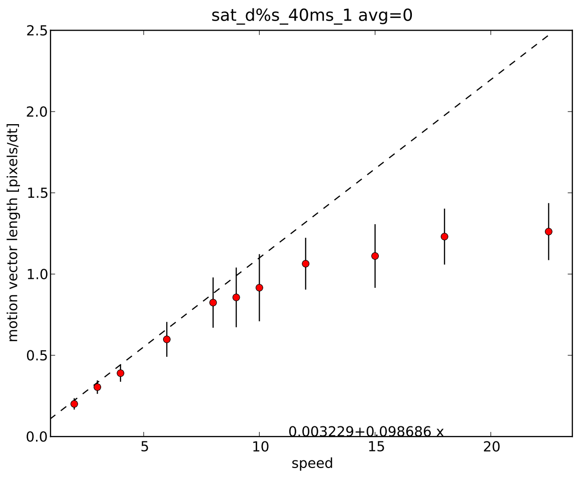

Figure 12: Shows how the motion estimation saturates as the real motion vector exceeds 1 pixel between two adjacent frames. The recording was carried out at 25 fps in order to achieve saturation at relatively low speeds that still produced clear images on the LCD screen (this was assessed qualitatively by looking at the frame data).

3.2 Hardware Performance

The PCB from section 2.2 was designed with three different power planes: analog, digital and "pad" (digital supply for the `MDC2D`). Because these nets are connected to the respective analog/digital voltage regulators with a jumper, the power supply for the different parts of the board can be interrupted and supplied by other means. For measuring the supply current, each of these connections was opened and a external power supply was connected over a 100 Ω resistor. The voltage was then regulated to reach exactly 3.28 V (corresponds to the operating voltage generated by the `TPS79328`) and the voltage drop over the resistor was measured and converted into current.

The results of this measurement – performed in normal operating conditions (streaming frames and calculating global motion vectors, without any energy saving enabled) – can be seen in table 1. Because three different chips are connected to the same analog power net, their respective current drain from analog V_{dd} could not be assessed directly. When all the `MDC2D`'s biases are turned off (by pulling its pin `pd` high), its power consumption approaches zero. The remaining current drain from analog V_{dd} is then mainly due to the `AD5391` and the analog modules of the `dsPIC33F`, while the motion chip's power consumption is represented by the difference in the measurements. The `FT232RL`'s power consumption was not measured, because it has to be powered from the USB cable and because it is only present on this evaluation board. Also, the `MDC2D`'s digital power consumption could not be

