CHALMERS

Design of a Plug'n'Play analysis interface for the communication protocol Flexray

Master of Science Thesis

Henning Colliander
Johan Säwing

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, September 2010

Design of a Plug'n'Play analysis interface for the communication protocol Flexray

Henning Colliander,
Johan Säwing

Examiner: Lars Bengtsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden September 2010

Abstract

This thesis is a report of our evaluation of the possibility to create a prototype of a Plug'n'Play hardware tool for analysing a Flexray Communications Systems network. The design of this tool was made with the intent of using an FPGA for decoding transmission and an ARM-processor for transmitting data, creating a basis for development. We design the prototype to only capture data sent on the network. The theory was that this would form a basis for further analysis. Due to hardware malfunction in the later parts of the project, no actual test of this theory could be done. Our results suggest that only using an FPGA for analysis and transmitting data is enough and additional analysis could be done on a different tool or using software on a larger system, such as a PC.

Table of contents

List of abbreviations

BSS	–	Byte Starting Sequence
CAS	–	Collision Avoidance Symbol
FES	–	Frame Ending Sequence
FPGA	–	Field Programmable Gate Array
FSS	–	Frame Start Sequence
MMU	–	Memory Management Unit
MTS	–	Media Access Test Symbol
RTOS	–	Real Time Operating System
TSS	–	Transmission Start Sequence
VHDL	–	VHSIC Hardware Description Language

List of figures

List of tables

1 Introduction

The purpose of this project is to create a prototype of a Plug'n'Play hardware tool for analysing communication on a Flexray Communications System. This hardware should consist of an FPGA for capturing data from the network and a ARM-based processor for relaying this information to a PC over Ethernet using UDP.

The project should evaluate the possibility of creating a prototype and try to establish the requirements of the hardware tool. This idea is based on the similar product for listening to MOST produced by Broccoli Engineering AB, who requested this project. The purpose of the prototype is to develop it into a commercial product. At the end of this project there will be a well documented and easy to use prototype.

During this project the following issues should be addressed.
Is it possible to receive data from the Flexray network without knowledge of the systems configuration?
Which are the demands of the product in terms of components, power and size?

2 Theory

2.1 Flexray

2.1.1 The Flexray Communication System

“The Flexray Communications System is a robust, scalable, deterministic, and fault-tolerant digital serial bus system designed for use in automotive applications. It was developed by the Flexray Consortium, a cooperation of leading companies in the automotive industry, from the year 2000 to the year 2009. The Flexray Consortium has concluded its work with the finalization of the Flexray Communications System Specifications Version 3.0. “[1]

The Flexray Communications System was created as an addition to current communications systems in automotive applications as these reaches their limits in speed and bandwidth. A full understanding of the Flexray Communications System is not needed within the scope of this project but a few facts have to be presented.

2.1.2 Topology

The Flexray Communications System communicates through two communication lines, named channel A and channel B in the specification. These channels can be used as either two separate communication lines or together for redundancy. Flexray allows for two basic network topologies that can be combined in several different ways. The two basic topologies are bus and star configuration and are shown in Figure 1 and Figure 2. As shown in Figure 1 a node can be connected to one channel in the network without necessarily be connected to the other channel. Each channel can be of a different topology, if one channel is a bus line the other can be an active star configuration. In combinations up to two active stars can be used to drive communications between different bus lines. In the scope of this project the actual benefits of different topologies are of no consequence, but since each topology needs a different configuration of the Flexray Communications System the existence of variations are important. For example each active star in the path between two different nodes will create a delay at the beginning of a transmission, and this has to be compensated by adjusting a parameter in the protocol, otherwise the transmission start might not be detected.

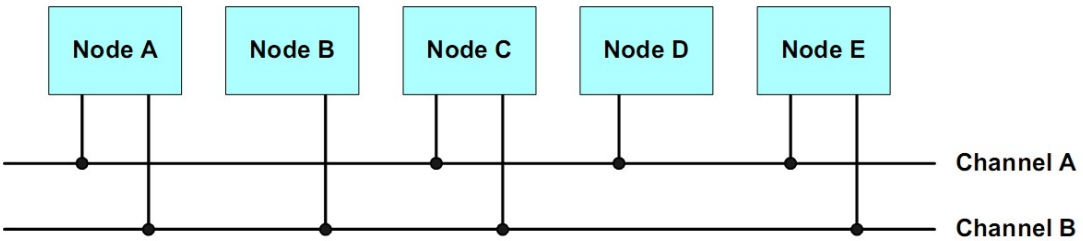


Figure 1: Flexray system with bus configuration.

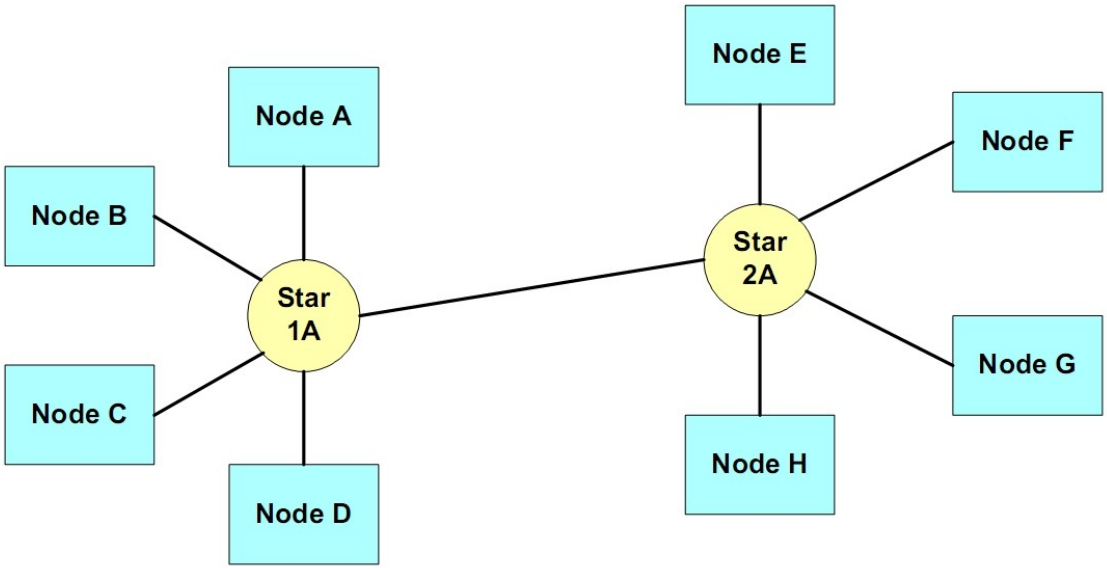


Figure 2: Flexray system with star configuration.

2.1.3 Time

There are four different ways of measuring time, the smallest is the microtick. A microtick has a duration set by the oscillator of the local device and is thus defined locally and not shared over the Flexray system. In this project it is defined as the same length of time as the sample clock, 1 µs. The second largest measurement is the macrotick, this is defined as a length of time and is the basic time unit of the Flexray network. Each node has a definition of macrotick as a number of microticks, based on the length of a microtick in that node. The communication slots are the basic measurement of time in the communication phases and they are defined as a number of macroticks.

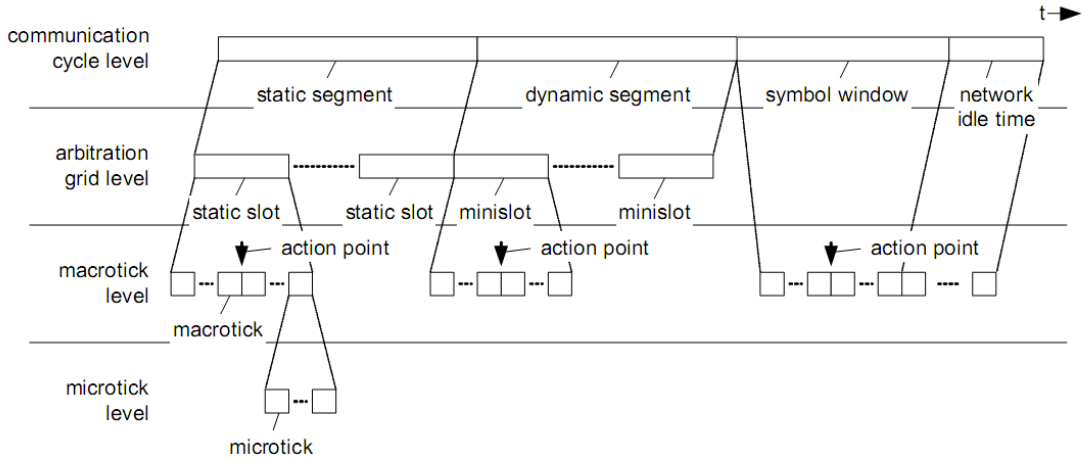


Figure 3: Timing hierarchy within the communication cycle. [2]

2.1.4 Communication phases

The communication in the Flexray Communications System is divided into four phases. The static segment, the dynamic segment, symbol window and the network idle time. Together they form one cycle of communication.

In the static segment a Time Division Multiple Access (TDMA) scheme is used to decide which node has access to the Flexray bus. The segment is divided into transmission slots and nodes that needs to transmit during this segment are allocated one or more slots. Each channel can have different slot assignment. Communication in this segment is done in frames which will be discussed more in detail in the next section.

The dynamic segment is divided into minislots, that are defined as a length of time. Each transmission is done in a dynamic slot composed of one or more minislots depending on the size of the transmission. If no transmission occurs during a dynamic slot it still uses one minislot. Since there is a predefined number of minislots, the number of dynamic slots may differ each cycle depending on how much data that is transmitted. Data is transmitted as frames in this phase. In addition to the ordinary frame a trailing sequence is added to increase the size of a transmission to fit an entire minislot if needed.

The symbol window is the phase were symbols can be sent and received. A symbol is a short bit pattern that is used by the Flexray system for synchronization and initialization. In the Flexray specification only three symbols exist, the Collision Avoidance Symbol (CAS) and the Media Access Test Symbol (MTS) which have the same pattern, and the Wakeup Symbol (WUS). Since this project is assuming connection to an already running system only the MTS symbol is considered.

The network idle time is a phase were no communication occours. This is used for synchronization as it can be considered a safe time where no data can be lost.

2.1.5 Frames

A frame transmitted in the static or dynamic segment has a header segment, payload segment and a trailer segment. In the dynamic part it is followed by the dynamic trailing sequence. As the actual information in the frame is not used in this project a brief description will suffice. A frame is started by a Transmission Start Sequence (TSS) to announce the arrival of a frame start, this is a predefined number of low bits to allow for late transmission start detection. After this a Frame Start Sequence (FSS) is transmitted, a high bit, followed by the first byte of the frame. Each byte is preceded by a Byte Start Sequence (BSS) which is one high bit followed by one low bit. The last byte is followed by the Frame End Sequence (FES), one low bit followed by one high bit. In the dynamic segment the dynamic trailing sequence follows the end of a frame, this is a length of low bits to extend the transmission until a full minislot is used.

