

# Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

## (Embedded) C



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

---

### Aufgabe 1 [C] Die Programmiersprache C im Vergleich zu C++

---

Da C++ aus C entstand, sind viele Features von C++ nicht in C enthalten. Im Folgenden sollen die Hauptunterschiede verdeutlicht werden.

- Keine OO-Konzepte (Vererbung, ...)
- Strukturen (`struct`) statt Klassen (`class`)
- Keine Templates
- Keine Referenzen, nur Zeiger und Werte
- Kein `new` und `delete`, sondern `malloc()` und `free()` (`#include <stdlib.h>`)
- Je nach Sprachstandard müssen Variablen am Anfang der Funktion deklariert werden (Standard-Versionen bis einschließlich C99)
- Parameterlose Funktionen müssen `void` als Parametertyp haben, leere Klammern (bspw. `int foo();`) bedeuten, dass beliebige Argumente erlaubt sind.
- Keine Streams, stattdessen (`f`) `printf` zur Ausgabe auf Konsole und in Dateien (`#include <stdio.h>`)
- Kein `bool`-Datentyp, stattdessen wird 0 als `false` und alle anderen Zahlen als `true` gewertet
- Keine Default-Argumente
- Keine `std::string` Klasse, nur `char`-Arrays, die mit dem Nullbyte ('`\0`') abgeschlossen werden.
- Keine Namespaces

Da einige dieser Punkte sehr entscheidend sind, werden wir auf diese im Detail eingehen. Wichtig ist hierbei, dass alle im Folgenden vorgestellten Konzepte auch in C++ zur Verfügung stehen. Die Header der C-Bibliothek sind alle auch in C++ verfügbar. Möchte man bspw. `malloc` nutzen, kann man dies in C++ über `#include <stdlib.h>` oder über `#include <cstdlib>` (Allgemeines Muster: vorangestelltes 'c' und fehlendes '.h'). Im zweiten Fall sind alle Funktionen im Namensraum `std` eingebettet, man muss als `std::malloc` nutzen.

---

#### Aufgabe 1.1 Kein OO-Konzept

---

In C gibt es keine Klassen, weshalb die Programmierung in C eher Pascal statt C++ ähnelt. Stattdessen gibt es Strukturen (`struct`), die mehrere Variablen zu einem Datentyp zusammenfassen, was vergleichbar mit Records in Pascal oder – allgemein – mit Klassen ohne Methoden und ohne Vererbung ist.

Die Syntax dafür lautet

---

## Aufgaben zu Embedded C

---

```
struct MyStruct {  
    <Type1> <Name11>, <Name12>, ...;  
    <Type2> <Name21>, <Name22>, ...;  
};
```

Zum Beispiel

```
struct Point {  
    int x;  
    int y;  
};
```

Die Sichtbarkeit aller Attribute ist automatisch **public**. Um den definierten **struct** als Datentyp zu verwenden, muss man zusätzlich zum Namen das Schlüsselwort **struct** angeben:

```
void foo(struct Point *p) {  
    ...  
}  
  
int main(void) {  
    struct Point point;  
    foo(&point);  
}
```

Um den zusätzlichen Schreibaufwand zu vermeiden, wird in der Praxis oft ein **typedef** auf den **struct** definiert:

```
typedef struct Point Point_t;  
Point_t point;
```

Man kann die Definition eines **struct** auch direkt in den **typedef** einbauen:

```
typedef struct {  
    int x;  
    int y;  
} Point;
```

---

### Aufgabe 1.2 Kein new und delete

---

Anstelle von **new** und **delete** werden die Funktionen **malloc** und **free** verwendet, um Speicher auf dem Heap zu reservieren. Diese sind im Header **stdlib.h** deklariert.

```
#include <stdlib.h>  
  
// reserve memory for 10 points  
Point *points = (Point*) malloc(10 * sizeof(Point));  
// ...  
free(points);
```

Der Cast des Ergebnisses von **malloc** zu **Point\*** ist streng genommen nur notwendig, wenn du den gezeigten Code mit einem C++-Compiler kompilierst. Der C-Compiler nimmt die Konvertierung stillschweigend vor.

---

### Aufgabe 1.3 Ausgabe auf Konsole per printf

---

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner **./exercises/solutions/microcontroller\_c\_basics/3\_printf**.

Um Daten auf der Konsole auszugeben, kann die Funktion **printf** verwendet werden. **printf** nimmt einen Format-String sowie eine beliebige Anzahl weiterer Argumente entgegen. Der Format-String legt fest, wie die nachfolgenden

## Aufgaben zu Embedded C

Argumente ausgegeben werden. Mittels `\n` kann man einen Zeilenvorschub erzeugen. Um `printf` zu nutzen, muss der Header `stdio.h` eingebunden werden.<sup>1</sup> Der folgende Codeausschnitt zeigt, wie man Zahlen und Zeichen mit `printf` ausgeben kann. Wenn die Variablen `i` und `c` nicht definiert werden, ist die Aufgabe undefiniert. Zu beachten ist auch, dass `printf` nicht automatisch einen Zeilenumbruch einfügt.

```
#include <stdio.h>
printf("Hello World\n"); // Print 'Hello World' and add a line break

int i;
printf("i = %d\n", i); // Print integer
printf("i = %3d\n", i); // Print integer with a minimum width of 3

char c;
printf("c = %c, i = %d\n", c, i); // Print character and integer
```

Im Folgenden werden wir untersuchen, welche Folgen es hat, wenn man das „per Konvention“ erwartete Null-Byte ('`\0`') entfernt. In diesem Fall wird der Speicher Byte-weise solange ausgegeben, bis ein Null-Byte angetroffen wird.

- Beginne mit einer leeren `main`-Funktion.
- Lege einen Puffer der Größe 8 an:

```
char *buffer = (char*) malloc(8 * sizeof(char));
```

- Kopiere mittels `strcpy` (aus dem Header `string.h`) den String "`Hello!!`" in den Puffer:  
`strcpy(buffer, "Hello!!");`

- Nun gib den Inhalt des Puffers mittels `printf` aus:

```
printf("%s\n", buffer);
```

Wenn du das Programm jetzt kopierst und ausführst, sollte nur der String `Hello!!` erscheinen.

### Speicher auslesen (Heap)

Im Folgenden sehen wir uns den Effekt an, den das Weglassen des Nullbytes bewirkt. Überschreibe zunächst das Null-Byte mit einem beliebigen Zeichen (bspw. '+'):

```
buffer[strlen(buffer)] = '+'; // or: buffer[7] = '+'
```

Die Funktion `strlen` liefert die „gemessene“ Länge von `buffer` zurück, also die Anzahl an Bytes, bis zum Nullbyte. Wenn du jetzt den Inhalt des Puffers ausgibst, kann alles passieren („Undefined Behavior“). Die Funktion `printf` wird solange den Speicher auslesen, bis sie ein Null-Byte trifft. Du wirst allerdings nur das neu eingefügte Zeichen ('+') zusätzlich sehen, da das Betriebssystem den Heap-Speicher des Programms standardmäßig mit '`\0`' überschreibt.

Um einen beeindruckenderen Effekt zu sehen, musst du selber dafür sorgen, dass der Speicher „sinnvolle“ Daten enthält. Füge dazu den folgenden Code vor dem `malloc`-Aufruf von `buffer` ein.

```
char *dummyBuffer = (char*) malloc(19 * sizeof(char));
strcpy(dummyBuffer, "AlterTextAlterText");
free(dummyBuffer);
```

Wenn du jetzt erneut versuchst, über das Ende von `buffer` hinaus zu lesen, solltest du „Reste“ von `dummyBuffer` sehen. Falls du nur `Hello!!` siehst, kann es sein, dass der zweite `malloc`-Aufruf einen anderen Speicherbereich zurückliefert als der erste. Dies kannst du mithilfe der Formatierungsoption `%p` prüfen:

<sup>1</sup> Weitere mögliche Parameter etc. der Funktion `printf` findest du unter <http://www.cplusplus.com/reference/cstdio/printf/>.

## Aufgaben zu Embedded C

---

```
printf("Address of dummyBuffer: %p\n", dummyBuffer);
printf("Address of buffer:      %p\n", buffer);
```

Dieses Experiment zeigt, dass es sehr wichtig ist, bei der Manipulation von C-Strings gut aufzupassen – zahlreiche Sicherheitslücken basieren auf dieser Schwäche!

### Speicher auslesen (Konstanten)

Ändere nun die Art, wie der Puffer allokiert wird, wie folgt:

```
char *myString = "Hello!!";
```

Jetzt liegt der Puffer nicht mehr auf dem Heap (wie bei `malloc`) sondern im `data`-Segment. Wenn du nun versuchst, das Nullbyte von `"myString"` zu überschreiben, solltest du einen Speicherzugriffsfehler („Segmentation Fault“) erhalten, da Schreibzugriffe in diesem Speicherbereich verboten sind.

### Strings einkürzen

Es ist auch möglich, einen C-String *einzukürzen*, indem man das Null-Byte verschiebt innerhalb des Puffers nach vorne verschiebt.

- Verwende den Code aus dem vorherigen Abschnitt und diejenige Zeile an, in der der Puffer manipuliert wird. Statt `'_'` an Index 5 zu platzieren, platziere jetzt das Null-Byte (`'\0'`) an Index 2.
- Bei der Ausführung sollte jetzt nur noch `He` ausgegeben werden.

---

### Aufgabe 1.4 Strings zusammenbauen

---

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/microcontroller_c_basics/4_sprintf`.

Die Funktion `sprintf` dient dazu, formatierte Strings zusammenzusetzen und ist syntaktisch eng mit der Funktion `printf` verwandt.<sup>2</sup>

- Beginne mit einer leeren `main`-Funktion.
- Lege erneut einen Puffer auf dem Heap an:

```
char *buffer = (char*) malloc(100 * sizeof(char));
```

- Verwende den folgenden Aufruf, um einen Gruß mittels `sprintf` zusammenzusetzen. Lege dazu im Vorfeld eine Variable `name` an.

```
sprintf(buffer, "Hello, %s!\n", name);
```

- Den so gefüllten Puffer gibst du wie gewohnt über `printf` aus:

```
printf(buffer);
```

- Die Funktion `sprintf` kann natürlich auch zur Ausgabe (bspw.) von Zahlen und Zeichen genutzt werden:

```
sprintf(buffer, "c = %c, i = %3d", c, i);
```

Beachte auch hier, dass der Puffer auf dem Heap (statt im `data`-Segment) liegen muss, damit die Funktion `sprintf` hineinschreiben kann. Andernfalls erhältst du einen Segmentation Fault.

---

<sup>2</sup> Weitere mögliche Parameter etc. der Funktion `sprintf` findest du unter <http://www.cplusplus.com/reference/cstdio/sprintf/>.