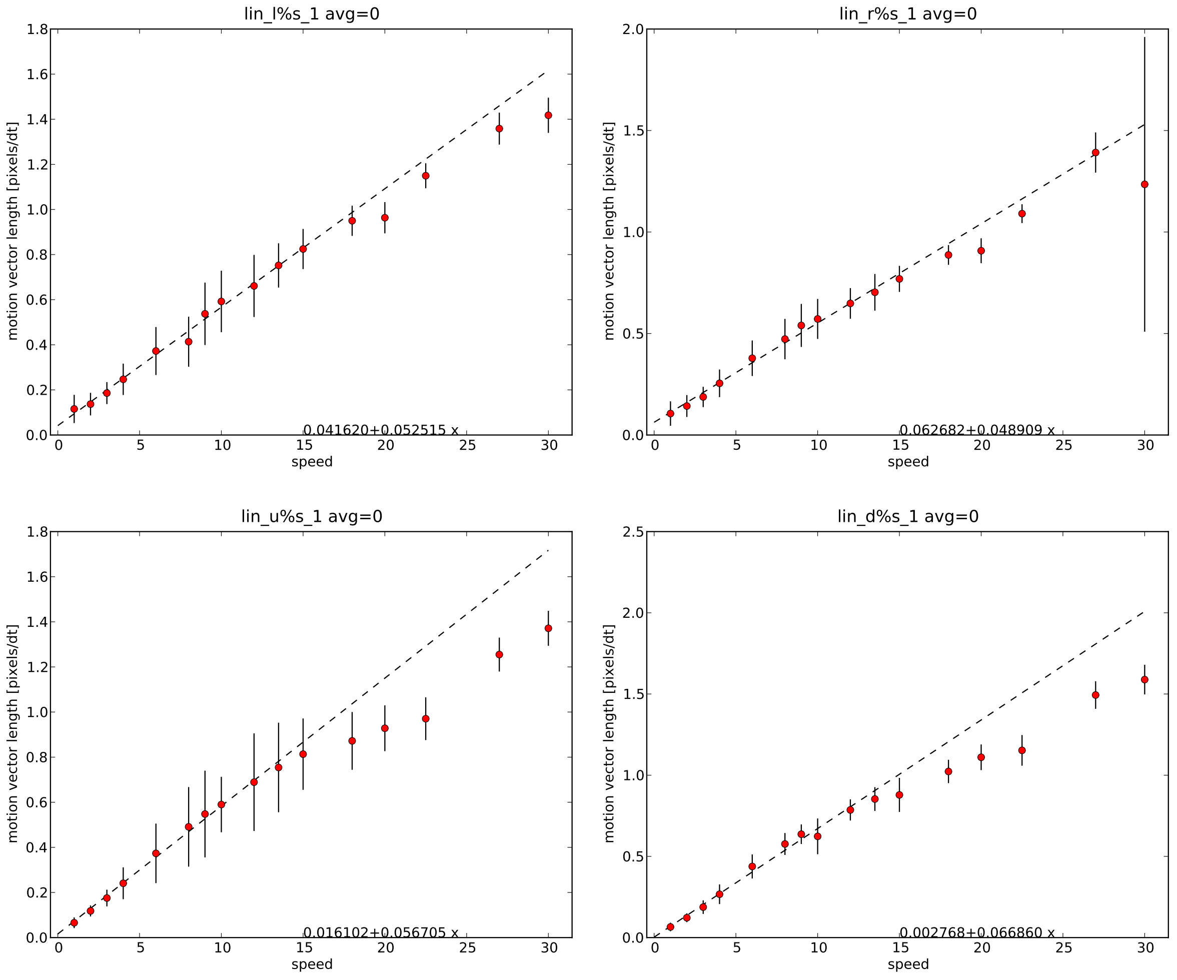

Figure 13: These plots show the average length of the motion vector (red dot) as well as the presumed standard deviation (vertical black line) for different recordings at varying stimulus speeds. The upper figures show the results for horizontal motion (left plot leftwards, right plot rightwards) and the lower row shows the results for vertical directions (left plot upwards, right plot downwards). The linear fit (black dashed line) was done on the values with a vector length below 0.8 pixels/Δt. As expected (see figure 12), the motion vector is underestimated at values > 1 pixel/Δt.

