# Discoverpixy

Ein Projekt zur Objekterkennung mittels der
Pixy CMUCam5 und der STM32 Mikrokontroller-Plattform


| | |
|---|---|
| Autoren: | Aaron Schmocker und Timo Lang |
| Dozent: | G. Krucker |
| Modul: | Softwaredesign und Softwareprojekte (BTE5052) |
| Abgabedatum: | 8.6.2015 |

# Inhaltsverzeichnis

# 1 Aufgabenstellung

Im Rahmen des Moduls BTE5052 soll eine Projektarbeit durchgeführt werden. Ziel ist es mithilfe der Videokamera Pixy CMUCam5 Objekte zu erkennen, ihnen zu folgen, die Erkennung geeignet darzustellen und zu konfigurieren. Als Entwicklungsplattform soll die STM32 Mikrokontroller-Serie dienen. Die effektive Objekterkennung erfolgt durch die Pixy-Kamera und die darunterliegende Technologie sowie die Thematik der Bildverarbeitung sei nicht Ziel dieser Arbeit. Mit der Pixy-Kamera ist via USB oder I2C zu kommunizieren. Als Mikrokontroller-Kit stehen das STM32F4Discovery Board und das Carme-Kit (BFH) zur Verfügung. Die erstellte Software ist in der Programmiersprache C zu schreiben und entsprechend zu kommentieren und dokumentieren.

Wir werden für die Realisierung dieser Arbeit das STM32F4Discovery Board verwenden und via USB mit der Pixy-Kamera kommunizieren. Um die Entwicklung zu vereinfachen wird nebenbei ein Emulator enwickelt, welcher das Ausführen der Applikation auf dem PC erlaubt.

# 2 Planung und Ziele

**Fokus der Arbeit:**

- Planung und Durchführung eines Mikrokontroller-Projekts im Zweierteam.
- Design einer erweiterbaren Architektur, so dass das Projekt von Dritten weitergeführt und optimiert werden könnte.
- Implementierung und Dokumentation einer funktionstüchtigen Lösung gemäss Aufgabenstellung.
- Abstraktion der Anwendung, zur Ausführung auf dem PC bzw. dem STM32F4Discovery

**Nicht Fokus der Arbeit:**

- Korrekte und schöne Implementierung des Usb-Hosts gemäss Spezifikation für die Pixy-Kamera auf dem STM32F4Discovery.
- Herstellen von produktionsreifen Bibliotheken zur Kommunikation mit Display, SD-Karte und Pixy
- Herstellen eines komplexen Emulators, dessen Funktionalität über die Funktionalität der Anwendung auf dem STM32F4Discovery hinausgeht.
- Perfomance-Optimierungen

## Zeitplanung / Meilensteine:

| Name | Bemerkungen | Datum |
|---|---|---|
| FSMC Display Funktionen | Mindestens diese die auch im Emulator vorhanden sind, Ziel: Pixy video auf dem Display | 20.4.2015 |
| Touch Controller, Basic | Ziel: zeichnen von Pixel auf dem Display via Touch | 27.4.2015 |
| Sd Charte, Basic | Ziel: anzeigen eines Bitmaps auf dem Display | 27.4.2015 |
| 2nd Display support | Nur wenn auch via FSMC möglich. | 27.4.2015 |
| Speisung | | 27.4.2015 |
| Schaltplan | | 27.4.2015 |
| Hardware ok | | 4.5.2015 |
| Display fertig abstrahiert | | 4.5.2015 |
| Touch fertig abstrahiert | | 4.5.2015 |
| Sd Karte fertig abstrahiert | | 4.5.2015 |
| Gui Library fertig | d.h. Buttons, Chechbox, Slider und ev NumUpDown, Radio, Dropdown | 11.5.2015 |
| Alle LowLevel funktionalitäten fertig | Ziel: ab hier wird nur noch der “common”-Order verwendet | 11.5.2015 |
| Objekte von Pixy empfangen | Ziel: die von Pixy erfassten Objekte werden auf dem Display dargestellt | 18.5.2015 |
| Objektracking | Ziel: Objekttracking funktioniert und ist konfigurierbar | 25.5.2015 |
| Abgabe | | 8.6.2015 |

# 3 Benutzung

## 3.1 Ordnerstruktur

Auf oberster Stufe befinden sich die 4 folgenden (haupt) Ordner:

- common
  - enthält allen Plattformunabhängigen Code, sowie die Applikation selbst
  - Würde man den app Unterordner löschen, hätte man ein “leeres Projekt” mit Treiber zur Ansteuerung von Display, Touch-Controller, SD-Karte und Pixy.
  - Der lowlevel Unterornder enthält die Prototypen für die Funktionen die von der jeweiligen Ziel-Plattform implementiert werden müssen.
- discovery
  - enthält allen lowlevel Code und den Initialisierungscode um die Software auf dem STM32F4Discovery zu betreiben. Die Pixy wird via UTB-OTG direkt ans Discovery angeschlossen.
  - Der libs Unterorder enthält Code von Dritten, der src Unterordner enthält den Hauptcode für die Plattform.
- emulator
  - enthält allen lowlevel Code und den Initialisierungscode um die Software auf dem PC (im sogenannten “Emulator”) laufen zu lassen. Die Pixy wird via USB direkt an den PC angeschlossen.
  - Der libs Unterorder enthält Code von Dritten, der qt Unterordner enthält den Hauptcode für die Plattform.
- doc
  - enhält alle Dokumentations-Resourcen sowie Referenzcode und Datenblätter.

## 3.2 Dokumentation und andere Resourcen

**Vorliegende Dokumentation**

In der vorliegenden Dokumentation (aktuelles Dokument) wurden vorallem einführende Themen, theoretische Grundlagen, sowie die Analyse und das Design des Projekts erläutert.

**Doxygen-Dokumentation**

In der Doxygen-Dokumentation wurden alle Methoden und Strukturen des plattformunabhängigen Teils dokumentiert. Die Schnittstellen aller Module sind in dieser Dokumentation ersichtlich.

Die Doxygen Dokumentation muss in einem Browser angeschaut werden. Sie kann entweder online auf http://t-moe.github.io/discoverpixy/ eingesehen werden oder selbst gebaut werden. Um die Doxygen-Dokumentation selbst zu bauen ist der Befehl *doxygen* im Hauptverzeichnis des Projekts auszuführen und anschliessend die Datei *doc/html/index.html* im Browser zu öffnen.

Es empfiehlt sich die “Modules” Seite als Ausgangspunkt zur Navigation zu benutzen.

**Dokumentation des Source-Codes**

Jeglicher von uns verfasster Source-Code wurde dokumentiert. Die grobe Beschreibung der Methoden und die Beschreibung der Parameter ist beim plattformunabhängigen Code dem Header-File bzw der Doxygen-Dokumentation zu entnehmen. Der Rest wurde jeweils direkt innerhalb der c bzw cpp Datei mithife von Kommentaren dokumentiert.

Der Source-Code von Dritten wurde nicht kommentiert. Jedoch ist die in einem späteren Kapitel der vorliegenden Dokumentation erklärt woher der Drittanbieter-Code stammt und allfällige Modifikationen sind erläutert.

**Datenblätter und Referenzcode**

Besonders bei der Arbeit mit der Ziel-Plattform (STM32F4Discovery) waren einige Internetrecherchen nötig. Die Datenblätter der verwendeten Controller sind im doc Ordner abgelegt. Ebenso sind Pinouts und Verdrathungspläne im diesem Ordner abgelegt.

Als Grundlage für die Entwicklung der Tft und GUI Module sowie des PID-Reglers wurde Code aus früheren Projekte verwendet und anschliessend komplett überarbeitet. Der ursprüngliche Code ist als Archiv im doc Ordner vorhanden.

## 3.3 Kompilierung des Projekts

Grundsätzlich sollte es möglich sein sowohl den Emulator, wie auch die Software für das STM32F4Discovery sowohl auf Linux wie auch auf Windows zu kompilieren.

Je nach Betriebssystem sind andere Voraussetzungen nötig:

| **Ziel-Plattform** | **Voraussetzungen zur Kompilierung auf Linux** | **Voraussetzungen zur Kompilierung auf Windows** |
|---|---|---|
| emulator | • qt5<br>  ◦ core<br>  ◦ gui<br>  ◦ widgets<br>  ◦ qmake<br>• boost<br>• make | Gleich wie bei Linux und zusätzlich:<br>• mysys / mingw<br>Siehe readme im emulator ordner |

| | | |
|---|---|---|
| | • g++ (x86-64)<br>• gdb (x86-64) | |
| discovery | • make<br>• gcc (arm-none-eabi)<br>• gdb (arm-none-eabi)<br>• st-link | Nicht getestet.<br>• Vermutlich sollte Atollic Studio inkl ST-Link Treiber genügen. |

Der Code muss für jede Ziel-Plattform separat kompiliert werden. Verwendet werden in beiden Fällen einfache Makefiles. Auf der Kommandozeile gelten folgende Befehle:

| **In den Ordnern "emulator" und "discovery"** | |
|---|---|
| *make* oder *make all* | Kompiliert den gesamten Source-Code für die jeweilige Plattform |
| *make clean* | Räumt alle erzeugen Objekt- und Binary-Files weg |
| *make debug* | Gleich wie *make all*.<br>Falls nötig wird zudem die Zielhardware geflashd.<br>Anschliessend wir gdb gestartet um den Code auf der jeweiligen Plattform zu debuggen. |
| **Nur im Ordner "emulator"** | |
| *make run* | Gleich wie *make all*.<br>Anschliessend wird der Emulator gestartet |
| **Nur im Ordner "discovery"** | |
| *make flash* | Gleich wie *make all*.<br>Anschliessend wird die Software via ST-Link auf die Zielhardware geflashd. |
| *make start* | Startet den ST-Link Server. Passiert normalerweise automatisch bei make all |
| *make stop* | Beendet den ST-Link Server. |
| *make backup* | Macht ein Backup der aktuellen geflashten Firmware und speichert das Backup als .bin Datei. |

Der Emulator Ordner kann zudem mit Eclipse geöffnet werden oder der Qt unterordner mit Qt-Creator. Für die Arbeit mit dem Discovery empfehlen wir das Arbeiten auf der Konsole.

