UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO, DE CIÊNCIAS EXATAS E EDUCAÇÃO
DEPARTAMENTO DE ENG. DE CONTROLE, AUTOMAÇÃO E COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Marcos Alexandre Muraro Grando

Estudo de Caso Sobre Uso de Rust em Sistemas Embarcados: desenvolvimento de um device driver do sensor BMP180 para microcontroladores AVR

Blumenau
2024

Marcos Alexandre Muraro Grando

Estudo de Caso Sobre Uso de Rust em Sistemas Embarcados: desenvolvimento de um device driver do sensor BMP180 para microcontroladores AVR

Trabalho de Conclusão de Curso de Graduação em Engenharia de Controle e Automação do Centro Tecnológico, de Ciências Exatas e Educação da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Engenheiro de Controle e Automação.
Orientador: Prof. Dr. Carlos Roberto Moratelli

Blumenau
2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Muraro Grando, Marcos Alexandre
Estudo de Caso Sobre Uso de Rust em Sistemas Embarcados : desenvolvimento de um device driver do sensor BMP180 para microcontroladores AVR / Marcos Alexandre Muraro Grando ; orientador, Carlos Roberto Moratelli, 2024.
56 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Blumenau, Graduação em Engenharia de Controle e Automação, Blumenau, 2024.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Sistemas Embarcados. 3. Rust. 4. Device Driver. 5. Arduino. I. Moratelli, Carlos Roberto. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Marcos Alexandre Muraro Grando

Estudo de Caso Sobre Uso de Rust em Sistemas Embarcados: desenvolvimento de um device driver do sensor BMP180 para microcontroladores AVR

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Engenheiro de Controle e Automação" e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação.

Blumenau, 03 de julho de 2024.

Banca Examinadora:

Documento assinado digitalmente
Carlos Roberto Moratelli
Data: 11/07/2024 12:57:48-0300
CPF: ***.999.369-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Carlos Roberto Moratelli
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Fabio Rafael Segundo
Data: 11/07/2024 15:20:24-0300
CPF: ***.593.909-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Fabio Rafael Segundo
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Guilherme Brasil Pintarelli
Data: 11/07/2024 14:30:41-0300
CPF: ***.523.129-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Guilherme Pintarelli
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Gostaria de agradecer, primeiramente, aos meus pais que, além de muito mais, custearam minha educação e estadia fora de minha cidade natal, e à minha irmã que me auxiliou a aqui me estabelecer, possibilitando-me cursar este ensino superior. Gostaria de agradecer, também, e especialissimamente, ao Prof. Dr. Carlos Roberto Moratelli por aceitar orientar-me e ajudar-me grandemente na execução deste documento final da minha graduação, bem como ao excelente corpo docente da UFSC do *campus* Blumenau por transmitirem seus conhecimentos a mim ao longo deste trajetória.

RESUMO

Este trabalho desenvolve um *device driver* do sensor BMP180 em Rust para microcontroladores AVR, neste caso um kit de desenvolvimento Arduino Mega 2560. Não havendo um *device driver* já existente, usa-se noções do análogo em linguagem C para fins de desenvolvimento e comparação. Utiliza-se critérios de comparação qualitativos, como qualidade do compilador, e quantitavos, como tamanho binário dd *firmware*, tempo de compilação e tempo de execução. Para a escrita do *driver* em Rust e sua utilização em placas AVR, é necessária uma camada de abstração de hardware específica para isso, havendo apenas uma opção pública que contém implementações do protocolo I2C, chamada AVR HAL. Foi concluído que o Rust pode ser capaz de entregar tamanhos binários significativamente menores a custo de uso de memória RAM. Para futuros trabalhos, recomendar-se-ia realizar comparações similares com softwares mais complexos, que já sejam conhecidos em C e que tenham notáveis problemas de segurança, com a finalidade de saber se o Rust pode entregar os mesmos benefícios aqui descritos, ao mesmo tempo que aumenta a segurança do código.

Palavras-chave: Rust, Device Driver, Arduino, Sistemas Embarcados

ABSTRACT

This work develops a device driver for the BMP180 sensor in Rust for AVR microcontrollers, in this case, an Arduino Mega 2560 development kit. In the absence of an existing device driver, notions from the analogous driver in the C language are used for development and comparison purposes. Both qualitative comparison criteria, such as compiler quality, and quantitative criteria, such as binary firmware size, compilation time, and execution time, are used. For writing the driver in Rust and its use on AVR boards, a specific hardware abstraction layer is required, with only one public option available which contains I2C protocol implementataions, called AVR HAL. It was concluded that Rust can deliver significantly smaller binary sizes at the cost of RAM usage. For future work, it is recommended to perform similar comparisons with more complex software that is already known in C and has notable security issues, with the aim of determining whether Rust can provide the same benefits described here while increasing code security.

Keywords: Rust; Device Driver; Arduino; Embedded Systems

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A linguagem de programação Rust tem ganhado tração nos últimos anos, com gigantes como Amazon, Microsoft, Google, entre outros, lançando projetos novos em Rust ou reimplementando sistemas já existentes. A Amazon já lança publicamente produtos em Rust desde 2018, como *Firecracker* e *Bottlerocket*, especialmente para aumentar a segurança de suas programações, alegando que o Rust fornece a eficiência do C/C++ sem os riscos de comportamento indefinido e problemas com memória, opinião compartilhada por Linus Torvalds, criador do Linux, que recentemente passou a aceitar *device drivers* em Rust em seu *kernel* (SHANE MILLER AND CARL LERCHE, 2022).

Há alguns anos a Microsoft revelou que a maioria dos bugs descobertos no Windows era devido a falhas de memória. Ela alega que não é possível reescrever tudo que ela quer em Rust, então optou pelo *Project Verona*, que é algo similar a uma linguagem Rust personalizada especialmente para a Microsoft, desenvolvida por ela mesma desde 2019. Membros relevantes da Microsoft dizem que a curva de aprendizado em Rust para programadores bem letrados em C/C++ é muito fácil, e, que, apesar de o projeto poder levar a lugar nenhum, já progrediu o suficiente para justificar a continuidade do projeto (ADAM BURCH, 2019).

A Google declarou no início de 2023 que iria começar lentamente, mas já produzindo uma *toolchain* para o Rust, citando ainda que códigos em Rust aparecerão no binário do Chrome até 2024. Citando benefícios de bibliotecas em Rust: "menos linhas de código para escrever, menos designs de documentos e menos revisões de segurança". Estão também experimentando e investindo na criação de uma ferramenta Rust/C++ bidirecional, ou seja, um software que permitiria comunicação entre componentes programados em cada linguagem, chamado Crubit (DANA JANSENS, 2023).

O Rust pode ser muito interessante para aplicações em sistemas embarcados. Especialmente no que concerne otimização e segurança do uso de memória e segurança contra *bugs*, devido a recursos como *Ownership*, *Borrowing*, Sem Ponteiros Nulos/Pendentes, Concorrência Previsível, etc. Em sistemas embarcados e computação em geral, ingerências no uso da memória pode resultar em uso desnecessário de memória, aumento do tempo de execução, corrupção de memória, vazamento de memória(*Memory Leaks*), estouro de pilha(*Stack Overflow*), condições de corrida(*Race Conditions*), etc. (KLABNIK; NICHOLS, 2023).

O Rust também provê abstrações de alto nível com custo zero e tem uma sintaxe mais clara e concisa, diferentemente do C/C++, em que várias abstrações vêm com custo mais alto de tempo de execução, bem como uma sintaxe que pode se tornar bem complexa dependendo do uso. Outros benefícios do Rust é a garantia de tempo de execução, *Tooling*(editores de texto, ferramentas de análise de códigos, ambientes de desenvolvimento integrado, etc.) mais simples e direto graças ao gerenciador de pacotes *Cargo*, etc. A

principal desvantagem do Rust é a falta de ecossistema e toda a bagagem de código legado existente para C/C++, visto que a linguagem C é muito mais antiga e popular.

Este trabalho visa contribuir para o ecossistema Rust para sistemas embarcados através da criação de um *device driver* para um sensor comum de temperatura e pressão, comparando posteriormente o resultado com o análogo em linguagem C. Para isso, serão usadas HAL e bibliotecas já presentes no ecossistema em Rust, bem como o *device driver* em C++ já existente na plataforma Arduino.

1.1 OBJETIVOS

Este trabalho propõem-se fazer uma comparação entre C/C++ e Rust através do desenvolvimento de um *device driver* em Rust para a plataforma Arduino Mega 2560, seguido de comparações de critérios qualitativos e quantitativos entre os dois *device drivers*.

1.1.1 Objetivos Específicos

- Expor principais características do Rust;
- Demonstrar implementações de *structs* de *device drivers* em Rust;
- Realizar comunicação I2C em Rust;
- Realizar tratamento de erros em Rust;
- Demonstrar *Pattern Matching* em Rust;
- Desenvolver o primeiro *device driver* público para um sensor BMP180 em Rust;
- Comparar dados qualitativos e quantitativos entre o resultado em Rust e seu análogo em C/C++;

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte forma: no Capítulo 2 será vista a fundamentação teórica do trabalho, explicando particularidades da linguagem Rust com exemplos, seguido de um panorama do atual ecossistema em Rust para sistemas embarcados, finalizando com uma noção sobre trabalhos relacionados ao tema deste trabalho; no Capítulo 3 será visto o desenvolvimento do *device driver*, em Rust, para o sensor escolhido, passando desde qual sensor e ferramentas foram utilizadas até detalhes do código-fonte do *device driver*; no Capítulo 4 serão vistos os resultados qualitativos e quantitativos do *device driver* desenvolvido e sua comparação com o *device driver* já existente para a plataforma Arduino e escrito em C++; no Capítulo 5 é feita uma breve revisão e conclusão do trabalho; no Anexo A são expostos integralmente os códigos fonte do *device driver* e da aplicação.

2 FUNDAMENTAÇÃO TEÓRICA

A linguagem Rust nasceu de um projeto pessoal de Graydon Hoare em 2006. A empresa na qual ele era um funcionário pesquisador, Mozilla, começou a patrocinar a linguagem a partir de 2009 como uma parte do software *Servo*, lançado em 2010. No ano seguinte, o compilador original escrito em OCAM foi substituído por um compilador feito em Rust, chamado rustc. A linguagem desenvolveu-se com o tempo e, em 2015, recebeu a primeira versão estável, diferentemente das versões anteriores, nas quais cada atualização gerava grandes alterações estruturais (ANDREW BINSTOCK, 2014).

Em 2020, após problemas internos na Mozilla, um grupo de funcionários que fazia parte do time de desenvolvimento do Rust planejaram criar uma fundação cujo primeiro objetivo seria obter todos os direitos de propriedade intelectual e de nome. Em 2021, a Fundação Rust foi anunciada por suas empresas fundadoras: AWS, Huawei, Google, Microsoft e Mozilla. Desde então, essa fundação e as empresas relacionadas são a ponta de lança do desenvolvimento da linguagem e de suas aplicações (WILLIAMS, 2021).

2.1 PRINCIPAIS CARACTERÍSTICAS DA LINGUAGEM RUST

2.1.1 Propriedade

Talvez a característica principal do Rust seja *Ownership*, ou propriedade. Explicando brevemente e fazendo uma analogia, há duas memórias que podem ser usadas: *stack* e *heap*. *Stack*, ou pilha, é como uma torre de pratos: adiciona-se cada prato novo em cima do outro e, para remover o do fundo da pilha, é preciso remover todos os outros antes. *Heap*, ou monte, é como um grande restaurante, onde há várias mesas, onde as pessoas vão sendo direcionadas pelos garçons a mesas livres de maneira aleatória, e, caso alguma outra pessoa queira sentar à sua mesa, ela terá de perguntar a um garçom em qual mesa você está. Além disso, é mais rápido para os garçons se as mesas vazias e ocupadas estiverem bem segregadas, e as mesas ocupadas pedirem em ordem, minimizando tempo de deslocamento e evitando operar de maneira malabarística para anotar os pedidos.

Pois bem, as mesas vazias são partes da memória *heap* livre, as ocupadas são partes da memória *heap* armazenando dados, o garçom é o processador, o ato de perguntar a ele a localização de uma pessoa é um ponteiro, e assim por diante. Esses métodos de gerenciamento inteligente, eficiente e seguro, evitando erro do garçom, é o que o conceito de *ownership* traz à linguagem Rust. Ele mantém um rastreio de quais partes do código usam determinados endereços de memória *heap*, minimiza a quantidade de dados duplicados na *heap* e remove dados de variáveis que saíram de escopo.

O programador não precisa se preocupar em alocar memória ou liberar memória manualmente ao longo do código, visto que o Rust faz isso automaticamente. Existem três regras que definem o *ownership*:

- Cada valor em Rust possui uma variável chamada de *owner* (dono);
- Pode haver apenas um único dono em um mesmo instante;
- Quando o dono sai do escopo, o valor é eliminado (*dropped*).

