

UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
DEPTO. DE ENGENHARIA ELÉTRICA E DE
COMPUTAÇÃO

**SOFIA - Um simulador de Arduino baseado em
Android**

Autor: Kollins Gabriel Lima, nº. USP 9012931

Orientador: Prof. Dr. Evandro Luis Linhari Rodrigues

São Carlos

2018

Kollins Gabriel Lima

SOFIA - Um simulador de Arduino baseado em Android

Trabalho de conclusão de curso apresentado à Escola de
Engenharia de São Carlos, da Universidade de São Paulo

Curso de Engenharia de Computação

ORIENTADOR: Prof. Dr. Evandro Luis Linhari Rodrigues

São Carlos

2018

Agradecimentos

Agradeço ao Prof. Dr. Evandro Luis Linhari Rodrigues por todo o apoio que recebi durante a execução deste trabalho e pela confiança em mim depositada.

Agradeço também aos meus colegas que contribuíram com dicas e sugestões e me ajudaram a melhorar a qualidade deste projeto.

"Only do what makes you love life"

Boglárka Csemer

Resumo

Introduzidas no mercado no ano de 2005, as placas de Arduino vêm se popularizando cada vez mais devido à sua simplicidade e seu custo reduzido em relação aos seus concorrentes. Este trabalho apresenta o projeto SOFIA, uma alternativa às placas de Arduino na forma de simulador para dispositivos Android. Aproveitando-se das características *Open-Source* do projeto Arduino, foi possível fazer modificações na IDE oficial para a criação de um sistema integrado Computador-*Smartphone*, facilitando o uso do aplicativo e se aproximando das condições reais de desenvolvimento. O que se obteve foi um simulador capaz de executar aplicações simples escritas para o Arduino UNO que, apesar de ainda possuir limitações de velocidade (quando comparado à implementações em *hardware*), consegue atender os objetivos propostos e se mostra competitivo em relação à outros aplicativos com funções semelhantes.

Palavras-Chave: Arduino, Android, Simulador.

Abstract

Introduced in 2005, the Arduino boards has become increasingly popular mainly because of its simplicity and its reduced cost. This work presents project SOFIA, an Android simulator as an alternative to Arduino boards. Taking advantage of Arduino's Open-Source code, it was possible to make changes in the official IDE to create an integrated Computer-Smartphone system, making it easier to use the application and getting closer to a real development experience. The result was a simulator capable of running simple projects written for Arduino UNO that achieves its objectives (although it still lacks in performance when compared with hardware implementations) and it's competitive against other applications with similar functions.

Keywords: Arduino, Android, Simulator.

Listas de Figuras

2.1	Diagrama de blocos da organização interna do ATmega328P	30
2.2	Diagrama de blocos da organização da CPU	31
2.3	Memória de programa ATmega328P	33
2.4	Memória de dados ATmega328P	35
2.5	Organização do módulo de Entrada/Saída (E/S) do ATmega328P	36
2.6	Organização do módulo de <i>Timer 0/2</i> do ATmega328P	38
2.7	Organização do módulo de <i>Timer 1</i> do ATmega328P. Para contar em 16-bits, os registradores TCNTn, ICRn, OCRnA e OCRnB são divididos em dois registradores de 8-bits (<i>LOW</i> e <i>HIGH</i>)	39
2.8	Diagrama de funcionamento do modo normal	40
2.9	Diagrama de funcionamento do modo CTC	40
2.10	Diagrama de funcionamento do modo <i>fast PWM</i>	41
2.11	Diagrama de funcionamento do modo PWM com Correção de Fase	42
2.12	Diagrama de funcionamento do modo PWM com Correção de Fase e Frequência	42
2.13	Alinhamento do resultado nos registradores ADCH e ADCL	43
2.14	Organização do conversor A/D	44
2.15	Organização da USART	46
2.16	Formato de um <i>frame</i> transmitido pela USART	47
2.17	Fluxo do processo <i>SCRUM</i> . A figura mostra o fluxo completo considerando <i>sprints</i> mensais e reuniões diárias. Neste projeto, os <i>sprints</i> foram semanais e as etapas em grupo foram omitidas.	49
3.1	Diagrama de classes das modificações realizadas na IDE do Arduino	54
3.2	Arquitetura do simulador	56
4.1	Localização do botão "Android"(selecionado) na IDE	61
4.2	Seletor de dispositivos	62

4.3	Cópia do arquivo realizada com sucesso para o <i>smartphone</i>	62
4.4	<i>Splash Screen</i> exibida ao abrir o simulador	63
4.5	Tela inicial do simulador	63
4.6	Falha ao abrir arquivo hexadecimal	65
4.7	Saídas digitais do simulador	65
4.8	Entradas digitais do simulador	65
4.9	Condição de círculo-círculo entre entradas	66
4.10	Entradas analógicas do simulador	66
4.11	Remoção manual de pinos de saída	67
4.12	Monitor Serial	67
4.13	Mapa de memória	68
4.14	Recurso de busca do mapa de memória	68
4.15	Configuração de tensão externa aplicada ao pino AREF, usada como base para conversão A/D	69
4.16	Ícone do frequencímetro	70
4.17	Frequencímetro em funcionamento	70
4.18	Gráfico linhas de código x esforço para resolução dos <i>bugs</i>	72
4.19	Gráfico linhas de código x esforço para resolução das vulnerabilidades	73
4.20	Gráfico linhas de código x esforço para resolução dos <i>Code Smells</i>	74
4.21	Gráfico linhas de código x linhas de código duplicadas.	75
4.22	Medida de cobertura do projeto obtida com o <i>Android Studio</i>	76
4.23	Medida de cobertura do projeto obtida com o <i>JaCoCo</i>	76
4.24	Cobertura por módulo (<i>Android Studio</i>)	77
4.25	Cobertura por módulo (<i>JaCoCo</i>)	77
4.26	Uso de CPU e memória do aplicativo para o projeto <i>Blink</i>	78
4.27	Tempo de uso da CPU (medido em uma janela de 5 minutos)	78
4.28	Tempo de uso da CPU após melhoria no método <i>run</i> (medido em uma janela de 5 minutos)	79
4.29	Consumo de memória (medido em uma janela de 5 minutos)	79
4.30	Tela principal do simulador <i>SOFIA</i>	81
4.31	Tela principal do simulador <i>BoardMicro - AVR Simulator</i>	82
4.32	Tela principal do simulador <i>BoardMicro - AVR Simulator</i> , versão para web	82
4.33	Tela principal do simulador <i>AndMCU</i>	83
4.34	Tela principal do simulador <i>Arduino Simulator Mini Free</i>	84

4.35 Tela de simulação do projeto <i>Blink</i> . O monitor serial no canto inferior direito pode ser escondido por meio do botão "Console"	84
4.36 Tela edição do código-fonte a ser simulado.	85
4.37 Distribuição acumulada de usuários Android. Como destacado, o simulador SOFIA poderá atingir pouco mais de 70% dos dispositivos Android ativos no momento, porém este número tente a aumentar com a modernização dos aparelhos	88
4.38 Tamanho do arquivo APK (MB) de cada aplicativo	88
4.39 Consumo médio de CPU (%) de cada aplicativo. Para cada processo, foram coletadas 397 amostras	89
4.40 Consumo médio de Bateria (%) de cada aplicativo em um período de 2 horas, com cada aplicativo executando o projeto <i>Blink</i> continuamente por 30 minutos. Foram realizadas 5 medições para cada aplicativo.	90
4.41 Consumo de memória (PSS em MB) de cada aplicativo. Foram realizadas 1000 medições para cada aplicativo	91

Lista de Tabelas

2.1	Vetor de interrupções ATmega328P	32
2.2	Formato dos registros do arquivo hexadecimal no padrão Intel	33
2.3	Modos de disparo do conversor A/D	45
4.1	Comparação das medições de frequência (<i>duty cycle</i> : 50%). Foram coletadas 110 amostras em cada frequência medida	71
4.2	Comparação das medições de <i>duty cicle</i> (projeto: <i>Timer1</i> , frequência: 488Hz). Foram coletadas 111 amostras para cada valor de <i>duty cycle</i> medido	71
4.3	Versão do Android requerida por cada aplicativo	87

Siglas

A/D	Analógico/Digital
ADB	<i>Android Debug Bridge</i> - Ponte de Depuração do Android
APK	<i>Android Package Kit</i> - Kit de Pacote do Android
CAN	<i>Controller Area Network</i> - Rede de Controladores de Área
CPU	<i>Central Process Unit</i> - Unidade Central de Processamento
CS	<i>Code Segment</i> - Segmento de código
CTC	<i>Clear Timer on Compare Match</i> - Limpar temporizador na correspondência de comparação
DIP	<i>Dual In-line Package</i> - Encapsulamento Duplo em Linha
EEPROM	<i>Electrically-Erasable Programmable Read-Only Memory</i> - Memória apenas de leitura programável e apagável eletronicamente.
E/S	Entrada/Saída
GDB	GNU Debugger
GVfs	<i>GNOME Virtual file system</i> - Sistema de arquivo virtual do GNOME
I^2C	<i>Inter-Integrated Circuit</i> - Circuito Inter-integrado
IDE	<i>Integrated Development Environment</i> - Ambiente de desenvolvimento integrado
IP	<i>Instruction Pointer</i> - Ponteiro de Instrução
IoT	<i>Internet of Things</i> - Internet das Coisas
LCD	<i>Liquid Crystal Display</i> - Display de Cristal Líquido
LDR	<i>Light Dependent Resistor</i> - Resistor Dependente de Luz
LED	<i>Light Emitting Diode</i> - Diodo Emissor de Luz
MIPS	<i>Millions of Instructions Per Second</i> - Milhões de Instruções por Segundo
MTP	<i>Media Transfer Protocol</i> - Protocolo de Transferência de Mídia
PC	<i>Program Counter</i> - Contador de Programa
PCB	<i>Printed Circuit Board</i> - Placa de Circuito Impresso
PDIP	<i>Plastic Dual In-line Package</i> - Encapsulamento de Plástico Duplo em Linha
PSS	<i>Proportional Set Size</i> - Tamanho Proporcional do Conjunto

PWM	<i>Pulse Width Modulation</i> - Modulação por Largura de Pulso
RISC	<i>Reduced Instruction Set Computer</i> - Computador com Conjunto de Instruções Reduzido
SDRAM	<i>Synchronous Dynamic Random-Access Memory</i> - Memória de acesso aleatório dinâmica síncrona
SPI	<i>Serial Peripheral Interface</i> - Protocolo Serial de Interface Periférica
TFT	<i>Thin Film Transistor</i> - Transistor de Película Fina
TWI	<i>Two-Wire Serial Interface</i> - Interface serial de duas linhas
USART	<i>Universal Synchronous Asynchronous Receiver Transceiver</i> - Transmissor/Receptor Universal Síncrono e Assíncrono

Sumário

1 Introdução	23
1.1 Motivação	25
1.2 Objetivo	27
1.3 Justificativa	27
1.4 Organização do Trabalho	27
2 Embasamento Teórico	29
2.1 Microcontrolador ATmega328P	29
2.1.1 Visão Geral	29
2.1.2 CPU	30
2.1.3 Memória de Programa	33
2.1.4 Memória de Dados	34
2.1.5 Módulo de Entrada e Saída Digital	35
2.1.6 Temporizadores	37
2.1.6.1 Modo Normal	37
2.1.6.2 Modo CTC	37
2.1.6.3 Modo <i>Fast PWM</i>	40
2.1.6.4 Modo PWM com Correção de Fase	41
2.1.6.5 Modo PWM com Correção de Fase e Frequência	42
2.1.6.6 Captura de Eventos	42
2.1.7 Conversor A/D	43
2.1.8 USART	46
2.2 Processo de <i>Software</i>	49
2.3 Teste de <i>Software</i>	50
3 Desenvolvimento do Projeto	51
3.1 Material	51

3.2 Método	54
3.2.1 Desenvolvimento na IDE do Arduino	54
3.2.2 Desenvolvimento Android	55
3.2.2.1 Módulo principal	55
3.2.2.2 CPU	57
3.2.2.3 Memória de Programa	57
3.2.2.4 Memória de Dados	58
3.2.2.5 Módulo de Interrupção	58
3.2.2.6 Módulo de E/S	59
3.2.2.7 Temporizadores	59
3.2.2.8 Conversor A/D	60
3.2.2.9 USART	60
4 Resultados e Discussões	61
4.1 Arduino IDE	61
4.2 Simulador	63
4.2.1 Interação com o sistema	64
4.2.2 Monitor Serial	67
4.2.3 Mapa de memória	68
4.2.4 Referência externa de tensão	69
4.2.5 Medição de frequência	69
4.3 Métricas de <i>Software</i>	72
4.3.1 Análise Estática	72
4.3.2 Cobertura	75
4.4 <i>Profiling</i>	76
4.5 Comparação entre aplicativos	80
4.5.1 Interface	80
4.5.2 Funcionalidades oferecidas	85
4.5.3 Requisitos do sistema e consumo de recursos	87
4.5.3.1 Sistema Operacional	87
4.5.3.2 Tamanho	87
4.5.3.3 Uso de CPU	89
4.5.3.4 Uso de Memória	90
4.5.4 Velocidade de simulação	91

4.5.5 Documentação	92
4.5.6 Disponibilidade	93
4.6 Código	94
5 Conclusão	95
5.1 Trabalhos futuros	96
Referências	97
Apêndice A Projeto <i>Blink</i>	103
Apêndice B Projeto <i>Blink</i> em assembly	105
Apêndice C Projeto <i>Input to Output</i>	107
Apêndice D Projeto <i>Interrupt</i>	109
Apêndice E Projeto <i>Timer</i>	111
Apêndice F Projeto <i>Analog Input</i>	113
Apêndice G Projeto <i>Serial</i>	115
Apêndice H Projeto <i>Analog Serial</i>	117
Apêndice I Projeto <i>Memory Measure</i>	119

Capítulo 1

Introdução

Os dispositivos móveis vem ganhando cada vez mais espaço no cotidiano das pessoas por trazer funcionalidades diversas em um dispositivo cada vez mais barato e portátil. Seja para fazer uma simples operação matemática, ou para navegar na *web*, tirar fotos, telefonar, etc., os *smartphones* e *tablets* tem evoluído cada vez mais tanto em questões de *hardware* quanto em *software*.

Em se tratando de *hardware*, os dispositivos móveis hoje carregam um grande poder computacional. Segundo uma matéria do Olhar Digital [1], o *desktop* top de linha em 2001 possuía 80 GB de HD, 128 MB de memória primária e um processador *single core* de 1,53 GHz. Hoje, um *smartphone* top de linha possui 8 núcleos de processamento, com frequências de até 2,8 GHz, memória primária de 4 GB e armazenamento interno de 256 GB (com possibilidade de expansão) [2], tudo isso em um *design* compacto que cabe no bolso. Graças à essa evolução, é possível realizar tarefas cada vez mais complexas em um dispositivo móvel.

Quanto a *software*, hoje existe uma maior padronização dos sistemas *mobile*, o que facilita o desenvolvimento de aplicações. O sistema operacional para dispositivos móveis líder de mercado é o Android [3]. Tendo sua primeira versão lançada em 2008, diversos foram os motivos pela sua grande popularidade, tais como o fato de ser gratuito, *open-source* (sob a licença *Apache 2.0*, principalmente [4]), o desenvolvimento em conjunto com as empresas interessadas (*Open Handset Alliance - OHA*) e o uso do Java para a criação de aplicativos, uma vez que essa é a linguagem de programação mais utilizada mundialmente [5], além de entregar ao usuário um sistema moderno, elegante e cheio de recursos.

Devido a constante presença dos dispositivos móveis (com sistema Android) e sua crescente capacidade de *hardware*, esta plataforma tem sido cada vez mais explorada por desenvolvedores. Uma prova disso é a loja oficial de aplicativos do Android (*Google Play Store*), que oferece uma infinidade de aplicativos, de calculadoras a jogos com gráficos realistas, muitos deles disponibilizados gratuitamente.

mente. É por este motivo também que esta plataforma foi escolhida para a realização deste trabalho, que visa desenvolver um simulador das funcionalidades de um Arduino UNO.

A utilização do poder computacional de *smartphones / tablets* em substituição à sistemas tradicionais não é uma ideia nova. Pode-se citar trabalhos como o de Junior [6], que utilizou um dispositivo Android para a implementação de um osciloscópio de baixo custo. Utilizando um microcontrolador ARM Cortex M4F para aquisição dos sinais e comunicação *Bluetooth* com o *smartphone*, foi possível construir um osciloscópio com um custo de projeto de US\$35, com erro médio de 0,2% no eixo do tempo, 0,02V no eixo da tensão e taxa máxima de aquisição de 150 mil amostras por segundos, o que, segundo o autor, torna este um sistema "aceitável para o uso do projeto no ambiente de aprendizado", considerando a diferença de preço com osciloscópios comerciais.

Em uma abordagem semelhante, Eberendu et al. [7] utiliza um *smartphone*, junto à uma placa de Arduino UNO, para a criação de um microscópio. O chamado *SmartScope* utiliza uma estrutura de suporte com uma lente plano-convexa para adaptar a câmera do *smartphone* a captura de imagens microscópicas. A placa de Arduino controla o LED (*Light Emitting Diode*) que serve como fonte de luz e permite o ajuste de intensidade do brilho por meio de botões, mostrando em um *display LCD (Liquid Crystal Display)* a configuração atual. O sistema tem funções de captura de imagem e vídeo e permite o armazenamento dos dados coletados em um banco de dados (*Microsoft Access*). Assim como o trabalho de Junior, o grande objetivo é criar um sistema de baixo custo como alternativa aos microscópios comerciais e ajudar alunos sem experiência a aprender a realizar leituras no equipamento.

Também é possível citar o trabalho de Teng et al. [8] que vai além do nível de aplicativo, fazendo modificações no *kernel Linux* para leitura de dados em um barramento CAN (*Controller Area Network*). Teng e sua equipe desenvolveram *drivers* e bibliotecas para realizar a leitura de dados de sensores em um automóvel por meio do barramento CAN. Dados relativos à velocidade, faróis, temperatura, chaves e alarme foram lidos diretamente das unidades de controle do veículo e mostrados na tela do *smartphone* em um aplicativo de instrumentação próprio, simulando o painel do carro. Seu trabalho evidencia que as possibilidades de uso dos sistemas Android podem se estender também para o nível de sistema.

Com uma pesquisa na *Google Play Store*, é possível encontrar diversos aplicativos relacionados com Arduino. Muitos deles se encontram na forma de "aplicativo-tutorial", mostrando exemplos de código e diagramas para o ensino da plataforma, mas também é possível encontrar IDEs (*Integrated Development Environment*) com capacidade de gravação das placas físicas, geradores de código automático, módulos para serem utilizados em projetos de automação (como controles *wireless* que se integram às *shields* do Arduino), etc.

Na parte de simuladores, vários foram os aplicativos de simulação/emulação de processadores e

microcontroladores encontrados. Um destaque fica para o aplicativo *AndMCU*¹ (também conhecido como *MCU Prototype Board Simulator*), que simula um *kit* de desenvolvimento (com botões, LEDs, LCD, etc.) e permite a execução de códigos *assembly* de um *subset* das instruções do microcontrolador 68705 da Motorola.

Quanto à simulação de placas de Arduino, pode-se citar o *BoardMicro - AVR Simulator*², que destina-se à simulação do microcontrolador ATMega32U4 a partir de um arquivo hexadecimal fornecido pelo usuário, e se baseia no Arduino Esplora. Também, o *CircSim Circuit Simulator*³ se propõe a fazer simulação de, entre outras coisas, placas de Arduino, no entanto, seu uso é praticamente impossibilitado dado que sua interface não se ajusta adequadamente em dispositivos móveis (fato que pode ser comprovado pelos comentários dos usuários na página do aplicativo).

Fora da loja oficial, foi encontrado ainda o *Arduino Simulator Mini Free*⁴, que é destinado à simulação do Arduino Diecimila, no entanto possui limitações por permitir a simulação apenas dos programas que vem junto com o aplicativo.

Sendo assim, o presente trabalho surge também para complementar esta área de aplicativos destinadas à simulação, oferecendo mais uma possibilidade para explorar o Arduino, seja por estudantes, hobistas ou qualquer pessoa que se interesse pelo assunto.

1.1 Motivação

O Arduino é uma plataforma de *hardware* e *software* abertos, destinada ao desenvolvimentos de projetos na área de eletrônica. Suas aplicações vem crescendo a cada dia e vão desde o uso educacional para ensino de robótica em escolas de ensino fundamental, até aplicações em IoT (*Internet of Things*) nas universidades.

Uma das principais características do Arduino é sua simplicidade. Em suas versões mais básicas, utilizam microcontroladores de 8-bits, o que torna o sistema menos complexo e mais barato quando comparada com outras plataformas concorrentes (tais como *Raspberry Pi* e *BeagleBone*) [9]. Com isso, se tornou uma plataforma ideal para prototipagem e para aprendizado, englobando um público-alvo de artistas, sem conhecimento prévio de eletrônica e programação, à engenheiros experientes, que usam a plataforma para prototipagem.

O desenvolvimento para essa plataforma demanda o uso de placas físicas de Arduino. Essas placas foram desenvolvidas com o objetivo de apresentarem baixo custo e facilitar a prototipagem, se

¹https://play.google.com/store/apps/details?id=com.hkonstas.andmcu&hl=en_US

²<https://www.amazon.com/Starlo-BoardMicro-AVR-Simulator/dp/B00LKX3VZC>

³https://play.google.com/store/apps/details?id=org.qtproject.CircSim&hl=pt_BR

⁴<https://www.amazon.com.br/Schogini-Systems-Arduino-Simulator-Mini/dp/B00LIUIX6Y>

integrando facilmente à módulos externos (*shields*) que adicionam ao sistema funções de interface, sensoriamento, etc. A desvantagem de se utilizar placas eletrônicas é, além do custo para adquiri-las, a necessidade de outros componentes externos, como LEDs, *protoboards*, multímetros, etc.