## Aufgaben zu Embedded C

### Aufgabe 1.5 Zahlen formatiert ausgeben

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/microcontroller_c_basics/5_numbers`. Schreibe ein C-Programm, welches alle geraden Zahlen von 0 bis 200 formatiert ausgibt. Die Formatierung soll entsprechend dem Beispiel erfolgen:

```
2   4   6   8   10  
12  14  16  ...
```

Mache die Spaltenzahl und Spaltenbreite mithilfe von Variablen konfigurierbar, sodass es auch leicht möglich ist, 15 Spalten und/oder Zahlen bis 10 000 auszugeben.

### Aufgabe 1.6 Bit-Operatoren in C (und C++)

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/microcontroller_c_basics/6_BitAndLogicOperations`.

In dieser Aufgabe machst du dich mit den Bit-Operatoren (`&`, `|`, `~`, `>>`, `<<`) in C vertraut. Alle Operatoren können exakt gleich in C++ verwendet werden. Bit-Operatoren sind für ganzzahlige Typen definiert (bspw. `(unsigned) int`, `(unsigned) char`). Ein Bit-Operator bezieht sich dabei auf jedes einzelne Bit. Im Gegensatz dazu beziehen sich logische Operatoren (`||`, `&&`) immer auf den gesamten Wert.

Zum Experimentieren stellen wir dir eine Vorlage zur Verfügung: <https://github.com/Echtzeitssysteme/tud-cppp/blob/master/exercises/templates/BitAndLogicOperations/BitAndLogicOperations.c>. Die Funktion `fmt` wandelt ein Byte in einen String um, der das Bit-Muster darstellt. Zum Beispiel ist die Ausgabe von `fmt(23)` der String `"0b00010111"` ( $19 = 1 + 2 + 4 + 16$ ).

- Fülle zunächst die mit `// TODO implement me` gekennzeichneten Zeilen aus, sodass die Ausgabe korrekt ist. Beispiele sind für NEG und NOT gegeben.

Stelle sicher, dass deine Ergebnisse für  $a = 23$ ,  $b = 3$  mit den erwarteten Ergebnissen in folgender Tabelle übereinstimmen

Ausdruck	Ergebnis
<code>a AND b</code>	3
<code>a OR b</code>	23
<code>a XOR b</code>	20
<code>a RIGHT s</code>	5
<code>a LEFT s</code>	92

Die folgende Tabelle enthält die erwarteten Ergebnisse der logischen Operatoren für alle Wertkombinationen von  $a$  und  $b$ . Mit `IMP` ist Implikation gemeint ( $\Rightarrow$ ) und mit `BIIMP` ist Biimplikation/Äquivalenz gemeint ( $\Leftrightarrow$ ).

Ausdruck	Ergebnis			
	$a=1, b=1$	$a=1, b=0$	$a=0, b=1$	$a=0, b=0$
<code>a LAND s</code>	1	0	0	0
<code>a LOR s</code>	1	1	1	0
<code>a XOR s</code>	0	1	1	0
<code>a IMP s</code>	1	0	1	1
<code>a BIIMP s</code>	1	0	0	1

- Im Allgemeinen entspricht eine Verschiebung nach links/rechts einer Multiplikation mit/Division durch 2. Dazu werden jeweils von rechts-links 0-Bits eingeschoben. Unter welchen Umständen gilt dies nicht mehr? Was passiert,

## Aufgaben zu Embedded C

wenn du den Datentyp von a auf `unsigned char` setzt? Beobachte auch, ob das angezeigte Bitmuster mit dem ausgegebenen Wert übereinstimmt.

- Ändere den Wert der Variablen a in -1. Was passiert bei einem Rechts-Shift? Was bei einem Links-Shift? Wie unterscheidet sich das Verhalten im Vergleich zu positivem a?
- Ändere den Wert von s zu -1. Wie sieht das Ergebnis des Rechts-/Links-Shifts aus? Ein Shift um einen negativen Wert ist „Undefined Behavior“. Welche Ergebnisse sind demnach erlaubt?

### Aufgabe 1.7 Structs und Unions (optional)

Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

```
struct Channels { // 1 + 1 + 1 + 1 = 4 Bytes
    unsigned char b;
    unsigned char g;
    unsigned char r;
    unsigned char a;
};
```

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/microcontroller_c_basics/7_Unions`. In diesem Aufgabenteil sehen wir uns an, wie sich die Datentypen `struct` und `union` kombinieren lassen, um zwei Perspektiven auf ein Farbojekt einzunehmen: einmal kanalweise und einmal als gesamter 32-Bit-Farbwert. Lege zunächst eine `struct` mit Namen `Channels` an, welche die vier Farbkanäle der ARGB-Farbcodierung darstellt (mit alpha-, rot-, grün- und blauem Farbkanal).

Neben dem Datentyp `struct` gibt es auch `union`, welcher syntaktisch ähnlich ist. Der Unterschied ist, dass hier die einzelnen Felder vereint werden und sich den gleichen Speicherplatz teilen. Verändert man also eine der Variable, ändert sich der gemeinschaftlich genutzte Speicherplatz und der Wert der anderen Variablen entsprechend. Demnach ist eine `union` dann sinnvoll, wenn der Wert einer Variablen unabhängig vom Typ gleichviel Speicherplatz benötigt und im selben Speicherbereich abgelegt werden soll.

Die ARGB-Farbcodierung lässt sich statt in vier einzelnen Variablen auch als eine einzige 32-Bit-Integer-Variablen darstellen. Lege nun eine `union` mit Namen `Color` an, die eine `uint32_t`-Variable (Header `stdint.h`) und eine Variable `channels` vom Typ `Channels` enthält. Lege nun ein `Color`-Objekt `c` an, das die Farbe blau repräsentiert und setze danach den Grün-Kanal auf den Wert 0xF0 (wie im folgenden Quelltext dargestellt).

```
#include <stdio.h>
#include <stdint.h>

struct Channels { // 1 + 1 + 1 + 1 = 4 Bytes
    unsigned char b;
    unsigned char g;
    unsigned char r;
    unsigned char a;
};

union Color {
    uint32_t argb;
    struct Channels channels;
};

int main(void)
{
    union Color c;
    c.argb = 0xFF;
    // blau: 0000000 00000000 00000000 11111111
```

## Aufgaben zu Embedded C

```
c.channels.g = 0xF0;  
// grün und blau: 00000000 00000000 11110000 11111111  
  
    return 0;  
}
```

Um ein besseres Verständnis der Speicherverteilung in einem `Color`-Objekt zu erhalten, lasse dir nun die Größe und Inhalt des Attributs `argb` und der einzelnen Farbkanäle ausgeben. Nutze dazu `printf` wie zuvor. Was stellst du fest? Die Darstellung des Speichers in Abbildung 1 sollte dir weiterhelfen.

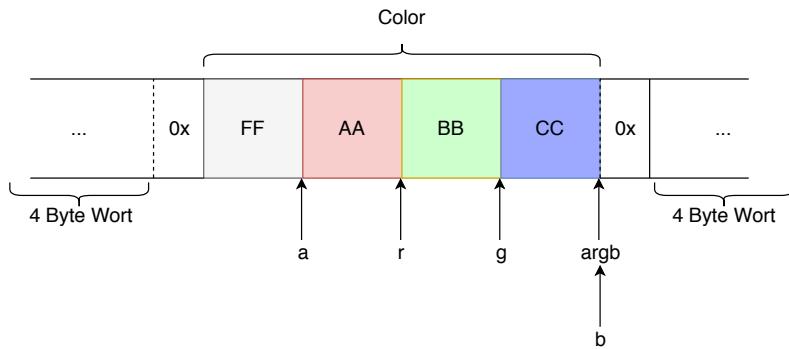


Abbildung 1: Skizze der Speicherverteilung eines `Color`-Objekts mit Farbwert 0xFFAABBCC

### Hinweise

- Mit `%x` kannst du eine Ganzzahl in Hexadezimaldarstellung formatieren.
- Mit `%p` kannst du eine Adresse formatieren.

### Aufgabe 1.8 C++-Aufgaben nachprogrammieren (optional)

Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

Dies ist eine freie Aufgabe, in der du versuchst, Programme der vergangenen Tage in reinem C auszudrücken. Der Schwierigkeitsgrad ist dabei sehr unterschiedlich!

### Niedrigerer Schwierigkeitsgrad

- **Sternen- oder Buchstabenmuster** ausgeben (??): Diese Aufgabe ist sehr ähnlich zu Aufgabe 1.5.
- **Werte analysieren und mit Arrays arbeiten** (?? und ??): Abgesehen von der fehlenden Unterstützung für Referenzen sollten die Ergebnisse sich nicht unterscheiden.
- **Funktionszeiger** (?? und ??): Funktionszeiger in C arbeiten genauso wie Funktionszeiger in C++. Du kannst dies testen, indem du die Programmlogik der vorigen Aufgabe in eine separate Funktion auslagerst, die neben der Spaltenbreite und Obergrenze der anzuzeigenden Zahlen zusätzlich einen Funktionszeiger-Parameter hat, der festlegt, was mit der jeweiligen Zahl vor der Ausgabe geschehen soll (bspw. verdoppeln, quadrieren).

### Höherer Schwierigkeitsgrad

- **Vector** (??): C bietet keine Objektorientierung, aber du kannst eine `struct` zur Datenhaltung anlegen und die Methoden der Klasse als Funktionen realisieren, deren Namen bspw. immer mit dem Präfix `Vector_` beginnen und die als zusätzlichen Parameter einen Pointer auf eine `Vector-struct` erhalten.

## Aufgaben zu Embedded C

---

- **Verkettete Listen (??):** Die reinen Datenstrukturen für die Liste (`List`) und für Listenelemente (`ListItem`) lassen sich als `struct` abbilden. Methoden kannst du wieder auf Funktionen mit Namenskonvention und zusätzlichem Pointer-Parameter abbilden.
- **Generischer Vektor und Liste (?? und ??):** Auch wenn es in C keinen eingebauten Mechanismus wie die C++-Templates gibt, kannst du die `Vector` und `List`-Klasse generisch machen, indem du die Einträge des Vektors bzw. den `content` der `ListItems` als `void*` deklarierst.
- **Eigene Array-Klasse (??):** Ebenso wie bei generischem Vektor und Liste kannst du natürlich auch deine eigene Array-Klasse schreiben.

 Die folgendenden Aufgaben sind nur zur Bearbeitung mit dem ausleihbaren Evaluationsboard gedacht. 

### Aufgabe 2 [C] Testprogramm auf den Microcontroller laden (optional)

---

Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

Alle nun folgenden, mit [C] markierten Aufgaben sind nicht klausurrelevant. Sie dienen dazu, dich in die Welt der Embedded-C-Programmierung einzuführen. Embedded C ist hierbei genau die gleiche Programmiersprache wie C. Jedoch gibt es einige technische Besonderheiten zu berücksichtigen.