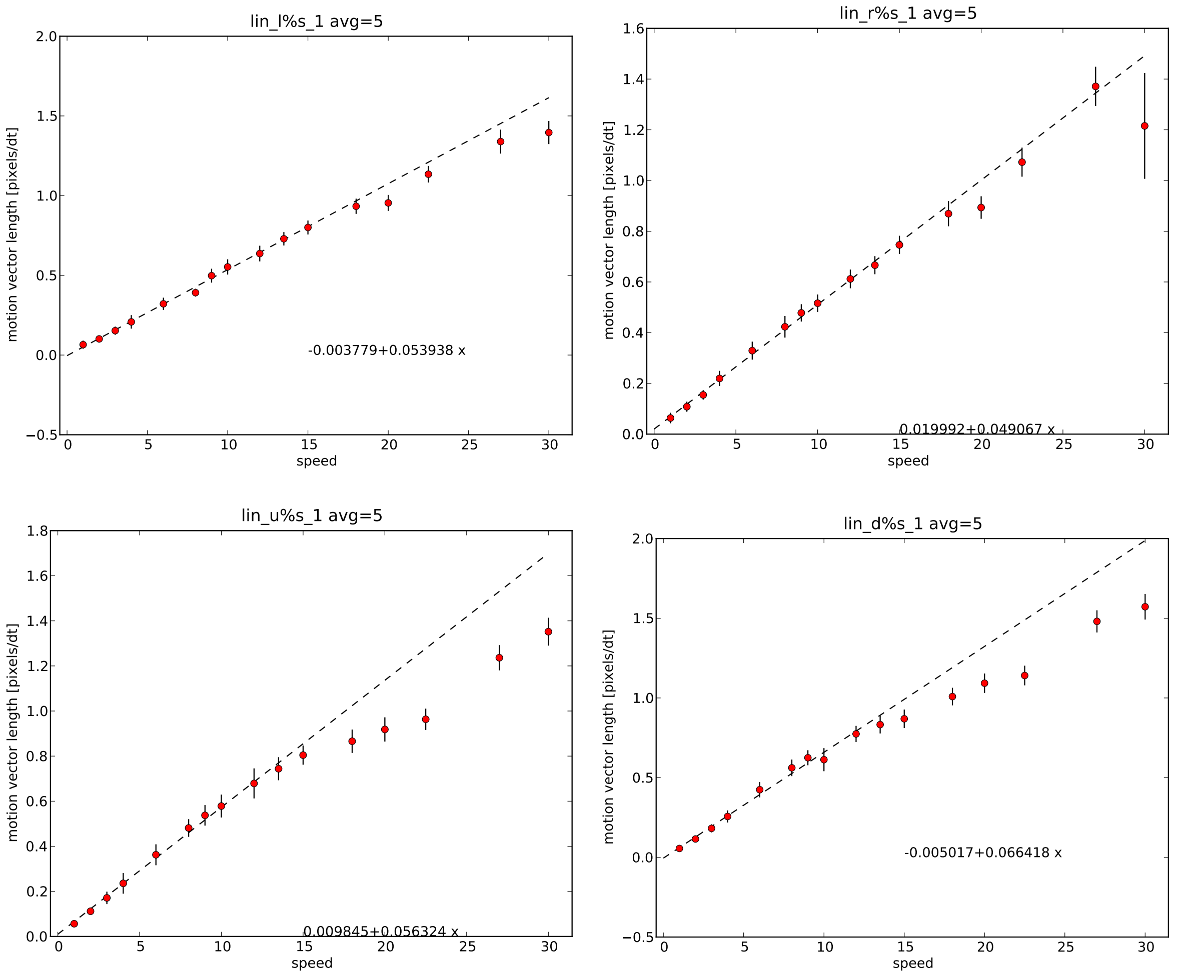

Figure 14: Shows exactly the same data as figure 13, with the only difference that the raw data of the recordings was *averaged* over 5 frames, resulting in a much smaller deviation from the mean (see also figure 11).

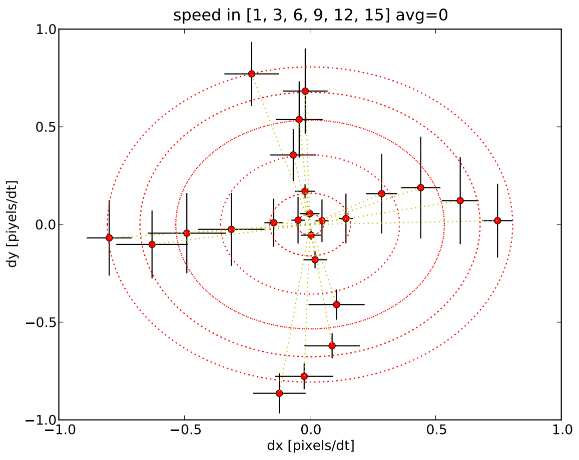

Figure 15: This plot shows a subset of the data that was already used to produce figures 13 and 14, but showing the mean of Δx as well as Δy and their respective presumed standard deviations. The red dashed circles correspond to the mean of the length of the global motion vector in the four directions at the same speed.

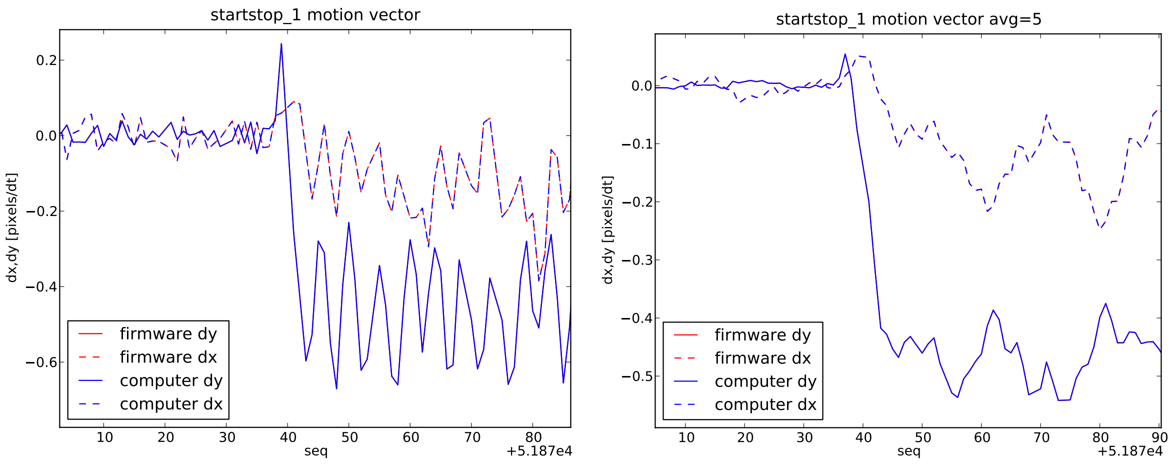

Figure 16: This graph shows the response to a sudden onset of motion. The transition from one steady state to the next is achieved in some few frames and a temporal averaging with 5 FIR coefficients does not noticeably delay this response (the individual graphs encode the same information as in figure 11).

