Informatik 11
Embedded Software

RWTH AACHEN UNIVERSITY

**Praktikum Systemprogrammierung**

# Versuch 6

# *PS/2-Tastatur*

Lehrstuhl Informatik 11 - RWTH Aachen

16. Juni 2023

Commit: c3b830de

# Inhaltsverzeichnis



---

Dieses Dokument ist Teil der begleitenden Unterlagen zum *Praktikum Systemprogrammierung*. Alle zu diesem Praktikum benötigten Unterlagen stehen im Moodle-Lernraum unter https://moodle.rwth-aachen.de zum Download bereit. Folgende E-Mail-Adresse ist für Kritik, Anregungen oder Verbesserungsvorschläge verfügbar:

support.psp@embedded.rwth-aachen.de

# 6 PS/2-Tastatur

Um die Kommunikation zwischen Programmen und Peripherie zu gewährleisten, werden in einem modernen Betriebssystem Treiber eingesetzt. Diese stellen Programmen standardisierte Schnittstellen zur Verfügung, mit denen auf die Peripherie des Rechners zugegriffen werden kann.

## 6.1 Versuchsinhalte

Exemplarisch für die Kommunikation mit standardisierter Hardware wird in diesem Versuch ein Treiber für den ATmega 644 zur Unterstützung einer PS/2-Tastatur erstellt. Zur Kommunikation mit der Tastatur wird die USART Komponente des ATmega 644 verwendet.

## 6.2 Lernziel

Das Lernziel dieses Versuchs ist das Verständnis der folgenden Zusammenhänge:

- Implementierung eines Treibers
- Schichtenmodell
- Scancodes
- Kommunikationsprotokoll einer PS/2-Tastatur

## 6.3 Grundlagen

Im Folgenden wird das Übertragungsprotokoll erklärt, welches zur Kommunikation mit der PS/2-Tastatur verwendet wird. Dazu wird zunächst die physikalische Verbindung zwischen Mikrocontroller und Tastatur beschrieben. Anschließend wird die Synchronisation der Kommunikationspartner und die einfache Übertragung von Bytes vorgestellt. Zum Ende des Grundlagenkapitels wird das Konzept von Scancodes und damit zusammenhängend die Semantik der übertragenen Bytes vorgestellt. Die Reihenfolge der Themen richtet sich nach ihrer Abstraktionsebene. Dabei bildet die physikalische Verbindung die niedrigste Ebene, während die Interpretation der Scancodes von den Details der Kommunikation komplett losgelöst ist und damit die höchste Ebene bildet.
Dieses Kapitel soll die generelle Funktionsweise der Kommunikation mit der Tastatur erklären. Details zur Implementierung in SPOS können Kapitel 6.4 entnommen werden.

### 6.3.1 Die PS/2 Schnittstelle

Zur Verbindung der Tastatur mit dem ATmega 644 wird ein PS/2-Port verwendet, welcher nicht von den bisher verwendeten Evaluationsboards unterstützt wird. Abbildung 6.1 zeigt eine Adapterplatine, welche auf diese Boards aufgesteckt wird und einen PS/2-Port zur Verfügung stellt.

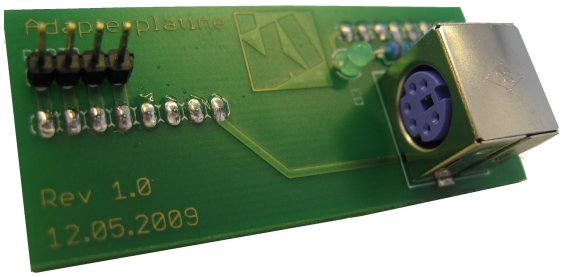


Abbildung 6.1: Adapterplatine für das Evaluationsboard

Die über die PS/2 Schnittstelle angeschlossene Tastatur verwendet vier der sechs verfügbaren Leitungen des PS/2 Ports. Zwei dieser Leitungen dienen der Spannungsversorgung. Die Versorgungsspannung $V_{cc}$ wird von der in diesem Versuch verwendeten Adapterplatine auf die Pins 4 bis 7 von Port B gelegt und $GND$ auf die Pins 1 bis 3 desselben Ports. Auf diese Weise kann die Tastatur softwareseitig ein- und ausgeschaltet werden. Die Versorgung der Tastatur mit nur einem Pin für $V_{cc}$ und einem für $GND$ ist nicht möglich, da die Tastatur beim Einschalten für kurze Zeit einen Stromverbrauch hat, der laut Spezifikation des ATmega 644 für einen einzelnen Pin nicht zulässig ist.
Der für die Datenübertragung genutzte Pin ist mit Pin 0 von Port D verbunden. Die Kommunikation mit der Tastatur erfolgt synchron, sodass eine zusätzliche Clockleitung benötigt wird, welche auf Pin 0 von Port B liegt. Eine Übersicht aller Pins ist in Tabelle 6.1 zu finden.

| Leitung | Pins | Port |
|---|---|---|
| $V_{cc}$ | 7..4 | B |
| $GND$ | 3..1 | B |
| $CLK$ | 0 | B |
| $Data$ | 0 | D |

Tabelle 6.1: Pinbelegung am Mikrocontroller

## Lernerfolgsfragen

- Wieso muss die Tastatur an mehr als zwei Pins für die Stromversorgung angeschlossen werden?
- Wie viele Leitungen werden insgesamt zur Kommunikation mit einer Tastatur benötigt?
- Können Sie bereits mit Hilfe des Datenblattes erklären, warum Daten- und Clockleitung an den genannten Pins angeschlossen werden müssen, bzw. was der Vorteil ist, genau diese Pins zu wählen?