#### Aufgabe 2.1 Überblick

---

Für die Arbeit mit dem Evaluationsboard nutzen wir die Entwicklungsumgebung *WinIDEA Open*<sup>3</sup>. Im Vergleich zu Code-Lite ist diese Umgebung speziell auf die Entwicklung von Embedded C zugeschnitten. Der Bauprozess für Embedded-C-Programme sieht teilweise anders aus als bei C++-Programmen:

- a) Das Ergebnis der **Link-Phase** ist kein auf dem PC ausführbares Programm, sondern ein sogenanntes *Image*. Dieses Image wird in den statischen Speicher des Microcontrollers geladen.
- b) Nach der Link-Phase folgt die **Flash-Phase** (in WinIDEA auch „Download“ genannt). Während dieser Phase wird das Image auf den Microcontroller übertragen.
- c) Anschließend beginnt die **Ausführung** direkt oder man muss den *Reset*-Knopf des Boards drücken, um den Programmzähler zurückzusetzen.
- d) Standardmäßig geht WinIDEA hierbei direkt in den **Debug-Modus**. Das bedeutet, dass die Ausführung des Programms am Beginn der `main`-Funktion angehalten wird.

Die weiteren Besonderheiten vom Embedded C sehen wir uns anhand eines einfachen Testprogramms an.

#### Aufgabe 2.2 Testprogramm

---

Für diese und alle weiteren Aufgaben stellen wir dir ein Codetemplate zur Verfügung, das von dir ergänzt wird. Wir beginnen mit einem kleinen fertigen Programm, das die RGB-LED des Evaluationsboards periodisch blinken lässt. Dies ist das „Hello World“-Programm der Embedded-C-Welt.

<sup>3</sup> <http://www.isystem.com/download/winideaopen>

## Aufgaben zu Embedded C

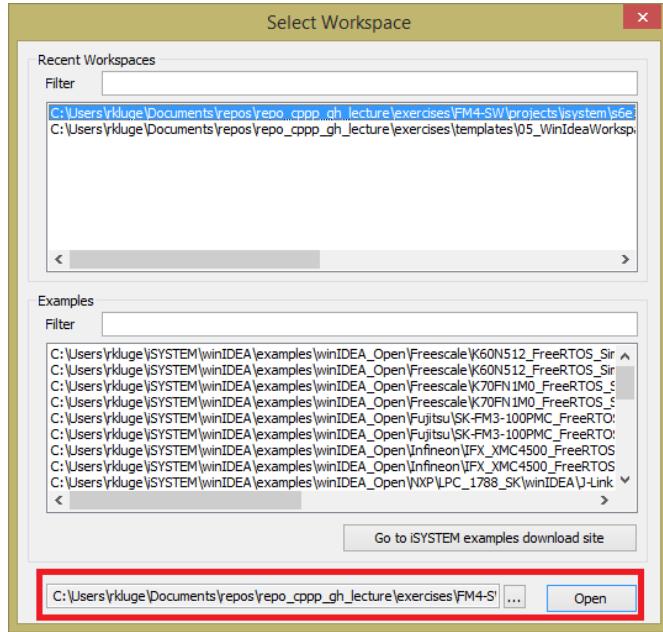


Abbildung 2: Workspace-Auswahl in WinIDEA

- a) Kopiere zunächst den **vorbereiteten WinIDEA-Workspace** (`/exercises/templates/05_WinIdea-WorkspaceTemplate`) in ein Verzeichnis **außerhalb** deines Benutzerverzeichnisses (bspw. nach `C:\tmp`).  
<sup>4</sup> Falls du deinen eigenen PC verwendest, ist es sehr wichtig, dass der WinIDEA-Workspace und die verwendeten Bibliotheken (bspw. `C:\PortableApps\Cypress\PDL`) auf dem gleichen Laufwerk liegen.
- b) **Öffne WinIDEA:** Das Programm liegt unter `C:\PortableApps\iSYSTEM\winIDEAOpen9\winIDEA.exe`. Bei Bedarf kannst du dir eine Desktop-Verknüpfung erstellen.
- c) **Wähle** in dem erscheinenden Dialogfenster den soeben **kopierten WinIDEA-Workspace** aus wie in Abbildung 2 gezeigt (roter Rahmen).
- d) Nun öffnet sich der eigentliche Workspace. Du findest links einen baumartig aufgebauten Datei-Browser, der die Datei-Gruppen **lib**, **src** und **Dependencies** enthält.
  - Die Gruppe **lib** enthält manuell konfigurierte Abhängigkeiten deines Projekts, die du normalerweise nicht anpassen musst.
  - Die Gruppe **src** enthält den Quelltext, mit dem du arbeiten wirst.
  - Die Gruppe **Dependencies** enthält automatisch von WinIDEA aufgelöste Abhängigkeiten und muss nicht angepasst werden.

Öffne nun die Datei **main.c** in der Gruppe **src**.

- e) Hier findest du die **main-Funktion**, deren Inhalt du im Folgenden immer wieder anpassen wirst. Im Moment delegiert sie an die Funktion `BlinkMain`. Diese ist in der Datei `blink.h` deklariert und in `blink.c` definiert.  
Öffne nun die Datei **blink.c**.
- f) Du findest den folgenden Quelltext (Listing 1) vor:

```
1 #include "blink.h"
2
```

<sup>4</sup> Leider ist es aus technischen Gründen nicht möglich, mit WinIDEA im Benutzerverzeichnis zu arbeiten, da dieses auf ein Netzlaufwerk abgebildet wird.

## Aufgaben zu Embedded C

---

```
3 #include <stdint.h>
4 #include "delay.h"
5 #include "s6e2ccxj.h"
6
7 #include "pins.h"
8
9 int BlinkMain() {
10     LED_BLUE_DDR |= (1 << LED_BLUE_PIN); // Configure blue LED pin as output.
11     LED_BLUE_DOR |= (1 << LED_BLUE_PIN); // Turn LED off.
12
13     const uint32_t sleepTime = 1000000;
14
15     // Main loop
16     while (1) {
17         // Clear bit -> Switch LED on
18         LED_BLUE_DOR &= ~(1 << LED_BLUE_PIN);
19         cppp_microDelay(sleepTime);
20
21         // Set bit -> Switch LED off
22         LED_BLUE_DOR |= (1 << LED_BLUE_PIN);
23         cppp_microDelay(sleepTime);
24     }
25 }
```

**Listing 1:** blink.c

- Zu Beginn wird der Pin, an den die blaue LED angeschlossen ist, als Ausgang konfiguriert (Zeile 10).
  - Dann wird der Pin auf 1 gesetzt, was die LED ausschaltet (Zeile 11).
  - In einer Endlosschleife wird die LED dann immer wieder ein- und ausgeschaltet. Zwischen den Schaltvorgängen wird eine Pause von einer Sekunde eingelegt.
- g) Um das Projekt jetzt zu **compilieren** und zu **linken**, wähle im Menü *Project* den Befehl *Make* (oder drücke F7). Dieser Befehl wird – seinem Namen entsprechend – nur diejenigen Teile neu kompilieren, die sich seit dem letzten Aufruf verändert haben. Mit dem Befehl *Project → Rebuild* wird das Projekt von Grund auf neu gebaut. Der Vorgang sollte im *Output*-Fenster<sup>5</sup> mit 0 Error (s) 0 Warning (s) enden.
- h) Um das erzeugte Image auf den Microcontroller zu flashen, verbinde den **Micro-USB-Anschluss CN2** des Boards mit dem **Data-USB-Port** des PCs. Die grüne LED in der Nähe des 2x5-Pin-Multicon-Blocks (CN12) sollte nun dauerhaft leuchten. Wähle anschließend in WinIDEA **Debug → Download** (oder Ctrl + F3).
- i) In WinIDEA wird nun ein **gelber Pfeil** neben der Signatur der **main-Funktion** erscheinen. Dies deutet an, dass du jetzt auf dem Microcontroller debuggen kannst.<sup>6</sup> Setze die Ausführung einfach fort, indem du **Debug → Run Control → Run** wählst (oder F5). Die blaue LED sollte nun blinken.
- j) Falls dein Programm einmal „unsauber“ startet oder du es neu starten möchtest, kannst du den Programmzähler mithilfe des **Reset Buttons** zurücksetzen. Er befindet sich oberhalb der MCU und ist auch mit „Reset“ beschriftet. Ganz in der Nähe befindet sich auch der **User Button**, den wir im Folgenden noch kennenlernen werden.

**Herzlichen Glückwunsch – du bist nun bereit, selber Hand an die Aufgaben zu legen!**

---

### Aufgabe 2.3 LED bunt blinken lassen

---

Als erste eigenständige Aufgabe erweiterst du die Funktion `BlinkMain` so, dass abwechselnd alle drei möglichen LEDs angesteuert werden. Führe dazu die folgenden Schritte aus:

<sup>5</sup> Kann über *View → Output* oder Alt+2 geöffnet werden.

<sup>6</sup> Dieses Verhalten lässt sich auch ausschalten: <https://github.com/Echtzeitsysteme/tud-cppp/wiki/WorkingWithWinIDEAOpen#how-can-i-prevent-winidea-from-stopping-at-main-after-downloading>

## Aufgaben zu Embedded C

---

- a) Öffne die Dateien Dateien `blinkrainbow.h` und `blinkrainbow.c` (in der Gruppe `src`).
- b) Nutze die Dateien `blink.h` und `blink.c` als Ausgangspunkt, um diese Aufgabe zu lösen.
- c) Erweitere nun den Code so, dass du auch auf die Ports der roten und grünen LED zugreifst und die LED abwechselnd rot, grün und blau leuchtet.

Listing 2 zeigt am Beispiel der blauen LED und des linken Joystick-Buttons wie man auf IO-Pins zugreifen kann. Die Bit-Operatoren wurden bereits in Aufgabe 1.6 behandelt. Die Namen der verschiedenen Register für die LEDs und Buttons sind in der Datei `pins.h` definiert. Ausdrücke wie `LED_BLUE_DDR |= (1 << LED_BLUE_PIN)` oder `BUTTON_LEFT_DDR &= ~(1 << BUTTON_LEFT_PIN)` werden verwendet, um ein einzelnes Bit in einem Register zu setzen oder zu löschen.

```
1 #include "s6e2ccxj.h"
2 #include "pins.h"
3
4 int io_example(void)
5 {
6     /*
7     Configure the pin of the blue LED as output by setting the corresponding
8     bit in the Data Direction Register (DDR).
9     */
10    LED_BLUE_DDR |= (1 << LED_BLUE_PIN);
11    /*
12    Set the pin to 1 by setting the corresponding bit in the Data Output
13    Register (DOR). This turns the LED off.
14    */
15    LED_BLUE_DOR |= (1 << LED_BLUE_PIN);
16
17    /*
18    Configure the pin of the left Joystick Button as input by clearing the corresponding
19    bit in the Data Direction Register (DDR).
20    */
21    BUTTON_LEFT_DDR &= ~(1 << BUTTON_LEFT_PIN);
22    /*
23    Enable the pull-up resistor in the pin by setting the corresponding bit
24    in the Pullup Configuration Register (PCR).
25    */
26    BUTTON_LEFT_PCR |= (1 << BUTTON_LEFT_PIN);
27
28    /*
29    Check the pin status by combining the Data Input Register (DIR) with the
30    corresponding bitmask.
31    The expression is inverted because the pin is 0 when the button is pressed.
32    */
33    if (!(BUTTON_LEFT_DIR & (1 << BUTTON_LEFT_PIN))) {
34        /*
35        Toggle the LED when the button is pressed. Toggling is done by XORing
36        the current pin state with 1.
37        */
38        LED_BLUE_DOR ^= (1 << LED_BLUE_PIN);
39    }
40 }
```

**Listing 2:** Beispiele zur Ansteuerung von Ein-/Ausgabepins

### Hinweise

- Für diese und die folgenden Kennenlernaufgaben stellen wir dir Musterlösungen direkt im WinIDEA-Workspace bereit. Du findest sie in der Gruppe `solution`. Beachte, dass die (nicht-statischen) Funktionen der Musterlösung und deren entsprechende Implementierungsdateien stets auf `_s` enden, um Namenskonflikte mit den Vorlagefunktionen zu vermeiden.

