Chapter 1

Implementation Details

1.1 Low Level Board[JE]

1.1.1 Neopixel Software Driver

We wrote our own software-based bitbanging driver to interact with the neopixel LED strips. When interacting with the neopixels, it pauses non-critical ChibiOS interrupts momentarily while alternating the (high/low) digital state of the output pin according to a specific timing protocol, where a certain timing of the high and low states means 0, and another means 1. This way, a sequence of bits are sent to the light strip.

Each neopixel reads 24 bits (8 for each three colors), and then forwards any excessive bits to the next pixel. Hence, to control 16 neopixels, we need to send 24 * 16 bits. When all bits are sent, a pause of a certain time lets the neopixel strip know we are finished writing, and that the corresponding LEDs should be turned on with the desired brightness.

1.1.2 Packet Parsing and Serial Connection Details

Received bytes from the serial connection are put into a buffer. When the buffer is large enough, packet parsing is attempted. The algorithm looks from the end (most recent) of the buffer and steps backward until it finds the end delimiter of a packet. When it does, it verifies that the start delimiter and size header are also present (following the netstring standard), and then reads the values transmitted in the packet body. The last byte in the packet body is a checksum byte, which is the result of computing the XOR value of all other value bytes. If the checksum byte indicates the packet is valid, the buffer is cleared, because any older packets will not be of interest. The green LED on the STM32 board is lit up to indicate valid communication.

Whenever we tried to parse the buffer but didn't find a valid packet, the orange LED is lit on the STM32 board. This can occur either because the checksum did not match the contents, or because several bytes (enough to expect a packet) were received that did not constitute a packet.

If no valid packets are received for a while, the red LED is lit. To make sure the output buffer is not filled, which would cause the thread to block while writing and could happen if the USB connection was disconnected, we make sure not to send anything in this state. However, since the serial connection is running in a dedicated thread, any blocking state would not in itself cause safety problems.

1.2 High Level Board

1.2.1 Proxy[ME & JK & IV]

The AutoTuxProxy component extends OpenDaVINCI time-triggered module and runs in the frequency provided by the user with the flag "–freq=". It consists of four separate namespaces, each responsible for a different task - camera, proxy, serial and containerfactory.

[ME] As stated above, within this component we have the handling of the camera. In this case a regular PC web-camera. To capture images we're using a third party library - OpenCV (Open Source Computer Vision). This is a library aimed towards real-time computer vision. Using this library gives us a nice and intuitive way to communicate with the physical camera attached to the Odroid.
Running this module creates a SharedImage (OpenDaVINCI object) from defined variables in the OpenDaVINCI configuration file. We then use the OpenCV library to open (connect) to the physical camera connected to the Odroid board, and fetch (grab), then read (retrieve, decode) the grabbed image into the SharedImage memory location.
In the earlier stages of the development we stored the whole image, which we later on changed to only copy the bottom half of the grabbed image, due to the fact that we're only looking at that part when using the image in the LaneFollowing module. By doing this we essentially cut the memory needed by this module in half, if we're lucky we also saved a couple of nanoseconds when copying the image to the memory location.
How ever we did have problems with the slice consumption within this module, this was mainly because of the different lighting conditions at the test track. As a darker room would mean that the camera needed a longer exposure time this would also increase our slice consumption time to unacceptable values, this did not pose as a problem when the room was better lit up.
To counteract this problem we tried to use OpenCV to set a reasonable static exposure time that would work ok for a more wider area of lightning conditions. Although this was not as easy as it would seem as OpenCV v.2 is using *Video4Linux* API version 1 and the camera properties we needed to access is not available in versions below API v.2. Hence this is still not completely

solved.

[JK] Each proxy iteration (as soon as a time slice is free), the time-triggered module reads the newest valid packet received via USB (wrapped in a vector), rechecks for its correct size, and sends out the data to the session. The data the proxy sends out, is of type SensorBoardData and VehicleData. To generate the containers we send out, two factories are used (part of the containerfactory namespace) which return a shared pointer (to automatically get rid of these objects after they aren't used anymore).
The SBDContainer factory basically takes our internal order and puts the data from the sensors in the order OpenDaVINCI is using in its simulation. The VDContainer factory is slightly more complex in it's behavior: after setting the speed it takes all four bytes we receive from the USB for calculating the absolute travelled path and bit shifts them together to one unsigned integer value.
Note that all values for distances and speed are getting converted from centimeter based values to meter based ones (as it is used in the high-level code). The last step is to receive the most recent containers for VehicleControl and LightSystem data from the session. The data gets processed (steering wheel angle from radians to degrees, speed to four different fixed values and the light settings bit shifted into a four bit sized unsigned char) and it's checksum is generated. Finally a vector out of these values is created which replaces the buffer wrappers send buffer which will be sent out back to the STM32 via USB.