Uma opção às placas de Arduino são os simuladores. Para um usuário iniciante, o simulador representa a possibilidade de iniciar seus estudos sem a necessidade de gastos com equipamentos eletrônicos e sem o risco de perder estes equipamentos por uso indevido. Para usuários experientes, um simulador permite explorar diferentes ambientes/situações de funcionamento, além de outros benefícios como o monitoramento interno do sistema, vasta gama de equipamentos eletrônicos virtuais, equipamentos de medição (voltímetro, osciloscópio, etc.), entre outros.

Diversas são as opções de simuladores existentes, cada qual com características próprias. Alguns que merecem destaque são:

- **Proteus**: Possui recursos para montagem e simulação de circuitos eletrônicos analógicos e digitais, bem como desenvolvimento de *layouts PCB* (*Printed Circuit Board*). Tem como desvantagem o fato de ser um *software* comercial, com a versão básica para simulação Arduino custando US\$248,00 [10]
- **Virtual Breadboard**: Permite a simulação de circuitos eletrônicos digitais em um ambiente virtual, bem como a programação de microcontroladores dentro do próprio sistema. Possui uma versão gratuita, no entanto, para simulação de Arduino é preciso comprar um módulo separadamente pelo valor de US\$49,00 [11].
- **Simuino**: Simulador de Arduino UNO e MEGA para terminal. Apesar de gratuito e *Open-Source*, possui apenas versões para Linux e é distribuído apenas em formato de código-fonte, sendo necessário fazer a compilação antes de usá-lo [12].
- **CodeBlocks**: Possui uma versão com ferramentas próprias para escrever e simular códigos Arduino, permitindo também o *upload* de código para placas físicas, com suporte a diversos modelos de *hardware*. A desvantagem fica por conta de não possuir uma interface gráfica para simulação, que ocorre toda em terminal apenas com textos indicando os estados de entradas e saídas [13].
- **Autodesk Tinkercad**: É um ambiente de aprendizado *on-line* gratuito que permite tanto a criação de projetos eletrônicos quanto de desenhos 3D. Possui um ambiente para codificação e depuração do código na própria ferramenta. A única desvantagem encontrada é o fato de funcionar apenas *on-line*, sendo necessário fazer um cadastro para utilizá-la [14].

- **UnoArduSim:** Desenvolvido pelo Professor Stan Simmons, da Universidade Queen's, no Canadá, o simulador conta com uma IDE integrada para codificação e um laboratório virtual com diversos equipamentos, tais como motores, geradores de sinal, LEDs, etc. É uma ferramenta gratuita com suporte apenas ao Arduino UNO [15].

É interessante destacar que o projeto aqui proposto é, em essência, um simulador do ATmega328P em nível de arquitetura, com o Arduino sendo utilizado como uma interface ao dispositivo. Isso significa que o usuário terá a possibilidade de estudar o processador em um nível mais baixo e não apenas utilizar funções em alto nível do Arduino (como é o caso de alguns simuladores), o que não é nenhuma imposição ou limitação do projeto, já que estas questões podem ser abstraídas caso o usuário não tenha interesse neste nível de detalhe.

Apesar da grande quantidade de simuladores já existentes, como foi mostrado anteriormente, há ainda uma falta deste tipo de *software* para dispositivos móveis. A grande motivação deste trabalho é poder contribuir com a comunidade que deseja aprender sobre desenvolvimento para Arduino, fornecendo uma opção de simulador para Android.

1.2 Objetivo

O objetivo principal do projeto é o desenvolvimento de um aplicativo Android que possibilite a execução de programas desenvolvidos para a plataforma Arduino UNO, permitindo assim a realização de testes sem a necessidade de uma placa de física ou qualquer outro componente eletrônico externo.

Também faz parte dos objetivos do trabalho a integração do simulador com a IDE oficial do projeto Arduino, permitindo a transferência dos códigos compilados para o simulador de maneira automática.

1.3 Justificativa

Este trabalho se justifica por atuar de forma a criar uma ferramenta *Open-Source* que beneficiará todos aqueles que desejam aprender/praticar o desenvolvimento para Arduino.

1.4 Organização do Trabalho

Este trabalho está distribuído em 5 capítulos, incluindo esta introdução, dispostos conforme a descrição que segue:

Capítulo 2: Apresenta um embasamento teórico, explicando o funcionamento dos módulos do microcontrolador ATmega328P implementados do simulador, bem como informações a respeito de processo e teste de *software* que se aplicam ao trabalho.

Capítulo 3: Descreve o processo de desenvolvimento do projeto e as principais estratégias de implementação. Além disso, apresenta as ferramentas utilizadas durante o desenvolvimento.

Capítulo 4: Discorre sobre os resultados obtidos, apresentando o sistema, algumas métricas de *software* e comparações feitas com outros aplicativos semelhantes. Também é feita uma discussão a respeito destes resultados.

Capítulo 5: Conclui a respeito do trabalho desenvolvido até o momento e apresenta algumas perspectivas para a continuação do projeto.

Capítulo 2

Embasamento Teórico

Neste capítulo será explicado o funcionamento e as características de cada módulo presente no microcontrolador do Arduino UNO (ATmega328P) que foi implementados no simulador, além de uma breve teoria a respeito de processo de desenvolvimento e teste de *software*. No que diz respeito ao ATmega328P, todas as informações foram retiradas da folha de dados do componente [16].

2.1 Microcontrolador ATmega328P

2.1.1 Visão Geral

O ATmega328P é um microcontrolador RISC (*Reduced Instruction Set Computer*) de 8-bits e arquitetura *Harvard* (memória de dados separada da memória de programa). Possui 28 pinos no encapsulamento PDIP (*Plastic Dual In-line Package*), sendo 23 programáveis e pode trabalhar com frequência máxima de operação de 20MHz.

Entre os periféricos que estão integrados neste dispositivo, pode-se listar:

- Dois temporizadores de 8-bits com *prescaler* separados;
- Um temporizador de 16-bits;
- 6 canais de PWM (*Pulse Width Modulation*);
- Conversor A/D (Analógico/Digital) de 10-bits (8 canais multiplexados);
- Duas interfaces de comunicação serial SPI (*Serial Peripheral Interface*);
- Uma USART (*Universal Synchronous Asynchronous Receiver Transceiver*) serial;
- Uma interface serial TWI (*Two-Wire Serial Interface*), compatível com I²C (*Inter-Integrated Circuit*) da Philips;

- *Watchdog Timer* programável com oscilador separado;
- Entre outros.

A figura 2.1 apresenta um diagrama de blocos da organização interna do microcontrolador.

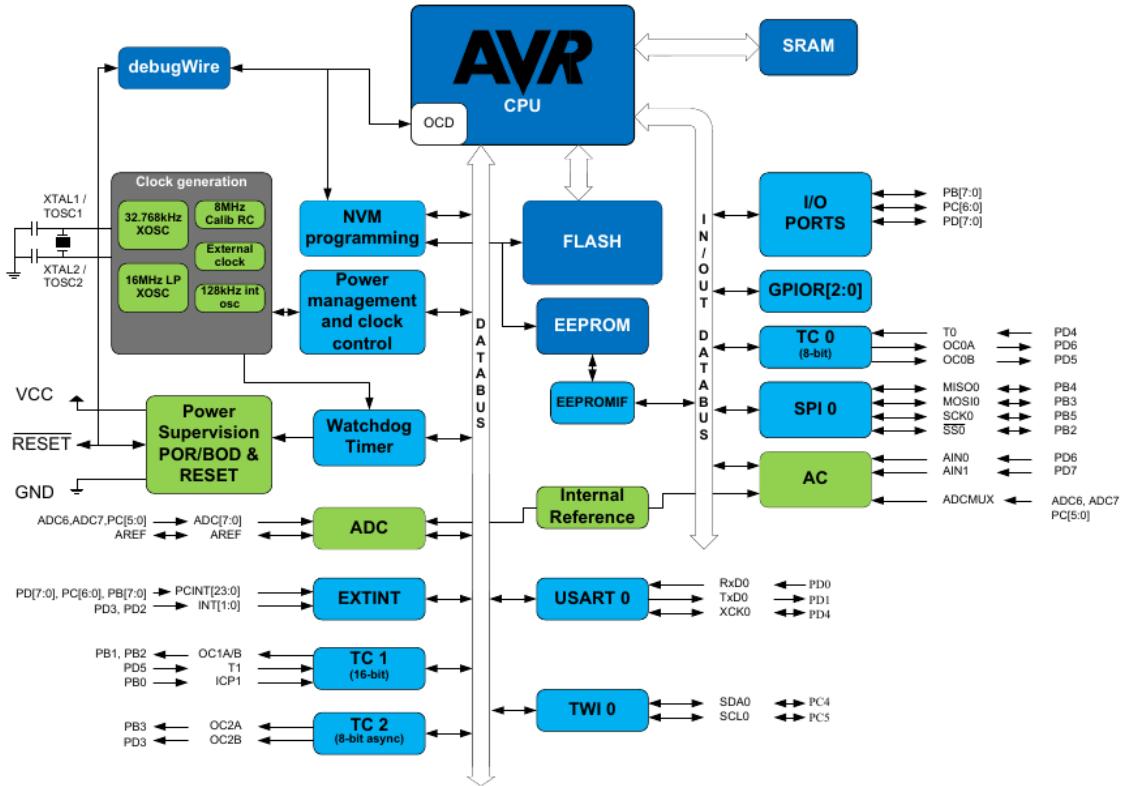


Figura 2.1: Diagrama de blocos da organização interna do ATmega328P

Fonte: Folha de dados ATmega328P

2.1.2 CPU

A CPU (*Central Process Unit*) do ATmega328P é apresentada na figura 2.2. Ela possui um banco de 32 registradores de 8-bits, com os 6 últimos podendo ser utilizados como registradores de 16-bits (chamados de registrador X (R27:R26), Y(R29:R28) e Z(R31:R30)); PC (*Program Counter*) de 14-bits; Registrador de *status* (8-bits), que armazena as *flags* geradas por cada operação aritmética/lógica (zero, *carry*, *overflow*, etc); *Stack Pointer* de 16-bits e demais registradores auxiliares. A CPU utiliza um estágio de *pipeline* que, junto com a arquitetura *Harvard*, permite que o sistema atinja uma velocidade máxima de 1 MIPS/MHz (*Millions of Instructions Per Second/MHz*).

Em chamadas de sub-rotinas e interrupções, a CPU utiliza uma pilha implementada diretamente na memória SDRAM (*Synchronous Dynamic Random-Access Memory*), cujo topo é apontado pelo registrador *Stack Pointer*. Esta estrutura de dados cresce do endereço mais alto da memória para o

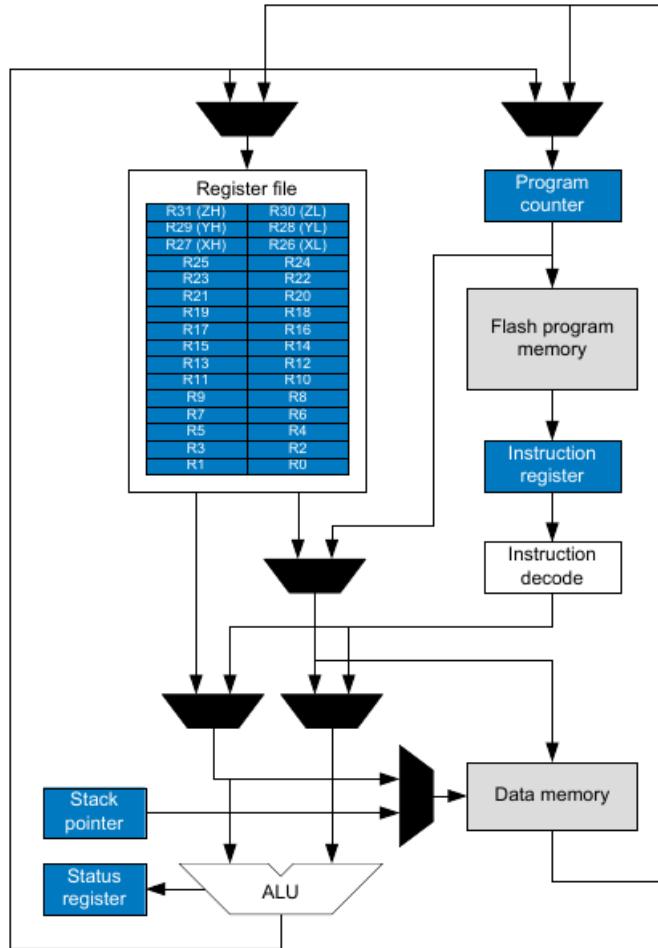


Figura 2.2: Diagrama de blocos da organização da CPU

Fonte: Folha de dados ATmega328P

endereço mais baixo, de forma que o *Stack Pointer* deve ser corretamente inicializado para o último endereço da memória SDRAM antes de ser utilizado.

As interrupções no ATmega328P são organizadas segundo sua prioridade. A tabela 2.1 mostra o vetor de interrupções, contendo o endereço de desvio para cada tipo de interrupção. Quanto mais baixo o endereço, maior é a prioridade (o *RESET* é a interrupção de maior prioridade no sistema). Importante ressaltar que as interrupções são desabilitadas automaticamente ao iniciar o tratamento de uma rotina de interrupção (e reabilitadas ao terminar), no entanto, este comportamento pode ser alterado por *software*, reabilitando as interrupções no começo da rotina.

As interrupções são classificadas em duas classes: as disparadas por evento e as disparadas por uma condição. Quando as interrupções são disparadas por eventos, é habilitada uma *flag* toda vez que o evento ocorre e, se a interrupção deste evento estiver ativa, ela será tratada ou enfileirada para tratamento posterior. Ou seja, em interrupções por evento, os eventos que não são tratados, são lembrados, e serão executados em ordem de prioridade assim que possível.

Tabela 2.1: Vetor de interrupções ATmega328P

Endereço de Desvio	Interrupção	Descrição
0x00	RESET	Interrupção de Reset
0x02	INT0	Interrupção Externa 0
0x04	INT1	Interrupção Externa 1
0x06	PCINT0	Interrupção de mudança de estado 0
0x08	PCINT1	Interrupção de mudança de estado 1
0x0A	PCINT2	Interrupção de mudança de estado 2
0x0C	WDT	Estouro do <i>Watchdog Timer</i>
0x0E	TIMER2_COMPA	Comparação <i>Timer 2</i> canal A
0x10	TIMER2_COMPB	Comparação <i>Timer 2</i> canal B
0x12	TIMER2_OVF	Estouro do <i>Timer 2</i>
0x14	TIMER1_CAPT	Captura de evento <i>Timer 1</i>
0x16	TIMER1_COMPA	Comparação <i>Timer 1</i> canal A
0x18	TIMER1_COMPB	Comparação <i>Timer 1</i> canal B
0x1A	TIMER1_OVF	Estouro do <i>Timer 1</i>
0x1C	TIMER0_COMPA	Comparação <i>Timer 0</i> canal A
0x1E	TIMER0_COMPB	Comparação <i>Timer 0</i> canal B
0x20	TIMER0_OVF	Estouro do <i>Timer 0</i>
0x22	SPI STC	Transferência SPI completa
0x24	USART_RX	Recepção USART completa
0x26	USART_UDRE	Registrador de dados vazio (USART)
0x28	USART_TX	Transmissão USART completa
0x2A	ADC	Conversão analógico-digital completa
0x2C	EE READY	EEPROM pronta
0x2E	ANALOG COMP	Comparador analógico
0x30	TWI	Interface serial I^2C
0x32	SPM READY	Armazenamento na memória de programa

Já quando o disparo ocorre por uma condição, a chamada para a rotina de interrupção permanece ativa enquanto a condição estiver presente. Este tipo de interrupção não necessariamente habilita *flag* de modo que, se a condição for removida antes que a CPU possa tratar a interrupção correspondente, esta não ocorrerá.

2.1.3 Memória de Programa

A memória de programa é uma *Flash* de 32kB x 8-bits, que está organizada da forma 16kB x 16-bits pois cada instrução do microcontrolador é de 16 ou 32-bits. Assim, o registrador PC de 14-bits pode fazer um endereçamento a palavra na memória de programa.

A figura 2.3 mostra a organização da memória de programa. Pode-se notar que o *Boot Loader* está posicionado em uma seção separada do restante da memória e isso ocorre por questões de segurança.

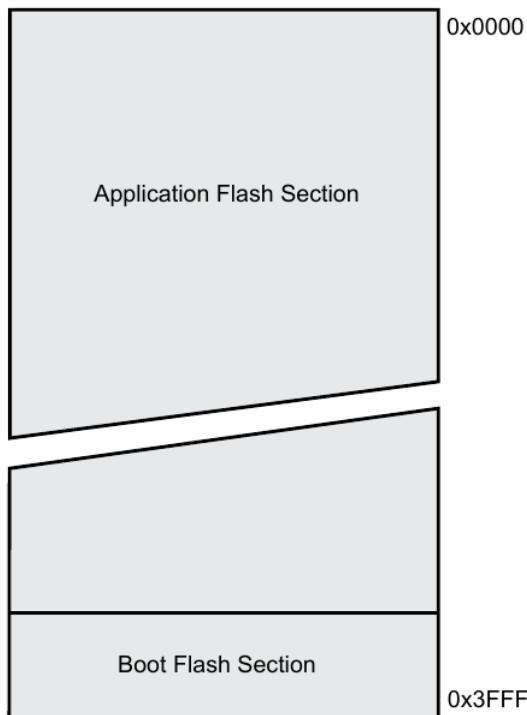


Figura 2.3: Memória de programa ATmega328P

Fonte: Folha de dados ATmega328P

As instruções a serem preenchidas na memória de programa são fornecidas pelo arquivo hexadecimal gerado pelo compilador. Este arquivo segue o padrão Intel e está disposto conforme mostra a tabela 2.2 [17]:

Tabela 2.2: Formato dos registros do arquivo hexadecimal no padrão Intel

Início do registro	Contagem de bytes	Endereço	Tipo de registro	Dado	<i>Checksum</i>
--------------------	-------------------	----------	------------------	------	-----------------

- **Início do registro:** Denotado pelo símbolo ":"(1 byte do registro).
- **Contagem de bytes:** Indica a quantidade de bytes a serem lidos no campo de dados (1 byte do registro).
- **Endereço:** Indica o endereço de memória no qual deve se iniciar o preenchimento dos dados. Este endereço pode não ser igual ao endereço físico da memória (2 bytes do registro).
- **Tipo do registro:** Indica o que o registro representa (1 byte do registro), podendo ser:
 - 00: Dado
 - 01: Fim do arquivo
 - 02: Segmento estendido de memória (o valor do campo "dados" é armazenado e o endereço físico de memória dos registros seguintes é calculado como sendo o campo de endereço somado ao segmento estendido multiplicado por 16).
 - 03: Início do segmento de memória (o valor contido no campo "dados" é carregado para os registradores CS (*Code Segment*) e IP (*Instruction Pointer*) para os processadores 8086 e 80186).
 - 04: Segmento estendido de memória Linear (o valor do campo "dados" é armazenado e o endereço físico de memória dos registros seguintes é calculado como sendo o campo de endereço concatenado ao segmento estendido, sendo este último a parte mais significativa do endereço).
 - 05: Início do endereço linear (aponta para o endereço de memória onde o programa deve iniciar a execução).
- **Dado:** Contém os dados do registro no formato *Little-endian* (primeiro byte é o menos significativo da palavra) (tamanho variável no registro).
- **Checksum:** Complemento de dois do byte menos significativo da soma de todos os bytes anteriores do registro (1 byte do registro).

2.1.4 Memória de Dados

O ATmega328P possui 2kB de memória de dados SDRAM, além do espaço de dados reservado aos registradores.

Apesar dos registradores não estarem fisicamente implementados na memória de dados, o microcontrolador faz um mapeamento linear da memória de modo a se obter, na prática, uma memória como mostrado na figura 2.4.

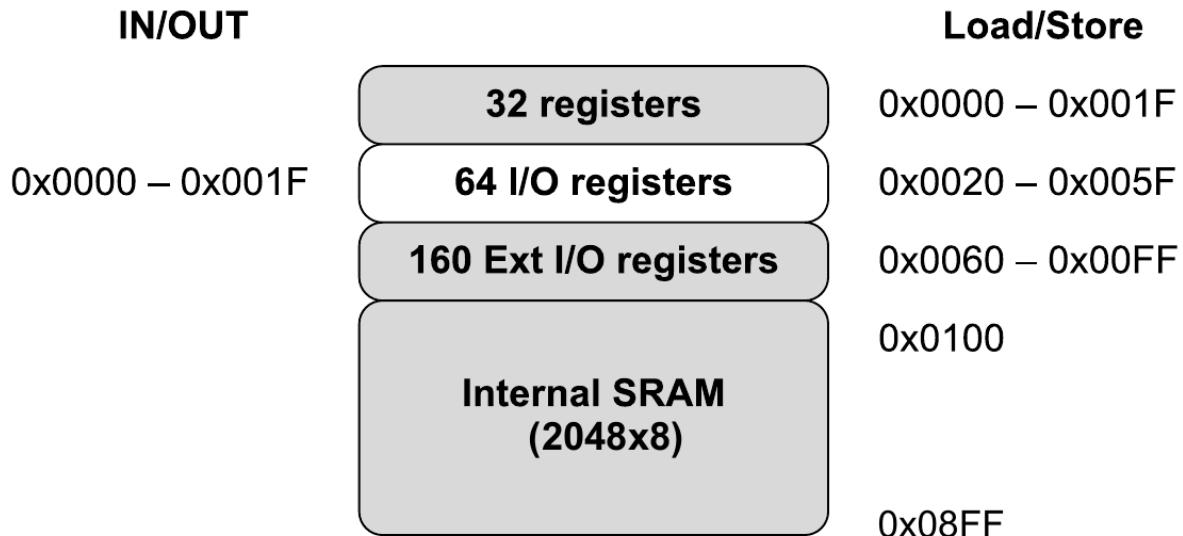


Figura 2.4: Memória de dados ATmega328P

Fonte: Folha de dados ATmega328P

Existem diferentes modos de endereçamento que são aplicados à memória de dados. Todo o espaço de endereçamento suporta qualquer um dos modos listados, são eles:

- **Direto:** Acesso direto ao endereço desejado.
- **Indireto com deslocamento:** Acesso à 63 endereços deslocados a partir do endereço base, dado pelos registradores Y ou Z.
- **Indireto:** Acesso ao endereço dado pelos registradores X, Y ou Z.
- **Indireto com pré-decremento:** Registradores X, Y ou Z são decrementados antes de serem utilizados como ponteiro para endereçamento.
- **Indireto com pós-incremento:** Registradores X, Y ou Z são incrementados após terem sido utilizados como ponteiro para endereçamento.