## 3.4 Ausführen und Bedienen des Projekts

### 3.4.1 Allgemeine Hinweise zur Ausführung

Die Benutzeroberfläche sieht auf der Zielhardware (STM32F4Discovery) und im Emulator gleich aus und sollte weitgehend selbsterklärend sein. Um alle Funktionalitäten nutzen zu können, ist eine Pixy-Kamera (http://www.cmucam.org/projects/cmucam5 ) erforderlich.

Besonders zu beachten ist, dass die Servos korrekt angeschlossen werden. Schritt 15 in der Online-Anleitung zum Bestücken der Servos

(http://www.cmucam.org/projects/cmucam5/wiki/Assembling_pantilt_Mechanism ) erläutert den Vorgang exakt.

Im folgenden werden die Unterschiede und Besonderheiten zur Ausführung auf der jeweilgen Zielplattform erläutert.

### 3.4.2 Ausführung des Emulators

| Voraussetzungen zur Ausführung auf Linux | Voraussetzungen zur Ausführung auf Windows |
|---|---|
| • qt5<br>  ◦ core<br>  ◦ gui<br>  ◦ widgets<br>• libusb | • Installierte Treiber für Pixy. Intstallation erfolgt durch Pixymon Installer: http://www.cmucam.org/projects/cmucam5/wiki/Installing_PixyMon_on_Windows_Vista_7_or_8<br>• Falls sie ein gepacktes Release erhalten haben sind alle Abhängigkeiten enthalten. Gepackte Releases finden sich auf: https://github.com/t-moe/discoverpixy/releases<br>• Sofern sie den Source-Code selbst kompiliert haben benötigen sie zur Ausführung die gleichen Voraussetzungen wie bei Linux. |

Die Pixy-Kamera kann direkt an den PC angeschlossen werden (am besten via USB 2.0). Eine Speisung mit einem externen Netzteil ist im normallfall **nicht** nötig.

### 3.4.3 Ausführung auf der Ziel-Plattform STM32F4Discovery

Sollten sie bereits ein STM32F4Discovery mit Display und mit geflashter Software erhalten haben, brauchen sie zur Ausführung nur eine Pixy-Kamera und ein Labornetzteil. Ansonsten muss die Software zuerst auf Linux gebaut und geflasht werden (siehe oben).

Die Servos benötigen ca einen Stromstärke von 1 Ampere. Das STM32F4Discovery ist auf dem USB-Ausgang auf 500mA begrenzt. Desshalb **muss** die Pixy-Kamera vor dem Anschliessen des USB-Kabels mit dem Labornetzteil über den Power-Connector mit Strom versorgt werden. Eine Anleitung dazu findet sich auf der Wiki Seite von Pixy:
http://www.cmucam.org/projects/cmucam5/wiki/Powering_Pixy#Power-connector

Sofern die Servos nicht angeschlossen sind, kann die Pixy-Kamera direkt ans Discovery angeschlossen werden.

**Achtung**: Wenn sie das Discovery-Board via USB speisen während die Pixy-Kamera über USB-OTG angeschlossen ist und vom Netzteil versorgt wird, sind die die Massen von Netzteil und die Masse vom

Mainboards ihres Computers miteinder vernetzt. Ist dies nicht erwünscht muss ein galvanisch getrenntes Netzteil oder ein Trenntrafo verwendet werden.

Sobald die Pixy-Kamera an dem Netzteil angeschlossen ist, kann das STM32F4Discovery auch gespeisen werden. Anschlissend sollte die Applikation mit dem Hauptbildschirm starten. Die Pixy-Kamera kann zur Laufzeit via USB-OTG ans Discovery angeschlossen werden oder davon getrennt werden.

# 4 Analyse

## 4.1 Highlevel

Unser System besteht aus einem Prozess, dessen Name identisch mit dem Projektnamen ist: “discoverpixy”. Im folgenden nennen wir diesen Prozess aber “Applikation”. Die Applikation kommuniziert mit verschiedenen externen Komponenten. Das Display (Tft) wird als Ausgabegerät und zur Anzeige von Informationen an den Benutzer benötigt. Der Benutzer bedient die Applikation in dem er seine Eingaben an dem Touch-Controller tätigt. Die Applikation kommuniziert je nach gewähltem Modus auch mit der Pixy-Kamera und der SD-Karte, jeweils in beide Richtungen.

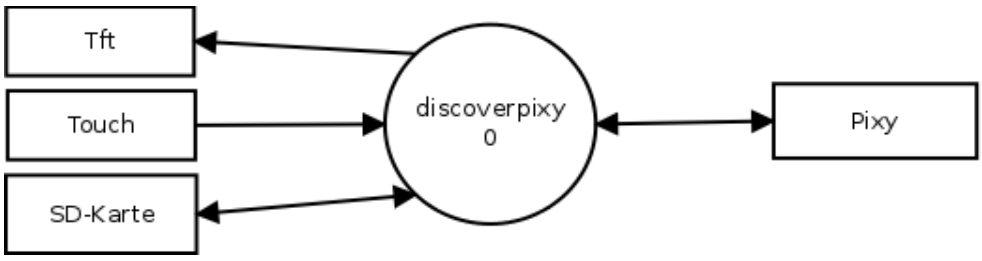



*Illustration 1: Kontextdiagramm*

Das folgende Anwendungsfalldiagramm zeigt wie der Benutzer mit der Applikation interagiert:

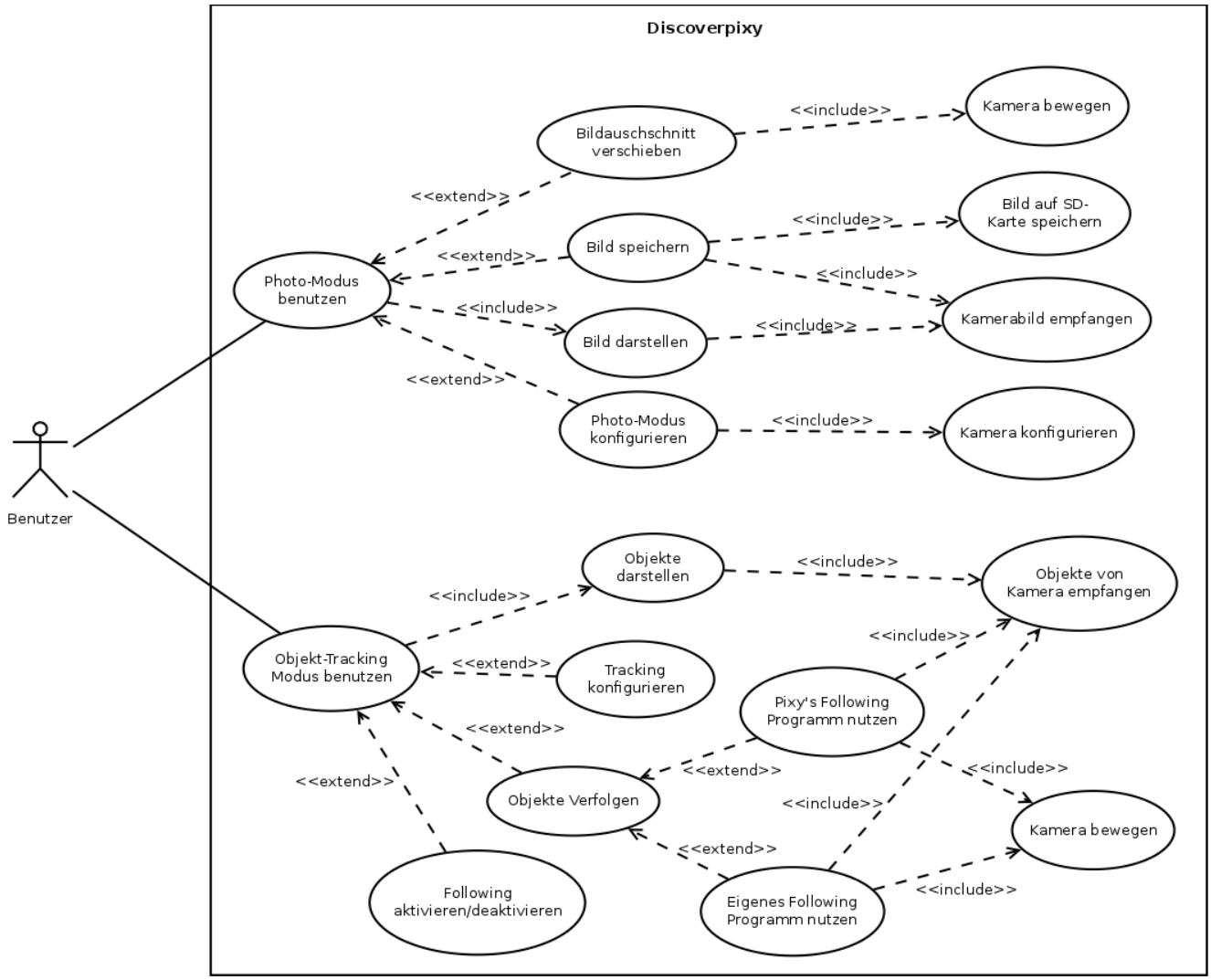



*Illustration 2: Anwendungsfalldiagramm*

Der Benutzer kann entweder den “Objekt-Tracking Modus” oder den “Photo-Modus” benutzen.