[IV] The serial module is responsible for the serial connection between the low-level board - the STM32 - and the high-level board - the Odroid. It reads from the usb sensor-board data, collected from the sensors, and writes to it control data, evaluated by the DecisionMaker component. The communication between the two is managed by a third-party library - libusb-1.0. Libusb is a lightweight library that is portable and easy to use. It supports all transfer types - control, bulk, interrupt and isochronous - and two types of transfer interfaces - synchronous and asynchronous. Furthermore, the library is thread-safe, it does not manage any threads by its own. The class implementing the calls for managing the usb connection is in SerialIOImpl.cpp. When an object of that class is created, a libusb_context is initialized by making a call to libusb_init() function and passing it a reference to the context struct. It is possible to call this function with the argument NULL , in which case a default context is created for the user. The problem that can occur by doing so is that - if there are multiple components in the application using libusb to communicate with any of the usb devices - the settings for the context might not be applicable, e.g. a part of the program needs to talk to the STM32 board but another can be talking to some other device connected to the system. Although, our program does not have that particular problem, we decided to bind a context to the device we use as a good practice. After the context is initialized, the list of available devices is obtained, and the device we are interested in is opened. To match the correct usb port, we use the STM32 vendor and product id which can be obtained by running lsusb in the terminal on a Linux-based machine. Once the device has been opened, we check if the interface we want to operate on is free, i.e. the kernel is not using it, and claim it if that is not the case. By claiming the interface of the device, we can exchange data through the usb serial connection.

The reading from and writing to the usb are performed using the synchronous API interface. The function to use to do the operation is libusb_bulk_transfer() which takes a number of parameters. In particular, the context to operate on, the usb endpoint, i.e. if it is read or write, indicated by the direction bits in the endpoint address, the buffer where to store the data, the length of the buffer, an int for the actual bytes transferred, and the timeout for the function call. The above mentioned function is blocking but since we do not have multiple components using the usb and there is a dedicated thread operating with it, we decided that is sufficient to use this API. The timeout for the call is set to 10 milliseconds for both the read and write. Libusb is also smart in a way that it does not wait for the entire buffer to be filled. In a case that less data is received than the size of the buffer, the read operation is terminated and the result is returned.
The buffer size for reading is set to size 128. We considered it to be enough with respect to the frequency we are running and the size of our data packets. For writing to the usb, the length of the buffer is equivalent to the size of the data that needs to be sent.

Parsing of the received packets is done when data is appended to the SerialBuffer object with a call to appendReceiveBuffer(), which takes a vector with unsigned chars as elements. The function traverses through all elements, starting from the back (most recent packet) and looks for the start and end delimiters of a correct packet. When one is detected, the values for the different sensors are obtained along with the checksum for this packet. The checksum is then checked against its validity and if so, the packet is considered valid and put in the internal buffer.

1.2.2 Configuration Tool[JK]

The main goal of this component was to have a tool at hand which can be used to tweak values, like the lane-follower gains or other camera setup values, on the run, without changing values in the source code or header files. This would've required recompilation every time and would therefore have been very time consuming.
An application with a GUI seemed optimal. Because we are running everything through a SSH session via the console, we decided to use GNU's ncurses library, which allowed us to create a Text-User-Interface (TUI) independent of the terminal emulator in use.

From the very beginning, we used the effc++ warnings. A lot of "Class contains pointer data members" warnings arose. Research on the internet showed a possible solution: wrapping these pointers into unique_ptrs. Unfortunately these wrapper pointer data types do not interwork with the ncurses library functions, so we ended up in removing the warnings.

To interact with the car's OpenDaVINCI session, a module was written which extends the *TimeTriggeredConferenceClientModule* (TTCCM). Depending on whatever frequency is set by running the application, the ATConfigurator will send out the setup values, as well as the selected state. To avoid lagging, a frequency of 2 Hertz is recommended.
To access this session data, a Singleton class, containing of different data members, as well as their getters and setters, was used. The TUI as well as the TTCCM is accessing this class.

To get some guidance for setting up the webcam to the correct angle, the SharedImage is fetched from the memory and transformed into an *ASCII picture*. Initially it was planned to use libcaca for doing that - a library, which even supports to display these images in color. While trying to integrate libcaca into the ATConfigurator we noticed that it spawned a separate window, instead of being in the same window as the configurator. This was fixed by using the libcaca ncurses driver, which unfortunately overwrote also our main ui window.
We ended up using an external tool to generate the picture and load it into the application. After unsuccessfully trying out img2txt (failed because of its usage of ANSI and UTF-8 chars), we switched over to jp2a, which generates a pure black and white ASCII representation of the picture.
While using this solution some small lags can be noticed: the image file is (over)written to whatever the frequency is set. Therefore it can happen that

the image is empty or corrupt, while jp2a tries to read it. We tried to avoid this by using a small buffer/tmpimage within the code, but some lags can be noticed still. Further improvements seemed not reasonable due to the late stage of the project phase.

One problem we faced while developing this application was navigating through it with arrow, return and back keys. The problem is that the values aren't as they are programmed in ncurses. Custom values were defined in the header file.