### 6.3.2 Kommunikation mit der Tastatur

Damit der Mikrocontroller mit der Tastatur kommunizieren kann, müssen sich beide Kommunikationspartner eine Datenleitung teilen. Das heißt, dass die Daten*richtung* dieser Leitung immer wohldefiniert sein muss; es dürfen niemals beide Kommunikationspartner *gleichzeitig* Daten senden. Zur Lösung dieses Problems gibt es im Allgemeinen mehrere Möglichkeiten. In diesem Fall ordnet sich einer der beiden Kommunikationspartner dem anderen unter, was als „Master-Slave“-Beziehung bezeichnet wird.
In dem hier vorgestellten Protokoll ist der Mikrocontroller der Master und die Tastatur der Slave. Der Mikrocontroller erlaubt oder verbietet der Tastatur explizit, Daten zu senden. Die Festlegung der Datenrichtung durch den Master geschieht über die Clockleitung. Schaltet der Master die Datenleitung auf $V_{cc}$ und die Clockleitung auf $GND$, so registriert der Slave (die Tastatur) dies und stoppt sofort das Senden von Daten. Die Tastatur schreibt also nicht mehr auf die Datenleitung, sondern liest davon.
Um die Leitung freizugeben, d.h. dem Slave das Senden von Daten zu erlauben, muss der Master die Clockleitung auf $V_{cc}$ setzen. Anschließend kann der Mikrocontroller die Datenleitung wieder als Eingang definieren, um Daten von der Tastatur zu empfangen. Dieses Verfahren wird in den zwei folgenden Abschnitten detaillierter erklärt.

#### Übertragung von der Tastatur zum Mikrocontroller

Ist die Clockleitung für mindestens 50 µs logisch 1, also auf dem gleichen Potential wie $V_{cc}$, wird der Tastatur dadurch mitgeteilt, dass diese Daten über die Datenleitung an den Mikrocontroller senden darf. Die Tastatur verwendet das in Tabelle 6.2 dargestellte Protokoll, welches Rahmen von je 11 Bits Länge definiert. Dabei bezeichnet in diesem Dokument *Rahmen* bzw. *Kommunikationsrahmen* das gesamte übertragene (11 Bit) Wort, wohingegen das *Datenbyte* nur die eigentliche (8 Bit) Nachricht bezeichnet.
Wird der Tastatur das Schreiben von Daten untersagt, so werden zu sendende Nachrichten im Tastaturcache zwischengespeichert, bis die Datenleitung wieder freigegeben

wird oder der Tastaturcache voll ist. Aus diesem Grund wird in diesem Versuch die Datenleitung immer auf den Empfangsmodus gesetzt, wenn keine Daten an die Tastatur gesendet werden.

| Bit | Bedeutung | Wert |
|---|---|---|
| 0 | Startbit | 0 |
| 1..8 | Datenbyte | Bit 1: LSB, Bit 8: MSB |
| 9 | Parität | ungerade Parität |
| 10 | Stoppbit | 1 |

Tabelle 6.2: Kommunikationsrahmen von der Tastatur zum Mikrocontroller

Das Empfangen von Daten wird mit Hilfe der USART Komponente des ATmega 644 umgesetzt. USART ist die Abkürzung für *Universal Synchronous and Asynchronous Receiver/Transmitter*. Dieser realisiert die serielle Übertragung von Daten. So müssen, nachdem die Codierung der Daten konfiguriert wurde, dem USART die zu übertragenen Daten (der sogenannte Payload) übergeben werden. Die Markierung mit Start- und Stoppbits, die für synchrone serielle Übertragung notwendig sind, wird vom USART selbstständig durchgeführt. Dadurch kann das Senden und Empfangen von Daten durch einfaches Schreiben in bzw. Lesen aus Prozessorregistern erfolgen. *Kapitel 17: USART* des Datenblattes beschreibt ausführlich die Funktionsweise und den Aufbau der Schnittstelle. Dort sind auch Codebeispiele angegeben, welche die Benutzung verdeutlichen. Für diesen Versuch wird es notwendig sein, sowohl gezielt auf den Empfang eines Bytes zu warten (*Polling*), als auch den Empfang eines Bytes mit einer *Interrupt Service Routine* zu behandeln, indem beim Empfang ein Interrupt ausgelöst wird. In Abschnitt 6.3.5 finden Sie hierzu weitere Hinweise. Informationen zum Einrichten von Interrupt Service Routinen sind im Datenblatt des Mikrocontrollers ATmega 644 zu finden.

### Übertragung vom Mikrocontroller zur Tastatur

Ist die Clockleitung für mindestens 100 µs logisch 0 und wird anschließend ein Startbit übertragen, also die Datenleitung auf logisch 0 gezogen, so wird der Tastatur dadurch mitgeteilt, dass ein Byte an diese übertragen werden soll. Zur Tastatur gesendete Daten werden von der Tastatur über die Clockleitung synchronisiert. Damit die Tastatur einen Takt erzeugt, muss die Clockleitung wieder auf logisch 1 gezogen werden.
Mit jeder fallenden Flanke des Clocksignals wird ein Bit des zu übertragenden Bytes, beginnend mit dem *LSB* übertragen. Die Tastatur liest das Bit jeweils an der nächsten steigendenden Flanke des Clocksignals.
Nach dem Datenbyte muss ein Paritäts- und ein Stoppbit übertragen werden. Das Paritätsbit ist eine einfache Checksum für den gesendeten Rahmen. In unserem Fall wird eine sogenannte ungerade Parität verwendet. Das bedeutet, dass die Anzahl der im Rahmen

vorhanden Bits mit dem Wert 1 (bis inklusive des Paritätsbits) ungerade ist. Die Tastatur beantwortet die Übertragung mit einem Bestätigungsbit (*Acknowledge-Bit*), indem sie die Datenleitung auf logisch 0 setzt.
Der Rahmen, der vom Mikrocontroller zur Tastatur gesendet wird, entspricht dem in Tabelle 6.2 vorgestellten Rahmen. Des Weiteren zeigt Abbildung 6.2 exemplarisch die Übertragung des Bytes $\texttt{CD}_h$[1] (in binärer Darstellung `11001101`).

