DEGREE PROJECT, IN SYSTEMS, CONTROL & ROBOTICS , SECOND LEVEL
STOCKHOLM, SWEDEN 2014

Development and Implementation of Star Tracker Electronics

MARCUS LINDH

KTH ROYAL INSTITUTE OF TECHNOLOGY

ELECTRICAL ENGINEERING, SPACE AND PLASMA PHYSICS DEPARTMENT

KTH Electrical Engineering

Development and Implementation of Star Tracker Electronics

MARCUS LINDH, MARCULIN@KTH.SE

Stockholm 2014

Space and Plasma Physics
School of Electrical Engineering
Kungliga Tekniska Högskolan

XR-EE-SPP 2014:001

Development and Implementation of Star Tracker Electronics

Abstract

Star trackers are essential instruments commonly used on satellites. They provide precise measurement of the orientation of a satellite and are part of the attitude control system. For cubesats star trackers need to be small, consume low power and preferably cheap to manufacture. In this thesis work the electronics for a miniature star tracker has been developed. A star detection algorithm has been implemented in hardware logic, tested and verified. A platform for continued work is presented and future improvements of the current implementation are discussed.

Utveckling och implementering av elektronik för en stjärnkamera

Sammanfattning

Stjärnkameror är vanligt förekommande instrument på satelliter. De tillhandahåller information om satellitens orientering med mycket hög precision och är en viktig del i satellitens reglersystem. För kubsatelliter måste dessa vara små, strömsnåla och helst billiga att tillverka. I detta examensarbete har elektroniken för en sådan stjärnkamera utvecklats. En algoritm som detekterar stjärnor har implementerats i hårdvara, testats och verifierats. En hårdvaruplattform som fortsatt arbete kan utgå ifrån har skapats och förslag på förbättringar diskuteras.

keywords

miniature star tracker, cubesat, real-time blob detection, FPGA image processing, CMOS image sensor, smartfusion2 SoC, attitude control, hardware development

Acknowledgements

During frustrating and seemingly impossible problems, my supervisors Nicola Schlatter and Nickolay Ivchenko have been of great support. The possibility to discuss the problems has been invaluable. Special thanks goes to Daria for posting memes cheering up in the lab, Ruslan for unforgettable moments, Francesco for serving us Italian delicacies and home-made ice cream and Niko for just being you. You will all be remembered.

I wish to thank the ones supporting me throughout this time. All of my friends at KTH and my family. Without you, it would never have been possible.

Let the next adventure begin!

Contents

List of Figures

Abbreviations

CCD - Charged-coupled Device
CCTV - Closed-circuit Television
CMOS - Complementary Metal–oxide–semiconductor
EMC - Electromagnetic Compatibility
FIFO - First In First Out
FOV - Field Of View
FPGA - Field Programmable Gate Array
FPS - Frames Per Second
GUI - Graphical User Interface
I^2C - Inter-Integrated Circuit
iLCC - Leadless Chip Carrier
IO - Input Output
KTH - Kungliga Tekniska Högskolan
LASER - Light Amplification by Stimulated Emission
LED - Light Emitting Diode
LISA - Lost In Space Algorithm
LVTTL - Low-voltage Transistor-transistor Logic
MP - Megapixel
PCB - Printed Circuit Board
SEU - Single Event Upset
SOC - System on a Chip
SPI - Serial Peripheral Interface
SPP - Space and Plasma Physics
UART - Universal Asynchronous Receiver/Transmitter
USB - Universal Serial Bus
VHDL - (Very High Speed Integrated Circuit) Hardware Description Language

1 Introduction

Celestial navigation is a technique that has been used for thousand of years. In its simplest form, it uses angular measurements between the horizon and a celestial object. An instrument used for this kind of measurement is often referred to as an sextant, see Figure 1. Technology has made more precise and automated techniques available. A camera can replace the human observer and automated software can search through digital star catalogues trying to find a matching field of view. All of this can be done in a fraction of a second. The core of the principle, usage of celestial objects for navigation, is still the same as several thousand of years ago.

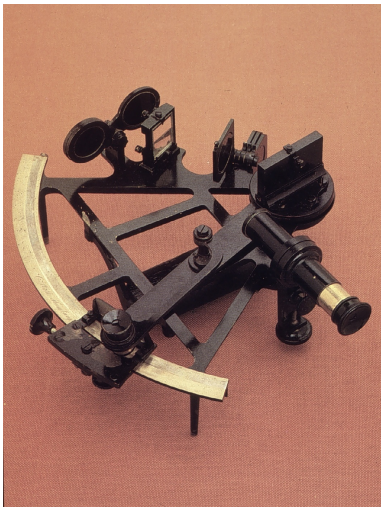


Figure 1: Sextant. An early instrument for celestial navigation. Picture originally taken by U.S. National Oceanic and Atmospheric Administration [2].

It is particularly important for spacecraft to get information about its current orientation. By constantly measuring the deviation from the desired orientation, actuators can apply torque as needed to reorient the object. A system used for keeping the orientation is usually referred to as an attitude control system. The attitude control system often contains a multitude of sensors and feedback inputs. One of the most precious input to this system is the data from a star tracker.

The particular star tracker developed in this project is to be used in the SEAM (Small Explorer for Advanced Missions) nano satellite project. It is a three unit's high (3U) cubesat which can be used for advanced scientific experiments. A one unit (1U) cubesat has the dimensions $10 \times 10 \times 10$ cm giving SEAM the overall dimensions $30 \times 10 \times 10$ cm. The SEAM project is sponsored by FP7, 7^{th} Framework Programme for Research and Technological Development [3]. The rocket containing the actual satellite system is planned to be launched in 2016.

Several commercial star tracker solutions exist but are not easily accessible. They are often expensive and too large for being used in small satellites such as cubesats. By developing a cheaper and smaller substitute more satellites can be equipped with the technology in the future. As satellites are frequently used for various research projects the data from a star tracker are essential. It enables higher precision of the attitude determination and can be combined with advanced measurements. In the long run the solution may become more widespread and help getting higher precision of measurements and attitude determination which can improve research quality.

1.1 Objectives

Most of the available trackers are aimed for large satellites and are as previous mentioned rather expensive. The objective of this project is to develop a hardware solution for a miniature sized star tracker. The tracker should be low cost, consume low power, have a small footprint and high accuracy. A robust platform is to be developed on which further work can be done to finalize the star tracker. Some of the design objectives are given in Table 1.

Table 1: Design objectives.

Acquisition time	<5 s
Update rate	4 Hz
Field of view	$6^\circ \times 6^\circ$
Image sensor	5 MP (2594×1944) CMOS
Power consumption	<1 W

This degree project focuses on the practical hardware implementation and creation of a robust platform on which further work can be done. Most of the focus has been on developing an efficient star detection algorithm and a software for doing various tests. The project of developing the star tracker involves two other students focusing on their respective fields. The identification and tracking algorithm is developed by Francesco Vallegra [4]. Magnitude correction and database compilation are done by Nikola Shterev [5].

One of the biggest questions was if it would even be possible to do the star detection in the hardware proposed. The suggested FPGA is very small and limited. No memories should be used for storing frames from the camera requiring all image processing to be done on the fly.

2 Prestudy

Star trackers have been used in various space applications for several decades. In the beginning they were very basic with limited functionality. As technology now allows it they have become more advanced and with higher performance.

2.1 Different Types of Trackers

"Star sensing and tracking devices can be divided into three major classes: star scanners, which use the spacecraft rotation to provide the searching and sensing function; gimbaled star trackers, which search out and acquire stars using mechanical action; and fixed head star trackers, which have electronic searching and tracking capabilities over a limited field of view." [6]

Nowadays most of the trackers use an image sensor, CCD or CMOS, to capture a picture of the stars and perform the tracking. This degree project is about a fixed head CMOS sensor based tracker with a fixed field of view (FOV). Most of the commercially available trackers are designed to be mounted on medium sized satellites and not many miniature trackers exist. Since the development of technology has come quite far it is now possible to construct a relatively small tracker and still get high precision. In particular, the availability of small sensitive cameras, powerful processors, small flash memories and field programmable gate arrays (FPGA) have made this possible. For comparison, ASTROS is a star tracker from 1985. It weighted 41 kg and had a power consumption of 43 W [7]. Today, equivalent performance can be achieved from units weighting less than 0.5 kg with a power consumption less than 1 W [8].

2.2 Comparision of Commercial Star Trackers

A range of commercially available star trackers have been studied and are compared in Table 2. Generally, the acquisition time, the time after power up to a fixed position acquired, is in the range of a couple of seconds. Refresh rates are 4-10 Hz. Most of the sensors used have a quite low resolution and CCD sensors seem to be most dominating. The Nano Star Tracker is the one most comparable to our star tracker development since it is also aimed for small cubesats.

2.3 Principle of Operation

At initial start up the tracker does not know its position. It is lost in space. When an algorithm has detected all visible stars by the image sensor a lost in space algorithm (LISA) will be performed on the data. The LISA is what finds the orientation of the star tracker by comparing the detected stars with a star catalogue stored in memory. After the orientation is acquired by LISA, the tracking of the stars can be performed. Frame by frame from the camera are compared to find the motion of the stars between the frames. A prediction is made which stars should be visible in the next frame and incorporates them in the tracking. The result of the tracking algorithm is a quaternion describing the orientation of the star tracker. Inputs to the two algorithms are

Table 2: Comparison of some commercial trackers.

Name	**Acquisition time**	**Update rate**	**FOV**	**Sensor resolution**
TERMA HE5AS [6]	3-5 s	4 Hz	$22° \times 22°$	1024x1024 CCD
A-STR (autonomous star tracker) [6], [9]	<6 s	10 Hz	$16.4° \times 16.4°$	512x512 CCD
Nano Star Tracker [8]	2 s	5 Hz	$12.0° \times 9.0°$	?

preprocessed data from the camera and information from a database containing a star catalogue. A conceptual diagram of the LISA and tracking structure is shown in Figure 2.

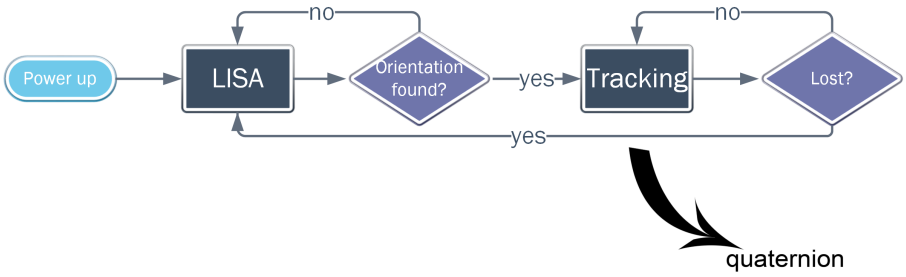


Figure 2: Conceptual diagram of the LISA and tracking part of the star tracker. The output from the tracking is a quaternion.

There are several ways of representing the orientation of an object in 3D space. One of the most common method is by the use of quaternions. Since the attitude control system in the cubesat is using quaternions it is natural to design the output from the star tracker to be in the same format.

The brightness of a star is classified as its magnitude. The lower the number, the brighter the star is. The catalogue is compiled in such a way that it contains only stars of sufficient magnitude to be detected by the sensor. There is a strong relationship between which magnitudes of stars can be seen and the sensors sensitivity, speed of the optics and exposure time.

3 Components of the Star Tracker

The star tracker being developed for SEAM at the Space and Plasma Department (SPP) at KTH can be subdivided in three major parts. The first is the image sensor which captures the image of the stars. The second is the camera FPGA performing the image processing and star detection. The third is the Smartfusion2 SOC for LISA and tracking.

A simplified overview of the star tracker is shown in Figure 3. The actual hardware setup used for implementing and testing is shown in Figure 4. In this section, technical details are presented for each component and subsystem.

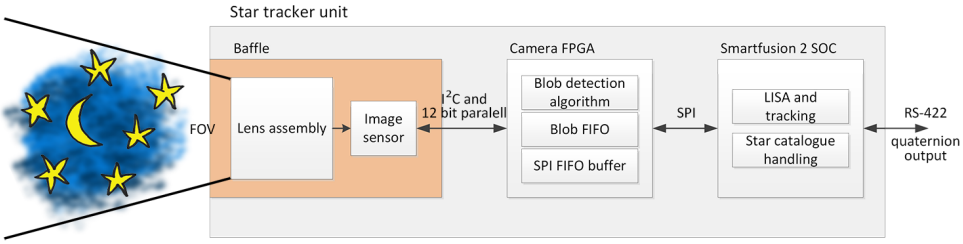


Figure 3: Conceptual overview of the star tracker unit.

3.1 Optical Assembly

The lens used during the development and testing is a standard C-mounted CCTV lens from Fujinon. The specific model of the lens is HF16HA-1B. It has a 16 mm fixed focal length, adjustable focus and an aperture between F1.4 and F16. The field of view is fixed at $17^\circ \times 12^\circ$. A custom made C-mount bracket has been manufactured for use together with the lens and a custom made image sensor PCB. The bracket and the image sensor PCB have been developed at the Space and Plasma Physics Department at KTH. The camera assembly containing the image sensor PCB, C-mount bracket and CCTV lens is shown in Figure 5.

For the flight version of the star tracker a single lens system is proposed to be used. If such a method is used it allows for higher light input to the sensor. A higher light input to the sensor makes it possible to detect dimmer object with the same exposure time. It is being investigated if a single lens system is feasible to use in this application. As a single lens system gives a smaller field of view, it may result in too few stars being detected to be able to run the star identification algorithm.

