AAUSAT6 camera solution

- an image capturing unit for use in a cubesat

P4 project report
Group 412

Aalborg University
Electronic Engineering and IT
Frederik Bajers vej 7
DK-9000 Aalborg

This report is compiled in LaTeX, originally developed by Leslie Lamport, based on Donald Knuth's TeX. The main text is written in *Computer Modern* pt 11, designed by Donald Knuth. Flowcharts and diagrams are made using Microsoft Visio.

Institute of Electronic Systems
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø

AALBORG UNIVERSITY
STUDENT REPORT

Title:
AAUSAT6 camera solution

Abstract:
This project is going to be awesome!

Theme:
Digital Systems Design

Project Period:
P4: 2. February 2015 - 27. May 2015

Project Group:
15gr412

Participants:
Amalie Vistoft Petersen
Mikkel Krogh Simonsen
Rasmus Gundorff Sæderup
Simon Bjerre Krogh
Thomas Kær Juel Jørgensen
Thomas Rasmussen

Supervisor:
Jens Dalsgaard Nielsen

Copies: 8

Page Numbers: ??

Date of Completion:
January 25, 2016

Contents

Preface

This project is centered around designing a functional prototype for a camera payload to a cubesat.
The reader of the report is assumed to have a basic understanding of electronics, especially digital electronics. A glossary is made, in which the abbreviations used throughout the report are listed. Concepts not familiar for the average reader will be explained, such as the workings of a camera and the communication interfaces used in the project. The code is written in VHDL and C and the reader of the report is assumed to have a basic understanding of these languages. In the implemented code the return values will not be checked to verify success.

The report is split into three parts. The first part covers a technical description of the things that are needed to be able to design the system. The second part covers the actual design and implementation phase, where the functionalities of the system are found, and hardware modules are identified and chosen. The final section includes an acceptance test, in which the overall functions of the system are tested, to see if the modules work together as intended. Finally, it is discussed how the prototype can be improved.

The group would like to thank mediaology student Joakim Bruslund Haurum for his help with providing information about imaging and demosaicing. The group would also like to thank computer science student Jens Hegner Stærmose for his help with discussing ideas for a compression algorithm. A big thanks also goes to the project groups supervisor, associate professor Jens Frederik Dalsgaard Nielsen, for his great help throughout the project, with everything from debugging code, discussing the ideas for implementation the group has come up with, and providing feedback at the groups work.

Aalborg Universitet, 17. december 2014

Amalie "Chewie" Vistoft Petersen
apet13@student.aau.dk

Mikkel 'Hulk' Krogh Simonsen
mksi13@student.aau.dk

Rasmus 'Pynte-over-politi-kommisær' Gundorff Sæderup
rsader13@student.aau.dk

Simon 'Nightmare' Bjerre Krogh
skrogh13@student.aau.dk

Thomas 'T-bone' Kær Juel Jørgensen
tkjj13@student.aau.dk

Thomas "Godlike" Rasmussen
trasm12@student.aau.dk

Part I

Preanalysis

1 | Satellite analysis

Since 2003, students at Aalborg University (AAU) have been designing, building and launching small satellites called cubesats. So far, five satellites has been built, with payloads varying from imaging (AAU Cubesat), magnetic measurements (AAUSAT II) and tracking ships using automatic identification system (AIS) signals (AAUSAT3 to 5) [AAU Student Space, 2015]. As part of developing the next generation of satellites, a payload has to be determined and designed.
A camera is one possible payload, capable of capturing images of the earth. A prototype of such a payload will be designed and implemented in this project. Due to economic and time constraints, only a prototype is made, since this shows off the basic idea of the system, but can be made in a realisitic timeframe.

Designing electronics for use in a space environment sets a lot of requirements in regards to power consumption, data transmission speed (due to a downlink speed down to earth) and the physical size among others. These considerations needs to be pointed out, before they can be taken into account in the design phase.

This will be done in the following sections, where the knowledge necessary to design the project will be obtained. This includes an analysis of the AAUSAT satellite platform, how a camera works, and the communication interfaces to be used in the project.

In this section, an overall look of the AAUSAT's topology and its psychical limitations will be given.

1.1 AAUSAT

Before building a satellite, it must be considered how big the satellite shall be. A cubesats size is measured in what is called "units". 1U is equivalent to approximately 1 kg within the size of $10 \times 10 \times 11$ centimeters, see Figure 1.2. A unit is scalable giving a 2 unit (2U) cubesat an increase in weight to 2 kg and an increase in dimension to $10 \times 10 \times 22$. Same scaling proportion goes for the 3 unit cubesat. The size of the satellite determines the cost of the deployment into space. If the satellite requires more space than one unit, the cost of the deployment will be bigger. Cubesats can be any whole multiple of units in length, but 1U, 2U and 3U satellites are the most common sizes. AAU Student Space has chosen the size of the satellite to one unit. Therefore, the following physical systems have to fit within the boundaries of one unit.

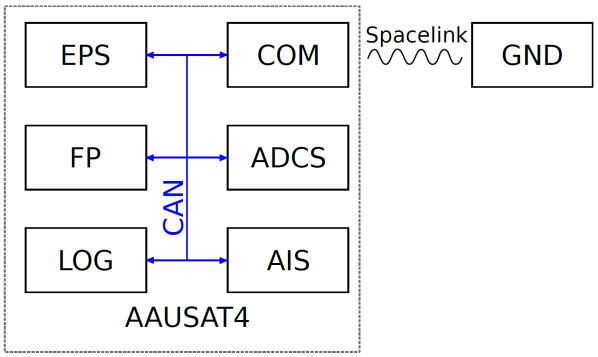


Figure 1.1: The AAUSAT subsystem topology, describing how all system are intertwined and communicating via the same controlled area network (CAN) bus

A satellite can consists of multiple subsystems, each having a different functionality. The hardware-implemented subsystems used in AAUSAT3 to AAUSAT5 are the following, and can be seen on Figure 1.1:

- **Electronic power system (EPS)**

  This subsystem is the satellites power distribution unit. It is in charge of the satellite's health, (de)charging of the batteries and turning the other subsystems on or off when needed. The primary function is to keep the batteries healthy and operational. The EPS can be thought of as the primary subsystem of the satellite, since it controls all the other subsystems. It is the first system to turn on when there is power enough and the last to turn off when there isn't [Space, 2015b, p. 12].

- **Communication (COM)**

  COM is a bidirectional communication system. It communicates with the ground station on earth and transmits the status of the satellite, including power consumption, error log, payload data etc. When the COM is transferring data to ground it can have different downlink speeds depending on how much data there is needed to be send. The downlink speed can be set between 1200 bps to 19200 bps, but is typically set to 9600 bps [Space, 2013a]. When the satellite has to send data to ground, it is sent as cubesat space protocol (CSP) packages, see section 1.3. Each package have the size of either 128 bytes or 250 bytes, in which 23 or 84 bytes are the payload-data that is needed transferred [Space, 2013c]. The ground station cannot connect to the satellite whenever needed, since the satellite has to be in the right position in regards to the ground station. From the ground station in Aalborg, a connection to the satellite can be established about 10 times a day in a time interval about 8 to 15 minutes, summing to approximately 120 minutes each day [Space, 2013b]. In this time, the satellite can send and receive data (but not simultaneously) depending on which commands the satellite has to execute [Space, 2015b, p. 12].

- **Attitude determination and control system (ADCS)**

  When the satellite is released into orbit it might begin to tumble around itself. The ADCS's job is to make sure it de-tumbles, as much as possible. This happens when the ADCS activates the magnetorquers (coils generating a magnetic field when current is sent through them) in a certain order. When this happens, the satellite should detumble down to two rotations around its own axis for each orbit around the Earth. [Space, 2015b, p. 13]

  One orbit takes about 100 minutes. Thus, the satellite should have rotation around itself lesser than 0,12 $\frac{\text{deg}}{\text{s}}$ [Space, 2015a, p. 10].

- **Payload (AIS)**

  The primary mission/payload for AAUSAT3 to 5 is to observe ship activity around Greenland. All larger ships have to transmit AIS data, containing information about its position, destination, speed etc. On the satellite, this observation happens through the AIS which receives AIS signals from the ships, saves them on board the satellite and then sends them to the ground station through COM [Space, 2015b, p. 16].

Furthermore, there are two pure software based subsystems, the Flight Planner and the system log (LOG). These can be programmed on any of the previously described subsystems and thereby they do not have a specified board just for them [Space, 2015b, p. 14]. In AAUSAT3-5, they are implemented on the COM subsystem.

- **Flight planner (FP)**

  If the ground station wishes to turn on certain parts of the satellite or perform tasks at a certain time, the flight planner is used. Normally when the satellite is in orbit it begins to take AIS samples without interference from the FP, when there is enough power to turn on the AIS. When the FP is activated, it acts as a delay for the CSP [Space, 2015b, p. 14], see 1.3, and thereby makes it possible to start the tasks at a different time. This means that samples can be taken at a certain time, i.e. when the satellite is over a certain location, for instance Greenland. The FP's task is therefore to push the experiments to another time, where it is not possible to communicate with the satellite from ground.

- **LOG**

  The LOG's task is to record all logs from the subsystems and save them. It records many things such as: time, errors, warnings, subsystem messages etc. These can then be transmitted to ground during a later pass.

These are the most essentials subsystem in the satellite. Each of these subsystems are built on their own PCB, except FP and LOG but these can be implemented on any of the physical printed circuit board (PCB)'s. The EPS is the only system that requires more space than the other systems, because of the batteries.

The voltage across the batteries is between 6,4 V − 8,0 V[Space, 2015a]. This sets the maximum voltage requirement for each subsystem to max 8 volt. It is specified in [Space, 2015a, p. 11] that if all subsystems is active at the same time the EPS must be capable of powering all the subsystems simultaneously for one hour. This means that the solarpanels and the batteries must have the ability to deliver enough energy for a solid hour to power all subsystems at once. When all subsystems is running, the power consumption is around 1 W[Space, 2014].

The satellite is build upon a decentralized philosophy. This means that each individual subsystem can work on its own, since each subsystem have its own processor and storage unit. If one of the system crashes, the EPS ensures that it will not take down the whole satellite but only kills the system which have failed. The satellite can therefore not be seen as one system but as one unit with four cooperating independent subsystems. There are lots of advantages of a decentralized system. The main advantage is that each subsystem can work on its own and thereby is not dependent on an availability of any computer, but it also enables different teams to work on different parts of the satellite at the same time.

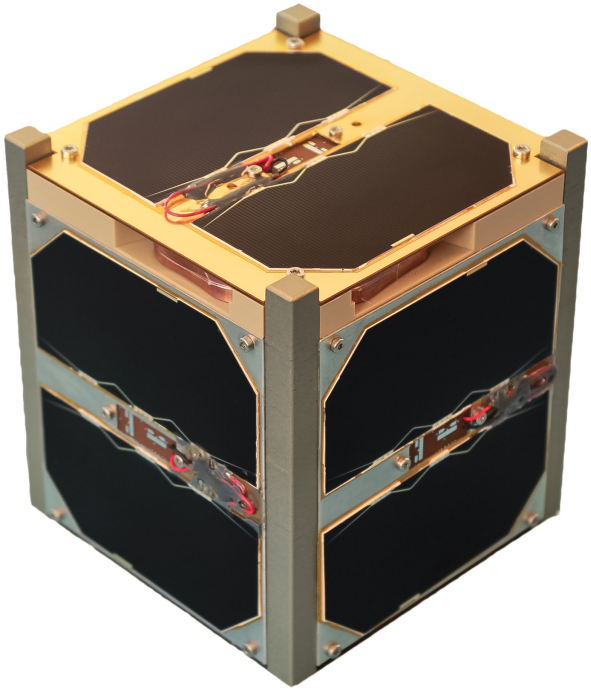


Figure 1.2: AAUSAT5, having the size of 1U [Saederup, 2015]

For the subsystems to be able to talk to each other, some kind of communication protocol is needed. The communication between the subsystems in the AAUSAT's is implemented by using CAN and CSP at the rate of 500 Kbps [team, 2015, step 1.7.4]. These will be described in the following section.

1.2 Communication to the satellite

To integrate any system into the satellite, some form of communication is necessary for the system can communicate with the other subsystems. It has not yet been decided what the interface of this communication should be in AAUSAT6, but it is likely that it will be CAN given that it has been used on AAUSAT3 through 5. CAN also provides a great deal of advantages over interfaces such as serial peripheral interface bus (SPI) or inter-integrated circuit (I^2C).

One of the biggest advantages of CAN is that no individual subsystem is the master. This is ideal for a distributed system which makes for a more redundant system. The standard CAN protocol also comes with a build in error detections and automatic retransmission of corrupted messages.

The CAN protocol can be derived into two sub CAN's known as

- CAN 2.0A
- CAN 2.0B

The difference in them being primarily CAN 2.0B having a 29-bit identifier where the CAN 2.0A only have a 11-bit identifier. In this project the CAN 2.0A will be used since CAN 2.0A already is implemented on the AAUSAT5. It is however still possibly for the two system to coexist on the same bus. The remaining bits on the CAN 2.0B identifier can be neglected if needed.

1.2.1 CAN nodes

Each entry point to the CAN bus is called a CAN node. A CAN node consists of a 4-layered structure as shown in Table 1.1 [Bosch, 1991, p. 5].

Application layer
Object layer
Transfer layer
Physical layer

Table 1.1: Layers of a CAN node

Physical layer

The CAN consists of two wires, CANH and CANL, connected as a differential pair, meaning the sum of the voltages on the two wires will always be the same, i.e. when one goes high, the other goes low [Microchip, 2002, p. 2]. This is done to eliminate common-mode noise.

At each end of the bus, the wiring should be terminated in such a manor that reflections are avoided. The CAN implementation provides only two separate states: A recessive state with high impedance and a dominant state with low impedance. In the recessive state, the bus tends to be half of the rail voltage which can vary depending on the designers choice. In a dominant state, CANH is switched to supply voltage (usually 3,3 V or 5 V) while simultaneously switching CANL to 0 V (ground). A recessive state is only present on the bus if no nodes are in a dominant state [Microchip, 2002, p. 2-3, 7]. A dominant state is equal to a logical 0 in the datastream, and a recessive is equal to a logical 1.

Transfer layer

The transfer layer makes sure the data gets transmitted properly. There are 4 different types of frames[Bosch, 1991, p. 10] that can be transmitted:

Data frame: Carries data from transceiver to receiver

Remote frame: Request a Data frame from the bus with the same identifier as the frame just transmitted

Error frame: Is transmitted if any bus unit detects and error

Overloading frame: Is used to create an additional space between preceding and succeeding Data or Remote frames

The Error and Overloading frames are a little different than the Data frame and Remote frame, and will be explained later.

The format of the Data and Remote frames has seven primary fields:
Start of frame, Arbitration field, Control field, data field, cyclic redundancy check (CRC) field, acknowledge (ACK) field, End of frame.
These seven fields are shown in Figure 1.3:

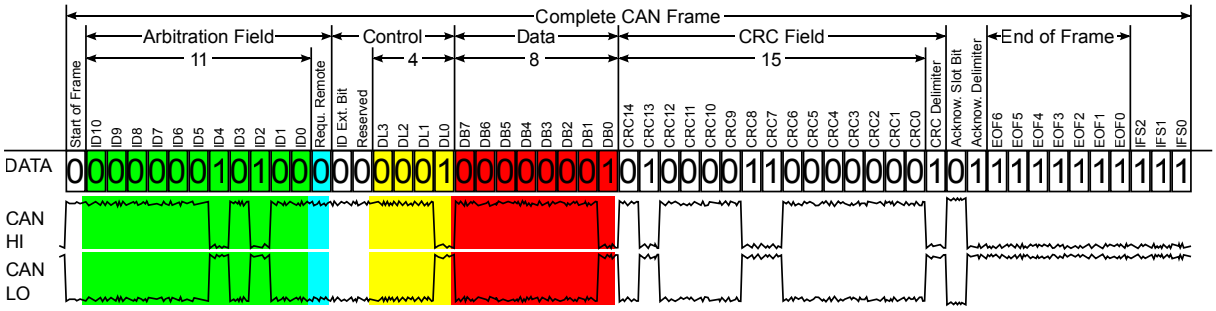


Figure 1.3: CAN frame with electrical properties at the bottom [Wikipedia, 2015]

Field name	Number of bits	Purpose
Start of frame	1	Announces start of frame
Identifier	11	A unique identifier which also states priorities
remote transmission request (RTR)	1	Request a transmission of a frame with same identifier
Reserved bits	2	Not used for anything, but should be dominant(0)
data length code (DLC)	4	Used to describe number of bytes in data field (0-8 bytes)
Data	0 - 64	Data to be transmitted (0 to 8 bytes)
CRC	15	Redundancy check
CRC delimiter	1	Must be recessive(1)
ACK state byte	1	Any receivers can drive dominant(0) to report error
ACK delimiter	1	Must be recessive (1)
End of Frame	7	Must be recessive (1)
Interframe state spacing	3	Must be recessive (1) unless errorframe or Overload frame is present

If two CAN nodes transmit simultaneously, the one with the highest priority wins the transmission. This means that the identifier represents the priority of the message. The priority is defined by the dominant bits. So, if multiple CAN nodes are transmitting at the same time, the one transmitting the dominant bit (logical 0) will "win" the bus. So, if three nodes start at the same time as in Figure 1.4, node 2 will win the bus, since its identifier has the longest stream of dominant bits. [Motorola, 1999, p. 12]. This also means that if two nodes transmit two identical messages it will appear as one.

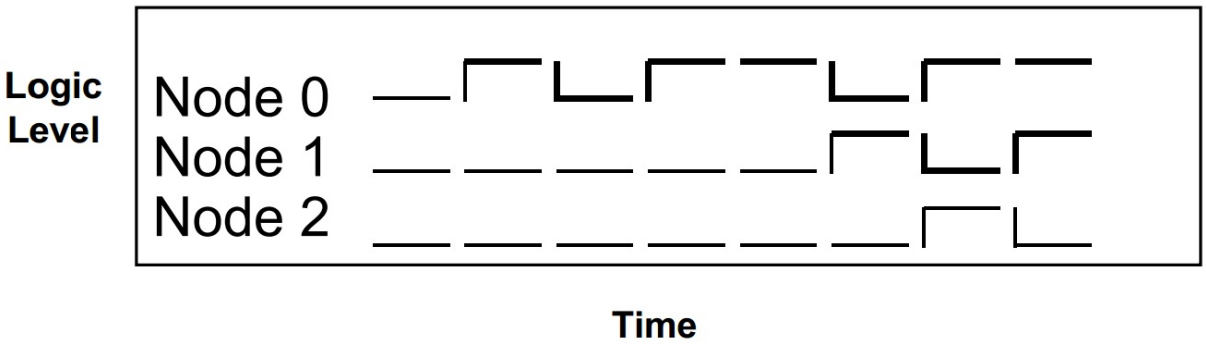


Figure 1.4: Three nodes using the bus simultaneously. Node 2 "wins" the bus since it has the longest stream of dominant bits [Motorola, 1999, p. 12]

Start of frame
The start of frame bit takes the bus from idle mode to transmission mode. It is a single dominant bit which all nodes use for synchronisation. It also sets all other nodes to receive mode [Bosch, 1991, p. 11].

Arbitration field
The arbitration field consists of 11 identifier bits and one RTR bit. The identifier is a unique number that gives the messages its priority. The identifier is transmitted with most significant bit (MSB) first. There must be at least one dominant bit (logical 0) in the eight MSB. The RTR bit determines whether the frame is a Data frame or a Remote frame: A dominant bit equals a Data frame and a recessive bit equals a Remote frame [Bosch, 1991, p. 11].

Control field
The Control field consists of six different bits. The first two bits are reserved and must be dominant (logic 0). The last four bits make up the DLC and contains the length of the Data in bytes, set by recessive bits. In a remote frame this will be 0, and therefore no Data are transmitted [Bosch, 1991, p. 11-12].

Data field
The Data field varies from 0 to 64 bits, i.e. 0 to 8 bytes, and contains the data to be transmitted within the Data frame. This means that no more than eight bytes must be transmitted with each frame [Bosch, 1991, p. 13].

CRC field
The CRC field consists of 15 CRC bits called the CRC sequence, and one CRC delimiter bit which the transmitter must set as recessive. It should be mentioned that the 15 CRC bits are specific for the CAN system. For any other arbitrary system the CRC bits can differ. The CRC sequence is determined by a polynomial division. Because there are 15 bits in the CRC sequence the CRC polynomial should therefore be of the order 15. The denominator polynomium for CAN is:

$$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + x^0$$

Because the highest order of the polynomial is always 1 it is left out and thus the polynomial can be represented as a binary number, based on the coefficients of the polynomial: 0b 0100 0101 1001 1001 or in hexadecimal as 0x4599. The enumerator polynomium, i.e. the polynomium to be divided with the CRC polynomium, is derived from the bit stream from Start of frame to the end of the Data field followed by 15 0's. Each bit matches a polynomial coefficient like the above polynomial. To understand the process of determining the CRC bits better, it is not necessary to think about the bit streams as polynomials but rather as bit strings which are xor'ed together multiple times in a certain way, since an xor operation in some ways is similar to a polynomial division.

When this division is done, only the last 15 bits will have a value greater than 0 [Bosch, 1991, p. 13-14], meaning this field will only be 15 bits long, no matter the number of input bits.

Acknowledge field

The Acknowledge field consists of two bits - one acknowledge bit and one acknowledge delimiter, both of which should be transmitted as recessive [Bosch, 1991, p. 14]. The ACK bit can be set from any nodes and if any nodes sets it the package is discarded regardless which node sets the ACK bit.

End of transmission field

To signal the end of a transmission, seven recessive bits are transmitted. After a Data or Remote frame comes an interframe spacing which can be terminated by an Overload frame [Bosch, 1991, p. 14].

Bit stuffing

Bit stuffing is applied to the parts that does not have a fixed format alias the Data and Remote frames from Start of frame to the end of the CRC sequence. Bitstuffing is not applied to from the CRC delimiter to the End of frame bit since these are of fixed format. Bit stuffing is introduction as an opposite bit if five identical bits either recessive or dominant is present in a row, see Figure 1.5 [Bosch, 1991, p. 22].

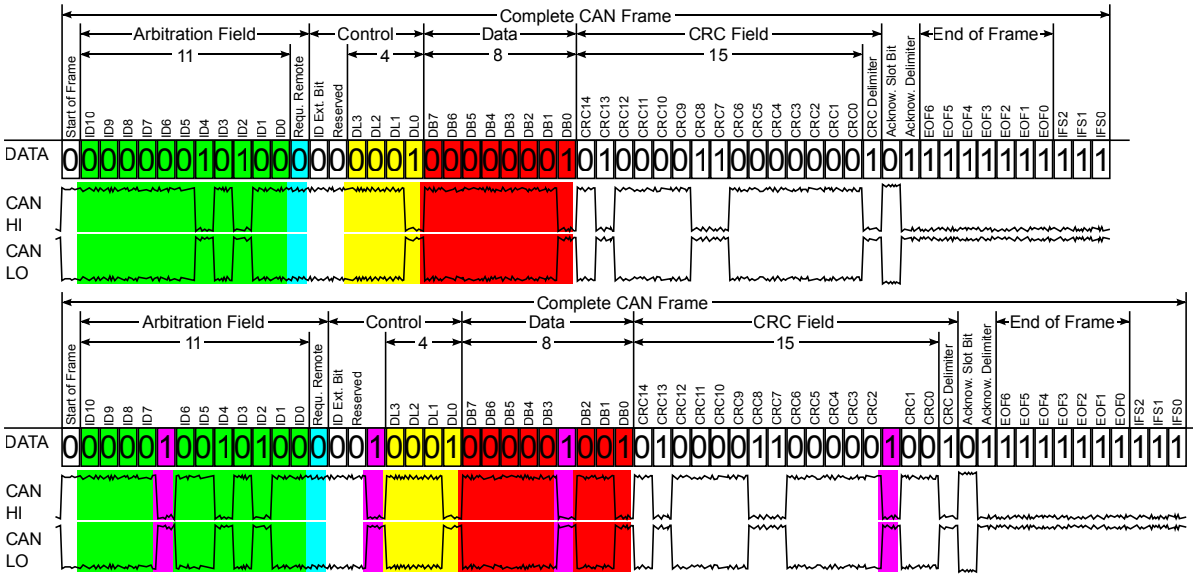


Figure 1.5: A CAN messages before and after bit stuffing [Wikipedia, 2015]

This has two positive implications: First, it ensures that a rising edge occurs regularly, which the receivers use to synchronize. Secondly and error flag violates this rule and thereby if more than five bits are identical, all nodes will know that either the error flag has been set or the end of the frame has been transmitted [Instruments, 2008, p. 7].

Error and Overload frames

An Error frame is transmitted if any of the five error types described in *Fault confinement* is detected. Thus it can be transmitted anywhere between Start of frame and the last bit of End of frame. An Overload frame is transmitted if a node needs more time to process data or for any other reason needs more time between two successive Data and or Remote frames. It is transmitted in the intermission described in *Interframe spacing*.

Both the Error frame and the Overload frame consists of six flag bits and six delimiter bits. In both cases the flag bits should be dominant and the delimiter bits should be recessive. The major difference between an Error frame and an Overload frame is its time of execution as described above. The node which is trying to execute any of these two frames will first transmit its flag and then start transmitting recessive bits until it monitors six recessive bits on the bus. The reason for this is that when a node detects an Error frame or an Overload frame, it will itself begin to transmit one such frame starting at the next bit.
This means that the node transmitting an Error or Overload frame will have to continue to transmit its delimiter until six consecutive recessive bits have been monitored - this ensures that all frames will know that an Error or Overload frame has been transmitted. Given this nature the error flag in an Error frame can vary from 6 to 12 bits seen on the bus as seen in Figure 1.6. Likewise, the overload flag will be 7 bits long seen from the bus as seen on Figure 1.6 [Bosch, 1991, p. 16-18].

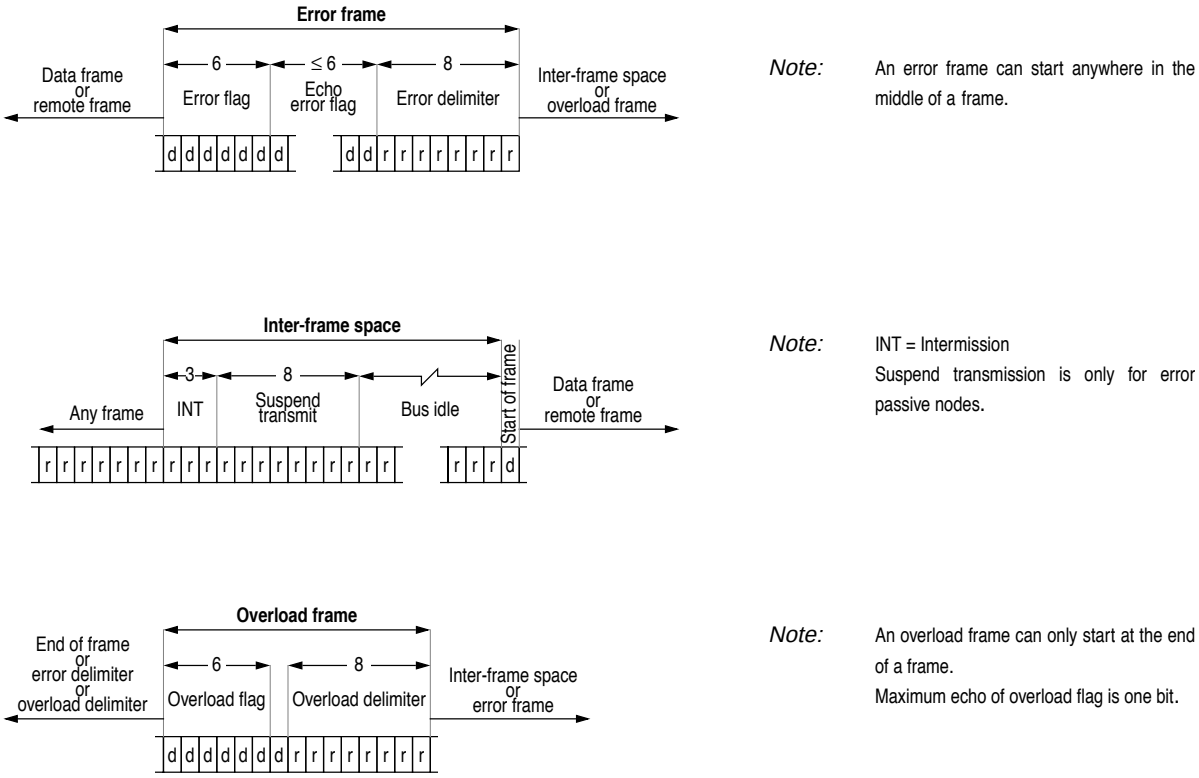


Figure 1.6: Error frame, Overload frame and interframe spaceing [Freescale Semiconductor, 2000]

Interframe spacing
Between two frames a space is introduces to let all nodes catch up and synchronize this space is called interframe spacing. The interframe spacing consists of three parts: Intermission, Suspend transmission and Bus idle. The intermission part consist of three recessive bits and is transmitted by all frames. It is during these bits that an Overload frame can be executed. Note that if a dominant bit is detected on the last of the three intermission bits it is interpreted as a Start of frame bit, but the next bits transmitted by the Overload frame means it will result in a Stuff error.
The suspend transmission part is only present for the transceiver node, also called a passive error node, and consists of eight recessive bits. Bus idle is of arbitrary length and during this time any node can start a transmission. The interframe spacing is shown at Figure 1.6 [Bosch, 1991, p. 18-19].

Error handling
In a CAN transmission there are 5 different kinds of error statements [Bosch, 1991, p. 23]:

Bit error:
The transceiver of a bit monitors a bit that is different than the bit to be transmitted. The exceptions from this error is the arbitration field and the acknowledge bit.

Stuff error:
A stuff error occurs on the 6'th consecutive equal bit in the area that should be coded with bit stuffing.

CRC error:
A CRC error occurs if the receiver calculates a different CRC sequence than the one it receives.

Form error:
A form error occurs when one or more illegal bits have been detected during a fixed form bit field. The only exception to this is the last recessive bit on an end of frame.

ACK error
An ACK error occurs if the transceiver does not detect a dominant bit on the acknowledge bit.

An error frame should be executed as soon as an error occurs. It is possible that more than one type of error can occur together.

Fault confinement
To confine faults from the data line, a unit may be in one of three states:

- Error active
- Error passive
- Bus off

An error active unit may send an active error flag during a transmission. An error passive unit may only send a error passive flag. Also, an error passive unit will wait before sending another transmission, see *Interframe spacing*. A bus off unit may not take part in the communication at all, and thus the output drivers might be switched off [Bosch, 1991, p. 24].

For fault confinement, two counters are implemented: A transmit error count (TEC) and a receive error count (REC). These counters are modified according to 12 rules, see [Bosch, 1991, p. 24-25]. The state of a node depends on those two counters and is shown in Table 1.2.

REC \TEC	< 128	> 127	> 255
< 128	Error active	Error passive	Bus off
> 127	Error passive	Error passive	Bus off
> 255	Bus off	Bus off	Bus off

Table 1.2: Table of node states based on REC and TEC

Object layer

The object layer takes care of message filtering, message handling and status handling. It is also the object layer that provides an interface to the application layer. The scope of the object layer thus includes [Bosch, 1991, p. 4-5]:

- Finding out which messages are to be transmitted
- Deciding which messages received by the transfer layer are to be used
- Providing an interface to the application layer

The object layer is, however, usually implemented in the software controlling the systems using the CAN bus. Fulfilling the scope of the object layer is therefore something that is to be dealt with when implementing CAN, but is not something which the CAN bus sets special rules for the exact implementation of. Therefore, it will not be discussed further in this section.

Summarizing

- The user can send data to the object layer
- The user can request remote frames
- only 8 bytes can be transmitted in each package

Because CAN only support 8 bytes og data per frame the CSP has been used as a layer on top of CAN to transport more data which will be described in the following.

1.3 Cubesat Space Protocol (CSP)

CSP is a communication protocol designed to be used within an embedded distributed system. This system can be a cubesat, meaning that CSP is the protocol that determines how individual subsystems should communicate with each other.

CSP is different from, say, transmission control protocol/internet protocol (TCP/IP), in that CSP is designed for much smaller networks than TCP/IP, which is primarily used for communication via the internet.

By using CSP, the developer of each subsystem does not have to worry about the workings of the other subsystems, since CSP establishes a common interface between all subsystems.

In CSP, all nodes (i.e. subsystems) have their own address, ranging from 0 to 15 (this has been expanded to 32 addresses, but usually only the first 16 are used). Nodes on the ground station also have CSP addresses, with the convention that ground-segment addresses start with a binary 1.
CSP does not support automatic routing, meaning that the topology of the satellite has to be predetermined, so all node addresses are know ahead of time.
The router in CSP can handle a pre-determined number of connections at a time, and each connection has to be free'd after use. [Wikipedia, 2014]

Because of the small size of a cubesat (both physically and in a network context) the handshakes, error detection and -corrections used in TCP/IP are not necessary in CSP since they limit bandwidth databus/spacelink unnecessarily. In this way, CSP can be compared to user datagram protocol (UDP) which also does not use any handshakes nor error correction [Gomspace, 2015, p. 5]. However, in CSP, a more reliable protocol also exists called reliable data protovol (RDP) which includes three-way handshakes, packet re-ordering, retransmission etc. This protocol is not used in AAUSATs however, because this handshake is not necessary for internal communication.

CSP compiles on AVR-8, AVR-32, ARM and PC, and supports the following network interfaces: I^2C, CAN, RS.232 [Gomspace, 2015, p. 2].
In the satellite the CAN is used which only allows for 8 byte transfer per package where CSP can have up to 250 bytes in a frame. Therefore a frame is broken into smaller packages before being transmitted and stitched together again in the receiver.

Note that all CSP should be big-endian [Gomspace, 2015, p. 9] - ARM/AVR32 systems are typically big-endian, whereas PC's are little endian, so some kind of conversion has to be made.

The CSP header
Two versions of the CSP header exist - version 0.9 and 1+, released in 2010. Here, version 1 will be used. The CSP header can be seen on Table 1.3. CSP 1+ support package to a maximum length of 84 data bytes, more than that and it needs to be send in multiple packages.

As can be seen, the header contains information about the priority of the package, source and destination as well as the port of the source and destination. However, merging this information to form the CSP header should be performed by the CSP protocol. Nonetheless, the destination

and destination port should be specified when setting up the CSP connection, together with the source and source port, so the CSP header can be constructed correctly.

Bit offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Priority		Source					Destination					Destination Port						Source Port						Reserved				HMAC	XTEA	RDP	CRC
32	Data (0 – 65535 bytes)																															

Table 1.3: The CSP Header, version 1 [Wikipedia, 2014]

To setup CSP on a system, the CSP buffer system device drivers and router core has to be initialized. Here, the buffer has to be declared, with respect to buffer size in number of packages, and maximum package size. Also, the interface has to be specified, for instance CAN. Finally, the routing has to be setup, including the router stack size [GomSpace, 2014]. A deeper explanation of this setup will not be given. When the setup is done, CSP is ready to be used to transmit data between a client and a server.

Hello world

A simple "Hello world"-example code, in which a client transmits a string to a server using CSP, can be seen in Listing 1.1 and Listing 1.2.

On the client side, a connection and a package has to be made. The connection is set up to connect to the server, using the server address and port known in advance, along with a timeout period, see line 3 in Listing 1.2.

Then, the string is loaded into the packet, along with the length of the package. Afterwards the package is sent via the connection after which the connection is closed.

On the server side, see Listing 1.1, a connection is made that binds to the the socked specified by the servers address and port. Then, the server enters listen mode, where it listens on the specified socket. When a package is recieved, it is printed, the buffer is free'd and the connection is closed. It shows that the CSP protocol can function on a minimum of resources just as long the buffer is continuously free'd.

```
1  csp_conn_t * conn;
2  csp_packet_t * packet;
3  csp_socket_t * socket = csp_socket(0);
4  csp_bind(socket, PORT_4);
5  csp_listen(socket, MAX_CONNS_IN_Q);
6  while(1) {
7      conn = csp_accept(socket, TIMEOUT_MAX);
8      packet = csp_read(conn, TIMEOUT_NONE);
9      printf("%S\r\n", packet->data);
10     csp_buffer_free(packet);
11     csp_close(conn);
12 }
```

Listing 1.1: Server side of Hello World

```
csp_conn_t * conn;
csp_packet_t * packet;
conn = csp_connect(PRIO_NORM, SERVER, PORT_4, TIMEOUT, 0);
packet = csp_buffer_new(sizeof(csp_packet_t));
sprintf(packet->data, "Hello World");
packet->length = strlen("Hello World");
csp_send(conn, packet, TIMEOUT_NONE);
csp_close(conn);
```

Listing 1.2: Client side of Hello World

Note that the example shown above is a minimum-working example. In reality, multiple checks would be performed on both client and server-side, to ensure that the data exists and gets transmitted and received correctly. There is also a limit to how long packages can be sent over CSP as determined by the buffer size, which has to be handled when transmitting over CSP. Also, the client side of the code would be implemented as a function, so the user would only have to call a function *void send_packet(void)* to send the package that has previously been put in the CSP buffer. The same goes for the server, where the code would be implemented as a *void csp_task(void *parameters)*, to create a server task that listens for incoming packages [GomSpace, 2014].

Obviously, there is more to CSP than has been shown here, but this brief run-down of the overall functionality should provide the reader a good overview of the capabilities and possible implementations of CSP.

2 | Camera

In this chapter, the workings of a camera will be discussed, from the light hitting the lens to the final output picture. This chapter will only discuss the camera in general, while the considerations that must be done in regards to the implementation in the satellite will be discussed in chapter 4. The signal way through a camera from light to final picture can be seen in Figure 2.1.

Figure 2.1: The signal way through a camera

2.1 The Optical System

A camera is an optical device that records images which can be stored directly. Most cameras use a similar design: Light from the visible spectrum enters an enclosed box through a piece of optics and an image is captured on a light-sensitive medium (image sensor).

The optical systems main physical element is the barrier between the incoming light from the surroundings and the light-sensitive medium. The optical systems barrier consist of a piece of glass (or more); this is also called a lens. A lens focuses the light from the surroundings onto the image sensor. In Figure 2.2 an illustration is displayed of an object (to the left of the lens) with two points, each point reflecting three light beams. When the three light beams collide with the lens surface, each beam with a different angle, the lens will "bend" the three beams towards the same location. An image of the object is formed at that location (to the right of the lens), and by placing image sensors at exactly this position, a picture can be captured. As seen on the figure, the picture captured is inverted, this has to be considered when extracting the image from the sensor [Moeslund, 2012].

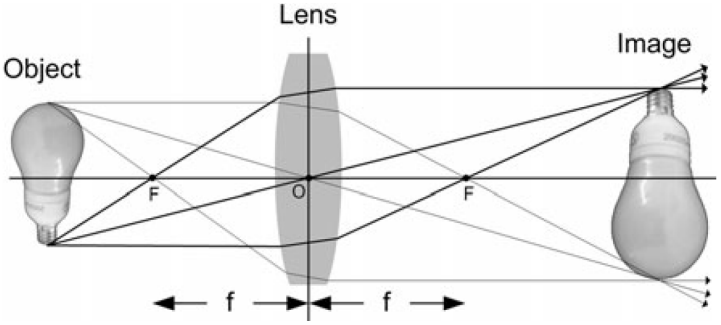
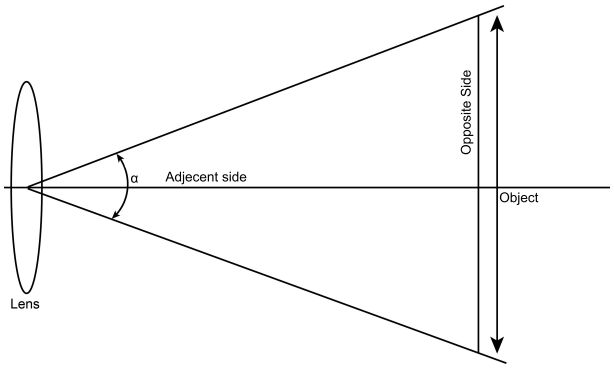


Figure 2.2: The light beams from the object is focused by the lens. The light bulb to the left of the lens is the real object and the inverted light bulb to the right of the lens is the image created [Moeslund, 2012].

The point F where the beams intersect with the parallel line in the figure (the optical axis) is called the focal point. The distance from the focal point to the center of the lens (the optical center O) is called the focal length. The focal length determines the distance in which beams are brought to a focus. If an optical system has a short focal length it bends the beams with a high angle, and with a longer focal length, the beams are bent with a smaller angle. The size of the image and the focal length is proportional. When the focal length is increased, the image will become larger - this is called optical zoom [Moeslund, 2012]. In practice, optical zoom is adjusted by rearranging the optics. Thus it is necessary to have movable parts to adjust the optical zoom, if a different image of the observable world has to be taken instead of the images taken with the optics original settings. Another method of enlarging an image is called digital zoom. Digital zoom crops the resulting image and enlarges the resulting image individual pixels, thus reducing overall quality.

The part of of the observable world that is desired to be captured on the image sensor, spans a certain angle as seen from the camera - this is called the field of view (FOV) or the angle of view. In Figure 2.3a the observable world is the object (to the right of the lens) and the field of view is the angle α made by the two separating lines (at the lens). This means that a smaller field of view results in a smaller section of the observable world being seen on the image sensor - the reverse is true for a wider field of view.



(a) Here the Field of View α of a camera can be seen

Focal length [mm]	Field of view [°]
14	114
28	75
35	63
50	46
85	28
100	24
200	12
600	4

(b) The relationship between commonly used focal lengths and their field of view of a camera having a sensor size of 24mm $\times$ 36mm. (full frame)

Figure 2.3: Figure 2.3a shows trigonometric correlation between focal length and Field of view. Figure 2.3b shows the field of view in correlation with standard photo objective.

The field-of-view depends on the focal length and the physical size of the image sensor. If the image sensor has an aspect ratio of 1:1 the field of view in both the horizontal and vertical direction is similar. Typically, the image sensor in cameras has a rectangular aspect ratio. Equation 2.1 and Equation 2.2 displays the formula for calculating field-of-view for the horizontal and vertical axis respectively.

$$FOV_x = 2 \cdot \tan^{-1}\left(\frac{\text{width of sensor} \cdot 0.5}{f}\right) \tag{2.1}$$

$$FOV_y = 2 \cdot \tan^{-1}\left(\frac{\text{height of sensor} \cdot 0.5}{f}\right) \tag{2.2}$$

Where:

FOV_x	is the horizontal FOV angle [°]
FOV_y	is the vertical FOV angle [°]
f	is the focal length [m]

The two equations displays that by changing the focal length and the aspect ratio of the image sensor, the desired field of view can be achieved.
Looking a little deeper into the camera the light is recorded on an image sensor, this will be described in the following.

2.2 The Image Sensor

The light bent by the optics has to be recorded, which is done with an image sensor. An image sensor consists of cells each referred to as a pixels, which are placed in a 2D-array. The number of pixels in each axis, together with the size of each pixel, determines the size of the sensor. When capturing an image, the pixel is able to measure the amount of incoming light that hits the pixel by converting the light to a voltage which is then converted to a digital value. Therefore, a pixel exposed to a larger amount of light will generate more voltage and be converted to a bigger digital number than a pixel exposed to less light. The number of pixels used in the image sensor defines the resolution of the camera [Moeslund, 2012].

Each pixel voltage is converted to a digital value, with a analog to digital converter (ADC). The bit depth of the image tells how many shades of gray exist between total black and total white. Usually, 8 bits are used (but up to 14 bits are used by some moderne high-end cameras), to represent the light intensity for each pixel. This way, the incoming light intensity of each pixel can be represented as a digital value ranging from 0 to 255, as shown on Figure 2.4.

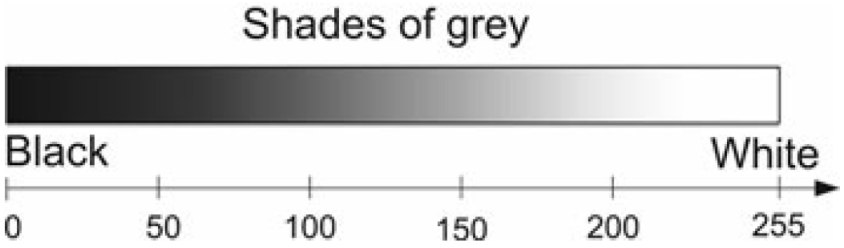
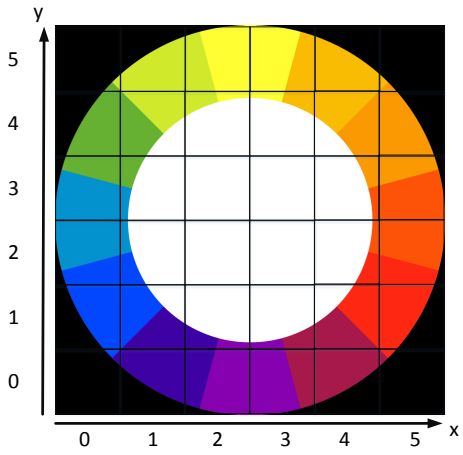
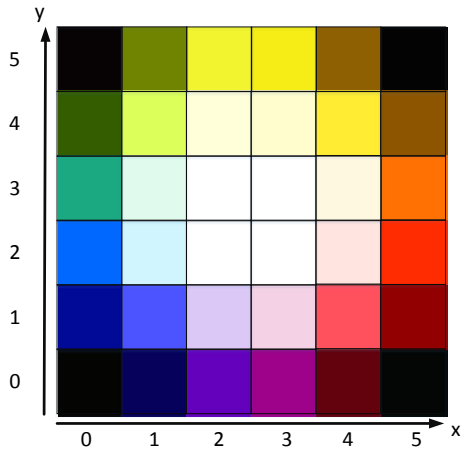


Figure 2.4: The connection between the digital values and the shades of grey [Moeslund, 2012].

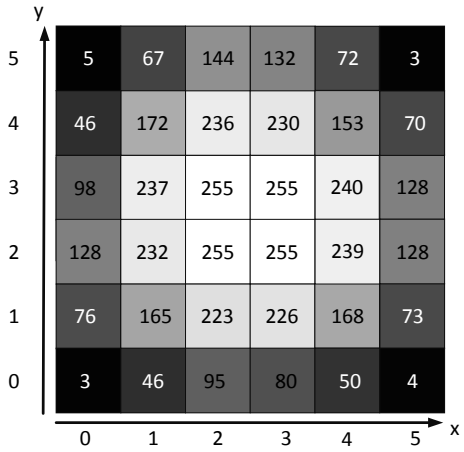
On Figure 2.5a, an image is overlayed with a 6×6 image sensor, with each pixel shown as a square. The average color hitting each of the 36 pixels is shown on Figure 2.5b. Figure 2.5c is showing the light intensity hitting each pixel, displayed as an 8-bit digital number [Moeslund, 2012].



(a) The original image shown on a 6×6 pixel sensor [Wikimedia, 2015]



(b) A 6×6 pixelated array of Figure 2.5a.



(c) An image similar to Figure 2.5b, showing the light intensity hitting each pixel, converted to an 8-bit digital value

Figure 2.5: An example of an image converted into pixels and light intensities

If the camera or the object is in motion, the time in which the pixels receives light from the object needs to be limited, to avoid blur. This is done by using a shutter which is a device that allows light to pass a certain amount of time. The amount of time the shutter is open is called the exposure time. Additionally, the exposure time also has to be adjusted in order to avoid the pixels in the image sensor to become overexposed or underexposed by light [Moeslund, 2012]. There exist different kinds of image sensors, the two biggest is complementary metal-oxide-semiconductor (CMOS) and charge-coupled device (CCD), these will be described in the following.

2.2.1 CMOS vs. CCD sensors

CCD and CMOS rely on two different sensor technologies. They both convert the incoming light to a voltage as described earlier [Moeslund, 2012]. The difference is how they read the accumulated charge of each cell/pixel. A CCD sensor transports the charge across to one corner of the chip. Then, in the corner, it converts the charge to a digital value with an ADC [Waltham, 2015]. The CMOS sensor uses several transistors at each pixel to amplify and move the charge on the CMOS. By doing this, the CMOS can read each pixel individually with an ADC[Howstuffworks, 2015].

There is a range of both positive and negative sides to both CCD's and CMOS sensors:

- **CCD**
  - \+ Higher image quality than a CMOS
  - \+ Low-noise images compared to a CMOS
  - \+ Higher pixel density (more pixels per area) than a CMOS
  - − Consumes a lot more power than a CMOS
  - − Expensive compared to CMOS sensor
- **CMOS**
  - \+ Consumes a lot less power than the CCD
  - \+ Inexpensive compared to CCD
  - − More sustainable to noise than CCD
  - − Lower light sensitivity than CCD
  - − Reads each pixel individually

The CCD use a special process to transport the accumulated charge from each pixel to the corner of the sensor with minimum distortion [Waltham, 2015]. This makes the CCD a higher quality when speaking of fidelity and light sensitivity. Furthermore, the pixel density (pixels per area) is higher because CCD's do not have a transistor located at each pixel[Howstuffworks, 2015].

The CMOS consumes significantly less power and is a lot less expensive compared to the CCD sensor, but lacks the quality of the CCD [Howstuffworks, 2015].

So forth only a grayscale version of the image have been explained. In the following the color part of the image will be described.

2.3 Color image detection

To obtain an image, luminance levels for each pixel in the imaging sensor must be read. Depending on the sensor this can be done rather simply when a monochrome image is needed, since it is only the light intensity at each pixel that need to be know.

For a color image things are different. A color image consist of information about light intensity and the light frequency (i.e. the color).

Having to measure the intensity and frequency of light in a high-megapixel sensor array is not an ideal solution, due to the amount of data such a signal would contain.

Instead, a different approach is typically used. The color information of each pixel is quantized into the three primary colors, named *channels*, namely the *red, green* and *blue* channels, thus forming what is know as the RGB color space. Each channel has a certain intensity, whose resolution is dependent on the bit depth of the sensor. Typically, 8 bits are used per channel, meaning $2^8 = 256$ different intensities are available per channel.

The three channels can be see as the orthonormal basis vectors of a 3-dimensional color space (the RGB color space). In this color space, any color can be expressed as a linear combination of the 3 primary colors, whose coefficients can be any integer from 0 to $2^8 - 1 = 255$, see Equation 2.3.

$$Color = (0 \dots 255) \cdot \bar{R} + (0 \dots 255) \cdot \bar{G} + (0 \dots 255) \cdot \bar{B} = \begin{bmatrix} L_R \cdot R \\ L_G \cdot G \\ L_B \cdot B \end{bmatrix} \tag{2.3}$$

Where:

L_N is the luminosity in the luminosity in the N'th channel $[\cdot]$

Now the challenge is how these three color-intensities are obtained for each pixel. The most common way to register the color of each channel, is to coat the pixel with a filter of that color. This means, that only light with that color can reach the sensor (i.e., only red light can penertrate a red filter), meaning that the light intensity of that color is measured.

So, pixels of each of the 3 primary colors are needed, to measure the light intensities of each channel. This can be done in a number of ways, for instance by having 3 sensors - each coated with a filter of each color, and then use mirrors to guide the light to each sensor. However, a more practical and cheaper alternative, which is also the most common way to get the light information, is by using a *bayer filter*, as shown on Figure 2.6.[Moeslund, 2012, p. 28]. The sensor thus consists of rows of pixels - one row with green and red pixels interchanged, and the other row with blue and green pixels interchanged.