2.1.2 Referências

Há outros dois conceitos importantes relacionados ao *ownership*: *reference* e *lifetime*, literalmente referência e tempo de vida. Em Rust, o ato de criar uma referência é chamado de *borrowing*, ou "pegar emprestado"e, assim como o nome diz, consiste em pegar emprestado algo de alguém que ela não esteja usando no momento e, assim que terminar, devolvê-la. É exatamente isso que o Rust faz, permitindo que uma variável pegue emprestado um dado de um *owner*, faça o que precisa fazer e, ao final do escopo, esse valor é devolvido para o dono original.

Antes de explicar mais, é preciso entender que, em Rust, quando uma variável é criada, ela é por padrão uma variável de valor imutável. Caso se queira alterar o valor de uma variável futuramente, é preciso criar ela usando "mut", como pode ser visto no Código 2.1.

```
1         let x = 5
2
3         let mut x = 5
4         }
5
```

Código 2.1 – Exemplo mut

Existem, em Rust, dois tipos diferentes de referências, um em que o dado original volta para o dono com o exato mesmo valor, caso o dono seja uma variável do tipo imutável, ou um que o dado pode ser alterado por quem pegou emprestado, e volta para o dono da maneira que foi alterado. Caso a variável original seja imutável, pode-se haver tantas referências simultâneas a ela quando for preciso, mas, caso seja mutável, pode-se haver apenas uma referência simultânea, evitando, assim, que o valor original seja alterado simultaneamente, causando problemas. Além disso, de forma simultânea, pode-se usar apenas um dos dois tipos de referências.

Há ainda um tipo especial de referência chamado slice, que é basicamente pegar emprestado apenas uma fatia de uma string, ao invés da string completa. Assim, ao

invés de ler a string byte por byte e pedir emprestado byte por byte, slice permite pegar emprestado determinado números de bytes da string de uma vez só.

2.1.3 Tempo de vida

Finalmente, *lifetime*, ou tempo de vida, se refere até onde vai à validade de uma referência. Por padrão, essa validade é o final do escopo atual. Basicamente, o tempo de vida é uma forma para evitar referências pendentes, o que poderia gerar futuros bugs no código. O compilador do Rust possui uma função chamada *Borrow Checker*, que verifica se o alvo de uma referência é chamado além do escopo da referência, gerando um erro de compilação caso esta verificação seja verdadeira.

2.1.4 Enum

Em Rust, *enum* é um tipo de declaração de variável, equivalente ao *enum* da linguagem C. Basicamente, um *enum* lista opções de classe para atribuir um valor a ele passado. Dentro do próprio *enum* já se pode definir o tamanho e formato de cada classe, podendo, dentro da mesma *enum*, ter classes nomeadas ou não, sendo strings, inteiros, vetores, etc., ou mesmo não ter valor nenhum, como pode ser visto no Código 2.2.

```
1    enum Mapa {
2  Sair,
3  Coordenada { x: i16, y: 16},
4  Mensagem(String),
5  Cor(i32, i32, i32),
6    }
7
```

Código 2.2 – Enum

O motivo do Rust permitir uma opção nula ao invés do clássico *null* conforme criado por Tony Hoare, é que se um valor nulo for usado como um valor não nulo, muitos problemas podem aparecer, descrito por ele mesmo como seu "erro de um bilhão de dólares"(TONY HOARE, 2009). Para evitar isso, o Rust simplesmente tem a opção de usar uma *enum* com o valor estando presente ou não.

Caso se queira fazer operações com elementos de uma *enum*, é preciso que a *enum* não seja genérica, caso contrário a compilação acusará erro. Isso ocorre porque o Rust, na busca de evitar o problema comum de assumir que algo é não nulo quando, na verdade, o é, decidiu limitar essa característica em *enums* genéricos por problemas quando desenvolviam a linguagem.

2.1.5 Pattern Matching

O Rust possui um comando chamado "match", que permite comparar um valor com uma série de padrões e então executar o código baseado no padrão em que o valor se encaixar. O comando match também permite agregar parte de valores que correspondem a padrões. Isto pode ser visto no Código 2.3 (KLABNIK; NICHOLS, 2023).

```
enum UsState {
 Alabama,
 Alaska,
 ...
}

enum Coin {
 Penny,
 Nickel,
 Dime,
 Quarter(UsState),
}

fn values(coin: Coin) -> u8 {
    match coin {
     Coin::Penny => 1,
     Coin::Nickel => 5,
     Coin::Dime => 10,
     Coin::Quarter(state) => {
        println!("State quarter from {:?}!", state);
        25
     }
    }
}

```

Código 2.3 – Match

O comando match aceita genéricos como entradas, possibilitando também operações entre eles, desde que sejam da mesma variante. Como o comando match é exaustivo, o uso do match deve levar em conta todas as possibilidades, caso contrário haverá erro de compilação. Se, por exemplo, match for usado para determinar padrões de itens de uma *enum* que contenha um valor nulo, e o match tiver apenas uma operação aritmética, haverá erro de compilação, pois um valor nulo não pode executar essa operação, protegendo o programa do erro inerente aos *nulls*.

Há ainda o comando *if let*, que é basicamente um match que ignora todos os valores quando ele encontra o padrão desejado. Ele diminui a verbosidade e deixa o código mais conciso, porém perdendo a exaustividade do match original.

2.1.6 Tipos Genéricos

Variáveis em Rust que sejam usadas em definições de funções ou *structs* podem ter tipo, ou serem genéricas. Ou seja, pode-se definir que determinada variável de entrada seja i32, ou i32 genérico. O tipo i32 normal, caso receba um valor que ocupe apenas 8 bits, irá completar o restante do espaço com zeros, utilizando memória desnecessária para adequar o valor recebido ao tamanho da variável. O i32 genérico estabelece um limite, mas, caso a variável de entrada seja de 8 bits, quando a função estiver sendo executada no código, a função abstrai isso e opera com um dado i8.

Em *structs*, basta passar diferentes itens na construção, como pode ser visto no Código 2.4.

```
1        struct Teste<A, B, C> {
2                x: A,
3              y: B,
4              z: C,
5        }
6
```

Código 2.4 – Struct Genérica

O compilador abstrai esta *struct* para que, quando for usada no código, qualquer um dos elementos x, y ou z tenham até 3 tipos diferentes. Dessa forma, podem ser, por exemplo, todos i16, ou podem ser i32, u8 e float16. Tipos podem ser usados também em *enums* ou métdos da mesma forma.

Em Rust, funções definidas dentro do contexto de *structs*, *enums* ou implementações são chamadas de *Methods*, ou métodos. Métodos implementam uma funcionalidade específica a uma variável de uma *struct*, utilizando o comando *impl* para adicionar essa dita funcionalidade, desde que ela já tenha sido definida. Isto pode ser visto no Código 2.5.

```
1        struct Retangulo<T> {
2         largura: <T>,
3         altura: <T>,
4        }
5
6        impl Retangulo {
7         fn area(&self) -> <T> {
8         self.largura * self.altura
9         }
10       }
11
```

Código 2.5 – Impl

2.1.7 Traits

Traits, ou traços, são um tipo especial de implementação. Ao definir um *trait* via comando *trait*, pode-se chamá-lo em uma implementação "para"o tipo desejado. Ficará mais claro analisar o Código 2.6 (KLABNIK; NICHOLS, 2023).

```
1         pub trait Summary {
2             fn summarize(&self) -> String;
3         }
4
5         pub struct NewsArticle {
6              pub headline: String,
7              pub location: String,
8              pub author: String,
9              pub content: String,
10        }
11
12        impl Summary for NewsArticle {
13             fn summarize(&self) -> String {
14             format!("{}, by {} ({})", self.headline, self.author, self.
    location)
15             }
16        }
17
```

Código 2.6 – Traits

Dessa forma, caso seja sabido que um método com um mesmo modelo será usado em vários tipos diferentes, mas que possuem uma mesma natureza condutora a mesmas funcionalidades, pode-se simplesmente defini-lo como um *trait* e chamar individualmente os atributos desejados de cada tipo na implementação. Artigos de jornal, livros e artigos científicos são distintos, mas, ao mesmo tempo, formam um grupo com similaridades estruturais. Todos eles terão resumo, autor, títulos, etc.

Ressalta-se que, caso seja criada uma biblioteca, em Rust chamadas de *crate*, com tipos e *traits* já definidos, quando essa *crate* for chamada num código, é necessário que pelo menos o tipo ou traço usado no código exista na *crate*. Se os dois elementos forem externos às *crate* utilizadas, o compilador acusará erro. Isso é chamado de *Orphan Rule* em Rust, e existe para evitar possíveis conflitos na compilação caso duas *crates* externas tenham definido o mesmo tipo e/ou traço.

2.1.8 Implementações Padrão

Também é possível definir uma implementação padrão em um, ou todos, métodos de um *trait*. Isso pode evitar repetidas chamadas de comandos de implementações em comportamentos mais comuns, conforme pode ser visto no Código 2.7 (KLABNIK; NICHOLS, 2023).

```
1       pub trait Summary {
2           fn summarize(&self) -> String {
3               String::from("(Read more...)")
4           }
5       }
6
7       pub struct NewsArticle {
8            pub headline: String,
9            pub location: String,
10           pub author: String,
11           pub content: String,
12      }
13
14      impl Summary for NewsArticle {
15           fn summarize(&self) -> String {
16           format!("{}, by {} ({})", self.headline, self.author, self.
    location)
17           }
18      }
19      let article = NewsArticle {
20          headline: String::from("Penguins win the Stanley Cup
    Championship!"),
21          location: String::from("Pittsburgh, PA, USA"),
22          author: String::from("Iceburgh"),
23          content: String::from(
24              "The Pittsburgh Penguins once again are the best \
25              hockey team in the NHL.",
26          ),
27      };
28
29      println!("New article available! {}", article.summarize());
30
31      This code prints New article available! (Read more...).
32
```

Código 2.7 – Implementação Padrão

Dessa forma, um *trait* pode ter muitas funções diferentes, mas permite que o programador especifique rapidamente apenas uma pequena parte dele. O exemplo foi trivial, mas pode-se ver como isso pouparia bastante tempo em um código longo e repetitivo.

2.1.9 Traits como Parâmetros

Considere o Código 2.8 (KLABNIK; NICHOLS, 2023).

```
1       pub fn notify(item: &impl Summary) {
2           println!("Breaking news! {}", item.summarize());
3       }
```

```
4
5          This code prints Breaking news! (Read more...).
6
```

Código 2.8 – Trait Bounds 1

Levando em conta que o resto do código imediatamente anterior a este já esteja feito, este exemplo permite visualizar que uma específica implementação de um *trait* pode ser usado como parâmetro de uma função, desde que o tipo de função seja compatível com o parâmetro usado.

Pode-se ter mais de um parâmetro na função ao mesmo tempo, bem como garantir que eles sejam do mesmo tipo chamando-os de forma genérica (*Trait Bounds*), conforme pode ser visto respectivamente no Código 2.9 (KLABNIK; NICHOLS, 2023).

```
1        pub fn notify(item1: &impl Summary, item2: &impl Summary) {
2
3        pub fn notify<T: Summary>(item1: &T, item2: &T) {
4
```

Código 2.9 – Trait Bounds 2

Ademais, é possível também em uma mesma função chamar implementações de *traits* diferentes usando "+", o que pode ser feito também de maneira genérica, como se vê no Código 2.10 (KLABNIK; NICHOLS, 2023).

```
1        pub fn notify(item: &(impl Summary + Display)) {
2
3        pub fn notify<T: Summary + Display>(item: &T) {
4
```

Código 2.10 – Trait Bounds 3

Porém, é fácil de perceber que usar muitos *Trait Bounds* torna facilmente a leitura bem desagradável, o que é resolvido por uma sintaxe alternativa em que eles são postos em um laço *where*. Compare esses dois exemplos equivalentes do ponto de vista do compilador, que pode ser visto no Código 2.11 (KLABNIK; NICHOLS, 2023).

```
1        fn some_function<T: Display + Clone, U: Clone +...
2                         Debug>(t: &T, u: &U) -> i32 {
3
4        fn some_function<T, U>(t: &T, u: &U) -> i32
5        where
6           T: Display + Clone,
7           U: Clone + Debug,
8        {
9
```

Código 2.11 – Trait Bounds 4

Certamente o uso do laço *where* facilita muito a leitura por programadores quando há uso dois ou mais *Trait Bounds* como parâmetros em uma mesma função.

