

Master-Projekt

Embedded Rust
Rust auf AVR-Mikrocontrollern

31. August 2018

Eingereicht von:

Niklas Kühl

inf102861@fh-wedel.de

Betreuer:

Prof. Dr. Ulrich Hoffmann

uh@fh-wedel.de

Inhaltsverzeichnis

1	Einleitung	1
2	Einrichtung der Entwicklungsumgebung	2
2.1	Systemkonfiguration	2
2.2	Installation des AVR-Rust-Compilers	3
2.2.1	Grundlegendes Vorgehen	3
2.2.2	Übersetzungsvorgang (make)	4
2.2.3	Installationsvorgang (make install)	4
2.2.4	Rust-Toolchain konfigurieren (rustup toolchain)	4
2.2.5	Installation der AVR-Toolchain	5
2.2.6	Optional: udev-Regel anlegen	5
2.3	Hello World AVR-Rust	6
3	Grundlagen von AVR-Rust	7
3.1	Registerzugriff	7
3.2	Interrupt-Routinen	8
3.3	Verwendung des EEPROM	9
3.4	Makros	10
3.4.1	Übersicht	10
3.4.2	Fragment Specifier	10
3.4.3	Pattern Matching	11
3.5	Panic-Funktion	12
3.6	Buffer Overflow	13
3.7	Integer Overflow	13
3.8	Uninitialisierte Variablen	14
3.9	Performance	14
4	Beispielprojekt	16
4.1	Hardware	16
4.2	WiFi-Modul	17
4.2.1	Protokoll des ESP-8266	17
4.2.2	DSL	18
4.2.3	Makro-basierte DSL	20
4.3	Temperatursensor	22
4.3.1	Protokoll des Dallas 18B20	22
4.3.2	Implementierung des Protokolls	23
4.4	Probleme bei der Entwicklung	24
4.4.1	Fehlerhafter Code	24
4.4.2	Nicht kompilierbares Projekt	25

5 Fazit	26
6 Quellcode	27
Literaturverzeichnis	28

Abbildungsverzeichnis

3.1	Aufruf der benutzerdefinierten panic-Funktion in der libcore	13
4.1	Schaltplan	16
4.2	Struktur der DSL	19

Tabellenverzeichnis

2.1	Verwendete Software	2
2.2	Verwendete Hardware	2
3.1	Ergebnisse der Performance-Messung	15
3.2	Größe der erzeugten Binärdateien	15
4.1	Verwendete Befehle des DS18B20	23

Quellcodeverzeichnis

2.1	Kompilierung und Installation des AVR-Rust-Compilers [1]	3
2.2	Anlegen des Installationsordners	4
2.3	Konfiguration der Rust-Toolchain	4
2.4	Installation der AVR-Toolchain	5
2.5	Installation avrdude 6.3	5
2.6	udev-Regel	5
2.7	Zur Kompilierung nötige Umgebungsvariablen [2]	6
2.8	Übersetzungsbefehl [2]	6
3.1	Registerzugriff in Rust	7
3.2	Timer-Initialisierung mit direktem Registerzugriff	8
3.3	Timer-Initialisierung mit <i>arduino</i> -Bibliothek	8
3.4	Interrupt Service Routine in C	8
3.5	Interrupt Service Routine in Rust	9
3.6	Import einer ISR im Hauptmodul	9
3.7	Verwendung des EEPROM in Rust	9
3.8	Verwendung des EEPROM in C	10
3.9	Makro in Rust	10
3.10	Rekursives Makro für benutzerdefinierte Syntax	11
3.11	Panic-Funktion	12
3.12	unsafe-Block mit Verwendung eines Raw-Pointers	14
4.1	Beispielsitzung mit AT-Kommandos	17
4.2	Beispiel AT-Kommando in Rust	18
4.3	Rust-Beispielsitzung mit Verwendung von Methoden	20
4.4	Makro zur Parameterverarbeitung	21
4.5	Einige Beispiele für die Verwendung des DSL-Makros	21
4.6	Expansion des Makros <code>esp_cmd</code>	21
4.7	Rust-Beispielsitzung mit Verwendung von Makros	22
4.8	ISR zum Lesen des 1-Wire-Bus	24
4.9	Nicht korrekt übersetzter Codeabschnitt	25
4.10	Fehlermeldung des Compilers (1)	25
4.11	Fehlermeldung des Compilers (2)	25

1 Einleitung

In der heutigen Welt spielt Automatisierung eine wichtige Rolle. Für die Steuerung von technischen Geräten werden dabei häufig Mikrocontroller eingesetzt. Programme für Mikrocontroller werden meistens in der Programmiersprache C geschrieben. Diese hardwarenahe Sprache gibt dem Entwickler nur wenig Hilfe zur Erkennung oder Vermeidung von Fehlern. Dadurch werden Programmierfehler oft erst im produktiven Einsatz entdeckt.

Die Programmiersprache Rust soll eine sichere Alternative zu C darstellen. Durch in die Sprache integrierte Überprüfungen werden bestimmte Fehlerarten vermieden, wodurch robustere Programme geschrieben werden können. In dieser Arbeit soll die Tauglichkeit von Rust auf AVR-Mikrocontrollern (AVR-Rust) untersucht werden.

In Kapitel 2 werden die notwendigen Schritte für die Einrichtung einer Entwicklungsumgebung zur Programmierung von AVR-Mikrocontrollern in Rust beschrieben.

In Kapitel 3 werden grundlegende Eigenschaften von AVR-Rust untersucht und mit C verglichen. Dazu gehören notwendige Konstrukte zur Programmierung von Mikrocontrollern wie Registerzugriff und Interrupt-Funktionen. Weiterhin werden Vor- und Nachteile von AVR-Rust analysiert.

In Kapitel 4 wird dann ein Beispielprojekt vorgestellt, welches die Möglichkeiten von Rust auf AVR-Mikrocontrollern weiter untersucht. Eine wichtige Aufgabe ist hierbei die Entwicklung einer sicheren Sprache zur Kommunikation mit einem WiFi-Modul.

In Kapitel 5 wird die Arbeit dann zusammengefasst und ein Ausblick gegeben.

2 Einrichtung der Entwicklungsumgebung

In diesem Kapitel wird die Installation der Programme beschrieben, welche für die Programmierung von AVR-Mikrocontrollern mit Rust erforderlich sind. Dabei wird auch auf mögliche Fehler bei der Installation eingegangen.

2.1 Systemkonfiguration

Folgende Programmversionen wurden für die Entwicklung genutzt:

Anwendung	Version	Beschreibung
Betriebssystem	Ubuntu 16.04.4 LTS (64-Bit)	-
rustc (Desktop)	1.26.2	Rust-Compiler für Desktop-Programme
rustup	1.11.0	Rust Update Tool
cargo	1.26.0	Rust-Buildsystem und Paketmanager
xargo	0.3.11	Alternative zu cargo, erlaubt die Verwendung einer benutzerdefinierten Standardbibliothek
rustc (AVR)	1.22.0-dev	Rust-Compiler für AVR-Programme
avr-gcc	4.9.2	AVR-C-Compiler und Linker
avrdude	6.3	AVR-Programmiertool
g++	5.4.0	C++ Compiler

Tabelle 2.1: Verwendete Software

Folgende Hardware kam bei der Entwicklung zum Einsatz:

Hardware	Beschreibung
Mikrocontroller (Entwicklungsboard)	ATmega328P Xplained Mini
Temperatursensor	Dallas 18B20
WiFi-Modul	ESP-8266l

Tabelle 2.2: Verwendete Hardware

2.2 Installation des AVR-Rust-Compilers

2.2.1 Grundlegendes Vorgehen

Um die Entwicklungsumgebung einzurichten, sollte eine normale Rust-Installation vorhanden sein (siehe [3]). Außerdem sollte das alternative Rust-Buildsystem *xargo* installiert sein.

