UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE ENGENHARIA MECATRÔNICA

ANDRÉ LUIGI BONOTE

DESENVOLVIMENTO DIRIGIDO A MODELO PARA BOOTLOADER DE MICROCONTROLADOR

Joinville
2021

ANDRÉ LUIGI BONOTE

DESENVOLVIMENTO DIRIGIDO A MODELO PARA BOOTLOADER DE MICROCONTROLADOR

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de Bacharel em Engenharia Mecatrônica, no curso de Engenharia Mecatrônica, da Universidade Federal de Santa Catarina, Centro Tecnológico de Joinville.

Orientador: Prof. Dr. Gian Ricardo Berkenbrock

Joinville
2021

RESUMO

Em diversas áreas críticas, faz-se uso de sistemas embarcados, como automóveis e sistemas de automação industrial, entre outros. Neste contexto, é interessante usar ferramentas que facilitem e tornem mais confiáveis os métodos de desenvolvimento de tais sistemas. Sabendo que boa parte desses sistemas são baseados em microcontroladores, a compreensão do uso de ferramentas que atendam esse requisito no desenvolvimento dessa categoria de sistemas. Dessa maneira, este trabalho traz uma abordagem de desenvolvimento dirigido a modelo do bootloader para o microcontrolador ATmega328P, de modo a demonstrar a aplicabilidade do uso de ferramentas automatizadas de geração de código para tal finalidade. Buscando-se também um código-fonte cujos componentes podem ser reutilizados por outras aplicações. Baseado no código-fonte fornecido para o Arduino Duemilanove, desenvolveu-se um modelo UML no software Papyrus, aperfeiçoando-o até atingir o desacoplamento desejado. Os modelos são traduzidos em código pelo Papyrus Software Designer. Os componentes desacoplados são utilizados em uma aplicação desenvolvida para demonstrar sua usabilidade. Por fim, os códigos gerados são analisados estaticamente e melhorados conforme os resultados.

Palavras-chave: Bootloader. Microcontrolador. Análise estática. UML. Desenvolvimento dirigido a modelo.

ABSTRACT

Embedded systems are used in several critical areas, such as automobiles and industrial automation systems, among others. In this context, it is interesting to use tools that facilitate and make the development methods of such systems more reliable. According to the fact that these systems are largely based on microcontrollers, it is important to understand the use of tools that meet the requirement in code development for this systems. In that manner, this one brings a model-driven development approach to the ATmega328P microcontroller bootloader, in order to demonstrate the applicability of using automated code generation tools for this purpose. This was done by looking for a source code whose components can be reused by other applications. A UML model was generated in the Papyrus software, based on the source code provided for the Arduino Duemilanove, and was imporoved it until reaching the desired decoupling. The models were translated into code by Papyrus Software Designer and the decoupled components were used in an application developed to demonstrate their usability. Finally, the generated codes were subjected to a statical analysis and improved according to the results.

Keywords: Bootloader. Microcontroller. Static analysis. UML Model-driven development.

AGRADECIMENTOS

Dirijo meu breve agradecimento aos que me ajudaram a concluir este trabalho e principalmente a etapa que ele encerra.
Agradeço à minha mãe que me manteve e aos meus avós que me apoiaram nesta jornada.
À minha noiva que chegou no meio do caminho e me aturou até aqui.
Ao meu orientador que não desistiu neste sinuoso percurso.
E finalmente a intercessão de São José e da nossa terna mãe, Maria, para que eu nunca esquecesse de meu propósito.
Obrigado!

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE SIGLAS

CPU Central Process Unit

EEPROM Electrically-Erasable Programmable Read-Only Memory

GPIO General Purpose Input and Output

LSB Least Significant Bit

MCU Microcontrolador

MDA Model Driven Architecture

MDD Model Driven Development

MISO Master Input/Slave Output

MOSI Master output/Slave Input

MSB Most Significant Bit

NRWW No Read-While-Write

OMG Object Management Group

PIM Platform Independent Model

PSM Platform Specific Model

RAM Random Access Memory

RWW Read-While-Write

SCK Serial Clock

SPI Serial Peripheral Interface

SS Slave Select

UART Universal Asynchrounous Receiver and Transmiter

UML Unified Modeling Language

USART Universal Synchronous and Asynchrounous Receiver and Transmiter

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuits

XML Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Sistemas eletrônicos são uma constante no dia-a-dia da sociedade, presentes até mesmo em lugares pouco óbvios. Um bom exemplo disso é a escassez de diversos modelos de automóveis 0km no mercado mundial causados pela falta de processadores (GOMES, 2021). As tentativas dos governos em conter o espalhamento de uma epidemia causaram diversas distorções no mercado de semicondutores, o que gerou a escassez de produtos baseados em tais componentes, fazendo com que muitas pessoas passassem a perceber um bem que outrora era tido como totalmente mecânico, como um complexo sistema eletrônico (GOMES, 2021).

Dado que o controle dos sistemas em questão é feito em grande parte por microcontroladores cujo mercado está crescendo continuamente e em poucos anos deve atingir a receita de 20 bilhões de dólares (NETWORK, 2021), uma pergunta deve ser feita:

"O que acontece com esses produtos [...] quando milhões de unidades já foram entregues e um ’aprimoramento’ de software precisa ser feito? Cada unidade precisa ser devolvida ao fabricante sempre que o software é atualizado?"(BENINGO, 2015). A solução desse problema é o uso do bootloader. Um programa com a capacidade de gravar e inicializar uma nova aplicação, instalado na memória desses sistemas (BENINGO, 2015).

Assim, o trabalho se propõe a demonstrar a aplicabilidade da geração automática de códigos, a partir de modelos UML, para a criação de bootloaders de microcontroladores, demonstrando com isso alguns benefícios desse método de desenvolvimento.

1.1 Objetivo Geral

O objetivo deste trabalho é demonstrar a aplicabilidade do desenvolvimento dirigido a modelo para bootloaders de microcontroladores.

1.2 Objetivos Específicos

Os objetivos específicos são:

a) Estudar o comportamento do bootloader original do Arduino Duemilanove baseado no microcontrolador ATmega328P;

b) Criar modelos UML baseados no código original, propondo alterações que permitam um maior desacoplamento das estruturas do código;
c) Por meio de ferramentas de geração de código, gerar códigos funcionais a partir dos diversos modelos produzidos;
d) Utilizar as estruturas desacopladas do bootloader em uma aplicação para o Arduino;
e) Avaliar, com o uso de análises estáticas, a qualidade dos códigos gerados.

2 REVISÃO TEÓRICA

Este capítulo apresenta os principais conceitos utilizados no trabalho de modo a embasar a compreensão do que foi realizado. Iniciando com uma visão da forma de desenvolvimento escolhida para o trabalho, o desenvolvimento dirigido a modelo, e os conceitos que este utiliza: a automação de processos de transformação, modelos UML e geração de código a partir de modelos. Na sequência, o conceito de análise estática de código é apresentado, técnica escolhida para avaliar a qualidade do código gerado. Análise esta feita verificando sua conformidade com os padrões de programação, abordados no tópico seguinte. Então, introduz-se o microcontrolador, hardware que recebe os códigos gerados no trabalho, aprofundando-se nos conceitos da comunicação serial e do bootloader. Por fim, apresentam-se alguns trabalhos relacionados.

2.1 Desenvolvimento dirigido a modelo

O Desenvolvimento Dirigido a Modelos, do inglês Model Driven Development (MDD) é um processo de desenvolvimento de software guiado por modelos que descrevem o sistema (SOMMERVILLE, 2019; KLEPPE; WARMER; BAST, 2003). Esse processo busca aumentar a produtividade, portabilidade e interoperabilidade entre plataformas, além de facilitar a documentação e manutenção do software (KLEPPE; WARMER; BAST, 2003).

Tem o ciclo de vida dividido em:

a) Platform Independent Model (PIM): modelo de alta abstração que busca descrever apenas o comportamento de um software, independente da plataforma onde será implementado.
b) Platform Specific Model (PSM): o nível de abstração é menor que na etapa anterior, considerada a plataforma onde será implementado. Traz termos específicos da plataforma e só pode ser plenamente compreendido por quem tem conhecimento da plataforma.
c) Código: Trata-se da tradução dos modelos em um código funcional.

Figura 1 – Passos no processo de transformação dos modelos em código

Fonte: (KLEPPE; WARMER; BAST, 2003).

As etapas do desenvolvimento dirigido a modelo ocorrem em sequência e podem ser automatizadas por meio do uso de ferramentas de automação, como ilustrado na figura 1, sendo que um PIM pode gerar um ou mais PSMs e este, por fim, gera um código. Ter mais de um PSM para um mesmo PIM aumenta consideravelmente a complexidade do projeto (KLEPPE; WARMER; BAST, 2003).

A grande vantagem do uso do desenvolvimento dirigido a modelo vem da possibilidade de desenvolver sistemas com um alto nível de abstração e grande facilidade em portar sistemas e componentes entre plataformas (SOMMERVILLE, 2019). Porém, não é muito utilizada, especialmente por limitações nas ferramentas de automação dos processos de transformação e utilização de outros métodos de desenvolvimento, como o ágil (SOMMERVILLE, 2019).

2.1.1 Automação dos processos de transformação

Usando do Model Driven Architecture (MDA) as etapas de transformações de PIM para PSM e, por fim, para código, podem ser automatizadas. Partindo do modelo anterior, o modelo seguinte é gerado automaticamente (KLEPPE; WARMER; BAST, 2003). Especialmente na última etapa de transformação, do PSM para o código, o uso de automação é uma maneira eficiente de eliminar uma série de erros humanos ao transcrever a lógica do modelo para código. Ajudando, assim, a criar códigos mais confiáveis, além de muito mais rapidamente produzidos (USMAN; NADEEM; KIM, 2008).

A automatização da geração de código permite maior separação entre ele e o modelo, melhorando as capacidades de portabilidade e manutenção, além de proporcionar a rápida prototipagem e rápida visualização de resultados. Também possibilita a geração de testes automatizados e que um mesmo comportamento tenha o mesmo código gerado (RUMPE, 2017).

Porém, essas técnicas de automação na geração de código não são largamente adotadas. A principal barreira para a total automação das etapas de transformação, ilustradas na figura 1 é a indisponibilidade de ferramentas para realizar a transformação de maneira satisfatória, pois elas ainda requerem muita intervenção com códigos

escritos manualmente. (KLEPPE; WARMER; BAST, 2003).

2.1.2 UML

Unified modeling language (UML) é um sistema de elementos gráficos que formam uma linguagem capaz descrever e modelar o comportamento de softwares (FOWLER, 2003). Criada pela Object Management Group (OMG) com o objetivo de fornecer um ferramental para análise, design e implementação de sistemas, principalmente os baseados em software (OMG, 2017).

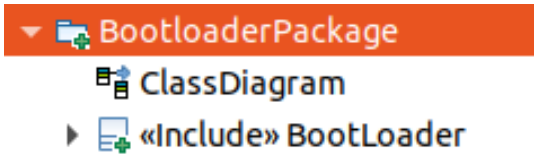
Boa parte do sucesso da linguagem UML e sua ampla adoção vem do fato de ter sua especificação não proprietária, extensível e suportada por diversas ferramentas, destacando-se, entre outras linguagens de modelagem gráfica que também permitem grande abstração (FOWLER, 2003; BHULLAR; CHHABRA; VERMA, 2016).

Atualmente, na segunda versão, provém uma ferramenta de modelagem visual de objetos interoperáveis adotada na indústria (OMG, 2017). Dentre os elementos fornecidos pela UML, temos o diagrama de classe, que como o nome sugere, descreve as classes do programa e também suas relações estáticas. Mas não são apenas classes que podem ser representadas. Ele pode ser utilizado para a descrição de diversos modelos estáticos que representem os mais variados conceitos (FOWLER, 2003).

Os diagramas de classe são compostos pelos seguintes elementos principais:

Namespace: Representa um local onde as classes estão organizadas, o diretório em que ela se encontra no projeto (OMG, 2017).

Figura 2 – Namespace representado graficamente no software Papyrus



Fonte: o Autor (2021).

Na figura 2 observa-se um namespace chamado BootloaderPackage contendo um diagrama de classe chamado ClassDiagram e uma classe chamada BootLoader.

Classe: A classe representa um elemento básico do modelo. É descrita pelos campos presentes no quadro 1 e na figura **??** (OMG, 2017):

Quadro 1 – Campos da classe UML

Nome	Nome do elemento que esta classe representa.
Rótulo	Uma forma de referenciar a classe sem ser seu nome. Funciona como um apelido.
Nome qualificado	Traz o nome da classe e o namespace dela da seguinte forma: Namespace::Nome. Utilizado para diferenciar a classe se outras possíveis classes com o mesmo nome em outros pacotes do projeto.
É ativa	Representa de forma binária se uma classe tem todas as suas instâncias como objetos capazes de realizar atividades (caso verdadeiro) ou se apenas são passivos, manipulados por outros objetos (caso falso).
É abstrata	Representa de forma binária se ela pode ser instanciada por si (caso falso) ou se deve ser implementada por uma herança (caso verdadeiro).
Visibilidade	será tratada a seguir nesse capítulo.
Atributos	Serão tratados a seguir nesse capítulo.
Operações	Serão tratados a seguir nesse capítulo.
Recepção	Representa a entrada de um sinal ao qual a classe irá reagir ao receber. Também possuem uma representação complexa como atributos e operações, porém não faz parte do escopo de interesse deste trabalho.
Estereótipo	Permite a criação de um novo tipo de elemento baseado na classe. Ampliando assim o vocabulário da UML.

Fonte: o Autor (2021).

Atributos: Os atributos são membros estruturais que armazenam dados no elemento e comumente representam as variáveis de uma classe ou então o fim de uma relação. Em representações mais abstratas podem simbolizar características (mutáveis ou não) do elemento (FOWLER, 2003).

Os atributos contêm elementos que os descrevem internamente, assim como as classes. Esses elementos estão descritos no quadro 2 e na figura **??** (OMG, 2017):

Quadro 2 – Campos do atributo UML

Nome	Nome do elemento que esse atributo representa.
Rótulo	Uma forma de referenciar o atributo sem ser seu nome, funciona como um apelido.
É derivado	Representa de forma binária se o valor do atributo pode (caso falso) ou não (caso verdadeiro) ser obtido a partir do valor de outro atributo da classe.
É derivado de união	O mesmo do anterior, porém o valor pode ser obtido a partir de um conjunto de outros elementos.
É ordenado	Representa de forma binária se os valores do atributo são (caso verdadeiro) ou não (caso falso) ordenados. Válido apenas para elementos com multiplicidade maior que 1.
É apenas leitura	Representa de forma binária se o valor do atributo pode ser alterado (caso falso) ou apenas consultado (caso verdadeiro).
É estático	Representa de forma binária se o valor do atributo é persistente (caso falso) ou se é eliminado ao sair do seu escopo (caso verdadeiro).
É único	Representa de forma binária se o atributo pode (caso falso) ou não (caso verdadeiro) ser duplicado.
Tipo	O tipo do elemento que aquele atributo representa. Aparece após o nome do atributo.
Multiplicidade	Será tratada em sequência no capítulo.
Visibilidade	Será tratada em sequência no capítulo.
Valor padrão	O valor que o atributo terá ao ser instanciado.
Subconjunto	Caso o atributo seja um subconjunto de outros atributos, é onde eles são referenciados.
Propriedade redefinida	Caso o atributo redefina outros atributos, é onde eles são referenciados.
Estereótipo	Permite a criação de um novo tipo de elemento baseado no atributo. Ampliando assim o vocabulário do UML.

Fonte: o Autor (2021).

Operações: Se os atributos representam as características de um elemento, as operações representam suas ações, que podem ser as funções de um código (FOWLER, 2003). São descritas pelos elementos mostrados no quadro 3 e na figura **??** (OMG, 2017):

Quadro 3 – Campos da operação UML

Nome	Nome da operação.
Rótulo	Uma forma de referenciar a operação sem ser seu nome. Funciona como um apelido.
É abstrato	Representa de forma binária se ela pode ser utilizada diretamente (caso falso) ou se deve ser implementada por uma herança (caso verdadeiro).
É Consulta	Representa de forma binária se a operação altera (caso falso) ou não (caso verdadeiro) o estado de qualquer instância de qualquer elemento.
É estático	Representa de forma binária se a operação é presente no escopo da classe (caso falso) ou se não é limitada ao escopo da classe (caso verdadeiro).
Condição de guarda	Condição que precisa ser cumprida para garantir o retorno correto da operação.
Visibilidade	Será tratada em sequência no capítulo.
Concorrência	Define se a operação é sequencial, guardada ou concorrente.
Método	É a representação do funcionamento do método. Pode ser um diagrama UML, como um diagrama de atividades, ou então um comportamento opaco.
Parâmetros	Parâmetros que o método utiliza. Cada parâmetro possui um elemento idêntico aos atributos além da direção que define se o método recebe (in), recebe e retorna (input) ou retorna (out) o parâmetro.
Precondições	Condições exigidas para a execução da operação.
Pós condições	Condições que serão verdadeiras após a execução da operação.
Exceção levantada	Exceção invocada quando uma operação é executada sem os requisitos de pré condições ou condições de guarda serem cumpridos.

Fonte: o Autor (2021).

Comportamento opaco: O comportamento opaco é uma maneira de inserir um trecho de código na linguagem de programação do código que o diagrama representado será escrito. É utilizado em substituição a um diagrama que represente o comportamento daquele código composto pelos elementos presentes no quadro 4 e na figura **??** (OMG, 2017):

Quadro 4 – Campos do comportamento opaco

Nome	Nome do comportamento opaco.
Rótulo	Uma forma de referenciar o comportamento opaco sem ser seu nome. Funciona como um apelido.
Linguagem	A linguagem de programação em que se descreveu o comportamento opaco.
Corpo	Código.
É abstrato	Representa de forma binária se ela pode ser utilizada diretamente (caso falso) ou se deve ser implementada por uma herança (caso verdadeiro).
É ativa	Representa de forma binária se um comportamento opaco tem todas as suas instâncias como objetos capazes de realizar atividades (caso verdadeiro) ou se apenas são passivos, manipulados por outros objetos (caso falso).
É recursivo	Representa de forma binária se um comportamento opaco é (caso verdadeiro) ou não (caso falso) recursivo.
Visibilidade	Será tratada em sequência no capítulo.
Especificação	Operação ou outro elemento UML ao qual o comportamento opaco pertence
Caso de uso	Diagrama de caso de uso que representa o comportamento opaco (caso exista).

Fonte: Autor.

Visibilidade: Define um elemento entre as classificações de visibilidade (OMG, 2017):

Pública Pode ser acessado por qualquer elemento.

Privada Pode ser acessado pelo elemento que o detém ou por elementos que têm certas relações com aquele que o detém.

Protegida Similar ao privado, porém também pode ser acessado por herdeiros do membro que o detém.

Associação: Setas sólidas saindo de uma classe de origem para uma classe alvo, são formas de representar as diversas interações entre os elementos do modelo (FOWLER, 2003). De modo geral, são muito próximas a atributos, porém usadas para representar itens mais complexos, como outras classes (FOWLER, 2003).

Composição: Uma categoria de associação onde a classe de origem possui a classe alvo, representada pela adição de um losango preenchido à base da seta (FOWLER, 2003). Ela determina que uma instância da classe alvo pode ser possuída apenas pela classe de origem. Então, no caso de uma classe alvo com várias composições, cada classe de origem da composição deverá ter n (de acordo com a multiplicidade) instâncias da classe alvo, e essas instâncias pertencem apenas às suas respectivas classes de origem (FOWLER, 2003).

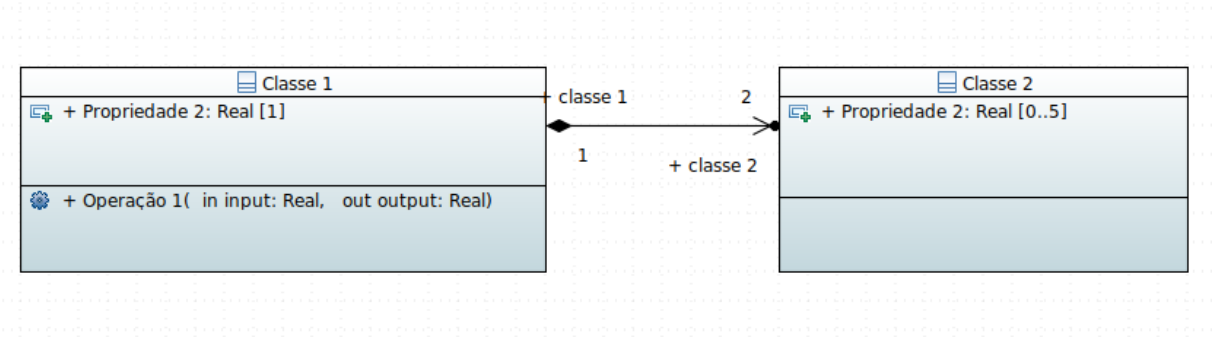
Multiplicidade: Representa quantidades de cada elemento, sejam eles propriedades,

operações ou classes (OMG, 2017).

[n] Representa um número fixo, n, deste item.

[n..m] Representa um número variável, entre m e n, deste item.

Figura 3 – Exemplo de diagrama de classes



Fonte: o Autor (2021).

Na figura 3 é possível observar um diagrama de classes com duas classes. A Classe 1 possui um atributo do tipo real, chamado Propriedade 1, de multiplicidade 1 e uma operação, chamada operação 1 que recebe uma entrada, input, do tipo real e retorna o output, do tipo real, e tem multiplicidade 1. A Classe 2 possui apenas uma propriedade, chamada Propriedade 2, do tipo real, que tem a multiplicidade de 0 até 5. Também se pode observar a existência de uma associação do tipo composição, que representa que a Classe 1 possui duas instâncias da Classe 2.

Para a transformação do PSM descrito em UML em código são usadas ferramentas automatizadas de geração de código, conforme descrito a seguir.

2.1.3 Geração de Código

As ferramentas de geração de código fazem parte do processo de automatização, especificamente na etapa de gerar a partir do PSM um código funcional. Isso pode ser feito extraindo informações de um modelo por consultas e convertendo-as em fragmentos de texto baseados em regras previamente estabelecidas (OMG, 2008).

Os processos de tradução de modelos UML para código variam conforme o tipo do modelo UML que será transformado (BHULLAR; CHHABRA; VERMA, 2016). De maneira geral, o método usado pelas ferramentas para a tradução é a busca por alvos que restringem os elementos do modelo. A eficácia dos tradutores vem da qualidade das buscas que fazem, e também dos algoritmos que interpretam como transcrever cada elemento no código (TAN; YANG; XIE, 2010).

Para compreender, de maneira simplificada, como os algoritmos de busca funcionam, é possível imaginar os modelos como tendo várias camadas de elementos que contêm outros elementos dentro de si. Os algoritmos então navegam pelos elementos do modelo, e sucessivamente pelos elementos contidos pela camada acima

deles, até ter um completo mapa do modelo (TAN; YANG; XIE, 2010).

Para realizar essa navegação eles são alimentados com o aquivo do tipo Extensible Markup Language (XML), o qual pode ser compreendido como a representação em texto de um modelo UML, gerado pelas ferramentas de criação de diagramas UML.

Enquanto navegam, os algoritmos buscam padrões no XML, fazendo comparações para saber como caracterizar o modelo e como encaixar cada elemento encontrado no código (TAN; YANG; XIE, 2010).

Para exemplificar pode-se observar o trecho de XML na figura 4, parte do XML do diagrama apresentado na figura 3.

Figura 4 – Exemplo de XML

```
<packagedElement xmi:type="uml:Class" xmi:id="_OLeewPWHEeur0I2kesmT2g" name="Classe 1" visibility="public">
  <ownedAttribute xmi:type="uml:Property" xmi:id="_PwvzcPWHEeur0I2kesmT2g" name="Propriedade 1">
    <type xmi:type="uml:PrimitiveType" href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#Real"/>
  </ownedAttribute>
```

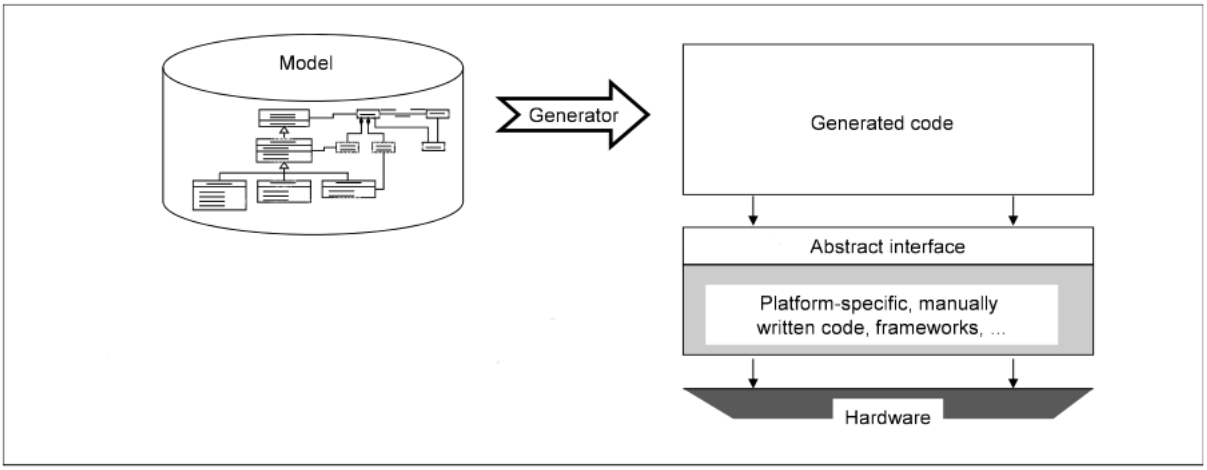
Fonte: o Autor (2021).

Ao chegar nesse trecho do XML, um tradutor poderia extrair a informação necessária para gerar uma classe pública chamada Classe 1 contendo um atributo chamado Propriedade 1 do tipo real.

De modo geral, a geração de código baseada em diagramas de classe é satisfatória, porém os comportamentos dos métodos presentes nas classes precisa ser feito de maneira manual (RUMPE, 2017). Como, por exemplo, pela inserção de comportamentos opacos nos diagramas UML.

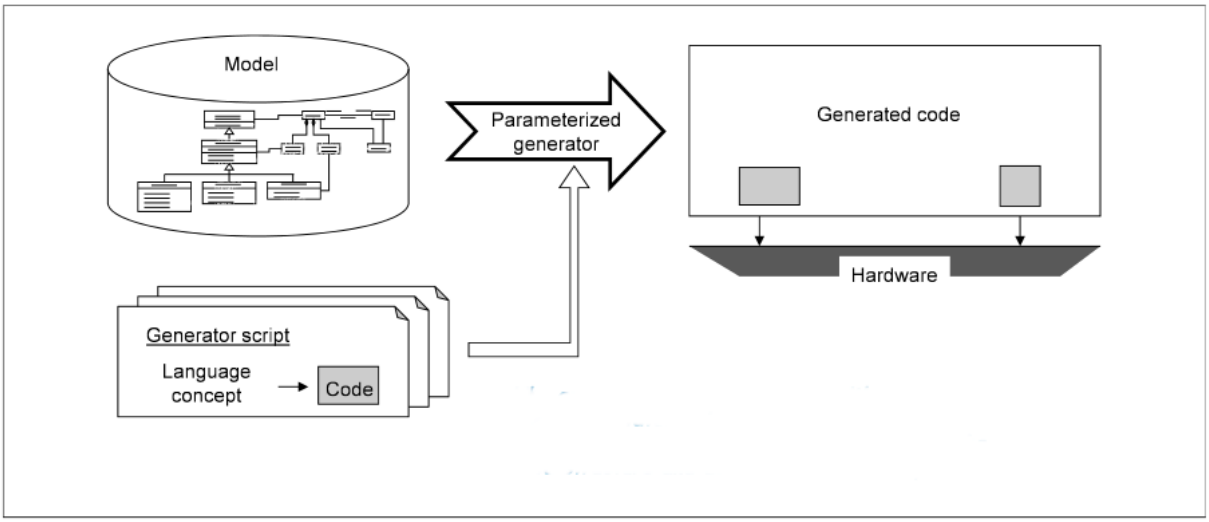
Os mecanismos de geração de código diferem conforme a plataforma para a qual o sistema é projetado. Duas abordagens existem na geração de código: a geração de interfaces abstratas, mostrada na figura 5, e o de geração parametrizado, mostrado na figura 6. Ambas exigem intervenção manual, porém em aspectos diferentes. A primeira exige que partes de código sejam adicionadas aos gerados pelo modelo, de modo a determinar comportamentos, especificações de hardware e outros. Na segunda, a intervenção ocorre no gerador onde um código é escrito ou parâmetros são dados para determinar a interpretação que o gerador dará para cada elemento do modelo. Assim, é possível obter o código gerado sem intervir nele após a geração (RUMPE, 2017).

Figura 5 – Geração de código para interfaces abstratas



Fonte: adaptado de (RUMPE, 2017).

Figura 6 – Código de geração parametrizado



Fonte: adaptado de (RUMPE, 2017).

Para analisar a qualidade dos códigos gerados, é possível utilizar uma série de métodos, entre eles a análise estática.

2.2 Análise Estática do Código

A análise das características de um código é dita estática quando não ocorre em seu tempo de execução. Ou seja, apenas o texto do código é analisado em busca de erros, inconsistências e inseguranças no código (NOVIKOV et al., 2018).

Esses testes podem detectar erros nos estágios iniciais da produção de um código, permitindo que as correções sejam feitas, evitando atualizações que representariam um enorme custo, especialmente para sistemas embarcados (NOVIKOV et al., 2018). Pois, detectam erros antes dos testes comumente empregados na indústria

para sistemas embarcados, que em sua maioria são realizados tentando reproduzir a utilização do sistema dentro e fora do seu modo projetado de utilização (REINBACHER et al., 2009).

As ferramentas de análise estática aumentam a segurança de sistemas embarcados ao procurar por revelar defeitos no código-fonte, tais como: divisões cujo denominador pode assumir o valor 0; memórias alocadas que não são liberadas em seu escopo, causando vazamentos de memória; ponteiros apontando para o endereço de memória 0; variáveis não inicializadas; tentativas de acesso a posições inexistentes de um buffer (CHELF; EBERT, 2009); dentre muitos outros que podem ser definidos por padrões de programação.

Por atuarem diretamente no código-fonte, essas ferramentas são rápidas, de baixo custo e simples por não precisarem de nenhuma sintaxe adicional. Porém, podem não cobrir todas as classes de falhas que podem ser encontradas com o uso de outros métodos de teste e verificação. Isso além de poderem apresentar alertas falsos, o que exige um trabalho extra no desenvolvimento para reduzir esses alertas (SOMMERVILLE, 2019).

2.2.1 Padrões de Programação

O uso de padrões no desenvolvimento de software busca atingir maior qualidade de software, de uma maneira viável a serem aplicados (SOMMERVILLE, 2019). Também permitem que conhecimentos baseados em tentativa e erro sejam facilmente aplicados em novos casos, ajuda a definir a qualidade desejada para um código e garantem a continuidade do projeto (SOMMERVILLE, 2019).

Os padrões abrangem diversos aspectos. Tem-se os de arquitetura os quais trazem formas de organizar um sistema de maneiras bem sucedidas, baseadas em sistemas já testados. Há padrões de projeto que trazem soluções comprovadas para problemas comuns. E também de software, os quais são uma seleção de diretrizes que devem ser adotadas no desenvolvimento de um software, comumente baseados em normas nacionais e internacionais (SOMMERVILLE, 2019).