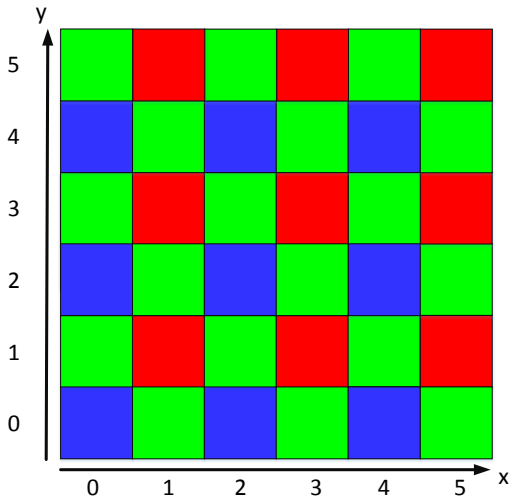


Figure 2.6: The bayer arrangement of color filters on an image sensor

The reason why there are twice as many green pixels as red and blue respectively, is that the human eye is more sensitive to green than the other colors [Moeslund, 2012, p. 28]. This method however has a problem, because only one color is measured in each pixel a lot of information is missing. This problem will be attended in the following.

2.4 Demosaicing

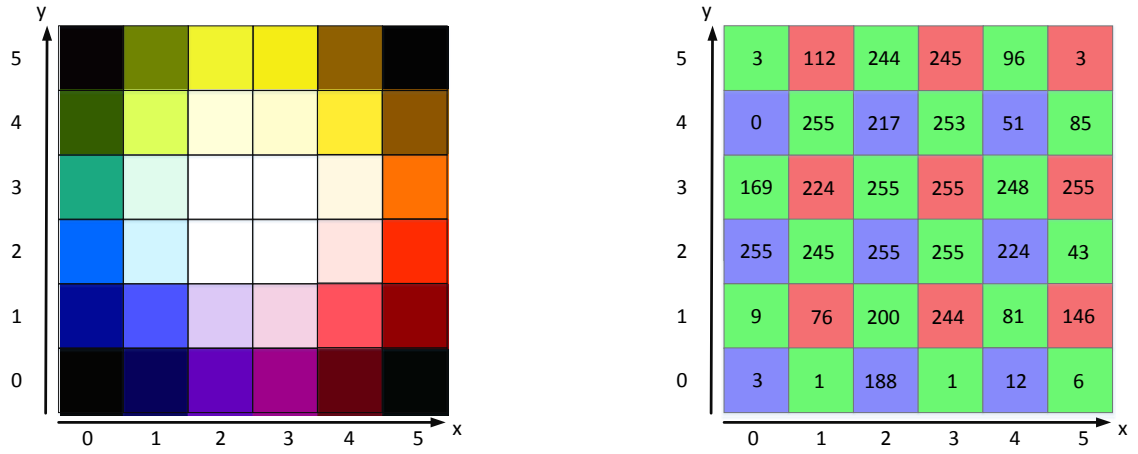
To form a full color image, it is necessary to estimate the missing R, G and B values for each pixel, so that a full RGB image can be made, i.e. the level of green light hitting the blue pixels, etc. has to be estimated. This is called *demosaicing*, since it removes the mosaic pattern formed by the Bayer filter. The algorithms used for demosaicing are usually called *interpolation* algorithms. Interpolation is usually defined as the process of increasing the resolution of an image, which is partially done here, since two color channels are generated per pixel when demosaicing. Therefore, in this context, demosaicing and interpolation are used interchangeably, meaning to recreate all three color channels for all pixels.

A variety of demosaicing algorithms exist, of which the most common ones of which will be mentioned here.

A demosaicing algorithm should, among others, have the following traits [Rani og Hans, 2013, p. 859]:

- Avoid the introduction of undesired color artefacts, in the form of colored fringes around the edges of the photo
- Maintain resolution as high as possible
- Low computational complexity to maximize speed and minimize power consumption

Some of the typical undesired artifacts resulting from demosaicing are blurring and the generation of false color, which is called color aliasing [Alleysson et al., 2002, p. 1]. In the following, the same image which has been used earlier is used to illustrate the effect of various demosaicing algorithms. The original 6×6 image is shown on Figure 2.7a, while the R, G and B values as registered by the pixels in the Bayer filter is shown on Figure 2.7b.



(a) An example of measured RGB values in a bayer filter image sensor

(b) The complete RGB image after nearest neighbor interpolation

Figure 2.7: The values of a 4x4 image before and after nearest-neighbor interpolation

Nearest neighbor

Nearest neighbor interpolation is the simplest type of interpolation. Here, the RGB values of a certain pixel are approximated by simply setting their value equal to the RGB value of their nearest neighbor.

In a Bayer filter, the distance from pixel to pixel are all equal, and thus there are many nearest neighbors to each pixel. Therefore, when designing an algorithm to produce this interpolation, one chooses which neighbor to copy the color value from. Also, because of the Bayer filter, the red value of a pixel is copied from one of the nearby red pixels, whereas the blue value is copied from one of the blue pixels. This principle can be seen in Equation 2.4 [Moeslund, 2012, p. 29], which states how to peform the nearest neighbor interpolation from a mathematical point of view.

$$g(x,y) = \begin{cases} [R,G,B]_B = & [f(x+1,y+1) & ,f(x+1,y) & ,f(x,y)] \\ [R,G,B]_{GB} = & [f(x,y+1) & ,f(x,y) & ,f(x-1,y)] \\ [R,G,B]_{GR} = & [f(x+1,y) & ,f(x,y) & ,f(x,y-1)] \\ [R,G,B]_R = & [f(x,y) & ,f(x-1,y) & ,f(x-1,y-1)] \end{cases} \tag{2.4}$$

Where:

f(x,y)	is the input image (Bayer pattern)
g(x,y)	is the output RGB image
$[R,G,B]_B$	is the function to be used for pixels sensitive to blue light
$[R,G,B]_{GB}$	is the function to be used for pixels sensitive to green light followed by a blue pixel
$[R,G,B]_{GR}$	is the function to be used for pixels sensitive to green light followed by a red pixel
$[R,G,B]_R$	is the function to be used for pixels sensitive to red light

As an example, say one wants to calculate the RGB values for the pixel at (1,1), see Figure 2.7a. This is done as shown in Equation 2.5.

$$\begin{aligned} g(1,1) &= [R,G,B]_R \\ &= [f(x,y),\ f(x-1,y),\ f(x-1,y-1)] \\ &= [R_{1,1},\ G_{0,1},\ B_{0,0}] \\ &= [17,9,16] \end{aligned} \tag{2.5}$$

If this is done for the entire image, the image shown in Figure 2.8 is obtained.

The advantage of the nearest neighbor method is it's low complexity and thus it's computational speed. The disadvantage is that the resolution of the image is not increased. Instead, one could say that the pixels just increase their size, since the pixel values are just copied to nearby pixels. This means that the end result is not of very good quality, nor very smooth, which is not desirable.

Bilinear interpolation
Based on Equation 2.4, one could say that the nearest neighbor method is a one-dimensional interpolation, since only one pixel of each color is used to determine the RGB values of a certain pixel. Bilinear interpolation on the other hand, uses 2 pixels per channel to determine the RGB values. This approach can be implemented easily, simply by taking the average of the nearby pixels of a certain color.

Two different formulas are needed for bilinear interpolation - one for when then four horizontal and vertical neighbors are known, see Equation 2.6, and one for when the diagonal neighbors are known, see Equation 2.7 [Rani og Hans, 2013, p. 859].

$$f(x,y) = \Big(f(x-1,y) + f(x,y-1) + f(x,y+1) + f(x+1,y)\Big)/4 \tag{2.6}$$

$$f(x,y) = \Big(f(x-1,y-1) + f(x-1,y+1) + f(x+1,y-1) + f(x+1,y+1)\Big)/4 \tag{2.7}$$

Note that, in the instance that only some of the neighboring pixels are known, the unknown pixels are just removed from the equations.

So, to find the B- and G-values of the red pixel at (1,1), see Figure 2.7a, the nearby blue and green pixels are all taken into account:

$$B_{1,1} = (B_{0,0} + B_{0,2} + B_{2,0} + B_{2,2})/4 \tag{2.8}$$
$$G_{1,1} = (G_{1,0} + G_{0,1} + G_{1,2} + G_{2,1})/4 \tag{2.9}$$

To find the B-channel at (2, 1), see Figure 2.7a, the fact that the horizontal blue values are not known, results in the following equation:

$$B_{2,1} = (B_{2,0} + B_{2,2})/2 \tag{2.10}$$

Bilinear interpolation can be implemented by the use of *kernels*. A kernel is a $n \times n$ matrix, which, through a mathematical expression, typically matrix multiplication, is applied to the part of the picture. The kernel is then moved throughout the frame, so the kernel gets applied to all parts of the image. In a software implementation of the demosaicing algorithm, this is more efficient due to the nature of the PC and it's computing capabilites. A kernel has a radius, R, describing it's size. A 3×3-matrix has radius $R = 1$, a 5×5-matrix has radius $R = 2$ etc.

To interpolate the green channel, the following kernel is used[Lukac, 2008, p. 224]:

$$F_G = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} /4 \tag{2.11}$$

To interpolate the red and blue channel, the following kernel is used [Lukac, 2008, p. 224]:

$$F_{RB} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} /4 \tag{2.12}$$

The kernel can be understood as a convolution filter that gets applied on the entire image when it is moved throughout the frame [Lukac, 2008, p. 224]. Thus, the operation can be written as:

$$g(x, y) = h(x, y) * f(x, y) \tag{2.13}$$

Where:

- $f(x, y)$ is the input image
- $h(x, y)$ is the kernel
- $g(x, y)$ is the resulting image

The convolution of two discrete functions f and g, at index $[n]$ is defined as [Oppenheim og Schafer, 2010, p. 159]:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] \cdot g[n - m] \tag{2.14}$$

If this definition is applied to each channel of the image with it's corresponding kernel (thus seeing each channel as a seperate image with the unknown pixels defined to have the value 0), the resulting RGB-channels can be found using Equation 2.15 [Moeslund, 2012, p. 76].

$$\begin{aligned} R(x,y) &= \frac{1}{4} \cdot \sum_{i=-R}^{R} \sum_{j=-R}^{R} F_{RB}(i,j) \cdot R(x+i, y+i) \\ G(x,y) &= \frac{1}{4} \cdot \sum_{i=-R}^{R} \sum_{j=-R}^{R} F_{G}(i,j) \cdot G(x+i, y+i) \\ B(x,y) &= \frac{1}{4} \cdot \sum_{i=-R}^{R} \sum_{j=-R}^{R} F_{RB}(i,j) \cdot B(x+i, y+i) \end{aligned} \tag{2.15}$$

Where:

C(x,y)	is the C'th color channel (R, G or B) at the position (x, y)
R	is the radius of the kernel, here $R = 1$
F_x	is the kernel used for the C'th channel

The results from this method are smoother than those from the nearest neighbor method, at the cost of additional computation. Because of this, the bilinear interpolation method is a commonly used demosaicing algorithm.

Using Matlab, the bilinear interpolation has been performed on the image from Figure 2.7a, using the kernels in Equation 2.11 and Equation 2.12. The resulting image can be seen on Figure 2.8.

Bicubic interpolation

While bilinear interpolation can be understood as a linear interpolation since it only looks at the "first degree" of nearby pixels, this idea can be expanded to become bicubic, meaning that "second degree" pixels are included in the approximation of pixel values. This means that, in order to determine the green pixel at say (1,1), instead of only looking at the green pixels surrounding it (i.e. touching it), the pixels surrounding these neighboring pixels are taken into consideration as well [Rani og Hans, 2013, p. 5].

As an example, the G-value at (3,3) is calculated in Equation 2.17. Note that indices reaching outside the image are simply set to 0.

$$\begin{aligned} G_{3,3} &= \frac{G_{2,3} + G_{3,2} + G_{3,4} + G_{4,3}}{4} + \frac{G_{0,3} + G_{3,0} + G_{6,3} + G_{3,6}}{8} \\ &= \frac{255 + 255 + 248 + 253}{4} + \frac{169 + 1 + 0 + 0}{8} && (2.16) \\ &= 255 && (2.17) \end{aligned}$$

This interpolation algorithm can, just like bilinear interpolation, be implemented using kernels. Here, the same kernels as the ones in bilinear interpolation. However, the kernel's radius is increased so it includes the pixels that are 2 pixels away from the pixel whose value is to be determined. The outermost pixels of this kernel are weighed less than the innermost, since it is still the pixels nearest to the pixel to be determined who have the most affect. It has not been possible to obtain a bicubic interpolation kernel, but one ha been predefined in Matlab, based on the bilinear kernels, to be able to perform the bicubic interpolation. The result of this interpolation can be seen on Figure 2.8. As can be seen on the figure, the bicubic interpolation causes a checkerboard-like pattern on the image. The exact cause of this pattern is not known, but it is most likely due to a incorrect implementation of the algorithm. The pattern can, however, be reduced by using a mean filter, which simply blurs the photo, by averaging the pixels. The mean filter is implemented by using the kernel shown in Equation 2.18 [Moeslund, 2012, p. 76], and the resulting image is shown on Figure 2.8.

$$F_{mean} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} /9 \tag{2.18}$$

Mathematically speaking, this is equal to using the derivatives of the color channels around each pixel. This is because the difference between the pixels surrounding the current pixel and the pixels surrounding these, can be seen as the change in color value, i.e. the derivative. As one knows from piecewise functions, the piecewise function is smooth when the derivative of the two functions are equal at their intersection point. The same principle applies here: To ensure that the resulting image is smooth, the derivative is used.

The bicubic interpolation is smoother than bilinear, but requires more computational force to use. It produces good results when applied to a grayscale image, but still shows some artifacts [Rani og Hans, 2013, p. 9].

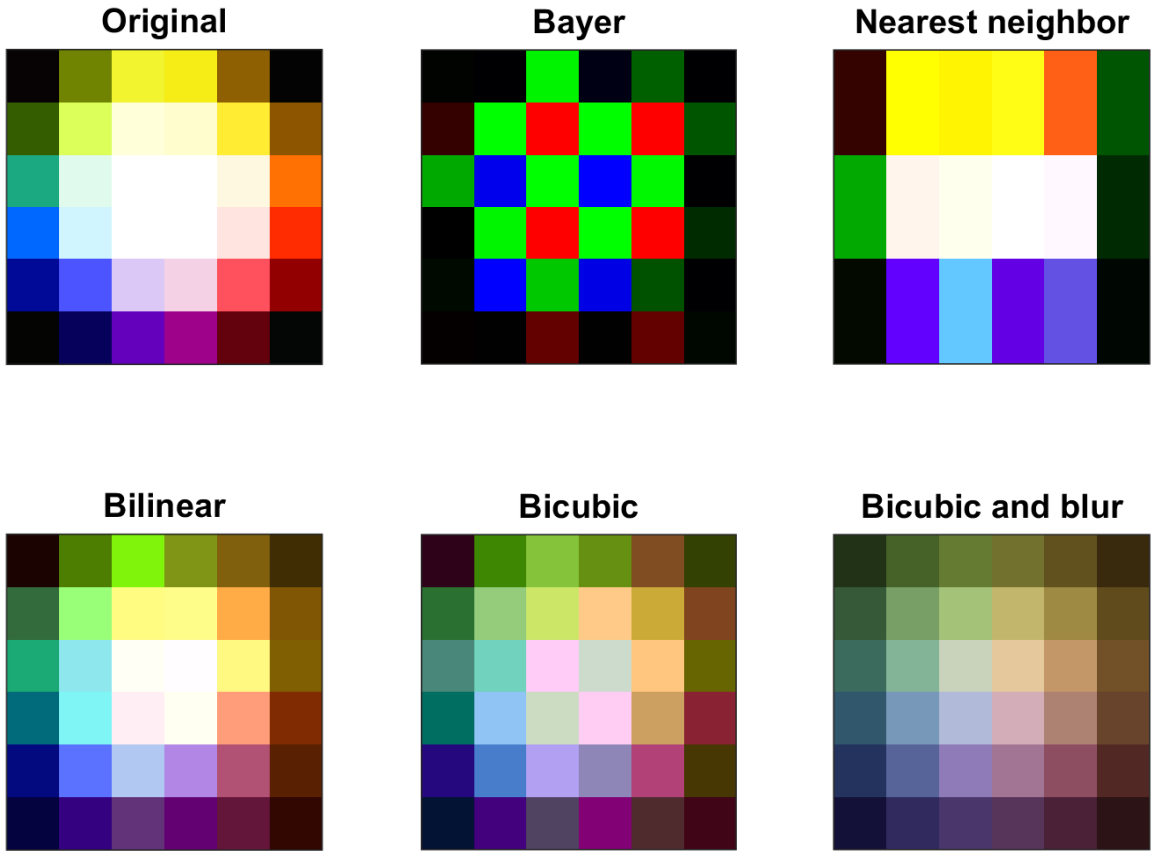


Figure 2.8: The original 6 × 6 px color image **(top left)** and the image after the Bayer filter **(top center)**. Also shown are the demosaiced images using the interpolation algorithms mentioned: Bilinear **(bottom left)**, bicubic **(bottom center)** and bicubic together with a mean filter (blur) **(bottom right)**.

An example of the demosaicing algorithms applied to a real image is shown on Figure 2.9, where the algorithms are applied to a 500 × 500 px image, from which a 75 × 75 px crop is shown.

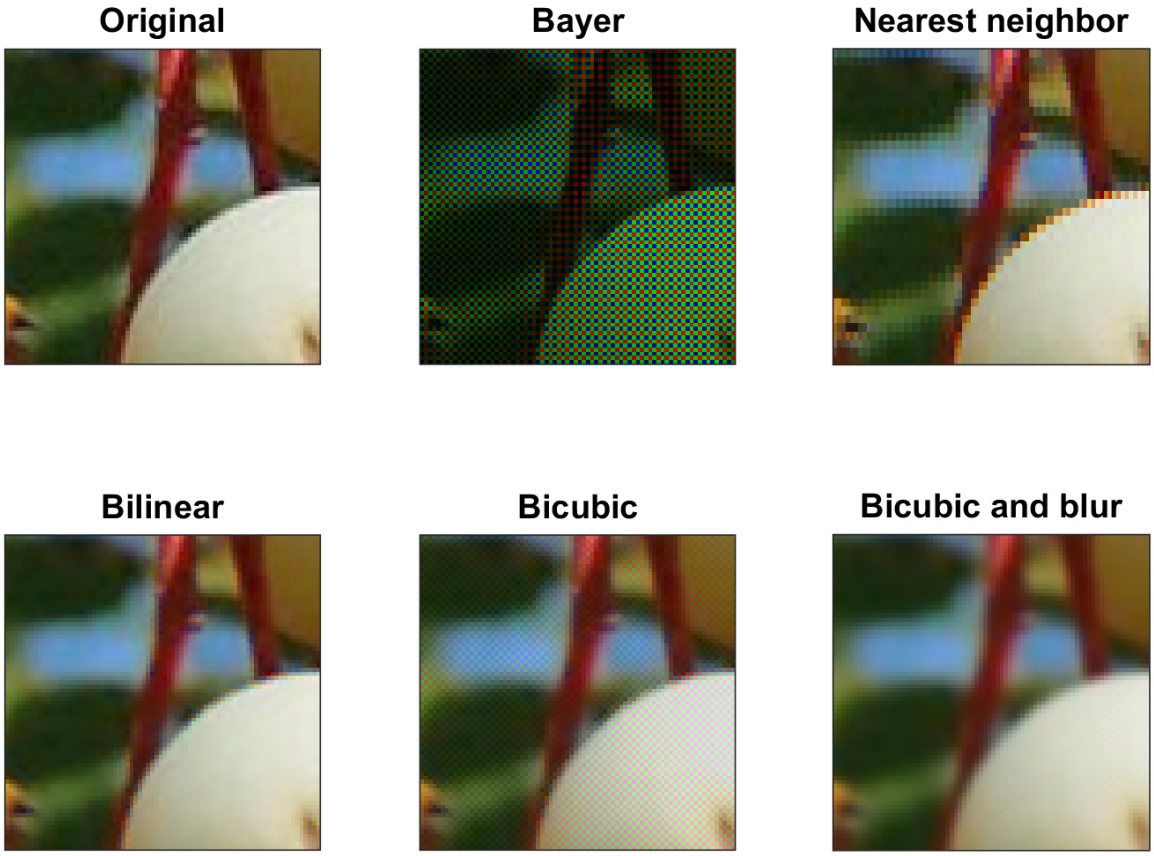


Figure 2.9: A 75×75 px crop of the original 500×500 px image **(top left)** and the image after the Bayer filter **(top center)**. Also shown are crops of the demosaiced images using the interpolation algorithms mentioned: Bilinear **(bottom left)**, bicubic **(bottom center)** and bicubic together with a mean filter (blur) **(bottom right)**.

From Figure 2.9 it is clear that nearest neighbor interpolation produces a very edgy images, with many artifacts. The bicubic interpolation produces smoother result than nearest neighbor and bilinear, but just like in Figure 2.8, it produces an undesired checkpattern througout the image, which can be reduced by using a mean filter. Overall, it seems as if it is the bilinear demosaicing algorithm that performs the best, but this is primarily because of the undesired checkboard pattern appearing in the bicubic interpolation. All the kernels will automatically make any undefined variables, i.e. the edges, into a black pixel and preform the interpolation as the individual procedure describes.

Other methods

A wide variety of other demosaicing methods exists. One method is to move the image into the frequency domain by means of the Fourier transform. Before going into the frequency domain, the image is split into 2 pars - a luminance part, and a chrominance part. The luminance determince the black and white properties of the image, i.e. how bright and dark each pixel is. The chrominance determines the color and color intensity of each pixel.
The advantage of this method is that the luminance of each pixel can be determined accurately,

simply as the light intesity of each pixel, regarding it's color. It is therefore just the chrominance that needs to be interpolated. This can be done in the frequency domain, because the chromiance and luminance parts are split up in the frequency domain. If the splitting is done perfectly, it would be possible to restore the chrominance perfectly, but this is not possible in real images. This is what leads to the artifacts seen in demosaicing; the aliasing between the luminance and chromiance in the frequency domain, which means the chrominance cannot be interpolated perfectly. [Alleysson et al., 2002]

Another method is based on the observation that hue (i.e. color) does not change abruptly - it changes rather smoothly over a small area. Using this method, the green channel is interpolated using bilinear interpolation. Then, the ratios R/G and B/G are calculated for each of the known pixels, and the red and blue channels of the missing pixels can then be estimated using this ratio, simply by multiplying the ratio with the green value of the current pixel. [Lukac, 2008, p. 224] In The End it comes down to what is on the image. A Picture containing more or less the same pixel, black, white etc. would not become a problem for any of the demsaicing. When concerning about what method would be best it comes to the fact that depends on the situation, so making the demosaicing filters interchangeable would be preferable.

For example, the blue pixel at (1,1), see Figure 2.7, can be found as:

$$\begin{aligned} B_{1,1} &= G_{1,1}\frac{\frac{B_{0,0}}{G_{0,0}} + \frac{B_{0,2}}{G_{0,2}} + \frac{B_{2,2}}{G_{2,2}} + \frac{B_{2,0}}{G_{2,0}}}{4} \\ &= 114\frac{\frac{3}{3} + \frac{255}{106} + \frac{255}{239} + \frac{188}{51}}{4} \\ &= 233 \end{aligned} \tag{2.19}$$

Since all the green pixel values are known from the bilinear interpolation, of the green channel, see Figure 2.8, from which the green channel values have been obtained using Matlab.

Now that an RGB image has been formed, it would be relevant to look at the next step in the image processing chain, namely image compression.

2.5 Image Compression

When sending information from a satellite in space to a ground station, it is desired to send as little information as possible while maintaining the most important details, because of the limited downlink bandwidth. In this section, different forms of compression and formats will be examined to reduce the amount of data to be transferred.

The information received from the camera is called RAW data. RAW data is the digital pixel value from the camera sensor. RAW data cannot be processed by the computer into an image, therefore it has to be converted into another format. Often, the bitmap (BMP) format is used. This format stores the RAW data in a specific structure with headers. By doing this, the computer can identify the file and its content. Thereby can the information be read and displayed as an image without any loss or compression. In addition, it is possible to convert bitmap format into a compressed format with loss or no loss.

2.5.1 Types of Data Compression

There are two overall types of compression: Lossless and lossy data compression. Lossless data compression is used when it is important not to lose any data - this could be in a text document, executable programs or source code. Examples of lossless compression methods are ZIP, FLAC, PNG. Thus, in lossless compression every single bit of data from the original file is present in the decompressed file [Harris, 2015]. For instance, when compressing a text document with a lossless data compression, it exchanges frequently used lines with a smaller associated signature, and by only storing one of the commonly used lines, the size of the file will become smaller. When decompressing the compressed file, the associated signature will be replaced with the commonly used line and all the original data will be restored. This is called run-length encoding [Rouse, 2005]., and this form of compression is used in tagged image file format (TIFF) file formats [Atkins, 2004].

Lossy data compression does not restore all the data from the original file. Lossy compression reduces the file size by removing data. The data which is removed is prioritized [Rouse, 2005]. Thus redundant information is removed first, and critical information is retained. For instance, a picture may contain more detail than the human eye can perceive, this information is unnecessary and is the first data removed. Because lossy compression removes data, the resulting size of the file will become significantly smaller than the original file size [Harris, 2015].

There is a wide variety of encoding methods, but only two different methods of encoding will be described in the following, namely Huffman coding and joint photographic experts group (JPEG).

2.5.2 Huffman Coding

Huffman coding is a type of lossless data compression and is used frequently in different file compression formats, both lossless and lossy. It is a compression technique used to reduce the number of bits used to send a message. The general concept of Huffman coding is to identify frequently used pixels in the image and then assign them a smaller signature [Brown, 2012]. The less frequently occurring pixels is assigned a longer signature than those pixels occurring often [WhyDoMath, 2015]. To show how the Huffman coding algorithm works, an example with the word *Abrakadabra* is examined.

Abrakadabra has 11 characters to represent the word. In a programming context, each character is of the variable type char, which consists of 8 bits. 11 characters multiplied with 8 bits will give a total of 88 bits.

There are 5 different characters in the word *Abrakadabra*. The most frequently appearing character is *a* which appear 5 time. *b* and *r*, both appear in the word 2 times. The less frequently appearing characters are *d* and *k*, which only appear in the word 1 each. Each of the 5 characters will be sorted together with their frequency of occurrence, where the greatest number will be on the left hand side and the least number on the right hand side, see Figure 2.10.

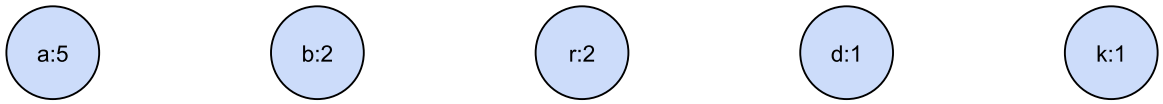


Figure 2.10: The 4 characters with there frequencies of occurrence.

The order in this example is a:5, b:2, r:2, d:1 and m:1. The 5 nodes are the roots of a binary tree, also called leafs. The binary tree is created by merging together nodes and combining them to one. The two nodes with the lowest numbers are combined first and the sum of each node together will be the sum of the new combined node. The combined node of d:1 and k:1, dk:2, has a frequency of occurrences value of 2, see Figure 2.11. This is done until all nodes are combined. In this example, b:2 is combined with the leaf r:2 and the total sum is 4, see Figure 2.11. The two new combined nodes, br:4 and dk:2, is combined to the node brdk:6. The last node, a:5, is merged together with the node brdk:6, and their total sum is 11.

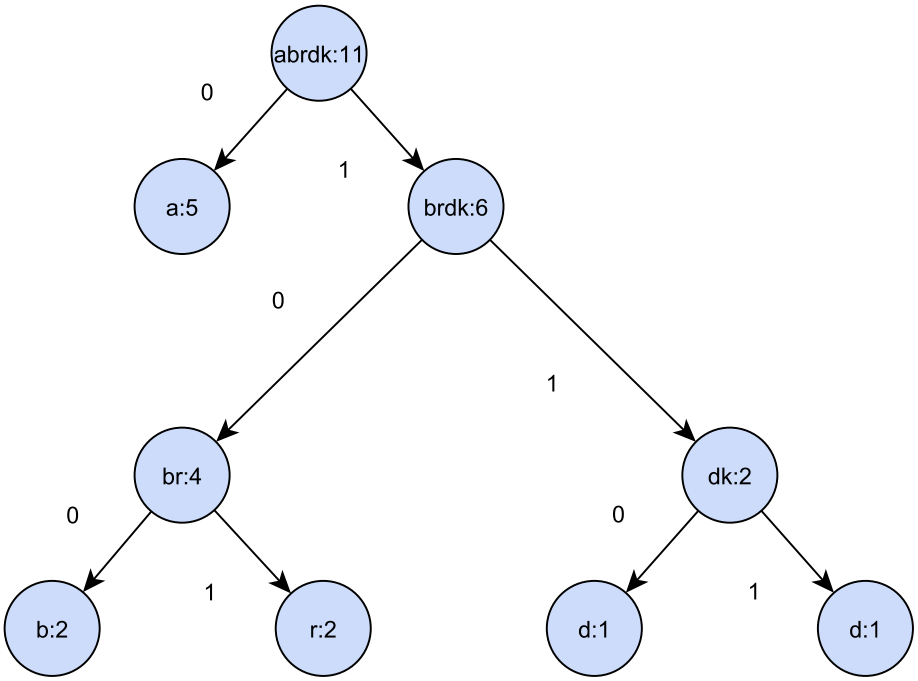


Figure 2.11: A Huffman binary tree created for the word Abrakadabra

The frequency of occurrence of the top node has to be identical to the number of characters in the original word. To make the huffman binary tree complete, the left edge at each node should be marked with a 0 and the right edge at each node should be marked with a 1, see Figure 2.11.

The binary code for each character is found by following each edge from the top of the tree down to the leaf containing the letter. The binary code for the most often used character a is 0, for the less used character d the binary code is 111. As described earlier, the general concept of Huffman coding is to assign frequently used characters or values the smallest signature, as seen with the character a.

The bit representation of the word *Abrakadabra* will then be: *01001010111011001001010* - this consist of 23 bits. The original word consisted of 88 bits, and thus, by using the Huffman coding, the amount of bits used was 26 percent of the original amount of bits. In this example Huffman coding was used on a word, but it can also be used on an image as well.

When decompressing the encoded data each signature for the associated character is needed. By replacing the character with the associated signature the original file will be restored. When the Huffman algorithm generates a signature the binary combination cannot be misunderstood. Therefore the replacement of the signature with the associated character cannot generate another image than the original one.

2.5.3 JPEG

One of the more frequently used file formats for lossy image compression is JPEG. To go from a bitmap to a JPEG file a series of steps has to be performed. First, the bitmap file (consisting of RGB colors) is converted to another color scheme called YCbCr. YCbCr consists of three components, Y, Cb and Cr. Y is the luminance (the light intensity), which is essentially a greyscale of the original picture, Cb is the blue difference and Cr is the red difference [Techradar., 2009]. The three components with the original picture is displayed in Figure 2.12.

Figure 2.12: Picture number one from the left is the YCbCr picture, number two is the Y channel, number three the Cb-component and the last picture is the Cr-component[Bilsen, 2015].

The human eye is more efficient at discriminating light intensity than colors. JPEG exploits this by primarily quantizing the Cb and Cr components [Techradar., 2009], also known as down sampling, since the resolution of the color channels are reduced.

After this, the images with the color components and luminance is split into 8x8 pixel blocks, which will give 64 pixel intensities in each block. The blocks are each transformed individually with the use of the discrete cosine transform (DCT) into a new matrix. Before the transformation, the block (matrix) is represented by a digital value for each pixel. When the DCT has been applied, the matrix is instead described by 64 frequency coefficients. DCT is similar to the Fourier transformation, because it is using cosine waves oscillating at different amplitudes and frequencies to analyse the original digital values of the pixel and its location [WhyDoMath., 2011]. Located towards the top left corner of the picture are the coefficients representing the low frequencies. The coefficient with the highest value is found in the top left corner of the picture and is called the DC coefficient, since it has the lowest frequency. The further away from the top left corner the coefficients get, the values represent gradually higher frequencies, and the values also get smaller [Enginering, 2015].

When using the JPEG file format the user has influence on the trade-off between quality and file size. This gives the user the opportunity of deciding the quantity of data removed. If the user wants a highly compressed file, the intensity of the quantization is increased [Enginering, 2015].

The DCT coefficients in the Cb and Cr matrices is divided with an standard 8x8 JPEG quantization matrix, defined by the JPEG standard. The values in the standard 8x8 JPEG quantization matrix will increase if the user wants an highly compressed file and the opposite will occur if the

user wants a smaller file [WhyDoMath., 2011]. An example of a standard JPEG quantization matrix (the Z matrix) and a DCT matrix divided with the standard JPEG quantization matrix (the Q matrix) is displayed in Equation 2.20.

$$Z = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \quad Q = \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -3 & 4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.20}$$

Where:

Z is the standard JPEG quantization matrix

Q is an example of a DCT matrix divided with the a JPEG quantization matrix

Because of the specific changes in coefficient values in the quantization matrix, the top left corner of the DCT matrix (low frequencies) is divided with a lesser value than those at the bottom right corner (higher frequencies). This will often result in a large number of zeroes at the bottom left corner in the Q matrix, which makes the image more compressible because of the repeating value. Higher frequencies gives the image more detail, whereas the low frequencies is the average light intensity. Higher frequencies are not as influential as lower frequencies for the human eye, and therefore it is preferable that higher frequencies has a lower coefficient value. After the reduction, all values will be rounded to integers. The quantization process is when JPEG becomes a lossy compression, because the original values of the coefficients in the matrix are lost [WhyDoMath., 2011].

The next and last step in JPEG file compression is to encode the quantized image. This is done with the algorithm called Huffman coding, which was explained earlier. The JPEG compression is thus complete and, if selected by the user, the file size has been dramatically reduced. Also, do note that the effect of the JPEG compression can be very different from image to image.

Based on the image seen in Figure 2.13, examples of the relationship between the selected amount of compression and the file size can be seen in Figure 2.14.

Figure 2.13: A 1 Mpx image used for demonstration

In Figure 2.14 the image has been compressed with different user selected settings in Adobe Photoshop, and the factor with which the image has been compressed is displayed - smaller numbers indicate greater compression. Each factor represents a quantization matrix which can be found in [Impulseadventure, 2008].

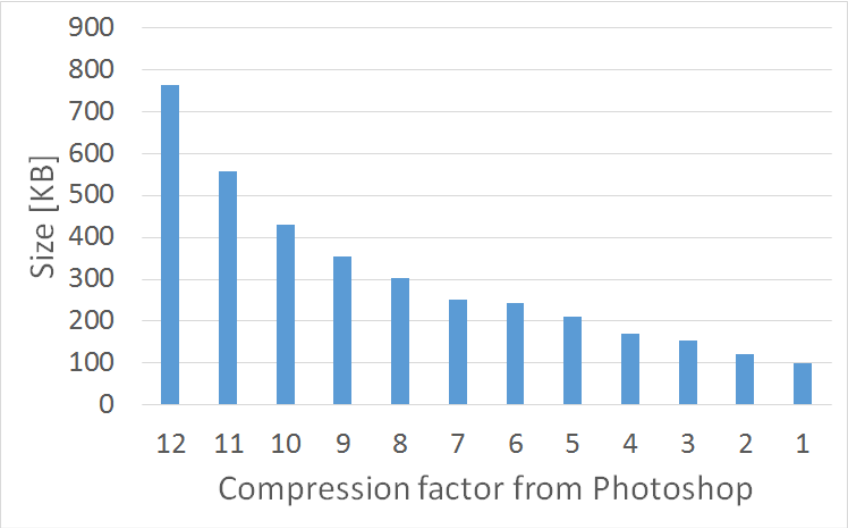
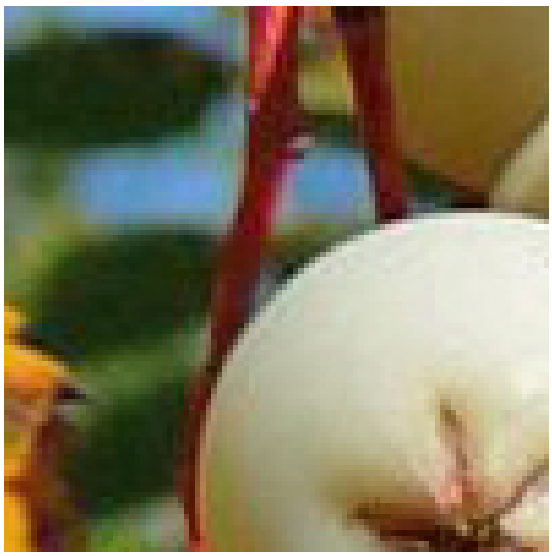
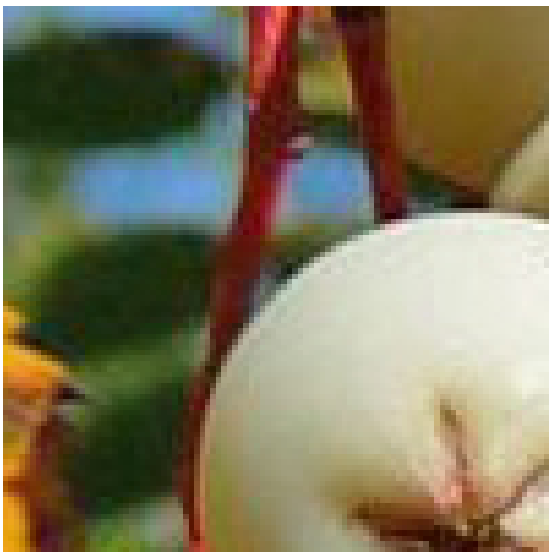


Figure 2.14: A graph comparing file size and compression factor of the image in Figure 2.13. Note that a compression factor of 10 is equal to no compression, and a factor of 1 applies the most compression.

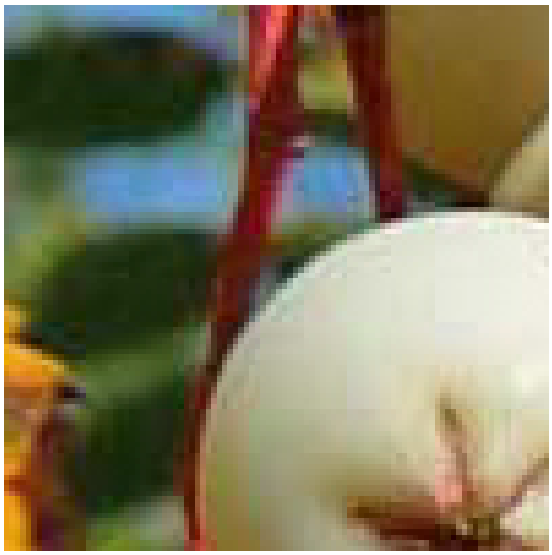
To see the difference in compression, Figure 2.13 has been compressed several times with different factors and the results can be seen in Figure 2.15



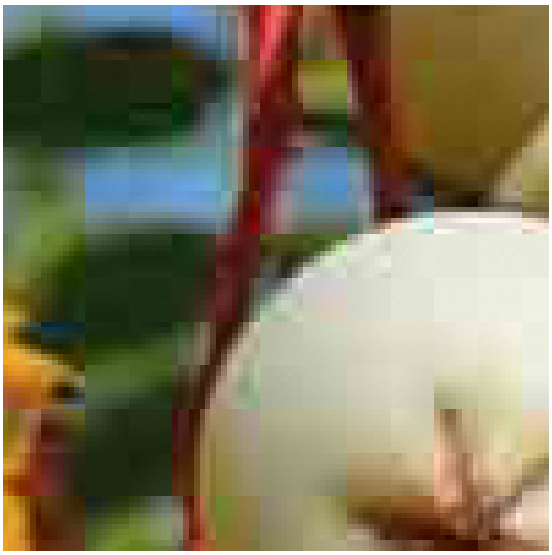
(a) Original image. File size: 761 KB



(b) Factor 9 compression. Size: 354 KB



(c) Factor 4 compression. Size: 169 KB



(d) Factor 1 compression. Size: 99.4 KB

Figure 2.15: An image compressed using JPEG in Photoshop showing the quality and the file size

As seen in Figure 2.15 the image looks mostly the same across all compressions, which is what JPEG compression is about, though the edges of the picture degrade when the compression factor decreases.

A basic explanation of the overall different compression types has been explained, and an example has been given. In the next chapter, considerations when sending a camera into space are examined.

3 | Communication interfaces

In this chapter, communication protocols that are to be used in the project will be described, to establish an overview and understanding of their workings. A brief introduction will be given into different random access memory (RAM) types in order to expose eventual weaknesses and discover their strengths. The mobile industry processor interface (MIPI) will be explained, since many cameras utilize this standard for communication. The I²C communication bus is also explained, since this is a commonly used interface across multiple electronic components.

3.1 MIPI

MIPI is a set of standards, created by the MIPI Alliance, for communication between components in mobile devices. The standards contain specifications for everything from camera interfaces to storage and radio frequency (RF). For this project, the most relevant part is the camera interface. As seen on Figure 3.1, the camera serial interface (CSI) exists in multiple versions, with three different physical layer (PHY)s.

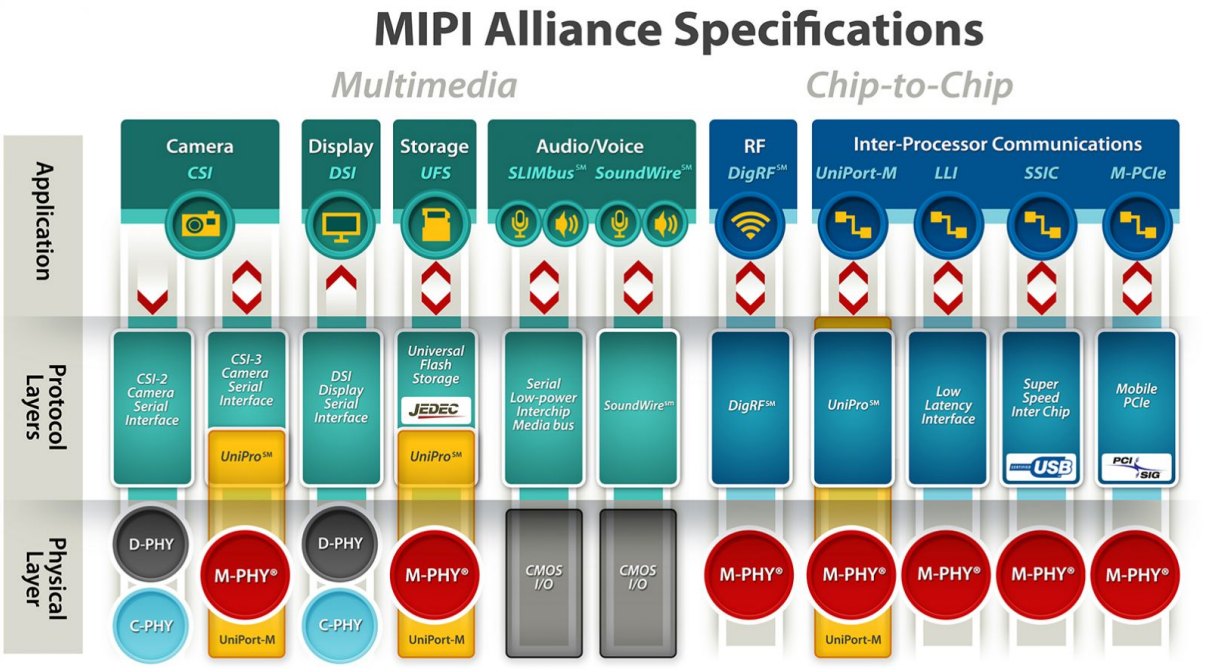


Figure 3.1: Overview of the MIPI specifications [source: `mipi.org/specifications/specifications-framework`]

As the standards are confidential, and only for paying members of the MIPI alliance, the details about which version is used is not made clear in camera datasheets. This can however be deducted, by close examination of the pin out of the camera module. The interface has three

different layers, and multiple variations of the physical interfaces, of which only the physical layer type D (D-PHY) physical layer will be described.

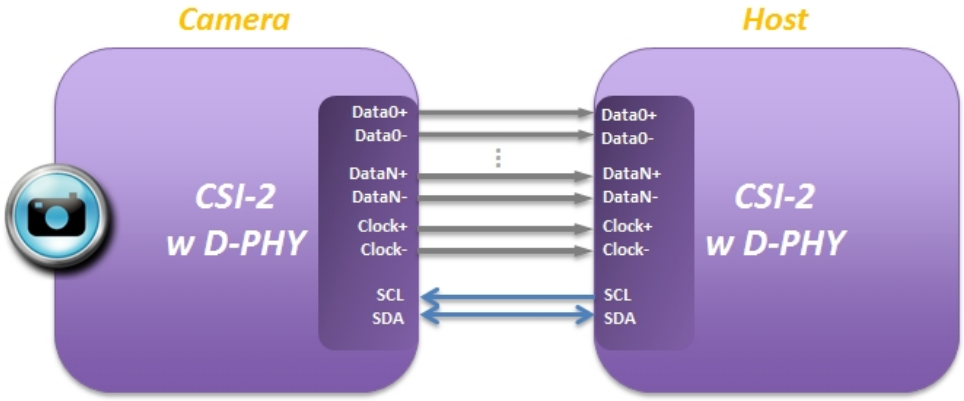


Figure 3.2: D-PHY physical layer [source: `mipi.org/specifications/camera-interface`]

3.1.1 D-PHY

MIPI D-PHY is the physical (electrical) layer. It is a bus designed for camera and display interfacing. It incorporates one clock lane, and from one to four data lanes. The intrinsic impedance is 100Ω differential and/or 50Ω single ended [Defossez, 2014, p. 5]. The bus can switch between 200mV differential signaling (high state (HS) mode) and 1.2V single ended CMOS logic (low power (LP) mode) on demand, see Figure 3.3.

HS mode is used when transferring image or video data. In between the HS packets, the bus switches to LP mode to conserve power. In HS mode the bus is unidirectional. The clock is generated by the master, and data is transferred on both the rising edge and the falling edge. This is knows as double data rate (DDR). The master is always the transmitter, and the slave always the receiver. All lanes (clock and data) must be terminated in HS mode, to avoid reflections. In a typical scenario with a camera connected to a processor, the camera will therefore be the master, and the processor the slave.

The data rate per lane goes from 80 Mbit/s to a maximum of 1 Gbit/s in HS mode, and $\leq$ 10 Mbit/s [Defossez, 2014, table 1].

In LP mode the bus becomes bidirectional on lane 0, making two way communications possible. This is most commonly used for sending configuration parameters to a display. In order to conserve power, the clock and data lanes are not terminated in this mode.

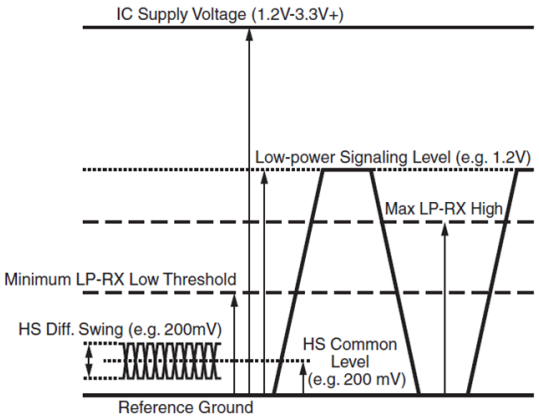


Figure 3.3: D-PHY electrical signal levels [source: [Defossez, 2014, fig. 1]]

3.2 I²C

The I²C protocol was developed in 1982 by Philips. Originally, it had a limited transfer rate of max 100 kHz, but it was later increased to 400 kHz. Most devices only support up to this transfer rate, but different other modes has been made, enabling speeds up to 5 MHz [Sparkfun, 2015a]. Also, I²C only supported up to 7 address bits, but this has since been increased to 10 bits. However, 7 bits are often enough for must applications [Sparkfun, 2015a].

Why I²C ?
Serial ports communicate via universal asynchronous receive/transmitter (UART). Since this is an asynchronous transmission, the two devices communicating need to be set up with the same transfer rate before the start of transmission. This also means, that if either units clock is unstable, the communication will fail. In UART, the units are "equal", i.e. there is no *master* nor *slave*. Finally, UART only allows communication between two modules [Sparkfun, 2015a].

SPI fixes this problem by introducing a clock pin. In SPI, the units are not equal - there is one master, and multiple slaves. However, more pins are required - at least 4: master in slave out (MISO), master out slave in (MOSI), serial clock (SCK) and chip select (CS). For each additional slave above 1, another CS pin has to be added from master to that slave. One of the strengths of SPI is its high transfer speed, which is due to SPI being *full duplex*[1]. Another strength is its clock pin, giving all units the same clock to rely on, thus reducing timing problems. Finally, SPI allows transfer rates up to 10 MHz[Sparkfun, 2015a].

I²C is a compromise between UART and SPI, giving the low wire count of UART and some of the pros of SPI, namely multiple slaves and a clock pin. The I²C protocol is based upon two-wire interface (TWI). The two lines used in I²C is serial clock (SCL) and serial data (SDA). One of the strengths of I²C is its ability to let multiple masters and slaves communicate on the same two wires. The clock signal is generated by the bus master, but some slaves may slow down the clock to force the master to transmit at slower rate - this is called "clock stretching" [Sparkfun,

[1]Data can be sent both ways simultaneously

2015a].

In the I²C protocol, communication lines have a pull-up resistor - typically 4,7 kΩ. This means that a signal is generally high and can only be pulled low. This also removes the potential problem of two units trying to pull the line both high and low at the same time, since the signal can only be driven low [Sparkfun, 2015a].

The I²C protocol

I²C messages are broken up into two types of frames: Address frames (the master letting the slaves note which slave the master is talking to) and message frames. Data is put on the SDA line after the SCL goes low, and is read after the SCL goes high.

An I²C message begins, when SDA goes low before SCL. After this, 7 address bits are sent, followed by a read/write bit, saying if the master is requesting or sending data.
After the 7 address bits and the read/write bit have been sent, the slave is to pull the SDA low, to acknowledge. Hereafter, the 8 data bits are sent, followed by an acknowledge by the slave.
Finally, to signify the end of the message, the SDA goes high after SCL.

This can be seen on Figure 3.4 [Sparkfun, 2015b].

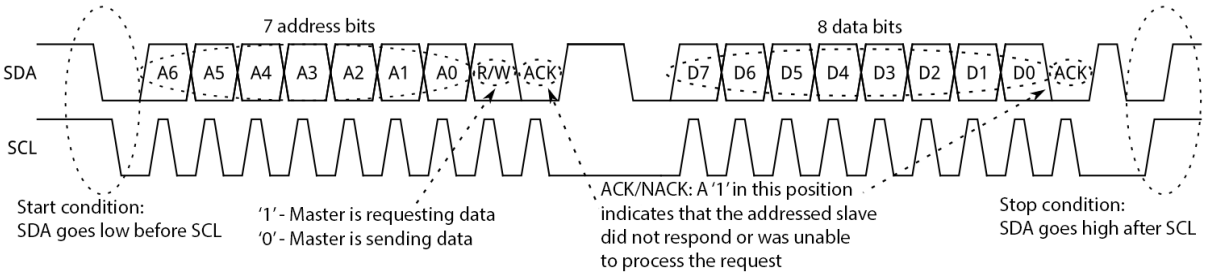


Figure 3.4: A timing diagram of the I²C protocol

In I²C, 10 bit addresses are possible. These are implemented by sending an address package preceeding the address package mentioned above. This preceeding package starts with the bitstream 0b1110, followed by address bits 9 and 8. This is clever because the pattern 0b1110 is not a legal address, meaning the package cannot be misinterpreted as a 7-bit data package. [Sparkfun, 2015a]

The following addresses are reserved in I²C, and must thus not be used [I2C-bus, 2015]:

- 0b0000000 0: General call
- 0b0000000 1: Start byte
- 0b0000001 x: CBUS addresses
- 0b00001xx x: High-speed Master Code
- 0b0000011 0: Reserved for future purposes
- 0b0000010 x: Reserved for different bus formats
- 0b11110xx x: 10-bit slave addressing
- 0b11111xx x: Reserved for future purposes

Where x signifies any bit value.