Für den Rust-Compiler für AVR-Mikrocontroller gibt es keine vorkompilierten Binärdateien. Daher muss der Compiler für das verwendete System neu übersetzt werden. Das grundlegende Vorgehen ist in Listing 2.1 beschrieben [1]. In den folgenden Kapiteln wird auf Eigenheiten und Probleme bei der Installation eingegangen.

```
1  # Grab the avr-rust sources
2  git clone https://github.com/avr-rust/rust.git
3
4  # Create a directory to place built files in
5  mkdir build && cd build
6
7  # Generate Makefile using settings suitable for an experimental
   ↪  compiler
8  ../rust/configure \
9  --enable-debug \
10 --disable-docs \
11 --enable-llvm-assertions \
12 --enable-debug-assertions \
13 --enable-optimize \
14 --prefix=/opt/avr-rust
15
16 # Build the compiler, optionally install it to /opt/avr-rust
17 make
18 make install
19
20 # Register the toolchain with rustup
21 rustup toolchain link avr-toolchain $(realpath $(find . -name
   ↪  'stage1'))
22
23 # Optionally enable the avr toolchain globally
24 rustup default avr-toolchain
```

Listing 2.1: Kompilierung und Installation des AVR-Rust-Compilers [1]

2.2.2 Übersetzungsvorgang (make)

Die ersten Schritte bis zum eigentlichen Übersetzungsvorgang sollten ohne Probleme ausgeführt werden. Der aufwändigste Schritt ist der eigentliche Übersetzungsprozess. Dieser kann einige Zeit (mehrere Stunden) in Anspruch nehmen. Im folgenden sind einige Fehler aufgelistet, welche beim Übersetzungsvorgang auftraten, sowie mögliche Lösungsansätze:

- Der Übersetzungsprozess bricht mit einem Speicherfehler ab
Dieser Fehler trat auf einem 32-Bit-System auf, weswegen auf ein 64-Bit-System gewechselt wurde. Zum Zeitpunkt der Entwicklung war die Übersetzung auf 32-Bit-Systemen nicht möglich. Dies ist ein bekannter Fehler¹.
- Der Übersetzungsprozess wird vom Betriebssystem beendet
Es war nicht genügend Arbeitsspeicher verfügbar. In einer virtuellen Maschine mit 4GB RAM schlug die Übersetzung fehl. Nachdem der Arbeitsspeicher auf 6GB erhöht wurde, war die Übersetzung erfolgreich.
- Fehler beim Linken des Programms: *cannot find -lffi*
Es mussten die fehlenden Pakete *libffi6* und *libffi-dev* installiert werden.

2.2.3 Installationsvorgang (make install)

Der Befehl `make install` installiert den erstellten AVR-Rust-Compiler in das Verzeichnis `/opt/avr-rust`. Dies schlug zunächst fehl, da hierfür Root-Rechte erforderlich sind. Eine Ausführung von `make install` mit Root-Rechten schlug ebenfalls fehl. Daher wurde der Ordner `/opt/avr-rust` manuell mit vollen Berechtigungen angelegt (Listing 2.2). Dadurch konnte die Installation erfolgreich ausgeführt werden konnte. Im Anschluss wurden die Rechte des Ordners dann wieder eingeschränkt.

```
sudo mkdir -m 777 /opt/avr-rust
```

Listing 2.2: Anlegen des Installationsordners

2.2.4 Rust-Toolchain konfigurieren (rustup toolchain)

Damit die in `/opt/avr-rust` installierte AVR-Toolchain genutzt wird wurde der Befehl in Listing 2.3 genutzt.

```
rustup toolchain link avr-toolchain /opt/avr-rust
```

Listing 2.3: Konfiguration der Rust-Toolchain

¹<https://github.com/rust-lang/rust/issues/39394>

2.2.5 Installation der AVR-Toolchain

Zum Programmieren des Mikrocontrollers muss eine entsprechende Toolchain vorhanden sein, welche mit dem in Listing 2.4 dargestellten Befehl installiert werden kann.

Das Programm avrdude dient zum Aufspielen eines Programms auf einen Mikrocontroller. Zum Zeitpunkt der Entwicklung war in den Paketquellen avrdude nur in der Version 6.2 enthalten. Das verwendete Entwicklungsboard *ATmega328P Xplained Mini* wird jedoch erst ab avrdude 6.3 unterstützt. Die Installation ist in Listing 2.5 dargestellt.

```
apt-get install avr-libc binutils-avr gcc-avr avrdude
```

Listing 2.4: Installation der AVR-Toolchain

```
1 add-apt-repository ppa:ubuntuhandbook1/apps
2 apt-get update
3 apt-get install avrdude
```

Listing 2.5: Installation avrdude 6.3

2.2.6 Optional: udev-Regel anlegen

Unter Linux sind zur Verwendung des AVR-Programmers standardmäßig Root-Rechte erforderlich. Um die Programmierung auch für normale Benutzer zu erlauben, kann die in Listing 2.6 aufgeführte Zeile in der Datei `/etc/udev/rules.d/99_usbprog.rules` hinzugefügt werden.

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="03eb", ATTRS{idProduct}=="2145",
↪ GROUP="plugdev", MODE="0660"
```

Listing 2.6: udev-Regel

2.3 Hello World AVR-Rust

Ein Beispiel Hello World Programm für AVR-Rust ist auf GitHub verfügbar [2]. Dieses lässt eine LED blinken. Vor der Kompilierung müssen die in Listing 2.7 aufgeführten Umgebungsvariablen gesetzt werden. Anschließend kann der Übersetzungsprozess mit dem in Listing 2.8 dargestelltem Befehl gestartet werden. Die ELF-Datei wird im Unterverzeichnis `target/avr-atmega328p/release/` erstellt. Diese kann dann anschließend mithilfe von `avrdude` auf den Mikrocontroller aufgespielt werden.

```
1  # Needed until https://github.com/japaric/xargo/pull/205 goes
   ↪ through,
2  # to tell it where to find avr-atmega328p.json:
3  export RUST_TARGET_PATH=`pwd`
4
5  # Likely needed if you've just compiled avr-rust from source:
6  export XARGO_RUST_SRC=/opt/avr-rust
```

Listing 2.7: Zur Kompilierung nötige Umgebungsvariablen [2]

```
rustup run avr-toolchain xargo build --target avr-atmega328p
   ↪ --release
```

Listing 2.8: Übersetzungsbefehl [2]

Zum Zeitpunkt der Entwicklung war auf der GitHub-Seite der Hinweis auf die Umgebungsvariable `RUST_TARGET_PATH` noch nicht vorhanden. Daher wurde ein Workaround [4] verwendet: Es wurden die vom Programm referenzierten Repositories *arduino*² und *libcore*³ lokal geklont und die Datei `avr-atmega328p.json` in den Projektordnern hinzugefügt.

²<https://github.com/avr-rust/ruduiino>

³<https://github.com/avr-rust/libcore>

3 Grundlagen von AVR-Rust

In diesem Kapitel werden wichtige Eigenschaften von AVR-Rust untersucht und die Vor-/Nachteile gegenüber C dargelegt. Zu diesen Eigenschaften gehören unter anderem die Verwendung von Registern und Interrupt-Funktionen sowie Rust-Makros und die Rust-Panic-Funktion.

3.1 Registerzugriff

Zur Verwendung der I/O-Funktionen eines Mikrocontrollers ist der Zugriff auf die einzelnen Register des Controllers erforderlich. Unter C gibt es für jedes Register einen Namen (z.B. PORTC), sodass der Registerzugriff nicht über die Speicheradresse des Registers erfolgen muss.

Für AVR-Rust gibt es die Bibliothek *arduino* [5], in welcher entsprechende Konstanten für die Adressen aller Register definiert sind. Somit kann der Registerzugriff ähnlich wie in C erfolgen.

Da sich Registerwerte zu jedem beliebigen Zeitpunkt ändern können, müssen Compiler-optimierungen bei Zugriffen deaktiviert werden. In C gibt es hierfür die Möglichkeit, einzelne Variablen als **volatile** zu markieren. In Rust ist dies nicht möglich, stattdessen gibt es die Funktionen `read_volatile()` und `write_volatile()`. Die Funktionen sind als unsicher markiert, somit ist der Aufruf nur in einem **unsafe** Block erlaubt. Listing 3.1 zeigt einen beispielhaften Registerzugriff.