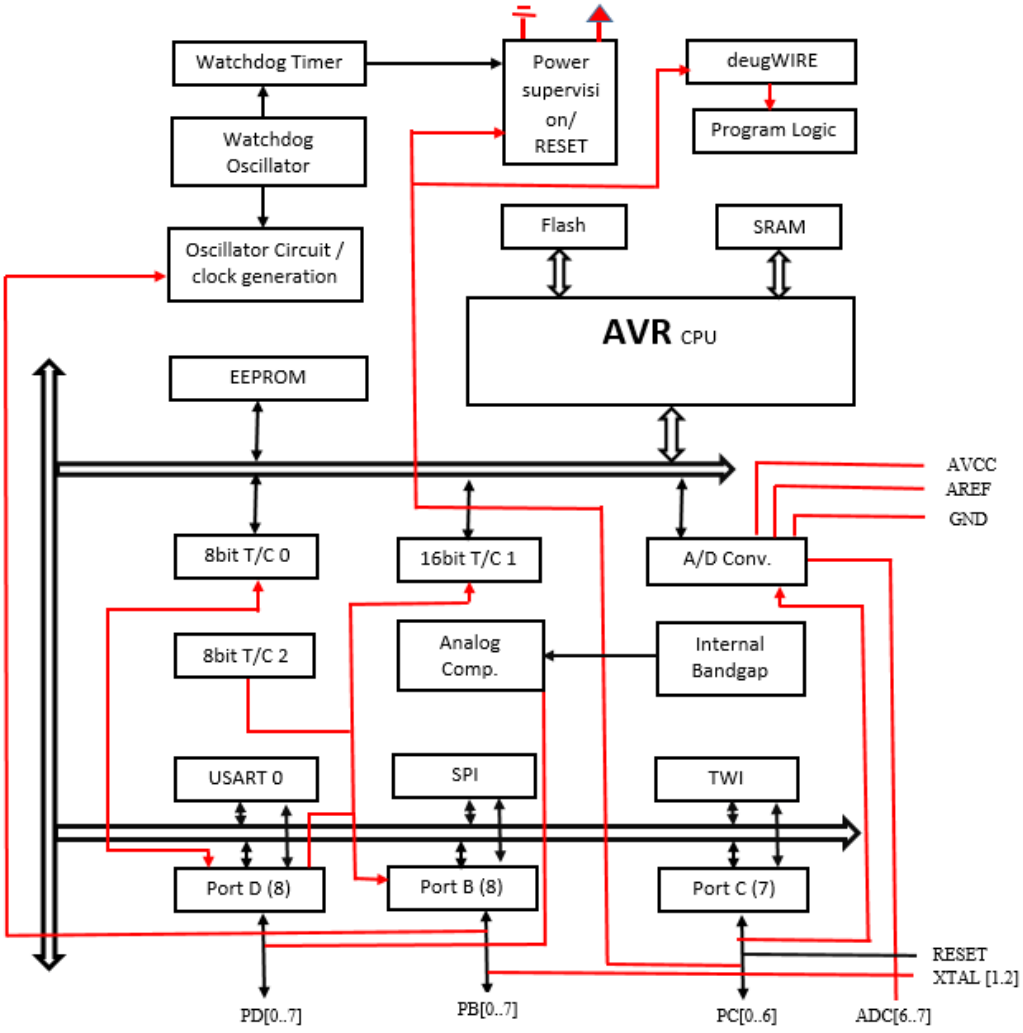
Ao teste estático aplicam-se principalmente os padrões de codificação, que buscam definir a maneira correta de se utilizar uma determinada linguagem de programação (CPPCHECK TEAM, 2021; SOMMERVILLE, 2019). Para isso, é fornecida para a ferramenta de testes uma série de regras aplicáveis à linguagem de programação utilizada, que vão evitar comportamentos indefinidos, padrões perigosos de código e falhas na construção do código (CPPCHECK TEAM, 2021).

Os padrões de programação também podem ser utilizadas para aplicações específicas, como no caso de sistemas embarcados, onde situações que não seriam problemáticas em diversas aplicações passam a apresentar grandes riscos. A maioria desses sistemas são baseados em microcontroladores.

2.3 Microcontrolador

Microcontroladores, ou MCUs, são dispositivos eletrônicos programáveis com uma interface de entradas e saídas de informação, utilizadas para interagir com outros dispositivos. Capazes de executar um programa por vez, lendo os estados de suas entradas e manipulando os estados de suas saídas durante a execução de tais programas (DEEPAK, 2016).

Figura 7 – Arquitetura de um microcontrolador ATmega328



Fonte: (PICKERILL, 2008).

Compostos de uma unidade central de processamento, do inglês Central Process Unit (CPU), acrescida de alguns periféricos, como ilustrado na figura 7, que servem como acessórios com funções específicas. Analisando mais detalhadamente é possível separar alguns dos principais componentes de um microcontrolador (WILLIAMS, 2014):

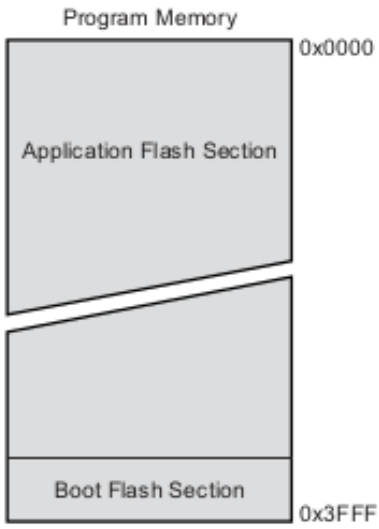
CPU: Central Process Unit, pode asumir duas arquiteturas: arquitetura do tipo reduced

instruction set computer (RISC), com um conjunto pequeno de operações pré definidas, consegue realizá-las de maneira mais rápida, em detrimento do tamanho de código. Arquitetura do tipo complex instruction set computer (CISC), que traz uma redução na complexidade dos compiladores e execução de tarefas complexas com maior velocidade, devido à sua construção com capacidade de operar instruções mais complexas (HEATH, 1995).

A CPU é o componente responsável por acessar todas as memórias, executar cálculos, controlar os periféricos e lidar com as interrupções, garantindo o funcionamento da aplicação (ATMEL CORPORATION, 2015).

Memória flash: armazenamento não volátil onde é alocado o código que o microcontrolador deve executar (WILLIAMS, 2014).

Figura 8 – Mapa de memória flash do ATmega328P



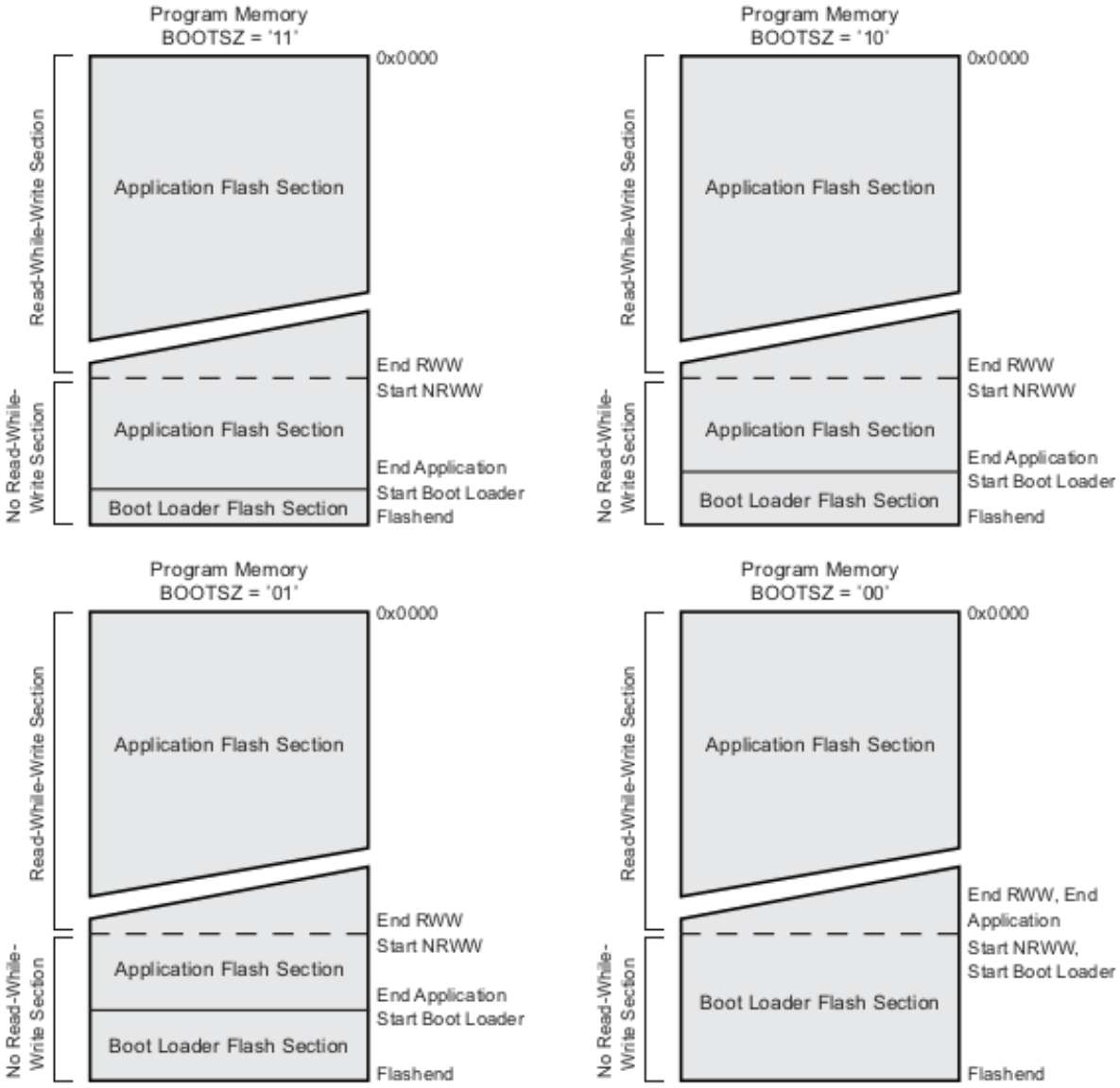
Fonte: (ATMEL CORPORATION, 2015).

Em muitos MCUs, na memória flash também é alocado um espaço exclusivo para o bootloader, figura 8. Essa secção de memória consegue acessar a si mesma e também a secção da aplicação (ATMEL CORPORATION, 2015), onde está armazenada a aplicação.

Outra divisão existente na memória flash é entre secção Read-While-Write (RWW), e No Read-While-Write (NRWW). Quando a primeira é apagada ou escrita, a segunda pode ser lida durante a operação de boot, já a segunda, quando é apagada ou escrita, não permite outra operação, pois a CPU fica suspensa (ATMEL CORPORATION, 2015).

Conforme ilustrado na figura 9, o bootloader sempre estará na secção NRWW, podendo ou não ocupar ela integralmente. Desse modo, o bootloader pode ser lido durante a gravação da aplicação (ATMEL CORPORATION, 2015).

Figura 9 – Secções de memória



Fonte: (ATMEL CORPORATION, 2015).

RAM: Random Access Memory, realiza um armazenamento volátil de dados, onde serão guardados dados usados apenas em tempo de execução (WILLIAMS, 2014).

EEPROM: Electrically-Erasable Programmable Read-Only Memory, armazenamento mais lento e não volátil. Pode ser usado como memória externa, salvando dados processados pelo microcontrolador em um local separado da memória de programa (WILLIAMS, 2014).

Uma das formas de memórias externas mais flexíveis, podendo manter os dados armazenados por mais de 200 anos sem nenhum suprimento de energia e trabalha com 5V de alimentação, tensão nominal da maioria dos MCUs (WILLIAMS, 2014).

Clock: ou relógio, oscilador que cria um pulso elétrico periódico que determina a passagem de tempo para o microcontrolador (WILLIAMS, 2014).

Saídas: pinos cuja tensão elétrica pode ser controlada pelo microcontrolador, representando uma saída de informação. De modo geral são digitais, ou seja, possuem apenas 2 valores pré definidos de tensão possíveis. (WILLIAMS, 2014).

Entradas: pinos cuja tensão pode ser lida pelo microcontrolador, representando uma entrada de informação. Essa leitura pode ser digital ou analógica (WILLIAMS, 2014).

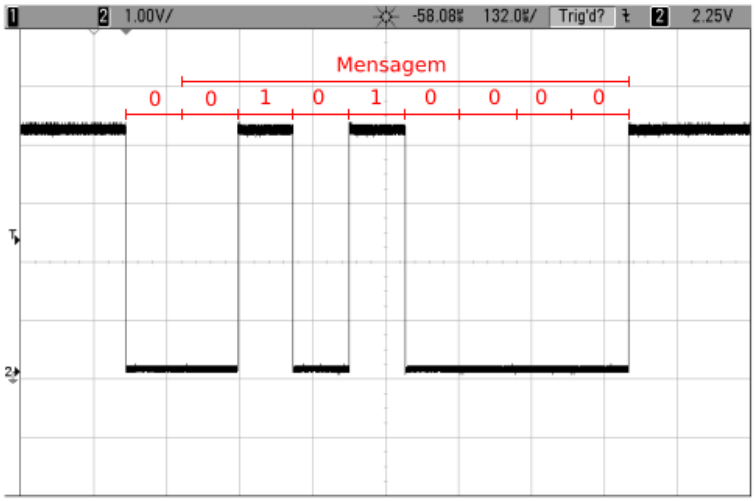
Comunicação serial: os microcontroladores possuem periféricos que implementam protocolos de comunicação serial, eles permitem que dados sejam trocados de modo mais eficiente com outros equipamentos que têm os mesmos protocolos de comunicação (WILLIAMS, 2014). Detalhada no tópico seguinte.

Interrupções: periférico que permite interrupção da execução de um programa em um determinado estado para que outro programa seja executado. Ele faz isso no momento de ativação de algum gatilho (como a mudança de estado de uma entrada digital), e retorna a execução do programa principal no mesmo estado em que foi interrompida anteriormente (WILLIAMS, 2014).

2.3.1 Comunicação serial

A comunicação serial é uma maneira dos microcontroladores trocarem informações mais complexas entre si e com outros equipamentos (computadores, sensores, telas, etc.) (WILLIAMS, 2014). Essa categoria de comunicação ocorre por de uma sequência de sinais elétricos de tensão, representando bits, trocada entre os equipamentos que estão participando da comunicação. Um protocolo de comunicação serial é responsável por garantir a correta codificação e decodificação dessa informação (WILLIAMS, 2014).

Figura 10 – Número 10 enviado via comunicação serial

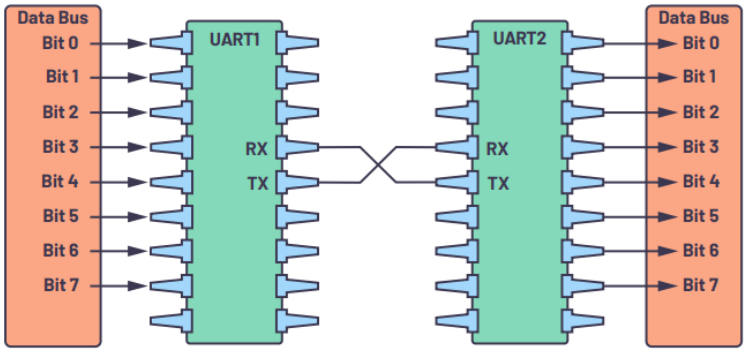


Fonte: adaptado de (WILLIAMS, 2014)

Na figura 10, pode ser observado por um sinal de tensão lido por um osciloscópio o valor decimal 10, 1010 em binário, enviado via comunicação serial. Na imagem tem-se o sinal alto, seguido de um bit 0 que representa o início da comunicação serial no protocolo utilizado. Então são contados 8 bits, que é o tamanho da mensagem, que contém o número dez: 00001010. Por fim, o sinal volta para alto significando o fim da mensagem. A transmissão começa pelo bit menos significativo (Least Significant Bit, LSB), e termina com o mais significativo (Least Significant Bit, LSB), portanto aparece espelhada na imagem (WILLIAMS, 2014).

Em sistemas micro controlados destaca-se o Universal Asynchrounous Receiver and Transmiter (UART), hardware que permite um protocolo assíncrono de comunicação de dispositivos eletrônicos com a arquitetura emissor-receptor (PEÑA; LEGASPI, 2020). Nele o emissor lê uma memória, de forma paralela e a envia bit a bit para o receptor, o qual disponibilizará esses bits, de forma paralela, em uma memória. Os dispositivos conseguem fazer tanto o papel de emissor quanto de receptor (PEÑA; LEGASPI, 2020), a ligação de dois equipamentos via UART pode ser melhor visualizada na figura 11.

Figura 11 – Comunicação UART



Fonte: (PEÑA; LEGASPI, 2020).

Uma variação comum é o Universal Synchronous and Asynchrounous Receiver and Transmiter (USART), derivado do UART pela adição de um pino de relógio a cada componente, o que permite que a transmissão de dados seja feita de maneira síncrona (WILLIAMS, 2014), graças ao sinal de relógio que pode ser fornecido pelo mestre ou pelo escravo (ATMEL CORPORATION, 2015).

Com esse hardware é possível implementar protocolos de comunicação mais complexos, como o Serial Peripheral Interface (SPI), que é um protocolo de comunicação serial, síncrono, full-duplex (ambos os participantes podem enviar e receber dados de forma simultânea), empregado entre dispositivos eletrônicos (Motorola Inc., 2004).

A comunicação serial é o que permite ao bootloader receber uma nova aplicação e gravá-la na memória.

2.3.2 Bootloader

Bootloaders são programas cuja finalidade primeira é criar um mapa de memória do dispositivo onde se encontra e iniciar a aplicação (UZLU; SAYKOL, 2016). Como apresentado de maneira simplificada na figura 12, onde a sua única ação é de apontar a execução do programa para a posição de memória 0, onde está a aplicação.

Figura 12 – Exemplo de um bootloader básico

```
void (*app_start)(void) = 0x0000;

int main(void)
{
    app_start();
}
```

Fonte: o Autor (2021).

Nos microcontroladores, também é competência do bootloader realizar a gravação da aplicação na memória do microcontrolador diretamente de um computador, sem a necessidade de utilizar um equipamento externo. Para isso ele gerencia a comunicação com o computador por um procolo de comunicação serial (MISCHIE; PAZSITKA, 2019).

De forma geral, para MCUs são escritos buscando a redução em seu tamanho, para reduzir o seu impacto na memória, que costuma ser um recurso escasso. Além disso, não é comum ser prevista a alteração do bootloader, escrito e gravado na memória pelo próprio fabricante do microcontrolador, utilizando um equipamento dedicado (MISCHIE; PAZSITKA, 2019).

O bootloader é executado quando o micrcontrolador é ligado. Por alguns instantes espera receber comandos pela comunicação serial, de modo que possa realizar as ações para o qual foi escrito, como gravar informações na memória de programa ou apagar a memória de programa. Caso não receba nenhuma instrução, ele passa o controle da execução para a aplicação (STRICKLAND, 2016).

Os microcontroladores que possuem o bootloader contam com ao menos duas seções distintas da memória: uma para o bootloader e outra para a aplicação (BENINGO, 2015). Alocado no fim da memória de programa do microcontrolador, ele é automaticamente executado quando o MCU é ligado. Dependendo das instruções que recebe via comunicação serial, ele pode acessar e escrever na parte inicial da memória de programa, onde fica alocada a aplicação (WILLIAMS, 2014).

O bootloader precisa cumprir alguns requisitos para seu correto funcionamento. Os mais básicos são (BENINGO, 2015):

a) Habilidade de alternar entre o bootloader e a aplicação;
b) Possuir uma interface de comunicação;
c) Capacidade de manipular as memórias de modo a gravar de forma correta a aplicação no microcontrolador.

As diferenças do bootloader para uma aplicação é o fato de ele ser alocado em um espaço específico de memória que será executado ao ligar o microcontrolador e a sua capacidade de manipular a memória de programa. Porém, a forma de se escrever um bootloader é similar a de uma aplicação comum (BENINGO, 2015).

2.4 Trabalhos Relacionados

Nessa seção são apresentados alguns trabalhos relacionados à geração automática de código para sistemas embarcados, especialmente baseando-se em diagramas UML.

Moreira et al. (2010) apresenta uma proposta de geração automatizada de código para sistemas embarcados. Neste artigo é sugerida uma solução para a

geração de códigos em VHSIC Hardware Description Language (VHDL), algo que não é oferecido por ferramentas comerciais. Para isso ele propõe uma maneira de descrever o código em VHDL com diagramas UML de classes e de atividades. Por fim, apresenta-se um código VHDL funcional, aplicado a um sistema mecatrônico gerado automaticamente partindo dos modelos UML.

Krunić et al. (2013) traz uma perspectiva interessante sobre a geração de código-fonte de firmwares. No artigo, apresenta-se uma abordagem de geração de código baseada em módulos divididos conforme a linguagem, o tipo de arquivo a ser gerado e o comportamento de função. Por fim, ele aponta a vantagem do uso dessa abordagem em comparação a produção manual de código, pois ela apresenta menos erros e produz o código em menos tempo, o que deixa os desenvolvedores livres para abstrair problemas.

Outra abordagem interessante para o uso de geração de código para microcontroladores é exposta por Bai, Chng e Bhanu (2007). No artigo, é proposta a modelagem da arquitetura do microcontrolador, e não do código. Dessa forma é possível especificar todos os seus periféricos e gerar códigos de inicialização específicos para os periféricos contidos no microcontrolador.

Já Lennis e Aedo (2013) descreve a geração automática de um código em linguagem C, a partir de um modelo UML. Ele demonstra que, utilizando apenas programas de código aberto, é aplicável tal abordagem de produção de código para sistemas microcontrolados.

Abaixo a tabela 1 apresenta um comparativo desses trabalhos com este trabalho.

Tabela 1 – Comparação entre os trabalhos relacionados.

	Modelo de entrada	Técnica de geração	Linguagem gerada
Moreira et al. (2010)	UML	Parametrizada	VHDL
Krunić et al. (2013)	XML	Parametrizada	C e Assembly
Bai, Chng e Bhanu (2007)	XML	Parametrizada	C
Lennis e Aedo (2013)	UML	Parametrizada	C
Este trabalho	UML	Interfaces abstratas	C

Fonte: o Autor (2021).

3 MATERIAIS E MÉTODOS

Este capítulo apresenta as ferramentas e os métodos utilizados para a realização deste trabalho. Iniciando pelo hardware utilizado, seguido das técnicas de gravação na memória do microcontrolador, ferramenta de criação de modelos, geração automatizada de códigos e, por fim, as ferramentas de análise do código gerado.

3.1 Arduino

Arduino é um eco-sistema que fornece uma plataforma de prototipagem de código aberto (DUKISH, 2018), incluindo placas de fácil programação baseadas em microcontroladores da família ATmega, sendo genericamente chamadas pelo nome Arduino (ARDUINO, 2018b). Essas placas comumente trazem o microcontrolador ATmega328P. De arquitetura RISC, ele tem baixo consumo elétrico e capacidade de processamento de 8 bits (ATMEL CORPORATION, 2015). Por ser de código aberto, existe uma série de bootloaders disponíveis para essas placas. O próprio fabricante os fornece permitindo, assim, a sua utilização e modificação. Além disso, ele também oferece ferramentas que facilitam a gravação no microcontrolador.

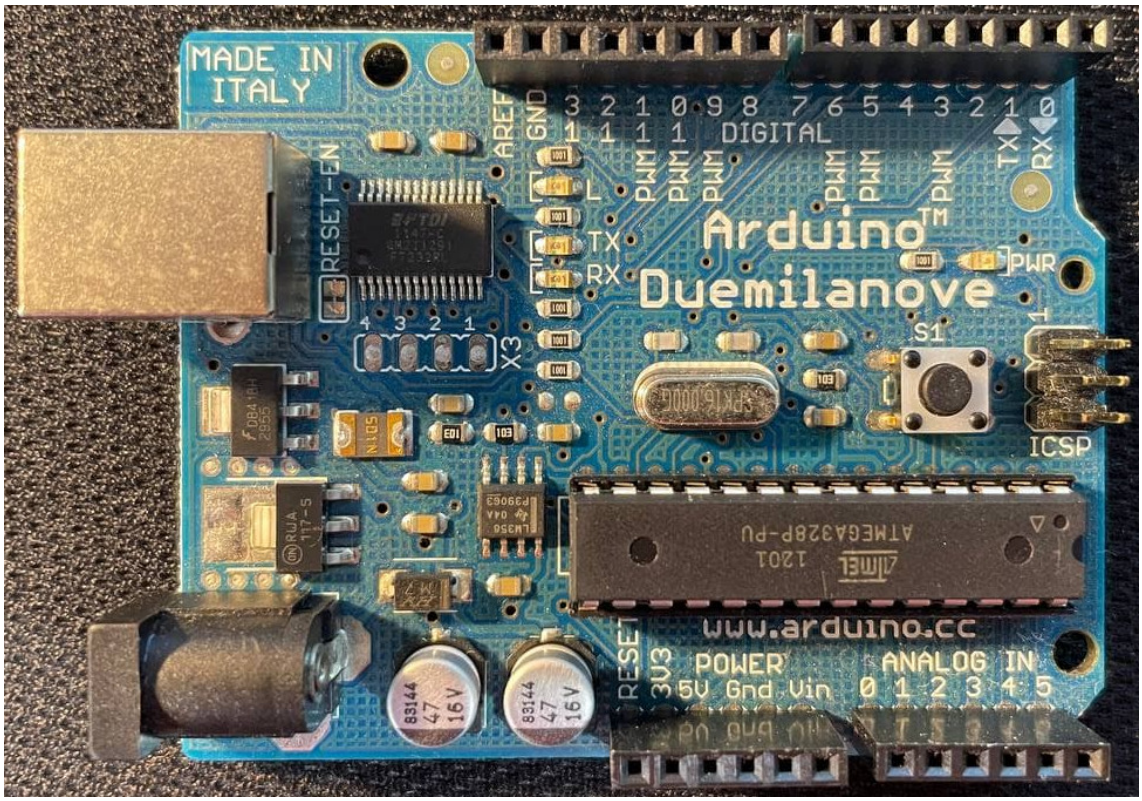
3.1.1 ATmega328P

O ATmega328P traz o suporte nativo para bootloader, com alocação flexível de memória. Com um mecanismo de RWW de auto programação, ele suporta o carregamento de códigos de programa sem o uso de periféricos, por meio do uso do bootloader. Esse carregamento não se limita ao programa principal, pois possibilita até mesmo alterar, regravar e apagar o próprio bootloader (ATMEL CORPORATION, 2015). Essa gravação é possível com o uso de uma interface de comunicação serial. São nativas do modelo as interfaces de comunicação serial: uma interface serial programável USART, interface SPI e uma interface compatível com o protocolo I2C (ATMEL CORPORATION, 2015). Dentre as placas de prototipagem Arduino as quais usam o ATmega328P como base destacam-se as versões Duamilanove e Uno.

3.1.2 Arduino Duemilanove e Uno

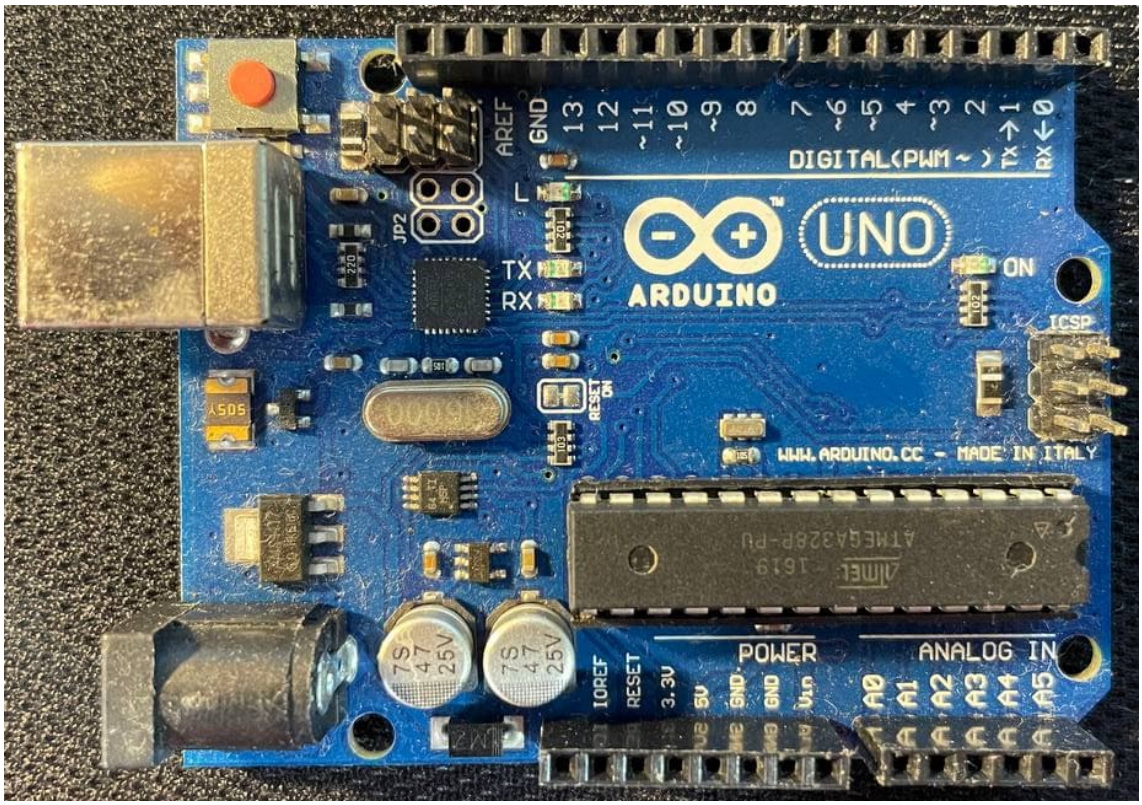
Ambas são placas de prototipagem baseadas no microcontrolador ATmega168 ou ATmega328P. Elas fornecem diversos periféricos para o microcontrolador como os periféricos de comunicação USB, conectores de 2,54 mm, alimentação de energia, cristal oscilador dentre outros (NASIR, 2018).

Figura 13 – Arduino Duemilanove



Fonte: o Autor (2021).

Figura 14 – Arduino Uno



Fonte: o Autor (2021).