2.1.10 Concorrência

Outro ponto forte do Rust é sua concorrência. Para evitar prolongamento, visto que a concorrência em Rust não será explorada no objeto desde trabalho, os pontos fortes da concorrência são os seguintes (KLABNIK; NICHOLS, 2023):

- Utilização do sistema de *Ownership* e *Borrowing* em *threads*;
- Transferência e compartilhamento de tipos entre *threads*;
- Funções Assíncronas;
- Tipos específicos para concorrência de memória compartilhada;
- Operações atômicas.

Estes itens tornam a concorrência menos propensa a erros. Além disso, como há vários comandos associados especificamente a concorrência, exitem muitas *crates* que permitem a expansão de seu uso.

2.2 ARDUINO MEGA 2560

As placas Arduino oferecem um ambiente de teste acessível, simples e rápido para aplicações embarcadas, sendo ideais para este trabalho. O Arduino Mega 2560 foi escolhido para diminuir a probabilidade de problemas de memória e velocidade de execução, visto que ele é uma versão mais potente do muito utilizado Arduino Uno.

2.3 ECOSSISTEMA RUST EM SISTEMAS EMBARCADOS

Rust tem uma sintaxe muito próxima do C/C++, e foi criada principalmente pensando em auxiliar o desenvolvimento de sistemas concorrentes, o que gerou suas características de garantia de segurança de memória, abstrações de custo-zero, iteradores, etc. Essas características, juntamente com uma eficiência ao nível de C/C++, bem como previsibilidade de execução, tornam essa linguagem muito atrativa a aplicações embarcadas. Há, ainda, um estudo de caso da vulnerabilidade *Heartbleed* que concluiu que se o OpenSSL tivesse sido implementado em Rust, ataques maliciosos ainda poderia ter acontecido, mas nenhum dado teria sido vazado (GETREU, 2016).

Sendo segurança, limitação de memória, e velocidade de execução em tempo real talvez as maiores preocupações em sistemas embarcados, é intuitivo pensar que Rust aparente ser a linguagem perfeita para este caso. No entanto, como a linguagem é nova, e foi originalmente pensado para desenvolvimento de sistemas, não para sistemas embarcados, há um ecossistema de Rust embarcado muito limitado se comparado com, por exemplo, C/C++.

Apesar de já haver alguns *device drivers* em Rust, eles estão dispersos para placas diferentes, como ESP32, Cortex e Arduino. Contudo, muita coisa não está unificada, ou

seja, determinados *device drivers* só serão encontrados para uma das placas. Além disso, poucos autores explicam os *device drivers* produzidos em detalhe, o que torna a pesquisa em ecossistemas Rust para embarcados uma procura de códigos sem grandes comentários. Para pessoas sem longa experiência em embarcados pode ser complicado.

Um periférico muito típicos é o motor de passos e, foi encontrado, novamente, apenas um único repositório público em Rust para microcontroladores, descrito pelo autor como "longe de maduro"(HANNOBRAUN, 2020). Portanto, analisando casos como este, percebe-se que o ecossistema *Open Source* de embarcados em Rust está ainda num período nascente, em que nem mesmo todos os periféricos mais simples e típicos estão suportados.

Falando mais especificamente em relação ao Arduino, foi encontrada apenas uma Camada de Abstração de Hardware(*Hardware Abstraction Layer - HAL*) para microcontroladores AVR, possibilitando a criação de códigos em Rust para essas placas. Essa AVR-HAL é fornecida pelo usuário do github Rahix (RAHIX, 2019).

2.4 TRABALHOS RELACIONADOS

Apesar de poucos, alguns trabalhos já foram produzidos sobre Rust para sistemas embarcados e determinadas comparações entre Rust e C. Em 2021 foi publicado: *The Rust Programming Language for Embedded Software Development (Nico Borgsmüller, 2021*), que arbitrou alguns critérios para comparar a adequação de Rust e C para sistemas embarcados, posteriormente testando esses critérios em microcontroladores realizando a mesma função, alternando a linguagem dos códigos.

Borgsmüller chegou à conclusão que Rust e C produzem resultados bastante similares, mas com algumas distinções notáveis. Apesar da velocidade de execução e uso de memória serem muito similares entre os dois, o Rust, no momento, nem sempre permite as mesmas otimizações tão facilmente, além de, no geral, produzir arquivos com tamanhos binários maiores. Além disso, o Rust atualmente só possui suporte para um número limitado de arquiteturas e bibliotecas, ao passo que se assume que o C tenha um compilador para qualquer hardware existente. Finalmente, o ecossistema do C foi considerado muito mais maduro (BORGSMÜLLER, 2021).

As vantagens encontradas no Rust foram paradigmas de programação maiores, incluindo orientação a objetos e programação funcional. As ferramentas de desenvolvimento também foram consideras da mais alta qualidade, o que aumentaria a produtividade, ao passo que o C, ainda que as tenha, são de difícil integração. Mas a motivação principal do autor para o desenvolvimento da tese sobre Rust foi a segurança de memória durante a compilação. Enquanto o programador em C deve tomar cuidado em cada linha de código, o em Rust apenas segue o *feedback* do compilador. Códigos em Rust também foram considerados livres de condições de corrida(*data races*) e de alta segurança de tipos (BORGSMÜLLER, 2021).

Por último, o Rust também possui ótima certificação. As diretrizes revisadas do

padrão 61508 da Comissão Eletrotécnica Internacional, que consiste em métodos sobre sistemas relacionados à segurança, mostram que, de uma perspectiva técnica, a linguagem Rust é muito bem adequada para aplicações de segurança crítica. A linguagem C, por sua vez, é considerada adequada somente dentro de certas limitações (BORGSMÜLLER, 2021).

Em *Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body (Manuel Costanzo, Enzo Rucci, Marcelo Naiouf and Armando De Giusti, 2021)*, há uma comparação em um estudo de caso levando em conta multi-threading, vetorialização, processamento em bloco, e outros parâmetros. Foi encontrado que o Rust não otimiza operações matemáticas tão bem quanto o C em determinados tipos de dados, como um algoritmo de *Simulação de N Corpos*.

No entanto, chegou-se à conclusão de que a escrita e manutenção do código em Rust é mais fácil devido a características de linguagem de alto nível que o C não possui, o que se soma a um código mais compacto em Rust graças a características de linguagens de programação orientada a objetos e de linguagens funcionais. Ademais, foi encontrado que o Rust gerencia a memória de maneira mais eficiente, e, em alguns casos, sem que seja necessária qualquer modificação na lógica computacional, como no próprio trabalho dos autores (COSTANZO *et al.*, 2021).

Eles chegam à conclusão de que o Rust oferece os benefícios das linguagens de mais alto nível que o C, orientadas a objeto e funcionais, sem perder desempenho significativa nessas abstrações. Assim sendo, recomendou-se o Rust como alternativa ao C em para computação de alto desempenho em certos contextos, tendo, ainda mais, o benefício de ser uma linguagem em evolução, com potencial para ser uma opção cada vez mais viável (COSTANZO *et al.*, 2021).

Em 2020, o trabalho *Exploring Rust for Embedded and Critical Systems (André Brandão de Pinho, 2020)* compara a segurança e desempenho entre as duas linguagens para sistemas críticos e apresenta dois estudos de caso: escalonamento preemptivo e coordenativo.

Escalonamento preemptivo é uma técnica que consiste em interromper a execução de uma tarefa para executar outra tarefa de maior prioridade. Isto permite pontualidade em detrimento de simplicidade e custo de operação. O escalonador cooperativo é o oposto: uma tarefa só libera a CPU voluntariamente, ou seja, quando termina sua atividade ou cede explicitamente o controle da CPU para outra tarefa. Isso simplifica o escalonador e reduz a sobrecarga de troca de contexto em detrimento da pontualidade (PINHO, 2020).

Como resultado da operação desses dois escalonadores, um programado em C e outro em Rust, o autor chegou à conclusão de que a linguagem Rust incentiva a escrita de um código seguro, com o sistema de propriedade agregando à segurança de memória, mas, ao mesmo tempo, exigindo escrita mais criteriosa, visto que um código aceitável ao compilador do C pode não ser aceito pelo do Rust. Também elogiou os *Tipos* que

a linguagem Rust oferece, alegando que usar essa característica significa detectar erros durante a compilação. A não existência do ponteiro nulo também agregou para a segurança do código (PINHO, 2020).

Porém, algumas limitações do Rust apareceram conforme o andamento do projeto. O tempo de compilação fica cada vez maior com o aumento dos projetos e as ferramentas de *debugging* não funcionaram perfeitamente. No entanto, há a possibilidade de que alguns desses erros tenham sido causados pelo sistema operacional *Windows 10*, dado que não ocorreram em outros, como o *MacOS*. Por fim, houve um problema de *overflow* de um inteiro que causava pânico no sistema durante execução do código em Rust (PINHO, 2020).

De maneira geral, os resultados destes trabalhos foram o seguinte:

- Velocidade de execução e uso de memória similares;
- Códigos em Rust mais compactos;
- Códigos em Rust com tamanhos binários maiores;
- Paradigmas e *tooling* melhores em Rust;
- Rust é livre de condições de corrida, mais seguro, e mais indicado para aplicações de segurança crítica;
- C otimiza melhor operações matemáticas para determinados tipos de dados;
- Poucas arquiteturas suportadas em Rust, muitas em C;
- Maturidade, popularidade e ecossistema maiores em C.

3 PROJETO DE IMPLEMENTAÇÃO DE UM DEVICE DRIVER PARA PLATAFORMA ARDUINO EM RUST

Este capítulo apresenta o projeto e implementação de um *device driver*, detalhes do sensor usado e sua integração com o microcontrolador AVR, utilização do protocolo I2C para comunicação, e, por fim, a programação do *device driver* em si e a aplicação sugerida.

3.1 O QUE SÃO *DEVICE DRIVERS*?

Device Drivers são *softwares* que criam uma interface entre o sistema operacional/-processador e um *hardware* externo, possibilitando comunicação. Dito isso, o *device driver* é específico, pois é escrito em determinada linguagem e para determinado SO. Então, se ele precisar ser portado para outro SO, ou outra linguagem, precisará ser reescrito. O *device driver* usualmente se comunica com o dispositivo via um barramento(*databus*).

A importância dos *device drivers* é permitir que dispositivos terceiros sejam modulares. Ao invés de ter um único e indivisível *hardware* com todas as funcionalidades desejadas, pode-se simplesmente ter vários *hardwares* separadamente, chamados de periféricos, e os *device drivers* de cada um deles instalados em um dispositivo, conectando-os quando for conveniente.

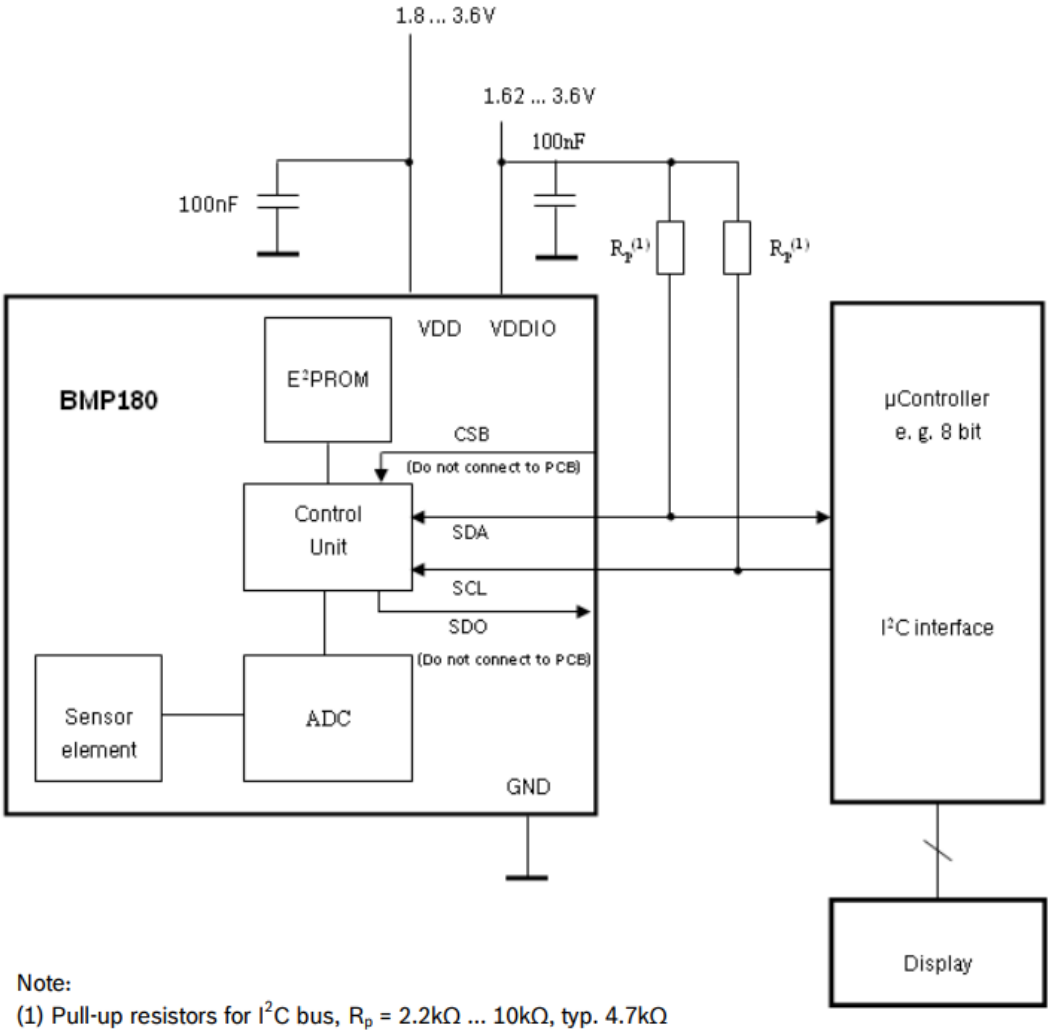
Com as memórias amplamente disponíveis hoje para computadores comuns, quantidade e tamanho em memória de *device drivers* não são uma questão de vida ou morte para não-embarcados, apesar de ainda terem certa importância, mas o mesmo não é necessariamente verdade para sistemas embarcados. O axioma de embarcados de uso mais conservador de memória se aplica também a *device drivers*. Além disso, visto que *device drivers* são *softwares* dos mais fundamentais possíveis, é de extrema relevância que seus códigos sejam seguros e confiáveis, o que depende em boa parte da linguagem usada.