```
1 let x = unsafe { read_volatile(PINB) }  
2 unsafe { write_volatile(PORTC, 0xFF) }
```

Listing 3.1: Registerzugriff in Rust

Die *arduino*-Bibliothek enthält auch eine Abstraktionsschicht zur Initialisierung der Timer und des UART, sodass entsprechender Code dann einfacher zu verstehen ist (Siehe Listings 3.2 und 3.3).

```

1  unsafe {
2      write_volatile(TCCR0A, WGM01);
3      write_volatile(TCCR0B, CS01 | CS00);
4      write_volatile(OCR0A, OCR0A_VAL);
5      write_volatile(TIMSK0, OCIE0A);
6  }

```

Listing 3.2: Timer-Initialisierung mit direktem Registerzugriff

```

1  timer0::Timer::new()
2      .waveform_generation_mode(
3      timer0::WaveformGenerationMode::ClearOnTimerMatchOutputCompare)
4      .clock_source(timer0::ClockSource::Prescale64)
5      .output_compare_1(Some(OCR0A_VAL))
6      .configure();

```

Listing 3.3: Timer-Initialisierung mit *arduino*-Bibliothek

3.2 Interrupt-Routinen

Mit Interrupt-Routinen können Ereignisse verarbeitet werden, auf die direkt reagiert werden muss. Tritt ein solches Ereignis auf, wird die normale Programmausführung unterbrochen und die entsprechende Interrupt Service Routine (ISR) aufgerufen. Zur Programmierung echtzeitfähiger Anwendungen sind solche Routinen zwingend erforderlich.

In C kann eine ISR anhand eines symbolischen Namens definiert werden. Eine Beispiel-ISR ist in Listing 3.4 dargestellt. Dieser Name wird dann in eine Interrupt-Vektor-Nummer aufgelöst. In AVR-Rust sind diese symbolischen Namen nicht vorhanden, daher muss der Interrupt direkt mit der Vektor-Nummer definiert werden (Beispiel in Listing 3.5). Durch Rust-Makros ist es allerdings möglich, entsprechende symbolische Namen einzuführen.

```

1  ISR(TIMER0_COMPA_vect){
2      Time++;
3  }

```

Listing 3.4: Interrupt Service Routine in C

```

1  #[no_mangle]
2  pub extern "avr-interrupt" fn __vector_14() {
3      unsafe { TIME += 1; }
4  }

```

Listing 3.5: Interrupt Service Routine in Rust

Bei der Verwendung von Interrupts ist zu beachten, dass die ISR im globalen Namensraum liegen muss. Daher müssen in Untermodule definierte Interrupt-Funktionen im Hauptmodul in den globalen Namensraum importiert werden (siehe Listing 3.6). Für in externen Bibliotheken definierte Interrupt-Funktionen ist dies allerdings nicht notwendig.

```

1  use timer;
2  pub use timer::__vector_14;

```

Listing 3.6: Import einer ISR im Hauptmodul

3.3 Verwendung des EEPROM

Der EEPROM dient zur persistenten Speicherung von Daten. Da zur Verwendung lediglich der Zugriff auf bestimmte Register erforderlich ist, kann der EEPROM auch in AVR-Rust problemlos benutzt werden (Siehe Listing 3.7).

```

1  const EEPROM_VAL_ADDR: u16 = 0x0004;
2  eeprom::write_u8(EEPROM_VAL_ADDR, 0x42);
3  eeprom::read_u8(EEPROM_VAL_ADDR);

```

Listing 3.7: Verwendung des EEPROM in Rust

In C gibt es allerdings die Möglichkeit, Variablen mit dem EEMEM-Attribut zu kennzeichnen (siehe Listing 3.8). Dies teilt dem Compiler mit, dass eine Variable im EEPROM abgelegt werden soll. Dies bietet bei der Verwendung des EEPROM folgende Vorteile:

1. Für EEMEM-Variablen müssen keine expliziten Speicheradressen angegeben werden. Die EEPROM-Speicherverwaltung wird vom Compiler übernommen.
2. Einer als EEMEM deklarierten Variable kann ein Standardwert zugewiesen werden, der beim Programmieren des Mikrocontrollers in den EEPROM geschrieben wird.

Da es das EEMEM-Attribut noch nicht in Rust gibt, ist die Verwendung des EEPROM somit in C einfacher als in Rust.

```
1  uint8_t eeVal EEMEM = 0x42;  
2  eeprom_read_byte(&eeVal);
```

Listing 3.8: Verwendung des EEPROM in C

3.4 Makros

3.4.1 Übersicht

Makros sind Befehle, die zur Ergänzung von Programmiersprachen dienen. Die meisten Makros werden wie normale Funktionen verwendet.

In C sind Makros durch einen Präprozessor realisiert, welcher eine simple Textersetzung ausführt. Dadurch weisen C-Makros einige Nachteile auf. Beispielsweise wird in dem C-Makro `#define MUL2(x) 2 * x` der Operatorvorrang nicht berücksichtigt, sodass ein Aufruf von `MUL2(1+1)` einen falschen Wert von 3 ergibt.

In Rust werden Makros direkt vom Compiler unterstützt und besitzen daher nicht die Nachteile von C-Makros. Rust-Makros werden auf syntaktischen Elementen wie Ausdrücken oder Syntaxbäumen definiert und erlauben sichere syntaktische Umformungen [6]. In Listing 3.9 ist ein Makro zum Auslesen eines Registers dargestellt.

```
1  // Definition  
2  macro_rules! reg_read {  
3      ($reg:ident) => (  
4          unsafe{ read_volatile($reg as *mut u8) }  
5      );  
6  }  
7  
8  // Verwendung  
9  let x = reg_read!(PINB);
```

Listing 3.9: Makro in Rust

3.4.2 Fragment Specifier

Die Parameter eines Makros (im vorigem Beispiel `$reg`) heißen **Metavariablen**. Solche Metavariablen besitzen einen **fragment specifier** genannten Typ, der festlegt, welche syntaktischen Strukturen für die Metavariable erlaubt sind. Es sind unter anderem folgende fragment specifier möglich:

- *ident* - Ein gültiger Rust-Bezeichner
- *block* - Ein von geschweiften Klammern umschlossener Block von Anweisungen
- *expr* - Ein Ausdruck
- *tt* - Ein Token Tree (kann nahezu beliebige syntaktische Konstrukte repräsentieren)

Eine vollständige Liste der fragment specifier findet sich in der Rust-Dokumentation [6].

3.4.3 Pattern Matching

Die Metavariablen von Makros werden per Pattern Matching erkannt. Die Muster können aus beliebig vielen fragment specifiern bestehen, aber auch beliebige andere Symbole enthalten. Beispielsweise passt das Muster `E $val:expr` auf den Ausdruck `E 1 + 2`.

Für jedes Makro können mehrere dieser Muster definiert werden, von denen jeweils das erste passende ausgeführt wird. Außerdem können Makros rekursiv definiert werden. Dadurch ist es möglich, eine eigene Syntax für den Inhalt eines Makros zu definieren. Dies wird in folgendem Beispiel genauer erklärt.