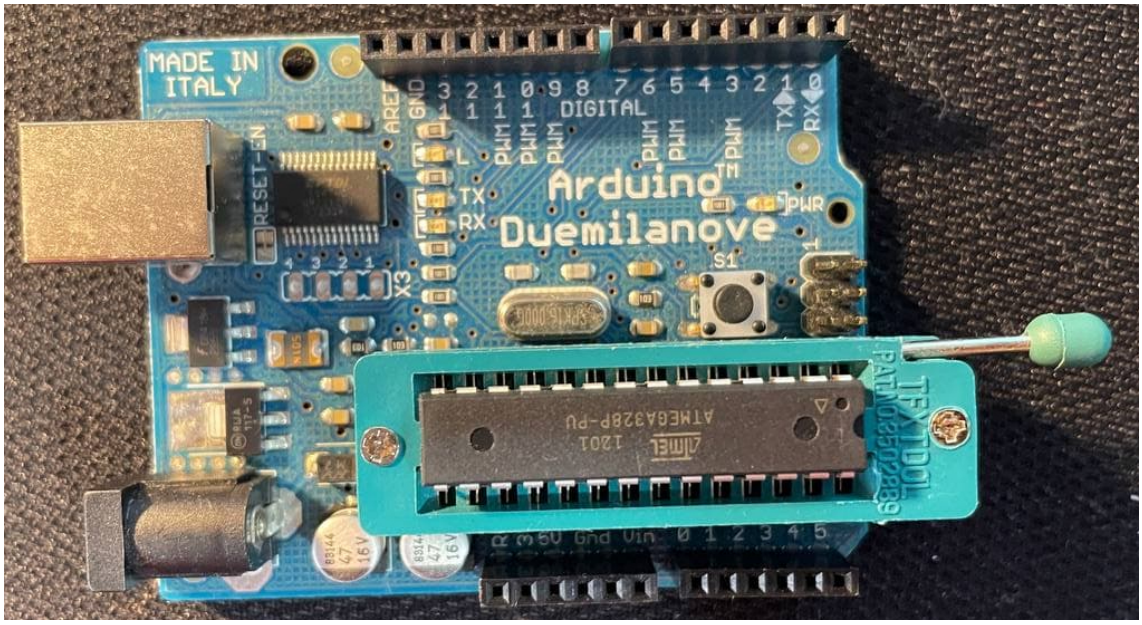
A principal diferença construtiva entre elas é o modo com o qual se comunicam via USB. No projeto da Duemilanove (figura 13), é utilizado o chip FT232RL que faz a interface entre a USB e a comunicação UART do ATmega328p (ARDUINO, 2008; FUTURE TECHNOLOGY DEVICES INTERNATIONAL LIMITED, 2020). No Uno (figura 14) essa interface é realizada por um segundo microcontrolador, o ATMEGA8U2 (ARDUINO, 2010).

A escolha da placa Arduino Duemilanove deu-se por ter um bootloader com o código-fonte fornecido pela fabricante, escrito majoritariamente na linguagem C e pela disponibilidade de duas unidades para a realização desse trabalho. Isto garante a continuidade desse trabalho caso a primeira placa fosse danificada durante sua realização dos experimentos. Porém, esta é uma placa descontinuada pela fabricante, substituída no mercado pela placa Arduino Uno. Portanto, utilizpu-se a placa Duemilanove para receber os bootloaders gerados. Enquanto se adquiriu uma placa Uno para ser utilizada como interface de gravação para os bootloaders gerados. As duas possuem o microtontrolador ATmega328P.

3.2 Gravação do bootloader

Para a gravação do bootloader adicionou-se um soquete ZIF de 28 pinos diretamente no soquete original do microcontrolador da placa Duemilanove, conforme visto na figura 15. Isso para permitir que se colocasse e retirasse o ATmega328P diversas vezes na placa, pois para a gravação do bootloader era necessário removê-lo da placa original e conectá-lo na placa de gravação.

Figura 15 – Arduino Duemilanove com soquete ZIF e ATmega328P

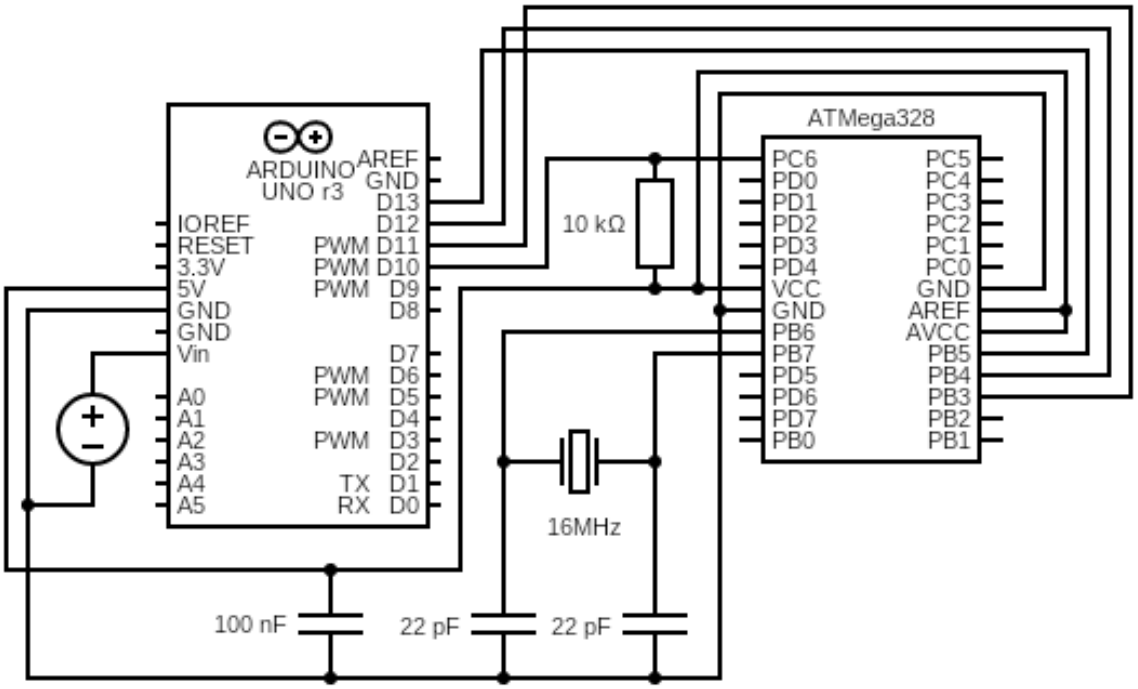


Fonte: o Autor (2021).

3.2.1 Placa para gravação

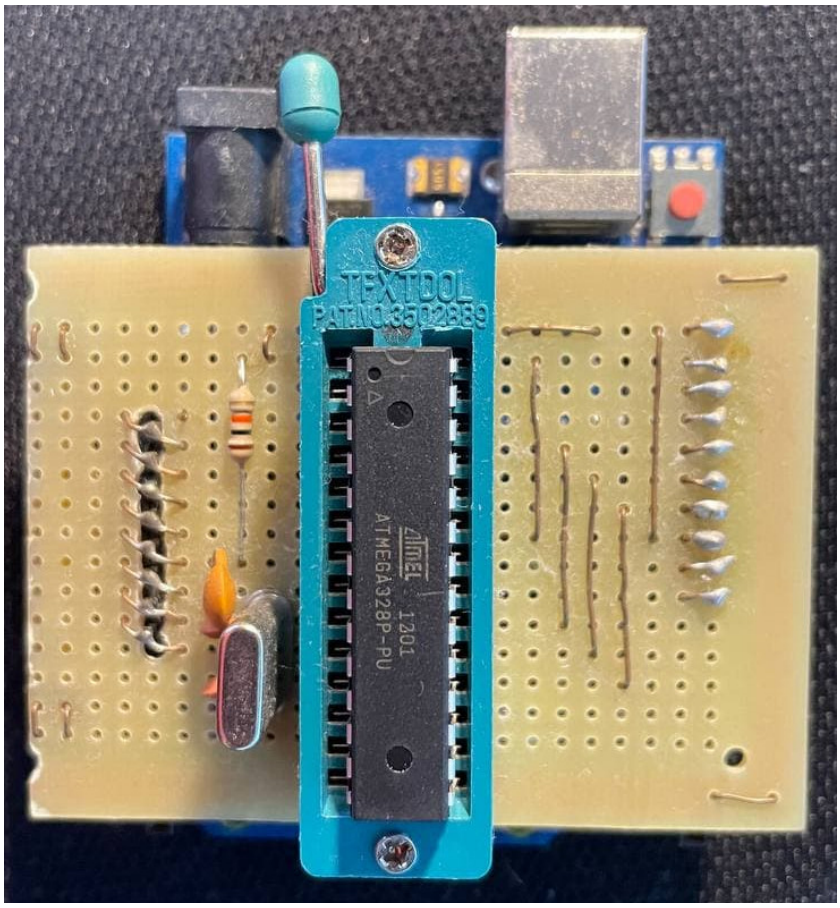
Confeccionou-se manualmente a placa de gravação, figura 17, de acordo com o esquemático apresentado na figura 16. Esse circuito permite que seja usado uma placa de desenvolvimento Arduino, juntamente com o software de desenvolvimento fornecido pelo fabricante, Arduino IDE, para realizar-se a gravação do bootloader no ATMega328P (ARDUINO, 2018a).

Figura 16 – Circuito para a gravação do bootloader no ATmega328P através da placa Arduino Uno



Fonte: o Autor (2021).

Figura 17 – Placa confeccionada de acordo com o esquemático



Fonte: o Autor (2021).

3.2.2 Método de gravação

Por ser código aberto, os bootloaders para as placas Arduino são facilmente encontrados e alterados nos diretórios do Arduino IDE. Na versão utilizada, para distribuições Linux Debian, o bootloader para a placa Duemilanove se encontra no diretório: `arduino-1.8.13/hardware/arduino/avr/bootloaders`. Esse diretório é criado automaticamente na instalação do software. Nele há um "makefile", que traz os comandos de compilação do bootloader, um arquivo ".hex" (hexadecimal) que se trata do bootloader já compilado e um arquivo ".c", o qual é o código-fonte.

Ao realizar a gravação do bootloader, o Arduino IDE grava o arquivo ".hex" no microcontrolador. Então é necessário compilar o código caso se deseje um bootloader diferente da versão original. Isso pode ser feito por qualquer compilador compatível com o código escrito e com o microcontrolador utilizado.

Para este trabalho, utilizou-se o compilador avr-gcc, presente no "makefile" original, já instalado com o Arduino IDE. Para compilação fez-se o uso do arquivo de "makefile", dando-se o comando "`make atmega328`", conforme figura 18. Em versões do

bootloader desenvolvidas neste trabalho, foram necessárias pequenas alterações para adicionar novos arquivos para a compilação ou para remover algum dado que passou a integrar o código.

Figura 18 – Compilando o código do bootloader

```
luigi@Ubuntu:~/Downloads/arduino-1.8.13-linux64/arduino-1.8.13/hardware/arduino/
avr/bootloaders/atmega$ make atmega328
avr-gcc -g -Wall -Os -mmcu=atmega328p -DF_CPU=16000000L  '-DMAX_TIME_COUNT=F_CPU
>>4' '-DNUM_LED_FLASHES=1' -DBAUD_RATE=57600   -c -o ATmegaBOOT_168.o ATmegaBOOT
_168.c
avr-gcc -g -Wall -Os -mmcu=atmega328p -DF_CPU=16000000L  '-DMAX_TIME_COUNT=F_CPU
>>4' '-DNUM_LED_FLASHES=1' -DBAUD_RATE=57600 -Wl,--section-start=.text=0x7800 -o
 ATmegaBOOT_168_atmega328.elf ATmegaBOOT_168.o
avr-objcopy -j .text -j .data -O ihex ATmegaBOOT_168_atmega328.elf ATmegaBOOT_16
8_atmega328.hex
rm ATmegaBOOT_168_atmega328.elf ATmegaBOOT_168.o
```

Fonte: o Autor (2021).

Com o bootloader compilado, realizou-se a gravação usando a Arduino IDE. Ao abrir o programa, o código que está em seu editor não é relevante para a gravação, bastando configurar a IDE, na aba “Tools”, conforme quadro 5.

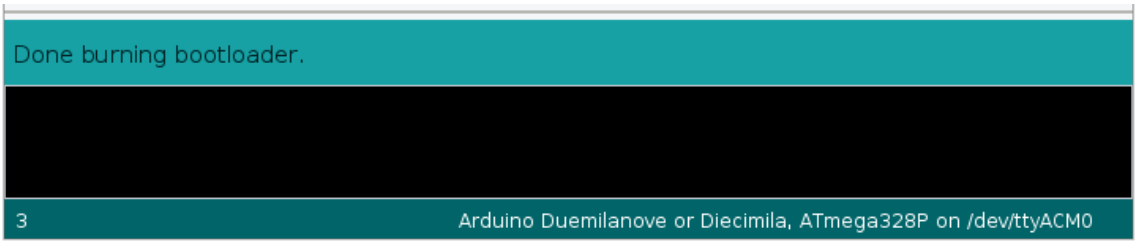
Quadro 5 – Configurando a IDE para a gravação do bootloader

Board	Arduino Duemilanove. Placa que receberá o microcontrolador onde o bootloader será gravado.
Processor	ATmega328P. Microcontrolador que receberá o bootloader.
Programer	Arduino as ISP. A gravação será feita utilizando o Arduino Uno para comunicação.
Port	Identificação da porta USB que será utilizada.

Fonte: o Autor (2021).

Com a pplaca conectada ao computador a placa de gravação, pela porta USB, usou-se o comando de gravação usando o item de menu “Burn Bootloader”, na aba “Tools”. O progresso e a conclusão do processo de gravação são apresentados na parte abaixo do editor do Arduino IDE conforme ilustrado na figura 19.

Figura 19 – Arduino IDE após gravação do bootloader



Fonte: o Autor (2021).

Neste ponto, o bootloader está gravado e já é possível utilizar o microcontrolador. Basta colocá-lo no Arduino Duemilanove e realizar a gravação de uma aplicação pelo Arduino IDE.

3.3 Gravação de aplicação no Arduino Duemilanove

Para a gravação de uma aplicação pelo Arduino IDE é necessário abri-la no editor com o comando de teclado *Ctrl+O*, ou escreve-la diretamente no editor do Arduino IDE. Neste trabalho utilizou a gravação da aplicação “Blink”. Tal aplicação é parte do Arduino IDE. Pode ser acessada no item do item “01.basics”, presente no item “Examples” da aba “Files”. Com a aplicação aberta, fez-se a configuração, na aba “Tools”, descrita no quadro 6.

Quadro 6 – Configurando a IDE para a gravação da aplicação

Board	Arduino Duemilanove. Placa que receberá a aplicação.
Processor	ATmega328P. Microcontrolador presente na placa.
Programer	AVRISP mkII. Isso definirá o protocolo de comunicação STK500, que é o protocolo de comunicação originalmente utilizado para a gravação via USB (ATMEL CORPORATION, 2003).
Port	Identificação da porta USB que será utilizada.

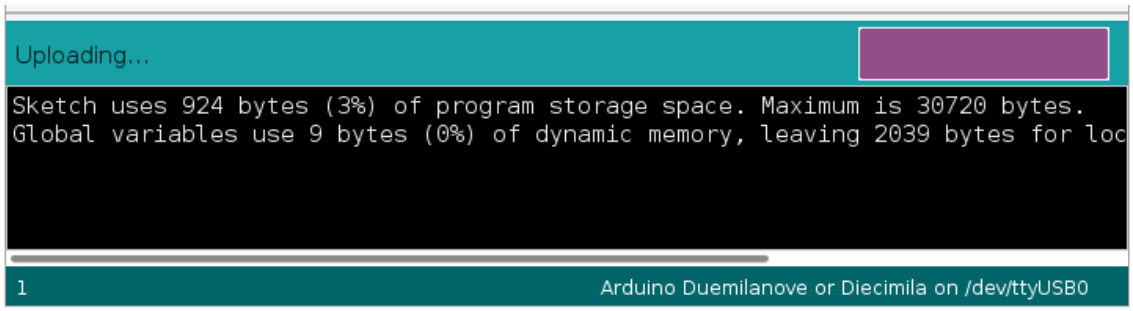
Fonte: o Autor (2021).

Por fim, com o Arduino Duemilanove conectado à porta USB, a compilação e upload são executados clicando num ícone dedicado a essa ação, representado como seta localizada na parte superior esquerda da IDE, mostrado na figura 20. Durante o processo, apresenta-se uma barra de progresso, conforme figura 21 e, ao final, é exibida uma mensagem sinalizando a conclusão do processo, como indica a figura 22.

Figura 20 – Ícone para compilação e gravação do código

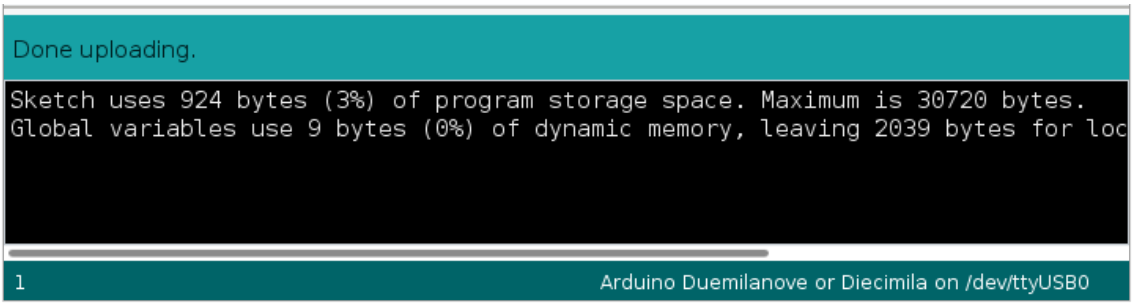
Fonte: o Autor (2021).

Figura 21 – Arduino IDE durante gravação de aplicação



Fonte: o Autor (2021).

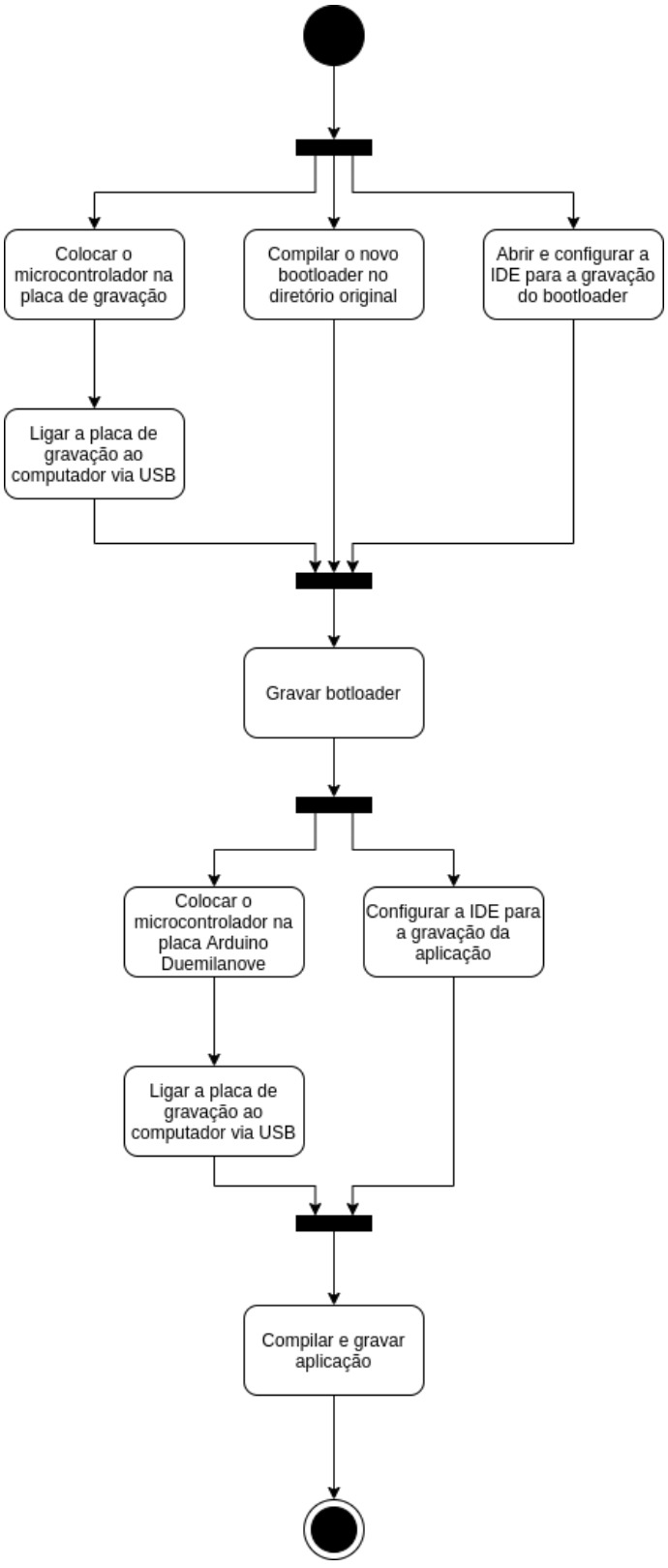
Figura 22 – Arduino IDE após gravação de aplicação



Fonte: o Autor (2021).

Na figura 23 apresenta-se, o processo de gravação de um novo bootloader de maneira simplificada pelo diagrama de atividades.

Figura 23 – Diagrama de atividade da gravação de um novo bootloader



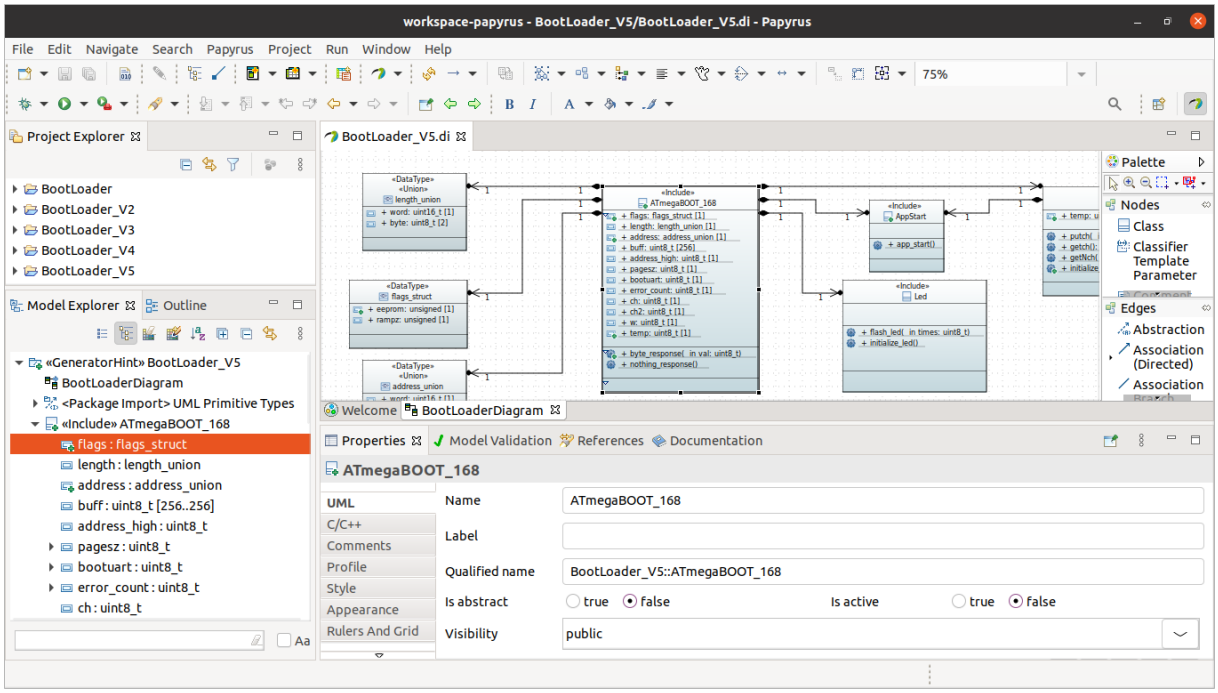
Fonte: o Autor (2021).

3.4 Papyrus

Consolidada a maneira de realizar a gravação do bootloader modificado, iniciou-se a modelagem do bootloader. Para isso, fez-se o uso do programa Papyrus.

A escolha do Papyrus, ilustrado na figura 24, deu-se por ser uma ferramenta de engenharia, código aberto, de modelagem gráfica (PAPYRUS, 2020). O Papyrus dispõe de uma série de ferramentas úteis para projetos baseados em modelagem e é personalizável, além de ser utilizado na indústria (PAPYRUS, 2020), (PAPYRUS, 2019).

Figura 24 – Papyrus



Fonte: o Autor (2021).

O uso da ferramenta possibilitou a utilização dos elementos UML descritos no capítulo 2.1.2, de modo a descrever os códigos desejados para o trabalho. O uso do perfil nativo do software para modelagem em linguagem C e C++ permitiu criar os modelos desejados de maneira satisfatória.

3.5 Papyrus Software Designer

Além da modelagem, o Papyrus também permite realizar a geração de código conforme descrito no capítulo 2.1.3 com o uso do Papyrus Software Designer, que realiza a geração de códigos em C, C++ e Java a partir dos modelos criados no Papyrus (ECLIPSE, 2021).

Também existem algumas limitações na geração de código. Como a geração de uniões (tipo union do C), que não é suportada, sendo necessária uma modificação no gerador de código para corrigir a geração do tipo “union”, bem como outros detalhes

necessários para realizar esse processo. Outra limitação é a necessidade do uso de comportamentos opacos para descrever comportamentos. Isso, pois o gerador não realiza a tradução de modelos comportamentais.

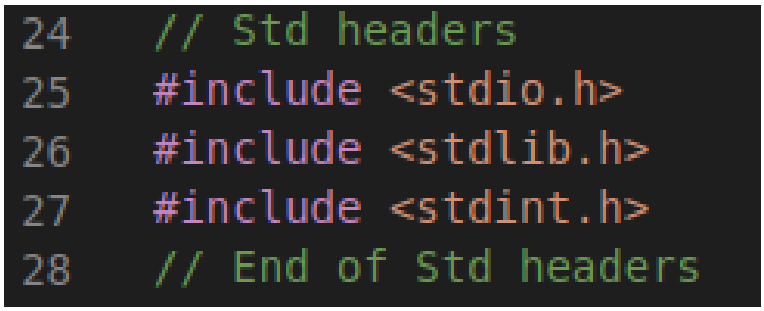
Para a geração de código em C funcional, as seguintes escolhas de modelagem foram adoradas para todo o trabalho:

Classes: Todas as classes são marcadas como não abstratas e recebem um estereótipo de um elemento da linguagem C. Desse modo o gerador cria um arquivo de cabeçalho (".h") para cada classe, contendo uma estrutura (tipo struct do C), com os atributos e operações, caso existam. Além isso, também gerou uma função fora da estrutura que pode ser utilizada como uma espécie de construtor para a instância da estrutura. Tal função não é utilizada neste trabalho. Também inclui outros cabeçalhos que tenham associação do tipo composição com essa classe. Quando existem operações na classe, é criado um arquivo ".c" com funções geradas a partir das operações. Desse modo, é possível criar o encapsulamento desejado com o uso de arquivos.

Na maior parte das classes, o estereótipo de C utilizado foi o padrão presente no Papyrus, o que adiciona os campos de corpo, pré-condição e cabeçalho ao elemento da classe, independentes do comportamento opaco, com exceção das uniões e estruturas, detalhadas no próximo capítulo. O corpo é uma maneira de incluir texto, ou código, diretamente no corpo do ".c", utilizado para as definições (define), que não são contempladas pelo gerador e também em casos específicos comportamentais que serão apresentados. O campo de pré-condição não foi utilizado e o campo de cabeçalho foi utilizado, quando necessário, para acrescentar bibliotecas por meio de includes. A utilização desse estereótipo também ocasiona na inclusão automática de algumas bibliotecas, o que é apresentado na figura 25.

Todos os elementos da classe (operações e atributos) foram colocados fora da estrutura. Fez-se isso para que o comportamento estático do bootloader fosse melhor representado, bem como o acesso dos elementos fosse realizado por qualquer instância onde seu ".h" fosse incluído, mantendo o comportamento mais próximo do comportamento original. A maneira como isso é feito é apresentada a seguir, onde as considerações de cada elemento são detalhadas.

Figura 25 – Bibliotecas incluídas por conta do uso do estereótipo de C



Fonte: o Autor (2021).

Aributos: Todos os atributos são marcados como estático e únicos, e recebem um tipo compatível com a linguagem C. Desse modo, o gerador os coloca fora da estrutura criada para a classe, garantindo seu caráter estático na memória do microcontrolador.

Operações: Por padrão, ao utilizar esteriótipos de C na classe, as operações são representadas como funções do C. Porém, assim como os atributos, todas as operações foram marcadas como estáticas e também como não consultivas. Desse modo, eles ficam fora da estrutura, garantindo seu comportamento estático. Também tiveram sua concorrência marcada como sequencial, considerado como comportamento padrão de execução de sistemas microcontrolados.

Os métodos são descritos por comportamentos opacos ativos, não recursivos e não abstratos, com a linguagem definida como C e corpo escrito em C, garantindo o perfeito funcionamento das funções, e permitindo que sejam reutilizados os códigos do bootloaders originais. Entretanto, essa decisão se dá principalmente por limitações do gerador de código, que não lida com modelos comportamentais na descrição de operações, apenas comportamentos opacos.

Os parâmetros, quando existem, são sempre compatíveis com a linguagem C e definidos unicamente em casos de entrada ou retorno das funções. Isso porque os parâmetros utilizados apenas no escopo da própria função já estarão contemplados no corpo do comportamento opaco.

Associações: Toda associação é uma composição com multiplicidade [1..1], gerando a inclusão do cabeçalho da classe alvo na classe base.

Após gerados, testaram-se os códigos, realizando a gravação de uma aplicação de exemplo no microcontrolador que recebeu o bootloader, e analisados estaticamente.

3.6 MISRA C

Padrão de programação desenvolvido pela colaboração de diversos fabricantes, que traz uma série de diretrizes e regras para o desenvolvimento seguro de aplicações e softwares em C. Desenvolvida para sistemas embarcados automotivos, o padrão visa oferecer uma maneira de reduzir ou remover erros em códigos escritos na linguagem C, por meio da definição de um subconjunto da linguagem (MISRA, 2013). A sua escolha se dá por ser o principal padrão de programação para desenvolvimento de sistemas embarcados. A verificação das conformidades do código com as regras do padrão é feita por ferramentas de análise estática de código.

3.7 Ferramentas de análise estática de software

Técnicas de análise estática são quaisquer métodos usados para inspecionar um código sem este ser executado (EGELE et al., 2012). Quando realizadas por ferramentas de análise, possibilitam detectar anomalias no código analisando o seu texto, verificando se códigos estão ou não sendo utilizados de maneira correta, comparando-os com padrões de código ou, até mesmo, calculando todos os valores possíveis para dados do programa (SOMMERVILLE, 2019).

Por serem feitas de maneira rápida, com baixo custo e sem a necessidade de utilização de nenhuma outra codificação além daquela a ser analisada, são facilmente implementadas no desenvolvimento de softwares quando comparadas com outras técnicas (SOMMERVILLE, 2019). Por essas características, escolheu-se o método de análise para a realização deste trabalho.

Vale lembrar que existem erros não cobertos pela análise estática, como erros de lógica. Também é comum a ocorrência falsos positivos. Isso tendo em vista que o uso da análise estática, neste trabalho, busca por erros típicos de programação e por não conformidades com o padrão de programação. Porém, seu uso se justifica pela excelente relação entre aumento de qualidade do código e recursos necessários para a realização da análise.

As ferramentas escolhidas para a realização da análise são descritas nas seções a seguir.