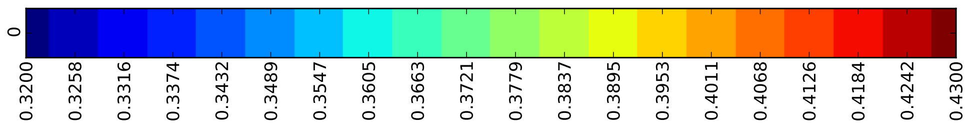

Figure 17: Color scale used for all the figures representing raw frame data; 1.00 corresponds to $aV_{DD} = 3.28\ V$ and 0.00 corresponds to $aV_{SS} = 0\ V$, as measured by the `dsPIC33F` on its analog port.

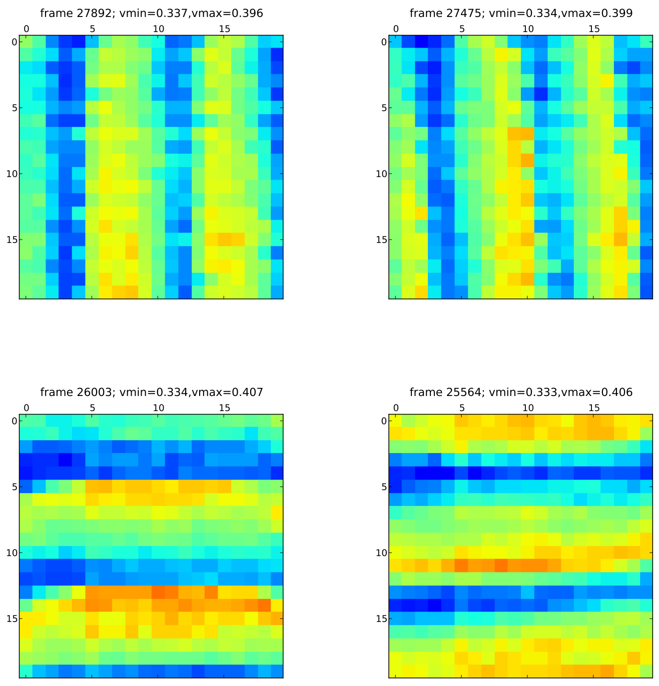

Figure 18: This figure shows snapshots of a moving bar stimulus in the four directions (top-left: leftwards, top-right: rightwards, bottom-left: upwards, bottom-right: downwards). The high-pass filtering property of the LMC circuit is illustrated by the fact that the leading edge creates a stronger signal. Although hard to see, it seems that the left border as well as the uppermost part of the picture are slightly out of focus (the blurred image would result in less contrast and therefore a smaller signal). Furthermore, the horizontal bars seem slightly bent (with a convex curvature upwards; this effect is less visible in the bars moving downwards due to their contrast enhancement at their lower edge). These findings could be an explanation of the non-linearities that can be seen in figure 15.

Chip	Measured Current	Power Consumption
MDC2D (analog)	2 mA	6.6 mW
dsPIC33F (digital)	50 mA	164 mW
other analog	16.8 mA	55.1 mW

Table 1: Power consumption of the different parts of the PCB; "other analog" is mainly the AD5391(the dsPIC33Fis not expected to draw a substantial current from its analog V_{dd}).

determined, because analog and digital power supplies are not completely separated on-chip and the digital supply current, which is expected to be very small compared to the analog supply current, is therefore provided indirectly by the analog rail when "pad" V_{dd} is disconnected.

In total, the current design is expected to consume approximately 230 mW at 3.3 V (excluding the FT232RL). But this figure has to be approached with caution, because the dsPIC33F's datasheet [13] cites an operating current of up to 74 mA. The power consumption of a new PCB design (see section 4.1) can easily be optimized by omitting the AD5391.

The firmware performance was assessed as follows: first, a very precise timing function was developed in assembler. After verifying the code in the MPLAB simulator (the clock speed of the simulator was set to 80 MHz to match the 40 MIPS on the actual device), the timing routine was used to generate a 10 min delay on the hardware which was in turn controlled by hand with a stopwatch – 10 min and 2 s were measured, which corresponds to a relative error of less than half a percent. The timing function was used to verify the accuracy of an interrupt-driven timer which was in turn used to measure the performance of the code. This interrupt driven timer is called every 10 μs when activated. This corresponds roughly to 400 instruction cycles and the interrupt is therefore not expected to result in a significant slow-down (the ISR executes in 6 instruction cycles and the setup and return-to-code make up for 8 instruction cycles [12]).