2.1.5 Módulo de Entrada e Saída Digital

Como dito anteriormente, o ATmega328P possui 23 pinos programáveis, que podem ser utilizados para entrada ou saída de sinal. A figura 2.5 mostra a organização interna do módulo de E/S (Entrada/Saída) do microcontrolador.

Os pinos podem ser configurados por meio dos registradores DDRxn e PORTxn, onde "x" corresponde à letra que identifica o *port* e "n" corresponde ao número do bit no registrador. O registrador DDRxn é utilizado para configuração da direção do pino (entrada ou saída), enquanto o PORTxn configura o

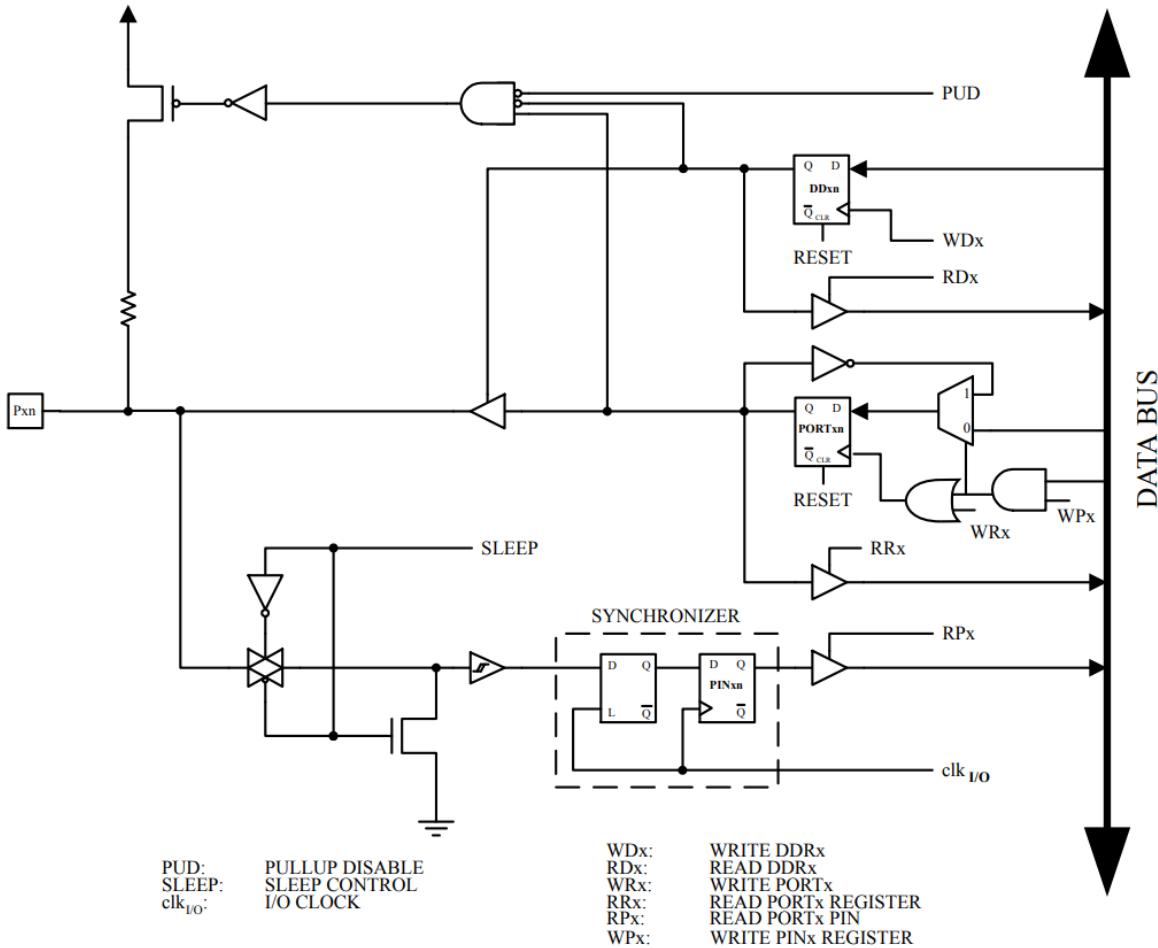


Figura 2.5: Organização do módulo de Entrada/Saída (E/S) do ATMega328P

Fonte: Folha de dados ATMega328P

estado do pino (nível alto ou baixo) se este for um pino de saída, caso contrário, seu efeito será ativar ou desativar o resistor de *pull-up* interno (se este estiver habilitado no registrador MCUCR).

Existe ainda o registrador PIN_{xn}, que é um registrador responsável por armazenar o valor da entrada do pino. Apesar de ser um registrador de leitura, é possível fazer a escrita do valor "1"(nível alto) via *software*. O efeito desta escrita será a inversão do valor contido no registrador PORT_{xn}, independente da configuração do pino como entrada ou saída.

Além de entrada e saída digital, alguns pinos possuem funções adicionais, tais como entrada analógica, saída de PWM, etc., que são multiplexadas ao funcionamento normal do pino.

O módulo de E/S pode disparar interrupções externas (pinos INT) ou interrupções por mudança de estado (pinos PCINT). Os pinos INT podem ser configurados, podendo gerar interrupção por mudança de estado, borda de subida/descida ou disparo por nível baixo. Já os pinos PCINT não podem ser configurados e apenas geram interrupção por mudança de estado na entrada. É interessante notar que as interrupções por nível são detectadas de maneira assíncrona, podendo ser utilizadas para despertar

o sistema se este estiver em determinados modos de hibernação.

2.1.6 Temporizadores

Estão integrados 3 temporizadores no ATmega328P, chamados *Timer 0*, *Timer 1* e *Timer 2*. Os modos de funcionamento disponíveis para cada temporizador são semelhantes, sendo eles: modo normal, modo CTC, *Fast PWM* e PWM com correção de fase. As fontes de *clock* para os temporizadores podem ser externa ou interna. Quanto interna, existe a possibilidade de controle da frequência por meio de um *prescaler*.

O *Timer 1* é um contador de 16-bits, enquanto os *Timers 0* e *2* são de 8-bits. O *Timer 2* possui uma função adicional de funcionamento assíncrono, podendo assim utilizar uma fonte de *clock* externa aplicada aos pinos TOSC1 e TOSC2 (os *Timers 0* e *1*, embora também possam ser acionados por *clock* externo, a detecção de borda que é realizada nos pinos T0 e T1 é feita de maneira síncrona). Quando em funcionamento assíncrono, o *Timer 2* pode ser utilizado para despertar o sistema caso este esteja em determinados modos de hibernação.

Os temporizadores podem atuar nos pinos de saída OCxA e OCxB, sobreescrivendo a operação normal do pino. Para isso, entretanto, é preciso que os pinos sejam configurados como saída no registrador DDRxn.

As figuras 2.6 e 2.7 apresentam a organização interna dos *Timers 0/2* e do *Timer 1* respectivamente. Os modos de operação disponíveis para os temporizadores são descritos a seguir.

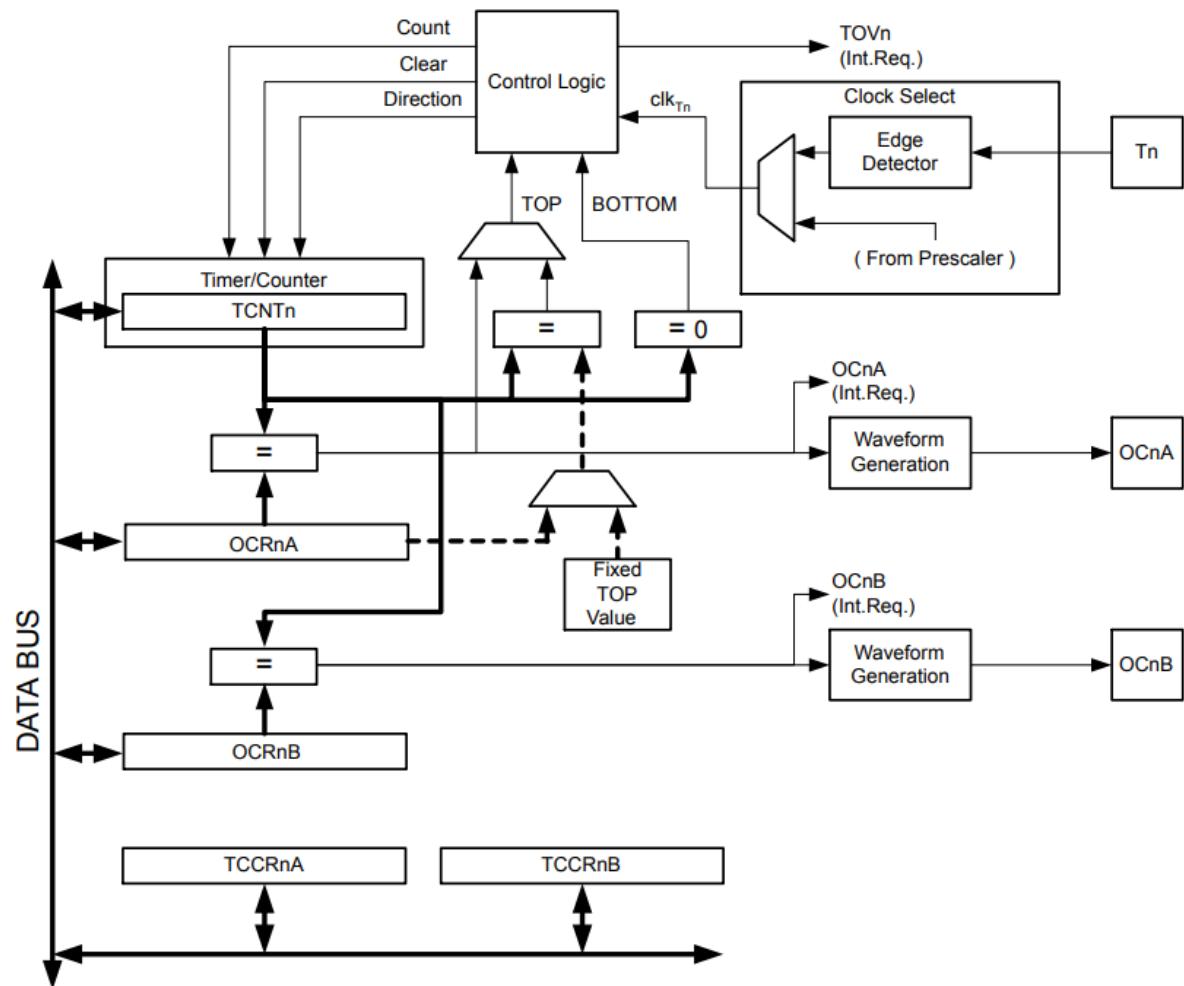
2.1.6.1 Modo Normal

O modo normal de operação é o mais simples. Nele, a contagem é feita continuamente até atingir o valor máximo (0xFF para 8-bits e 0xFFFF para 16-bits), quando ocorre um *overflow* e o sistema reinicia a contagem do zero, como mostrado no diagrama da figura 2.8. O estouro do contador pode ser utilizado para gerar uma interrupção.

Os registradores OCRnA e OCRnB são continuamente comparados com o valor de TCNTn (que armazena a contagem) e em caso de *match*, podem gerar interrupções no sistema e/ou atuar nos pinos OCxA e OCxB, podendo levá-los à nível alto, baixo ou inverter seus valores.

2.1.6.2 Modo CTC

O modo de funcionamento CTC (*Clear Timer on Compare Match*) apresenta as mesmas possibilidades do modo normal, no entanto o valor máximo de contagem é igual ao valor contido no registrador OCRxA, que pode ser ajustado a qualquer momento (para o *Timer 1*, existe também a opção de utilizar o registrador ICR1). O diagrama de funcionamento deste modo é mostrado na figura 2.9.

Figura 2.6: Organização do módulo de *Timer 0/2* do ATmega328P

Fonte: Folha de dados ATmega328P

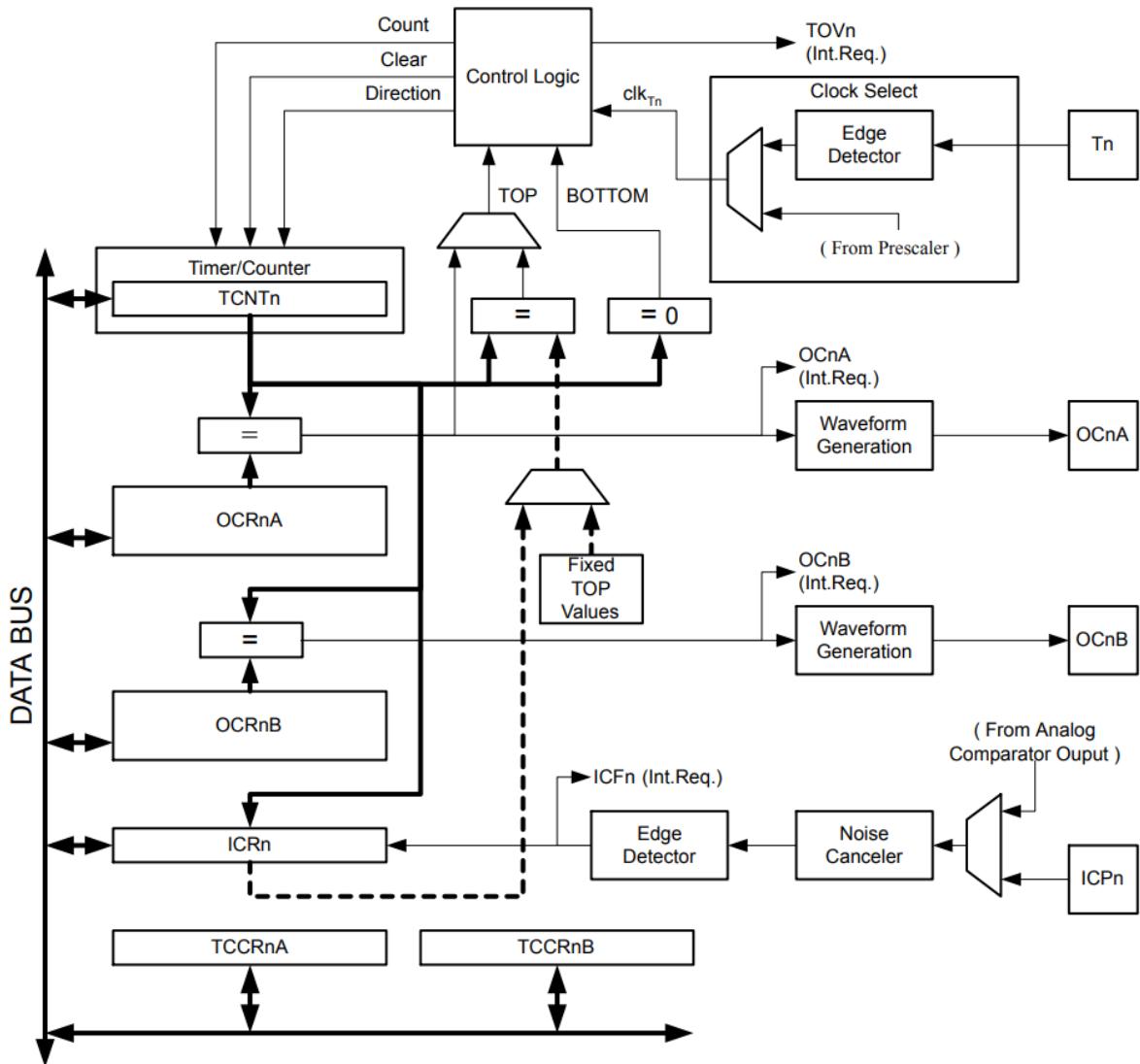


Figura 2.7: Organização do módulo de *Timer 1* do ATmega328P. Para contar em 16-bits, os registradores TCNT_n, ICR_n, OCR_{nA} e OCR_{nB} são divididos em dois registradores de 8-bits (*LOW* e *HIGH*)

Fonte: Folha de dados ATmega328P

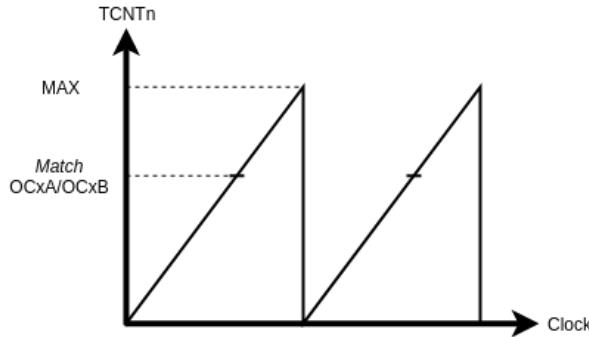


Figura 2.8: Diagrama de funcionamento do modo normal

Fonte: Autor

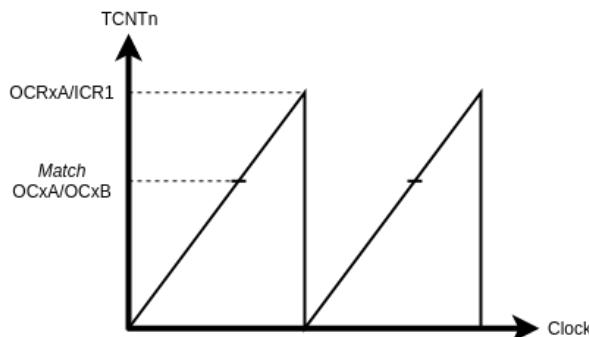


Figura 2.9: Diagrama de funcionamento do modo CTC

Fonte: Autor

No entanto, diferente do modo normal, o reinício da contagem não pode gerar interrupção de *overflow*. Isso só é possível caso OCRxA (ou ICR1) seja igual à 0xFF (8-bits) ou 0xFFFF (16-bits). Neste caso, o modo CTC e o modo normal se comportam de maneiras idênticas.

2.1.6.3 Modo *Fast PWM*

No modo *fast PWM*, assim como no modo normal, a contagem é feita continuamente do valor mais baixo (0) ao valor mais alto (0xFF para 8-bits ou, no caso do *Timer 1*, este valor pode ser configurado para 0xFF, 0x1FF ou 0x3FF), havendo a possibilidade de alteração deste valor utilizando o registrador OCRnA (ou ICR1 para o *Timer 1*), como ocorre no modo CTC. Também como no modo normal, o estouro do contador pode gerar interrupção de *overflow*. A figura 2.10 mostra o diagrama de funcionamento deste modo.

A diferença do modo *fast PWM* está na maneira como um *match* entre TCNTn e OCRnA/OCRnB é tratado. Além das possibilidades de disparo de interrupção, os pinos OCxA/OCxB podem ser configurados para apresentar nível baixo em caso de *match* e nível alto no estouro do contador (modo de funcionamento não-invertido), ou o contrário (modo de funcionamento invertido) de modo a gerar

uma onda quadrada.

Outra característica do modo *fast PWM* está na atualização dos valores de OCRnA/OCRnB. Enquanto no modo normal e no modo CTC estes valores são atualizados imediatamente, no modo *fast PWM* a atualização dos valores ocorre apenas quando TCNTn atinge o valor máximo da contagem. Com isso, evita-se que uma comparação seja perdida caso o valor de OCRnA/OCRnB seja menor que o valor de TCNTn, o que pode ocorrer nos modos normal e CTC.

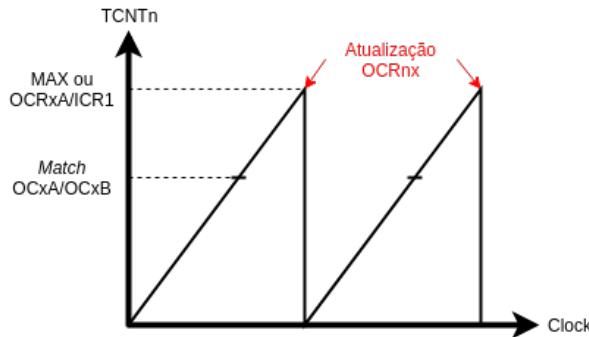


Figura 2.10: Diagrama de funcionamento do modo *fast PWM*

Fonte: Autor

2.1.6.4 Modo PWM com Correção de Fase

O modo PWM com correção de fase faz a contagem progressiva até o valor máximo (fixo ou variável, da mesma forma como ocorre no modo *fast PWM*), seguido de uma contagem regressiva até o valor mínimo (0), quando é disparada a condição de *overflow*. Esta característica, faz com que este modo atinja velocidades 2x menor que o modo *fast PWM*. A figura 2.11 apresenta o diagrama de funcionamento deste modo.

