Ausarbeitung

Embedded Rust

4. August 2018

Eingereicht von:
Niklas Kühl
inf102861@fh-wedel.de

Betreuer:
Prof. Dr. Ulrich Hoffmann
uh@fh-wedel.de

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **2**
2.1 Rust . . . 2
2.2 C vs Rust . . . 2
2.3 Mikrocontroller-Programmierung . . . 3

3 Einrichtung der Entwicklungsumgebung **4**
3.1 Systemkonfiguration . . . 4
3.2 Installation des AVR-Rust-Compilers . . . 5
3.2.1 Grundlegendes Vorgehen . . . 5
3.2.2 Übersetzungsvorgang (make) . . . 5
3.2.3 Installationsvorgang (make install) . . . 6
3.2.4 Rust-Toolchain konfigurieren (rustup toolchain) . . . 6
3.2.5 Installation der AVR-Toolchain . . . 6
3.2.6 Optional: udev-Regel anlegen . . . 7
3.3 Hello World AVR-Rust . . . 7
3.4 Entwicklungsumgebungen . . . 7

4 Entwicklung **8**

5 Ergebnisse **11**

6 Fazit **12**

7 Notizen **13**
7.1 DSL . . . 13

Literaturverzeichnis **14**

Quellcodeverzeichnis

3.1 Kompilierung und Installation des AVR-Rust-Compilers [1] 5
3.2 Installation avrdude 6.3 7

4.1 Test Rust Listing 9
4.2 Und das gleiche mit Makros 9
4.3 Test Rust Listing 2 10

1 Einleitung

2 Grundlagen

2.1 Rust

- Programmiersprache - Sichere Alternative zu C - Dies soll v.a. erreicht werden durch ...

- Makros in Rust - Makros können zur Erweiterung des Sprachumfangs genutzt werden. Problematische Eigenschaften wie sie C-Makros besitzten, werden allerdings vermieden - Makros sind im Compiler integriert Es gibt also keinen Präprozessor wie bei C - Da Rust-Makros nicht wie in C auf Textersetzung basieren, können daraus resultierende Fehler nicht auftreten: - Falsche Ergebnisse durch Operator-vorrang - Macro Hygine: Innerhalb von Makros definierte Variablen können keine anderen Variablen verstecken, wenn beide den gleichen Bezeichner haben - Makros stellen wenig anforderungen an das, was in ihnen stehen kann - Es können auch Symbole definiert werde - Es lässt sich relativ frei eine eigene Syntax spezifizieren - Parameter = metavariable - Type = fragment specifier - Da Rust Makros zu einem frühen Zeitpunkt der Kompilierung aufgelöst werden, können keine Rust-Typen wie u8 verwendet werden. Stattdessen besitzen die Parameter eines Makros (=metavariable) Typen (=fragment specifier) wie - expr (Beliebiger Rust-Ausruck, z.B. 2+2, func(x)) - token tree

- Was ist Rust - Vor/Nachteile - Vorteile - Sichere autom. Speicherverwaltung (ohne GC) - Eingebauter Index-Check - Kein Zugriff auf nicht initialisierten Speicher möglich - Nachteile - Für Neulinge erstmal schwer zu verstehen??

- Anwendungsbereiche - Desktop Entwicklung (Firefox) - Systemprogrammierung

2.2 C vs Rust

- Buffer Overflow - Z.B. Es wird Protokoll zelenweise übertargen -> Zeile ist zu lang und passt nicht in Buffer C: Es passiert irgendwas Rust: In der offiziellen AVR-LibCore wird dann eine Eindlosschleife ausgelöst -> ein wenig besser Wenn die libcore angepasst wird, ist auch beliebiges Verhalten möglich, zb. Fehlemeldung oder Reset

- Rust: gwöhnungsbedürftig wg. borrowing-Konzept

2.3 Mikrocontroller-Programmierung

- Standardmäßig in C - Daher auch Fehler von C enthalten - Rust kann hier Vorteile bringen (wo? in welchen Bereichen?) -> Array-Index-Check -> Zugriff auf unitialisierten Speicher

Evtl die Dreiteilung Analyse-Design-Implementierung??