To prevent stray light from the sun or other bright objects to enter the image sensor,

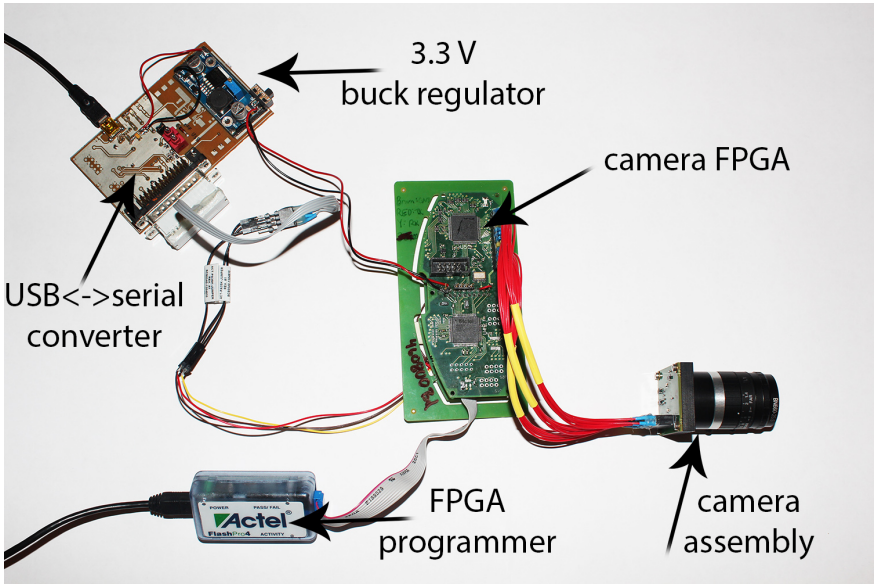


Figure 4: Actual hardware setup. The PCB with the camera FPGA is stacked on top of the Smartfusion2 board.

the optical assembly will be shielded with a black baffle. The baffle will prevent stray light from entering the optical path and is a common practice in such systems.

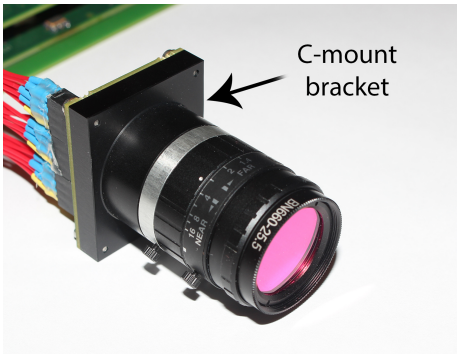


Figure 5: Camera assembly.

3.2 Image Sensor

Aptina manufactures the image sensor used. The specific model chosen is MT9P031 [1], and is a CMOS sensor. It comes in a 48-pin iLCC package and provides 12 parallel data lines for clocking out the pixels. The reason for choosing this sensor is superior low-light performance, low dark current and being simple to interface. The sensor has a native resolution of 2592×1944 pixels (5 MP) but can be lowered if needed by enabling internal binning modes.

Registers define the operation mode and settings of the sensor. These can be accessed and written by the user by means of I^2C communication. The registers are here called *REGXX*, where *XX* is the number of the register. The settings most relevant to the startracker are exposure time and analogue signal gain. Exposure time is affected by the value of several registers and needs to be calculated.

The exposure time t_{EXP} is calculated [1, p. 31] using

$$t_{EXP} = SW \times t_{ROW} - SO \times 2 \times t_{PIXCLK}, \tag{1}$$

where SW is shutter width, t_{ROW} is the row time, SO is shutter overhead and t_{PIXCLK} is the period $\frac{1}{f_{PIXCLK}}$ of the pixel clock. f_{PIXCLK} is the frequency of the clock signal to the sensor.

Shutter width SW is calculated as

$$SW = \max(1, (2^{16} \times Shutter\_Width\_Upper) + Shutter\_Width\_Lower),$$

where $Shutter\_Width\_Upper$ is the value of *REG08*. $Shutter\_Width\_Lower$ is the value of *REG09*. t_{ROW} is the period from the first pixel output in a row to the first pixel output in the next row. t_{ROW} is calculated as

$$t_{ROW} = 2 \times t_{PIXCLK} \times max((\frac{W}{2} + \max(HB, HB_{min})), (41 + 346 \times (Row\_Bin + 1) + 99)).$$

W is the output image width. In this project $W = 2592$. HB is horizontal blanking $HB = REG05 + 1$. Row_Bin is the row-wise binning configuration. HB_{min} is calculated as

$$HB_{min} = 346 \times (Row\_Bin + 1) + 64 + \frac{W_{DC}}{2}.$$

Shutter overhead SO is calculated as

$$SO = 208 \times (Row\_Bin + 1) + 98 + \min(SD, SD_{max}) - 94,$$

where $SD = Shutter\_Delay + 1$. $Shutter\_Delay$ is the value of *REG12*. In our case to obtain the full 5 MP resolution $Row\_Bin = 0$. $SD_{max} = 1232$ if $SW < 3$, $SD_{max} = 1504$ otherwise.

The exposure time is affected by several parameters but can be changed mainly by altering the value of *REG09*. Complete Matlab code for calculating the exposure time

is given in Appendix A.

A clock signal of 32.768 MHz has been used during this project. By using the default register values as listen in Table 3 in Equation 1, one yields the exposure time t_{EXP} = 0.2070 s. Since stars are very dim, a rather long exposure time is required. A frame rate of 4 fps has been used during the development phase. This value is chosen due to the standard exposure time of the sensor and the limiting clock frequency used.

Table 3: Default register values.

REG03	0x0797
REG04	0x0A1F
REG05	0x0000
REG06	0x0019
REG08	0x0000
REG09	0x0797
REG12	0x0000

The image sensor requires steady voltage supplies at 1.8 V and 2.8 V. A PCB, shown in Figure 6, for the image sensor and voltage regulators has been designed at SPP. This makes it possible to run the image sensor PCB from one single 3.3 V power source. The voltage levels of the IO interface of the sensor are standard LVTTL.

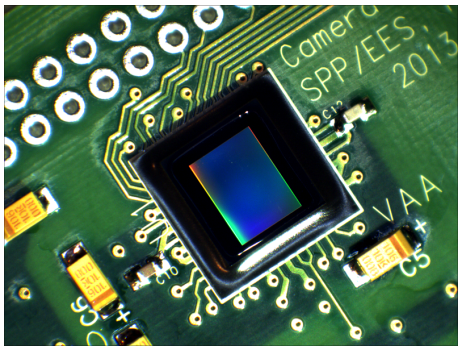


Figure 6: Closeup of the image sensor mounted on a specially made PCB.

From the factory image sensor may be affected by a problem called hot pixels. These are individual pixels stuck at unusual high values. This is a problem generally affecting a very low number of pixels when the image sensor is brand new. This is something to be aware of and it is handled in the star detection algorithm.

3.3 Camera FPGA

The logics for communicating with the camera are implemented in an Actel Proasic 3 FPGA. The specific model used is A3P250 and is shown in Figure 7 soldered on a PCB. The FPGA contains a fabric with 250 K gates. It is widely used in aviation, space and military applications due to its robustness. It is reliable even if exposed to radiation which is one key factor when designing for space applications.

The main task for the camera FPGA is to control and process data from the image sensor. It loads appropriate parameters to the sensor by using the I^2C interface. The image processing and star detection is done entirely on the camera FPGA in near real-time as the data are clocked out from the image sensor. The outputted data from the camera FPGA are a message containing X, Y and I for each detected star in the field of view. Details follow in section 4.

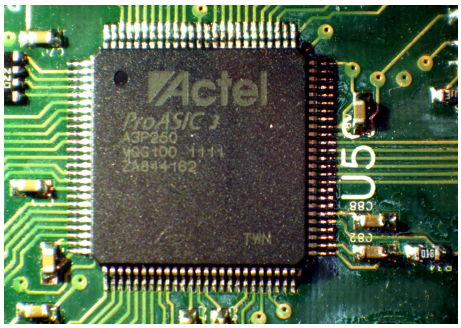


Figure 7: Actel A3P250 FPGA used for image processing.

3.4 Smartfusion2 SoC

The Smartfusion2 SOC is a system on a chip containing both an FPGA fabric and an ARM Cortex M3 processor. It communicates with the camera FPGA and will perform the star identification algorithm (LISA) and tracking. Figure 8 shows the Smartfusion2 mounted on a PCB.

There are a couple of manufacturers (Xilinx, Altera, Lattice etc.) providing system on a chip solutions. They have however a very high Single Event Upset (SEU) failure rate at high altitudes caused by radiation. The Smartfusion2 is designed with this in mind and has zero failures in the event of SEU [10]. This makes it a good choice for space applications.

The Smartfusion2 SOC is intended to perform the star identification and tracking algorithms. These are however not fully developed yet and not implemented. A simple software for the Smartfusion2 to access star data from the camera FPGA through SPI and relaying it on the UART connection have been implemented.

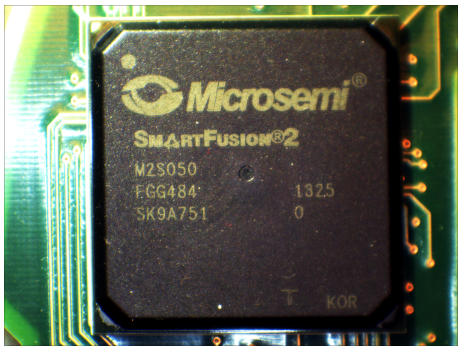


Figure 8: M2S050 Smartfusion2 SOC used for LISA and tracking.

3.5 Communication

To have a reliable data transfer between the star tracker unit and other satellite systems a RS-422 interface is suitable. Since RS-422 is using balanced data transmission, it can reliably be used even in harsh environments with EMC noise. For flexibility, both balanced and unbalanced connection should be available for interfacing. The unbalanced connection could easily be connected to a serial port on a computer for testing and debugging.

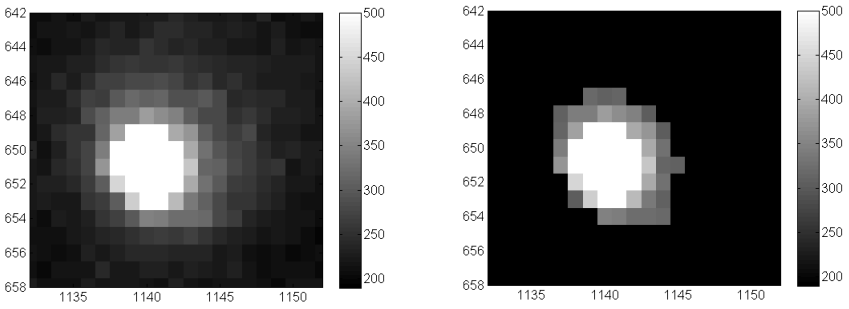
3.6 Housing

The housing should be constructed of a material shielding the electronics from radiation and still be lightweight. It should secure all electronics component in a robust fashion. The optical assembly should preferably be hermetically sealed to prevent particles entering the optical path.

4 Camera FPGA and Image Processing

When a star is in the field of view of the star tracker it is projected on the image sensor through the optics. The star will span a certain number of pixels depending on the optics. Throughout this section the word blob is frequently used. The term blob here refers to a collection of pixels with a value higher than a specified threshold level. Those pixels represent a star as seen by the sensor. To express a single row of pixels in the blob the word rowblob is used.

Figure 9a shows an close-up view of a star as seen by the sensor. After applying a threshold filter the image of the resulting blob is obtained as seen in Figure 9b. The blob detection algorithm uses a threshold value to separate the stars from the background.



(a) Original image.

(b) Thresholded image.

Figure 9: Close-up view of a star before and after applying a threshold filter. The right image shows the resulting blob after applied threshold. In this example the threshold value was set to $th = 305$. The star is successfully separated from the background.

4.1 Realtime Centroid Calculation

Blob detection and centroid calculations are key challenges for the functionality of a star tracker. There are several ways of detecting objects in an image. Some are more advanced than others. In the case of the star tracker, it is the simplest possible scenario. The source image has a black background with white spots (stars) on it. The extracted spots by applying a threshold filter are what we call blobs. A centroid calculation is needed to get one single value of the location of a blob. This location is the one closest to the real location of the star on the image plane. The centroid calculation uses the intensity of the pixels within the blob to calculate the center of intensity by means of weighted sums. In this way sub-pixel accuracy can be obtained. In the algorithm presented in section 4.2 sub-pixel accuracy is not implemented. The algorithm is however easy to modify to obtain sub-pixel precision. Due to the relatively high native resolution of the image sensor sub-pixel accuracy is probably not needed.

The image from the image sensor is a 12 bit 5 MP grayscale image. The interesting

spots vary in size and intensity. The first step is to separate the background from the rest of the image keeping only the spots. This is where the threshold value is used. If a pixel is detected to have a value higher than the threshold, it is handled as potentially being part of a blob. Objects having an intensity lower than the threshold will not be possible to detect. How the threshold value is chosen is explained in Section 4.2.4.

Many of the existing solutions of blob detection need full access to a whole image from the camera. For this project such a solution is not desirable. It would have needed an external memory to temporarily save the picture for later processing. The use of external memory would result in larger physical design, higher power consumption and longer processing times.

The blob detection method considered in this thesis uses data directly from the image sensor as it is clocked out. This results in an extremely low latency. The solution uses a rotating FIFO buffer to accomplish the task and has according to research not been used in star trackers before. The method has similarities with sliding window methods but is more dynamic. Since the algorithm is very efficient and runs on the fly in near real-time it could certainly be useful in high speed applications. Some examples of this could be camera based touch screens and infrared reflective eye tracking.

Because of the complexity of the algorithm, it would have been hard and time-consuming to directly implement the algorithm in the FPGA. To simplify the developing phase the algorithm was first implemented and simulated in C#. The coding style was done in a way to fake a hardware implementation. The algorithm was tested and debugged until it behaved satisfactory. The software developed for simulating the blob detection algorithm grew larger than initially planned. Many other features were added to simplify the developing and testing. Because of the many features implemented in the software section 5.4 is dedicated for explaining them further.

The image sensor works in such a way that it clocks out pixel by pixel, row by row. Figure 10 schematically shows how a scene is projected on the image sensor. A result of the imaging is that the scene is mirrored on the sensor. After exposure the pixels are clocked out starting from the bottom right corner on the sensor corresponding to the upper left corner of the scene.

The algorithm computes the coordinates and intensity for each detected blob in the frame. The output from the algorithm is a set of values, X, Y and I, for each blob. X corresponds to the horizontal location of the blob and has a value between 0 and 2592. Y corresponds to the vertical location of the blob and has a value between 0 and 1944. I is the intensity of the whole blob and have a value between $2 \cdot th$ and 2^{24}, where th is the threshold level. It is calculated as the sum of all pixels with a value higher than the threshold level contained in the blob. Since each blob contains at least two pixels the minimum possible value of I is $2 \cdot th$.