3.2 INTEGRAÇÃO COM O SENSOR BMP180

O BMP180(Bosch) é um sensor de pressão e temperatura de alta precisão, contendo quatro diferentes resoluções, as maiores sendo mais precisas, mas consumindo mais energia e com maior tempo de execução. Ele é otimizado para uso em celulares, GPS e equipamentos externos, como drones. É baseado em tecnologia piezo-resistiva, o que lhe permite alta acuracidade e linearidade, assim como estabilidade de longo prazo. Foi projetado para usar comunicação I2C. A Figura 1 exemplifica um sistema que utiliza tal sensor, mostrando as principais conexões e componentes eletrônicos nele usados.

A primeira coisa a se preocupar são os calibradores. O BMP180 possui uma E2PROM(*Electrically Erasable Programmable Read-Only Memory*) com 11 palavras de 16 bits, totalizando 176 bits. Essas 11 palavras são justamente os 11 calibradores utilizados pelo sensor para compensar os valores brutos de temperatura e pressão, que devem ser

Figura 1 – Circuito de Aplicação BMP180



Fonte: (BOSCH SENSORTEC, 2013)

lidos pelo microcontrolador na inicialização do *device driver*. A Figura 2 mostra a estrutura dos calibradores disposta no BMP180, contendo os parâmetros e endereços de memória de sensor que serão utilizados.

- AC1-AC6: usados em conjunto com a temperatura não compensada para calcular a temperatura real;
- B1 e B2: usados em conjunto com a pressão não compensada para calcular a pressão real;
- MB, MC e MD: usados no cálculo matemático da pressão.

Como os endereços dos registradores são pré-determinados pelo sensor, é preciso apenas implementar sua leitura. Com os calibradores prontos, o sensor pode realizar a primeira medição e seguir num ciclo até quando necessário. O fluxograma demonstrando o ciclo de medição pode ser visto na Figura 4.

Figura 2 – Calibradores BMP180

	BMP180 reg adr	
Parameter	**MSB**	**LSB**
AC1	0xAA	0xAB
AC2	0xAC	0xAD
AC3	0xAE	0xAF
AC4	0xB0	0xB1
AC5	0xB2	0xB3
AC6	0xB4	0xB5
B1	0xB6	0xB7
B2	0xB8	0xB9
MB	0xBA	0xBB
MC	0xBC	0xBD
MD	0xBE	0xBF

Fonte: (BOSCH SENSORTEC, 2013)

Figura 3 – Resoluções de Medição BMP180

Mode	**Parameter** *oversampling_setting*	**Internal number of samples**	**Conversion time pressure max. [ms]**	**Avg. current @ 1 sample/s typ. [µA]**	**RMS noise typ. [hPa]**	**RMS noise typ. [m]**
ultra low power	0	1	4.5	3	0.06	0.5
standard	1	2	7.5	5	0.05	0.4
high resolution	2	4	13.5	7	0.04	0.3
ultra high resolution	3	8	25.5	12	0.03	0.25

Fonte: (BOSCH SENSORTEC, 2013)

No entanto, precisa-se especificar a resolução da medição. O sensor calcula a pressão absoluta, ou seja, o resultado leva em conta a pressão atmosférica. Portanto, como uma diferença de alguns metros tem um impacto quase irrisório no cálculo da pressão sobre um objeto na superfície do planeta devido ao escopo da atmosfera, a resolução de medição visa aumentar a precisão da medição. Como a temperatura não apresenta esse problema prático, a resolução não é usada em seu cálculo. Os parâmetros, especificações temporais, físicas e de energia de cada tipo de resolução estão dispostas na Figura 3.

O tipo de resolução utilizada, obviamente, não é definida ao nível de *device driver*,

mas na aplicação. O *device driver* deve ser flexível o suficiente para permitir que uma aplicação use qualquer uma das resoluções fornecidas pelo sensor.

3.2.1 Comunicação I2C

A comunicação I2C(*Inter-Integrated Circuit*) é um protocolo de comunicação muito comum que permite comunicação entre microcontroladores e periféricos. Foi o protocolo usado por ser exigido pelo sensor. Em resumo, ela opera da seguinte forma:

- Configuração *Master-Slave*: o *Master*(microcontrolador) se comunica com todos os *Slaves*(periféricos), mas estes se comunicam apenas com o *Master*, e não entre si;
- Conexão Física Bidirecional: na SDA(*Serial Data Line*) transitam os dados *Master-Slave*, ao passo que na SCL(*Serial Clock Line*) transita o sinal do *clock* que sincroniza os dados *Master-Slave*;
- Endereçamento: cada *Slave* conectado ao barramento I2C possui um endereço único, o qual é usado pelo *Master* para iniciar comunicação;
- Transferência de Dados: os dados são transferidos em pacotes. Cada pacote inicia com um byte de endereçamento, seguido de um ou mais bytes de dados úteis;
- Condições de Início e Parada: a comunicação começa com o *Master* deixando o SDA em estado baixo e SCL alto, seguido do envio do pacote, e, quando o pacote tiver sido transmitido, o *Master* gera uma condição de parada, pondo o SDA em estado alto e mantendo o SCL também em estado alto.

3.3 FERRAMENTAS UTILIZADAS

Durante a execução deste trabalho, não foi encontrado nenhum *device driver* para o BMP180 para uma HAL AVR em Rust. A escrita deste *device driver* parte de noções de *device driver* em outras linguagens e de especificações do *datasheet* do sensor. Mas, para que um *device driver* possa ser escrito em Rust e compreendido pela arquitetura AVR dos microcontroladores presentes nas placas Arduino, é preciso uma HAL em Rust.

Uma HAL é uma camada de interface de *software* entre o *hardware* e o *software* de aplicação. Ela oculta detalhes do *hardware*, possibilitando o desenvolvimento de softwares de forma portável e independente, facilitando manutenção e desenvolvimento do *software*, abstraindo complexidades via fornecimento de interface de programação de aplicativos(API) simplificada, entre outras funcionalidades.

Neste caso, a HAL é necessária para fazer interface entre o Arduino e qualquer *software* programado em Rust. No entanto, só foi encontrada apenas uma única HAL pública que faz este papel para microcontroladores AVR, criada e disponibilizada por um

indivíduo chamado *Rahix* (RAHIX, 2019). Desenvolver uma HAL própria seria por si só um trabalho muito maior do que o desenvolvimento de um *device driver* para um sensor. Desta forma, será utilizada esta HAL.

3.4 IMPLEMENTAÇÃO

Nesta seção são apresentados os detalhes de implementação do *device driver*.

3.4.1 Algoritmo de Medição

É preciso entender qual o algoritmo de medição dado pelo fornecedor do sensor para correto funcionamento. A Figura 4 mostra o fluxograma detalhado do ciclo de medição, obtido do *datasheet* do fabricante (BOSCH SENSORTEC, 2013). Portanto, o *device driver* deve implementar o fluxograma apresentado. Nota-se uso de matemática custosa, em termos de processamento embarcado, nos cálculos dos valores reais.

3.4.2 Estrutura de um Projeto Rust

Como está sendo usada a HAL do *Rahix*, a estrutura de diretório do projeto leva isso em consideração. Normalmente, se parecerá com algo como o Código 3.1.

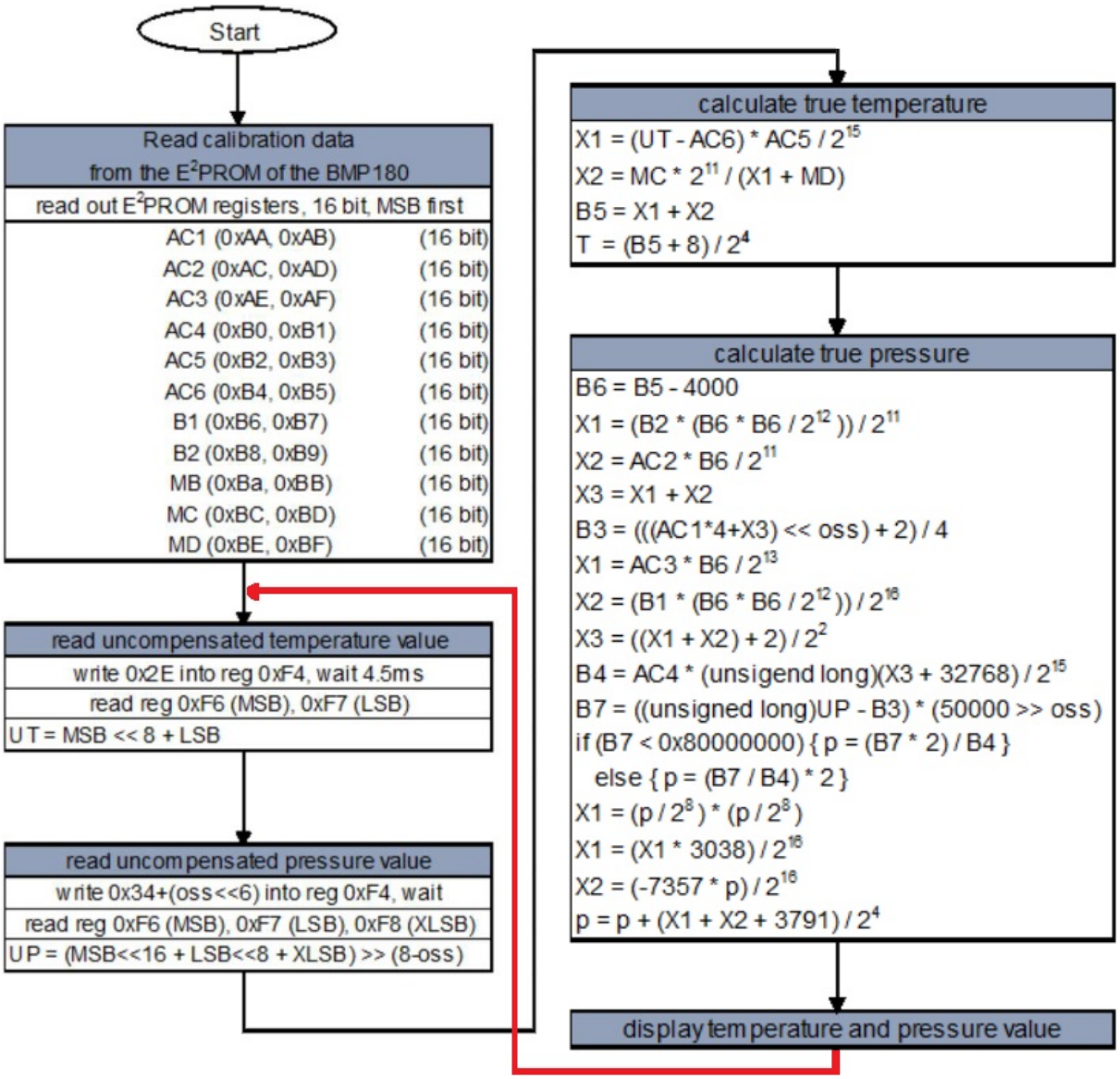
```
1  bmp180-driver
2    -> Cargo.lock
3    -> Cargo.toml
4    -> LICENSE-MIT
5    -> rust-toolchain.toml
6    -> src
7      -> bmp180.rs
8      -> main.rs
9    -> target
10   -> avr-atmega2560
11     -> debug
12       ->bmp180.elf
```

Código 3.1 – Estrutura de Diretório de um Projeto Rust

O arquivo *Cargo.lock* grava cada dependência do projeto desde a última fez que o comando *cargo build* ou *cargo run* foi executado. O arquivo *rust-toolchain.toml* garante a versão do compilador Rust e outras ferramentas associadas. Os arquivos dentro da pasta *src* são os arquivos de fato programados pelo projetista, sendo, neste caso, um driver e uma aplicação. O arquivo *.elf*, assim como em outras linguagens, é o arquivo compilado. Há, também, o arquivo de licença.