```

1  macro_rules! process_expr {
2      ($obj:expr; $func:tt) => {
3          call_func!($obj, $func)
4      };
5      ($obj:expr; $func:tt $($tail:tt)* ) => {
6          process_expr!(call_func!($obj, $func); $($tail)*);
7      };
8  }
9
10 macro_rules! call_func {
11     ($obj:expr, +)          => {$obj.inc()};
12     ($obj:expr, ?)          => {$obj.get()};
13 }

```

Listing 3.10: Rekursives Makro für benutzerdefinierte Syntax

Die in Listing 3.10 definierten Makros dienen zur Generierung einer Kette von Methodenaufrufen. Beispielsweise kann aus dem Aufruf `process_expr!(x; + ?)` der Ausdruck `x.inc().get()` generiert werden.

Das Hilfsmakro `call_func` dient dabei lediglich zur Übersetzung der Symbole in Funktionsbezeichner, beispielsweise wird `call_func!(x, ?)` zu `x.get()` umgeformt.

In den Mustern von `process_expr` muss die erste Metavariable (`$obj:expr`) von den anderen Metavariablen getrennt werden. Da diese vom Typ `expr` ist, muss ein in Ausdrücken verbotenes Trennsymbol verwendet werden. Hier wird das Semikolon verwendet.

Das erste Muster von `process_expr` stellt den nicht-rekursiven Fall dar, bei dem lediglich ein Objekt und ein Funktionssymbol übergeben werden. In diesem Fall wird einfach die dem Symbol entsprechende Funktion aufgerufen.

Das zweite Muster von `process_expr` bekommt ein Objekt, ein Funktionssymbol sowie eine Restliste von Symbolen (`$(tail:tt)*`). Auf dem Objekt wird die entsprechende Funktion aufgerufen und das Resultat dient als neues Objekt für den rekursiven Aufruf des Makros.

3.5 Panic-Funktion

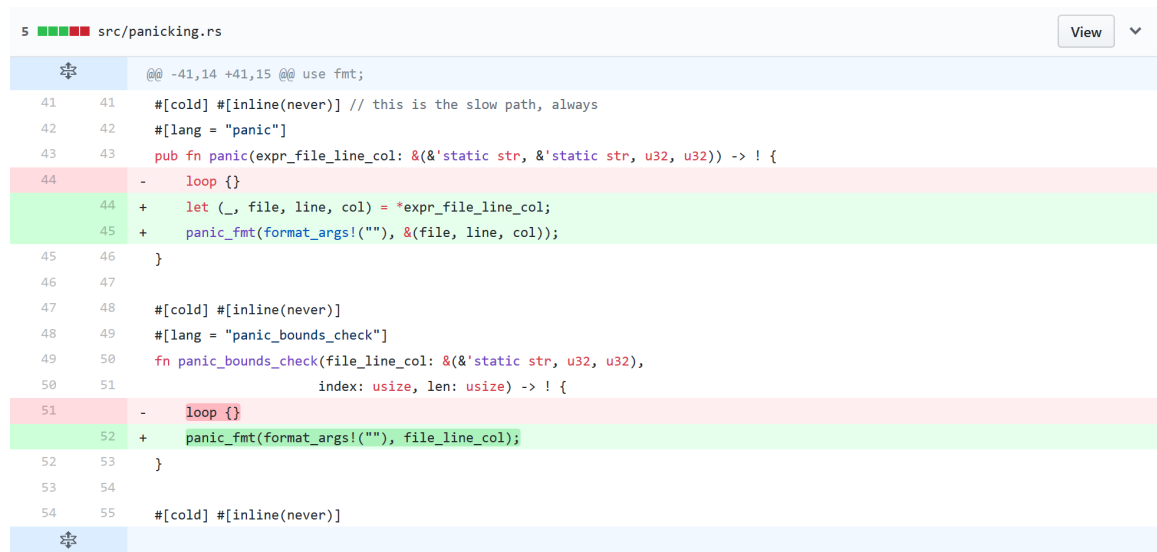
In Rust gibt es eine spezielle Funktion, die *panic*-Funktion, welche bei Laufzeitfehlern aufgerufen wird. Auf PC-Systemen wird ein *panic* z. B. bei Buffer Overflows, Integer Overflows oder fehlgeschlagenen Assertions ausgelöst und führt zum Programmabbruch mit einer entsprechenden Fehlermeldung.

Auch auf AVR-Rust gibt es die *panic*-Funktion, allerdings muss sie hier vom Benutzer definiert werden. Vom Rust-Typsystem wird dabei sichergestellt, dass die Funktion nicht zurückkehrt: Durch den speziellen Rückgabotyp `!` wird erzwungen, dass die Funktion mit einer Endlosschleife endet. Abgesehen davon ist beliebiges Verhalten möglich. Ein Beispiel für eine *panic*-Funktion ist in Listing 3.11 gezeigt.

```
1  #[lang = "panic_fmt"]
2  #[unwind]
3  pub extern fn rust_begin_panic(_msg: (), _file: &'static str, _line:
   ↳ u32, _col: u32) -> !
4  {
5      cli!();
6      uart::put_str_no_isr("ERROR!");
7      restart();
8      loop { } // Wait for restart
9  }
```

Listing 3.11: Panic-Funktion

Bei Tests fiel auf, dass die benutzerdefinierte *panic*-Funktion nie aufgerufen wurde — in der Standardbibliothek wird an der entsprechenden Stelle lediglich eine Endlosschleife ausgeführt. Daher wurde der Code der *libcore* angepasst, um die benutzerdefinierte *panic*-Funktion aufzurufen (Abbildung 3.1). Nach dieser Anpassung wurde bei einem Laufzeitfehler die *panic*-Funktion korrekt aufgerufen, wodurch die Fehlerbehandlung möglich wird.



```

5 src/panicking.rs
@@ -41,14 +41,15 @@ use fmt;
41 41 #[cold] #[inline(never)] // this is the slow path, always
42 42 #[lang = "panic"]
43 43 pub fn panic(expr_file_line_col: &(&'static str, &'static str, u32, u32)) -> ! {
44 - loop {}
44 + let (_, file, line, col) = *expr_file_line_col;
45 + panic_fmt(format_args!("{}", &(file, line, col)));
45 46 }
46 47
47 48 #[cold] #[inline(never)]
48 49 #[lang = "panic_bounds_check"]
49 50 fn panic_bounds_check(file_line_col: &(&'static str, u32, u32),
50 51 index: usize, len: usize) -> ! {
51 - loop {}
52 + panic_fmt(format_args!("{}", file_line_col));
52 53 }
53 54
54 55 #[cold] #[inline(never)]

```

Abbildung 3.1: Aufruf der benutzerdefinierten panic-Funktion in der libcore

3.6 Buffer Overflow

Ein Buffer Overflow ist ein sicherheitskritischer Fehler, welcher auftritt, wenn eine zu speichernde Datenmenge größer ist als der dafür vorgesehene Speicherbereich. Die Grenze des Speicherbereichs wird dabei ignoriert und die Daten in die folgenden Speicherzellen geschrieben. Die darin enthaltenen Daten (z. B. der Programmcode) werden überschrieben. Somit ist bei einem Buffer Overflow beliebiges undefiniertes Verhalten des Programms möglich.

Diese Art Fehler ist sehr häufig in C-Programmen anzutreffen, da die Speicherverwaltung hier manuell vom Benutzer erfolgt. Auch auf Mikrocontrollern ist dies der Fall, etwa wenn eine empfangene Nachricht länger als das dafür vorgesehene Array ist.

Bei Rust kann diese Art von Fehler nicht auftreten, da bei jedem Arrayzugriff eine Indexüberprüfung stattfindet. Schlägt die Prüfung fehl, so wird ein *panic* ausgelöst, wodurch die normale Programmausführung gestoppt wird. In der *panic*-Funktion kann dann eine Fehlerbehandlung stattfinden.

3.7 Integer Overflow

Ein Integer Overflow tritt auf wenn das Ergebnis einer Berechnung nicht im Wertebereich des verwendeten Datentyps liegt. Das Ergebnis ist in einem solchen Fall ein anderer mit dem Datentyp darstellbarer Wert, weswegen so Rechenfehler entstehen können.

Auf PC-Systemen wird bei einem Integer Overflow ein *panic* ausgelöst, wodurch es zu einem Programmabbruch kommt. Um diesen zu verhindern, muss der Entwickler dafür sorgen, dass keine Überläufe auftreten.

Bei AVR-Rust werden Überläufe hingegen ignoriert, wie es auch in C der Fall ist. Dieses Verhalten ist auf Mikrocontrollern durchaus sinnvoll. Denn hier gibt es einige Anwendungen wie Timer, bei denen das Überlaufverhalten erwünscht ist.

3.8 Uninitialisierte Variablen

Die Verwendung von uninitialisierten Variablen ist ein weiterer typischer Fehler in C. Wird auf eine Variable vor der ersten Zuweisung zugegriffen, so besitzt sie einen nicht festgelegten Wert, wodurch undefiniertes Programmverhalten entstehen kann. Da der C-Compiler in solchen Fällen nicht zwingend eine Warnung produziert, können so schwerwiegende Fehler entstehen.

Bei Rust hingegen müssen Variablen zwingend initialisiert werden. Durch eine Datenflussanalyse kann der Compiler zu Übersetzungszeit feststellen, ob es einen Zugriff auf eine nicht initialisierte Variable gibt. Wenn ein solcher Fall festgestellt wird, lässt sich das Programm nicht übersetzen.

An bestimmten Stellen kann es allerdings auch erforderlich sein, auf beliebige (möglicherweise uninitialisierte) Speicheradressen zuzugreifen. Hierzu bietet Rust das Schlüsselwort *unsafe*. In *unsafe*-Blöcken ist der Zugriff auf sogenannte Raw-Pointer möglich, welche auf beliebige Speicheradressen verweisen oder den Wert *null* haben dürfen (siehe Listing 3.12). Mit diesem speziellen Typ ist somit ein Zugriff auf uninitialisierten Speicher möglich. Für alle anderen Typen gilt allerdings auch in *unsafe*-Blöcken, dass sie vor Verwendung initialisiert werden müssen.