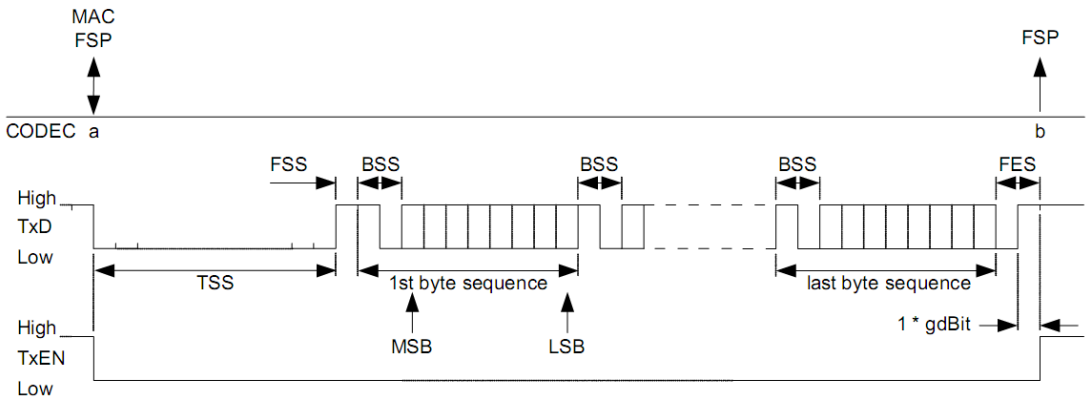


Figure 4: Frame encoding in the static segment. [2]

2.1.6 Synchronization

The Flexray Communications System is synchronized by a combination of four different synchronization mechanisms.

The first is a requirement that no clock used for communication may drift more than 0.15% from a reference clock, this means the maximum difference between two clocks in the system may only be 0.3%.

The second synchronization method is configuration of parameters in the system, this means every node in the network should know when to expect transmissions and when they are allowed to transmit.
The third synchronization method is bitstrobing, this means that the bitstrobing process synchronize the strobing of a bit to the falling edge of an incoming bit, that is synchronizes the bit detection to the bits transmitted on the Flexray bus.
The fourth synchronization method is the synchronization of cycles and their sizes. This is done by adapting the number of microticks per cycle and shifting the start of a cycle in the network idle phase.

2.1.7 Parameters

The Flexray Communications System contains about 50 different parameters that need to be configured at the construction of the system. These need to be known to safely transmit and receive data on the Flexray bus, but there is no need to understand everyone of these for the completion of this project. Therefore only a handful parameters are important to the understanding of the project and will be presented.

Table 1: Flexray parameter prefix.

c	Protocol constant
g	Global parameter, same value in all nodes
p	Node parameter
d	Duration, is combined with any other prefix.

Table 2: Flexray constants.

Parameter Name	Description	Value
cChannelIdleDelimiter	This is the number of high bits that marks the start of the transition from transmission on the bus to channel idle.	11gdBit
cClockDeviationMax	This is the maximum clock frequency deviation accepted in the network.	0.0015
cdCASRxLowMin	This is the minimum number of low bits preceding a CAS/MTS symbol.	29 gdBit
gdCASRxLowMax	This is the maximum number of low bits preceding a CAS/MTS symbol. This value can be between 67 and 99, but 99 is used in this project.	99 gdBit
gdTSSTransmitter	This is the number of low bit in the TSS. Can be as low as 3 but this project uses 15.	15
cPayloadLenghtMax	This is the maximum number of two byte words in the payload of a frame.	127
cSamplesPerBit	This is the number of samples that must be taken of each bit received from the Flexray bus.	8
cStrobeOffset	This means the fifth sample of a bit is strobed by the bitstrobing process.	5
cVotingDelay	This is the delay of the bit value received caused by the sampling algorithm.	2
cVotingSamples	The number of samples that makes up the voting window of the sampling algorithm.	5
gdBit	This is the nominal length of a bit on the Flexray bus. This value is based on the sample clock and pSamplesPerMicrotick.	1 µs
gdBitMin	The minimal bit length due cClockDeviationMax.	0.9995 µs
gdBitMax	The maximum bit length due to clock deviation.	1.0015 µs
gdSampleClockPeriod	This is the clock period of the sample clock, given by the speed of the network, 10 Mbps and the samples per bit (8) => 80 MHz sample clock => 0.125 µs clock period.	0.125 µs
pdMicrotick	In this case the microtick is the same as the sample clock, but it is allowed to have as microtick as one, two or four times as slow as the sample clock.	0.125 µs
pSamplesPerMicrotick	This could be one, two or four if the sample clock is faster than the microtick clock.	1

2.2 FPGA

Field-programmable gate array, FPGA, is an integrated circuit which is programmable by using hardware description language, HDL.

For this project an FPGA is used as development hardware because it is reprogrammable and can handle two parallel communication channels. This is needed since the Flexray network has two channels and there will be two transceivers connected to the FPGA. An other device to consider is the CPLD, but it has fewer logic gates then an FPGA and this limits the development choices. Even when the eventual product will be constructed an FPGA would be used rather than an ASIC, since product volume is assumed to be small.

In order to be able to keep up with a speed of the Flexray system of 10Mbps the FPGA need to have at least a clock frequency of 80 MHz.

2.3 ARM

2.3.1 Microprocessor

A microprocessor is a single integrated circuit containing most functions of a CPU. Microprocessors are mostly used in embedded devices and as controllers in computer peripherals as hard drives and printers.
The need for a microprocessor in this project comes from the possibilities for expanding the project and adding more analysis beyond the scope of the FPGA. By using the microprocessor from the beginning the additional work of adding more hardware and reprogramming the FPGA can be avoided.

2.3.2 ARM

ARM is an architecture for 32-bit embedded microprocessors, used in a wide range of different microcontrollers with different applications. As ARM processors are used in most 32-bit mobile embedded solutions moving from one microcontroller to another will be easier since there will be no need for changing architecture. The microcontroller chosen for this project will be part of a proof of concept and another processor might be chosen, for the next version, based on the now known requirements. One of the requirements of this project is that the microcontroller should be able to run Linux, this demands a microcontroller containing a Memory Management Unit (MMU).

2.3.3 Linux

Linux is an open source operating system based on the Unix operating system. The Linux OS is used on a wide range of different platforms, among those are various embedded platforms with ARM microprocessors. Using Linux as the OS it is possible to write code which can easily be adapted for other Linux installations on other platforms.

2.3.4 Alternatives

2.3.4.1 No microcontroller

The project could be constructed with only an FPGA implementing an Ethernet controller. The reason this was not chosen is that a microcontroller provides a good foundation for implementing communication with the outside world on a higher level than could be expected of an FPGA implementation.

2.3.4.2 MMU-less microcontroller

An microcontroller lacking MMU can't run Linux as it would normally run, this would take away some of the advantages of using an easily ported OS. The other options are using a Real Time Operating System (RTOS) or writing assembler. These options add to the complexity of creating the code and to the complexity of adding new code and programs to the device.

2.4 Definition of indata types

To analyses the contents of the Flexray communication, different types of data has to be identified. These definitions are used in the project as these types of data are of most interest for the analysis.

2.4.1 Symbol

A Symbol is a single high bit after a given number of low bits. Time of symbol recognition point should be recorded but no data should be attached. Symbol is recognized at the first high bit within the CAS.

2.4.2 Whole Frame

A whole frame is a frame that can decode from FSS to FES. Time of frame recognition point should be recorded and all bits within should be treated as part of bytes and the BSS part should be removed in the decoding.

2.4.3 Broken Frame

A broken frame has a correct FSS and is decoded in the same way but do not contain an FES, instead there is some kind of fault. As a fault is found the already decoded part of the frame is kept but all bits after this point, unless channel is idle, are decoded as raw data. The following faults should be detected, failure to decoded a BSS or FES, channel idle without FES, or a too long frame. When BSS or FES decoding fails or a frame is to long a raw data package should follow. When early channel idle is detected no raw data package should follow.

2.4.4 Raw Data

Raw data are data not recognized as any other type. Data of this type will have a timestamp at the raw data recognition point and simply contain any data found until a channel becomes idle. The channel idle detection bits, the eleven bits needed to recognize a channel idle, should not be saved but dropped. Data before the raw data recognition point should not be part of this package. The recognition point should be the first bit after an error has occurred. If a raw data packages becomes to large another raw data package should follow.

2.4.5 Examples of data combinations

2.4.5.1 Raw data only

If FES fails and no symbol is found within CAS, a raw data packages should start at the first bit outside the CAS and only raw data packages should be used until channel has been idle.

2.4.5.2 Symbol follow by raw data.

The symbol package should be constructed and a raw data package should be constructed starting at the first low bit unless channel is recognized as idle.

2.4.5.3 Whole Frame follow by raw data

The frame should be constructed then raw data should be collected from the first bit after the FES.

2.4.5.4 Broken Frame follow by raw data

Unless frame is broken by early channel idle detection raw data should be collected from the first bit after an error is found.

2.4.6 Flexray Information Package

Header(6 bytes)

Type of Data (4 bits)

Type of data in this package Whole Frame(00), Broken Frame(01), Symbol(10) or Raw Data(11). First two bits are used but the next two bit are reserved for future use.

Length of Data (9 bits)

The length of the data in bytes, this does not include the header. Max 262 bytes.

Channel (1 bits)

The channel (0 for A, 1 for B) this data was received from.

Time (32 bits)

Time in number of microticks from an unspecified point in time. This is in the form of a little-endian integer and will be reset as it reaches its maximum.

Data (0-262 bytes)

The decoded data, no data when a symbol is decoded otherwise it contains at least the header of a frame even when en empty frame is decoded.

3 Method/experiment

3.1 Testing

To verify the operation of the projects different components, a way of testing each component have to be constructed. For testing purposes the project can be divided in three different parts, the HDL part with the VHDL code written for the FPGA, the hardware part with the physical components of the different boards and the software part with the C/C++ code meant to be running on the ARM microcontroller or a PC.

The hardware will be tested in three steps, first a schematic inspection where the schematic for the hardware will be inspected after its creation preferably by someone else but the creating person. The second step will be inspection of the constructed part. Both to ensure that it is constructed as planned out in the schematic and that each connection carries current only to the correct places. The last part of the hardware testing will be to turn it on and ensure correct voltages and currents where those measurements are possible.

The software part for the C/C++ code will mostly be based on trial and error. This part of the project is relatively small and the created software is expected to be quite minimal both in size and complexity. Most of the function will be handling connections to the hardware with use of Ethernet and USB. This is hard to simulate so the testing will be done through trial and error of real data.

The HDL part will more or less move from software testing to hardware testing as the each step of the testing procedure is passed. This part of the coding is expected to be more error prone than the C/C++ part as it will be far larger in size and more complex. Testing will be done in two parts, the first part using a VHDL testbench with a simulated Flexray environment. The second part will be using physical signals generated by a Flexray Testing System.