É importante que o arquivo *Cargo.toml* (Código 3.2) esteja configurado corretamente. Ele é um arquivo de definição da *build*, incluindo principalmente as dependências

Figura 4 – Algoritmo de Medição



Fonte: (BOSCH SENSORTEC, 2013)

de *crates*(e suas versões) e a versão de compilação. Qualquer *crate* importada para o código deve ser declarada neste arquivo, do contrário, o compilador acusará erro.

```
...
[dependencies]
ufmt = "0.2.0"
ufmt_float = "0.2.0"
embedded-hal = "0.2.7"
avr-device = "0.5.4"

[dependencies.arduino-hal]
git = "https://github.com/rahix/avr-hal"
rev = "45a37eb746d264166c3ea382fed323a204104998"
features = ["arduino-mega2560"]
```

```
12          ...
```

Código 3.2 – Excerto do arquivo Cargo.toml

Foram utilizadas cinco *crates*, sendo dois na implementação do *device driver* e quatro para a implementação da aplicação.

3.4.3 Principais *Crates* Utilizadas

A principal *crate* utilizada é a *arduinohal*, fornecendo uma HAL que implementa sua prórpia API para Arduino e permite programação em linguagem Rust. É também dependência das outras 4 *crates* utilizadas, sendo essencial para o desenvolvimento do *device driver*. A *crate avrdevice* é genérica para microcontroladores AVR, que fornece acesso de baixo nível para os registradores da placa, periféricos e outros usos mais específicos sem replicar uma API para Arduino, ou seja, sem prover funções, estruturas e comportamentos que mimetizam a interface e funcionamento do ambiente de programação do Arduino. Pânicos são funções em Rust que ativam quando um erro crítico acontece durante a execução da aplicação, limpando o *stack* e terminando a execução. A *crate embedded hal* abstrai interfaces comuns a muitos *hardwares* como GPIO(*General Purpose Input/Output*), SPI(*Serial Peripheral Interface*), I2C, UART(*Universal Asynchronous Receiver-Transmitter*), etc. Foi utilizada no *device driver* para definição da *struct* que estabelece o I2C como método de comunicação entre placa e sensor. A *crate ufml* permite formatação de dados para saídas sem depender da *crate std*, que é limitada ou desabilitada em embarcados. Ou seja, permite que uma variável calculada seja impressa na saída como uma informação legível, como o *print* da linguagem *C*. A *crate ufmt_float* permite que os dados formatados para saída sejam do tipo *float*, ou seja, números com sete ou mais casas decimais.

3.4.4 Versões de Compilação

O Rust, assim como qualquer outra linguagem, possui dois métodos de compilação diferentes: *debug* e *release*. *Debug* é a versão de desenvolvimento, usado principalmente para facilitar *debugging*, utilizando otimização mínima, consumindo mais memória e sendo mais lento. *Release* é versão final de aplicação, usando otimizações agressivas durante a compilação para melhorar desempenho e reduzir tamanho do binário. Mesmo placas mais robustas podem não ter memória para uso da versão *debug*, causando erro de difícil identificação para programadores inexperientes. Portanto, foi utilizada a versão *release*.

3.4.5 Inicialização de Parâmetros

O primeiro passo da implementação do *device driver* é a inicialização das variáveis que representam endereços e comandos que devem ser enviados ao dispositivo. O Código 3.3 mostra este caso.

```
const BMP180_ADDR: u8 = 0x77;
const CONTROL_REGISTER: u8 = 0xF4;
const MSB_DATA: u8 = 0xF6;
const LSB_DATA: u8 = 0xF7;
const XLSB_DATA: u8 = 0xF8;
const CMD_READ_TEMPERATURE: u8 = 0x2E;
const CMD_READ_PRESSURE_0: u8 = 0x34;
const CAL_AC1_MSB: u8 = 0xAA;
```

Código 3.3 – Inicialização

Estes endereços de memória são as posições da memória do sensor que possuem funções definidas no *datasheet*, como:

- BMP180_ADDR: registrador contendo identificação do sensor;
- CONTROL_REGISTER: registrador contendo o controle de comunicação I2C do sensor;
- MSB/LSB/XLSB/_DATA: registradores contendo níveis graduados de dados usados no cálculo dos calibradores e de uma resolução de medição mais avançada;
- CMD_READ_TEMPERATURE/PRESSURE: registradores contendo os comandos de realização de leitura de temperatura e pressão;
- CAL_AC1_MSB: registrador contendo a primeira metade dos dados do calibrador AC1.

Foi declarado apenas o endereço do primeiro registrador de calibração, pois o código foi escrito de tal forma que os calibradores são alocados de forma sequencial após o primeiro, assim como consta na Figura 2.

3.4.6 Resolução e Erro

Foi definida a *enum* descrevendo os modos de operação, ou seja, resolução, bem como a do erro. Isto pode ser visto no Código 3.4.

```
pub enum Mode {
 #[allow(dead_code)]
 Bmp085UltraLowPower,
 #[allow(dead_code)]
 Bmp085Standard,
 #[allow(dead_code)]
 Bmp085HighRes,
 #[allow(dead_code)]
 Bmp085UltraHighRes,
}

pub enum Error {
 CommunicationError,
```

```
14          }
```

Código 3.4 – Resolução e Erro

A diretiva de compilação *#[allow(dead_code)]* é usada para o compilador suprimir código que é declarado, mas nunca usado. Nesse caso, a aplicação define uma resolução que não é alterada durante a execução, o que significa que, apesar do *device driver* ter de definir resoluções para qualquer aplicação, todas as que não forem usadas numa determinada aplicação serão inúteis.

3.4.7 *Struct* de Definição e Implementação

O Código 3.5 define a *struct* do *device driver*. Nela constam o método de comunicação (I2C), resolução (oss) e calibradores. Esses são os únicos parâmetros que ele precisa receber para operar.

```
1           pub struct BMP180<I2C> {
2              i2c: I2C,
3              oss: i32,
4              ac1: i16,
5              ac2: i16,
6              ac3: i16,
7              ac4: u16,
8              ac5: u16,
9              ac6: u16,
10             b1: i16,
11             b2: i16,
12             mb: i16,
13             mc: i16,
14             md: i16,
15          }
```

Código 3.5 – Struct de Definição

O Código 3.6 mostra parta da implementação desta estrutura. Ele é parametrizada sobre um tipo I2C, onde irá ler e escrever, e inicia os valores de resolução e calibradores.

```
1           impl<I2C> BMP180<I2C>
2           where
3               I2C: Write + WriteRead,
4   {
5               pub fn new(i2c: I2C) -> Self {
6                   BMP180 {i2c,
7                           oss: 0,
8                           ac1: 0,
9                           ac2: 0,
10                          ac3: 0,
11                          ac4: 0,
12                          ac5: 0,
```

```
13                          ac6: 0,
14                          b1: 0,
15                          b2: 0,
16                          mb: 0,
17                          mc: 0,
18                          md: 0
19                      }
20                  }
```

Código 3.6 – Implementação Struct

3.4.8 Leitura de Calibradores e Cálculo do *b5*

O Código 3.7 mostra a implementação do método *begin()*. Utiliza-se um vetor de 22 *bytes* para armazenas os 11 calibradores. O comando *write_read* realiza uma operação de comunicação I2C bidirecional, ou seja, escreve algo no endereço alvejado e realiza a leitura. Nesse caso, escreve-se no sensor os endereços a serem usados para os calibradores, seguido da leitura desses endereços para serem armazenados no vetor *data*, possibilitando posterior uso no código.

Seguindo a ordem do *datasheet*, a primeira metade do dado de um calibrador é o *byte* de endereço de memória menor, ou seja, o armazenamento é feito em *big endian*. Também é selecionada a resolução escolhida nas linhas 8-13 e feita a transformação de possíveis erros na linha 6.

```
1   pub fn begin(& mut self, mode: Mode) -> Result<(), Error>{
2       let mut data = [0; 22];
3
4       self.i2c.
5           write_read(BMP180_ADDR, &[CAL_AC1_MSB], &mut data)
6           .map_err(|_| Error::CommunicationError)?;
7
8       self.oss = match mode {
9           Mode::Bmp085UltraLowPower => 0,
10          Mode::Bmp085Standard => 1,
11          Mode::Bmp085HighRes => 2,
12          Mode::Bmp085UltraHighRes => 3,
13      };
14
15      self.ac1 =  i16::from_be_bytes([data[0], data[1]]);
16      self.ac2 = i16::from_be_bytes([data[2], data[3]]);
17      self.ac3 = i16::from_be_bytes([data[4], data[5]]);
18      self.ac4 = u16::from_be_bytes([data[6], data[7]]);
19      self.ac5 = u16::from_be_bytes([data[8], data[9]]);
20      self.ac6 = u16::from_be_bytes([data[10], data[11]]);
21      self.b1 = i16::from_be_bytes([data[12], data[13]]);
22      self.b2 = i16::from_be_bytes([data[14], data[15]]);
```

```
23          self.mb = i16::from_be_bytes([data[16], data[17]]);
24          self.mc = i16::from_be_bytes([data[18], data[19]]);
25          self.md = i16::from_be_bytes([data[20], data[21]]);
26
27          Ok(())
28      }
```

Código 3.7 – Leitura dos Calibradores

Faz-se necessário determinar o valor da variável *b5*, conforme o *datasheet*, que será usado nos cálculos da temperatura e pressão. Para isso, é feita uma função que segue à risca o que é dito no *datasheet* e pode ser vista no Código 3.8. Observe que a função toma como entrada *ut* (ver Figura 4) e utiliza certas constantes dos calibradores para determinação de *b5*.

```
1       fn calculate_b5(&mut self, ut: i32) -> i32 {
2           let x1 = (ut - self.ac6 as i32) * (self.ac5 as i32) >> 15;
3           let x2 = (self.mc as i32) * 2048 / (x1 + (self.md as i32));
4           x1 + x2
5       }
```

Código 3.8 – Cálculo b5

3.4.9 Leitura e Cálculo da Temperatura

Entra-se na função da temperatura. É enviado o comando para inicializar o procedimento de medição de temperatura, sendo necessário aguardar 5ms antes da leitura da temperatura não compensada. Lê-se os dados feitos pela leitura(*MSB_DATA, LSB_DATA*), armazena-se eles no vetor *data* e rearranja-se a ordem dos elementos do vetor para obtenção do valor correto. Após a medição, o valor obtido é testado para erro, e, caso não haja erro, é calculado o valor real conforme a Figura 4. Isso pode ser visto no Código 3.9.

```
1       fn i2c_read_raw_temperature(&mut self) -> Result<i32, Error> {
2           let mut data = [0; 2];
3
4           self.i2c
5               .write(BMP180_ADDR, &[CONTROL_REGISTER, CMD_READ_TEMPERATURE
    ])
6               .map_err(|_| Error::CommunicationError)?;
7
8           arduino_hal::delay_ms(5);
9
10          self.i2c
11              .write_read(BMP180_ADDR, &[MSB_DATA, LSB_DATA], &mut data)
12              .map_err(|_| Error::CommunicationError)?;
13
14          Ok((data[0] as i32) << 8 | data[1] as i32)
15      }
```

```
16
17      pub fn read_temperature(& mut self) -> Result<f32, Error> {
18          let ut: i32;
19
20          match self.i2c_read_raw_temperature() {
21              Ok(value) => ut = value,
22              Err(_) => return Err(Error::CommunicationError),
23          }
24
25          let b5 = self.calculate_b5(ut);
26
27          let temperature = (b5 + 8) >> 4;
28
29          Ok((temperature as f32) / 10.0)
30      }
```