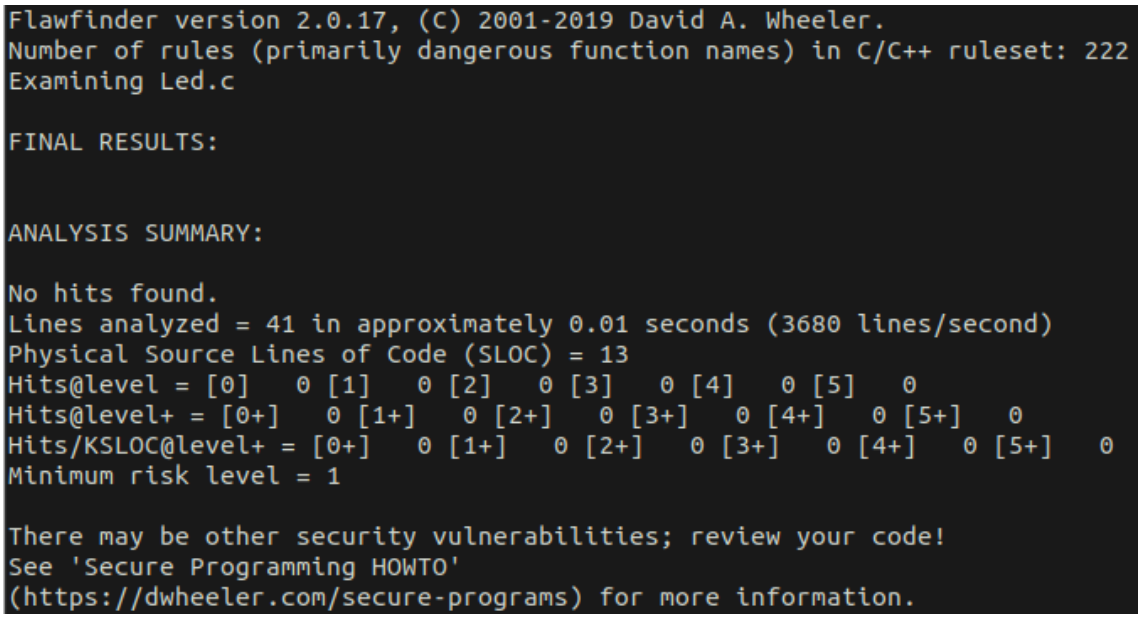
3.7.1 Flawfinder

Flawfinder é uma ferramenta que busca por prováveis falhas de segurança em códigos escritos nas linguagens C e C++. Ele pode buscar essas falhas em um arquivo de código ou em vários arquivos que estejam em um diretório analisado. Seu resultado elenca os erros em níveis que variam do 0, mais brando, ao 5, grande risco (FLAWFINDER, 2017).

As buscas por erros são baseadas numa base de dados interna do programa, os quais contêm uma série de erros conhecidos em ambas as linguagens. Ao encontrar um padrão entre o código e um erro do banco de dados ele acusará a possível falha (FLAWFINDER, 2017).

Na figura 26 é possível observar uma análise que não retornou nenhuma falha encontrada.

Figura 26 – Resultado de uma análise estática realizada com Flawfinder

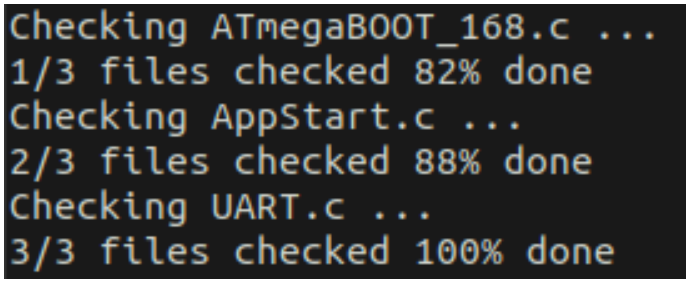


Fonte: o Autor (2021).

3.7.2 CppCheck

Similar ao Flawfinder, o CPPCheck é uma ferramenta de análise estática de código. Ela foi desenvolvida para avaliar códigos escritos em C e C++ para vários compiladores e também com trechos escritos em Assembly (CPPCHECK TEAM, 2021). A figura 27 mostra o resultado de uma análise realizada pela ferramenta.

Figura 27 – Resultado de uma análise estática realizada com CppCheck

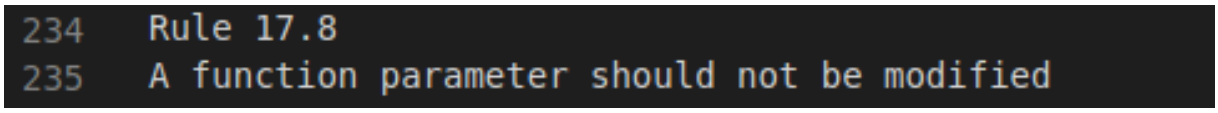


Fonte: o Autor (2021).

Uma característica importante do CppCheck é a capacidade de definir

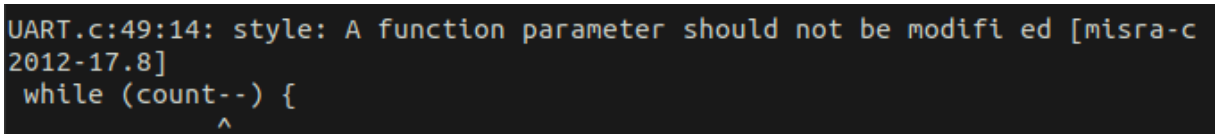
facilmente um padrão de código para que a ferramenta analise se o código escrito é ou não compatível com um padrão (CPPCHECK TEAM, 2021). Diversos desses padrões são nativos, mas podem ser adicionados outros padrões com o uso de complementos, como o misra.py (CPPCHECK TEAM, 2021). O misra.py é um complemento que verifica a conformidade do código com o padrão MISRA C, descrito em 3.6, (CPPCHECK TEAM, 2021). Por ser proprietário, o conjunto de regras do MISRA C precisa ser fornecido em um arquivo de texto para que a análise possa ser realizada pela ferramenta (CPPCHECK TEAM, 2021). Na figura 28, é possível observar um trecho do arquivo de texto necessário para a análise. E na figura 29 é apresentado um resultado de uma análise estática utilizando o complemento misra.py.

Figura 28 – Regra do MISRA C formatada em arquivo de texto para uso do CppCheck



Fonte: o Autor (2021).

Figura 29 – Resultado de uma análise estática realizada com o complemento para o CppCheck, misra.py



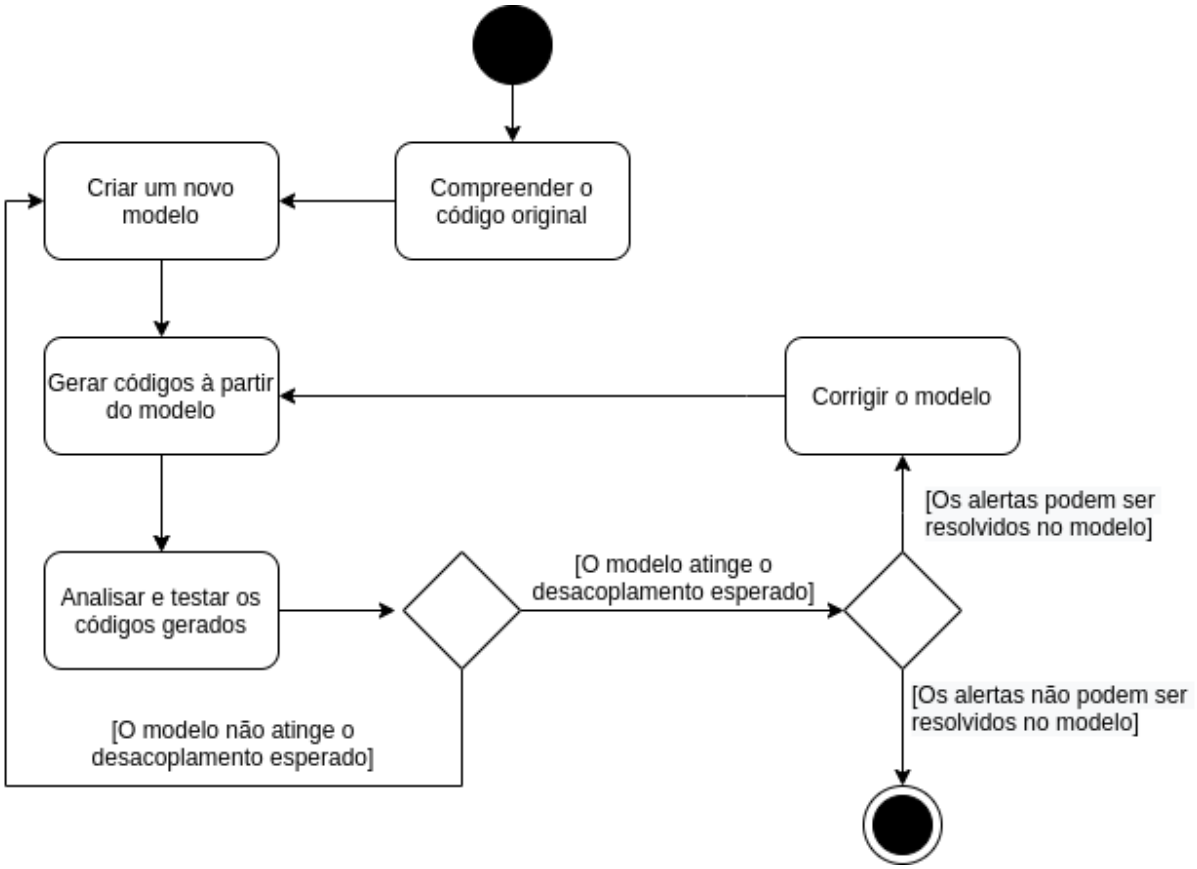
Fonte: o Autor (2021).

4 RESULTADOS

Os resultados, obtidos por meio do método exploratório, estão apresentados neste capítulo.

Gerou-se um modelo inicial, baseado na engenharia reversa do código original de bootloader fornecido pelo Arduino. A partir dele, geraram-se sucessivos modelos buscando, de maneira iterativa, aproximar-se do resultado desejado para o trabalho, conforme apresentado na figura 30. As alterações em cada modelo deram-se nos seus aspectos estáticos, tendo como primeiro objetivo um desacoplamento que permita a reutilização dos componentes do bootloader em outra aplicação e a busca da melhoria da qualidade do código, principalmente segundo sua conformidade com o MISRA C.

Figura 30 – Diagrama de atividades do processo de desenvolvimento dos modelos

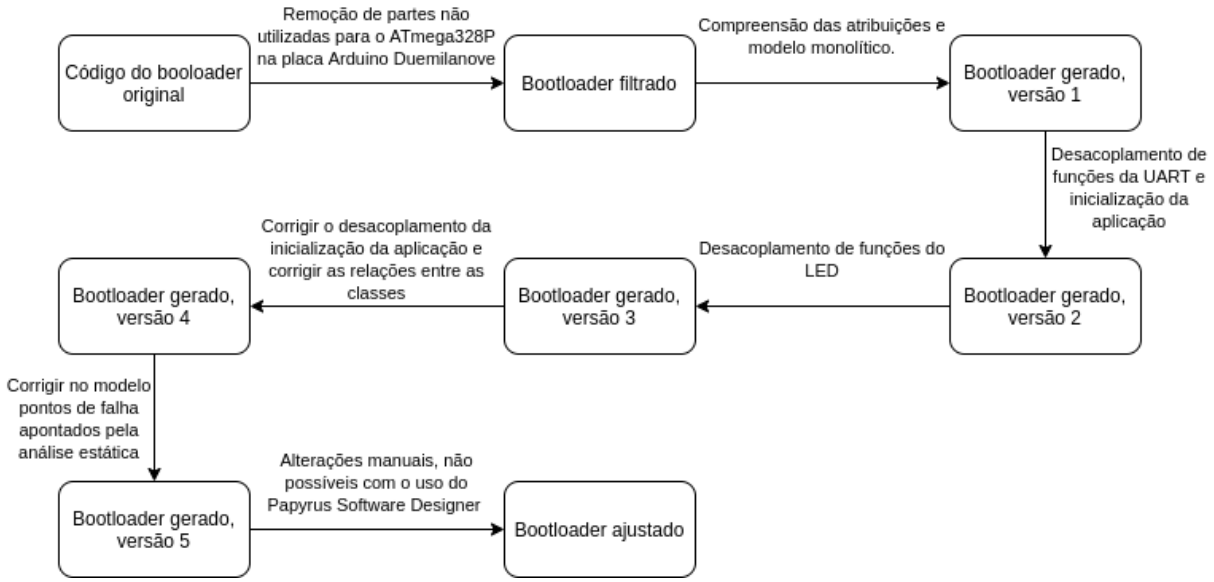


Fonte: o Autor (2021).

Após isso, fez-se uma alteração manual na última versão dos códigos gerados, feita de modo a apresentar um resultado proposto, que não pôde atingir-se com o

gerador de código sem alterações na maneira que a ferramenta traduz os modelos. Esse processo é representado, juntamente com os bootloaders resultantes, na figura 31.

Figura 31 – Diagrama de atividades dos códigos obtidos



Fonte: o Autor (2021).

Por fim, apresenta-se uma aplicação construída para demonstrar que o desacoplamento desejado entre as atribuições do bootloader foi alcançado.

Testaram-se todas as versões de maneira prática por sua gravação no microcontrolador conforme descrito em 3.2. Seguida da gravação de uma aplicação de exemplo, conforme descrito em 3.3. A existência de erros no funcionamento bootloader não permitiria que a aplicação fosse gravada com sucesso. Também submeteu-se a análise estática todos os códigos de bootloaders obtidos, sendo que apenas o CppCheck, quando utilizado com o complemento misra.py, demonstrou a presença de não conformidade com o MISRA C. O Flawfinder não apresentou possíveis falhas em nenhuma das análises.

4.1 Estudo do bootloader original

O bootloader fornecido para o Arduino Duemilanove é compatível com diversas placas da família Arduino. Assim, para possibilitar a compilação com diversas placas, é necessário lidar com as diferenças presentes entre elas. Essas diferenças incluem periféricos e, até mesmo, o uso de diferentes microcontroladores. Algo que pode ser ilustrado pelas figuras 32 e 33, que apresentam respectivamente os fragmentos de comandos para a compilação do código para o ATmega328P e também para o ATmega1280, que apresentam diferentes posições iniciais de memória.

Figura 32 – Trecho do “makefile” com as flags para o microcontrolador ATmega328p

```
153   atmega328: TARGET = atmega328
154   atmega328: MCU_TARGET = atmega328p
155   atmega328: CFLAGS += '-DMAX_TIME_COUNT=F_CPU>>4' '-DNUM_LED_FLASHES=1' -DBAUD_RATE=57600
156   atmega328: AVR_FREQ = 16000000L
157   atmega328: LDSECTION  = --section-start=.text=0x7800
158   atmega328: $(PROGRAM)_atmega328.hex
```

Fonte: o Autor (2021).

Figura 33 – Trecho do “makefile” com as flags para o microcontrolador ATmega1280

```
198   mega: TARGET = atmega1280
199   mega: MCU_TARGET = atmega1280
200   mega: CFLAGS += '-DMAX_TIME_COUNT=F_CPU>>4' '-DNUM_LED_FLASHES=0' -DBAUD_RATE=57600
201   mega: AVR_FREQ = 16000000L
202   mega: LDSECTION  = --section-start=.text=0x1F000
203   mega: $(PROGRAM)_atmega1280.hex
```

Fonte: o Autor (2021).

Essa possibilidade de uso em diversas placas tem reflexos diretos no código-fonte do bootloader. Vários trechos do código, como a mostrada na figura 34, servem apenas para lidar com essa pluralidade de aplicações. Desse modo, muitos desses trechos do código são necessários apenas para a compilação para uma configuração específica de placa, como mostrado na figura 35. Nela é possível observar uma parte do código útil apenas o microcontrolador ATmega1280.

Figura 34 – Trecho do bootloader original que lida com a possibilidade de uso para diversas configurações de placa

```
972   #elif defined(__AVR_ATmega168__)  || defined(__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
973       uint32_t count = 0;
974       while(!(UCSR0A &  BV(RXC0))){
```

Fonte: o Autor (2021).

Figura 35 – Trecho do bootloader original utilizado apenas para o ATmega1280

```
155   #if defined __AVR_ATmega1280__
156   #define SIG2    0x97
157   #define SIG3    0x03
158   #define PAGE_SIZE   0x80U   //128 words
```

Fonte: o Autor (2021).

Como este trabalho explora apenas o uso da placa Duemilanove na versão que possui o microcontrolador ATmega328P, a primeira etapa removeu manualmente os trechos de código desnecessários para essa aplicação. Isso também facilitou a compreensão de seu funcionamento para realizar a engenharia reversa. Assim, determinaram-se as atribuições do bootloader em questão.

Após as edições do código original, do código resultante (apêncice A),

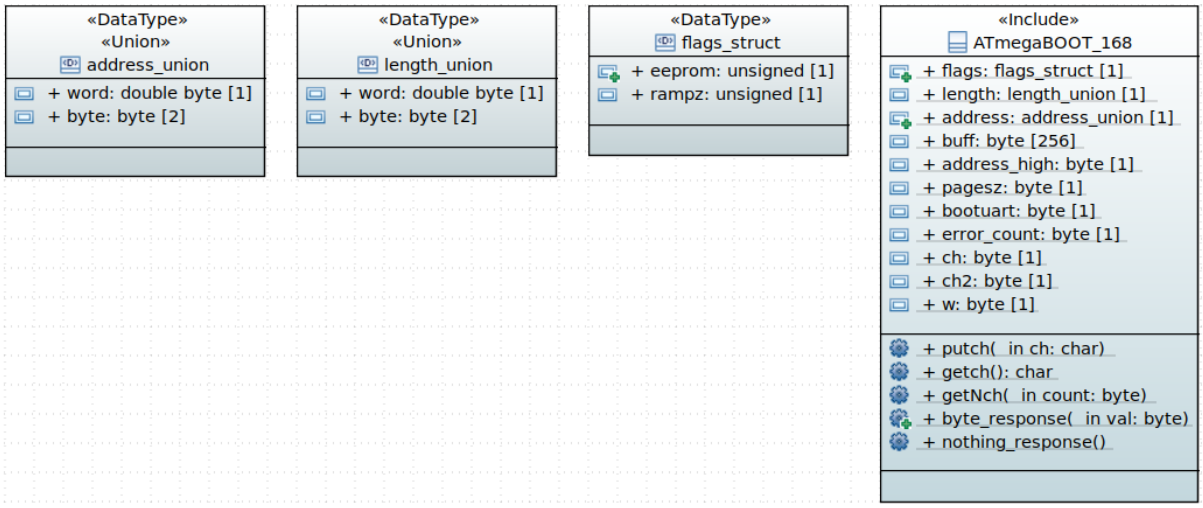
constatou-se que ele tem as seguintes capacidades:

a) Realizar a chamada a aplicação;
b) configurar e controlar o LED presente na placa Arduino Duemilanove;
c) inicializar e realizar comunicação UART (2.3.1); e
d) interpretar as mensagens enviadas pelo protocolo STK500 (3.3), realizando as ações requeridas, por exemplo gravar a aplicação na memória do microcontrolador.

4.2 Bootloader gerado, versão 1

Considerando inicialmente o código resultante da etapa descrita em 4.1, iniciou-se a modelagem. Nessa primeira etapa, criou-se um modelo monolítico de modo a manter a proximidade com o código anterior, com o único desacoplamento do modelo feito pela separação dos tipos union e struct. Pois, esses necessitaram do uso de um esteriótipo para esses tipos, diferentes do estereótipo padrão de C utilizado para as demais classes (3.4 e 3.5).

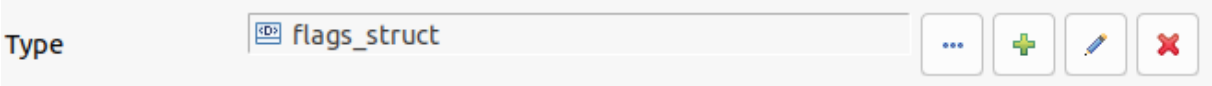
Figura 36 – Primeira versão do modelo



Fonte: o Autor (2021).

Nesse modelo existe uma classe que representa o código principal (ATmegaBOOT_168), onde as uniões e estruturas são adicionadas manualmente como atributos, bem como as outras variáveis do código. Nas uniões e estruturas, referenciou-se o modelo de classe de cada uma como o tipo do atributo, conforme figura 37.

Figura 37 – Usando modelo de classe como tipo do atributo



Fonte: o Autor (2021).

Adicionaram-se, também manualmente, as operações na classe. Elas são a representação das funções do código analisado. Todas têm seu comportamento definido por comportamentos opacos retirados das funções do código original.

No campo de corpo da classe ATmegaBOOT_168 acrescentou-se o ponteiro de função usado para iniciar a aplicação e a função principal, ou função *main*. Por essa última adição, define-se esta como a classe principal do bootloader que será gerado.

Ao gerar o código (Apêndice B), criou-se um arquivo de cabeçalho para cada classe. Os cabeçalhos das uniões e estrutura são referenciados no cabeçalho da classe principal pelo gerador, por conta do uso dos modelos como tipos nos atributos da classe principal. Isso é observado na figura 38. Gerou-se, também, um arquivo de código contendo o comportamento das funções, a “main” e outros elementos descritos em 3.5.

Figura 38 – Referência para uniões e estruturas

```
17  // Derived includes
18  #include "address_union.h"
19  #include "flags_struct.h"
20  #include "length_union.h"
21  // End of Derived includes
```

Fonte: o Autor (2021).

Com o código gerado realizaram-se as análises estáticas, embora elas ainda não fossem consideradas para as alterações do código, pois o modelo ainda não apresentava o desacoplamento desejado. Os resultados das análises do Flawfinder e do CppCheck não retornaram alertas, porém a análise realizada com o misra.py tem o retorno detalhado na tabela 2.

Tabela 2 – Resultado da análise estática com misra.py do primeiro código gerado.

Regra	Ocorrências
2.7	1
10.4	46
11.9	2
12.1	3
12.3	1
14.4	19
15.6	65
15.7	3
17.7	8
17.8	3
19.2	6
20.1	3
20.7	1
21.6	1

Fonte: o Autor (2021).

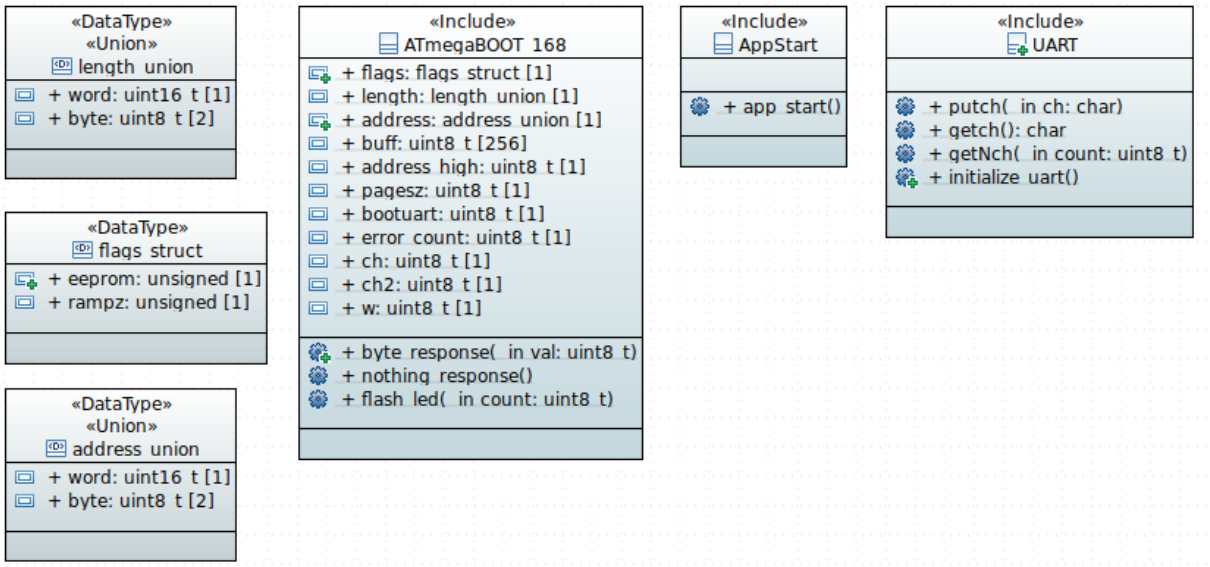
Cada valor da coluna de regra, representa a regra do padrão que não está sendo respeitada, e a de ocorrências mostra quantas vezes essa regra não é respeitada em todo o código-fonte. Serão apresentadas, a seguir, mudanças nos modelos que visam reduzir o número de ocorrências dessas não conformidades.

4.3 Bootloader gerado, versão 2

Partindo do primeiro modelo, iniciou-se o desacoplamento das atribuições do microcontrolador. Para isso criaram-se duas novas classes, uma para a manipulação da UART e uma segunda responsável por iniciar a aplicação.

Conforme observado na figura 39, removeram-se as operações referentes a UART de ATmegaBOOT_168 e passaram-se, então, para a classe UART. Também criaram-se duas novas operações, uma para iniciar a aplicação e outra para inicializar a UART. Essas operações tinham suas atribuições presentes na classe ATmegaBOOT_168 diretamente na função *main*.

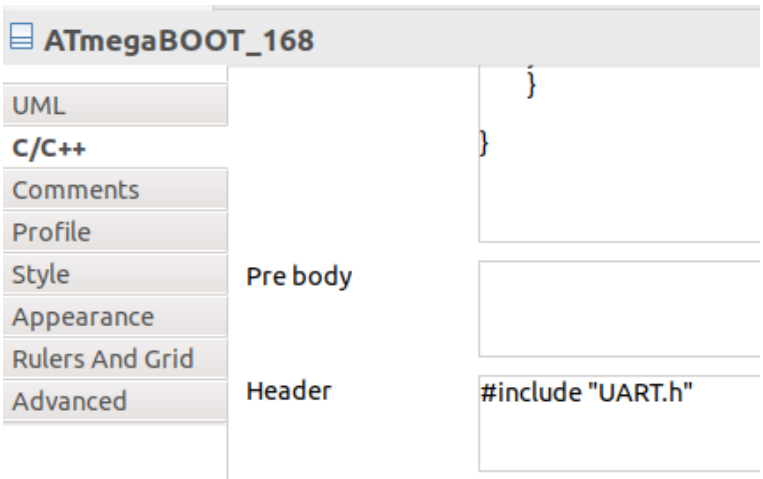
Figura 39 – Segunda versão do modelo



Fonte: o Autor (2021).

Geraram-se, assim, dois novos arquivos de cabeçalho e dois de código, seguindo o padrão do gerador de códigos. A inclusão desses novos cabeçalhos deu-se pelo uso do estereótipo de C, que traz o campo de cabeçalho, conforme ilustrado na figura 40, e se reflete no código como mostrado na figura 41.

Figura 40 – Uso do estereótipo de C para adição de texto ao cabeçalho no modelo UML



Fonte: o Autor (2021).

Figura 41 – Cabeçalho incluído com o uso do estereótipo

```
26   // Include from Include stereotype (header)
27   #include "UART.h"
```

Fonte: o Autor (2021).

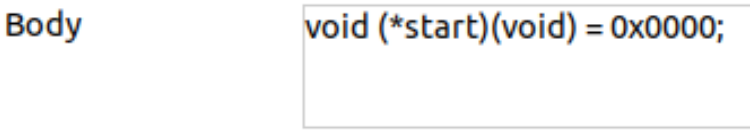
Dessa maneira, a classe do bootloader (ATmegaBOOT_168) recebeu a adição do cabeçalho da classe de UART. A classe de UART recebeu da classe AppStart, o que também torna essa última acessível à classe do bootloader. Isso foi necessário, pois originalmente a função que recebe um caractere via comunicação UART tem a capacidade de iniciar a aplicação. Ela faz isso como um tratamento de erro, após um número de tentativas falhas ao ler o valor da UART. Esse modelo segue a construção do bootloader original, porém isso se torna um problema para a ideia de desacoplamento das atribuições a ser corrigido em modelos seguintes.

No corpo do estereótipo de C da classe UART, acrescentou-se a biblioteca que mapeia as entradas do microcontrolador conforme a placa, apresentado na figura 42. Em AppStart, no campo de corpo, adicionou-se o ponteiro de função que faz o mapeamento de memória para iniciar a aplicação, conforme mostrado na figura 43, e que é chamado pela operação “app_star”.

Figura 42 – Adição de biblioteca no campo Header da classe UART

Fonte: o Autor (2021).

Figura 43 – Ponteiro de função usado para iniciar a aplicação



Fonte: o Autor (2021).

Submeteram-se, então, os códigos gerados às análises estáticas, tendo possíveis erros apontados apenas pelo misra.py, apresentados na tabela 3. Tais resultados demonstraram uma sensível redução nos alertas apontados pela análise quando comparados aos resultados do primeiro modelo, apesar de ainda não ser o foco das alterações a correção de tais erros.

Tabela 3 – Resultado da análise estática com misra.py do segundo código gerado.

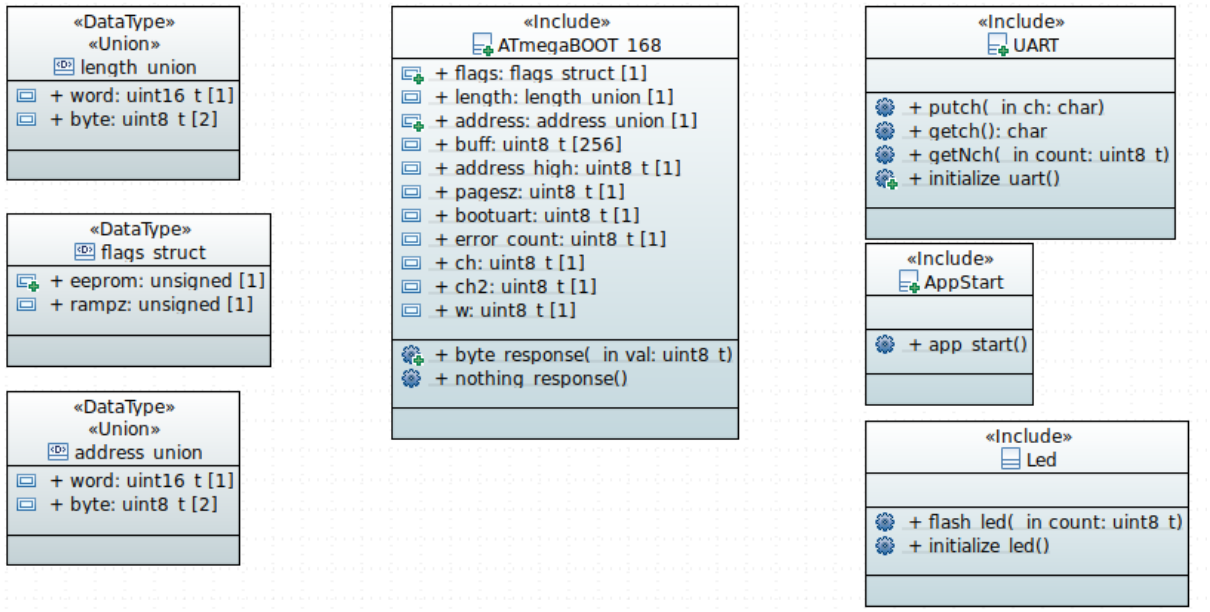
Regra	Ocorrências
2.7	1
10.4	23
11.9	1
12.1	1
14.4	8
15.6	23
15.7	1
17.7	4
17.8	2
19.2	2
20.7	3
21.6	2

Fonte: o Autor (2021).

4.4 Bootloader gerado, versão 3

O terceiro modelo teve a mesma finalidade da versão anterior, porém, dessa vez, separaram-se as atribuições de controle do LED. Passando a operação de manipulação do LED da classe ATmegaBOOT_168 para a classe Led, e com comportamento de inicializar o LED retirado da função *main* para tornar-se uma operação em LED, conforme observado na figura 44.