The Flexray Testing System is a Flexray Starter kit from Freescale Semiconductors, this has been used in a previous project and has wires attached to the Flexray connectors on one of the boards. This gives access to BM (bus line minus) and BP (bus line plus) on both channel A and channel B and this is the connection to a Flexray bus that will be used to verify the solution. The Starter kit comes with a number of example files for generating test data and is programmable with Freescales CodeWarrior Development Studio, where C code can be written to create new communication.

The testing of the HDL code were made harder by the differences in simulated VHDL and synthesis but most of these troubles could be avoided by restructuring code and accepting delays in the test result.

To make software testing more successful a simulated Flexray environment had to be created. This led to the creating of a VHDL component called Node that is made to simulate the communication of a single Flexray node and library of procedures and functions to create Flexray simulation data. With these tools the VHDL code written for use on the FPGA were easier to validate. To automate the testing the generated signal had to be copied and delayed nine clock cycles, this to take in account the delay caused by the processes when simulated. These automated tests were successful in detecting faults in the code but the lack of actual Flexray data casts some doubt on the tests, as they were created using the same understanding of the system that was used to build the code. It can be argued that any fault except pure programming faults can be found in both the testing code and the code to be tested.

Since the FPGA broke down followed by one of the components of the Flexray testing system the actual verification of the solution created in this project is not possible. Even worse a record of actual Flexray data from the system could not be acquired and therefore no simulation of this data could be constructed.

3.2 Power supply

3.2.1 Theory

Some of the components may have their own power supply but in order for the system to be more reliable all the components will get their power from a central power supply.

3.2.2 Construction of the power supply

In order to build the power supply for the project following the components need to be included; fuse, voltage regulators, LED, resistors, capacitors and heat sinks, see Figure 5. Following chapters contains more detailed information about the components.

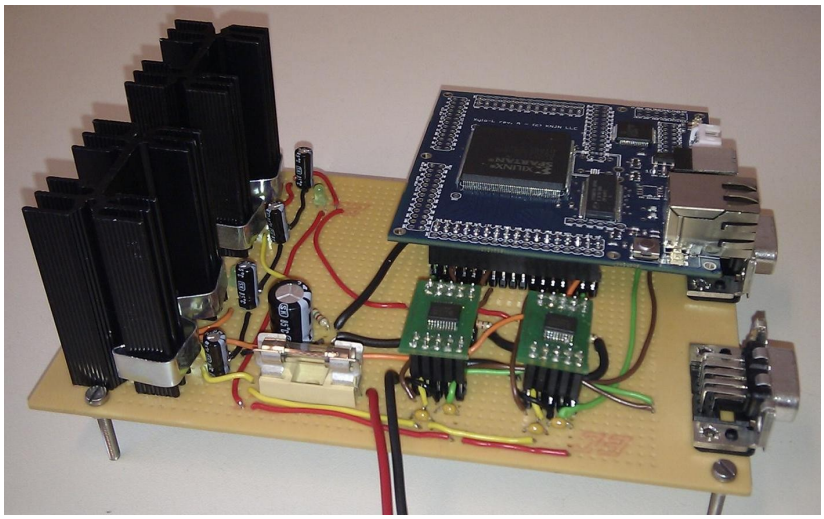


Figure 5: Power supply hardware.

3.2.2.1 Fuse

A fuse is needed to secure current through both voltage regulators. Calculations shows that the maximum current needed in total for the different devices is about 1400mA (See Table 3). To prevent the fuse from breaking due to the current surge when the capacitors are charge as current is first applied, there need to be a fuse that can manage a current larger then 1400mA. An adequate fuse value is 2000mA.

Table 3: Maximum current consumption for the hardware for the three voltages.

Hardware	12V	5V	3.3V
FPGA	-	-	500mA
Transceivers	60mA	70mA	40mA
LEDs	20mA	20mA	20mA
ARM-processor	-	640mA	-
Totally	80mA	730mA	560mA

3.2.2.2 Voltage regulators

The components need different voltages as seen in Table 3. To be able to provide different voltages there need to be two voltage regulators, one for 12V down to 5V and one for 12V down to 3.3V. At first the plan was to put the regulators in series but to prevent one regulator from having to handle all the current the regulators were connected parallel, see Figure 6.

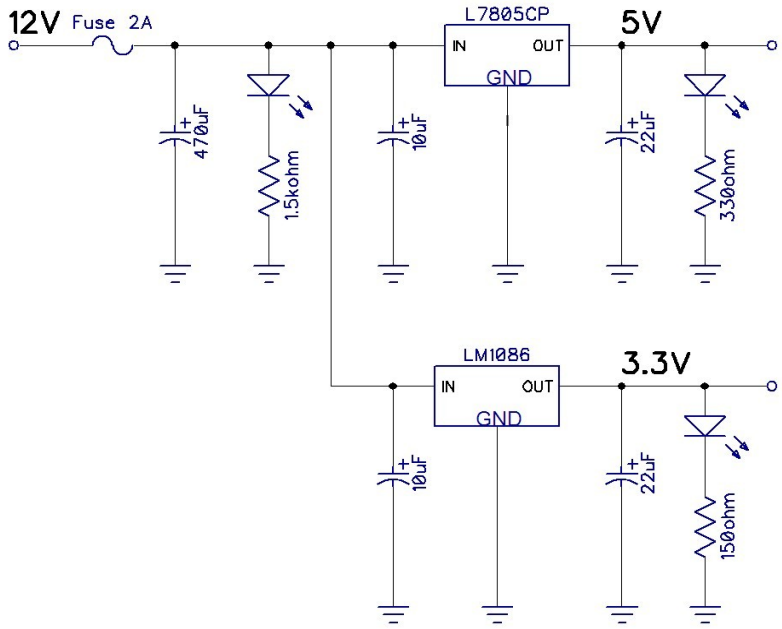


Figure 6: Power supply schematics.

3.2.2.2.1 12V → 5V

To regulate from 12V down to 5V the model L7805CP was chosen. There will be a approximated maximum output current of 800mA, this makes it important to attach a heat sink to prevent the regulator from overheating. The regulator model is ISOWATT-220, which have an insulated case and will not need an extra insulator when attached to a heat sink. L7805CP is able to deliver over one ampere if adequate heat sink is attached. [3]

3.2.2.2.2 12V → 3,3V

Because L7805CP can't regulate the voltage down to 3.3V there will need to be another regulator for this. Several regulators were compared. The models compared was: LM1117T (3.3V), LM1086CT (3.3V), LD1117V (3.3V), LD1117CV (3.3V) and LD1117AV (3.3V).

LM1086CT (3.3V) was chosen because it met the necessary requirements listed below. Some of the data compared were:

•Suitable model for the lab board, TO-220.

•Output current: needs to be about 600mA.

•Input voltage: The power source will have input voltage of about 12V and the voltage regulator need to have input voltage range including 12V. [4]

3.2.2.3 LED

Green LED was chosen to display an active-mode from the three different voltages; 12V, 5V and 3.3V. For the LED to glow it need about 10mA.

3.2.2.4 Resistors

To be able to produce 10mA for the LED there have to be different resistors for the three voltages. The equations 3.1 and 3.2 is used to get the values in the Table 4.

$$U_{tot} = U_{LED} + U_R \qquad \text{(eq. 3.1)}$$

$$U_R = R \cdot I \qquad \text{(eq. 3.2)}$$

Table 4: Resistances and currents for the different voltages.

Voltage (V)	Resistance (Ω)	Current (A)
12V	1500Ω	6.7mA
5V	330Ω	9.0mA
3.3V	150Ω	8.7mA

3.2.2.5 Capacitors

The data sheets of the voltage regulators gives a good idea of what values needed for the capacitors. To be able to withstand large current peaks from the 12V voltage source there is a larger capacitor after the fuse. See Figure 6 for the chosen values of capacitors. [3] [4]

3.2.2.6 Heat sink

Heat sinks are needed because there is high current going through the voltage regulators. The most important value of a heat sink is its thermal resistance value for surface to air, $R_{\theta SA}$. To be able to find a good value of $R_{\theta SA}$ the equation for T_j (see equation 3.3), temperature at junction of voltage regulator, needs to be considered. T_R is the room temperature, $R_{\theta JC}$ is the junction to case thermal resistance, and $R_{\theta CS}$ is the case to surface thermal resistance. [4]

$$T_j = T_R + (R_{\theta JC} + R_{\theta CS} + R_{\theta SA}) \cdot P \qquad \text{(eq. 3.3)}$$

For the voltage regulators in this project $R_{\theta SA}$ = 5.8ºC/W is a good value for the heat sink to keep T_j within the working range of the voltage regulators. See Figure 7 for details about which values of $R_{\theta SA}$ that results in low values of T_j. Another aspect to the choice of heat sink is that lower values of $R_{\theta SA}$ most often means larger heat sink. This need to be considered depending on how much lab board area that can be used for the heat sink.

For the voltage regulator LM1086 there need to be an insulating material between regulator and heat sink. This is because the metallic plate in the regulator is connected to output voltage. Silicon pads were used as insulators since they do not need grease for good contact between the materials.

Figure 7 shows T_j depending on different $R_{\theta SA}$ for both regulators. Without the heat sinks the voltage regulators would reach a theoretic temperature of about 350-400 ºC. [5]

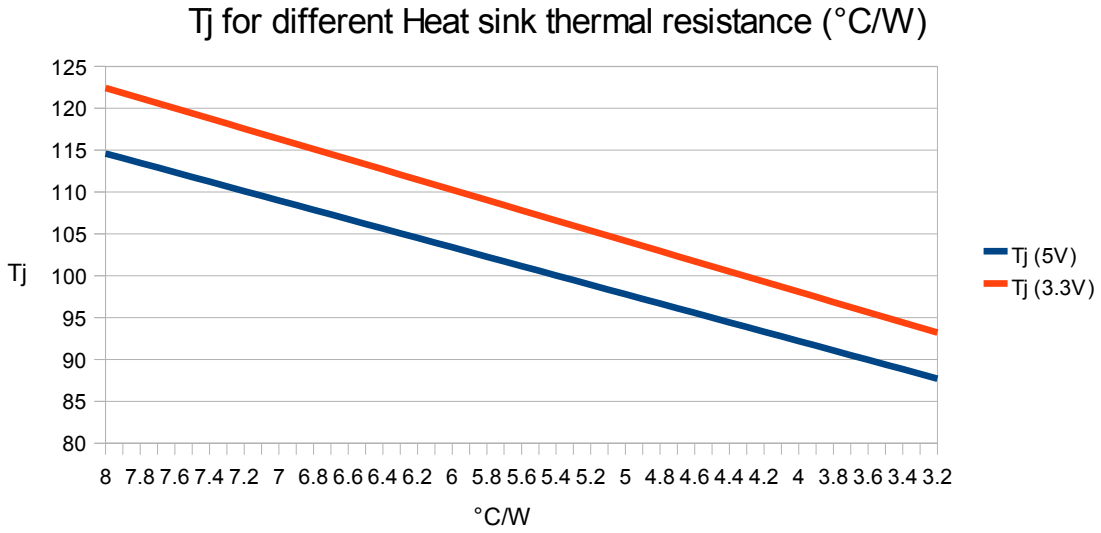


Figure 7: T_j for different Heat sink thermal resistance (°C/W).