Código 3.9 – Funções da Temperatura

3.4.10 Leitura e Cálculo da Pressão

Para a leitura da pressão não compensada, a função é praticamente a mesma da temperatura, à exceção de que o comando de leitura está em um registrador diferente e que, podendo haver uma resolução de medição maior, esse dado extra é armazenado no registrador de constante *XLSB*. Para o cálculo do valor real, porém, há matemática mais complexa. Primeiro, alonga-se o número de *bits* dos valores de temperatura e pressão não compensadas para 32 *bits*, evitando erro de compilação acusando cálculos de variáveis de tamanhos não compatíveis. Por fim, segue-se o algoritmo da Figura 4. Certo cuidado deve ser tomado no cálculo da variável *b7*, pois ele pode resultar em um número de 64 *bits*.Isto pode ser visto no Código 3.10.

```
1       pub fn read_pressure(&mut self) -> Result<i32, Error> {
2           let ut: i32;
3           let up: i32;
4
5           match self.i2c_read_raw_pressure() {
6               Ok(value) => up = value,
7               Err(_) => return Err(Error::CommunicationError),
8           }
9
10          match self.i2c_read_raw_temperature() {
11              Ok(value) => ut = value,
12              Err(_) => return Err(Error::CommunicationError),
13          }
14
15          let b5 = self.calculate_b5(ut);
16          let b6 = b5 - 4000;
```

```
17          let x1 = ((self.b2 as i32) * (b6 * b6 >> 12)) >> 11;
18          let x2 = (self.ac2 as i32) * b6 >> 11;
19          let x3 = x1 + x2;
20          let b3 = (((self.ac1 as i32 * 4 + x3) << self.oss) + 2) >> 2;
21          let x1 = (self.ac3 as i32) * b6 >> 13;
22          let x2 = (self.b1 as i32) * ( b6 * b6 >> 12) >> 16;
23          let x3 = ((x1 + x2) + 2) >> 2;
24          let b4 = (self.ac4 as u32) * (x3 + 32768) as u32 >> 15;
25
26          let b7 = (up - b3) as u64 * (50000 as u64 >> self.oss);
27
28          let p;
29          if b7 < 0x80000000 {
30              p = (b7 * 2) / b4 as u64 ;
31          } else {
32              p = (b7 / b4 as u64) * 2;
33          }
34
35          let x1 = (p >> 8) * (p >> 8);
36          let x1 = (x1 * 3038) >> 16;
37          let x2 = (-7357 * p as i64) >> 16;
38          let p = p as i64 + ((x1 as i64 + x2 + 3791) >> 4);
39          Ok(p as i32)
40      }
41  }
```

Código 3.10 – Funções da Pressão

3.4.11 Aplicação

Com o *device driver* já completo, segue-se para a aplicação, que pode ser visto no Código 3.11:

- Linhas 1 e 2: declaração de não utilização de *crates* inapropriadas para embarcados;
- Linhas 3 e 4: declaração das *crates* necessárias para aplicação;
- Linhas 6 a 8: importação de outro arquivo Rust no mesmo diretório do arquivo da aplicação e uso de *enum* de escolha de resolução;
- Linhas 10 a 35: inicialização do *Panic Handler* que, em caso de pânico, desativa interrupções do *firmware* nativo da placa, inicia o Arduino, seus pinos e sua comunicação serial e informa o local do erro enquanto o LED pisca com frequência de 10Hz.

```
1 #![no_std]
2 #![no_main]
3 use ufmt_float::uFmt_f32;
```

```
4  use arduino_hal::prelude::*;
5
6  mod BMP180;
7  use BMP180::BMP180;
8  use BMP180::Mode;
9
10 #[panic_handler]
11 fn panic(info: &core::panic::PanicInfo) -> ! {
12     avr_device::interrupt::disable();
13
14     let dp = unsafe { arduino_hal::Peripherals::steal() };
15     let pins = arduino_hal::pins!(dp);
16     let mut serial = arduino_hal::default_serial!(dp, pins, 57600);
17
18     ufmt::uwriteln!(&mut serial, "Firmware panic!\r").unwrap_infallible
       ();
19     if let Some(loc) = info.location() {
20         ufmt::uwriteln!(
21             &mut serial,
22             "  At {}:{}:{}\r",
23             loc.file(),
24             loc.line(),
25             loc.column(),
26         )
27         .unwrap_infallible();
28     }
29
30     let mut led = pins.d13.into_output();
31     loop {
32         led.toggle();
33         arduino_hal::delay_ms(100);
34     }
35 }
```

Código 3.11 – Inicialização da Aplicação

Partindo para a execução da aplicação, abre-se o laço *main*, que é o laço de aplicação. Então, são estabelecidos os seguintes itens, conforme Código 3.12:

- Linha 3: escolha da resolução de medida;
- Linhas 5 a 7: declaração dos pinos do Arduino e inicialização da comunicação serial;
- Linhas 9 a 14: inicialização do I2C, escolha dos pinos usados para SCL e SDA, e frequência de comunicação do barramento(100k *bits*s);
- Linha 16: criação de variável de tipo da *struct* definida do *device driver*;
- Linhas 18 a 20, 27 a 29 e 37 a 39: tratamento de erro de comunicação;

- Linhas 22 a 43: inicialização de *loop* com intervalos de 1 segundo, onde é medida a temperatura e pressão.

```
1  #[arduino_hal::entry]
2  fn main() -> ! {
3      let mode = Mode::Bmp085UltraHighRes;
4
5      let dp = arduino_hal::Peripherals::take().unwrap();
6      let pins = arduino_hal::pins!(dp);
7      let mut serial = arduino_hal::default_serial!(dp, pins, 57600);
8
9      let i2c = arduino_hal::I2c::new(
10         dp.TWI,
11         pins.d20.into_pull_up_input(),
12         pins.d21.into_pull_up_input(),
13         100000,
14     );
15
16     let mut BMP180 = BMP180::new(i2c);
17
18     if let Err(_e) = BMP180.begin(mode) {
19         ufmt::uwriteln!(&mut serial, "Error communicating with the
    BMP180 sensor.").unwrap();
20     }
21
22     loop {
23         match BMP180.read_temperature() {
24             Ok(temperature) => {
25                 ufmt::uwriteln!(&mut serial, "Tempature: {}", uFmt_f32::
    Two(temperature)).unwrap();
26             }
27             Err(_e) => {
28                 ufmt::uwriteln!(&mut serial, "Error communicating with
    the BMP180 sensor.").unwrap();
29             }
30         }
31
32         match BMP180.read_pressure() {
33             Ok(pressure) => {
34                 ufmt::uwriteln!(&mut serial, "Pressure {}", pressure).
    unwrap();
35             }
36             Err(_e) => {
37                 ufmt::uwriteln!(&mut serial, "Error communicating with
    the BMP180 sensor.").unwrap();
38             }
39         }
40
```

```
41          arduino_hal::delay_ms(1000);
42      }
43  }
```

Código 3.12 – Execução da Aplicação

Lembrando que para compilar em versão *release* deve-se utilizar o comando *cargo build –release*. Então, antes de gravar o arquivo compilado na memória do microcontrolador, deve-se identificar qual barramento de conexão está sendo utilizado pelo computador. Caso seja, por exemplo, COM3, deve-se utilizar o comando *$env:RAVEDUDE_PORT = "COM3"* para a HAL compreender que é por esse barramento que deve ser transmitido o arquivo compilado durante o comando *cargo run*.

4 RESULTADOS

Com o *device driver* escrito e implementado em aplicação, pode-se, por fim, obter dados comparativos em relação ao *device driver* já existente em C/C++. Especificamente, o *device driver* em C/C++ utilizado para comparação foi o *Adafruit* (GUNDRY, 2020).

4.1 ANÁLISE QUALITATIVA DA IMPLEMENTAÇÃO EM RUST

Inicialmente, serão analisados aspectos qualitativos, em modelo de vantagens e desvantagens.

4.1.1 Desvantagens

As desvantagens começaram logo no início, com falta de opções de HALs. Havendo apenas uma única HAL pública, não há segunda opção caso haja qualquer problema decorrente. Para este projeto de implementação não houve problema real, mas, para outros projetos que visam programar em Rust para Arduino, a única opção seria informar o desenvolvedor da HAL e aguardar resposta. Porém, isso também significa que criar uma HAL própria e mais otimizada pode dotar o nome do criador da dita HAL de grande prestígio no ecossistema Rust.

Como foi dito, não houve problema real, essencial, na utilização da HAL do *Rahix* para este projeto, mas houve problemas contingentes. Como a versão de compilação padrão da HAL é a de *debug*, a compilação sem especificação da versão *release* casou travamento do dispositivo, possivelmente por falta de memória. Até ser percebido o problema e alterado para a versão *release*, perdeu-se certo tempo. Além disso, houve problema de compatibilidade com a versão 1.0.0 da *crate embedded_hal*, o que forçou o uso da versão 0.2.7.

Exemplos de implementação em Rust para embarcados, diferentemente de opções de HAL, existem em certa quantidade no ecossistema, como o próprio motor de passos (HANNOBRAUN, 2020), ou o mesmo sensor BMP180 utilizado, porém para outras placas, especialmente ARM Cortex. Portanto, quem quiser programar um novo software em Rust para embarcados não parte do zero, ainda que tenha muito menos abundância de pesquisa do que se estivesse programando em C. Dito isso, como o ecossistema é recente, a chance de encontrar algo similar ao que se pretende é inversamente proporcional ao nível de especificidade e complexidade do projeto pensado. Para se ter uma ideia, o *device driver* do motor de passo citado, que é o mais popular do ecossistema, ainda não está 100% otimizado.

Pode-se resumir as desvantagens qualitativas da seguinte forma:

- Opção única de HAL(e suas consequências);
- Ecossistema limitado e de baixa complexidade.

4.1.2 Vantagens

A primeira vantagem que foi visível durante este projeto foi a checagem de erros em compilação. É muito frequente códigos em C não funcionarem e não acusarem exatamente o erro e a linha de código do erro. À primeira vista, os erros de compilação do Rust podem ser incômodos, visto que não é possível nem mesmo testar a aplicação, ainda que errônea ou mal otimizada, dado que o compilador do Rust acusa erro não só em falha de execução, mas também em potenciais riscos à execução.

Normalmente, os potenciais riscos que normalmente causam problemas em C e que o compilador do Rust não deixa passar são os seguintes:

- Tamanho de vetores: frequentemente, em C, cria-se um vetor de tamanho X, mas ao longo de seu uso no código, o seu tamanho acaba ficando, digamos, 3X. Isso é uma das causas mais comuns de falha de execução em C. O compilador do Rust detecta isso e acusa como erro, interrompendo a compilação;
- Compartilhamento de variáveis: também acaba ocorrendo em C que uma mesma variável é utilizada em mais de um local, e, ao ser alterada em um deles, não opera corretamente dentro de outro. O sistema de *ownership* do Rust previne isso, e mesmo variáveis mutáveis têm o seu controle;
- Impossibilidade de variáveis globais: com a mesma finalidade do item anterior, o Rust tradicionalmente não permite variáveis globais, apenas constantes globais.

Qualquer um que tenha programado em C/C++ por algum tempo sabe que esses itens são causas de muita dor de cabeça em códigos mais longos e complexos. Por mais que a insistência do Rust por vezes seja incômoda em não permitir nem sequer o teste de uma execução sem completa correção desses itens, isso é compensado pelo aviso da exata da linha do erro, exposição do erro e mesmo sugestão de correção. Determinados erros têm códigos associados a eles, que estão em uma coletânea feita pelos desenvolvedores do Rust, dando explicações e soluções detalhadas para cada erro.

Outra vantagem notável é o gerenciamento, ou tratamento de erros. Basicamente, sempre que um dado é transmitido via I2C, verifica-se para erro de comunicação. Isso pode parecer trivial, mas faz com que não se tenha que depurar erros espúrios somente para descobrir mais tarde que o problema.

Pode-se resumir as vantagens qualitativas da seguinte forma:

- Compilação criteriosa;
- Auxílio em identificação e correção de erros.

4.2 ANÁLISE QUANTITATIVA DA IMPLEMENTAÇÃO EM RUST

Para análise quantitativa, arbitraram-se os parâmetros mais comuns ligados à velocidade e memória.

4.2.1 Tamanho Binário e Uso de Memória

O tamanho binário do arquivo compilado e uso de memória, especialmente *flash*, são critérios de suma importância para sistemas embarcados. Tamanhos binários menores tendem a significar os seguintes pontos:

- Menos espaço de armazenamento usado;
- Carregamento mais rápido do armazenamento para a RAM;
- Consumo de bateria em taxa reduzida;
- Dissipação de menos calor;
- Quantidade menor de vulnerabilidades devido à quantidade menor de dados.