```
1  let mem = 0x20 as *const u8;  
2  unsafe {  
3      uart::put_u8(*mem);  
4  }
```

Listing 3.12: unsafe-Block mit Verwendung eines Raw-Pointers

3.9 Performance

Um die Performance von AVR-Rust mit C zu vergleichen, wurde ein Algorithmus in beiden Sprachen implementiert und anschließend getestet. Dabei handelt es sich um einen Algorithmus zur Berechnung der Fibonacci-Zahlen. Um auch die Performance bei Speicherzugriffen zu testen, wurden die Ergebnisse in einen globalen Buffer geschrieben. Ein Auszug der Ergebnisse ist in Tabelle 3.1 dargestellt. Es ist zu erkennen, dass bei den längeren Berechnungen C etwa um den Faktor 1.5 schneller als Rust ist. Dies ist ein relativ gutes Ergebnis, da der C-Compiler über Jahre hinweg optimiert werden konnte. Der AVR-Rust-Compiler hingegen befindet sich noch in einem frühem Stadium und hat derartige Optimierungen noch nicht erfahren.

Iteration	Insgesamt benötigte Zeit (C)	Insgesamt benötigte Zeit (Rust)
1	100 ms	100 ms
2	201 ms	200 ms
...
24	3.2 s	3.6 s
25	3.8 s	4.5 s
26	4.7 s	5.8 s
27	6.0 s	7.9 s
28	8.2 s	11 s
29	11.7 s	16 s
30	17 s	25 s
31	26 s	39 s
32	40 s	61 s
33	63 s	97 s
34	101 s	156 s
35	161 s	250 s

Tabelle 3.1: Ergebnisse der Performance-Messung

Weiterhin wurde die Größe der Binärdateien zweier verhaltensgleicher Programme überprüft (siehe Tabelle 3.2). Es ist ersichtlich dass Rust hier ähnliche Ergebnisse wie C produziert, abhängig von der Optimierungsstufe für die Kompilierung des C-Programms.

Sprache	Größe der Binärdatei
C mit Optimierung auf Größe (-Os)	1516 Byte
Rust	2658 Byte
C mit Optimierung auf Geschwindigkeit (-O3)	2818 Byte

Tabelle 3.2: Größe der erzeugten Binärdateien

4 Beispielprojekt

Um tiefer gehende Eigenschaften von Rust zu testen, wurde ein Beispielprojekt umgesetzt, in dem externe Hardware mit AVR-Rust angesprochen wurde. Das Programm misst die aktuelle Umgebungstemperatur und kann diese an einen Server senden. Zur Messung der Temperatur wurde der Temperatursensor Dallas 18B20 verwendet. Zur TCP-basierten Kommunikation mit einem Server wurde das WiFi-Modul ESP-8266l verwendet.

4.1 Hardware

Die Verkabelung der einzelnen Komponenten ist in Abbildung 4.1 dargestellt.

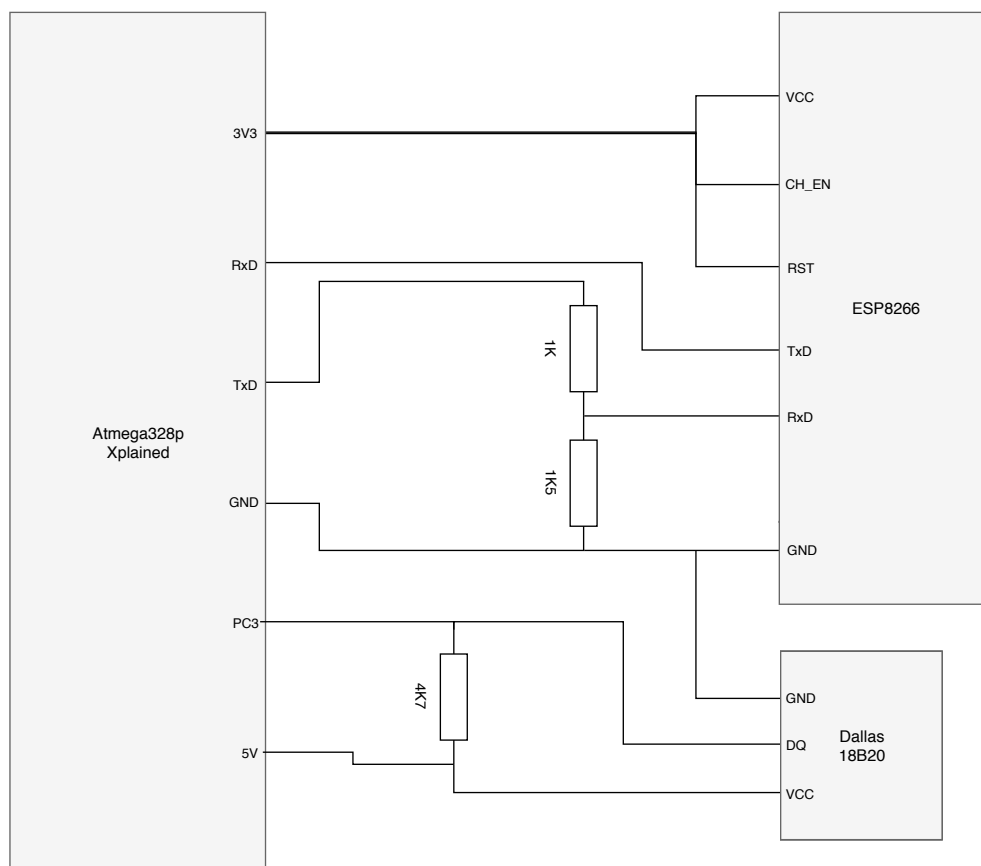


Abbildung 4.1: Schaltplan

4.2 WiFi-Modul

4.2.1 Protokoll des ESP-8266

Der ESP8266 nutzt eine serielle Schnittstelle zur Kommunikation. Als Protokoll werden die sogenannten AT-Kommandos verwendet. Die meisten der unterstützten Befehle sind dabei *extended commands*, welche nicht im Standard-AT-Befehlssatz enthalten sind. Diese Kommandos sind folgendermaßen aufgebaut:

- Ausführen eines Kommandos: AT+<cmd>
- Ausführen eines Kommandos mit Parametern: AT+<cmd>=<value0>,<value1>,...
- Abfrage eines Wertes: AT+<cmd>?

Die Kommandos werden mit CR LF abgeschlossen. In Listing 4.1 ist eine Beispielsitzung mit einigen AT-Kommandos dargestellt. Der vollständige Befehlssatz ist unter [7] beschrieben.