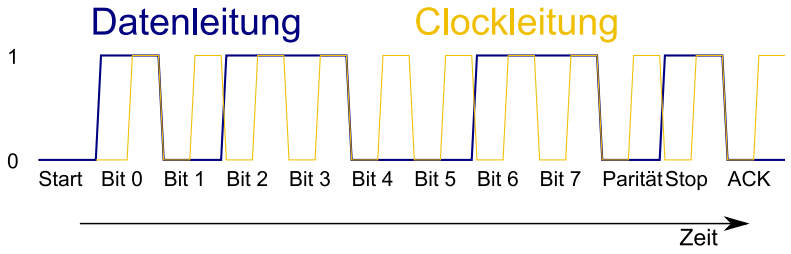


Abbildung 6.2: Übertragung des Bytes $\texttt{CD}_h$

Die übertragenen Datenbytes müssen allerdings noch interpretiert werden. Die Lokalisierung der Tastatur beeinflusst nur die aufgedruckten Zeichen auf den Tasten, die übertragenen Bytes sind davon unabhängig. Die Tasten werden anhand ihrer Position auf der Tastatur codiert: Die Tastatur besteht aus einer Matrix, auf der alle Tasten verdrahtet sind. Diese Matrix wird *gescannt*, um gedrückte Tasten auszumachen sowie zu codieren. Die letztendlich generierten Datenbytes sind abhängig vom *Scancode*.

## Lernerfolgsfragen

- Was versteht man unter einer „Master-Slave“-Beziehung?
- Welches Gerät ist in diesem Versuch der Master und welches der Slave?
- Was ist die primäre Aufgabe der Clockleitung?
- Wie unterscheidet sich ein Kommunikationsrahmen von einem Datenbyte?
- Wie kündigt der Mikrocontroller an, dass er Daten an die Tastatur senden möchte?
- Wieso ist diese Ankündigung notwendig?
- Wie beeinflusst das Tastaturlayout die durch die Tastatur gesendeten Daten?

[1] $\text{x}_h$ steht für Hexadezimal

### 6.3.3 Scancodes

Das Kommunikationsprotokoll muss es der Tastatur ermöglichen, vom Benutzer durchgeführte Aktionen an den Master zu übertragen. Drückt der Benutzer beispielsweise die Taste `X`, muss die Tastatur dem Master die Information „*Taste gedrückt*“ und die Information „*Taste `X`*“ übertragen. Die Abbildung zwischen der ausgeführten Aktion (etwa das Drücken der Taste `X`) und der Folge übertragener Bytes wird als *Scancode* bezeichnet. Die in diesem Praktikum verwendeten Tastaturen verfügen über drei verschiedene Scancodes, welche von 1 bis 3 durchnummeriert sind und zur Laufzeit beliebig umgestellt werden können.
Obwohl Tastaturen standardmäßig Scancode 2 verwenden, wird im Rahmen dieses Praktikums empfohlen, Scancode 3 einzustellen. Dieser ist aufgrund fester Nachrichtenlängen einfacher zu behandeln. Im Codegerüst sind zudem bereits Funktionen enthalten, die mit Scancode 3 codierte Daten direkt auf Tasten und Tastentypen abbilden können.

#### Benutzeraktionen

Die im obigen Abschnitt eingeführten Aktionen beinhalten das Drücken und Loslassen von Tasten. Das Drücken einer Taste wird als *Make* und das Loslassen als *Break* bezeichnet. Entsprechend wird das Wort, welches beim Drücken einer Taste gesendet wird, als *Makecode* und das, welches beim Loslassen einer Taste gesendet wird, als *Breakcode* bezeichnet.
In Scancode 3 bestehen alle Makecodes aus genau einem Byte, welches die gedrückte Taste angibt. Der zugehörige Breakcode entsteht aus dem Makecode durch Voranstellen von $\texttt{F0}_h$, ist also zwei Byte lang. In Tabelle 6.3 ist ein kleiner Ausschnitt von Scancode 3 Codes aufgeführt.

| Taste | Makecode | Breakcode |
|---|---:|---:|
| Shift-Rechts | `59` | `F0 59` |
| Shift-Links | `12` | `F0 12` |
| P | `4D` | `F0 4D` |
| S | `1B` | `F0 1B` |

Tabelle 6.3: Auswahl von Scancode 3 Codes

Beispielsweise wird die Eingabe des Textes „PSP“ durch nachstehende Folge von Bytes übertragen (Bei der Eingabe wird Shift-Links und Shift-Rechts benutzt).

```
59 4D F0 4D F0 59 12 1B F0 1B 4D F0 4D F0 12
```

## Lernerfolgsfragen

- Was versteht man unter einem Scancode?
- Wie wird das Drücken und Loslassen einer Taste unterschieden?

### 6.3.4 Tastaturbefehle

Zur Konfiguration der Tastatur gibt es einen Satz von Kommandos, die an die Tastatur gesendet werden können. Die für diesen Versuch relevanten Kommandos werden in den nachfolgenden Abschnitten vorgestellt. Diese Befehle dürfen nicht mit Scancode Codes verwechselt werden, da sie keine Tasten codieren, sondern Befehle an die Tastatur.

#### Scancode ändern: F0

Wie in Abschnitt 6.3.3 dargestellt, gibt es verschiedene Scancodes. In diesem Versuch wird die Verwendung von Scancode 3 empfohlen. Um den Scancode zu ändern, muss das Kommando $\texttt{F0}_h$ gesendet werden. Daraufhin antwortet die Tastatur mit einer Bestätigung ($\texttt{FA}_h$). Nachdem der Mikrocontroller die Bestätigung empfangen hat, wird der numerische Bezeichner (1, 2 oder 3) des zu setzenden Scancodes gesendet. Dies wird von der Tastatur wiederum mit einer Bestätigung ($\texttt{FA}_h$) quittiert.
Alternativ kann anstatt eines neuen Scancodes eine 0 gesendet werden. Nach der Bestätigung sendet die Tastatur in diesem Fall den aktuell eingestellten Scancode.

**Beispiel** Das Setzen von Scancode 3 bzw. Lesen des Scancodes wird wie folgt erreicht:

```
Setzen:  M:F0 S:FA M:03 S:FA
Lesen:   M:F0 S:FA M:00 S:FA S:03
```

Hier kennzeichnet das Voranstellen von `M:`, dass das jeweilige Datenbyte vom Mikrocontroller (Master) versandt wurde und das Voranstellen von `S:`, dass das jeweilige Datenbyte von der Tastatur (Slave) versandt wurde.