Wenn der Benutzer den Photo-Modus nutzt, so sieht er das akutelle Kamerabild. Er kann den Bildausschnitt verschieben und/oder das Bild auf die SD-Karte speichern. Falls nötig, kann er auch Einstellungen vornehmen.

Wenn der Benutzer den Objekt-Tracking Modus nutzt, so sieht er die aktuell erkannten Objekte. Wenn er möchte kann er das automatische Verfolgen von Objekten einschalten oder ausschalten. Ist das automatische Verfolgen eingeschaltet, so verfolgt die Kamera die Objekte mithilfe der Servos. Weiter kann der Benutzer das Tracking konfigurieren, um z.B. auszuwählen welche Objekte verfolgt werden sollen.

## 4.2 Aufbau der Benutzeroberfläche/Bedienung

Die Anwendung soll aus mehreren “Bildschirmen” (engl. Screens) aufgebaut sein.

Gestartet wird auf dem Hauptbildschirm, von wo aus man mit Buttons zu den entsprechenden Unterbildschirmen kommt. Zurück zum Hauptbildschirm kommt man jeweils mit dem “Back” Button.

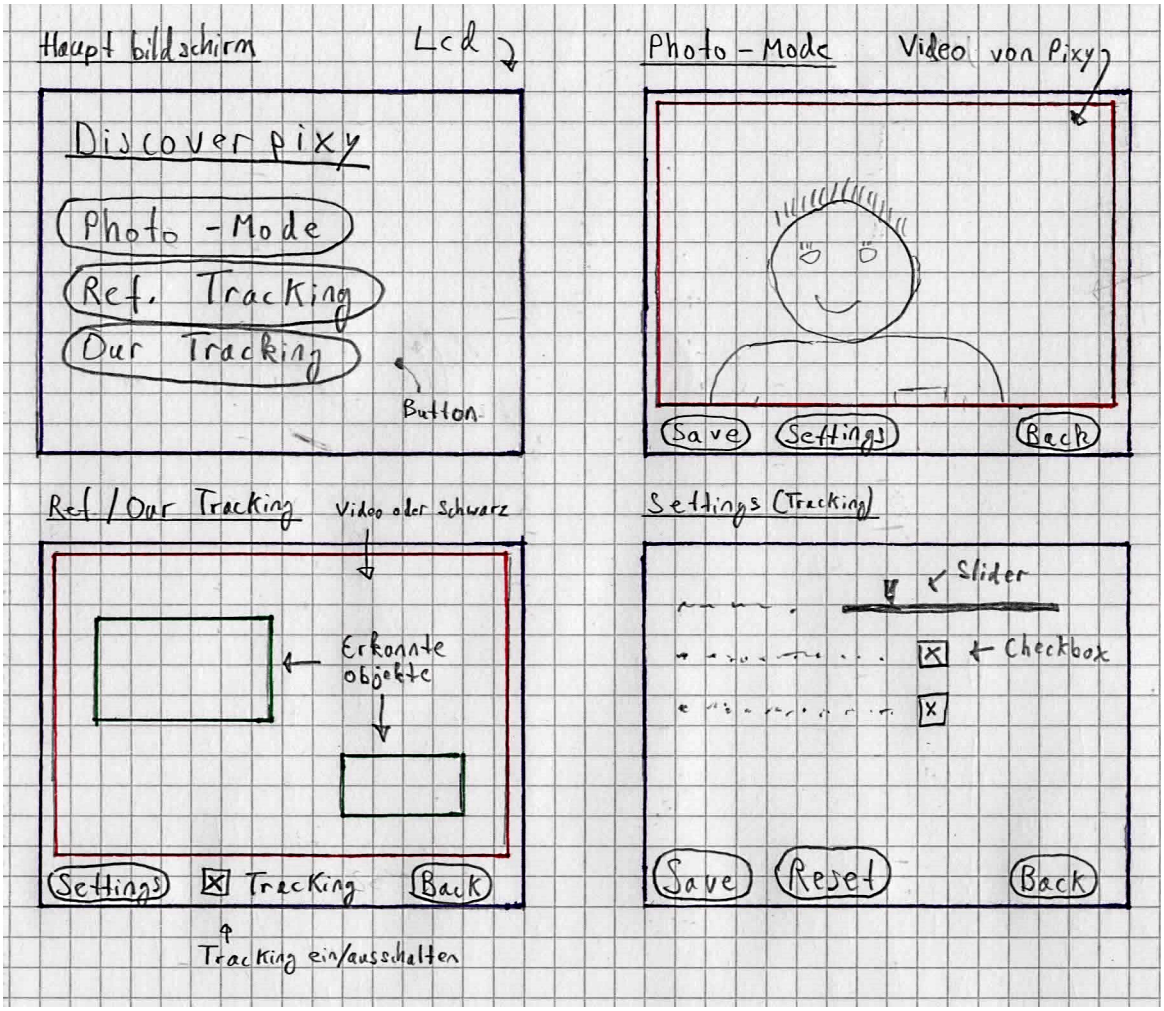



*Illustration 3: Prototypen verschiedener “Screens”*

**Photo-Mode Bildschirm:**

Der Benutzer sieht im Zentrum das aktuelle Bild von der Pixy-Kamera. Der Bildausschnitt kann verschoben werden in dem das Bild “gepackt und gezogen” (engl. Click&Drag) wird. Mithilfe des “Save” Buttons kann ein Snapshot auf die SD-Karte gespeichert werden. Sollten für die “Photo-Mode” Funktionalität eigene Einstellungen von nöten sein, so können diese über den Settings Button erreicht werden.

**Ref. Tracking / Our Tracking Bildschirm:**

Die Pixy Kamera hat bereits ein Programm zum automatischen Tracking von Objekten. Dieses kann über den “Ref. Tracking” (Ref für Reference) im Hauptbildschrim erreicht werden. Wird im

Hauptbildschirm “Our Tracking” ausgewählt, soll unsere eigene Implementierung für das Verfolgen der Objekte verwendet werden. Die Bildschirme der beiden Modi sind identisch:

In der Mitte ist wieder das Bild er Pixy-Kamera sichtbar. Sollte es aus Performancegründen nicht möglich sein das Video anzuzeigen, ist der Hintergrund eingefärbt. Auf dem Hintergrund werden die Rechtecke der erkannten Objekte anzeigt. Wenn die Checkbox “Tracking” (bzw. “Following”) aktiviert ist, wird automatisch das grösste Objekt verfolgt. Ansonsten bleibt die Kamera fixiert und optional kann der Bildausschnitt wie beim “Photo-Mode” verschoben werden. Jegliche Konfiguration folgt über den “Settings”-Bildschirm.

**Settings (Tracking):**

Zum Einstellen der Objekt-Erkennung und des Objekt-Trackings soll es auf den “Settings”-Bildschirm verschiedene Optionen geben. Die Optionen haben jeweils eine Beschriftung und entweder eine Checkbox oder einen Slider um den Wert zu verändern. Der Button “Save” speichert die Einstellungen, “Reset” verwirft sie. Mögliche Einstellungen: Parameter für den Regelkreis (PID-Regler?), Grenzen für die Servos, Empfindlichkeit der Objekt-Erkennung, usw.

## 4.3 Abstraktion

Remove this?

Die gesamte Software soll modular aufgebaut werden. Die einzelnen Module sollen entweder unabhängig agieren oder ihre Abhängigkeiten zu den anderen Modulen klar spezifizieren. Zudem soll die Anwendung mindestens auf zwei Plattformen laufen: Auf dem PC (“Emulator”) und auf dem Zielsystem (STM32F4Discovery). Läuft die Anwendung im Emulator, so wird die Pixy-Kamera direkt an den PC angeschlossen. Läuft die Anwendung auf dem STM32F4Discovery, so wird die Pixy-Kamera via USB-OTG ans Discovery angeschlossen.

# 5 Grobdesign

Die Software soll in Module unterteilt werden. Das folgende Diagramm zeigt die vorhandenen Module und deren Abhängigkeiten. Findet eine Kommunikation zwischen 2 Modulen statt so wird dies durch einen Pfeil dargestellt.