3.3 RAM types

Today's vast expanding digital world gives a vast variety of RAM to choose from. The different RAM types differ in respect to size, cost, efficiency etc. They are utilized differently according to what part of the system they are communicating with. A closer inspection is needed to determine what type of RAM is needed for handling the data being received from the camera and read from the micro controller. The following section describes four possible RAM types that could be suitable for this task.

Static RAM

Static random access memory (SRAM) are constructed using several transistors, typically four or six for each cell. A SRAM cell has the advantages of making RAM refresh obsolete. The cell be constantly keeping its given value until overwritten. The large number of components used have a large impact on the cost, making the SRAM a rather expensive solution.

Dynamic RAM

Dynamic random access memory (DRAM) are constructed using a transistor and a capacitor. In this cell, the capacitor will be keeping the charge i.e. the value of the bit. This calls for an external refresh system keeping all the cells accounted for. Advantages of these cells are their low cost

Synchronous Dynamic RAM

Synchronous Dynamic random access memory (SDRAM) are primarily like DRAM, but these RAM utilize the concept of burst modes and thereby increasing the speed of the RAM. This is realised using a SRAM buffer where the specific row is loaded onto, treated, and then loaded back into the SDRAM. This burst or full page[2] feature are beneficial when work whit data which should be handled sequentially, like a series of pixels.

Double Data Rate Synchronous Dynamic Ram

Double Data Rate Synchronous Dynamic random access memory (DDR SDRAM) are the same type as SDRAM however they utilize both the rising and falling edge of the clock, increasing the data transfer rate by a factor of 2. They have the same advantages and disadvantages as SDRAM.

Concluding this chapter knowing what interface a possible solution could end up using and what possible RAM types there are available for memory, it would now be preferable to look at what specifications needed when choosing the camera. In the following chapter a closer look on field of view and pixel density will be discussed.

[2] full page read/write tells that the entire row should be treated before changing address

4 | Cameras in space

When putting a camera in space onboard a satellite, some measures has to be taken to ensure the functionality of the camera. Also, the requirements there are to the camera based on its purpose in the mission has to be taken into account. These things will be investigated in this chapter.

4.1 Field of view

The basics concepts of field of view are thoroughly described in section 2.1. With that in mind, the field of view can be described by a right triangle, where the adjacent side corresponds to the focal length[1]. The relations are described in Equation 2.1 and Equation 2.2, and depicted in figure Figure 2.3a. Using those relations and the diagonal of a standard 35 mm image sensor being roughly 42,7 mm $\pm$ 1 mm[Dempsey, 2015]. These relations yield Equation 4.1 which correlates FOV and the focal length.

$$FOV = 2 \cdot tan^{-1}\left(\frac{42{,}7\,\text{mm} \cdot 0.5}{f}\right) \tag{4.1}$$

Where:

FOV	is the cameras field of view [degrees]
f	is the focal length of the camera [mm]

Figure 4.1 shows a graphical representation of the expression in *Equation: (4.1)*, showing an increase in focal length will reduce the FOV rather rapidly.

[1] focal length being the length from the image sensor to the lens

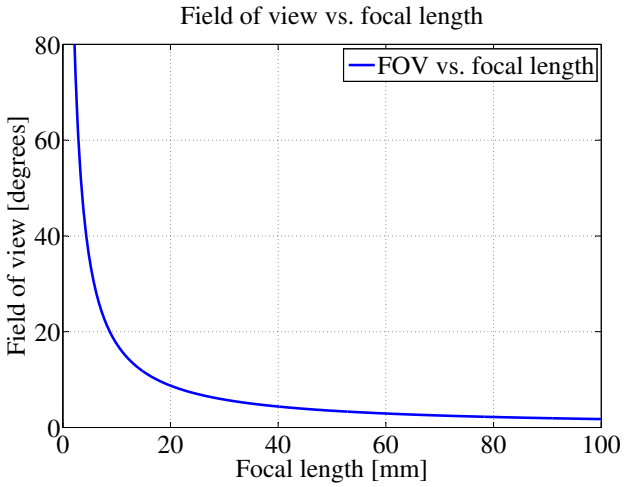


Figure 4.1: Graphic representation of the satellites FOV in degrees vs the focal length in mm. The reference image sensor is a standard 35 mm image sensor.

After determining the FOV, it raises the question of how large an area the camera is able to cover as a direct outcome of the altitude the satellite will be orbiting.

4.2 Size of target

The main purpose of the satellites camera is to capture images from space that can be used for either scientific or humanitarian purposes. The size of the target area will mainly depend on the cameras FOV and the orbit altitude. The possible area able to be covered by the camera is explained in Equation 4.4. In order to understand the elements of the equation, an assumption has to be made about the camera having a sensor aspect ratio of 4:3. Knowing a rectangular area can be determined by $a \cdot b$ and the fact that a rectangle with an aspect ratio of 4:3 has a diagonal ratio of 5. This gives the desired information for determining the area of the picture, see Equation 4.2.

$$A = a \cdot b = \left(\frac{4}{5} \cdot d\right) \cdot \left(\frac{3}{5} \cdot d\right) = \frac{12}{25} \cdot d^2 \tag{4.2}$$

Where:

- A is the area covered by the picture[km^2]
- d is the diagonal distance of the camera [km]

Using a right triangle, where half of the diagonal d corresponds to the opposite side, the orbit attitude h corresponds to the adjacent side and the FOV is the angle between the adjacent side and the hypotenuse, a correlation for the diagonal is found in Equation 4.3.

$$d = 2 \cdot \left(tan\left(\frac{FOV}{2}\right) \cdot h \right) \tag{4.3}$$

Equation 4.4 can now be determined by combining equation 4.2 and 4.3. This equation is depicted on Figure 4.2 where the FOV has an incremental rise of 10^o, the lowest being 10^o.

$$A = \frac{12}{25} \cdot \left(2 \cdot \left(tan\left(\frac{FOV}{2}\right) \cdot h \right) \right)^2 \tag{4.4}$$

Where:

A	is the area covered by the picture[km^2]
FOV	is the cameras field of view [degrees]
h	is the orbit height [km]

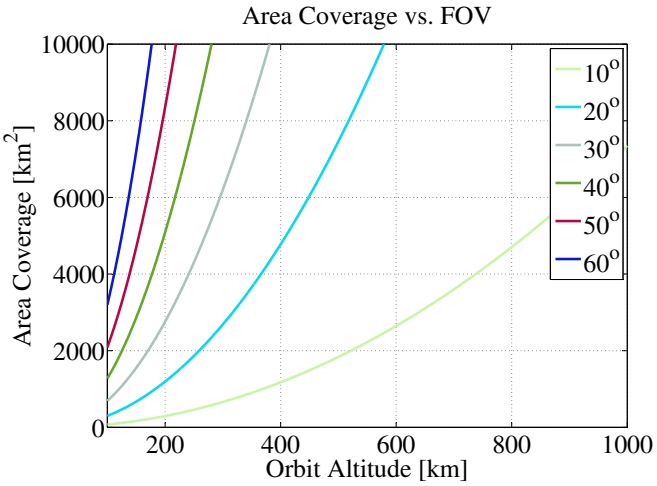


Figure 4.2: Graphic representation of the satellites camera coverage in km^2 vs the orbit height in km. The lowest line have a FOV of 10 °and increments with 10 °

Figure 4.2 is showing a linear rise in area coverage due to the orbit altitude. With an increase in area coverage, it comes in mind that there is only a finite amount of pixels to describe the area, which begs the question about the resolution, i.e. pixels pr. km^2.

4.3 Pixel Density

When the satellite is in orbit and ready to capture frames, the pixel density comes into play. The pixel density describes the amount of pixels pr. area covered. The camera image sensor will be given with a specific resolution which sets the baseline for pixel pr. area covered.

Equation 4.5 shows the relation between with pixel density, FOV and orbit height. The equation is combined under the assumption that the image sensor has a 5 megapixel resolution. The expression is showing the behaviour of pixel density as a function of the orbit height. In reality however, the expression is a function of ratio between the resolution and area, $\frac{\text{megapixel}}{\text{area}}$.

$$\text{PxD} = \frac{5 \cdot 10^6}{\frac{12}{25} \cdot \left(2 \cdot \left(tan\left(\frac{FOV}{2}\right) \cdot h\right)\right)^2} \tag{4.5}$$

Where:

PxD	is the pixel density of the photo $[pixel/km^2]$
FOV	is the cameras field of view [degrees]
h	is the orbit height [km]

A graphical representation of the pixel density is depicted in Figure 4.3 with respect to the FOV. The graph is created by assuming a height of 600 km.

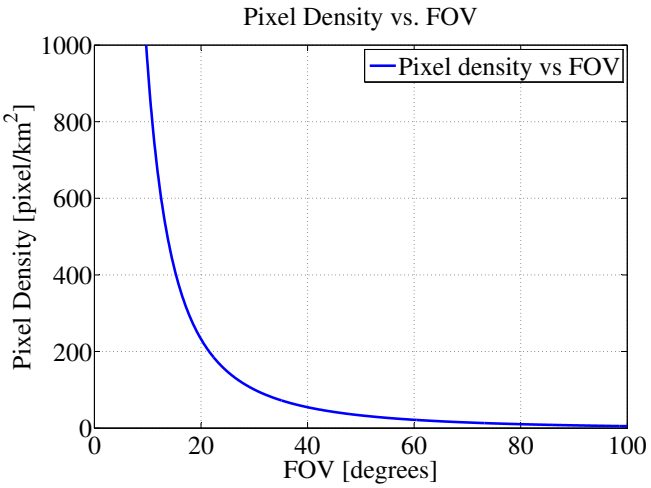


Figure 4.3: Graphic representation of the satellites picture resolution in pixel vs the diagonal FOV in degrees. The function is displaying an orbit altitude of 600 km

As can be seen on Figure 4.3, an increase in field of view degrades the pixel density. It also shows that a small FOV has a relatively large pixel density. The pixel density will again increase or decrease with respect to the orbit height but still following the contour of the graph. when pixel density is know it raises a couple of questions, such as what to photograph, physical limitations hardware limitations and so forth.

4.4 Other considerations

A lot of different things need to be considered when launcing a camera inside a cubesat. First of is what should be photographed but also the physical and hardware boundaries.

When in orbit, only a couple of things can be photographed. The things considered are:

- The Earth
- The stars
- The Sun
- Other Satellites

Depending on the target, different kinds of cameras are preferred e.g. if the stars are the target a camera with long exposure time and big pixel cells should be used to be able to gather as much light as possible. What should also be considered is that the satellite might be tumbling and therefore it is not given that the camera is pointing the desired direction, meaning that not all pictures are of the intended subject. This could prove troublesome for instance when trying to take a picture of the stars but pointing at the sun instead might overload and burn the sensor.

Because the satellite might tumble it is necessary to consider what to do with pictures that do not picture the desired target. An example of this problem could be if the satellite tumbles with 1 degree per minute in an orbit of 600 km and the target is the earth. Without considering the field of view, the satellite will only be pointing towards the earth in roughly 132 degrees, see Figure 4.4.

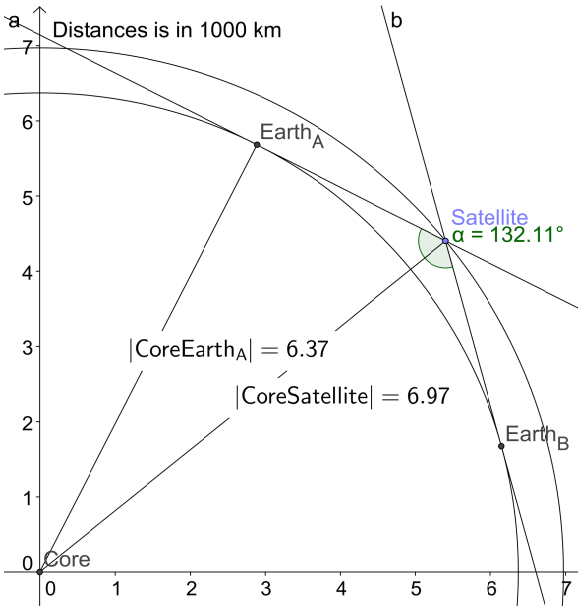


Figure 4.4: A simplifed drawing of the Earth and the satellite orbiting it

That means that only in roughly $\frac{132}{360} \approx 36.7\,\%$ of the time is the satellite actually pointing towards the Earth. Since the tublerate is 1 degree per minute, this ratio can be understood as for each 132 minutes the satellite is pointing towards the earth and take pictures, it spends 228 minutes pointing the wrong way.

There are a couple of different ways around this problem. One is to track where the camera is pointing and then only take pictures when pointing in the right direction. Another is to either automatically or manually delete pictures that have been taken in a wrong direction.

A completely different problem is the physical limitations. Given the size of a cubesat section 1.1 and taking into account that there should be room for a power supply, a radio, an attide control system and the payload as a minimum, it does not leave much room for the camera and much less for the lens. When choosing a camera this has to be taken into account.

Also, when taking a picture and processing it two things are needed to store the data; RAM and flash memory. In a satellite, these things can be a problem especially if large quantities are needed. Therefore, it is necessary to consider how much RAM is needed and how many pictures need to be stored on-board the satellite. Depending on the chosen method to convert the RAW image data to a full coloured image and the chosen method of compressing it, different sizes of RAM are needed. In the worst case scenario, the whole picture has to be stored in the RAM at a given point. In the best case, only a fraction of the picture needs to be stored. For instance, if choosing JPEG compression, see subsection 2.5.3, only 8 rows of pixels need to be in memory at any given time. Either way, it is highly unlikely that enough RAM and flash can be provided from any microcontroller, FPGA ect. Therefore, when choosing a chip (either microcontroller or FPGA) it should be considered that the one chosen is compatible with external storage devices.

From Section 1.1: AAUSAT it is know that the downlink speed of AAUSAT is typically 9600 bits per second. If a picture is 2 megapixels (Mpx) and in 8-bit RGB colors, i.e. 8 bits per color channel per pixel, then the whole picture would have a size of $2\,\mathrm{Mp} \cdot 8\,\frac{\mathrm{bit}}{\mathrm{px}} \cdot 3 = 48\,\mathrm{Mbit}$. If CSP is used for the downlink, 84 bytes out of 250 bytes are data - the rest is part of the CSP format definition. Assuming data is streamed continuously, the time to download an image can be seen in Equation 4.6.

$$\frac{48\,\mathrm{Mbit} \cdot 250}{84} \cdot \left(9600\,\frac{\mathrm{bit}}{\mathrm{s}}\right)^{-1} = 4{,}1\,\mathrm{hours} \tag{4.6}$$

From Figure 1.1 it is known that communication with the satellite is possible around 180 minutes a day. Considering this, it would take:

$$\frac{4{,}1\,\mathrm{hours} \cdot 60\,\mathrm{minutes/hour}}{180\,\mathrm{minutes/day}} = 1{,}4\,\mathrm{days} \tag{4.7}$$

When choosing a camera design this should be considered so the downlink time can be minimized.

Also, when taking a picture and processing it, two things are needed to store the data; RAM and flash memory. In a satellite these things can be a problem especially in large quantities are needed. Therefore it is necessary to consider how much RAM is needed and how many pictures need to be stored on-board the satellite. Depending on the method of converting the

RAW data to a full coloured image and the method of compressing it, different sizes of RAM is needed. Worst case scenario the whole picture has to be stored in the RAM at a given point, best case only a fraction of the picture needs to be stored e.g. choosing JPEG compression see subsection 2.5.3 only 8 rows of pixels need to be in memory at any given time. In any way it is highly unlikely that enough RAM and flash can be provided from any micro controller, field programmable gate array (FPGA) ect. The chip used should therefore be compatible with external storage devices.

4.4.1 Temperature in space

In space, a crucial thing to consider is temperature. The satellite is stressed due to the temperature in space for two main reasons. First, the outside temperature can vary from -40 °C to 65 °C [Space, 2015a, p.8]. The other element is that it is difficult to release heat to the surroundings. Therefore, every component used in the satellite should have an operating range which covers these criteria. Simultaneously, if one side of the satellite is facing the sun, the other side is not, meaning there is a large temperature gradient over the frame of the satellite since the side facing the sun is much hotter than the one facing away. Therefore, the satellite and it's components must also be able to withstand this gradient.

Now that all the basics are covered the design of the satellite can commence.

Part II

Design & implementation

5 | Design considerations

In this section, the system will be designed with a top-down approach, starting with a use-case description of the overall functionalities of the system. Hereafter, the constraints set by the satellite, with regards to physical limitations, downlink speed etc. is considered. Based on the use-case descriptions and the constraints, it is possible to make an overall system design, describing the systems building blocks. The requirements for the system will also be listed.

5.1 Use case design

Before the payload can be designed, a few design considerations has to be made, in order to establish what the payload must be able to do. This will be done by means of a UML use case diagram, see Figure 5.1.

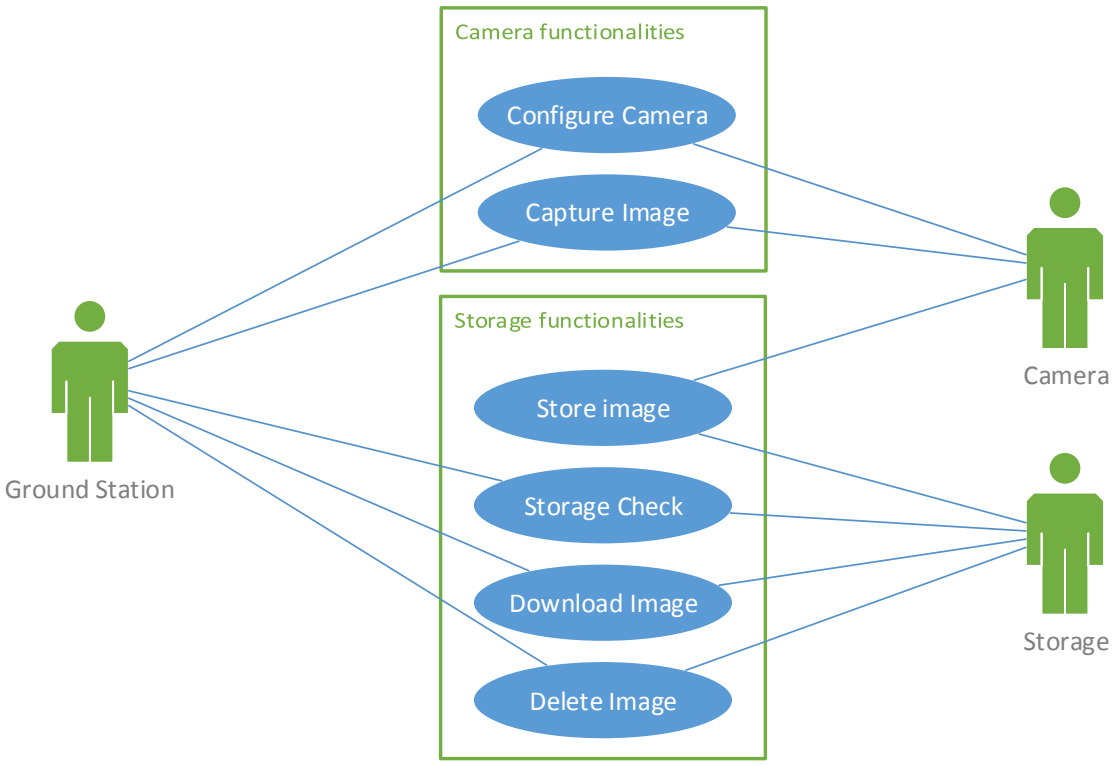


Figure 5.1: A usecase diagram showing the overall functionalities of the system

As previously mentioned, the main purpose of the payload is to automatically capture images with a interval set by Ground Station, store them on board the satellite, and be able to transmit them to the ground, so the images can be viewed.

The ground stations function is to set up the payload. Therefore, all the functionalities of the payload are to be controlled from the ground station. This includes configuring the camera, i.e. change settings, and downloading images from the satellite.

The ground station should also be able to delete images and do a storage check of the satellite, so the storage space can be managed manually to prevent overload.
Also, being able to remotely capture an image from the ground station should be possible, if it is wished to take an images during a pass.

5.2 Payload constraints

Before the building blocks of the system can be established, some considerations has to be done with respect to the limitations set by the platform the system is to be implemented in, i.e. a cubesat. In this section, the considerations needed to be made for the overall system to be implemented in a satellite will be described.

To minimize the amount of data to be sent from the satellite to the ground station, it is decided that the demosaicing will be performed on the ground station. This is because the demosaicing procedure produces 3 times as much data as the original image, which is not ideal when downlink time has to be considered.
The images must also be saved in a commonly used file format, so they can be seen and used by other users who do not have any special software available that might otherwise be needed to view the image. Such formats could be JPEG og BMP images. Here, BMP would be the better choice since this format is much easier to save images as than JPEG, which can be seen in Section 2.5: Image Compression. However, the actual conversion of the raw image data to a BMP file will not be done in this project.

Since the payload is to be implemented in a cubesat having a size of minimum 1U, the size of the entire satellite must not be greater than $10 \times 10 \times 11$ cm. see Section 1.1: AAUSAT. A maximum payload height of 4 cm has been established as a reasonable and realistic size, when considering the PCB height of the other subsystems. Also, since the satellite has to fit the frame, solar panels etc. the PCBs cannot be larger than 87×87 mm in size, as can be seen in the schematics/board layouts of the AAUSAT PCBs.

Because the satellite has a power budget of 1.5 watts on average, see Section 1.1: AAUSAT, the payload must not use more power than this, since the satellite will then be drained for all power. Also, there should also be enough power to supply the other subsystems. The exact power consumption of the subsystems is not known, but a rough estimate is that half the power of the satellite goes to the payload and the other half to the rest of the subsystems. The exact power consumption is not critical, since the EPS will shut down the payload if the battery voltage drops below a certain voltage. Also, since the 1.5 Watts available on the satellite is an average power, a spontaneously higher power consumption of the payload is allowed, since it will only be on some of the time, thus averaging down below the allowed power consumption threshold. Nonetheless, a low power consumption is desired, to ensure the payload to be used for as long as possible per battery charge.

As investigated in Section 1.1: AAUSAT, the downlink speed sets a limitation on how big the image files should be, to be able to download them in a reasonable amount of time. Here, it is decided that a reasonable download time is 1 day. This can be achieved either by taking small photos or compressing the images. Since it is not desired to decrease image quality, a lossless compression scheme is to be used.
However, it would be very impractical to wait so long to download every image before it is known if the image actually contains useful image data or is just a "black frame"[1]. This can be solved by rendering small resolution previews of all the images. These previews can then be sent to the ground station, where the operator can choose which images to download in full resolution. Assuming an 8-bit black and white preview, with a size of 100 × 100 pixels, a pass duration of 10 minutes and a downlink speed of 9600 bps, 24 previews per pass should theoretically be obtainable, as can be calculated from Equation 5.1. Based on these ideal conditions, it is estimated that in reality, the ability to download at least 20 previews per pass is both a sufficient and realistic amount.

$$\frac{\text{baud rate} \cdot \text{pass length}}{\text{preview size} \cdot \text{bit depth} \cdot \text{data ratio}} = \frac{\text{previews}}{\text{pass}} \tag{5.1}$$

Where:

Baud rate	is the transfer ratio [bits/s]
pass length	is the duration of a pass [s/pass]
preview size	is the resolution of the preview [pixels/preview]
bit depth	is the number of bits per pixel [bits/pixel]
Data ratio	is the ratio between data and package size in CSP, i.e. 84/250 [-]

To further improve the amount of useful previews being downloaded per pass, the payload could automatically discard the black frames so they do not have to be downloaded.

It would be preferable to make a time lapse video of an entire orbit. This could be done by capturing an image each second for a full orbit, and then stitch these together to form a movie. This would however set a requirement to the storage capabilities of the system so it can store these images. Another option is to allow the time lapse functionality to delete old images, to make room for new ones. It should however be possible to turn this feature off, in case it is desired to keep all images.
Assuming a 5 megapixel image, with 8 bits per pixel (since the demosaicing is not done on the spacecraft), one image would take up 5 MB of storage space. Since an orbit takes about 100 minutes, see Section 1.1: AAUSAT, this would mean that 6000 images has to be stored, summing to a total of 30 GB. There should of course also be room to store low-resolution previews of these images, but these would only take up 60 MB, assuming an 8-bit black and white preview, with a size of 100 × 100 pixels, which is negligible.

Based on the analysis performed in Section 4.2: Size of target and Section 4.3: Pixel Density, it is possible to consider the optimum object area that the camera photographs, as well as the

[1] A black frame is defined as a frame containing a certain percent of only black pixels

necessary pixel density, also known as spatial resolution. This decision is however left up to the designers of AAUSAT6, since it depends on the desired object size which is not known at this time. It also depends on the orbit altitude, which is also not known. When these factors are known, the field of view and spatial resolution can be calculated based on the graphs in Section 4.2: Size of target and Section 4.3: Pixel Density. The field of view and spatial resolution can be changed as wished, by choosing a suitable lens and sensor.

5.3 Payload requirements

Based on the preanalysis, the considerations and system overview, see Section 5.2: Payload constraints, it is possible to set up the requirements for the system to be integrated in the satellite.

The requirements are highlighted with **bold**, and the reason behind each requirement is listed underneath each requirement.

A The payload shall be able to capture an image and store it on onboard memory

The purpose of the payload is to capture images as well as store it for further processing, e.g. compression and downlink to ground station, see Section 5.1: Use case design.

B The payloads dimensions should not exceed 87x87x50 mm

This is the PCB size used in the current AAUSATs, see Section 5.2: Payload constraints

C The payload shall be able to communicate with other AAUSAT subsystems using CAN and CSP

Since the subsystems communicate via CAN and CSP, the payload should support these interfaces, see Section 1.1: AAUSAT.

D The power consumption of the payload shall not exceed 0.75 watt on average[2]

The payload should not use more than half of the available power on the satellite on average, see Section 5.2: Payload constraints

E The payload shall be able to send an image from storage through the COM, to the ground station where it can be saved as an image

Captured images should be viewable to the operator, see Section 5.2: Payload constraints

F From the ground station, it shall be possible to change the workings of the payload, in terms of resolution and when images should be captured

[2]Meaning the average power consumption measured during the entire mission, i.e. the time when the system is off is taken into account

It shall be possible to change settings depending how and when images shall be captured, see Section 5.1: Use case design

G The payload shall be able to compress the image lossless such that it can be downloaded from space in less than 1 day

Downloading an image shall be done in a reasonable amount of time without degrading image quality, Section 5.2: Payload constraints

H The payload must be able to detect black frames and not store these

Images not containing any useful information shall not be stored, see Section 5.2: Payload constraints

I At least 10 previews must be able to be downloaded from space per pass

The payload must be able to render and downlink small resolution previews of the captured images, as per Section 5.2: Payload constraints.

J From the ground station, it shall be possible to request previews and full resolution images as well as remote capture an image

Since the ground station acts as the main interface between the operator and the payload, it shall be possible to download previews and full resolution images through it. The operator shall also be able to manually remote capture an image, see Section 5.1: Use case design.

K From the ground station, it shall be possible to delete pictures and view on-board memory statistics

It is important that the memory can be managed manually from the ground station in case of memory issues that the payload cannot solve automatically, see Section 5.2: Payload constraints

L The payload must have at least 30 GB of storage space

To ensure that images from a full pass can be stored in on-board memory, see Section 5.2: Payload constraints

Note that the prototype developed in this project will *not* be tested in regards to these requirements. Instead, requirements for the prototype are listed in section 5.5, based on the prototype constraints that are determined in the following section.

5.4 Prototype constraints

In this project, a prototype of the payload will be made instead of the full system due to time limitations, but made so that the prototype still show how the main functionalities of the payload shall be.
This means that the prototype is to demonstrate the ability to capture an image, generate a preview and compress the full resolution image. The preview and compressed image must be saved in on-board storage, from where they can be fetched and transmitted through CAN. It shall also demonstrate the ability to view storage information about the files saved in storage and storage space left. It must also be possible to configure the camera to captures images with a set time interval.

The choice of lens and sensor in the prototype is not critical, since it can be changed if necessary, as long as the interface to the camera unit is identical. Therefore, optimizing the choice of camera, lens and sensor will not be made in the prototype.

Apart from the camera, will it primarily be the external communication with the rest of the satellite and the ground station that will be disregarded in the prototype, since it is not vital for the functionalities of the system itself. Thus, it will not be a key point to ensure compatibility with the rest of the satellite and the ground station. Instead, a simple CAN dongle connected to a PC is used as the prime communication with the prototype, and a terminal is used to visualize output. For the prototype it has been decided, that it is sufficient to implement CAN and not CSP. This is because of time constraints and the possibility of implementing the CSP layer as an expansion.

Power consumption is considered when choosing hardware modules for the prototype, since the hardware chosen for the prototype and that for the final system is to be as similar as possible, to ease the final integration. Changing hardware for a less power consuming variant may therefore change the system radically, which is not desired, and thus this has to be considered as early as possible in the design phase.

Finally, due to financial restrictions, the PCB and components used will not be of space grade.

5.5 Prototype requirements

Based on the payload requirements in section 5.3 and the prototype considerations made in section 5.4, overall requirements for the prototype made in the project can be deduced.

1. **It shall be possible to send request via CAN messages for: previews, full resolution images, remote capture of an image, deletion of images and for on-board storage statistics.**

   This requirement is identical to req. J and K in section 5.3, but with the interfaces described in section 5.4.

2. **The prototype shall be able to capture an image and store it on onboard memory**

This requirement is identical to req. A in section 5.3.

3. **The prototype shall be able to send a compressed image from storage through to a terminal on a PC, where the data can be viewed.**

   This requirement is based on req. E in section 5.3, but with the change that the image data is to be viewed in a terminal, but it will not be a requirement that the prototype can transmit its messages through the COM subsystem to the ground station.

4. **From CAN, it shall be possible to change the configurations of the payload, in terms of resolution and when images shall be captured**

   This requirement is based on req. F in section 5.3.

5. **The payload shall be able to compress the image lossless such that it can be downloaded from space in less than 1 day**

   This requirement is identical to req. G in section 5.3.

6. **The prototype must be able to detect black frames and not store these**

   This requirement is identical to req. H in section 5.3.

7. **The prototype must be able to generate previews that can be viewed in a terminal**

   This requirement is based on req. I in section 5.3, but with the note that the previews are not needed to be viewable as an image. Instead, it is sufficient to view the data in a terminal.

8. **At least 10 previews must be able to be downloaded from space per pass**

   This requirement is identical to req. I in section 5.3.

9. **The payload must have at least 30 GB of storage space**

   This requirement is identical to req. L in section 5.3.

It is now possible to design the system, based on these requirements for the prototype.

6 | Design

The design of the system is discussed in the following chapter, together with how the different requirements can be achieved through different functionalities.
The flow of the design starts with determining the functionalities, after which the interfaces between those functionalities is described and lastly the physical interfaces between the hardware modules can be designed.

6.1 Functional design

Based on the use case seen in 6.1 and the prototype requirements seen in section 5.5, the following functionalities of the system have been deducted:

- Capture image (the process of capturing an image on a sensor)
- Data extraction (the process of getting that image out of the sensor)
- Preview generation (the process of scaling the image down to a very compact size)
- Pixel intensity counting (the process of counting the frequency of all intensities)
- Temporary storage capability (the process of storing the image until it can be processed further)
- Data compression (the process of compressing the image for a faster download)
- Permanent storage capability (the process of storing the image for when it is needed)
- CAN communication (the ability to communicate via CAN with the rest of the satellite)
- CAN message handling (the process of handling messages from the satellite and sending messages to rest of the satellite)
- Black frame detection (the process of detecting an image containing mainly black pixels)
- Timelapse functionality (the process of taking multiple pictures with a specified time interval)

These functionalities can be seen together with the the desired data to be passed between the functionalities on Figure 6.1. On the figure, the order of execution will be following the arrows in the clockwise direction, starting in the upper left corner with the receiving of CAN messages, and ending with the data being stored on the secure digital (SD) card, where it can be fetched when an image is requested from the ground station.

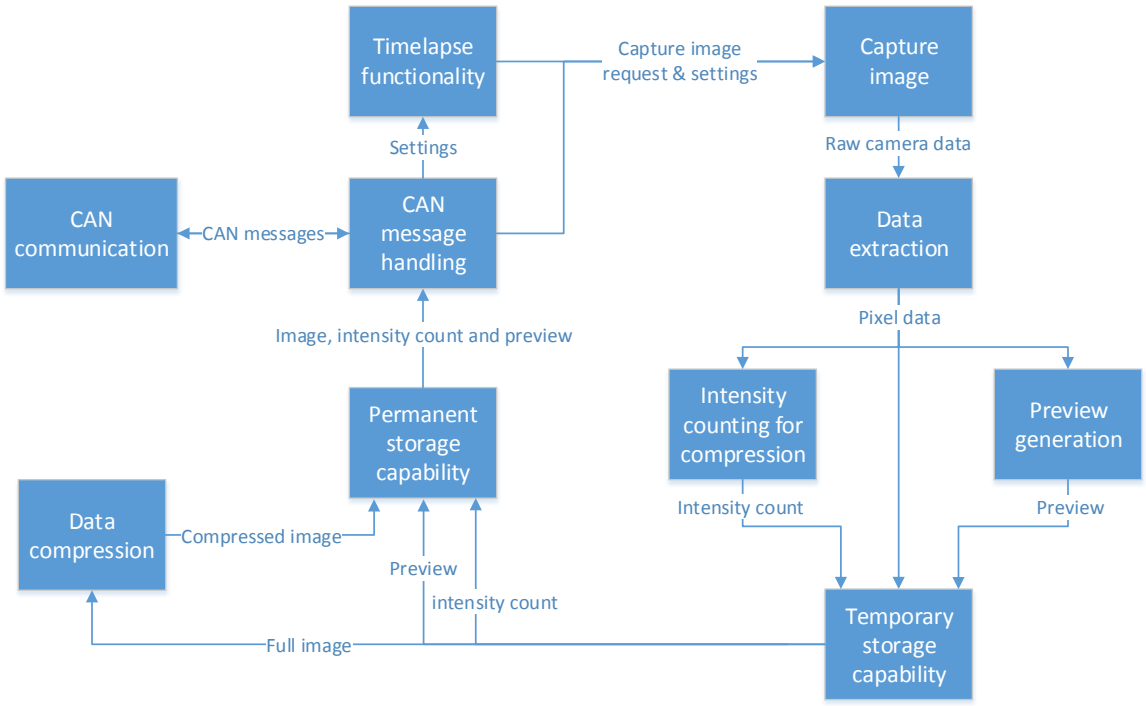


Figure 6.1: An overview of the functionalities of the system and the information flowing between them. Note that all arrows indicate data, except the *capture image*-request.

A more in depth description of the functionalities seen on Figure 6.1 is done in the following.

Capture image

The capture image functionality shall be able to load the intensity of the incoming light and process it as electrical signals. For this functionality a prefabricated camera will be used, since it is beyond the scope of this project to fabricate a camera.

Data extraction

This functionality shall be able to get the image from the camera and pass it along to the rest of the system.

Preview generation

This functionality shall be able to receive the image from the data extractor and scale it down to a size where the COM is able to send 10 pictures to the ground station in one pass, see section 5.5.

Intensity counting for encoding

This functionality is tightly connected to the data compressor. The functionality shall be able to count the frequency of all possible pixel intensities (i.e. how many times they occur in the image) based on the data passed from the data extractor. When done counting, the data compressor shall have access to this information.

Permanent storage capability

This functionality shall be able store the compressed images together with the previews on a storage unit, that can keep the data even when the system is turned off. It shall also be able to check whether or not there is enough free space on the storage unit to store another image, and delete images. It shall be able to store files using a file system. It shall also be possible to fetch data from the storage.

Temporary storage capability

This functionality shall be able to store the image from the extractor and make it available to the data compressor unit. It shall also hold the counted pixel intensities and the preview before before it can be stored in the permanent storage space.

CAN communication

The prototype shall be able to communicate using CAN. Therefore the functionality has to support this, see section 5.5.

CAN message handling

Because communication and handling is two different things, it is necessary to have a message handler that can take the appropriate action depending on the message. This functionality shall also be able to handle outgoing data streams.

Data compression/encoder

The system shall be able to compress images to a size such that even a full resolution picture can be sent to the ground station in less than 1 day, see section 5.5. Huffman encoding is used as the compression scheme since it delivers lossless compression, and uses an algorithm that can be implemented by the group.

Black frame detection

The system shall be able to detect and discard images containing mainly black pixels, i.e. images captured while Earth is not in the field of view, seesection 5.5

Timelapse functionality

The system shall be able to maintain operation i.e. take pictures even without manual commands. This shall be done in a predetermined interval that can be changed from the ground station. For instance, the timelapse can be configured to capture 10 or 20 pictures during an

orbit. The functionality shall be able to either delete/overwrite the oldest image or cancel the capture function if there is not enough room in storage.

6.2 Choice of hardware

Based on the preanalysis, the functional design and choices made throughout this section, the overall design of the payload can be determined with respect to the hardware modules to be used, as can be seen on Figure 6.2.

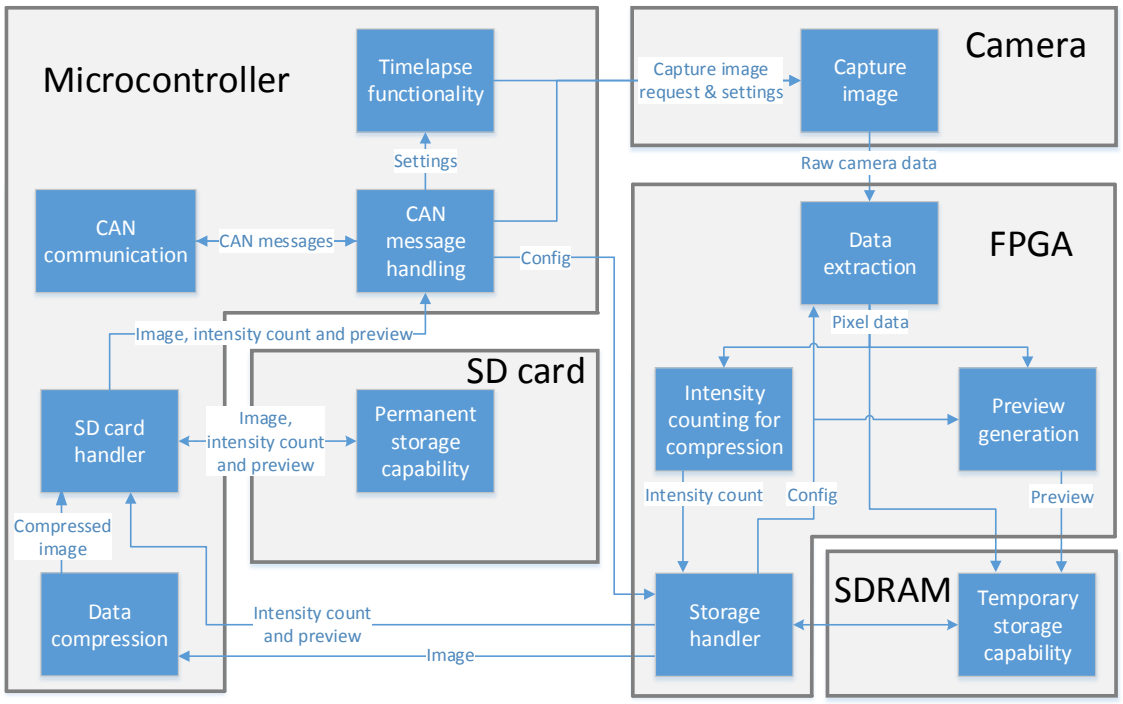


Figure 6.2: A diagram showing how the functionalities are incorporated in the system hardware blocks.

Camera

For the camera unit, there are many considerations to be made in regards to which camera type to choose: Both in regards to the optics and the sensor of the camera, but also the electrical interface, i.e. how the pixel data is obtained. To ensure maximum quality, the ideal solution would be to choose a dedicated image sensor and design the optics to suit the needs for such a device on board a satellite, however this is beyond the scope of this project. Therefore, a prefabricated camera is chosen. Here, the Raspberry Pi camera (picam) is chosen, because of its low cost and the easy accessibility of the camera. The datasheet for the camera can be seen in [FOUNDATION, 2015]. Disregarding the low cost and accessibility the picam also gives a broad spectrum of advantages.

The Rasperry Pi Cameras advantages:

10-bit RAW Data	Makes a higher dynamic range possible than the 8 bit data typically available from cameras. The 10 bit RAW data is not used in the prototype to make a higher dynamic range, but it is converted to 8 bit by the pixel gate, so each pixel takes up one byte of data, which will be described later in section 6.4.
62^o Field of view	In chapter 4, Figure 4.2 shows that a field of view of 62^o from an orbit altitude of approximately 200 km will give an area coverage of $10.000km^2$, meaning that from a more realistic altitude of 600-800 km, it will be possible to see more than Europe without any trouble. The Rasperry Pi camera also comes in a wide range of variants with different optics. This means that a different lens can be used if this turns out to be better than using the supplied lens, thus changing the field of view, see chapter 2.
2592 x 1944 Pixel resolution	This image width and heigh will result in a resolution of about 5 megapixels, which is seen as a good compromise between the level of detail, see section 4.3, and the time to transfer each image, see section 4.4.
MIPI Interface	The MIPI interface is widely used among camera manufactures and this increase the arsenal of usable cameras without having to modify the system to a large extent, if it is chosen to use a different camera.
CMOS	The camera uses CMOS sensors, which consumes a lot less power than the CCD sensor, see subsection 2.2.1. This helps satisfy the strict power demands described in section 5.3.

The Raspberry Pi camera is equipped with two differential data lanes and a differential clock lane, see [Omnivision, 2009, p. 16]. Looking over the MIPI standard, see Figure 3.1, the only physical layer with a dedicated differential clock lane is D-PHY. The only protocol layer for D-PHY is camera serial interface 2 (CSI-2), and the interface has therefore been identified as D-PHY with CSI-2, see Figure 3.2. The picam also has an I^2C interface, which is used for setting up the camera [Omnivision, 2009, p. 16].

FPGA

It is decided that an FPGA is to be used in the payload. This is partly because it is very relevant to include such a device from a learning perspective, but also due to the FPGA's ability to do parallelized computations, as it enables the ability to process data received from parallel data lanes simultaneously. Also, since CSI-2 is based on a high data rate of up to 1 Gbit/s, see section 3.1, a processing unit capable of handling MIPI data with this speed shall be used, and here an FPGA is an option because it's ripple time is lower than the computation speed of a typical microcontroller unit (MCU).

This is also why the intensity counting and the preview generation is performed in the FPGA,

since this can be done in parallel while the image is being loaded from the camera, instead of doing it serialized in an MCU.

Because MIPI is a closed standard, being able to interface with this standard is one of the main concerns when choosing an FPGA. Therefore, an FPGA with MIPI compatibility would be a great advantage, since it would ease the communication between the camera chipset and the FPGA.

The FPGA must be able to communicate with the MCU, in order to forward image data to the MCU and receive configuration data from the MCU.

Another concern is the limitations of equipment from Aalborg University. Due to this, it have been decided not to choose any FPGA chips with BGA connections. This is done due to the difficulty of good soldering of a BGA chip without the right equipment. Also, if the system is to be used in a space environment, BGA chips are not good practice since the solder joints are difficult to inspect for quality checks.

Because of the limitations of the power available on the AAUSATs, see section 1.1, it would be preferable to choose an FPGA with low power consumption. This will allow a longer period of time in which the FPGA can be active.

The FPGA that has been chosen is the model "Lattice MachXO2 - 7000, 4 speed" [latticestore, 2014]. This FPGA has been chosen because it is a low powered FPGA, using 4 mA in static mode[Semiconductor, 2015, p. 46] where i.e. a similar sized SPARTAN 3 uses 15 mA in quiescent mode [Xilinx, 2011, p. 62]. It comes in a TQFP-144 package (i.e. non-BGA) and furthermore it has a free MIPI IP-core available [Corporation, 2014b].

The MachXO2 has the following specifications:

- 114 I/O pins
- RAM
  - Distributed RAM - 54 kbit
  - Embedded Block RAM - 240 kbit
- 6864 logic array blocks
- A total memory of 550 kbit
- 256 kbit flash memory

Further data can be seen in [Corporation, 2014a].

As it can be seen, this FPGA has a storage capacity of 550 kbit which can be programmed through a JTAG interface. This is a great advantage, since it means that the configuration of the FPGA is stored internally, rather than on an external flash chip. This reduces overall design complexity, and gives one less point of failure to account for.

SD card

There is a need to store the images in some type of storage unit. Different storage types are possible, but an SD card is chosen as the storage unit, since it can deliver transfer of at least 2 MB/s and up to 30 MB/s, depending on the card chosen [Association, 2014]. The SD card will be interfaced through the MCU, since data then can be sent via CAN to the rest of the satellite and thus to the ground station, without having to turn on the FPGA. The SD card's small physical size and large storage capacity makes it ideal compared to other storage methods with the same capacity, meaning that many images can be stored on a smaller physical unit.

RAM

Due to the fact that an SD card cannot be written to with the necessary speed i.e. up to 1 Gbit/s see section 3.1, the image has to be stored in a buffer before it can be written. This buffer is implemented by using RAM. The FPGA stores the image in RAM, which is then read by the MCU and stored in the SD card.
The interface between the FPGA and the MCU, as well as the interface between the FPGA and the RAM is chosen to be parallel busses. This is because a parallel bus is capable of having higher transfer rates than a serial bus (assuming identical clock speeds). This is a nice feature to have when transferring large amounts of data in a short timespan, and also because a parallel interface is the default when interfacing RAM. By having a parallel interface between the MCU and FPGA, it is possible for the MCU to interface the stored image data in the RAM through the FPGA, without knowing that the FPGA is even there.

The RAM used will be SDRAM - this is because large quantities of SRAM are difficult to obtain in non-BGA packages, and they also have a high price/storage ratio, as opposed to SDRAM. Due to the high cost of SRAM, the SDRAM will therefore be used in the project.

For the SDRAM a AS4C4M16S is chosen because it supports up to 8 MB storage as 4Mx16bit words and a clock of up to 143 MHz which is sufficient to store the incoming data from the camera. A datasheet for the chosen RAM can be found in [Memory, 2011].

Microcontroller

It is known that FPGAs are very power consuming [Bishop, 2009], although a low power model has been chosen. On top of that, the SDRAM consumes a lot of power as well e.g. 85 mA at 3.3 V [Memory, 2011, p. 19]. Therefore, a dedicated MCU shall also be included in the payload, so that the FPGA and RAM does not need to be turned on all the time. The MCU is responsible for the external communication, by means of CAN. The MCU is also responsible for turning on and off the FPGA and associated RAM, so that they are only on when needed.

It is decided to do the compression and black frame detection in the MCU, since this is easier to implement in a C-environment than on an FPGA.

While the FPGA is responsible for extracting image data from the camera, the setup of the camera is handled by the MCU via I^2C. It must be possible to change the setup from the ground station, and that is also why it would be ideal to handle it on the MCU, since this is

directly connected to the CAN bus, and thus the space link.
The timelapse functionality also has to be handled in the MCU, since this uses the same setup commands as those already used in the MCU. Also, since this functionality is running over a long period of time, it would be ideal to place it in a low-power module.

As previously mentioned, the MCU shall be connected to the SD-card, so that the FPGA does not have to be turned on when sending images to the ground station.

The MCU shall also have a handler, which ensures that the preview and intensity count, received from the FPGA, and the compressed image, received from the Data compression functionality, is sent to the SD-card.

That gives some criteria when choosing an MCU. The MCU must:

- be a low power model
- have a CAN controller
- have an I^2C controller
- have an SD-card controller with secure digital high capacity (SDHC) interface
- have an interface for external RAM

The MCU that has been chosen is a Freescale K10P100M100SF2V2. It has an ARM Cortex M4 core and supports a clock of 100 MHz [Freescale, 2013]. More features includes:

- Operating voltage: 1.7 V - 3.6 V
- 512 KB program flash memory
- up to 128 KB RAM
- Flexbus external bus interface
- Multiple low power modes
- 2 CAN modules
- 2 I^2C modules
- SDHC interface

This MCU is ideal because it supports CAN communication in low power mode, where it uses only 1.7 mA at 3 V (4 MHz system clock) [Freescale, 2013, p. 16]. In this mode it supports CAN speed up to 512 Kbps[Freescale, 2012a, p. 132], which is faster than the 500 Kbps currently used in the AAUSATs see section 1.1.
When more processing power is needed the MCU can be changed to run in normal mode (100 MHz), where the typical power consumption is 47 mA at 3V (25 °C) [Freescale, 2013, p. 15].

These features makes it ideal for the system, though a K60 will be used instead of a K10 in the proof-of-concept model. It comes from the same sub-family, and being a higher-up model, therefore has the same features as the K10, plus some additional ones [semiconductor, 2010, p. 34]. The reason why it is chosen is because it comes with a development board where many of the interfaces have been brought out for easy access.

6.3 Communication interfaces

Now that the physical components have been determined the communication between the blocks and their interfaces can be determined. First the focus will be on how the data is passed around. This can be seen on Figure 6.3.

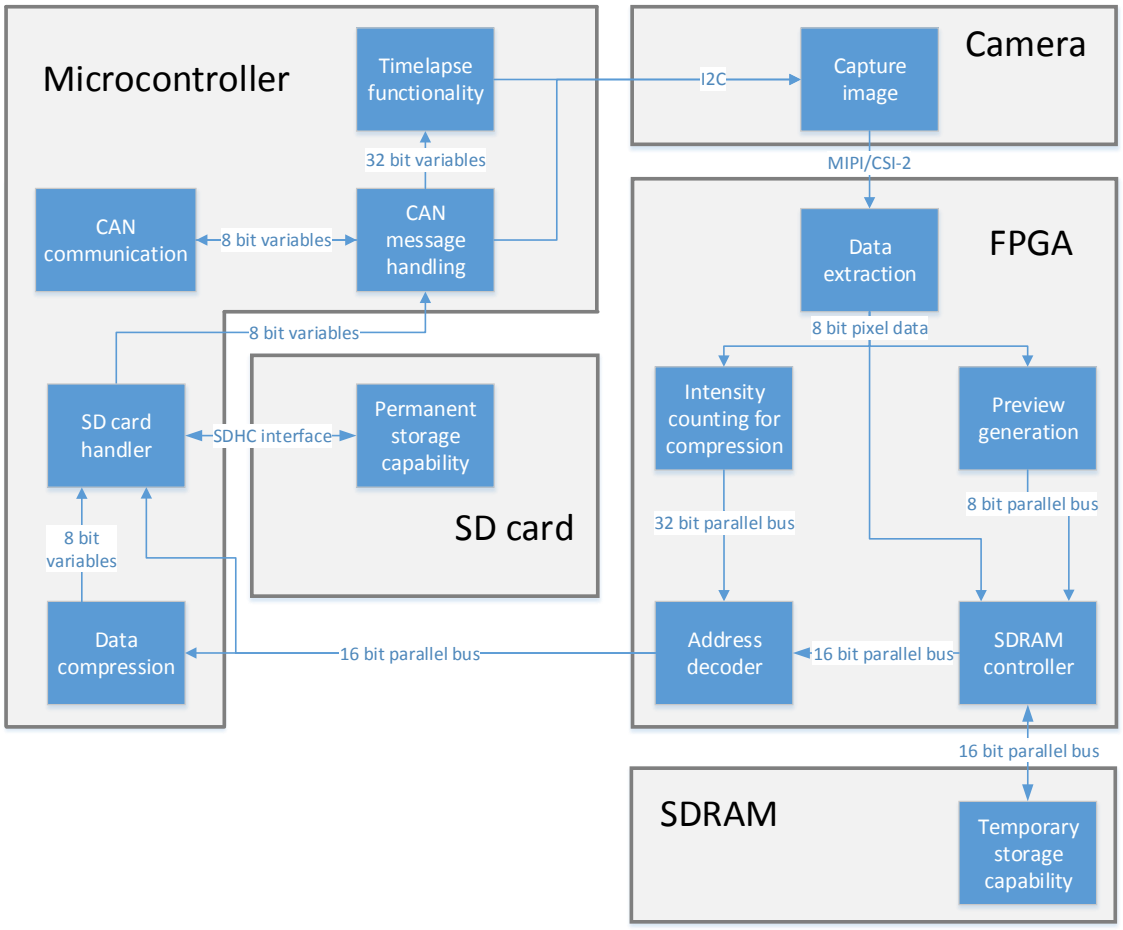


Figure 6.3: A diagram showing how the functionalities are incorporated in the system blocks.