## Aufgaben zu Embedded C

### Aufgabe 3 [C] Taster abfragen (optional)

Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

In dieser Aufgabe erweiterst du die vorherige Aufgabe um eine Benutzerinteraktion über den Taster des linken Joysticks (**Joystick 1**). Ziel dieser Aufgabe ist es, mithilfe des Tasters die RGB-LED in zwei verschiedenen Szenarien zu kontrollieren. Im ersten Szenario soll der Taster als Lichtschalter arbeiten: Wird der Taster einmal betätigt, schaltet sich die blaue LED ein; wird der Taster erneut betätigt, schaltet sie sich wieder ab. Im zweiten Szenario soll die blaue LED solange leuchten, wie der Taster gedrückt gehalten wird.

- a) In dieser Aufgabe wirst du mit den Dateien `button.c` und `button.h` arbeiten.
- b) Zunächst werden wir die nötigen Variablen in `button.c` deklarieren: Um den aktuellen Zustand der LED zu speichern wird eine vorzeichenlose 8-Bit-Integer-Variablen `ledStatus` angelegt.
- c) Implementiere zunächst die Funktion `initLED()`. Diese soll `ledStatus` und den Pin der blauen LED initialisieren.
  - Der LED-Status soll zu Beginn 0 (= „aus“) sein.
  - Der Pin der blauen LED muss als Ausgang konfiguriert werden.
  - Das Daten-Register der blauen LED soll entsprechend initialisiert sein, dass die LED ausgeschaltet ist.
- d) Implementiere nun die Funktion `toggleBlueLED()`. Diese soll den Status von `ledStatus` umkehren: War der Wert zuvor 1, soll er danach 0 sein und umgekehrt. Der aktuelle Status der LED soll mit `setBlueLED(uint8_t status)` gesetzt werden.
- e) Implementiere nun die Funktion `isButtonPressed`, die zurück gibt, ob der Taster gerade gedrückt ist. Beachte, dass der Taster durch den Pull-Up-Widerstand genau dann gedrückt ist, wenn am Pin ein niedriger Pegel (0) anliegt. Ein Beispiel für die Abfrage des Tasters findest du in Listing 2.
- f) Implementiere nun mithilfe der zuvor erstellten Hilfsfunktionen die Hauptfunktionen `ButtonToggleBlueLED()` und `ButtonHoldBlueLEDOn()`. Die erste soll dem Button die Funktion eines Lichtschalters geben und die zweite soll die LED zum Leuchten bringen, solange der Taster gedrückt gehalten wird.

### Aufgabe 4 [C] Display ansteuern (optional)

Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

In diesem Abschnitt lernst du die Ansteuerung des Displays kennen. Das Display hat eine Auflösung von 480 \* 320 Pixeln. Zunächst wirst du den Bildschirm in verschiedenen Farben ausfüllen und die dafür benötigte Farbcodierung kennenlernen. Anschließend wirst du Funktionen implementieren, um auf dem Bildschirm regelmäßige Muster und Texte auszugeben. Alle in dieser Aufgabe verwendeten Funktionen **müssen in der Datei `display.c` implementiert werden**. Tabelle 1 dokumentiert das erwartete Verhalten der zu implementierenden Funktionen.

#### Aufgabe 4.1 Bildschirm umfärben

Das Display verwendet eine RGB565-Codierung für Farben (auch als „High Color“ bekannt<sup>7</sup>). Bei der RGB565-Codierung werden 5 Bits für Rot, 6 Bits für Grün und 5 Bits für Blau verwendet (siehe Abbildung 3).

Um auch eigene Farben nutzen zu können, implementierst du zunächst die Funktion `cppp_color565(uint8_t r, uint8_t g, uint8_t b)`, welche RGB888-Farben in RGB565-Farben umwandelt. Der Umwandlungsalgorithmus arbeitet für drei gegebene Bytes (`uint8_t` oder `unsigned char`) `r`, `g`, `b` wie folgt:

<sup>7</sup> [https://en.wikipedia.org/wiki/High\\_color](https://en.wikipedia.org/wiki/High_color)

## Aufgaben zu Embedded C

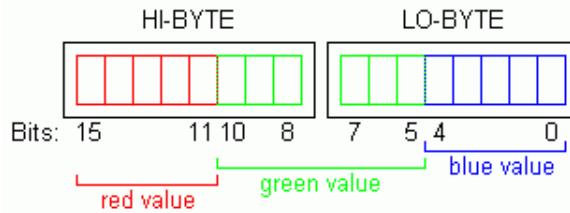


Abbildung 3: RGB565 Codierung

- a) Extrahiere die höchstwertigen 5 Bits von `r`, die höchstwertigen 6 Bits von `g` und die höchstwertigen 5 Bits von `b`. Nutze den Bit-Und-Operator mit einer passenden Bitmaske, um nur die entsprechenden Bits beizubehalten und alle anderen Bits auf 0 zu setzen. Verwende anschließend den Shift-Operator, um die extrahierten Bits ans untere Ende des Bytes zu schieben. Es folgt eine Skizze für den Rot-Kanal.

```
uint8_t r          = 0xA7;      // --> 0b10100111
uint8_t hiR        = r ____;   // --> 0xA0; 0b10100000
uint8_t hiRShifted = r ____; // --> 0x14; 0b00010100
```

- b) Füge die extrahierten Bits mittels des Bit-Oder-Operators zusammen, um den RGB565-Wert zu erhalten: Beachte dabei, dass du mithilfe des Links-Shift-Operators die einzelnen Teile korrekt ausrichtest. Beispielsweise muss `hiRShifted` vor der Verroderung um 11 Bits nach links verschoben werden. Beachte, dass das Ergebnis 16 Bit lang sein muss und wir deshalb den Typ `uint16_t` verwenden.

```
uint16_t result=(hiRShifted ____) | (hiGShifted ____ ) | (hiBShifted ____);
```

Tabelle 1: Wichtige Funktionen und Variablen für das Display

Funktionen/Variablen	Beschreibung
<code>void cppp_drawRect(int16_t x, int16_t y, int16_t w, int16_t h, uint16_t color)</code>	Zeichnet die Umrandung eines Rechtecks in der Farbe <code>color</code> mit der Breite <code>w</code> und der Höhe <code>h</code> an die Stelle <code>x,y</code> .
<code>void cppp_fillRect(int x1, int y1, int w, int h, uint16_t fillcolor)</code>	Zeichnet ein ausgefülltes Rechteck in der Farbe <code>fillcolor</code> mit der Breite <code>w</code> und der Höhe <code>h</code> an die Stelle <code>x,y</code> .
<code>void cppp_drawPixel(int16_t x, int16_t y, uint16_t color)</code>	Zeichnet ein Pixel an <code>x,y</code> mit der Farbe <code>color</code> .
<code>void cppp_fillScreen(uint16_t color)</code>	Füllt den gesamten Bildschirm mit der Farbe <code>color</code> .
<code>uint16_t cppp_color565(uint8_t r, uint8_t g, uint8_t b)</code>	Umwandlung von RGB in HEX 565.

Tabelle 2: RGB565-Farbwerthe

Farbe	RGB888	RGB565
Cyan	0x00EAFF	0x075F
Rosa	0xFC00FF	0xF81F
Orange	0xFFB400	0xFDA0

## Aufgaben zu Embedded C

---

Zum Testen deiner Implementation kannst du die Vergleichswerte in Tabelle 2 nutzen. Gehe dazu wie folgt vor:

a) Lege dir eine leere main-Funktion an und binde den Header `display.h` ein.

b) Füge die folgende Zeile ein, um deinen Code für die Farbe Cyan zu testen:

```
color565(0x00, 0xEA, 0xFF);
```

c) Nun erstelle das Projekt wie gewohnt, aber, bevor du das Programm auf den Mikrocontroller lädst, setze einen *Breakpoint* in `color565`. Öffne dazu die Datei `display.c`. Rechtsklicke in die Zeile, in der die Variable `result` zurückgegeben wird. Wähle *Set Breakpoint* (oder drücke F9). Links neben der Zeile erscheint ein rotes Rechteck.

d) Lade nun den Code auf den Microcontroller. Die Ausführung sollte an dem eingestellten Breakpoint anhalten (gelber Pfeil).

e) Bewege nun den Mauszeiger über die Variable `result`. Es erscheint ein kleiner Tooltip mit dem Inhalt `result = 1887`. Dies ist das richtige Ergebnis, aber leider in einer nicht-hexadezimalen Darstellung.

f) Um dies zu ändern, öffne die *Watch View* mittels *View → Watch* (oder Alt + 3).

g) Betätige die dritte Schaltfläche von links („Toggle Hex Mode“) am oberen Rand des *Watch*-Fensters.

h) Wenn du mit dem Mauszeiger nun erneut auf `result` zeigst, erfährst du, dass `result=0x075F`.

i) Um das Programm weiterlaufen zu lassen, wähle *Debug → Run Control → Run* (oder F5).

Die Datei `display.h` enthält einige vordefinierte Farben, die du ebenfalls bei den folgenden Aufgaben nutzen kannst: BLACK (0x0000), BLUE (0x001F), RED (0xF800), GREEN (0x07E0), YELLOW (0xFFE0), WHITE (0xFFFF).

Im letzten Teil dieser Aufgabe färbst du den Bildschirm mit der Farbe deiner Wahl.

a) Beginne wieder mit einer leeren main-Funktion.

b) Binde die Header `init.h` (aus `lib`) ein und füge den folgenden Initialisierungscode für das Board in `main` ein:

```
initBoard();
```

c) Um das Display in Cyan einzufärben, füge folgende Zeile an:

```
fillScreen(color565(0x00, 0xEA, 0xFF));
```

---

### Aufgabe 4.2 Regelmäßiges Muster ausgeben

---

In diesem Abschnitt implementierst du die Funktion `void printPattern(backgroundColor, foregroundColor)`. Diese Funktion ordnet auf dem Display Quadrate (4 x 4 Pixel) als Schachbrettmuster an.

#### Hinweise

- Nutze die Funktion `cppp_fillScreen`, um zunächst den Bildschirm in der Farbe `backgroundColor` zu füllen. Nutze anschließend die Funktion `fillRect`, um die einzelnen Quadrate in der Farbe `foregroundColor` zu erzeugen.
- Eine einfache Möglichkeit ist, zwei geschachtelte `for`-Schleifen zu verwenden, die den Schleifenzähler in X-Richtung jeweils um die Blockgröße und in Y-Richtung um die doppelte Blockgröße weiterschalten.