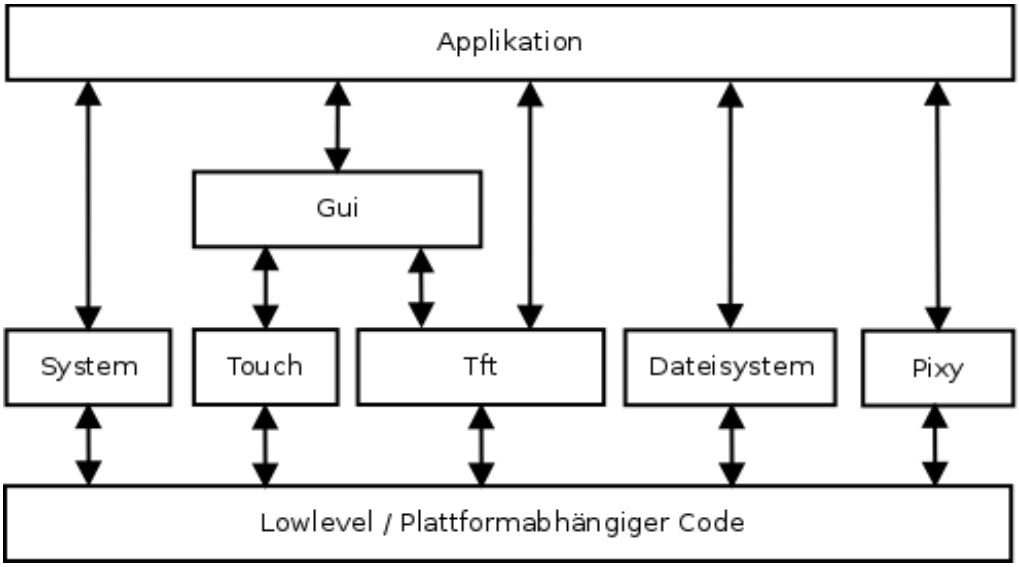



*Illustration 4: Übersicht über die Module und deren Abhängigkeiten*

Alle Module sollen aus Plattformunabhängigem Code bestehen. Die Module greifen wenn nötig auf Plattformspezifischen Code zu. Dieser wird aber abstrahiert und muss dann für jedes Zielsystem separat implementiert werden. Details zu der Implementierung folgen in einem späteren Kapitel. Im folgenden soll jedes Modul und seine Schnittstellen kurz beschrieben werden

## 5.1 Applikation

| | |
|---|---|
| Ordername: | app |
| Beschreibung: | Das Applikations Modul beinhaltet den effektiven Code zur Steuerung aller Geräte und Interaktion mit dem Benutzer. |
| Benötigte Module: | System, Gui, Tft, Dateisystem, Pixy |
| Benötigt von: | Keinem Modul, aber dem Gesamtsystem. |
| Benötigte Low-Level Funktionen: | Keine |
| Bereitgestellte Funktionen: | void app_init();<br>void app_process(); |

## 5.2 System

| | |
|---|---|
| Ordername: | system |
| Beschreibung: | Das System Modul kontrolliert die darunterliegende Hardware und bietet Sleep-Funktionen an |
| Benötigte Module: | keine |
| Benötigt von: | Applikation |
| Benötigte Low-Level Funktionen: | bool ll_system_init();<br>void ll_system_delay(uint32_t msec);<br>void ll_system_process(); |
| Bereitgestellte Funktionen: | bool system_init();<br>void system_delay(uint32_t msec);<br>void system_process(); |

## 5.3 Tft

| | |
|---|---|
| Ordername: | tft |
| Beschreibung: | Das Tft Modul ermöglicht das Zeichnen von Formen und Texten auf ein Display. |
| Benötigte Module: | keine |
| Benötigt von: | Gui, Applikation |
| Benötigte Low-Level Funktionen: | bool ll_tft_init();<br>void ll_tft_clear(uint16_t color);<br>void ll_tft_draw_line(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint16_t color);<br>void ll_tft_draw_pixel(uint16_t x,uint16_t y,uint16_t color);<br>void ll_tft_draw_rectangle(uint16_t x1,uint16_t y1,uint16_t x2,uint16_t y2, uint16_t color);<br>void ll_tft_fill_rectangle(uint16_t x1,uint16_t y1,uint16_t x2,uint16_t y2, uint16_t color);<br>void ll_tft_draw_bitmap_unscaled(uint16_t x, uint16_t y, uint16_t width, uint16_t height, const uint16_t *dat);<br>void ll_tft_draw_circle(uint16_t x, uint16_t y, uint16_t r, uint16_t color);<br>uint8_t ll_tft_num_fonts();<br>uint8_t ll_tft_font_height(uint8_t fontnum);<br>uint8_t ll_tft_font_width(uint8_t fontnum);<br>void ll_tft_draw_char(uint16_t x, uint16_t y, uint16_t color, uint16_t bgcolor, uint8_t font, char c); |
| Bereitgestellte Funktionen: | bool tft_init();<br>void tft_clear(uint16_t color);<br>void tft_draw_line(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint16_t color); |

| | |
|---|---|
| | void tft_draw_pixel(uint16_t x,uint16_t y,uint16_t color);<br>void tft_draw_rectangle(uint16_t x1,uint16_t y1,uint16_t x2,uint16_t y2, uint16_t color);<br>void tft_fill_rectangle(uint16_t x1,uint16_t y1,uint16_t x2,uint16_t y2, uint16_t color);<br>void tft_draw_bitmap_unscaled(uint16_t x, uint16_t y, uint16_t width, uint16_t height, const uint16_t* dat);<br>void tft_draw_circle(uint16_t x, uint16_t y, uint16_t r, uint16_t color);<br>uint8_t tft_num_fonts();<br>uint8_t tft_font_height(uint8_t fontnum);<br>uint8_t tft_font_width(uint8_t fontnum);<br>void tft_print_line(uint16_t x, uint16_t y, uint16_t color, uint16_t bgcolor, uint8_t font, const char* text); |

## 5.4 Touch

| | |
|---|---|
| Ordername: | touch |
| Beschreibung: | Das Touch Modul empfängt Benutzereingaben und löst Benachrichtigungen im Falle einer Interaktion aus. |
| Benötigte Module: | keine |
| Benötigt von: | Gui |
| Benötigte Low-Level Funktionen: | bool ll_touch_init(); |
| Bereitgestellte Funktionen: | bool touch_add_raw_event(int x, int y, Event_Flags flags);<br>bool touch_init();<br>bool touch_has_empty(unsigned char num);<br>bool touch_register_area(TOUCH_AREA_STRUCT* area);<br>bool touch_unregister_area(TOUCH_AREA_STRUCT* area); |

## 5.5 Dateisystem

| | |
|---|---|
| Ordername: | filesystem |
| Beschreibung: | Das Dateisystem Modul bietet Funktionen zum lesen und schreiben auf einen Datenträger (z.B. SD-Karte) |
| Benötigte Module: | keine |
| Benötigt von: | Applikation |
| Benötigte Low-Level Funktionen: | bool ll_filesystem_init();<br>…. noch unklar |
| Bereitgestellte Funktionen: | bool filesystem_init();<br>… noch unklar |

## 5.6 Gui

| | |
|---|---|
| Ordername: | gui |
| Beschreibung: | Das GUI Modul bietet Funktionen zum erstellen von und interagieren mit, Gui-Elementen (z.B. Buttons). |
| Benötigte Module: | Tft, Touch |
| Benötigt von: | Applikation |
| Benötigte Low-Level Funktionen: | keine |
| Bereitgestellte Funktionen: | bool gui_init();<br>bool guiAddButton(BUTTON_STRUCT* button);<br>void guiRemoveButton(BUTTON_STRUCT* button);<br>void guiRedrawButton(BUTTON_STRUCT* button);<br>bool guiAddBitmapButton(BITMAPBUTTON_STRUCT* button);<br>void guiRemoveBitmapButton(BITMAPBUTTON_STRUCT* button);<br>void void guiRedrawBitmapButton(BITMAPBUTTON_STRUCT* button);<br>.. und noch mehr |

## 5.7 Pixy

| | |
|---|---|
| Ordername: | pixy |
| Beschreibung: | Das Pixy Modul bietet Funktionen zur Interaktion mit der Pixy Cam. |
| Benötigte Module: | keine |
| Benötigt von: | Applikation |
| Benötigte Low-Level Funktionen: | Keine ll_* funktionen. Alle bereitgestellten Funktionen werden direkt von der Plattform-Implementierung bereitgestellt. |
| Bereitgestellte Funktionen: | int pixy_init();<br>int pixy_blocks_are_new();<br>int pixy_get_blocks(uint16_t max_blocks, struct Block * blocks);<br>int pixy_service();<br>int pixy_command(const char *name, ...);<br>void pixy_close();<br>int pixy_led_set_RGB(uint8_t red, uint8_t green, uint8_t blue);<br>int pixy_led_set_max_current(uint32_t current);<br>int pixy_led_get_max_current();<br>int pixy_cam_set_auto_white_balance(uint8_t value);<br>int pixy_cam_get_auto_white_balance();<br>uint32_t pixy_cam_get_white_balance_value();<br>int pixy_cam_set_white_balance_value(uint8_t red, uint8_t green, uint8_t blue); |

| | |
|---|---|
| | int pixy_cam_set_auto_exposure_compensation(uint8_t enable);<br>int pixy_cam_get_auto_exposure_compensation();<br>int pixy_cam_set_exposure_compensation(uint8_t gain, uint16_t compensation);<br>int pixy_cam_get_exposure_compensation(uint8_t * gain, uint16_t * compensation);<br>int pixy_cam_set_brightness(uint8_t brightness);<br>int pixy_cam_get_brightness();<br>int pixy_rcs_get_position(uint8_t channel);<br>int pixy_rcs_set_position(uint8_t channel, uint16_t position);<br>int pixy_rcs_set_frequency(uint16_t frequency);<br>int pixy_get_firmware_version(uint16_t * major, uint16_t * minor, uint16_t * build); |

# 6 Spezielle, erwähnenswerte, verwendete Techniken

## 6.1 Verkettete Listen

Verkette Listen sind ein bekanntes Konzept in der Softwareentwicklung. Der Code zum bearbeiten und durchsuchen der Liste erfordert meistens die dynamische allozierung von Speicher und ist sehr repetitive. Leider ist in der Standartlibrary von C99 keine Listenimplementierung vorhanden.