Camera

The picam has 2 interfaces, 1 for setup and 1 for image data. The picam expect its setup as I^2C data. That means the data flowing between the MCU and the picam is transmitted with

an I^2C bus interface. The camera then transmit its data through a MIPI bus thereby requiring a CSI-2 interface.

SDRAM

The chosen SDRAM has a 16 bit data interface therefore the FPGA must comply with this.

FPGA

As mentioned, the image is transmitted on a CSI-2 interface to the data extraction unit. There from the data is transmitted in parallel buses allowing high data rates at lower clock speed. The data will be transmitted as 8 bit parallel data matching 1 pixel per transfer. This data is then transmitted to 3 different units: intensity counter, preview and directly to the SDRAM-controller. After the scaling in the preview functionality, the data is then transmitted on another parallel bus, also in 8 bit width. The intensity counter will need to be able to count to 5 mio. due to cameras ability to take 5 MPx pictures. This means that at least 23 bits is needed to store the number, but since this is later used in the MCU, a standard width is preferred and thus 32 bit is used. Because the counting array and the picture data is stored in two different memory units, an address decoder shall be implemented, to handle the multiplexing of signals, when data is requested from the different memory units by the MCU. The SDRAM-controller will have a 16 bit bus interface to the SDRAM and a 16 bit bus interface to the Address decoder. This is ideal because the interface on the flexbus interface is 16 bit.

SD-card

An SD-card can utilize two types of interfaces. SPI and SDHC. For this design SDHC is chosen since it provides the highest data rate, and because the MCU supports it.

MCU

The MCU has the *Flexbus* connection to the FPGA, the flexbus is 16 bits wide due to the fact that the development board chosen has not mapped all 32 connections available to the flexbus. 16 bits matches the width of the SDRAM so in that matter 1 address to the flexbus will match 1 address in the SDRAM. All communication to the SD-card will be through the SDHC-interface due to its higher data rate. The internal communication bus in the satellite is CAN, so this will be the outgoing interface from the MCU. CAN transfer bytes aka. 8 bit variables. Internally in the MCU the variables are transferred as 32 bit variables due to the MCU architecture being 32 bit.

6.4 Electrical Interfaces

Now that the interfaces have been determined according to how the data shall be transferred, the rest of the signals can be described e.g. clocks, address lanes and so forth. This can be seen on Figure 6.4 and will be described in the following section.

Figure 6.4 is based on Figure 6.2, with a focus on electrical interfaces. This is why the MCU has become one module in Figure 6.4 and the FPGA module Data Extraction from Figure 6.2 has been split into to modules, MIPI bridge and Pixelgate, which functions will be described later in the section.

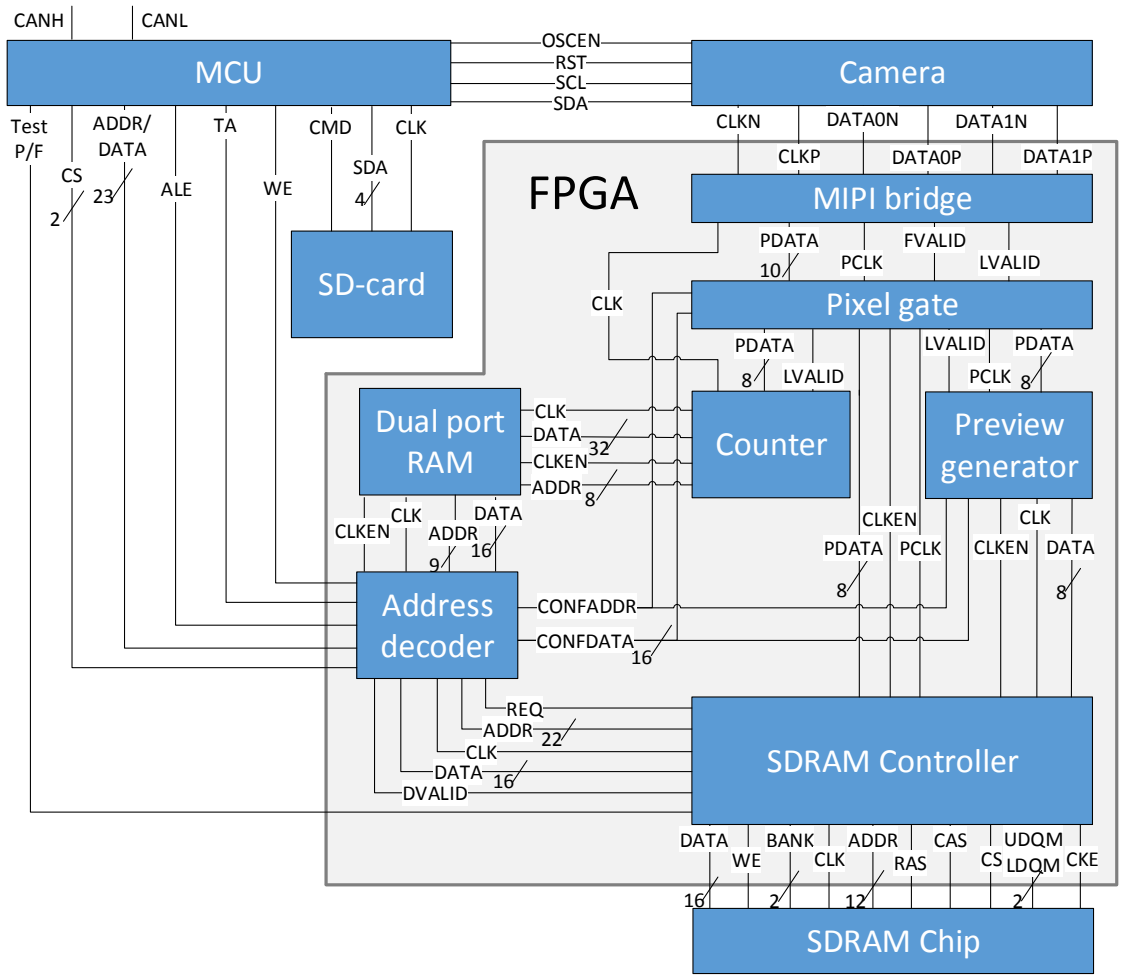


Figure 6.4: Layout of the hardware interfaces between the modules in the system.

MCU interfaces

The external interface connecting the satellite to the system, is the CAN bus, see section 1.2. This consist of a differential pair, CANH and CANL, which is connected to the MCU. The MCU is appointed the job of handling all the external communication to and from the system, because

of its ability to switch to a low power mode with a still functioning CAN module, making it possible to listen on the CAN bus without using a lot of power. Furthermore, it handles the flow of data to and from the SD card, since a prefabricated driver-code is already available for the chosen MCU.

The MCU is used to handle the setup and control of the camera, which needs to be done every time this is powered on and off.

The connection between the camera and MCU is made by using an I²C bus, which consist of a serial clock-, SCL, and a serial data line, SDA, see section 3.2. The camera is turned off when not in use, this is done by the 2 connections reset, RST, and oscillator enable, OSCEN, to ensure minimal power consumption.

The MCU shall also be able to send camera configurations stored on the SD Card to the camera, when it is turned on. Depending on the configurations, the camera will be set to capture image with a certain resolution, white balance settings, exposure settings etc.

To ensure that the MCU is aware of the SDRAM is operating as intended, the connection *Test P/F* between the MCU and the SDRAM Controller is made. When the SDRAM Controller is turned on, it will test the memory chip by writing a standard message to the SDRAM and then read it back. Depending on the results the SDRAM Controller will either send a pass or fail to the MCU. If the result is a pass the MCU will send the configuration to the camera, and the image capture can begin. The Final MCU connections are to the Address decoder.

Address Decoder interfaces

The purpose of the Address decoder is to locate the addresses that the MCU is requesting - this is necessary to have, to be able to distinguish between the FPGAs internal ram and the external SDRAM Chip. The 6 lines connected between the Address decoder and the MCU is the two Chip Selects, CS, the 23 combined Address/Data pins, ADDR/DATA, the Address Latch, ALE, the Transmit Acknowledges, TA and the Write Enable, WE. ALE is used to signal, that the current data on the bus is an address. The Address decoder is connected to the Dual Port RAM, which is a part of the FPGAs internal RAM, through a clock, 9 address pins and 16 data pins. Finally, the address decoder is connected to the SDRAM Controller through a clock, 22 Address pins, ADDR, 16 Data pins, DATA, a request pin, REQ, a data valid pin, VALID, a clock pin, CLK and a clock enable pin, CLKEN.

SDRAM Controller interfaces

The purpose of the SDRAM Controller is to ensure communication between the SDRAM Chip and the other modules. The SDRAM has three access ports 1 for read and 2 for write. The read port is connected to the address decoder through the previously described connections, the write port connects to two First in first out queue (FIFO), which functions as a buffer each holding requests from the Preview generator and the Pixel gate. Both of which is connected by 8 data pins, DATA, a clock, CLK and a clock enable pin, CLKEN. This particular setup makes it possible to change from one clock domain to another, which is important because the pixel clock stops when the camera stops sending pictures. The FIFO also ensures that requests from

each of the two will be buffered until the SDRAM controller is ready to handle the requests. The SDRAM Controller is connected to the SDRAM Chip through 16 Data pins, DATA, 12 Address pins, ADDR, the ability to switch between the 4 banks using the 2 BANK pins, a clock, CLK, a write enable, WE, a row address strobe, RAS, a column address strobe, CAS, 2 Data Input/Output Mask, UDQM and LDQM, and finally a Clock enable, CKE.

MIPI bridge interfaces

The Camera transmits data through two differential data lines to the MIPI bridge. These two are Line 0 that consists of a positive and negative data pin, DATA0P and DATA0N, and line 1 which also consists of a positive and negative data pin, DATA1P and DATA1N. Furthermore, a differential clock is connected, CLKP and CLKN.

The MIPI bridge is an IP core from the chosen Lattice FPGA, which converts MIPI data (the data from the Camera) to 10 bit pixeldata (RAW 10 bit data). The 10 bit pixeldata is transmitted to the Pixel gate through the 10 data pins, PDATA. Additionally, it also sends the Line valid, LVALID, and frame valid, FVALID. LVALID is high when it is sending a line. FVALID is high as long as it is sending a whole image, also called a frame. Finally, the MIPI bridge converts the MIPI clock received from the Camera to a pixel clock, PCLK, that goes high when a new pixel arrives.

Pixel gate interfaces

The Pixel gate is connected to the Address Decoder, SDRAM Controller, Preview generator and Counter. The Pixel gate's function is to convert from 10 bit pixel data to 8 bit data, and determine when the pixel data should be relayed to the other modules. The Pixel gate supplies the pixel clock, PCLK, and the 8 bit pixel data to the Counter and the Preview generator. The MCU can send configurations to the Preview generator and the Pixel gate through the address decoder, using the 16 bit CONFDATA bus, and a configuration address, CONFADDR, alternating between the preview and counter as the data destination.

Counter interfaces

The last block is the Counter which receives data from the Pixel gate and uses the SRAM in the FPGA to store an array. The Counter uses 8 address pins and 32 data pins, furthermore it receives a clock foam the MIPI bridge which is four times as fast as the pixel clock, PCLK, to communicate to the SRAM to be able to read and write from the SRAM multiples times per pixel clock.

Now, the integration of the prototype can begin. The hardware modules determined in Section 6.2: Choice of hardware will each be described in a chapter. Each chapter will start with the requirements for that module, which are based on the functionalities that module is to fulfill, see Section 6.1: Functional design. Hereafter will follow the integration of the module, closing off each chapter with a module test.

7 | Camera

In this chapter, the workings of the Raspberry Pi camera chosen as the camera unit in the prototype will be described. Focus will be on determining the registers needed to be set in the camera, in order for it to capture images as desired.

7.1 Requirements

Based on the usecase, see section 5.1, prototype requirements, see section 5.5 and the functional design, it is possible to set up requirements for the camera modules. The requirements are based on the functionality that the camera module has, here the "capture image" functionality, see section 6.1. This functionality can be formulated as the following requirements:

Capture image

The camera functionality shall be able to:

1. Change the configuration of the camera via I²C

   See Section 6.2: Choice of hardware

2. Capture an image based on the configuration

   See Section 5.1: Use case design and Section 6.1: Functional design

3. Transmit the captured image through the MIPI bus

   See Section 6.2: Choice of hardware

Now that the requirements for the module have been defined, it is possible to do the integration, which will be done in the following sections.

7.2 Reverse engineering the Raspberry Pi camera

The Raspberry Pi camera is chosen for the project, as it is readily available at the university. This means it will be easy and quick to get a new camera, if it should stop working.

Unfortunately, there is currently no complete documentation available for the camera, so most of the register setup must be done by reverse engineering the camera and the interaction between the Raspberry Pi and the camera.

Hardware interface

The schematic for the camera module is not being provided by the Raspberry Pi Foundation, so the first step is to determine the pinout of the camera. Most of it can be found in the schematic for the Raspberry Pi, see Figure 7.1

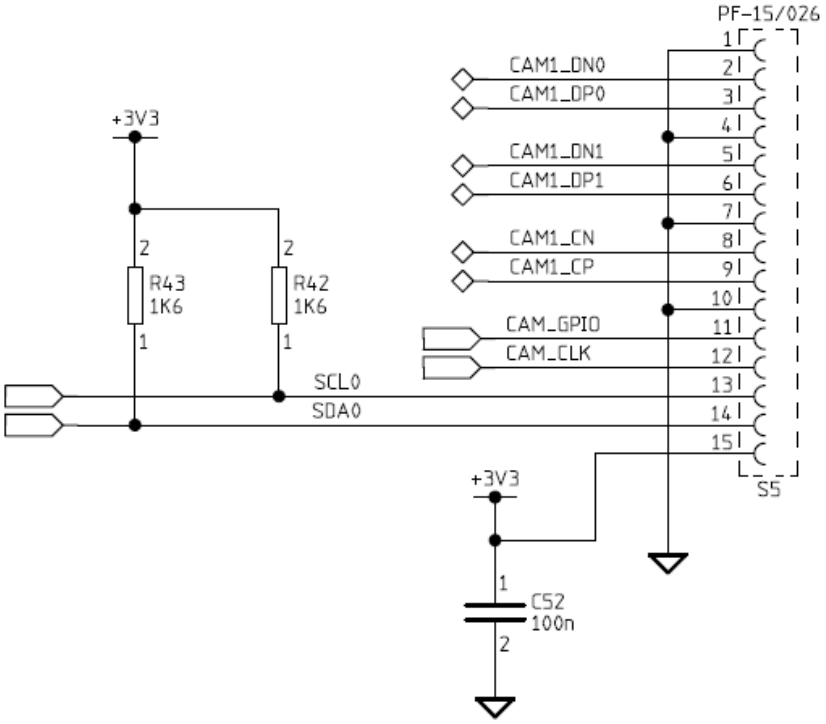


Figure 7.1: Pinout of the Raspberry Pi camera connector [source: http://www.rs-online.com/designspark/electronics/knowledge-item/r-pi-ffc-connectors]

The majority of these connections can be recognized from the CSI-2 specifications, and a preliminary table can be put together of the camera pinout. Only two connections remains unknown, see Table 7.1.

Pin number	Name	Description
1, 4, 7 and 10	GND	Supply Ground
2	CAM1_DN0	D-PHY Differential Data Lane 0
3	CAM1_DP0	
5	CAM1_DN1	D-PHY Differential Data Lane 1
6	CAM1_DP1	
8	CAM1_CN	D-PHY Differential Clock
9	CAM1_CP	
11	CAM_GPIO	Undocumented
12	CAM_CLK	Undocumented
13	SCL0	I²C bus
14	SDA0	
15	+3V3	Supply 3.3V

Table 7.1: Camera pinout

As the pinout is now known, a breakout board can be made to connect between the camera and the Raspberry Pi. The board provides easy access to the high speed data lines via BNC connectors, and pinheaders for listening on the I²C bus.

The PCB can be seen on Figure 7.2.

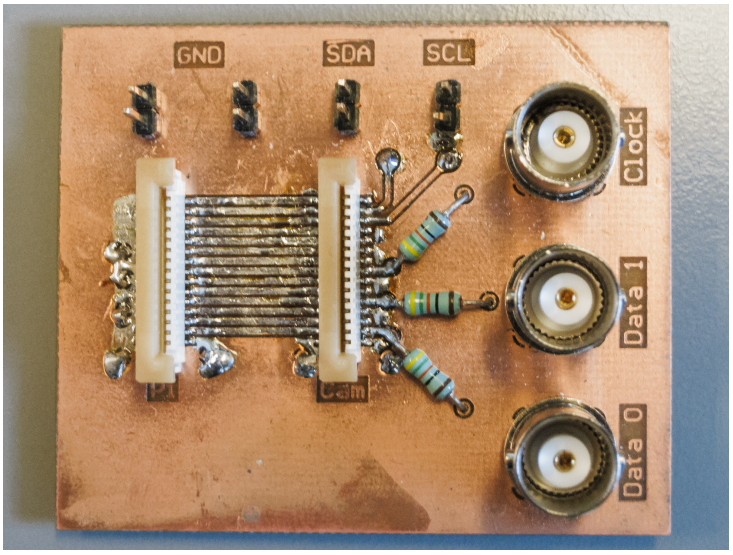


Figure 7.2: Breakout board for monitoring camera signals.

Software/Protocol interface

Now step two can begin, the actual reverse engineering. The Raspberry Pi will be commanded to take pictures in various resolution, while the I²C bus is monitored with an oscilloscope, a Rigol DS1102E. This oscilloscope features a "Deep memory"-Mode, where up to a million samples can be captured, and afterwards saved as a .csv file. The test setup can be seen on Figure 7.3

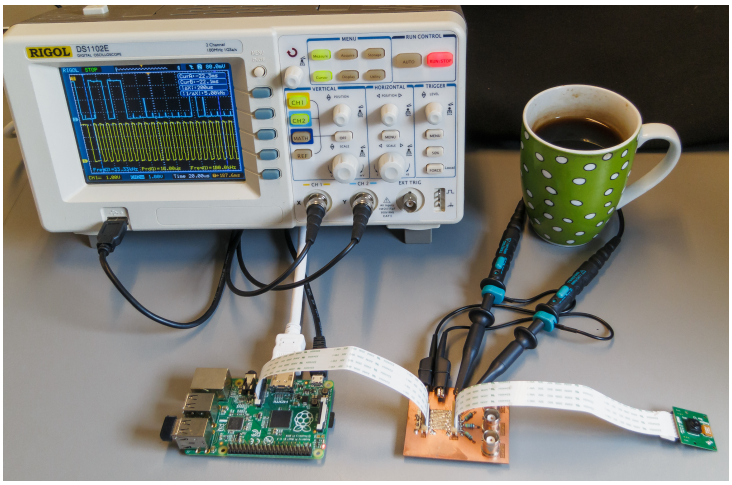
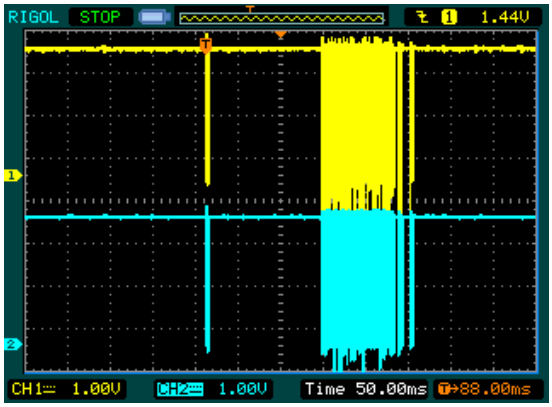


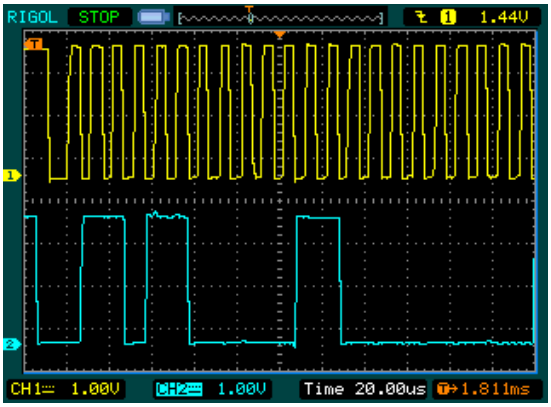
Figure 7.3: Test setup for monitoring the I²C bus

A screenshot of a capture can be seen on Figure 7.4a. As seen on the screenshot, the Raspberry Pi sends a few short data burst, followed by a long burst. The short bursts contains very little data, and are easily decoded manually, by watching the waveforms.

A zoomed in view of the initial short burst can be seen on Figure 7.4b.



(a) Captured I²C data



(b) Zoomed in short burst

Figure 7.4: Image captures containing I²C burst from the setup on Figure 7.3

Decoding a couple of transfers in this data results in the following bit sequences:

Binary:	011 0110	0	0	0000 0001	0	0000 0000	0	0000 0000	0
Hex:	0x36	0	0	0x01	0	0x00	0	0x00	0
Description:	address	write	ack	data	ack	data	ack	data	ack

Translation: "Write 0x01, 0x00 and 0x00 to I²C address 0x36"

Binary:	011 0110	0	0	0000 0001	0	0000 0000	0
Hex:	0x36	0	0	0x01	0	0x00	0
Description:	address	write	ack	data	ack	data	ack

Translation: "Write 0x01 and 0x00 to I²C address 0x36"

Binary:	011 0110	1	0	0000 0000	1
Hex:	0x36	1	0	0x00	1
Description:	address	read	ack	data	ack

Translation: "Read from I²C address 0x36" (0x00 is read from the slave)

Binary:	011 0110	0	0	0000 0001	0	0000 0011	0	0000 0001	0
Hex:	0x36	0	0	0x01	0	0x03	0	0x01	0
Description:	address	write	ack	data	ack	data	ack	data	ack

Translation: "Write 0x01, 0x03 and 0x01 to I²C address 0x36"

According to [Omnivision, 2009, p. 77] I²C slave address of the camera is 0x6C for writes, and 0x6D for reads. As the I²C standard calls for 7 bit addresses, followed by a read/write bit, looking at the provided addresses in binary gives the answer why the decoded address is different from what the datasheet claims:

Hex:	0x6C	0x6D
Binary:	0110 1100	0110 1101

The datasheet is simply interpreting the read/write bit as part of the address.

According to [Omnivision, 2009, p. 23], a software sleep can be commanded, by writing 0x00

to register address 0x0100. This matches the first transfer. The first two bytes are the register address, and the last is the data to write.

Starting the configuration with a sleep command makes sense - there is no need for the camera to be active until it has been fully configured. The next two transfers reads back the written data from the last transfer. Why this is done is unknown, but it might be for making sure the camera is working correctly.

Now that the transfer format, has been undestood, it is clear that the last of the four transfers must be "Open register 0x0103, and write 0x01". According to the datasheet, this is the command to reset the sensor, again something which makes sense to do during initialization, to make sure all registers are in a well known state.

The long databurst now needs to be decoded. Because of the large amount of data, this will need to be automated. The oscilloscope can save the captured data to a Comma Separated Values (CSV) file.

Now Sigrok, an open source signal analysis software suite will be used (`http://www.sigrok.org`). This package contains a tool for reading a .csv file from a logic analyser, and finding I²C packets. As the data has been captured with an oscilloscope, and not a logic analyzer, the data will need to be reformatted in a way sigrok understands.

The oscilloscope saves the captured data as the voltages of the two channels, together with a timestamp. Sigrok expects only 1's and 0's. The conversion from voltages to binary values is easily done using Matlab. The data is imported, and the clock and data channels identified, see Figure 7.5

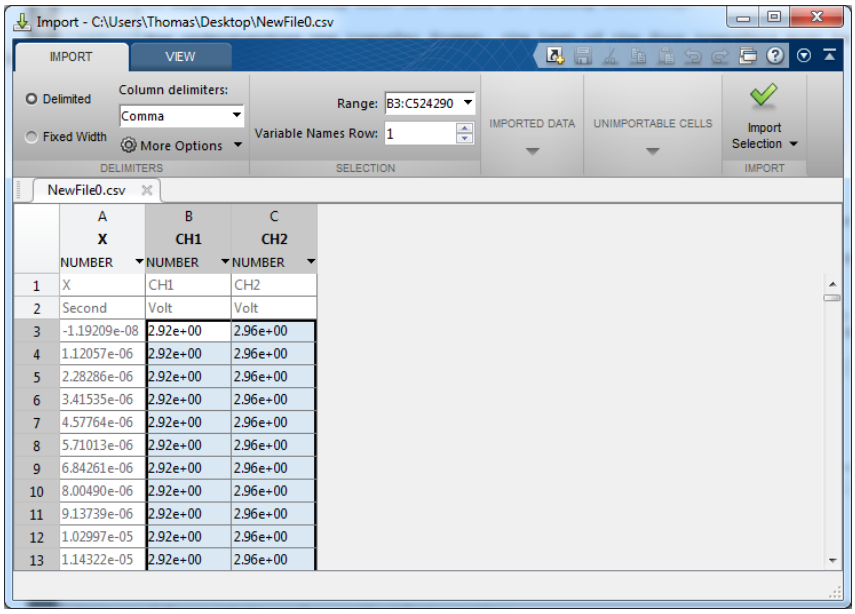
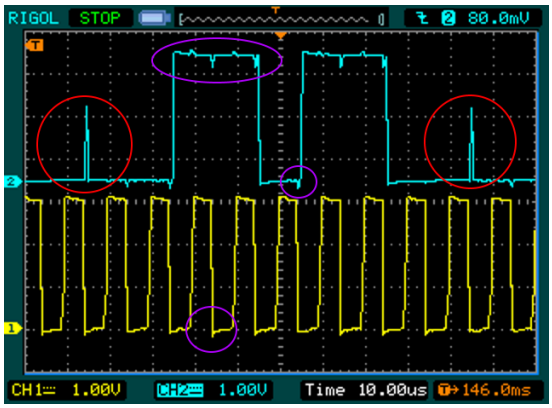
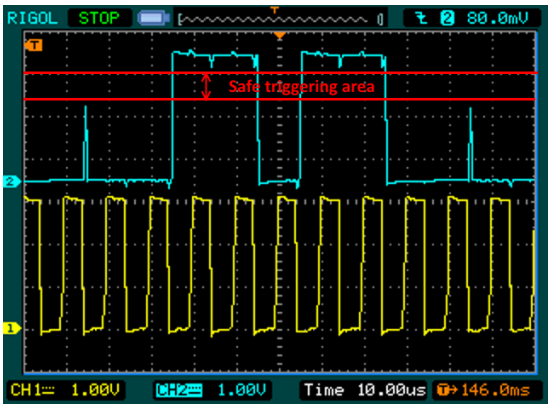


Figure 7.5: Importing oscilloscope data.

The thresholds for when to interpret the data as a '1' or a '0' is determined by observing the data. Care must be taken to not capture "false positives" caused by ringing. Also, when the master releases the bus, to let the slave send an acknowledge, a small spike occurs, which must also be filtered out. The voltage range between 2.0 V and 2.8 V is free of artifacts, and 2.7V is chosen as the threshold. See Figure 7.6



(a) Image capture showing ringing(purple) and bus release spikes(red)



(b) No false readings occur between the red lines

Figure 7.6: Determining the thresholds for a '1' or a '0'

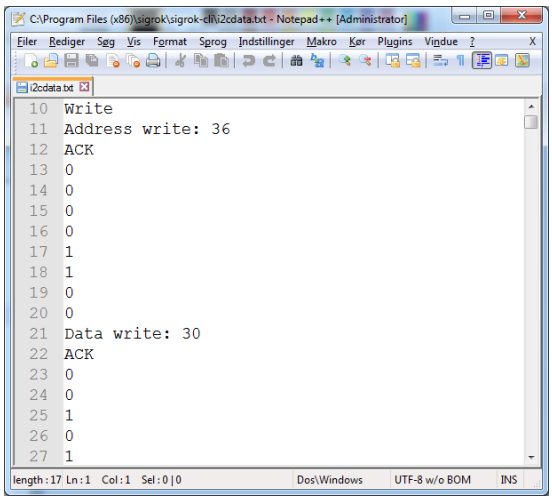
The voltages can now be converted to 1's and 0's, and saved as a new .csv file using the following commands:

```
CH1(CH1<2.7)=0
CH2(CH2<2.7)=0
CH1(CH1>2.6)=1
CH2(CH2>2.6)=1
DATA = [CH1, CH2]
csvwrite('i2cdata.csv',DATA)
```

Listing 7.1: Matlab commands

The generated .csv file can now be processed by Sigrok. This will analyse the data and create a text file with the I²C transfers, see Figure 7.7a.

Now that the actual I²C data has been obtained, it has to be converted to a format that can be easily included in the microcontroller software. To do this, a small C program is written, that takes the sigrok data in and outputs a C header file, containing an array with the data to be written to the camera, see Figure 7.7b



(a) Sigrok output data **(b)** Generated C header file

Figure 7.7: Screenshots describing output data and the generated C header file

The above procedure with the decoding of I²C packages and generation of registers based on these is repeated for the relevant image resolutions, to get the register set for each. An example of such a register set can be seen in Table 7.2.

Register Address	Description	Default data	Configured data
0x3034	SC_CMMN_PLL_CTRL0: Bit[6:4]: pll_charge_pump Bit[3:0]: mipi_bit_mode 0000: 8 bit mode 0001: 10 bit mode Others: Reserved to future use	0x1A	0x1A

Table 7.2: Example of a register description from the datasheet. Note that the Default configuration is "Reserved to future use", which is rather odd.

With the register setups at hand, the camera can now be powered on without the Raspberry Pi. During the testing, an MSP430 MCU development board is used to send the I²C data, as it is a simple architecture to get up and running, and is configured for 3.3V operation just like the camera. An Arduino could have been used, but this would require I²C level shifters, as the Arduino uses 5V logic levels.

As an initial test, the sleep command is sent to the camera and then read back. Unfortunately, the camera ignores the I²C transfer, and appears completely dead.

Probing around the camera board with the oscilloscope, it turns out that the oscillator providing the camera clock is not running. This must be caused by one of the two undocumented pins on the camera board. It is presumed that the CAM_CLK pin could be responsible for activating the oscillator, and pulling this pin to a logic high, in fact makes in oscillator starts up. At the same time an LED on the camera board lights up.

With the oscillator now running, the sleep command is sent to the camera again, again without result. According to [Omnivision, 2009, p. 23: reset], using a hardware reset pin is recommended, even though the camera supports software reset.

As it is very likely that the camera board designers have read the same section, it is almost certain that the last pin (CAM_GPIO) controls this. Pulling this pin to a logic high, the camera starts responding to I²C commands.
Sending one of the captured data bursts causes the CSI bus to become active and start streaming data. The camera is now ready for connecting to the MIPI bridge in the FPGA.

7.3 Test

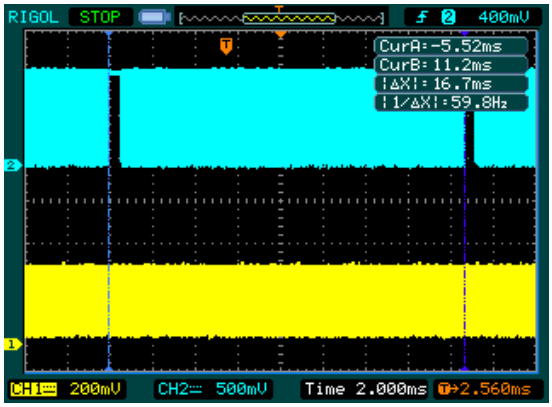
The camera is now ready to be tested, to verify that it meets the requirements previously set for it.

To test the camera, the equipment used for the reverse engineering is used again, see section 7.2.

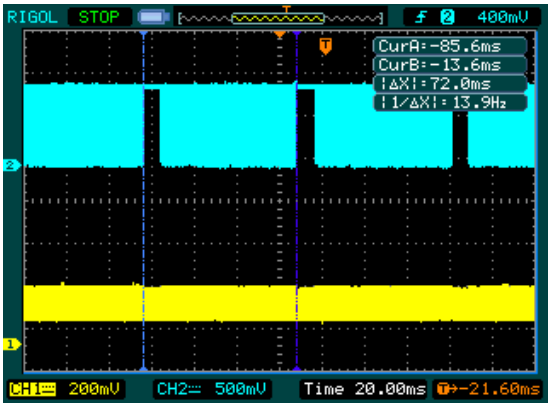
Two different configurations are sent over the I²C bus from the MSP430 development board:

- 640x480 (VGA) at 60 frames per second
- 2592x1944 (5 Mpx) at 14 frames per second

The oscilloscope is connected to the MIPI clock line, and one of the data lines, and the waveforms are observed, see Figure 7.8 a and b.



(a) 640x480 60Hz



(b) 2592x1944 14Hz

Figure 7.8: MIPI data (blue) and clock (yellow) lines when the camera is active

As seen on the Figure 7.8a, the VGA configuration results in data bursts, repeating every 16.7ms. This translates to ≈ 60 Hz, which is the expected frame rate for the VGA configuration.

With the 5 Mpx configuration (Figure 7.8b), the burst repeat rate is 72 ms, which again is consistent with the expected 14 fps.

It can therefore be concluded, that the camera configuration can be changed via the I²C bus. Furthermore, the camera is clearly transmitting the configured number of frames per second on the MIPI bus. This means the test requirements: changing the camera configuration, capture an image and transmit it through the MIPI us, is met.

The integration of the camera has been described and the camera requirements has been tested. In the following chapter the MCU be described and implemented.

8 | Microcontroller

This chapter's main focus will be on the MCU's tasks. The MCU controls when and how the different subsystems are turned on and initialize communication with the FPGA and the camera. The MCU receives commands from the CAN system and executes various commands according to ground controls needs.

8.1 Microcontroller requirements

Based on the functionalities that the microcontroller has to facilitate, as determined in section 6.1, the following requirements for the MCU can be established:

Capture image

- Send camera configurations to the camera via I²C

  See Section 6.3: Communication interfaces

- Power on and off the camera

  See Section 6.4: Electrical Interfaces

CAN communication

- Send and receive CAN messages

  See Section 6.1: Functional design

CAN message handling

The following requirements all originate from the use case diagram on Figure 5.1. The communication to the system happens through CAN, where the following messaged handlings has been found:

- Initiate the capturing of an image when requested.
- Display files and storage space in a terminal when requested.
- Delete a chosen image on the storage unit when requested.
- Receive camera configurations and setup the camera accordingly.
- Display image data in a terminal when requested.
- Receive settings for the timelapse functionality.

Data compression/encoder

- Detect frames containing too many pixel intensities below a pre set threshold.

  See Section 6.1: Functional design

- Create Huffman signatures based on the counter array

  See Section 6.1: Functional design

- Replace picture data with Huffman signatures

  See Section 6.1: Functional design

- The compressed image must take up less space than the original

  See req. 5 in Section 5.5: Prototype requirements

Timelapse functionality

- Request an image capture with a set interval autonomously

  See Section 6.1: Functional design

- Toggle between not capturing an image or deleting the oldest image if there is not enough room for more images

  See Section 6.1: Functional design

- The interval of which images are captured must be changeable

  See req. 4 in Section 5.5: Prototype requirements

The different requirements based on the functionalities in section 6.1 has been given. Now the MCUs main functionalities can be described and determined.

8.2 Overall microcontroller functionality

The overall functionality is described via a graphical interpretation on *Figure: 8.1*.

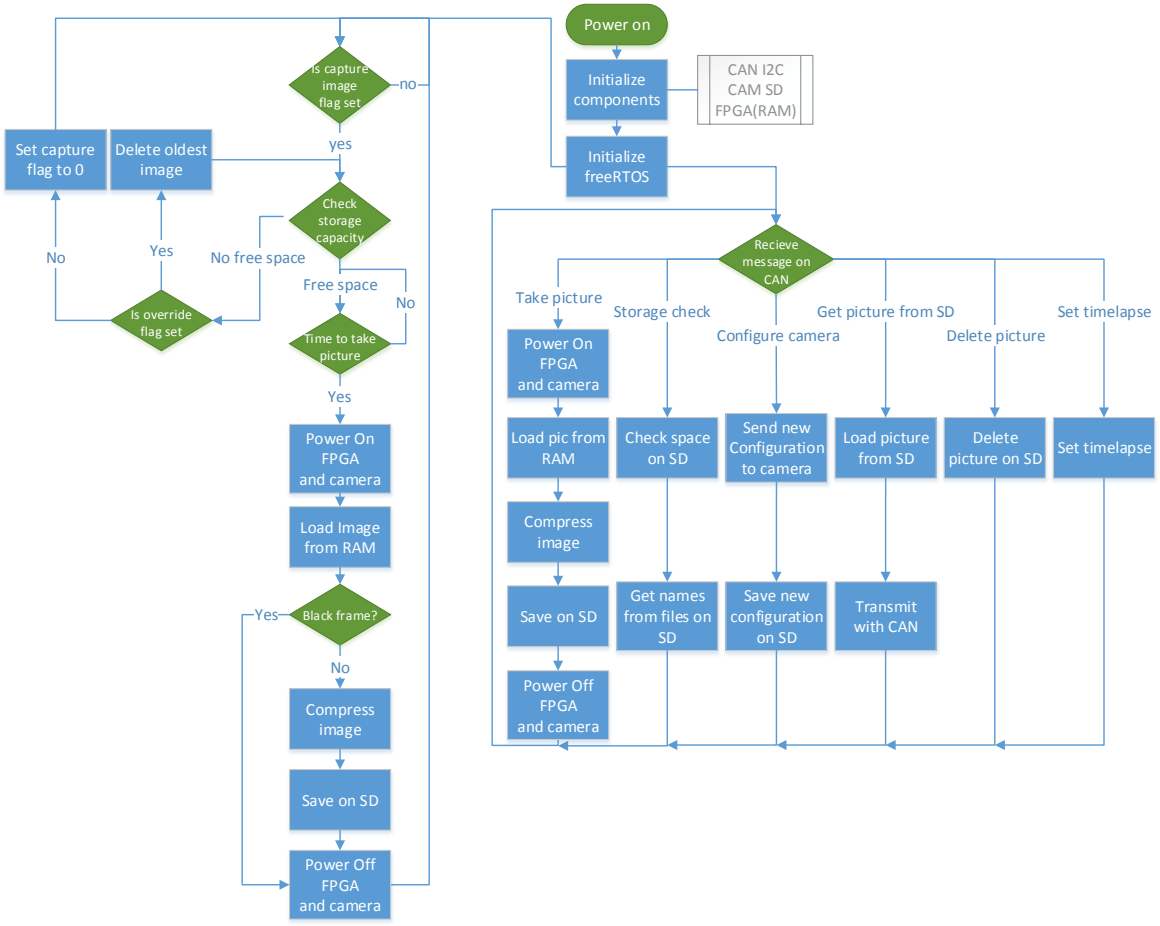


Figure 8.1: General overview of the Micro controller functionality

When the MCU is powered on, it will initialize the internal oscillators and the CAN module. Following a successful setup, it continues to initialize its operating system, FreeRTOS, and start listening for commands from the CAN bus.

When the MCU is operational it will utilize FreeRTOS to switch between two tasks, which can be broken into:

- Receive a message on the CAN network involving
  - Manually taking pictures and compressing them
  - SD-card handling functions
    - Delete picture
    - Storage check
    - Get Picture from SD
  - Setting up configuration for camera and/or timelapse timing

- Take a picture in accordance to a predefined timing schedule

Receiving messages on CAN

The MCU will be responsible for communication with the FPGA and the camera. Therefore the MCU have a wide variety of function concerning picture handling. Using the previously described list as basis, the MCU will be executing one of the following functions, preceding that the commands have been received from the CAN network.

Take picture is the manual picture function of the satellite, making it possible for ground control to activate the camera and capture an image. The progress for taking a picture starts with powering up the FPGA and camera. After this, configuration data is sent to the FPGA through the Flexbus, and through I^2C to the camera. When the configuration is done, the camera starts capturing images. One of these pictures, depending on FPGA configuration, will be loaded into an external RAM due to the high amount a raw pixel data. From the RAM, the image will be loaded into the MCU where the image will be compressed, using a huffmann algorithm and then saved on an external SD-card. The functions finishes with powering down the camera and FPGA.

Storage check performs a scan of the SD-card, creating a list of the folders and files contained on the SD-card. This function makes it possible for ground control to receive a status and create an overview of the current storage capacity.

Configure Camera will be sending configurations to the camera i.e. setting the needed registers to use when capturing an image. To make it possible to change the configuration from the ground station, the configuration files will be saved on the external SD-card.

Get picture from SD is used for transferring images to the terminal. The MCU will load the image from the SD-card, preparing the image by breaking it into packages and sending it by CAN.

Delete picture will be a crucial function for maintaining control over the SD-card. The function will permanently delete pictures on the SD-card giving room for newer taken pictures.

Set timelapse will set a flag causing the second task of the MCU to trigger. The timelapse function will described in detail in the next section. This specific function will configure how often the task is going to execute i.e. timing and duration.

This concludes the first task of the MCU. As described these functions form the general behaviour of the MCU and will be how ground control communicate with the system.

Timed picture functionality

The Timed picture functionality is utilized for taking a series of picture. This function gives ground control the possibility for capturing images when the satellite is out of reach. The configuration of this function will be preformed by the previous described commands.

Capture image flag set is checking if the timelapse flag has been set, described in the ***Set timelapse***. This task will be waiting for this flag since initialization of the MCU and will be running parallel to the previous handling functions described.

Check storage capacity will, if the capture image flag has been set, preform a storage check to ensure enough space for a picture. Different possibilities will be made available to handle the event of a full SD-card. An overwrite flag can be set, granting the MCU the option of overwriting the already stored images. If this flag has not been set, and no space is available on the SD-card, the *capture image flag* will be set to 0 and no image will be captured.

Time to take picture is the final stage in the timelapse function before it enters a ***Take picture*** state. When reaching this function the MCU have been granted enough storage and will be waiting for a predefined time to capture the image. When the time arises it will trigger the ***Take picture*** function, which was described earlier in this section.

The main tasks of the MCU has now been described. The underlying submodules will be described in the following sections.

8.3 Other functionalities

In the following section the operating system *FreeRTOS*, the *CAN* protocol, the I^2C protocol and the *Flexbus* interface will be explained and parts of the implementation will be illustrated.

FreeRTOS

As can be seen on Figure 8.1, a Real Time Operating System (RTOS) is used in the microcontroller. The reason for this is that it should be able to receive and transmit CAN messages, while also capturing images, performing storage checks etc. It could be done by using interrupts, so that when a message is received over the CAN bus, an interrupt occurs and the Interrupt Service Routine (ISR) for receiving CAN messages is run.

This is not done, because the group finds it an interesting challenge to utilize an RTOS, and an RTOS enables more features such as semaphores, priorities and critical regions, than inter-

rupts do. FreeRTOS is also chosen because it is used in other satellite subsystems, and it has been ported to the selected MCU. An example project demonstrating FreeRTOS exists in the Integrated Development Environment (IDE) used for the processor, and is used as a basis for getting an RTOS up and running.

An example in which 3 tasks are created can be seen in Listing 8.1. Here, the function *xTaskCreate* takes the function that is to be run in the task as the first parameter, along with a task name, a stack size for that tasks, any parameters to be passed to the task, and finally an optional task handle. The task handle can be used, when one desires to change priority of the task, delete it etc.

```
1  if (xTaskCreate(can_TX, "CANTX", configMINIMAL_STACK_SIZE, (void*)NULL, 3, NULL) != ↩
       pdPASS)
2      {
3          for(;;){} //Stop program if task cannot be created
4      }
5
6  if (xTaskCreate(can_RX, "CANRX", configMINIMAL_STACK_SIZE, (void*)NULL,3,NULL) != ↩
       pdPASS)
7      {
8          for(;;){} //Stop program if task cannot be created
9      }
10
11 if (xTaskCreate(blinky_task, "blinkGPIO",(unsigned short) 50,(void*)(uintptr_t)CAM_GPIO↩
       ,3,NULL) != pdPASS)
12     {
13         for(;;){} //Stop program if task cannot be created
14     }
15 vTaskStartScheduler();
```

Listing 8.1: Creating three FreeRTOS tasks with priority 3. Notice how the last task takes *CAM_GPIO* as a parameter when creating the task.

5 tasks are created using FreeRTOS:

- CAN RX (responsible for receiving CAN messages)
- Capture image (capturing an image when requested)
- Timelapse (takes photos with a set interval)
- Get picture from SD (prints picture data to the terminal from the SD card)
- Storage check (prints a list of files on the SD card and the storage space available in a terminal)

All tasks except CAN RX and Timelapse are waiting as default. This is because the tasks are not to be run unless requested. This is done by letting those task wait for a binary semaphore each, also known as mutex. These semaphores are given by the CAN TX task, depending on what message it receives, which can be seen in Listing 8.2. Since the semaphore is binary, the semaphore can only take a maximum value of one, meaning that it is not possible to "spam" the system with requests to capture images, and halting the entire system.

```
void can_RX(){
    while(1){
            result = can_recieve(ch2, &temp);
            if(result == 0){
                if(ch2[0] == 'p'){                          // take picture
                    xSemaphoreGive(Take_picture);
                }
                else if(ch2[0] == 's'){                     // storage check
                    xSemaphoreGive(Storage_check);
                }
                else if(ch2[0] == 'I'){                     // download image to ground
                    xSemaphoreGive(Get_picture_from_SD);
                }
                else if(ch2[0] >= '0' && ch2[0] <= '9'){    // set timelapse
                    taskENTER_CRITICAL();
                    timelapse_interval = atoi(ch2);
                    taskEXIT_CRITICAL();
                }
                else{
                    // message not understood, ignore it
                };
            }
    }
}
```

Listing 8.2: The CAN RX task, showing how semaphores are given depending on what message is received

The timelapse task is enabled as default, since this task has to capture an image with a set interval, and can be seen in Listing 8.3. The timelapse interval can be changed when configuring the timelapse via CAN, thus changing how often images are captured.

```

uint32_t timelapse_interval = 60; // capture an image every 60 seconds as default

void timeLapse(){
    TickType_t xLastWakeTime;

    while(1){
    vTaskDelayUntil(&xLastWakeTime, timelapse_interval*1000*portTICK_PERIOD_MS);
    CaptureImage();
    }
}
```

Listing 8.3: The FreeRTOS delayUntil function

As with any RTOS, the utilization should be considered, to ensure that the Central Processing Unit (CPU) is not overloaded, and that all deadlines set for the tasks are met.

However, there are no strict deadlines to be met in the prototype. Therefore, it is not necessary to calculate the utilization, since it depends on the completion time of all tasks, and the period of which the tasks are called, which is not known. Each task is however given a priority, matching its time interval. The CAN RX task has the smallest period, and is therefore assigned the highest priority. The timelapse task is given the next highest priority since this runs continuously but less often than CAN RX. The remaining tasks are given the same, but lower priority.

CAN

Likewise, example projects exist for the CAN bus, demonstrating the use of this bus. However, some work has to be done in order to extract the functionalities needed in this particular project, and to create higher-level functions (APP-functions) enabling an easy use of the CAN bus, without bothering the programmer of the main application with the low-level workings of the CAN functionality.

The three functions made for the CAN functionalities can be seen in Listing 8.4.

```
1 int can_init(uint32_t rx_indentifier, uint32_t bitrate,
2             uint32_t tx_timeout, uint32_t rx_timeout);
3
4 int can_recieve(uint8_t *data, uint8_t *msgsize);
5
6 int can_send(uint8_t *data, uint8_t msgsize, uint32_t tx_identifier);
```

Listing 8.4: The three CAN handler functions

Note that in the initialization function, only the RX and identifiers (i.e. the CAN "addresses" of the system) as well as the bitrate and timeouts are to be specified, to ease the use of the functions. Bitrates can be selected from a list of predefined rates, ranging from 125 kHz to 1 MHz.

Likewise in the receive function, a pointer to where the data is to be stored and a pointer to store the length of the message is parameters to the function.

Finally, the transmit function takes a pointer to the data to transmit, together with the length and the TX identifier, i.e. the "address" to send the data to.

I²C

Since I²C is used to setup the camera, it is not needed to make functions that receive data via I²C in the MCU. Thus, only two driver functions are made, which can be seen in Listing 8.5. Note that the address that the function I2CSend takes, is the register address, and not the I²C address of the camera. This is because this address is always to be used when sending data to the camera, and is therefore not needed as a parameter for the APP-programmer to specify, since it will always be the same.

```
1 uint8_t CameraI2CInit();
2 uint8_t I2CSend(uint16_t addr, uint8_t data);
```

Listing 8.5: The two I²C handler functions

The functions are based on an example project in the IDE used to program the MCU, from which the core functionalities and initializations have been taken. It is therefore left up to the functions taken from the example project to ensure that I²C data is sent as expected, as described in section 3.2.

Flexbus

As opposed to the other functionalities described in this section, the Flexbus is not implemented based on example projects, but is instead based on registers found in the microcontroller datasheet.

When setting up the Flexbus, two registers are of particular interest. The FB_CSCR0 (*Chip Select Control Register*) and FB_CSMR0 (*Chip Select Mask Register*), since it is these two which determine the workings of the bus. The setup of these registers can be seen in Listing 8.6.

```
1  // Configure FB control register
2      FB_CSCR0 = FB_CSCR_BLS_MASK     // right justified mode, data is sampled on ↩
                                          fb_ad[15:0]
3               | FB_CSCR_PS(2)        // 16-bit port
4               | FB_CSCR_ASET(0x0)    // assert chip select on first clock edge after ↩
                                          address is asserted
5               | FB_CSCR_WS(0x0)      // 0 wait states from CS assert to internal TA ↩
                                          occurs
6               | FB_CSCR_BSTR_MASK;   // enable burst reads
7
8                                      // Configure FB address masking:
9      FB_CSMR0  = FB_CSMR_BAM(0x070F) // Mask address space
10              | FB_CSMR_WP_MASK      // Enable write protect
11              | FB_CSMR_V_MASK;      // Enable chip select 0
```

Listing 8.6: Setup of the FB_CSCR0 and FB_CSMR0 registers

The Flexbus supports 32-bit addresses and data bus widths, but it can be multiplexed, to support other widths, see Figure 8.2 and [Freescale, 2012b, p. 736]. It is desired to set the multiplexing to use a 16 bit data bus, since this is the word length of the RAM, see section 6.2. The data is right-justified so that it is present on bits [15 : 0], which is done because Flexbus pins [32 : 24] are not routed out to physical pins on the selected MCU breakout board. Thus, the address used in the Flexbus is 24-bit, and the data is 16 bit, as can be seen in section 6.4.