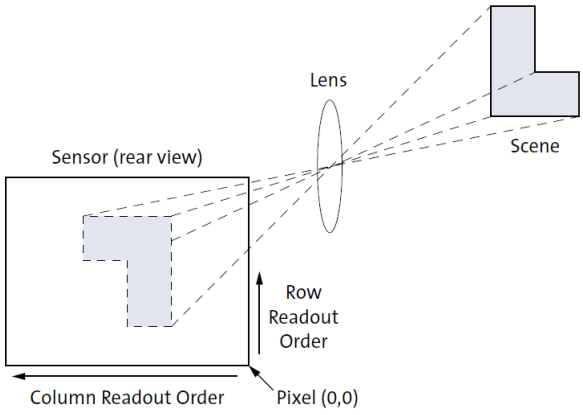


Figure 10: Projection of the scene on the image sensor [1, p. 11].

The algorithm presented in the next section is mainly designed for circular blobs. However due to its dynamic detection it is useful even for slightly deformed blobs. This happens if the exposure time of the camera is too long resulting in smeared stars.

4.2 Realtime Centroid Calculation Algorithm

The blob detection and centroid calculation algorithm is divided in four layers for easier overview. The bottom layer is in direct contact with the raw pixel data from the image sensor. The top layer is more abstract and is the layer which computes the final values of X, Y and I for each blob. The layering principle is visualized in Figure 11. The figures presented in this section explaining the algorithm do not contain all details. Flags and signals between the layers are omitted for simplicity reasons. The complete VHDL code for the algorithm is given in Appendix C.

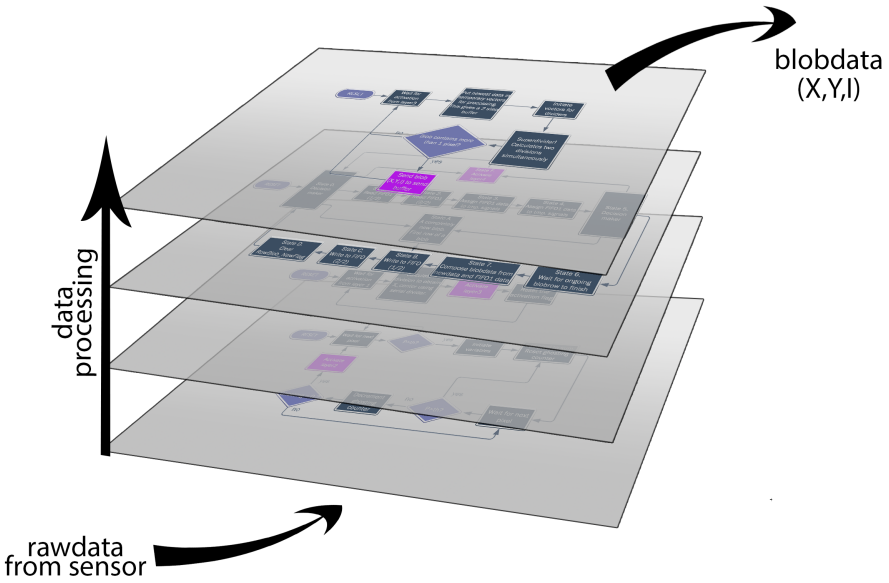


Figure 11: Visualization of the blob detection algorithm as four layers. The first layer (bottom) has direct contact with the raw data from the sensor. The fourth and topmost layer is the most abstract layer outputting processed blobdata.

Following is the principle of the blob detection algorithm presented. The four layers of the algorithm all run in parallel and are synchronized with flags.

Assume a stream of pixel values entering the detection algorithm. The first criterion needed to trigger the detection is a threshold override. A pixel P with intensity higher than the threshold level th triggers this event. When a threshold override is detected, the algorithm goes to the next step to create two important sums. The horizontal position on the row on which the event was triggered is saved as X_{start}. The first sum is $\sum(n \cdot P)$ where n is the current number of pixels in the ongoing rowblob. n increases with the number of pixels detected for the rowblob. This is a weighted sum of intensities by their location on the row. The next sum $I_{row} = \sum P$ is the integrated intensity of the ongoing rowblob. The two sums are the numerator and denominator in an equation used to calculate which column in the rowblob has the center of intensity. The center column by intensity is called X_{center} and is calculated as

$$X_{center} = \frac{\sum_{n=1}^{w}(n \cdot P)}{\sum_{n=1}^{w} P} + X_{start}, \tag{2}$$

where w is the width of the rowblob.

Figure 12 shows fundamental parts of the detection algorithm. When the next pixel in the stream entering the algorithm, is detected to have an intensity lower than the threshold, a ghosting phase is initiated. The ghosting phase gives a distance (ghosting distance) in which pixels can be detected and still be grouped with the ongoing rowblob. If the next pixel overriding the threshold value appears outside the ghosting distance, it will be handled as a completely new blob. The ghosting phase is implemented to improve the functionality of the algorithm and help in the event of broken pixels.

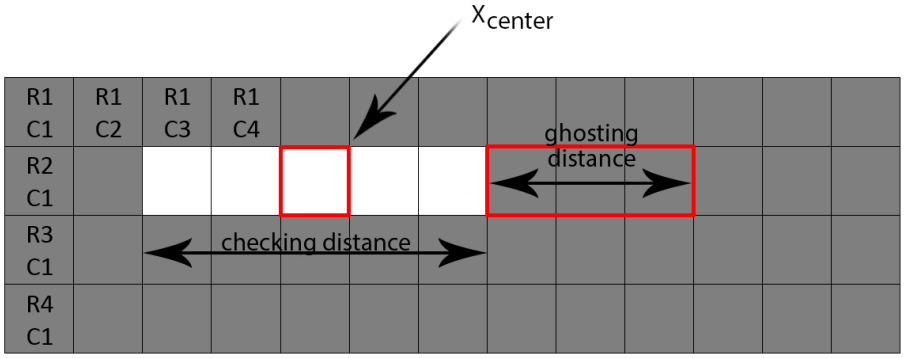


Figure 12: Ghosting and checking distance. X_{center} is the center of intensity column in the detected blob on row 2. In this example located on row 2, column 5 (2,5). The ghosting level is here set to 3 giving a ghosting distance of 3 pixels.

The ghosting level is set with parameter *Ghosting_Level*. The value of the parameter gives an equal ghosting distance. If no pixel overrides the threshold in the ghosting distance, the calculations for the rowblow can be taken to the next step and finalized. Equation 2 is now calculated using the completed sums in the numerator and denominator. The detection of threshold override and sum creation are implemented in layer 1 as pictured in Figure 13. The calculation of Equation 2 is done in layer 2 as pictured in Figure 14.

The data for each blob are saved in a FIFO buffer as 2×62 bits. 128 bits in total for each blob. Some of the bits are however not used and reserved for future improvements. When a new rowblob is discovered it is always compared with the first element in the FIFO buffer. The checking distance is the area in which a newly detected rowblob needs to have pixels located to be merged with the blob in FIFO. The checking distance is shown in Figure 12. If this occurs data are appended to the blob in the FIFO and put in the FIFO's last position. The FIFO rotates in such a way that the first entry is always the one to possible append data to. If no new pixels are discovered in the checking distance to the blob in the FIFO the blob is ready to be released. A flag triggers the fourth layer which makes the final processing on the blob. The logics for

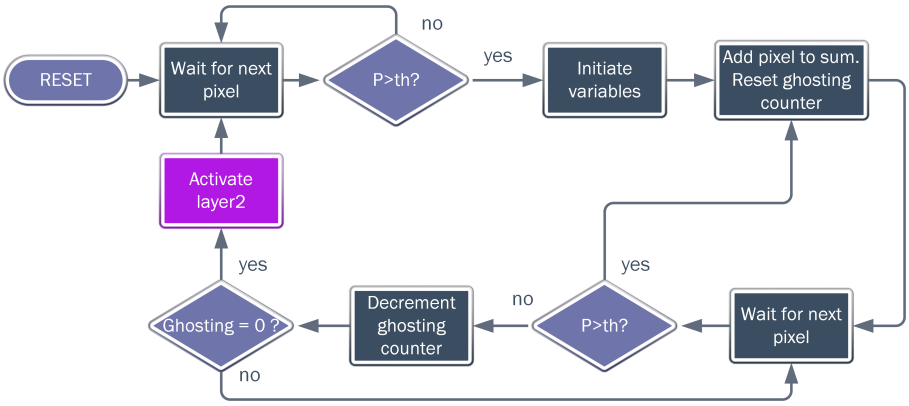


Figure 13: Layer 1 (bottom layer). This layer is in direct contact with the pixels clocked out from the sensor. P is the value of the current pixel. The threshold level is denoted th.

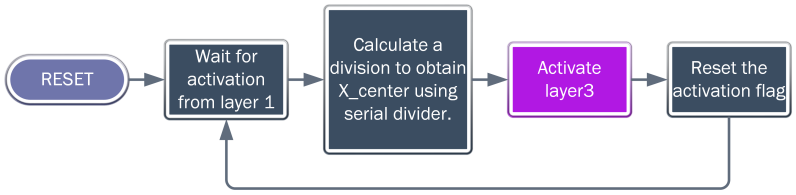


Figure 14: Layer 2.

FIFO rotation and merging of blobs is implemented in layer 3 as shown in Figure 15. The fourth layer with the final processing is shown in Figure 16.

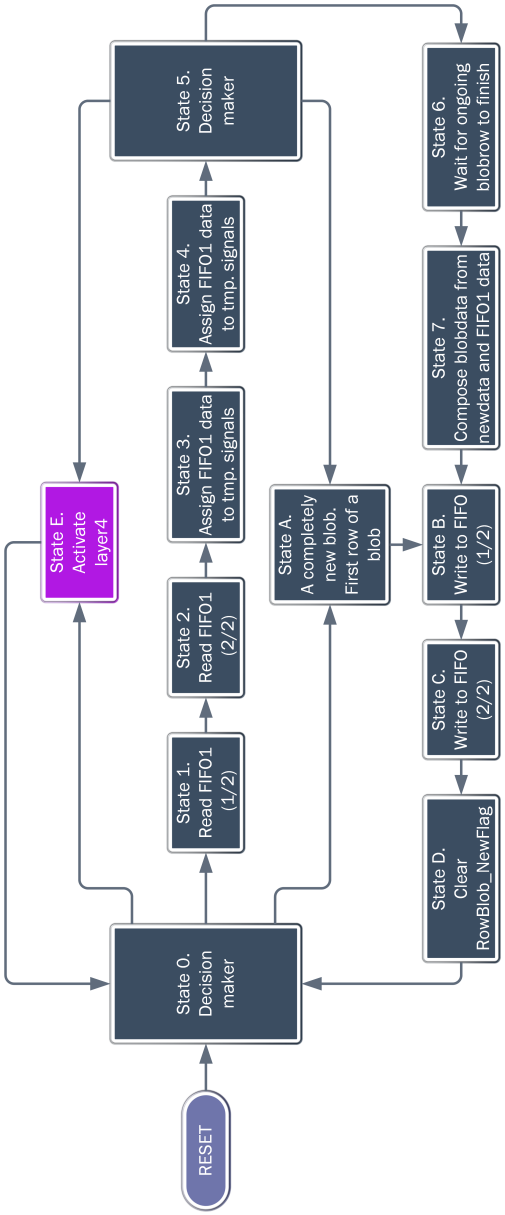


Figure 15: Layer 3. Logics for FIFO rotation and blob merging.

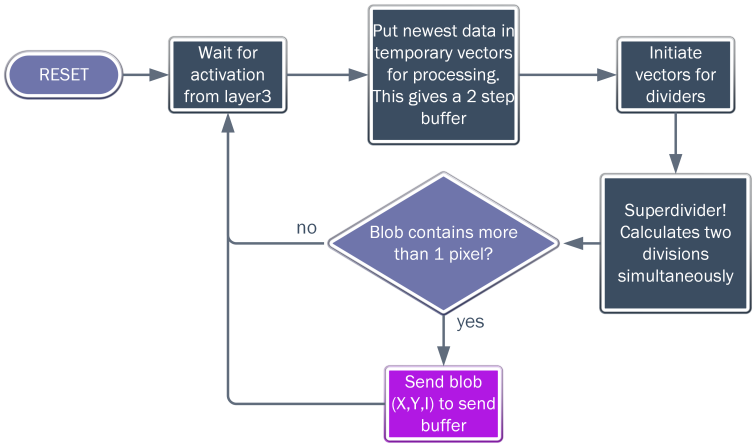


Figure 16: Layer 4.

How the FIFO packages are constructed and what data they contains are shown in Table 4 and Table 5.

Table 4: FIFO data. Package 1 of 2.

Value	$\sum X_{center}$	X_{center}	Startrow	Rowspan	PX count
Number of bits	24	12	12	6	10
Bits in package	63:40	39:28	27:16	15:10	9:0

Table 5: FIFO data. Package 2 of 2.

Value	$\sum_{n=1}^{h} (I_{row} \cdot n)$	$\sum_{n=1}^{h} I_{row}$	unused
Number of bits	24	24	16
Bits in package	63:40	39:16	15:0

A final check is done on the blob before it is released to the SPI send buffer. It must contain at least two pixels. To prevent the send buffer to overflow in the event of too many detected blobs a limiter is set. The limiter is called $MaxStarsInFrame$ and specifies how many blobs are allowed to be detected in one frame. The limiter in the current implementation is set to 25. For comparison the commercial star tracker A-STR has a maximum tracking capability capability of 10 stars [9, p. 6].

When a blob finally is ready to be released X_{blob}, Y_{blob} and I_{blob} are calculated.
X_{blob} is calculated as

$$X_{blob} = \frac{\sum X_{center}}{Rowspan},$$

where $Rowspan$ is the number of rows in the blob. This is an ordinary mean value.

The center of intensity in Y-direction is calculated using weighted sums similar to Equation 2. The center position in Y-direction is called Y_{blob} and is calculated as

$$Y_{blob} = \frac{\sum(I_{row} \cdot Y)}{\sum I_{row}} + Y_{start}, \tag{3}$$

where I_{row} is the integrated intensity of each individual row. Y is the current row in the blob. Y_{start} is the vertical starting position of the blob on the image sensor.

The intensity I_{blob} for the whole blob is calculated as the sum of intensities of all rows in the blob

$$I_{blob} = \sum I_{row}.$$

All calculations containing divisions are implemented in hardware VHDL using serial dividers. The implementation of the divider is explained further in Section 4.2.3.