Começando a análise quantitativa pelo tamanho do binário e uso de memória *flash* e RAM dos arquivos compilados, deve-se realizar o comando de análise binária chamado *objdump*, alvejando o arquivo compilado. Esse comando faz parte da coleção de ferramentas de programação desenvolvido pelo *GNU Project* chamado *GNU Binutils*. A execução desse comando fornece informações no estilo segundo Código 4.1, conforme utilizado para o arquivo compilado do tipo elf gerado.

```
1  PS C:\Users\Marcos\bmp180> avr-objdump ./target/avr-atmega2560/release/
       bmp180-final.elf --section-headers
2
3  ./target/avr-atmega2560/release/bmp180-final.elf:     file format elf32-
       avr
4
5  Sections:
6  Idx Name          Size      VMA       LMA       File off  Algn
7    0 .data         000000f6  00800200  00001f9a  00002050  2**4
8                    CONTENTS, ALLOC, LOAD, READONLY, DATA
9    1 .text         00001f9a  00000000  00000000  000000b4  2**1
10                   CONTENTS, ALLOC, LOAD, READONLY, CODE
11   2 .bss          00000001  008002f6  008002f6  00002146  2**0
12 ...
```

Código 4.1 – Comando objdump no device driver bmp180 em Rust.

A coluna nomeada (*size*) para *.data*, *.text* e *.bss* têm valores de *bytes* em base hexadecimal, significam o seguinte:

- .data: esta seção contém as variáveis de dados inicializadas usadas pelo programa. O tamanho desta seção representa a quantidade de memória RAM necessária para armazenar dados inicializados durante a execução do programa;
- .text: esta seção contém o código do programa, ou seja, as instruções executáveis. O tamanho desta seção representa a quantidade de memória *flash* necessária para armazenar o código compilado;

- .bss: esta seção contém as variáveis de dados não inicializadas usadas pelo programa. O tamanho desta seção representa a quantidade de memória RAM necessária para armazenar dados não inicializados durante a execução do programa.

O tamanho binário total do arquivo compilado nada mais é do que a soma dos *bytes* usados por esses 3 itens, ao passo que o uso de memória *flash* é a quantidade de bytes da seção *.text* e o uso de memória RAM a quantidade de bytes somadas das seções *.data* e *.bss*. Portanto, para o *device driver* desenvolvido, tem-se:

- Tamanho binário: 8337 *bytes*;
- Uso de memória *flash*: 8090 *bytes*;
- Uso de memória RAM: 247 *bytes*.

Realizando o mesmo procedimento para o *device driver* em C/C++, obtiveram-se os resultados conforme Código 4.2. Lembrando que, embora o *device driver* do *Adafruit* também tenha a capacidade de cálculo de altitude, tudo que se refere a isso foi apagado do código. Isso garante uma comparação justa entre os dois *device drivers*, levando em conta somente medição de temperatura e pressão, e cálculos necessários a isso.

```
1  PS C:\Users\Marcos\AppData\Local\Temp\arduino\sketches\
       D131775C7F57EEA33413E780D5D7BBA2> avr-objdump BMP085test.ino.elf --
       section-headers
2
3  BMP085test.ino.elf:     file format elf32-avr
4
5  Sections:
6  Idx Name          Size      VMA       LMA       File off  Algn
7    0 .data         00000090  00800200  000023f8  0000248c  2**0
8                    CONTENTS, ALLOC, LOAD, DATA
9    1 .text         000023f8  00000000  00000000  00000094  2**1
10                   CONTENTS, ALLOC, LOAD, READONLY, CODE
11   2 .bss          00000189  00800290  00800290  0000251c  2**0
12 ...
```

Código 4.2 – Comando objdump no device driver bmp180 em C++.

- Tamanho binário: 9360 *bytes*;
- Uso de memória *flash*: 9216 *bytes*;
- Uso de memória RAM: 144 *bytes*.

Nota-se que o Rust consome mais memória RAM devido a recursos mais avançados do Rust em relação ao C, como sistema de empréstimo de variáveis, propriedade, tempo de vida, etc. Além disso, na mesma lógica, o compilador em Rust checa mais para segurança de memória, condições de corrida, e outros recursos. Em suma, aquilo que diferencia o Rust do C também é responsável por exigir mais recursos de memória.

Para a memória flash, apesar do ecossistema em C estar muito bem otimizado para gerar o menor tamanho possível, a declaração do não uso da *crate std* e uso de *crates* específicas para embarcados, como a *embedded_hal*, que foi usada neste trabalho, permitem que códigos embarcados em Rust sejam competitivos em relação ao C. Apesar de este *device driver* apresentar tamanho binário menor em Rust, a disputa é acirrada nos softwares na totalidade, geralmente com o C tendendo a apresentar tamanhos binários menores.

Verifica-se, contudo, que o Rust tem potencial para ser seriamente considerado por desenvolvedores para uso em embarcados, especialmente para aplicações mais sensíveis, devido à sua vantagem de segurança, uma vez que parece possível, com as otimizações corretas e a depender a aplicação, obter tamanho binário menor que o C com praticamente a mesma eficiência operacional. As comparações entre os *device drivers* fornecem os seguintes resultados:

- Tamanho binário: C 12,27% maior que Rust;
- Uso de memória *flash*: C 13,92% maior que Rust;
- Uso de memória RAM: C 41,6% menor que Rust.

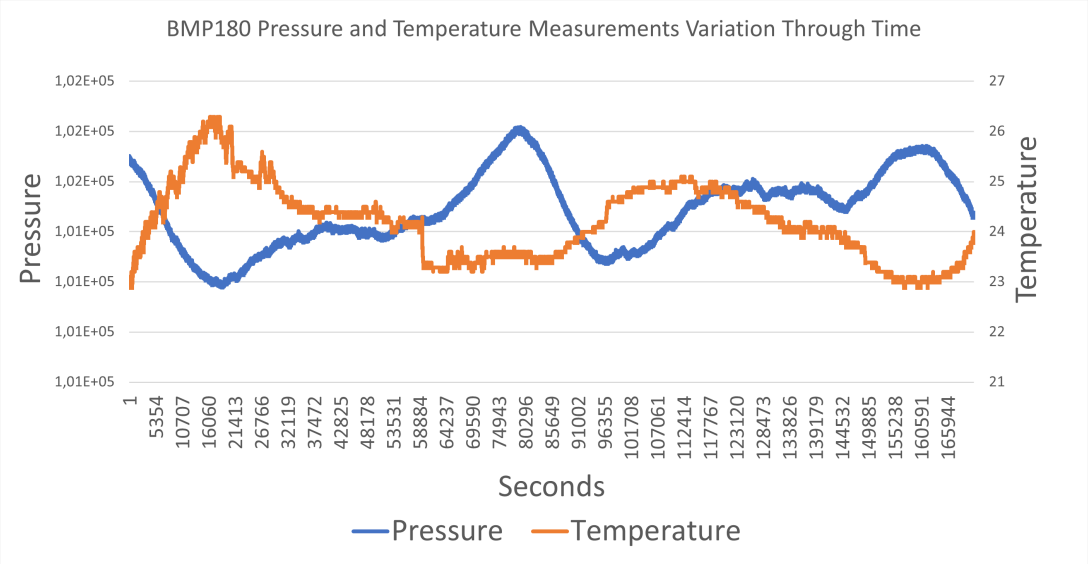
4.2.2 Tempo de Compilação e Execução

Para os tempos de compilação e execução, levando em consideração uma *build* limpa, ou seja, sem nenhum código fonte pré-compilado, tem-se o seguinte:

- Tempo de compilação em Rust: 10 a 12 segundos;
- Tempo de compilação em C: 2 a 2,5 segundos;
- Tempo de execução em Rust: 3 a 3,5 segundos;
- Tempo de execução em C: 3 a 4 segundos.

Há diferença notável entre os tempos de compilação, com o do Rust sendo, em média, quatro a cinco vezes mais lento. No entanto, isso é esperado. É consenso na comunidade de Rust, incluindo por parte dos autores do livro *The Rust Programming Language*, que a compilação em Rust demora mais tempo pelos motivos já explicados de compilação mais custosa e complexa em relação a compiladores do C. Ainda assim, é importante salientar que, por mais que a razão entre os tempos de compilação tenda a seguir por aqueles valores obtidos, o tempo em absoluto depende muito do hardware e outros fatores.

Figura 5 – Teste de Execução



4.3 EXEMPLO DE FUNCIONAMENTO

Como teste de funcionamento do sensor, colheu-se dados de funcionamento de cerca de 170k iterações, ou seja, cerca de 170k segundos, o que equivale a aproximadamente a 2 dias de funcionamento ininterrupto. Com esses dados, o gráfico visto na Figura 5 foi gerado.

A curva azul, a pressão, é medida em Pascal, à esquerda do gráfico, ao passo que a curva laranja, a temperatura, é medida em Celsius, à direita do gráfico. Como o ar atmosférico é mais denso quanto mais baixa a temperatura a que ele estiver submetido, e uma densidade(concentração de massa) maior do ar significa uma pressão maior exercida pelo ar, faz completo sentido que a pressão atmosférica seja menor durante o período quente do dia, e que ela seja maior durante o período frio do dia, o que é exatamente o que se vê no gráfico. Para o ar atmosférico, são duas grandezas inversamente proporcionais, com valores de temperatura e pressão condizentes com o mês de junho em Blumenau, em 2024. Portanto, pode-se concluir que o sensor funcionou corretamente.

5 CONCLUSÃO

Neste trabalho, selecionou-se um sensor BMP180 para ser alvo do desenvolvimento de um *devide driver* escrito em Rust para operar em um Arduino Mega 2560. Após selecionar todos os instrumentos necessários, desde a HAL, *crates* e *device driver* análogo em C/C++, foi escrito o *device driver* em Rust e realizados testes comparativos de compilação e execução, comparando aspectos qualitativos como vantagens e desvantagens, e quantitativos como tamanho do binário, uso memória RAM, tempo de compilação e execução. Também foi produzido um gráfico mostrando o funcionamento do sensor com o *device driver* desenvolvido, demonstrando ser um *device driver* funcional e atendendo aos requisitos do proposto pelo sensor.

O uso da memória RAM durante compilação foi muito maior em Rust em relação ao C, devido ao compilador do Rust realizar abstrações maiores e mais complexas. Surpreendentemente, o tamanho binário do *device driver* foi significativamente menor em Rust, um resultado oposto ao de outros trabalhos já citados. Provavelmente isso se deve ao fato de o *device driver* produzido ser relativamente simples, quase não utilizando nenhum recurso exclusivo da linguagem Rust, o que indica, porém, que o tamanho binário de softwares em Rust se deve aos recursos mais avançados do Rust, e, em uma situação onde tais recursos não sejam tão necessários ou convenientes, o Rust possa fornecer uma opção de arquivo compilado menor em relação ao C. Ademais, é da opinião do autor deste trabalho que, para quem tem certa experiência em programar em C, a linguagem Rust é muito acessível e de rápida adaptação.

O ecossistema de software ainda em formação pode ser problemático caso queira-se construir uma aplicação mais complexa. Apesar da única HAL pública não ter causado problemas neste trabalho, não há como dizer em quais casos ela falharia. Contudo, como a linguagem parece estar ganhando um bom impulsionamento por grandes empresas, como demonstrado no Capítulo 1, espera-se que o ecossistema amadureça nos próximos anos.

Os aparentemente intermináveis erros de compilação por potenciais falhas de segurança também consumiram certo tempo, coisa comum caso seja a primeira vez que um programador C use Rust. Além disso, é importante salientar a compilação e execução em modo *release* para embarcados, visto que o padrão *debug* simplesmente falha sem qualquer *feedback* ao usar a HAL.

Para trabalhos futuros seria interessante analisar maiores complexidades de códigos em Rust e seus análogos em C/C++ para verificar o gradiente de tamanho binário do arquivo compilado. Desta forma, será possível ter uma noção mais acurada das forças do compilador do Rust. Seria também muito interessante, mas muito mais complexo, comparar as duas linguagens em códigos maiores, que requiram mais recursos da linguagem Rust e sejam complexos, com o intuito de aferir uma comparação mais condizente com o cenário profissionalmente competitivo.

REFERÊNCIAS

ADAM BURCH. Using Rust in Windows. **Microsoft Security Responce Center**, 2019.

ANDREW BINSTOCK. The Rise And Fall of Languages in 2013. **Dr. Dobb's Journal**, 2014.

BORGSMÜLLER, Nico. **The Rust programming language for embedded software development**. 2021. F. 63, lxvii. Ingolstadt.

BOSCH SENSORTEC. **BMP180 Digital pressure sensor**. [*S.l.*], 2013. Disponível em: https://cdn-shop.adafruit.com/datasheets/BST-BMP180-DS000-09.pdf.

COSTANZO, Manuel; RUCCI, Enzo; NAIOUF, Marcelo; GIUSTI, Armando De. **Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body**. [*S.l.*: *s.n.*], 2021. arXiv: 2107.11912 [cs.PL].

DANA JANSENS. Supporting the Use of Rust in the Chromium Project. **Google Security Blog**, 2023.

GETREU, Jens. Embedded system security with Rust. **Tallinn University of Technology: Tallinn, Estonia**, 2016.

GUNDRY, Tyeth. **Adafruit-BMP085-Library**. [*S.l.*], 2020. Disponível em: https://github.com/adafruit/Adafruit-BMP085-Library.