In unserem Projekt finden verkette Listen an diversen Stellen verwendung. Oftmals werden jedoch nur gewisse Element-Operationen benötigt und nicht alle (z.B. nur anfügen/löschen am ende der Liste und lineares durchlaufen). Dadurch ist der Code jeweils überschaubar und es ist nicht nötig eine generische Listenimplementierung zu verwenden.

Wichtig zu bemerken ist an dieser Stelle, dass verkettete Listen auch ohne dynamische Speicherallokation funktionieren können. Dazu wird das einzufügende Element vom Benutzer der Library als globale Variable definiert und anschliessend wird die Adresse davon an die Library übergeben um das Element der Liste anzuhängen. Ein Beispiel dazu bilden die *Screen* Strukturen unserer Applikation, welche jeweils ein Listenelement einer verketteten Liste darfstellen.

## 6.2 Objektorientiertes programmieren in C

Obwohl C eigentlich keine Objektorientierte Sprache ist, ist es möglich ansatzweise Objektorientiert zu programmieren. Funktionen können zwar nicht als Member einer Struktur definiert werden, aber sie können entsprechend benannt werden und mit einem weiteren Parameter “this” versehen werden.

Nehmen wir als Beispiel folgende C++ methode:

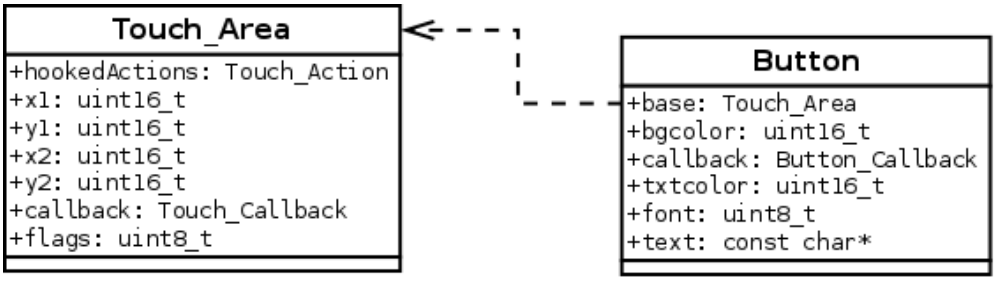
```
void Auto::fahre(int x, int y) { ... }
```

In C kann die Klassenmethode  wie folgt realisiert werden:

```
void auto_fahre(struct Auto* this, int x, int y) { ... }
```

Anstatt die Methode *fahre* auf dem Objekt aufzurufen ( *meinauto.fahre(1,3) )* wird die freie Methode *auto_fahre* aufgerufen mit dem Objekt als ersten Parameter ( *auto_fahre(meinauto,1,3)* ).

Auch das Vererben von Feldern ist möglich. Nehmen wir als Beispiel 2 Strukturen aus dem GUI Modul des Projekts. Die Struktur *Button* soll alle Member der Struktur *Touch_Area* besitzen und noch um einige Member erweitert werden ("*Button* erbt von *Touch_Area*"). Um dies zu realisieren wird in der *Button* Struktur zu oberst ein Feld eingefügt "*base*" vom Typ *Touch_Area*.




Hat man ein Element vom Typ *Button* vorliegen und will auf Member von *Touch_Area* zugreifen muss zusätzlich noch "*base.*" vor den Membernamen geschrieben werden. z.B. *button1.base.x1 = 100.* Pointer vom Typ *Touch_Area* können zudem in einen Pointer vom Typ *Button* gecasted werden und umgekehrt, da *base* einen Strukturoffset von 0 hat.

Etwas schwieriger gestaltet sich das überschreiben von Methoden der Basisklasse (im Beispiel *Touch_Area*). Der C++ Kompiler nutzt dazu sogenannte vtables. Darauf soll aber nicht näher eingegangen werden, da wir diese Funktionalität nicht benötigen.

# 7 Implementierung der Module

## 7.1 Applikation

| **Status:** | Stabil und abgeschlossen |
|---|---|
| **Doxygen-Dokumentation:** | http://t-moe.github.io/discoverpixy/group__app.html |
| **Quellen:** | Keine |
| **Zugehörige Quell- und Headerdateien:** | Ordner: common/app (100% Eigenentwicklung) |
| **Durchgeführte Tests:** | |

**Beschreibung:**

Das Applikationsmodul besteht aus mehreren Teilen:

- Applikations Initialisierungs- und Update-Methoden
- Die einzelnen Bildschirme
  - Hauptbildschirm
  - Tracking
  - File, Gui, Pixy Test
  - Photo-Mode
- Hilfsfunktionen zum Steuern von Pixy
- Hilfsfunktionen zur PID Regelung

**Applikations Initialisierungs- und Update-Methoden**

Dateien: *app.c* und *app.h*
Hier gibt es nur 2 Funktionen *app_init()* und *app_process()*.
*app_init* sollte zu Beginn der Main-Methode in der jeweilgen Ziel-Plattform aufgerufen werden. *app_init* sorgt dafür das alle anderen Module initialisiert werden in dem es die *init* Funktion jedes Moduls aufruft.
*app_process* sollte innerhalb einer Endlosschleife im main der jeweiligen Ziel-Plattform aufgerufen werden. *app_process* zeichnet den aktuellen Bildschirm neu und ruft *system_process* auf, damit die Ziel-Plattform ihre Events abarbeiten kann.

**Hauptbildschirm**

Der Hauptbildschirm und alle anderen Bildschirme basieren auf der “Screen-Idee” welche im Kapitel

| |
|---|
| 7.6 beschrieben wird.<br><br>Beim Betreten des Screens (in der *enter* Methode)wird das komplette GUI erstellt. Dass heisst es werden Buttons erzeugt und registriert, welche zu den Unterbildschirmen führen. Zusätzlich werden 2 Logos vom Filesystem gezeichnet. Die *update* Methode des Screens ist leer, da der ganze Screen statisch ist. Beim verlassen (in der *leave* Methode) werden die erzeugten Buttons wieder unregistriert.<br><br>Wird auf ein Button geklickt, so wird mithilfe der *gui_screen_navigate M*ethode markiert dass jetzt zu einem Unterbildschirm gewechselt werden soll. Sobald die Applikation das nächste mal den Hauptloop betritt, wird dann der Bildschirm gewechselt, indem zuerst auf dem Hauptbildschirm *leave* aufgerufen wird und anschliessend auf dem neuen Bildschirm *enter* aufgerufen wird.<br><br>**Tracking Bildschirm**<br><br>**File, Gui, Pixytest**<br><br>**Photomode Bildschirme**<br><br>**Hilfsfunktionen zum Steuern von Pixy**<br><br>**Hilfsfunktionen zur PID-Regelung** |

| | |
|---|---|
| **Offene Punkte:** | |

## 7.2 System

| | |
|---|---|
| **Status:** | Stabil und abgeschlossen |
| **Doxygen-Dokumentation:** | http://t-moe.github.io/discoverpixy/group__system.html |
| **Quellen:** | Keine |
| **Zugehörige Quell- und Headerdateien:** | Ordner: common/system (100% Eigenentwicklung) |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| Dieses Modul enthält keinerlei Logik. Jede einzelne Funktion wird direkt an einen Low-Level Funktion mit exakt gleicher Signatur weitergeleitet. Grund: Eine Funktion wie sleep (Funktion die eine bestimmte Zeit wartet) kann nicht generisch implementiert werden, denn die Implementierung ist plattformabhängig. | |
| **Offene Punkte:** | Keine |

## 7.3 Tft

| **Status:** | Stabil und abgeschlossen |
|---|---|
| **Doxygen-Dokumentation:** | http://t-moe.github.io/discoverpixy/group__tft.html |
| **Quellen:** | Keine |
| **Zugehörige Quell- und Headerdateien:** | Ordner: common/tft (100% Eigenentwicklung) |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| 90% der Methoden dieses Moduls besitzen keine Logik und werden direkt an eine Low-Level Funktion mit gleicher Signatur weitergeleitet. Theoretisch könne man auch nur eine einzige Low-Level Funktion *ll_draw_pixel* erstellen und alle anderen Funktionen wie *draw_line* implementieren in dem man die Low-Level Funktion mehrfach aufruft. Aus performancegründen wurde jedoch darauf verzichtet. Die Ziel-Plattform können *draw_line* vermutlich effizienter implementieren als wir in der Lage wären dies im plattformunabhängigen Code zu tun.<br><br>Eine Ausnahme bilden die Methoden *tft_print_line* und *tft_print_formatted.* Diese dienen dazu Text auf das Display zu zeichnen. Die Ziel-Plattform muss dazu nur die Lowlevel Funktion *ll_tft_draw_char* bereitstellen. Diese wird dann mehrfach aufgerufen, für jedes Zeichen des Strings einmal.<br><br>Eine weitere Ausnahme bildet die Methode *tft_draw_bitmap_file_unscaled*. Diese dient dazu ein Windows Bitmaps (.bmp) vom Dateisystem auf das Display zu zeichnen. Dateisystem Operationen erfolgen durch das plattformunabhängige Filesystem Modul. Es macht daher keinen Sinn eine Lowlevel-Funktion zu erzeugen welche das Bitmap zeichnet, da ein Grossteil der Methode sowieso identische Aufrufe zum Filesystem Modul enthalten würde. | |
| **Offene Punkte:** | keine |