Neste modo de operação, os pinos OCxA/OCxB podem ser configurados para apresentar nível baixo em caso de *match* (TCNTn e OCRxA/OCRxB) na contagem progressiva, e nível alto em caso de *match* na contagem regressiva (ou o contrário), gerando uma onda quadrada. No caso do *Timer 1*, ainda existe a possibilidade de inverter o valor de OC1A a cada *match*, seja em contagem progressiva ou regressiva.

Assim como no modo *fast PWM*, a atualização do valor de OCRxA/OCRxB não é instantânea, ocorrendo apenas no momento em que a contagem atinge o valor máximo. Importante ressaltar que, tanto para o modo de correção de fase, quanto para o *fast PWM*, o valor de ICR1 é atualizado imediatamente, o que pode ocasionar em uma perda na comparação com TCNTx caso este registrador estiver sendo usado para definir o topo da contagem.

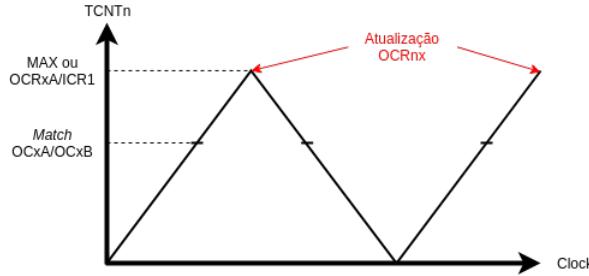


Figura 2.11: Diagrama de funcionamento do modo PWM com Correção de Fase

Fonte: Autor

2.1.6.5 Modo PWM com Correção de Fase e Frequência

Este modo de operação está disponível apenas para o *Timer 1*. Seu funcionamento é idêntico ao modo de correção de fase, a diferença está no momento da atualização dos registradores OCRxA/OCRxB, que não ocorre no topo da contagem mas sim ao atingir o valor mínimo, como mostra a figura 2.12. Esta mudança faz com que os pulsos gerados sejam sempre simétricos.

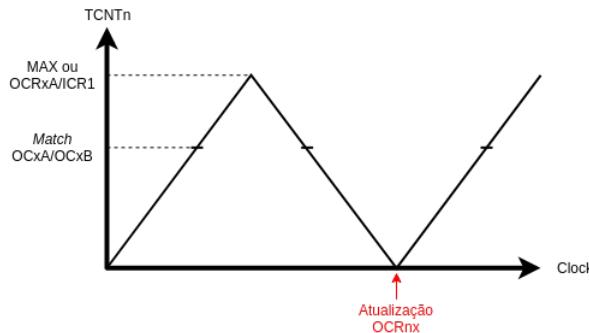


Figura 2.12: Diagrama de funcionamento do modo PWM com Correção de Fase e Frequência

Fonte: Autor

2.1.6.6 Captura de Eventos

O *Timer 1* apresenta uma funcionalidade extra que é a captura de eventos. Esta funcionalidade permite que o valor da contagem presente em TCNTx seja capturado e salvo no registrador ICR1.

O disparo do evento de captura pode ser dado pela saída do comparador analógico ou pelo pino ICP1, que pode ser configurado para disparo por borda de subida ou descida.

Se o registrador ICR1 não estiver sendo utilizado como valor máximo do contador, esta é a única forma de escrever neste registrador.

2.1.7 Conversor A/D

O ATmega328P possui um conversor A/D de aproximações sucessivas com resolução de 10-bits e 8 canais de entrada (ADC0-ADC7) multiplexados, além de duas entradas fixas (0V e 1,1V) e um sensor de temperatura integrado. A figura 2.14 apresenta a organização interna do conversor A/D.

O conversor possui uma entrada de alimentação separada que é feita por meio do pino AVcc. Esta entrada pode ser utilizada como tensão de referência (V_{ref}) para conversão, ou ainda, podem ser escolhidas outras opções por meio do registrador ADMUX, tais como o pino AREF ou referência interna de 1,1V.

Por ser um conversor de 10-bits, são necessários 2 registradores para armazenar o resultado da conversão, são eles o ADCL e o ADCH. O resultado da conversão é alinhado à esquerda por padrão, no entanto esta opção pode ser alterada para alinhamento à direita por meio do registrador ADMUX. A figura 2.13 mostra como o resultado é armazenado nos registradores ADCL e ADCH para cada modo.

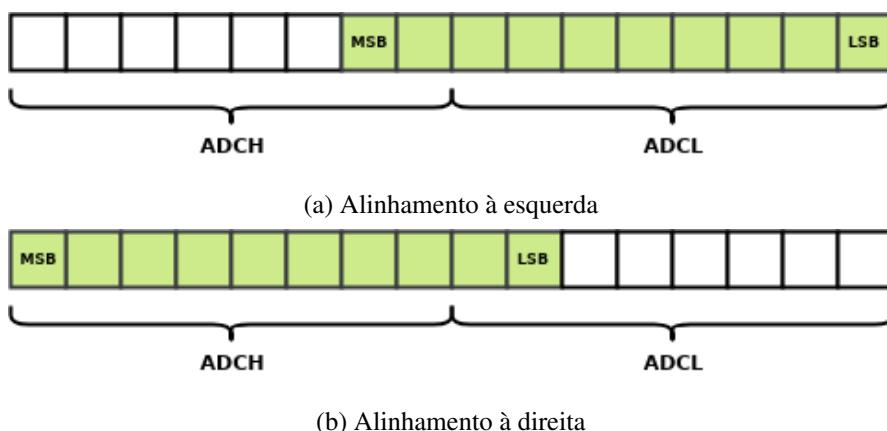


Figura 2.13: Alinhamento do resultado nos registradores ADCH e ADCL

Fonte: Autor.

Por segurança, uma leitura no registrador ADCL bloqueia a permissão de escrita nos registradores ADCL e ADCH, garantindo assim que o dado lido é referente à mesma conversão (o acesso é liberado novamente ao realizar uma leitura em ADCH). Caso o resultado esteja alinhado à direita, pode-se obter um valor convertido de 8-bits apenas lendo o registrador ADCH.

Para iniciar uma conversão é necessário colocar em nível alto os bits ADEN (habilita o conversor) e ADSC. A conversão é então inicializada e ADSC permanece em nível alto durante todo o tempo de conversão. Ao final, o bit ADSC é resetado por *hardware* e a flag ADIF é setada, podendo gerar uma interrupção se esta estiver configurada. O resultado de uma conversão é dada pela equação 2.1.

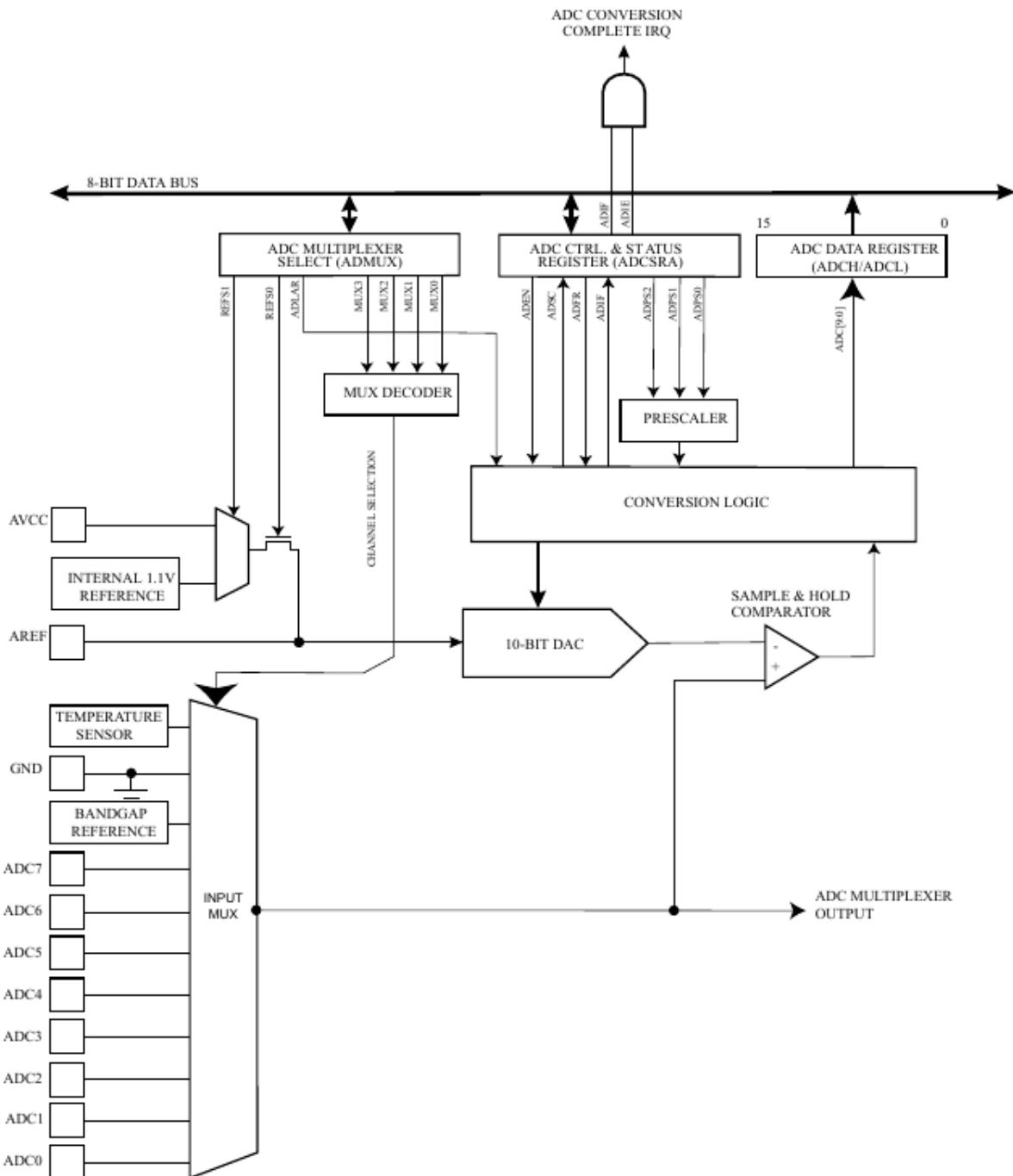


Figura 2.14: Organização do conversor A/D

Fonte: Folha de dados ATmega328P

$$ADC = \frac{V_{in} \cdot 1024}{V_{ref}} \quad (2.1)$$

Onde V_{in} é a tensão aplicada à entrada do microcontrolador e ADC é o valor (decimal) convertido. Valores de entrada superiores à V_{ref} terão valores convertidos próximos à 0x3FF (valor máximo para 10-bits).

É possível configurar um evento para o disparo do conversor A/D ou utilizá-lo no modo *Free Run*, em que o disparo é feito pela própria *flag* do conversor (ADIF), fazendo com que uma nova conversão comece imediatamente após a outra. Para utilizar este modo, a *flag* ADIF precisa ser limpa a cada conversão, o que é feito automaticamente se for utilizada interrupção. A tabela 2.3 apresenta todas as possibilidades de disparo do conversor A/D.

Tabela 2.3: Modos de disparo do conversor A/D

ADTS[2:0]	Disparo
000	Modo <i>Free Run</i>
001	Saída do comparador analógico
010	Interrupção externa 0
011	<i>Match A Timer 0</i>
100	<i>Overflow Timer 0</i>
101	<i>Match B Timer 1</i>
110	<i>Overflow Timer 1</i>
111	Captura de evento <i>Timer 1</i>

Um recurso extra oferecido pelo microcontrolador é o sensor de temperatura integrado. Este sensor é capaz de realizar medições entre -45°C e 85°C, com precisão de $\pm 10^{\circ}\text{C}$.

Para utilizar este sensor, é preciso configurar o V_{ref} para a entrada interna de 1,1V. A temperatura (em °C) é dada pela equação 2.2.

$$T = \frac{[ADCH << 8 | ADCL] - T_{OS}}{k} \quad (2.2)$$

Onde T_{OS} é um valor inserido na EEPROM (*Electrically-Erasable Programmable Read-Only Memory*) de cada componente como parte dos testes em produção, ADCH e ADCL são os valores contidos nos respectivos registradores e k é um valor a ser determinado na calibração. (Na prática, utiliza-se $T_{OS} = 324,31$ e $k = 1,22$ [18])

2.1.8 USART

A USART é um dos módulos do Atmega328P que permitem a comunicação serial deste com outros dispositivos. Este módulo possui múltiplos modos de operação, suportando comunicação *full duplex*, operação síncrona e assíncrona, detecção de erros em *frames*, modo de comunicação com multiprocessadores, etc., além de poder ser utilizada para comunicação SPI (como Mestre). A figura 2.15 apresenta a organização interna deste módulo no microcontrolador.

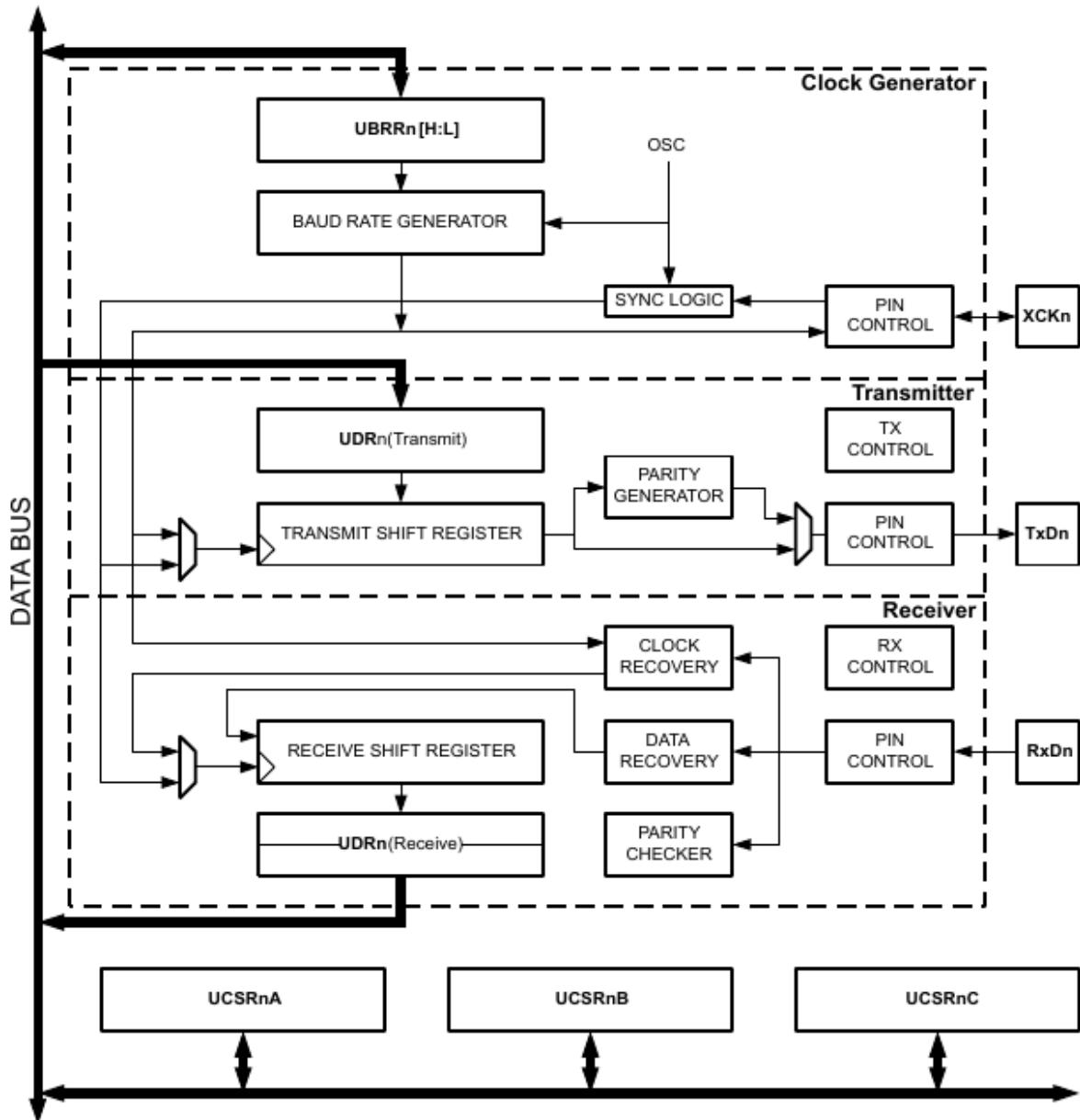


Figura 2.15: Organização da USART

Fonte: Folha de dados ATmega328P

O protocolo de comunicação da USART utiliza *frames* que podem ter 5,6,7,8 ou 9 bits, com 1 ou 2 bits de parada, além da possibilidade de adição de bits de paridade par ou ímpar. Os bits de paridade

são utilizados para detecção de erro na transmissão e são calculados pelas equações 2.3 (paridade par) e 2.4 (paridade ímpar).

$$P_{even} = d_{n-1} \oplus \dots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0 \quad (2.3)$$

$$P_{odd} = d_{n-1} \oplus \dots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1 \quad (2.4)$$

Onde d_n é o n-ésimo bit de dado a ser transmitido.

A transmissão de um *frame* se inicia com o bit de *START*, fazendo a mudança do estado da linha alto (*IDLE*) para o nível baixo, indicando que uma comunicação deve ser iniciada. Os bits do *frame* são então transmitidos um a um, começando pelo bit menos significativo e terminando com o bit de paridade (se houver). Por fim, são enviados os bits de parada (1 ou 2, conforme configurado), indicando o fim do *frame*. Este procedimento está ilustrado na figura 2.16.

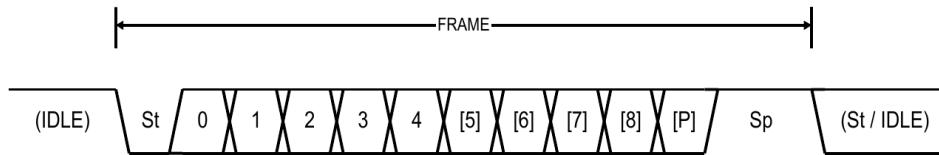


Figura 2.16: Formato de um *frame* transmitido pela USART

Fonte: Folha de dados ATmega328P

Durante a transmissão e a recepção, a USART utiliza um registrador auxiliar UDRn (como mostrado na figura 2.15). Este registrador é na verdade composto por dois espaços de memória, um acessado por operações de escrita e outro por operações de leitura, e consiste em um *buffer* que recebe os dados a serem transmitidos (antes de serem enviados ao registrador de deslocamento para transmissão) ou os dados recebidos (quando a recepção for concluída).

Para utilizar a USART, o programador precisa inicialmente informar a velocidade de comunicação (*BAUD rate*), formato do *frame* e habilitar o transmissor e o receptor. A transmissão se inicia ao carregar um dado no registrador UDRn e a recepção ao receber o bit de *START*. A velocidade de comunicação é ajustada por meio do registrador UBRRn de 16-bits e pode ser calculada pela equação 2.5 (para o modo assíncrono).

$$BAUD = \frac{f_{osc}}{16(UBRRn + 1)} \quad (2.5)$$

Onde BAUD é a velocidade de comunicação (bits/s), f_{osc} é a frequência de *clock* do sistema e UBRRn é o valor contido no registrador.

A USART pode ser utilizada de modo dirigido por interrupção. Três eventos podem ser configuradas gerar interrupção, são eles:

- **Recepção completa:** Indica que existe dado não lido no *buffer* de recepção.
- **Registrador de dados vazio:** Indica que o *buffer* de transmissão está pronto para receber um novo dado, ou seja, o dado anterior já foi movido para o registrador de deslocamento. Esta é uma interrupção disparada por condição, persistindo até que um novo dado seja escrito no registrador UDRn ou a interrupção seja desabilitada manualmente.
- **Transmissão completa:** Indica que todos os dados presentes no registrador de deslocamento já foram transmitidos.

Assim como ocorre com os temporizadores, quando em operação, a USART sobrescreve o funcionamento normal dos pinos PD0 (Rx) e PD1 (Tx).

2.2 Processo de Software

Para a criação de *software* de modo profissional, é fundamental o uso de técnicas que propiciem estabilidade, controle e organização do processo de desenvolvimento [19]. Diversos são os processos de *software* existentes, cada um com suas características próprias, entre eles, o processo ágil que surge para sanar fraquezas da engenharia de *software* convencional, priorizando a entrega mais do que a análise de projeto [19].

O processo ágil mais utilizado no mundo é o *SCRUM* [20]. Desenvolvido em 1993 por Jeff Sutherland, o *SCRUM* baseia seu desenvolvimento em ciclos curtos (chamados *Sprints*), onde, ao final de cada ciclo deve ser desenvolvido um módulo funcional do sistema (e este deve estar finalizado). O desenvolvimento em ciclos curtos tem a grande vantagem de gerar resultados constantes, revelando erros rapidamente e permitindo assim que estes sejam corrigidos rapidamente. O fluxo do processo *SCRUM* é mostrado na figura 2.17.