During the image acquisition, the pixels are sampled individually by performing the following steps : *sampling* an analog value from an analog input pin (during this time the internal sample capacitor is filled), *converting* this analog value to a digital representation and telling the MDC2D's scanner to multiplex the next pixel to the analog output pad. The conversion time for a 10 bit value is $12 \cdot T_{AD}$ which translates to 900 ns (75 ns is the minimum ADC clock cycle). The sample time is set by the timer interrupt interval (see section 2.3). Setting an interval of 2.5 μs per frame has shown to be sufficient for moving the scanner and sampling the new value (assessed qualitatively by comparing captured frames, see figure 19) and results in a total acquisition time of 1 ms for a 20×20 pixel frame.

The implementation of the motion algorithm from section 2.5 performs

Operation	time
Image acquisition	1 ms
Algorithm performace	0.12 ms

Table 2: How much time the firmware spends on acquiring one image from the `MDC2D` and how long it takes to perform the Srinivasan algorithm on two frames.

a 7 instruction long cycle over the whole frame (minus its border pixels) to create the "reference images" and then performs 5 MAC instructions over these arrays to calculate (19) to (23). Since these instructions take only one cycle and because the other instructions are not repeated a significant number of times, the whole calculation is expected to take approximately $(7 + 5) \times 320 = 4320$ instruction cycles; this would correspond to 108 μs at 40 MIPS. The time stopped via ISR has shown to be 120 μs and fits well with the estimate, considering that the estimation did not include any divisions, bit shifting, preparation of the loops and other other glue code.

It is somewhat harder to assess the stability of the implementation, but during the recordings of all the data for this report, the firmware didn't crash a single time after a successful connection to the host. For producing figure 13, 12832 messages containing pixel and global motion data were recorded. The firmware generated 35 "overflow" errors (see section 2.3), out of which 33 were due to Δx or Δy being outside the maximum 2 $pixels/\Delta t$. This leaves only 2 errors due to an "internal overflow" or division by zero.

3.3 Software Framework

The software framework that was developed for this project (see sections 2.3 and 2.4) can easily be adapted for new projects that are based on `dsPIC33F` microcontrollers, use the USB port for communication and Java for the host-side integration. The following procedure is suggested for development of new software based on the framework presented in this report:

1. In a first step, the firmware should be adapted to the specific needs. Please have a look at the documentation of the firmware source code (in doxygen[3] format), especially the files `main.c`, `message.[ch]` and `command.[ch]`

2. For testing the new firmware, the class `StreamCommandTest` from the package `ch.unizh.ini.jaer.projects.dspic.serial` can be used to establish a connection, as well as sending text commands and receiving the firmware's answers; the `set` and `get` commands provide a convenient way of dumping status informations and setting parameters. The

[3]a source code documentation generator; see http://www.doxygen.org

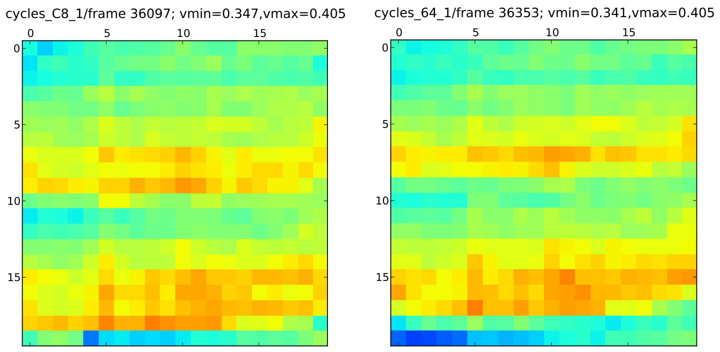

Figure 19: These two frames were acquired with different settings for the ISR timer interval (see section 2.3); on the left side, a setting of 5 μs per pixel (resulting in 2 ms per frame) was used while on the right side, only half the time was spent on sampling the the individual pixels, resulting in an acquisition time of only 1 ms per frame. The shorter acquisition time does not seem to adversely affect the image quality (the two frames are not identical because they were produced by a moving stimuli that does therefore not exactly match in two different frames).

class `StreamCommandTest` currently contains some GUI elements that are specific for this project, but this can easily be changed or extended.

3. In a second step, a complete Java application can be developed, using the class `StreamCommand` from the package `ch.unizh.ini.jaer.projects.dspic.serial` to interface asynchronously to the firmware for streaming messages and sending command sequences. The documentation of this class should be read carefully.

It should be stressed that the current version of the different packages is by no means mature; it should rather be seen as a development platform that can be used as a starting point. If this platform is used in further projects, people working on it are kindly asked to separate their new code into application specific tasks and general improvements of the interface that can then be merged into the jAER tree so that new projects can be based on those improved versions.