3.3 Choice of hardware

The demands of this project were a broad estimate since the software itself would probably use the most of the resources. The maximum size of a Flexray frame is 262 bytes, with the assumption that a frame need to be assembled to be analysed, at least 262*2 = 524 bytes need to be stored per channel. Most likely only one frame can be transmitted to the microcontroller at a time. This demands extra memory for a queue, but with the assumption that data is transmitted with an average of 20 Mbps, 3 frames queue per channel should be enough. That is while one frame is sent, one is analysed and one is decoded. 3 frames per channel means 3*2*262 = 1572 bytes needs to be stored. This applies both the FPGA and the microprocessor. The Flexray communication system has a maximum data transfer rate of 10 Mbps on two data channels, this means 20 Mbps in serial communication is needed, at least as an average over time. The Flexray specification requires each sent bit to be sampled at a rate of 8 times the transfer rate. This means a sample rate of 80*10^6 samples /s and a clock frequency of 80 MHz for the FPGA that does the sampling. [2]

The amount of memory needed to be able to run Linux is 4 MB of Flash and 8-16 MB SDRAM.[6] This project needs both a microprocessor and an FPGA, the best solution would be to have both in the same card. After search for a possible card two cards suitable for our purpose were found. Other cards found were all in the range of 10 000 kr and often more powerful than needed for the project. A comparison of the two card (see Table 5) was made and the Xylo-L was chosen because of its large FPGA, lesser estimated price and their support staff were quick to answer our questions.

Table 5: Board comparison.

Board	Xylo-L	TS-7500
FPGA	Spartan-3E	Lattice XP2
Distributed RAM	73 kB	10 kB
Block RAM	360 kB	166 kB
ARM-processor	ARM7TDMI-S	ARM926-EJ
Micro-controller	LPC3132	CNS2132
SDRAM	32 kB	64 MB
FLASH	64 kB	4 MB

When the preparations to install Linux were made it was discovered that the chosen card, as can be seen in the table, wasn't able to run Linux. This is partly because the lack of memory but also because the ARM7TMI-S processor lacks an MMU, or an EMI (External Memory Interface). This was overlooked when the decision to buy a card was made both due to the lack of knowledge of the ARM architecture and human error.

3.4 Flexray Transceiver TJA1080

To be able to receive information from the Flexray system there needs to be two transceivers, one for each channel. The Flexray nodes from the Flexray Starter kit uses the transceiver TJA1080, this model are also used in this project.
The transceivers in this project will only receive data from the Flexray system, this makes it easier to configure it when not all the I/O pins are used. [7]
The schematic for the transceivers connected to the FPGA pins and the bus pins is shown in Figure 8. See the Table 6 for more details about the pins of the transceiver.

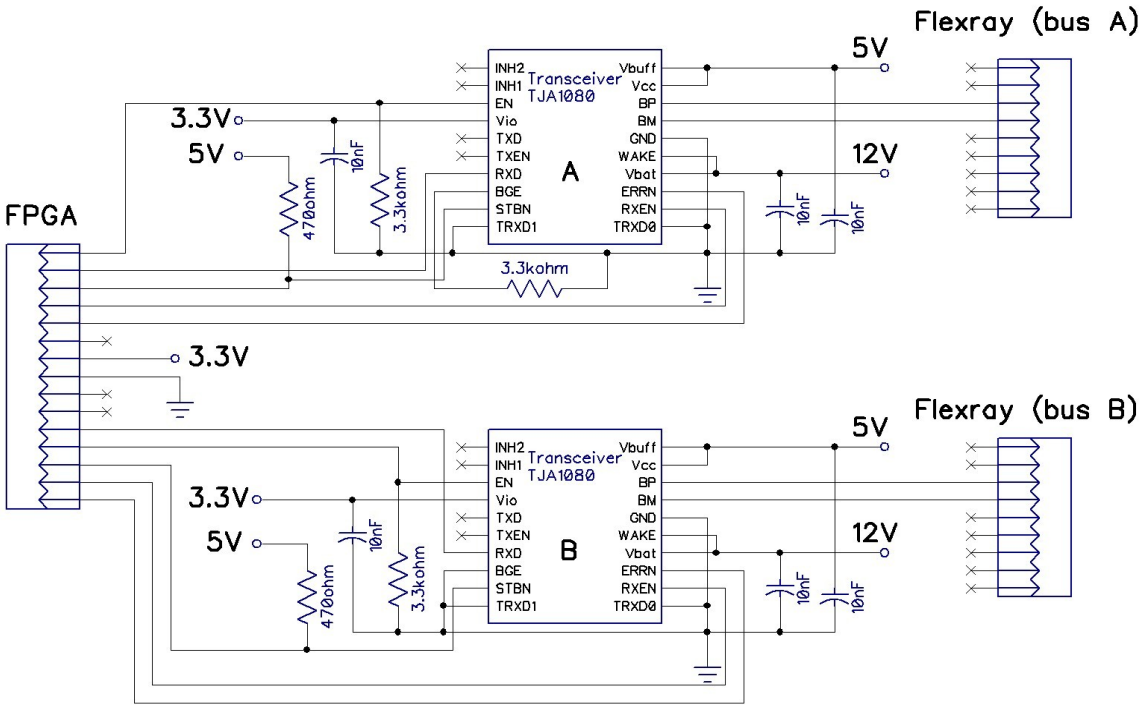


Figure 8: Schematics of the two Flexray transceivers.

Table 6: Pin declaration.

Pins	Description
INH1-2	They are used for switching external voltage regulators. These are not used in this project and are therefore disconnected.
EN	EN together with STBN controls which mode (*Normal, Receive only, Go-to-sleep, Standby or Sleep*) the transceivers are using. *Receive only* is the only mode that is wanted for this project. EN has an internal pull-down and to make sure it stays down there is an external pull-down as well.
VIO	Same voltage as the external system which in this case is the FPGA with 3,3V.
TXD	This pin transmits data from the transceiver to the Flexray system, but in this project there will not be any data sent and that is why the pin is disconnected.
TXEN	Controls when to write data to the Flexray system. In this project there will not be any data sent and that is why the pin is disconnected. TXEN has an internal pull-up and when it is high the transmitter is disabled.
RXD	This pin transmits data from the transceiver to the FPGA.
BGE	Bus guardian enable, it has an internal pull-down which makes the transmitter disabled. It also has an external pull-down to force a low value.
STBN	STBN together with EN controls which mode the transceivers are using. For *Receive only* mode STBN need to be set high. It has an internal pull-down. To always make it high it has an external pull-up.
TRXD0-1	They are only used in the star configuration and are therefore disconnected.
RXEN	Gives a low signal to the FPGA when activity on the Flexray bus is detected.
ERRN	Detects errors in the transceivers and send out a low signal when error is detected.
VBAT	It is needed to be able to start the transceivers. (12V)
WAKE	It has internal-pull and pull-down depending on which state it had before. To prevent local wake-ups to be detected the WAKE pin is connected to an external pull-up.
GND	Ground.
BM	Flexray bus line minus.
BP	Flexray bus line plus.
VCC	Supply voltage. (5V)
VBUF	Buffer to supply voltage. (5V)

To set the transceivers in *Receive only* mode EN need to be set low and STBN high. To prevent the transceivers from changing mode both EN pin and STBN pin have external pull-ups and pull-downs. This is done by connecting external resistors of 3.3kΩ, which is about 10% of the internal resistor, to the pins.

The transmitter functions in the transceivers are disabled by setting the BGE pin to low, by connecting it to ground, and the TXEN pin to high.
To prevent current spikes the voltage pins each are connected to a 10nF capacitor. See Figure 8 for the schematics of the transceivers.
Each transceiver connected to the FPGA, need two I/O pins (EN and STBN) and three input pins (RXD, RXEN and ERRN). The ERRN pin is connected to the FPGA but the information from it will not be of interest in this project, but might be of interest to include in future studies.
In order to use the transceiver VBAT need at least 6.5V, and in this case it will be given about 12V. VCC and VBUF need 5V, VIO is set to the same as what the connected system are using in voltage, in this case the FPGA are using 3.3V.

3.5 Microcontroller board

3.5.1 Building the microcontroller board

Since the Xylo-L board didn't have the ARM-based microcontroller with the specifications needed for the construction of the the prototype, a replacement was needed. There were three choices for this replacement, buying another board with both FPGA and ARM-based microcontroller, buying a pure ARM-based microcontroller board and connect it to the first FPGA board, or construct a new ARM-based microcontroller board and connect it to the first FPGA board.

3.5.1.1 FPGA and microcontroller board

Buying a new FPGA and microcontroller would mean redoing the original work of establishing requirements, for both FPGA and microcontroller, given the greater understanding of the project gained during the already completed parts. This could also mean that the new board may not have the same FPGA or even same brand of FPGA, which means new development tools must be acquired and studied. The advantages of this solution is that the FPGA and microcontroller will most likely have hard wired communication lines and include code to ease the writing of the code for communication between the FPGA and ARM. Since this was eight weeks into the project this choice was to expensive in time and money.

3.5.1.2 Buying an microcontroller board

Buying an ARM-based microcontroller board means re-evaluate the requirements only for the ARM-based parts then searching for possible candidates. The advantage of this solution is a complete board usually comes with a preconfigured embedded Linux distribution ready to install. The downside is a lesser understanding of both the principles behind creating a board and the individual parts. Since getting hardware requirements of the product was part of the project this solution was not used.

3.5.1.3 Constructing a new microcontroller board

Constructing a new ARM-based microcontroller board is a harder and more time-consuming solution but it will also give a greater understanding of the board itself and the adaptation of Linux to the hardware. This is beneficial since more precise demands of the final product can be estimated. An estimate of the time required was made and it seemed possible to finish this part in time, especially since the microcontroller wasn't needed until the later part of the project. Also none of this extra time would go to reconstructing the already written parts, as those of the FPGA and possibly the power supply.

3.5.1.4 Building the microcontroller board

To build the microcontroller board a schematic is needed and creating one from scratch can be a long process requiring knowledge and a lot of trial and error. It is possible to adapt a schematic that already had been written and used to construct a board, if a suitable board is found with a schematic.
Olimex Ltd is a company that designs and constructs microcontroller board, since they provide schematics for each board they design their site were used as a start in the search for schematics. It turns out that finding a microcontroller with the right packaging was problematic. Most controllers are made with the FBGA (Fine-pitch Ball Grind Array), a package type that need special ovens to solder.
At a suggestion from the project supervisor Björn, the search were redirected to companies that constructs microcontrollers to find if any of these companies were willing to provide free samples of their products. Using the Olimex website and wikipedia, a long list of possible companies were found. This list was quickly reduced to three companies, NXP Semiconductors, Texas Instrument and Freescale Semiconductors, that were the only companies that had a samples program. After reviewing the selection of possible microcontrollers the IMX233CAG from Freescale Semiconductors was the only

microcontroller in the samples programs that suited the requirements. This microcontroller could be bought with an evaluation board to which Freescale provided a schematic. This schematic was used as basis for the new schematic. But before it could be used the components of the schematic had to be checked to ensure there existed similar components with a package actually possible to assemble with the available tools.