## Achtung

Wenn ein zur Tastatur gesendeter Befehl nicht durch die Tastatur quittiert wurde, kann das bedeuten, dass dieser durch die Tastatur nicht richtig verarbeitet wurde. Um sichergehen zu können, dass die Tastatur den Befehl erhalten hat, empfiehlt es sich, den Befehl solange periodisch neu zu versenden, bis dieser quittiert wurde.

### Alle Tasten aktivieren: FA

Nach Ändern des Scancodes kann es vorkommen, dass Tasten deaktiviert werden. In dem Fall besteht die Möglichkeit, dass für manche Tasten der Break- oder Makecode nicht gesendet wird oder die Tastenwiederholung deaktiviert ist. Mit Hilfe des Befehls $\texttt{FA}_h$ werden alle Make- und Breakcodes sowie die Tastenwiederholung aktiviert. Dies wird von der Tastatur mit $\texttt{FA}_h$ bestätigt.

**Hinweis**

Der Bestätigungscode und der hier vorgestellte Aktivierungscode werden durch die gleiche Zahl repräsentiert. Die Bedeutung ist allerdings davon abhängig, welcher der beiden Kommunikationspartner diese versandt hat. Gleiches gilt für den im vorherigen Abschnitt vorgestellten Code zum Ändern des Scancodes, welcher durch die gleiche Zahl dargestellt wird wie der Präfix eines Breakcodes.

### LEDs steuern: ED

Mit dem Befehl $\texttt{ED}_h$ lassen sich die drei LEDs der Tastatur an- und ausschalten. Nach der Übertragung von $\texttt{ED}_h$ antwortet die Tastatur mit dem Bestätigungscode $\texttt{FA}_h$. Anschließend kann ein Byte übertragen werden, welches den neuen Zustand der drei LEDs codiert, siehe dazu Tabelle 6.4. Dieses Byte wird von der Tastatur wieder mit $\texttt{FA}_h$ bestätigt.

| LED Name | Bit-Position |
|---|---|
| Num (Ziffernblock) | $\texttt{000000x0}_b$ |
| Feststelltaste (CAPS) | $\texttt{00000x00}_b$ |
| Rollen (Scroll) | $\texttt{0000000x}_b$ |

Tabelle 6.4: Bit-Positionen der LEDs

**Achtung**

Das Verändern der LEDs ändert nicht die Funktionen der Tastatur. Die Semantik der LEDs muss der Master festlegen, wenn die von der Tastatur gesendeten Codes verarbeitet werden. Aus Sicht der Tastatur handelt es sich hier um drei LEDs ohne jegliche Bedeutung.

**Beispiel** Das Setzen der Num und Rollen LEDs wird wie folgt erreicht:

```
M:ED S:FA M:03 S:FA
```

### 6.3.5 Behandlungsfunktionen

Es kommt häufig zu Situationen, in denen auf das Eintreten von Ereignissen reagiert werden muss. Bei einer Schichtenarchitektur, wie sie im nachfolgenden Abschnitt eingeführt wird, gibt es dazu grundsätzlich zwei Möglichkeiten.

#### 1. Polling

Erstens kann auf das Eintreten des Ereignisses aktiv gewartet werden (*Polling*). Dabei ruft jede Schicht die direkt darunter liegende Schicht auf, bevor das Ereignis auftritt. Sobald die unterste Schicht (der Empfänger) eine Nachricht erhält, terminieren die aufgerufenen Funktionen und die Kontrolle wird an die höchste beteiligte Schicht zurückgegeben.

#### 2. Interrupts

Zweitens können Ereignisse durch die Schichten nach oben weitergeleitet werden, sobald ein Ereignis in einer der unteren Schichten verzeichnet wird. Das Feststellen eines Ereignisses wird mit Hilfe eines Interrupts realisiert. Dieser Ansatz erfordert, dass die untere Schicht weiß, welcher oberen Schicht das jeweilige Ereignis mitgeteilt werden soll. Dafür wird eine Funktion der oberen Schicht in Form eines Funktionszeigers als sogenannte *Callback-Funktion* registriert. Dieser *Callback*, aufgerufen von der unteren Schicht, behandelt dann das Ereignis (engl. *event handling*).

## Lernerfolgsfragen

- Was versteht man unter dem Begriff *Polling*?
- Was versteht man unter dem Begriff *Callback*?
- Wie unterscheiden sich diese Konzepte?
- In welche Richtung (im Sinne der Schichten) läuft der Kontrollfluss bei diesen zwei Herangehensweisen?

## 6.4 Hausaufgaben

### Achtung

Passen Sie das `define VERSUCH` auf die aktuelle Versuchsnummer an.

Implementieren Sie die in den nächsten Abschnitten beschriebenen Funktionalitäten. Halten Sie sich an die hier verwendeten Namen und Bezeichnungen für Variablen, Funktionen und Definitionen.
Lösen Sie alle hier vorgestellten Aufgaben zu Hause mithilfe von Microchip Studio 7 und schicken Sie die dabei erstellte und funktionsfähige Implementierung über Moodle ein. Ihre Abgabe soll dabei die `.atsln`-Datei, das Makefile, sowie den Unterordner mit den `.c/.h`-Dateien inklusive der `.xml/.cproj`-Dateien enthalten. Beachten Sie bei der Bearbeitung der Aufgaben die angegebenen Hinweise zur Implementierung! Ihr Code muss ohne Fehler und ohne Warnungen kompilieren sowie die Testtasks mit aktivierten Compileroptimierungen bestehen. Wie Sie die Optimierungen einschalten ist dem begleitenden Dokument in Abschnitt *6.3.2 Probleme bei der Speicherüberwachung* zu entnehmen.

### Achtung

Verwenden Sie zur Prüfung auf Warnungen den Befehl „Rebuild Solution“ im „Build“-Menü des Microchip Studio 7. Die übrigen in der grafischen Oberfläche angezeigten Buttons führen nur ein inkrementelles Kompilieren aus, d.h. es werden nur geänderte Dateien neu kompiliert. Warnungen und Fehlermeldungen in unveränderten Dateien werden dabei nicht ausgegeben.