4 Outlook

4.1 Miniaturization

For use of the design presented in this report on a flying platform, where severe weight restrictions apply, the following steps should be considered to miniaturize the dimensions, while maintaining the same functionality:

- The lenses should be replaced by lightweight plastic lenses with a fixed focus that need to be adapted to the particular needs of the application. Because this will change the optical properties, the results from section 3.1 would need to be reproduced. Despite of the lower quality of such plastic lenses, this step could eventually improve the quality of the gathered data because the lenses could not fully be centered onto the chip in this project, due to its off-center packaging.

- The `MDC2D` could be removed from its package and the bare silicon die would then be glued on a board, connecting the pads manually with small bonding wires.

- The `FT232RL` and the USB connector should be replaced by a minimalistic serial interface for development an debugging purposes.

- The `AD5391` would not be needed anymore, because the on-chip bias-generator can be used to set the operating point of the analog circuits. The only bias that has to be provided externally is `Vrefminbias` (see section 2.2). Since this bias is not sensitive regarding its exact value and needs not be changed during operation, it could be generated by a resistive voltage divider.

- The `dsPIC33F` used in this project could be replaced by a smaller variant with fewer pins and depending on the architecture, the microcontroller could be used to perform different tasks, because the implementation presented in sections 2.3 and 2.5 performs the image acquisition and global flow estimation in as little as 1.12 ms.

4.2 Algorithm

The results that were presented in section 3.1 are certainly not sufficiently detailed for directly using the `MDC2D/dsPIC33F` on a flying platform. Therefore, the platform clearly needs to be tested for application specific requirements, that vary widely, depending on the exact use of the global optical flow sensor. For example, if the chip is to be used as a sensor in a closed loop for a control task where large angular velocities can occur and the steering mechanism must react accordingly, the timing requirements might be very demanding and it might not be possible to average the value in time

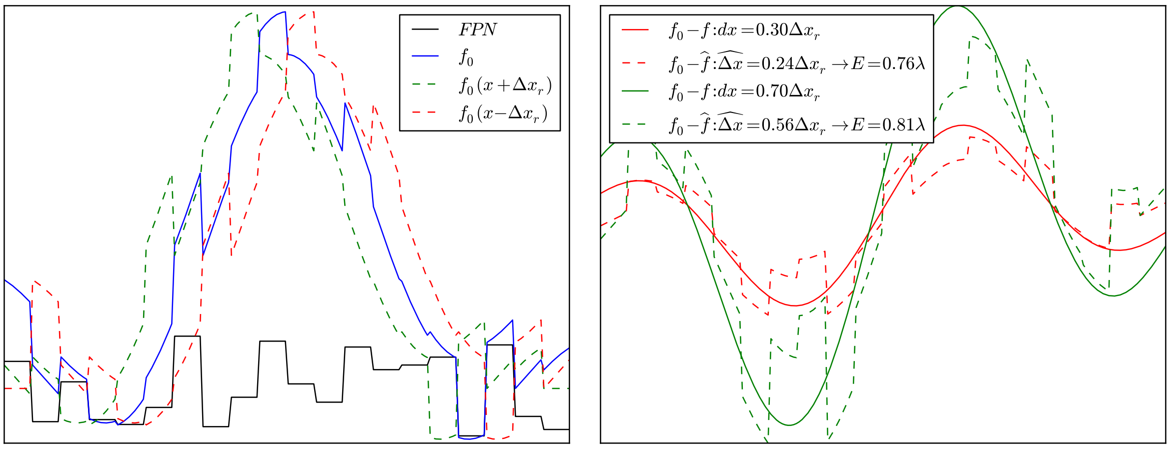

Figure 20: This figure shows the same simulated 1D data as in figures 2 and 3, but this time with some added *fixed pattern noise* (FPN, black line in left plot). On the right side, the effects of the FPN is not visible in the difference between the moved and the original function, because they both share the same noise at the same places. But it is visible in the difference between the original function and its shifted versions, because the FPN has also been *shifted*. This results in a decreased performance of the algorithm compared with the results from figure 3

to decrease noise. In another scenario, where the sensor would be used for odometry[4], the timing requirements would be relaxed and it would be more important to get an unbiased measurement to minimize errors of integrated values.

As mentioned earlier (see figures 13, 14) simple averaging in time can result in a signal that is much less noisy than the raw signal. This can easily be implemented in the firmware, but the details of the averaging filter as well as the adaptation of its parameters depending on the signal is highly application specific.

The algorithm presented here is relatively robust to noise [15]. The mismatch in the analog circuits between the different pixels of the sensor gives raise to *fixed pattern noise* (see figure 21). This noise is different from the noise explored in [15], since it disappears in the difference images of adjacent frames (the offset of every pixel changes slowly in time), but is included in the difference of shifted images; figure 20 illustrates this. The results of motion estimation could probably be improved by filtering out the fixed pattern noise.