To created the ARM board a prototype board will be used on which the components will be assembled. This is 10x20 cm large with 50x20 holes positioned 8 mm apart. This means the ARM board will be using only components with through-hole connections or attached to SMD-adapters.

3.5.2 The i.MX233CAG4B

The i.MX233 was not ideal for this project since it was made for other purposes. It contains a large amount of sound and video technology that were of no use for the project. On the plus side it contains a power management unit that given a 5V supply voltage provides peripherals with needed voltages as 3.3, 2.5 and 1.8. The micro-controller board has no JTAG connection, the reason for this is that a connection would have to be built, but since a real Freescale programmer is very expensive another JTAG-to-USB programmer have to be used. More precisely the “ARM Tiny USB” connector from Olimex since this is supported by openocd an open source ARM debugger. But the i.MX233CAG4B only has an SJTAG pin which need a Freescale made CPLD to convert information to JTAG pins. To created an ordinary JTAG connection the microcontroller had to be booted with instructions to use the JTAG pins, this means the microcontroller must be booted by an SD-card first, which just as easily can boot Linux directly. The JTAG connection was therefore deemed unnecessarily complicated and omitted.
As the most important part of the project was the acquiring of data from the Flexray system, the ARM board was put on hold when the FPGA broke down. As a result it was never finished and the board was only half constructed.

3.6 VHDL

In order to control the FPGA a hardware description language, such as VHDL or Verilog, is needed. Both are good choices and both have their pros and cons. Due to previous knowledge the decision was made to use VHDL.

The goal of the VHDL code is to correctly read the bits from the signal RXD, from the Flexray system, and detect the type of data received. When this is completed the contents of the bytes are sent to the ARM processor.
The program used was ISE Webpack from Xilinx because the chosen FPGA, Spartan 3E, was from Xilinx.

The code have been divided into three parts, sampling, bitstrobing and framedecoder. First the signal is sampled in the sampling process to prevent small glitches in the signal, then sent through the bitstrobing process which synchronizes the bit timing of the output value, then through the framedecoder process which checks the different parts of the frame. When these steps are done the contents of the bytes are sent to the ARM processor. More detailed description of the three processes is in following chapters. The VHDL code is shown in appendix A.

3.6.1 Sampling

This process samples the signal RXD from the transceiver as shown in Figure 9. Each clock cycle from the channel sample clock one new sample of RXD is temporary stored. The last five samples are temporary stored in voting window which is used to determine the output signal, VotedValue. The majority of the stored sample values, ones or zeroes, in the voting window is what is sent out on VotedValue each channel sample clock cycle. This method makes a delay of cVotingDelay between the signals RXD and VotedValue, see Figure 9. By using this method small glitches of maximum two channel sample clock cycles is removed when sent to output VotedValue. [2]

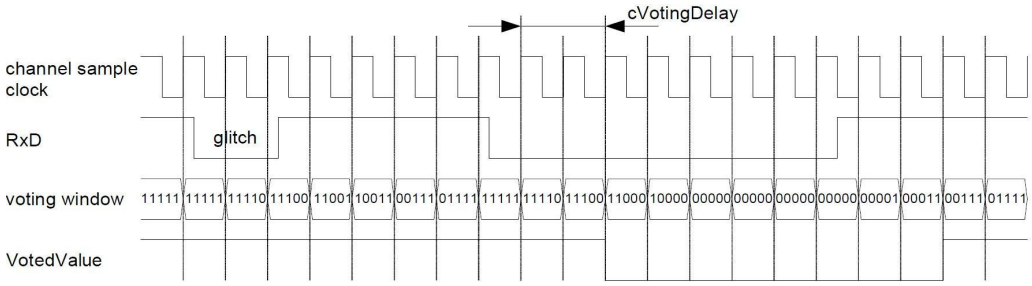


Figure 9: Sampling process.

3.6.2 Bitstrobing

When the framedecoder is processing a byte the bitstrobing only strobes, without synchronize the bits, and send them through to the framedecoder process. Otherwise the strobed bit is synchronized when the VotedValue has a fallen edge, *bit synchronization edge* (see Figure 10). This is done by resetting the sample counter.

The process strobes the value from VotedValue when the sample counter reaches *cStrobeOffset* (see Figure 10). [2]

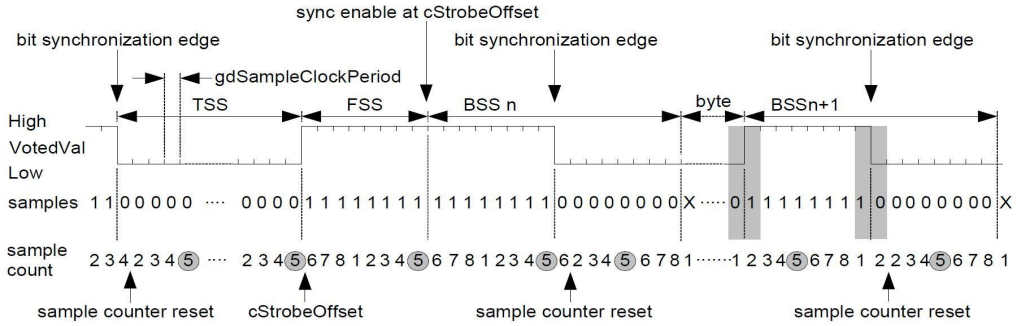


Figure 10: Bitstrobing process.

3.6.3 Framedecoder

Now when sampling and bitstrobing have checked the correct value of a bit and synchronize the signal, it is time for the framedecoder to check what kind of information is sent through the Flexray system.

The VHDL code for framedecoder is divided into seven states, this to make it possible to loop through smaller code parts. The overall flowchart for framedecoder is shown in Figure 11 and also in Appendix B.

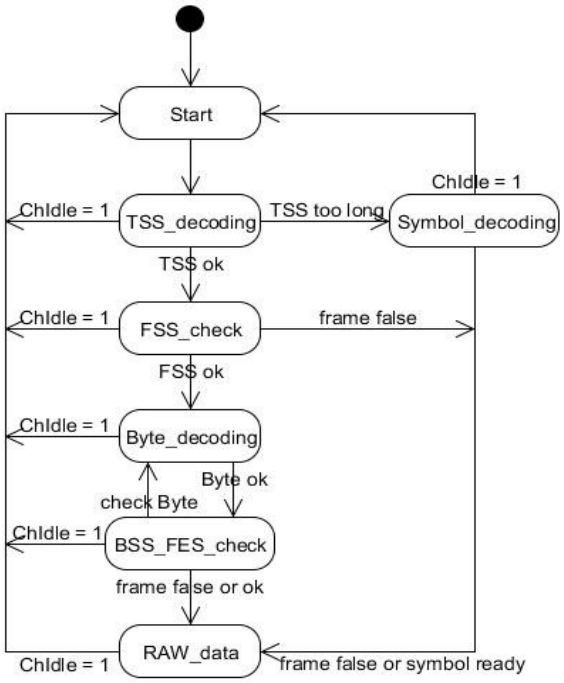


Figure 11: Flowchart for framedecoder.

There is one flowchart for each of these states, see Appendix B.

In order for the process to do anything it must detect a bit from bitstrobing and the channel cannot be idle. If the channel idle becomes high in the middle of the framedecoder process it will go to the Start state and send a signal to tell that the frame is false.

Table 7: Summary of the different states.

States	Description
Start	Reset some of the signals.
TSS_decoding	Checks if the TSS is correct. TSS is allowed to have up to *gdTSSTransmitter* (variable between 3 and 15) zero bits followed by a one bit. If it is more then *gdTSSTransmitter* zero bits the package is not a frame but might be a symbol. *gdTSSTransmitter* is set when the Flexray system is constructed. If the first one bit comes before too many zeroes have passed the process will continue with FSS_check state otherwise it goes to Symbol_decoding state.
Symbol_decoding	Checks if the data package is a symbol or just a false frame. If there are more zeroes then *gdCASRxLowMax* (variable between 67 and 99) or if there is a one bit before there has been at least *cdCASRxLowMin* (constant of 29) zeroes the frame is considered false and the process continues in the RAW_data state. In this cases the process also sends out a frame false signal. If there is between *cdCASRxLowMin* and *gdCASRxLowMax* zeroes followed by a one bit the data package is considered a symbol and a symbol identification is sent out and the process continues in the RAW_data state.
FSS_check	Checks if the FSS is correct. If FSS is OK it should contain one to two 1 bits. Then the process will continue with Byte_decoding state otherwise the frame is considered false and the process will go to RAW_data state and a frame false signal is sent out.
Byte_decoding	It loops through eight bits (one byte) and sends out each bit after it arrives from the bitstrobing process. This state and RAW_data state is the only states that sends out data bits. After the Byte_decoding state is finished it continues to BSS_FES_check state.
BSS_FES_check	After every sent byte it checks if it is a BSS or an FES. If the sequence is a one bit then a zero bit it is a BSS and if it is a zero bit then a one bit it is an FES. If a correct FES is detected a signal is sent out telling that a complete frame has been received. After an FES the process continues to RAW_data state to see if there is any more bits to store as raw data after the frame. If there are no more bits coming and the signal channel idle goes high the framedecoder process continues to the Start state and waits for the next bit. After a BSS the process increase the byte counter and keeps track of how many bytes the frame contains. The maximum number of bytes in a frame is 262 and if the counter exceeds 262 bytes the process sends out a signal telling that it is a false frame and continues to RAW_data state for the following bits. Also if detected that it is not a BSS or an FES the process sends out a signal telling that it is a false frame and continues to RAW_data state for the following bits.
RAW_data	This parts takes care of all data bits which follows a correct frame, a false frame or a symbol. The state loops through the bits and sends out the bits until the channels idle is set high.

3.6.4 Simulation

When the VHDL code was to be put into a simulation some things to be considered. There are three predefined variables, *gdTSSTransmitter*, *cdCASRxLowMin* and *gdCASRxLowMax*, of which *cdCASRxLowMin* is fixed but the other two varies within a range of possible values. Their values are set in the construction of a new Flexray system. This project will not receive the correct variables because it will not be present during construction of the new Flexray system. After some consideration it was decided that the best results for the project, without knowing the exact values,was when the highest possible values were chosen.
Other problems with the simulation was to consider how the simulation dealt with the signals of the project. To make sure the simulation was the same as expected from using the FPGA some of the VHDL code need to be changed to fit the working process of the simulation.
In the simulation example, see Figure 12, it is shown that strobed_value is slightly delayed in comparison to RXD because of the sampling and bitstrobing processes delays the signal with three clock cycles. strobed_value is the signal which arrives from bitstrobing and RXD is what arrives from the transceiver. strobed_flag goes high every time *cStrobeOffset* is reached and it tells the framedecoder it is OK to receive the next bit. data_flag tells the ARM-processor that it is OK to get the next data_out bit from the framedecoder. States from framedecoder process are also seen in Figure 12.