Port size and phase		FB_AD			
		31–24	23–16	15–8	7–0
32-bit	Address phase	Address			
	Data phase	Data			
16-bit	Address phase	Address			
	Data phase	Address		Data	
8-bit	Address phase	Address			
	Data phase	Address			Data

Figure 8.2: The combinations of address and data bus widths for FB_CSCRn[BLS] = 1, depending on mode of operation

The Flexbus has a run mode clock frequency of up to 50 MHz [Freescale, 2012b, p. 186]. Based on a timing diagram of the Flexbus read burst [Freescale, 2012b, p. 758], that can also be seen on Figure 10.13, a burst read can be completed in a minimum of 5 clock cycles, where the last

clock cycle in a burst read can be the same as the beginning of the next burst read. This means that the maximum 32 bit reading rate will be 12.5 MHz (= 400Mbps), but at this speed, there is little time for the MCU to perform other computations than handling the Flexbus.

Another aspect to be considered is the memory map of the Flexbus. As can be seen on Figure 8.3, the Flexbus support a variety of addresses. Here, 0xA000 0000 is chosen as the base address, to prevent the MCU from accidentally executing data in the external RAM as code. At the same time the addresses chosen to match the SDRAM is also set to be write protected. Because of this 2 chip selects are used, so that another memory mask can be used to pass config to the FPGA.

0x6000_0000–0x7FFF_FFFF	FlexBus (External Memory - Write-back)	All masters
0x8000_0000–0x9FFF_FFFF	FlexBus (External Memory - Write-through)	All masters
0xA000_0000–0xDFFF_FFFF	FlexBus (External Peripheral - Not executable)	All masters

Figure 8.3: The addresses supported by the flexbus [Freescale, 2012b, p. 170]

If a 32-bit word is requested, the Flexbus automatically handles transmitting 2 16-bit words over the flexbus, and merge them to a single 32-bit word. An example of how data can be fetched via the Flexbus can be seen in Listing 8.7, where the data is simply accessed by means of a pointer to where it is stored.

```
#define FB_START_ADDRESS 0xA0000000
volatile uint32_t* data = FB_START_ADDRESS; // volatile to ensure pointer points out ↩
    into RAM
int i;

for(i = 0; i < 256; i++){
        printf("\n\rData %i : \t %10x", counter, *data); // print data as HEX value
        data++; // increment data pointer
}
```

Listing 8.7: An example of how memory can be accesed through the Flexbus. 256 32-bit words are fetched and printed in a terminal in this function.

FreeRTOS, CAN, I²C and Flexbus has been described and illustrated. Now the activities in the functionality *Capture image* will be explained.

8.4 Capture image

In the following section will the capture image function be described. A UML sequence diagram, describing the overall activities needed to capture an image, seen from the MCU, can be seen on Figure 8.4.

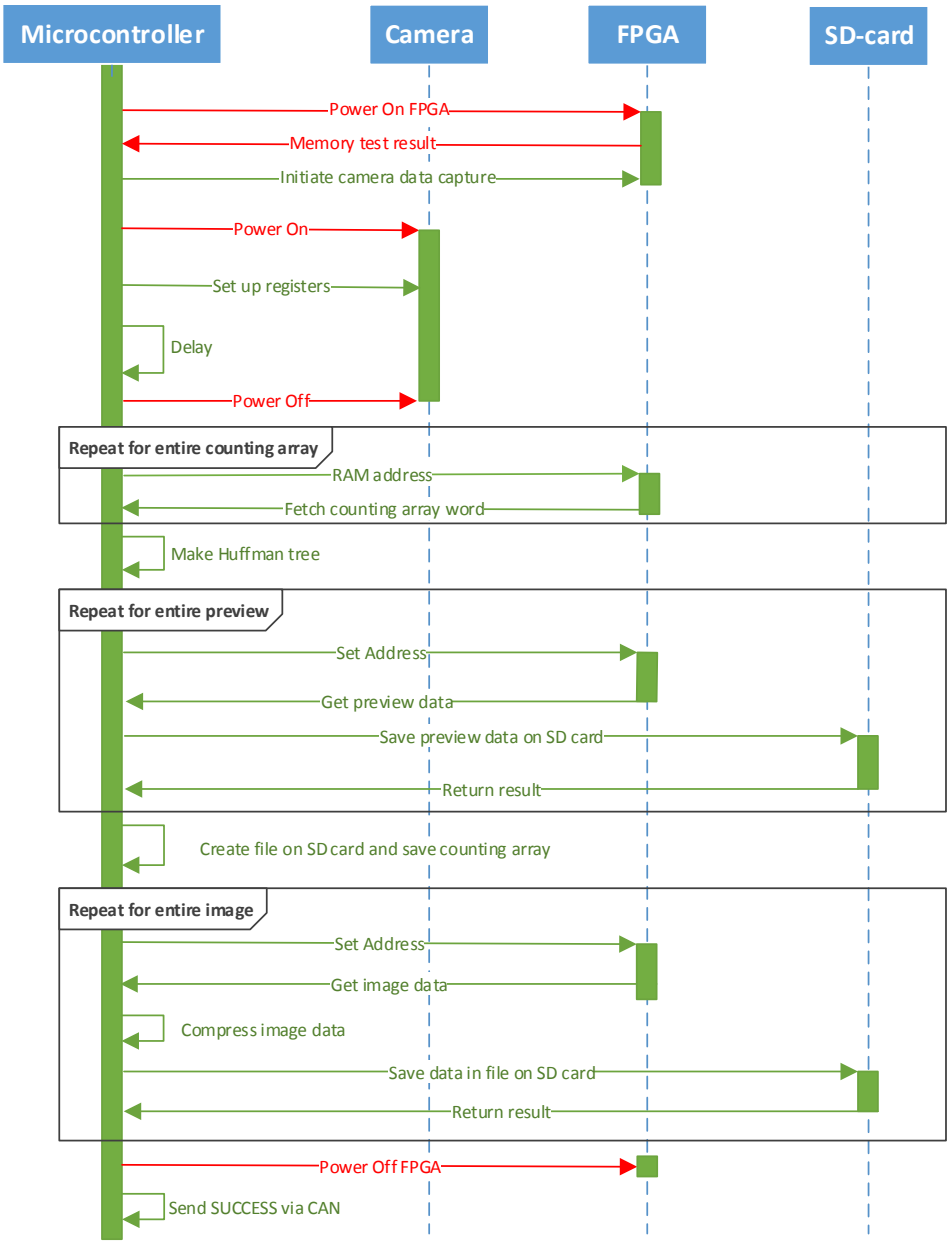


Figure 8.4: A sequence diagram showing the signal and dataflow between the elements of the system, when capturing an image. Red arrows indicate commands and green ones indicate data

In order to save power, it is decided that the FPGA and camera shall always be turned off, unless it is used to capture an image. This is because they are more power consuming than the MCU, just like they are not needed to be permanently on for the system to function.

The sequence diagram will be described in the following.

- To start, the FPGA is turned on, and a configuration is sent to the FPGA. This configuration contains information about the width and height of the image to be captured, and the number of frames the pixel gate is to skip before it starts relaying image data.
- Configuring the camera is also necessary which is done by setting up registers in the camera. The exact registers to be set can be seen in section 7.2.
- When the camera has been set up, the camera starts transmitting images to the FPGA. While this happens, the MCU enters a delay for a specified period of time, after which it powers off the camera, since the camera data has been saved in RAM by the FPGA.
- Now, the MCU can begin receiving the counting array from the RAM through the FPGA. The counter array is fetched from the RAM one 16-bit word at a time. Based on the counter array, the Huffman tree is generated. The counter array is stored in the header of the file that is created for the image, so it can be decompressed on earth.
- After this, the preview is fetched from RAM, the same way as with the counting array. When the preview is loaded from the RAM, it is saved in a file on the SD-card. This is done so that every time the MCU receives a word of the preview, this is appended to the file on the SD card.
- Hereafter the full resolution image can be fetched from RAM. Using the counting array, this is compressed, as described in section 8.5. The compressed image part is appended to file the SD card.
- When the full file has been written to the transmits an image via CAN to signify success.

Note that how the FPGA distinguishes if it is the counting array, the preview or the full image that is requested from the MCU is determined by the address specified.

A snippet from the code used to capture an image can be seen in Listing 8.8. The code shows how a file is created on the SD card with the counting array, and how the image is compressed and saved in the file.

```
1
2 SD_FileCreate(KOMP, counterArrayBytes, 1024, &nameNumber);  // create a new file, where↩
      we write the counter array as the first data
3
4 fileOffset = 1024;
5
6 // while the size of the image read is smaller than the total image size, read data, ↩
    compress it and save it on the SD card
7 while(ChunkCounter*dChunkSizeRam < IMAGESIZE){
8
```

```
9       ReplacePictureData(imageBytes, dChunkSizeRam, HuffmanSignature[1], HuffmanSignature↩
            [0], EncodedImage, &iEncodedByte);
10
11      // append the compressed data to the file with the correct offset.
12      SD_FileWrite(KOMP, fileOffset, EncodedImage, iEncodedByte, &nameNumber, fileOffset+↩
            iEncodedByte);
13
14      fileOffset += iEncodedByte; // increase offset with the length of data just written
15
16      ChunkCounter++;
17
18      *imageWords += dChunkSizeRam;   // increase file pointer
19      imageBytes = (uint8_t*)imageWords; // update 8-bit pointer
20  }
```

Listing 8.8: A snippet from the capture image function

In the snippet above, a 16-bit pointer called *imageWords* is used to point to the image data in the SDRAM, since the word length of the Flexbus and SDRAM is 16 bit. However, since the functions in the code expect 8-bit pointers, the pointer is also casted to an 8-bit pointer, called *imageBytes*. In the snippet, a file is created on the SD card in the folder containing the compressed images. The counter array is stored in the file, as shown in line 2. Hereafter, the file offset, keeping track of how much data has been written to the file, is incremented.

A while loop is made, ensuring that the code runs until an entire image is written in the SD card. In the loop, a picture chuck is compressed by using the *ReplacePictureData* function. The compressed data, saved in EncodedImage, is then written to the SD card, in the file that the counter array was saved in. The file offset is increased, and the 16-bit pointer that points to the image in the SDRAM is increased with the chunksize. Also, the 8-bit pointer that is used in the code is also updated, based on the new 16-bit pointer address. This is repeated until the entire image has been compressed.

Now that the overall workings of the capture image functionality has been described, it is relevant to look at how the functions called by capture image are implemented. In the following section, the compression algorithm will be explained.

8.5 Compression and Black frame detection

When designing the compression algorithm to be used, some considerations has to be made. The two primary factors determining how well a compression algorithm functions are:

1. Speed
2. Compression ratio

Based on these two factors, it is not optimal to generate the Huffman tree as a binary tree. This is due to the fact that when the signatures are to be generated using the Huffman tree, the tree has to be traversed from top to bottom for each leaf (i.e. each light intensity in the photo). This is because the position of the leaf in regards to the branches determines the signature, see section 2.5. Traversing the tree takes time, and the greater the tree is, the greater this time factor gets. Therefore, implementing the algorithm in such a way that the signatures would be created simultaneously with the tree would be advantageous, such that once the tree had been generated, so had all the signatures.

In regards to the second criterion, it should also be kept in mind that it is necessary to transmit the Huffman tree to the recipient, for him to be able to decode the messages. Note that this has to be done for each image, since the Huffman tree will be different from image to image. Having to send the information in the Huffman tree takes up part of the data that can be downlinked per pass. Thus, having to send less data for the recipient to be able to decode the images is desired, since the Huffman tree should also be counted in when calculating how much storage space the compressed file takes up.

Thus, the design constraints for the Huffman tree have been determined.

Overall functionality

As the compression functionality will be reading the whole image, it is decided that the Black frame detection is to be done at the same time. The number of pixels with intensities below a certain threshold is summed, and compared with a predetermined maximum number of dark pixels. If the maximum is exceeded, the image will be rejected, thereby implementing the black frame detection functionality.

The compression algorithm consists of two functions - one for making the Huffman signatures, and one for replacing the picture data with the signatures. A chunksize is necessary to ensure the compressed data does not take up too much memory before it is sent to the SD card. The chunksize will therefore determine how much of the image is compressed at a time.

A flowchart describing how the compression algorithm works can be seen on Figure 8.5.

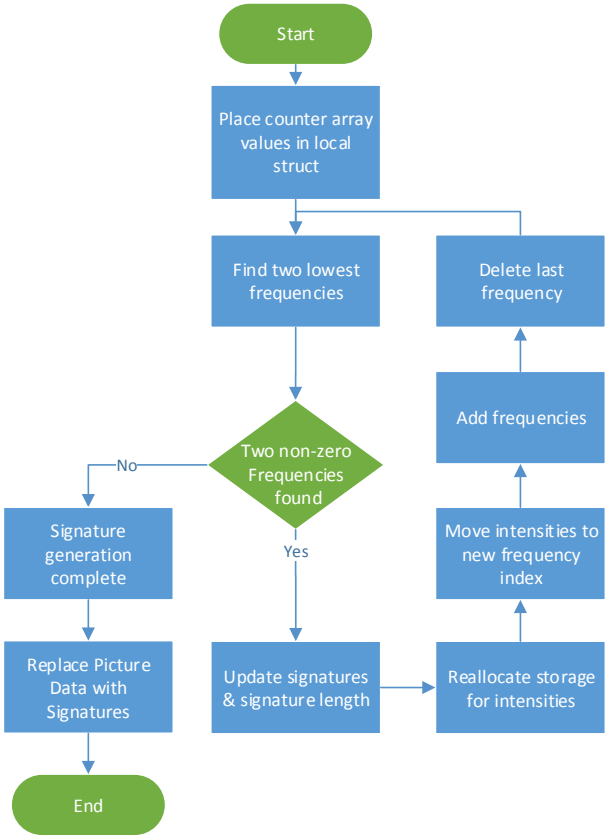


Figure 8.5: A flowchart displaying the workings of the compression algorithm

The overall workings of the code are based around the procedure described in section 2.5. Here, the two lowest frequencies are identified, and these are placed as the lowest leaves in the tree. However, instead of just generating the tree, the signatures are generated simultaneously. Space is reallocated to make room for the intensities that are put in the tree, and the two frequencies are summed, to make their parent node. It is in fact now possible to delete one of the two nodes, since their contribution to the tree has been established, and they are thus no longer needed.

When the algorithm can no longer find two non-zero frequencies, the top of the tree has been reached, and the compression algorithm is therefore done. The only thing left to do, is to replace the original image data with the generated signatures.

In practice, the implementation of the code is not done as a classical binary tree due to a desire of avoiding the traversing of the tree. The approach used to generate the tree is described in the following section.

Signature generation

To generate the Huffman tree and thus the signatures, a counting array is used, an example of which can be seen on Figure 8.6. This counting array contains information about the frequency of all intensities in the image, i.e. how many times each light intensity appears in the image.

The counting array is generated in the FPGA to save computation time, since this can be done in the FPGA simultaneously while the FPGA stores the image in the RAM.

A struct array is made, where each struct contains an intensity and its corresponding frequency, thus the array contains this information for all the intensities. Note that the intensities in the struct are made as an array of pointers, initially containing only a single intensity, but it is capable of pointing to more than one intensity. The fact that pointers are used will be useful later, when the tree is constructed.

Also, a two-dimensional array is made, containing the signature and signature length for all the intensities.

To explain the algorithm, an example is used. In this example, there are 5 intensities (0 to 4) with a corresponding frequency that can be seen on Figure 8.6.

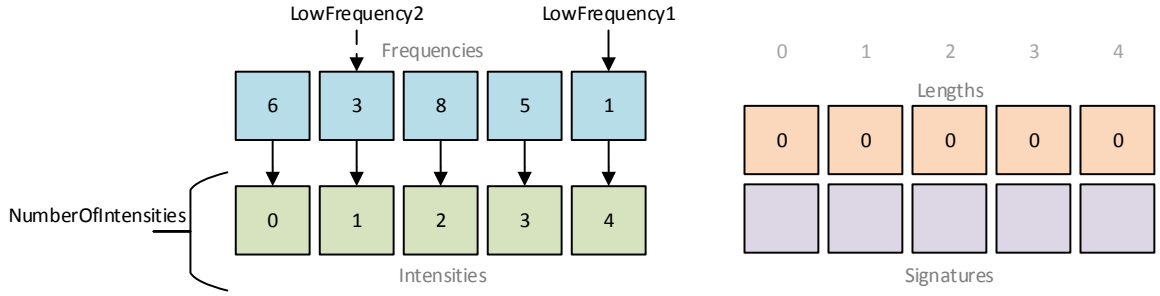


Figure 8.6: The starting point of the compression algorithm. To the *left*, a struct is seen containing the frequencies and pointers to the intensity corresponding to each frequency. A two-dimensional array (*right*) contains the signatures and the length of these.

The first step is to identify the two lowest, non-zero intensities. The reason why they need to be non-zero is that it is not necessary to assign a signature to intensities with a frequency of zero. These are called *LowFrequency1* and *LowFrequency2*, where LowFrequency1 is the smallest of the two.

The intensities that this frequency points to, i.e. intensity nr. 4 for the case of LowFrequency1, are assigned a binary *"1"* in the MSB position of the signature, and the signature length is incremented. The intensities that correspond to the frequency that LowFrequency2 points to are assigned a *"0"* and the length is incremented.
Now, the two frequencies are summed, and placed in the frequency that occurs first in the array, in this case LowFrequency2. Also, the intensities that were the last of the two, here LowFrequency1, are moved to the intensities of the first, i.e. LowFrequency2. This can easily

be done by utilizing the pointer array previously described, since the next pointer in the array is simple assigned to point at the address of the intensity to be moved. Please note that after this is done, the last of the two, i.e. what LowFrequency1 is pointing at, is deleted since it is no longer needed. This means that the struct array gets gradually smaller, and thus the amount of frequencies to be searched to identify LowFrequency1 and LowFrequency2 is reduced each time, thereby greatly improving speed.

This results in the configuration seen on Figure 8.7, where the new LowFrequency1 and LowFrequency2 have been identified.

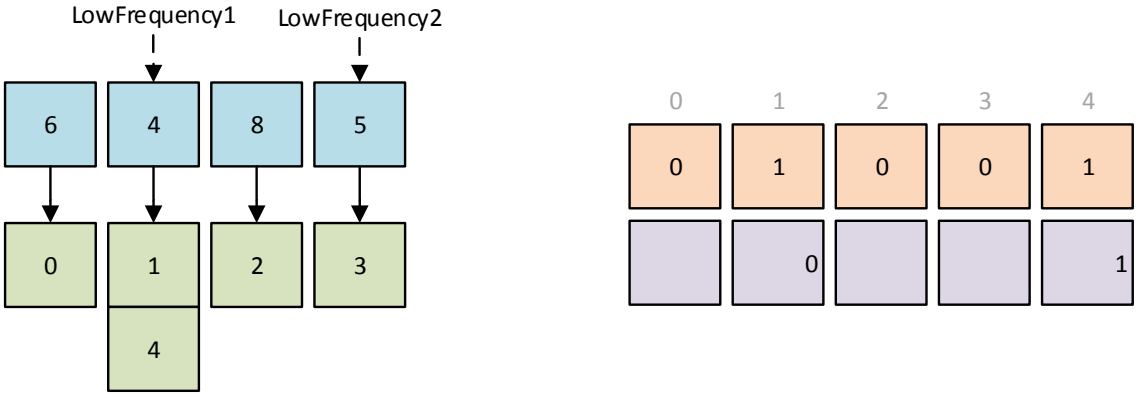


Figure 8.7: The two structs after the algorithm has run once. The signatures have been updated and the two smallest frequencies have been summed.

Now the procedure is repeated for these two frequencies. But, since LowFrequency1 now contains two intensities, (1 and 4), both of these have to have their signature updated. This is done like before, meaning the algorithm puts a binary "1" in the MSB position of the signatures that LowFrequency1 points to. As before, the frequencies are summed and placed in the first of the two entries, and the intensities of the last entry is put behind the intensities of the first, as can be seen on Figure 8.8.

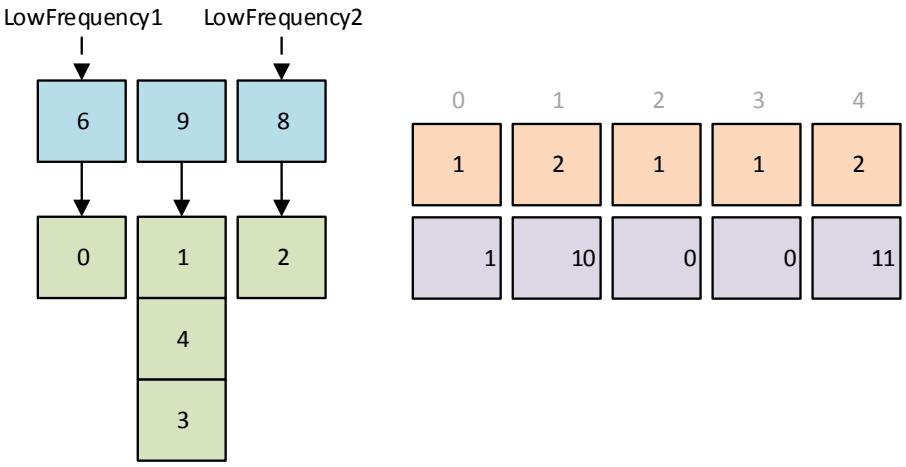


Figure 8.8: The two structs after the algorithm has run twice.

This is done two more times, see Figure 8.9 and Figure 8.10. The final state, Figure 8.10, contains a single frequency, that should be identical to the sum of all the original frequencies.

It can also be seen that all intensities have obtained a signature, and that the length of the signature depends on the original frequencies of each intensity. For instance, intensity 1 and 4 have the longest signatures, since they have the lowest frequencies. Also, intensity 2 has the largest amount of zeros in its signature, since it has the highest frequency of the five.

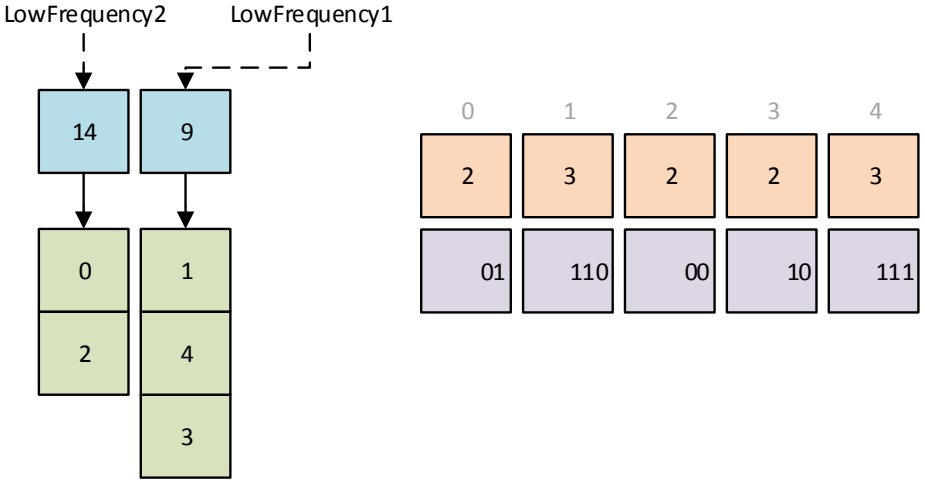


Figure 8.9: The two structs after the algorithm has run three times.

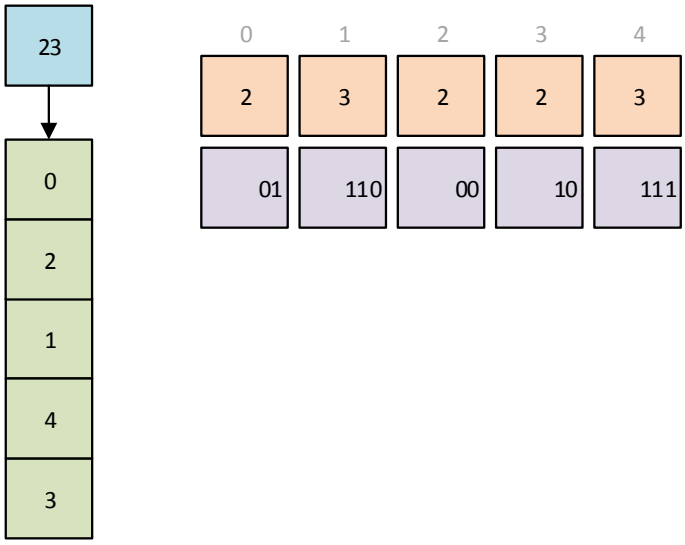


Figure 8.10: The final state when the algorithm is finished after 5 runs.

In Figure 8.11, a Huffman tree has been made from the same intensities as used in the example. Here, it can be seen how the frequencies determine each intensity's placement in the tree - the higher frequency, the higher placement in the tree. Note how the signature for each intensity is made based on the edges of the tree, and how these matches the signatures seen in Figure 8.10.

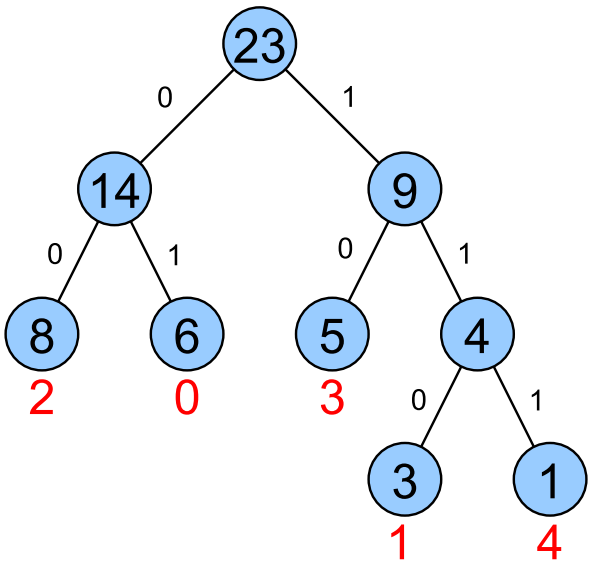


Figure 8.11: The Huffman tree equivalent of the finished algorithm, showing the link between the signatures and the intensities placement in the tree. Red text indicates intensities, and the number in each node is the frequency

Function prototypes for the two functions used in the compression algorithm, namely *Huffmantree()* and *ReplacePictureData*, can be found Listing 8.9, where the inputs and outputs for both functions can be seen.

```
int HuffmanTree(unsigned int aCounterArray[], unsigned int aHuffmanSignatures[], unsigned int aHuffmanLengths[]);
/*  inputs:
        aCounterArray:  the counting array
    outputs:
        aHuffmanSignatures: Array to store the Huffman signatures
        aHuffmanLengths: Array to store the length of the Huffman signatures
    returns:
        0 for success, 1 for fail
*/


void ReplacePictureData(uint8_t aPictureData[], uint16_t dChunkSizeRam, uint32_t aHuffmanSignatures[], uint32_t aHuffmanLengths[],
        uint8_t *EncodedImage, uint16_t *iEncodedByte);
/*  inputs:
        aPictureData:   image data array
        dChunkSizeRam:  How many pixels are to be compressed in this chunk
        aHuffmanSignatures: Array to store the Huffman signatures
        aHuffmanLengths: Array to store the length of the Huffman signatures
    outputs:
        EncodedImage:   array of all the encoded byte of this image chunk
        iEncodedBytes:  number of encoded bytes in this chunck
    returns:
        none
*/
```

Listing 8.9: The prototype for the HuffmanTree function and the ReplacePictureData function, with inputs and output listed

Applying the algorithm to a real image

The encoding algorithm is applied to the image in Figure 2.7, to demonstrate the generation of signatures for a real image. From the algorithm, signatures are obtained for all the intensities in the image. This can be seen in Table 8.1, where each intensity in the image and its corresponding frequency is listed, along with the signature of each intensity.

Intensity:	**0:**	**1:**	**3:**	**6:**	**9:**	**12:**	**43:**	**51:**
Frequency:	1	2	3	1	1	1	1	1
Signature:	11111	1110	0010	11110	11011	11010	11001	11000

76:	**81:**	**85:**	**96:**	**112:**	**146:**	**169:**	**188:**	**200:**	**217:**
1	1	1	1	1	1	1	1	1	1
10100	10001	10000	01111	01110	01101	01100	01011	01010	01000

224:	**244:**	**245:**	**248:**	**253:**	**255:**
2	2	2	1	1	7
0011	1001	1011	01001	10101	000

Table 8.1: The Huffman signatures generated for the intensities in the image shown in Figure 2.7

A Huffman tree has also been generated for the image shown on Figure 2.7. This tree can be seen on Figure 8.12. It is seen, that if the edges in the tree are followed from the top down to each leaf, the leafs signature is obtained, thus showing the link between the Huffman tree and the algorithm.

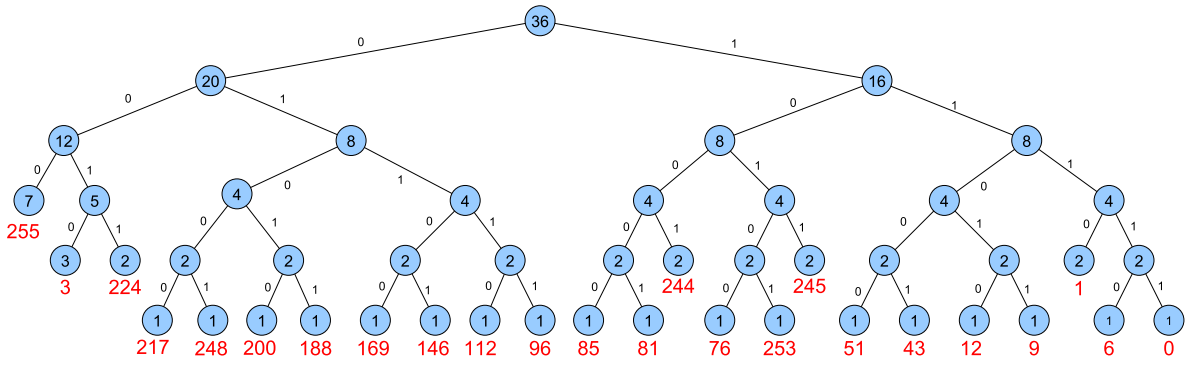


Figure 8.12: A Huffman tree generated based on the image in Figure 2.7. Red text indicates intensities, and the number in each node is the frequency. Note how the top node has the frequency 36, equal to the amount of pixels in the image on Figure 2.7

Generation of compressed image

After the signatures has been made, the original image has to have its data replaced with the signatures. This is not entirely straightforward, since the length of the signatures are not identical. And since compressed image has to be stored in some type of data structure, the signatures have to be split over the entries in this structure where each entry is to have the same length. A uint8 array is chosen as this structure, since each uint8 has the size of a byte. So, the image has to have each pixel values replaced with its corresponding signature, so that one long bitstream is obtained. Then, this bitstream has to be split into 8 bit blocks, which can then be saved in the uint8 array.

An example of this can be seen on Figure 8.13, where the signatures with varying lengths are shown on top, and the byte array containing the signatures is seen below. Note that stuffing bits with the value 0 are added to the end of the last byte, to fill out the remaining bits.

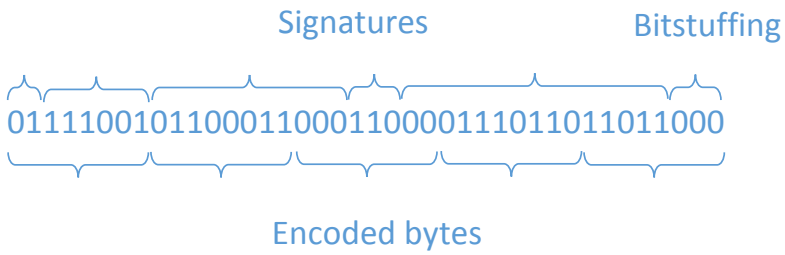


Figure 8.13: The concatenated signatures of a part of an image, and the signatures after they have been split into 8-bit blocks.

Maximum signature length

In order to determine the datatype to hold the signatures, the maximum signature length has to be determined, so that a datatype is chosen that can in fact hold all possible signatures.

The longest signature length is equal to the height of the tree, i.e. the number of edges from the top node to the lowest leaf node. To obtain the deepest tree possible, the tree has to be as unbalanced as possible. Assuming the image is 5 megapixel, the top node of the tree will always have a value of $5 \cdot 10^6$, since the frequencies have to sum to the total amount of pixels in the image.

To obtain as deep a tree as possible, the longest chain of nodes from top to bottom is desired. This is done by having as small steps from node to node as possible, but without having so small steps that the tree branches, instead of being one long chain. This is illustrated on Figure 8.14.

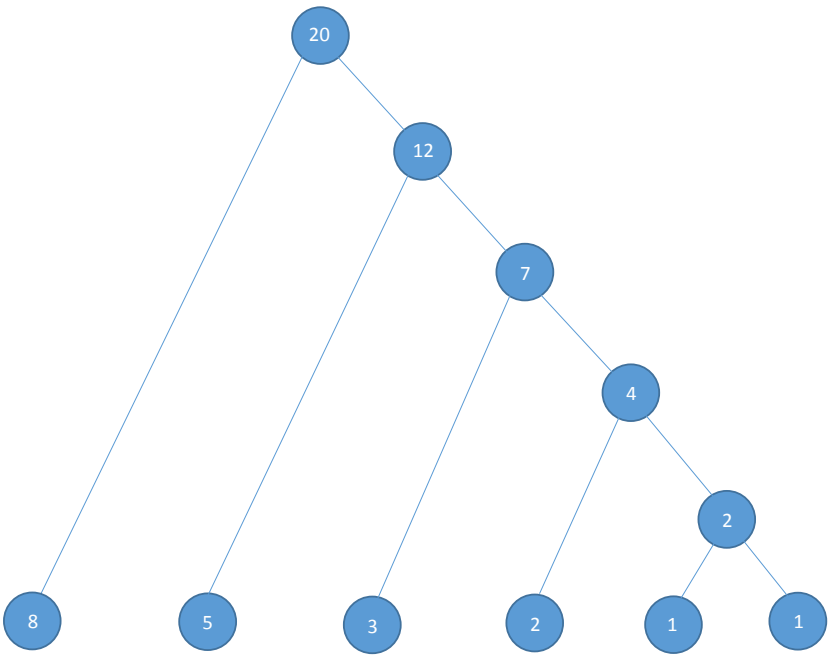


Figure 8.14: A Huffman tree with a frequency interval (see the value of the leaves) equal to the Fibonacci sequence.

This occurs when the next leaf has a value that is one smaller than the sum of the previous nodes, as can be seen on Figure 8.14. This pattern lets the leaves have the smallest possible interval, without two leaves being linked directly together, which would increase branching and thus decrease the height.

As can be seen on Figure 8.14, the leaves form the Fibonacci sequence, and the top node of the tree is equal to the second-next Fibonacci number minus 1, relative to the biggest leaf.

As previously mentioned, the top node has the value $5 \cdot 10^6$. Since the parent nodes are the sum of the previous nodes, it is desired to find the index i (i.e. the height of the tree), that satisfies the equation:

$$\sum_{i=1}^{N} F_i \leq 5 \cdot 10^6 \tag{8.1}$$

By proof of induction, see *Appendix A*, the following can be shown:

$$\sum_{i=1}^{N} F_i = F_{N+2} - 1 \tag{8.2}$$

No whole Fibonacci number is equal to $5 \cdot 10^6$, and so the previous Fibonacci number just before $5 \cdot 10^6$ is chosen. This is because it will not be possible to have a top node with a value of more than $5 \cdot 10^6$, and thus a height leading to a top node having a value higher than this is not possible. Thus, some branching will be seen, but this does not change the height of the tree.

From [Knott, 2011], it can be seen that the 33'rd Fibonacci number, 3524578, is the highest Fibonacci number less than $5 \cdot 10^6$. Therefore, $i + 2 = 33$ and thus $i = 31$. This means that the maximum height of the tree is 31, and the maximum signature length is thus 31 bits.

The conclusion is therefore, that signatures can be saved in a uint32 datatype.

Decompression

To decompress the image, the opposite of when generating the compressed image is used.

On the ground, the counting array that is transmitted along with the image, is used to generate the sequences. Reading the image bitstream from the beginning to end, the signatures generated from the counting array is used as a look up table, to convert the bits read, with the actual pixel values.

As with the compression, bitshifting is used to get the signatures that spans across multiple encoded bytes. In Listing 8.10 the *Decompression* parameters is explained.

```
1  int Decompression(uint8_t *EncodedImage, uint16_t iEncodedBytes, uint8_t *↩
       decompressedData, unsigned int aCounterArray[]);
2  /*  inputs:
3          EncodedImage:   array of all the encode bytes to be deciphered
4          iEncodedBytes:  number of encoded bytes
5          aCounterArray:  the counting array
6      outputs:
7          decompressedData: a pointer to the deciphered data
8      returns:
9          zero:           if success
10 */
```

Listing 8.10: The prototype for the decompression function, with inputs and output listed

Now that the MCUs functionalities have been described and illustrated, it is possible to test them, to ensure the implemented functions operates as intended.

8.6 Test

The following section describes how testing of the MCU functionality will be performed. The test procedure will make sure that every subsystem is tested accordingly with the requirements in section 5.5.

Testing FreeRTOS and CAN

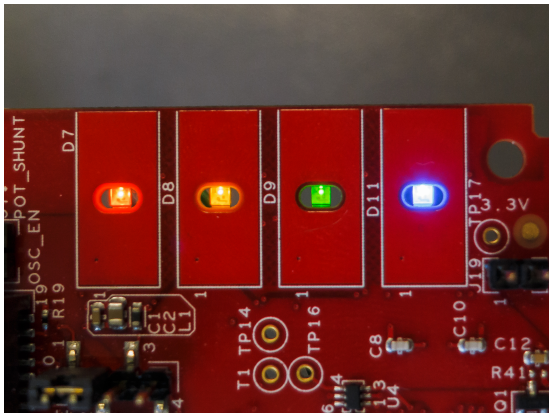
A simple test will be executed to ensure the proper functionality of FreeRTOS and the CAN bus. The following requirement is tested:

- That FreeRTOS can toggle between different task
- That the MCU can send and receive CAN messages.

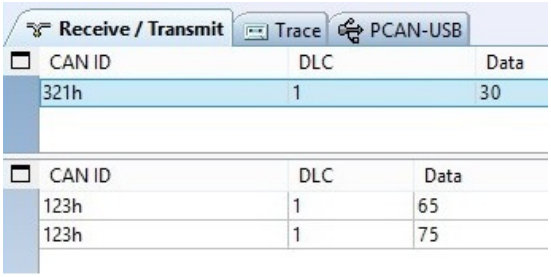
To ensure that FreeRTOS can toggle between different tasks, four LED's mounted on the microcontroller breakout board, is set to blink in its own task. Simultaneously, a CAN RX task is created, as well as a CAN TX task.

The CAN TX task is responsible for sending the data written in a terminal over the CAN bus to a receiver, while the CAN RX task prints the data received from the CAN bus in the terminal. To test this, the CAN L and CAN H pins are connected to a CAN to USB adapter, and into a PC, with a CAN transceiver program installed, called *PCAN*.

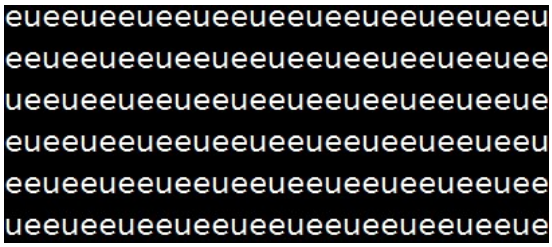
Result:
The tasks are created, and it is seen in Figure 8.15a how all four LED's are lit. When data, in this case an ASCII *0*, is written in the terminal on the PC, it is received as HEX in the PCAN software. See the the upper data field in Figure 8.15b an 0x30 has been received, which is the ASCII code for a *0*, proving that the CAN TX task works.



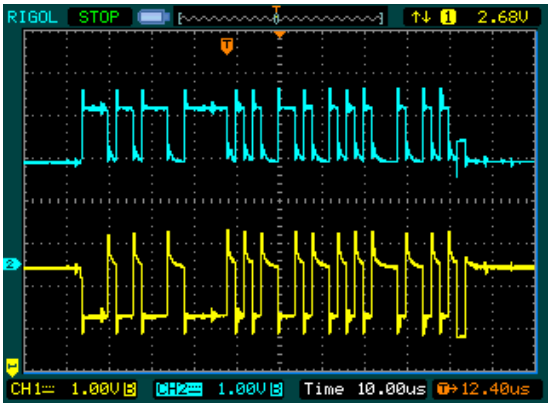
(a) Four individual tasks created in FreeRTOS. Each task makes one individual LED blink.



(b) The CAN transceiver program, PCAN. Receiving the Hex data 0x30 and transmitting the Hex data 0x65 and 0x75.



(c) The printed CAN message in the terminal as ASCII characters *eu*.



(d) Here one of the CAN messages can be seen, represented by CAN H (BLUE) and CAN L (YELLOW)

PCAN is configured to transmit CAN messages with a predetermine set interval, see the lowermost data fields in Figure 8.15b, where the data 0x65 and 0x75 are transmitted. One of the data packages can be seeing on Figure 8.15d. The printed data in the terminal is seeing as the ASCII characters *eu*, see Figure 8.15c, which is equivalent to the data sent by PCAN, with the same interval, where the PCAN is configured to transmit an *e* twice as often as a *u*.

Conclusion:
It can be concluded that FreeRTOS and the CAN transceiver is fully working.

Testing I²C

The test of I²C will be described in this section. Since no I²C data is to be received by the MCU but only transmitted, the ability to receive I²C data will not be tested. However the following requirement will be tested.

- Send camera configurations to the camera via I²C .

It is vital that the data transmitted is of the correct format and that the data is transmitted in one long stream, instead of an alternating stream of addresses and data.

To test this, the I²C pins on the MCU are connected to an oscilloscope, so the bus can be monitored. The pins are also connected to the picam, since a unit has to ACK the messages sent by the MCU for the test to be successful.

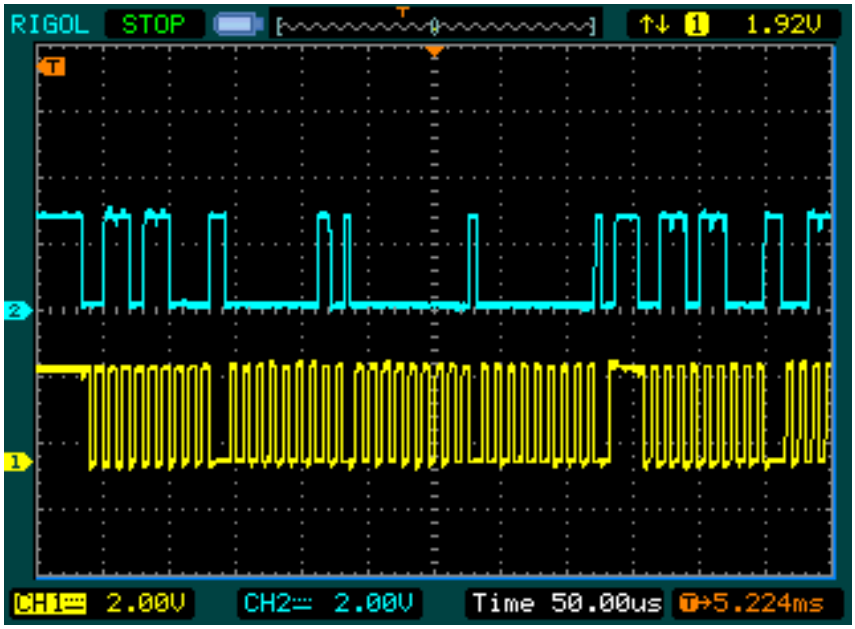


Figure 8.16: I²C transmission test. Yellow [SCL], Blue [SDA].

The MCU is set to transmit the registers to set the camera specifications, see *Appendix 7.2*.

Result:
It is seen on Figure 8.16 that the data is transmitted over the bus, and that the acknowledges expected by the camera occurs.

On Figure 8.17 the expected response can be seen more clearly. At the beginning, the SDA goes low before the SCL and the I²C message begins. The first seven bits are the address bits follow by the read/write bit. Then the cameras acknowledge can be seen, where the slave pulls the SDA low. Hereafter, the 8 bits of data is seen followed by another acknowledge.

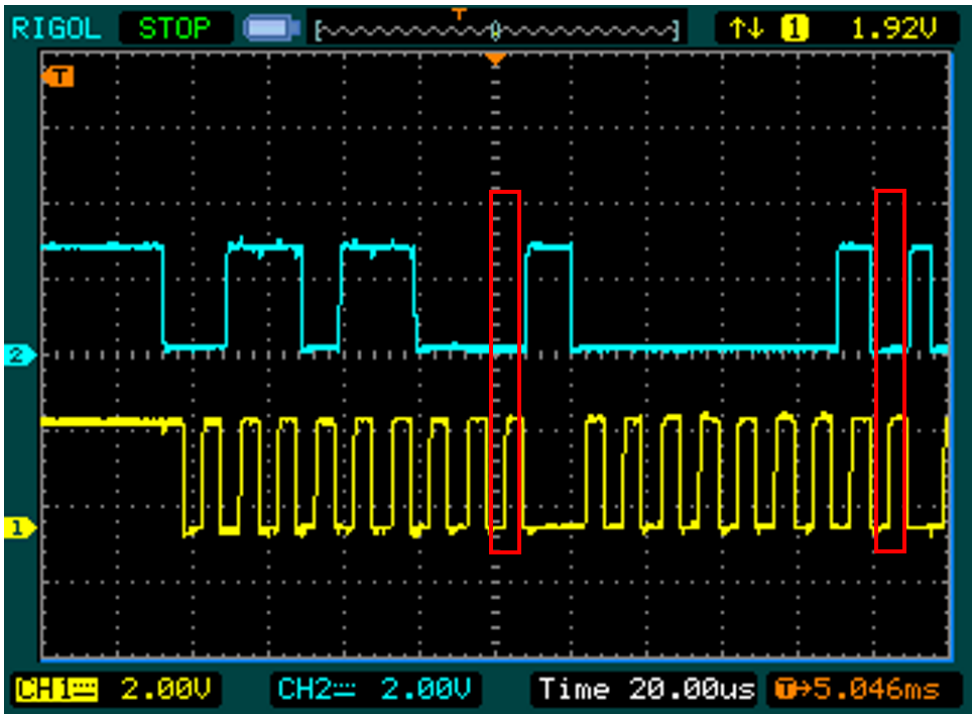


Figure 8.17: The ack occuring on the ninth rising edge of the clock. Yellow [SCL], Blue [SDA]

Conclusion:
The I²C works as intended, since the transmission observed matches the expected, see section 3.2.

Data compression/encoder

In accordance to the four specified requirements for the *Data compression/encoder* in section 8.1, the MCU should be able to do the following

- Detect frames containing too many pixel intensities below a pre set threshold.
- Create Huffman signatures based on the counter array.
- Replace picture data with Huffman signatures.
- The compressed image must take up less space than the original.

Detect frames containing too many pixel intensities below a pre set threshold, will however not be tested because it has not been implemented.

In Section 8.5: Compression and Black frame detection, an example with an image with 36 pixels was used. The example will be tested to verify if the created program generates the expected signatures. Hereafter, the 36 pixel image will be encoded with the use of the signatures, to see if the program replaces the pixel intensities correct with the signatures. This is manageable to see with a 36 pixel image compared to a 5Mpx image. After the test with the 36 pixel image, a test with 5Mpx will be completed, to see if it 5Mpx encoded image is smaller than the original image.

Test procedure for the 36 pixel image:
First a counter array is generated, which is needed to make the Huffman signatures. The counter array is compared to the counter array in Section 8.5: Compression and Black frame detection, to ensure the generated code for making the counter array is correct.

```
Counterarray: 0 1   Counterarray: 112 1
Counterarray: 1 2   Counterarray: 146 1
Counterarray: 3 3   Counterarray: 169 1
Counterarray: 6 1   Counterarray: 188 1
Counterarray: 9 1   Counterarray: 200 1
Counterarray: 12 1  Counterarray: 217 1
Counterarray: 43 1  Counterarray: 224 2
Counterarray: 51 1  Counterarray: 244 2
Counterarray: 76 1  Counterarray: 245 2
Counterarray: 81 1  Counterarray: 248 1
Counterarray: 85 1  Counterarray: 253 1
Counterarray: 96 1  Counterarray: 255 7
```

Figure 8.18: Generated counter array for the 36 pixel image

The generated counter array, seen in Figure 8.18, and the counter array found in Section 8.5: Compression and Black frame detection, see Table 8.1, is identical. The generated counter array can now be used to generate the signatures, with the use of the function *HuffmanTree*, see section 8.5. The function *HuffmanTree* is called and in Figure 8.19 the generated signatures can be seen.

```
HuffmanSignature[0][5]: 11111  HuffmanSignature[112][5]: 01110
HuffmanSignature[1][4]: 1110   HuffmanSignature[146][5]: 01101
HuffmanSignature[3][4]: 0010   HuffmanSignature[169][5]: 01100
HuffmanSignature[6][5]: 11110  HuffmanSignature[188][5]: 01011
HuffmanSignature[9][5]: 11011  HuffmanSignature[200][5]: 01010
HuffmanSignature[12][5]: 11010 HuffmanSignature[217][5]: 01000
HuffmanSignature[43][5]: 11001 HuffmanSignature[224][4]: 0011
HuffmanSignature[51][5]: 11000 HuffmanSignature[244][4]: 1001
HuffmanSignature[76][5]: 10100 HuffmanSignature[245][4]: 1011
HuffmanSignature[81][5]: 10001 HuffmanSignature[248][5]: 01001
HuffmanSignature[85][5]: 10000 HuffmanSignature[253][5]: 10101
HuffmanSignature[96][5]: 01111 HuffmanSignature[255][3]: 000
```

Figure 8.19: The generated Huffman signatures, their intensity and the length of the intensities.

In Figure 8.19 the intensity, the length of the Huffman signature and the signature is displayed. An example of this is *HuffmanSignature[0][5]: 11111*, the number [0] is the intensity, the number [5] is the length of the Huffman signature and the last number 11111 is the generated binary Huffman signature, for that specific intensity. By comparing the generated signatures, seen in Figure 8.19, with the signatures made in section 8.5, it can be seen that they are identical, therefore it can be concluded that the generating of signatures, by using the created function, for a 36 pixel image was a success.