## 7.4 Touch

| **Status:** | Stabil und abgeschlossen |
|---|---|
| **Doxygen-Dokumentation:** | http://t-moe.github.io/discoverpixy/group__touch.html |
| **Quellen:** | • Grundlegende Idee mit Touch Library aus früherem Projekt Code siehe Github oder doc-Ordner: idpa_software.tar.gz |
| **Zugehörige Quell- und Headerdateien:** | Ordner: common/touch (Code aus Referenzprojekt zu 100% Überarbeitet) |
| **Durchgeführte Tests:** | |

| **Beschreibung:** | |
|---|---|
| Das Touch Modul stellt eine einfache API bereit zum Interagieren mit dem Touch-Screen bereit. Die Idee ist dass die Ziel-Plattform die Touch-Events (berühren, verschieben, loslassen) unverabeitet weiterreicht an das Touch-Modul. Das Touch-Modul rechnet die Koordinaten dann um (entsprechend der aktuellen Kalibrierung) und gibt Events weiter an die Interessenten. Benutzer des Touch-Moduls können für eine rechteckige Region auf dem Bildschirm (sogennante Touch_Area) ein Callback registieren. Zusätzlich kann spezifiziert werden bei welchen Arten von Event das Callback ausgelöst werden soll. Sobald dann die Ziel-Plattform ein Event ans Touch-Modul weiterreicht welches die gwünschte Region betrifft, wird das benutzerdefinierte Callback ausgeführt.<br><br>Dieses Modul wird in Anwendung (app Modul) selten direkt genutzt. Es dient eher als Grundlage für die Gui Elemente (siehe Kapitel 7.6). | |
| **Offene Punkte:** | • Pointer-Array durch Verkettet Liste ersetzen (siehe Kommentar im Dateikopf) |

## 7.5 Dateisystem

| | |
|---|---|
| **Status:** | Stabil und abgeschlossen |
| **Doxygen-Dokumentation:** | http://t-moe.github.io/discoverpixy/group__filesystem.html |
| **Quellen:** | • Petit FatFS http://elm-chan.org/fsw/ff/00index_p.html |
| **Zugehörige Quell- und Headerdateien:** | Ordner: common/filesystem (100% Eigenentwicklung) |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| Dieses Modul enthält keinerlei Logik. Jede einzelne Funktion wird direkt an einen Low-Level Funktion mit exakt gleicher Signatur weitergeleitet. In den Signaturen und den Strukturen stecken jedoch einige Überlegungen: Diese wurden so designt dass die Implementierung auf dem STM32F4Discovery möglichst einfach ist und dass sich die Low-Level Funktionen durch weiterleiten an Funktionen der Petit FatFs Library implementieren lassen. Petit FatFs ist eine FAT16 Implementierung für 8-bit Mikrokontroller, inkl SD-Karten Treiber. Da die Petit FatFs einige Einschränkungen besitzt, wurde auch die High-Level-Api (Code dieses Moduls) nicht enorm aufgeblasen. So können z.B. nur Dateien beschrieben werden die bereits existieren und ihre Grösse kann nicht verändert werden. Weiter wurde darauf geachtet, dass es auch möglich ist die Low-Level Funktionen im Emulator umzusetzen, in dem z.B. Filehandles übergeben werden können auch wenn Petit FatFs diese nicht braucht. Trotz dem dass die Funktionen und Strukturen nach dem Vorbild der Petit FatFs Implementierung designt wurden, befindet sich in diesem Modul keinerlei Fremdcode. | |

| | |
|---|---|
| **Offene Punkte:** | Keine |

## 7.6 Gui

| | |
|---|---|
| **Status:** | Stabil und abgeschlossen |
| **Doxygen-Dokumentation:** | http://t-moe.github.io/discoverpixy/group__gui.html |
| **Quellen:** | • Grundlegende Idee mit GUI Library aus früherem Projekt Code siehe Github oder doc-Ordner: idpa_software.tar.gz |
| **Zugehörige Quell- und Headerdateien:** | Ordner: common/gui (Referenzcode zu 100% Überarbeitet oder neu geschrieben) |
| **Durchgeführte Tests:** | |

**Beschreibung:**

Das Gui-Modul besteht aus mehreren Komponenten:

- Button
- Checkbox
- NummericUpDown
- Screen

Button, Checkbox und NummericUpDown basieren alle auf dem Touch Modul. Sie zeichnen ein grafisches Bedienelement mithilfe des TFT-Moduls welches durch Events von Touch-Moudul bedient werden kann. Der Code dieser Komponenten macht sich ein paar Techniken zunutze die im Kapitel 6.2 beschrieben sind (“Vererbung in C”). Das grundlegende Konzept hinter diesen Komponenten wurde aus dem Code von einem früheren Projekt übernommen (siehe Quellenangabe oben). Der Code wurde jedoch komplett überarbeitet und ausführlich kommentiert. Jede dieser Komponenten bietet ein Callback an, bei dem der User eine aufzurufende Funktion angeben kann für den Fall dass ein Event ausgelöst wird (z.B. Button oder Checkbox angeklickt).

Im Rahmen dieses Projekts wurde zusätzlich noch die Screen Komponente entwickelt. Die Idee dahinter kommt aus der Erkentnis dass grundsätzlich jede Ansicht (=Screen) aus den folgenden 3 Phasen besteht:

- Bildschirm betreten / Initialisierung (*enter* Methode)
- Wiederholtes aktualisieren / neu zeichnen (*update* Methode)
- Aufräumen / Bildschirm verlassen (*leave* Methode)

Desshalb wurde eine Struktur erstellt mit 3 Callback-Funktionen (*enter*, *update*, *leave*) , welche die Funktionalität eines einzelnen Screens kapseln sollen. Zwischen den Bildschirmen kann dann mithilfe

von Methoden des Screen Moduls navigiert werden. So gibt es z.B. eine Methode zum wechseln auf einen anderen Screen und eine Methode zum navigieren zum vorherigen Screen. Das Screen Modul stellt sicher dass ein Wechsel des Bildschirms immer aus dem Hauptloop ( *app_process()* )ausgeführt wird, selbst wenn der Auftrag zu wechseln in einem Interrupt erteilt wurde. Die Chronik der betretenen Screens wird in einer verketten Liste abgelegt. Die Liste stellt einen Stack dar, neue Screens werden oben auf den Stack gelegt. Beim zurücknavigieren wird der oberste Screen wieder entfernt und der zweitoberste wird angezeigt. Da ein Screen zu keinem Screen navigieren darf der bereits auf dem Stack liegt, kann die Screen Struktur gerade als Listenelement verwendet werden. Siehe Bemerkung im Kapitel 6.1 .

Das Wechseln des Screens läuft wie folgt ab:

- Die Methode *gui_screen_navigate* wird aufgerufen (z.B. in einem Button-Callback) um zu signalisieren dass der Screen gewechselt werden soll.
- Beim nächsten Aufruf von *gui_screen_update* (in dem Hauptloop) passiert anschliessend folgendes:
- Auf dem aktuellen Screen wird die *leave* Methode aufgerufen
- Der Screen wird gewechselt in dem auf dem neuen Screen die *enter* Methode aufgerufen wird
- Von nun an wird ein Aufruf von *gui_screen_navigate* (im Hauptloop) die *update* Methode des aktuellen Screens aufrufen, bis ein weiterer Screen-Wechsel erfolgen soll.

| | |
|---|---|
| **Offene Punkte:** | • Screen: Sicherstellen dass zu keinem Screen navigiert werden kann, der sich bereits in der Chronik befindet<br>• Button: 3D-Effekt verbessern in dem der Button um 1-Pixel verschoben wird während er gedrückt ist<br>• Button: Event hinzufügen der wiederholt ausgelöst wird wenn der Button längere Zeit gedrückt wird. Siehe auch nächsten Punkt:<br>• NummericUpDown: Zahl schneller hochzählen wenn der Button für eine längere Zeit gedrückt wird. |

## 7.7 Pixy

| | |
|---|---|
| **Status:** | Stabil und abgeschlossen |
| **Doxygen-Dokumentation:** | http://t-moe.github.io/discoverpixy/group__pixy.html |
| **Quellen:** | • Pixy libpixyusb (charmed labs): https://github.com/charmedlabs/pixy/tree/5561258623492034a19a47c420f255679f2522ae/src/host/libpixyusb |
| **Zugehörige Quell- und Headerdateien:** | Ordner:<br>• common/pixy (kleinere Anpassungen am Original)<br>• discovery/libs/Pixy (einige Anpassungen am Original)<br>• emulator/libs/Pixy (einige Anpassungen am Original ) |
| **Durchgeführte Tests:** | |

**Beschreibung:**

Als Grundlage für die Pixy-Kamera Ansteuerung diente die libpixyusb von den Pixy-Entwicklern. Die Library ist in C++ geschrieben und nutzt libusb um die Pixy-Kamera, welche am PC angeschlossen wird, anzusteuren. Die Ganze USB-Schicht befindet sich in einer Klasse *USBLink*. Zur USB Übertragung wird der Bulk-Modus verwendet. In der Mittelschicht befindet sich ein RPC-Framework namens *Chirp*, welches es erlaubt Funktionen entfernt auzufrufen, als wären sie lokal.