[4]The use of data from moving sensors to estimate change in position over time (source: Wikipedia)

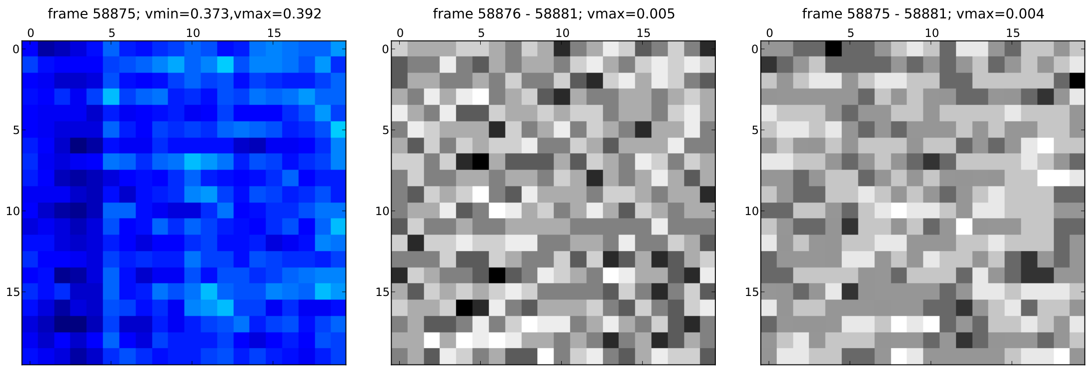

Figure 21: This figure shows two different types of noise that occur in the `MDC2D`: the colored image on the left illustrates the *fixed pattern noise* (FPN) that is mainly caused by the mismatch of the transistors in the analog pixels. This FPN has a peak-to-peak amplitude that corresponds to approximately 25% of the peak-to-peak signal of a high contrast stimulus, such as in figure 18. Additionally, there is also some high frequency noise, as illustrated by the two grayscale images that show the difference of two adjacent frames (compared to an arbitrary baseline image to remove the FPN). The peak-to-peak amplitude of the not-FP noise is relatively small (around 5% of the signal's peak-to-peak amplitude, corresponding to about 4× the LSB of the ADC).

A Download, Usage

All software design and testing was done on a Windows XP SP3 machine. Some programs used in the development are only available for the Windows operating system. The host-side software should theoretically also work under Linux (with an adapted baudrate, see section 2.4), but has not been tested yet.

PCB design : the software used for the PCB design was *Altium Designer Winter 09 Edition.* This program exists only for Windows computers. The project files can be downloaded from the following subversion repository:

https://svn.ini.uzh.ch/repos/shih/chips/MDC2D/dsPIC33F_testboard

Firmware : the source code for the `dsPIC33F` is contained in the `deviceFirmwarePCBLayout/dsPICserial/MDC2D` subdirectory of the jAER project[5]. Please read the `README.txt` and `INSTALL.txt` files included in the project for further instructions.

Host-side software : all code used in this project is also included in the jAER project; it can be found in the directory `host/java/src`. The following steps should be performed to use the new PCB in conjunction with the motion viewer

1. Download the whole jAER trunk (at least revision #2781 that represents the state after the last commit for this project).
2. Connect the PCB to a USB port of the computer. Verify that a new virtual serial communication port shows up (i.e. a new `COMx` port in the Windows device manager or a new device called `ttyUSBx` under Linux). Eventually install a virtual com port (VCP) driver from FTDI[5].
3. Read the documentation of the jAER package `ch.unizh.ini.jaer.projects.dspic.serial`; it contains an executable class called `StreamCommandTest` that can be used to verify the connection to the microcontroller, check the version of the installed firmware and control the microcontroller via a command-line interface. If the connection could not be established (e.g. under Linux), change the baudrate to a standard 115.2 kbaud; of course, a new firmware with the same speed setting has to be downloaded to the microcontroller for this to work; see the firmware documentation.
4. Start the motion viewer by executing the Java class `MotionViewerMain_MDC2D` from the package `ch.unizh.ini.jaer.projects.opticalflow`.
5. Select the "hardware interface" `dsPIC33F_COM` in the combo-box of available hardware interaces in the motion viewer. Choose the right serial port (the one was created when the device was connected). If the jumpers between the `AD5391` and the `MDC2D` are set, voltage biases will be used and the "on chip biases" check-box should be deselected. If the jumpers are not set, the on-chip bias generator has to provide the bias currents and the check-box must therefore be enabled. Also activate the checkbox labeled "streaming".
6. Eventually load the biases `MDC2D_dsPIC.xml` from the directory `biasgenSettings/`.