HANNOBRAUN. **Universal Stepper Motor Interface**. [*S.l.*], 2020. Disponível em: https://github.com/hannobraun/stepper.

KLABNIK, S.; NICHOLS, C. **The Rust Programming Language, 2nd Edition**. [*S.l.*]: No Starch Press, 2023. ISBN 9781718503106.

PINHO, André Brandão de. **Exploring Rust for Embedded and Critical Systems**. 2020. Diss. (Mestrado) – Universidade do Minho.

RAHIX. **AVR-HAL**. [*S.l.*], 2019. Disponível em: https://github.com/Rahix/avr-hal.

SHANE MILLER AND CARL LERCHE. **Sustainability with Rust**. [*S.l.*], 2022. Disponível em: https://aws.amazon.com/pt/blogs/opensource/sustainability-with-rust/.

TONY HOARE. Null References: The Billion Dollar Mistake. *In*: QCON. [*S.l.*: *s.n.*], 2009.

WILLIAMS, Ashley. Hello World! **Rust Foundation News**, 2021.

ANEXO A – Códigos do *device driver* e aplicação

Neste anexo estão expostos em sua integralidade os códigos fonte do *device driver* desenvolvido e da aplicação.

```
use embedded_hal::blocking::i2c::{Write, WriteRead};

/* Declaration of constants, according to the
   BMP180 Datasheet Digital pressure sensor - Bosch */
const BMP180_ADDR: u8 = 0x77;
const CONTROL_REGISTER: u8 = 0xF4;
const MSB_DATA: u8 = 0xF6;
const LSB_DATA: u8 = 0xF7;
const XLSB_DATA: u8 = 0xF8;
const CMD_READ_TEMPERATURE: u8 = 0x2E;
const CMD_READ_PRESSURE_0: u8 = 0x34;
const CAL_AC1_MSB: u8 = 0xAA;

/* Operation modes */
pub enum Mode {
    #[allow(dead_code)]
    Bmp085UltraLowPower,
    #[allow(dead_code)]
    Bmp085Standard,
    #[allow(dead_code)]
    Bmp085HighRes,
    #[allow(dead_code)]
    Bmp085UltraHighRes,
}

pub enum Error {
    CommunicationError,
}

pub struct Bmp180<I2C> {
    i2c: I2C,
    oss: i32,
    ac1: i16,
    ac2: i16,
    ac3: i16,
    ac4: u16,
    ac5: u16,
    ac6: u16,
    b1: i16,
    b2: i16,
    mb: i16,
    mc: i16,
    md: i16,
```

```
}

impl<I2C> Bmp180<I2C>
where
    I2C: Write + WriteRead,
{
    /* New instance of Bmp180 device. */
    pub fn new(i2c: I2C) -> Self {
        Bmp180 {i2c,
                oss: 0,
                ac1: 0,
                ac2: 0,
                ac3: 0,
                ac4: 0,
                ac5: 0,
                ac6: 0,
                b1: 0,
                b2: 0,
                mb: 0,
                mc: 0,
                md: 0
        }
    }

    /* Drivers initialization. Must be called on device initialization.
       Load the calibration data, necessary for temperature and pressure
    calculation.*/
    pub fn begin(& mut self, mode: Mode) -> Result<(), Error>{
        let mut data = [0; 22];

        self.i2c.
            write_read(BMP180_ADDR, &[CAL_AC1_MSB], &mut data)
            .map_err(|_| Error::CommunicationError)?;

        self.oss = match mode {
            Mode::Bmp085UltraLowPower => 0,
            Mode::Bmp085Standard => 1,
            Mode::Bmp085HighRes => 2,
            Mode::Bmp085UltraHighRes => 3,
        };

        self.ac1 =  i16::from_be_bytes([data[0], data[1]]);
        self.ac2 = i16::from_be_bytes([data[2], data[3]]);
        self.ac3 = i16::from_be_bytes([data[4], data[5]]);
        self.ac4 = u16::from_be_bytes([data[6], data[7]]);
        self.ac5 = u16::from_be_bytes([data[8], data[9]]);
        self.ac6 = u16::from_be_bytes([data[10], data[11]]);
```

```
90          self.b1 = i16::from_be_bytes([data[12], data[13]]);
91          self.b2 = i16::from_be_bytes([data[14], data[15]]);
92          self.mb = i16::from_be_bytes([data[16], data[17]]);
93          self.mc = i16::from_be_bytes([data[18], data[19]]);
94          self.md = i16::from_be_bytes([data[20], data[21]]);
95
96          Ok(())
97      }
98
99      /* "Calculation of the variable b5 used for temperature and pressure
    . */
100     fn calculate_b5(&mut self, ut: i32) -> i32 {
101         let x1 = (ut - self.ac6 as i32) * (self.ac5 as i32) >> 15;
102         let x2 = (self.mc as i32) * 2048 / (x1 + (self.md as i32));
103         x1 + x2
104     }
105
106     /*************************************************************
107         Routines for temperature calculation.
108     *************************************************************/
109
110     /* Activates the sensor for temperature return and reads it in RAW
    format */
111     fn i2c_read_raw_temperature(&mut self) -> Result<i32, Error> {
112         let mut data = [0; 2];
113
114         // write the command to sample raw temperature.
115         self.i2c.
116             write(BMP180_ADDR, &[CONTROL_REGISTER, CMD_READ_TEMPERATURE
    ])
117             .map_err(|_| Error::CommunicationError)?;
118
119         arduino_hal::delay_ms(5);
120
121         // read the raw temperature data.
122         self.i2c
123             .write_read(BMP180_ADDR, &[MSB_DATA, LSB_DATA], &mut data)
124             .map_err(|_| Error::CommunicationError)?;
125
126         Ok((data[0] as i32) << 8 | data[1] as i32)
127     }
128
129     /* Determines the temperature in Celsius. */
130     pub fn read_temperature(& mut self) -> Result<f32, Error> {
131         let ut: i32;
132
133         match self.i2c_read_raw_temperature() {
```

```
134             Ok(value) => ut = value,
135             Err(_) => return Err(Error::CommunicationError),
136         }
137 
138         let b5 = self.calculate_b5(ut);
139 
140         let temperature = (b5 + 8) >> 4;
141 
142         //float division to obtain the temperature in Celsius with one
    decimal place.
143         Ok((temperature as f32) / 10.0)
144     }
145 
146     /************************************************************
147         Routines for pressure calculation.
148     ************************************************************/
149 
150     /* Reads the registers with pressure information in RAW format. */
151     fn i2c_read_raw_pressure(&mut self) -> Result<i32, Error> {
152         let mut data = [0; 2];
153 
154         let cmd = CMD_READ_PRESSURE_0 | ((self.oss as u8) << 6);
155 
156         // write the command to sample raw pressure.
157         self.i2c
158             .write(BMP180_ADDR, &[CONTROL_REGISTER, cmd])
159             .map_err(|_| Error::CommunicationError)?;
160 
161         arduino_hal::delay_ms(
162             match self.oss {
163                 0 => 5,
164                 1 => 8,
165                 2 => 14,
166                 3 => 26,
167                 _ => 0,
168             }
169         );
170 
171         // read the raw pressure data.
172         self.i2c
173             .write_read(BMP180_ADDR, &[MSB_DATA], &mut data)
174             .map_err(|_| Error::CommunicationError)?;
175 
176         let raw_pressure = (data[0] as i32) << 8 | data[1] as i32;
177 
178         // read additional precision bits for pressure calculation.
179         if self.oss > 0{
```

```
180             self.i2c
181                 .write_read(BMP180_ADDR, &[XLSB_DATA], &mut data)
182                 .map_err(|_| Error::CommunicationError)?;
183         }
184 
185         let raw_pressure = (raw_pressure << 8) | data[0] as i32;
186 
187         Ok((raw_pressure >> (8 - self.oss)) as i32)
188     }
189 
190 
191     /* Read raw data and performs calculations for pressure. */
192     pub fn read_pressure(&mut self) -> Result<i32, Error> {
193         let ut: i32;
194         let up: i32;
195 
196         match self.i2c_read_raw_pressure() {
197             Ok(value) => up = value,
198             Err(_) => return Err(Error::CommunicationError),
199         }
200 
201         match self.i2c_read_raw_temperature() {
202             Ok(value) => ut = value,
203             Err(_) => return Err(Error::CommunicationError),
204         }
205 
206         let b5 = self.calculate_b5(ut);
207         let b6 = b5 - 4000;
208         let x1 = ((self.b2 as i32) * (b6 * b6 >> 12)) >> 11;
209         let x2 = (self.ac2 as i32) * b6 >> 11;
210         let x3 = x1 + x2;
211         let b3 = (((self.ac1 as i32 * 4 + x3) << self.oss) + 2) >> 2;
212         let x1 = (self.ac3 as i32) * b6 >> 13;
213         let x2 = (self.b1 as i32) * ( b6 * b6 >> 12) >> 16;
214         let x3 = ((x1 + x2) + 2) >> 2;
215         let b4 = (self.ac4 as u32) * (x3 + 32768) as u32 >> 15;
216 
217         /* Caution! This operation results in a 64-bit integer. */
218         let b7 = (up - b3) as u64 * (50000 as u64 >> self.oss);
219 
220         let p;
221         if b7 < 0x80000000 {
222             p = (b7 * 2) / b4 as u64 ;
223         } else {
224             p = (b7 / b4 as u64) * 2;
225         }
226 
```

```
227         let x1 = (p >> 8) * (p >> 8);
228         let x1 = (x1 * 3038) >> 16;
229         let x2 = (-7357 * p as i64) >> 16;
230         let p = p as i64 + ((x1 as i64 + x2 + 3791) >> 4);
231         Ok(p as i32)
232     }
233 
234 }
```

Código A.1 – Device Driver

```
1  #![no_std]
2  #![no_main]
3  use arduino_hal as hal;
4  use ufmt_float::uFmt_f32;
5  use arduino_hal::prelude::*;
6  use avr_device;
7  
8  mod bmp180;
9  use bmp180::Bmp180;
10 use bmp180::Mode;
11 
12 #[panic_handler]
13 fn panic(info: &core::panic::PanicInfo) -> ! {
14     // disable interrupts - firmware has panicked so no ISRs should
    continue running
15     avr_device::interrupt::disable();
16 
17     // get the peripherals so we can access serial and the LED.
18     //
19     // SAFETY: Because main() already has references to the peripherals
    this is an unsafe
20     // operation - but because no other code can run after the panic
    handler was called,
21     // we know it is okay.
22     let dp = unsafe { arduino_hal::Peripherals::steal() };
23     let pins = arduino_hal::pins!(dp);
24     let mut serial = arduino_hal::default_serial!(dp, pins, 57600);
25 
26     // Print out panic location
27     ufmt::uwriteln!(&mut serial, "Firmware panic!\r").unwrap_infallible
    ();
28     if let Some(loc) = info.location() {
29         ufmt::uwriteln!(
30             &mut serial,
31             "  At {}:{}:{}\r",
32             loc.file(),
33             loc.line(),
```

```
            loc.column(),
        )
        .unwrap_infallible();
    }

    // Blink LED rapidly
    let mut led = pins.d13.into_output();
    loop {
        led.toggle();
        arduino_hal::delay_ms(100);
    }
}

#[arduino_hal::entry]
fn main() -> ! {
    let mode = Mode::Bmp085UltraHighRes;

    let dp = arduino_hal::Peripherals::take().unwrap();
    let pins = arduino_hal::pins!(dp);
    let mut serial = hal::default_serial!(dp, pins, 57600);

    let i2c = hal::I2c::new(
        dp.TWI,
        pins.d20.into_pull_up_input(),
        pins.d21.into_pull_up_input(),
        100000,
    );

    let mut bmp180 = Bmp180::new(i2c);

    if let Err(_e) = bmp180.begin(mode) {
        ufmt::uwriteln!(&mut serial, "Error communicating with the BMP180 sensor.").unwrap();
    }

    loop {
        match bmp180.read_temperature() {
            Ok(temperature) => {
                ufmt::uwriteln!(&mut serial, "Tempature: {}", uFmt_f32::Two(temperature)).unwrap();
            }
            Err(_e) => {
                ufmt::uwriteln!(&mut serial, "Error communicating with the BMP180 sensor.").unwrap();
            }
        }

```

```
78          match bmp180.read_pressure() {
79              Ok(pressure) => {
80                  ufmt::uwriteln!(&mut serial, "Pressure {}", pressure).
    unwrap();
81              }
82              Err(_e) => {
83                  ufmt::uwriteln!(&mut serial, "Error communicating with
    the BMP180 sensor.").unwrap();
84              }
85          }
86
87          arduino_hal::delay_ms(1000);
88      }
89  }
```

Código A.2 – Aplicação