### Hinweis

Da die Ports B und D für andere Zwecke benötigt werden, muss in diesem Versuch von der Benutzung des externen SRAMs abgesehen werden. Somit ist es ratsam, den externen SRAM wie auch den dazugehörigen Heap aus der Liste Ihrer Speichermedien bzw. Heaps zu entfernen.

### 6.4.1 Einleitung

Im Grundlagenkapitel 6.3 wurde bereits eine Unterteilung in verschiedene Abstraktionsebenen dargestellt. Die erste Ebene beinhaltet die Außenbeschaltung des Mikrocontrollers. Diese wurde im Grundlagenkapitel 6.3.1 vorgestellt. Die zweite in Abschnitt 6.3.2 vorgestellte Ebene beinhaltete den Austausch von Datenbytes. Die dritte Ebene, welche in Kapitel 6.3.3 vorgestellt wurde, realisiert die Interpretation der übertragenen Datenbytes.
Diese Drei-Schichten-Architektur wird im Folgenden genauer erläutert. Die erste Schicht beinhaltet die Konfiguration der I/O Schnittstelle. Die darüberliegende Schicht realisiert das Senden und Empfangen einzelner Bytes. In Schicht 3 werden die empfangenen Bytes interpretiert. Schließlich wird die darauf aufbauende Anwendungsschicht 4 betrachtet, welche die erkannten Tastendrücke verarbeitet.
Diese Unterteilung wird in Abbildung 6.3 veranschaulicht. Die Interaktion der Schichten wird durch die ausgetauschten Daten angegeben.

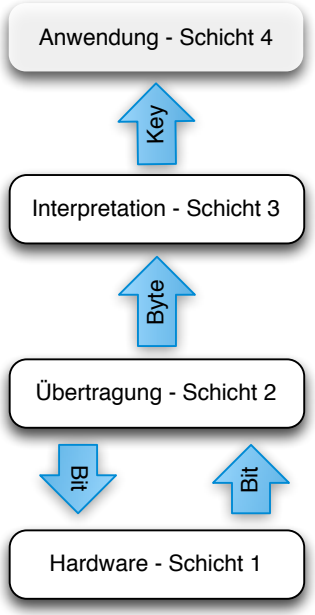


Abbildung 6.3: Illustration aller Schichten

### 6.4.2 Schicht 1: Übertragungsrichtung

Schicht 1 initialisiert die Hardwareschnittstellen für die Übertragung zwischen Tastatur und dem Mikrocontroller.
Vervollständigen Sie dazu zunächst in `keyb_usart.h` bereits angelegte Definitionen, welche die verwendeten Pins und Ports spezifizieren (Pinbelegung 6.6). Achten Sie bei jeder

Verwendung eines Pins darauf, ob dieser korrekt als *Input* oder *Output* konfiguriert wurde. Implementieren Sie anschließend die Initialisierungsfunktion `keyb_usart_init` in `keyb_usart.c`, die sich an den folgenden Schritten orientiert:

1. Da die Tastatur ihre Stromversorgung aus den Ports des Mikrocontrollers bezieht, müssen Sie diese zunächst geeignet auf $V_{cc}$ und $GND$ schalten.
2. Stoppen Sie anschließend die Kommunikation, indem Sie die Datenleitung auf 1 und die Clockleitung auf 0 setzen.
3. Initialisieren Sie die USART0 Komponente (ungerade Parität, 8 Bit Datenrahmen und ein Stoppbit, Receiver aktiv). Beachten Sie dazu im Datenblatt des Mikrocontrollers Kapitel *17.10.4*. Außerdem ist zu beachten, dass die Clock nicht vom Mikrocontroller, sondern von der Tastatur erzeugt wird. Insbesondere muss also keine Baudrate konfiguriert werden. Berücksichtigen Sie dabei außerdem die Hinweise im nächsten Abschnitt.
4. Teilen Sie der Tastatur mit, dass sie in den Sendemodus wechseln darf, indem Sie die Daten- und Clockleitung auf 1 setzen und anschließend den Datenpin als Eingang definieren.

**Achtung**

Achten Sie darauf, dass die Initialisierung nicht unterbrochen wird. Deaktivieren Sie also die Interrupts für die Dauer der Funktion.

### 6.4.3 Schicht 2: Bytes senden und empfangen

Schicht 2 implementiert den Austausch von Datenbytes und Befehlen, sowie die Initialisierung des Tastaturtreibers.

#### Senden von Bytes

Im Folgenden wird es nötig sein, Befehle an die Tastatur zu übermitteln. Benutzen Sie dazu die im Codegerüst bereits vorgegebene Funktion `keyb_transmit`. Diese gibt wahr zurück, wenn die Übertragung vollständig war. Beachten Sie, dass die Funktion nicht auf den Empfang des Bestätigungsbytes wartet, sondern lediglich bestätigt, dass die Daten korrekt gesendet wurden. Die Übertragung wird automatisch vorzeitig abgebrochen, falls die Tastatur zu lange nicht antwortet. Dies verhindert ein Einfrieren des Tastaturtreibers.

### Setzen des Scancodes

Implementieren Sie die Funktion `keyb_setScanCodeSet` in `keyb_usart.c`. Beachten Sie die Hinweise im Grundlagenkapitel 6.3.4. Sie können hier die Funktion `keyb_receive` benutzen, die erst im Anschluss implementiert wird, um die Bestätigung der Tastatur zu empfangen. Die Funktion soll den aktiven Scancode zurückgeben und, analog zu `keyb_usart_init`, atomar ablaufen.

### Setzen der LEDs

Implementieren Sie die Funktion `keyb_updateLEDs`, die den Zustand jeder LED als Parameter übergeben bekommt und an die Tastatur überträgt. Befolgen Sie das in Kapitel 6.3.4 beschriebene Protokoll. Auch diese Funktion soll atomar ablaufen.