The function *ReplacePictureData*, explained in Listing 8.9, is called and used on the image in Figure 2.7b page 25, by executing the code in Listing 8.11.

```
1
2  while(ChunkCounter*dChunkSizeRam < DataArraySize){
3
4  fresult = readFilePartly("agurk.txt", FA_READ, FL_OPEN | FL_CLOSE, aPictureData, ↩
       ChunkCounter*dChunkSizeRam, dChunkSizeRam);
5
6  ReplacePictureData(aPictureData, dChunkSizeRam,                                ↩
       HuffmanSignature[1], HuffmanSignature[0],                                      ↩
       EncodedImage, &iEncodedByte);
7
8  /* create a new file, where we write the counter array as the first data */
9  SD_FileWrite(PREVIEW, fileOffset, EncodedImage,                                ↩
       iEncodedByte, &nameNumber, fileOffset+iEncodedByte);
10
11 fileOffset += iEncodedByte;
12
13 /* increase offset with the length of data just written */
14 iEncodedByte = 0;
15 ChunkCounter++;
16 }
```

Listing 8.11: The prototype for the decompression function, with inputs and output listed

The code in Listing 8.11 uses a *while loop* to ensure that all the picture data is read from the SD-card. The *while loop* contains:

1. Picture data, received from the SD-card, with the size of dChunkSizeRam is placed in the *fresult*. To ensure that the data read from the SD-card is not the same, there is an offset at the size of ChunkCounter multiplied with dChunkSizeRam. At the end of the *while loop* the ChunkCounter is incremented with 1.
2. The dChunkSizeRam sized picture data, containing pixel intensities, is encoded in *ReplacePictureData*, by swapping the Huffman signatures with the pixel intensities.
3. *SD_FileWrite* writes the encoded data back to the SD-card, by using an offset at the size of fileOffset, which is added with iEncodedByte. iEncodedByte is a variable received from the *ReplacePictureData* containing the size of the encoded data.

An image containing the encoded picture data has been produced, and can be seen in Figure 8.20.

```
EncodedImage[0]: 00100111 EncodedImage[10]: 10110000
EncodedImage[1]: 01001101 EncodedImage[11]: 00001111
EncodedImage[2]: 10111100 EncodedImage[12]: 01110100
EncodedImage[3]: 10111110 EncodedImage[13]: 01010100
EncodedImage[4]: 00010001 EncodedImage[14]: 11000101
EncodedImage[5]: 01011100 EncodedImage[15]: 10100101
EncodedImage[6]: 01000001 EncodedImage[16]: 11001011
EncodedImage[7]: 10000110 EncodedImage[17]: 11101101
EncodedImage[8]: 00000010 EncodedImage[18]: 01111011
EncodedImage[9]: 01000000 EncodedImage[19]: 00100000
```

Figure 8.20: The encoded 36 pixel image.

As seen in the figure, the image size is 20 bytes, which is 16 bytes smaller than the original 36 bytes file. It can thereby be concluded that for images with few different pixel intensities, in this case 24, or pixel occurring in the pattern of the 36 pixel image, the two compression algorithm works. Now the same steps will be performed on the 5Mpx image.

A snippet of the occurring signatures for the 5Mpx image, can be seen in Figure 8.21.

```
HuffmanSignature[0][11]: 11111111111        HuffmanSignature[10][15]: 0000000010111110
HuffmanSignature[1][18]: 000000010001001010 HuffmanSignature[11][13]: 0000010111111
HuffmanSignature[2][11]: 01111111111        HuffmanSignature[12][13]: 0000010111110
HuffmanSignature[3][10]: 1111111110         HuffmanSignature[13][11]: 00010011111
HuffmanSignature[4][16]: 0000000011111111   HuffmanSignature[14][10]: 0011111110
HuffmanSignature[5][16]: 0000000100010011   HuffmanSignature[15][9]: 011111110
HuffmanSignature[6][10]: 0111111110         HuffmanSignature[16][9]: 010111111
HuffmanSignature[7][9]: 111111110           HuffmanSignature[17][8]: 10111110
HuffmanSignature[8][9]: 101111110           HuffmanSignature[18][9]: 010101110
HuffmanSignature[9][15]: 000000001111110    HuffmanSignature[19][8]: 10011110
```

Figure 8.21: A snippet of the signatures generated for the 5Mpx image.

Results:
As it can be seen on Figure 8.21, none of the shown signatures have a length under 8 bits. For

the entire image, very few signatures have a length smaller than 8 bits, which is the baseline size that the compression aims at reducing. Thus, by not having primarily signatures under 8 bits, the length of the compressed image is in fact bigger than the original. Of course, the amount of signatures with a length under 8 bits is not entirely determining for the length of the compressed file, since it depends on the frequency of the intensities as well, but the average signature length is a good indicator for whether or not the code works as desired. The exact reason for why the generated signatures are too long is not known, but it has to do with how the Huffman signatures are generated, but the fact that the image is made of random data is also a contributing factor (i.e. the frequencies are almost identical).

The entire size of the compressed image can be found as shown in Listing 8.12, resulting in a compressed image size of 5658725 bytes, which is 13 % larger than the original.

```
for(i = 0; i<256; i++ ){
    sumOfEncodedBytes += HuffmanSignature[0][i]*aCounterArray[i];
    }
printf("\n\r total size: %i", sumOfEncodedBytes/8); // print total size in bytes
    }
```

Listing 8.12: The code used to calculate the size of a compressed image

Another issue in regards to the code is when the image is to have its picture data replaced with corresponding signature, where the code might crash from time to time. The function uses malloc/realloc, which might be causing problems.

Conclusion: It can be concluded that for an image with few intensities the created algorithm works as intended, see Figure 8.20. But for an image using all 256 intensities the algorithm makes a lot of signatures with a length longer then 8 bit see Figure 8.21, and the image is thereby not compressed see Table 8.2. The recent for the large signatures could be two things

1. The use of malloc/realloc could be causing trouble with memory leaks or allocation errors.
2. The 5 Mpx image is normalized, so the frequency of occurrence for pixel intensity is almost identical, which makes the tree more *balanced* and thereby making long signatures for every intensity.

	36 Px	5 MPx
Original size	36 B	5 MB
Compressed size	20 B	5.6 MB

Table 8.2: The size of data before and after compression

Testing CAN message handling & timelapse

In chapter 8, the requirements to the CAN message handler is discussed. This chapter is to document, through tests, if the following requirements are met.

- Initiate the capturing of an image when requested via CAN
- Display files and storage space in a terminal when requested via CAN
- Delete a chosen image on the storage unit when requested via CAN
- Receive camera configurations via CAN and setup the camera accordingly.
- Display image data in a terminal when requested via CAN
- Receive settings for the timelapse functionality via CAN

The timelapse function is to fulfil the following requirements:

- Request an image capture with a set interval autonomously
- Toggle between not capturing an image or deleting the oldest image if there is not enough room for more images
- The interval of which images are captured must be changeable

Test procedure

For testing purposes, the functions that the message handler calls, are replaced with functions printing their name in a terminal, so it can be seen which functions is currently running. Since it has not been possible to get the *SD card delete* function to work properly, see section 9.3, the message handler for this function has not been implemented and will therefore not be tested.

Likewise, the configuration receive function has not been implemented The timelapse function has not been fully implemented, so the toggle between capturing and deleting an image is not implemented, however the functionality of toggle between waiting and capturing an image is implemented and tested. Thus, 4 functions are called in the test, see Listing 8.13.

e funktioner
r overflødige at
fx er CAN RX
aptureImage
FreeRTOS
tet

```
1
2  void getPictureFromSD(){
3
4      while(1){
5      xSemaphoreTake(Get_picture_from_SD, portMAX_DELAY);
6      taskENTER_CRITICAL();
7      printf("\n\rGet picture");
8      taskEXIT_CRITICAL();
9      }
10 }
```

```
11
12 void storageCheck(){
13
14     while(1){
15     xSemaphoreTake(Storage_check, portMAX_DELAY);
16     taskENTER_CRITICAL();
17     printf("\n\rStorage Check");
18     taskEXIT_CRITICAL();
19     }
20 }
21
22 void timeLapse(){
23     TickType_t xLastWakeTime;
24
25     while(1){
26         vTaskDelayUntil(&xLastWakeTime, 1000*timelapse_interval*portTICK_PERIOD_MS);
27         taskENTER_CRITICAL();
28         printf("\n\r\tTimeLapse");
29         taskEXIT_CRITICAL();
30     }
31 }
32
33 void CaptureImage(){
34
35     while(1){
36         xSemaphoreTake(Take_picture, portMAX_DELAY);
37
38         taskENTER_CRITICAL();
39         printf("\n\rCapture Image");
40         can_send("D", 1, tx_ID);
41         taskEXIT_CRITICAL();
42     }
43 }
```

Listing 8.13: The 4 functions called by the CAN handler

PCAN is configured to transmit messages enabling the various tasks, at a predetermined interval. Messages changing the timelapse interval to 5 or 10 seconds are also available, see Figure 8.23. The tasks are initiated, the scheduler is started, and PCAN transmit the messages. After about 30 seconds, the timelapse interval is changed to 5 seconds using PCAN. The interval of which the timelapse message is printed in the terminal is monitored with a stopwatch.

```
Get picture
Capture Image
Storage Check
Capture Image
Storage Check
        TimeLapse
Get picture
Capture Image
Storage Check
Capture Image
Storage Check
        TimeLapse
Get picture
Capture Image
Storage Check
Capture Image
Storage Check
        TimeLapse
Get picture
Capture Image
Storage Check
Capture Image
Storage Check
        TimeLapse
Get picture
```

Receive

CAN ID	DLC	Data	Cycle Time
321h	1	44	5002

Transmit

CAN ID	DLC	Data	Cycle Ti...
123h	1	73	5000
123h	1	70	5000
123h	6	49 4D 47 5F 31 30	10000
123h	2	31 30	Wait
123h	1	35	Wait

Figure 8.23: Three messages "s", "p" and "IMG_10", converted to hexadecimal, are sent via CAN with a set interval to call the various tasks. The timelapse interval can be toggled from 10 seconds to 5 seconds. A "D" (44) has been received via CAN, as an acknowledge that the Capture image function was called

Results

The messages printed in the terminal can be seen in Figure 8.22. Here, it can be seen how the *Capture image* and *Storage check* functions are called twice as often as *Get Picture*, which matches their cycle times in PCAN as seen in Figure 8.23. It is seen how the *Timelapse* function gets called twice as often in the lower part of the image, which is due to the timelapse configuration sent to the MCU.

Conclusion

All tasks gets called correctly when requested via CAN, and the timelapse interval can be changed by sending a message via CAN. Thus the CAN Message Handler works as intended, except from the tasks that are not implemented.

The timelapse function performs as expected, as the time interval can be changed, and matches the set interval. However, the ability to toggle between deleting images or not capturing images has not been implemented.

Most of the MCU requirements from section 8.1, have been implemented and tested. The functionalities of the *Data compression/encoder* are however not entirely working as intended and have to be looked in to. The black frame detector was never implemented on the MCU and have thereby never been tested.

The next chapter will discuss the functionalities and requirements to the SD card and how these have been implemented on the system.

9 | SD card

The SD card functions as the permanent storage unit in the system. Its purpose is to save data and keep it when the system is turned off. The SD card in it self is not important nor interesting from a design perspective, but can be seen in section 6.2, and it will therefore just be the functions written to interface with the card that will be described in the chapter.

9.1 Requirements

Permanent storage capability

The SD card functionalities must be able to fulfill the following requirements, which are based on the considerations made in Section 6.1: Functional design:

- Create files
- Organize files based on their contents
- Save previews on the SD card
- Save the compressed image and counter array on the SD card
- Save camera setup information on the SD card

  Since it would be preferable to save the setup on board, instead of having to upload it for each image captured
- Obtain a list of files on the SD card
- Obtain information about the storage space available on the SD card
- Delete files
- Show the contents of a file in a terminal

9.2 SD card implementation

In the following section, the implementation of the functions needed to interface with the secure digital (SD) card will be described. These functions build upon other functions obtained from the Freescale Software Development Kit (SDK), which handle the physical interface to the SD card. The file system that is used on the SD card is given from the freescale community[Freescale, 2014] The main focus will be on the following functions:

- SD_CreateFile
- SD_WriteFile

- SD_ReadFile

Other functions related to the SD card, will briefly be explained.

SD card abstraction layers

An overview of the SD card abstraction layers can be seen on Figure 9.1.

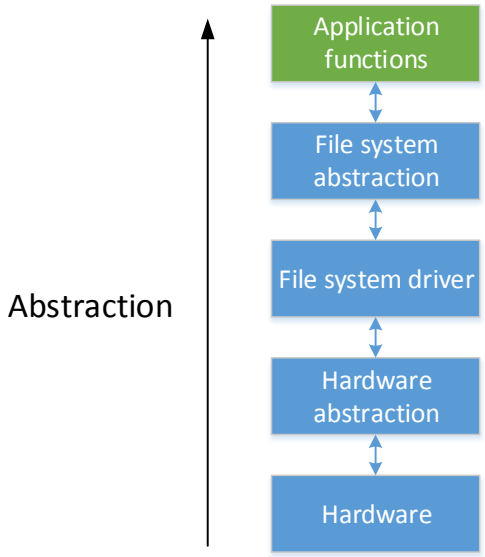


Figure 9.1: A diagram showing the abstraction layers of the SD card. The green box indicates that these functions are made by the group.

The SD card functionalities are based on an example project in the MCU IDE, showing how blocks of data can be written and read from the SD card. This project is responsible for the Hardware Abstraction Layer (HAL), by having functions for reading and writing sectors, check if an SD card is mounted etc. In the IDE, a file system module is found, called FatFS, which enables the use of volumes, files and folders on the SD card. This module is based on the FAT file system, and adds an abstraction on top of the HAL meaning that keeping track of where file data is located on the drive in regards to clusters and sectors is handled automatically, based on file object structures.

However, an abstraction layer more is added to the SD card functionalities, the file system drivers, to further ease the use of the card. This abstraction layer is found in [Freescale, 2014], and enables easy navigation between files and folders using names, instead of file- and directory objects. It also enables the *ls*-command, also known from Linux, that lists the files in the current folder. Finally, on top of this layer, are the application functions written by the group which will be described in the following.

SD_CreateFile

The create function is the an essential function to the SD card. The main purpose of the create function is to create a new file and to store data in it.

The flowchart seen in 9.2 will be use to describe the flow of the create function.

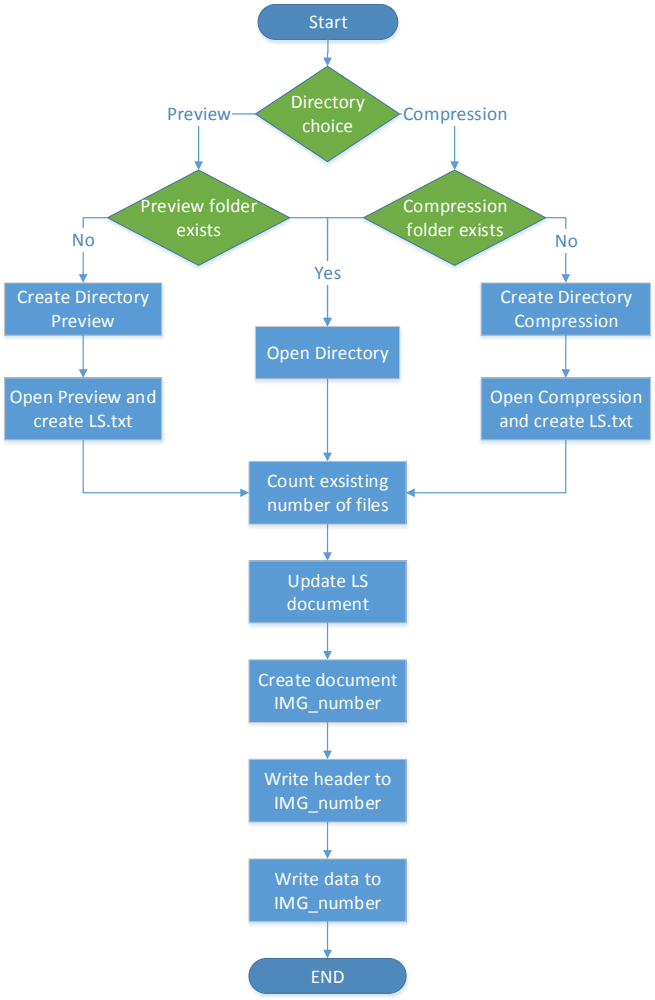


Figure 9.2: Flowchart of the function to create new files and storage the first data packed

When the function is called, the first statement is to determine which folder to store a certain data in, for instance it could either be a preview or a compressed image. The next statement is to determine if the folder exist or not. If the folder exist, it will be opened. If not the folder will be created, opened and then the first file, LS, will be created. This file is to store all the file names there is in the specific folder. After this, all the files are counted. When the counting is done, an integer is returned which is used under naming of the new file and to save the file's location in the folder.

After the file is created the first data to be written is a header to the file. The header contains the total length of the file and how many packages of 84 bytes it contains. The size of packages is predetermined from the maximum data length that can be sent from the satellite, see 1.1. When the header is written, the first package is written to the file.

SD_WriteFile

When the file is created, by the *SD_CreateFile* and the first chunk of data is written, the SD_Write function has to be called to write any further data.

The flow of the write function can be seen in 9.3

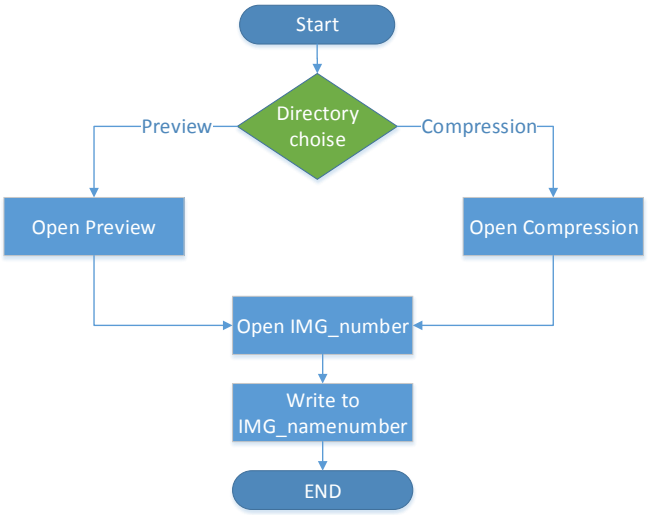


Figure 9.3: Flowchart of the function that write the remaining packages to a file

The first statement is to determining if a preview or a compression folder should be opened, hereafter, the chosen folder is opened. Then the function will search after a predetermined file, for instance IMG_143. When the file is found, it will be opened and the new data will be written with an offset, so any pre-existing data does not get overwritten.

SD_ReadFile

The function SD_Read is used to read the stored data from any of the existing files and then return the data to the microcontroller, so it can be sent via the CAN bus to the satellites COM-subsystem and down to earth.

The flow of the read function can be seen in 9.4

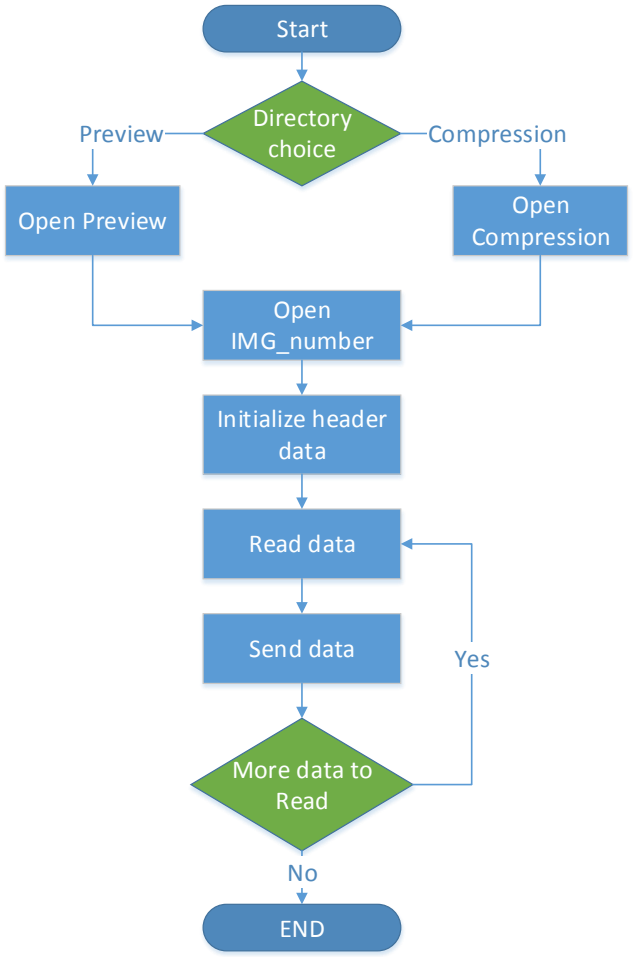


Figure 9.4: Flowchart of the function that read and send a data packet

The first statement is to check which folder the data is stored in and, hereafter, opening the folder. Thereafter the file is opened and the header of the file, which contains the length of the file and packaged number, is initialized. Then the function goes into a loop state, where it will read and return a data string of 84 bytes, until the file is fully read.

Other functions

Furthermore there is other two functions:

- SD_Delete
  - A function that deletes a specific file in a certain path.
- SD_LS
  - A function that returns all the file names there are in a certain folder.

Now that the functions used to interface with the SD card have been described, it is possible to test them to ensure that they function as intended.

9.3 Test

The SD-card will be tested according to the SD-card requirements stated first in the chapter. It should be able to create and organize files. Save previews, compressed images, counter arrays and camera setup informations. also, obtain a list of files stored on the SD-card and obtain information on how much storage space left available. furthermore it should be able to delete files and, finally, a file's content should be able to be shown in a terminal.

Test procedure

During the test of the SD-card the code described in Listing 9.1 will be used. The code is written with a loop using the different needed functions. The loop will ensure redundancy of the system making sure every file is handled in the same manor. The test procedure relies on the functions described in subsubsection 9.2 through subsubsection 9.2.

```
1  void createSDFile(){
2      // Initialize needed variables for test
3      uint8_t counter = 0;
4      uint16_t nameNumber = 0;
5      static char temp[1025] = "...";
6      // Create 8 files
7      while(counter <= 8){
8              // char array for printing the output in terminal
9              char out[25];
10 // Creates file in preview folder, writing temp which is 1024 long and is given address↩
       nameNumber
11             SD_FileCreate(PREVIEW, temp, 1024, &nameNumber);
12 // Writes in the created file, adding another 1024 characters after the existing string
13             SD_FileWrite(PREVIEW, 1024, temp, 1024, &nameNumber, 1024*2);
14 // Saves the file with respective number and name
15             sprintf(out, "IMG_%i.txt", counter);
16 // Read the file
17             SD_Read(out, PREVIEW);
18
19             counter++;
20       }
21     // Read the SD card, counting the amount of files and print them
22       SD_LS(PREVIEW, &nameNumber);
23 }
```

Listing 9.1: The code showing the creation of a file on the SD-card used for the test

The code in Listing 9.1 starts by initialising the needed variables for use in the while loop. The *counter* is used to ensure only 9 files is created, and the *nameNumber* is used to maintain control of the files position in the folder. A static char array is initialized with a test string containing 1024 signs and a terminating character. The while loop will be creating 9 files by using the initialised variables and the earlier described functions. The loop will be performing the following operation:

1. Create a char array used for storing the file name momentarily

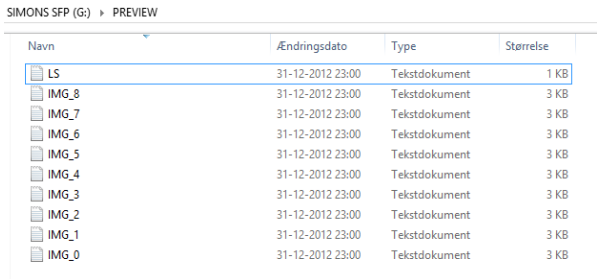
2. Create a file and a folder called PREVIEW, using the predefined static string and save the file's location in the preview folder by using the *nameNumber*.
3. Writes in the newly created file using the predefined string and shifting it 1024 places in the file, before writing, thereby not overwriting existing data.
4. The file will now be given a number correlating to the position in the counter (which is the same number *nameNumber* has been given).
5. Finally, the *SD_Read* will read what is in the file.

When the loop has been performed 9 times the ls function will be executed, counting every file in the folder and saving the list in a file named LS.

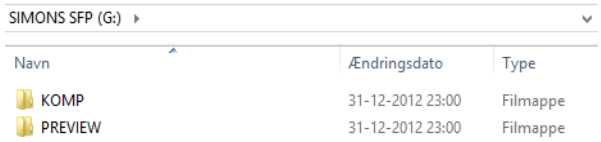
Results

In the following test sequence, different screen captures were made to document the output of the MCU. These results/images governs Figure 9.5a to Figure 9.6. The results are as follows:

Read from a Windows 7 computer using an embedded SD-card reader, the expected list of files, shown on Figure 9.5a, is stored on the SD-card:



(a) List of the files contained on the SD-card after the test has been performed



(b) List of folders contained on the SD-card

Figure 9.5: Showing to screen captures of windows 7 computer showing the content saved on the SD-card

It can be seen in Figure 9.5a the folder created is named preview as expected and 9 files plus a LS file is created and stored in the folder.

Using the MCU SD_Readfile to read one of the files the expected information, seen on Figure 9.6 is read from the SD-card.

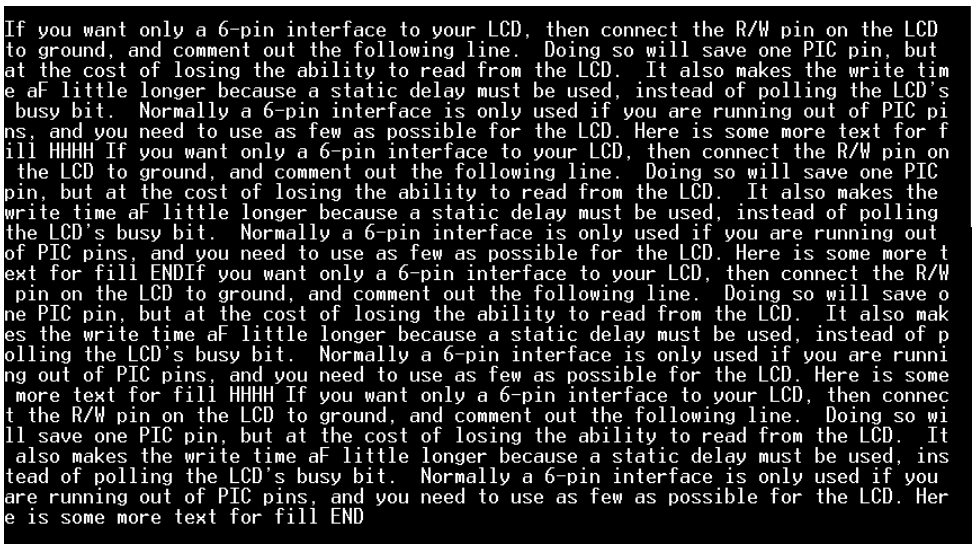


Figure 9.6: Terminal output showing what is read in the file that has just been created.

Using the MCU *LS* function an output in the terminal is given, which displays the different files contained in the specific folder, preview, and how many bytes there is left on the SD-card, The result is displayed on Figure 9.7.

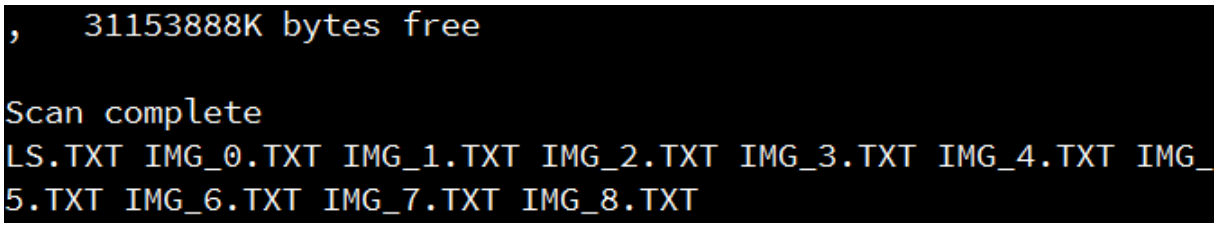


Figure 9.7: Terminal output showing the LS function writing a list of files contained on the SD-card.

The 9 file names is printed in the terminal and the storage space which is left on the SD-card is displayed.

Listing 9.2 displays the code used for deleting, *delFile0rDir*, and accessing a specific folder stored on the SD-card, *openDirectory*.

```
1       openDirectory("Komp", 4);
2       delFileOrDir("IMG_2.txt");
3       openDirectory("", 0);
```

Listing 9.2: Code snippet showing functions which can delete files and access folders stored on the SD-card

Figure 9.8 shows how the SD_delete, deletes a specific file and the ls function creates an updated list of files.

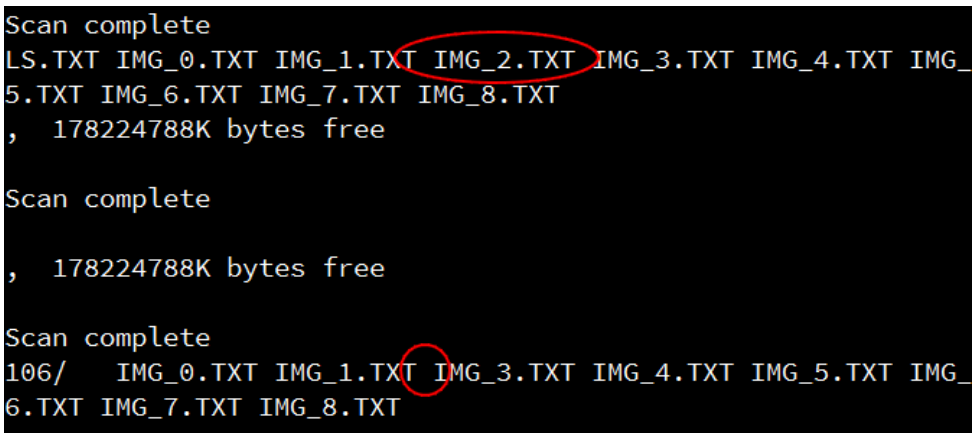


Figure 9.8: Terminal output showing the LS function writing after a delete task have been executed

Conclusion

The results is compared to the requirements listed at the start of the chapter. Files has been created and organized in their specific folder, in this case, the preview folder, see Figure 9.5a. The expected contents of a file has been read and is displayed in Figure 9.6. A list is obtained of the files in the preview folder and information on the storage space available on the SD-card is given, see Figure 9.7. Deletion of a file, as seen on Figure 9.8, acts as expected as seen in Figure 9.8. However, an error occurred when the file that is being deleted is not the latest generated. If the file that is being deleted i.e. is the picture before the latest, only one new picture can be stored on the SD card, but it will however overwrite the latest picture when it is generated. This means that every time a new picture is generated it will overwrite the previous generated picture. This error occur because of the name generator. The name generator works by counting all the file in a specific directory and then naming the new file by the file count. If a file is deleted the count will be one less than before and the generation of a new file, will therefore have a name identical to one of the existing files.
Furthermore in Figure 9.5b it shows that different folders have been created and stored on the SD card. The function to save camera setup information on the SD card has not been implemented and has therefore not been tested.

After evaluating the test results it has been concluded, that the SD-card functions are working as intended and it therefore fulfils its requirements provided in the start of the chapter, except from the ability to delete images and save camera setups.

10 | FPGA

In this chapter, the functionalities implemented in the FPGA will be described, in regards to their functionality and the implementation of them. These functionalities are the data extraction of data from the camera by using a MIPI Bridge, and the generation of the preview and counting array.

The implementation of the SDRAM controller and the interfacing of this in the FPGA will, however, be described in the next chapter, see chapter 11.

10.1 FPGA requirements

The FPGA functionalities, see Section 6.1: Functional design, must be able to fulfill the following requirements:

Data extraction

The following requirements are based on the functionality description as seen in Section 6.1: Functional design:

- Receive data from the camera
- Convert the received data to 8-bit pixel data
- Relay the pixel data to the preview generator, counting array and the SDRAM, after a set number of frames, see subsubsection 10.4.

Preview generation

- Generate a preview based on the image captured

  See Section 6.1: Functional design

- Save the preview in SDRAM

  See Section 6.4: Electrical Interfaces

Pixel intensity counting for encoding

- Keep track of the frequency of occurence of the pixel intensities in the captured image

  See Section 6.1: Functional design

Storage handling

- Transmit data from the memory units to the MCU

See Section 6.1: Functional design

- Multiplex between the 2 memory units

  See Section 8.3: Other functionalities

- Pass configuration to the desired registers

  See Section 6.1: Functional design

10.2 Overall functionality

In the following section the overall functionality of the FPGA will be described by using the flowchart in Figure 10.1.

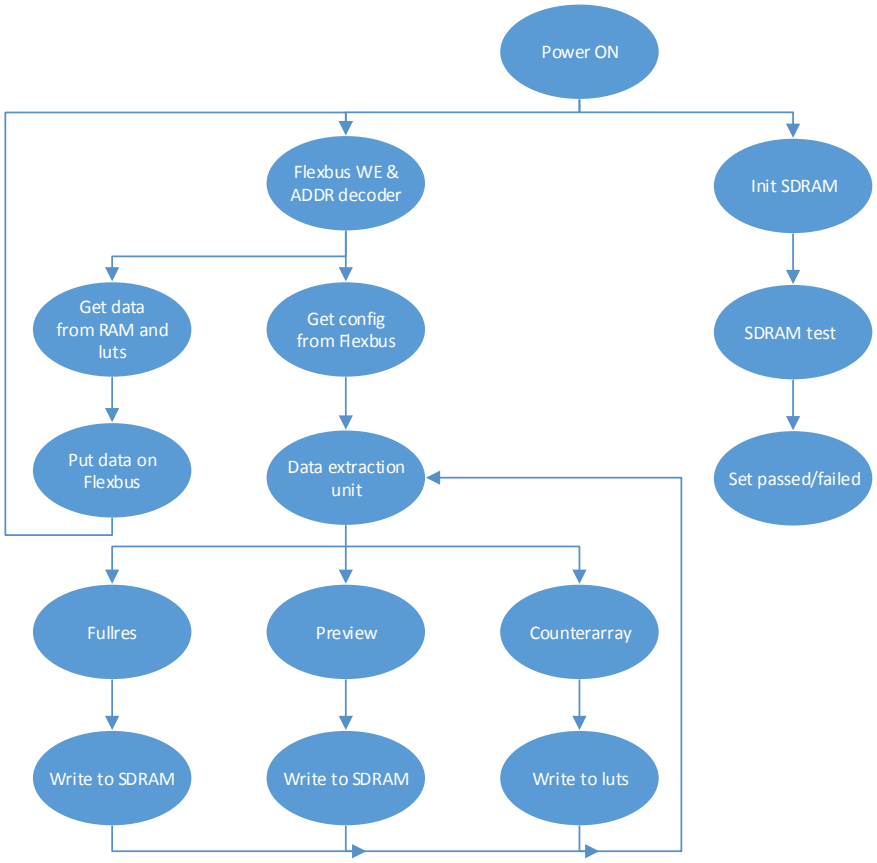


Figure 10.1: Flowchart for the FPGAs functionalities

Handling whether or not the FPGA should be powered on is determined by the MCU, see section 6.2.

Initialize RAM will be performed when the FPGA is powered on. One of the processes starting will be initialization of the SDRAM chip. The details of this process is described in chapter 11. When the SDRAM is active it will perform a test, making sure the chip is fully operational and every address is available. If the system fails during the test, it will report an error to the MCU and the SDRAM will not be accessible. If the test is successful an acknowledges will be sent to the MCU and the MCU will begin transmitting configurations through the Flexbus.

will be listening for instruction on the flexbus. The FPGA will be either writing configuration data to the Data extraction unit or requesting data from the SDRAM or the internal gates (luts).

if the WE is down the MCU is requesting data, and the specific data will be retrieved from the SDRAM or the internal luts. Hereafter, it will be transmitted through the Flexbus to the MCU.

Get config from flexbus is activated when the Flexbus *Write Enable* is high, and the MCU is transmitting data to the FPGA. The FPGA will be receiving the configuration data for the extraction unit (pixel gate) and the preview. The configuration to the extraction unit is which frame to save and, to the preview, the frame size and how much the image should be scaled down.

Data extraction unit receives transmitted data from the camera and simultaneously be handling:

- Storing the full resolution image in the SDRAM
- Creating a preview, by using the extracted data, which will be stored the image in the SDRAM
- Creating a counter array needed for the compression, by using the extracted data, which will be stored in FPGA gates (luts).

The flow in the FPGA when it is turned on by the MCU has been described. Now the order of activities in the FPGA will be explained, in the following Section 10.3: Overall activities.

10.3 Overall activities

To better visualise the order of activities, when the FPGA is activated, a UML sequence diagram can be seen on Figure 10.2

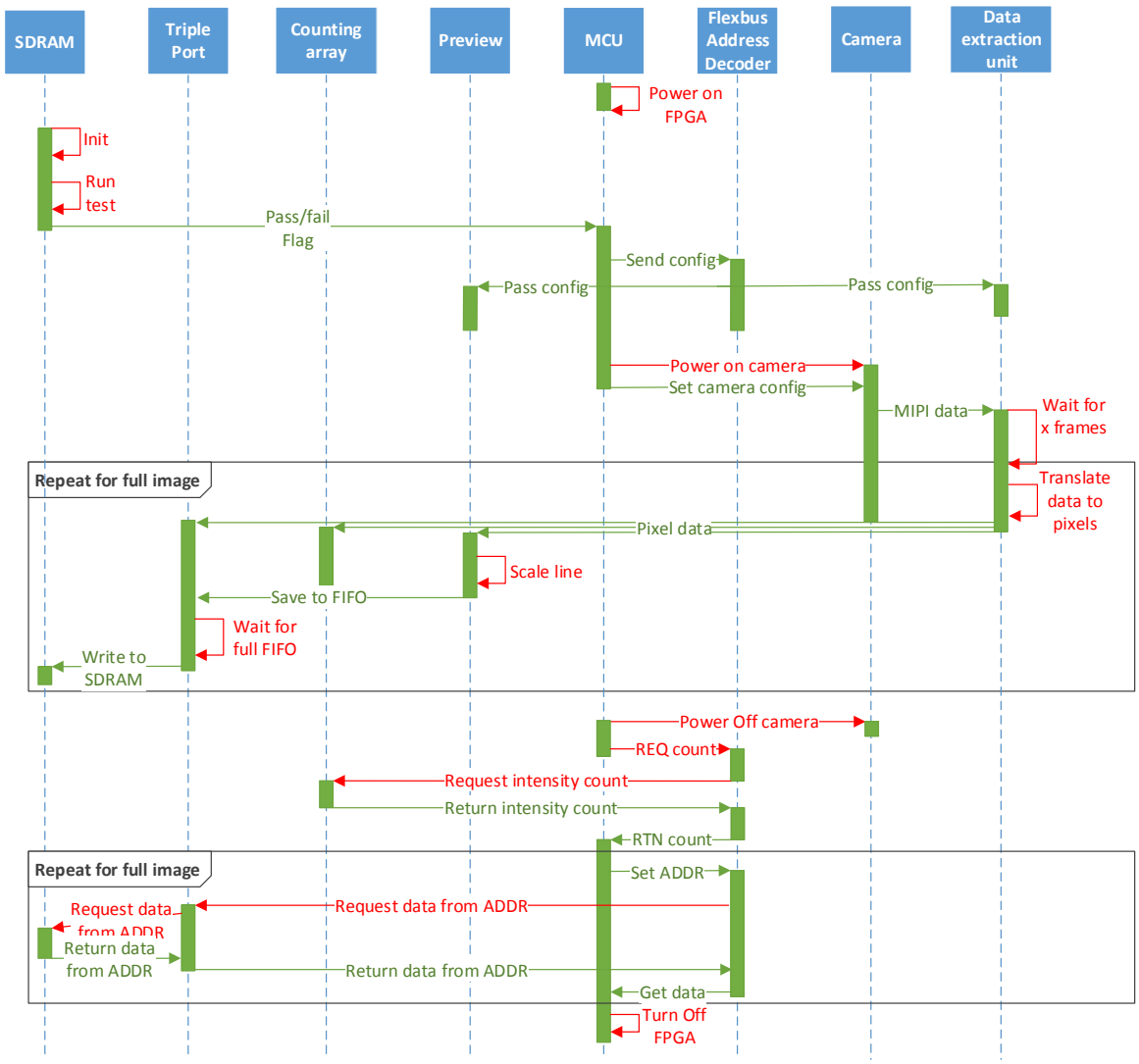


Figure 10.2: A sequence diagram showing the signal and dataflow inside of the FPGA when capturing an image. Red arrows indicate commands and green indicates data.

The diagram shown in Figure 10.2 is described in the following:

- The sequence starts when the MCU powers on the FPGA. When powered up, the SDRAM is initialized, and a self test is performed. If successful, a flag is set, to tell the MCU to go on.
- Now the MCU sends the configuration, containing image size and scaling factor, to the Flexbus Address Decoder. The relevant pieces of the configuration are then passed on to the data extraction unit and the Preview generator.

- The FPGA is now configured and ready to receive an image from the camera.
- The MCU turns on the camera and configures it via the I²C bus, which causes the camera to start streaming image frames via the MIPI bus to the data extraction unit. The MCU now waits for a predetermined time, while the image is being processed.
- The camera needs to take a few pictures in order to adjust white balance and exposure, so the data extraction unit ignores the first X[1] number of frames, according to the configuration.
- After the preset number of frames has been received, the data extraction unit starts sending decoded pixel data to the Preview generator, the counting array and one of the write ports for the SDRAM in parallel. This is done for one whole frame, where after all following frames from the camera are ignored.
- Four things happens in parallel for the X frame:
  - The pixel data received is passed directly on to a FIFO in the Triple port controller.
  - The Preview generator scales down the image, one line at a time, by a preconfigured amount and sends each line to a FIFO in the Triple port controller.
  - The Counting array counts the number of occurrences of each pixel intensity.
  - The Triple port controller monitors its FIFOs, and writes data to the SDRAM when there is enough data for a burst-transfer.
- When the whole image has been processed, the MCU powers down the camera. The counting array is read via the Flexbus address decoder, thereafter the full image and the scaled image are read.
- After this, the FPGA is powered off.

Since the overall activities now has been explained, a more detailed description of the different subs system follows. Starting from the source of the image the extraction of data from the camera will now be described.

10.4 Data extraction

The section will describe how the data extraction unit, i.e. the MIPI bridge and the Pixel gate works. The MIPI bridge and Pixel gate are essential part for decrypting and extraction the MIPI data and transmitting the wanted pixel data through to the Pixel intensity counter, Preview generator and to the SDRAM.

MIPI bridge

The MIPI bridge is responsible of receiving serial data from the camera, deserializing it, and translating from MIPI CSI2 to parallel pixeldata. Futhermore it handles the generation of the

[1] X being an integer defined by the configuration data

pixelclock, via a PLL. As the bridge contains a pre-synthesized IP-block provided by Lattice Semiconductors (a blackbox), only the major building blocks will be described.

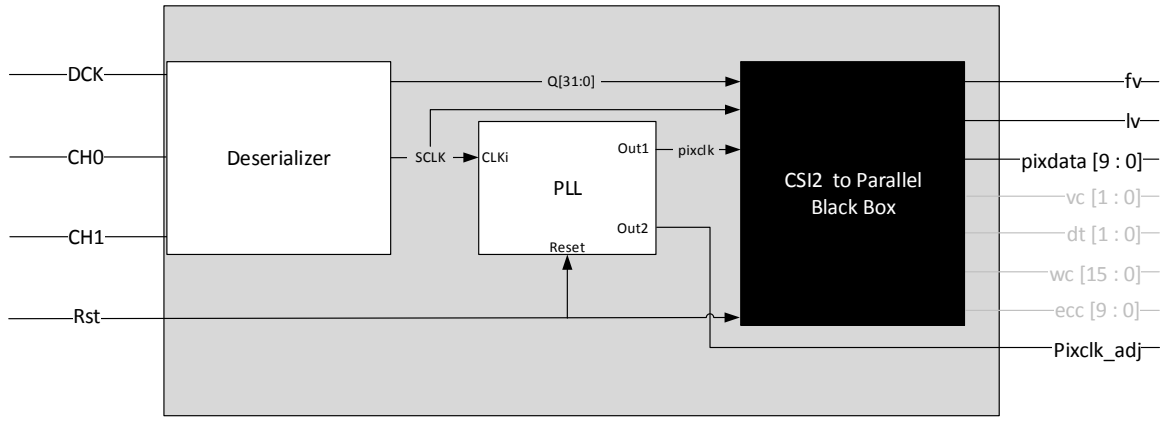


Figure 10.3: MIPI CSI2 bridge input and output.

The deserializer in the bridge receives three signals from the camera, see Figure 10.3, DCK, CH0 and CH1. DCK is the received camera clock which is configured to operate at 107.5 MHz. CH0 and CH1 is the two serial data lanes which each transfers DDR serial data to the MIPI bridge.

The job of the deserializer is to convert serial data to parallel data. Each lane contains 1 bit DDR data, which is converted to 8 bit parallel data.[semiconductor, 2012, p.4] At the same time, the clock, DCK, is scaled down accordingly to match the parallel data rate.

As the input clock is 107.5 MHz DDR, the output clock (SCLK) will therefore be:

$$\frac{107.5\text{MHz} \cdot 2\text{bits}}{8\text{bits}} = 26.875\text{MHz}$$

This clock is sent to the blackbox together with the parallel data. Note that the parallel databus is 32 bits wide, and not 16 (8 bit · 2 lanes). The remaining 16 bits are there to support 4-lane CSI2 configurations.

In addition to the blackbox, the clock is sent to a PLL, which creates two new clocks, a pixelclock for the blackbox and a phase adjusted pixelclock for the output of the bridge.

According to [semiconductor, 2012, p.4], the pixelclock can be calculated as:

$$\text{pixclk} = \frac{1}{(\text{buswidth})} \cdot (\text{number of lanes}) \cdot 2 \cdot 4 \cdot \text{SCLK}$$

In our case the buswidth is 10 bit, number of lanes is 2, and SCLK is 26.875 MHz. The pixelclock

is therefore:

$$\text{pixclk} = \frac{1}{10} \cdot 2 \cdot 2 \cdot 4 \cdot 26.875\text{MHz} = 43\text{MHz}$$

Rst is the reset, this is permanently pulled down and therefore not used. This is because the FPGA will be powered off each time a picture has been captured, thus making the reset functionality unnecessary.

The blackbox has 7 outputs, see the right side of Figure 10.3 which are fv, lv, pixdata, vc, dt, wc, and ecc. The fv, frame valid, is high as long as a frame, an image, is sent through the pixdata. The lv, line valid, is high when a row is being sent through the pixdata. The pixdata is a 10 bit SDR parallel bus, capable of transmitting one pixel in each pixelclock cycle. The vc, virtual channel, dt, datatype, wc, wordcounter, and ecc, error correction, are CSI2 specific protocol data [semiconductor, 2012, p.2], and is therefore confidential and thus undocumented.

Pixel Gate

When the camera is active, it sends a continuous stream of images to the MIPI bridge. The pixel gate is responsible for deciding which of the images to store, and at the same time convert from 10 bit to 8 bit pixel data.

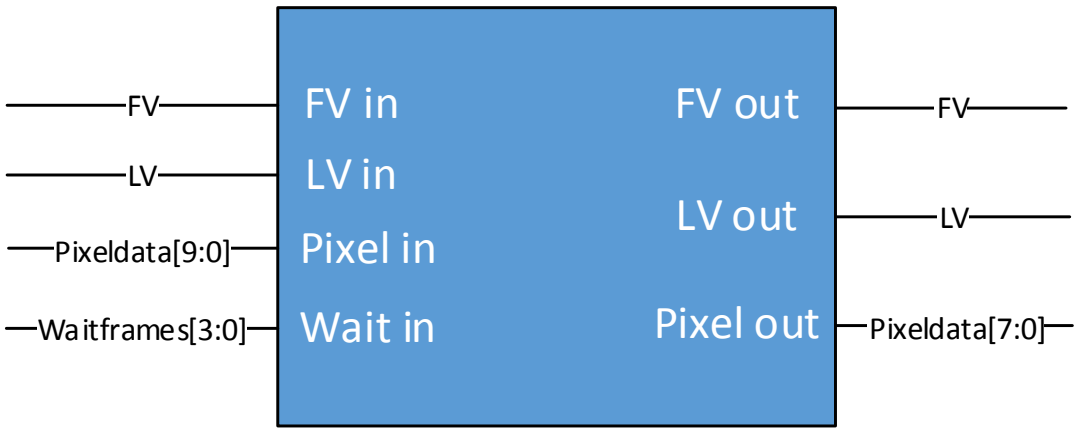


Figure 10.4: Pixelgate input and output.

The pixel gate has four inputs, see Figure 10.4. Three input signals are from the MIPI bridge: The 10 bit parallel pixeldata, the Line valid (LV) signal, and the Frame valid (FV) signal. The last input, Waitframes, controls which image the gate should let through to the rest of the system. This is done by specifying the number of images to ignore, before letting one through. The Waitframes is received from the MCU through the configuration functionality.

As waitframes is 4 bits wide, the number of images to ignore can be set between 0 and 15.

Citing an Engineer from the Raspberry Pi team:

"Do bear in mind that the first frame that the OV5647 produces after starting streaming is always corrupt [..]" [2]

The first frame should therefore never be used, so setting waitframes to 0 is not allowed.

The output (the right side of Figure 10.4) is the 8 bit pixeldata to the rest of the system, alongside a gated version of FV and LV.

As the VHDL code for the gate is fairly short, the functionality of the bridge will be described by going over the actual code. The architecture section of the code can be seen in Listing 10.1

```
1  architecture behavioral of pixelgate is
2
3  signal framecount : std_logic_vector(3 downto 0) := "0000";
4  signal done       : std_logic := '0';
5  begin
6    lv_out <= lv_in and fv_out;
7    pixels_out(7 downto 0) <= pixels_in(9 downto 2);
8    process (fv_in)
9    begin
10     if fv_in'event and fv_in='1' then
11       if (framecount = wait_frames) then
12         if done = '0' then
13           fv_out <= '1';
14           framecount <= "0000";
15           done <= '1';
16         end if;
17       else
18         fv_out <= '0';
19         framecount <= framecount + 1;
20       end if;
21     end if;
22   end process;
23 end behavioral;
```

Listing 10.1: VHDL code for the Pixel Gate

As seen in the code, the translation from 10 bit to 8 bit pixels is done by simply letting the 8 most significant bits of the input be the output.

The Line Valid output (lv_out) is made by and'ing the Line Valid input with the Frame valid output. This ensures that lv_out is only active while the predetermined image is being processed.

The gate needs to count how many frames it has received from the MIPI bridge. This is done in the signal *framecount*. On power up, this is set to "0000", just to make sure it is well-defined

[2] (https://www.raspberrypi.org/forums/viewtopic.php?f=43&t=109137, post by user 6by9, Tue May 05, 2015 5:18 pm)

before proceeding.

Another signal, *done*, is used as a flag to stop the gate after an image has been allowed to pass through.

To count the frames, the gate triggers on a rising edge on the Frame Valid input (fv_in). Every time this happens, the framecounter is compared to the input value on the wait_frames input. If it matches, the *done*-signal is checked, to see if an image has already been allowed to pass. If no image has been passed yet, three things happen:

- The *fv_out*-signal is set high, letting the rest of the system know that the current frame should be captured. As described before, this also activates the *lv_out*-signal.
- The framecounter is reset to 0. This forces the evaluation of **"framecount = wait_frames"** to be false, as waitframes is not allowed to have the value 0. This ensures that *fv_out* is set to low on the next frame.
- The *done* flag is set high, to make sure no more images are let through.

If the framecounter does not match the preset value, the framecounter is incremented, and *fv_out* is set low to tell the system that the current frame should not be captured.

The MIPI bridge and Pixel gate have been explained. Now the Preview, which the Pixel gate transmits pixel data to, will be described and illustrated.

10.5 Preview

This section is to explain how the the preview functionality works. As described in section 5.3 the preview picture is a low resolution version of the original.

Overview of the functionality

The preview is generated by taking the average of a series of pixels and making a new one, in this case 32x32 pixels. For an easier explanation, the explanation will be based on Figure 10.5 and Figure 10.6

Figure 10.5: An illustration on how an row of pixels in the original picture is scaled down, in this case 1/4 the original size.

