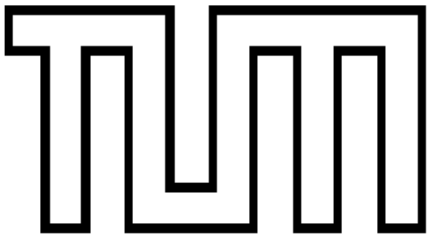

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Practical Course Report

Hardware / Software Co-Design

Marker following and car2x integration

Dharmil Shah, Matthias Grimm, Neeraj Sujan, Bashar Al-Ani

WS15

Contents

1. Introduction

1.1 Motivation

The number of electronic control units (ECUs) in cars is constantly rising. Modern cars have already more than 100 ECUs and more than 50% of the value of the car is from electronic components. This leads to a problem, since the car manufacturers are running out of space for new ECUs. Also the wiring to connect all the ECUs with all the components needs a lot of space as well. Furthermore, this adds a lot of weight, which dramatically increases the fuel consumption. One way to solve this problem is to introduce multi-core ECUs. So far the ECUs have been single-core processors. The reason for that is, that in the car there are a lot of safety-critical processes running, which require hard real-time guarantees. Proving these guarantees on multi-core systems is hard, since multi-core systems are very complex and not that well understood. The overall idea behind this project was to build a proof of concept that multi-core ECUs can in fact handle an autonomous driving car.

1.2 Problem Description

The task of this project was to build a car is that capable of building so-called cartrains and can handle Car2x communication. The idea of a cartrain is that a lot of autonomous driving cars, which have to go to the same direction drive very close to each other. Thereby, they can save fuel. In order to achieve this cartrain, the car was supposed to detect a marker (which could then be mounted to the license plate of other cars) and then follow this marker. We were building on top of previous groups work.

1.3 Approach

The car has several operation modes. These include Preoperational, Idle, Autodrive, Manudrive and Emergency Break. The relationship between these modes is illustrated in the following picture:

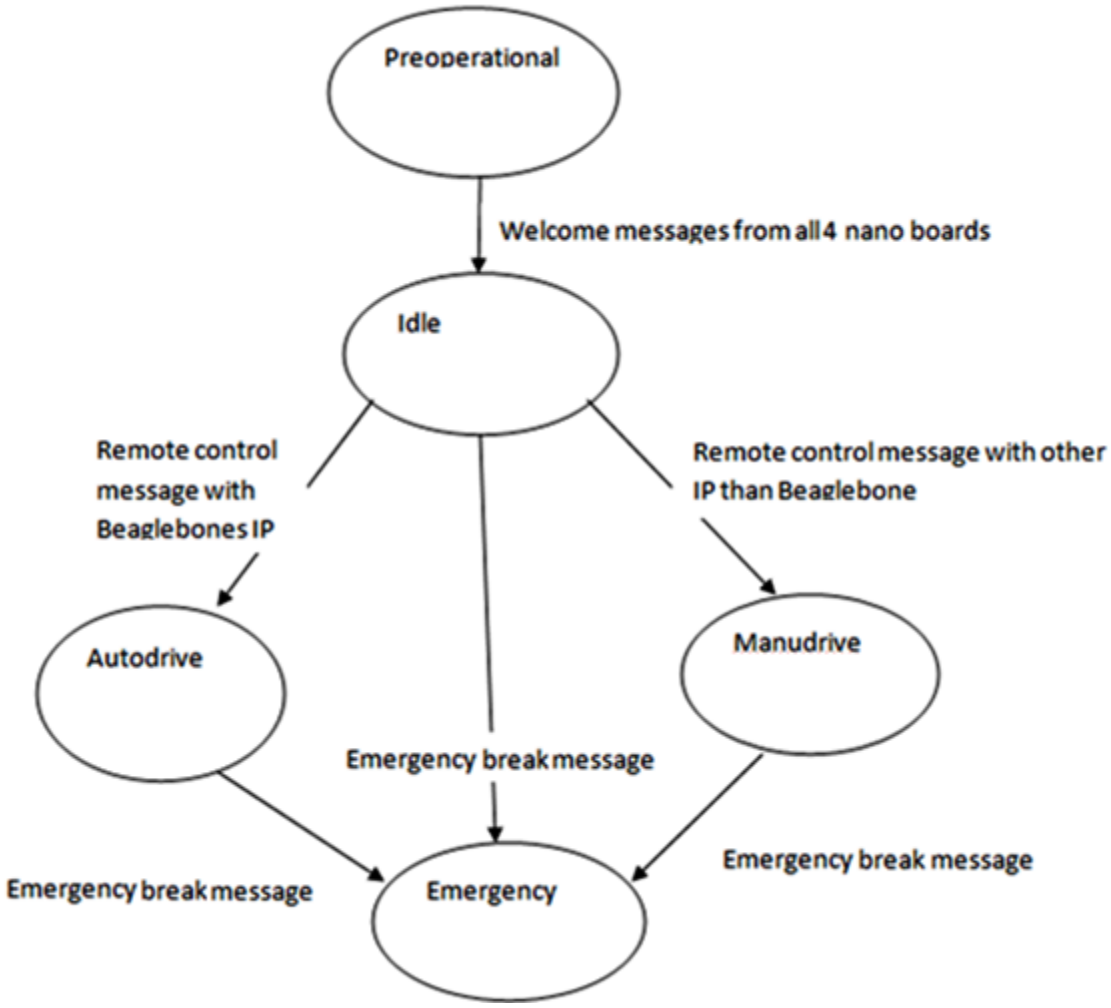

Figure 1.1 Car operating modes

The idea is that the autonomous driving component will send messages to the car, which make the car go into the Autodrive mode. If a roadside unit however sends a message, the car will go either into the Manudrive mode (if the roadside unit sends velocities to the car) or into the emergency break mode (if the roadside unit sends an emergency break message).

2. Concepts

2.1 Overview

Figure 1 shows the main conceptual components of the system. The system is intended to handle requests coming from measuring the distance to the marker and the car2x messages received from the base station (which in our case is a laptop). The system has several modes and each mode receives certain type of messages. All messages are processed by the communication core. In the system the car control unit is only responsible for switching between the modes. Both cores share a common memory region in the main memory. This region stores the data structure of the current car state.

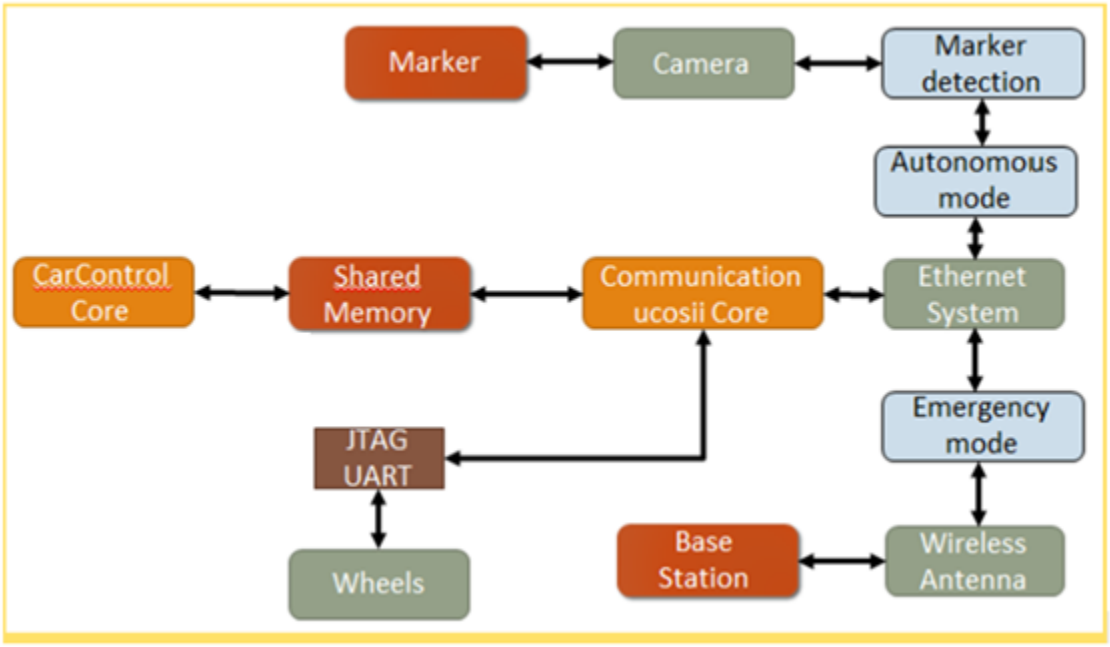

Figure 2.1: System structure

The marker detection provides a measurement of how far the marker is from the car. Then a velocity message is generated using PID control unit (both marker detection and PID control are on the beaglebone) and sent to the communication core on the main FPGA. This message is intended to make the system drive autonomously (follow the marker). Meanwhile a velocity message can also be generated on the base station (the laptop) and sent wirelessly to the wireless antenna attached to the

system. In real world this velocity message is intended to be an emergency message which makes the system stop in order to avoid accidents.

Once the message is received correctly by the communication core, the contained velocity value is sent as a message to each nano-board (the peripheral FPGAs).Each of these nano-boards implements PWM circuit in hardware and is connected to H-bridge circuit and to an external Ethernet to UART converter. Once converted, the message is processed by the motor connected to each wheel and the wheel is turned according to the received velocity value.