3 Einrichtung der Entwicklungsumgebung

3.1 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 3.1: 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 3.2: Verwendete Hardware

3.2 Installation des AVR-Rust-Compilers

3.2.1 Grundlegendes Vorgehen

Um die Entwicklungsumgebung einzurichten, sollte eine normale Rust-Installation vorhanden sein. Zur Installation siehe www.rust-lang.org/en-US/install.html. Außerdem sollte das alternative Rust-Buildsystem xargo installiert werden: `cargo install xargo`.

Für den Rust-Compiler für AVR-Mikrocontroller gibt es keine vorkompilierten Binärdateien. Daher muss er für des verwendete System neu übersetzt werden. Das grundlegende Vorgehen ist auf der entsprechenden GitHub-Seite beschrieben (github.com/avr-rust/rust). In den folgenden Kapiteln wird auf Eigenheiten und Probleme bei der Installation eingegangen.

```
# Grab the avr-rust sources
git clone https://github.com/avr-rust/rust.git

# Create a directory to place built files in
mkdir build && cd build

# Generate Makefile using settings suitable for an experimental compiler
../rust/configure \
--enable-debug \
--disable-docs \
--enable-llvm-assertions \
--enable-debug-assertions \
--enable-optimize \
--prefix=/opt/avr-rust

# Build the compiler, optionally install it to /opt/avr-rust
make
make install

# Register the toolchain with rustup
rustup toolchain link avr-toolchain $(realpath $(find . -name 'stage1'))

# Optionally enable the avr toolchain globally
rustup default avr-toolchain
```

Listing 3.1: Kompilierung und Installation des AVR-Rust-Compilers [1]

3.2.2 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 und auch mit einem Fehler abbrechen. 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 Fehler[1].

- 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.

3.2.3 Installationsvorgang (make install)

Der Befehl `make install` installiert den 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:

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

Dadurch konnte die Installation erfolgreich ausgeführt werden konnte. Im Anschluss wurden die Rechte des Ordners dann wieder eingeschränkt.

3.2.4 Rust-Toolchain konfigurieren (rustup toolchain)

Damit die in `/opt/avr-rust` installierte AVR-Toolchain genutzt wird, sollte folgender Befehl genutzt werden:

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

3.2.5 Installation der AVR-Toolchain

Zum Programmieren des Mikrocontrollers muss eine entsprechende Toolchain vorhanden sein, welche mit den folgendem Befehl installiert werden kann:

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

[1] https://github.com/rust-lang/rust/issues/39394

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, welcher mit folgenden Befehlen installiert werden kann:

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

Listing 3.2: Installation avrdude 6.3

3.2.6 Optional: udev-Regel anlegen

Da unter Linux zum Verwenden des Programmers standardmäßig Root-Rechte erforderlich sind, kann folgende Zeile in der Datei `/etc/udev/rules.d/99_usbprog.rules` hinzugefügt werden, um die Programmierung auch für normale Benutzer zu erlauben:

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

3.3 Hello World AVR-Rust

3.4 Entwicklungsumgebungen

- Diskussion IDEs - Eclipse mit RustDT - VS Code

4 Entwicklung

- Erst kleinere Testprogramme - Dann größeres Programm mit Peripherie-Ansteuerung - DSL mit Rust - Method Chaining - Makro DSL - Hello World AVR-Rust - Tests Rust vs C (AVR)

- Test des avr-periphery Programms erfolgte mit einem durch netcat beireigestellten Server, dadurch konnten die durch den µController gesendeten nachrichten auf einem PC angezeigt werden