---

### Aufgabe 4.3 Text ausgeben

---

In diesem Aufgabenteil wirst du einen Cursor implementieren, der es erlaubt, auf einfache Art und Weise Zeichen auf dem Bildschirm auszugeben. Zum Anzeigen von Buchstaben auf dem Bildschirm stellen wir dir eine kleine Fontbibliothek

## Aufgaben zu Embedded C

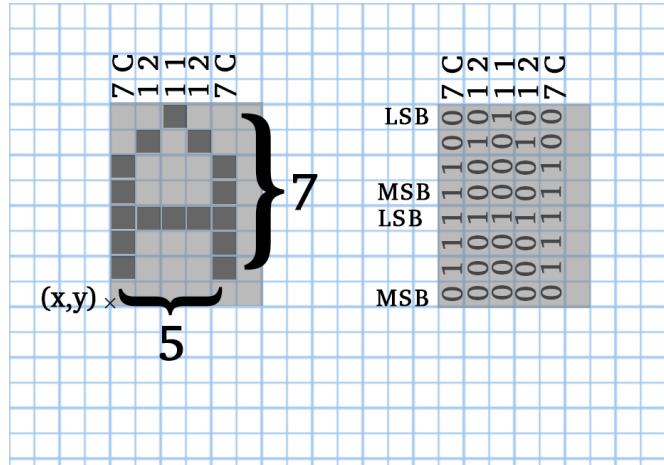


Abbildung 4: Schematische Darstellung des Zeichens 'A' in einer ASCII-5\*7-Schrifart

im Extended-ASCII 5\*7-Format zur Verfügung. Du findest diese in der Datei `glcdfont.h` (Gruppe lib). Die Bibliothek besteht aus einem Array mit 255 Buchstaben, die jeweils als 5 Bytes abgespeichert sind (siehe Abbildung 4). Das 5\*7-Format eignet sich für den 480 \* 320 großen Bildschirm, da die Darstellung dieser Schrifart mit einer Fläche von 35 Pixeln je Buchstaben immer noch gut lesbar ist. Wenn man von einem Pixel Abstand zwischen den Buchstaben ausgeht, passen auf das Display bis zu 3200 Zeichen.

Du wirst in dieser Aufgabe mehrere Funktionen in der Datei `display.c` implementieren. Der Koordinatenursprung (0,0) des Bildschirms ist links unten. In der Vorlagendatei `display.c` ist die Fontbibliothek `glcdfont.h` bereits eingebunden. Um globale Grundeinstellungen für die Schrifart festzulegen, wurden bereits die Variablen `cursorX`, `cursorY`, `textColor`, `textSize` und `textBackground` deklariert.

- a) Implementiere die Setter für den Cursor (`setCursor(int16_t x, int16_t y)`), die Textfarbe (`setTextColor(uint16_t c)`), die Textgröße (`setTextSize(uint8_t s)`) sowie die Hintergrundfarbe (`setBackgroundColors(int bg)`).

Zudem sollen in der Funktion `initCursor` folgende Grundeinstellungen vorgenommen werden: Der Cursor ist im Koordinatenursprung, die Textfarbe ist Weiß, die Hintergrundfarbe ist Schwarz und die Textgröße ist 2.

- b) Die Funktion `drawChar(x, y, c, color, bg, size)` soll einen Buchstaben `c` in ASCII-Schrifart auf dem Display an der Position `x, y` in der Farbe `color` mit der Hintergrundfarbe `bg` und in der Größe `size` abbilden.

- Zunächst muss überprüft werden, ob die angegebene Position gültig ist. Sofern die Werte die Grenzen des Bildschirms überschreiten, soll die Funktion `drawChar` beendet werden.
- Als nächster Schritt soll auf die richtige Stelle des Arrays `font` zugegriffen werden und die gesetzten Bits der Hexadezimalwerte als farbiges Pixel interpretiert werden. Abbildung 4 zeigt den Aufbau der Daten im Array `font` am Beispiel des Buchstabens 'A'. Ein beliebiger Buchstabe `c` ist in `font` an der Position `c * 5` gespeichert. Für diesen Buchstaben werden 5 Byte/40 Bits durchteriert. Für jedes gesetzte Bit wird an der entsprechenden Stelle ein Pixel (für `size == 1`) oder Rechteck (für `size > 1`) auf dem Display ausgegeben.
- Zusätzlich soll nach dem Buchstaben ein Leerraum gesetzt werden. Der Leerraum soll die Breite `size` haben.

- c) Um die Textausgabe auf dem Display für Strings zu ermöglichen, müssen weitere Funktionen implementiert werden, die einen automatischen Cursor verwenden. Dieser Cursor speichert die Position des letzten geschriebenen Buchstabens. Die Funktion `writeAuto(char c)` soll die Aufgabe übernehmen, einen Buchstaben `c` auf dem Display mithilfe von `drawChar` zu schreiben und die Cursorposition zu verändern. Hierbei sind Display-Grenzen und Zeilenumbrüche zu beachten.

## Aufgaben zu Embedded C

- d) Implementiere die Funktion `writeText(const char *text)`, welche einen C-String als Parameter erhält und diesen mithilfe von `writeAuto` auf das Display schreibt.
- e) Implementiere abschließend die Funktion `writeTextln(const char *text)`, die sich ähnlich wie `writeText` verhält, am Ende der Textausgabe jedoch zusätzlich einen Zeilenumbruch einfügt.
- f) Implementiere zur Ausgabe von Zahlen die Funktionen `writeNumberOnDisplay(const uint8_t *value)` und `writeNumberOnDisplayRight(const uint8_t *value)`. Erstere soll die per Pointer übergebene Zahl linksbündig ausgeben, Zweitere soll die Zahl rechtsbündig ausgeben. Rechtsbündig bedeutet hier, dass die Ausgabe immer die gleiche Breite hat, egal ob der Wert 0 oder 255 ist (also: 3 Ziffern). Verwende die Funktion `itoa` (aus `stdlib.h`<sup>8</sup>), um den per Zeiger übergebenen Wert in ein `char`-Array zu schreiben und anschließend mittels `writeText` auf dem Display aus.
- g) Implementiere die Funktion `write16BitNumberOnDisplay(const uint8_t 16, uint8_t mode)` die eine 16 Bit Zahl auf dem Display ausgeben soll. Hierbei soll zwischen linksbündige und rechtsbündige Schreibweise unterschieden werden. Gilt `mode == 0`, dann soll die Zahl linksbündig angezeigt werden, sonst rechtsbündig.

### Hinweise

- Falls du Sonderzeichen darstellen möchtest, kannst du dich an folgender Tabelle orientieren: <http://www.theasciicode.com.ar/american-standard-code-information-interchange/ascii-codes-table.png>.

Um beispielsweise ein 'ß' auszugeben, definierst du es einfach als `char ss = '\xE1';`

<sup>8</sup> <http://www.cplusplus.com/reference/cstdlib/itoa/>

## Aufgaben zu Embedded C

### Aufgabe 5 [C] Joysticks abfragen (optional)

Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

In diesem Abschnitt nutzen wir die Funktionen der vorigen Aufgabe, um die analogen Werte der Joysticks auszugeben. Darauf aufbauend entwickelst du eine Aufgabe, die mithilfe des Joysticks die Farbe der RGB-LED verändert. Die zu implementierenden Funktionen befinden sich in der Datei `joystick.c`. Solltest du die vorherige Aufgabe nicht (vollständig) bearbeitet haben, kannst du auf die Musterlösung in der Datei `display_s.c` zurückgreifen. Um diese statt deiner eigenen Lösung zu nutzen, hängst du an jeden Funktionsnamen das Suffix `_s` an (bspw. `writeChar_s` statt `writeChar`).

#### Aufgabe 5.1 Analoge Werte auf dem Display anzeigen

Jeder Joystick besitzt zwei analoge Leitungen, welche die X- oder Y-Position des Steuerknüppels auslesen. Die analogen Werte entsprechen dabei der Spannung des jeweiligen Drehpotentiometers der Achse, welche zwischen 0 V und 5 V liegt. Die Spannungswerte der Joysticks werden durch den Analog-Digital-Wandler des Microcontrollers auf einen 8-Bit-Wert abgebildet (Wertebereich: 0 bis 255). Die Aufteilung der Wertebereiche sowie die Orientierung der X- und Y-Richtung sind in Abbildung 5 dargestellt. Die Aufteilung ist nicht gleichmäßig. Stattdessen ergibt sich in Neutralstellung der Wert

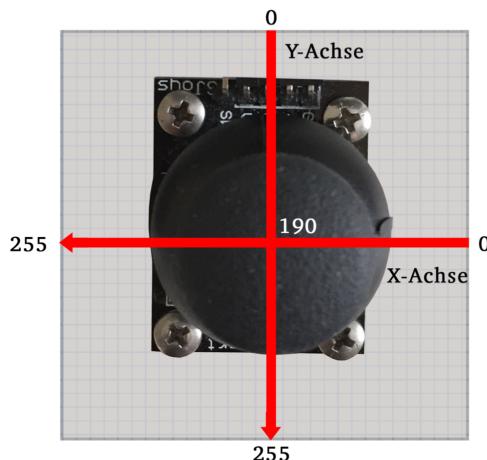


Abbildung 5: Wertebereich der Joysticks

(190, 190). Joystick 1 ist mit den analogen Anschlüssen AN16 (X) und AN19 (Y) und Joystick 2 ist mit AN13 (X) und AN23 (Y) verbunden.

- Implementiere die Funktion `printValues`, welche über Zeiger auf die analogen Leitungen AN13, AN16, AN19 und AN23 die Werte der Joysticks ausliest und diese auf dem Bildschirm ausgibt. Nutze dazu die Funktionen in Tabelle 3 und folgendes Codefragment, mithilfe dessen man die Analog-Kanäle ausliest:

```
// #include "analog.h"
uint8_t analog11;
uint8_t analog12;
uint8_t analog13;
uint8_t analog16;
uint8_t analog19;
uint8_t analog23;
uint8_t analog17;
cppp_getAnalogValues(&analog11, &analog12, &analog13, &analog16, &analog17, &analog19,
&analog23);
```

## Aufgaben zu Embedded C

- b) Um fortlaufend die Positionsdaten des Joysticks auszugeben, rufe `printValues` in einer Schleife auf:

```
#include "init.h"

int main(){
    initBoard();
    while(1) {
        cppp_microDelay(1000);
    }
}
```

**Tabelle 3:** Wichtige Funktionen und Variablen für die Verwendung der Joysticks

Funktionen/Variablen	Beschreibung
<code>setCursor(0, 319)</code>	Setzt den Cursor auf die linke obere Ecke
<code>void writeTextln(char *text)</code>	Schreibt <code>text</code> auf das Display und verschiebt den Cursor mit Zeilensprung
<code>void writeText(char *text)</code>	Schreibt <code>text</code> auf das Display und verschiebt den Cursor ohne Zeilensprung
<code>void writeNumberOnDisplayRight(const uint8_t *number)</code>	Schreibt die per Pointer referenzierte Zahl <code>number</code> an die Position des Cursors und verschiebt den Cursor

### Aufgabe 5.2 LED mit Joystick 1 kontrollieren

In dieser Aufgabe soll die Funktion `controlLEDs` geschrieben werden, um die Farbe der RGB-LED durch die Bewegung des Joysticks 1 nach links oder rechts zu verändern. Table 4 zeigt die verschiedenen möglichen Wertebereiche mit den anzusteuernden Ports und Pins der LEDs (wie in `pins.h` definiert).