```
1  AT // Test der Kommunikation
2  AT+RST // Reset
3  ATE // Echo einschalten
4  AT+CWMODE? // Betriebsmodus abfragen
5  AT+CWMODE=1 // Betriebsmodus auf "Client" setzen
6  AT+CIPMUX=1 // Mehrere parallele Verbindungen erlauben
7
8  AT+CWLAP // Verfügbare WiFi-Netzwerke auflisten
9  AT+CWJAP="SSID","pw" // Mit Netzwerk verbinden
10 AT+CIPSTART=0,"TCP","192.168.2.3",80 // Baut über Verbindung 0
    ↳ eine TCP-Verbindung zum angegebenen Server auf
11 AT+CIPSEND=0,16 // Sendet 16 Bytes an Verbindung 0
12 Temperatur=19.25 // Daten für den Server (16 Bytes)
13
14 AT+CIPCLOSE=0 // Schließt Verbindung 0
15 AT+CWQAP // Trennt Verbindung vom Netzwerk
```

Listing 4.1: Beispielsitzung mit AT-Kommandos

4.2.2 DSL

Die AT-Kommandos sollen im Quelltext nicht als einfache Zeichenketten hinterlegt werden. Denn dies stellt eine potenzielle Fehlerquelle dar und kann dazu führen, dass fehlerhafte Kommandos gesendet werden. Stattdessen sollen die Befehle mit einer DSL (Domain-specific language), einer anwendungsspezifischen Sprache, zusammengebaut werden. Durch die Verwendung der DSL wird eine Überprüfung der Kommandos zur Übersetzungszeit möglich.

Zur Implementierung der DSL werden Traits genutzt, welche es erlauben für einen Typen eine Menge von Methoden zu definieren. Für jeden Teil der AT-Kommandos (z.B. AT, +, CW, ...) gibt es eine entsprechende Methode (z.B. `at()`, `ext()`, `wifi()`, ...). Jede dieser Methoden besitzt als Rückgabewert ein *Trait Object*. Auf diesem Objekt sind dann weitere Methoden definiert. Auf diese Weise lässt sich kontrollieren, welche Methode bei welchem Objekt aufgerufen werden kann, da immer nur die im jeweiligen Trait definierten Methoden zur Verfügung stehen. In Listing 4.2 ist ein Beispiel-Kommando dargestellt.

```
at().ext().wifi().connection().set().name(b"SSID").pw(b"pw");
```

Listing 4.2: Beispiel AT-Kommando in Rust

In Abbildung 4.2 ist ein Auszug aus der Struktur der DSL dargestellt. Die Funktion `at()` ist die einzige Funktion, welche ohne vorhandenes Trait-Objekt aufgerufen werden kann und dient somit als Einstiegspunkt jedes Kommandos. Der Rückgabewert ist vom Typ `ATBase`, auf welchem dann unter anderem die Methode `ext()` definiert ist. Nach diesem Schema wird dann weiter verfahren, um das komplette Kommando zu bilden.

Die Funktion `set()` gibt ein Trait-Objekt zurück, welches die Parameter für das entsprechende Kommando speichert (im obigen Beispiel `name` und `pw`). Diese Parameter können dann durch Setter-Funktion geändert werden.

Für alle Traits, welche ein valides Kommando repräsentieren, wird die Methode `send()` implementiert, welche das Kommando an den ESP8266 sendet. Diese Methode liefert dann ein Trait-Objekt zurück auf der eine von zwei (optional aufrufbaren) Methoden definiert ist:

1. `get()` - Für alle abfragenden Kommandos implementiert. Wartet bis die Antwort empfangen wurde und gibt den abgefragten Wert zurück.
2. `wait()` - Für alle Befehle implementiert, welche keinen Wert abfragen. Wartet bis die Antwort (Echo) empfangen wurde.

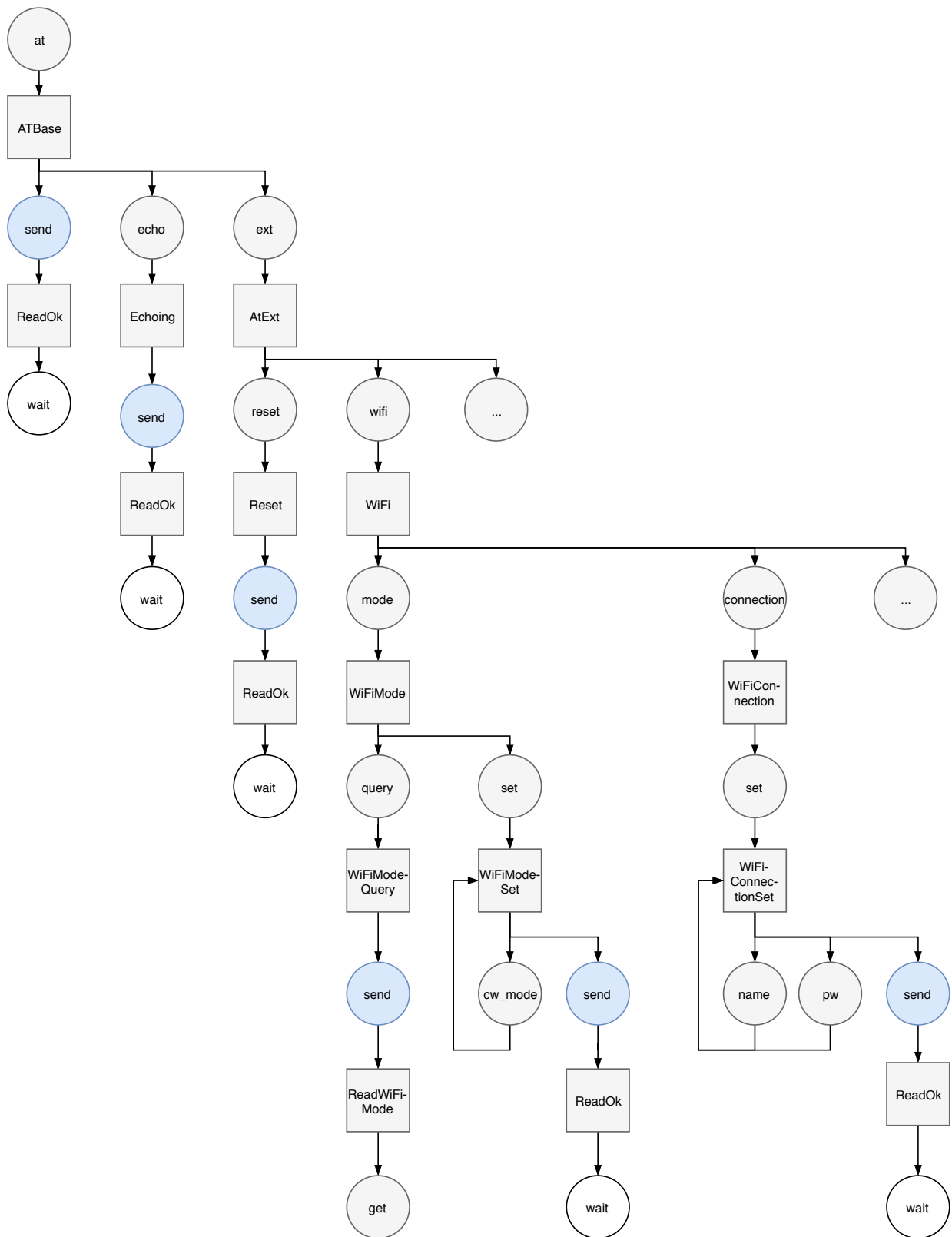


Abbildung 4.2: Struktur der DSL

In Listing 4.3 ist eine Beispielsitzung in Rust dargestellt (diese entspricht dem Beispiel in Listing 4.1).

```
1  at().send().wait();
2  at().ext().reset().send().wait();
3  at().echo(true).send().wait();
4  let mode = at().ext().wifi().mode().query().send().get();
5  at().ext().wifi().mode().set().cw_mode(CWMode::Client).send().wait();
6  at().ext().multi_connections().set().enabled(true).send().wait();
7
8  at().ext().wifi().scan().send().wait();
9  at().ext().wifi().connection().set().name(b"SSID")
↪ .pw(b"pw").send().wait();
10 if let Ok(conn) =
↪ at().ext().tcp().open().set().tcp_handle(TCPHandle::Multi1)
↪ .hostname(b"192.168.2.3").port(80).send().wait()
11 {
12     conn.send_str(b"Temperatur=19.25").wait();
13     conn.close().wait();
14 }
15 at().ext().wifi().disconnect().send().wait();
```

Listing 4.3: Rust-Beispielsitzung mit Verwendung von Methoden

4.2.3 Makro-basierte DSL

Die in Abschnitt 4.2.2 definierte Sprache lässt sich durch die Verwendung von Makros vereinfachen. Wie in Abschnitt 3.4.3 erläutert, kann für die Inhalte eines Rust-Makros nahezu beliebige Syntax definiert werden. Für die DSL wurde ein Makro definiert, dessen Syntax sich an der normalen AT-Kommandosyntax orientiert.

Das grundlegende Prinzip des Makros wurde bereits in Listing 3.10 als Beispiel vorgestellt: Aus einer Folge von Symbolen wird eine Kette von Methodenaufrufen generiert. Für die DSL wurde die Möglichkeit hinzugefügt, Parameter festzulegen. Dabei wird angenommen, dass zunächst beliebig viele parameterlose Funktionen aufgerufen werden und im Anschluss daran beliebig viele Funktionen mit je einem Parameter.

Listing 4.4 zeigt die Parameterverarbeitung. Der Name der Funktion und ihr Parameter werden jeweils durch einen Doppelpunkt getrennt. Die einzelnen Paare aus funktion:parameter werden durch Komma getrennt. Ein Beispielaufruf sieht somit folgendermaßen aus: process_params!(obj, setX:5, setY:6, setZ:7). Dieser Aufruf wird dann zu obj.setX(5).setY(6).setZ(7) umgeformt.