2.2 Car structure

The overall structure of the car can be seen in the following figure.

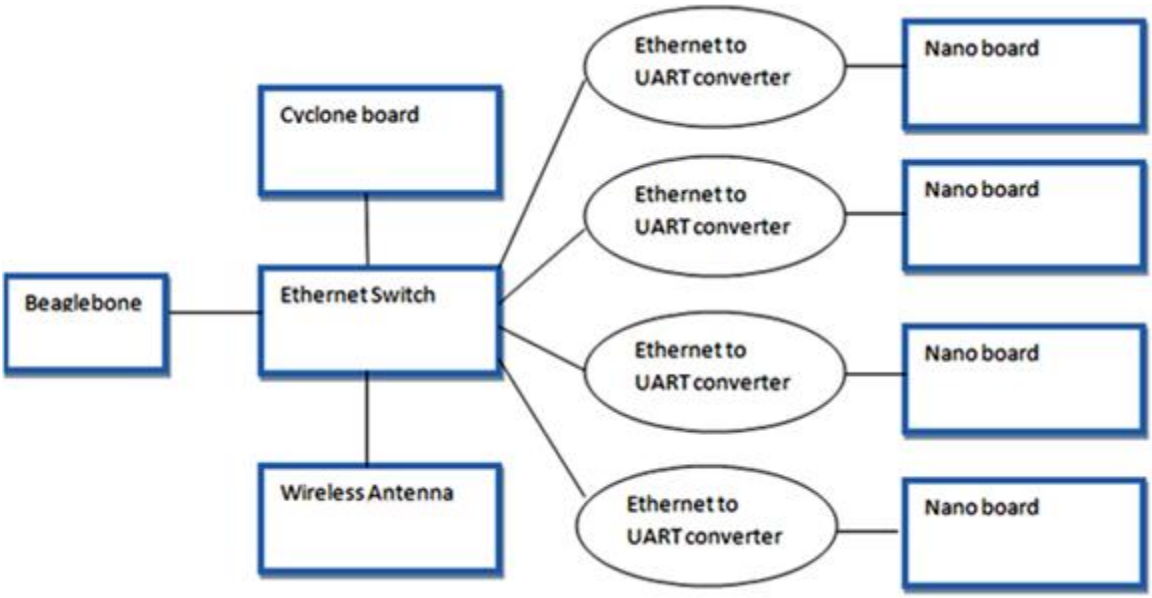

Figure 2.2 Car organization

The car consists of a Beaglebone black connected to a camera, a cyclone board, an ethernet switch, a wireless antenna, and 4 nano boards, each connected to a motor and an ethernet to UART converter.

The Beaglebone is using the camera to track the marker and then it sends the velocities which are required to follow the marker to the cyclone board. The cyclone board has 2 Nios 2 cores. One of them is doing all the communication stuff and the other one is doing control things. Hence, they are named communication core and control core. They communicate through a shared memory region. The communication core receives the message from the Beaglebone and processes it,

which might involve the control core. Afterwards, it sends the velocities the Beaglebone wants, to the nano boards. Each nano board controls exactly one wheel. The wheels of this car can not steer, therefore if the car wants to turn left or right, it has to set different speeds to the left and right sides of the wheels. This is similar to the way a tank moves around. The car can also receive wireless messages with the help of a wireless antenna. These messages could be sent from roadside units for example. All communication is done via the Ethernet switch.

2.3 CarProtocol and Car2X protocol

The CarProtocol or CARP Protocol was developed originally by the group of Florian Hisch as a communication protocol to recognise data structures in the data stream. The CARP protocol was then extended by a set of messages to ease the integration of the NIOS processor with the communication module of the car. One NIOS2 core is handles the internal state of the car and the other core is in charge of all the communication between internal modules of the car and the external communication partners like other cars or car2x stations.

The central unit now is a server to which external clients can connect to. Keeping that in mind, there now is an external communication interface featuring the so called Car2X-Messages as an extension of the CARP protocol. Each message of this type, which is sent to the car, will be answered by the car after being processed or getting outdated.

All this behavior is handled on the communication-core of the NIOS2 system. The communication core performs different steps depending on a specific message.

2.4 WiPort

This component was added to the car to allow communication from various third-party users with the car. The idea is that it broadcasts a network “HF-A11x_AP” to which other cars/users can connect to. Connecting to that wireless network results in the connecting car getting on the same local network as our car’s (192.168.0.x).

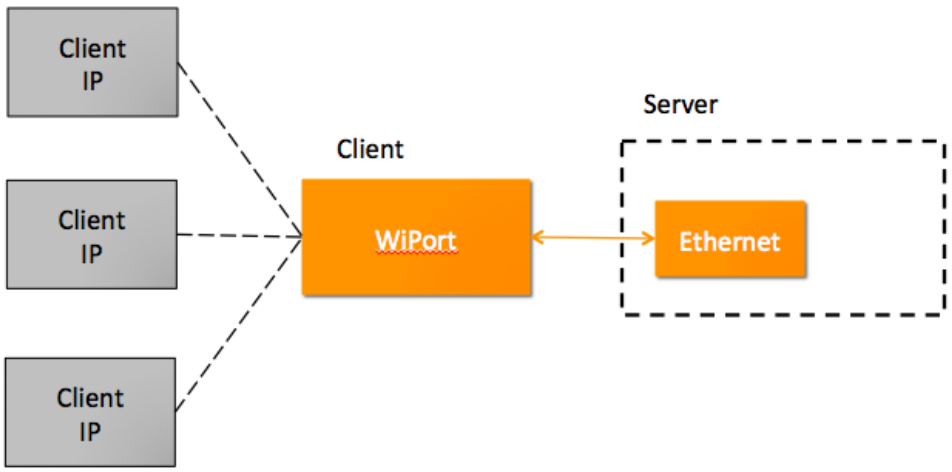

Figure 2.3 Wiport connectivity

As it can be seen from the figure, the WiPort is directly connected to our car via Ethernet cable going into the switch. Thus communication from any user with the car is possible. The dotted lines indicate a wireless connection.

2.5 BeagleBone Black

2.5.1 Basic functionality

Beaglebone is embedded board which is capable of storing Linux distribution through SD card. The current Linux distribution installed is Ubuntu 12.4 furthermore ROS hydro is installed as well. For a more detailed overview of beagleboard as well as a comparison with other Linux boards (Arduino Yun, Raspberry Pi, Intel Galileo) please refer to [1]. In general, there 5 basic ways to connect to the board:

- Access through USB, this port is used to connect to the camera used to detect the marker.
- Access through Ethernet, this is used to connect to the main FPGA board. It can also be used to connect to a laptop directly but a cross over Ethernet is needed in that case.
- Micro HDMI through adapter, this enables you to see the program on LCD display
- Add Linux image to the SD card and boot from there. In our case the Linux image is burned on the SD card and thus it is possible to boot the board directly.

- SSH via the WiPort [in our project].

Note: Win32 disk imager was used to burn the image on the SD Card. make sure you select the right destination! Writing takes some time (about 30 min).

2.5.2 Marker detection

The following are the conceptual steps for marker detection

1) Image preprocessing: The function adaptiveThreshold converts a gray image into a binary image. We use adaptive thresholding, because of changing lighting conditions.
2) Find rectangular contours: The function findContours finds contours in a binary image. In a further step we approximate a polygon and check if the found contours are rectangular and convex.
3) Check pattern: Found pattern is warped (warpPerspective) via homography to get a normalized region of interest and is checked via normalized cross correlation (NCC) against the reference pattern (identifyPatter). The reference pattern has to be defined at the beginning of the program.
4) Calculate marker position: The class "Moments" calculates the position (in pixel coordinates) of the marker in the current image. Assuming the vertical axis is fixed and the size of marker is known, it is possible to calculate the depth (z-direction) of the marker. In this case the size of the marker is defined to an edge length of 8x8 cm. This is important, because the calculation of depth depends on this known size. It is possible to use another size by changing the value markerSize in the main function car vision.

2.5.3 PID control on Beaglebone

A PID-Controller is used for controlling the steering and the velocity of the car in autonomous mode. PID is short for proportional-integral-derivative. This controller minimizes the error between the measured value and our set-points, dependent on the chosen parameters of the controller. With our two tasks (line and marker following), we need the controller, as mentioned, for the steering and the velocity of our autonomous car. In the marker following task, the car should follow the marker

while maintaining the position of the marker in the centre of the car and ensuring a constant distance.

Note: For full details on the functionality of the Beaglebone please refer to the documentation report titled “Marker Tracking and Following” of the previous team.

3. System Software

3.1 Communication core

Figure 3.1 the main cyclone IV FPGA board

3.1.1 Basic functionality

The communication core is one of the two Nios 2 cores on the cyclone board. Its task is to do communication both with external devices, like roadside units, and interior components of the car, for example the nano boards. The communication core usually has ID 0. Since this requires multi-threading, the communication core runs MicroC OS2.