### Kommunikation starten

So wie Schicht 1, muss auch diese Schicht initialisiert werden. Dazu wird die Funktion `keyb_start` implementiert:

1. Rufen Sie `keyb_usart_init` auf, um sicherzustellen, dass die Hardware korrekt konfiguriert ist. Warten Sie anschließend für mindestens 10 ms den Startvorgang der Tastatur ab.
2. Konfigurieren Sie die Tastatur, damit diese Scancode 3 anstatt Scancode 2 verwendet. Benutzen Sie hierfür die Funktion `keyb_setScanCodeSet`. Aktivieren Sie anschließend alle Tastenaktionen durch Senden des in Abschnitt 6.3.4 erläuterten Befehls. Warten Sie auch hier die Bestätigung ab. Dieser Schritt sollte ebenfalls atomar geschehen.
3. Rufen Sie `keyb_init_input_processor` auf, um die dritte Schicht zu initialisieren.
4. Aktivieren Sie den Interrupt der USART Komponente des Mikrocontrollers, der bei einem empfangenen Byte ausgelöst wird.

**Hinweis**

Wiederholen Sie das Senden eines Befehls solange, bis eine Bestätigung von der Tastatur empfangen wird.

### Aktivierung/Deaktivierung des Receivers

Die USART Komponente des Mikrocontrollers erlaubt es, Receiver und Transmitter einzeln zu aktivieren oder zu deaktivieren. Diese Funktionalität wird in `keyb_transmit` be-

nötigt. Implementieren Sie dazu die Funktion `keyb_activateReceiver`, die den Receiver des USART aktiviert, und `keyb_deactivateReceiver`, die diesen wieder deaktiviert.

#### Empfangen von Bytes

Das Empfangen von Bytes kann, wie im Grundlagenkapitel 6.3 beschrieben, durch die USART Schnittstelle übernommen werden.
Da viele Funktionen explizit auf eine Antwort der Tastatur warten müssen, ist es notwendig, für das Empfangen *beide* der nachfolgend aufgeführten Empfangsfunktionen zu implementieren.

**Polling:** Implementieren Sie zunächst eine Funktion `keyb_receive`. Fragen Sie in der Funktion solange das `RXC0`-Bit im Register `UCSR0A` der USART Schnittstelle ab, bis signalisiert wird, dass Daten zum Empfang bereit stehen. Lesen Sie anschließend das empfangene Datenbyte aus dem Register `UDR0`. Beachten Sie außerdem die Hinweise in den Kapiteln *17.7.1* und *17.7.3* des Datenblatt des Mikrocontrollers.
Beachten Sie die Möglichkeit, dass ein Befehl nicht richtig von der Tastatur empfangen wurde und diese deswegen auch keine Antwort liefert. Um diesen Fall abzufangen, soll `keyb_receive` nach einem Timeout abbrechen.
Hierzu können Sie das vorgegebene Makro `POLL_EVENT_PANIC` verwenden. Dieses Makro erhält die Argumente `CONDITION` und `PANIC`: `CONDITION` bezeichnet dabei eine Bedingung, auf die gewartet werden soll und `PANIC` einen Codeblock, welcher ausgeführt wird, sobald ein Timeout für die angegebene `CONDITION` eintritt. Die Funktion soll im Fall eines Timeouts eine 0 zurückliefern und die vorangehende Übertragung sollte (außerhalb der Funktion) wiederholt werden.

**Interruptgesteuert:** Implementieren Sie eine *Interrupt Service Routine* (ISR), die in Register UDR0 eingehende Datenbytes verarbeitet. Die zugehörige Funktionssignatur wurde bereits im Codegerüst angelegt.
Rufen Sie in dieser ISR die Funktion `keyb_input_process` mit dem empfangenen Byte als Parameter auf, um die Verarbeitung in der dritten Schicht anzustoßen.

### 6.4.4 Schicht 3: Daten interpretieren

Schicht 3 ist für die Interpretation der Daten zuständig. Dabei wird auf die bereits implementierte Schicht 2 zurückgegriffen.

#### Klassifizierung der Tasten

Für diesen Versuch wurden alle Tasten klassifiziert. Die gewählten *Tastentypen* sind

1. Druckbare Zeichen auf dem Hauptblock (`KEYB_KT_PRINTABLE`)
2. Druckbare Zeichen auf dem Nummernblock (`KEYB_KT_NUMPRINTABLE`)
3. Steuertasten links von der Leertaste (`KEYB_KT_LCONTROL`)

4. Steuertasten rechts von der Leertaste (`KEYB_KT_RCONTROL`)
5. F-Tasten (`KEYB_KT_FKEY`)
6. Pfeiltasten (`KEYB_KT_ARROW`)

Die Tastentypen sind im Codegerüst bereits als `enum KeyType` angelegt worden. Die Angaben in den Klammern entsprechen den gewählten Bezeichnern im Codegerüst. Die genaue Aufteilung wird in Abbildung 6.4 gezeigt.

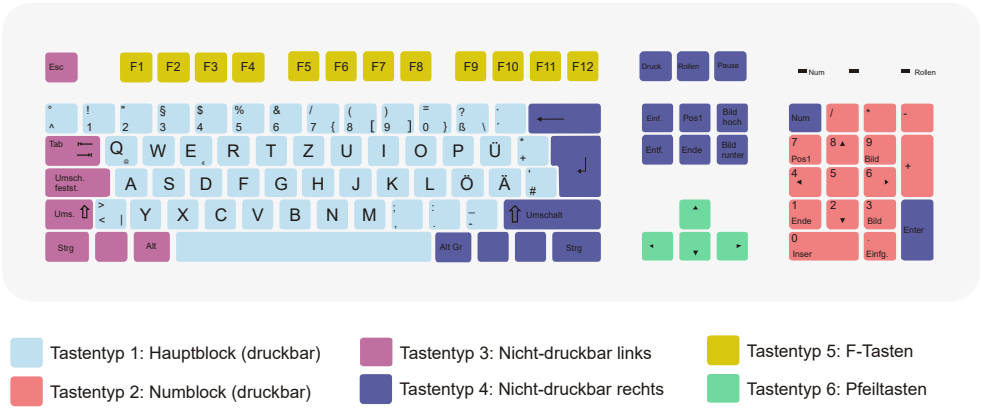


Abbildung 6.4: Klassifizierung der Tasten

Dies erlaubt eine einfache Bearbeitung von ähnlichen Tasten und die Darstellung aller Tasten durch ein kleines Alphabet. Das hat zur Konsequenz, dass eine Taste nur eindeutig von einem Tupel, bestehend aus einem Tastentyp und einem Zeichen, identifiziert werden kann. Beispielsweise repräsentiert (1,`'5'`) die Taste mit der Aufschrift $\frac{\%}{5}$ über der Taste *R*, während (2,`'5'`) die Taste mit der Aufschrift $\boxed{5}$ in der Mitte des Nummernblocks repräsentiert. Die komplette Zuordnung der Tasten ist in den Lookup-Tabellen der Datei `keyb_scancode.c` codiert. Passende Definitionen für nicht-druckbare Tasten sind in der Datei `keyb_usart.h` definiert.