On the left side of the arrow on Figure 10.5, an example of a pixel row is shown and at the right side the downscaled version. This is done, in this case, by taking the average of four pixels and creating a new one, which contains the intensity corresponding to the average.

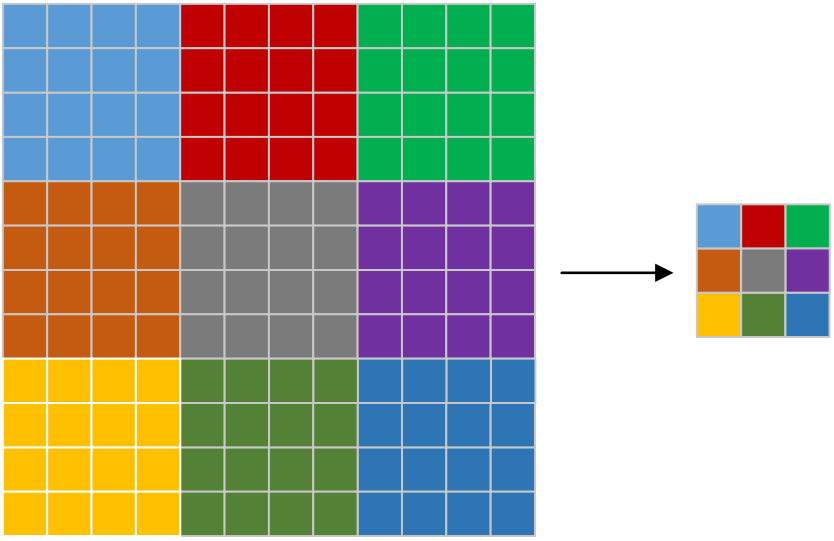


Figure 10.6: An illustration on how a fully sized picture is scaled down to a 1/4 the size of the original picture aka "Preview".

With the description to Figure 10.5 in mind, see Figure 10.6. This figure is an example on a fully sized picture before and after a downscaling, i.e a full resolution and low resolution picture. When the first row is downscaled to a small resolution, the next row is also down scaled. When this is done, the two rows are added. This is done for four rows. When the four down scaled rows are added, the average is taken to create a new pixel, see Figure 10.7 with the pixel intensity corresponding to the average.

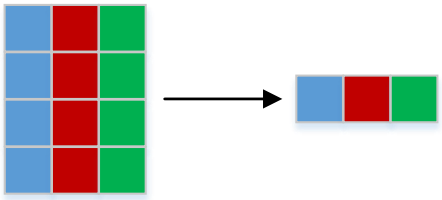


Figure 10.7: An illustration on how four rows are combined to create one new row.

In this example, this happens 4 times for each small resolutions row, i.e the average of four small resolutions rows are taken to create one new row, with the intensity corresponding to the average.

When this happens on a picture taken by the camera, the small resolutions picture should be a factor 32 smaller in length and height than the full resolution picture and instead of just taking the average of 4x4 pixels to create a new, the average of 32x32 pixels is done instead. The full resolution of the picam is 2592 x 1944 pixels, as described in section 6.2. This results in a preview size of 81 x 60 pixels.

The conclusion is, that the preview picture is a lot smaller than the original full sized picture, with the preview having an approximate size of 5 kB. A lot of details will however be cut off. This will make it possible to send down a lot of small pictures instead of one big picture over the same time. Ground control can thereby pick out the best pictures and request for a full resolution version of these.

The preview generator has been implemented by means of a state machine, written in VHDL, and will be described in the following.

Previews states

Here, a closer look will be taken at which states the preview functionality can be in. For an easier explanation, see the state machines at Figure 10.8 and Figure 10.9.

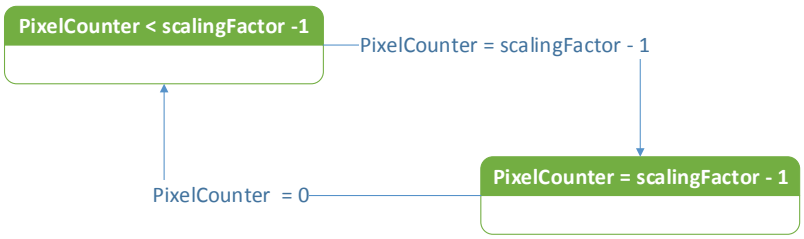


Figure 10.8: The state machine for the counter which counts and sums all the pixels values

Figure 10.8 shows the first state of the preview function. This state counts 32 pixels from a row and sum the pixels values. When the 32 pixels are summed, a flag is set to signal to the state machine that it should enter the next state, waiting, as seen on Figure 10.9. This state waits on the counter state, seen on Figure 10.8, to set the flag sumComplete high.

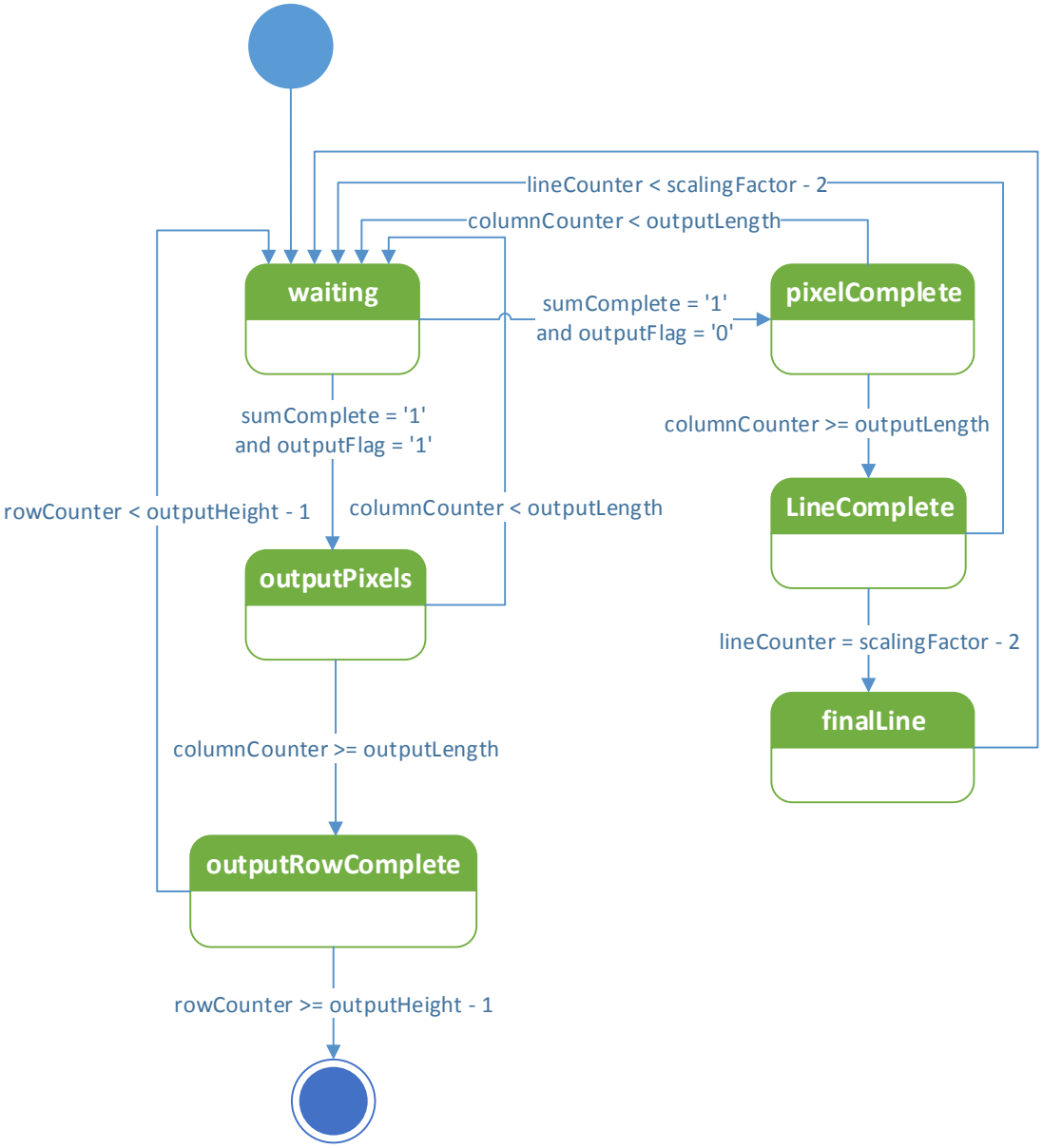


Figure 10.9: The state machine of which states the preview generator can be in, when it generates the small resolution picture

After the state, *waiting*, there are two possible ways of transmission. To describe the state machine seen in Figure 10.9, the flow of the description will be described in the same order as the states flows when the state machine is running.

The next state is *pixelComplete*. This is the most interesting state and therefore the code to

this state is shown, see Listing 10.2.

```
1  when pixelComplete =>
2      case SRAM_ps is
3          when do_read =>
4              SRAM_WE <= '0';
5              SRAM_clkEN <= '1';
6              SRAM_address <= columnCounter(8 downto 0);
7              SRAM_ns := do_write;
8          when do_write =>
9              SRAM_WE <= '1';
10             result := (others => '0');
11             result(14 - scalingPower downto 0) := pixelsum(14 downto scalingPower);
12             SRAM_data <= SRAM_Q + result;
13             SRAM_ns := done;
14         when done =>
15             SRAM_WE <= '0';
16             SRAM_clkEN <= '0';
17             columnCounter := columnCounter + 1;
18             SRAM_ns := do_read;
19             if(columnCounter >= outputLength) then
20                 columnCounter := (others => '0');
21                 lineCounter := lineCounter + 1;
22                 SM_ns <= lineComplete;
23             else
24                 SM_ns <= waiting;
25             end if;
26         when others =>
27             SRAM_ns := do_read;
28     end case;
```

Listing 10.2: Code that shows what happens in the state pixelComplete

In this state there is three cases: *do_read*, *do_write* and *done*. These will each be described in the same flow as shown in Listing 10.2.

- do_read

  In this state the preview enables the read function on the FPGA so it is possible to read from the SRAM. The address which is to be read from is determined in line 6 Listing 10.2. After *read* is enabled the FPGA will read from the specified address.

- do_write

  In this state the preview enables the *write* function, so that the FPGA can write to the SRAM. In line 11, the average of the pixelSum is taken. This is done by bit shifting five times, which is equal to dividing pixelSum with 32, corresponding to the 32 pixel values. Afterwards, in line 17, the value is added to the read data from do_read and stored on the address do_read reads from.

  Line 12 is responsible for summing the lines to be made into one. When entering *pixelComplete* for the first time at the beginning of a new line, the first pixel stored in the row is read. Then a new pixel is received from the pixel counter, see Figure 10.8. This pixel bit shifted five times to get the average (since pixel counter only sums pixel values). The new pixel value is then added to the previous values read from the SRAM, and the sum is stored in SRAM.

This process will take place 31 times for each pixel address, i.e 31 rows with a length of 81 pixels, an example of which is shown on Figure 10.7.

- done

  This state keeps track of whether the preview generator is finished with making a line of the preview. It counts the numbers of columns that has been summed, to determine which column it is currently making, out of the 81 columns in the preview. The if statement seen on line 19 is used to determine if the row is complete, which happens when the row is 81 pixels long. If it is 81 pixels long, the state machine will go to *lineComplete*, if not it will go back to *waiting* and start over till it reaches a full row of 81 columns.

The next state is *lineComplete*. Each time this state is entered, a counter will add one to *linecounter*, a variable that contains the number of how many lines that have been made. When this state has been reached 31 times, it will signal the next state *finalLine* to set the flag outputFlag high and then return to *waiting*. This signifies that when the next line of pixels arrive, it will be the final line needed to generate that particular line of the preview.

When a new pixel is received from the pixel counter, and sumComplete and ouputFlag are set high, the state *outputPixels* will be entered. This state is similar to *pixelComplete* but instead of storing the pixel value in SRAM it will be stored in SDRAM, so the MCU can read the data. The states "do_read" in Listing 10.3 will not be explained here, because it is identical to the case "do_read" in Listing 10.2 and the case "done" will only be explained briefly.

```
1  when outputPixels =>
2      case SRAM_ps is
3          when do_read =>
4          when do_write =>
5              SDRAM_WE <= '1';
6              result := (others => '0');
7              result(14 - scalingPower downto 0) := pixelsum(14 downto scalingPower);
8              outputValue := (SRAM_Q + result);
9              SDRAM_data <= outputValue(15 downto scalingPower);
10             SRAM_ns := done;
11         when done =>
12             SRAM_WE <= '0';
13             SRAM_clkEN <= '0';
14             SDRAM_WE <= '0';
15             columnCounter := columnCounter + 1;
16             SRAM_ns := do_read;
17             if(columnCounter >= outputLength) then
18                 columnCounter := (others => '0');
19                 SM_ns <= outputRowComplete;
20             else
21                 SM_ns <= waiting;
22             end if;
23 end case;
```

Listing 10.3: Code that shows what happens in the state *outputPixel*

- do_write

  When this state is entered, the last state for a low resolution pixel is entered. On line 5 the read function to the SRAM is enabled, where the previous stored pixel data is read. On

line 7 the received pixel sum, from the state *pixelSum*, is bit shifted 5 times to calculate the average. The two values are in line 8 added and in line 9 bit shifted 5 times to calculate the average of the 32 averages and then stored in the SDRAM.

- done

  The only difference between this case and the case *done* in Listing 10.2, is that the next state can be *outputRowComplete* instead of *finalLine*. If the row is 81 pixel long the next state will be *outputRowComplete*, if not, *waiting*. I.e. the state machine switch go between the waiting state and the *outputPixel* state, until it has outputted an entire line and then go to *outputRowComplete*.

The state *outputPixel* is the second last state of the state machine. This state keeps track of where the low resolution picture is in its process i.e if one row is fully completed or only half way. The state final state, *outputRowComplete*, keeps track of how many rows the preview has. When the total count of rows is equal to 60, the preview is completed.

Given that the preview has been explained it now follows to explain another crucial part in creating information for compressing data, namely the counter array which will be described in the following section.

10.6 Counting array

In the following section, the flow, functionality and implementation of the counter array, is described and illustrated.

The purpose of the counting array is to register each pixel intensity and count the quantity of each in their equivalent address. In Figure 10.10 an illustration of a counting array is displayed. The array contains the frequency of occurrences. Each frequency of occurrence is in the address which is equivalent to the intensity. An example of this would be, as shown in Figure 10.10, intensity 0 in the array address 0 with an frequency of occurrence of 6.

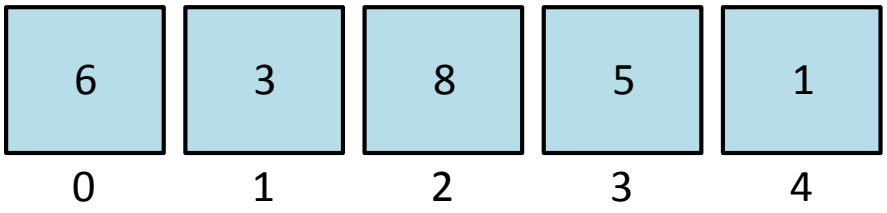


Figure 10.10: An illustration representing a counting array.

10.6.1 Overall flow and functionality

The flowchart in Figure 10.11 describes and illustrates how the counter array is implemented.

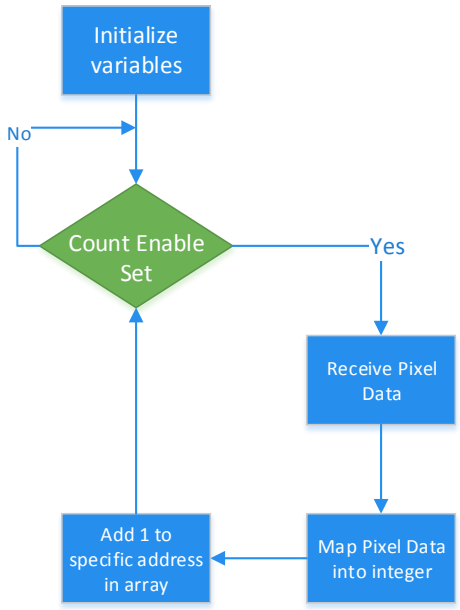


Figure 10.11: A flowchart displaying how the counter array is implemented.

The code will perform the task of creating the counter array needed for the compression in the MCU. To describing the process, the following combined steps will happen on every rising clock if the *count enable* is set;

1. Pixel data[3] is received from the pixel gate.
2. The pixel data is cast to an unsigned and then casted to an integer. The integer which have a value between 0 and 255, is used as an index for the counter array.
3. The value of a specific address is incremented with 1.

The Counter array i 256 long and 24 bits deep. This size makes it possible to store all the pixel occurrences in one address, if the situation of only one pixel value is presented in a whole image. One address can thereby contain a value higher than 5 millions.

The Counter array, which is needed for the Huffman compression, has been described. Now the Address decoder which has to multiplex between the temporary storage spaces from which the MCU wants to connect is to be explained in the following section.

[3] A binary value

10.7 Address decoder

The address decoder has one purpose to act as a linker between the different storages units and the MCU. It needs to be able to communicate with all the different interfaces and link the data between those.

Implementation

To fulfil the requirements set in Section 10.1: FPGA requirements, the address decoder needs to be able to multiplex the bus between the different storage and configuration units. The way of how this is done can be seen on Figure 10.12.

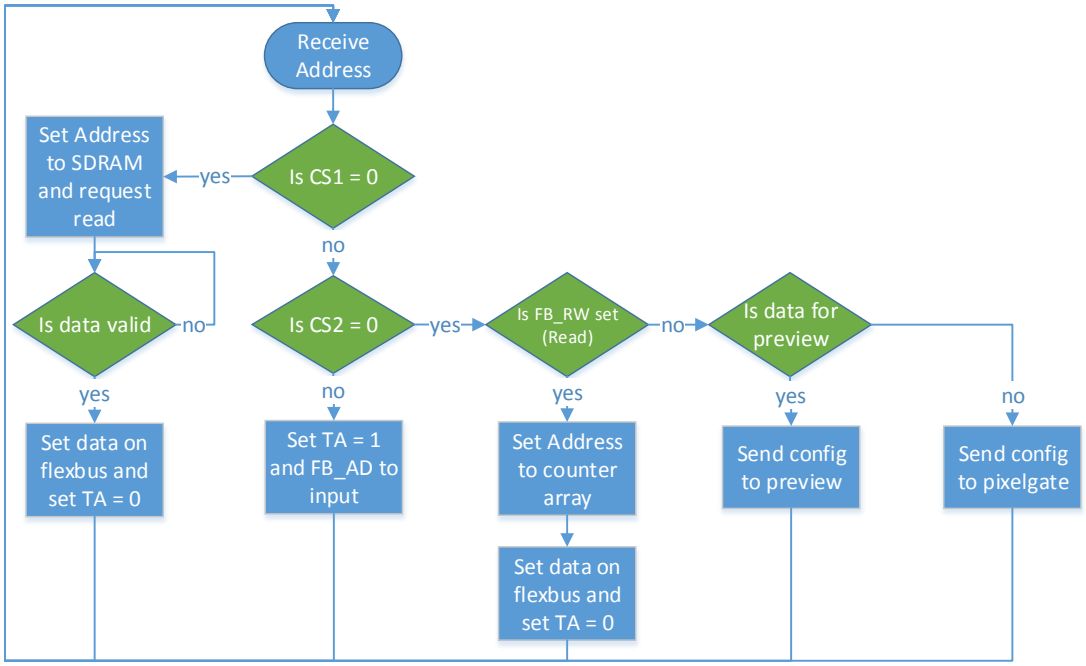


Figure 10.12: A flowdiagram showing how the address decoder handles the multiplexing between the different storage units and configuration registers

From Section 8.3: Other functionalities it is known that two chip selects are used, one for the data stored in the SDRAM, memory unit 1, and one for the counting array and configuration handling, memory unit 2. When receiving an address, the address decoder first checks if the address belongs to memory unit 1 or 2. If memory unit 1 is selected the data is first fetched from the SDRAM and then set on the flexbus pins. If memory unit 2 is selected the address decoder first check if a read or write is requested. When a write is requested the address decoder, then checks if the data should be passed to the preview generator or the pixelgate. When a read is requested, the address is passed to counter array and the corresponding counter data is then set on the flexbus. If no chip selects are 0 the flexbus pins are set to input and TA is set to 1.

Flexbus interface

The main interface is the flexbus, and a timing diagram of a burst read from the Flexbus can be seen on Figure 10.13.

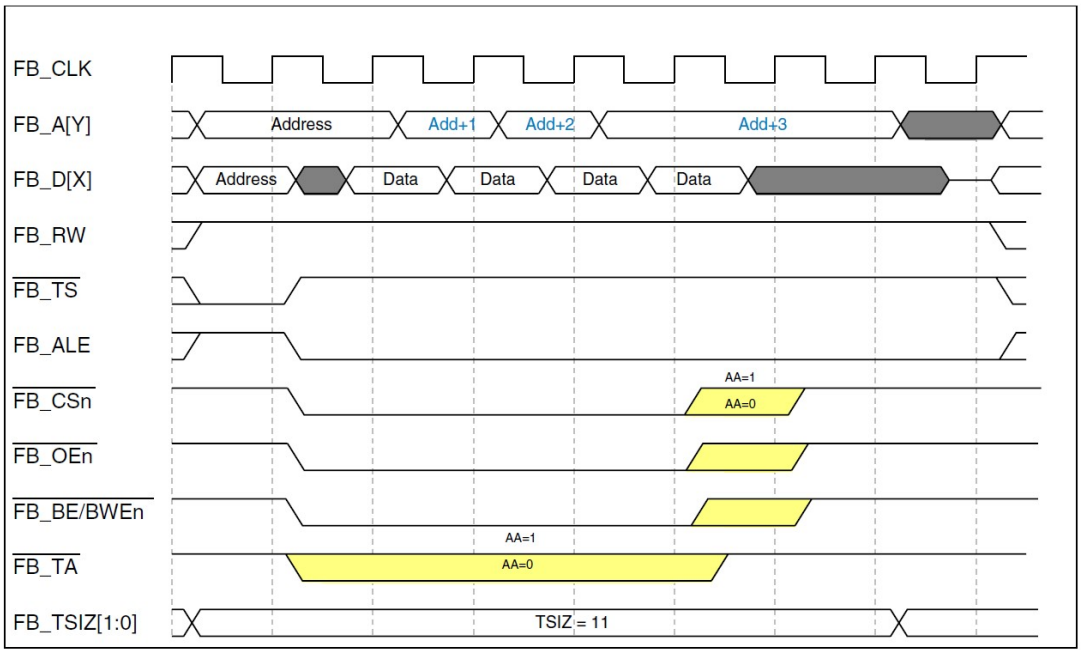


Figure 10.13: The timing diagram of a Flexbus 32-bit-Read burst from an 8-Bit port [Freescale, 2012b, p. 758]

To uphold the timing set in Figure 10.13 the address decoder is clocked via the flexbus. The address is latched at a falling edge where the signals are set to be stable. The address is then send to three places both of the memory units and a temporary register as shown in Listing 10.4.

```
1 if falling_edge(FB_CLK) then
2     if FB_ALE = '1' then
3         temp_addr <= FB_AD(21 downto 0);
4         counter_addr <= FB_AD(9 downto 2);
5         MCUAddr <= FB_AD(21 downto 0);
6     end if;
7 end if;
```

Listing 10.4: Snippet of address decoder showing how the address is latched to intern registers

When the address have been latched, the MCU pulls the desired chip select low, the address decoder then checks whether it is the images (SDRAM) or the counter array (internal memory unit), that should be active.

The communication to the two memory units can be seen on the CD. From the MCU all addresses under FB_CS1 is write protected section 8.3 and therefore is it only necessary to handle a read operation to from the SDRAM. Because the flexbus uses *burst read*, two reads are needed, to get

32 bit of data from the SDRAM. The address decoder first sets the data on the flexbus, when all 32 bit has been stored in a temporary register. This can be seen in the snippet Listing 10.5

```
if FB_CS1 = '0' then            -- SDRAM
    if (read_done = '0') then
        if ((ready = '1') and (request = '0')) then
            MCURequest <= '1';
            request := '1';
        elsif ((ready = '0') and (request = '1')) then
            MCURequest <= '0';
        elsif ((MCUDatavalid = '1') and (request = '1'))then
            if SDRAM_LSB => '1' then
                temp_data(15 downto 0) <= MCUout;
                MCUAddr <= temp_addr + 1;
                SDRAM_LSB <= '0';
                request := '0';
            else
                temp_data(31 downto 16) <= MCUout;
                MCUAddr <= temp_addr + 1;
                SDRAM_LSB <= '1';
                request := '0';
                read_done <= '1';
            end if;
        end if;
```

Listing 10.5: The part of the address decoder showing how 2 words are fetched from the SDRAM

After this is done the data is then passed on two consecutive clock cycles to the MCU.

An equal implementation is done for memory unit 2, but here the system first checks whether the request is a read or write. For a read, the data is passed directly. A write uses an address pin to see if the configuration shall be passed to the preview generator or pixelgate. This can be seen in Listing 10.6.

```
elsif FB_CS2 = '0' then         -- counter array
    if FB_RW = '1' then
        if FB_OE = '0' then
            if MSB_en = '1' then
                FB_AD(15 downto 0) <= counter_out(31 downto 16);
            else
                FB_AD <= counter_out(15 downto 0);
                MSB_en <= '1';
            end if;
            FB_TA <= '0';
        end if;
    else
        if temp_addr(10) = '1' then
            -- pass configuration to preview
        else
            -- pass configuration to pixelgate
        end if;
    end if;
```

Listing 10.6: The part of the address decoder handling memory unit 2

The actual passing of the configuration has not been implemented, since all subsystems has their configuration hard wired. Given that every part of the FPGA has been described, the test of the implemented subsystem follows.

10.8 Test

The requirements for the FPGA: Data extraction, preview generator and pixel intensity counting for encoding, is tested in the following section.

Data extraction

The data extraction blocks can now be tested, to ensure they comply with the requirements in section 10.1.

Pixel gate test
The Data extraction functionality that the pixel gate is to fulfil will be tested partly in the following. However, the functionality in regards to relaying the output of the gate to both the SDRAM, counter array and preview generator is not tested, since it is the top file in the VHDL project that is responsible for this. Therefore, the following is tested:

Requirements to be fulfilled:

- Relay the input data to the output, after a set number of frames, see 10.4.
- Convert the received data to 8-bit pixel data.

Testing Procedure:

The *FV in* and *FV out*, see Figure 10.4, signals are brought out as pins on the FPGA, so they can be monitored with an oscilloscope.

The camera is set to transmit image data which goes through the MIPI bridge. For testing purposes, the pixel gate is set to wait for 8 frames, then relay one frame, after which it repeats the cycle.

Results:
The *FV in* and *FV out* signals are monitored using an oscilloscope, and the result can be seen on Figure 10.14.

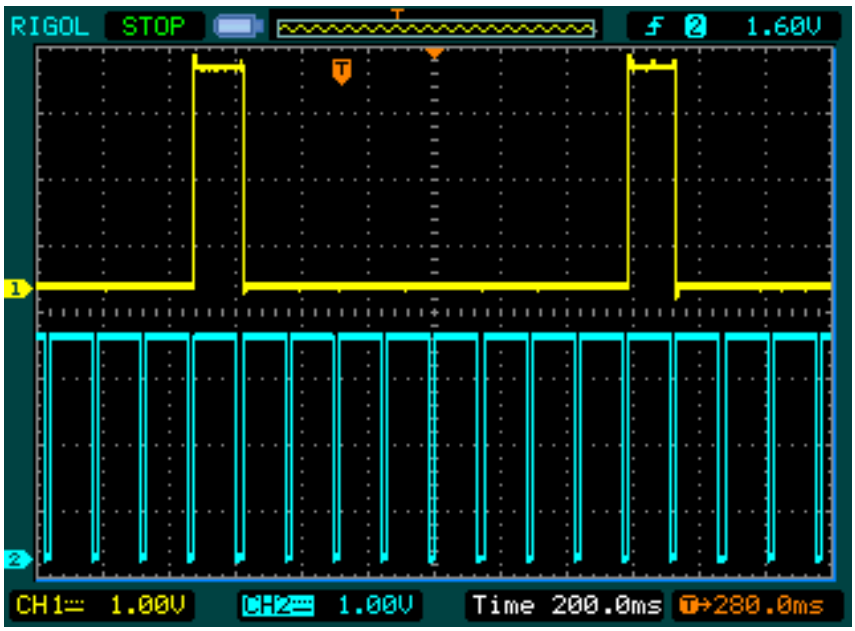


Figure 10.14: The *FV in* pin (bottom) and the *FV out* pin (top). Notice how *FV out* goes high once every 8'th time the *FV in* goes high.

In regards to the second requirement, the following is found in the pixel gate VHDL file:

```
entity pixelgate is port(
            pixels_in      : in std_logic_vector(9 downto 0);
            ...
            pixels_out     : out std_logic_vector(7 downto 0)
    );
end pixelgate;
```

Listing 10.7: A part of the declaration of the pixelgate entity, where the pixel input and output is shown

```
    pixels_out(7 downto 0) <= pixels_in(9 downto 2);
```

Listing 10.8: The mapping of 10 bit input data to 8 bit output data

Showing how the 10 bit input data is converted to 8 bit output data, through bit shifting.

Conclusion:
The input and output data is connected as a wire, meaning that these are always connected. As can be seen on Figure 10.17, the pixel gate is able to relay a single FV-signal, after a set interval of wait frames. Therefore, the pixel gate works as desired.

MIPI bridge test
The MIPI bridge handles the data-reception from the camera, and converts the CSI2 data to parallel pixel data. To verify that the output data is valid, and not random noise, the bridge is tested by sending the pixel intensities to a VGA monitor.

Test Procedure:
This is done using a dual port RAM as a buffer, and a VGA sync generator for generating the VGA timings. The inner working of the sync generator is not essential for this project, and it is therefore treated as a blackbox.

The overall connections between the components used in the test can be seen on Figure 10.15.

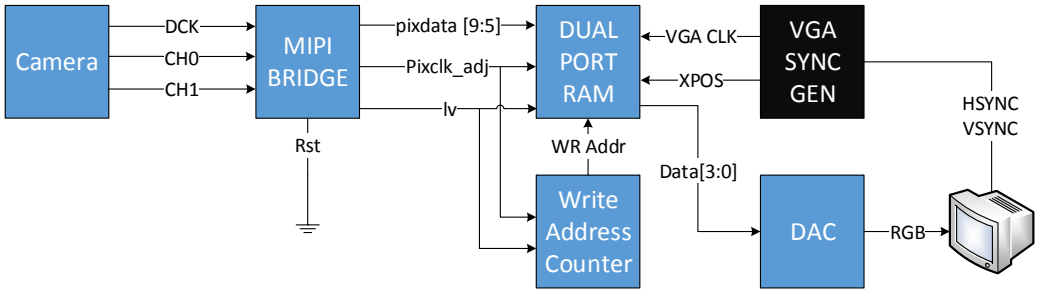


Figure 10.15: Diagram showing the components used for the test

The dual port RAM is generated using the build-in memory block generator in the FPGA IDE. It is configured to have 640 addresses, and a word length of 4 bits. One of the RAM ports is used for writing, and is connected to the MIPI bridge by the following connections:

MIPI Bridge:	RAM:
Pixel clock	Write clock
Line valid	Write enable
Line valid	Write clock enable
Pixeldata [9:5]	Data in

By connecting it this way, data is only written to the RAM when valid pixeldata is available. The VGA DAC has 4 input bits per color (Red, Green and Blue). Only the 4 most significant bits of the pixel data is used, as the same value has to be written to all 3 color channels to produce a greyscale image.[4]

The Write address is handled by a counter, which counts the pixelclock while Line valid is high, and resets when Line valid is low.

This ensures that the first pixel of each line is written to RAM address 0, the next to address 1, etc.

[4] A greyscale representation of the image is chosen, as a bayer-data processor is needed to produce a color image

The other port of the RAM is used for reading. This port is clocked by the VGA sync generator. The address input is connected to an Xpos output on the sync generator. This signal represents the horizontal position of the pixel being displayed on the screen at any given time. The RAM output data is sent to the VGA DAC, which sends the analog VGA data to the screen.

Note that no attempt has been made to synchronize the vertical axis between the camera and the screen. The side effect of not syncing, is an image on the screen that rolls slowly from top to bottom. Syncing is certainly possible, but not needed for this test, as it only serves to actually see the data from the camera.

The test itself consists of configuring the camera to output 640x480 images at 60 frames per second, and watch the screen to see if data is received.

Results:

The test result can be seen in Figure 10.16.

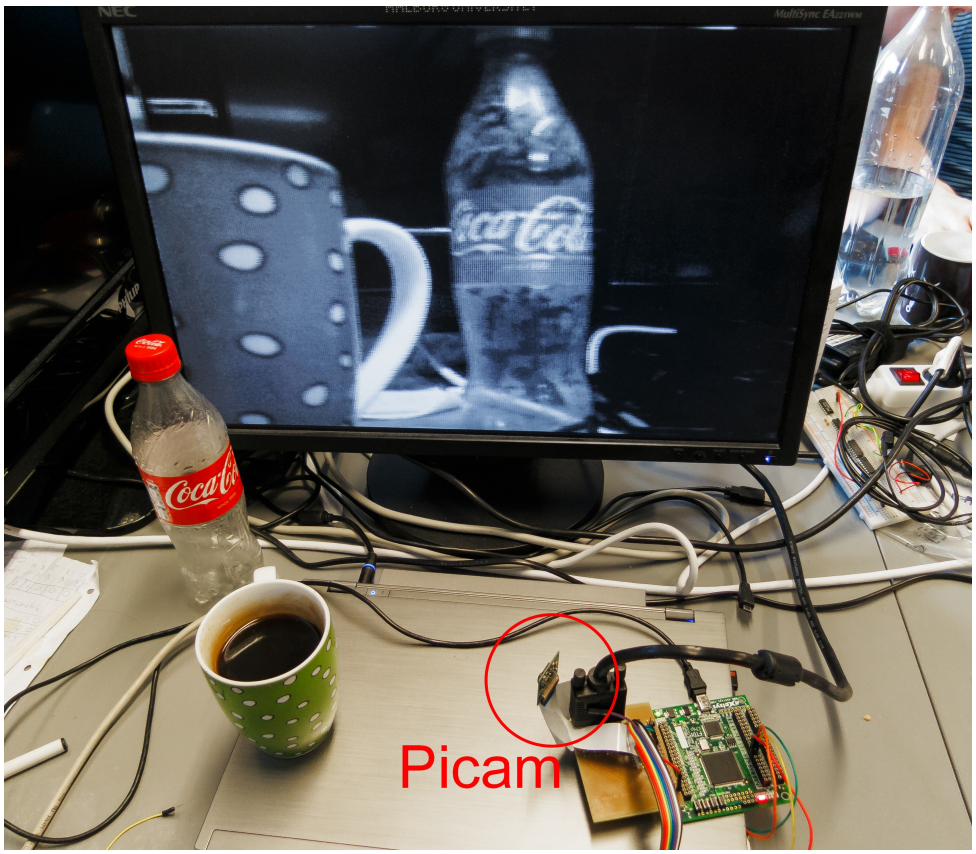


Figure 10.16: The camera displaying an image on the screen

Conclusion:
As seen, the test was a success, and the MIPI bridge has therefore passed the requirement, for

receiving data from the camera.

Address decoder & Flexbus test
The address decoder and Flexbus are tested simultaneously, since they are deeply connected and their functionalities are based on the same requirement, which is *Transmit data from the memory units to the MCU*.

Test Procedure:
An SRAM block is made in the FPGA, where the data is pre-initialized. The SRAM block is set to have an data width of 32 bits, and an address width of 8 bits. The initialization means that all addresses are assigned the same 32 bit data, namely the eight HEX characters *0xdeadbeef*. However, address number 211 is given the data *0xfeedbabe*. The reason why the test is based on an SRAM block in the FPGA is that it was not possible to get the SDRAM controller working in time for this test.

The address decoder is set up to relay the data from the SRAM, when data requests occur on the Flexbus.

In the MCU, the Flexbus is initialized, code that can be seen in Listing 8.7 is used to step through the addresses and the print the data.

Results:
The result of the test can be seen in Figure 10.17, where a section of the printed data is viewed. Notice how all addresses show the same data, except address 211.

```
counter 200 :      deadbeef
counter 201 :      deadbeef
counter 202 :      deadbeef
counter 203 :      deadbeef
counter 204 :      deadbeef
counter 205 :      deadbeef
counter 206 :      deadbeef
counter 207 :      deadbeef
counter 208 :      deadbeef
counter 209 :      deadbeef
counter 210 :      deadbeef
counter 211 :      feedbabe
counter 212 :      deadbeef
counter 213 :      deadbeef
counter 214 :      deadbeef
counter 215 :      deadbeef
```

Figure 10.17: Data read by the MCU over the Flexbus. Notice how the data at address 211 is different than the rest, as it is supposed to.

Conclusion:
Since the MCU is able to read all the data from the SRAM successfully through the address decoder and the Flexbus, both of these work as intended.

Data extraction conclusion
The first requirement for the test of Data extraction, *receive data from the camera*, has been passed both shown in *MIPI bridge test* and *Pixel gate test*. In the *MIPI bridge test* in Figure 10.16 the data has been extracted from the camera and is displayed on a screen. In *Pixel gate*, in Figure 10.14, the received data from the MIPI bridge (and the camera) can be seen. Also in this figure the requirement, *Relay the recieved data to preview generator, counting array and the SDRAM, after a set number of frames*, has been tested with a oscilloscope, and the output only goes high every 8'th time. This shows the test is passed. Finally, in *Address decoder & Flexbus test* the last requirement in Data extraction, *Transmit data from the memory units to the MCU*, is fulfilled, seen in Figure 10.17 the transmitted data is consistent with the displayed data in the terminal.

The test is passed and the requirements for data extracting has therefore been fulfilled.

Preview generation

Because of time constraints it have not been possible to implement the preview generation. It has therefore not been possible to perform the two test, from the start of the chapter, *generate a preview and save the preview in SDRAM*.

Pixel intensity counting for encoding

To ensure the requirement, *Keep track of the frequency of occurrence of the pixel intensities in the captured image* is fulfilled to following test has been performed;

Test procedure:
The test consists of two states:

1. Send data to be incremented in the counter array from address 0 to 255 in the FPGAs gates.
2. Second, read the counter array and guarantee that the expected result is obtained.

In the first state of the test, the address and the frequency of occurrence is identical. The frequency of occurrence for address 0 will be 0, for address 1, 1 etc.. The second state reads the values of the counter array from the FPGAs gate, see section 6.4. Each value, with its associated address, is displayed with a small interval on the FPGAs LEDs. An example of this is displayed in Figure 10.18.

Results: The 8 LEDs displays each value from 0 to 255 with a short interval, on Figure 10.18 the value 50 is seen in address 50.

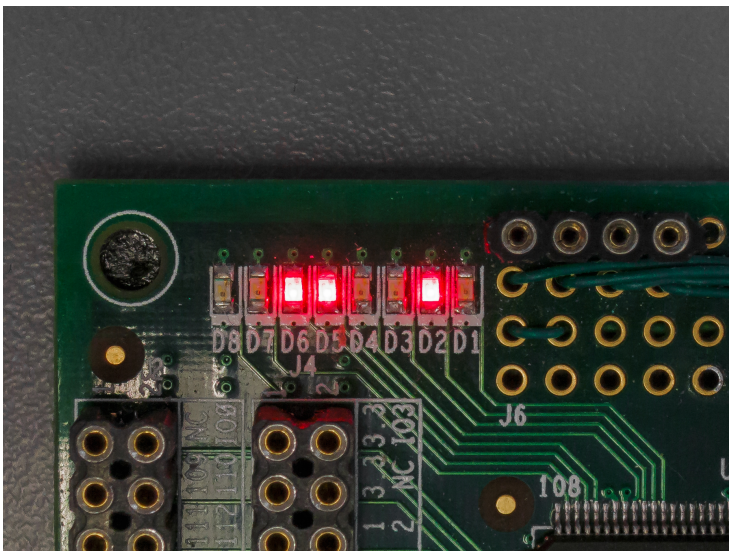


Figure 10.18: A test using 8 LEDs, which displays a binary value of 50 from address 50.

Conclusion: The requirement, *Keep track of the frequency of occurrence of the pixel intensities in the captured image*, has been fulfilled through the performed test.

The functionalities and their implementation in the FPGA, the extraction of data from the Camera using a MIPI bridge, the preview and the counter array, has been described, illustrated and tested. In the next chapter the implementation and test of the external RAM will be described.

11 | External RAM

In section 10.6 it was determined how the counting array would be stored. The following is therefore about how the full resolution picture and the low resolution picture, Preview, is stored.

11.1 RAM requirements

Continuing from what was already explained in section Section 6.2: Choice of hardware this section will describe the needed requirements for SDRAM in explicit detail.

Temporary storage capability

The functionality of this function is to be able to store an image from the extractor and make it available to the data compressor unit. It should also contain the preview, before storing it on the SD-card.

The requirements can generally be split in 6 general requirements determined in section 6.1 and 2 specific requirements for the AS4C4M16S SDRAM.

General requirements

There are, generally, 6 requirements to be met. These 6 requirements are deemed important for upholding easier timing constraints, in accordances to read and write timing in the system. Given that these terms are met, the system will be able to unload an entire image into the external SDRAM, giving the MCU more headroom when storing and collecting data. The general requirements are as follows:

- The external SDRAM shall be able to store both the entire image and the preview image at the same time. See section 6.1

  A standard full resolution image needs 5 MB storage space Section 6.2: Choice of hardware

  A Preview image needs 5 kB storage space Section 10.5: Preview

- The read port, chosen in Section 6.2: Choice of hardware, should be made, to make it possible to mimic SRAM.

  This is to ease the mapping in the address decoder see section 10.7

- Both the full image and the preview should have separate port entrances

  See chapter 6

- The SDRAM controller should be able to write data with 1 Gbit/s

See Chapter 3.1: MIPI

- The SDRAM controller should be able to read data up to 400 Mbit/s

  The MCU's maximum transfer rate is estimated to less than 400 Mbit/s based on Section 8.3: Other functionalities.

- The SDRAM should be able to store the data as long as power is supplied

  This is necessary because the time it takes to process the image can vary and therefore is unknown.

Specific requirements

If the RAM is to fulfil the listed requirements, more specific SDRAM controller requirements must be met. These requirements concern the more hardware specific needs, required to create a functional RAM storage. The following requirements tackle both the RAM controller and the RAM chip.

- The SDRAM shall be refreshed minimum 4096 times in a period of 64 ms in order to withhold the stored image data.

  See [Memory, 2011].

- The SDRAM must run at a 133 Mhz clock speed, maximizing the data transfer rate[1].

  See [Memory, 2011].

Finishing the list of requirements, it comes down to implementation of the SDRAM.

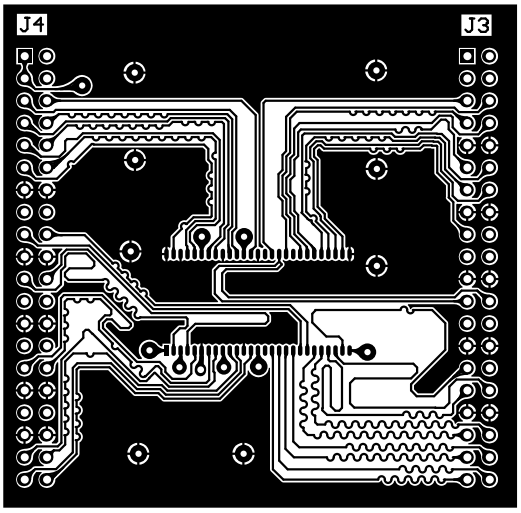
11.2 Implementation

The SDRAM does not have a comprehensive list of requirements concerning the physical chip. There are however few design rules which can improve timing when working with synchronous RAM. The traces on the PCB board, from chip to FPGA, is to be of equal length, giving the signals the same propagation delay through the board. This create an optimum environment for creating the controller. The routing on the FPGA breakout board has not been taken into account, and it is assumed that Lattice have matched the length of all traces. Figure 11.1b shows a rough simulation[2] in Altium Designer where a unit step on a data trace is measured. It shows a ripple which could be caused due to capacitive and inductive parasites on the board. The problem could be solved with proper termination of the pins, but is not handled in this project.

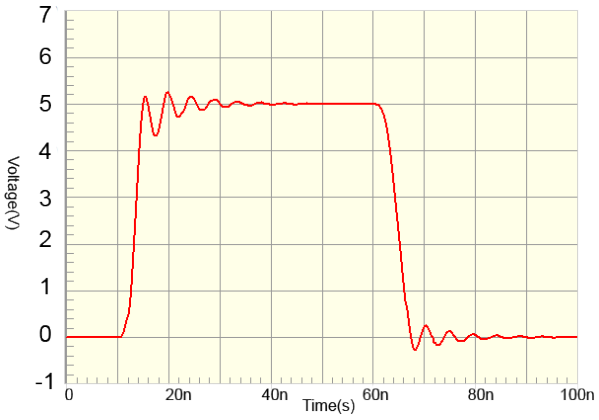
[1] Maximum clock speed is 143 MHz but a margin is used for reliability reasons

[2] Some variables have note been taken into account, such as copper thickness of the trace, board composition and slew rate of the FPGA

The board has been routed upholding a 40.5 mm length ± 0.5 mm on every trace. The PCB layout can be seen on 11.1a. The chip's VCC has been routed to the FPGA boards VCC pins and have been fitted with decoupling capacitors for a stable supply.



(a) Outline of the PCB board



(b) Simulation of a 5 volt and unit step on the data traces

Figure 11.1: The PCB board design and simulation of a unit step on the board

Concluding the physical hardware implementation, the FPGA programming for control of the SDRAM is next.

Design of an SDRAM Controller

The design of the SDRAM controller is broken into two parts:

- A controller which communicates with the SDRAM, handling refreshes and read/write operations.
- A triple port which communicates with other external components tunnelling request to the controller.

Beginning with the larger perspective, the triple port will be described first, since it holds the least timing constraints. An overview of the operations, which the triple port has to control, can be seen by a graphical representation on Figure 11.2.

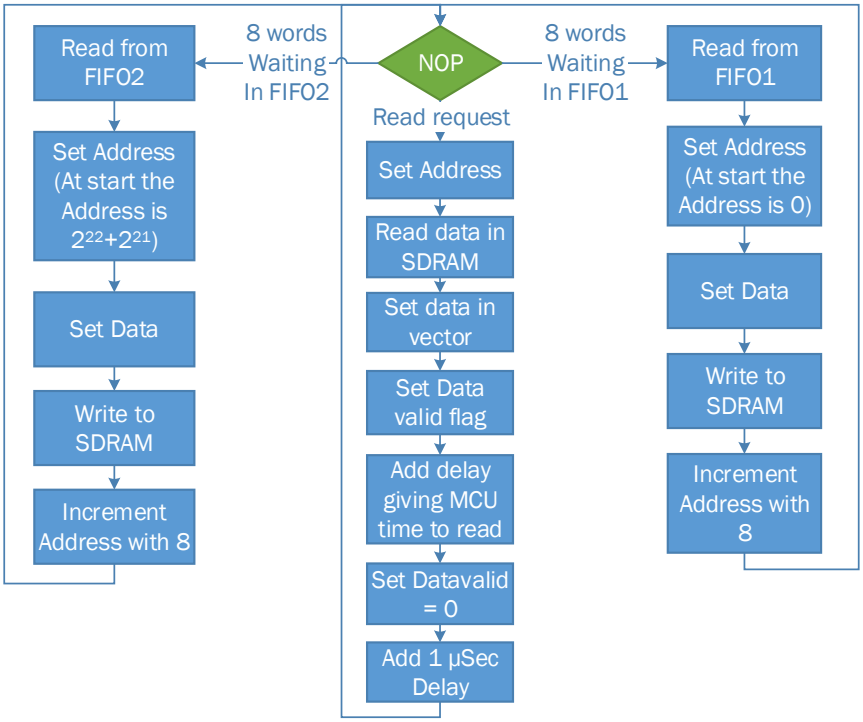


Figure 11.2: Flowchart describing the overview of the triple port controller

Going through the flow on Figure 11.2, it shows three main tasks:

1. Receive data from the pixelgate
2. Receive data from the preview generator
3. Transfer data to the MCU

The triple port will be acting like a buffer for the three components, giving possibility of asynchronous communication. When the RAM has been initialized, it will be waiting in a NOP[3] state, waiting to either receive or transmit data.

Pixelgate and Preview communication
When the preview generator or pixelgate transmit data to the controller, the information will be stored in a FIFO. The preview generator and pixelgate each have their own FIFO. The two FIFO's also handle the task of putting 2 pixels together to a 16 bit word. The controller will detect the FIFO filling up and once it reaches a total of 8-words, it will copy the data stored in the FIFO into the SDRAM. After storing the data it increments the address in the SDRAM by 8 and prepares for the next transfer. The difference between the pixelgate port and preview

[3]No-Operation

port, besides that they are two separate FIFO's, is that the start address is 0x0 for the full resolution image, while the preview image starts at address 0x600000. Using these addresses as start addresses, guarantees that there is enough storage available for the full resolution picture and the preview in the SDRAM.

MCU communication

The communication with the MCU is done with a emulated SRAM interface where 16-bits are latched to the output registers. This is done since the MCU only needs to access one address at a time. Since outgoing communication from the triple port to the MCU is asynchronous, a data valid flag is utilized for telling the MCU when the data has been loaded. The data will be valid until a new read is requested where the data valid will be cleared.

Having now explained how the triple port controller works, the SDRAM controller is next.

Setup and Timing

This part of the controller is the most time critical since many timing constraints needs to be met in order to have a functional RAM chip. When using a RAM chip it first has to be initialized. The initialization is interpreted via the timing diagram on Table 11.1.

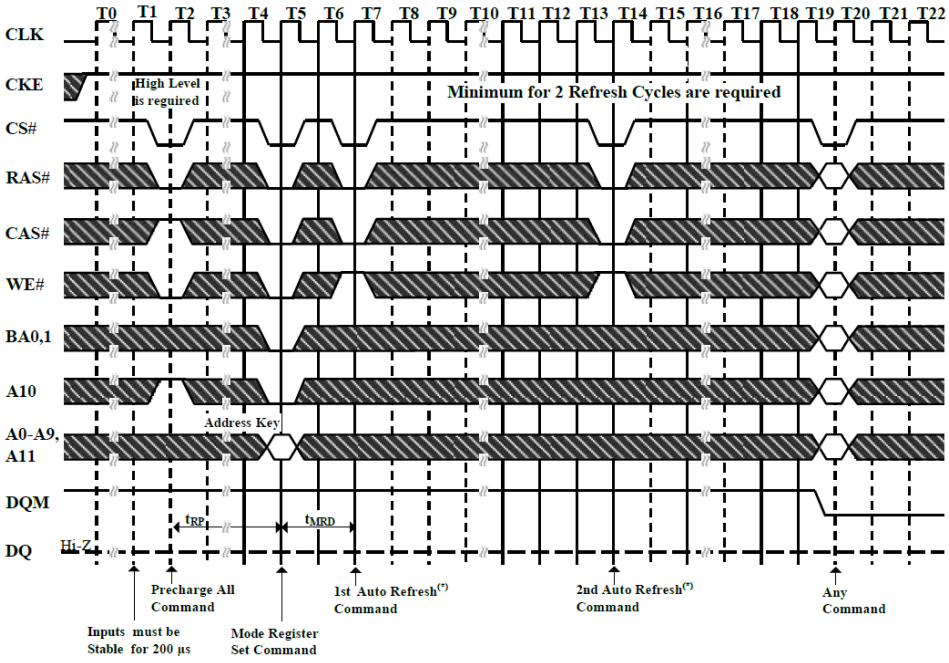


Table 11.1: Table showing the timing constraints needed for proper communication with the external RAM. The table is from the SDRAM Datasheet [Memory, 2011, p. 24]

.