B Acknowledgments

I would like to thank my advisor Shih-Chii Liu (INI[6]) for her support and ideas during this project, as well as Tobi Delbrück (INI), who answered my questions regarding jAER integration and helped tracking down tricky dynamic library and driver related problems. Also, the support and detailed advice from Rico Möckel (EPFL[7]) during the PCB realization and firmware development were very helpful. Finally, I would like to thank my track

[5]http://www.ftdichip.com/FTDrivers.htm
[6]Institute for Neuroinformatics, http://www.ini.uzh.ch/
[7]Ecole polytechnique fédéral de Lausanne, http://www.epfl.ch

advisor in biomedical engineering, Janos Vörös (ETH[8]), for his approval of this semester project and all the administrative support.

I would also like to acknowledge the free software movement, respectively its innumerable contributors, for creating all the high quality tools without which this project could not have been realized in its current form; a non exhaustive list of programs used for developing the software, testing it and writing the report : Doxygen, GNU Image Manipulation Program, Inkscape, Java, Kile, LaTeX, Python/PyLab, NetBeans, Vim editor.

C Bibliography

References

[1] Markus Bernet. Chapter 5, printed circuit board design. 8

[2] Mandyam V. Srinivasan Dario Floreano, Jean-Christophe Zufferey. *Flying Insects and Robots.* Springer, 2009. 2, 3

[3] Analog Devices. *AD5390/AD5391/AD5392 : 8-/16-Channel, 3 V/5 V, Serial Input, Single-Supply, 12-/14-Bit Voltage Output DACs*, 2004. 9

[4] Martin Egelhaaf and Alexander Borst. Motion computation and visual orientation in flies. *Comparative Biochemistry and Physiology Part A: Physiology*, 104(4):659 – 673, 1993. 2

[5] Tobi Delbrück et al (Institute for Neuroinformatics ETH/University Zürich). jAER - java tools for address event representation (AER). `http://jaer.sourceforge.net`. 14, 15, 19, 31

[6] IEEE. *Analog VLSI Adaptive, Logarithmic, WideDynamic-Range Photoreceptor*, Circuits and Systems, ISCAS, May 1994. 3

[7] Texas Instruments. *TPS793xx : Ultralow-noise, high PSRR, fast RF 200mA low-dropout linear regulators in NanoStar wafer chip scale and SOT23*, 2001-2007. 9

[8] Silicon Labaratories. *C8051F320/1 : Full Speed USB, 16k ISP FLASH MCU Family*, 2003. 8

[9] Shih-Chii Liu. A neuromorphic avlsi model of global motion processing in the fly. *IEEE Transactions on Circuits and Systems*, 47(12):1458–1467, Dec 2000. 3

[10] Beyond Logic. USB in a nutshell. `http://www.beyondlogic.org/usbnutshell/usb1.shtml`, 2011. [Online; accessed March-2011]. 9

[8]Eidgenössische technische Hochschule Zürich, http://www.ethz.ch

[11] Future Technology Devices International Ltd. *FT232R USB UART IC*, 2010. 9

[12] Microchip. *dsPIC33F Family Reference Manual Part 1 : Section 06. Interrupts - dsPIC33F FRM*, 2008. 26

[13] Microchip. *dsPIC33FJ32MC302/304, dsPIC33FJ64MCX02/X04 and dsPIC33FJ128MCX02/X04 Data Sheet*, 2011. 8, 18, 26

[14] M. G. Nagle, M. V. Srinivasan, and D. L. Wilson. Image interpolation technique for measurement of egomotion in 6 degrees of freedom. *J. Opt. Soc. Am. A*, 14(12):3233–3241, Dec 1997. 4

[15] Mandyam V. Srinivasan. An image-interpolation technique for the computation of optic flow and egomotion. *Biological Cybernetics*, 71:401–415, 1994. 10.1007/BF00198917. 2, 4, 5, 20, 30

[16] Mandyam V. Srinivasan, Michael Poteser, and Karl Kral. Motion detection in insect orientation and navigation. *Vision Research*, 39(16):2749 – 2766, 1999. 2

[17] Reto Thürer. Optical flow sensor: USB interfacing & optical flow algorithms. 2011. 2, 4, 8, 10, 20

[18] Wikipedia. Universal asynchronous receiver/transmitter — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter`, 2011. [Online; accessed March-2011]. 9

[19] D Zufferey, JC; Floreano. Fly-inspired visual steering of an ultralight indoor aircraft. *IEEE Transactions on Robotics*, 22(1):137–146, 2006. 2