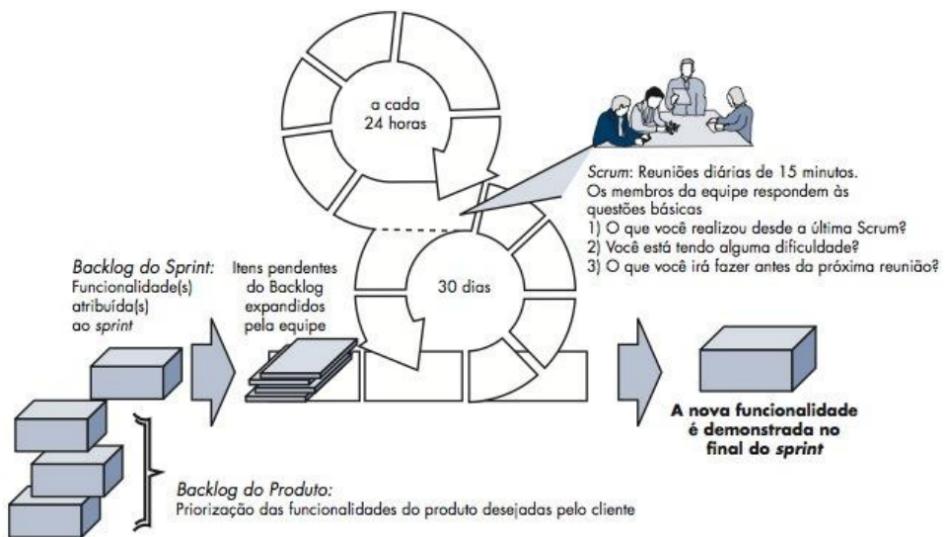


Figura 2.17: Fluxo do processo *SCRUM*. A figura mostra o fluxo completo considerando *sprints* mensais e reuniões diárias. Neste projeto, os *sprints* foram semanais e as etapas em grupo foram omitidas.

Fonte: Pressman [19]

Como aponta Jeff Sutherland [21], a realidade do desenvolvimento de *software* é diferente do que propõe alguns processos convencionais, como o método cascata, onde o planejamento é inteiramente e minunciosamente detalhado para o desenvolvimento posterior. Alteração nos requisitos é algo comum em projetos de *software* e abordagens convencionais não preveem esta possibilidade, ao passo que os métodos ágeis procuram incorporar este dinamismo do desenvolvimento real. Este foi o motivo pelo qual o *SCRUM* foi escolhido para o desenvolvimento deste trabalho.

2.3 Teste de *Software*

O desenvolvimento de *software* é uma tarefa complexa e diversos fatores podem contribuir para que a execução de um programa resulte em um comportamento inesperado, sendo que a maioria dos defeitos em *software* são de origem humana, ou seja, um erro cometido pelo programador durante o desenvolvimento [22]. Diferentes tipos de erros podem ser cometidos durante o desenvolvimento e por isso a atividade de teste é dividida em fases, cada qual com um objetivo distinto. Neste projeto, foram realizados apenas testes de unidade, ou seja, testes que focam nas menores unidades de um programa [22], neste caso, os métodos em Java.

Diversas são as técnicas e os critérios que podem ser utilizados para testar um *software*. Este trabalho focou no uso de duas técnicas: teste funcional e teste estrutural.

O teste funcional é aquele onde a atividade de teste é voltada para as especificações, ou seja, o funcionamento desejado, sem se preocupar na maneira como o programa foi escrito. O critério adotado para a realização deste teste foi a Análise do Valor Limite. Neste tipo teste, o domínio de entrada (ou seja, o conjunto de todos os possíveis valores que podem ser utilizados como entrada do programa [22]) é dividido em classes, de forma que se espera um mesmo comportamento do programa para todos os elementos de uma mesma classe (por isso, estas são chamadas classes de equivalência) [22]. O critério se baseia em testar valores que estão nos limites de cada classe de equivalência, ou seja, na transição dos valores de entrada onde o comportamento do programa deve mudar.

O teste estrutural, por outro lado, observa a estrutura interna do código para a geração dos casos de teste (par formado por dado de entrada e saída desejada [22]). No caso deste projeto, o critério utilizado foi o baseado em fluxo de controle, mais especificamente, o critério de Todos-Nós, onde busca-se a criação de casos de teste que exerce, pelo menos uma vez, cada comando do programa [22].

Todos os principais módulos foram testados com técnicas funcionais e estruturais, sendo que o teste estrutural foi o principal (exceto no módulo de CPU).

Capítulo 3

Desenvolvimento do Projeto

Neste capítulo serão detalhadas as etapas de desenvolvimento do projeto, bem como as ferramentas utilizadas.

3.1 Material

Para a execução do projeto foram necessárias diversas ferramentas para projetar, desenvolver e testar o sistema. A seguir são listados todos os materiais utilizados ao longo do projeto.

- Para o desenho de diagramas de classe e mapas mentais na fase de projeto, foram utilizadas as ferramentas *Dia*¹ e *Draw.io*², por serem ambas gratuitas, *Open-Source* e com recursos que suprem as necessidades deste projeto.
- Como já mencionado anteriormente, o método ágil *SCRUM* foi o escolhido para a construção do sistema. Apesar de não ter sido aplicado por completo, com reuniões semanais e outras atividades em grupo (já que é o trabalho foi desenvolvido por uma única pessoa), o desenvolvimento em ciclos, a priorização de tarefas e outras características do *SCRUM* foram aplicadas para ajudar na organização e no desenvolvimento do simulador. A plataforma *Taiga*³ foi utilizada para organização e planejamento dos *Sprints* e sua escolha se deve pelo fato de ser gratuito e já ser conhecida pelo autor.
- Para controle de versão utilizou-se o *Git*⁴ sincronizado à um repositório *on-line* no *Github*⁵. Nenhum motivo especial na escolha destas ferramentas, a não ser a familiaridade do autor com

¹<http://dia-installer.de/>

²<https://www.draw.io/>

³<https://taiga.io/>

⁴<https://git-scm.com/>

⁵<https://github.com/>

seu uso.

- Para documentação do projeto, foi escolhido o *Gitbook*⁶ pela possibilidade de hospedar a documentação *on-line*, além de contar com um *design* próprio que torna a documentação mais agradável para leitura e mais organizada.
- Para o realizar modificações no código da IDE do Arduino, foi utilizado o *InteliJ IDEA*⁷ para escrever o código e o *Apache Ant*⁸ para a compilação (uma vez que esta ferramenta já é utilizada pelo projeto Arduino).
- O *Inkscape*⁹ foi usado para a alteração no *design* da IDE (inserção do botão "Android"). Nenhum motivo especial suporta esta escolha, apenas o fato de estar disponível facilmente nos repositórios do Arch Linux, sistema operacional utilizado durante o desenvolvimento.
- Para o desenvolvimento *mobile*, o *Android Studio*¹⁰ foi a escolha para este projeto. A opção de desenvolver o código nativamente veio principalmente da experiência do autor com esta plataforma, mas também por questões de desempenho. Além disso, o *Android Studio* conta com diversas ferramentas que foram utilizadas na fase de teste, como o *JUnit*¹¹ e o *PowerMock*¹², utilizados para criação de teste de unidade, além de ferramentas de *profiling*.
- O *SonarQube*¹³ foi escolhido para fazer a análise estática do código do simulador. A escolha desta ferramenta se deve pelo fato de ser possível extrair diversas métricas de *software* em uma única aplicação, além de sugestões de correção de *bugs*, vulnerabilidades, entre outros. Além disso, o *plugin JaCoCo*¹⁴ foi utilizado em conjunto ao *SonarQube* para obter medidas de cobertura de código.
- Para montar códigos *Assembly* escritos para o ATmega328P, foi utilizado o *AVRA*¹⁵. Também não há motivos especiais para o uso desta ferramenta a não ser por sua facilidade de instalação e uso.

⁶www.gitbook.com

⁷<https://www.jetbrains.com/idea/>

⁸<https://ant.apache.org/>

⁹<https://inkscape.org/pt-br/>

¹⁰<https://developer.android.com/studio/>

¹¹<https://junit.org/junit4/>

¹²<https://github.com/powermock/powermock>

¹³<https://www.sonarqube.org/>

¹⁴<https://www.eclemma.org/jacoco/>

¹⁵<http://avra.sourceforge.net/>

- Para a criação dos gráficos, foram criados *scripts* em *Python*, fazendo uso da biblioteca *Matplotlib*¹⁶, que possui recursos avançados para geração de gráficos e é de fácil uso. Para gráficos simples, foi utilizado o *Google Sheets*¹⁷.
- Em termos de hardware, foi utilizado um Arduino UNO R3 para comparar os resultados obtidos pelo aplicativo com o sistema real, principalmente comparações feitas em medidas de frequência e *duty cycle*, no qual se fez uso também de um osciloscópio *InfiniiVision DSOX2002A*, da Keysight.
- Por fim, o aparelho disponível para realizar os testes foi o *smartphone ASUS ZenFone 2*, com 4 núcleos de processamento (2,33GHz), 4GB de memória principal e 32GB de armazenamento, executando o Android 5.0.

¹⁶<https://matplotlib.org/>

¹⁷<https://www.google.com/sheets/about/>

3.2 Método

3.2.1 Desenvolvimento na IDE do Arduino

A primeira parte do desenvolvimento ocorreu na IDE do Arduino. Esta etapa focou na criação do botão "Android", cuja função é compilar o código e transferi-lo para o aparelho Android conectado ao computador, da mesma forma que o botão "*Upload*" faz com a placa de Arduino.

A figura 3.1 apresenta o diagrama de classes do código desenvolvido nesta etapa. Para fazer as modificações necessárias, foi necessário inserir o pacote *android_usb* na pasta app/src/processing/app/ do projeto Arduino, além de editar as classes *EditorToolbar* e *Editor* da mesma pasta para inserir o botão e definir sua funcionalidade, respectivamente (o arquivo contendo a imagem do botão está localizado em build/shared/lib/theme/ e build/linux/work/lib/theme/)

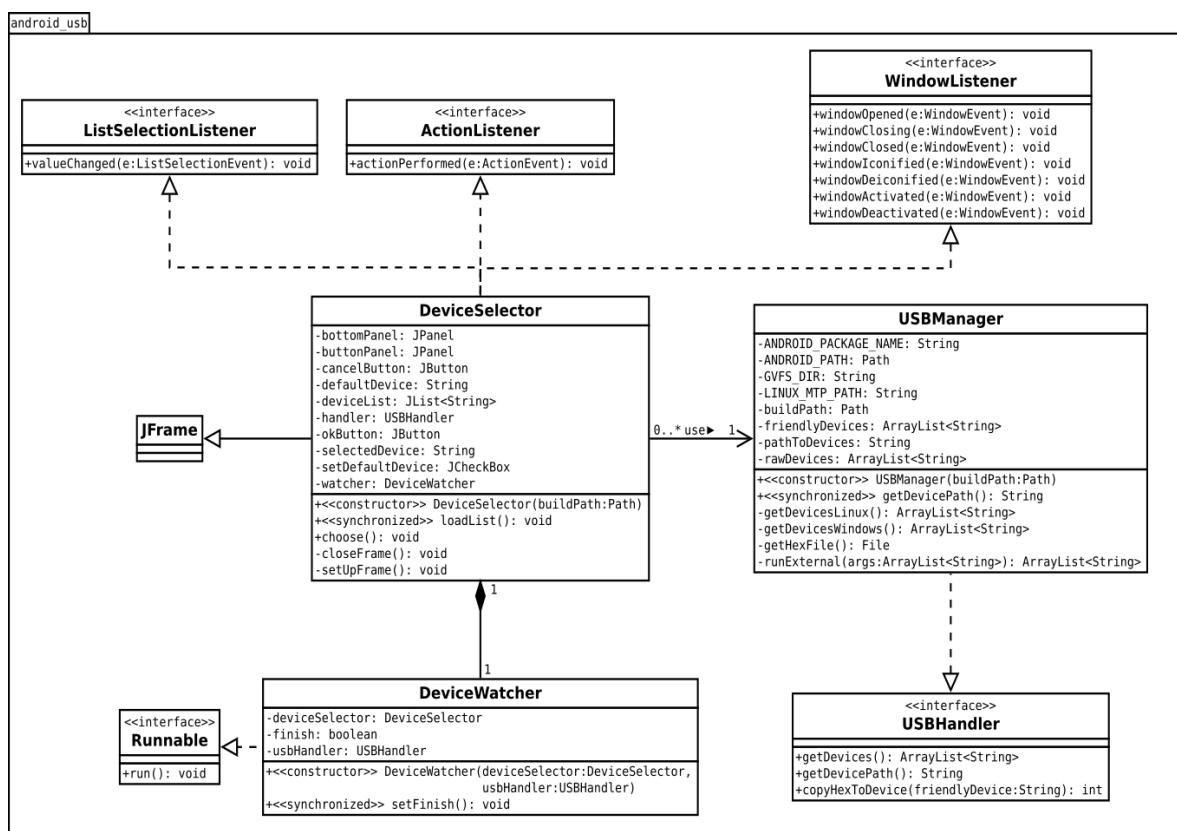


Figura 3.1: Diagrama de classes das modificações realizadas na IDE do Arduino

Fonte: Autor

A classe *DeviceSelector* é a entrada do sistema. Ela recebe como argumento o local onde o hexadecimal é gerado (que pode ser obtido por meio do método *getBuildPath* da classe *Sketch* já existente no projeto do Arduino) e cria a janela para selecionar o dispositivo. Todos os aparelhos Android monitorados no sistema são exibidos em uma lista em caso de sucesso na compilação. Em conjunto, atua a

classe *DeviceWatcher*, que é uma *thread* que fica a procura de novos dispositivos. Desta forma, caso algum aparelho seja conectado após a exibição da janela de seleção, a lista de dispositivos é atualizada automaticamente.

Já a classe *USBManager* cuida de toda a comunicação com o sistema e o dispositivo Android, além de fazer a identificação dos dispositivos conectados e retornar ao seletor um *friendly name* para que o usuário possa reconhecer seu dispositivo facilmente. Ao pressionar o botão "Ok" do seletor de dispositivos, o método *copyHexToDevice* é chamado para copiar o arquivo compilado para o dispositivo Android.

No Linux, todos os dispositivos MTP (*Media Transfer Protocol*) são montados por meio do GVfs (*GNOME Virtual file system*). Utilizando a variável de ambiente *\$XDG_RUNTIME_DIR* é possível acessar o dispositivo como uma pasta no sistema de arquivos. Desta forma, o arquivo hexadecimal é copiado para o local *XDG_RUNTIME_DIR/gvfs/<Dispositivo>/DCIM/SOFIA* e tem o nome genérico de *code.hex*, tornando o código disponível para o simulador. No Windows, dispositivos MTP são tratados de forma diferente, não sendo montados diretamente no sistema de arquivos, tornando o acesso bastante trabalhoso. Por este motivo, nesta primeira versão do sistema, não foi implementado um método automático para copiar o arquivo hexadecimal para o dispositivo Android no Windows.

A classe *USBManager* utiliza alguns comandos externos do sistema, tais como *lsusb*, *gio copy* e *gvfs-copy*. O primeiro comando é necessário apenas para retornar um nome mais legível do dispositivo para o usuário (*friendly name*), mas caso este não seja encontrado o sistema ainda será capaz de funcionar. Já o segundo e o terceiro comando são utilizados para transferir o arquivo para o dispositivo móvel, sendo executados na ordem: primeiro o *gio copy*, seguido do *gvfs-copy* em caso de falha (*gvfs-copy* é um comando antigo, mas foi adicionado para compatibilidade). Se o usuário não tiver pelo menos um destes comandos disponíveis no computador, o sistema informará falha ao copiar o arquivo. Estes três comandos fazem parte do pacote básico de instalação das principais distribuições Linux.

3.2.2 Desenvolvimento Android

A segunda parte do desenvolvimento foi a criação do aplicativo para fazer a simulação do código. A figura 3.2 apresenta um diagrama simplificado da arquitetura do simulador.

3.2.2.1 Módulo principal

Tudo tem início na classe *UCModule*, que constitui o módulo principal do sistema. Esta classe é responsável por inicializar todos os demais módulos e fazer a sincronização entre eles, além de fornecer serviços para estes módulos no que diz respeito às características do sistema simulado, como a quantidade de pinos, tensão de alimentação, etc.

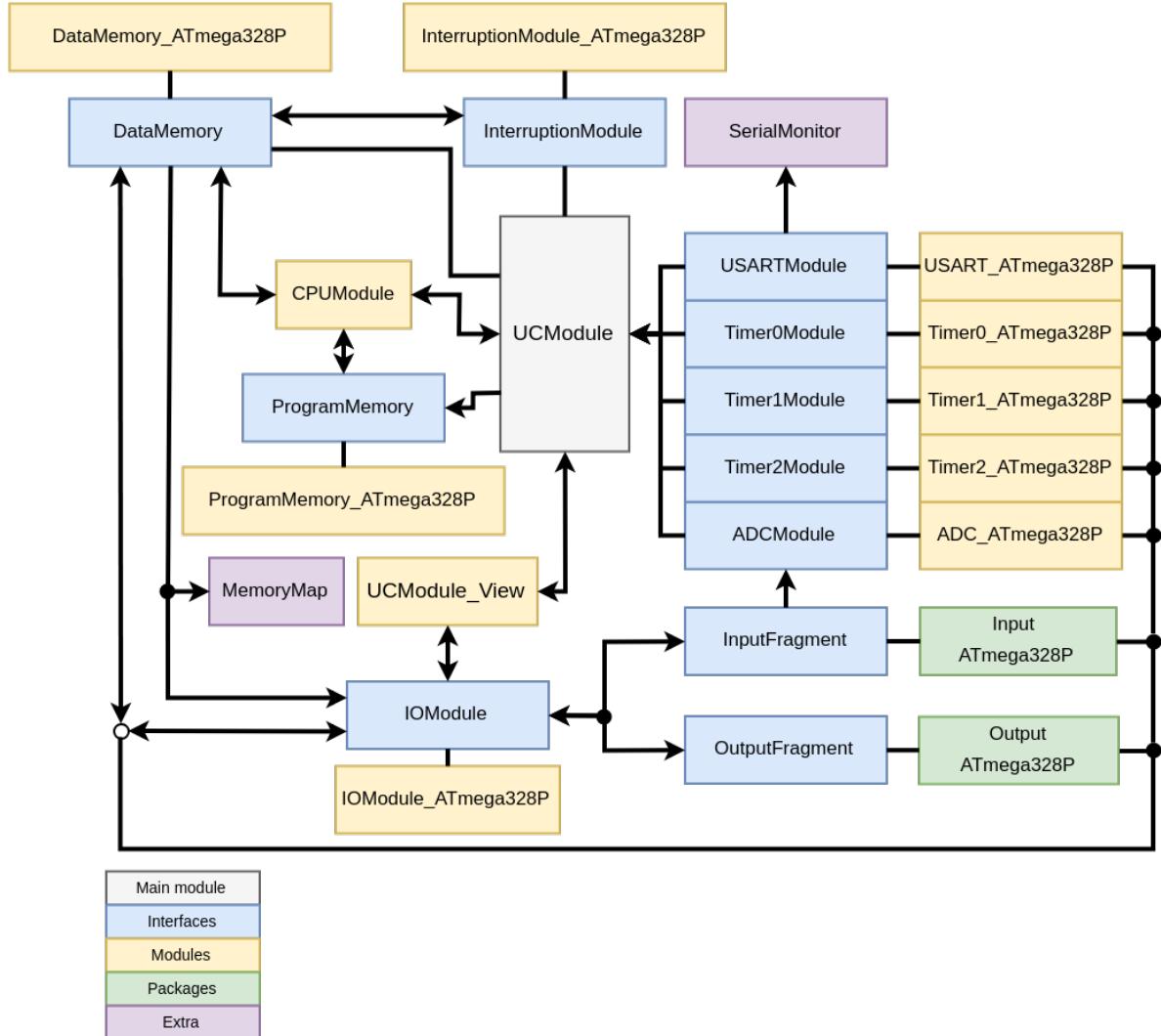


Figura 3.2: Arquitetura do simulador

Fonte: Autor

Esta classe possui uma extensão que é a classe *UCModule_View*. Ela é responsável por toda a manipulação das telas e recursos visuais do aplicativo, além de fornecer *feedback* à *UCModule* quanto às ações dos botões (como botão *Reset*) e fazer a contagem do tempo simulado. É na *UCModule_View* que os módulos de entrada e saída são inicializados.

Outra tarefa da *UCModule* é atuar como escalonador. Todos os módulos do sistema (CPU, conversor A/D, temporizadores, USART) além do módulo de visualização (*UCModule_View*) são executados um a um, em uma fila circular, sendo calculado um ciclo de *clock* a cada iteração. A velocidade de execução deste *loop* não foi limitada de nenhuma forma, para que a velocidade de simulação seja a maior possível. Essa abordagem, no entanto, faz com que o simulador sofra variações de velocidade em função da carga no sistema Android.

Ainda quanto à sincronia dos módulos, buscou-se preservar o número correto de ciclos de *clock*

que cada instrução da CPU leva para executar, bem como chamadas para rotinas de interrupção e *prescalers* dos temporizadores. A exceção fica apenas para o conversor A/D, cujo tempo de conversão é de apenas 1 ciclo de *clock* (originalmente, o Arduino leva cerca de 13 ciclos) e os *prescalers* foram desabilitados para este módulo. Essa mudança se justifica por não afetar de maneira significativa o realismo da simulação, ao mesmo tempo que proporciona um ganho de performance, tornando o sistema mais agradável ao usuário.

3.2.2.2 CPU

O próximo módulo, a CPU (*CPUModule*), é responsável pela execução das instruções contidas na memória de programa. Ao final de cada instrução, é feita a verificação por interrupções, que são executadas em ordem de prioridade conforme apresentado na tabela 2.1. Todas as instruções do microcontrolador ATmega328P foram implementadas, com exceção das instruções *BREAK*, *SLEEP* e *WDR*.