### Datentypen von Schicht 3

Wie dem vorgegebenen Codegerüst in `keyb_usart.h` entnommen werden kann, werden vier Datentypen eingeführt: `struct Key`, `enum KeyType`, `enum KeyMeta` und `enum KeyModifier`. Die Struktur `Key` wird verwendet, um eine Aktion eindeutig zu codieren. Dazu gehört die Information, ob es sich um einen Make- oder Breakcode handelt, welche im Attribut `Key.meta` (des Datentyps `KeyMeta`) gespeichert wird. Die tatsächlich gedrückte Taste wird in `Key.key` (des Datentyps `uint8_t`) und der Tastentyp, der zu der dieser Taste gehört, in `Key.type` (des Datentyps `KeyType`) gespeichert. Während des Tastendrucks aktive Modifier (etwa die Shift-Taste) werden im Feld `Key.modifier` (des

Datentyps `KeyModifier`) gespeichert. Beachten Sie, dass bei einem Tastendruck mehrere Modifier gleichzeitig aktiv sein können.

### Datenprozessoren

Implementieren Sie zunächst die Funktion `keyb_init_input_processor`, die die internen Zustände und LEDs der Tastatur auf einen definierten Zustand zurücksetzt. Die in der dritten Schicht vorzuhaltenden Daten werden im Folgenden näher erläutert.

**Input Prozessor** Implementieren Sie in der Datei `keyb_processor.c` die Funktion `keyb_input_process`, welche die von Schicht 2 empfangenen Bytes decodiert. Die Informationen, welche Taste gedrückt oder losgelassen wurde und welchem Tastentyp diese angehört, hinterlegen Sie in einer globalen Variablen vom Typ `Key`. Sie können die vorgegebenen Funktionen `keyb_translateInputToKey` und `keyb_translateInputToKeyType` zum Interpretieren von Scancode 3 Codes verwenden. Beachten Sie, dass Sie bei einem Breakcode zwei Byte verarbeiten müssen, um ein Zeichen zu erzeugen. Die Funktion `keyb_input_process` soll `keyb_char_modifiers` aufrufen, falls ein gültiger Wert decodiert werden konnte. Das erste Byte eines Breakcodes und ungültige Eingaben erzeugen keinen neuen Wert.

**Modifiers** In `keyb_char_modifiers` soll nun der neu decodierte Wert weiter verarbeitet werden. Zunächst muss der interne Zustand des Tastaturtreibers aktualisiert werden. Wurde etwa die Shift-Taste gedrückt, muss intern vermerkt werden, dass nun der Shift-Modifier (`KEYB_MOD_SHIFT`) aktiv ist. Dies kann in der oben bereits erwähnten globalen Variablen vom Typ Key vermerkt werden. Analog gilt dies für alle weiteren Modifier, die im Codegerüst bereits als `enum KeyModifier` vorgegeben wurden. Ebenfalls muss der Zustand der Feststell-, Num- und Rollen-Taste mit den ihnen zugeordneten LEDs aktualisiert werden. Handelt es sich bei dem Wert um ein druckbares Zeichen, müssen je nach Zustand der Modifier und der Feststell- bzw. Num-Taste weitere Translationen vorgenommen werden. Dazu finden Sie im Codegerüst die Funktionen `keyb_shiftChar` und `keyb_unNumChar`. Rufen Sie anschließend die Funktion `keyb_char_process` auf, deren Aufgabe es ist, die zuvor durch die Anwendungsschicht registrierten Behandlungsfunktionen aufzurufen.

**Beispiel** Das Verarbeiten eines Tastendrucks in der dritten Schicht ist beispielhaft in Abbildung 6.5 dargestellt. Im Beispiel wurde angenommen, dass vorher die Shift-Taste gedrückt, aber nicht losgelassen wurde. Daher wird `key` zunächst in `keyb_input_process` auf '`0`' gesetzt und anschließend in `keyb_char_modifiers` zu '`=`' übersetzt.

### Behandlungsfunktionen von Schicht 3

Schicht 3 ermöglicht der Anwendungsschicht Behandlungsfunktionen zu registrieren, die bei Tastendrücken aufgerufen werden. Es soll die Möglichkeit geboten werden, sowohl eine Behandlungsfunktion für alle Tastendrücke, als auch Behandlungsfunktionen für