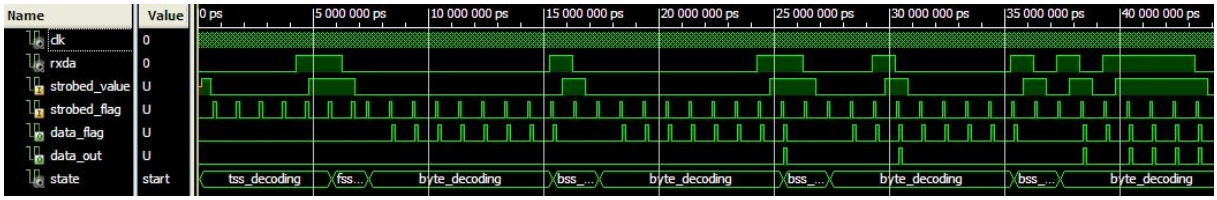


Figure 12: Simulation of the first 40µs of a frame.

Because of the broken FPGA this was not possible to test in other ways than simulating the VHDL code. The VHDL code is behaving as expected.

4 Results

The hardware verification of the solution was not possible since the hardware broke down as the final test were to begin. The tests that were performed did not verify a stable connection between the Flexray Communication System and the FPGA. The reasons for this are unknown but the two transceivers became warm, creating a suspicion of malfunction. Since the FPGA was damaged and they are all connected to each other both with power and data lines, it may be the same event causing all problems.

No real measurement of the power consumption was possible so the estimated values are used to form an estimate of the prototypes consumption. This estimated values are based on the information gain from the data sheets for the components and are given in Table 3.

5 Conclusion

To address the goals and issues of this project:
Is it possible to receive data from the Flexray communication System without knowledge of the systems configuration?
Our conclusion is that it should be possible, the problem will be to evaluate the information gain from the system. The solution we constructed is focused on just receiving the frames and symbols sent on the system, but this is not enough to analyse a network in any depth. Without knowledge of the configuration it is impossible to know if a frame is a faulty transmission or not. Our suggestion is to use an FPGA as data receiver and then send this information to another system, either an embedded analyser or a PC and analyse this data with the help of knowledge of the system configuration or use of statistical analysis.

Which are the demands of the product in terms of components, power and size?
In this project the ARM-based part running embedded Linux would actually just be a very inefficient Ethernet controller. The theory is that this part could be developed to do further analysis beyond the scope of the FPGA. But given the fact that the microcontroller has a high power demand separating the components is a better option. The FPGA is able to do the first part of analysis and data gathering without extra help. By separating the components it is possible to choose if the data transmitted by the FPGA should be picked up by the microcontroller or a PC. If a embedded Linux solution is used it must do more than just transferring data, to motivate the extra hardware and power needed. Instead it could be used to analyse data and prepare it for presentation.

References

[1] www.flexray.com, accessed on 2010-08-10

[2] FlexRay Communications System Protocol Specification, v2.1 Revision AA, FlexRay Consortium, December 2005

[3] data sheet L7800 series, http://pdf1.alldatasheet.com/datasheet-pdf/view/22614/STMICROELECTRONICS/L7800.html, accessed on 2010-08-05

[4] reference data sheet LM1086, http://pdf1.alldatasheet.com/datasheet-pdf/view/8592/NSC/LM1086.html, accessed on 2010-08-02

[5] Heat sink BW50-2G, http://www.aavidthermalloy.com/cgi-bin/stdisp.pl?Pnum=bw50-2g, accessed on 2010-08-02

[6] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef & Phillipe Gerum, Building Embedded Linux Systems, O'Reilly, 2008, ISBN 987-0-596-52968-0

[7] TJA1080 Flexray transceiver, http://pdf1.alldatasheet.com/datasheet-pdf/view/164398/PHILIPS/TJA1080.html, accessed on 2010-08-10

Appendix A (VHDL code)

Sampling:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Entity for sampling the incoming bitstream from the Flexray bus.
-- Check the Flexray protocol for details and background.
entity Sampling is
Port (clk : in std_logic;  -- RxD from the bus driver (Received data signal)
      RxD : in std_logic;  -- clocked with the sampling frequency
      VotedValue : out std_logic);-- current bit given by the voting process,
end Sampling;              -- out to the bitstrobing process

architecture Behavioral of Sampling is

-- stored data for the voting process
signal VotingWindow : std_logic_vector (0 to 4) :=  ('1','1','1','1','1');

begin
-- Sample one bit every sampling clock cycle and
-- vote on the collected samples.
Sample: process(clk)
--variable votingwindow : std_logic_vector (0 to 4);
begin
-- On a rising edge shift out the longest know value
-- and add the new value to the voting window.
if rising_edge(clk) then
   VotingWindow(0 to 3) <= VotingWindow(1 to 4);
   VotingWindow(4) <= RxD;

   case VotingWindow is
      when "00000" => VotedValue <= '0';
      when "00001" => VotedValue <= '0';
      when "00010" => VotedValue <= '0';
      when "00011" => VotedValue <= '0';
      when "00100" => VotedValue <= '0';
      when "00101" => VotedValue <= '0';
      when "00110" => VotedValue <= '0';
      when "00111" => VotedValue <= '1';
      when "01000" => VotedValue <= '0';
      when "01001" => VotedValue <= '0';
      when "01010" => VotedValue <= '0';
      when "01011" => VotedValue <= '1';
      when "01100" => VotedValue <= '0';
      when "01101" => VotedValue <= '1';
      when "01110" => VotedValue <= '1';
      when "01111" => VotedValue <= '1';
      when "10000" => VotedValue <= '0';
      when "10001" => VotedValue <= '0';
      when "10010" => VotedValue <= '0';
      when "10011" => VotedValue <= '1';
      when "10100" => VotedValue <= '0';
      when "10101" => VotedValue <= '1';
      when "10110" => VotedValue <= '1';
      when "10111" => VotedValue <= '1';
      when "11000" => VotedValue <= '0';
      when "11001" => VotedValue <= '1';
      when "11010" => VotedValue <= '1';
      when "11011" => VotedValue <= '1';
      when "11100" => VotedValue <= '1';
      when "11101" => VotedValue <= '1';
      when "11110" => VotedValue <= '1';
      when "11111" => VotedValue <= '1';
      when others => VotedValue <= '0';
   end case;

end if;
end process;
end Behavioral;
```

Bistrobing

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity bitstrobing is
   Port (clk           : in  STD_LOGIC;
         voted_value   : in  STD_LOGIC;
         EnEdgeEnable  : in  STD_LOGIC;
         strobed_value : out STD_LOGIC;
         strobed_flag  : out STD_LOGIC;
         chIdle        : out STD_LOGIC);
end bitstrobing;

architecture Behavioral of bitstrobing is

-- Constants for the Flexray protocol
constant SamplesPerBit        : integer range 0 to 8 := 8;
constant StrobeOffset         : integer range 0 to 5 := 5;
constant ChannelIdleDelimiter : integer range 0 to 11 := 11;

-- Variables as defined by the flowcharts in
-- in the Flexray protocol.
signal preStrobedValue  : STD_LOGIC := '1';
signal enEdgeDetect     : STD_LOGIC := '1';
signal preVotedValue    : STD_LOGIC := '1';
signal sampleCounter    : integer range 0 to 8 := 0;
signal channelIdleCount : integer range 0 to 11 := 0;

begin

-- strobed_value only need to have a defined
-- value when the strobed_flag is set high.
-- strobed_value <= voted_value;

chIdle <= '1' when channelIdleCount = 11 else '0';

process(clk)
begin
   if rising_edge(clk) then
      -- Strobed_flag is set low since it only should be held
      -- high for one clock cycle.
      strobed_value <= voted_value;
      strobed_flag <= '0';
      enEdgeDetect <= EnEdgeEnable AND enEdgeDetect;

      if( EnEdgeDetect = '1' AND preVotedValue = '1' AND voted_value = '0') then
         enEdgeDetect <= '0';
         sampleCounter <= 2;
      else
         if(samplecounter = StrobeOffset) then
            strobed_flag <= '1';
            enEdgeDetect <= EnEdgeEnable AND voted_value;

            if(voted_value = '0') then
               channelIdleCount <= 0;
            else
               if(channelIdleCount < ChannelIdleDelimiter) then
                  channelIdleCount <= channelIdleCount +1;
               end if;
            end if;
            preStrobedValue <= voted_value;
         end if;

         if(sampleCounter < samplesPerBit) then
            sampleCounter <= sampleCounter + 1;
         else
            samplecounter <= 1;
         end if;
      end if;
      preVotedValue <= voted_value;

   end if;
end process;

end Behavioral;
```

Framedecoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FutureFrame is
   Port (clk          : in STD_LOGIC;
         strobed_value: in STD_LOGIC;
         strobed_flag : in STD_LOGIC;
         chIdle       : in STD_LOGIC;
         data_out     : out STD_LOGIC;
         data_flag    : out STD_LOGIC;
         data_type    : out STD_LOGIC_VECTOR(1 downto 0);
         data_done    : out STD_LOGIC;
         EnEdgeEnable : out STD_LOGIC
         );
         --data_types: frame_ok=00, frame_false=01,
         --symbol=10, RAW data=11

end FutureFrame;

architecture Behavioral of FutureFrame is

type state_type is (Start,TSS_decoding,FSS_check,Byte_decoding,
                    BSS_FES_check,Symbol_decoding,RAW_data);

signal state    : state_type := Start;
signal zBitCnt  : integer range 0 to 200;
signal zBit     : std_logic;
signal zByteCnt : integer range 0 to 300;
signal BF_count : integer range 0 to 1;
signal BF_error : boolean;
signal BSS_FES  : std_logic;  --BSS_FES = '1' => BSS, BSS_FES = '0' => FES

procedure BSS_FES_DECODING (zStrobed             : in std_logic;
                            signal BSS_FES_error : out boolean
                           ) is

variable zB : std_logic;
begin
   zB := zStrobed;
   case BF_count is
         when 0 =>BF_count <= BF_count + 1;  --Always check 2 bits for BSS and FES.
                  BSS_FES_error <= false;
                  if zB = '1' then
                     BSS_FES <= '1';  --It's a beginning of a BSS.
                  else
                     BSS_FES <= '0';  --It's a beginning of a FES.
                  end if;

         when 1 =>BF_count <= 0;
                  if (zB = '0' and BSS_FES = '1') or
                     (zB = '1' and BSS_FES = '0') then
                     BSS_FES_error <= false;
                  elsif (zB = '0' and BSS_FES = '0') or
                     (zB = '1' and BSS_FES = '1') then
                     BSS_FES_error <= true;
                  end if;
         end case;
end BSS_FES_DECODING;

begin

process(clk)