4.2.1 Row by Row Explanation of FIFO Events

How the blobs are discovered row by row is shown in Figure 17. What happens for each row and the current state of the FIFO buffer are as follow:

(Row 1)
i. Blob 1 is discovered and enqueued. →[1]→

(Row 2)
i. Data are added to blob 1. Dequeued and enqueued. →[1]→
ii. Blob 2 is discovered and enqueued. →[2,1]→

(Row 3)
i. Blob 3 is discovered and enqueued. →[3,2,1]→
ii. Data are added to blob 1. →[1,3,2]→
iii. Data are added to blob 2. →[2,1,3]→

(Row 4)
i. Data are added to blob 3. →[3,2,1]→
ii. Blob 1 is released. →[3,2]→
iii. Data are added to blob 2. →[2,3]→

(Row 5)
i. Data are added to blob 3. →[3,2]→
ii. Blob 4 is discovered. →[4,3,2]→
iii. Blob 2 is released. →[4,3]→

(Row 6)
i. Blob 3 is released. →[4]→
ii. Data are added to blob 4. →[4]→

(Row 7)
i. Data are added to blob 4. →[4]→
ii. Blob 5 is discovered. →[5,4]→

(Row 8)
i. Blob 4 is released. →[5]→
ii. Data are added to blob 5. →[5]→

(Row 9)
i. Data are added to blob 5. →[5]→

(Row 10) After the last row blob 5 is still in the FIFO. The reason for this is that it was on the bottom border and could therefore not be released. The whole FIFO is now cleared before the start of next frame.

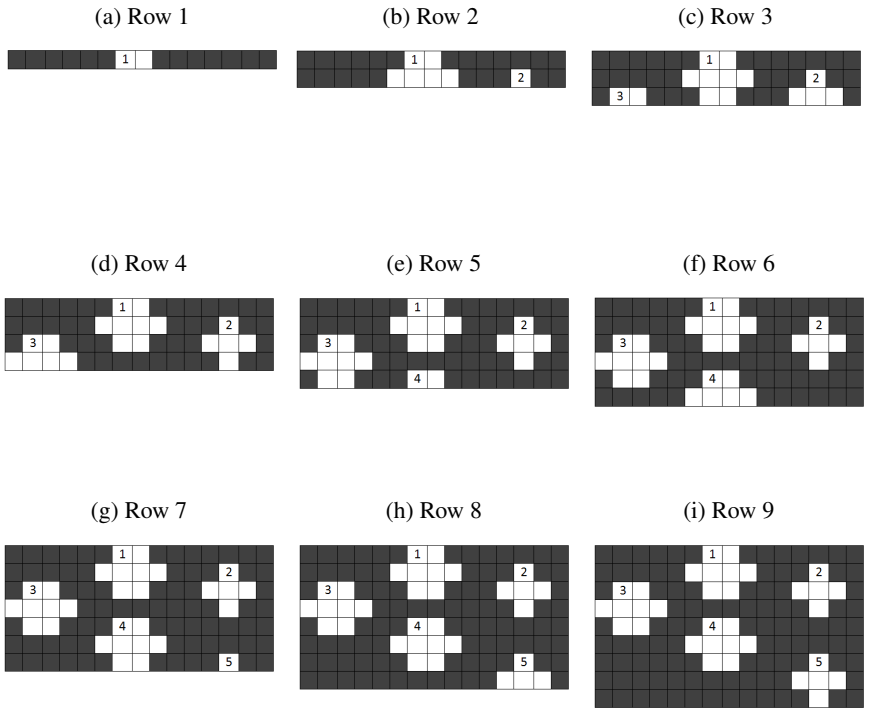


Figure 17: The image from the sensor is read out rowwise, left to right. Figure a-i shows how a picture with five stars is detected row by row.

4.2.2 Design Constraints

By implementing an algorithm in hardware, physical design constraints will be present. Vectors of fixed lengths are used for storing values and only integers can be used without doing ”fancy tricks” using floats, which require more space. A rectangular blob with height h, width w and all pixels with the same intensity P is considered to investigate how wide signal vectors are needed. Let k be the bit-width of the signal vector.

The relationship that must be fulfilled is

$$\sum_{n=1}^{h}(P \cdot w) \cdot n = 2^k. \tag{4}$$

This constraint originates from the numerator in Equation 3. This part can grow very large and is the main limitation. For simplicity the blob is assumed to be square shaped with sides $h = w = x$. Equation 4 can be rewritten as

$$\sum_{n=1}^{x}(P \cdot x) \cdot n = 2^k.$$

By rewriting the sum one get the equation

$$P \cdot (x^3 + x^2) = 2^{k+1}. \tag{5}$$

Solving for the real-roots of x in Equation 5 for some chosen values of P and k one obtains the maximum blob dimensions as in Table 6. Since each pixel P is 12 bit it can have a value between 0 and 4095.

Table 6: Vector Constraints.

P	k	x
2048	24	25.069
4095	24	19.832
2048	32	160.94
4095	32	127.68

In Table 6 it shows that increasing the vector width from 24 bits to 32 allows for significantly larger values of x. It seems beneficial to use a 32 bit vector since it can handle very large square blobs. It can thus handle even larger circular blobs. A 32 bit vector however require more space in the FPGA fabric and results in slower division. A slower divider may introduce other problems such as timing issues. A vector width of 24 bits is therefore chosen and used for all divisions in the algorithm. This is reasonable since a star usually spans over just a couple of pixels and should generally not cause any troubles. A vector width of 24 bits allow blobs containing 25×25 pixels with $P = 2048$ to be detected without overflowing.

4.2.3 Digital Division in the Algorithm

The blob detection algorithm relies on a set of sums and divisions to be computed correctly. Sums are easy to compute in hardware but divisions generally cause trouble. One special case is division with 2^n which is particularly easy as it is only a right shifting of bits n steps. For the blob detection algorithm a complete division had to be implemented.

In hardware logic, divisions can be implemented in two ways. The first type is a pipelined implementation which solves the division in only one clock cycle. This can be beneficial with smaller signal vectors in the division. In this case the division is done with two 24 bit vectors and the hardware logics therefore becomes huge. It occupies near 100 % of the FPGA core which is not feasible. Instead a serial divider is implemented. The downside is that a serial divider becomes slower and requires 24 clock cycles to compute. The VHDL code for the serial divider is attached in Appendix B.

4.2.4 Threshold Value for the Algorithm

The background of the image is not perfectly flat. Depending on conditions, it may vary and does not have a constant value. One way of determining this value is by the use of histogram. Since the majority of the picture is black background the most frequently occurring value in the histogram can be said to be the background level. The threshold value for the blob detection algorithm has to be set slightly higher than the background level. For this application it should give satisfactory results to analyse histograms once for some sample pictures of stars and obtain the suitable value of the threshold level. A more dynamic way of doing it would be to calculate a histogram in the camera FPGA on the fly and adjust the threshold value automatically giving a dynamic threshold.

The values of the pixels vary between frames due to noise. When the threshold value is picked this has to be kept in mind. One has to make sure that the noise level from the background wont disturb the blob detection algorithm and give false threshold overrides. In the current implementation, the threshold value has been set experimentally for indoor tests using simulated stars. For the final version of the tracker, the threshold level needs to be tested carefully together with the optics and different exposure times to get the best possible performance.

4.2.5 Timing Issues

Since the blob detection algorithm uses a serial divider it can cause timing issues between the different layers. If a new blob is detected before the division is complete, one blobrow may be skipped. This can be resolved by increasing the ghosting value. The value should be increased so that a new blobrow is not released until the previous division is complete. This results however in a constraint limiting how close two stars can be to each other and still be separable. The ghosting level is the limiting factor how close to each other stars can be detected. The checking distance is another pa-

rameter that should be set with the ghosting level in mind. A rule of thumb is to keep $checking\_distance \leq ghosting\_level$.

4.3 Camera FPGA Firmware

There are essentially two firmwares developed for camera FPGA in the star tracker. The basic functionality of the camera and I^2C communication in both of them originates from the ISAAC Experiment [11]. The main firmware contains the blob detection algorithm and SPI communication. The second is only designed for testing the image sensor.

4.3.1 Firmware 1, Blob Detection

This is the main firmware in which the blob detection algorithm is implemented. An SPI interface is implemented to communicate between the Smartfusion2 and the camera FPGA. Since this particular model of FPGA is small with only 250 K grids most of it becomes occupied with the blob detection algorithm. In the current implementation 87 % the FPGA core is used. The camera is set to run at 4 fps.

The blockdiagram for the FPGA is drawn in Viewdraw by Microsemi. Viewdraw is part of Libero IDE which is the development tool used for the FPGA. The Viewdraw schematic concerning the blob detection algorithm is shown in Figure 18. It shows how the block containing the algorithm is connected with the FIFO buffer and SPI send buffer.

4.3.2 Firmware 2, Camera Testing

The camera test firmware allows saving of images to flash memory. The images will however not have the full 5 MP resolution. The pixels are binned 4×4 giving an image resolution of 648×486 pixels. The firmware allows images to be saved continuously at 2 fps giving a sequence of images.

The camera testing firmware communicates with a PC using UART and a serial terminal. For reading the images saved in the memory a terminal with logging capabilities such as Teraterm [12] is needed. The resulting logfile after a readout session can then be opened in the multi-purpose software. If the logfile contains a sequence of images it can then be played as a movie in the software. All communication is done using UART and by sending keystrokes corresponding to different commands. The commands for communicating are listed in Table 7.

4.4 SPI Communication with Camera FPGA

The communication with the camera FPGA is done with a SPI bus. The camera FPGA is a SPI slave and the Smartfusion2 is the SPI master. SPI communication is always initiated by the master. The protocol implemented has eight message codes specifying

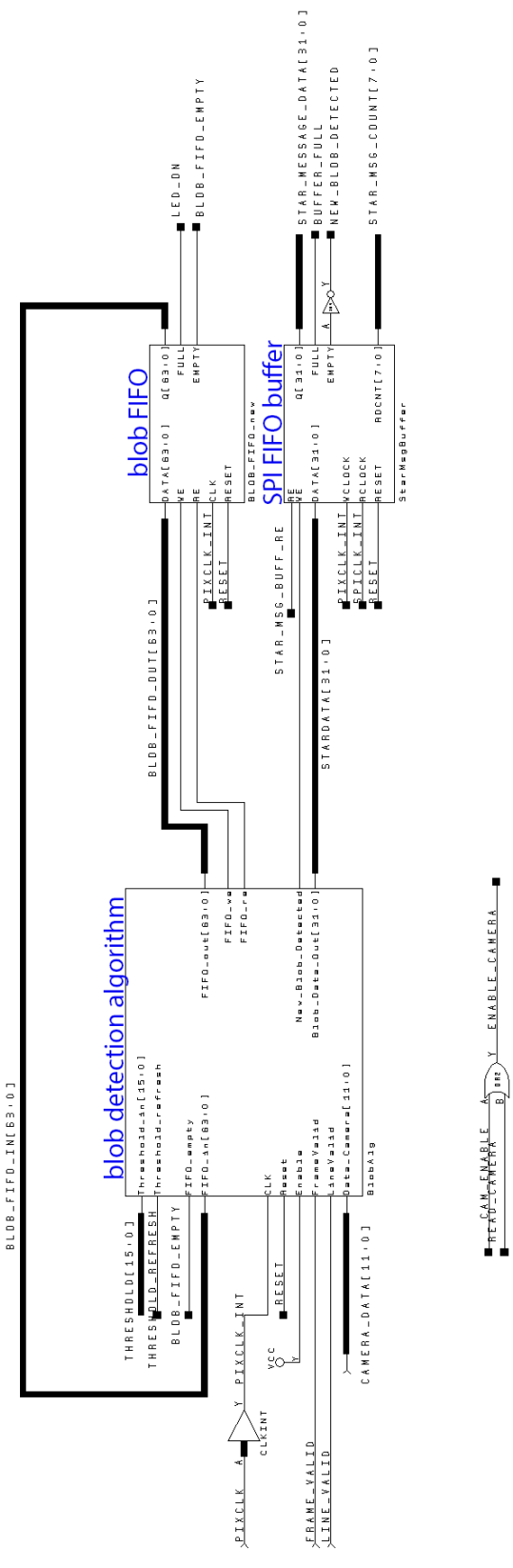


Figure 18: Viewdraw schematic.

Table 7: Commands in the Camera Test Firmware

Command	**Keypress**
Toggle memory readout on/off	a
Toggle image saving on/off	y
Rewind memory	R
Erase memory	Q
Print current memory position	F

what kind of package is currently being transmitted or received. The SPI packet size is set to a fixed length of 32 bits. The first 8 bits is the command sent to the slave, and the remaining 24 bits are the actual data. In this implementation only the first 4 bits of the command is processed. The leftmost bit of these four is not to be used since it can have an unknown state. This gives three usable bits for command codes resulting in a total of eight possible commands. The 24 bit data can be sent either to the slave or be read from the slave depending on the command used. A list of all SPI commands for communication with the camera FPGA is shown in Table 8.

Table 8: SPI commands

	Code (Bits 0-7)	**Payload** (Bits 8-31)	**Response** (Bits 8-31)
Request status	0x00	-	Status message
Request star data (Part 1 of 3)	0x10	-	Star data, 1 of 3
Request star data (Part 2 of 3)	0x20	-	Star data, 2 of 3
Request star data (Part 3 of 3)	0x30	-	Star data, 3 of 3
(not implemented)	0x40	-	-
(not implemented)	0x50	-	-
(not implemented)	0x60	-	-
(not implemented)	0x70	-	-

The SPI master always checks the status message (code 0x00) if any new data are available in the slave. The structure of the status message is shown in Table 9. If bit 8 (Buffer not empty flag) is 1, new data are available. One should then read part 1, part 2 and part 3 sequentially to receive all information. The status message is checked and the read action is repeated until no more data are available, eg reading data until *Buffer_not_empty_flag* becomes zero. The data contained in the three parts corresponds to data for one blob and can be either real star data or a new frame message. The structure of the three SPI packages are shown in Table 10, Table 11, and Table 12 respectively.

As seen in Table 10, the first part contains a bit called NF indicator. This is the bit specifying if the package is an actual star package or a new frame indicator. 1 = new frame. 0 = stardata.

The data from the camera FPGA needs to be read as soon as possible after it has become available. This is done by repeatedly polling the status message shacking if any new data are available. If the Smartfusion2 for any reason is unable to keep up the pace of reading the star data an overflow may occur in the camera FPGA buffer resulting in lost stars. Another reason to read the star buffer as quick as possible is to get as low delay as possible between the data acquisition and processing.

Table 9: Status message

Bits	23:12	11:9	8	7:0
Contains	0xAAA	0b000	Buffer_not_empty_flag	Stars_in_buffer

Table 10: Part 1 of star message.