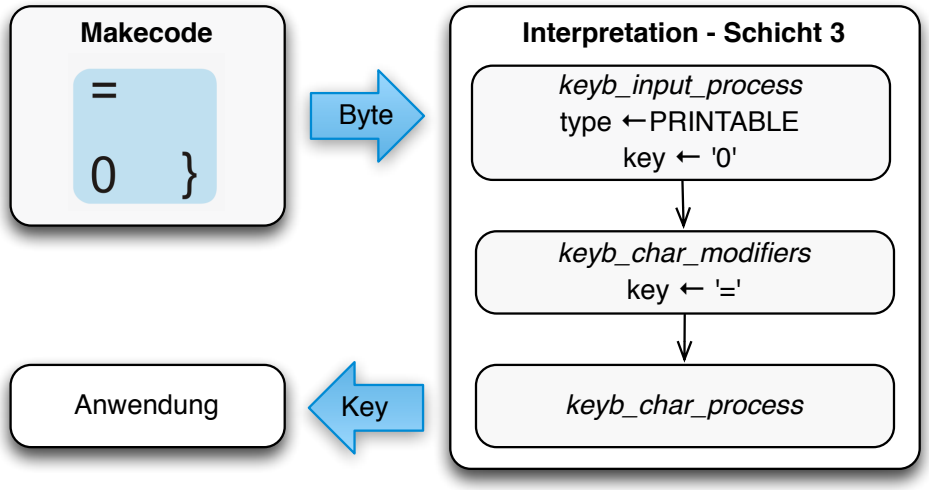


Abbildung 6.5: Interpretation eines Tastendrucks

jeden unterschiedlichen Tastentypen (vgl. 6.4.4) zu registrieren. Behandlungsfunktionen sind vom Typ `KeyHandler` und erhalten als einzigen Parameter einen `Key`. Die beispielhafte Verwendung kann im zur Verfügung gestellten Texteditor nachvollzogen werden. Implementieren Sie die Funktion `keyb_set_processor`, die eine Behandlungsfunktion für einen spezifischen Tastentypen registriert. Die Behandlungsfunktionen können Sie etwa in einem globalen Array hinterlegen. Vervollständigen Sie anschließend die Funktion `keyb_set_main_processor`, die die Behandlungsfunktion festlegt, die alle Tastendrücke empfangen soll.

### 6.4.5 Schicht 4: Anwendungsschicht

In Schicht 4 liegen die Anwendungsprogramme. Diese nutzen die bereits implementierte Schicht 3 zur Realisierung ihrer Funktionalität.

#### Texteditor

Im Moodle steht eine Testanwendung bereit, die alle relevanten Funktionen der Tastatur nutzt. Diese verarbeitet alle Tastatureingaben und gibt diese auf dem LCD aus. Alle Tasten und LEDs müssen sich so verhalten, wie es von einem Texteditor erwartet würde. Die einzige Ausnahme bilden die Funktionstasten, die den zur Verfügung stehenden Zeichensatz auf dem LCD anzeigen. Stellen Sie sicher, dass diese Anwendung korrekt funktioniert.

### Eigene Anwendung

Spezifizieren Sie eine eigene Tastaturanwendung und implementieren Sie diese. Sie können sich hier eine beliebige Anwendung ausdenken, solange diese auf sinnvolle Art und Weise Gebrauch von der Tastatur macht. Ihre Anwendung muss sich von dem vorgegebenen Texteditor unterscheiden. Mögliche Anwendungsfelder sind etwa Spiele, Systemprogramme oder wissenschaftliche Programme. Beispiele für solche Anwendungen sind etwa eine Systemshell oder ein endlicher Automat.

### 6.4.6 Zusammenfassung

Folgende Übersicht listet alle Typen, Funktionen und Aufgaben auf. Alle aufgelisteten Punkte müssen zur Teilnahme am Versuch bis zur Abgabefrist bearbeitet und hochgeladen werden. Diese Übersicht kann als Checkliste verwendet werden und ist daher mit Checkboxen versehen.

- ☐ Schicht 1: Übertragungsrichtung (`keyb_usart`)
  - ☐ Anlegen und Ergänzen der Definitionen in der `keyb_usart.h`
  - ☐ `void keyb_usart_init()`
  - ☐ `void keyb_activateReceiver()`
  - ☐ `void keyb_deactivateReceiver()`
- ☐ Schicht 2: Bytes senden und empfangen (`keyb_usart`)
  - ☐ `void keyb_start()`
  - ☐ `uint8_t keyb_receive()`
  - ☐ `uint8_t keyb_setScanCodeSet(uint8_t scanCodeSet)`
  - ☐ `void keyb_updateLEDs(bool num, bool caps, bool roll)`
  - ☐ *Interrupt Service Routine* für empfangene Bytes
- ☐ Schicht 3: Bytes interpretieren (`keyb_processor`)
  - ☐ `void keyb_init_input_processor()`
  - ☐ `void keyb_input_process(uint8_t input)`
  - ☐ `void keyb_char_modifiers()`
  - ☐ `void keyb_char_process()`
  - ☐ `void keyb_set_processor(KeyType, KeyHandler)`
  - ☐ `void keyb_set_main_processor(KeyHandler)`
- ☐ Schicht 4: Anwendungsschicht
  - ☐ Testen der Implementierung mit vorgegebenem Texteditor
  - ☐ Entwicklung einer eigenen Tastaturanwendung in der `progs.c`

## 6.5 Präsenzaufgaben

### Texteditor

Demonstrieren Sie einem Betreuer die korrekte Funktionsweise des vorgegebenen Texteditors.

### Eigene Anwendung

Erläutern Sie einem Betreuer das Einsatzgebiet Ihrer Anwendung und führen Sie deren Funktion vor.

## 6.6 Pinbelegung

| Port | Pin | Belegung |
|---|---|---|
| Port A | A0 | LCD Pin 1 (D4) |
| | A1 | LCD Pin 2 (D5) |
| | A2 | LCD Pin 3 (D6) |
| | A3 | LCD Pin 4 (D7) |
| | A4 | LCD Pin 5 (RS) |
| | A5 | LCD Pin 6 (EN) |
| | A6 | LCD Pin 7 (RW) |
| | A7 | frei |
| Port B | B0 | Tastatur: Clock |
| | B1 | Tastatur: GND |
| | B2 | Tastatur: GND |
| | B3 | Tastatur: GND |
| | B4 | Tastatur: $V_{cc}$ |
| | B5 | Tastatur: $V_{cc}$ |
| | B6 | Tastatur: $V_{cc}$ |
| | B7 | Tastatur: $V_{cc}$ |
| Port C | C0 | Button 1: Enter |
| | C1 | Button 2: Down |
| | C2 | Reserviert für JTAG |
| | C3 | Reserviert für JTAG |
| | C4 | Reserviert für JTAG |
| | C5 | Reserviert für JTAG |
| | C6 | Button 3: Up |
| | C7 | Button 4: ESC |
| Port D | D0 | Tastatur: Data |
| | D1 | frei |
| | D2 | frei |
| | D3 | frei |
| | D4 | frei |
| | D5 | frei |
| | D6 | frei |
| | D7 | frei |

Pinbelegung für Versuch 6 (*PS/2-Tastatur*).