--For best results without knowing the below constants
--these values are chosen as the best alternatives.
constant gdTSSTransmitter  : integer := 15; -- 3-15 gdBits (possible values)
constant cdCASRxLowMin     : integer := 29; -- 29 gdBit
constant gdCASRxLowMax     : integer := 99; -- 67-99 gdBit (possible values)

begin

if rising_edge(clk) then

   data_flag <= '0';
   data_out <= '0';
   data_done <= '0';
```

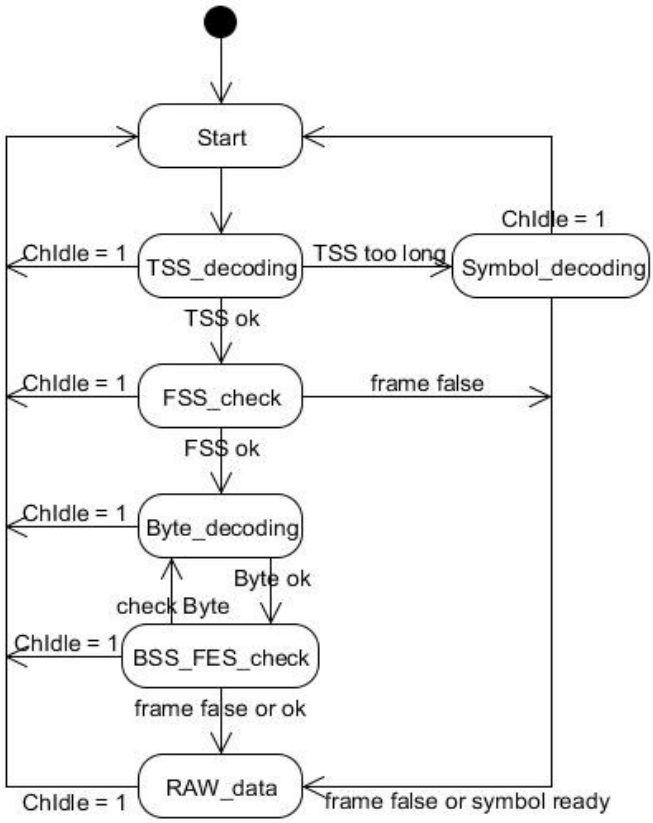
```
--Set all necessary signals into start values
--and then continue to the next state.
if state = Start then
   state <= TSS_decoding;
   zBitCnt <= 1;
   zBit <= '0';
   zByteCnt <= 0;
   data_type <= "00";
   EnEdgeEnable <= '1';
   BF_count <= 0;
   BF_error <= false;
   BSS_FES <= '0';
end if;

if strobed_flag = '1' then

   case state is
         when Start => null;

         --Check if it's a frame or a symbol.
         --It will continue to look for a 1 bit, If there
         --are to many 1 bits the process will continue
         --to see if it's a symbol.
         when TSS_decoding =>
               if chIdle = '0' then   --This checks so the
                  zBit <= strobed_value;            --channel is not idle.
                  if strobed_value = '1' then
                     --A 1 bit is found within the numbers of bits allowed.
                     state <= FSS_check;
                     zBitCnt <= 1;
                  elsif strobed_value = '0' and zBitCnt > gdTSSTransmitter then
                     --gdTSSTransmitter - The number of zeroes
                     --that's allowed for a symbol.
                     state <= Symbol_decoding;
                     zBitCnt <= zBitCnt + 1;
                  else
                     state <= TSS_decoding;
                     zBitCnt <= zBitCnt + 1;
                  end if;
               else
                  state <= Start;
                  data_type <= "01";
               end if;

          --Check if correct FSS and first BSS of the frame.
         when FSS_check =>
               if chIdle = '0' then
                  zBit <= strobed_value;
                  if strobed_value = '1' and zBitCnt > 2 then
                     --There are too many 1 bits => set frame
                     --as false and continue to read RAW_data.
                     data_type <= "01";
                     data_done <= '1';
                     state <= RAW_data;
                     zBitCnt <= 1;
                  elsif strobed_value = '1' and zBitCnt <= 2 then
                     --There are still not too many 1
                     --bits and the FSS_check continues.
                     data_type <= "00";
                     data_done <= '0';
                     state <= FSS_check;
                     zBitCnt <= zBitCnt + 1;
                  elsif strobed_value = '0' then
                     --Now a 0 bit is found and the process
                     --starts to check for Bytes.
                     data_type <= "00";
                     data_done <= '0';
                     state <= Byte_decoding;
                     zBitCnt <= 1;
                  end if;
               else
                  state <= Start;
                  data_type <= "01";
                  data_done <= '1';
               end if;
```

```
when Byte_decoding =>
      if chIdle = '0' then
         data_type <= "00";
         data_done <= '0';
         if zBitCnt > 7 then
         --One whole Byte is done and next is to
         --check for BSS/FES.
            EnEdgeEnable <= '1';
            zBitCnt <= 1;
            zBit <= strobed_value;
            data_out <= strobed_value;
            data_flag <= '1';
            state <= BSS_FES_check;
            zByteCnt <= zByteCnt + 1;
         else
         --Still not finished with a whole Byte.
         --Sends out one bit eight times.
            EnEdgeEnable <= '0';
            zBitCnt <= zBitCnt + 1;
            zBit <= strobed_value;
            data_out <= strobed_value;
            data_flag <= '1';
            state <= Byte_decoding;
            zByteCnt <= zByteCnt;
         end if;
      else
         state <= Start;
         data_type <= "01";
         data_done <= '1';
      end if;

--Checks if it's a BSS, FES or BF_error.
when BSS_FES_check =>
      if chIdle = '0' then
         zBit <= strobed_value;
         BSS_FES_DECODING(strobed_value,BF_error);
         --if (BF_count = 0) and (not BF_error) and BSS_FES = '1' then
         --In order for the simulation to work the below code line
         --must be used, but in real tests the code line above
         --must be used.
         if (BF_count = 1) and (not BF_error) and BSS_FES = '1' then
            if zByteCnt > 263 then
               --If zByteCnt = 264 and BSS_FES=1 then there
               --are to many Bytes and the frame is false.
               --The following data is continued in RAW_data.
               data_type <= "01";
               state <= Raw_data;
               data_done <= '1';
               zBitCnt <= zBitCnt;
            else
               --BSS is ok and then continue to check next byte.
               data_type <= "00";
               state <= Byte_decoding;
               data_done <= '0';
               zBitCnt <= 1;
            end if;
         --elsif (BF_count = 0) and (not BF_error) and BSS_FES = '0' then
         --In order for the simulation to work the below code line
         --must be used, but in real tests the code line above
         --must be used.
         elsif (BF_count = 1) and (not BF_error) and BSS_FES = '0' then
            --FES is ok and following data will be included in RAW_data.
            data_type <= "00";
            state <= RAW_data;
            data_done <= '1';
            zByteCnt <= 0;
            zBitCnt <= 1;
         elsif (BF_count = 1) and BF_error then
            --Error encountered and following data
            --will be included in RAW_data.
            data_type <= "01";
            state <= Raw_data;
            data_done <= '1';
         else
            --Still on first bit.
            data_type <= "00";
            state <= BSS_FES_check;
            data_done <= '0';
         end if;
      else
         state <= Start;
         data_type <= "01";
         data_done <= '1';
      end if;
```

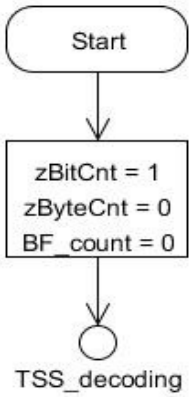
```
            --To be a symbol there is has to be between cdCASRxLowMin
            --and gdCASRxLowMax zeroes before the first one bit.
            --Otherwise the rest of the frame is seen as RAW_data.
            when Symbol_decoding =>
                  if chIdle = '0' then
                     zBit <= strobed_value;
                     if strobed_value = '1' and zBitCnt >= cdCASRxLowMin then
                        data_done <= '1';
                        zBitCnt <= zBitCnt;
                        data_type <= "10";
                        state <= RAW_data;  --Directed to RAW_data to look
                                            --for data after finished Symbol.
                     elsif (strobed_value = '1' and zBitCnt < cdCASRxLowMin) or
                           (strobed_value = '0' and (zBitCnt + 1) > gdCASRxLowMax) then
                        data_done <= '1';
                        zBitCnt <= zBitCnt;
                        data_type <= "01";
                        state <= RAW_data;  --Directed to RAW_data to look
                                            --for data after false frame.
                     elsif strobed_value = '0' and (zBitCnt + 1) > gdCASRxLowMax then
                        data_done <= '0';
                        zBitCnt <= zBitCnt + 1;
                        data_type <= "10";
                        state <= Symbol_decoding;
                     end if;
                  else
                     state <= Start;
                     data_type <= "01";
                     data_done <= '1';
                  end if;

            --When starting to register the RAW_data the
            --BSS/FES checkbits will not be included.
            when RAW_data =>
                  zBit <= strobed_value;
                  EnEdgeEnable <= '0';
                  data_out <= strobed_value;
                  data_type <= "11";
                  data_flag <= '1';
                  if zBitCnt > 7 then
                     zByteCnt <= zByteCnt + 1;
                     zBitCnt <= 1;
                  else
                     zByteCnt <= zByteCnt;
                     zBitCnt <= zBitCnt + 1;
                  end if;
                  --When chIdle is set channel Idle is detected and the
                  --next state will be Start waiting for new frame/data.
                  --Otherwise RAW data will continue to be registered.
                  if chIdle = '1' then
                     state <= Start;
                     data_done <= '1';
                  else
                     state <= RAW_data;
                     data_done <= '0';
                  end if;

            when others => null;

      end case;
   end if;
end if;