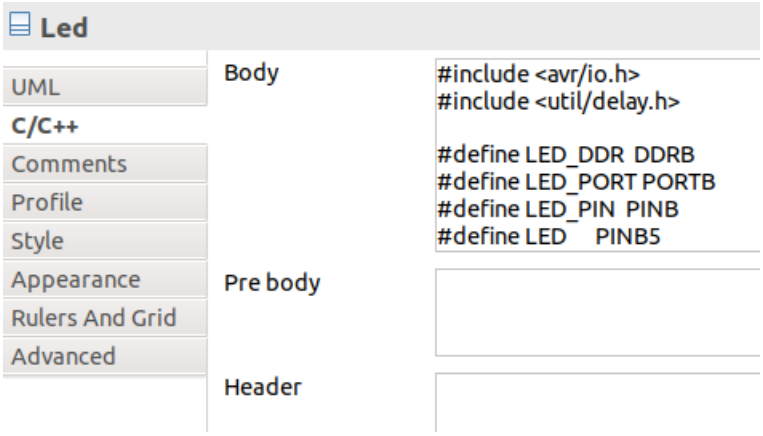
Figura 44 – Terceira versão do modelo



Fonte: o Autor (2021).

No estereótipo de C da classe Led, são adicionadas definições ligadas a essa atribuição que estavam na classe ATmegaBOOT_168 e também as bibliotecas necessárias, conforme figura 45. Por sua vez, a inclusão do cabeçalho da classe Led na classe principal se dá de maneira similar ao que se fez na classe UART com o uso do estereótipo de cabeçalho da classe.

Figura 45 – Uso do estereótipo de C na classe Led



Fonte: o Autor (2021).

Ao submeter o código gerado à análise estática, apenas o misra.py apontou possíveis erros, descritos na tabela 4.

Tabela 4 – Resultado da análise estática com misra.py do terceiro código gerado.

Regra	Ocorrências
2.7	2
10.4	23
11.9	1
12.1	1
14.4	8
15.6	23
15.7	1
17.7	4
17.8	2
19.2	2
20.7	4
21.6	3

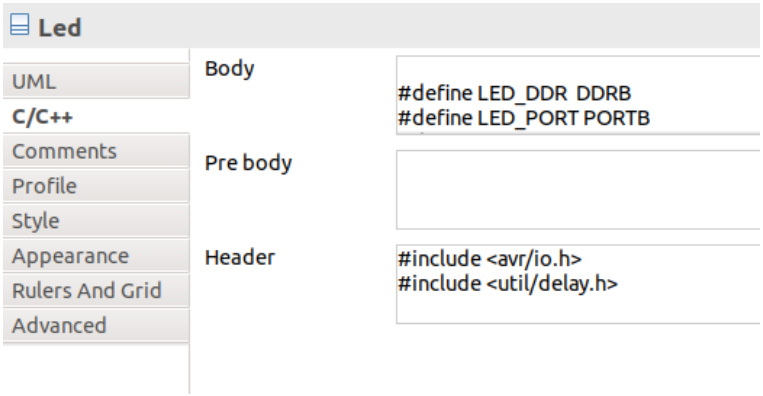
Fonte: o Autor (2021).

4.5 Bootloader gerado, versão 4

A quarta versão teve como proposta corrigir a modelagem anterior de modo a garantir o desacoplamento correto dos componentes, corrigir a maneira como as classes são referenciadas umas nas outras e incluir as bibliotecas no campo correto do estereótipo.

Para isso, os “includes” das classes saíram dos estereótipos sendo usadas associações do tipo composição. Fazendo, assim, com que as classes alvos apareçam como atributos das classes de origem. Os “includes” de bibliotecas que estavam no corpo dos estereótipos de C das classes passaram ao campo de cabeçalho, e removeram-se os cabeçalhos das classes que eram adicionados manualmente, como mostrado na figura 46.

Figura 46 – Uso do estereótipo de C sendo feito da maneira correta



Fonte: o Autor (2021).

Para a correção do encapsulamento, fez-se a remoção da referência de AppStart em UART. Para isso, alterou-se o comportamento opaco da função *getch* apresentado na figura 47 para o apresentado na figura 48. Ou seja, modificou-se a chamada da função que inicia a aplicação pelo retorno de um valor que não é tratado como uma mensagem do protocolo STK500, tornando, assim, a classe ATmegaBOOT_168 contabilizá-lo como um erro. Isso se dá, pois após um número de comandos não compreendidos, o comportamento da função *main* é o de realizar a chamada da aplicação. Isso gera um novo erro na modelagem, pois cria dois pontos de retorno em uma mesma função. Porém, o foco deste trabalho é nos aspectos estáticos do modelo. Assim, este erro é mantido para evitar maiores alterações nos comportamentos do bootloader. E deixam-se tais melhorias como sugestão para trabalhos futuros.

Figura 47 – Função getch antes do encapsulamento de AppStart

```
uint32_t count = 0;
    while(!(UCSR0A & _BV(RXC0))){

        count++;
        if (count > MAX_TIME_COUNT)
            app_start();
    }
    return UDR0;
```

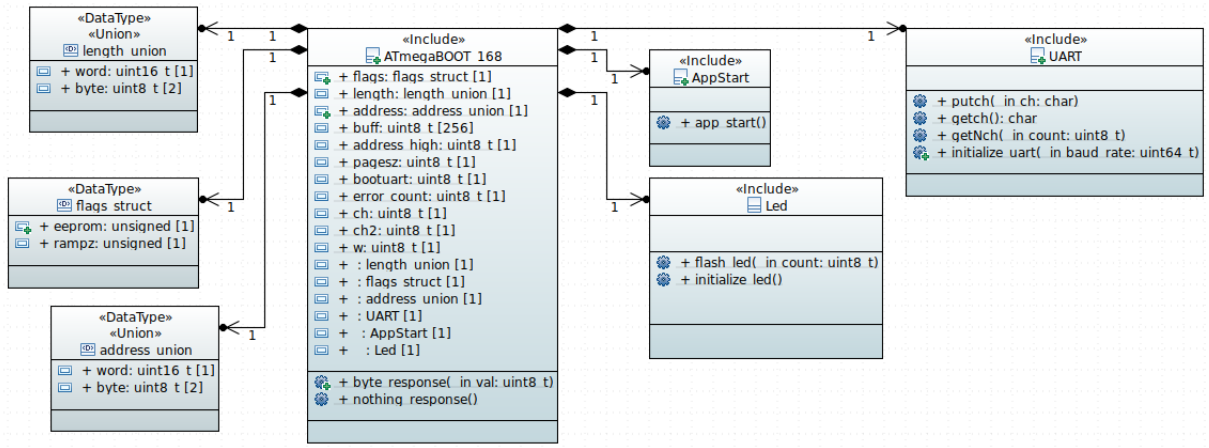
Fonte: o Autor (2021).

Figura 48 – Função getch após o encapsulamento de AppStart

```
uint32_t count = 0;
    while(!(UCSR0A & _BV(RXC0))){

        count++;
        if (count > MAX_TIME_COUNT)
            return 0x00;
    }
    return UDR0;
```

Fonte: o Autor (2021).

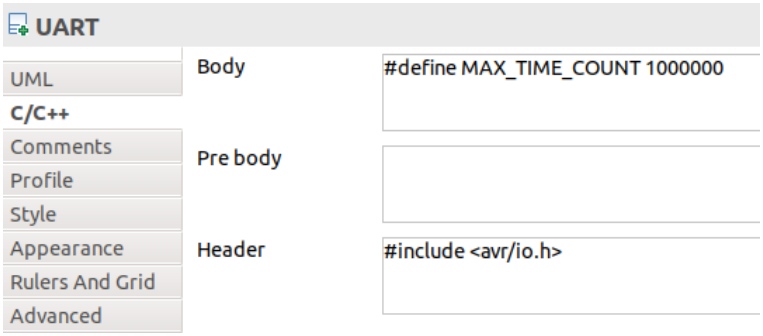
Figura 49 – Quarta versão do modelo



Fonte: o Autor (2021).

Utilizou-se, então, o modelo mostrado na figura 49 para o teste de desacoplamento com a aplicação descrita em 4.9. Para isso, alteraram-se duas características no modelo visto que alguns valores usados pelas classes desacopladas eram originalmente passados na compilação pelo “makefile”. Como as aplicações de Arduino desenvolvidas diretamente na IDE não fazem uso desse tipo de arquivo, essas mudanças tornam os códigos gerados funcionais no contexto. A primeira alteração deu-se pela definição do valor de “MAX_TIME_COUNT” dentro de UART, conforme figura 50. O valor usado é o mesmo que estava presente no “makefile” original. A outra ocorreu tornando-se o valor da taxa de transmissão da UART um parâmetro de entrada da função *initialize_uart*. Assim, a “main” recebe esse valor definido no “makefile” e o repassa na função que inicia a UART. Essa alteração é observada na figura 49.

Figura 50 – Definição de MAX_TIME_COUNT



Fonte: o Autor (2021).

Assim como as versões anteriores, ao analizar-se o código gerado, apenas a análise estática com misra.py apresentou avisos de possíveis erros, descritos na tabela 5.

Tabela 5 – Resultado da análise estática com misra.py do quarto código gerado.

Regra	Ocorrências
2.7	1
10.4	25
11.9	1
12.1	1
14.4	8
15.5	1
15.6	23
15.7	1
17.7	4
17.8	2
19.2	2
20.7	4
21.6	4

Fonte: o Autor (2021).

4.6 Bootloader gerado, versão 5

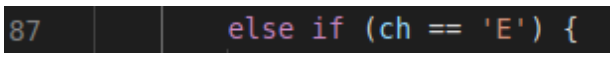
A quinta versão, e última gerada de maneira automatizada (Apêndice C), não teve alterações no modelo. Representada pela mesma imagem da figura 49. Buscou-se, nessa versão, ajustar os comportamentos opacos do modelo de modo a reduzir os alertas apontados pelo misra.py. Utilizou-se como método para isso compreender a causa de cada um dos erros. E então encontrar no código o que o causou e buscar uma solução para a correção. Dessa maneira temos por alerta:

2.7 Causado pelas estruturas geradas com base nas classes, não é possível resolver

sem alterações no gerador de código.

10.4 Causado pela diferença de tipos em comparações como na figura 51. As tentativas de corrigir o erro, alterando os tipos dos elementos envolvidos nas comparações, não surtiram efeito e aumentaram o número de ocorrências do erro. Pelo conhecimento de que a comparação é feita diretamente entre os valores armazenados endereços de memória e que todos os tipos que participam dessas comparações têm o mesmo tamanho, manteve-se da maneira original.

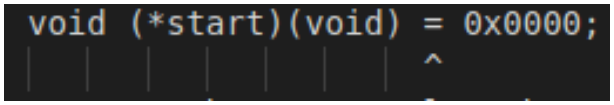
Figura 51 – Comparação que causa o erro 10.4



Fonte: o Autor (2021).

11.9 Falso positivo, o ponteiro na figura 52 não está apontando para um nulo como o erro assinala, mas sim para o endereço de memória na posição 0.

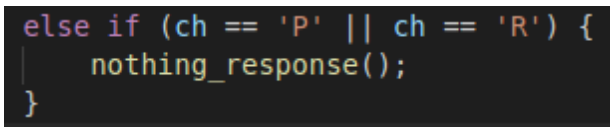
Figura 52 – Ponteiro que causa o erro 11.9



Fonte: o Autor (2021).

12.1 Solucionado com a troca do código apresentado na figura 53 pelo apresentado na figura 54.

Figura 53 – Comparação que causa o erro 12.1



Fonte: o Autor (2021).

Figura 54 – Solução do erro 12.1

```
else if (ch == 'P') {
    nothing_response();
}

else if (ch == 'R') {
    nothing_response();
}
```

Fonte: o Autor (2021).

14.4 Falso positivo, previsto na documentação, causado pela ferramenta não conseguir interpretar o retorno de operações como booleanas.

15.5 Causado pela existência de dois pontos de retorno em uma função. Não corrigido, conforme descrito em 4.5.

15.6 Causado pela falta de chaves em condicionais e iterações, resolvido com a adição das mesmas.

15.7 Solucionado com a troca do código apresentado na figura 55 pelo apresentado na figura 56.

Figura 55 – Comparação que causa o erro 15.7

```
else if (++error_count == MAX_ERROR_COUNT) {
    app_start();
}
```

Fonte: o Autor (2021).

Figura 56 – Solução do erro 15.7

```
else {
    if (++error_count == MAX_ERROR_COUNT) {
    app_start();
    }
}
```

Fonte: o Autor (2021).

17.7 Causado pelo não uso de retornos de funções e resolvido com o uso de variáveis temporárias para os retornos de funções.

17.8 Causado por alterar variáveis passadas para funções, corrigindo com o uso de variáveis temporárias com o valor copiado das de entrada das funções.

19.2 Causado pelo uso do tipo union, não pode ser solucionado sem mudanças no modelo.

20.7 Causado pelo modo como o gerador de código cria as estruturas baseadas nas classes, não é possível resolver sem alterações no gerador de código.

21.6 Causado pela adição de bibliotecas pelo gerador de código, não é possível resolver sem alterações no gerador de código.

Assim, o resultado da análise após as alterações são mostrados na tabela 6:

Tabela 6 – Resultado da análise estática com misra.py do quinto código gerado.

Regra	Ocorrências
2.7	1
10.4	27
11.9	1
14.4	8
15.5	1
19.2	2
20.7	4
21.6	4

Fonte: o Autor (2021).

4.7 Bootloader Ajustado

A sexta e última versão buscou corrigir os erros apresentados pelo misra.py no quinto modelo, ocasionados pelo gerador de código. Desse modo, editou-se manualmente o código já gerado na etapa anterior. Buscando explorar um código hipotético (Apêndice D) que poderia ser gerado a partir da última versão, descrita em 4.6, com alterações no modo como o gerador traduz o modelo.

As bibliotecas padrão do C são adicionadas automaticamente pelo gerador de código em todos os arquivos de cabeçalho quando é utilizado o estereótipo de C. Nesta versão, removeram-nas, apresentadas na figura 25.

Então, retirou-se a função de inicialização da classe ATmegaBOOT_168, apresentada na figura 57. Essa função é gerada para agir como um método construtor em classes que possuem elementos dentro da sua estrutura. Como configuraram-se todos os atributos e operações para fora das estruturas, conforme descrito em 3.5, apenas a classe ATmegaBOOT_168 apresentou elementos em sua estrutura, conforme visto na figura 58. Os elementos nessa estrutura são as outras classes que lhe pertencem pela associação de composição, conforme observado no modelo 49. A remoção dessa função é possível, pois essa estrutura nunca é utilizada.

Figura 57 – Função de inicialização de ATmegaBOOT_168

```
80    * Default value initialization
81    * @param ATmegaBOOT_168 structure instance pointer
82    * @return void
83    */
84   void ATmegaBOOT_168_init(ATmegaBOOT_168 *self);
```

Fonte: o Autor (2021).

Por fim, eliminaram-se as estruturas de UART, Led, AppStart e ATmegaBOOT_168. Essas estruturas geradas automaticamente conforme descrito em 3.5 não são utilizadas na arquitetura escolhida para o código. Portanto, puderam ser retiradas sem comprometer sua funcionalidade.

Figura 58 – Struct gerada a partir do modelo de ATmegaBOOT_168

```
52   typedef struct ATmegaBOOT_168 ATmegaBOOT_168;
53
54   struct ATmegaBOOT_168 {
55
56       length_union length_union;
57
58       flags_struct flags_struct;
59
60       address_union address_union;
61
62       UART uart;
63
64       AppStart appstart;
65
66       Led led;
67
68   };
```

Fonte: o Autor (2021).

Essas alterações no código refletiram na análise estática realizada pelo misra.py, conforme tabela 7.

Tabela 7 – Resultado da análise estática com misra.py do quinto código gerado.

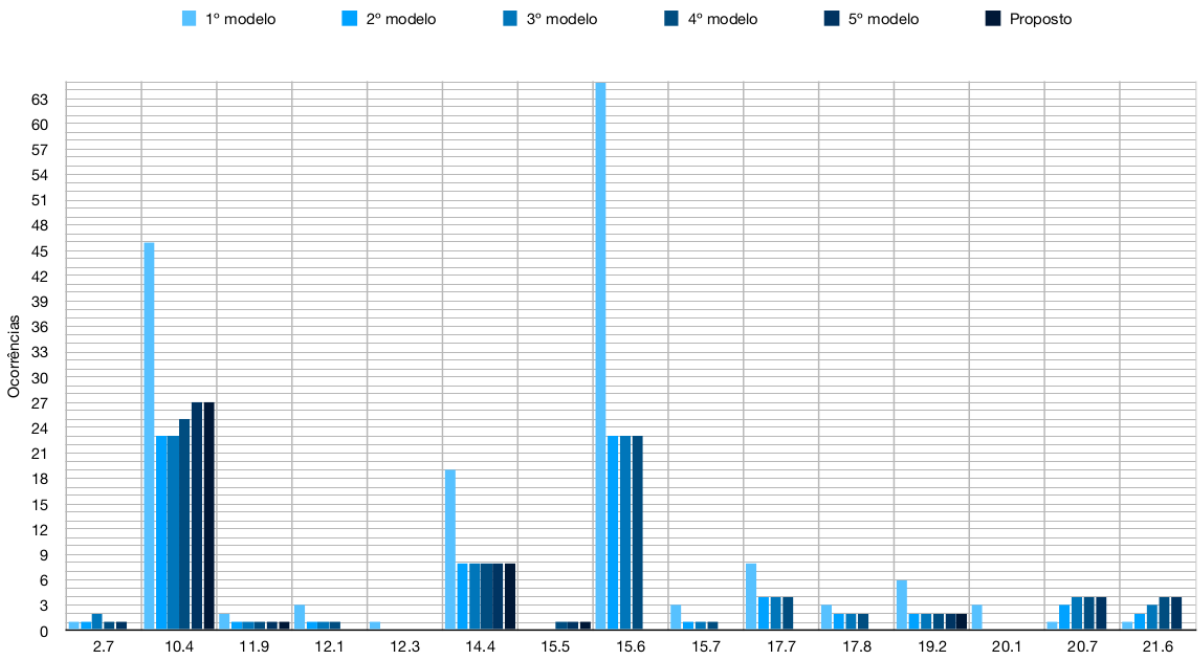
Regra	Ocorrências
10.4	27
11.9	1
14.4	8
15.5	1
19.2	2

Fonte: o Autor (2021).

4.8 Considerações sobre a análise estática

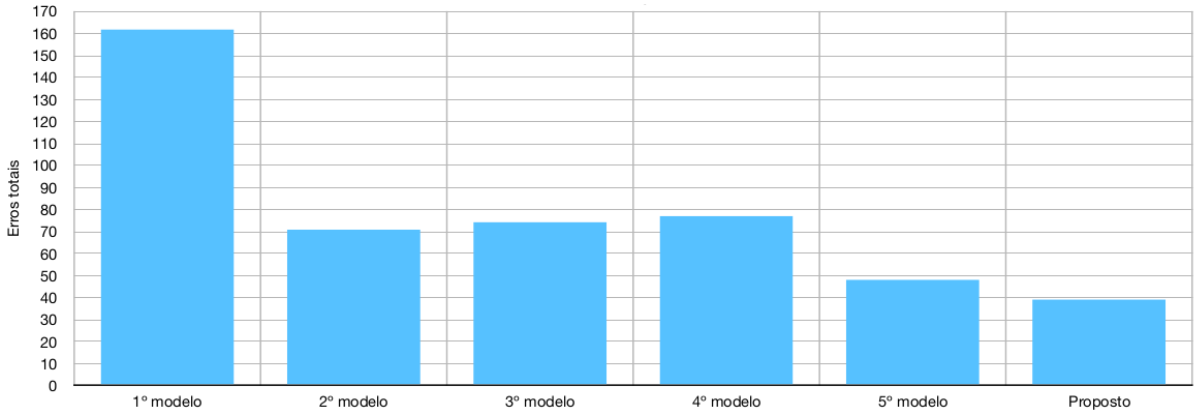
Para melhor analisar os dados das análises estáticas, são apresentados os gráficos de número de ocorrências de cada erro em cada versão do código (figura 59) e de número de ocorrências totais de erros em cada versão do código (figura 60).

Figura 59 – Ocorrências de cada erro por versão



Fonte: o Autor (2021).

Figura 60 – Ocorrência total de erros por versão



Fonte: o Autor (2021).

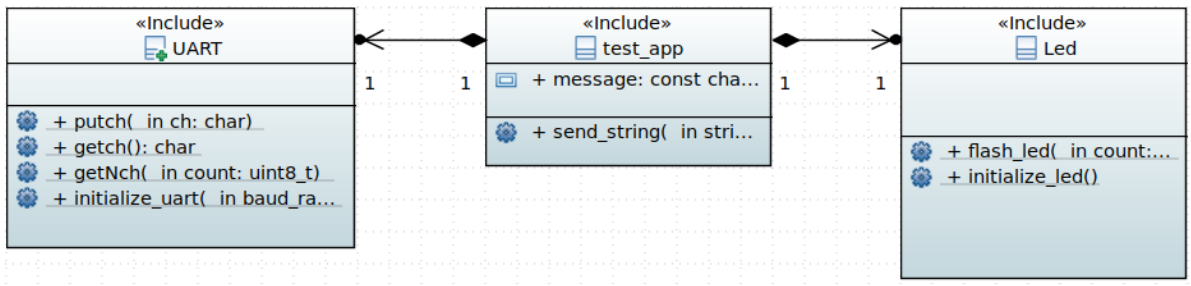
Tendo como critério as diretrizes do padrão de código MISRA C, observa-se uma melhora na qualidade do código com o refinamento dos modelos. Ao comparar-se o primeiro código gerado com a versão proposta, nota-se que apenas um erro 15.5 aumentou. Isso foi causado por uma decisão que visou melhorar o desacoplamento das atribuições do bootloader sem se distanciar muito do código original. Conforme descrito

em 4.5. Em uma nova versão, dessa vez focada em corrigir aspectos comportamentais do modelo, pôde eliminar esse erro, bem como reduzir ou eliminar outros erros restantes.

4.9 Aplicação

Conforme descrito em 4.5, as atribuições de manipulação da UART e do LED possibilitaram o desenvolvimento de uma aplicação com o uso dos códigos gerados a partir das classes UART e Led com o objetivo de mostrar a usabilidade desses códigos fora do contexto do bootloader. Seu modelo é observado na figura 61.

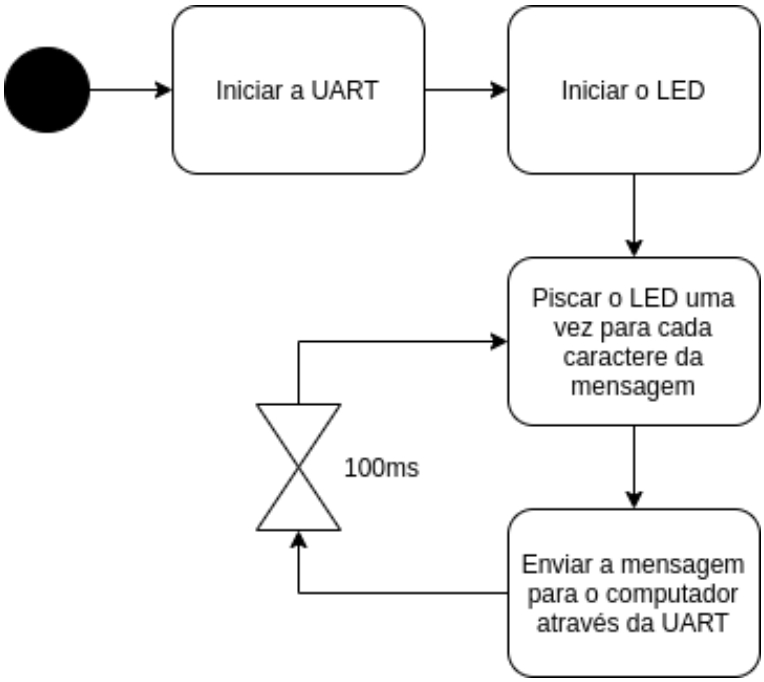
Figura 61 – Modelo da aplicação de teste



Fonte: o Autor (2021).

Construída com o objetivo de ser compatível com as versões de UART e Led descritas em 4.5, 4.6 e 4.7, a aplicação contém uma mensagem preestabelecida e tem o comportamento descrito pelo diagrama da figura 62.

Figura 62 – Diagrama de atividade da aplicação



Fonte: o Autor (2021).

A aplicação deveria ser carregada no Arduíno pela IDE, conforme descrito em 3.2.2, e a IDE usa como arquivo principal do projeto a extensão “.ino”. Isso implicou uma série de diferenças entre o código para a IDE e um código em C convencional como a maneira de referenciar cabeçalhos de outros arquivos. Tais características exigiriam alterações na maneira que o gerador (3.5) constrói os códigos, para que fosse possível utilizá-lo para gerar a aplicação para o Arduino. Por conta disso, junto da simplicidade da aplicação, fez-se a escolha de escrever o código manualmente, resultando no código apresentado na figura 63.

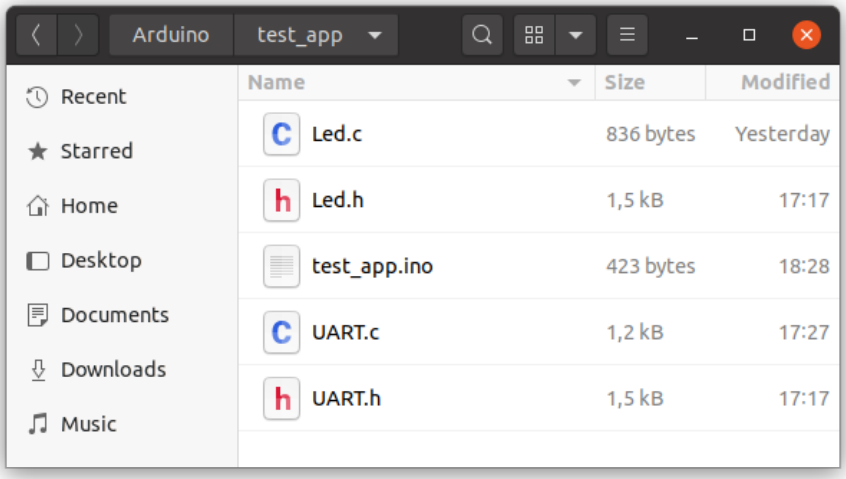
Figura 63 – Código da aplicação

```
extern "C"{
#include "Led.h"
#include "UART.h"
}

const char message[]="UFSC CTJ";

void send_string(const char * string);

void setup() {
  initialize_led();
  initialize_uart(57600);
}

void loop() {
  send_string(message);
  delay(1000);
}


void send_string(const char * string){
  int str_length = strlen(string);
  flash_led(str_length);
  for (int i=0; i<str_length;i++){
    putch(string[i]);
  }
  putch('\n');
}
```

Fonte: o Autor (2021).

Para utilização de Led e UART, mantiveram-se os arquivos “.c” e “.h” da versão desejada desses elementos, no mesmo diretório o arquivo da aplicação (”.ino”), conforme figura 64.

Figura 64 – Diretório da aplicação



Fonte: o Autor (2021).

Após o carregamento da aplicação no Arduino, observou-se seu correto funcionamento. Isso pelo comportamento do LED e também da mensagem recebida pelo monitor serial, conforme figura 65. Todas as versões de Led e UART a partir de 4.5 apresentaram compatibilidade com a aplicação. Isso demonstrou o correto desacoplamento desses elementos do bootloader.

Figura 65 – Mensagens recebidas via USB com o uso da UART

Fonte: o Autor (2021).

5 CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo apresentar e testar o uso de técnicas de desenvolvimento dirigido a modelos no desenvolvimento de bootloaders para microcontroladores. Buscou-se uma versão funcional com um maior desacoplamento entre as atribuições.

Inicialmente, alterou-se manualmente o código-fonte do bootloader fornecido pelo fabricante para a placa Arduino Duemilanove na sua versão com o microcontrolador ATmega328P. Essa alteração buscou eliminar partes do código-fonte não relevantes para essa versão de placa. O bootloader foi testado por meio da gravação no microcontrolador com o auxílio de uma placa desenvolvida para esse fim. Em seguida, armazenou-se uma aplicação de exemplo diretamente pela placa Duemilanove.

Desenhou-se, então, no software Papyrus, um modelo em UML representando tal código obtido na etapa anterior. Desse modelo originou-se um novo código, gerado pela ferramenta Papyrus Software Designer, com um único desacoplamento alcançado entre os componentes dessa versão, causado por limitações da ferramenta. Testou-se seu funcionamento da mesma maneira que o código da versão anterior. Finalmente, analisou-se estaticamente o código com as ferramentas Flawfinder e CppCheck.

As etapas utilizadas para o desenvolvimento da primeira versão modelada repetiram-se em outras duas versões sendo essas responsáveis pelo desacoplamento das atribuições do bootloader, o controle da inicialização da aplicação, o controle da UART e do LED presente na placa.

Melhorou-se o modelo numa quarta versão. Nela também alteraram-se alguns aspectos que permitiram o uso dos componentes desacoplados em outras aplicações. A partir dessa versão, passou-se a testar o desacoplamento por meio do uso de uma aplicação com esse fim.

Essa aplicação, utilizada para realizar o teste do desacoplamento, utilizou os componentes de controle da UART e do LED tendo como comportamento piscar o LED um determinado número de vezes e enviar uma frase via UART.

Fez-se ainda uma quinta versão alterando-se alguns aspectos do modelo anterior. Isso buscando aumentar a conformidade do código com o padrão de código MISRA C.

Finalmente, propôs-se uma nova versão feita a partir da edição manual da quinta versão. Também buscando uma maior conformidade com o MISRA C. Essa versão não pôde ser alcançada com a geração automática, pois exigiria alterações na

maneira como a ferramenta, Papyrus Software Designer, traduz os modelos para código. Submeteu-se também essa versão às análises estáticas e testes práticos realizados nas anteriores.

Demonstrou-se, assim, a aplicabilidade de métodos de desenvolvimento dirigido a modelo para bootloader de microcontroladores, gerando-se um código-fonte com maior desacoplamento entre seus componentes, reutilizáveis em outra aplicação. Também observou-se uma melhoria na qualidade do código, pelos parâmetros do padrão MISRA C, com pequenas alterações realizadas entre as versões dos modelos.

Ocorreram, porém, algumas limitações no desenvolvimento desse trabalho. Um exemplo é a necessidade do uso de comportamentos opacos no gerador de códigos utilizado. Também destaca-se a utilização de apenas métodos estáticos para analisar o código. Desse modo, sugere-se como exploração para trabalhos futuros:

a) O uso de uma ferramenta de geração de código que lidem com modelos comportamentais ao invés de comportamentos opacos.
b) A avaliação dos códigos gerados por outros métodos de análise que considerem outros fatores importantes para sistemas embarcados como o tamanho e o tempo de execução do bootloader.
c) A implementação de alterações no gerador de códigos utilizado, permitindo o uso de modelos comportamentais e também permitindo alterar os aspectos editados manualmente para o bootloader sugerido.

REFERÊNCIAS

ARDUINO. **Arduino 2009 schematic**. 2008. Disponível em: https://www.arduino.cc/en/uploads/Main/arduino-duemilanove-schematic.pdf. Acessado em 10 jun. 2020.

ARDUINO. **Arduino UNO Reference Design**. 2010. Disponível em: https://www.arduino.cc/en/uploads/Main/arduino-uno-schematic.pdf. Acessado em 10 jun. 2020.

ARDUINO. **From Arduino to a Microcontroller on a Breadboard**. 2018. Disponível em: https://www.arduino.cc/en/Tutorial/BuiltInExamples/ArduinoToBreadboard. Acessado em 10 jun. 2020.

ARDUINO. **What is Arduino?** 2018. Disponível em: https://www.arduino.cc/en/Guide/Introduction. Acessado em 16 jul 2020.

ATMEL CORPORATION. Stk500 communication protocol. Atmel Corporation, San Jose, CA, 2003.