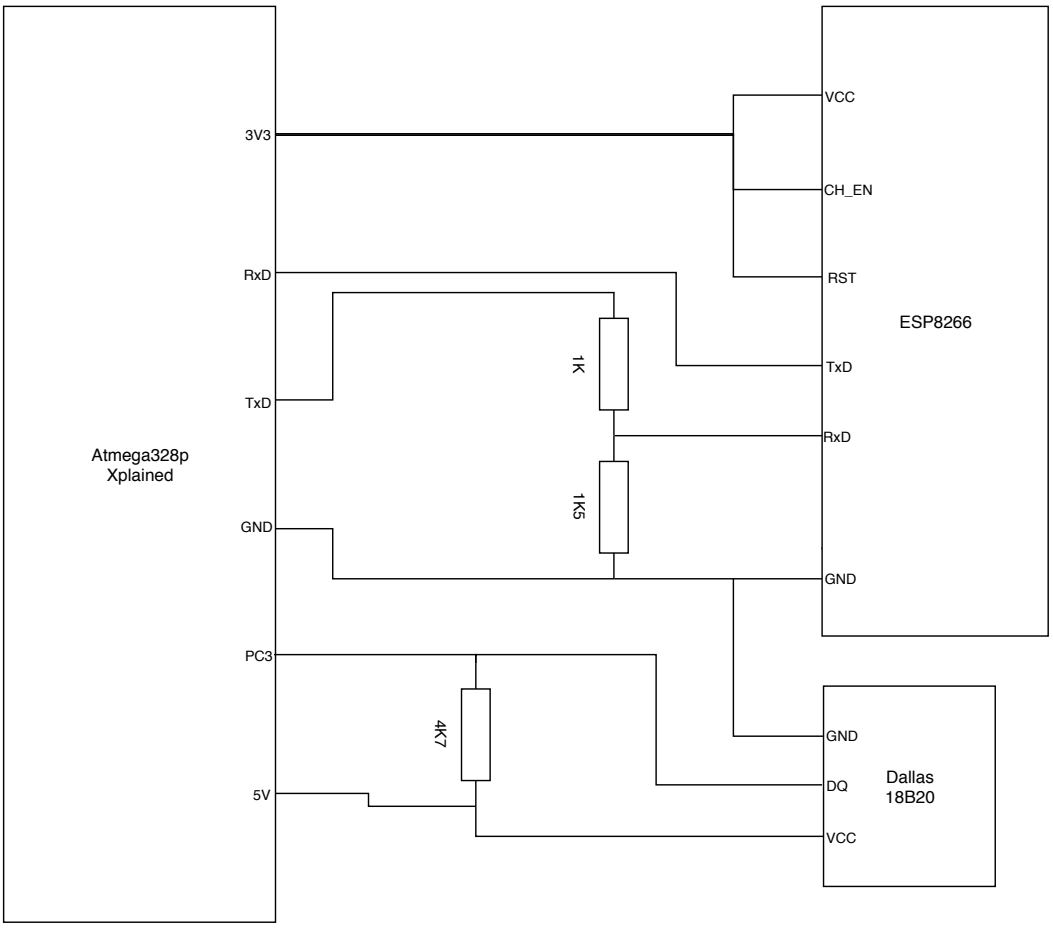

Abbildung 4.1: Schaltplan

```rust
  fn open_server_connection() -> Result<TCPConnection, ()> {
    esp8266::at().send().wait();
    esp8266::at().ext().reset().send().wait();

    esp8266::at().ext().wifi().mode().set().cw_mode(CWMode::Client)
    .send().wait();
    esp8266::at().ext().multi_connections().set().enabled(true)
    .send().wait();
    esp8266::at().ext().wifi().scan().send().wait();

    let connect_wifi =
    esp8266::at().ext().wifi().connection().set().name(WIFI_NAME)
    .pw(WIFI_PW).send().wait();
    if let Ok(_) = connect_wifi {
      return esp8266::at().ext().tcp().open().set()
      .tcp_handle(TCPHandle::Multi1)
    .hostname(HOST).port(PORT).send().wait();
    }
    else {
      return Err(());
    }
  }
```

Listing 4.1: Test Rust Listing

```rust
fn open_server_connection() -> Result<TCPConnection, ()> {
  esp_cmd!(send wait AT)?;
  esp_cmd!(send wait AT+RST)?;

  esp_cmd!(send wait ATE true)?;
  esp_cmd!(send wait AT+CWMODE = cw_mode:(CWMode::Client))?;
  //let mode = esp_cmd!(send get AT+CWMODE?)?;
  esp_cmd!(send wait AT+CIPMUX = enabled: true)?;
  esp_cmd!(send wait AT+CWLAP)?;

  esp_cmd!(send wait AT+CWJAP = name:WIFI_NAME, pw:WIFI_PW)?;
  return esp_cmd!(send wait AT+CIPSTART =
    tcp_handle:(TCPHandle::Multi1), hostname: HOST, port:PORT);
}
```