3.1.2 Code

The main function of the communication does mostly microC stuff and starts the threads. One of the threads is for the Ethernet stuff and not really interesting. The files tse_my_system.c, pendingAnswers.cpp and network_utilities.c are either more or less empty and unimportant or consist of code provided by Altera or other companies. Therefore they will be ignored in this documentation. The application runs on the other threads. The application code is mostly in the socketserver.cpp file. The application starts in line 1081 of the socketserver.cpp file with the function SSSSimpleSocketServer. In order to understand this function, it is useful to know the basics of socket programming. First of all, the function uses the socket() call to create an endpoint for TCP or UDP communication. The corresponding file descriptor for the socket is stored in fd_listen. Second, the socket gets binded to a

port. Then the socket starts to listen via the listen() command. Now (line 1144) all socket stuff is done. The socket was successfully started and is listening for connection and now the initialization of the car starts.

After the socket initialization the car performs the mode change from Preoperational mode to Idle mode. That means, the car must get into a state where its ready to drive. Therefore at first the memory controller for accessing the shared memory is initialized. Afterwards the shared memory region is filled with a carstate and the carstate is initialized with zeros. The ifterminator Boolean in line 1148 is mainly for debugging purposes. It ensures a double check that the initialization sequence is not performed twice. Theoretically, it could be removed.

Afterwards, an infinite while loop starts. In this loop first the select call is done with the socket as argument. This call waits until someone tries to connect or until there is data to read. Next, the code checks whether there is a new emergency message, a new remote control message or a new control message. If that is the case, the current action is executed (e.g. if it is an emergency break message, the wheel speed is set to zero and the emergency break mode is entered), and then sends an answer to whoever sent the message. The test, whether the message is up to date is done via the stateVersion <= state.counterCarControl condition. The counterCarControl variable holds the ID of the last packet the control core processed. IDs are assumed to be incremental. If the message is not up to date, it is not processed. This logic continues until line 1563.

Now the original initialization process starts. Therefore it is tested whether the socket is set, that means whether there is a connection request. If this is the case, a new socket for this connection is created and stored in a socket list. If this fails and then the socket is set because there is incoming data. In this case a function (sss_handle_receive_new) is called to receive the data.

Afterwards (line 1601) it is tested whether the car is still in preoperational mode and whether all four nanoboards already requested the connection. If this is the case, the car sends some messages to the nanoboards. That will make the nanoboards turn forward and backward. If all messages are sent correctly, the communication core requests the idle mode. This is the end of the while(1).

Now since the initialization is done, the communication core is ready to receive messages from the outside world. That is done by the previously mentioned

sss_handle_receive_new() function, which starts in line 977. This function parses the byte stream incoming in the socket and checks whether there is a valid message according to our protocol. If this is the case, it checks which type of message there is and stores the bytes accordingly. Afterwards, this function calls the sss_exec_command function with the packet, which just arrived.

The sss_exec_command function starts in line 177. It checks whether the packet which was received was valid. If that is the case, a for loop starts which iterates over all messages of the packet. At the beginning of the loop, the carstate is accessed through the memory controller, which will block all accesses from the control core. Next, if the current or requested mode is not the preoperational mode (that means the car is in a mode where it can actually drive), a switch statement starts which determines the type of message. This switch statement goes on until line 873. In this statement it is basically determined what type the current message has and then the actions corresponding to this type of message are done.

Recall that this is only done if the current mode is not preoperational. If the current mode is preoperational, then the else statement in line 875 is executed. In this mode, only welcome messages from the components of the car are interesting. Therefore it is checked whether one of these messages arrived and then the corresponding component is registered. This registration is important when the cyclone board wants to communicate with the corresponding component.

3.2 The control core

3.2.1 Basic functionality

The control core is supposed to control the car, that means it authorizes mode changes, velocity changes and so on. It communicates with the communication core via a shared data structure, the carstate. The usual control flow is that the communication core requests something and then the control core authorizes this request.

3.2.2 Code

In the code, first the memory controller for accessing the shared memory is initialized. Afterwards, a carstate is put into the shared memory and filled with zeros. Next, the control core enters an infinite while loop. At the beginning of this loop it waits. This is done because the Memory controller uses a hardware mutex. This mutex has no fairness. It can be seen as a first come, first get lock. The waiting wants to stop the control core, so the communication core gets a chance to acquire the mutex as well. After the waiting is done, the control core accesses carstate in the shared memory region through the controller. This will get him the mutex and block all other accesses from the communication core. Afterwards, the control core checks whether the current mode equals the requested mode. If you want to know more about the modes, please refer to the section about the communication core. If this is not the case,it performs a mode switch. That means, it will set the maximum speed to the maximum speed of the requested mode and then set the current mode to the desired mode.

Afterwards the counterCarControl is set to the counterCarComm. Thereby, the communication core can see, that the control core has catched up to it. After that, the control core calls the set speed function, which will check whether the average of the four requested velocities for the four wheels exceeds the maximum velocity. If this is not the case, the control core will authorize the new velocity, by setting the desired velocities for each wheel as velocities in the motorEcus[] array. The last thing the control core does in one iteration is updating the IP of the device that currently controls the car, if requested. It does that by assigning reqip1 to ip1, reqip2 to ip2, and so on. Then it pushes the carstate onto the shared memory again. This will unblock the mutex. After that, the iteration begins again.

3.3 The nano boards

3.3.1 Code

The code starts in the main.cpp. First of all the init() function is called, which sets the speed to zero (in case the motors were running before starting the nano board due to an error or due to a sudden restart). Then it opens a socket. Afterwards main calls

the sendWelcome() function. This function mainly consists of a while(1) loop. In this loop, the nano board first sends a welcome message to the cyclone board. After that, it checks whether it received a correct answer from the cyclone board. If it did not receive any answer at all, or if the received message is not correct, then the loop starts again. Otherwise, the nano board returns true. The code was altered in a way that the nano board only sends a welcome message every 50th iteration (line 166) but checks for an answer every iteration. Otherwise, the nano boards would simply flood the cyclone board with messages way too fast.

Figure 3.2 The nano board FPGA

The next step which main does is calling the setUpPIController() function. This function will make the corresponding wheel of the nano board go forward and then backward very fast. This should be considered when starting the car, since it will cause the car to move. I suggest to always place the car on top of a box, so that the wheels do not have contact with the ground. After making the motor go forward the function reads out the wheel encoder. It does the same after making the wheel turn backward. Thereby, all nano boards can have the same effective maximum speed in both directions despite the sensors not being perfectly aligned. However for debugging purposes the maximum speeds in both directions are currently always hard coded +500 and -500. After that a PI controller (defined in pidcontroller.cpp) is constructed. This is a classic PI controller. Afterwards, the function just waits from another message from the cyclone board. Then the function ends.

Afterwards, the main function reaches an infinite while loop. At the start of this loop the nano board first waits for a new packet. This is done via the function waitForNextPacket(). Afterwards the controlSpeed() function is called. Using the current (coming from the wheel encoder) and the desired speed (coming from the latest packet), the function calls the controller to calculate the next speed value. This value is then added as an offset to the current speed, and the resulting speed is set to the motor. Afterwards the function shifts to the next message in the packet. In

order to understand this measure, recall that a packet can consists of several message. However, in practice packets only consist of one message. Afterwards the next iteration of the while loop starts.

It should also be noted that a packet is called a protocol in the code. This nomenclature was introduced by previous groups. We did not change it in order to stay consistent with their documentation.

The code of Car2X_nanoMotorCtrlRechts is mostly the same as the code as the code of Car2X_nanoMotorCtrl. The only difference is that whenever the speed is measured, it is multiplied by -1. That is necessary, because the motors on the right side are actually turned 180 degrees with respect to the once on the left side. So when the motor on the left side and on the right side do the same movement, it would cause the left side to move forward, but the right side to move backwards. There are no other changes between Car2X_nanoMotorCtrlRechts and Car2X_nanoMotorCtrl.

3.4 The CarProtocol

This protocol was developed initially by the group of Florian Hisch and was later extended by the group of Hagen Schmidtchen. As such, there are some changes that were made in functionality by the later group. The group of Florian developed their own communication protocol named CarProtocol to recognize data structures in the telnet stream. CarProtocol is organized as packet of messages. The total structure is shown in the below figure.

As it can be seen in the above figure, the different packet fields are:

- ‘C A R P’: Start sequence of the packet to mark it is a CARP message.
- PacketNumber: Consecutive increasing 16 bit number which works as an ID. The number is never changed by the Nios2, only by the central ECU (Linux-PC). Thereby the central ECU can detect a protocol fail if the response packet does not have the same PacketNumber as the request packet. It needs to wrap around (go to zero) after the PacketNumber 65535!
- PayloadLength: Total length of the payload in Bytes. This length information contains not the packet-header length!