ATMEL CORPORATION. Atmega328p datasheet: 8-bit avr microcontroller with 32k bytes in-system programmable flash. Atmel Corporation, San Jose, CA, 2015. Rev.: 7810D–AVR–01/15.

BAI, Y.; CHNG, E. S.; BHANU, G. P. An mcu description methodology for initialization code generation software. In: **2007 International Conference on Parallel and Distributed Systems**. [S.l.: s.n.], 2007. p. 1–7.

BENINGO, J. Bootloader design for microcontrollers in embedded systems. **Embedded Software Design Techniques**, Beningo engineering, jun. 2015.

BHULLAR, N. S.; CHHABRA, B.; VERMA, D. A. Exploration of uml diagrams based code generation methods. In: **In Proceedings of the 2016 INTERNATIONAL CONFERENCE ON INVENTIVE COMPUTATION TECHNOLOGIES (ICICT)**. [S.l.]: IEEE, 2016.

CHELF, B.; EBERT, C. Ensuring the integrity of embedded software with static code analysis. **IEEE Software**, IEEE, v. 26, p. 96 – 99, abr. 2009.

CPPCHECK TEAM. Cppcheck manual. Cppcheck team, mar. 2021. Ver. 2.5.

DEEPAK, R. **MACHINE VISION AND ITS APPLICATION IN UNMANNED AERIAL VEHICLE**. Tese (Doutorado), 05 2016.

DUKISH, B. **Coding the Arduino**. [S.l.]: Apress, 2018.

ECLIPSE. **Papyrus Software Designer**. 2021. Disponível em: https://marketplace.eclipse.org/content/papyrus-software-designer. Acessado em 20 ago. 2021.

EGELE, M. et al. A survey on automated dynamic malware-analysis techniques and tools. **ACM Computing Surveys**, ACM Computing, v. 44, n. 6, p. 42, fev. 2012.

FLAWFINDER. **Flawfinder**. 2017. Disponível em: https://dwheeler.com/flawfinder/flawfinder.pdf. Acessado em 25 ago. 2021.

FOWLER, M. **UML Distilled**: A brief guide to the standard object modeling language. [S.l.]: Addison-Wesley Professional, 2003.

FUTURE TECHNOLOGY DEVICES INTERNATIONAL LIMITED. Future technology devices international ltd. ft232r usb uart ic datasheet. Future Technology Devices International Limited, mai. 2020. Ver.: 2.16.

GOMES, F. **Falta de processadores está a estagnar produção automóvel**. 2021. **Razão automóvel**. Disponível em: https://www.razaoautomovel.com/2021/02/falta-de-processadores-esta-a-estagnar-producao-automovel. Acessado em 09 jul. 2020.

HEATH, S. **Microprocessor Architectures**: Risc, cisc and dsp. 2. ed. Jordan Hill, Oxford: Newnes, 1995.

KLEPPE, A.; WARMER, J.; BAST, W. **MDA Explained**: The model driven architecture™: Practice and promise. 1. ed. Boston, MA: Addison-Wesley Professional, 2003.

KRUNIć, M. et al. Automatic source code generation of peripheral hardware modules firmware. In: **2013 21st Telecommunications Forum Telfor (TELFOR)**. [S.l.: s.n.], 2013. p. 833–836.

LENNIS, L.; AEDO, J. Generation of efficient embedded c code from uml / marte models. In: . [S.l.: s.n.], 2013.

MISCHIE, S.; PAZSITKA, R. Designing a msp430 bootloader. In: **In Proceedings of the 2019 INTERNATIONAL CONFERENCE ON APPLIED ELECTRONICS (AE)**. [S.l.]: IEEE, 2019.

MOREIRA, T. G. et al. Automatic code generation for embedded systems: From uml specifications to vhdl code. In: **2010 8th IEEE International Conference on Industrial Informatics**. [S.l.: s.n.], 2010. p. 1085–1090.

Motor Industry Software Reliability Association. Misra c:2012: guidelines for the use of the c language in critical systems. MIRA, mar. 2013.

Motorola Inc. Spi block guide. Freescale Semiconductor, jul. 2004.

NASIR, S. Z. **Introduction to Arduino Duemilanove**. 2018. **the engineering projects**. Disponível em: https://www.theengineeringprojects.com/2018/10/introduction-to-arduino-duemilanove.html. Acessado em 18 jul. 2020.

NETWORK, E. N. **Microcontroller Market Revenue To Cross $20 BIllion By 2027: Global Market Insights**. 2021. **Electronics B2B**. Disponível em: https://www.electronicsb2b.com/industry-buzz/microcontroller-market-revenue-to-cross-20-billion-by-2027-global-market-insights/. Acessado em 18 jul. 2020.

NOVIKOV, A. S. et al. Detecting the use of unsafe data in software of embedded systems by means of static analysis methodology. In: **In Proceedings of the 7TH MEDITERRANEAN CONFERENCE ON EMBEDDED COMPUTING**. [S.l.]: IEEE, 2018.

Object Management Group INC. Mof model to text transformation language. Object Management Group inc., jan. 2008.

OBJECT MANAGEMENT GROUP, INC. Omg® unified modeling language® (omg uml®). Object Management Group, Inc, dez. 2017. Ver. 2.5.1.

PAPYRUS. **Papyrus overview**. 2019. Disponível em: https://wiki.eclipse.org/Papyrus#Overview. Acessado em 20 jun. 2020.

PAPYRUS. **Papyrus**. 2020. Disponível em: https://www.eclipse.org/papyrus/. Acessado em 10 ago. 2021.

PEÑA, E.; LEGASPI, M. G. Uart: A hardware communication protocol understanding universal asynchronous receiver/transmitter. **Analog Dialogue**, Analog Devices, v. 54, n. 4, dez. 2020.

PICKERILL, S. **Spi diagram**. 2008. **CENTER FOR ROBOTICS AND BIOSYSTEMS**. Disponível em: http://hades.mech.northwestern.edu/index.php/File:Spi-diagram.png. Acessado em 18 jul. 2020.

REINBACHER, T. et al. Refining assembly code static analysis for the intel mcs-51 microcontroller. In: **In Proceedings of the 2009 IEEE INTERNATIONAL SYMPOSIUM ON INDUSTRIAL EMBEDDED SYSTEMS**. [S.l.: s.n.], 2009.

RUMPE, B. **Agile Modeling with UML: Code Generation, Testing, Refactoring**. 1. ed. New York, NY: Springer, 2017.

SOMMERVILLE, I. **Engenharia De Software**. 10. ed. São Paulo: Pearson Universidades, 2019.

STRICKLAND, J. R. **Junk Box Arduino**: Ten projects in upcycled electronics. [S.l.]: Springer Science+Business Media, 2016.

TAN, L.; YANG, Z.; XIE, J. Ocl constraints automatic generation for uml class diagram. In: **In Proceedings of the 2010 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND SERVICE SCIENCES**. [S.l.]: IEEE, 2010.

USMAN, M.; NADEEM, A.; KIM, T. hoon. Ujector: A tool for executable code generation from uml models. In: **In Proceedings of the 2008 ADVANCED SOFTWARE ENGINEERING AND ITS APPLICATIONS**. [S.l.]: IEEE, 2008.

UZLU, T.; SAYKOL, E. Utilizing rust programming language for efi-based bootloader design. In: **In Proceedings of the 2ND INTERNATIONAL CONFERENCE ON RECENT TRENDS AND APPLICATIONS IN COMPUTER SCIENCE AND INFORMATION TECHNOLOGY**. [S.l.]: computer science bibliography, 2016.

WILLIAMS, E. **Make: AVR Programming**: Learning to write software for hardware. 1. ed. Sebastopol, CA: Maker Media, 2014.

APÊNDICE A - CÓDIGO RESULTANTE DA REMOÇÃO DAS PARTES NÃO UTILIZADAS PARA O ATMEGA328P NA PLACA DUEMILANOVE DO BOOTLOADER ORIGINAL

ATmegaBOOT_168.c :

```
1
2  #include <inttypes.h>
3  #include <avr/io.h>
4  #include <avr/pgmspace.h>
5  #include <avr/interrupt.h>
6  #include <avr/wdt.h>
7  #include <util/delay.h>
8  #include <avr/eeprom.h>
9
10 #define MAX_ERROR_COUNT 5
11
12 /* Versoes de hardware e software */
13 #define HW_VER    0x02
14 #define SW_MAJOR 0x01
15 #define SW_MINOR 0x10
16
17 // Portas para comunicacao UART do 328P:
18 #define BL_DDR  DDRD     // The Port D Data Direction Register
19 #define BL_PORT PORTD    // The Port D Data Register
20 #define BL_PIN  PIND     // The Port D Input Pins Address
21 #define BL      PIND6    //pino D6
22
23 // Led integrado:
24 #define LED_DDR  DDRB    // The Port B Data Direction Register
25 #define LED_PORT PORTB   // The Port B Data Register
26 #define LED_PIN  PINB    // The Port B Input Pins Address
27 #define LED      PINB5   //pino B5
28
29 // Tres bytes de assinatura definidos pelo datasheet do atmega:
30 #define SIG1   0x1E   //ATmega
31 #define SIG2   0x95     //328
32 #define SIG3   0x0F     //P
33
34 #define PAGE_SIZE 0x40U //64 words
35
36 // Prptotipos de funcoes:
```

```
37 void putch(char);
38 char getch(void);
39 void getNch(uint8_t);
40 void byte_response(uint8_t);
41 void nothing_response(void);
42 void flash_led(uint8_t);
43
44 // Variaveis:
45 union address_union {
46   uint16_t word;
47   uint8_t  byte[2];
48 } address; //essa variavel vai armazenar o endereco para a proxima operacao
     de leitura ou escrita
49
50 union length_union {
51   uint16_t word;
52   uint8_t  byte[2];
53 } length; //essa variavel armazena o byte inicial e final de um bloco a ser
     gravado
54
55 struct flags_struct {
56   unsigned eeprom : 1;
57   unsigned rampz  : 1;
58 } flags; //salva algumas flags uteis
59
60 uint8_t buff[256]; // buffer de armazenamento
61 uint8_t address_high;
62 uint8_t pagesz=0x80;
63
64 uint8_t bootuart = 0;
65 uint8_t error_count = 0; //contador de erros
66 uint8_t ch,ch2;
67 uint16_t w;
68
69 void (*app_start)(void) = 0x0000; //Isso seta um ponteiro de funcao para a
     posicao 0
70 //isso faz com que "app_start(); // -> Passa o controle para a aplicacao "
     transfira o controle do bootloader para a aplicacao
71
72 int main(void)
73 {
74   asm volatile("nop\n\t"); //Instrucao em assembly que da o delay de um
     ciclo
75
76     //inicializa UART e LED
77   UBRR0L = (uint8_t)(F_CPU/(BAUD_RATE*16L)-1);
78   UBRR0H = (F_CPU/(BAUD_RATE*16L)-1) >> 8;
```

```
79  UCSR0B = (1<<RXEN0) | (1<<TXEN0);
80  UCSR0C = (1<<UCSZ00) | (1<<UCSZ01);
81  DDRD &= ~_BV(PIND0);
82  PORTD |= _BV(PIND0);
83  LED_DDR |= _BV(LED);
84  LED_PORT |= _BV(LED);
85  flash_led(3);
86  //Loop 'infinito'
87  //o loop ficara se repetindo ate que app_start() seja chamado
88  for (;;) {
89
90    //Le um char da UART
91  ch = getch();
92
93  if(ch=='0') { //0x30 - Cmnd_STK_GET_SYNC - Get Synchronization
94    nothing_response();
95  }
96
97  else if(ch=='1') { //0x31 - Cmnd_STK_GET_SIGN_ON - Check if Starterkit
      Present
98    if (getch() == ' ') { //0x20 - Sync_CRC_EOP - "Command terminator"
99      putch(0x14); // toda resposta inicia em 0x14
100     putch('A');
101     putch('V');
102     putch('R');
103     putch(' ');
104     putch('I');
105     putch('S');
106     putch('P');
107     putch(0x10); // toda mensagem termina em 0x10
108   } else {
109     if (++error_count == MAX_ERROR_COUNT)
110       app_start(); // -> Passa o controle para a aplicacao
111   }
112 }
113
114 else if(ch=='@') {  //0x40 - Cmnd_STK_SET_PARAMETER - Set the value of a
      valid parameter in the STK500 starterkit
115   ch2 = getch();
116   if (ch2>0x85) getch(); //uma serie de parametros esta definida entre 0
      x80 e 0x96, se >85 enviara o Resp_STK_INSYNC
117   nothing_response();
118 }
119
120 else if(ch=='A') { //0x41 - Cmnd_STK_GET_PARAMETER - Get the value of a
      valid parameter from the STK500 starterki
121   ch2 = getch();
```

```
122     if(ch2==0x80) byte_response(HW_VER);     // 0x80 - Parm_STK_HW_VER -
        This parameter defines the version of the starterkit hardware
123     else if(ch2==0x81) byte_response(SW_MAJOR); // 0x81 - Parm_STK_SW_MAJOR
        - The major version of the starterkit MCU software
124     else if(ch2==0x82) byte_response(SW_MINOR); // 0x82 - Parm_STK_SW_MINOR
        - The minor version of the starterkit MCU software
125     else if(ch2==0x98) byte_response(0x03);    // Unknown but seems to be
        required by avr studio 3.56
126     else byte_response(0x00);        // Covers various unnecessary responses
        we don't care about
127   }
128
129   else if(ch=='B') { //0x42 - Cmnd_STK_SET_DEVICE - Set the device
        Programming parameters for the current device
130     getNch(20);
131     nothing_response();
132   }
133
134   else if(ch=='E') { //0x45 - Cmnd_SET_DEVICE_EXT - Set extended
        programming parameters for the current device
135     getNch(5);
136     nothing_response();
137   }
138
139   else if(ch=='P' || ch=='R') { //0x50 - Cmnd_STK_ENTER_PROGMODE - Enter
        Programming mode for the selected device, 0x52 - Cmnd_STK_CHIP_ERASE -
        Erase device
140     nothing_response();
141   }
142
143   else if(ch=='Q') { //0x51 - Cmnd_STK_LEAVE_PROGMODE - Leave programming
        mode
144     nothing_response();
145   }
146
147
148   /* Set address, little endian. EEPROM in bytes, FLASH in words  */
149   /* Perhaps extra address bytes may be added in future to support > 128kB
        FLASH.  */
150   /* This might explain why little endian was used here, big endian used
        everywhere else.  */
151   else if(ch=='U') { //0x55 - Cmnd_STK_LOAD_ADDRESS - Load 16-bit address
        down to starterkit
152     address.byte[0] = getch();
153     address.byte[1] = getch();
154     nothing_response();
155   }
```

```
156
157  /* Universal SPI programming command, disabled.  Would be used for fuses
      and lock bits.  */
158  else if(ch=='V') { //0x56 - Cmnd_STK_UNIVERSAL - Program Fuse bits (Fuse
      bits are 1 bit controls in a programmmable component that are activated
      or de-actovated (fused) once when programming the device to define a
      certain behavior.)
159    if (getch() == 0x30) { //0x30 - Cmnd_STK_GET_SYNC - Use this command to
      try to regain synchronization when sync is lost
160      getch();
161      ch = getch();
162      getch();
163      if (ch == 0) {
164        byte_response(SIG1);
165      } else if (ch == 1) {
166        byte_response(SIG2);
167      } else {
168        byte_response(SIG3);
169      }
170    } else {
171      getNch(3);
172      byte_response(0x00);
173    }
174  }
175
176  /* Write memory, length is big endian and is in bytes  */
177  else if(ch=='d') { //0x64 - Cmnd_STK_PROG_PAGE - Load 16-bit address down
      to starterkit
178    length.byte[1] = getch();
179    length.byte[0] = getch();
180    flags.eeprom = 0;
181    if (getch() == 'E') flags.eeprom = 1; //0x45 - Cmnd_SET_DEVICE_EXT -
     Set extended programming parameters for the current device
182    for (w=0;w<length.word;w++) {
183      buff[w] = getch();                            // Store data in buffer,
     can't keep up with serial data stream whilst programming pages
184    }
185    if (getch() == ' ') { //0x20 - Sync_CRC_EOP - "Command terminator"
186      if (flags.eeprom) {                    //Write to EEPROM one byte at a
     time
187        address.word <<= 1;
188        for(w=0;w<length.word;w++) {
189
190          while(EECR & (1<<EEPE));
191          EEAR = (uint16_t)(void *)address.word;
192          EEDR = buff[w];
193          EECR |= (1<<EEMPE);
```

```
194           EECR |= (1<<EEPE);
195
196           address.word++;
197         }
198       }
199       else {                    //Write to FLASH one page at a time
200         // if (address.byte[1]>127) address_high = 0x01;  //Only possible
    with m128, m256 will need 3rd address byte. FIXME
201         // else address_high = 0x00;
202         address.word = address.word << 1;         //address * 2 -> byte
    location
203         if ((length.byte[0] & 0x01)) length.word++; //Even up an odd number
     of bytes
204         cli();          //Disable interrupts, just to be sure
205         while(bit_is_set(EECR,EEPE));      //Wait for previous EEPROM writes
     to complete
206         asm volatile(
207             "clr r17   \n\t" //page_word_count
208             "lds r30,address \n\t" //Address of FLASH location (in bytes)
209             "lds r31,address+1 \n\t"
210             "ldi r28,lo8(buff) \n\t" //Start of buffer array in RAM
211             "ldi r29,hi8(buff) \n\t"
212             "lds r24,length  \n\t" //Length of data to be written (in bytes)
213             "lds r25,length+1  \n\t"
214             "length_loop:    \n\t" //Main loop, repeat for number of words
    in block
215             "cpi r17,0x00  \n\t" //If page_word_count=0 then erase page
216             "brne  no_page_erase \n\t"
217             "wait_spm1:    \n\t"
218             "lds r16,%0    \n\t" //Wait for previous spm to complete
219             "andi  r16,1           \n\t"
220             "cpi r16,1           \n\t"
221             "breq  wait_spm1       \n\t"
222             "ldi r16,0x03  \n\t" //Erase page pointed to by Z
223             "sts %0,r16    \n\t"
224             "spm     \n\t"
225             "wait_spm2:    \n\t"
226             "lds r16,%0    \n\t" //Wait for previous spm to complete
227             "andi  r16,1           \n\t"
228             "cpi r16,1           \n\t"
229             "breq  wait_spm2       \n\t"
230
231             "ldi r16,0x11  \n\t" //Re-enable RWW section
232             "sts %0,r16    \n\t"
233             "spm     \n\t"
234             "no_page_erase:    \n\t"
235             "ld  r0,Y+   \n\t" //Write 2 bytes into page buffer
```

```
236             "ld  r1,Y+   \n\t"
237
238             "wait_spm3:    \n\t"
239             "lds r16,%0    \n\t" //Wait for previous spm to complete
240             "andi  r16,1           \n\t"
241             "cpi r16,1           \n\t"
242             "breq  wait_spm3       \n\t"
243             "ldi r16,0x01  \n\t" //Load r0,r1 into FLASH page buffer
244             "sts %0,r16    \n\t"
245             "spm     \n\t"
246
247             "inc r17   \n\t" //page_word_count++
248             "cpi r17,%1          \n\t"
249             "brlo  same_page \n\t" //Still same page in FLASH
250             "write_page:   \n\t"
251             "clr r17   \n\t" //New page, write current one first
252             "wait_spm4:    \n\t"
253             "lds r16,%0    \n\t" //Wait for previous spm to complete
254             "andi  r16,1           \n\t"
255             "cpi r16,1           \n\t"
256             "breq  wait_spm4       \n\t"
257             "ldi r16,0x05  \n\t" //Write page pointed to by Z
258             "sts %0,r16    \n\t"
259             "spm     \n\t"
260             "wait_spm5:    \n\t"
261             "lds r16,%0    \n\t" //Wait for previous spm to complete
262             "andi  r16,1           \n\t"
263             "cpi r16,1           \n\t"
264             "breq  wait_spm5       \n\t"
265             "ldi r16,0x11  \n\t" //Re-enable RWW section
266             "sts %0,r16    \n\t"
267             "spm     \n\t"
268             "same_page:    \n\t"
269             "adiw  r30,2   \n\t" //Next word in FLASH
270             "sbiw  r24,2   \n\t" //length-2
271             "breq  final_write \n\t" //Finished
272             "rjmp  length_loop \n\t"
273             "final_write:    \n\t"
274             "cpi r17,0   \n\t"
275             "breq  block_done  \n\t"
276             "adiw  r24,2   \n\t" //length+2, fool above check on length
    after short page write
277             "rjmp  write_page  \n\t"
278             "block_done:   \n\t"
279             "clr __zero_reg__  \n\t" //restore zero register
280             : "=m" (SPMCSR) : "M" (PAGE_SIZE) : "r0","r16","r17","r24","r25"
    ,"r28","r29","r30","r31"
```

```
281           );
282       }
283       putch(0x14); // 0x14 Resp_STK_INSYNC is sent after Sync_CRC_EOP has
    been received.
284       putch(0x10); // 0x10 - Resp_STK_OK
285     } else {
286       if (++error_count == MAX_ERROR_COUNT)
287         app_start();
288     }
289   }
290
291   /* Read memory block mode, length is big endian.  */
292   else if(ch=='t') { //0x74 - Cmnd_STK_READ_PAGE - Read a block of data
    from FLASH or EEPROM of the current device
293     length.byte[1] = getch();
294     length.byte[0] = getch();
295     address.word = address.word << 1;          // address * 2 -> byte
    location
296     if (getch() == 'E') flags.eeprom = 1;  //0x45 - Cmnd_SET_DEVICE_EXT -
    Set extended programming parameters for the current device
297     else flags.eeprom = 0;
298     if (getch() == ' ') {  //0x20 - Sync_CRC_EOP - "Command terminator"
                  // Command terminator
299       putch(0x14); // 0x14 Resp_STK_INSYNC - is sent after Sync_CRC_EOP has
    been received.
300       for (w=0;w < length.word;w++) {            // Can handle odd and even
    lengths okay
301         if (flags.eeprom) {                         // Byte access EEPROM
    read
302           while(EECR & (1<<EEPE));
303           EEAR = (uint16_t)(void *)address.word;
304           EECR |= (1<<EERE);
305           address.word++;
306         }
307         else {
308
309           if (!flags.rampz) putch(pgm_read_byte_near(address.word));
310           address.word++;
311         }
312       }
313       putch(0x10); // 0x10 - Resp_STK_OK
314     }
315   }
316
317   /* Get device signature bytes  */
318   else if(ch=='u') { //0x75 - Cmnd_STK_READ_SIGN - Read signature bytes
319     if (getch() == ' ') { //0x20 - Sync_CRC_EOP - "Command terminator"
```

```
320       putch(0x14); // 0x14 Resp_STK_INSYNC - is sent after Sync_CRC_EOP has
      been received.
321       putch(SIG1);
322       putch(SIG2);
323       putch(SIG3);
324       putch(0x10); // 0x10 - Resp_STK_OK
325     } else {
326       if (++error_count == MAX_ERROR_COUNT)
327         app_start(); // -> Passa o controle para a aplicacao
328     }
329   }
330
331   else if(ch=='v') { //0x76 - Cmnd_STK_READ_OSCCAL - Read Oscillator
      calibration byte
332     byte_response(0x00);
333   }
334
335
336   else if (++error_count == MAX_ERROR_COUNT) {
337     app_start();
338   }
339   } //fim do loop 'infinito'
340 LED_PORT &= ~_BV(LED);
341 }
342
343
344 // Funcoes:
345
346
347 //    Envia um char via UART
348 void putch(char ch){
349   while (!(UCSR0A & _BV(UDRE0))); // o bit UDRE0 indica que o buffer de
      transmissao pode receber novos dados, se 1 pode enviar.
350   UDR0 = ch;
351 }
352
353
354 //    Recebe um char via UART
355 char getch(void){
356     uint32_t count = 0;
357   while(!(UCSR0A & _BV(RXC0))){ //O bit RCXCn determina que existem dados
      nao lidos no buffer de recebimento do UART0, se 0 entao nao tem mais
      dados
358     count++;
359     if (count > 5)
360       app_start(); // Passa o controle para a aplicacao
361   }
```

```
362   return UDR0; // Retorna o conteudo do buffer de dados do UART0 (Leitura e
        escrita tem esse mesmo endereco)
363 }
364
365
366 //   Recebe 'count' chars via UART 0
367 void getNch(uint8_t count)
368 {
369   while(count--) {
370     getch();
371   }
372 }
373
374
375 \\ Envia 'val' via UART
376 void byte_response(uint8_t val)
377 {
378   if (getch() == ' ') {
379     putch(0x14); // 0x14 Resp_STK_INSYNC - is sent after Sync_CRC_EOP has
      been received.
380     putch(val);
381     putch(0x10); // 0x10 - Resp_STK_OK
382   } else {
383     if (++error_count == MAX_ERROR_COUNT)
384       app_start(); // -> Passa o controle para a aplicacao
385   }
386 }
387
388 // Usada para sincronizacao
389 void nothing_response(void)
390 {
391   if (getch() == ' ') {
392     putch(0x14); // 0x14 Resp_STK_INSYNC - is sent after Sync_CRC_EOP has
      been received.
393     putch(0x10); // 0x10 - Resp_STK_OK
394   } else {
395     if (++error_count == MAX_ERROR_COUNT)
396       app_start(); // -> Passa o controle para a aplicacao
397   }
398 }
399
400 // Pisca o led 'count' vezes
401 void flash_led(uint8_t count)
402 {
403   while (count--) {
404     LED_PORT |= _BV(LED);
405     _delay_ms(50);
```

```
406     LED_PORT &= ~_BV(LED);
407     _delay_ms(300);
408   }
409 }
```

APÊNDICE B - CÓDIGO GERADO À PARTIR DO PRIMEIRO MODELO

address_union.h :

```
1  // ---------------------------------------------------------
2  // Code generated by Papyrus C
3  // ---------------------------------------------------------
4
5  // This template is called by the main module file
6  /*
7   * File generated from the BootLoader_Vsimp::address_union uml class
8   * Generated by the Papyrus C Generator (CEA LIST)
9   *
10  */
11
12 #ifndef ADDRESS_UNION_H_
13 #define ADDRESS_UNION_H_
14
15 // Explicit import of the class
16 /*
17  * -------------------- Includes and declares --------------------
18  */
19 // Derived includes
20 // End of Derived includes
21 // Derived declarations
22 // End of Derived declares
23
24 // Std headers
25 #include <stdio.h>
26 #include <stdlib.h>
27 #include <stdint.h>
28 // End of Std headers
29
30 // header body
31 /*
32  * ------------------- Public Class Description ------------------
33  */
34 //  Structure
35 typedef union address_union address_union;
36
37 union address_union {
38
```

```
39   uint16_t word;
40
41   uint8_t byte[2];
42 };
43
44 // Property initialisation declarations
45
46 // ----------------------------------Public Global VariableDescription
     ----------------------------------
47 // global variable declaration
48
49 uint16_t word;
50 // global variable declaration
51
52 uint8_t byte[2];
53
54 #endif /*ADDRESS_UNION_H_*/
```

flags_struct.h :

```
1  // ----------------------------------------------------------
2  // Code generated by Papyrus C
3  // ----------------------------------------------------------
4
5  // This template is called by the main module file
6  /*
7   * File generated from the BootLoader_Vsimp::flags_struct uml class
8   * Generated by the Papyrus C Generator (CEA LIST)
9   *
10  */
11
12 #ifndef FLAGS_STRUCT_H_
13 #define FLAGS_STRUCT_H_
14
15 // Explicit import of the class
16 /*
17  * -------------------- Includes and declares --------------------
18  */
19 // Derived includes
20 // End of Derived includes
21 // Derived declarations
22 // End of Derived declares
23
24 // Std headers
25 #include <stdio.h>
26 #include <stdlib.h>
27 #include <stdint.h>
28 // End of Std headers
```

```
29
30 // header body
31 /*
32  * ------------------- Public Class Description ------------------
33  */
34 //  Structure
35 typedef struct flags_struct flags_struct;
36
37 struct flags_struct {
38
39   unsigned eeprom;
40
41   unsigned rampz;
42 };
43
44 // Property initialisation declarations
45 /*
46  * ----------- Default value initialization prototypes -----------
47  */
48 /*
49  * Default value initialization
50  * @param flags_struct structure instance pointer
51  * @return void
52  */
53 void flags_struct_init(flags_struct *self);
54
55 // ----------------------------------Public Global VariableDescription
    ---------------------------------
56 // global variable declaration
57
58 unsigned eeprom;
59 // global variable declaration
60
61 unsigned rampz;
62
63 #endif /*FLAGS_STRUCT_H_*/
```

length_union.h :

```
1 // ------------------------------------------------------
2 // Code generated by Papyrus C
3 // ------------------------------------------------------
4
5 // This template is called by the main module file
6 /*
7  * File generated from the BootLoader_Vsimp::length_union uml class
8  * Generated by the Papyrus C Generator (CEA LIST)
9  *
```

```
10  */
11
12 #ifndef LENGTH_UNION_H_
13 #define LENGTH_UNION_H_
14
15 // Explicit import of the class
16 /*
17  * -------------------- Includes and declares --------------------
18  */
19 // Derived includes
20 // End of Derived includes
21 // Derived declarations
22 // End of Derived declares
23
24 // Std headers
25 #include <stdio.h>
26 #include <stdlib.h>
27 #include <stdint.h>
28 // End of Std headers
29
30 // header body
31 /*
32  * -------------------- Public Class Description ------------------
33  */
34 //  Structure
35 typedef union length_union length_union;
36
37 union length_union {
38
39   uint16_t word;
40
41   uint8_t byte[2];
42 };
43
44 // Property initialisation declarations
45
46 // ----------------------------------Public Global VariableDescription
    ----------------------------------
47 // global variable declaration
48
49 uint16_t word;
50 // global variable declaration
51
52 uint8_t byte[2];
53
54 #endif /*LENGTH_UNION_H_*/
```

ATmegaBOOT_168.h :

```
// ----------------------------------------------------------
// Code generated by Papyrus C
// ----------------------------------------------------------

/*
 * File generated from the BootLoader_Vsimp::ATmegaBOOT_168 uml class
 * Generated by the Papyrus C Generator (CEA LIST)
 *
 */

#ifndef ATMEGABOOT_168_H_
#define ATMEGABOOT_168_H_

/*
 * -------------------- Includes and declares --------------------
 */
// Derived includes
#include "address_union.h"
#include "flags_struct.h"
#include "length_union.h"
// End of Derived includes

// Derived declarations
// End of Derived declares

// Include from Include stereotype (header)
void (*app_start)(void) = 0x0000;
//pagesz=0x80;
//bootuart = 0;
//error_count = 0;
// End of Include stereotype (header)

// Std headers
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
// End of Std headers

/*
 * ------------------- Public Class Description -----------------
 */
//  Structure
/* Class Macro definition */
#define ATmegaBOOT_168(OBJ) ((ATmegaBOOT_168*)OBJ)

typedef struct ATmegaBOOT_168 ATmegaBOOT_168;
```

```
47
48 struct ATmegaBOOT_168 {
49
50 };
51
52 // Constructor and destructor declarations
53 /*
54  * --------- Default constructor & destructor prototypes ---------
55  */
56
57 // Property initialisation declarations
58 /*
59  * ----------- Default value initialization prototypes -----------
60  */
61 /**
62  * Default value initialization
63  * @param ATmegaBOOT_168 structure instance pointer
64  * @return void
65  */
66 void ATmegaBOOT_168_init(ATmegaBOOT_168 *self);
67
68 // Class methods declarations
69
70 // Class receptions declarations
71
72 // ----------------------------------Public Global VariableDescription
    ---------------------------------
73
74 /*
75  * ------------- Global Public Variable Declarations -------------
76  */
77 // global variable declaration
78 flags_struct flags;
79 // global variable declaration
80
81 length_union length;
82 // global variable declaration
83
84 address_union address;
85 // global variable declaration
86
87 uint8_t buff[256];
88 // global variable declaration
89
90 uint8_t address_high;
91 // global variable declaration
92
```

```
93 uint8_t pagesz = 0x80;
94 // global variable declaration
95
96 uint8_t bootuart = 0;
97 // global variable declaration
98
99 uint8_t error_count = 0;
100 // global variable declaration
101
102 uint8_t ch;
103 // global variable declaration
104
105 uint8_t ch2;
106 // global variable declaration
107
108 uint8_t w;
109
110 // ----------------------------------Global Public Functions
    ----------------------------------
111 /*
112  * ------------- Global Public Function Declarations -------------
113  */
114 void
115 putch(char ch);
116
117 char
118 getch();
119
120 void
121 getNch(uint8_t count);
122
123 void
124 byte_response(uint8_t val);
125
126 void
127 nothing_response();
128
129 void
130 flash_led(uint8_t count);
131
132 // --------------------------------Signal Event Process Functions
    Implementations--------------------------------
133
134 // ----------------------------------Call Event Process Functions
    ----------------------------------
135
136 #endif /*ATMEGABOOT_168_H_*/
```