Folgende Änderungen wurden an der originalen Software gemacht:

- Alle printf statements wurden durch fprintf(stderr,.....) ersetzt um Debug-Meldungen auf den Standarterror-Stream umzuleiten
- Die Library wurde von Multithreaded auf Singlethreaded umgerüstet. Dazu wurde der Code-Block der normalerweise in einem eigenen Thread ausgeführt wurde in eine Methode ausgelagert *pixy_service()* welche nun vom Benutzer regelmässig aufgerufen werden muss. Betroffene Dateien: pixyinterpreter.cpp, pixyinterpreter.hpp, pixy.h, pixy.cpp
- Das öffentliche Headerfile pixy.h wurde bereinigt so dass es nichts mehr USB-Spezifisches enthält und keine unnötigen Includes.

Um dieses Modul plattformunabhängig zu machen wurde ein etwas anderer Ansatz gewählt als für die anderen Module: Die öffentlichen Headerdateien pixy.h und pixydefs.h wurden im Ordner common/pixy platziert. Die Implementierung ist dann aber komplett in die libs Ordner der entsprechenden Plattformen gelegt worden. Die entsprechenden Plattformen bauen aus dem Quellcode eine statische Library die dann beim bauen der entsprechenden Plattform dazugelinkt werden kann. Die öffentlichen Headerfiles sind komplett in C gehalten.Vom C++ Code ist also für die Applikation selbst nichts sichtbar.

| | |
|---|---|
| **Offene Punkte:** | • Hinzufügen einer Methode zum abfragen ob die Pixy-Kamera aktuell verbunden ist<br>• Hinzufügen einer Methode zum Anfragen und dekodieren |

| | |
|---|---|
| | eines Frames, damit nicht immer *pixy_command* aufgerufen werden muss. |

# 8 Plattform STM32F4Discovery

## 8.1 Aufbau und Pinbelegung

| | |
|---|---|
| **Status:** | Instabil |
| **Quellen:** | FSMC beispiel?<br>Stm datenblätter?<br>Display pinout? |
| **Zugehörige Quell- und Headerdateien:** | Keine |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| Grundlegendes Konzept des Aufbaus und Pinbelegung beschreiben<br><br>Probleme mit Doppelt belegten pins erklären, mögliche Lösungen angeben | |
| **Offene Punkte:** | • Print herstellen<br>• Mems Sensor auslöten |

## 8.2 System

| | |
|---|---|
| **Status:** | Stabil und abgeschlossen |
| **Quellen:** | • Beispielprojekte aus Atollic-Studio<br>• STM32F4xx_StdPeriph_Driver aus STM32F4-Discovery_FW_V1.1.0: http://www.st.com/web/en/catalog/tools/PF257904<br>• Newlib Stubs http://stm32discovery.nano-age.co.uk/open-source-development-with-the-stm32-discovery/getting-newlib-to-work-with-stm32-and-code-sourcery-lite-eabi |
| **Zugehörige Quell- und Headerdateien:** | Ordner: discovery/libs/StmCoreNPeriph (kaum angepasst, Original)<br>Dateien:<br>• discovery/main.c (Eigenentwicklung)<br>• discovery/ll_system.c (Eigentenwicklung) |

| | |
|---|---|
| | • discovery/startup.s (übernommen von Atollic)<br>• discovery/newlib_stubs.c (kaum angepasst, Original)<br>• discovery/system_stm32f4xx.c (übernommen von Atollic) |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| Stm beschreiben. StmCoreNPeriph beschreiben.<br>Initialisierungs prozedere (sysclock, systick) | |
| **Offene Punkte:** | Keine |

## 8.3 Tft

| | |
|---|---|
| **Status:** | Stabil und abgeschlossen |
| **Quellen:** | • Beispiel, SSD1289 Ansteuerung mit FSMC: http://mikrocontroller.bplaced.net/wordpress/?page_id=1357<br>• Grundlegende Zeichenfunktionen und Fonts aus früherem Projekt.<br>Code siehe Github oder doc-Ordner: idpa_software.tar.gz |
| **Zugehörige Quell- und Headerdateien:** | Dateien:<br>• discovery/ll_tft.c (90% Eigententwicklung, basierend auf Referenzcode Original aus idpa_software.tar.gz und Beispiel)<br>• discovery/font.c (kaum Anpassungen, Original aus idpa_software.tar.gz)<br>• discovery/font.h (kaum Anpassungen, Original aus idpa_software.tar.gz) |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| Lcd ansteuerung beschreiben. Auf controller verweisen. Quellen angeben (timo bms, fsmc aus dem netz) | |
| **Offene Punkte:** | • *ll_tft_draw_circle* und *ll_tft_draw_bitmap_unscaled* implementieren |

## 8.4 Touch

| | |
|---|---|
| **Status:** | Stabil und abgeschlossen |
| **Quellen:** | • Controller-Ansteuerung aus früherem Projekt.<br>Code siehe Github oder doc-Ordner: idpa_software.tar.gz |
| **Zugehörige Quell- und Headerdateien:** | Dateien:<br>• discovery/ll_touch.c (90% Eigententwicklung, basierend auf Referenzcode Original aus idpa_software.tar.gz) |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| SPI erklären, Controller ansteuerung beschreiben. | |
| **Offene Punkte:** | Keine |

## 8.5 Sd-Karte

| | |
|---|---|
| **Status:** | Nicht implementiert |
| **Quellen:** | • Petit FatFS<br>http://elm-chan.org/fsw/ff/00index_p.html |
| **Zugehörige Quell- und Headerdateien:** | Dateien:<br>• discovery/ll_filesystem.c (Praktisch leer) |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| SPI erklären, Controller ansteuerung beschreiben. | |
| **Offene Punkte:** | • Implementierung |

## 8.6 Pixy

| | |
|---|---|
| **Status:** | Stabil und abgeschlossen |
| **Quellen:** | Code aus Kapitel 7.7 |
| **Zugehörige Quell- und Headerdateien:** | Ordner: discovery/libs/Pixy<br>(100% Eigentenwicklung, nebst Code aus Kapitel 7.7 ) |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| Als Grundlage für die Pixy-Modul Implementierung für die Discovery-Platform diente der angepasste | |

Code von libpixyusb (siehe Kapitel 7.7 ). An dieser Anpassung wurde aber für die Discovery-Plattform noch einiges verändert: Für jegliche USB-Kommunikation, sowie für die Timer-Verwendung wurden Funktionsprototypen definiert:

```
int USBH_LL_open();
int USBH_LL_close();
int USBH_LL_send(const uint8_t *data, uint32_t len, uint16_t timeoutMs);
int USBH_LL_receive(uint8_t *data, uint32_t len, uint16_t timeoutMs);
void USBH_LL_setTimer();
uint32_t USBH_LL_getTimer();
```

Jegliche Funktionen der *USBLink* Klasse wurden "geleert" und mit einer einzigen Zeile befüllt die jeweils die enstprechende Funktion der oben definierten Prototyen aufruft. *USBLink::send* ruft jetzt also direkt *USBH_LL_send* auf. Die *USBLink* Klasse selbst enthält keine Logik mehr.

Die Implementierung dieser Prototypen hat an sich nichts mehr mit Pixy zu tun. Sondern nur noch mit Usb (im Bulk Modus). Desshalb wurden diese Funktionen auch im Usb Modul implementiert (siehe Kapitel 8.7).

Der Code des Pixy-Moduls für die Discovery-Plattform wird zu einer statichen Library kompiliert, welche später beim linken die von den öffentlichen Header-Files angeforderten Funktionalität bereitstellt.

| | |
|---|---|
| **Offene Punkte:** | Keine |

## 8.7 Usb

| | |
|---|---|
| **Status:** | Instabil |
| **Quellen:** | • STM32F4-Discovery_FW_V1.1.0: http://www.st.com/web/en/catalog/tools/PF257904<br>• STM32F4_USB_MP3 https://github.com/vanbwodonk/STM32F4_USB_MP3 |
| **Zugehörige Quell- und Headerdateien:** | Ordner:<br>• discovery/libs/StmUsbHost (kaum angepasst, Original)<br>Dateien:<br>• discovery/src/usb_bsp.c (übernommen aus USB Example)<br>• discovery/src/usbh_msc_core.c (etliche Anpassungen, Original aus MSC-Klasse)<br>• discovery/src/usbh_msc_core.h (kaum angepasst, Original aus MSC-Klasse)<br>• discovery/src/usbh_usr.c (kaum angepasst, Original aus MP3-Beispiel)<br>• discovery/src/usbh_usr.h (kaum angepasst, Original aus MP3-Beispiel) |

| **Durchgeführte Tests:** | |
|---|---|

**Beschreibung:**

Pixy nutzt zur Kommunikation via USB den Bulk Modus. Dafür müssen eine *send* und eine *receive* Funktion bereitgestellt werden. Leider gibt es für STM32 keine Beispiele von Applikationen die den Bulk Modus verwenden. Alle Beispiele bauen auf einem enorm grossen USB-Framework auf, welches mit sogenannten Geräteklassen funktioniert.

Usb definiert für gewisse Typen von Geräte eine Geräteklasse. So z.B. Maus, Kamera, Datenträger. Geräte dieser Klasse implementieren dann die definierten Schnittstellen-Funktionen der jeweilgen Klasse. Betriebssysteme unterstützen in der Regel die meisten dieser standartisierten Geräteklassen, dadurch ist für den Benutzer keine manuelle Treiberinstallation von nöten.