The initialization sequence shows, that all input signals must be stable for 200 μs before start-up. After the signal is stable a *PrechargeAll* command is executed, which precharges all banks. The next command to be executed is the *"Mode Register Set"* command. This command specifies how the SDRAM is to operate, i.e:

CAS Latency Describes the delay in clock cycles, from a read is requested, to the SDRAM chip sends back the data.

Burst Length Describes how many words 1 read/write request handles, ranging from 1 word to a full page of 256 words

Table 4. Truth Table (Note (1), (2))

Command	State	CKE_{n-1}	CKE_n	DQM	BA0,1	A10	A0-9,11	CS#	RAS#	CAS#	WE#
BankActivate	Idle(3)	H	X	X	V	Row address		L	L	H	H
BankPrecharge	Any	H	X	X	V	L	X	L	L	H	L
PrechargeAll	Any	H	X	X	X	H	X	L	L	H	L
Write	Active(3)	H	X	V	V	L	Column address (A0 ~ A7)	L	H	L	L
Write and AutoPrecharge	Active(3)	H	X	V	V	H		L	H	L	L
Read	Active(3)	H	X	V	V	L	Column address (A0 ~ A7)	L	H	L	H
Read and Autoprecharge	Active(3)	H	X	V	V	H		L	H	L	H
Mode Register Set	Idle	H	X	X	OP code			L	L	L	L
No-Operation	Any	H	X	X	X	X	X	L	H	H	H
Burst Stop	Active(4)	H	X	X	X	X	X	L	H	H	L
Device Deselect	Any	H	X	X	X	X	X	H	X	X	X
AutoRefresh	Idle	H	H	X	X	X	X	L	L	L	H

Table 11.2: Truth table showing how the different pins on the external RAM needs to be set in order to execute the desired command [Memory, 2011]

The pin setting for the correlating command are described in Table 11.2. It is decided that a burst length of 1 is to be used since the address decoder only reads 1 word at a time see section 10.7. When a row is activated, more than one column can be written to in the consecutive clock cycles as long as the address is provided.

A CAS Latency of 3 is chosen since it allows the SDRAM to be clocked higher (although a margin is made to ensure stability in the performance). When the mode registers has been set, the process of initialization will be complete when two auto refresh cycles has been executed. After 2 refresh cycles the SDRAM has been initialized and is ready for use. This makes an initialization time of 200 μs + 18 clock cycles as a minimum e.g. 200,135 µs.

11.2.1 VHDL Implementation

The signals is controlled from an FPGA, which give the advantages of parallel communication and speed. A key part of the following code is to ensure valid data from and to the FPGA. To change state a primary and temporary variable is used. The temporary variable is use to store the next state. The temporary variable is then transferred to the primary variable, which triggers the new state. This ensures that a change of state first happens when all the commands has been executed. This makes the code more reliable because it ensure that every step happens in the right order.

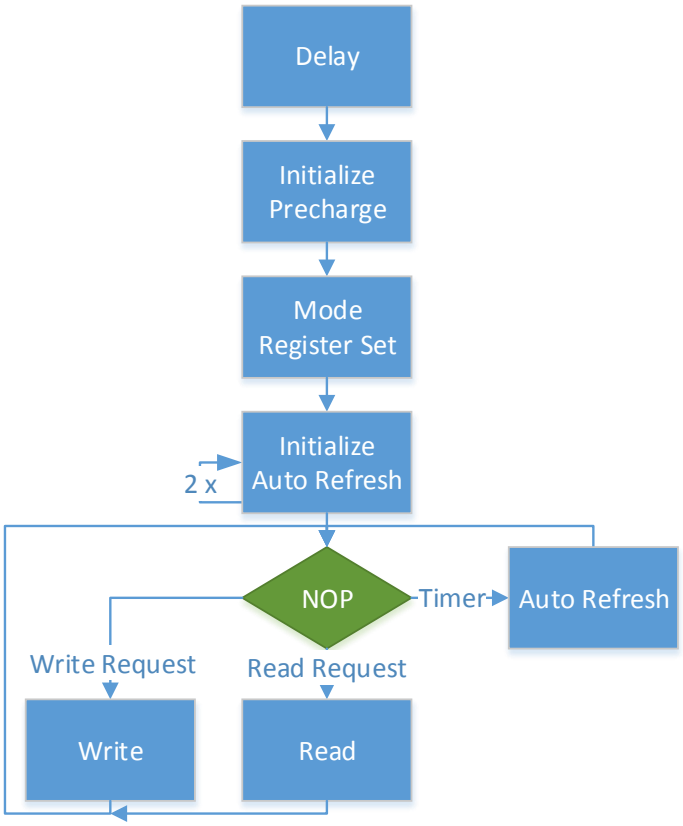
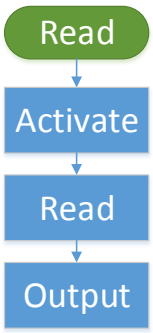
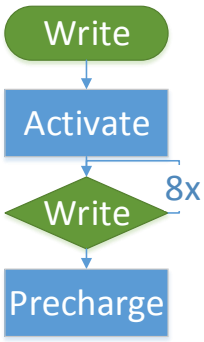


Figure 11.3: The FSM describing the SDRAM controller. The functions Read and Write are expanded in figure Figure 11.4a and 11.4b

In the following, two examples of the main controller and triple port VHDL code is given. The code is build with a state machine in mind. The basis of the state machine is built upon Figure 11.3. When performing the actual reading and writing it is handled by secondary state machines, the read and write, described on Figure 11.4a and 11.4b respectively. The primary FSM will initialize the SDRAM, setting it up and keeping the auto refresh active, if the SDRAM controller is in NOP state, as described in *Setup and Timing*. When in NOP state it will wait for either a read or write request from the triple port. When one of these request has been set, it will go into a secondary state machine described in Figure 11.4.



(a) The secondary FSM which handle reading in SDRAM



(b) The secondary FSM which handle writing in SDRAM

Figure 11.4: The read and write FSM used to read and write from the SDRAM

The secondary FSM is setting all the needed data and address pins to the required states. When the pins have been set a read or write will be performed. An incremental loop has been added to the write function, Figure 11.4b. This way eight words can be written into the SDRAM in eight cycles optimizing the write loop for speed. When a read is requested, only one word will be accessed since the address decoder only requests for 1 word see section 10.7.

The following two examples are snippets of the VHDL code used to control the SDRAM. Regarding the rest of the code, these snippets will in the following part be described in detail, giving an understanding of how the code is constructed.

Read State

In the following, the read state will be described though Listing 11.1.

```
when iRead =>
    cmd_ready <= '0';
    case cState is
        when cActivate =>
            RAM_CMD <= cmd_Activate;
            sdr_Addr <= ROW_Addr;
            sdr_BA <= BA_Addr;
            cState_r := cRead;

            Delay_cnt := 0;
            delay_to := 3;
            iState_r := iDelay;
            iStateTemp := iRead;

        when cRead =>
            RAM_CMD <= cmd_Read;
            sdr_Addr(sdr_Addr_code_pin) <= '1';
            sdr_Addr(COL_Addr'range) <= COL_Addr;
            sdr_Data <= (others => 'Z');
            cState_r := cWaitDone;
            WaitForRead <= True;
```

```
22
23              Delay_cnt := 0;
24              delay_to := 9;
25              iState_r := iDelay;
26              iStateTemp := iRead;
27
28          when cWaitDone =>
29              WaitForRead <= False;
30              cState_r := cActivate;
31              iState_r := iNop;
```

Listing 11.1: State describing the Read state of the FSM

When the state of the code is changed to read according to Figure 11.3, the code in Listing 11.1 will be executed. Next the secondary state machine in Figure 11.4a will be activated. First it will activate a desired row in a desired bank.

This is done by setting an activate command according to Table 11.2 while simultaneously setting the bank- and address pins. Then the case changes from cActivate to cRead. A delay is added for the SDRAM chip to activate the row according to [Memory, 2011]. Next a read command is set as described in Table 11.2, setting up the SDRAM for a read. This is done together with address pin 10 [4], which command the current active bank to auto precharge, making sure the bank is precharged after the read.

The address pins are set to the desired column address and the data pins are set to high impedance (input) making it possible for the SDRAM to output data on the data pins.

The WaitForRead flag is then set, signalling another process that data is incoming, and the state is then changed to cWaitDone. A delay is made ensuring that both the data has arrived and the row has been precharged see [Memory, 2011], before going to the cWaitDone state. The cWaitDone state then prepare for another read and set the primary state machine to the NOP state again.

The other process utilizes a feed back clock from the SDRAM and thereby complies with the CAS latency. This process also signals the triple port that the data is valid.

Triple port SRAM emulation

The code from Listing 11.2, describes the state where the triple port receives pixel data from the pixelgate. A FIFO is handling the interface, making sure the triple port receives 16-bit words, as described in *Design of SDRAM Controller*. Before entering the state described in Listing 11.2 the triple port checks if the FIFO has at least 8 words in its stack, if so, load them into Temp_Data and then the following happens:

```
1 when tWriteFullres =>
2     if ((cmd_ready = '1') and (make_request = '0')) then
3             doRead <= '0';
4             make_request <= '1';
5             host_addr <= Addr(21 downto 0);
```

[4] This pin is multiplexed between auto precharge and a normal address pin

```
6               host_Data_in <= Temp_Data;
7       elsif ((cmd_ready = '0') and (make_request = '1')) then
8               make_request <= '0';
9               Addr <= Addr + 8;
10              tdelay_cnt := 0;
11              tdelay_to := 20;
12              tState_r := tDelay;
13              tStateTemp := tNOP;
14      end if;
```

Listing 11.2: Triple port - state when receiving the full resolution image

When entering the Write state of the triple port, it will first check if the controller is busy. After ensuring the controller is available for a write, the doRead is cleared so a write can be performed. The desired address is loaded and the data from the FIFO is loaded, and the write request is made to the controller. This is done until the controller clears the cmd_ready and then the triple port stops requesting a write, and increments the address by 8 to prepare for new write cycle. A delay is added, making sure the controller finishes its previous action.

11.3 Reliability and test

The final process in implementing the RAM into the system is testing the RAM, making sure the controller behaves as wanted and is reading the correct data from every address. The system is being tested separately making sure the individual systems work, since the SDRAM controller needs to function properly in order for the triple port to function. The RAM is tested against the requirements described in section 11.1:

SDRAM Controller

Requirements to be fulfilled

- The external SDRAM shall be able to store both the entire image and the preview image at the same time.
- The SDRAM controller should be able to write data with 1 Gbits/s
- The SDRAM controller should be able to read data up to 400 Mbps
- The SDRAM shall be refreshed minimum 4096 times in a period of 64 ms in order to withhold the stored image data
- The SDRAM must run at a 133 Mhz clock speed, maximizing the data transfer rate
- The SDRAM should be able to store the data as long as power is supplied

Testing Procedure To check these requirements, two tests is performed.

1. To checks whether or not the SDRAM transfer data fast enough.

2. To checks that the clock and refresh command is as compliant with the requirements.

For the first test, a loop is used to input data, emulating what the triple port delivers, to all addresses of the SDRAM chip. When every address in the RAM has been written, it will revert to reading every address and compare the data with the written. Before any reading or writing is performed, a second of waiting in is done in the NOP state, making sure the SDRAM has had time to be properly initialized. The writing procedure is repeated 1000 times and the read/compare procedure is repeated 50 times.

For the second test a logic analyzer is set up to measure the clock and an FPGA pin is configured to go high, when an auto refresh is performed. Parallel to this, a long duration test is performed, where a piece of data is written to the RAM and read 1 minute later[5]. The last test it to ensure that the refreshing process of the RAM works.

Results

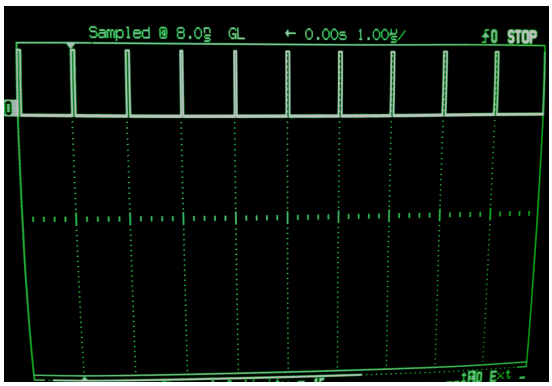
The results of the test were as follows:

Test 1:

- Writing 1000x8MB took 56.6 Sec = 1130.7 Mbits/s
- Reading 50x8MB took 3 Minutes and 54 seconds = 13.7 Mbits/s

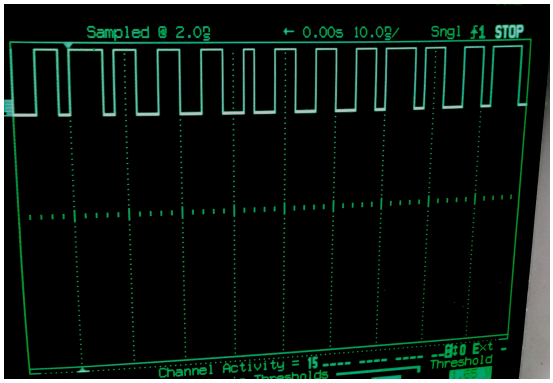
Test 2:

[5] 1 minute is deemed enough due to the short term memory of an SDRAM (64 ms [Memory, 2011] meaning that if it can store data for 1 minute it will be able to store it for longer periods as well



(a) A picture of the entry into auto refresh measured on a logic analyser

(b) A picture of the clock speed measured by a counter



(c) A picture of the clock signal measured on a logic analyser (The reason the clock is not very symmetrical might be that the logic analyser only measure 500 MSa/s and that might not be enough to give a symmetrical clock at 133.3 MHz)

Figure 11.5: Pictures of clock and refresh interval measured during test

The write, read and compare for the RAM over a period of 1 minute, showed that the RAM still had the right data stored after one minute. This test was done by writing a specific series of data to the RAM, waiting one minute and then compared the data to the original written data.

Conclusion

Concluding the test of the SDRAM controller it shows that the write speed is sufficient to receive data from the Camera FIFO in correlation with the demands written in section 11.1. Reading from the SDRAM controller was not quite as fast as expected. This was due to an implemented delay between reads, which the controller needed for reasons assumed to be in correlation with ripple or timing issues. Since the read speed is not as essential as the write speed, due to the fact that the MCU can be programmed to handle the wait which the camera can not, it is deemed sufficient and the test is therefore passed. Since all 8 MB has been written to and then read from, the requirement about storing the full image and the preview at the same time is met too. The second test is also passed since all requirements set in section 11.1 is met.

Triple Port

Requirements to be fulfilled

- The read port is to mimic SRAM
- Both the full image and the preview have separate port entrances
- The SDRAM controller should be able to write data whit 1 Gb/s
- The SDRAM controller should be able to read data up to 400 Mbits/s.

Testing Procedure

The triple port is tested by emulating the camera's write process to the FIFO and the preview's write process to the other FIFO and finally by reading the data from the read port and comparing it to the written data. This will be tested against the requirements set in section 11.1.

Results

The results of the test were:

- The camera port could write to the FIFO without filling it up and loosing data
- The read port could read the data back

Conclusion

The requirements set have not been met due to a couple of reasons, foremost only two of the three ports have been implemented, and second most that the data speed tested with was only half of that expected in a final product. This was due to some extra delays needed for the same reasons as the SDRAM controller. Also a bug was discovered but not fixed. The data shall be read consecutively if not the system will stall. Therefore is the test not passed.

Part III

Test & conclusion

12 | Acceptance test

A test of the prototype system is to be performed in order to ensure all of the requirements determined in Section 5.5: Prototype requirements are met. This is done to validate the prototype as a reliable payload for a cubesat. The procedure will be explained in Section 12.1: Test procedure and result are explained in Section 12.2: Test Results by the two following tables.

12.1 Test procedure

Req. no.	Requirements:	Test procedure	Expected output
1	It shall be possible to send requests via CAN messages for: previews, full resolution images, remote capture of an image, deletion of images and for onboard storage statistics.	This requirement is tested through the test for requirement 2 to 10.	It is possible to perform the test for requirement 2 to 10.
2	The prototype shall be able to capture an image and store it on onboard memory.	First, all files on the SD-card are deleted, and the SD-card is then placed back in the SD-card slot on the MCU development board. A CAN dongle is connected to the two CAN pins. The prototype is then powered on. The PCAN software is configured to transmit at 500 Kbps and send a single 'p'. It is verified in PCAN that 'D' is returned by the prototype. The SD card is then placed in a PC and it is checked that KOMP\IMG_0 and PREVIEW\IMG_0 exists.	The two files exist on the SD-card, and contains the counter array and encoded image data.
3	The prototype shall be able to send a compressed image from storage through to a terminal on a PC, where the data can be viewed.	A CAN dongle is connected to the two CAN pins. The prototype is then powered on. A terminal is setup to a baudrate of 115200 and the connected COM port. The PCAN software is configured to transmit at 500 Kbps and send "IMG_0".	The terminal is filled with binary data.

4	From CAN, it shall be possible to change the configuration of the payload, in terms of resolution and when images shall be captured.	A CAN dongle is connected to the two CAN pins. The prototype is then powered on. A terminal is setup to a baudrate of 115200 and the connected COM port. '5' is sent over CAN. After 20 seconds "10" is sent over CAN.	The terminal prints the word "Timelapse", first with 5 seconds interval and after the second command is executed the interval change to 10 seconds.
5	The payload shall be able to compress the image lossless such that it can be downloaded from space in less than 1 day.	The SD-card is placed in a PC, and the filesize of KOMP\IMG_0 is checked.	The filesize is below 4.3 MB see *Max Image size* below.
6	The prototype must be able to detect black frames and not store these.	First all files on the SD-card are deleted, and the SD-card is then placed back in the SD-card slot on the MCU development board. A CAN dongle is connected to the two CAN pins. The prototype is then powered on. The camera is then covered. The PCAN software is configured to transmit at 500 Kbps and send a single 'p'. The SD card is then placed in a PC and it is checked that KOMP\IMG_0 and PREVIEW\IMG_0 does not exist.	The files does not exist.
7	The prototype must be able to generate previews that can be viewed in a terminal.	A CAN dongle is connected to the two CAN pins. The prototype is then powered on. A terminal is setup to a baudrate of 115200 and the connected COM port. First a 'p' is sent to take a picture and then "kIMG_0" is sent.	The terminal is filled with binary data.
8	At least 10 previews must be able to be downloaded from space per pass.	The SD-card is placed in a PC, and the filesize of *PREVIEW\IMG_0* is checked.	The total number of bytes is below 24.2 KB see *Max preview size* below.
9	The payload must have at least 30 GB of storage space.	A CAN dongle is connected to the two CAN pins. The prototype is then powered on. A terminal is setup to a baudrate of 115200 and the connected COM port. A 's' is sent over CAN.	Output to the terminal is free space 30+ GB.

Max image size

Based on Equation 4.7, the amount of data that can be transferred in 1 day is:

$$size = \frac{\left(3\,\text{hours} \cdot 3600\,\frac{\text{s}}{\text{hour}} \cdot 9600\,\frac{\text{bit}}{\text{s}}\right) \cdot 84}{250} \tag{12.1}$$

$$size = 4{,}35\,\text{MB} \tag{12.2}$$

Max preview size From Section 1.1: AAUSAT it is known that a pass is approximately 10 min. The time to download a preview must therefore be below 1 min. In the same manor as Equation 12.1 the size of a preview can be calculated as:

$$size = \frac{\left(1\,\text{min} \cdot 60\,\frac{\text{s}}{\text{min}} \cdot 9600\,\frac{\text{bit}}{\text{s}}\right) \cdot 84}{250} \tag{12.3}$$

$$size = 24{,}2\,\text{KB} \tag{12.4}$$

12.2 Test Results

The following table shows the results of the test described in Section 12.1: Test procedure and if the requirement has been fulfilled.

Req. no.	Results	Requirements fulfilled?
1	It is possible to remotely request a *capture image*, but because the SDRAM is not fully implemented, see section 11.3, it is not possible for the data to reach the MCU. Furthermore it is not possible to request either a full resolution image or a preview due to time constraints.	No
2	The 'D' is received in PCAN as can be seen in Figure 12.1. Also the file KOMP\IMG_0 exist and the data in it can be seen on Figure 12.3. But as expected the file PREVIEW\IMG_0 does not exist due to the fact that the preview functionality has not been implemented. Also the compressed file does not contain image data but only the counter array. This is due to the fact that the triple port is not fully implemented Section 11.3: Triple Port.	No
3	The terminal prints a large string of binary data as ascii values as can be seen in Figure 12.2, when compared to Figure 12.3, it can be seen that the data is identical except for values not contained in standard ascii.	Yes
4	From the test section 8.6 *Testing CAN message handling & timelapse* it can be seen that the timelapse functionality can be reconfigured via CAN. To change the resolution via CAN, has not been implemented due to time constraints.	No
5	From section 8.6 *Data compression/encoder* it can be seen that the compression algorithm, inflates the full image. Because of this the 5 MB image ends up being 5.6 MB and is therefore 29 % larger than then calculated max size at 4.35 MB, see Equation 12.1.	No
6	This functionality has not been implemented due to time constraints.	No
7	This functionality has not been implemented due to time constraints.	No

8	The Preview functionality has not been implemented due to time constraints. When designing the preview it was intended to scale the image 32 times. This would make it possible to send 10 previews, since a produced preview would have the size of 4.86 KB, see section 10.5. This is lower than the calculated max preview size at 24.3 KB, see Equation 12.4.	No
9	The terminal prints a 31.154.400 KB out on the screen, as can be seen on Figure 12.4. This value is higher than the requirement of 30+ GB. the requirement is therefore fulfilled and the test is a success.	Yes

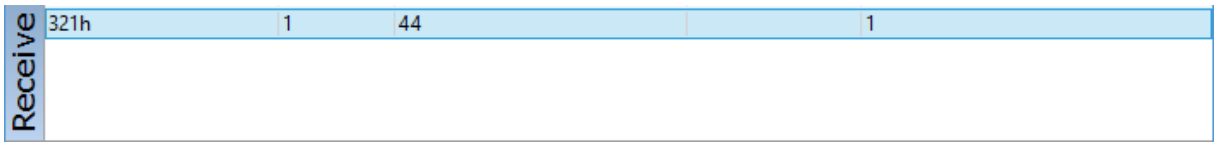


Figure 12.1: PCAN receiving 0x44 or 'D' from the prototype

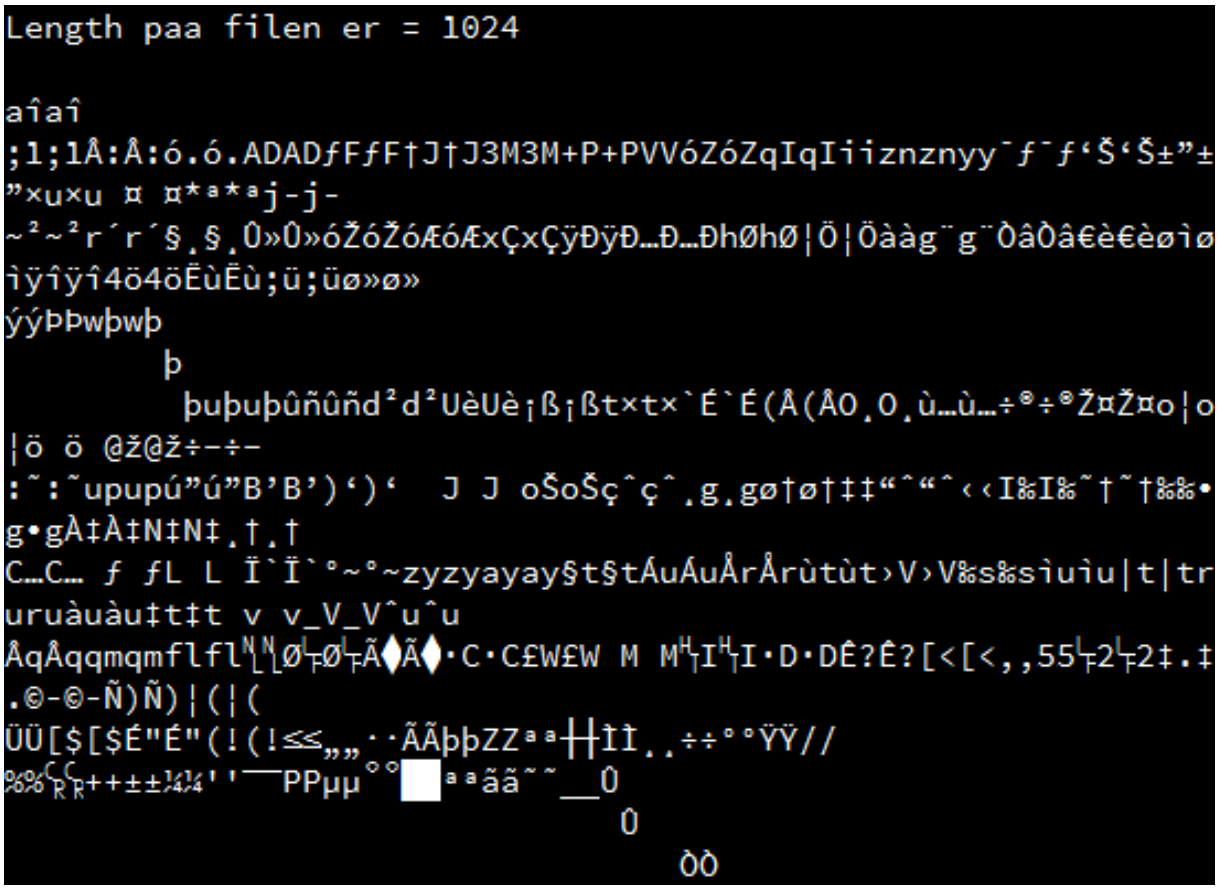


Figure 12.2: The terminal output of IMG_0 file

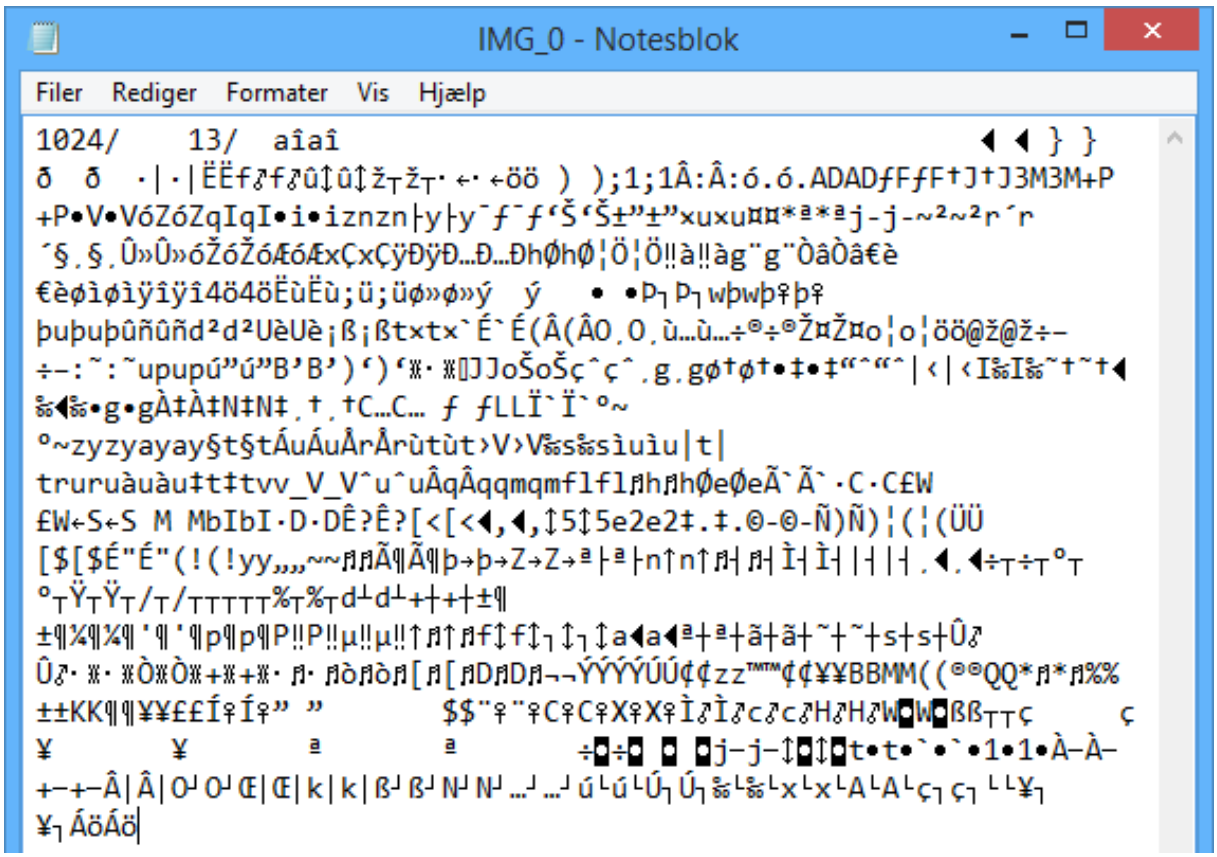


Figure 12.3: The data in IMG_0 file viewed with notepad

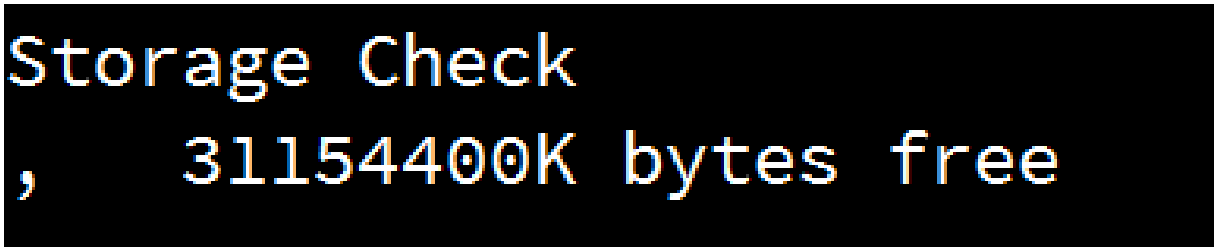


Figure 12.4: The output of the storage check

The prototype has been tested in correlation with the requirements set for the prototype. 2 out of 9 requirements were met during the test.

A picture of the complete prototype can be seen on Figure 12.5, showing the MCU, FPGA, SDRAM and Pi camera connect with their respective breakout board.

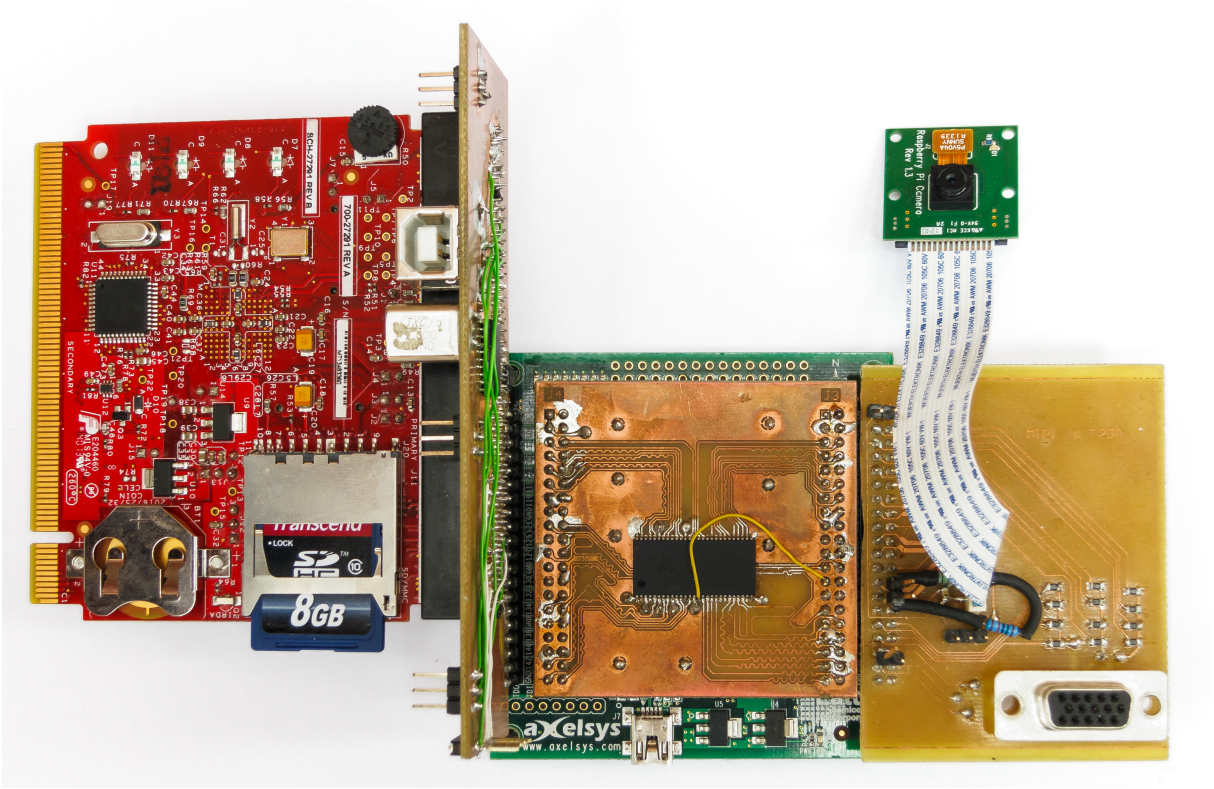


Figure 12.5: The entire prototype system

13 | Conclusion

In this project, a camera payload for a cubesat has been designed, and a prototype has been developed.

The theory needed to understand the workings of a camera has been investigated, as well as image compression methods. The communication interfaces ranging from CAN and CSP to I²C and the MIPI protocol has been described.

Based on this analysis, requirements for both the finished payload and the prototype have been made. Form the requirements, functionalities of the system could be established, and from these the data interfaces and electrical interfaces could be made.

A Raspberry Pi camera has been chosen as the camera unit in the system, however the registers needed to be set in the camera to begin its capture had to be found by reverse engineering the camera.

The designed system is able to receive messages via CAN, capture an image, compress it and store it on an SD card along with small resolution preview of the image. Depending on the message sent via CAN, the system can also display the files on the SD card, or display the data of an image in a terminal. A timelapse feature has also been made, capturing an image with a changeable interval.

To be able to download the image from the satellite quicker, a Huffman encoding algorithm is used, however on a 5 Mpx image, the compressed image takes up about about 13 % more storage space than the original image. The reason for this is not known.

An FPGA is used to receive data from the MIPI bus on the camera, and store it in SDRAM, so it can be fetched by the MCU. The FPGA is also generating the counting array that the Huffman algorithm needs, as well as the preview. It is also acting as the interface between the MCU and the RAM, so the image data can be read directly by the MCU, as if it was stored in internal memory. This is done by having a Flexbus interface as well as an SDRAM controller.

It was not possible to make the SDRAM controller in the FPGA fully functional, and thus the image could not be stored and fetched from the RAM. Nor has the preview generator been implemented fully. The FPGA also contains a MIPI bridge that is responsible for converting the MIPI data to 8-bit pixel data.

In the acceptance test, it is shown that the prototype does not live up to most of the requirements, the primary reason being that the SDRAM controller is not working. However, the CAN interface and associated handling functions are working. As is the compression algorithm, but for small images only. Also, when requested over CAN, the Raspberry Pi camera captures an image, and a counting array made which is stored on the SD card. It is thereby shown that the overall dataflow of the system works as intended, despite a non-functioning RAM controller.

14 | Discussion

In the following chapter, the developed prototype will be discussed, in regards to the improvements that can be made. The improvements are both those needed to get the prototype to work as intended, but also what should be done to get it in a state that it can be implemented in a satellite.

Instead of having a task running checking if data has been received over CAN, this could be implemented as a hardware interrupt, since this will only be triggered when necessary. This would increase performance since the scheduler would then have one less task to handle, and it would not consume CPU time. In fact, it could render the RTOS obsolete, since all others tasks in principle can run in series, for instance by using a state machine.

To make the system more redundant, it would be an idea to have a backup to fall back on, should the SD card fail. This could be done by having a small flash storage on board, where a single image can be saved. Likewise, the flash storage could be used to save the preview. The SDRAM can be seen as having a built in redundancy, since it has four banks that work independently of each other.

A redesign of the RAM board would be preferred where the design shall compensate for potential inductive/capacitive parasites. As shown on Figure 11.1b there is risk of a too large ringing with the current layout. Fixing this would ensure more stable communication with the RAM granting the triple port greater speeds. A configuration of the FPGA board output pins, making the timing of every pin consistent should also have a positive affect of the triple port speed, since the speed can be increased when the timing is more accurate. The triple port should also be fully implemented, so that the RAM can be accessed by the MCU, the preview generator and the full resolution image.

Also, the *delete image* function should be improved, so that after deletion, it is possible to store more than one picture without overwriting existing images. One way to solve this problem is when the file counting in a directory is complete, instead of just creating the new file, a search on the new file name should be made before creating the new file. If the file exist, file count integer, should be increased until the counter exceeds the highest picture value. By implementing this, the new file should not overwrite any existing files, regardless of whether any files has been deleted on the SD card.

When requesting the LS function it could be a good idea to implement a date and time to each picture, so it is possible to chose pictures from a specific date instead of just knowing their names. A time and date stamp function on the MCU was not implemented with the current design, but would greatly increase overview of what is stored on the SD-card. The date and time could also be used to name files.

Since the compression algorithm compresses chunks at a time, stuff bits are appended to the end of each chunk. The decompression algorithm can be made so it can facilitate this, but to reduce the size of the compressed image, it would be better to only add stuff bits to the final

byte in the image.

The preview functionality has not been fully implemented, which it should be. It could be possible to make the preview size configurable, by sending a desired scaling factor with the configuration to the preview.

Also, the configurations sent to the various modules should be implemented fully, thus allowing different camera setups, configuring the pixel gate with how many frames it should ignore, and the preview scaling factor.

CSP is also to be implemented in the finished project, so the system can communicate with other satellite subsystems, and the ground station. However, since CAN has been implemented and tested, this should be doable as CSP is a communication layer added on top of CAN.

The size of the system should also be reduced, so it can fit into a cubesat. Power pins to the different hardware elements of the system should also be made, so they can be turned on and off.

The black frame detector should be implemented, so that images, not containing data of any significance, are not stored. This is most easily done in the MCU. Making it configurable would be ideal, so the threshold can be changed based on what the images actually look like.

It would also be a good idea to implement and check up on return values, to see if functions return success, and handle it if they do not.

Compressing the image is a vital part of the system, since it allows for a faster download time to the ground. Therefore, getting the compression algorithm to work for a full 5 Mpx image is important. It is seen that the error is that the signatures generated are too long, see Figure 8.21, when applied to a big image, but the reason for this is unknown. Also, the compression algorithm should be changed so it does not use malloc/realloc, since this can cause allocations errors and memory leaks when done "on-the fly".

For use in a space environment, it could be an idea to not use an FPGA at all, but instead having the system in one single chip, by having less hardware capable of failing. Another camera solution could also be used that does not use MIPI (for instance having a parallel bus with the pixel data, thus making the interfacing simpler.

If these improvements are implemented, it is estimated that the system will function as desired, and may be ready for implementation in a satellite, although a lot of work and testing is to be done before a system ready for flight is obtained.

Bibliography

AAU Student Space (2015). AAU Satlab. http://www.space.aau.dk. Downloaded Feburary 20 2015.

Alleysson, D., Süsstrunk, S., og Hérault, J. (2002). Color demosaicing by estimating luminance and opponent chromatic signals in the fourier domain. In *Color and Imaging Conference*, volume 2002, pages 331–336. Society for Imaging Science and Technology.

Association, S. (2014). Sd standard overview - speed class. https://www.sdcard.org/developers/overview/speed_class/. Downloaded March 23 2015.

Atkins, B. (2004). Raw, jpeg and tiff. http://photo.net/learn/raw/. Downloaded March 7 2015.

Bilsen, E. V. (2015). Aic - color conversion. http://www.bilsen.com/aic/colorconversion.shtml. Downloaded Feburary 24 2015.

Bishop, P. (2009). A tradeoff between microcontroller, dsp, fpga and asic technologies. http://www.eetimes.com/document.asp?doc_id=1275272. Downloaded March 13 2015.

Bosch (1991). Can specification. http://esd.cs.ucr.edu/webres/can20.pdf. Downloaded Feburary 17 2015.

Brown, B. (2012). Text compression with huffman coding. https://www.youtube.com/watch?v=ZdooBTdW5bM. Downloaded March 7 2015.

Corporation, L. S. (2014a). Lcmxo2. http://www.latticestore.com/products/tabid/417/categoryid/9/productid/519/searchid/1/default.aspx?searchvalue=lcmxo2-7000*. Downloaded March 16 2015.

Corporation, L. S. (2014b). Product selector guide. http://www.latticesemi.com/~/media/Documents/ProductBrochures/NZ/ProductSelectorGuide.PDF. Downloaded March 16 2015.

Defossez, M. (2014). D-phy solutions. http://www.xilinx.com/support/documentation/application_notes/xapp894-d-phy-solutions.pdf. Downloaded March 24 2015.

Dempsey, T. (2015). Compare digital camera sensor sizes: full frame 35mm, aps-c, 4/3, 1 inch-type. http://photoseek.com/2013/compare-digital-camera-sensor-sizes-full-frame-35mm-aps-c-micro-four-thirds-1-inch-type/. Downloaded Feburary 18 2015.

Enginering, I. (2015). How does the jpeg compression work? http://www.image-engineering.de/index.php?option=com_content&view=article&id=522. Downloaded Feburary 23 2015.

FOUNDATION, R. P. (2015). Raspberry pi camera datasheet. http://www.raspberrypi.org/documentation/hardware/camera.md. Downloaded Feburary 15 2015.

Freescale (2012a). K10 sub-family reference manual. http://cache.freescale.com/files/32bit/doc/ref_manual/K10P100M100SF2V2RM.pdf. Downloaded may 23 2015.

Freescale (2012b). K60 sub-family reference manual. http://cache.freescale.com/files/32bit/doc/ref_manual/K60P144M100SF2V2RM.pdf. Downloaded march 31 2015.

Freescale (2013). K10 sub-family data sheet. http://cache.freescale.com/files/32bit/doc/data_sheet/K10P100M100SF2V2.pdf. Downloaded may 23 2015.

Freescale, M. B. (2014). Sd-card fat file system. https://community.freescale.com/docs/DOC-100913. Downloaded May 15 2015.

Freescale Semiconductor, I. (2000). Can bosch controller area network (can) version 2.0. http://www.freescale.com/files/microcontrollers/doc/data_sheet/BCANPSV2.pdf. Downloaded Feburary 17 2015.

GomSpace (2014). Client and server example. https://github.com/GomSpace/libcsp/blob/master/doc/example.md. Downloaded Feburary 27 2015.

Gomspace (2015). Cubesat space protocol (csp). http://gomspace.com/documents/GS-CSP-1.1.pdf. Downloaded Feburary 27 2015.

Harris, T. (2015). How file compression works. http://computer.howstuffworks.com/file-compression3.htm. Downloaded Feburary 27 2015.

Howstuffworks (2015). What is the difference between ccd and cmos image sensors in digital camera? http://electronics.howstuffworks.com/cameras-photography/digital/question362.htm. Downloaded Feburary 20 2015.

I2C-bus (2015). I2c bus addressing. http://www.i2c-bus.org/addressing/. Downloaded Feburary 24 2015.

Impulseadventure (2008). Jpeg compression quality from quantization tables. http://www.impulseadventure.com/photo/jpeg-quantization.html. Downloaded February 13 2015.

Instruments, T. (2008). Introduction to the controller area network (can). http://www.ti.com/lit/an/sloa101a/sloa101a.pdf. Downloaded Feburary 17 2015.

Knott, R. (2011). The fibonacci numbers. http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibtable.html. Downloaded May 5 2015.

latticestore (2014). lcmxo2-7000. http://www.latticestore.com/products/tabid/417/categoryid/9/productid/519/searchid/1/default.aspx?searchvalue=lcmxo2-7000*. Downloaded March 10 2015.

Lukac, R. (2008). *Single-sensor imaging: Methods and applications for digital cameras*. CRC Press.

Memory, A. (2011). As4c4m16s 4m x 16 bit synchronous dram. http://www.alliancememory.com/pdf/dram/64m-as4c4m16s.pdf. Downloaded February 15 2015.

Microchip (2002). A can physical layer discussion. http://ww1.microchip.com/downloads/en/AppNotes/00228a.pdf. Downloaded Feburary 17 2015.

Moeslund, T. B. (2012). *Introduction to video and image processing*. SPringer, 1. edition. ISBN: 978-1-4471-2502-0.

Motorola (1999). Overview of can. https://www.mi.fu-berlin.de/inf/groups/ag-tech/projects/Z_Finished_Projects/ScatterWeb/moduleComponents/CanBus_canover.pdf?1346661370. Downloaded Feburary 17 2015.

Omnivision (2009). Ov5647 datasheet. http://www.seeedstudio.com/wiki/images/3/3c/Ov5647_full.pdf. Downloaded February 13 2015.

Oppenheim, A. og Schafer, R. (2010). *Discrete-time Signal Processing*. Pearson.

Rani, K. S. og Hans, W. J. (2013). Fpga implementation of bilinear interpolation algorithm for cfa demosaicing. In *Communications and Signal Processing (ICCSP), 2013 International Conference on*, pages 857–863. IEEE.

Rouse, M. (2005). Lossless and lossy compression. http://whatis.techtarget.com/definition/lossless-and-lossy-compression. Downloaded Feburary 27 2015.

Saederup, R. G. (2015). Aausat5. https://www.flickr.com/photos/118193674@N04/sets/72157648831307898/. Downloaded Feburary 24 2015.

semiconductor, F. (2010). Beyond bits: Next-generation microcontrollers. http://cache.freescale.com/files/microcontrollers/doc/brochure/BR8BITBYNDBITS5.pdf. Downloaded may 23 2015.

semiconductor, L. (2012). Mipi csi2-to-cmos parallel sensor bridge. http://www.latticesemi.com/view_document?document_id=50533. Downloaded may 19 2015.

Semiconductor, L. (2015). Machxo2 family datasheet. http://www.latticesemi.com/~/media/LatticeSemi/Documents/DataSheets/MachXO23/MachXO2FamilyDataSheet.pdf. Downloaded may 23 2015.

Space, A. S. (2013a). AAUSAT3. http://www.space.aau.dk/aausat3/index.php?n=Ham.HAMAndRadioInfo. Downloaded Feburary 27 2015.

Space, A. S. (2013b). AAUSAT3 passing Aalborg. http://www.space.aau.dk/aausat3/index.php?n=Main.PassingAalborg. Downloaded Feburary 27 2015.

Space, A. S. (2013c). Spacelink format. http://www.space.aau.dk/aausat3/index.php?n=Ham.HAMAndRadioInfo. Downloaded Feburary 27 2015.

Space, A. S. (2014). Critical items list. Technical report, Aalborg university. Downloaded March 18 2015.

Space, A. S. (2015a). Aausat5 - system specification document. Annex.

Space, A. S. (2015b). Aausat5 - user manual. Annex.

Sparkfun (2015a). I2c. https://learn.sparkfun.com/tutorials/i2c. Downloaded Feburary 20 2015.

Sparkfun (2015b). I2c basic timings. https://dlnmh9ip6v2uc.cloudfront.net/assets/6/4/7/1/e/51ae0000ce395f645d000000.png. Downloaded Feburary 20 2015.

team, A. (2015). Reduced functional test, rev 1.3. not published. Downloaded may 23 2015.

Techradar. (2009). All you need to know about jpeg compression. http://www.techradar.com/news/computing/all-you-need-to-know-about-jpeg-compression-586268/2. Downloaded Feburary 23 2015.

Waltham, N. (2015). Ccd and cmos sensors. http://download.springer.com/static/pdf/412/chp%253A10.1007%252F978-1-4614-7804-1_23.pdf?auth66=1424701194_3ffa52b17811c9e7fbdbbac3eb33be44&ext=.pdf. Downloaded Feburary 23 2015.

WhyDoMath. (2011). Image compression: How math led to the jpeg2000 standard. http://www.whydomath.org/node/wavlets/basicjpg.html. Downloaded Feburary 23 2015.

WhyDoMath (2015). Image compression: how math led to the jpeg2000 standard. http://www.whydomath.org/node/wavlets/imagecompression.html. Downloaded Feburary 27 2015.

Wikimedia (2015). Color wheel. http://upload.wikimedia.org/wikipedia/commons/thumb/3/38/BYR_color_wheel.svg/2000px-BYR_color_wheel.svg.png.

Wikipedia (2014). Cubesat space protocol. http://en.wikipedia.org/wiki/Cubesat_Space_Protocol. Downloaded Feburary 27 2015.

Wikipedia (2015). Can bus. http://en.wikipedia.org/wiki/CAN_bus. Downloaded Feburary 17 2015.

Xilinx (2011). Extended spartan-3a family overview. http://www.xilinx.com/support/documentation/data_sheets/ds706.pdf. Downloaded may 23 2015.

Appendix

A | Fibonacci proof using induction

The Fibonacci sequence is defined as:

$$F_i = F_{i-1} + F_{i-2} \qquad \text{for} \quad i \geq 2 \tag{14.1}$$
$$F_1 = 1, F_2 = 1 \tag{14.2}$$

We want to prove that:

$$\sum_{i=1}^{n} F_i = F_{n+2} - 1 \tag{14.3}$$

Basis step: We want to show that our statement holds for the smallest i, namely i = 2:

$$F_1 + F_2 \stackrel{?}{=} F_{2+2} - 1 = F_4 - 1 \tag{14.4}$$
$$F_1 + F_2 \stackrel{?}{=} (F_3 + F_2) - 1 \tag{14.5}$$
$$F_1 + F_2 \stackrel{?}{=} ((F_2 + F_2) + F_2) - 1 \tag{14.6}$$
$$F_1 + F_2 \stackrel{?}{=} 1 + 1 + 1 - 1 \tag{14.7}$$
$$2 = 2 \tag{14.8}$$

Induction step: Now that we have shown that the statement holds for n, we want to show that the statement holds for $n + 1$, by replacing n with $n + 1$:

$$\sum_{i=1}^{n+1} F_i \stackrel{?}{=} F_{(n+1)+2} - 1 \tag{14.9}$$
$$F_1 + F_2 + F_3 + ... + F_n + F_{n+1} \stackrel{?}{=} F_{n+3} - 1 \tag{14.10}$$
$$F_1 + F_2 + F_3 + ... + F_n + F_{n+1} \stackrel{?}{=} F_{n+2} + F_{n+1} - 1 \tag{14.11}$$
$$\left(\sum_{i=1}^{n} F_i\right) + F_{n+1} \stackrel{?}{=} F_{n+2} + F_{n+1} - 1 \tag{14.12}$$

Now, we can use what we have proven in the basis step, to reduce this:

$$\left(\sum_{i=1}^{n} F_i\right) + F_{n+1} \stackrel{?}{=} F_{n+2} + F_{n+1} - 1 \tag{14.13}$$
$$F_{n+1} = F_{n+1} \tag{14.14}$$

Q.E.D

Since this is true, the original statement is also true, and the proof is complete.