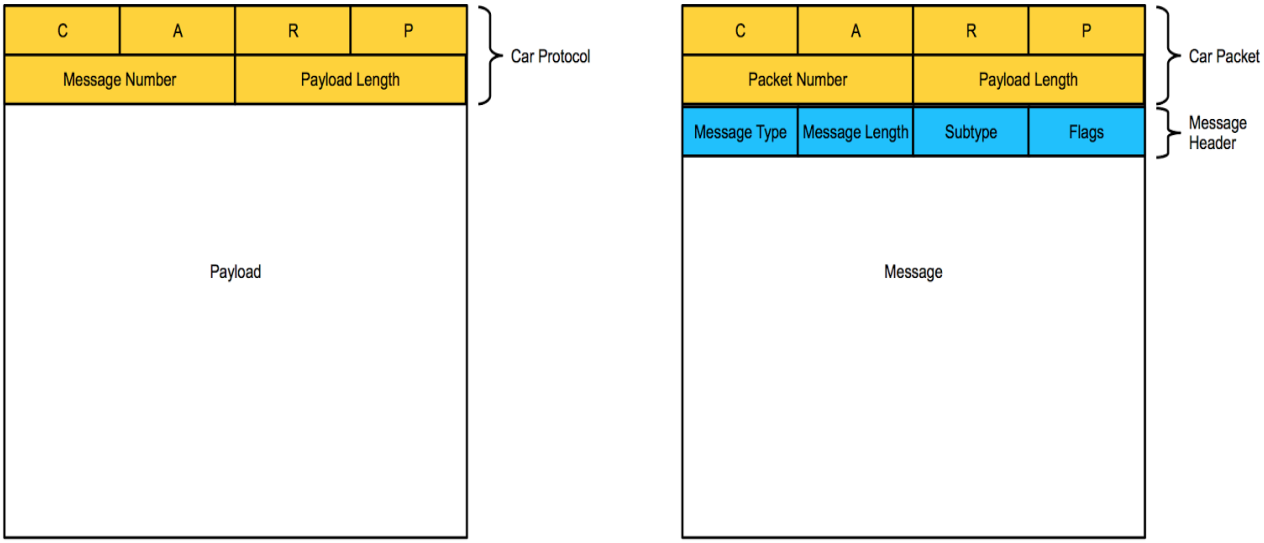

Figure 3.3 Car Protocol structure

The payload contains at least one message and max. 8 messages. All messages start at a multiple of 4 assuming that all messages have a multiple of 4 as length. There must be no gap between two messages!
The order of the messages in a packet is quite important:
If a packet contains a WelcomeMessage, this message needs to be and will be interpreted first. All other messages in this packet will be ignored! Otherwise the first message is the most important followed by the second and so on.
Note: As the velocity-message is the most important in normal mode it will be assumed to be the first message!

CarMessage - Requests and answers:

A CarMessage is the atomic communication part. The central ECU writes some (request) messages and packs them into a CarProtocol packet. The packet is transferred and read by the Nios2 processor. Every message is handled in one of the cycle runs. Handled means that the message activates a sensor. This sensor fills the empty fields of the message. We called this filled message answer although it looks exactly like the request.
Every sensor understands at least one message type. So we send only those messages to a specific FPGA which can be understand by a connected sensor. For

example: Only one FPGA has an A/D-Converter so only this FPGA gets a packet with an ADCMessage.

A message consists of a message-header and a message-body. The message-header is implemented in a superclass. Due to different body-types the body must be implemented in different subclasses.

The Structure of the message header is shown is before and described as:

- Type is the ID of the message class. The type follows this rule:
  - Types between 0 and 3 are for NETWORKING (such as WelcomeMessage)
  - Types between 4 and 7 are BASIC messages and have to be understood by every networking client
  - Types between 8 and 255 are GENERAL purpose messages
- Length = Length of header (always 4 bytes) + Length of PAYLOAD The Length of request and answer should always be identical!
- SubType: If a sensor requires more than one message type, SubType distinguishes between these messages. Is there just one message type this field should be 0.
- Flags: Bits are numbered from 0 (least significant bit) to 7: Bit 0: Request(0) / Answer(1)

One example of a CarMessage is the WelcomeMessage. This message has two aims:

1) Synchronize the 4 Motor-ECU with the central-ECU and
2) inform the central-ECU about the messages which can be understood by this Motor-ECU.

The central-ECU sends (at the same time) a WelcomeMessage to each Motor-ECU. The Motor-ECUs answer with the same message but additional with a list of those messages which are available at this ECU. The list is marked with Operation 1, Operation 2 and so on. The message has the structure given in Figure 3.4. Only those MessageTypes are necessary to list in the WelcomeMessage with a type >= 8. The list has to be filled up with 0x00 if less than 4 additional messages are understood.

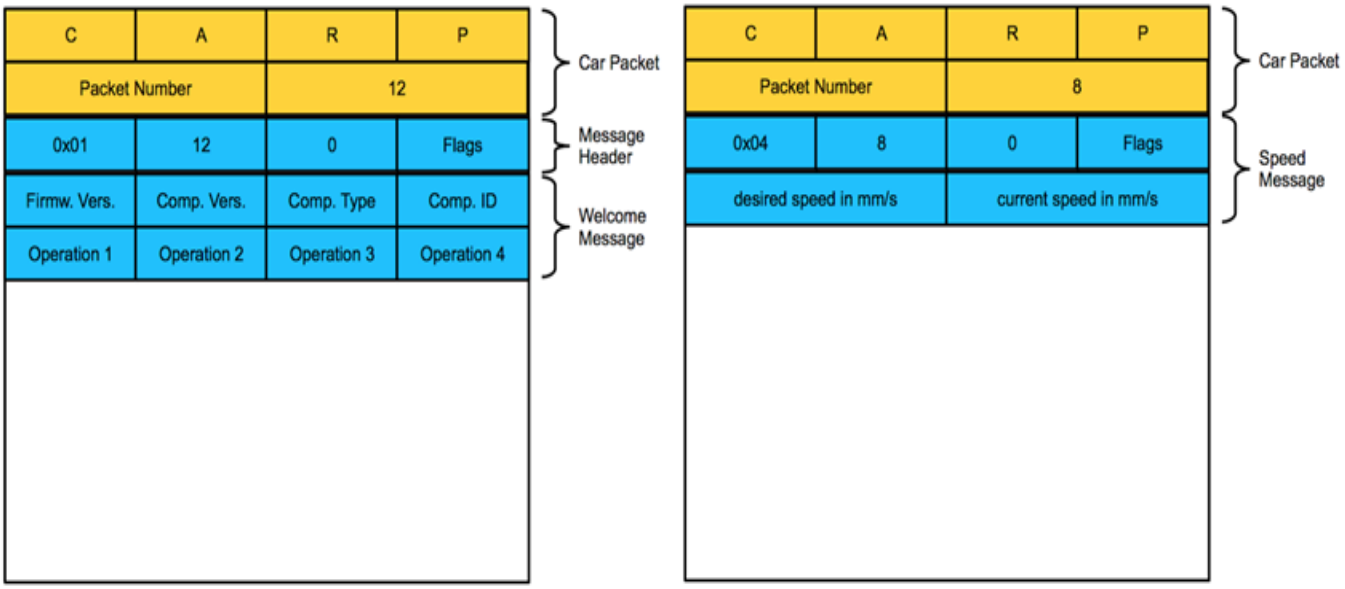

Figure 3.4 Welcome message (left), CarVelocity message (right)

3.5 The C2X extensions

The “CARP Protocol” was later extended to include a new set of messages to allow the communication between the central communication core and the nano boards by the group of Hagen Schmidtchen. These messages are known as the “Car2X Messages”. Now, one NIOS2 cyclone core handles the internal state of the car and the other one is in charge of all the communication between the nanoboards and external communication partners like other cars or “car2x” stations. All the internal communication between the nanoboards and the central board is based on the work of Florian Hisch and had not been modified. The only change was a role switch in terms of server-client relationship. The central unit now is a server, to which the external clients can connect.

Keeping that in mind, there is an external communication interface featuring the following messages as an extension of the “CARP protocol”. Each message is answered by the central core after being processed or outdated. While parsing the received messages in the “socketserver.cpp” file from TCP/IP, a new message object created for every new message.

Directly after that, the “sss_exec_command()” function handles the received message by checking the message type and then taking various steps depending on the specific message.

In case of simple “polling messages” that don’t interact with the car state, like the “CInfoStateMessage” and the “CInfoSensorMessage” which do only read some information out of the current state, the response is immediately created and sent, containing the required data.

The other three messages like the CControlMessage, EmergencyBreakMessage have an influence on the car state and therefore have to be queued until the Car State gets updated by the control core. Once the Car State has been updated, the main loop of the socketserver checks if the requested state change like for example an emergency brake process has been performed or not, and then deletes the message from the queue and sends an answer message. Depending on this check the produced answer message contains one of the three following flags:

- “A” for successful execution
- “F” for failed execution
- “O” for the message being outdated

The main challenge and reason for this check is the fact that communication and state control are performed by different NIOS2 cores. While the state is being updated periodically, incoming messages might arrive more frequently. If for example 3 “CControlMessages”, which contain the information to set the motors to a specific speed, are received within one state update cycle, it is obvious that only the last one should be executed in the new state and the older ones get outdated as soon as a new message of the same type is received. Whenever a message gets queued, and there is already a message in the queue, the outdated one gets immediately answered with an “O” flag. It is outdated and doesn’t have to be executed any more, since it has been overwritten by the latest message, resulting in only one queued message at maximum for every message type at every state update. For a more detailed description of the queueing process see next chapter.

In the following part all types of CAR2X messages are explained using examples. Note that there is always the protocol header included. For answer messages the

protocol header consists of CARP, “ControlCoreCounter”, “CommCoreCounter” payloadLength, success-flag and message type.

CEmergencyBrakeMessage

Example packet sent to the car:

Byte	1	2	3	4	5+6	7+8	9	10	11	12
Content	C	A	R	P	PackedID	PayloadLength: 0x04	Type: 0x20	Length: 0x04	Subtype:0x00	Flags: 0x00

After receiving this message, the control core is requested to change the car state into emergency braking and this message is queued to wait for the next state update, or being outdated. The payload of the answer consists of the PacketNumber of the received message.