**Tabelle 4:** Anzeigebereiche der LEDs

Position des Joystick	LED-Farbe	Werbereich AN16	Daten-Port	Ausgabe-Pin
Links	Grün	255 ... 200	LED_GREEN_DOR	LED_GREEN_PIN
Mitte	Blau	200 ... 180	LED_BLUE_DOR	LED_BLUE_PIN
Rechts	Rot	180 ... 0	LED_RED_DOR	LED_RED_PIN

- Implementiere zunächst die Hilfsfunktion `controlLEDsInit`, welche die Leitungen der RGB-LEDs initialisiert. Die analogen Kanäle der LEDs sollen ausgeschaltet werden. Definiere die Pins der LEDs als Ausgänge und initialisiere sie mit 1u (= „aus“).
- Implementiere nun die Funktion `controlLEDs`, welche die Position der X-Achse des Joystick über den analogen Kanal AN16 ausliest und die Farbe der RGB-LED gemäß Tabelle 4 verändert.
- Dein Testcode sollte in etwa wie folgt aussehen:

```
#include "init.h"

int main(){
    initBoard();
    controlLedsInit();
    while(1) {
        controlLeds();
        delay(1000);
    }
}
```

## Aufgaben zu Embedded C

### Aufgabe 6 [C] Touchscreen ansteuern (optional)

Eingebaut im mit dem Microcontroller verbundenen Bildschirm ist eine resistive 4-Wire Touchschicht. Der Name setzt sich zusammen aus den Eigenschaften dieser Schaltung. Resistiv steht für die Messung von Widerständen zum Erkennen von Touch-Gesten und 4-Wire bedeutet, dass für diese Technik vier Datenleitungen gebraucht werden. Ein resistiver 4-Wire-Touchscreen ist wie in Abbildung 6 aufgebaut.

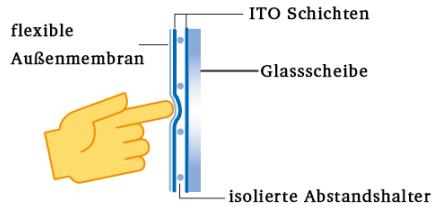


Abbildung 6: Aufbau eines resistiven Touchscreens

Dieser besteht aus a) einer Glas- oder Acryl-Schicht, b) einer äußeren resistiven Schicht, die mit Indium-Zinn-Oxid („indium tin oxide“, ITO) beschichtet ist, c) isolierenden Punkten, d) der inneren resistiven Schicht aus ITO und e) einem Polyester-Film. Die beiden resistiven Schichten sind jeweils an 2 Polen angeschlossen (Abbildung 7).

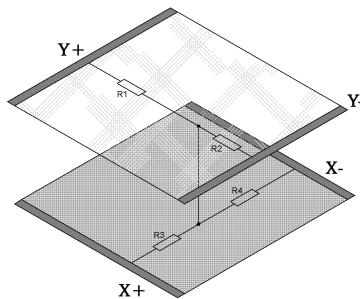


Abbildung 7: Aufbau eines 4-Wire resistiven Touchscreens

Die Schichten sind in Bezug auf ihre Pole um 90 Grad zueinander gedreht. Dies ist wichtig, um später die X- und Y-Koordinaten des Druckpunkts zu lesen. Sobald ein Objekt die oberste Glas- oder Acryl-Schicht berührt und genügend Druck ausübt, wird sich die oberste ITO-Schicht mit der unteren verbinden. Die X- und Y-Koordinate des Druckpunkts wird bestimmt, indem die Spannungen an den Polen gemessen werden. Zur Messung des X-Wertes werden X+ und X- über Gleichspannung geschaltet. Das heißt, X+ ist beispielsweise auf Vcc und X- ist mit GND verbunden. Durch die Verbindung der beiden ITO-Schichten entsteht ein Stromfluss durch beide Schichten und es kommt zu einem Spannungsteiler in der X-Schicht. Die Spannungen zwischen X+ und dem Druckpunkt sowie dem Druckpunkt und X- lassen sich durch das Ausmessen von Y- und Y+ bestimmen. Diese Information wird vom Microcontroller ausgelesen, der die gemessene Spannung in Relation zur Auflösung des Displays setzt. Die Y-Koordinate wird gemessen, indem Y+ und Y- an eine Gleichspannung gelegt und die Spannungen an X+ und X- ausgelesen werden.

Für dein Projekt stellen wir dir die Funktionen `cppp_readTouchX()`, `cppp_readTouchY()` und `cppp_readTouchZ()` zur Verfügung, die X-, Y- und Z-Werte eines Druckpunkts auf dem Touchscreen auslesen können (`analog.h`). Ist der Z-Wert größer als ein bestimmter Grenzwert, kann von einer Berührung des Touchscreens ausgegangen werden.

#### Aufgabe 6.1 Werte des Touchscreens debuggen

Implementiere zunächst die Funktion `debugTouch()`, die kontinuierlich die X-, Y- und Z-Werte des Touchscreens auf dem Bildschirm ausgibt.

## Aufgaben zu Embedded C

### Aufgabe 6.2 Zeichnen auf dem Touchscreen

In diesem Abschnitt implementierst du eine kleine Mal-Anwendung für den Touchscreen. Mit dieser soll es möglich sein, verschiedene Farben auszuwählen und mithilfe des Fingers auf dem Bildschirm zu zeichnen. Vervollständige hierfür die Funktion `paintTouch()` sowie `loopPaintTouch()`. Bereits implementiert sind die Farbpaletten und der Lösch-Button auf der unteren Seite des Bildschirms. Die Funktion `loopPaintTouch()` muss noch um eine Touch-Logik ergänzt werden, die Berührungspunkte auf dem Bildschirm erkennt und diese korrekt interpretiert. Hierbei gibt es folgende Szenarien:

- Die Farbpalette wird berührt und somit verändert sich die aktuelle Malfarbe.
- Bild erneuern wurde gedrückt und der Malbereich wird zurückgesetzt.
- Wird der freie Zeichen-Bereich berührt, so soll an dieser Stelle in der ausgewählten Farbe ein ausgefüllter Kreis mit dem Radius `PENRADIUS` gezeichnet werden.

Abbildung 8 zeigt, wie die fertige Anwendung aussehen kann.

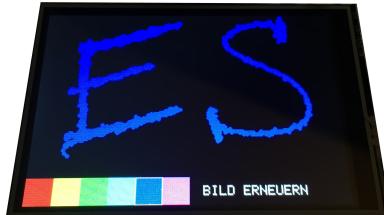


Abbildung 8: Mal-Anwendung auf dem Touchscreen

## Aufgaben zu Embedded C

### Aufgabe 7 [C] Eigenes Microcontroller-Projekt umsetzen (optional)

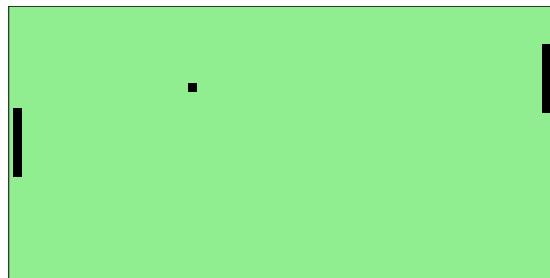
Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

Nachdem du einige der Ein- und Ausgabemöglichkeiten des Boards kennengelernt hast, besteht deine Aufgabe in diesem Teil darin, ein kleines Projekt deiner Wahl umzusetzen. Du hast hierbei die freie Wahl, die folgenden Vorschläge sollen nur als Anregung dienen.

#### Vorschlag: Pong

Zwei Gegner sollen je einen Balken (Rechteck) am linken oder rechten Rand des Spielfeldes mit den Schiebereglern steuern können, um einen Ball (ein Quadrat) im Spiel zu halten. Erreicht der Ball den linken oder rechten Rand des Spielfelds, so bekommt der Spieler auf der anderen Seite einen Punkt und der Ball wird an seine Anfangsposition (die Mitte des Spielfelds) zurückversetzt. Erreicht der Ball den oberen oder unteren Rand sowie einen der Balken der Spieler, so wird der Ball reflektiert - verlässt also niemals das Spielfeld.

Gewonnen hat der Spieler, der zuerst eine definierte Anzahl an Punkten erreicht. Der aktuelle Punktestand könnte ebenfalls auf dem Display angezeigt werden.

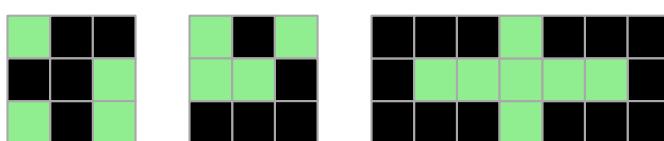


#### Vorschlag: Game of Life

„Game of Life“<sup>9</sup> besteht aus einem zweidimensionalen Spielfeld. Jedes Feld steht für eine Zelle, die *tot* oder *lebendig* ist. Die Farben für den Zustand kannst du natürlich frei wählen. Jede Zelle hat acht Nachbarzellen, die ebenso tot oder lebendig sein können. Zu Beginn gibt es eine vordefinierte Anfangsgeneration. Durch festgelegte Regeln wird die nachfolgende Generation ermittelt:

- Eine **lebende Zelle** ...
  - mit 1 oder 0 lebenden Nachbarn stirbt aus Einsamkeit.
  - mit 4 oder mehr lebenden Nachbarn stirbt wegen Übervölkerung.
  - mit 2 oder 3 lebenden Nachbarn bleibt am Leben.
- Eine **tote Zelle** mit genau 3 lebenden Nachbarn wird in der nächsten Generation geboren werden, andernfalls bleibt sie tot.

Als Anfangsgeneration eignen sich zufällige Populationen oder eine der folgenden Figuren:



<sup>9</sup> siehe auch [http://de.wikipedia.org/wiki/Conways\\_Spiel\\_des\\_Lebens](http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens)

## Aufgaben zu Embedded C

---

### Hinweise

- Da das Spielfeld begrenzt ist, soll es torusförmig aufgebaut werden. Das heißt: Alles, was am unteren Rand des Spielfelds verschwindet, kommt oben wieder heraus – das gleiche gilt für den linken und rechten Rand.
- Verwende als Spielfeld ein mehrdimensionales Array
- Ein weiteres mehrdimensionales Array bietet sich an, um die zukünftige Generation erzeugen zu können.
- Achte beim torusförmigen Feld unbedingt darauf, dass du nicht über die Grenzen des Spielfelds hinaus zugreifst! Das kann zu unvorhersehbarem und schwer zu debuggendem Verhalten des ganzen Displays führen!