ATmegaBOOT_168.c :

```
// ----------------------------------------------------------
// Code generated by Papyrus C
// ----------------------------------------------------------

/*
 * File generated from the BootLoader_Vsimp::ATmegaBOOT_168 uml class
 * Generated by the Papyrus C Generator (CEA LIST)
 *
 */

// Include self header
#include "ATmegaBOOT_168.h"
// Derived includes
// Include from Include stereotype (body)
#include <inttypes.h> // NOLINT
#include <avr/io.h> // NOLINT
#include <avr/pgmspace.h> // NOLINT
#include <avr/interrupt.h> // NOLINT
#include <avr/wdt.h> // NOLINT
#include <util/delay.h> // NOLINT
#include <avr/eeprom.h> // NOLINT

#define MAX_ERROR_COUNT 5

#ifndef BAUD_RATE
#define BAUD_RATE    19200
#endif

#define HW_VER    0x02
#define SW_MAJOR 0x01
#define SW_MINOR 0x10

#define BL_DDR   DDRD
#define BL_PORT  PORTD
#define BL_PIN   PIND
#define BL       PIND6

#define LED_DDR   DDRB
#define LED_PORT  PORTB
#define LED_PIN   PINB
#define LED       PINB5

#define SIG1   0x1E

#define SIG2   0x95
#define SIG3   0x0F
```

```
47 #define PAGE_SIZE 0x40U
48
49 int main(void) {
50   asm volatile("nop\n\t");
51
52   UBRR0L = (uint8_t)(F_CPU / (BAUD_RATE * 16L) - 1);
53   UBRR0H = (F_CPU / (BAUD_RATE * 16L) - 1) >> 8;
54   UCSR0B = (1 << RXEN0) | (1 << TXEN0);
55   UCSR0C = (1 << UCSZ00) | (1 << UCSZ01);
56   DDRD &= ~_BV(PIND0);
57   PORTD |= _BV(PIND0);
58   LED_DDR |= _BV(LED);
59   flash_led(3);
60   for (;;) {
61
62     ch = getch();
63
64     if (ch == '0') {
65       nothing_response();
66     }
67
68     else if (ch == '1') {
69       if (getch() == ' ') {
70         putch(0x14);
71         putch('A');
72         putch('V');
73         putch('R');
74         putch(' ');
75         putch('I');
76         putch('S');
77         putch('P');
78         putch(0x10);
79       } else {
80         if (++error_count == MAX_ERROR_COUNT)
81           app_start();
82       }
83     }
84
85     else if (ch == '@') {
86       ch2 = getch();
87       if (ch2 > 0x85)
88         getch();
89       nothing_response();
90     }
91
92     else if (ch == 'A') {
93       ch2 = getch();
```

```
94        if (ch2 == 0x80)
95          byte_response(HW_VER);
96        else if (ch2 == 0x81)
97          byte_response(SW_MAJOR);
98        else if (ch2 == 0x82)
99          byte_response(SW_MINOR);
100       else if (ch2 == 0x98)
101         byte_response(0x03);
102       else
103         byte_response(0x00);
104     }
105
106     else if (ch == 'B') {
107       getNch(20);
108       nothing_response();
109     }
110
111     else if (ch == 'E') {
112       getNch(5);
113       nothing_response();
114     }
115
116     else if (ch == 'P' || ch == 'R') {
117       nothing_response();
118     }
119
120     else if (ch == 'Q') {
121       nothing_response();
122     }
123
124     else if (ch == 'U') {
125       address.byte[0] = getch();
126       address.byte[1] = getch();
127       nothing_response();
128     }
129
130     else if (ch == 'V') {
131       if (getch() == 0x30) {
132         getch();
133         ch = getch();
134         getch();
135         if (ch == 0) {
136           byte_response(SIG1);
137         } else if (ch == 1) {
138           byte_response(SIG2);
139         } else {
140           byte_response(SIG3);
```

```
141         }
142       } else {
143         getNch(3);
144         byte_response(0x00);
145       }
146     }
147 
148     else if (ch == 'd') {
149       length.byte[1] = getch();
150       length.byte[0] = getch();
151       flags.eeprom = 0;
152       if (getch() == 'E')
153         flags.eeprom = 1;
154       for (w = 0; w < length.word; w++) {
155         buff[w] = getch();
156       }
157       if (getch() == ' ') {
158         if (flags.eeprom) {
159           address.word <<= 1;
160           for (w = 0; w < length.word; w++) {
161 
162             while (EECR & (1 << EEPE))
163               ;
164             EEAR = (uint16_t) (void*) address.word;
165             EEDR = buff[w];
166             EECR |= (1 << EEMPE);
167             EECR |= (1 << EEPE);
168 
169             address.word++;
170           }
171         } else {
172           if (address.byte[1] > 127)
173             address_high = 0x01;
174           else
175             address_high = 0x00;
176           address.word = address.word << 1;
177           if ((length.byte[0] & 0x01))
178             length.word++;
179           cli();
180 
181           while (bit_is_set(EECR, EEPE))
182             ;
183           asm volatile(
184               "clr  r17   \n\t" //page_word_count
185               "lds  r30,address \n\t"//Address of FLASH location (in bytes)
186               "lds  r31,address+1 \n\t"
187               "ldi  r28,lo8(buff) \n\t"//Start of buffer array in RAM
```

```
188             "ldi  r29,hi8(buff) \n\t"
189             "lds  r24,length  \n\t"//Length of data to be written (in
    bytes)
190             "lds  r25,length+1  \n\t"
191             "length_loop:   \n\t"//Main loop, repeat for number of words
    in block
192             "cpi  r17,0x00  \n\t"//If page_word_count=0 then erase page
193             "brne no_page_erase \n\t"
194             "wait_spm1:   \n\t"
195             "lds  r16,%0    \n\t"//Wait for previous spm to complete
196             "andi r16,1           \n\t"
197             "cpi  r16,1           \n\t"
198             "breq wait_spm1       \n\t"
199             "ldi  r16,0x03  \n\t"//Erase page pointed to by Z
200             "sts  %0,r16    \n\t"
201             "spm      \n\t"
202             "wait_spm2:   \n\t"
203             "lds  r16,%0    \n\t"//Wait for previous spm to complete
204             "andi r16,1           \n\t"
205             "cpi  r16,1           \n\t"
206             "breq wait_spm2       \n\t"
207
208             "ldi  r16,0x11  \n\t"//Re-enable RWW section
209             "sts  %0,r16    \n\t"
210             "spm      \n\t"
211             "no_page_erase:   \n\t"
212             "ld r0,Y+   \n\t"//Write 2 bytes into page buffer
213             "ld r1,Y+   \n\t"
214
215             "wait_spm3:   \n\t"
216             "lds  r16,%0    \n\t"//Wait for previous spm to complete
217             "andi r16,1           \n\t"
218             "cpi  r16,1           \n\t"
219             "breq wait_spm3       \n\t"
220             "ldi  r16,0x01  \n\t"//Load r0,r1 into FLASH page buffer
221             "sts  %0,r16    \n\t"
222             "spm      \n\t"
223
224             "inc  r17   \n\t"//page_word_count++
225             "cpi r17,%1         \n\t"
226             "brlo same_page \n\t"//Still same page in FLASH
227             "write_page:    \n\t"
228             "clr  r17   \n\t"//New page, write current one first
229             "wait_spm4:   \n\t"
230             "lds  r16,%0    \n\t"//Wait for previous spm to complete
231             "andi r16,1           \n\t"
232             "cpi  r16,1           \n\t"
```

```
233             "breq wait_spm4       \n\t"
234             "ldi  r16,0x05  \n\t"//Write page pointed to by Z
235             "sts  %0,r16    \n\t"
236             "spm      \n\t"
237             "wait_spm5:   \n\t"
238             "lds  r16,%0    \n\t"//Wait for previous spm to complete
239             "andi r16,1           \n\t"
240             "cpi  r16,1           \n\t"
241             "breq wait_spm5       \n\t"
242             "ldi  r16,0x11  \n\t"//Re-enable RWW section
243             "sts  %0,r16    \n\t"
244             "spm      \n\t"
245             "same_page:   \n\t"
246             "adiw r30,2   \n\t"//Next word in FLASH
247             "sbiw r24,2   \n\t"//length-2
248             "breq final_write \n\t"//Finished
249             "rjmp length_loop \n\t"
250             "final_write:   \n\t"
251             "cpi  r17,0   \n\t"
252             "breq block_done  \n\t"
253             "adiw r24,2   \n\t"//length+2, fool above check on length
    after short page write
254             "rjmp write_page  \n\t"
255             "block_done:    \n\t"
256             "clr  __zero_reg__  \n\t"//restore zero register
257             : "=m" (SPMCSR) : "M" (PAGE_SIZE) : "r0","r16","r17","r24","
    r25","r28","r29","r30","r31"
258           );
259       }
260       putch(0x14);
261       putch(0x10);
262     } else {
263       if (++error_count == MAX_ERROR_COUNT)
264         app_start();
265     }
266   }
267
268   else if (ch == 't') {
269     length.byte[1] = getch();
270     length.byte[0] = getch();
271     address.word = address.word << 1;
272     if (getch() == 'E')
273       flags.eeprom = 1;
274     else
275       flags.eeprom = 0;
276     if (getch() == ' ') {
277       putch(0x14);
```

```
278         for (w = 0; w < length.word; w++) {
279           if (flags.eeprom) {
280             while (EECR & (1 << EEPE))
281               ;
282             EEAR = (uint16_t) (void*) address.word;
283             EECR |= (1 << EERE);
284             putch (EEDR);
285             address.word++;
286           } else {
287
288             if (!flags.rampz)
289               putch(pgm_read_byte_near(address.word));
290             address.word++;
291           }
292         }
293         putch(0x10);
294       }
295     }
296
297     else if (ch == 'u') {
298       if (getch() == ' ') {
299         putch(0x14);
300         putch(SIG1);
301         putch(SIG2);
302         putch(SIG3);
303         putch(0x10);
304       } else {
305         if (++error_count == MAX_ERROR_COUNT)
306           app_start();
307       }
308     }
309
310     else if (ch == 'v') {
311       byte_response(0x00);
312     }
313
314     else if (++error_count == MAX_ERROR_COUNT) {
315       app_start();
316     }
317   }
318   LED_PORT &= ~_BV(LED);
319 }
320
321 // End of Include from Include stereotype (body)
322
323 /*
324  * ------- Default constructor & destructor implementations ------
```

```
325  */
326
327 void ATmegaBOOT_168_init(ATmegaBOOT_168 *self) {
328
329 }
330
331 /**
332  *
333  */
334 void putch(char ch) {
335   while (!(UCSR0A & _BV(UDRE0)))
336     ;
337   UDR0 = ch;
338 }
339
340 /**
341  *
342  */
343 char getch() {
344   uint32_t count = 0;
345   while (!(UCSR0A & _BV(RXC0))) {
346
347     count++;
348     if (count > MAX_TIME_COUNT)
349       app_start();
350   }
351   return UDR0;
352 }
353
354 /**
355  *
356  */
357 void getNch(uint8_t count) {
358   while (count--) {
359     getch();
360   }
361 }
362
363 /**
364  *
365  */
366 void byte_response(uint8_t val) {
367   if (getch() == ' ') {
368     putch(0x14);
369     putch(val);
370     putch(0x10);
371   } else {
```

```
372     if (++error_count == MAX_ERROR_COUNT)
373       app_start();
374   }
375 }
376
377 /**
378  *
379  */
380 void nothing_response() {
381   if (getch() == ' ') {
382     putch(0x14);
383     putch(0x10);
384   } else {
385     if (++error_count == MAX_ERROR_COUNT)
386       app_start();
387   }
388 }
389
390 /**
391  *
392  */
393 void flash_led(uint8_t count) {
394   while (count--) {
395     LED_PORT |= _BV(LED);
396     _delay_ms(50);
397     LED_PORT &= ~_BV(LED);
398     _delay_ms(300);
399   }
400 }
```

APÊNDICE C - CÓDIGO GERADO À PARTIR DO QUINTO MODELO

address_union.h :

```
1  // ------------------------------------------------------
2  // Code generated by Papyrus C
3  // ------------------------------------------------------
4
5  // This template is called by the main module file
6  /*
7   * File generated from the BootLoader_V5::address_union uml class
8   * Generated by the Papyrus C Generator (CEA LIST)
9   *
10  */
11
12 #ifndef ADDRESS_UNION_H_
13 #define ADDRESS_UNION_H_
14
15 // Explicit import of the class
16 /*
17  * -------------------- Includes and declares --------------------
18  */
19 // Derived includes
20 // End of Derived includes
21 // Derived declarations
22 // End of Derived declares
23
24 // Std headers
25 #include <stdio.h>
26 #include <stdlib.h>
27 #include <stdint.h>
28 // End of Std headers
29
30 // header body
31 /*
32  * ------------------- Public Class Description ------------------
33  */
34 //  Structure
35 typedef union address_union address_union;
36
37 union address_union {
38
```

```
39   uint16_t word;
40
41   uint8_t byte[2];
42 };
43
44 // Property initialisation declarations
45
46 // ----------------------------------Public Global VariableDescription
     ----------------------------------
47 // global variable declaration
48
49 uint16_t word;
50 // global variable declaration
51
52 uint8_t byte[2];
53
54 #endif /*ADDRESS_UNION_H_*/
```

flags_struct.h :

```
1  // ----------------------------------------------------------
2  // Code generated by Papyrus C
3  // ----------------------------------------------------------
4
5  // This template is called by the main module file
6  /*
7   * File generated from the BootLoader_V5::flags_struct uml class
8   * Generated by the Papyrus C Generator (CEA LIST)
9   *
10  */
11
12 #ifndef FLAGS_STRUCT_H_
13 #define FLAGS_STRUCT_H_
14
15 // Explicit import of the class
16 /*
17  * -------------------- Includes and declares --------------------
18  */
19 // Derived includes
20 // End of Derived includes
21 // Derived declarations
22 // End of Derived declares
23
24 // Std headers
25 #include <stdio.h>
26 #include <stdlib.h>
27 #include <stdint.h>
28 // End of Std headers
```

```
29
30 // header body
31 /*
32  * ------------------- Public Class Description ------------------
33  */
34 //  Structure
35 typedef struct flags_struct flags_struct;
36
37 struct flags_struct {
38
39   unsigned eeprom;
40
41   unsigned rampz;
42 };
43
44 // Property initialisation declarations
45 /*
46  * ----------- Default value initialization prototypes -----------
47  */
48 /*
49  * Default value initialization
50  * @param flags_struct structure instance pointer
51  * @return void
52  */
53 void flags_struct_init(flags_struct *self);
54
55 // ----------------------------------Public Global VariableDescription
    ---------------------------------
56 // global variable declaration
57
58 unsigned eeprom;
59 // global variable declaration
60
61 unsigned rampz;
62
63 #endif /*FLAGS_STRUCT_H_*/
```

length_union.h:

```
1 // ------------------------------------------------------
2 // Code generated by Papyrus C
3 // ------------------------------------------------------
4
5 // This template is called by the main module file
6 /*
7  * File generated from the BootLoader_V5::length_union uml class
8  * Generated by the Papyrus C Generator (CEA LIST)
9  *
```

```
10  */
11
12 #ifndef LENGTH_UNION_H_
13 #define LENGTH_UNION_H_
14
15 // Explicit import of the class
16 /*
17  * -------------------- Includes and declares --------------------
18  */
19 // Derived includes
20 // End of Derived includes
21 // Derived declarations
22 // End of Derived declares
23
24 // Std headers
25 #include <stdio.h>
26 #include <stdlib.h>
27 #include <stdint.h>
28 // End of Std headers
29
30 // header body
31 /*
32  * -------------------- Public Class Description ------------------
33  */
34 //  Structure
35 typedef union length_union length_union;
36
37 union length_union {
38
39   uint16_t word;
40
41   uint8_t byte[2];
42 };
43
44 // Property initialisation declarations
45
46 // ----------------------------------Public Global VariableDescription
    ----------------------------------
47 // global variable declaration
48
49 uint16_t word;
50 // global variable declaration
51
52 uint8_t byte[2];
53
54 #endif /*LENGTH_UNION_H_*/
```

ATmegaBOOT_168.h:

```
// ----------------------------------------------------------
// Code generated by Papyrus C
// ----------------------------------------------------------

/*
 * File generated from the BootLoader_V5::ATmegaBOOT_168 uml class
 * Generated by the Papyrus C Generator (CEA LIST)
 *
 */

#ifndef ATMEGABOOT_168_H_
#define ATMEGABOOT_168_H_

/*
 * -------------------- Includes and declares --------------------
 */
// Derived includes
#include "AppStart.h"
#include "Led.h"
#include "UART.h"
#include "address_union.h"
#include "flags_struct.h"
#include "length_union.h"
// End of Derived includes

// Derived declarations
// End of Derived declares

// Include from Include stereotype (header)
#include <inttypes.h>
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include <util/delay.h>
#include <avr/eeprom.h>
// End of Include stereotype (header)

// Std headers
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
// End of Std headers

/*
 * ------------------- Public Class Description -----------------
```

```
47  */
48 //  Structure
49 /* Class Macro definition */
50 #define ATmegaBOOT_168(OBJ) ((ATmegaBOOT_168*)OBJ)
51
52 typedef struct ATmegaBOOT_168 ATmegaBOOT_168;
53
54 struct ATmegaBOOT_168 {
55
56   length_union length_union;
57
58   flags_struct flags_struct;
59
60   address_union address_union;
61
62   UART uart;
63
64   AppStart appstart;
65
66   Led led;
67
68 };
69
70 // Constructor and destructor declarations
71 /*
72  * --------- Default constructor & destructor prototypes ---------
73  */
74
75 // Property initialisation declarations
76 /*
77  * ----------- Default value initialization prototypes -----------
78  */
79 /**
80  * Default value initialization
81  * @param ATmegaBOOT_168 structure instance pointer
82  * @return void
83  */
84 void ATmegaBOOT_168_init(ATmegaBOOT_168 *self);
85
86 // Class methods declarations
87
88 // Class receptions declarations
89
90 // ----------------------------------Public Global VariableDescription
    ----------------------------------
91
92 /*
```

```
93  * ------------ Global Public Variable Declarations ------------
94  */
95 // global variable declaration
96 flags_struct flags;
97 // global variable declaration
98
99 length_union length;
100 // global variable declaration
101
102 address_union address;
103 // global variable declaration
104
105 uint8_t buff[256];
106 // global variable declaration
107
108 uint8_t address_high;
109 // global variable declaration
110
111 uint8_t pagesz = 0x80;
112 // global variable declaration
113
114 uint8_t bootuart = 0;
115 // global variable declaration
116
117 uint8_t error_count = 0;
118 // global variable declaration
119
120 uint8_t ch;
121 // global variable declaration
122
123 uint8_t ch2;
124 // global variable declaration
125
126 uint8_t w;
127
128 // ----------------------------------Global Public Functions
    ----------------------------------
129 /*
130  * ------------ Global Public Function Declarations ------------
131  */
132 void
133 byte_response(uint8_t val);
134
135 void
136 nothing_response();
137
138 // ----------------------------------Signal Event Process Functions
```

```
    Implementations------------------------------

// ----------------------------------Call Event Process Functions
    ---------------------------------

#endif /*ATMEGABOOT_168_H_*/
```

ATmegaBOOT_168.c:

```
// ----------------------------------------------------------
// Code generated by Papyrus C
// ----------------------------------------------------------

/*
 * File generated from the BootLoader_V5::ATmegaBOOT_168 uml class
 * Generated by the Papyrus C Generator (CEA LIST)
 *
 */

// Include self header
#include "ATmegaBOOT_168.h"
// Derived includes
// Include from Include stereotype (body)

#define MAX_ERROR_COUNT 5

#define HW_VER    0x02
#define SW_MAJOR 0x01
#define SW_MINOR 0x10

#define SIG1   0x1E

#define SIG2   0x95
#define SIG3   0x0F
#define PAGE_SIZE 0x40U

int main(void) {
  asm volatile("nop\n\t");

  initialize_uart (BAUD_RATE);

  initialize_led();
  flash_led(3);

  for (;;) {

    ch = getch();

```

```
40      if (ch == '0') {
41        nothing_response();
42      }
43
44      else if (ch == '1') {
45        if (getch() == ' ') {
46          putch(0x14);
47          putch('A');
48          putch('V');
49          putch('R');
50          putch(' ');
51          putch('I');
52          putch('S');
53          putch('P');
54          putch(0x10);
55        } else {
56          if (++error_count == MAX_ERROR_COUNT) {
57            app_start();
58          }
59        }
60      }
61
62      else if (ch == '@') {
63        ch2 = getch();
64        if (ch2 > 0x85) {
65          getNch(1);
66        }
67        nothing_response();
68      }
69
70      else if (ch == 'A') {
71        ch2 = getch();
72        if (ch2 == 0x80) {
73          byte_response(HW_VER);
74        } else if (ch2 == 0x81) {
75          byte_response(SW_MAJOR);
76        } else if (ch2 == 0x82) {
77          byte_response(SW_MINOR);
78        } else if (ch2 == 0x98) {
79          byte_response(0x03);
80        } else {
81          byte_response(0x00);
82        }
83      }
84
85      else if (ch == 'B') {
86        getNch(20);
```

```
87        nothing_response();
88      }
89
90      else if (ch == 'E') {
91        getNch(5);
92        nothing_response();
93      }
94
95      else if (ch == 'P') {
96        nothing_response();
97      }
98
99      else if (ch == 'R') {
100       nothing_response();
101     }
102
103     else if (ch == 'Q') {
104       nothing_response();
105     }
106
107     else if (ch == 'U') {
108       address.byte[0] = getch();
109       address.byte[1] = getch();
110       nothing_response();
111     }
112
113     else if (ch == 'V') {
114       if (getch() == 0x30) {
115         getNch(1);
116         ch = getch();
117         getNch(1);
118         if (ch == 0) {
119           byte_response(SIG1);
120         } else if (ch == 1) {
121           byte_response(SIG2);
122         } else {
123           byte_response(SIG3);
124         }
125       } else {
126         getNch(3);
127         byte_response(0x00);
128       }
129     }
130
131     else if (ch == 'd') {
132       length.byte[1] = getch();
133       length.byte[0] = getch();
```

```
134       flags.eeprom = 0;
135       if (getch() == 'E') {
136         flags.eeprom = 1;
137       }
138       for (w = 0; w < length.word; w++) {
139         buff[w] = getch();
140       }
141       if (getch() == ' ') {
142         if (flags.eeprom) {
143           address.word <<= 1;
144           for (w = 0; w < length.word; w++) {
145
146             while (EECR & (1 << EEPE)) {
147               ;
148             }
149             EEAR = (uint16_t) (void*) address.word;
150             EEDR = buff[w];
151             EECR |= (1 << EEMPE);
152             EECR |= (1 << EEPE);
153
154             address.word++;
155           }
156         } else {
157           if (address.byte[1] > 127) {
158             address_high = 0x01;
159           } else {
160             address_high = 0x00;
161           }
162           address.word = address.word << 1;
163           if ((length.byte[0] & 0x01)) {
164             length.word++;
165           }
166           cli();
167
168           while (bit_is_set(EECR, EEPE)) {
169             ;
170           }
171
172           asm volatile(
173               "clr  r17   \n\t" //page_word_count
174               "lds  r30,address \n\t"//Address of FLASH location (in bytes)
175               "lds  r31,address+1 \n\t"
176               "ldi  r28,lo8(buff) \n\t"//Start of buffer array in RAM
177               "ldi  r29,hi8(buff) \n\t"
178               "lds  r24,length  \n\t"//Length of data to be written (in
    bytes)
179               "lds  r25,length+1  \n\t"
```

```
180             "length_loop:   \n\t"//Main loop, repeat for number of words
    in block
181             "cpi  r17,0x00  \n\t"//If page_word_count=0 then erase page
182             "brne no_page_erase \n\t"
183             "wait_spm1:   \n\t"
184             "lds  r16,%0    \n\t"//Wait for previous spm to complete
185             "andi r16,1           \n\t"
186             "cpi  r16,1           \n\t"
187             "breq wait_spm1       \n\t"
188             "ldi  r16,0x03  \n\t"//Erase page pointed to by Z
189             "sts  %0,r16    \n\t"
190             "spm      \n\t"
191             "wait_spm2:   \n\t"
192             "lds  r16,%0    \n\t"//Wait for previous spm to complete
193             "andi r16,1           \n\t"
194             "cpi  r16,1           \n\t"
195             "breq wait_spm2       \n\t"
196
197             "ldi  r16,0x11  \n\t"//Re-enable RWW section
198             "sts  %0,r16    \n\t"
199             "spm      \n\t"
200             "no_page_erase:   \n\t"
201             "ld r0,Y+   \n\t"//Write 2 bytes into page buffer
202             "ld r1,Y+   \n\t"
203
204             "wait_spm3:   \n\t"
205             "lds  r16,%0    \n\t"//Wait for previous spm to complete
206             "andi r16,1           \n\t"
207             "cpi  r16,1           \n\t"
208             "breq wait_spm3       \n\t"
209             "ldi  r16,0x01  \n\t"//Load r0,r1 into FLASH page buffer
210             "sts  %0,r16    \n\t"
211             "spm      \n\t"
212
213             "inc  r17   \n\t"//page_word_count++
214             "cpi r17,%1         \n\t"
215             "brlo same_page \n\t"//Still same page in FLASH
216             "write_page:    \n\t"
217             "clr  r17   \n\t"//New page, write current one first
218             "wait_spm4:   \n\t"
219             "lds  r16,%0    \n\t"//Wait for previous spm to complete
220             "andi r16,1           \n\t"
221             "cpi  r16,1           \n\t"
222             "breq wait_spm4       \n\t"
223             "ldi  r16,0x05  \n\t"//Write page pointed to by Z
224             "sts  %0,r16    \n\t"
225             "spm      \n\t"
```

```
226             "wait_spm5:   \n\t"
227             "lds  r16,%0    \n\t"//Wait for previous spm to complete
228             "andi r16,1           \n\t"
229             "cpi  r16,1           \n\t"
230             "breq wait_spm5       \n\t"
231             "ldi  r16,0x11  \n\t"//Re-enable RWW section
232             "sts  %0,r16    \n\t"
233             "spm      \n\t"
234             "same_page:   \n\t"
235             "adiw r30,2   \n\t"//Next word in FLASH
236             "sbiw r24,2   \n\t"//length-2
237             "breq final_write \n\t"//Finished
238             "rjmp length_loop \n\t"
239             "final_write:   \n\t"
240             "cpi  r17,0   \n\t"
241             "breq block_done  \n\t"
242             "adiw r24,2   \n\t"//length+2, fool above check on length
    after short page write
243             "rjmp write_page  \n\t"
244             "block_done:    \n\t"
245             "clr  __zero_reg__  \n\t"//restore zero register
246             : "=m" (SPMCSR) : "M" (PAGE_SIZE) : "r0","r16","r17","r24","
    r25","r28","r29","r30","r31"
247           );
248       }
249       putch(0x14);
250       putch(0x10);
251     } else {
252       if (++error_count == MAX_ERROR_COUNT) {
253         app_start();
254       }
255     }
256   }
257
258   else if (ch == 't') {
259     length.byte[1] = getch();
260     length.byte[0] = getch();
261     address.word = address.word << 1;
262     if (getch() == 'E') {
263       flags.eeprom = 1;
264     } else {
265       flags.eeprom = 0;
266     }
267     if (getch() == ' ') {
268       putch(0x14);
269       for (w = 0; w < length.word; w++) {
270         if (flags.eeprom) {
```

```
271             while (EECR & (1 << EEPE)) {
272               ;
273             }
274             EEAR = (uint16_t) (void*) address.word;
275             EECR |= (1 << EERE);
276             putch (EEDR);
277             address.word++;
278           } else {
279
280             if (!flags.rampz) {
281               putch(pgm_read_byte_near(address.word));
282             }
283             address.word++;
284           }
285         }
286         putch(0x10);
287       }
288     }
289
290     else if (ch == 'u') {
291       if (getch() == ' ') {
292         putch(0x14);
293         putch(SIG1);
294         putch(SIG2);
295         putch(SIG3);
296         putch(0x10);
297       } else {
298         if (++error_count == MAX_ERROR_COUNT) {
299           app_start();
300         }
301       }
302     }
303
304     else if (ch == 'v') {
305       byte_response(0x00);
306     }
307
308     else {
309       if (++error_count == MAX_ERROR_COUNT) {
310         app_start();
311       }
312     }
313   }
314 }
315
316 // End of Include from Include stereotype (body)
317
```

```
318 /*
319  * ------- Default constructor & destructor implementations ------
320  */
321
322 void ATmegaBOOT_168_init(ATmegaBOOT_168 *self) {
323
324 }
325
326 /**
327  *
328  */
329 void byte_response(uint8_t val) {
330   if (getch() == ' ') {
331     putch(0x14);
332     putch(val);
333     putch(0x10);
334   } else {
335     if (++error_count == MAX_ERROR_COUNT) {
336       app_start();
337     }
338   }
339 }
340
341 /**
342  *
343  */
344 void nothing_response() {
345   if (getch() == ' ') {
346     putch(0x14);
347     putch(0x10);
348   } else {
349     if (++error_count == MAX_ERROR_COUNT) {
350       app_start();
351     }
352   }
353 }
```

AppStart.h:

```
1 // ----------------------------------------------------------
2 // Code generated by Papyrus C
3 // ----------------------------------------------------------
4
5 /*
6  * File generated from the BootLoader_V5::AppStart uml class
7  * Generated by the Papyrus C Generator (CEA LIST)
8  *
9  */
```

```
10
11 #ifndef APPSTART_H_
12 #define APPSTART_H_
13
14 /*
15  * -------------------- Includes and declares --------------------
16  */
17 // Derived includes
18 // End of Derived includes
19 // Derived declarations
20 // End of Derived declares
21
22 // Std headers
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <stdint.h>
26 // End of Std headers
27
28 /*
29  * -------------------- Public Class Description ------------------
30  */
31 //  Structure
32 /* Class Macro definition */
33 #define AppStart(OBJ) ((AppStart*)OBJ)
34
35 typedef struct AppStart AppStart;
36
37 struct AppStart {
38
39 };
40
41 // Constructor and destructor declarations
42 /*
43  * --------- Default constructor & destructor prototypes ---------
44  */
45
46 // Property initialisation declarations
47 // Class methods declarations
48 // Class receptions declarations
49 // ----------------------------------Public Global VariableDescription
    ---------------------------------
50
51 // ----------------------------------Global Public Functions
    ---------------------------------
52 /*
53  * ------------- Global Public Function Declarations -------------
54  */
```

```
55 void
56 app_start();
57
58 // ----------------------------------Signal Event Process Functions
   Implementations-------------------------------
59
60 // ----------------------------------Call Event Process Functions
   ---------------------------------
61
62 #endif /*APPSTART_H_*/
```