```

1 macro_rules! process_params {
2     ($obj:expr, $func:tt : $val:tt) => {
3         $obj.$func($val)
4     };
5     ($obj:expr, $func:tt : $val:tt, $($tail:tt)*) => {
6         process_params!($obj.$func($val), $($tail)*)
7     };
8 }

```

Listing 4.4: Makro zur Parameterverarbeitung

Als Einstiegspunkt der Makro-basierten DSL dient das Makro `esp_cmd!(...)`. Dieses erzeugt das erste Trait-Objekt mittels `at()` und ruft auf diesem dann die gewünschten Methoden auf. Außerdem können vor das erzeugte Kommando die optionalen Schlüsselwörter `send`, `wait` und `get` geschrieben werden, welche die entsprechenden Methoden aufrufen (siehe Listing 4.5).

```

1 let x = esp_cmd!(AT+CWLAP);
2 x.send();
3 esp_cmd!(send AT+CWLAP);
4 esp_cmd!(send wait AT+CWLAP);
5 esp_cmd!(send AT+CWMODE?);
6 esp_cmd!(send get AT+CWMODE?);

```

Listing 4.5: Einige Beispiele für die Verwendung des DSL-Makros

In Listing 4.6 werden die einzelnen Schritte bei der Expansion des Makros an einem Beispiel gezeigt. Jede Zeile stellt dabei einen Expansionsschritt dar (Codeabschnitte welche sich bei einem Expansionsschritt nicht ändern, sind ausgegraut).

```

1 esp_cmd!(AT+CWJAP = name:b"SSID")
2 process_expr!(at(), +CWJAP = name:b"SSID")
3 process_expr!(call_func!(at(), +), CWJAP = name:b"SSID")
4 process_expr!(at().ext(), CWJAP = name:b"SSID")
5 process_expr!(call_func!(at().ext(), CWJAP), = name:b"SSID")
6 process_expr!(call_func!(call_func!(at().ext(), CW), JAP), =
   ↳ name:b"SSID")
7 process_expr!(call_func!(at().ext().wifi(), JAP), = name:b"SSID")
8 process_expr!(at().ext().wifi().connection(), = name:b"SSID")
9 process_params!(call_func!(at().ext().wifi().connection(), =),
   ↳ name:b"SSID")
10 process_params!(at().ext().wifi().connection().set(), name:b"SSID")
11 at().ext().wifi().connection().set().name(b"SSID")

```

Listing 4.6: Expansion des Makros `esp_cmd`

In Listing 4.7 ist die Beispielsitzung aus Listing 4.3 unter Verwendung des DSL-Makros dargestellt.

```
1  esp_cmd!(send wait AT);
2  esp_cmd!(send wait AT+RST);
3  esp_cmd!(send wait ATE true);
4  esp_cmd!(send wait AT+CWMODE = cw_mode:(CWMODE::Client));
5  let mode = esp_cmd!(send get AT+CWMODE?);
6  esp_cmd!(send wait AT+CIPMUX = enabled:true);
7
8  esp_cmd!(send wait AT+CWLAP);
9  esp_cmd!(send wait AT+CWJAP = name:b"SSID", pw:b"pw");
10
11 return esp_cmd!(send wait AT+CIPSTART =
    ↪ tcp_handle:(TCPHandle::Multi1), hostname: b"192.168.2.3",
    ↪ port:80);
12
13 if let Ok(conn) = esp_cmd!(send wait AT+CIPSTART =
    ↪ tcp_handle:(TCPHandle::Multi1), hostname: b"192.168.2.3", port:80)
14 {
15     conn.send_str(b"Temperatur=19.25").wait();
16     conn.close().wait();
17 }
18 esp_cmd!(send wait AT+CWQAP);
```

Listing 4.7: Rust-Beispielsitzung mit Verwendung von Makros

4.3 Temperatursensor

4.3.1 Protokoll des Dallas 18B20

Der DS18B20 nutzt ein einfaches 1-Wire-Protokoll zur Kommunikation [8]. Dieses arbeitet folgendermaßen (Der Bus ist standardmäßig high):

Reset:

- Setze den Bus für mindestens $480\mu s$ auf low um einen Reset auszulösen
- Warte bis der Presence-Detect abgeschlossen ist (Der Sensor setzt den Bus für 60 bis $240\mu s$ auf low)

Daten bitweise schreiben (LSB zuerst):

- Bit '1': Setze den Bus für 1 bis $15\mu s$ auf low
- Bit '0': Setze den Bus für $60\mu s$ auf low

Daten bitweise lesen (LSB zuerst):

- Setze den Bus für 1 bis 15 μ s auf low
- Prüfe Bus innerhalb von 60 μ s:
 - Bus low: Bit '0'
 - Bus high: Bit '1'

In Tabelle 4.1 sind die Befehle des Temperatursensors dargestellt, welche zum Messen der Temperatur benutzt werden.

Befehl	Beschreibung
Skip Rom 0xCCh	Überspringt die Geräteadressierung
Convert T 0x44h	Misst die Temperatur und speichert diese im Scratchpad
Read Scratchpad 0xBEh	Liest das Scratchpad aus

Tabelle 4.1: Verwendete Befehle des DS18B20

Es ist zu beachten, dass zwischen den Befehlen regelmäßig ein Reset nach folgendem Schema erfolgen muss:

1. Reset
2. Adressierungsbefehl (Skip Rom)
3. Befehl (Convert bzw. Read Scratchpad)
4. Antwort lesen

Zum Auslesen des Scratchpad nach dem Convert-Befehl muss also zunächst ein Reset und eine Adressierung erfolgen (Die Inhalte des Scratchpad bleiben bei einem Reset erhalten).

4.3.2 Implementierung des Protokolls

Die Kontrolle des Eindrahtbus erfolgt durch eine ISR, welche von einem Timer kontrolliert wird. Der Timer wird nach jedem ISR-Aufruf angepasst, um die unterschiedlichen Wartezeiten zu realisieren. Die Methode basiert dabei auf einem Zustandsautomaten, je nach Zustand der globalen Variable STATE wird der entsprechende Codeabschnitt ausgeführt.

In Listing 4.8 ist als Beispiel die Routine zum Lesen des Bus dargestellt. Beim 1. Aufruf der ISR (Zustand: State::BeginCycle) wird der Bus auf low gesetzt und der Timer so konfiguriert, dass er nach 5 μ s erneut aufgerufen wird. Beim 2. Aufruf wird der Bus wieder auf high gesetzt. Beim 3. Aufruf (also nach 10 μ s) wird der Bus gelesen und das entsprechende Bit gespeichert. Außerdem wird der Timer auf eine Wartezeit von 60 μ s konfiguriert, da nach dieser Zeit der Schreibvorgang des Sensors auf jeden Fall abgeschlossen ist. Beim 4. Aufruf wird dann, abhängig davon, ob noch weitere Bits empfangen werden, zurück in den Initialzustand oder in den Endzustand gewechselt