Bits	23	22:16	15:12	11:0
Contains	NF indicator	Spare	0b0000	X-position

Table 11: Part 2 of star message.

Bits	23:20	19:12	11:0
Contains	0b0000	Pixel count	Y-position

Table 12: Part 3 of star message.

Bits	23:0
Contains	Intensity I

5 Testing

The blob detection algorithm has been tested extensively indoors. To be able to test the camera in daylight a red-pass filter from Edmund Optics is mounted on the camera. The filter allows only red light at a narrow wavelength to pass through and enter the image sensor. Small red light sources have been used to simulate stars indoor.

5.1 Simulated Stars

To simulate stars in the lab a pinpoint light source has been built. This has been of good use during the testing of the system, specifically the blob detection and centroid calculation. The light source consists of a plastic box containing a rechargeable Li-ion battery, dip-switches, a potentiometer to vary the intensity and a set of light sources. The charging of the internal battery is done with a USB connector. The light sources are red 1206 SMD LEDs soldered on a set of flat cables. The LEDs have inbuilt lenses to get a narrow light output. A fine fiber optic strand is also included in the box. The fibre is driven from a low power red laser giving a very good pinpointed (Ø1 mm) monochromatic light. The fiber is painted black to prevent light being emitted elsewhere than from the edge. The dip-switches select which light sources to be lit.

The constructed device is shown in Figure 19. The fibre optic strand giving a Ø1 mm light source is shown in Figure 20. The USB jack for charging the battery is shown in Figure 21. An indicator indicates when the battery is fully charged.

Figure 19: Rechargeable light source with adjustable intensity to simulate stars.

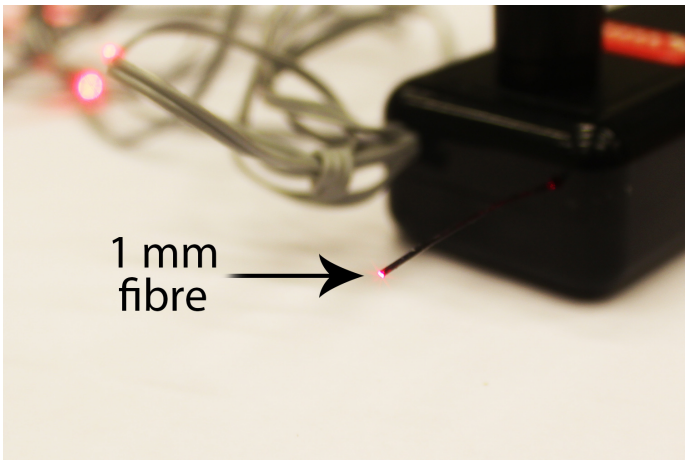


Figure 20: Fiber optic strand driven from a red laser diode.

5.2 More Complicated Star Patterns

To simulate actual star signs a video projector can be used to project stars modified to be realistic as seen from the star tracker. An example of software capably of projecting stars as seen with a specific field of view is the free and open source software Stellarium [13].

5.3 Star Photography

Several tests have been performed with the image sensor and the Fujinon CCTV lens outdoors a couple of dark nights in February and April. A set of star pictures with different settings have been taken with a development kit for the image sensor. This was done in order to have material to analyse to be able to characterise the behaviour of the lens and the image sensor. The result of the data are discussed further in the companion master thesis [5].

The blob detection algorithm was also tested in a real scenario during the photo sessions. The tests were however not successful. The image sensor had the default settings giving a too short exposure time. As explained in Section 3.2 the default exposure time is $t_{EXP} = 0.2070$ s. With the Fujinon lens set to F1.4 and the default settings of the image sensor, no stars could be detected. By increasing the signal gain and exposure time the blob detection algorithm would probably had worked. The exposure time should however be kept below 0.25 s to allow 4 fps from the image sensor. A single lens system with higher light input will also help in the detection of stars. Unfortunately, the settings could not be changed out in the field and no more photo sessions were made.

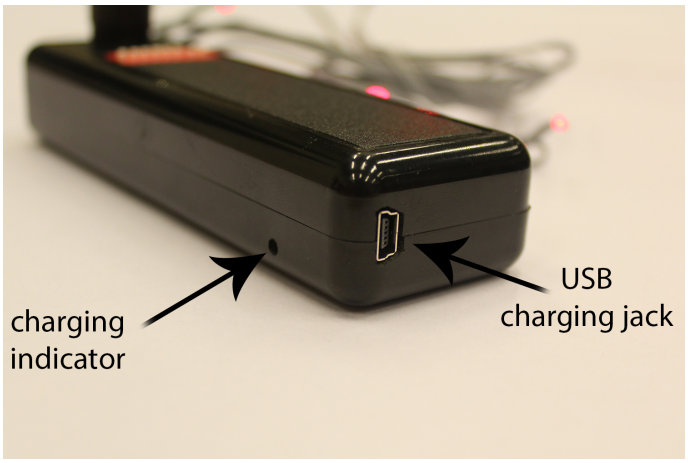


Figure 21: USB charging jack and charging indicator.

5.4 Multi-purpose Software

An application has been coded in C# to help the development of the star tracker. It allows the user to perform the blob detection algorithm in software on a loaded image and several other functions. It can read common image formats such as ordinary bmp, jpg and png files. Raw images saved by Aptina Sensor Demo Software [14] can also be loaded. The software have got the nickname ”The Blobber”.

Figure 22 shows the graphical user interface (GUI) and the arrowed controls are:

a: Menu item for loading log files saved by the camera test firmware.

b: Settings for the serial port. Open and close the port.

c: Controls for real-time star data. The received data are displayed in the gray box. Indicators for frame number and framerates are shown. If checkbox *Save logfile* is ticked the received realtime data will be recorded. When the checkbox is unticked the data will be saved on the harddrive in the directory logs/. Filename is created automatically containing the date and time. If *Trails* is ticked fading trails are shown between the detected real-time blobs.

d: Images located in the folder bilder/ in the software folder. Select which image to work with.

e: Settings for the blob detection algorithm. Threshold value is set with the slider or by typing in the text box right to it. The radius of the markers is adjusted with the second slider. Checkboxes selects if single pixel blobs should be made visible and if numbers should be shown next to the markers.

f: Button to run the blob detection algorithm.

g: Result after the blob detection process. All detected blobs and corresponding data such as width W, number of pixels PX, horizontal position X, vertical position Y and intensity I are visible. By scrolling down a matrix called Francesco containing X, Y and I for each detected blob is visible. The text can be copied and used in other locations.

h: If a log file containing a sequence of images have been loaded the *Play* button can be clicked to play the sequence. Frame indicator shows the current frame being displayed. Min-max indicator shows the minimum and maximum values detected for all pixels in the sequence.

i: *Load image* button loads the selected image and displays it in j.

j: Panel showing the original image or log file loaded.

k: Panel showing the thresholded image and the detected blobs.

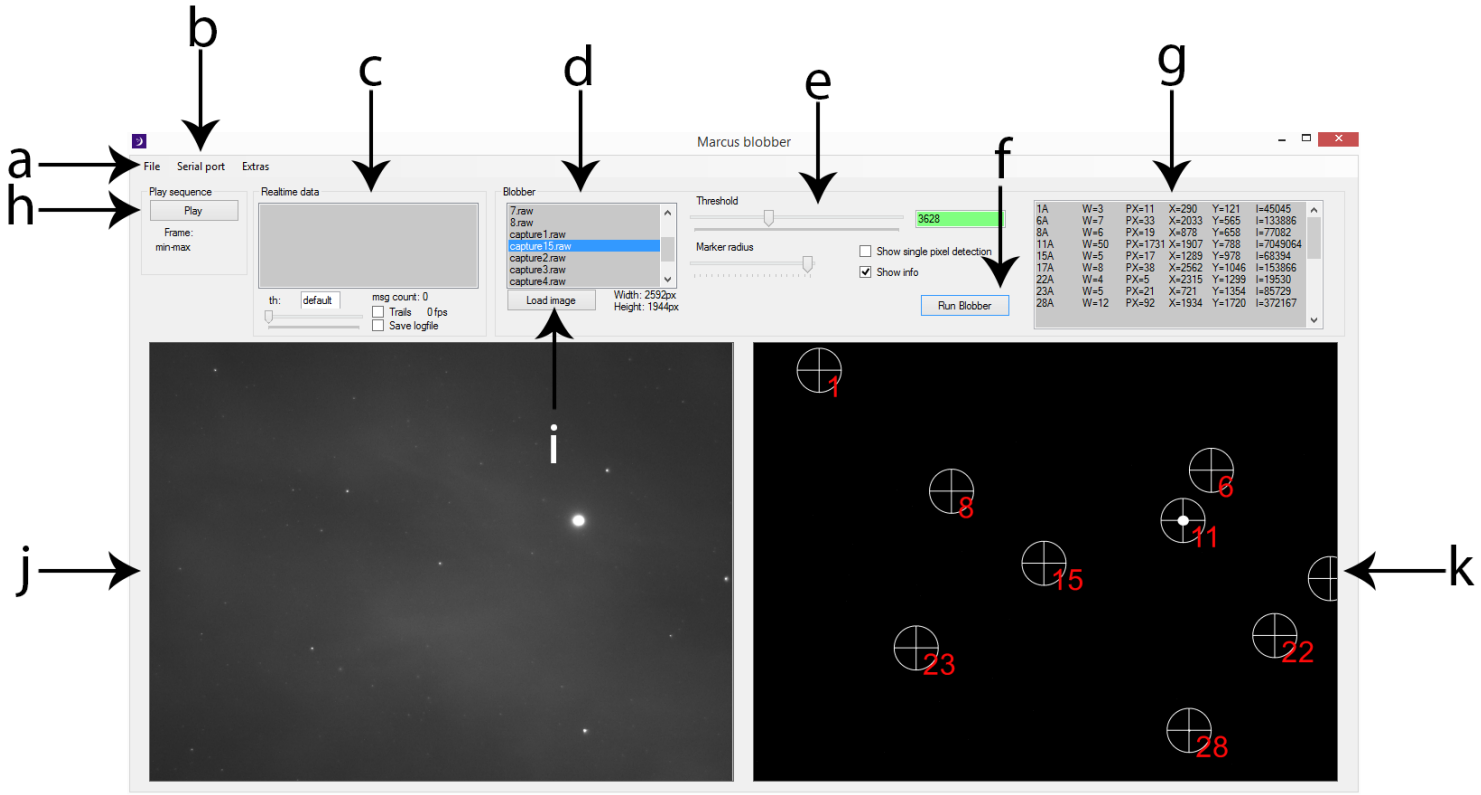


Figure 22: GUI of the multi-purpose software.

By loading an image file in the software the blob detection algorithm can be evaluated and images can be analysed with different threshold levels. The threshold can be changed and the result is shown in the right panel. A convenient matrix called Francesco is shown containing X, Y, I of each detected star in the image. The matrix is given in a Matlab friendly format. It can then be used together with star identification algorithms to perform various test.

Figure 23 shows an raw image of stars taken with the image sensor and Fujinon lens loaded in the left panel. The right panel shows the thresholded image with detected blobs. By checking the box *Show single pixel detection* the software will display all detected blobs, even ones containing only one single pixel. The result is shown in Figure 24. A significant amount of single pixel blobs is detected. This is due to the non-optimal picture taken with the camera containing a cloudy sky. Some pixels on the sensor are however defective (hot pixels) and will give some falsely detected single pixel blobs. These become visible when the *Show single pixel detection* box is ticked.

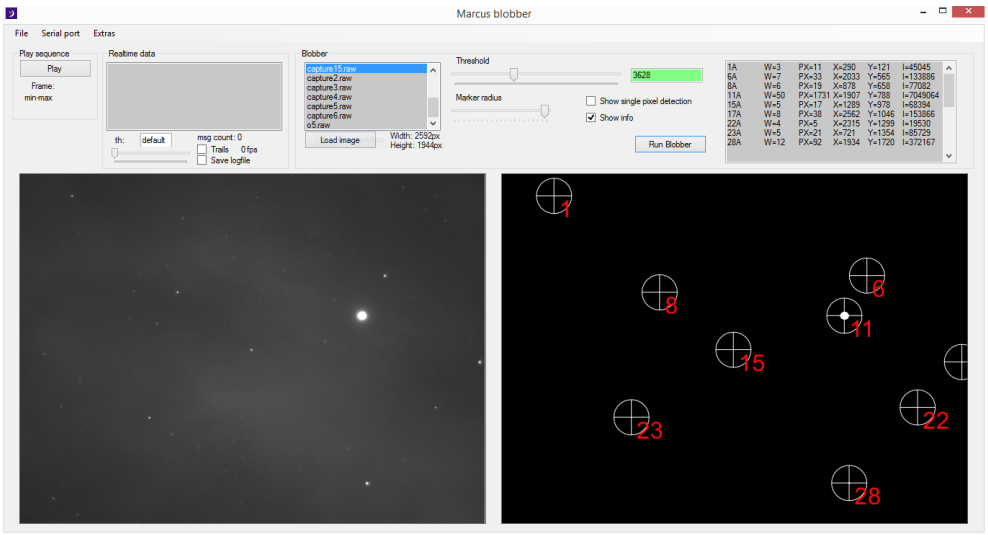


Figure 23: Blobber software. Left panel shows the loaded picture. The right panel shows the detected blobs.

Logfiles containing images saved with the camera test firmware can be used together with the software. Logfiles are opened by clicking *File-> Open logfile*. If the logfile contains several images they are loaded in a sequence. The sequence can then be played as a movie by clicking the *Play* button. Figure 26 shows a logfile loaded and displayed in the left panel.