Example answer message sent by the car:

Byte	1	2	3	4	5+8	9-12	13-16	17	18	19+20
Content	C	A	R	P	ControlCoreCounter	CommCorecounter	Payload Length	Success flag (A;F;O)	Type: 0x20	PackedID of the received message

CControlMessage:

This message allows to control all the four wheels of the car and in theory allows passing speeds between -32768 and +32768.

V1 -> Speed for the left front wheel

V2 -> Speed for the left rear wheel

V3 -> Speed for the right front wheel

V4 -> Speed for the right rear wheel

Part 1 of the speed (13, 15, 17 and 19) is the multiplication factor and Part 2 of the speed is the addition factor. Please keep it in mind it uses 2’s complement for the negative values.

Speed1 = V1[1] * 256 + V1[2]

Example packet sent to the car:

Byte	1	2	3	4	5+6	7+8	9	10	11	12	13+14	15+16	17+18	19+20
Content	C	A	R	P	PacketID	Payload Length : 0x0c	Type:0x30	Length:0x0C	Sub type : 0x0	Flags:0x0	V1	V2	V3	V4

After receiving this message, the control core is requested to set the specified motor velocities (V1...4 as 16-bit values), specified in the payload. It has to be mentioned that in case the sender of this message is not registered as the current source of control, the message is not queued but immediately replied as failed because the sender is not allowed to control the car. But if the control is allowed, this message is queued to wait for the next state update, or getting out of date.

Example answer message sent by the car

Byte	1	2	3	4	5-8	9-12	13-16	17	18	19+20	21+22	23+24	25+26	27+28
Content	C	A	R	P	Control Core Counter	CommCore Counter	PayloadLength	Success flag (A,F, O)	Type: 0x30	PacketID of the received message	V1	V2	V3	V4

V1 to V4 are now standing for the current desired motor speed values delivered to the specific PWM motor controllers. In case of a successful message, they are equal to the requested values of the message, sent to the car, otherwise they differ and the message is answered as “failed”, giving the controlling unit feedback about the current state of the car, to let it know what might have failed

CRemoteControlMessage

Example packet sent to the car

Byte	1	2	3	4	5+6	7+8	9	10	11	12	13	14	15	16
Content	C	A	R	P	PacketID	PayloadLength:0x60	Type:0x08	Length: 0x08	Subtype :0x0	Flags: 0x0	IP 1	IP 2	IP 3	IP 4

The payload of this message contains the IP of a source which should be allowed to control the car by using “CControlMessages”. Per default this source is the unit inside of the car featuring the ImageProcessingUnit with its IP 192.168.0.110. In the standard state of the car “AutoDrive”, the car is controlled from this IP by “CControlMessages”. By sending a “CRemoteControlMessage” to the car containing a different IP, the car is requested to set its state to “ManualDrive” with the new source IP locked for “CControlMessages”. Sending 0.0.0.0 as new IP will set the car back to “AutoDrive”, using the ImageProcessing IP again. As before, the answer gets queued until the state is updated for the next time.

Example answer message sent by the car:

Byte	1	2	3	4	5+8	9-12	13-16	17	18	19+20	21+22	23+24	25+26	27+28
Content	C	A	R	P	Control Core Counter	CommCoreCounter	Payload Length	Success flag (A,F, O)	Type :0x60	Packetld of the received message	IP1	IP2	IP3	IP4

IP1 to IP4 are the current state values of the locked IP. In case of success they equal the requested one, otherwise they give the feedback about who is currently controlling the car.

CInfoStateMessage

Example packet sent to the car:

Byte	1	2	3	4	5+6	7+8	9	10	11	12	13	14	15	16
Content	C	A	R	P	Packet ID	PayloadLength. =0x04	Type: 0x40	Length:=x04	Subtype: 0x0	Flags:=x0				

This message just polls the current state information from the car. It is answered immediately and thus has not to be queued. The answer always contains the “A” flag for success and features the biggest part of the state object from the shared memory as its payload. It can be used for debugging purposes or by the ImageProcessing unit or a station as additional odometry feedback

Example answer message sent by the car:

Byte	Content
1	C
2	A
3	R
4	P
5-8	ControlCoreCounter
9-12	CommCoreCounter
13-16	PayloadLength
17	Success flag (A,F,O)
18	Type: 0x40
19+20	PacketID of the received message
21-24	iMaxSpeed
25	IP1
26	IP2
27	IP3
28	IP4
29	reqIP1
30	reqIP2
31	reqIP3
32	reqIP4
33	currMode
34	reqMode
35-36	1st ECU state
57-78	2nd ECU state

As “ControlCoreCounter” and “CommCoreCounter” are already part of the answer header while also being part of the car-state, they are excluded from the payload. Its the same for the sensor values which are polled by the “CInfoSensorMessage” seperately to save bandwidth.

CInfoSensorMessage

Example packet sent to the car:

Byte	1	2	3	4	5+6	7+8	9	10	11	12	13	14	15	16
Content	C	A	R	P	PacketID	PayloadLength : 0x04	Type : 0x50	Length : 0x04	Subtype:0 x0	Flags: 0x0S				

This message just polls the current sensor information from the car state. It is answered immediately and thus has not to be queued. The answer always contains the ” A“flag for success and features the whole sensor information which are part of the car state. This message can be used to access the sensor data. Note that currently there is no sensor hardware connected and there is still no implementation to get the sensor information from this missing hardware. On the other hand, the state object already contains memory to store this data which is read out by one of these messages. The polling interface by CAR2X messages is completely working for an assumed amount of 2 sensors but will always deliver no information until the internal sensor read out will be implemented.

Example answer message sent by the car:

Byte	1	2	3	4	5-8	9-12	13-16	17	18	19+20	21-24	25-28
Content	C	A	R	P	ControlCore Counter	Comm CoreCounter	Payload Length	Success flag (A,F,O)	Type : =x50	PacketID of the received message	Sensor 1 Value	Sensor 2 Value.

3.6 BeagleBone Black

3.6.1 Code

The Following constructs are important to run "servoCommand":

- - servoCommand.cpp, servoCommand.h: inherited from "servoControl".

- - servoControl.cpp, servoControl.h: lowest level and communicates directly with motors (set PWM signals). If the hardware changes, this class has to be adapted.
- - PWM-values of the car are initialized in the constructor of the servoControl class (these parameters have to be changed if another car is used)
- - PWM8_13_PATH, PWM9_14_PATH: File path to set pwm signal of the Beaglebone black ("/sys/devices/ocp.3/pwm_test_P8_13.11/")

3.6.2 Integration with Car2x

The velocities generated based on the distance to the marker need to take in consideration the physical capabilities of the car model, features such fixed wheels versus rotating wheels affect the parameters that need to be sent to the main FPGA board which in turn are transmitted to the nano boards. In our system the velocities are generated according to the following formula:

Right Wheels Velocity =
(Car_max_speed * distance / max_Dist) * (1 + Model_width/(2*radius))
Left Wheels Velocity =
(Car_max_speed * distance / max_Dist) * (1 + Model_width/(2*radius))
Where:
Car_max_speed: max speed of the car, currently set to 0x7F (or 127)
distance: current distance to the marker
maxDist: max distance to the marker, currently set to 25. (the value affected by the camera resolution and the max number of frames that are allowed to be processed)
Model_width: the physical width of the car. currently set to 20 cm
radius: represents the angle of deviation from the center of the marker and is calculated using the following formula.
radius = Model_length/(cos(90 - degree))
Where:
Model_length: the physical length of the car. currently set to 30 cm
degree: The steering degree from the servo

4. Initializing, operating and debugging the car

4.1 General overview

The following assumptions are made to start the system:

- Quartus II sp1 web edition is installed on your laptop (we used windows version, so did the previous teams).
- A power source is available either using an external power supply set to 11 volts or using charged batteries attached to the system
- All the components are turned on, the main FPGA, the four nano boards, the beaglebone, the camera, the wireless antenna. You should be able to see the LEDs of each component working.

Figure 4.1 - The car is booted up

In order to start the system, the following steps are needed:

- Open the project code using quartus, as seen in figure 4.2 you should see the project title labelled 1.
- The nano-boards should be initialized and this is done by connecting each board to the laptop you are working with through mini USB to USB cable. An alternative would be to flash the board thus whenever they are powered up

the code is already uploaded to them. In order to do so press programmer icon - labeled 2 in figure 4.1. The programmer window will pop up. Press the hardware setup button and choose the usb cable. Then press the start button and close the programmer window.