Die STM32 Usb Libraries sind in mehrere Packete unterteilt: Usb Device Library, Usb Host Library, Usb OTG Library. Die OTG Library wird in jedem Falle benötigt, wenn man als Usb-Host fungieren will braucht es zusätzlich die Host Library, wenn man als Endgerät fungieren will braucht es die Device Library. Die Host Library erfordert die Implementation einer USB Geräteklasse. Die Library ruft dann Callback-Funktionen der Klasse auf wann immer es aus Sicht der Library nötig ist. Die Usb-Library kontroliert also die Applikation und nicht umgekehrt. Will die Applikation von sich aus eine Übertragung auslösen so müssen die Daten zwischengespeichert werden, bis die Usb-Library eine Übertragung auslösen möchte.

Diese Philosophie passt ganz und gar nicht zu der Art und Weise wie die Pixy Usb-Kommunikation implementiert ist. Die libpixyusb möchte die Usb-Schnittstelle selbst kontrollieren und wenn der Benutzer eine Übertragung tätigen will jeweils einen Bulk-Transfer auslösen.

Um trotzdem eine Kommunikation mit der Pixy zu ermöglichen wurde eine etwas “unsaubere” Variante gewählt: Ein Beispiel zur Ansteuerung eines USB-Sticks (aus dem MP3 Projekt) wurde genommen und überarbeitet. Dabei wurde die dort implementierte MSC-Klasse “geleert” und die Funktionalität wie folgt angepasst: Die Geräteklasse liest die Serial_Number, Hersteller und Produktname und geht nur weiter in den “Klassenmodus” falls es sich um die Pixy handelt. Im Klassenmodus werden dann 2 Channels (senden/empfangen) zur Bulk-Kommunikation geöffnet. Ansonsten macht die Klasse nichts; die Callback-Funktionen der Usb-Library sind leer. Um Daten zu senden und empfangen wurden die Prototypen aus Kapitel 8.6 implementiert. Darin werden die *USBH_BulkSendData* und *USBH_BulkReceiveData* Methoden aufgerufen. Diese Funktionen sollten eigentlich nur von einer Geräteklasse bedient werden, welche sich strikte an die Abläufe/Spezifikation der Klasse hält. Insbesonders sollten sie wohl nicht von der Applikation aufgerufen werden, sondern nur innerhalb eines Callbacks der Usb-Library. Trotzdem scheint die Usb-Kommunikation mithilfe dieses “Tricks” in einigen Fällen zu funktionieren. Wieso dieser Trick in manchen Fällen nicht funktioniert, konnte noch nicht genau evaluiert werden. Problematisch ist insbesondere das Probleme mit USB nicht mithife eines normalen Debuggers inspziert werden können: Beim erreichen und fortfahren von einem Breakpoint vergeht soviel Zeit, dass die USB-Library das Gerät als “getrennt” betrachtet und von vorne begonnen werden muss.

| | |
|---|---|
| Es sollte grundsätzlich aber möglich sein eine fehlerfreie USB-Kommunikation zwischen STM32F4Discovery und Pixy-Kamera zu implementieren. Dafür wären aber einige weitere Mannwochen und eine tiefgehende Analyse der STM Usb-Libraries von nöten. | |
| **Offene Punkte:** | • Genauere Untersuchung des Problems bei der aktuellen Implementierung<br>• Tiefgehende Analyse der STM Usb-Libraries<br>• Implementierung korrekter send/receive Funktionen<br>• Korrektes behandeln von Ereignissen durch unerwartetes Trennen und erneutes Anschliessen von Pixy |

# 9 Plattform Emulator

## 9.1 Aufbau & System

| | |
|---|---|
| **Status:** | Stabil und abgeschlossen |
| **Quellen:** | Keine |
| **Zugehörige Quell- und Headerdateien:** | Dateien:<br>• emulator/qt/ll_system.cpp<br>• emulator/qt/main.cpp |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| Die Idee des Emulators ist ziemlich simpel: Alle Low-Level Funktionen die normalerweise von der Ziel-Plattform (STM32F4Discovery) implementiert werden sollten werden stattdessen mithilfe von QT auf dem PC realisiert. QT ist eine quellenoffene C++ Klassenbibliothek für die plattformübergreifende Programmierung von grafischen Benutzeroberflächen. Der gesamte Code wird dann zu einer statischen Library verpackt und zusammen mit dem plattformunabhängigen C-Code (aus dem common Ordner) und der pixy library (siehe Kapitel 9.4) zu einer ausführbaren Datei gelinkt.<br><br>Fenster und grafische Elemente können mit dem “QT-Designer” (Tool) zusammengeklickt werden. Zur Laufzeit darf nur aus dem Hauptthread darauf zugegriffen werden. Zusätzlich sollte der Hauptthread nicht blockiert werden, weil sonst das GUI einfriert.<br><br>Zu beginn der Main-Methode wird der Haupt Event-Loop von QT gestartet und anschliesend wird *app_init()* aufgerufen. Danach wird ein Thread gestartet welcher von nun an *app_process()* ausführt. Jegliche Zeichenoperationen die innerhalb von *app_process()* stattfinden zeichnen nur in einen Buffer, welcher dann später im GUI Thread auf den Bildschirm gezeichnet wird (siehe Kapitel 9.2). Die Main-Methode ist danach nur noch für das Verarbeiten von QT-Events zuständig. Das GUI bleibt | |

somit immer ansprechbar und reagierend.

Die Low-Level Funktionen dieses Moduls (System) sind enorm simpel:
*ll_system_init()* ist leer. Es ist keine spezielle Initialisierung von nöten, nebst der Initialisierung in der main Methode.
*ll_system_delay* ruft *QThread:msleep* auf, was den Thread (der normalerweise *app_process* aufruft) in den Schlafzustand versetzt.

*ll_system_process()* verarbeitet ausstehende QT-Events und schläft dann 1 Milisekunde um die CPU-Last in Grenzen zu halten.

| | |
|---|---|
| **Offene Punkte:** | Keine |

## 9.2 Tft & Touch

| | |
|---|---|
| **Status:** | Stabil und abgeschlossen |
| **Quellen:** | Keine |
| **Zugehörige Quell- und Headerdateien:** | Dateien:<br>• emulator/qt/ll_tft.cpp<br>• emulator/qt/ll_touch.cpp<br>• emulator/qt/mainwindow.cpp<br>• emulator/qt/mainwindow.h<br>• emulator/qt/mainwindow.ui |
| **Durchgeführte Tests:** | |

**Beschreibung:**

Mithilfe des QT-Designers (Tools) wurde ein einfaches GUI erstellt (*MainWindow* Klasse). Es besteht aus einem “Exit”-Button und einem Dropdown zur Auswahl der Zoomstufe. Innerhalb von *ll_tft_init()* wird das erstellte GUI anschliessend gestartet. Alle *ll_tft_** Funktionen werden direkt weitergeleitet an eine Methode der *MainWindow* Klasse mit gleicher Signatur. Innerhalb dieser Methoden wird dann mithilfe der *QPainter* Klasse in ein Bild-Buffer gezeichnet. Zusätzlich wird markiert dass das Fenster neugezeichnet werden muss. Im Paint-Event der *MainWindow* Klasse wird dann schliesslich der Bild-Buffer - unter Berücksichtigung der aktuellen Zoomstufe - gezeichnet. Der Paint-Event wird vom GUI-Thread aufgerufen, sobald das Fenster neu gezeichnet werden muss.

Das Touch Modul besitzt nur eine Low-Level Funktion, *ll_touch_init()* welche in diesem Falle leer ist. Tritt in der *MainWindow* Klasse ein *MouseEnter*, *MouseLeave* oder *MouseMove* Event auf, so wird die Maus-Position – unter Berücksichtigung der aktuellen Zoomstufe – umgerechnet und an die Funktion *touch_add_raw_event()* des Touch Moduls übergeben.

| | |
|---|---|
| **Offene Punkte:** | Keine |

## 9.3 Dateisystem

| **Status:** | Stabil und abgeschlossen |
|---|---|
| **Quellen:** | Keine |
| **Zugehörige Quell- und Headerdateien:** | Datei: emulator/qt/ll_filesystem.cpp |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| Die Dateisystem Implementierung (*ll_filesystem_** Funktionen) basieren auf den *QFile* Klassen. *QFile* bietet betriebssystemunabhängigen Zugriff auf das Dateisystem an. Der Stammorder des Dateisystems wird dabei im Ordner “emulated” (innerhalb vom emulator Ordner) angenommen. Alle Pfade werden also als realtiv zum Ordner “emulated” angenommen. Die Implementierung der einzelnen Methoden ist realtiv selbsterklärend und die nötige Dokumentation ist direkt den Code zu entnehmen. | |
| **Offene Punkte:** | • Überprüfen ob die angegeben Dateipfade wirklich unterhalb des “emulated”-Ordner liegen oder ob versucht wird aus dem Ordner “auszubrechen”. |

## 9.4 Usb & Pixy

| **Status:** | Stabil und abgeschlossen |
|---|---|
| **Quellen:** | Code aus Kapitel 7.7 |
| **Zugehörige Quell- und Headerdateien:** | Ordner: emulator/libs/Pixy<br>(übernommen aus Anpassung aus Kapitel 7.7 ) |
| **Durchgeführte Tests:** | |
| **Beschreibung:** | |
| Als Grundlage für die Pixy-Modul Implementierung für die Emulator-Platform diente der angepasste Code von libpixyusb (siehe Kapitel 7.7 ). An dieser Anpassung wurde grundsätzlich nichts weiter verändert für die Emulator-Plattform. Der Code wird zu einer statichen Library kompiliert, welche später beim linken die von den öffentlichen Header-Files angeforderten Funktionalität bereitstellt. | |
| **Offene Punkte:** | Keine |