A feature is added in the multi-purpose software to display detected stars in real-time. To use this feature a simple relaying is performed in the Smartfusion2. It forwards

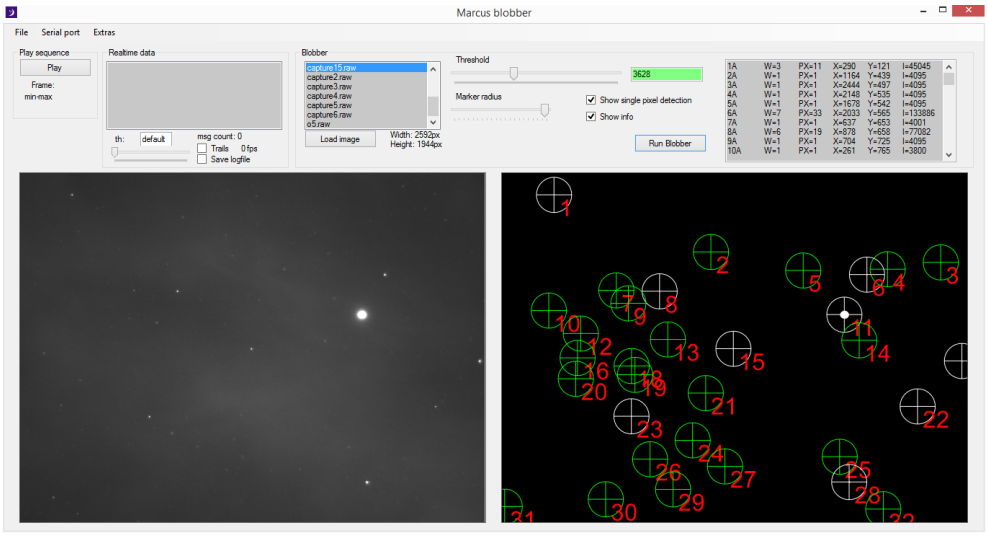


Figure 24: Blobber software showing single pixel detection with green markers.

the data received on the SPI bus from the camera FPGA to the UART connection to the computer. The data as seen on the SPI bus are repacked to a simpler format that can be interpreted by the software.

The UART package for use with real-time data is 7+3 bytes long. The message start with a header character (+) followed by an identifying bit specifying if it is a newframe message or a star. The payload in the message contains X, Y and I information of the blob. Even the number of pixels in the blob is contained in the message. Each message ends with a carriage return and line feed command. The structure of the real-time packet for usage with the software is shown in Table 13. Figure 25 shows the output format as it is captured by a serial terminal.

```
1  +0069A1E70352FA50
2  +0054540F02E90A45
3  +005A355901D3B330
4  +8000000000000000
5  +0069A1E8036D8757
6  +0054540F02E56C44
7  +005A355901D8E931
8  +8000000000000000
9  +0069A1E8035BC752
10 +0054540F02E96045
11 +005A355901D3B130
```

star 1-3, frame 1

new frame indicator

star 1-3, frame 2

new frame indicator

star 1-3, frame 3

Figure 25: Real-time data from the Smartfusion2 UART connection as captured by a serial terminal. Rows 1-3, 5-7 and 9-11 contains star data and row 4 and 8 are new frame indicators.

Table 13: Star message to be interpreted by the software.

	Value	**Number of bits**
Header	+ (0x2B)	8
Payload	Identifier (1=NF, 0=ST)	1
Payload	Spare	3
Payload	Spare	4
Payload	X	12
Payload	Y	12
Payload	I	24
Payload	PX count	8
Tail	carriage return (0x0D)	8
Tail	line feed (0x0A)	8

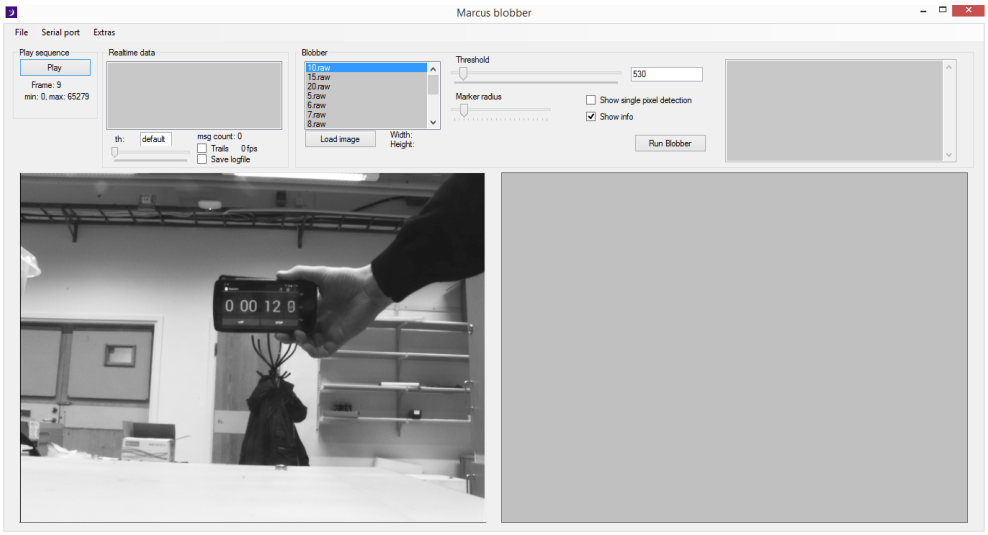


Figure 26: A logfile containing a video sequence being played in the blobber software. A phone with a stopwatch is seen on the recording for checking the frame rate.

If the UART connection of the Smartfusion2 is connected to the computer running the mult-purpose software real-time data can be displayed. The detected blobs appear in the right panel as they are received by the computer. The settings for the serial port are changed under the menu *Serial port*. The supported baud rates are:

- 38400 Baud
- 460800 Baud
- 2000000 Baud

where 460800 Baud is the default speed of the Smartfusion2 board.

6 Conclusions and Discussion

An electrical platform for a miniature star tracker has been developed and tested. An innovative star detection algorithm has been implemented in the camera FPGA and a set of communication protocols have been defined. The FPGA used for image processing is very limited but a working star detection algorithm could successfully be implemented running on the fly.

A multi-purpose software has been written to make the development of the star tracker easier and more convenient. What is yet to be done is to implement the actual LISA algorithm and the tracking of the stars in the Smartfusion2. The software currently written for the Smartfusion2 gives easy access to X, Y and I of each detected star in a frame. The SPI communication between the camera FPGA and the Smartfusion2 is fully functional but should be extended to add more features. Some of the features can be the ability to change threshold and exposure settings of the camera from the Smartfusion2.

To make the star tracker and blob detection algorithm more robust a couple of problems and design flaws need to be addressed. The blob detection method described herein contains several limitations. They can however be avoided if one is aware of them. Large object in the field of view will cause trouble. Since the hardware implementation contains physical limitations too large objects will result in overflow of signal vectors and strange phenomena's can happen. A big object will usually result in the detection of several false blobs. Big objects are the earth, sun and the moon.

Depending on the lens configuration used, the moon may not be a problematic object. A narrow field of view will make objects appear larger on the sensor and potentially cause trouble. However when the field of view is narrow a bright object such as the moon will stay in sight for shorter periods compared with a wider field of view.

A frame condition status can be implemented in the blob detection algorithm to resolve this issue. The percentage of the frame having a threshold override can be calculated and compared with a certain threshold. If the frame contains an unusually high number of pixels overriding the threshold value, the frame condition status should indicate a bad frame. If the Smartfusion2 receives a bad frame status it can decide to discard the whole frame and no tracking is performed.

A horizon in the field of view such as in Figure 27 is problematic since it is a big object. Since the image is read from top to bottom the image in Figure 27 could still work. Any other orientation of the horizon creates direct problems. In the case of horizon in the lower half of the image, the stars in the upper half will be detected as they should. When the algorithm scans the row containing the horizon a multitude of potential blobs will be detected until the number specified in $MaxStarsInFrame$ parameter is reached. The frame condition status previously mentioned can help the Smartfusion2 in this case to be aware of a bad frame captured by the camera.

Another issue is blobs located on the border of the image sensor. If a blob contains

Figure 27: Horizon in the FOV is a problem for the star tracker. The picture is a composition of an image of earth taken from MUSCAT Experiment [15] and a star picture taken with the actual image sensor for the star tracker.

pixels on any of the four borders one cannot be sure that the centroid calculation is done correct. The star might contain parts projected outside the sensor which is not detected. Hence, the blob detection algorithm may calculate the wrong centroid. In the current implementation there is no border control of detected blobs. This could easily be implemented and a blob condition flag can be set wherever it is a normal blob or a border blob detection. In that case, all blobs will still be captured and the decision to keep it or skip it is made in the Smartfusion2 software. Another way of dealing with this is to set a virtual border slightly smaller than the actual image sensor size. This can be implemented in the Smartfusion2 and be made to skip blobs detected outside the virtual border.

Fast moving object is another issue with this algorithm but heavily related to the frame rate and exposure time of the camera. Since fast moving objects in combination with relatively long exposure time results in trails this can cause issues. Horizontal trails can be handled very well but diagonal trails are more prone to cause trouble. Diagonal trails can move outside of the detecting range in the blob detection algorithm. This will make it hard for the algorithm to merge rowblobs together resulting in multiple detection of one object.

To get the best possible result from this system the image quality from the camera needs to be as good as possible. The sensor need to capture a clear image of the stars and the lens has to be in focus. The focusing of the lens is however problematic since the picture from the sensor cannot be viewed directly in an easy way. One suggestion to address this problem is to design a special PCB that can easily be attached to the connector on the image sensor board. The purpose of this PCB is to help focusing by outputting the picture from the image sensor to a monitor. The focusing process will be much easier if a visual feedback is available during the process.

The implemented protocol between the camera FPGA and the Smartfusion2 SOC is merely a suggestion and could be improved in several ways. More commands should be added so the Smartfusion2 can change more parameters inside the camera FPGA. This may involve threshold level, exposure time and sensor gain.

6.1 Social and Ethical Aspects

The star tracker presented in this thesis is targeted for space applications. By incorporating a star tracker in a satellite, it enables higher precision of measurements and more precise attitude control can be performed. Research on the environment on earth and various physical phenomena's are frequently carried out in space. The technology help us learn about phenomena's in nature where the results can benefit life on earth. By equipping more satellites with star trackers the research quality can be increased.

As always with technology, it can be used for other purposes than the one primarily intended. In the case of star trackers they have multiple uses. They are not only developed and used for space related research, but also for some less peaceful applications. The high accuracy of a star tracker is useful in airborne weapon system. Some cruise missiles use star trackers in their guidance system to increase target precision and attitude control.

The possibilities with technology are endless. There are always other ways of using technology than as primary intended. This is something to be aware of. The defence industry however pushes technology forward and has made many modern and socially appreciated innovations possible. Lockheed Martin is a company taking advantage of this by producing technology aimed for both space applications and defence. One of their products is an advanced star tracker.

References

[1] Aptina Imaging, " MT9P031 CMOS Digital Image Sensor Data Sheet (Rev.F)," 2011.

[2] Wikimedia Commons, Public Domain, "Plath sextant," http://commons.wikimedia.org/wiki/File:Sextant.jpg, 2014, [Online; accessed 17-June-2014].

[3] European Commission, "7 th Framework Programme for Research and Technological Development," http://ec.europa.eu/research/fp7/understanding/fp7inbrief/what-is_en.html, 2014, [Online; accessed 7-June-2014].

[4] "Francesco Vallegra, Private communication, MSc thesis due in 2014." 2014.

[5] "Nikola Shterev, Private communication, MSc thesis due in 2014." 2014.

[6] Space Alliance, http://www.spacealliance.ro/articles/view.aspx?id=201002250904, 2014, [Online; accessed 7-June-2014].

[7] A. R. Eisenman, C. C. Liebe, and J. L. Jorgensen, "The New Generation of Autonomous Star Trackers," http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/22609/1/97-1120.pdf, 1997, [Online; accessed 20-June-2014].

[8] Blue Canyon Technologies, http://bluecanyontech.com/all_products/star-trackers/, 2014, [Online; accessed 17-June-2014].

[9] F. Boldrini and A. Landi, "The Officine Galileo Autonomous Star Tracker," http://www.dlr.de/Portaldata/49/Resources/dokumente/archiv3/1409P.pdf, 2014, [Online; accessed 17-June-2014].

[10] Microsemi, "Single Event Effects (SEE)," http://www.microsemi.com/products/fpga-soc/reliability/see, 2014, [Online; accessed 17-June-2014].

[11] ISAAC (RX15), "SED v4.2," 2014.

[12] Sourceforge, "Teraterm v4.73," 2012. [Online]. Available: http://ttssh2.sourceforge.jp/

[13] Stellarium, http://www.stellarium.org/, 2014, [Online; accessed 11-June-2014].

[14] Aptina Imaging, "Aptina DevSuite," 2014. [Online]. Available: http://www.aptina.com/support/Devsuite.jsp

[15] MUSCAT (RX13), "SED v5.1," 2013.