A arquitetura trabalhada não fornece instruções com campos fixos para *opcode*, operadores, etc., o que dificulta a decodificação das instruções. Para que a legibilidade do código não fosse prejudicada com uma série de condicionais aninhadas, a estratégia adotada para a decodificação foi a utilização de um banco de dados com as instruções pré-decodificadas. Desta forma, foi criado um banco de dados com 2^{16} posições (tamanho da instrução) e para cada posição foi inserido um identificador (ID) da instrução referente àquela posição em binário. Com isso, ao ler uma instrução da memória de programa, a CPU simplesmente acessa um vetor na posição da instrução lida e recupera a instrução a ser executada (o banco de dados é carregado para a memória durante a exibição do *Splash Screen* ao iniciar o aplicativo).

Apesar do gasto de memória (o tamanho deste banco de dados em memória é de 128kB), esta solução torna o código muito mais legível.

3.2.2.3 Memória de Programa

A memória de programa (*ProgramMemory*) é responsável por armazenar o código a ser executado pela CPU. O código é lido de um arquivo hexadecimal no formato *Intel HEX* e formatado em um *array* de bytes. É também função da memória de programa verificar continuamente o arquivo hexadecimal e enviar uma mensagem de *Reset* para o módulo principal em caso de alterações.

Como dito na seção 2.1.3, a memória de programa do ATmega328P é organizada no formato 16kB x 16-bits, no entanto, no simulador, preferiu-se a organização 32kB x 8-bits. Desta forma, a estrutura de dados para armazenar o programa é mais simples (um vetor de bytes), além de facilitar a implementação da instrução "*LPM*", que realiza a leitura de um byte da memória de programa.

Cada vez que uma instrução é lida da memória pela CPU, dois bytes são lidos e concatenados. O valor do PC continua se referindo ao valor da próxima instrução enquanto que o endereço dos bytes desta instrução são calculados no método *loadInstruction*.

3.2.2.4 Memória de Dados

A memória de dados (*DataMemory*) é o módulo responsável por armazenar todas as informações dos registradores e da memória SDRAM externa. Além disso, a memória de dados notifica a classe *IOModule* em caso de alteração nos registradores de E/S (PINx, PORTx e DDRx).

Alguns registradores são tratados de maneira diferentes no ATmega328P. Pode-se citar, por exemplo, o registrador PINx. Este, como explicado na seção 2.1.5, apesar de ser um registrador de leitura, permite também a escrita de um valor, porém esta escrita não altera o valor do PINx, mas sim, o valor do PORTx. Estes casos foram todos tratados na memória de dados, nos métodos de escrita *writeByte* e *writeBit*. Assim, ao escrever um valor na memória, o endereço é verificado e se um caso especial for detectado, a operação realizada será diferente de uma simples escrita. Também foram escritos métodos especiais para a manipulação dos registradores que são atualizados apenas por *hardware* (como o PINx), bem como para a manipulação das *flags* de interrupção.

A memória de dados também fornece informações para o mapa de memória quando este é aberto pelo usuário. Por realizar mais operações de leitura e principalmente devido a maior atualização da tela, a velocidade de simulação diminui quando este recurso está ativo, ficando mais lento quanto mais dinâmica for a porção visível na tela.

3.2.2.5 Módulo de Interrupção

Foi desenvolvido um módulo de interrupção (*InterruptModule*) cuja função é verificar e organizar todos os eventos de interrupção que podem ser gerados. Este módulo recebe requisições dos temporizadores, E/S, conversor A/D e USART, organizando-as em ordem de prioridade. No caso do módulo de E/S, é feita ainda a verificação se houve ou não uma interrupção, por meio de detectores de borda, detecção de nível baixo, etc.

Este módulo também é responsável por armazenar os endereços de desvio das interrupções e fornecer à CPU para a execução das rotinas de interrupção.

Todas as classes do projeto podem acessar o módulo de interrupção estaticamente, por meio da classe *UCModule*.

3.2.2.6 Módulo de E/S

Seguindo para o módulo de entrada e saída, este é dividido em duas partes, cada uma tratando exclusivamente entrada ou saída. A parte de tratamento de entrada é responsável pelo gerenciamento de cada elemento gráfico de entrada, bem como o tratamento de suas ações, enquanto que o pacote de saída faz o mesmo para os elementos gráficos de saída. A classe *IOModule* fica acima destas duas, fazendo a integração para que a classe *UCModule_View* possa exibir corretamente os elementos de interface com o usuário.

Não foi imposta nenhuma restrição quanto à ligação de entradas e saídas no simulador. Isso significa que o usuário pode conectar múltiplas entradas/saídas no mesmo pino, conectar uma entrada analógica em um pino digital (neste caso, serão adotados os valores de tensão da folha de dados para definir nível alto, baixo ou indefinido), conectar entradas digitais em pinos analógicos, etc. Um mecanismo de detecção de curto-circuito (entre entrada e saída, também entre entradas) atua toda vez que uma entrada ou saída é alterada, parando o sistema se alguma condição indevida for detectada.

Foram definidos 3 níveis lógicos no sistema: alto, baixo e alta impedância. O nível de alta impedância é visto apenas na saída. Na entrada, existe ainda um estado indefinido, que envia um valor aleatório para a entrada, ou seja, pode ser interpretado como nível alto ou baixo (exceto se o resistor de *pull-up* interno estiver habilitado).

3.2.2.7 Temporizadores

Foram implementados os 3 temporizadores (*TimerxModule*) presentes no ATmega328P. Eles fornecem todos os modos de funcionamento dos apresentados na seção 2.1.6, com exceção do funcionamento assíncrono para o *Timer 2*.

Como mostrado no capítulo 2, alguns registradores, tais como a pilha, registradores do *Timer 1*, etc., trabalham em pares (*LOW* e *HIGH*). Em especial, o *Timer 1* usa um registrador temporário para que a leitura/escrita nos registradores ocorra de maneira sincronizada, ou seja, ao ler um valor de contagem do registrador *LOW*, o registrador *HIGH* é salvo imediatamente para que a leitura seja referente ao mesmo instante de tempo. Da mesma forma, a escrita em um registrador *HIGH* é armazenada em um registrador temporário e só é realizada de fato quando ocorrer uma escrita no registrador *LOW*, fazendo a escrita simultânea das duas partes.

Este mecanismo de registrador temporário só foi utilizado para escritas da CPU. No módulo de *Timer 1*, a escrita nos registradores de 16-bits ocorre por meio de um método especial na memória de dados, que faz a escrita das duas partes simultaneamente.

3.2.2.8 Conversor A/D

O conversor A/D (*ADCModule*), assim como os Temporizadores, apresenta todos os módulos de funcionamento descritos na seção 2.1.7, com exceção do sensor de temperatura e do disparo pela saída do comparador analógico (já que este módulo não foi implementado).

Em termos de implementação, este é o módulo menos fiel ao sistema físico original, no entanto, isso não reflete em uma simulação incorreta.

3.2.2.9 USART

Por fim, a USART (*USARTModule*), apesar de possuir diversos modos de operação no sistema físico original, possui apenas um modo de operação no simulador: *frames* de 8 bits, sem paridade e um bit de parada. Esta configuração foi escolhida por ser a inicialização padrão da função *Serial.begin()* do Arduino (embora outras configurações também sejam possíveis).

Este módulo se comunica com um monitor serial que foi integrado ao sistema e ocupa o mesmo espaço reservado aos pinos de saída na tela. A comunicação entre simulador e monitor serial ocorre em baixo nível, ou seja, ao enviar uma informação do monitor para o simulador, o texto é transformado em bytes e enviados um a um, sendo recebidos nos registradores dedicados a esta função no ATmega328P. O envio de informação ao monitor é feita da mesma forma (byte a byte), sendo que o monitor terá a tarefa apenas de decodificar estes bytes e os imprimir em um formato legível.

Uma característica deste módulo é que a configuração de velocidade (*BAUD rate*) não importa para o seu funcionamento, ou seja, o comportamento do sistema será o mesmo para qualquer velocidade de comunicação.

Voltando na figura 3.2, pode-se observar que as comunicações ocorrem sempre por meio de interfaces (exceto para envio de dados, que precisam partir dos módulos). Este *design* isola toda a parte de controle do funcionamento dos módulos específicos do ATmega328P, o que facilita a expansão do sistema já que, para suportar uma nova plataforma, basta escrever novos módulos de *Timer*, conversor A/D, etc., que implementem as mesmas interfaces. As únicas classes que são acessados diretamente são a *UCModule_View* (que é própria do aplicativo) e a *CPUModule* (que é própria da arquitetura AVR).

Capítulo 4

Resultados e Discussões

Nesta seção, serão apresentados os resultados obtidos com as modificações da IDE do Arduino e com o simulador para Android, bem como algumas métricas de *software* obtidas para o simulador e algumas comparações feitas com aplicativos semelhantes ao desenvolvido neste trabalho.

4.1 Arduino IDE

A figura 4.1 mostra como ficou a IDE do Arduino após a introdução do botão "Android".

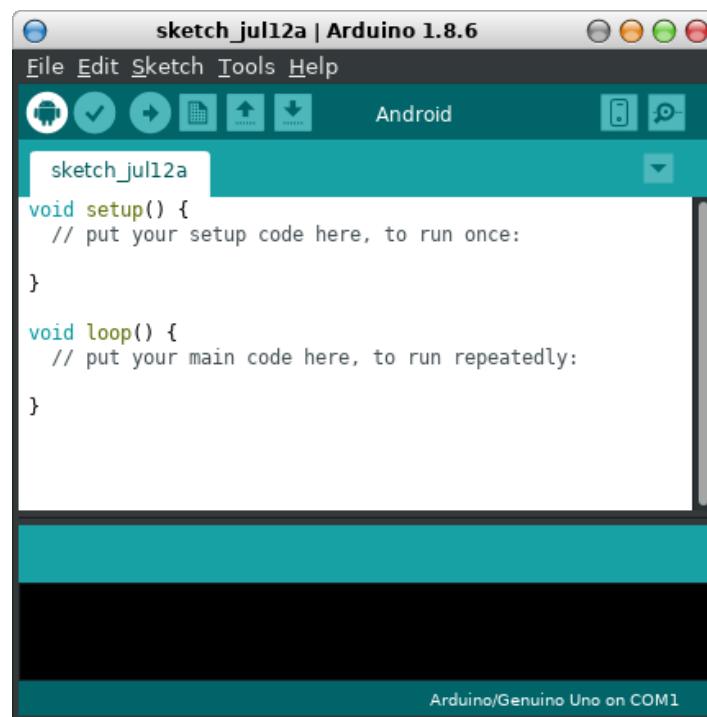


Figura 4.1: Localização do botão "Android"(selecionado) na IDE

Fonte: Autor

Ao pressionar o botão "Android", o processo de compilação se inicia e, em caso de sucesso, é exibida a janela para a seleção do dispositivo Android conectado ao PC, como mostrado na figura 4.2.

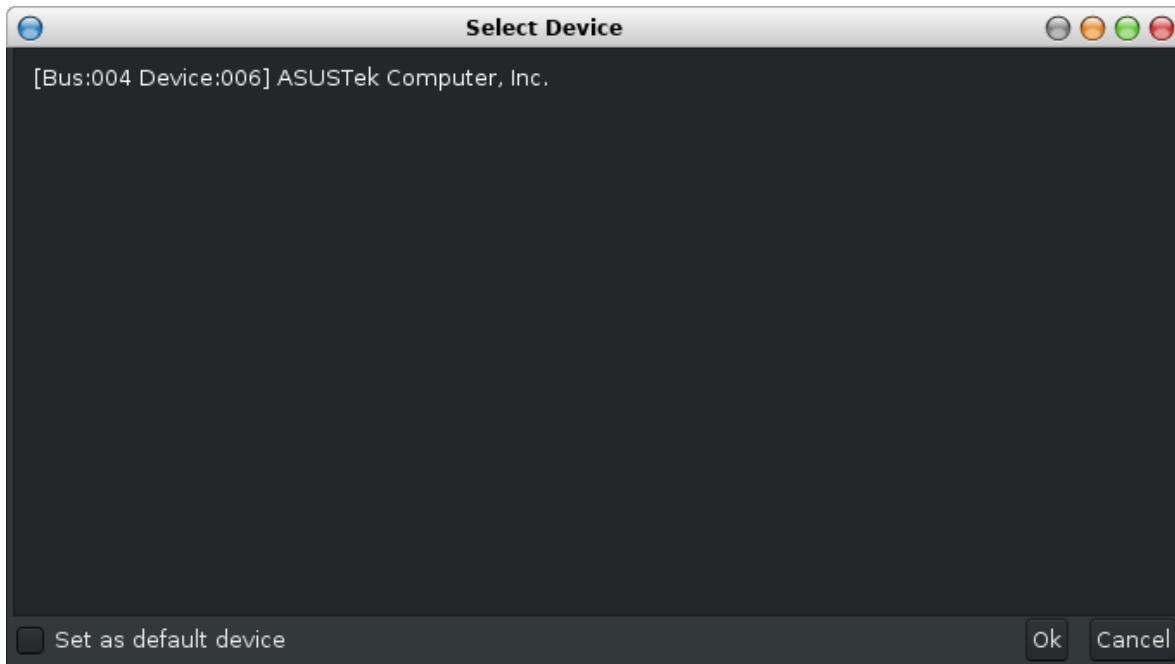


Figura 4.2: Seletor de dispositivos

Fonte: Autor

Ao selecionar o dispositivo, o usuário pode ativar a opção "*Set as default device*". Isso fará com que a opção escolhida seja salva e não exibirá o seletor de dispositivos nas próximas compilações.

Se tudo ocorreu como o esperado, o usuário deve ver a mensagem de cópia no *console* abaixo do editor, conforme mostra a figura 4.3.

```

Done compiling.
"/home/kollins/.arduino15/packages/arduino/tools/avr-gcc/4.9.2-atmel3.5.4-arduino
Sketch uses 734 bytes (2%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local
Copying...
Source: /tmp/arduino_build_332453/Blink.ino.hex
Target: [Bus:002 Device:003] ASUSTek Computer, Inc.
Done
Done
  
```

Figura 4.3: Cópia do arquivo realizada com sucesso para o *smartphone*

Fonte: Autor

4.2 Simulador

Ao abrir o simulador, o usuário é recebido com uma *Splash Screen*, mostrada na figura 4.4, enquanto o banco de dados é carregado para a memória principal, e posteriormente, o usuário é redirecionado para a tela inicial do simulador, mostrada na figura 4.5



Figura 4.4: *Splash Screen* exibida ao abrir o simulador

Fonte: Autor

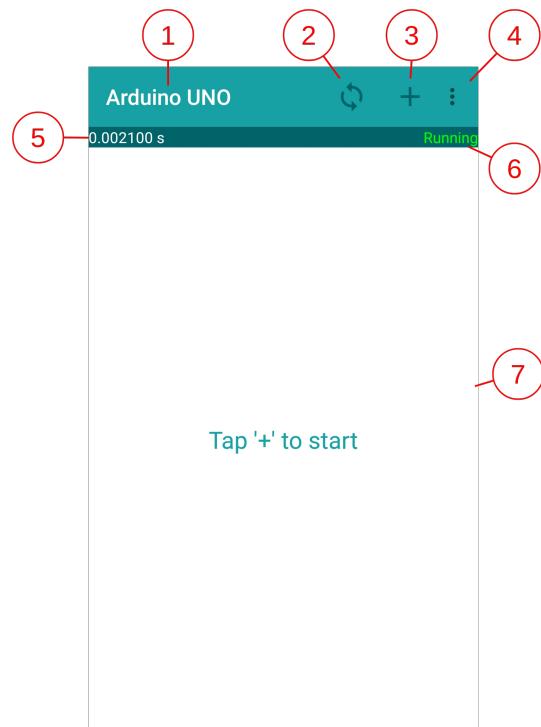


Figura 4.5: Tela inicial do simulador

Fonte: Autor

Na parte superior do simulador se concentram as opções para que o usuário comece a utilizar o sistema, bem como informações sobre o estado da simulação. Nesta *toolbar*, o usuário tem acesso às seguintes opções/informações:

1. **Modelo simulado:** Mostra qual placa de Arduino está sendo utilizada para a simulação.
2. **Botão Reset:** Permite o *reset* manual do sistema.
3. **Adicionar Entrada/Saída:** Permite adicionar uma entrada/saída digital ou uma entrada analógica, bem como um monitor serial.
4. **Outras opções:** Contém funções adicionais para importar código do *smartphone*, mapa de memória, configuração de tensão de referência (para o conversor A/D), remover todas as entradas/saídas, ajuda e acesso à informações do projeto.

Além disso, o usuário pode visualizar:

5. **Tempo simulado:** Exibe tempo simulado do sistema, baseado no cristal de 16MHz e atualizado a cada pulso de *clock* interno do sistema.
6. **Status:** Mostra o *status* da simulação.
7. **Área de Trabalho:** Onde serão dispostos os elementos de entrada e saída.

Caso não seja encontrado um arquivo hexadecimal ou ocorra alguma falha em sua abertura, o usuário deve ver uma mensagem na barra de *status* informando o problema, como mostrado na figura 4.6.

4.2.1 Interação com o sistema

O usuário interage com o sistema por meio de entradas e saídas no simulador. Uma saída digital é mostrada na figura 4.7. No lado esquerdo é possível selecionar o pino no qual a saída estará conectada e do lado direito é mostrado o estado do pino. Por padrão, o pino 13 é selecionado, uma vez que este é o pino onde o LED interno está conectado no Arduino UNO. A figura 4.7 mostra os três estados possíveis para uma saída no simulador.



Figura 4.6: Falha ao abrir arquivo hexadecimal
Fonte: Autor



Figura 4.7: Saídas digitais do simulador

Fonte: Autor

Uma entrada pode ser digital ou analógica. Uma entrada digital é mostrada na figura 4.8. Pode-se observar que existem 4 elementos em uma entrada digital, são eles (da esquerda para a direita):

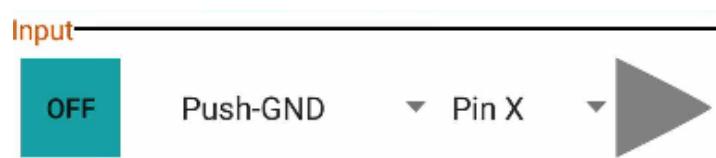


Figura 4.8: Entradas digitais do simulador

Fonte: Autor

- **Botão:** por onde o usuário envia sinais ao sistema.
- **Seletor de modo:** define a operação do botão, podendo ser:
 - *Push-GND*: envia nível baixo se pressionado, indefinido caso contrário (é a opção padrão).
 - *Push-VDD*: envia nível alto se pressionado, indefinido caso contrário.
 - *Pull-Up*: envia nível baixo se pressionado, alto caso contrário.
 - *Pull-Down*: envia nível alto se pressionado, baixo caso contrário.
 - *Toggle*: alterna seu nível a cada toque no botão.
- **Seletor de pino:** define para qual pino do Arduino o sinal deve ser enviado.
- **Saída do sinal:** mostra o que está sendo enviado para o pino selecionado

Por padrão, nenhum pino está selecionado. Isso garante que não haverá um curto-circuito ao adicionar uma entrada. Em caso de curto-circuito, é exibida uma mensagem na barra de *status* (como mostra a figura 4.9) e a simulação para, podendo ser reiniciada manualmente assim que a condição de curto-circuito for removida.

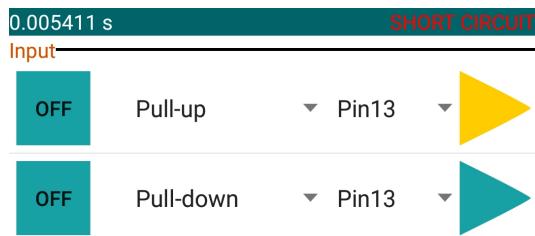


Figura 4.9: Condição de círculo-circuito entre entradas

Fonte: Autor

Uma entrada analógica é mostrada na figura 4.10. Ela possui um seletor de pino, uma barra deslizante e um voltímetro, indicando o valor de tensão enviado ao Arduino. Esta entrada permite ao usuário enviar valores de tensão entre 0V e 5V e assim como na entrada digital, nenhuma entrada está selecionada por padrão.

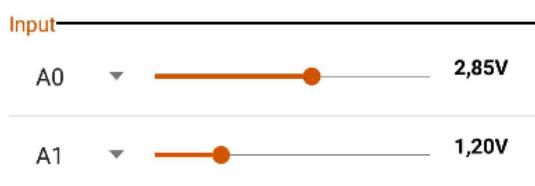


Figura 4.10: Entradas analógicas do simulador

Fonte: Autor

Caso se queira remover uma entrada/saída, pode-se utilizar um toque longo na célula desejada e selecionar quais elementos serão removidos, como mostra a figura 4.11. Alternativamente, pode-se utilizar a opção "*Clear I/O*" da toolbar para remover todas as entradas e saídas.

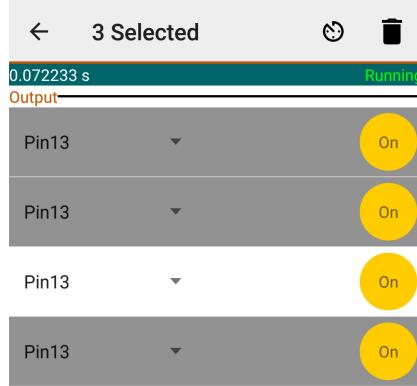


Figura 4.11: Remoção manual de pinos de saída

Fonte: Autor

4.2.2 Monitor Serial

Outro modo de interação com o sistema é por meio do monitor serial, mostrado na figura 4.12. Este monitor pode ser utilizado para receber ou enviar informações para a USART e ocupa o mesmo espaço reservado para as saídas digitais, ou seja, não é possível visualizar as saídas em conjunto com o monitor serial (é possível, no entanto, utilizá-lo simultaneamente com entradas).