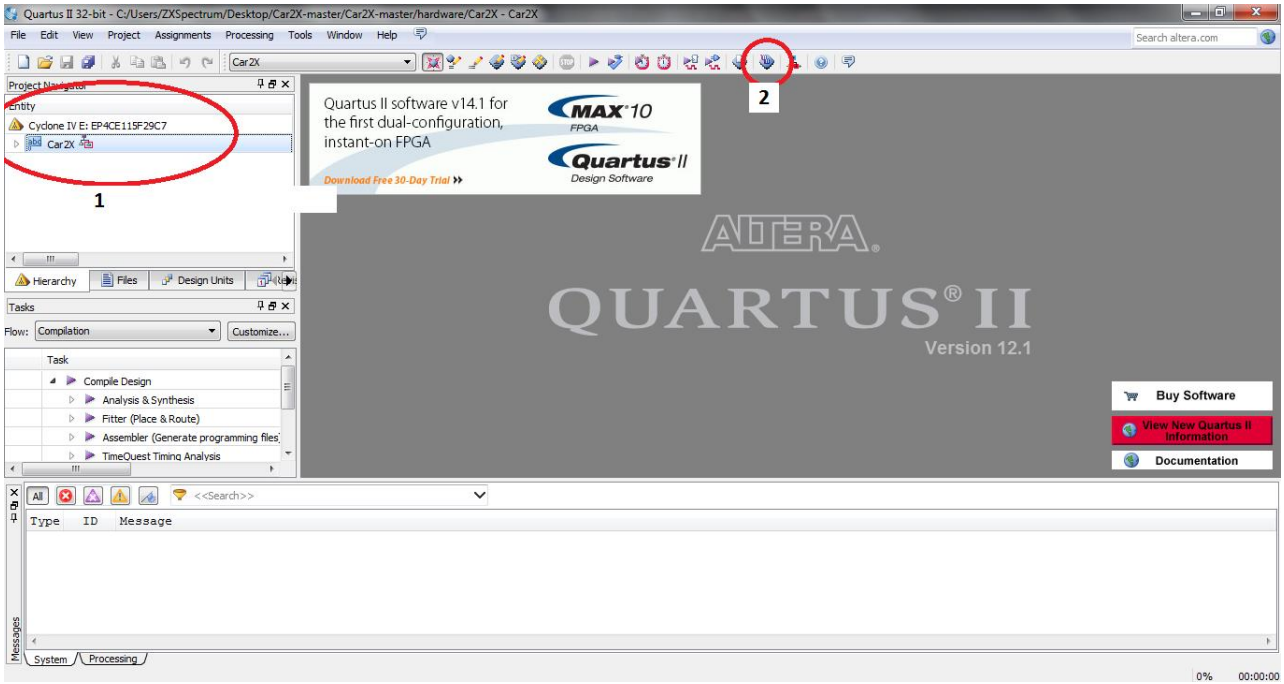

Figure 4.2 Quartus II GUI

- Next, initialize the main FPGA, this means uploading the code to both communication core and control core, this can also be done in any order. Although we used to start control core first then communication core. Start nios II built tools for eclipse and run Car2X_carcontrol and Car2X_carcommunication.
- at this point the car mode should change from pre-operational mode (0,0) to idle mode (1,1), this can be seen as each wheel will rotate forward and backward once.
- Next, log into the beaglebone black and start running roscore node first and then carvision node (your application). In order to log to beaglebone two options are available, the first to connect to the board using a laptop through a crossover cable via the ssh terminal. Another option is to connect to the board through the wireless network.

- Independently connect to the wireless network using a laptop; this enables you to send the wireless emergency message to the communication core. This can be done either using python script or using java GUI.
- Once the beaglebone is running you can connect it back to the main FPGA and the velocity message will be received by the communication core.
- Depending on the distance and orientation from the marker the wheels will start rotating accordingly. While the car is moving it possible to stop the system completely by sending an emergency message using the laptop.

Figure 4.3 The Beaglebone connected to the webcam (via USB) and the main FPGA (via ethernet)

4.2 Starting the nano boards

In order to start one of the nano boards, connect the nano boards via USB Blaster to your computer. If the computer has trouble detecting the USB Blaster cable, it could be, because one of the USB Blaster cables in the lab is broken. In this case just use another cable. After that, start the Nios 2 Software Build Tools for Eclipse. Now it depends whether the wheel, which should be started is on the right or on the left side. If it is unclear which side is left and right, just remember that the Ultrasound sensor points in the front of the car.

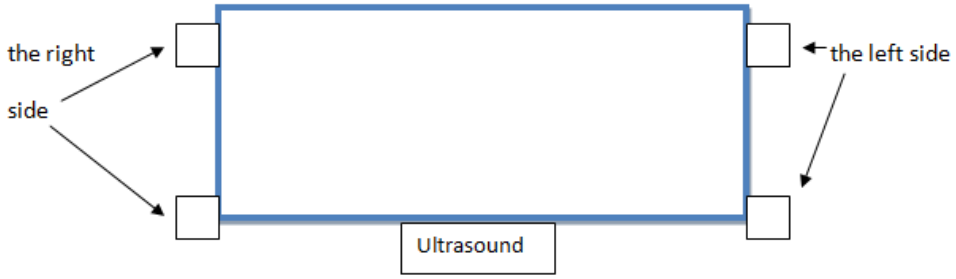

Figure 4.4 The right wheels and the left wheels (Ultrasound is in the front)

If one of the nano boards from the left side should be started, use Car2X_nanoMotorCtrl. In the project explorer of Eclipse (that's the list on the left hand side) do a right click on Car2X_nanoMotorCtrl and select "run as" and then "Nios 2 Hardware". This will start the nano board. Once the nano board is started, some of the LEDs will start to shine. Then you can unplug the USB Blaster. If the wheel which should be started is on the right side, do the exact same thing but use Car2X_nanoMotorCtrlRechts instead of Car2X_nanoMotorCtrl.

4.3 IPs and hostnames

The following are IP addresses we used. The port number used is 23.

Table 4.1 The IP addresses used in the project

Cyclone board	192.168.0.200
Wheel left front	192.168.0.12
Wheel left rear	192.168.0.14
Wheel right front	192.168.0.11
Wheel right rear	192.168.0.13
Wireless antenna	192.168.0.21
Beagle bone black	192.168.0.150

4.4 Communicating with Car2x from BeagleBone

In order to start Beaglebone. Type in terminal:

ssh root@192.168.0.150 with the password root [Preferred]

or

ssh ubuntu@192.168.0.150 with password temppwd

In order to clean and build make sure to always run in the carvision node directory

rm -rf build/ && catkin_make clean && catkin_make

To run the system run roscore in the background (so that when you replace the cable the system keeps running), to do so type roscore & in the terminal

To run the rosvision node

rosrun car_vision car_vision_node

4.5 Communicating with Car2x from Laptop via WiPort

As mentioned earlier, the WiPort broadcasts a wireless network access point “HF-A11x_AP” to which the users should connect. Once you connect to that wireless network, you’re on the same local network as the car’s and your IP should be something like 192.168.0.100 or 192.168.0.101 and so on. With this, in theory, you can communicate with the car directly by using something like telnet and connecting to Cyclone’s board IP (192.168.0.200) on port 23. This is exactly what the GUI tool and Python script do and, in general, just make the life easier by providing a degree of automation.

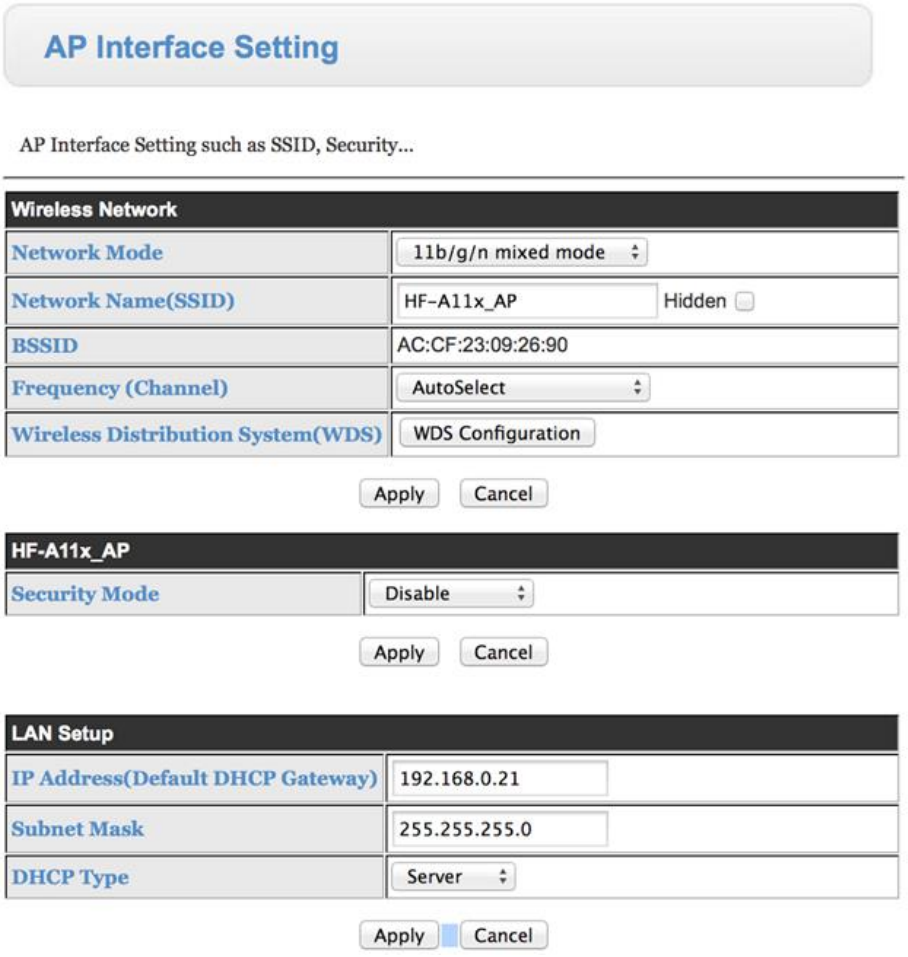

Figure 4.5 WiPort configuration window

4.5.1 Configuring the WiPort

The official device model of the WiPort is USR-WIFI232-610. You can find the documentation for it here: http://www.tcp232.net/download/USR-WIFI232-610_en.pdf You can configure the page by going to the 192.168.0.21 in the browser and using admin/admin as the username and the password. The only setting that needs to be changed from the default configuration is the IP Address in the LAN Setup section of the AP Interface Settings page. Below is a screenshot of the settings.