A Exposure Time Calculation

Matlab code written by Nicola Schlatter for calculation of exposure time.

```
Reg03=hex2dec('0797');Row_size=Reg03;
Reg04=hex2dec('0A1F');Column_Size=Reg04;
Reg05=0;HB=Reg05+1;
Reg06=hex2dec('0019');      %Vertical Blank          Def 25      Range
Reg08=0;                    %Shutter Width Upper;    Def 0;      Range 0-15
Reg09=hex2dec('0797');      %Shutter Width Lower;    Def 1943;   Range 0-65535
Reg12=0;                    %Shutter Delay;          Def 0;      Range 0-8191


t_PIXCLK=1/32.768e6; %32.768 MHz

Shutter_Width_Upper=Reg08;
Shutter_Width_Lower=Reg09;
Shutter_Delay=Reg12+1;

Row_Skip=0;
Column_Skip=0;
Row_Bin=0;
Column_Bin=0;



VB=Reg06+1;

%shutter width
SW=max([1,(2^16*Shutter_Width_Upper)+Shutter_Width_Lower]);

SD=Shutter_Delay+1;
if SW<3
    SDmax=1232;
else
    SDmax=1504;
end
SO=208*(Row_Bin+1)+98+min(SD,SDmax)-94;

if Column_Bin==0
    Wdc=80;
elseif Column_Bin==1
    Wdc=40;
elseif Column_Bin==3
    Wdc=20;
end

HBmin=346*(Row_Bin+1)+64+(Wdc/2);
W=2*ceil((Column_Size+1)/(2*(Column_Skip+1)));


t_ROW=2*t_PIXCLK*max([((W/2)+max(HB,HBmin)),(41+346*(Row_Bin+1)+99)]);

t_EXP=SW*t_ROW-SO*2*t_PIXCLK
```

B Serial Divider

Serial divider in VHDL.

```
-----------Divider1 signals------------
SIGNAL a_div1                       : std_logic_vector(23 DOWNTO 0);
SIGNAL b_div1                       : std_logic_vector(23 DOWNTO 0);
SIGNAL result_div1                  : std_logic_vector(23 DOWNTO 0);
SIGNAL buffer_div1                  : std_logic_vector(47 DOWNTO 0);
SIGNAL cnt_div1                     : std_logic_vector(4 DOWNTO 0);
SIGNAL Data_Prepare_State           : std_logic_vector(3 DOWNTO 0);

 case Data_Prepare_State is
    when x"0" => --Idle, wait for some data

        --Relevant stuff for the center of intensity calculation
        a_div1 <= XI_sum_New; --numerator
        b_div1 <= I_sum_New; --denominator

    when x"1" => --Start dividing!
        --Just check if there is any point in doing some looping.
        --You never know, sometimes you can be lucky and able to take a shortcut↩
            !
        if a_div1 < b_div1 then --Cannot divide if numerator is smaller than ↩
             denominator. result = 0
            result_div1 <= x"000000";
            Data_Prepare_State <= x"3";
        elsif a_div1 = b_div1 then --Can divide exactly one time. result = 1
            result_div1 <= x"000001";
            Data_Prepare_State <= x"3";
        else --We have to do some calculations...
            cnt_div1 <= "00000";
            buffer_div1 <= x"000000" & a_div1;
            Data_Prepare_State <= x"2"; --Go and do some looping! :)
        end if;

    when x"2" => --Divide loop state
        if buffer_div1((48-2) DOWNTO 23) >= b_div1 then
            buffer_div1(47 downto 24) <= '0' & (buffer_div1((48-3) downto 23) - ↩
                 b_div1((24-2) downto 0));
            buffer_div1(23 downto 0) <= buffer_div1(22 downto 0) & '1';
        else
            buffer_div1 <= buffer_div1(46 downto 0) & '0';
        end if;

        if cnt_div1 /= "11000" then --24 cycles
            cnt_div1 <= cnt_div1 + 1;
            Data_Prepare_State <= x"2";
        else
            result_div1 <= buffer_div1(23 downto 0);
            Data_Prepare_State <= x"3";
        end if;

    when x"3" => --Division is done

        --Use the result in result_div1(23 DOWNTO 0)

        ...

    when others =>
        Data_Prepare_State <= x"0";
end case;
```