---

### Vorschlag: Regentropfen

---

Das Touch-Display ist ein kleiner Teich; wenn du eine Stelle mit dem Finger berührst, breitet sich von dort eine konzentrische Welle aus. Die Geschwindigkeit der Welle kannst du zusätzlich abhängig machen vom ausgeübten Druck.

---

### Weitere Tipps und Vorschläge

---

Asteroids  
<https://goo.gl/aE7mgC>

Ausweichspiele à la Hugo  
<https://goo.gl/Ab8Go2>

Pacman  
<https://goo.gl/kXthKj>

Moorhuhn  
<https://goo.gl/X2xems>

Labyrinth  
<https://goo.gl/VCf85t>

Snake  
<https://goo.gl/iL2L5M>

### Hinweise

- Der folgenden Wiki-Artikel beschreibt, wie man schnell bestehende Bilder in das eigene Projekt einbinden kann:  
<https://github.com/Echtzeitsysteme/tud-cppp/wiki/RGB565-mit-Gimp>.
- Quick Start Guide für das Evaluationsboard: <http://www.cypress.com/file/290916/download>
- PDL Quick Start Guide: <http://www.cypress.com/file/307966/download>
- Schaltbild des Evaluationsboards: <http://www.cypress.com/file/290921/download>

## Aufgaben zu Embedded C

### Aufgabe 8 [C] DHT11 (optional)

Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

Für die folgende Aufgabe musst du dir folgendes zusätzliches Material bei den Betreuern beschaffen:

- DHT11 Temperatur- und Feuchtigkeitssensor
- 3 Steckbrücken Pin-Pin 20cm (jeweils ein verbundenes Steckbrückenpaar und eine einzelne Steckbrücke)
- 1 Steckbrücke Pin-Buchse 20cm
- 1 Widerstand 10 kΩ (oder höher)

Der DHT11 ist ein Temperatur- und Feuchtigkeitssensor, der mithilfe des Steckplatine an den Microcontroller angeschlossen werden kann. Der DHT11 hat 4 Anschlüsse (Pins), von denen im Rahmen des Praktikums 3 verwendet werden. Zwei Pins werden für Masse (GND) und Versorgungsspannung (VCC) genutzt und ein Pin für die digitale Datenübertragung zwischen dem DHT11 und dem Microcontroller. Abbildung 9 zeigt die Pin-Belegung des DHT11. Pin 1 wird mit VCC verbunden und Pin 4 mit GND. Pin 2 ist ein I/O-Pin zur digitalen Datenübertragung. In dieser Aufgabe ist es das Ziel die Temperatur- und Feuchtigkeitswerte des Sensors kontinuierlich auszulesen und diese auf dem Display darzustellen.

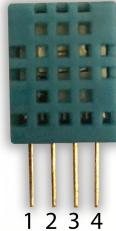


Abbildung 9: DHT11 Pinbelegung

- Zunächst trenne den Microcontroller von der Stromzufuhr und stelle sicher, dass dieser ausgeschaltet ist. Verbinde gemäß Abbildung 10 den DHT11 mit dem Microcontroller. Der DHT11 wird, von links gezählt, an den dritten Pin des CN17 Steckers des Microcontrollers angeschlossen (Port 5, Pin 2, siehe Präprozessorkonstante GPIO1PIN\_PF52). Nutze als Unterstützung Abbildung 11, die schematisch die GPIO-Verbindungen des Microcontrollers darstellt. Bevor der Microcontroller wieder an den Computer angeschlossen wird, lasse die Verkabelung von einem Tutor überprüfen.
- In der Funktion `readDHT11(uint8_t* humidity, uint8_t* temperature)` in der Datei `dht11.c` wird zunächst eine Präambel vom DHT11 an den Microcontroller gesendet. Hierdurch wird die Verbindung zwischen dem DHT11 und dem Microcontroller synchronisiert. Nach der Präambel beginnt der DHT11 40 Bits an den Microcontroller zu senden. Abbildung 12 zeigt die Aufteilung der Bitgruppen.

Implementiere die fehlenden Stellen der Funktion `readDHT11(uint8_t* humidity, uint8_t* temperature)`, indem du folgende Teilschritte umsetzt. Nach der Präambel müssen zunächst die ankommenden Signale als 40 Bits interpretiert und in einem Array mit 5 Einträgen je 8 Bits gespeichert werden. Lese mit `Gpio1pin_Get(GPIO1PIN_P52)` den aktuellen Wert des digitalen Pins aus und entscheide entsprechend der folgenden Logik, ob es sich um eine 1 oder 0 handelt. Ist der Pin länger als 30 µs auf HIGH gesetzt  $t_{HIGH} > 30 \mu s$ , dann interpretiere dieses Signal als eine 1 und im umgekehrten Fall  $t_{HIGH} < 30 \mu s$  als eine 0. Nachdem alle 40 Bits empfangen wurden, bilde die Checksumme, welche die Summe der ersten 32 Bits darstellt. Ist der folgende Pseudocode erfüllt, dann ist die Übertragung der Daten erfolgreich gewesen.

```
data[0]+data[1]+data[2]+data[3] == data[4]
```

## Aufgaben zu Embedded C

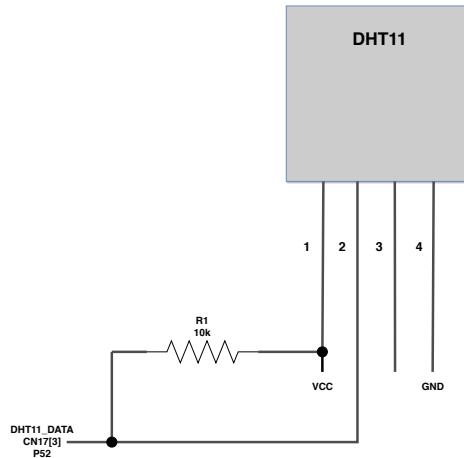


Abbildung 10: Verkabelung des DHT11

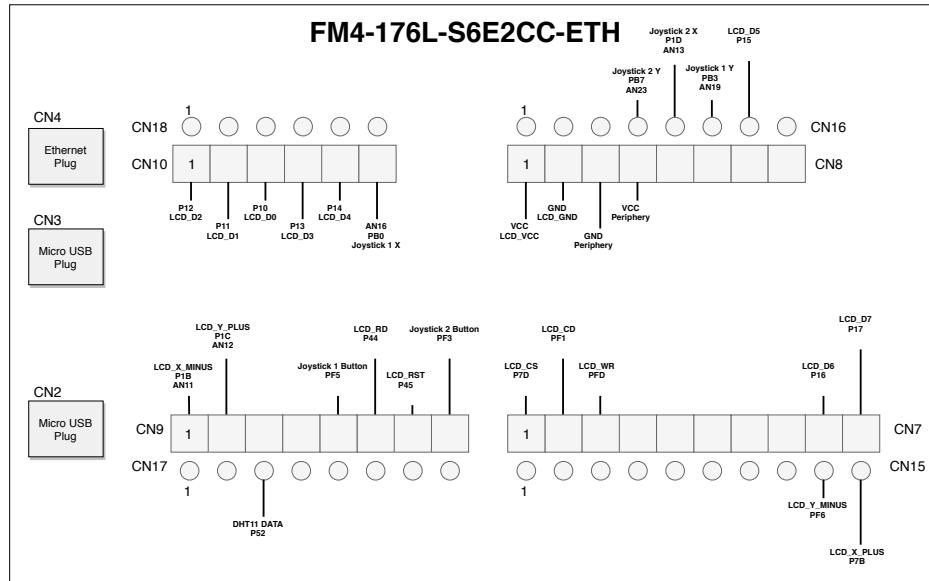


Abbildung 11: Verkabelung des DHT11

Beende die Methode `readDHT11`, sofern die Checksumme nicht stimmt. Ist die Checksumme richtig, speichere die aktuelle Temperatur und Feuchtigkeit in den übergebenen Parametern `humidity` und `temperature`.

1.

2.

3.

4.

5.

0011 0101	0000 0000	0001 1000	0000 0000	0100 1101
-----------	-----------	-----------	-----------	-----------

Feuchtigkeit

Temperatur

Prüfsumme

Abbildung 12: 40 Bit Datenstruktur eines DHT11 Pakets

## Aufgaben zu Embedded C

---

### Hinweise

- Bei der Abfrage des Pins mithilfe von `Gpio1pin_Get(GPIO1PIN_P52)`, kann es dazu kommen, dass fehlerhafte Signale dauerhaft auf HIGH oder LOW bleiben. Um diesen Fall zu filtern, verwende folgende Art der Abfrage.

```
timeout = 1000
while(Gpio1pin_Get(GPIO1PIN_P52) == 0) {
    if (!timeout--) {
        return 0;
    }
}
```

- Um einen Delay auf dem Microcontroller ausführen, kannst du die Methode `cppp_microDelay(uint32_t timeInMicroseconds)` verwenden. Zum Beispiel wird beim Aufruf `cppp_microDelay(20)` der Microcontroller 20 µs pausiert.
  - Der DHT11 sendet immer seine Daten geordnet nach dem höchsten Bit, sodass der höchstwertigste Bit zuerst gesendet wird.
- c) Nutze die Methoden `writeTextln` und `writeNumberOnDisplay` der Display-Library, um die Sensorenwerte auf dem Display auszugeben.

### Hinweise

- Falls du die Funktionen `writeTextln` und `writeNumberOnDisplay` noch nicht implementiert hast, kannst du die Methoden der Musterlösung mit der Konkatenation des Suffix `_s` nutzen (also bspw. `writeTextln_s` und `writeNumberOnDisplay_s`).

## Aufgaben zu Embedded C

### Aufgabe 9 [C] Beschleunigungssensor (optional)

Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

Auf dem Evaluationsboard ist ein Beschleunigungssensor integriert, den du verwenden kannst um die Orientierung des Entwicklungsboards auszulesen. Der Beschleunigungssensor wird von einem eigenen Microcontroller ausgelesen. Die Kommunikation zwischen FM4 und diesem Prozessor findet über die I2C-Schnittstelle statt.

In der Gruppe *lib* findest du die Datei *acceleration\_app.h*, mit der du den Beschleunigungssensor für deine eigene Applikation verwenden kannst. In dieser Aufgabe wirst du mithilfe des Beschleunigungssensors eine digitale Wasserwaage simulieren.