AppStart.c:

```
1  // ---------------------------------------------------------
2  // Code generated by Papyrus C
3  // ---------------------------------------------------------
4
5  /*
6   * File generated from the BootLoader_V5::AppStart uml class
7   * Generated by the Papyrus C Generator (CEA LIST)
8   *
9   */
10
11 // Include self header
12 #include "AppStart.h"
13 // Derived includes
14 // Include from Include stereotype (body)
15 void (*start)(void) = 0x0000;
16
17 // End of Include from Include stereotype (body)
18
19 /*
20  * ------- Default constructor & destructor implementations ------
21  */
22
23 /**
24  *
25  */
26 void app_start() {
27   start();
28 }
```

UART.h:

```
1  // ---------------------------------------------------------
2  // Code generated by Papyrus C
3  // ---------------------------------------------------------
4
5  /*
```

```
6  * File generated from the BootLoader_V5::UART uml class
7  * Generated by the Papyrus C Generator (CEA LIST)
8  *
9  */
10
11 #ifndef UART_H_
12 #define UART_H_
13
14 /*
15  * -------------------- Includes and declares --------------------
16  */
17 // Derived includes
18 // End of Derived includes
19 // Derived declarations
20 // End of Derived declares
21 // Include from Include stereotype (header)
22 #include <avr/io.h>
23 // End of Include stereotype (header)
24
25 // Std headers
26 #include <stdio.h>
27 #include <stdlib.h>
28 #include <stdint.h>
29 // End of Std headers
30
31 /*
32  * -------------------- Public Class Description ------------------
33  */
34 //  Structure
35 /* Class Macro definition */
36 #define UART(OBJ) ((UART*)OBJ)
37
38 typedef struct UART UART;
39
40 struct UART {
41
42 };
43
44 // Constructor and destructor declarations
45 /*
46  * --------- Default constructor & destructor prototypes ---------
47  */
48
49 // Property initialisation declarations
50 // Class methods declarations
51 // Class receptions declarations
52 // ----------------------------------Public Global VariableDescription
```

```
    ---------------------------------

// ----------------------------------Global Public Functions
    ---------------------------------
/*
 * ------------ Global Public Function Declarations ------------
 */
void
putch(char ch);

char
getch();

void
getNch(uint8_t count);

void
initialize_uart(uint64_t baud_rate);

// ----------------------------------Signal Event Process Functions
    Implementations--------------------------------

// ----------------------------------Call Event Process Functions
    ---------------------------------

#endif /*UART_H_*/
```

UART.c:

```
// ---------------------------------------------------------
// Code generated by Papyrus C
// ---------------------------------------------------------

/*
 * File generated from the BootLoader_V5::UART uml class
 * Generated by the Papyrus C Generator (CEA LIST)
 *
 */

// Include self header
#include "UART.h"
// Derived includes
// Include from Include stereotype (body)
#define MAX_TIME_COUNT 1000000
// End of Include from Include stereotype (body)

/*
 * ------- Default constructor & destructor implementations ------
```

```
20  */
21
22 /**
23  *
24  */
25 void putch(char ch) {
26   while (!(UCSR0A & _BV(UDRE0))) {
27     ;
28   }
29   UDR0 = ch;
30 }
31
32 /**
33  *
34  */
35 char getch() {
36   uint32_t count = 0;
37   while (!(UCSR0A & _BV(RXC0))) {
38     count++;
39     if (count > MAX_TIME_COUNT)
40       return 0x00;
41   }
42   return UDR0;
43 }
44
45 /**
46  *
47  */
48 void getNch(uint8_t count) {
49   uint8_t aux_count = count;
50   while (aux_count--) {
51     char temp = getch();
52   }
53 }
54
55 /**
56  *
57  */
58 void initialize_uart(uint64_t baud_rate) {
59   UBRR0L = (uint8_t)(F_CPU / (baud_rate * 16L) - 1);
60   UBRR0H = (F_CPU / (baud_rate * 16L) - 1) >> 8;
61   UCSR0B = (1 << RXEN0) | (1 << TXEN0);
62   UCSR0C = (1 << UCSZ00) | (1 << UCSZ01);
63   DDRD &= ~_BV(PIND0);
64   PORTD |= _BV(PIND0);
65 }
```

Led.h:

```
// ----------------------------------------------------------
// Code generated by Papyrus C
// ----------------------------------------------------------

/*
 * File generated from the BootLoader_V5::Led uml class
 * Generated by the Papyrus C Generator (CEA LIST)
 *
 */

#ifndef LED_H_
#define LED_H_

/*
 * -------------------- Includes and declares --------------------
 */
// Derived includes
// End of Derived includes
// Derived declarations
// End of Derived declares
// Include from Include stereotype (header)
#include <avr/io.h>
#include <util/delay.h>

// End of Include stereotype (header)

// Std headers
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
// End of Std headers

/*
 * ------------------- Public Class Description -----------------
 */
//  Structure
/* Class Macro definition */
#define Led(OBJ) ((Led*)OBJ)

typedef struct Led Led;

struct Led {

};

// Constructor and destructor declarations
```

```
47 /*
48  * --------- Default constructor & destructor prototypes ---------
49  */
50
51 // Property initialisation declarations
52 // Class methods declarations
53 // Class receptions declarations
54 // ----------------------------------Public Global VariableDescription
     ----------------------------------
55
56 // ----------------------------------Global Public Functions
     ----------------------------------
57 /*
58  * ------------- Global Public Function Declarations -------------
59  */
60 void
61 flash_led(uint8_t count);
62
63 void
64 initialize_led();
65
66 // ----------------------------------Signal Event Process Functions
     Implementations-------------------------------
67
68 // ----------------------------------Call Event Process Functions
     ----------------------------------
69
70 #endif /*LED_H_*/
```

Led.c:

```
1 // ----------------------------------------------------------
2 // Code generated by Papyrus C
3 // ----------------------------------------------------------
4
5 /*
6  * File generated from the BootLoader_V5::Led uml class
7  * Generated by the Papyrus C Generator (CEA LIST)
8  *
9  */
10
11 // Include self header
12 #include "Led.h"
13 // Derived includes
14 // Include from Include stereotype (body)
15
16 #define LED_DDR  DDRB
17 #define LED_PORT PORTB
```

```
18 #define LED_PIN  PINB
19 #define LED      PINB5
20 // End of Include from Include stereotype (body)
21
22 /*
23  * ------- Default constructor & destructor implementations ------
24  */
25
26 /**
27  *
28  */
29 void flash_led(uint8_t count) {
30   uint8_t aux_count = count;
31   while (aux_count--) {
32     LED_PORT |= _BV(LED);
33     _delay_ms(50);
34     LED_PORT &= ~_BV(LED);
35     _delay_ms(300);
36   }
37 }
38
39 /**
40  *
41  */
42 void initialize_led() {
43   LED_DDR |= _BV(LED);
44 }
```

APÊNDICE D - BOOTLOADER AJUSTADO

address_union.h :

```
// ----------------------------------------------------------
// Code generated by Papyrus C
// ----------------------------------------------------------

// This template is called by the main module file
/*
 * File generated from the BootLoader_V5::address_union uml class
 * Generated by the Papyrus C Generator (CEA LIST)
 *
 */

#ifndef ADDRESS_UNION_H_
#define ADDRESS_UNION_H_

// Explicit import of the class
/*
 * -------------------- Includes and declares --------------------
 */
// Derived includes
// End of Derived includes
// Derived declarations
// End of Derived declares

// header body
/*
 * ------------------- Public Class Description ------------------
 */
//  Structure
typedef union address_union address_union;

union address_union {

  uint16_t word;

  uint8_t byte[2];
};

// Property initialisation declarations
```

```
39
40 // ----------------------------------Public Global VariableDescription
   ----------------------------------
41 // global variable declaration
42
43 uint16_t word;
44 // global variable declaration
45
46 uint8_t byte[2];
47
48 #endif /*ADDRESS_UNION_H_*/
```

flags_struct.h :

```
1  // ----------------------------------------------------------
2  // Code generated by Papyrus C
3  // ----------------------------------------------------------
4
5  // This template is called by the main module file
6  /*
7   * File generated from the BootLoader_V5::flags_struct uml class
8   * Generated by the Papyrus C Generator (CEA LIST)
9   *
10  */
11
12 #ifndef FLAGS_STRUCT_H_
13 #define FLAGS_STRUCT_H_
14
15 // Explicit import of the class
16 /*
17  * -------------------- Includes and declares --------------------
18  */
19 // Derived includes
20 // End of Derived includes
21 // Derived declarations
22 // End of Derived declares
23
24
25 // header body
26 /*
27  * ------------------- Public Class Description -----------------
28  */
29 //  Structure
30 typedef struct flags_struct flags_struct;
31
32 struct flags_struct {
33
34   unsigned eeprom;
```

```
35
36   unsigned rampz;
37 };
38
39 // Property initialisation declarations
40 /*
41  * ----------- Default value initialization prototypes -----------
42  */
43 /*
44  * Default value initialization
45  * @param flags_struct structure instance pointer
46  * @return void
47  */
48 void flags_struct_init(flags_struct *self);
49
50 // ----------------------------------Public Global VariableDescription
    ---------------------------------
51 // global variable declaration
52
53 unsigned eeprom;
54 // global variable declaration
55
56 unsigned rampz;
57
58 #endif /*FLAGS_STRUCT_H_*/
```

length_union.h:

```
1 // ------------------------------------------------------
2 // Code generated by Papyrus C
3 // ------------------------------------------------------
4
5 // This template is called by the main module file
6 /*
7  * File generated from the BootLoader_V5::length_union uml class
8  * Generated by the Papyrus C Generator (CEA LIST)
9  *
10  */
11
12 #ifndef LENGTH_UNION_H_
13 #define LENGTH_UNION_H_
14
15 // Explicit import of the class
16 /*
17  * -------------------- Includes and declares --------------------
18  */
19 // Derived includes
20 // End of Derived includes
```

```
21 // Derived declarations
22 // End of Derived declares
23
24
25 // header body
26 /*
27  * ------------------- Public Class Description ------------------
28  */
29 //  Structure
30 typedef union length_union length_union;
31
32 union length_union {
33
34   uint16_t word;
35
36   uint8_t byte[2];
37 };
38
39 // Property initialisation declarations
40
41 // ----------------------------------Public Global VariableDescription
     ---------------------------------
42 // global variable declaration
43
44 uint16_t word;
45 // global variable declaration
46
47 uint8_t byte[2];
48
49 #endif /*LENGTH_UNION_H_*/
```

ATmegaBOOT_168.h:

```
1  // ----------------------------------------------------------
2  // Code generated by Papyrus C
3  // ----------------------------------------------------------
4
5  /*
6   * File generated from the BootLoader_V5::ATmegaBOOT_168 uml class
7   * Generated by the Papyrus C Generator (CEA LIST)
8   *
9   */
10
11 #ifndef ATMEGABOOT_168_H_
12 #define ATMEGABOOT_168_H_
13
14 /*
15  * -------------------- Includes and declares --------------------
```

```
16  */
17 // Derived includes
18 #include "AppStart.h"
19 #include "Led.h"
20 #include "UART.h"
21 #include "address_union.h"
22 #include "flags_struct.h"
23 #include "length_union.h"
24 // End of Derived includes
25
26 // Derived declarations
27 // End of Derived declares
28
29 // Include from Include stereotype (header)
30 #include <inttypes.h>
31 #include <avr/io.h>
32 #include <avr/pgmspace.h>
33 #include <avr/interrupt.h>
34 #include <avr/wdt.h>
35 #include <util/delay.h>
36 #include <avr/eeprom.h>
37 // End of Include stereotype (header)
38
39 /*
40  * ------------------- Public Class Description ------------------
41  */
42 //  Structure
43 /* Class Macro definition */
44
45 // Constructor and destructor declarations
46
47 // Class methods declarations
48
49 // Class receptions declarations
50
51 // ----------------------------------Public Global VariableDescription
    ---------------------------------
52
53 /*
54  * ------------- Global Public Variable Declarations -------------
55  */
56 // global variable declaration
57 flags_struct flags;
58 // global variable declaration
59
60 length_union length;
61 // global variable declaration
```

```
62
63 address_union address;
64 // global variable declaration
65
66 uint8_t buff[256];
67 // global variable declaration
68
69 uint8_t address_high;
70 // global variable declaration
71
72 uint8_t pagesz = 0x80;
73 // global variable declaration
74
75 uint8_t bootuart = 0;
76 // global variable declaration
77
78 uint8_t error_count = 0;
79 // global variable declaration
80
81 uint8_t ch;
82 // global variable declaration
83
84 uint8_t ch2;
85 // global variable declaration
86
87 uint8_t w;
88
89 // ----------------------------------Global Public Functions
   ---------------------------------
90 /*
91  * ------------- Global Public Function Declarations -------------
92  */
93 void
94 byte_response(uint8_t val);
95
96 void
97 nothing_response();
98
99 // ----------------------------------Signal Event Process Functions
   Implementations-------------------------------
100
101 // ----------------------------------Call Event Process Functions
   ---------------------------------
102
103 #endif /*ATMEGABOOT_168_H_*/
```

ATmegaBOOT_168.c:

```
// ----------------------------------------------------------
// Code generated by Papyrus C
// ----------------------------------------------------------

/*
 * File generated from the BootLoader_V5::ATmegaBOOT_168 uml class
 * Generated by the Papyrus C Generator (CEA LIST)
 *
 */

// Include self header
#include "ATmegaBOOT_168.h"
// Derived includes
// Include from Include stereotype (body)

#define MAX_ERROR_COUNT 5

#define HW_VER    0x02
#define SW_MAJOR 0x01
#define SW_MINOR 0x10

#define SIG1   0x1E

#define SIG2   0x95
#define SIG3   0x0F
#define PAGE_SIZE 0x40U

int main(void) {
  asm volatile("nop\n\t");

  initialize_uart (BAUD_RATE);

  initialize_led();
  flash_led(3);

  for (;;) {

    ch = getch();

    if (ch == '0') {
      nothing_response();
    }

    else if (ch == '1') {
      if (getch() == ' ') {
        putch(0x14);
        putch('A');
```

```
48          putch('V');
49          putch('R');
50          putch(' ');
51          putch('I');
52          putch('S');
53          putch('P');
54          putch(0x10);
55        } else {
56          if (++error_count == MAX_ERROR_COUNT) {
57            app_start();
58          }
59        }
60      }
61
62      else if (ch == '@') {
63        ch2 = getch();
64        if (ch2 > 0x85) {
65          getNch(1);
66        }
67        nothing_response();
68      }
69
70      else if (ch == 'A') {
71        ch2 = getch();
72        if (ch2 == 0x80) {
73          byte_response(HW_VER);
74        } else if (ch2 == 0x81) {
75          byte_response(SW_MAJOR);
76        } else if (ch2 == 0x82) {
77          byte_response(SW_MINOR);
78        } else if (ch2 == 0x98) {
79          byte_response(0x03);
80        } else {
81          byte_response(0x00);
82        }
83      }
84
85      else if (ch == 'B') {
86        getNch(20);
87        nothing_response();
88      }
89
90      else if (ch == 'E') {
91        getNch(5);
92        nothing_response();
93      }
94
```

```
95      else if (ch == 'P') {
96        nothing_response();
97      }
98
99      else if (ch == 'R') {
100       nothing_response();
101     }
102
103     else if (ch == 'Q') {
104       nothing_response();
105     }
106
107     else if (ch == 'U') {
108       address.byte[0] = getch();
109       address.byte[1] = getch();
110       nothing_response();
111     }
112
113     else if (ch == 'V') {
114       if (getch() == 0x30) {
115         getNch(1);
116         ch = getch();
117         getNch(1);
118         if (ch == 0) {
119           byte_response(SIG1);
120         } else if (ch == 1) {
121           byte_response(SIG2);
122         } else {
123           byte_response(SIG3);
124         }
125       } else {
126         getNch(3);
127         byte_response(0x00);
128       }
129     }
130
131     else if (ch == 'd') {
132       length.byte[1] = getch();
133       length.byte[0] = getch();
134       flags.eeprom = 0;
135       if (getch() == 'E') {
136         flags.eeprom = 1;
137       }
138       for (w = 0; w < length.word; w++) {
139         buff[w] = getch();
140       }
141       if (getch() == ' ') {
```

```
142       if (flags.eeprom) {
143         address.word <<= 1;
144         for (w = 0; w < length.word; w++) {
145
146           while (EECR & (1 << EEPE)) {
147             ;
148           }
149           EEAR = (uint16_t) (void*) address.word;
150           EEDR = buff[w];
151           EECR |= (1 << EEMPE);
152           EECR |= (1 << EEPE);
153
154           address.word++;
155         }
156       } else {
157         if (address.byte[1] > 127) {
158           address_high = 0x01;
159         } else {
160           address_high = 0x00;
161         }
162         address.word = address.word << 1;
163         if ((length.byte[0] & 0x01)) {
164           length.word++;
165         }
166         cli();
167
168         while (bit_is_set(EECR, EEPE)) {
169           ;
170         }
171
172         asm volatile(
173             "clr  r17   \n\t" //page_word_count
174             "lds  r30,address \n\t"//Address of FLASH location (in bytes)
175             "lds  r31,address+1 \n\t"
176             "ldi  r28,lo8(buff) \n\t"//Start of buffer array in RAM
177             "ldi  r29,hi8(buff) \n\t"
178             "lds  r24,length  \n\t"//Length of data to be written (in
    bytes)
179             "lds  r25,length+1  \n\t"
180             "length_loop:   \n\t"//Main loop, repeat for number of words
    in block
181             "cpi  r17,0x00  \n\t"//If page_word_count=0 then erase page
182             "brne no_page_erase \n\t"
183             "wait_spm1:   \n\t"
184             "lds  r16,%0    \n\t"//Wait for previous spm to complete
185             "andi r16,1           \n\t"
186             "cpi  r16,1           \n\t"
```

```
187             "breq wait_spm1       \n\t"
188             "ldi  r16,0x03  \n\t"//Erase page pointed to by Z
189             "sts  %0,r16    \n\t"
190             "spm      \n\t"
191             "wait_spm2:   \n\t"
192             "lds  r16,%0    \n\t"//Wait for previous spm to complete
193             "andi r16,1           \n\t"
194             "cpi  r16,1           \n\t"
195             "breq wait_spm2       \n\t"
196
197             "ldi  r16,0x11  \n\t"//Re-enable RWW section
198             "sts  %0,r16    \n\t"
199             "spm      \n\t"
200             "no_page_erase:   \n\t"
201             "ld r0,Y+   \n\t"//Write 2 bytes into page buffer
202             "ld r1,Y+   \n\t"
203
204             "wait_spm3:   \n\t"
205             "lds  r16,%0    \n\t"//Wait for previous spm to complete
206             "andi r16,1           \n\t"
207             "cpi  r16,1           \n\t"
208             "breq wait_spm3       \n\t"
209             "ldi  r16,0x01  \n\t"//Load r0,r1 into FLASH page buffer
210             "sts  %0,r16    \n\t"
211             "spm      \n\t"
212
213             "inc  r17   \n\t"//page_word_count++
214             "cpi r17,%1         \n\t"
215             "brlo same_page \n\t"//Still same page in FLASH
216             "write_page:    \n\t"
217             "clr  r17   \n\t"//New page, write current one first
218             "wait_spm4:   \n\t"
219             "lds  r16,%0    \n\t"//Wait for previous spm to complete
220             "andi r16,1           \n\t"
221             "cpi  r16,1           \n\t"
222             "breq wait_spm4       \n\t"
223             "ldi  r16,0x05  \n\t"//Write page pointed to by Z
224             "sts  %0,r16    \n\t"
225             "spm      \n\t"
226             "wait_spm5:   \n\t"
227             "lds  r16,%0    \n\t"//Wait for previous spm to complete
228             "andi r16,1           \n\t"
229             "cpi  r16,1           \n\t"
230             "breq wait_spm5       \n\t"
231             "ldi  r16,0x11  \n\t"//Re-enable RWW section
232             "sts  %0,r16    \n\t"
233             "spm      \n\t"
```

```
234             "same_page:    \n\t"
235             "adiw r30,2    \n\t"//Next word in FLASH
236             "sbiw r24,2    \n\t"//length-2
237             "breq final_write \n\t"//Finished
238             "rjmp length_loop \n\t"
239             "final_write:    \n\t"
240             "cpi  r17,0    \n\t"
241             "breq block_done  \n\t"
242             "adiw r24,2    \n\t"//length+2, fool above check on length
    after short page write
243             "rjmp write_page  \n\t"
244             "block_done:     \n\t"
245             "clr  __zero_reg__  \n\t"//restore zero register
246             : "=m" (SPMCSR) : "M" (PAGE_SIZE) : "r0","r16","r17","r24","
    r25","r28","r29","r30","r31"
247           );
248       }
249       putch(0x14);
250       putch(0x10);
251     } else {
252       if (++error_count == MAX_ERROR_COUNT) {
253         app_start();
254       }
255     }
256   }
257
258   else if (ch == 't') {
259     length.byte[1] = getch();
260     length.byte[0] = getch();
261     address.word = address.word << 1;
262     if (getch() == 'E') {
263       flags.eeprom = 1;
264     } else {
265       flags.eeprom = 0;
266     }
267     if (getch() == ' ') {
268       putch(0x14);
269       for (w = 0; w < length.word; w++) {
270         if (flags.eeprom) {
271           while (EECR & (1 << EEPE)) {
272             ;
273           }
274           EEAR = (uint16_t) (void*) address.word;
275           EECR |= (1 << EERE);
276           putch (EEDR);
277           address.word++;
278         } else {
```

```
279
280             if (!flags.rampz) {
281               putch(pgm_read_byte_near(address.word));
282             }
283             address.word++;
284           }
285         }
286         putch(0x10);
287       }
288     }
289
290     else if (ch == 'u') {
291       if (getch() == ' ') {
292         putch(0x14);
293         putch(SIG1);
294         putch(SIG2);
295         putch(SIG3);
296         putch(0x10);
297       } else {
298         if (++error_count == MAX_ERROR_COUNT) {
299           app_start();
300         }
301       }
302     }
303
304     else if (ch == 'v') {
305       byte_response(0x00);
306     }
307
308     else {
309       if (++error_count == MAX_ERROR_COUNT) {
310         app_start();
311       }
312     }
313   }
314 }
315
316 // End of Include from Include stereotype (body)
317
318
319 /**
320  *
321  */
322 void byte_response(uint8_t val) {
323   if (getch() == ' ') {
324     putch(0x14);
325     putch(val);
```

```
326     putch(0x10);
327   } else {
328     if (++error_count == MAX_ERROR_COUNT){
329       app_start();
330     }
331   }
332 }
333
334 /**
335  *
336  */
337 void nothing_response() {
338   if (getch() == ' ') {
339     putch(0x14);
340     putch(0x10);
341   } else {
342     if (++error_count == MAX_ERROR_COUNT){
343       app_start();
344     }
345   }
346 }
```

AppStart.h:

```
1  // ----------------------------------------------------------
2  // Code generated by Papyrus C
3  // ----------------------------------------------------------
4
5  /*
6   * File generated from the BootLoader_V5::AppStart uml class
7   * Generated by the Papyrus C Generator (CEA LIST)
8   *
9   */
10
11 #ifndef APPSTART_H_
12 #define APPSTART_H_
13
14 /*
15  * -------------------- Includes and declares --------------------
16  */
17 // Derived includes
18 // End of Derived includes
19 // Derived declarations
20 // End of Derived declares
21
22
23 /*
24  * ------------------- Public Class Description -----------------
```

```
 */
//  Structure


// Constructor and destructor declarations
/*
 * --------- Default constructor & destructor prototypes ---------
 */

// Property initialisation declarations
// Class methods declarations
// Class receptions declarations
// ----------------------------------Public Global VariableDescription
   ----------------------------------

// ----------------------------------Global Public Functions
   ----------------------------------
/*
 * ------------- Global Public Function Declarations -------------
 */
void
app_start();

// ----------------------------------Signal Event Process Functions
   Implementations-------------------------------

// ----------------------------------Call Event Process Functions
   ----------------------------------

#endif /*APPSTART_H_*/
```

AppStart.c:

```
// ---------------------------------------------------------
// Code generated by Papyrus C
// ---------------------------------------------------------

/*
 * File generated from the BootLoader_V5::AppStart uml class
 * Generated by the Papyrus C Generator (CEA LIST)
 *
 */

// Include self header
#include "AppStart.h"
// Derived includes
// Include from Include stereotype (body)
void (*start)(void) = 0x0000;
```

```
16
17 // End of Include from Include stereotype (body)
18
19 /*
20  * ------- Default constructor & destructor implementations ------
21  */
22
23 /**
24  *
25  */
26 void app_start() {
27   start();
28 }
```

UART.h:

```
1  // ----------------------------------------------------------
2  // Code generated by Papyrus C
3  // ----------------------------------------------------------
4
5  /*
6   * File generated from the BootLoader_V5::UART uml class
7   * Generated by the Papyrus C Generator (CEA LIST)
8   *
9   */
10
11 #ifndef UART_H_
12 #define UART_H_
13
14 /*
15  * -------------------- Includes and declares --------------------
16  */
17 // Derived includes
18 // End of Derived includes
19 // Derived declarations
20 // End of Derived declares
21 // Include from Include stereotype (header)
22 #include <avr/io.h>
23 // End of Include stereotype (header)
24
25 /*
26  * ------------------- Public Class Description -----------------
27  */
28 //  Structure
29 /* Class Macro definition */
30
31 // Constructor and destructor declarations
32 /*
```

```
33  * --------- Default constructor & destructor prototypes ---------
34  */
35
36 // Property initialisation declarations
37 // Class methods declarations
38 // Class receptions declarations
39 // ----------------------------------Public Global VariableDescription
    ----------------------------------
40
41 // ----------------------------------Global Public Functions
    ----------------------------------
42 /*
43  * ------------- Global Public Function Declarations -------------
44  */
45 void
46 putch(char ch);
47
48 char
49 getch();
50
51 void
52 getNch(uint8_t count);
53
54 void
55 initialize_uart(uint64_t baud_rate);
56
57 // ----------------------------------Signal Event Process Functions
    Implementations-------------------------------
58
59 // ----------------------------------Call Event Process Functions
    ----------------------------------
60
61 #endif /*UART_H_*/
```

UART.c:

```
1  // ---------------------------------------------------------
2  // Code generated by Papyrus C
3  // ---------------------------------------------------------
4
5  /*
6   * File generated from the BootLoader_V5::UART uml class
7   * Generated by the Papyrus C Generator (CEA LIST)
8   *
9   */
10
11 // Include self header
12 #include "UART.h"
```

```
13 // Derived includes
14 // Include from Include stereotype (body)
15 #define MAX_TIME_COUNT 1000000
16 // End of Include from Include stereotype (body)
17
18 /*
19  * ------- Default constructor & destructor implementations ------
20  */
21
22 /**
23  *
24  */
25 void putch(char ch) {
26   while (!(UCSR0A & _BV(UDRE0))) {
27     ;
28   }
29   UDR0 = ch;
30 }
31
32 /**
33  *
34  */
35 char getch() {
36   uint32_t count = 0;
37   while (!(UCSR0A & _BV(RXC0))) {
38     count++;
39     if (count > MAX_TIME_COUNT){
40       return 0x00;
41     }
42   }
43   return UDR0;
44 }
45
46 /**
47  *
48  */
49 void getNch(uint8_t count) {
50   uint8_t aux_count = count;
51   while (aux_count--) {
52     char temp = getch();
53   }
54 }
55
56 /**
57  *
58  */
59 void initialize_uart(uint64_t baud_rate) {
```

```
  UBRR0L = (uint8_t)(F_CPU / (baud_rate * 16L) - 1);
  UBRR0H = (F_CPU / (baud_rate * 16L) - 1) >> 8;
  UCSR0B = (1 << RXEN0) | (1 << TXEN0);
  UCSR0C = (1 << UCSZ00) | (1 << UCSZ01);
  DDRD &= ~_BV(PIND0);
  PORTD |= _BV(PIND0);
}
```

Led.h:

```
// ----------------------------------------------------------
// Code generated by Papyrus C
// ----------------------------------------------------------

/*
 * File generated from the BootLoader_V5::Led uml class
 * Generated by the Papyrus C Generator (CEA LIST)
 *
 */

#ifndef LED_H_
#define LED_H_

/*
 * -------------------- Includes and declares --------------------
 */
// Derived includes
// End of Derived includes
// Derived declarations
// End of Derived declares
// Include from Include stereotype (header)
#include <avr/io.h>
#include <util/delay.h>

// End of Include stereotype (header)


/*
 * -------------------- Public Class Description ------------------
 */
//  Structure


// Constructor and destructor declarations
/*
 * --------- Default constructor & destructor prototypes ---------
 */

```

```
39 // Property initialisation declarations
40 // Class methods declarations
41 // Class receptions declarations
42 // ----------------------------------Public Global VariableDescription
   ---------------------------------
43
44 // ----------------------------------Global Public Functions
   ---------------------------------
45 /*
46  * ------------ Global Public Function Declarations ------------
47  */
48 void
49 flash_led(uint8_t count);
50
51 void
52 initialize_led();
53
54 // ----------------------------------Signal Event Process Functions
   Implementations---------------------------------
55
56 // ----------------------------------Call Event Process Functions
   ---------------------------------
57
58 #endif /*LED_H_*/
```

Led.c:

```
1  // ----------------------------------------------------------
2  // Code generated by Papyrus C
3  // ----------------------------------------------------------
4
5  /*
6   * File generated from the BootLoader_V5::Led uml class
7   * Generated by the Papyrus C Generator (CEA LIST)
8   *
9   */
10
11 // Include self header
12 #include "Led.h"
13 // Derived includes
14 // Include from Include stereotype (body)
15
16 #define LED_DDR  DDRB
17 #define LED_PORT PORTB
18 #define LED_PIN  PINB
19 #define LED      PINB5
20 // End of Include from Include stereotype (body)
21
```

```
22 /*
23  * ------- Default constructor & destructor implementations ------
24  */
25
26 /**
27  *
28  */
29 void flash_led(uint8_t count) {
30   uint8_t aux_count = count;
31   while (aux_count--) {
32     LED_PORT |= _BV(LED);
33     _delay_ms(50);
34     LED_PORT &= ~_BV(LED);
35     _delay_ms(300);
36   }
37 }
38
39 /**
40  *
41  */
42 void initialize_led() {
43   LED_DDR |= _BV(LED);
44 }
```

APÊNDICE E - CÓDIGO DA APLICAÇÃO

test_app.ino :

```
1
2  extern "C"{
3  #include "Led.h"
4  #include "UART.h"
5  }
6
7  const char message[]="UFSC CTJ";
8
9  void send_string(const char * string);
10
11 void setup() {
12   initialize_led();
13   initialize_uart(57600);
14 }
15
16 void loop() {
17   send_string(message);
18   delay(1000);
19 }
20
21
22 void send_string(const char * string){
23   int str_length = strlen(string);
24   flash_led(str_length);
25   for (int i=0; i<str_length;i++){
26     putch(string[i]);
27   }
28   putch('\n');
29 }
```