C Blob Detection Algorithm

The Blob Detection Algorithm in VHDL.

```
-- BlobAlg.vhd
--
-- This module is running the blob detection algorithm in four flag-synchronized↩
     state machines. It is to be used in the miniature star tracker.
-- The alorithm takes the image from the camera, detects stars/blobs, groups ↩
    them together and outputs simple (X,Y,I)-data for each star.
--
-- Each state machine can be seen as representing a layer in the algorithm in ↩
    which the first one (layer 1) is in direct contact with the pixeldata from ↩
    the sensor. Low level
-- The fourth state machine (layer 4) is eventually putting the detected stars (↩
    X,Y,I) into the SPI buffer. Highest level
--
-- 2014-05-16
-- Marcus Lindh (marculin@kth.se)
-- 073-9786602
--


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity BlobAlg is
    generic(SIZE: INTEGER := 24);
    port
        (
            CLK, Reset              : in std_logic;
            Enable                  : in std_logic;
            FrameValid              : in std_logic;
            LineValid               : in std_logic;
            Data_Camera             : in std_logic_vector(11 DOWNTO 0);
            FIFO_in                 : in std_logic_vector(63 DOWNTO 0);
            FIFO_empty              : in std_logic;

            Threshold_in            : in std_logic_vector(15 DOWNTO 0);
            Threshold_refresh       : in std_logic;

            New_Blob_Detected       : out std_logic;
            FIFO_we                 : out std_logic;
            FIFO_re                 : out std_logic;
            Blob_Data_Out           : out std_logic_vector(31 DOWNTO 0);
            FIFO_out                : out std_logic_vector(63 DOWNTO 0)
        );

end BlobAlg;

architecture behavioral of BlobAlg is


-------------------------------------------------------------------------
-- Signal declaration
-------------------------------------------------------------------------
SIGNAL Threshold_Value              : std_logic_vector(11 DOWNTO 0);        -- ↩
    thresholding value

SIGNAL Threshold_refresh_old        : std_logic;

-------------------------Blob algorithm signals-------------------------
SIGNAL Ghosting                     : std_logic_vector(7 DOWNTO 0);         --
```

```
CONSTANT Ghosting_Level             : std_logic_vector(7 DOWNTO 0):=x"19";  -- ↩
     Sets the Ghosting Level to 25

SIGNAL Blobber_State                : std_logic_vector(3 DOWNTO 0);         --
SIGNAL Data_Prepare_State           : std_logic_vector(3 DOWNTO 0);         --
SIGNAL Release_State                : std_logic_vector(3 DOWNTO 0);         --


--Signals belonging to the state machine integrating blobrelated pixels
SIGNAL XI_sum                       : std_logic_vector(23 DOWNTO 0);        --
SIGNAL I_sum                        : std_logic_vector(23 DOWNTO 0);        --
SIGNAL Blob_Width                   : std_logic_vector(7 DOWNTO 0);         --


--Signals sent from the integrating state machine to be processed.
SIGNAL XI_sum_New                   : std_logic_vector(23 DOWNTO 0);        -- ↩
     This is a two step buffer, data transferred to XI_sum
SIGNAL I_sum_New                    : std_logic_vector(23 DOWNTO 0);        -- ↩
     This is a two step buffer, data transferred to I_sum
SIGNAL X_center                     : std_logic_vector(11 DOWNTO 0);        --
SIGNAL Blob_Row                     : std_logic_vector(11 DOWNTO 0);        --


SIGNAL RowBlob_Start_Column_New     : std_logic_vector(11 DOWNTO 0);        -- ↩
     This is a two step buffer, data transferred to RowBlob_Start_Column
SIGNAL RowBlob_Start_Column         : std_logic_vector(11 DOWNTO 0);        --

SIGNAL New_Blob_To_Calculate        : std_logic;                            -- ↩
     Triggers next state machine

SIGNAL New_Frame                    : std_logic;



-- Data related to the blob at first position in FIFO
SIGNAL FIFO1_X_center               : std_logic_vector(11 DOWNTO 0);        --
SIGNAL FIFO1_X_center_Sum           : std_logic_vector(23 DOWNTO 0);        --
SIGNAL FIFO1_Rowspan                : std_logic_vector(7 DOWNTO 0);         --
SIGNAL FIFO1_IrowN_sum              : std_logic_vector(23 DOWNTO 0);        --
SIGNAL FIFO1_Irow_sum               : std_logic_vector(23 DOWNTO 0);        --
SIGNAL FIFO1_Y_stopp                : std_logic_vector(11 DOWNTO 0);        --
SIGNAL FIFO1_PXcount                : std_logic_vector(9 DOWNTO 0);         --

SIGNAL FIFO1_Data_Snatched          : std_logic;                            -- ↩
     Data is waiting in the FIFO1 to be used. Only compare values with FIFO1 if ↩
     this is '1'.
SIGNAL FIFO1_Temp1                  : std_logic_vector(63 DOWNTO 0);        --
SIGNAL FIFO1_Temp2                  : std_logic_vector(63 DOWNTO 0);        --


SIGNAL FIFO1_Conditioner_State      : std_logic_vector(3 DOWNTO 0);         --




-- Data that is used from a blowrow when adding data to the FIFO
SIGNAL RowBlob_X_center             : std_logic_vector(11 DOWNTO 0);        --
SIGNAL RowBlob_X_center_Sum         : std_logic_vector(23 DOWNTO 0);        --
SIGNAL RowBlob_Irow_sum             : std_logic_vector(23 DOWNTO 0);        --
SIGNAL RowBlob_IrowN_sum            : std_logic_vector(31 DOWNTO 0);        --
SIGNAL RowBlob_Row                  : std_logic_vector(11 DOWNTO 0);        --
SIGNAL RowBlob_PXcount              : std_logic_vector(11 DOWNTO 0);        --
SIGNAL RowBlob_NewFlag              : std_logic;
SIGNAL RowBlob_Rowspan              : std_logic_vector(7 DOWNTO 0);         --
```

```
SIGNAL Tmp1_message1                : std_logic_vector(63 DOWNTO 0);        --
SIGNAL Tmp1_message2                : std_logic_vector(63 DOWNTO 0);        --

SIGNAL Tmp2_message1                : std_logic_vector(63 DOWNTO 0);        --
SIGNAL Tmp2_message2                : std_logic_vector(63 DOWNTO 0);        --


SIGNAL ReleaseFlag                  : std_logic;



-- The current column and Row we are reading camera data from
SIGNAL Column                       : std_logic_vector(11 DOWNTO 0);        --
SIGNAL Row                          : std_logic_vector(11 DOWNTO 0);        --


SIGNAL Newframe_Send_Flag           : std_logic;



-- Count the numbers of stars detected in the Frame.
SIGNAL StarsInFrame                 : std_logic_vector(4 DOWNTO 0);
CONSTANT MaxStarsInFrame            : std_logic_vector(7 DOWNTO 0):=x"19";




----------------------------Divider1 signals-----------------------------
SIGNAL a_div1                       : std_logic_vector(23 DOWNTO 0);
SIGNAL b_div1                       : std_logic_vector(23 DOWNTO 0);
SIGNAL result_div1                  : std_logic_vector(23 DOWNTO 0);
SIGNAL buffer_div1                  : std_logic_vector(47 DOWNTO 0);
SIGNAL cnt_div1                     : std_logic_vector(4 DOWNTO 0);



----------------------------Divider2 signals-----------------------------
SIGNAL a_div2                       : std_logic_vector(23 DOWNTO 0);
SIGNAL b_div2                       : std_logic_vector(23 DOWNTO 0);
SIGNAL result_div2                  : std_logic_vector(23 DOWNTO 0);
SIGNAL buffer_div2                  : std_logic_vector(47 DOWNTO 0);
----------------------------Divider3 signals-----------------------------
SIGNAL a_div3                       : std_logic_vector(23 DOWNTO 0);
SIGNAL b_div3                       : std_logic_vector(23 DOWNTO 0);
SIGNAL result_div3                  : std_logic_vector(23 DOWNTO 0);
SIGNAL buffer_div3                  : std_logic_vector(47 DOWNTO 0);

SIGNAL cnt_div23                    : std_logic_vector(4 DOWNTO 0);




-----Now the journey begins!-----------------------------------------
begin

    ThresholdUpdate:process(RESET,CLK)
    BEGIN
        IF(RESET = '1') THEN
            Threshold_refresh_old <= '0';
            Threshold_Value <= x"200"; --Default value, 512
        ELSIF falling_edge(CLK) THEN
```

```
            --Refresh threshold value on rising edge of the threshold refresh ↩
                signal
            if Threshold_refresh = '1' and Threshold_refresh_old = '0' then
                Threshold_Value <= Threshold_in(11 downto 0);
            end if;

            Threshold_refresh_old <= Threshold_refresh;

        END IF;

    END PROCESS;



    Blobbing:process (CLK, Reset) --Blob detection algorithm
    begin
        if Reset /= '0' then

            Blobber_State <= x"0";
            XI_sum <= x"000000";
            I_sum <= x"000000";
            Column <= x"000";
            Row <= x"000";
            Blob_Row <= x"000";

            RowBlob_Start_Column <= x"000";
            RowBlob_Start_Column_New <= x"000";

            Data_Prepare_State <= x"0";

            New_Blob_To_Calculate <= '0';

            FIFO_we <= '0';
            FIFO_re <= '0';

            FIFO1_Conditioner_State <= x"0";
            FIFO1_Data_Snatched <= '0';

            New_Blob_Detected <= '0'; --Important to have this here!. Otherwise ↩
                if it is 1 at start the state machine will get stuck
            FIFO1_Data_Snatched <= '0';

            RowBlob_NewFlag <= '0';
            New_Frame <= '0';

            --Fourth state machine for releasing blobs
            Release_State <= x"0";
            ReleaseFlag <= '0';

            Ghosting <= x"00";

            StarsInFrame <= "00000";

        elsif falling_edge(CLK) then

            --Outside the frame. Clear variables
            if FrameValid = '0' then
                Column <= x"000";
                Row <= x"000";
                Blobber_State <= x"0";
                StarsInFrame <= "00000"; --Clear the blob counter for each new ↩
                    frame
            else
                Newframe_Send_Flag <= '1';
            end if;


            --The ghosting helps for dead pixels in a row and decides when a ↩
```

```
            blob on row is ended
        if Ghosting > 0 then
            Ghosting <= Ghosting - 1;
        end if;









        ------------Algorithm layer 4
        case Release_State is
            when x"0" => --Idle state, wait for some data
                if ReleaseFlag = '1' then
                    Tmp2_message1 <= Tmp1_message1; --Put data to the ↩
                        temporary  signal vector
                    Tmp2_message2 <= Tmp1_message2; --Put data to the ↩
                        temporary  signal vector
                    Release_State <= x"1";

                    New_Frame <= '0'; --Ordinary message

                elsif FrameValid = '0' and Newframe_Send_Flag = '1' then --↩
                    Outside the frame, send one newframe message
                    Release_State <= x"D";
                    Newframe_Send_Flag <= '0';

                    New_Frame <= '1'; --This tells the Blobmessage that I ↩
                        should send a NewFrameMessage

                end if;


            when x"1" =>
                Release_State <= x"2";

                --Divider2 calculates the X_center by mean
                a_div2 <= Tmp2_message1(63 downto 40); --Sum of all ↩
                    X_centers                    --numerator.
                b_div2 <= "000000000000000000" & Tmp2_message1(15 downto 10)↩
                    ; --Number of rows  --denominator.

                --Divider3 calculates


                a_div3 <= Tmp2_message2(63 downto 40); --Sum of ↩
                    rowintensities*n    --numerator
                b_div3 <= Tmp2_message2(39 downto 16); --Sum of ↩
                    rowintensities      --denominator


            when x"2" =>
                Release_State <= x"3";

                buffer_div2 <= x"000000" & a_div2; --prepare buffer for ↩
                    dividing
                buffer_div3 <= x"000000" & a_div3; --prepare buffer for ↩
                    dividing
                cnt_div23 <= "00000"; --Reset the counter



            when x"3" => --Divide loop state

                --Divider 2
                if buffer_div2((48-2) DOWNTO 23) >= b_div2 then
                    buffer_div2(47 downto 24) <= '0' & (buffer_div2((48-3) ↩
```

```
                    downto 23) - b_div2((24-2) downto 0));
                buffer_div2(23 downto 0) <= buffer_div2(22 downto 0) & ↩
                    '1';
            else
                buffer_div2 <= buffer_div2(46 downto 0) & '0';
            end if;

            --Divider 3
            if buffer_div3((48-2) DOWNTO 23) >= b_div3 then
                buffer_div3(47 downto 24) <= '0' & (buffer_div3((48-3) ↩
                    downto 23) - b_div3((24-2) downto 0));
                buffer_div3(23 downto 0) <= buffer_div3(22 downto 0) & ↩
                    '1';
            else
                buffer_div3 <= buffer_div3(46 downto 0) & '0';
            end if;


            if cnt_div23 /= "11000" then --24 cycles
                cnt_div23 <= cnt_div23 + 1;
                Release_State <= x"3";
            else
                result_div2 <= buffer_div2(23 downto 0);
                result_div3 <= buffer_div3(23 downto 0);
                Release_State <= x"4";
            end if;



        when x"4" => --Division is done
            Release_State <= x"D";



        when x"D" =>
            Release_State <= x"E";

            if ((Tmp2_message1(9 downto 0) > 1) or New_Frame = '1') then↩
                --There where more than 1 pixels in the blob
                New_Blob_Detected <= '1'; --Enable writing to UART ↩
                    buffer

                if New_Frame = '1' then --New frame message

                    Blob_Data_Out <= "1" & "000" & "0000" & x"000000";

                else --ordinary message
                    --Blob_Data_Out <= result_div2(11 downto 0) & "0" & ↩
                        Tmp2_message1(5 DOWNTO 0) & "0" & Tmp2_message1↩
                        (27 downto 16);

                    Blob_Data_Out <= "0" & "000" & "0000" & result_div2↩
                        (11 downto 0) & ((Tmp2_message1(27 downto 16) -↩
                        ("000000" & Tmp2_message1(15 downto 10))) + ↩
                        result_div3(11 downto 0));

                    --Increase the detected star counter
                    StarsInFrame <= StarsInFrame + 1;
                end if;

            end if;


        when x"E" =>
            Release_State <= x"F";

            if New_Frame = '1' then --New frame message
                Blob_Data_Out <= x"00000000"; --The second message
```

```
                else
                    Blob_Data_Out <= Tmp2_message2(39 downto 16) & ↩
                        Tmp2_message1(7 DOWNTO 0);
                end if;



            when x"F" =>
                Release_State <= x"0";

                New_Blob_Detected <= '0'; --Stop writing to UART buffer
                ReleaseFlag <= '0'; --Reset the flag that started this state↩
                     machine


            when others =>
                Release_State <= x"0";
        end case;







        ------------Algorithm layer 3

        --This state machine gets the data from the FIFO buffer and stores ↩
             it in the temporary vectors.
        --It also checks if the FIFO1 blob should be released or not. It ↩
             also serves for adding new blobs to the FIFO,
        --regardless they are new or composed from FIFO1 data
        case FIFO1_Conditioner_State is
            when x"0" => --Idle state, wait for some data

                if FIFO1_Data_Snatched = '0' and FIFO_empty = '0' then
                    FIFO1_Conditioner_State <= x"1";
                    FIFO_re <= '1'; -- NOTE! Data is delayed 2clk cycles. ↩
                        Set re high 2clk cycles before accessing data.

                elsif FIFO1_Data_Snatched = '1' then
                    FIFO1_Conditioner_State <= x"5";     --Jump directly to ↩
                        the waiting state to release the blob or combine ↩
                        with new row data

                elsif RowBlob_NewFlag = '1' then
                    FIFO1_Conditioner_State <= x"A";     --Go and enqueue ↩
                        data. Since the FIFO is empty this is a completely ↩
                        new blob
                end if;

            --Grab data from the FIFO (it is actually 2x packages)
            when x"1" =>
                FIFO1_Conditioner_State <= x"2";

            when x"2" =>
                FIFO1_Conditioner_State <= x"3";
                FIFO1_Temp1 <= FIFO_in;             --Read first data ↩
                    package
                FIFO_re <= '0';                     --stop reading


            when x"3" =>
                FIFO1_Conditioner_State <= x"4";
                FIFO1_Temp2 <= FIFO_in;             --Read second data ↩
                    package

                --Extract data from first package
```

```
                FIFO1_X_center_Sum  <= FIFO1_Temp1(63 DOWNTO 40);
                FIFO1_X_center      <= FIFO1_Temp1(39 DOWNTO 28);
                FIFO1_Y_stopp       <= FIFO1_Temp1(27 DOWNTO 16);
                FIFO1_Rowspan       <= "00" & FIFO1_Temp1(15 DOWNTO 10);
                FIFO1_PXcount       <= FIFO1_Temp1(9 DOWNTO 0);


            when x"4" =>
                FIFO1_Conditioner_State <= x"5";

                FIFO1_Data_Snatched <= '1';       --We have snatched the ↩
                    data from the FIFO1 position. (dequeued)

                --Extract data from second package

                FIFO1_IrowN_sum <= FIFO1_Temp2(63 DOWNTO 40);
                FIFO1_Irow_sum <= FIFO1_Temp2(39 DOWNTO 16);



            --This is the waiting state. Check if the blob should be ↩
                released. If we are in this state and the
            --ghosting counter is noted to be > 0 in a certain region; the ↩
                FIFO1 blob belongs to the ongoing blob!
            --New blobs can be added.
            --If we are outside the frame and still have blobs in the FIFO, ↩
                the FIFO is cleared, blobs sent to SPI buffer. This
            --happens if there are blobs on the last row.
            when x"5" =>

                if FrameValid = '0' and FIFO_empty = '0' then --We need to ↩
                    clear the FIFO. This happens if there are blobs on the ↩
                    bottom border
                    FIFO1_Conditioner_State <= x"E";    --Send to SPI buffer
                else

                    if (Column > (FIFO1_X_center - x"00A")) and (Column < (↩
                        FIFO1_X_center + x"00A")) and (Ghosting > 0) then
                        FIFO1_Conditioner_State <= x"6";    -- Ongoing ↩
                            rowblob belongs to the FIFO1 blob. Combine!

                    elsif RowBlob_NewFlag = '1' then
                        FIFO1_Conditioner_State <= x"A";    -- A completely ↩
                            new blob detected!

                    elsif (Column > (FIFO1_X_center + x"00A")) and (↩
                        FIFO1_Y_stopp /= Row) then
                        FIFO1_Conditioner_State <= x"E";    -- Release the ↩
                            blob, Send to SPI buffer in next state machine ↩
                            (layer4)

                    end if;

                end if;



            when x"6" => --Wait for the ongoing rowblob to finish. We will ↩
                merge some data and then put it back into the FIFO
                if RowBlob_NewFlag = '1' then
                    FIFO1_Conditioner_State <= x"7";
                end if;


            when x"7" => --Here we combine the new rowblob data with the ↩
                data saved in the FIFO
                FIFO1_Conditioner_State <= x"B";
```

```
                    FIFO1_Data_Snatched <= '0'; --The previously snatched data ↩
                        is no longer valid. Go and resnatch in state 0.

                    RowBlob_X_center_Sum <= RowBlob_X_center + ↩
                        FIFO1_X_center_Sum;
                    RowBlob_PXcount <= RowBlob_PXcount + FIFO1_PXcount;
                    RowBlob_Rowspan <= FIFO1_Rowspan + 1;


                    --Build the sums used for calculating Y-center

                    RowBlob_Irow_sum <= RowBlob_Irow_sum + FIFO1_Irow_sum;
                    RowBlob_IrowN_sum <= RowBlob_IrowN_sum(23 DOWNTO 0)*↩
                        FIFO1_Rowspan + FIFO1_IrowN_sum;



                when x"A" => --We go here if it was a completely new blob
                    FIFO1_Conditioner_State <= x"B";

                    RowBlob_Rowspan <= x"01"; -- one row
                    RowBlob_X_center_Sum <= x"000" & RowBlob_X_center; --The ↩
                        first value to the sum

                --This part queues data to the FIFO
                when x"B" =>
                    FIFO1_Conditioner_State <= x"C";

                    --Clock out the first message
                    FIFO_out <= RowBlob_X_center_Sum(23 DOWNTO 0) & ↩
                        RowBlob_X_center(11 DOWNTO 0) & RowBlob_Row(11 DOWNTO ↩
                        0) & RowBlob_Rowspan(5 DOWNTO 0) & RowBlob_PXcount(9 ↩
                        DOWNTO 0);
                    FIFO_we <= '1';



                when x"C" =>
                    FIFO1_Conditioner_State <= x"D";

                    --Clock out the second packet
                    FIFO_out <= RowBlob_IrowN_sum(23 DOWNTO 0) &  ↩
                        RowBlob_Irow_sum(23 DOWNTO 0) & x"ABCD";



                when x"D" =>
                    FIFO1_Conditioner_State <= x"0";

                    FIFO_we <= '0';
                    RowBlob_NewFlag <= '0'; --Clear the sending flag




                when x"E" =>--Create the temp messages
                    FIFO1_Conditioner_State <= x"0";

                    ReleaseFlag <= '1'; --This starts the fourth state machine
                    Tmp1_message1 <= FIFO1_X_center_Sum(23 downto 0) & ↩
                        FIFO1_X_center(11 downto 0) & FIFO1_Y_stopp(11 downto ↩
                        0) & FIFO1_Rowspan(5 DOWNTO 0) & FIFO1_PXcount(9 downto↩
                         0);
                    Tmp1_message2 <= FIFO1_IrowN_sum(23 DOWNTO 0) & ↩
                        FIFO1_Irow_sum(23 DOWNTO 0) & x"0000";

                    FIFO1_Data_Snatched <= '0';         --We have consumed the ↩
                        data temporary stored from FIFO1. We need to get some ↩
```

```
                    new data
                RowBlob_NewFlag <= '0';


            when others =>
                FIFO1_Conditioner_State <= x"0";
        end case;









        ------------Algorithm layer 2

        --Prepare the data for enqueueing
        case Data_Prepare_State is
            when x"0" => --Idle, wait for some data
                if New_Blob_To_Calculate = '1' then
                    Data_Prepare_State <= x"1";

                    --Reset the flag that brought us here
                    New_Blob_To_Calculate <= '0';

                    --Relevant stuff for the center of intensity calculation
                    a_div1 <= XI_sum_New; --numerator
                    b_div1 <= I_sum_New; --denominator


                    --We alreade have this data so just move it to their ↩
                        respectively signals
                    RowBlob_Irow_sum <= I_sum_New;
                    RowBlob_IrowN_sum(31 DOWNTO 0) <= x"00" & I_sum_New; --↩
                        This is the first value so n=1 (I_sum_New*n = ↩
                        I_sum_New)

                    RowBlob_Row <= Blob_Row;
                    RowBlob_PXcount <= "0000" & Blob_Width;
                    RowBlob_Start_Column <= RowBlob_Start_Column_New;

                end if;

            when x"1" => --Start dividing!

                if Blob_Width = x"01" then --Can divide exactly one time if ↩
                    there is only 1 pixel in the rowblob
                    result_div1 <= x"000001";
                    Data_Prepare_State <= x"3";
                else --We have to do some calculations...
                    cnt_div1 <= "00000";
                    buffer_div1 <= x"000000" & a_div1;
                    Data_Prepare_State <= x"2"; --Go and do some division ↩
                        looping! :)
                end if;

            when x"2" => --Divide loop state

                if buffer_div1((48-2) DOWNTO 23) >= b_div1 then
                    buffer_div1(47 downto 24) <= '0' & (buffer_div1((48-3) ↩
                        downto 23) - b_div1((24-2) downto 0));
                    buffer_div1(23 downto 0) <= buffer_div1(22 downto 0) & ↩
                        '1';
                else
                    buffer_div1 <= buffer_div1(46 downto 0) & '0';
                end if;
```

```
            if cnt_div1 /= "11000" then --24 cycles
                cnt_div1 <= cnt_div1 + 1;
                Data_Prepare_State <= x"2";
            else
                result_div1 <= buffer_div1(23 downto 0);
                Data_Prepare_State <= x"3";
            end if;


        when x"3" => --Division is done
            Data_Prepare_State <= x"0";
            RowBlob_NewFlag <= '1'; -- This triggers the next state ↩
                machine
            RowBlob_X_center <= result_div1(11 DOWNTO 0) + ↩
                RowBlob_Start_Column; --X_center is the result from the↩
                 division


    when others =>
            Data_Prepare_State <= x"0";
    end case;









    ------------Algorithm layer 1

    --This state machine checks if pixelvalue > threshold. If so it ↩
        integrates the intensity and relevant sums.
    --When we are done integrating the blob on this row, the pixelvalue ↩
        is < threshold and ghosting = 0 we set the
    --flag New_Blob_To_Calculate <= '1'. This signals that we can ↩
        trigger the other state machine to start doing some ↩
        calculations.
    if LineValid = '1' then

        --Counting rows and columns
        if Column /= x"A1F" then --(2592-1)
            Column <= Column + 1;
        else
            Column <= x"000";
            Row <= Row + 1;
        end if;


        case Blobber_State is
            when x"0" =>        --- idle. Waiting for threshold override
                if Data_Camera > Threshold_Value then
                    Blobber_State <= x"1";
                    Ghosting <= Ghosting_Level;

                    --initiate variables
                    RowBlob_Start_Column_New <= Column;
                    Blob_Width <= x"01";
                    XI_sum <= x"000" & Data_Camera;
                    I_sum <= x"000" & Data_Camera;
                end if;

            when x"1" =>

                if Data_Camera > Threshold_Value then
                    Ghosting <= Ghosting_Level;
                    Blob_Width <= Blob_Width + 1;
```

```
                        XI_sum <= XI_sum + Blob_Width*(Data_Camera);
                        I_sum <= I_sum + Data_Camera;
                    end if;

                    if Ghosting = x"00" then --If ghosting is 0, next step
                        Blobber_State <= x"2";

                        XI_sum_New <= XI_sum;
                        I_sum_New <= I_sum;
                        Blob_Row <= Row;
                    end if;

                when x"2" =>

                    Blobber_State <= x"0"; --Go back

                    if StarsInFrame < MaxStarsInFrame then --Only process ↩
                         the star if we have detected less than the maximum ↩
                         number of stars in a frame
                        New_Blob_To_Calculate <= '1'; --Calculate the Center↩
                             of intensity of the blobrow, start next state ↩
                            machine
                    end if;


                when others =>
                    Blobber_State <= x"0";
            end case;
        end if;


    end if;
  end process;

end behavioral;
```