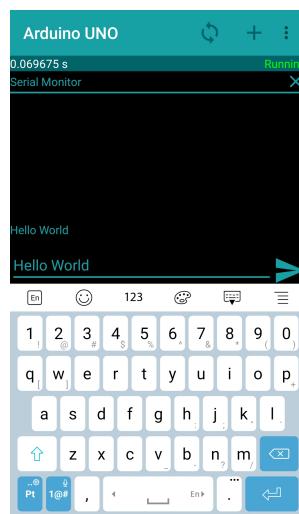


Figura 4.12: Monitor Serial

Fonte: Autor

O monitor serial pode ser removido também com a opção "*Clear I/O*" ou clicando no "x" na parte superior.

4.2.3 Mapa de memória

O recurso de mapa de memória é apresentado na figura 4.13. Este recurso permite ao usuário ver e acompanhar o estado de cada bit da memória de dados enquanto a simulação continua funcionando, bem como verificar qual o uso total de memória pelo sistema em bytes e em porcentagem. O estado dos bits não é atualizado automaticamente na tela, mas sim a cada 800ms, diminuindo o impacto na performance do simulador quando este recurso é aberto.

Figura 4.13: Mapa de memória

Fonte: Autor

Toda a memória (2kB), somado aos registradores, podem ser visualizados com esta função. Para facilitar a busca por um endereço específico, os registradores foram nomeados e adicionou-se o recurso de busca, que pode ser acessado pelo ícone de lupa no canto superior direito. A figura 4.14 mostra a busca sendo utilizada para encontrar o registrador TCNTx.

Figura 4.14: Recurso de busca do mapa de memória

Fonte: Autor

4.2.4 Referência externa de tensão

Outro recurso que foi inserido no simulador foi a configuração da referência externa de tensão para conversão analógica. Em *hardware*, esta referência é aplicada no pino AREF e utilizada como base para conversão. No simulador, pode-se utilizar o menu "AREF Config" para simular esta função. A figura 4.15 mostra a tela de configuração exibida.

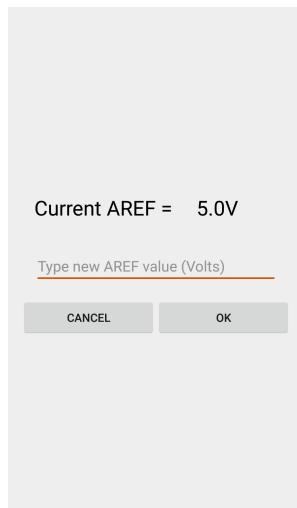


Figura 4.15: Configuração de tensão externa aplicada ao pino AREF, usada como base para conversão A/D

Fonte: Autor

Por padrão, o valor configurado é de 5V (como mostra a figura 4.15). Este valor pode ser alterado para qualquer valor real que esteja dentro das especificações mínimas e máximas apresentadas na folha de dados do ATmega328P (Mínimo: 1V, Máximo: 5V).

4.2.5 Medição de frequência

Caso a frequência de oscilação de um pino de saída seja muito alta, o simulador não será capaz de exibir esta intermitência corretamente devido a limitações de velocidade da atualização da tela do Android (máximo 60fps). Nestes casos, pode-se utilizar o recurso de frequencímetro integrado ao simulador para verificar a qual frequência a saída está alterando seu estado. (Na verdade, pode-se utilizar o frequencímetro em qualquer ocasião, este é apenas um caso onde seu uso se faz absolutamente necessário para depuração.)

Para medir frequência em um pino de saída, basta pressionar a saída desejada com um toque longo e selecionar frequencímetro (figura 4.16). Os valores de frequência e *duty cicle* aparecerão na célula de saída, como mostra a figura 4.17.

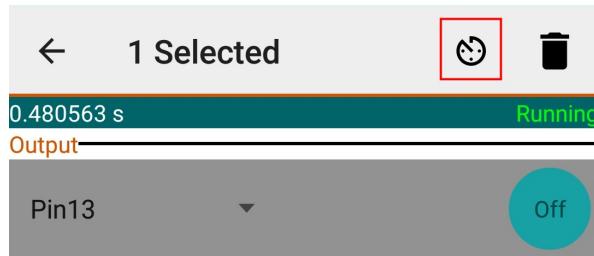


Figura 4.16: Ícone do frequencímetro

Fonte: Autor

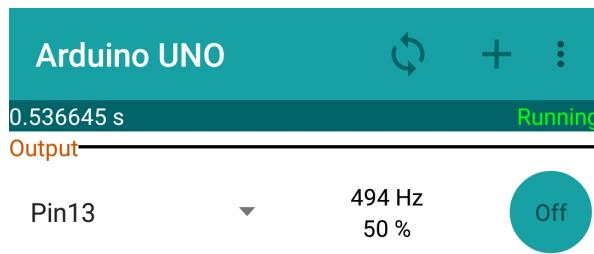


Figura 4.17: Frequencímetro em funcionamento

Fonte: Autor

A medição de frequência implementada se mostrou bastante precisa quando comparada com as medições reais de frequência. Para realizar os testes, foram utilizados os projetos *Blink* e *Timer1*, apresentados nos apêndices A e E respectivamente. O projeto *Blink* foi utilizado ora com a função *delay*, ora com função *delayMicroseconds*, desta forma podendo obter diferentes valores de frequência gerados por diferentes métodos, enquanto o projeto *Timer1* foi configurado com diferentes valores em TCNT1 para as diferentes configurações de frequência e *duty cycle*. As tabelas 4.1 e 4.2 apresentam os valores medidos de frequência obtidos com o simulador e com o osciloscópio ligado ao Arduino, bem como o erro relativo de cada medição comparado com o valor teórico esperado.

Como pode ser observado, o simulador e o Arduino se comportam de maneira semelhantes tanto na medição de frequência quanto nas relativas ao *duty cicle*, apresentando um erro relativo alto quando se trabalha com altas frequências.

Com isso, conclui-se que o frequencímetro incluso no simulador fornece medições satisfatórias, principalmente se a aplicação desenvolvida não tiver grandes exigências quanto à estes parâmetros. Notou-se apenas uma certa instabilidade nos valores quando as medições eram feitas em altas frequências, mas nada crítico ao ponto de não permitir o uso e a leitura das grandezas.

Tabela 4.1: Comparação das medições de frequência (*duty cycle*: 50%). Foram coletadas 110 amostras em cada frequência medida

Função	Período (Teórico)	Frequência (Hz)				Erro (%)	
		SOFIA		Arduino		SOFIA	Arduino
		Média	D. Padrão	Média	D. Padrão		
<i>delayMicroseconds</i>	2 μ s	107.062	37.946	133.254	4,93	78,59	73,35
	0,5 ms	1.967	12,04	1.968	11,97	1,63	1,58
	1 ms	989,16	0,98	988,36	0,03	1,08	1,16
<i>delay</i>	2 ms	495,71	0,55	495,44	9,47	0,86	0,91
	4 ms	248,96	0,19	248,64	0,21	0,41	0,55
	8 ms	125	0	124,59	0,03	0	0,32
<i>Timer 1</i>	125 ns	33.991	3.187	39.778	1,01	99,58	99,5
	2,048 ms	484	0	483,52	0,16	0,88	0,97
	4,129 ms	242,99	0,09	242,88	0,06	0,33	0,28
	8,192 ms	122	0	121,72	0,01	0,06	0,28

Tabela 4.2: Comparação das medições de *duty cicle* (projeto: *Timer1*, frequência: 488Hz). Foram coletadas 111 amostras para cada valor de *duty cycle* medido

Duty Cycle (%)					Erro (%)	
Teórico	SOFIA		Arduino		SOFIA	Arduino
	Média	D. Padrão	Média	D. Padrão		
25	26	0	25,58	0,034	4	2,31
50	50	0	50,01	0,012	0	0,01
75	74	0	74,43	0,003	1,33	0,76
99	98	0	97,90	0,002	1,01	1,11

4.3 Métricas de Software

4.3.1 Análise Estática

Com o uso do *SonarQube*, foi realizada a análise estática do código com o objetivo principal de identificar e corrigir problemas relacionados à segurança, vulnerabilidades e manutenibilidade. O *SonarQube* fornece diversas métricas a respeito do projeto, além de classificar a qualidade do *software* com notas variando de A (melhor qualidade) a E (pior qualidade).

A primeira métrica relevante que a ferramenta fornece é uma medida de confiabilidade do sistema, baseada no número de *bugs*. A figura 4.18 mostra um gráfico relacionando cada classe do projeto com a quantidade de *bugs* encontrados. Este gráfico (assim como os mostrados nas figuras 4.19, 4.20 e 4.21) apresentam as classes do projeto, que são representadas por círculos. Quanto maior o tamanho do círculo, maior a quantidade de problemas encontrados nela e a cor do círculo mostra a gravidade do problema. O eixo horizontal mostra a quantidade de linhas que a classe possui, enquanto o eixo vertical o esforço (medido em tempo) estimado para resolver todos os problemas. Pode-se observar na figura 4.18 que uma classe se destaca pela sua quantidade de *bugs* (círculo de maior raio), esta classe é a *DataMemory_ATmega328P*.



Figura 4.18: Gráfico linhas de código x esforço para resolução dos bugs

Fonte: Autor

É importante ressaltar que, por ser uma ferramenta de análise estática, muitos problemas apontados podem não se aplicar ao projeto ou ser falsos positivos. Em se tratando de *bugs*, dos problemas que não

foram resolvidos sobraram apenas *bugs* relacionados ao tratamento de exceções (que está sendo feito com o uso de *Logs*, no *Android Studio*) e à conversão (*cast*) de valores (os casos de teste criados com o *JUnit4* estão testando estas conversões). Devido à estes problemas o sistema receberia classificação C em termos de confiabilidade, no entanto foi atribuída a nota E devido à um *bug* identificado na classe *DataBaseHelper* relacionado à abertura e fechamento do banco de dados (apesar deste ter sido corrigido conforme a sugestão proposta).

Outra métrica fornecida diz respeito à segurança do sistema e se baseia na quantidade de vulnerabilidades encontradas. A figura 4.19 apresenta um gráfico relacionando cada classe do projeto com a quantidade de vulnerabilidades (tempo para correção dos defeitos). Novamente, observa-se uma classe em destaque (maior círculo), esta é a *OutputFragment_Atmega328P*.

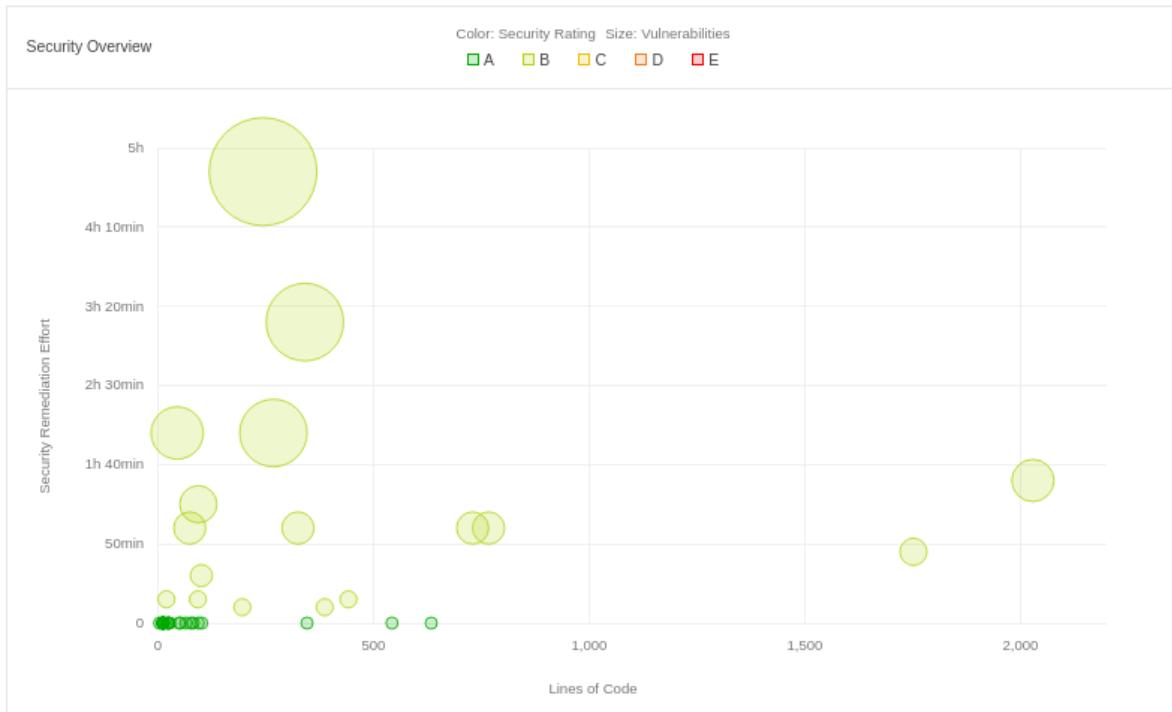


Figura 4.19: Gráfico linhas de código x esforço para resolução das vulnerabilidades

Fonte: Autor

Em termos de vulnerabilidades, o sistema apresenta melhores resultados em comparação aos *bugs*. Das que não foram resolvidas, sobraram apenas vulnerabilidades relacionadas à manipulação de variáveis estáticas. Muitas destas, no entanto, são variáveis privadas, de forma que não existem grandes problemas relacionados ao acesso delas por classes externas (como foi apontado pela ferramenta). O sistema foi classificado com o *ranking* B para segurança.

A próxima métrica obtida diz respeito a manutenibilidade do código. Esta medida é feita com base na complexidade cognitiva dos métodos e quanto ao uso devido/indevido de padrões de codificação

(esses são chamados *Code Smells*). A figura 4.20 apresenta um gráfico relacionando cada classe do projeto com a quantidade de *Code Smells* encontrados (tempo para correção dos defeitos). A classe com maior número de problemas para essa métrica é a *Timer1_ATmega328P*.

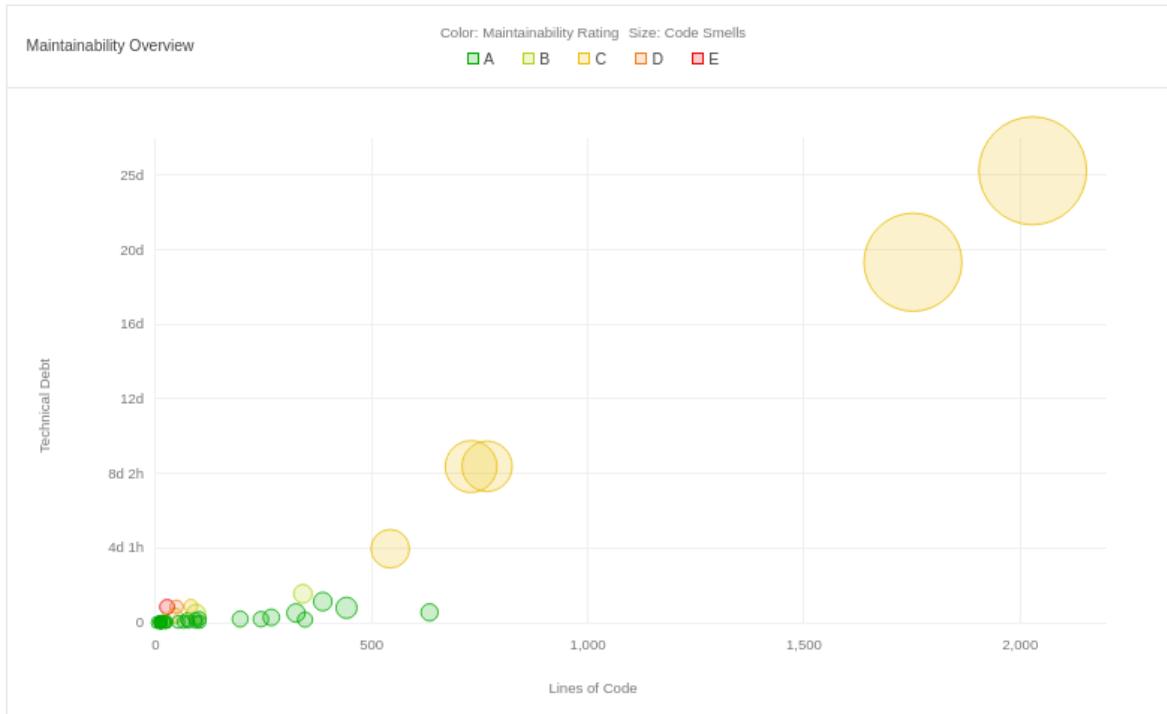


Figura 4.20: Gráfico linhas de código x esforço para resolução dos *Code Smells*

Fonte: Autor

Muitos dos problemas apontados nesta categoria são de menor importância, tais como remover linhas comentadas, mudar nomes de variáveis, etc. Os problemas de maior importância são os de complexidade cognitiva, que indicam que um método está muito grande, tornando-o de difícil compreensão. No entanto, alguns destes métodos não podem ser refatorados facilmente, já que comprometeria a arquitetura geral do projeto.

Outro problema apontado quanto à manutenibilidade foi a quantidade de níveis de herança que algumas classes possuem, excedendo o limite de 5 níveis. Não há muito o que fazer nestes casos, já que estas classes não herdam de nenhum código desenvolvido, mas de classes do próprio sistema Android (como é o caso da classe *AppCompatActivity*). Este foi o principal motivo para que o sistema fosse classificado com o *ranking C* para manutenibilidade.

Também foi obtida uma métrica de código duplicado. A figura 4.21 apresenta um gráfico relacionando cada classe do projeto com a quantidade de linhas de código duplicadas.

As classes com maior número de código duplicado são as classes referentes aos temporizadores (o *Timer1* é a classe em destaque na figura 4.21) e a CPU. Os modos de operação dos *Timers* se

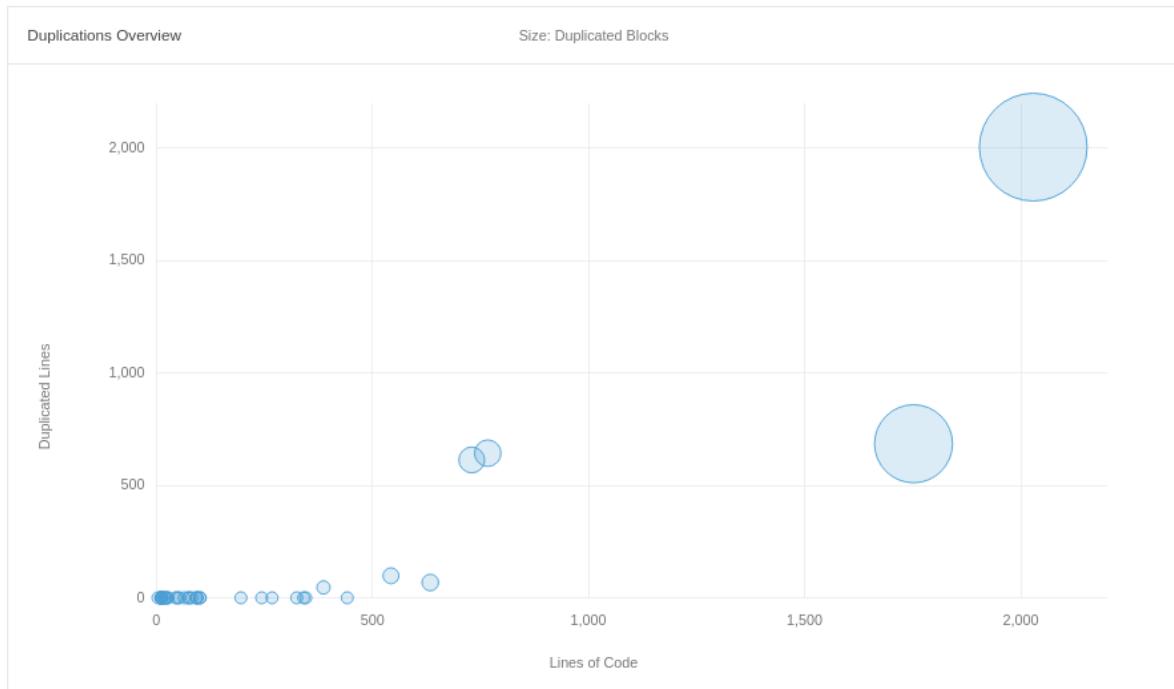


Figura 4.21: Gráfico linhas de código x linhas de código duplicadas.

Fonte: Autor

assemelham em muitos aspectos, no entanto, escrever um único método para gerenciar todos estes modos certamente o tornaria grande e complexo (gerando um problema na métrica de complexidade cognitiva) e portanto, foram criados diferentes métodos, cada um com pequenas variações, de forma a atender o modo de operação configurado. Um problema semelhante ocorre com a CPU, já que várias instruções realizam a leitura e a escrita dos valores na memória de maneira idêntica, mas suas operações são diferentes de modo que também não é possível unir códigos de diferentes instruções.

A última métrica importante de ser mencionada é a de complexidade ciclomática. Esta é uma métrica interessante pois diz qual é o número mínimo de casos de teste necessários para que eles cubram todo o projeto (cobertura de linhas). O valor obtido para complexidade ciclomática foi de 1.969.

4.3.2 Cobertura

Foram criados teste de unidade para todos os principais módulos do projeto (CPU, memórias de dado e de programa, conversor A/D, temporizadores, USART e módulo de interrupção), totalizando 808 casos de teste. O foco dos testes foi a cobertura de linhas e condicionais e análise de valor limite, este último aplicado principalmente aos testes da CPU.