end process;
end Behavioral;
```

Appendix B (Flowcharts)

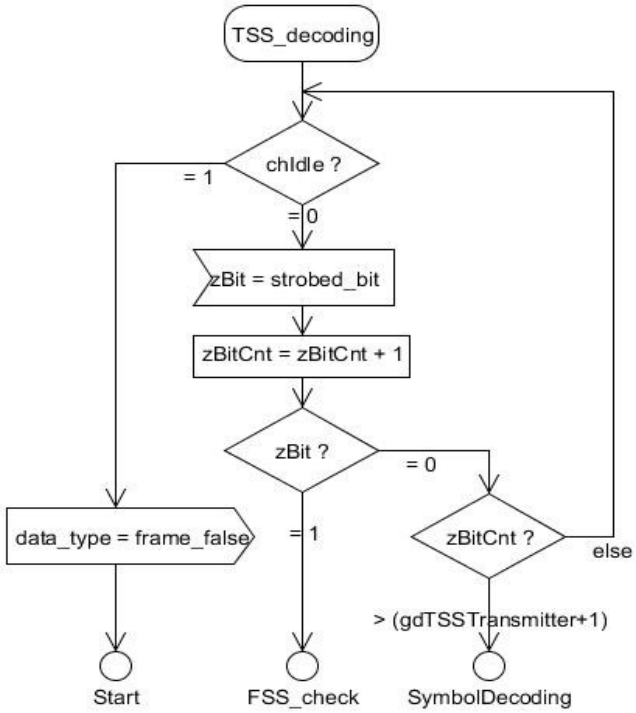
Framedecoder



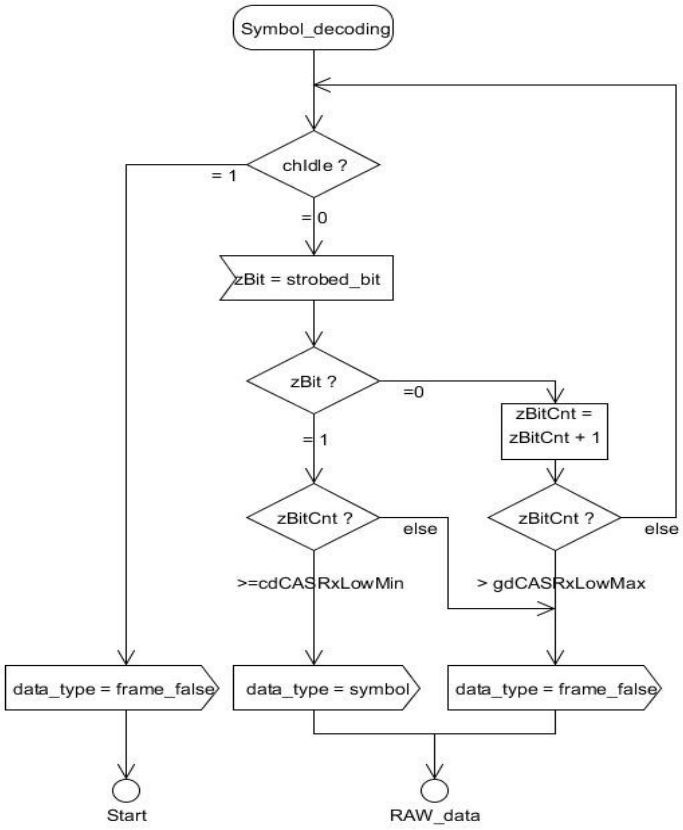
Framedecoder states



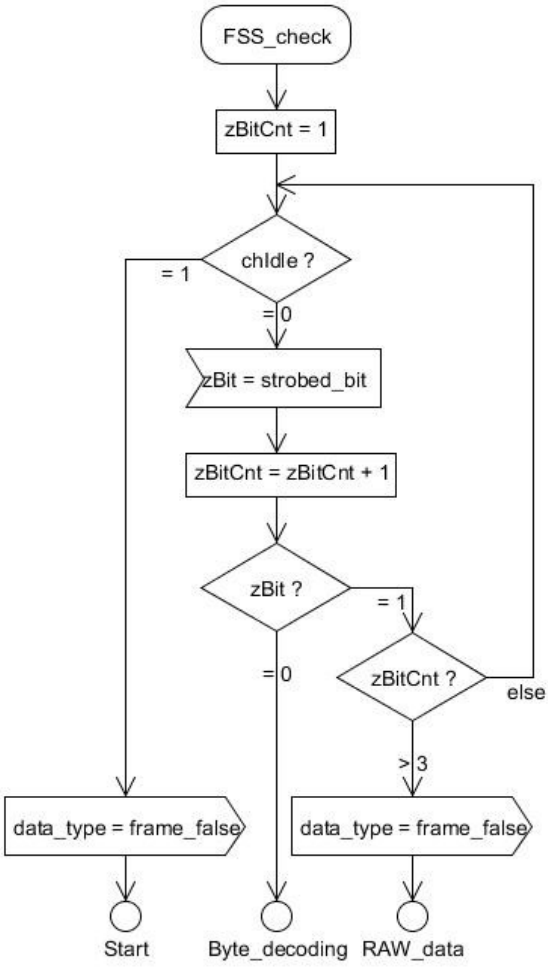
Start state of framedecoder



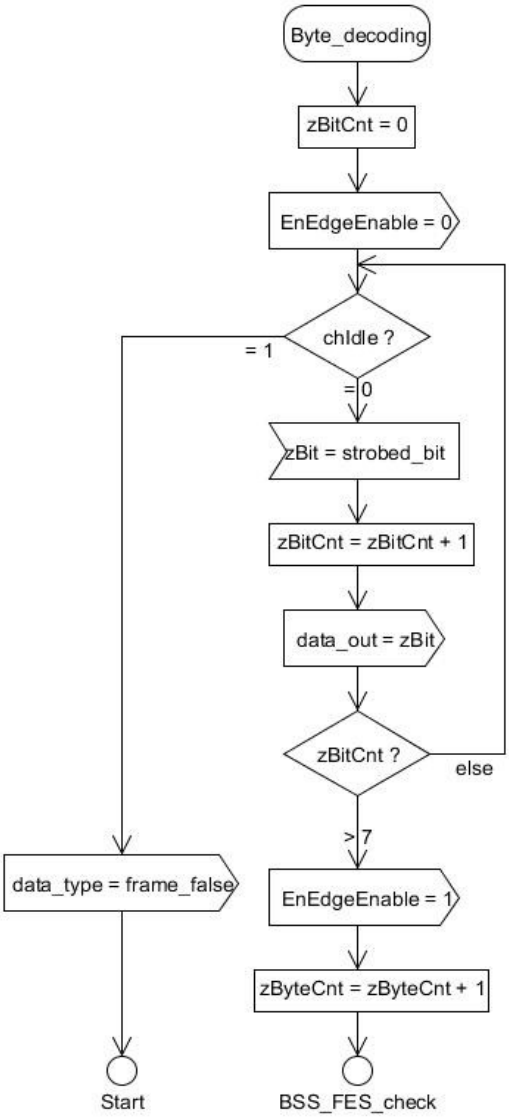
TSS_decoding state of framedecoder



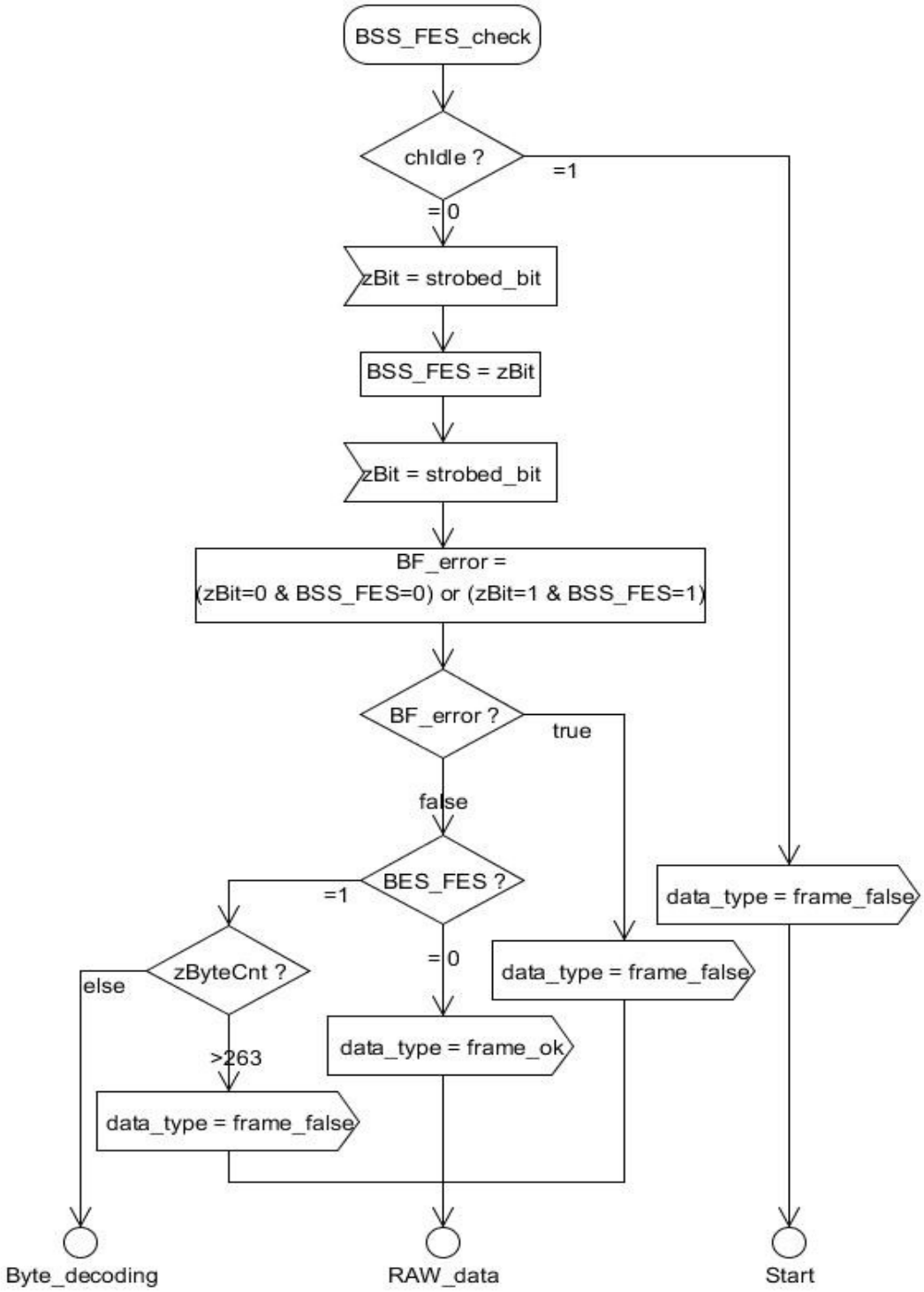
Symbol_decoding of framedecoder



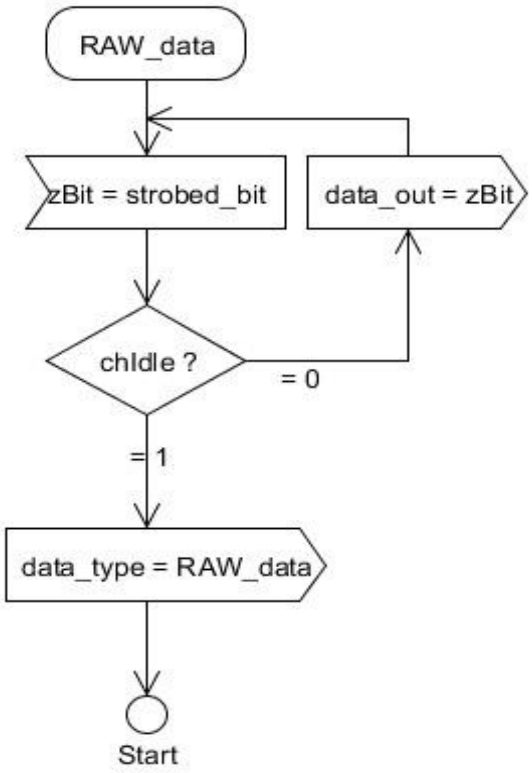
FSS_check of framedecoder



Byte_decoding of framedecoder



BSS_FES_check of framedecoder



RAW_data of framedecoder