4.5.2 GUI tool

This GUI tool was written in Java by the group of Hagen Schmidtchen and allows the remote control of the car from any device connected to the wireless network of the WiPort.

You can find it in the GUI folder of hwswcd-car2x project/folder.

You can just run the JAR file of the GUI by going to this folder first:

cd hwswcd-car2x/GUI/Car2xStation/Car2xStation

java -jar "dist/Car2xStation.jar"

In order to use the GUI, you have to first click on the Connect button and wait for the GUI to be connected to the cyclone board.

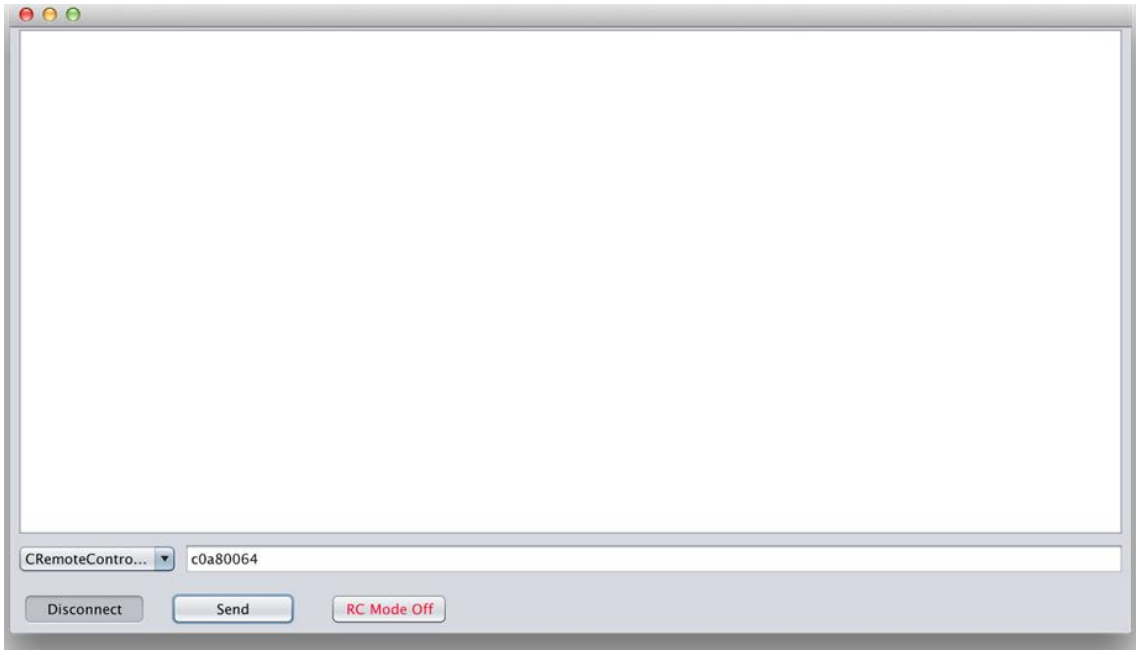

Figure 4.6 Accessing the car through Laptop GUI

Once it is pressed, you have to initially authenticate yourself using a CRemoteControlMessage from the drop down. In the text box, you need to enter the IP of your host machine in hexadecimal. For example:
If your IP is 192.168.0.100, you need to enter c0a80064 in the textbox without any spaces. Basically, you convert each part of the IP to hex: 192 -> c0, 168 -> a8, 0 -> 00, 100 -> 64.
Once you have entered it, you can press send and the message will be sent to the car and it will transfer authentication to your machine.
To control the car wheels/speeds, you need to send a CControlMessage and select it from the drop down.
As before, the speeds need to be converted to hex before the message can be sent:
For example, if you want to send 100, 100, 100, 100 as the speed for the four wheels, you need to convert 100 to hex: 64
Now as described in the earlier section about the CControlMessage, as the speeds are less than 255, you need to send 00s for the part1 of the speed.
The complete message you will type in the textbox will be:
0064006400640064 which is the hex for 0 100 0 100 0 100 0 100 without any spaces.
For the EmergencyBreakMessage, as the protocol expects no other parameters, it can be sent directly without typing anything in the textbox.
The GUI project is written in Java and is a Netbeans project and thus can be easily imported in Netbeans for any modifications. The steps for importing the project are:
File Menu -> Open Project -> In the path directory navigate to hwswcd-car2x/GUI/Car2xStation and click on the Car2xStation with the coffee symbol. To run the project, you can click on the play button.

4.5.2 Python script

This script allows an easy way to communicate with the cyclone board of the car from any device connected to the wireless network of the WiPort. This can be helpful to remote control the car manually or use it for debugging purposes.
As the script is written in Python, it is very easy to use and extend.

The suggested way of running the script is via an IPython console as it will allow for tab completion and quick documentation lookup.

In your terminal, first please navigate to the directory of the project by using the cd command:

cd /path/to/my/project

Then please launch the IPython console by typing ipython:

ipython

Then please import the script by the command:

- import car2xmessages as c2x (Note you can use tab completion by entering car2x and pressing tab)

To find out the instructions of the script, please use the command:

c2x? #The ? operator behind any variable shows the docstring of that script/variable/function

Now because the car by default allows the beaglebone to communicate with it, you will have to first send a RemoteControlMessage to manually authenticate yourself by sending an ip.

You can do that build a RemoteControlMessage by using the command:

rc = c2x. [TAB]

Response: c2x.BaseMessage c2x.WelcomeMessage

c2x.CARPMessage c2x.main

c2x.CarControlMessage c2x.math

c2x.CarVelocityMessage c2x.send

c2x.EmergencyBreakMessage c2x.socket

c2x.RemoteControlMessage

rc = c2x.RemoteControlMessage?

Response:

Init a RemoteControlMessage to auth my computer

@param ip - Complete ip in string like 255.255.0.0

Constructor information:

Definition:c2x.RemoteControlMessage(self, ip)

Ah, now we know how to call this function. Enter in the shell:

rc = c2x.RemoteControlMessage('192.168.0.100')

Now that we have built the message, we need to send it to the cyclone board by entering this in the shell:

c2x.send(rc)

And you should get a response from the cyclone board acknowledging your message! You should verify in the communication core logs that the message was indeed received and you're in the remote control mode! (Mode 3).

After that, you're authenticated to send any command messages to the car!

Say, you want to control the speed of the car, enter in the shell:

c2x.CarControlMessage?

Response:

Init a CarControlMessage.

@param speed1 - Speed for left front wheel

@param speed2 - Speed for left rear wheel

@param speed3 - Speed for right front wheel

@param speed4 - Speed for right rear wheel

Values need to be between -32768 and +32768

Now enter in the shell:

cc = c2x.CarControlMessage(200, 300, 400, 500) #build the message

c2x.send(cc) #send the message to the car

You can play around with different speeds and see that the car will indeed, after a delay, come to the required speeds. **Caution: While according to the protocol, speeds between (-32768, 32768) are permitted, the control core of the cyclone board restricts the speed around 250.** If you pass any more speeds, the car will not execute the message and will prevent any future messages from execution.

Say after you've played around with different speeds, you need to test a morbid reality where the car is about to crash into a building. At that point you need to send an EmergencyBreakMessage! Once you send this, the car will immediately stop all the wheels and will stop responding to any further messages.

eb = c2x.EmergencyBreakMessage()

c2x.send(eb)