Listing 4.2: Und das gleiche mit Makros

```rust
#[no_mangle]
fn send_temperature() { // Send Temperature to server
  if let Ok(conn) = open_server_connection() {
    for _i : u8 in 0.. 10 {
      conn.send_str(b"Temperature: ").wait();
      let temp = ds18b20::read_temperature();
      let temp_str = ds18b20::temperature_to_str(&temp);
      conn.send_str(&temp_str).wait();
      conn.send_str(b"\r\n").wait();
      delay_ms(1000);
    }
    close_server_connection(conn);
  }
}
```

Listing 4.3: Test Rust Listing 2

5 Ergebnisse

- Erkenntnisse aus den Tests (Vor/Nachteile zu C) - Probleme bei der Entwicklung

6 Fazit

- Im allgemeinen eignet sich rust zur systmprogrammierung. Durch A, B und C ergeben sich Vorteile gegenüber der Verwendung von C. Dadurch können robustere Programme entwickelt werden.

Die eingesetzte Compiler Version hatte jedoch einige Schwächen

Somit wird vom EInsatz von Rust auf AVR vorerst abgeraten. Wenn der Compiler stabil ist, kann Rust allerdings gut verwendet werden.

7 Notizen

- Sind Maybe-Typen von Rust effizient auf AVR? - Sind ISR's re-entrant bei Rust => Was passiert, wenn bei ISR-Ausführung ein weiterer Interrupt auftritt - Was gibt es für std-libraries für avr rust - unsafe Rust - Was ist daran unsicher - Echte Pointer möglich, die NULL sein dürfen => Spezieller Zeigertyp, die STd-Refs von Rust dürfen auch in unsafe Code nie null sein - Mit diesen Zeigertypen ist auch der Zugriff auf beliebige SPeicher-adressen möglich - wie kann man die unsafe-sectionen auf ein min reduzieren? => Durch Funktionen und Makros lassen sich unsafe-zugriffe kapseln - z.B. auch erzeugen einer vermeintlich sicheren Referenz auf globales Obj => trotz normaler Referenz nicht thread-safe

- Performance - Test: Fibonacci Berechnung inkl Speicherzugriffe Ergebnis: C etwa um Faktor 1.5 schneller, abhängig vom Optimierungslevel - Code größe 1 C mit opt-size 1500By 2 Rust 2600 3 C mit opt-speed 2800 - RAM Verbrauch C: Bei 120 Calls Stack overflow; 120*16 = 1920Bytes von 2048 Rust Bei 100 Calls 100*16 = 1600Bytes - ALLERDINGS: C COmpiler wurde über Jahre hinweg Optimiert, daher sind diese Zahlen schon ganz gut

- Rust Out of Bounds - panic wird ausgelöst - Nutzer muss panic Funktion definieren, welche feslegt was dann passieren soll - Die Funktion darf nicht zurückkehren; dies wird durch Rust zur Übersetzungszeit festgelegt - ABER: In der avr-libcore wird die benutzer-definierte Funktion nicht aufgerufen; stattdessen wird Endlossschleife ausgelöst - Wenn libcore angepasst wird (WO?) ist bei OOB beliebiges Verhalten wie zB Fehlermeldung oder Neustart möglich

- Vorteile durch Typsystem - Erkennung unf Verhinderung von uninitialisierten Variablen - Verwechselung von Variablen werden leichter erkannt (da Zwischen u8 und usize unterschieden wird) - Fehler bei Zuweisung unterschiedlicher Typen - Keine implizite Typumwandlung

- Was fehlt AVR-Rust - Stabiler Compiler - Std-Library - Delay-Funktion - ISR-Definitionen (wie in c die interrupt.h)

7.1 DSL

- Verbieten, dass andere Kommandos als die unterstützten zusammengesetzt werden - Fehler werden schon zur Übersetzungszeit erkannt => Wie sehen die Fehlermedlungen aus? (Makro-DSL) Wenn man sich beim Makro verschrieben hat, kommt dann eine aussagekräftige Fehlermeldung??? => Stichwort: Fluent Interface

Literaturverzeichnis

[1] Github-Seite des AVR-Rust-Projekts. https://github.com/avr-rust/rust. Abgerufen am 03.08.2018.

[2] Rust-Dokumentation. https://doc.rust-lang.org/. Abgerufen am 03.08.2018.

[3] Tutorial: AVR auf Ubuntu. https://wiki.ubuntuusers.de/AVR/. Abgerufen am 03.08.2018.