```

1  fn isr_read() {
2      match unsafe_read!(STATE) {
3          State::BeginCycle => {
4              bus_low();
5              unsafe_write!(STATE = State::After5us);
6              configure_timer(TIMER_CONF_5_US);
7          }
8          State::After5us => {
9              bus_high();
10             unsafe_write!(STATE = State::After10us);
11         }
12         State::After10us => {
13             unsafe_write!(BUFFER = bus_read() << 7 | BUFFER >> 1 );
14             unsafe_write!(STATE = State::After70us);
15             configure_timer(TIMER_CONF_60_US);
16         }
17         _ => { // State::After70us
18             unsafe_write!(BIT_CNT = BIT_CNT + 1);
19             if unsafe_read!(BIT_CNT) < 8 {
20                 unsafe_write!(STATE = State::BeginCycle);
21                 configure_timer(TIMER_CONF_10_US);
22             }
23             else {
24                 unsafe_write!(STATE = State::Done);
25                 disable_timer2();
26             }
27         }
28     }
29 }

```

Listing 4.8: ISR zum Lesen des 1-Wire-Bus

4.4 Probleme bei der Entwicklung

4.4.1 Fehlerhafter Code

Bei der Entwicklung fiel auf, dass ein bestimmter Codeabschnitt (Listing 4.9) nicht korrekt übersetzt wurde. Die Variable `y` hat nach Ausführung nur den Wert von `tmp_1`, die Variable `tmp_2` wird verworfen.

Wenn die Variablen `tmp_1` und `tmp_2` mit `read_volatile` gelesen werden, wird der Codeabschnitt korrekt übersetzt. Daher wird der Fehler wahrscheinlich durch Compiler-optimierungen erzeugt.

Dies ist ein schwerwiegender Fehler, da das Verhalten des kompilierten Programms vom Quellcode abweicht. Derartige Fehler sind extrem schwer zu finden.

```

1  let tmp_1: u8 = x << 1;
2  let tmp_2: u8 = x >> 7;
3  let y: u8 = tmp_1 | tmp_2;

```

Listing 4.9: Nicht korrekt übersetzter Codeabschnitt

4.4.2 Nicht kompilierbares Projekt

In einer späten Phase der Entwicklung trat ein Fehler bei der Kompilierung auf (Listing 4.10). Der Quellcode wies keine Fehler auf, es handelt sich dabei um einen internen Fehler im Compiler. Durch Umschreiben des Programms wurde der Fehler zunächst behoben, bei der weiteren Entwicklung trat jedoch ein ähnlicher Fehler auf (Listing 4.11).

Bei einer Analyse des Quelltextes konnte kein bestimmter Codeabschnitt identifiziert werden, der dieses Verhalten verursacht. Änderungen an unterschiedlichen Codeabschnitten im Projekt hatten Einfluss auf den Fehler. Teilweise wurde der Fehler auch durch das Auskommentieren bestimmter Codezeilen provoziert.

Dies ist ebenfalls ein schwerwiegender Fehler, da er die Entwicklung behindern oder sogar verhindern kann. Der Entwickler erwartet, dass ein fehlerfreies Programm vom Compiler übersetzt werden kann.

```

1 rustc: rust/src/llvm/lib/CodeGen/RegAllocBase.cpp:148: void
  ↳ llvm::RegAllocBase::allocatePhysRegs(): Assertion
  ↳ `!SplitVirtReg->empty() && "expecting non-empty interval"' failed.

```

Listing 4.10: Fehlermeldung des Compilers (1)

```

1 rustc: rust/src/llvm/lib/CodeGen/MachineBasicBlock.cpp:1299:
  ↳ llvm::MachineBasicBlock::livein_iterator
  ↳ llvm::MachineBasicBlock::livein_begin() const: Assertion
  ↳ `getParent()->getProperties().hasProperty(
  ↳ MachineFunctionProperties::Property::TracksLiveness) &&
  ↳ "Liveness information is accurate"' failed.

```

Listing 4.11: Fehlermeldung des Compilers (2)

Der Fehler in Listing 4.11 ist bereits bekannt und wurde in der aktuellen Version von AVR-Rust bereits behoben¹.

¹ <https://github.com/avr-rust/rust/issues/99>

5 Fazit

In diesem Projekt wurde die Tauglichkeit von Rust auf AVR-Mikrocontrollern untersucht.

Als erste Aufgabe musste ein entsprechender Compiler eingerichtet werden. Die einzelnen Schritte für die Installation wurden in Kapitel 2 beschrieben. Dabei wurde auch auf mögliche Probleme bei der Installation eingegangen.

Im Anschluss wurden verschiedene Eigenschaften von AVR-Rust analysiert und mit C verglichen. Alle für die Programmierung von Mikrocontrollern notwendigen Konstrukte wie Registerzugriff und ISRs wurden auch von Rust unterstützt. Beim Vergleich zeigten sich einige Vorteile von Rust gegenüber C, wie die automatische Indexprüfung zur Laufzeit und die Prüfung auf uninitialisierte Variablen zur Übersetzungszeit.

In einem Beispielprojekt wurden tiefergehende Eigenschaften von AVR-Rust untersucht. In dem Beispielprojekt wurde die Temperatur mithilfe eines Sensors gemessen und anschließend unter Verwendung eines WiFi-Moduls an einen Server gesendet.

Für die Kommunikation mit dem WiFi-Modul über AT-Kommandos wurde eine DSL entwickelt, durch welche sich die Kommandos flexibel zusammensetzen lassen. Fehlerhafte Kommandos werden schon zur Übersetzungszeit erkannt, wodurch garantiert wird, dass nur valide Kommandos gesendet werden. Durch die Verwendung von Makros wurde eine sichere, kompakte Notation der Kommandos ermöglicht, welche sich an den eigentlichen AT-Kommandos orientiert.

Für die Kommunikation mit dem Temperatursensor war eine Steuerung der Datenleitung im Mikrosekundenbereich notwendig, welche durch Interrupts implementiert wurde. Es traten hierbei keine Performance-Probleme auf. Somit kann AVR-Rust grundsätzlich auch für zeitkritische Anwendungen eingesetzt werden.

Zum Ende der Entwicklungsarbeit wurden einige Fehler in der eingesetzten Compiler-Version festgestellt. Einer der Fehler führte dazu, dass ein fehlerfreies Projekt nicht übersetzt werden konnte, wodurch die weitere Arbeit stark behindert wurde. Somit ist der produktive Einsatz von AVR-Rust noch nicht zu empfehlen.

Da der AVR-Rust-Compiler allerdings kontinuierlich weiterentwickelt wird, ist davon auszugehen, dass solche Fehler behoben werden und der Compiler stabiler wird. Wenn dies der Fall ist, kann AVR-Rust produktiv eingesetzt werden, um robuste Programme auf Mikrocontrollern zu ermöglichen.

6 Quellcode

Der Quellcode des Projekts ist auf folgender GitHub-Seite zu finden:

<https://github.com/GitNiklas/Embedded-Rust>.

Das in Kapitel 4 vorgestellte Beispielprojekt befindet sich im Unterverzeichnis [avr-periphery](#).

Literaturverzeichnis

- [1] GitHub: AVR-Rust-Compiler. <https://github.com/avr-rust/rust>. Abgerufen am 03.08.2018.
- [2] GitHub: Hello World AVR-Rust-Programm. <https://github.com/avr-rust/blink>. Abgerufen am 05.08.2018.
- [3] Rust Installation. <https://www.rust-lang.org/en-US/install.html>. Abgerufen am 28.03.2018.
- [4] Foreneintrag: AVR support not working in avr-rust. <https://users.rust-lang.org/t/solved-avr-support-not-working-in-avr-rust-xargo/14869/6>. Abgerufen am 28.03.2018.
- [5] GitHub: Rust-Arduino-Library. <https://github.com/avr-rust/ruduino>. Abgerufen am 05.08.2018.
- [6] Rust-Dokumentation: Makros. <https://doc.rust-lang.org/book/first-edition/macros.html>. Abgerufen am 14.04.2018.
- [7] Espressif Inc. [ESP8266 AT Instruction Set](#), 2017.
- [8] Maxim Integrated. [DS18B20 Datenblatt](#), 2015.
- [9] GitHub: Rust-Standarbibliothek für AVR. <https://github.com/avr-rust/libcore>. Abgerufen am 05.08.2018.
- [10] The Rust Project Developers. [The Rust Programming Language](#), 2011.
- [11] Tutorial: AVR auf Ubuntu. <https://wiki.ubuntuusers.de/AVR/>. Abgerufen am 28.03.2018.