Now the car will have stopped immediately and will stop responding to any further commands just like in a true emergency.

```
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import car2xmessages as c2x

In [2]: c2x.
c2x.BaseMessage            c2x.CarControlMessage      c2x.EmergencyBreakMessage  c2x.WelcomeMessage         c2x.math           c2x.socket
c2x.CARPMessage            c2x.CarVelocityMessage     c2x.RemoteControlMessage   c2x.main                   c2x.send

In [2]: rc = c2x.RemoteControlMessage('192.168.0.100')
[96, 8, 0, 0]
[96, 8, 0, 0]

In [3]: c2x.
c2x.BaseMessage            c2x.CarControlMessage      c2x.EmergencyBreakMessage  c2x.WelcomeMessage         c2x.math           c2x.socket
c2x.CARPMessage            c2x.CarVelocityMessage     c2x.RemoteControlMessage   c2x.main                   c2x.send

In [3]: c2x.send(rc)
['C', 'A', 'R', 'P', 0, 1, 0, 8, 96, 8, 0, 0, 192, 168, 0, 100]
['C', 'A', 'R', 'P', '\x00', '\x01', '\x00', '\x08', '`', '\x08', '\x00', '\x00', '\xc0', '\xa8', '\x00', 'd']
Sent data
Got CAR??d with len 16
['C', 'A', 'R', 'P', '\x00', '\x01', '\x00', '\x08', '`', '\x08', '\x00', '\x00', '\xc0', '\xa8', '\x00', 'd']

In [4]: rc = car
car2xmessages.py   car2xmessages.pyc

In [4]: rc = c2x.Car
c2x.CarControlMessage   c2x.CarVelocityMessage

In [4]: cc = c2x.CarControlMessage(100, 200, 300, 400)
[48, 12, 0, 0]
[100, 200, 300, 400]
[100, 200, 300, 400]
[48, 12, 0, 0]

In [5]: cc = c2x.send(cc)
['C', 'A', 'R', 'P', 0, 1, 0, 12, 48, 12, 0, 0, 0, 100, 0, 200, 1, 44, 1, 144]
['C', 'A', 'R', 'P', '\x00', '\x01', '\x00', '\x0c', '0', '\x0c', '\x00', '\x00', '\x00', 'd', '\x00', '\xc8', '\x01', ',', '\x01', '\x90']
Sent data
Got CARP
        0
         d?,? with len 20
['C', 'A', 'R', 'P', '\x00', '\x01', '\x00', '\x0c', '0', '\x0c', '\x00', '\x00', '\x00', 'd', '\x00', '\xc8', '\x01', ',', '\x01', '\x90']

In [6]: eb = c2x.EmergencyBreakMessage()
[32, 4, 0, 0]

In [7]: eb = c2x.send(eb)
['C', 'A', 'R', 'P', 0, 1, 0, 4, 32, 4, 0, 0]
['C', 'A', 'R', 'P', '\x00', '\x01', '\x00', '\x04', ' ', '\x04', '\x00', '\x00']
Sent data
Got CARP  with len 12
['C', 'A', 'R', 'P', '\x00', '\x01', '\x00', '\x04', ' ', '\x04', '\x00', '\x00']
```

Figure 4.7 Example of running the car2xmessages python script with IPython and the echo server.

In order to test this script on a local computer, you can run the echo_server.py script (also included) as a server:

sudo python echo_server.py #The sudo is needed because it starts a server on port 23

while running this, open another terminal and change these lines in car2xmessages.py from:

#TCP_IP = '127.0.0.1' #echo server ip

TCP_IP = '192.168.0.200' #Cyclone board ip

to

TCP_IP = '127.0.0.1' #echo server ip

#TCP_IP = '192.168.0.200' #Cyclone board ip

and then run the IPython console just like in the beginning.

4.6 Debugging with GDB

Being able to debug the car during run time is vital to any future effort aiming to extend or enhance the capabilities of the car. This important section deals specifically with debugging the Control/Communication core of the car. It can also be extended to debug the nanoboards themselves.

“GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed [GDB Website].” Indeed, it is a very powerful debugger that allows you to stop programs on breakpoints, execute custom code while the program is running, see stack traces and in general very helpful in finding and squashing bugs.

We will here detail the steps of running and debugging the Communication core with GDB as it is the meat of the car. However, the steps for debugging any other cores/nanoboards are the same. In general, we run GDB once the car has started (both communication and control cores are running and waiting for incoming messages).

Steps for GDB:

- Open a Nios II command shell on the Communication core project by right clicking on it in Eclipse
- In the command window, type: nios2-gdb-server --tcpport 2342 --tcppersist
- Open another Nios II command shell from the same project and type: nios2-elf-gdb Car2x_communication.elf. You will see a GDB shell open.
- In the GDB shell, enter: target remote localhost:2342

Now you can debug the core in real-time! First thing you would want to do is add a breakpoint at some line. Once the program gets to that line, it will automatically stop and gdb will allow you to enter custom code/debug.In the GDB shell, enter:

- break socketserver.cpp:1499
- break socketserver.cpp:400
- break socketserver.cpp:1042

You can add as many breakpoints in as many files as you want. Once you're done adding breakpoints, enter continue in the GDB shell to resume execution of the program.

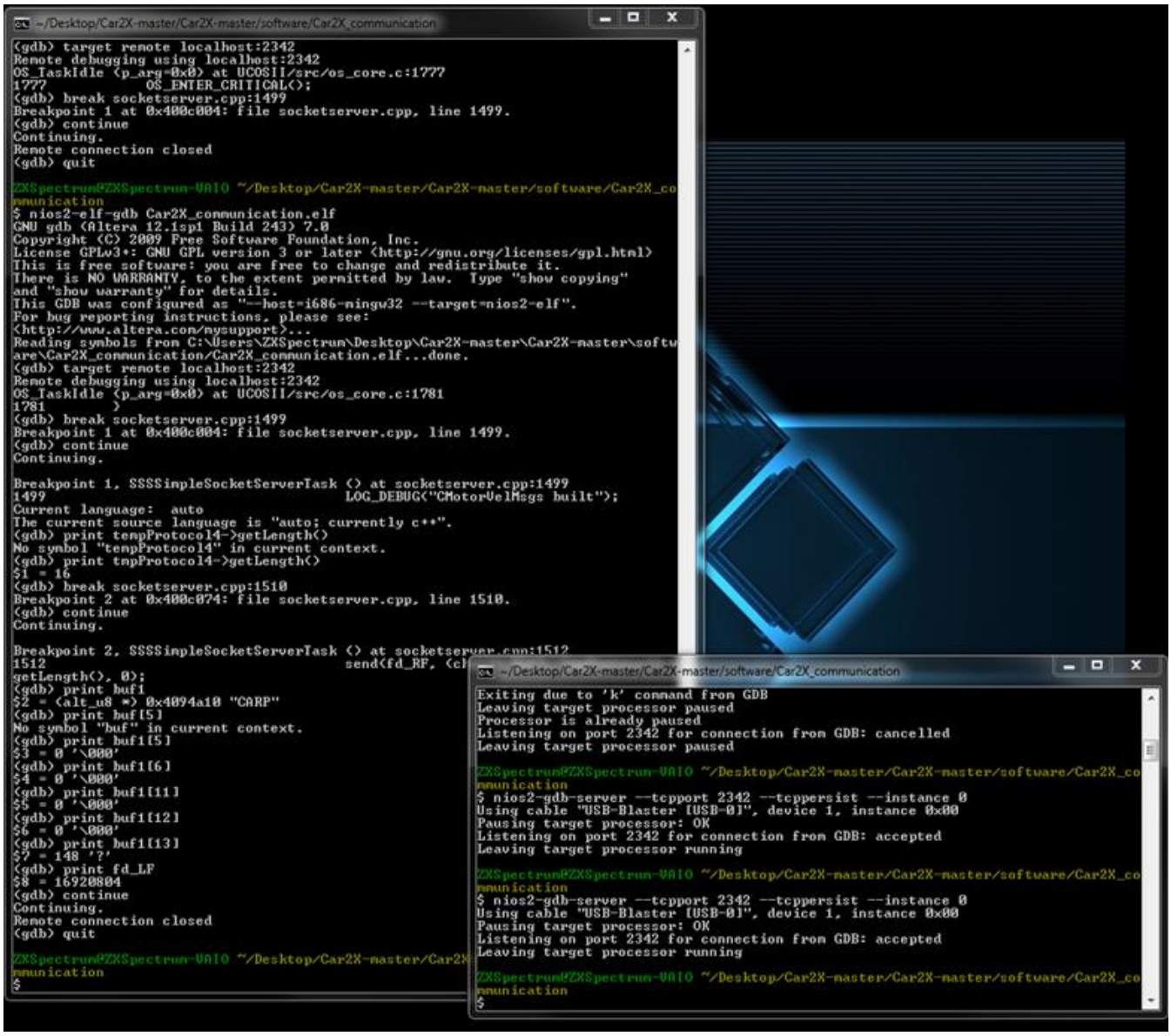

Figure 4.8 debugging using GDB

At some point, you will hit a breakpoint and GDB will allow you interact. Some of the useful commands you can try are:

- next: Execute next line of code. Will not enter functions.
- step: Step to next line of code. Will step into a function.
- where: It will print the stack trace of the program. Very useful to debug the program flow.
- print x: Will print the value of a variable

A handy guide for the GDB commands and their explanations can be found at: http://www.yolinux.com/TUTORIALS/GDB-Commands.html To quit your GDB session, you can use the quit command.

5. Summary

Throughout the semester we continued the work of the previous teams and achieved the following tasks:

- Connection between Beaglebone board and the main FPGA board
- Camera calibration as the camera was replaced
- Loose and misplaced cables were detected and reconnected
- Defected FPGAs were replaced and integrated into the system
- Deadlocks between cores were removed
- Code executing arbitrarily and Random car behavior was removed by implementing cache coherency
- Delays in buffers while sending TCP/IP packets were reduced by means of optimizing the code such as freeing Buffers before being reused.
- Conversion of velocity and steering from rotating wheels to fixed wheels.
- Receiving random velocities from Beaglebone was fixed

Acknowledgment

We would like to thank Mr. Hardik Shah for his support and patience. We also would like the previous teams for their feedback and support as well.

Appendix A: Source code

Git source link (All the source is in one repository including BB, Car2x (with new and old hardware), Python and Java GUI)

References

[1] T. DiCola, "Embedded Linux Board Comparison", White paper, adafruit learning system. Sep-2014.

[2] Nios II Command-Line Tools. Available: http://www.altera.com/literature/hb/nios2/edh_ed51004.pdf

[3] USR-WIFI232-610 WIPort Documentation: http://www.tcp232.net/download/USR-WIFI232-610_en.pdf