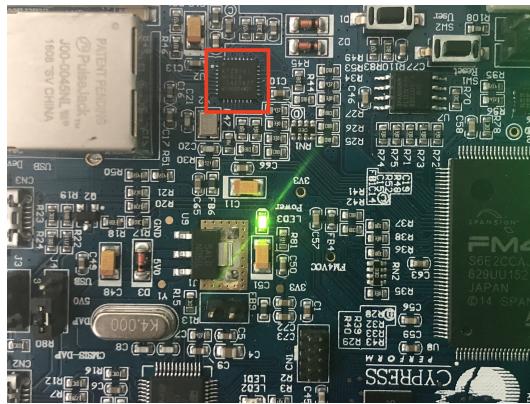


Abbildung 13: Der Beschleunigungssensor des FM4-Boards (rot umrandet)

In der Datei *acceleration.c* (Gruppe *src*) findest du eine Vorlage für diese Aufgabe. In dieser fehlt noch die fertige Implementierung der Funktion *cppo\_rgbLEDAcceleration*. Bei der Initialisierung des Boards wird eine Interrupt-Routine gestartet, sobald neue Messdaten des Beschleunigungssensor des Chips vorliegen. Sofern die Routine ausgelöst wurde, wird die globale Variable *cppo\_accelerationDataAvailable* auf 1 gesetzt. In dem Array *float cppo\_orientationValues[3]* werden die aktuellen X-,Y- und Z-Achsen-Orientierungen des Boards in alphabetischer Reihenfolge gespeichert. Gehe nun wie folgt vor:

- Gib auf dem Display kontinuierlich die aktuelle Orientierung des Boards aus. Setze hierzu den Cursor per *setCursor* an die Position (0, 319) (linke obere Ecke des Displays) und gib die Daten in folgendem Format aus:

```
//*** Beschleunigungssensor ***
//Orientierung X:
//Orientierung Y:
//Orientierung Z:

//Die Ebene ist (nicht) waagrecht.
```

#### Hinweise

- Verwende zur Ausgabe von Variablen des Typs *float* auf den Display die Funktion *cppo\_writeFloat* (im Header *gfx.h*).
- Nutze die Ergebnisse aus Teilaufgabe a um ein Schema zu erkennen, wann das Board sich nicht im Gleichgewicht befindet. Setze die RGB-LED auf Rot sofern sich das Board nicht im Gleichgewicht befindet und auf Grün falls ja. Gib ebenfalls auf dem Display aus, ob sich das Board im Gleichgewicht befindet (siehe Beispieldformat in Teilaufgabe a).

## Aufgaben zu Embedded C

---

### Hinweise

- Zum Ansteuern der RGB-LED kannst du in dieser Aufgabe als Vereinfachung den Header `rgbled.h` (Gruppe: *lib*) verwenden. Hierzu muss zunächst die Funktion `cppo_initLEDs` einmalig aufgerufen werden. Beispielsweise kannst du mithilfe der Funktionen `cppo_redLEDO`n und `cppo_redLEDO`ff die rote LED ein- bzw. ausschalten.

## Aufgaben zu Embedded C

### Aufgabe 10 [C] ABLE—Android Bluetooth Low Energy (optional)

ABLE (Android Bluetooth Low Energy) ist eine Applikation für Androidgeräte, um über BLE (Bluetooth Low Energy) eine drahtlose Verbindung zwischen einem Android-Smartphone und einem anderen BLE-Gerät herzustellen. Der Mikrocontroller des C/C++-Praktikums kann durch ein zusätzliches Modul mit BLE erweitert werden. Du kannst auf einem Androidgerät die ABLE-App installieren und dich mit dem Mikrocontroller verbinden. Abbildung 14 zeigt ein Beispielprogramm, in dem der Mikrocontroller den X-Wert des Joysticks 1 über BLE an ein Android-Smartphone sendet. Eine ausführliche Anleitung zur Nutzung von ABLE auf dem Mikrocontroller des Praktikums findest du unter diesem Link [1].

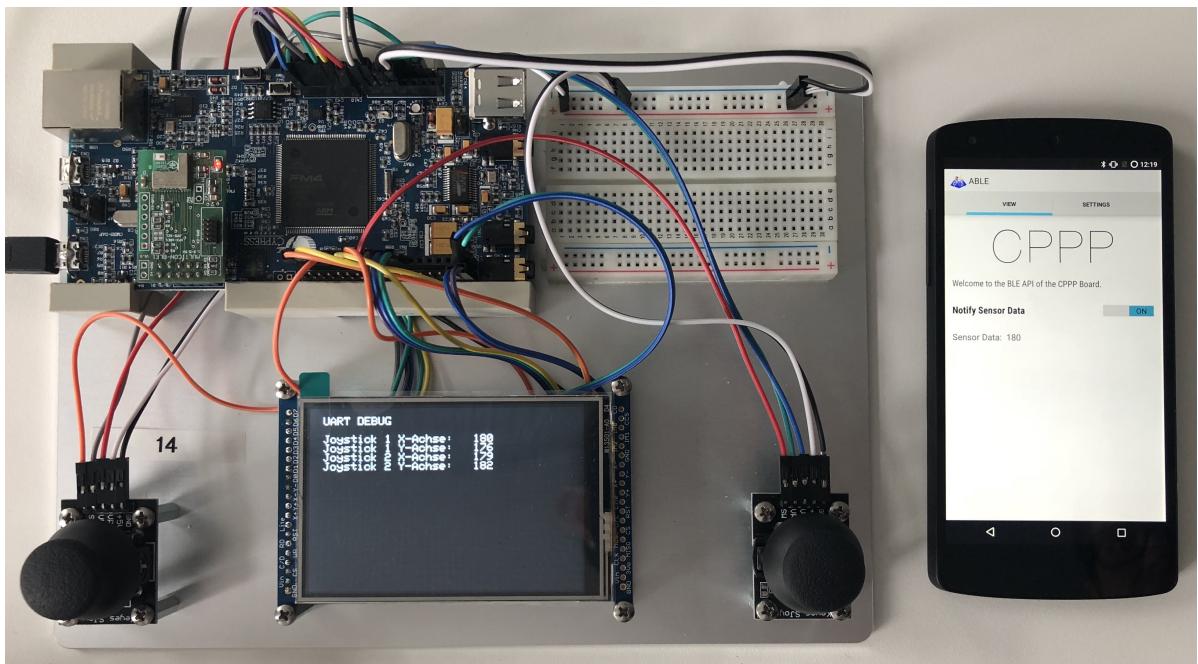


Abbildung 14: ABLE: Der Microcontroller verbunden über Bluetooth mit einem Androidgeräte

## Aufgaben zu Embedded C

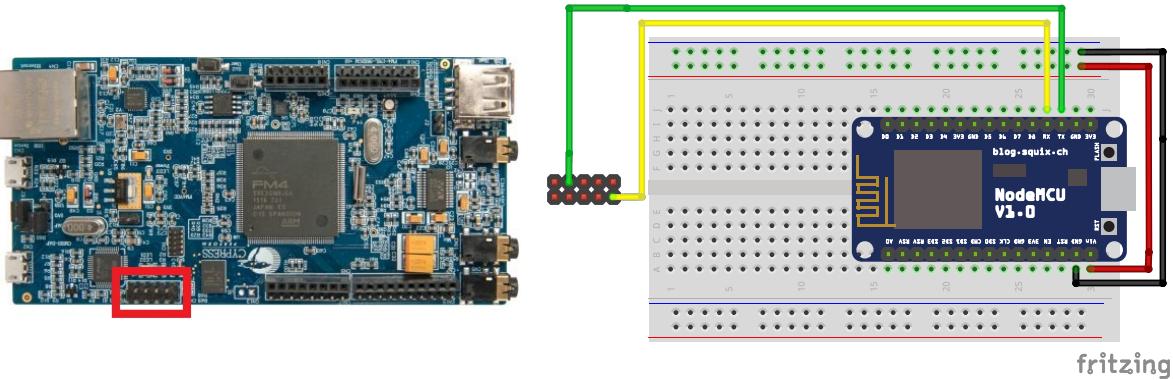
### Aufgabe 11 [C] WSN—Wide Sensor Network (optional)

Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

Ein WSN (Wide Sensor Network) ist ein Netzwerk aus verteilten Sensorknoten. Ziel dieser Aufgabe ist es, das Mikrocontrollerboard um weiteren Mikrocontroller mit WiFi-Schnittstelle zu erweitern, sodass mehrere Boards über ein drahtloses Netzwerk (WLAN) kommunizieren können. Dieses Netzwerk soll verwendet werden, um gemessene Sensordaten an eine zentrale Server-Anwendung zu senden.

Als Erweiterungsmodul nutzen wir das ESP-12E Modul. Dieses wird mit einer modifizierten Version der esp\_wifi\_repeater-Firmware betrieben. Die Kommunikation zwischen unserem Mikrocontroller und dem ESP-Board wird über UART realisiert.

- Unser Mikrocontrollerboard besitzt insgesamt 12 serielle Schnittstellen. In dieser Aufgabe verwenden wir die Schnittstelle UART3, welche über den Multicon-Pinheader (rote Markierung) verfügbar ist.



Zuerst muss das ESP-Board wie in den beiden Abbildungen zu sehen angeschlossen werden. Dazu sind 4 Leitungen nötig: Masse (GND), Versorgungsspannung (Vcc), von Tx (Transmit) am Cypress-Board zu Rx (Receive) am ESP-Modul und vice versa.

- Um nun Daten in Form von Zeichenketten zu übertragen, muss zuerst die UART-Schnittstelle initialisiert werden. Dazu reicht es, den Header `uart_multicon.h` einzubinden und die Funktion `cpp_initUart3Baud(115200)` aufzurufen. 115200 ist die Baudrate mit der die Schnittstelle arbeiten muss.

Zum Übertragen von ganzen Zeichenketten, ist es hilfreich zunächst eine Funktion zu implementieren, welche ein einzelnes Zeichen als Parameter entgegennimmt und überträgt (z.B. `void writeCharUart3(char c)`). In dieser Funktion muss gewartet werden, solange der Aufruf `Mfs_Uart_GetStatus(&UART3, UartTxEmpty)` nicht TRUE ergibt. TRUE und FALSE sind in diesem Fall Makros der Treiberbibliothek, und müssen daher genau so verwendet werden. Liefert der `Mfs_Uart_GetStatus`-Aufruf FALSE zurück, ist die Schnittstelle bereit um ein einzelnes Zeichen mit dem Aufruf von `Mfs_Uart_SendData(&UART3, 'c')` zu versenden.

- Implementiere nun noch eine Funktion die einen nullterminierten C-String entgegennimmt und durch wiederholte Aufrufe von `writeCharUart3(char c)` versendet.

Teste deine Funktion, indem du ihr einen Text (z.B. `"mqtt_pub /Hello World!\r\n"`) übergibst. `\r\n` markiert das Ende des Befehls. Auf der Serveranwendung sollte deine Nachricht nun ankommen.

- Erweitere nun dein Board um einen Sensor, und sende den gemessenen Sensorwert an die Zentrale Serveranwendung. Sende dazu den Befehl `"mqtt_wsn <Sensorwert>\r\n"`.

## Aufgaben zu Embedded C

---



Dieses Werk ist unter einer Creative Commons Lizenz vom Typ Namensnennung - Nicht kommerziell - Keine Bearbeitungen 4.0 International zugänglich. Um eine Kopie dieser Lizenz einzusehen, konsultieren Sie <http://creativecommons.org/licenses/by-nc-nd/4.0/> oder wenden Sie sich brieflich an Creative Commons, Postfach 1866, Mountain View, California, 94042 USA.