A cobertura dos testes criado foi medida tanto com a ferramenta interna do *Android Studio* quanto com o plugin *JaCoCo* (integrado ao *SonarQube*). Houve uma diferença de 10% nas duas medidas

de cobertura por estas ferramentas, com o *Android Studio* registrando 59% de cobertura e o *JaCoCo* 49%, como mostram as figuras 4.22 e 4.23

Element ▾	Class, %	Method, %	Line, %
sofia	68% (165/242)	54% (375/690)	59% (3729/6288)

Figura 4.22: Medida de cobertura do projeto obtida com o *Android Studio*

Fonte: Autor

Coverage	48.1%
Lines to Cover	6,206
Uncovered Lines	3,166
Line Coverage	49.0%
Conditions to Cover	2,899
Uncovered Conditions	1,562
Condition Coverage	46.1%

Figura 4.23: Medida de cobertura do projeto obtida com o *JaCoCo*

Fonte: Autor

A cobertura dos testes por módulo é apresentada nas figuras 4.24 e 4.25. A partir destes dois gráficos apresentados, pode-se dizer que a medição realizada pelo *Android Studio* se mostra mais exata ao esperado, já que as medidas do *JaCoCo* indicam uma cobertura de 0% para a USART, quando existem 7 casos de teste para este módulo, e 100% para módulo de E/S, quando não foi escrito nenhum caso de teste para este módulo, por este ser mais dedicado à operações com interface gráfica.

Além dos testes automatizados, foram realizados testes manuais das funcionalidades desenvolvidas. Todos os códigos testados podem ser vistos na seção de apêndice deste trabalho.

4.4 Profiling

Foi utilizada a ferramenta de *profiling* do próprio *Android Studio* para avaliar o consumo de recursos do aplicativo e tentar identificar pontos de otimização. O projeto *Blink* (apêndice A), foi utilizado no simulador durante a execução do *profiling*.

O sistema foi avaliado quanto ao seu uso de CPU e memória, a figura 4.26 mostra o desempenho do sistema considerando estes dois fatores. Pode-se observar que o uso de CPU fica em torno de 25%, enquanto que o uso de memória é de aproximadamente 30MB.

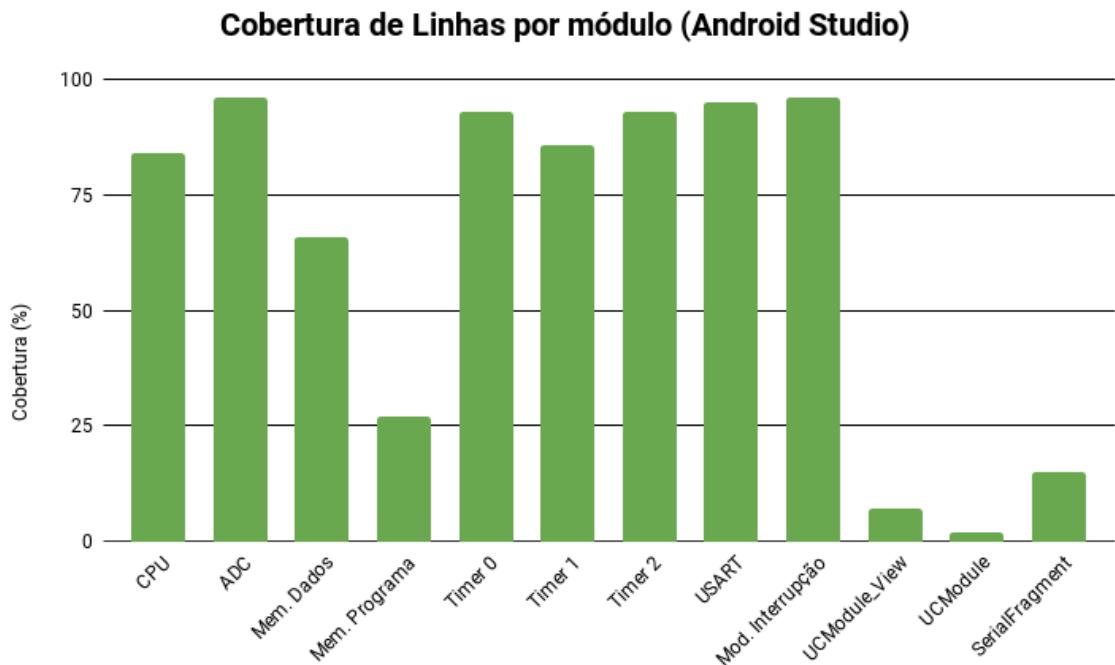


Figura 4.24: Cobertura por módulo (*Android Studio*)

Fonte: Autor

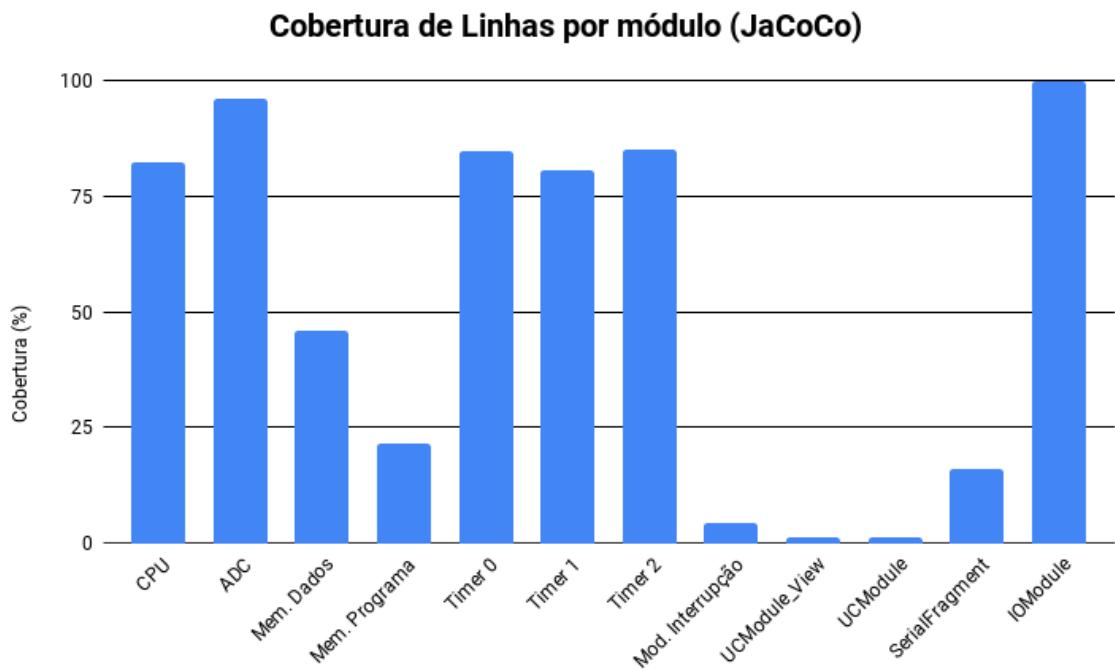


Figura 4.25: Cobertura por módulo (*JaCoCo*)

Fonte: Autor



Figura 4.26: Uso de CPU e memória do aplicativo para o projeto *Blink*

Fonte: Autor

Olhando especificamente para a CPU (utilizando a opção de gravação instrumentada para rastreamento de métodos), observou-se que muito do uso de CPU não está relacionado diretamente com as classes desenvolvidas, mas sim com chamadas à classes internas do Android. Do que foi desenvolvido, a classe *UCModule_View* se mostrou o principal ponto crítico, mais especificamente o método *run*, que faz a atualização do tempo simulado na tela. A figura 4.27 apresenta este resultado.

Name	Total (µs)	%	Self (µs)	%	Children ...	%
▼ m main() ()	1,249,116	100.00	15,072	1.21	1,234,044	98.79
▼ m dispatchMessage() (android.os.Handler)	221,829	17.76	3,849	0.31	217,980	17.45
▼ m handleCallback() (android.os.Handler)	217,980	17.45	4,374	0.35	213,606	17.10
▼ m run() (com.example.kollins.sofia.UCModule_View\$1)	213,606	17.10	8,855	0.71	204,751	16.39
▼ m setText() (android.widget.TextView)	200,767	16.07	3,730	0.30	197,037	15.77
▼ m setText() (android.widget.TextView)	197,037	15.77	4,507	0.36	192,530	15.41
▼ m setText() (android.widget.TextView)	192,530	15.41	43,733	3.50	148,797	11.91
▼ m checkForRelayout() (android.widget.TextView)	62,213	4.98	725	0.06	61,488	4.92
► m invalidate() (android.view.View)	31,587	2.53	264	0.02	31,323	2.51
► m requestLayout() (android.view.View)	29,729	2.38	1,238	0.10	28,491	2.28
m nullLayouts() (android.widget.TextView)	172	0.01	172	0.01	0	0.00

Figura 4.27: Tempo de uso da CPU (medido em uma janela de 5 minutos)

Fonte: Autor

Uma medida tomada quanto à este resultado foi atrasar a atualização do tempo simulado por um fator de 1024, ou seja, uma requisição de atualização da tela só é dada após 1024 passagens pelo método *run*. O valor de 1024 foi obtido experimentalmente de modo a não prejudicar a fluidez da interface (valores maiores fazem com que o tempo simulado salte de um valor para outro, dando a impressão que o sistema está a ponto de travar).

Essa medida teve um impacto significativo no desempenho deste método, que passou de um uso de CPU em torno de 17% para pouco menos de 0,5% , como mostra a figura 4.28.

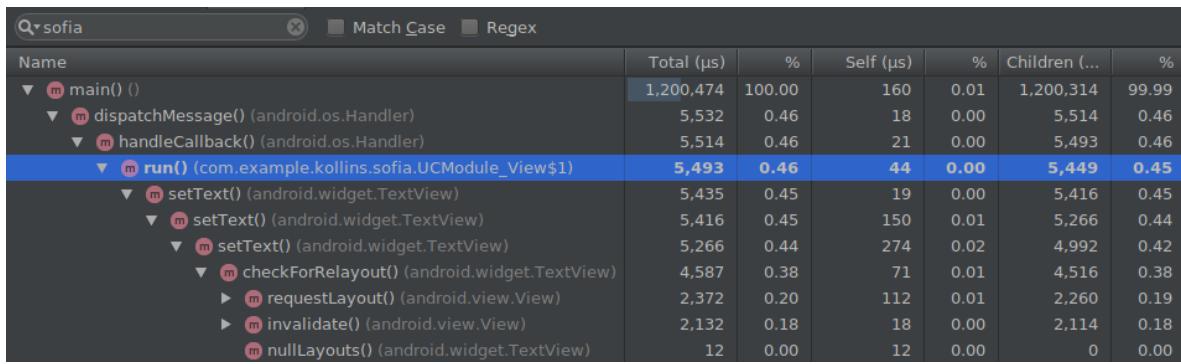


Figura 4.28: Tempo de uso da CPU após melhoria no método *run* (medido em uma janela de 5 minutos)

Fonte: Autor

Também foi feito um *profiling* específico para o uso de memória. Os resultados são mostrados na figura 4.29.

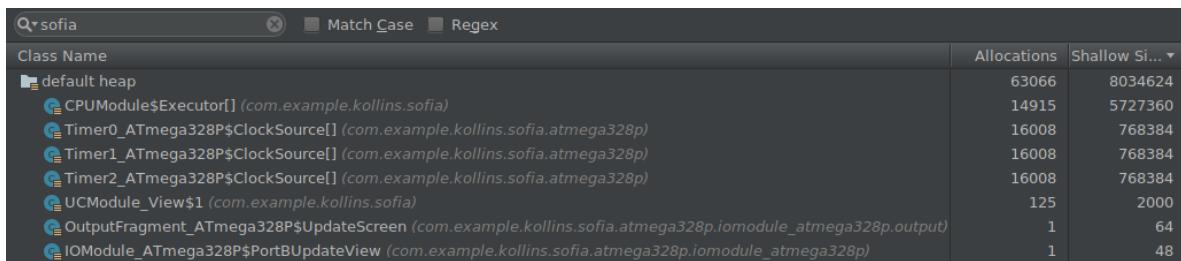


Figura 4.29: Consumo de memória (medido em uma janela de 5 minutos)

Fonte: Autor

O que se observa em termos de memória é que as estruturas de *Enum*, utilizadas na CPU e nos temporizadores, foram as responsáveis por fazer com que estes módulos se destacassem no uso de memória. De certa forma, este resultado já era esperado uma vez que estas são estruturas estáticas, o que demanda um maior tamanho em memória e não são limpas pelo *Garbage Collector* do Java.

Por outro lado, o uso de *Enum* permitiu obter uma funcionalidade próxima aos ponteiros de função que se tem na linguagem C, recurso bastante útil para fazer a decodificação das instruções na CPU (bem como a seleção do *prescaler* nos temporizadores).

Como mencionado na seção 3.2.2.2, manter a legibilidade do código durante a decodificação foi uma grande preocupação durante o desenvolvimento. O uso de memória é um preço a se pagar por essa facilidade obtida com a estrutura *Enum*. Mesmo assim, o aplicativo não apresenta valores elevados de consumo de memória quando comparado com outros aplicativos semelhantes, como mostra a seção 4.5.

4.5 Comparação entre aplicativos

Foram 3 os aplicativos selecionados para serem comparados com o projeto SOFIA, são eles:

- *Arduino Simulator Mini Free*
- *BoardMicro - AVR Simulator* (aplicativo e web).
- *AndMCU* (ou *MCU Prototype Board Simulator*)

Os aplicativos não são exatamente uma alternativa uns dos outros, mas apresentam características semelhantes e um mesmo propósito, que é a simulação de um microcontrolador.

Para a realização dos testes, foi utilizado novamente o projeto *Blink* do apêndice A nos simuladores *BoardMicro - AVR Simulator* e SOFIA. O aplicativo *Arduino Simulator Mini Free* não permite a edição de códigos, mas possui o mesmo projeto disponível para simulação. Quanto ao *AndMCU*, foi escrito um programa em *assembly* mostrado no apêndice B para a poder testá-lo, já que este simulador se destina à outra plataforma.

4.5.1 Interface

O primeiro ponto a ser comparado são as interfaces que cada aplicativo oferece para ao usuário.

O projeto SOFIA, como já foi mostrado na seção 4.2, procura oferecer uma interface simples, sem fazer referência à componentes eletrônicos. Na tela principal, o usuário tem sempre a sua disposição botões para realizar todas as tarefas que desejar no aplicativo, seja inserir novos elementos, reiniciar a simulação ou acessar funções adicionais, tudo isso sem que seja necessário acessar novas telas. A figura 4.30 mostra a interface principal do SOFIA com algumas entradas e saídas.

Uma desvantagem desta interface é que ela não é tão intuitiva aos usuários, que poderão levar algum tempo para se acostumar, principalmente aqueles que procuram por referências visuais dos componentes eletrônicos, como LEDs, *protoboards*, etc. Por este motivo, um botão de ajuda foi adicionado no aplicativo para levar o usuário à página do projeto, onde ele terá todas as informações necessárias para o uso do simulador.

Além disso, o tamanho da tela é reduzido, de forma que a inserção de muitos elementos pode prejudicar a experiência do usuário em termos de visualização dos resultados, já que apenas parte das entradas/saídas estarão visíveis (sem contar o possível uso do monitor serial).

O simulador *BoardMicro - AVR Simulator*, assim como no projeto SOFIA, também utiliza uma representação abstrata do sistema a ser simulado, como mostra a figura 4.31.

Na tela principal, apenas estão presentes a visualização de uma tela TFT (*Thin Film Transistor*)

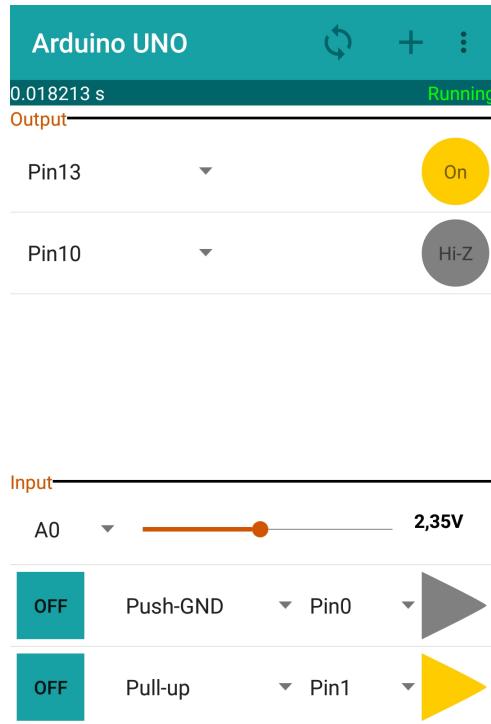


Figura 4.30: Tela principal do simulador SOFIA

Fonte: Autor

LCD na parte superior e dos PORTs de B a F (de cima para baixo). Na figura 4.31, a parte destacada em verde é o PORTC7, onde está ligado o LED interno do Arduino Esplora.

Não há botões nem outras telas no aplicativo e toda a navegação é feita por toques. Esta característica torna o sistema bastante difícil de se utilizar, principalmente por não haver uma ajuda indicando quais são as ações possíveis. Este simulador possui ainda uma versão *web*, com uma interface ligeiramente diferente, como mostra a figura 4.32, o que facilita seu uso.

Esta escolha de interface se justifica uma vez que a plataforma alvo deste simulador é o Arduino Esplora, que possui a forma de um *joystick* e pode ser conectado à um *display* externo (internamente, a tela TFT mostrada na parte superior está interligada nos pinos de comunicação SPI do ATmega32U4 para estabelecer esta conexão). Mesmo assim, a falta completa de informações/indicações para o usuário dificultam o entendimento do sistema, forçando o usuário a estudar o aplicativo por algum tempo antes de poder utilizá-lo.

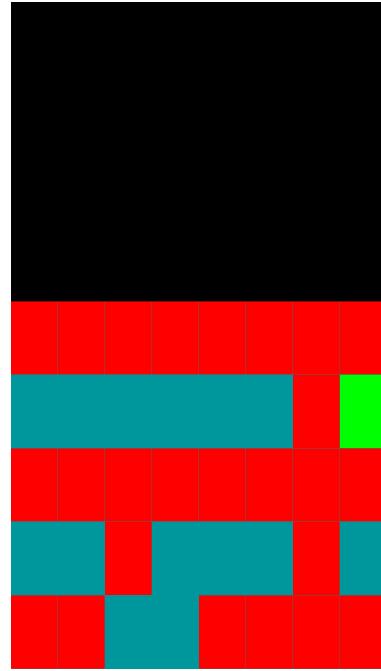


Figura 4.31: Tela principal do simulador *BoardMicro - AVR Simulator*

Fonte: Autor



Figura 4.32: Tela principal do simulador *BoardMicro - AVR Simulator*, versão para web

Fonte: Autor

Partindo para o *AndMCU*, este possui uma interface que lembra um *kit* de desenvolvimento PIC, mostrando inclusive o microcontrolador PIC18F458 (apesar do projeto funcionar com códigos assembly do microcontrolador 68705 da Motorola). A interface deste simulador é mostrada na figura 4.33.

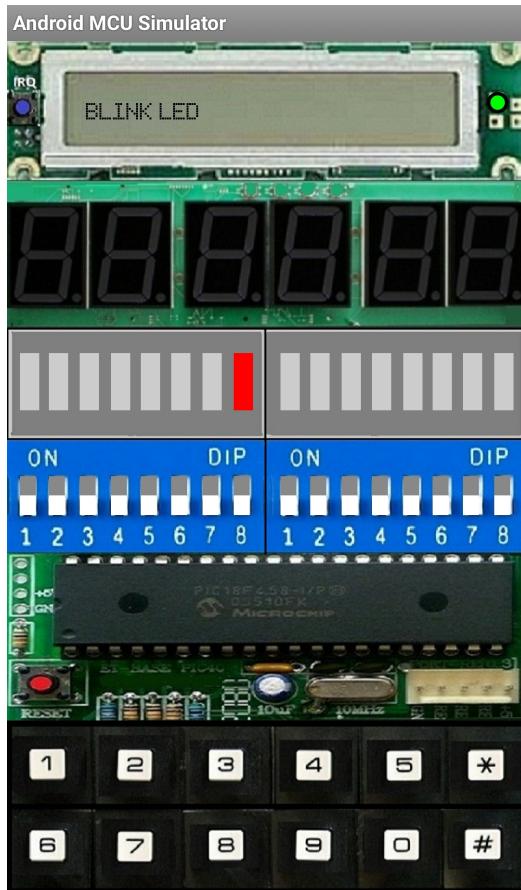


Figura 4.33: Tela principal do simulador *AndMCU*

Fonte: Autor

O *kit* apresentado conta com um *display* LCD de 16 caracteres, 6 *displays* de 7 segmentos, 2 barras gráficas com 8 LEDs cada, 2 *DIP* (*Dual In-line Package*) *Switchs* de 8 vias e um teclado numérico. Além disso, existe um botão azul no canto superior esquerdo para interrupções externas, um botão vermelho para *reset* manual no canto inferior esquerdo e um LED de status no canto superior direito. Também, o sistema utiliza o sensor de luminosidade do *smartphone* como entrada analógica e a documentação do projeto menciona ainda um *buzzer* [23].

Este simulador também não conta com menus de ajuda ou qualquer instrução de uso no aplicativo. Na verdade, uma vez escolhido o código a ser executado, não há mais nada a se fazer no simulador a não ser trabalhar em cima da simulação atual. A troca de arquivos para simulação exige sair do aplicativo e iniciá-lo novamente.

Apesar disso, o *AndMCU* apresenta um visual compacto, com boa disposição dos elementos na tela e com diversos recursos que devem ser facilmente reconhecidos por usuários já experientes com microcontroladores e eletrônica. O *hardware* integrado dá a possibilidade para que o usuário possa testar os programas mais comuns e mesmo alguns mais avançados.

Por fim, o *Arduino Simulator Mini Free* também prefere um visual mais realista do *hardware*, como mostra a figura 4.34.

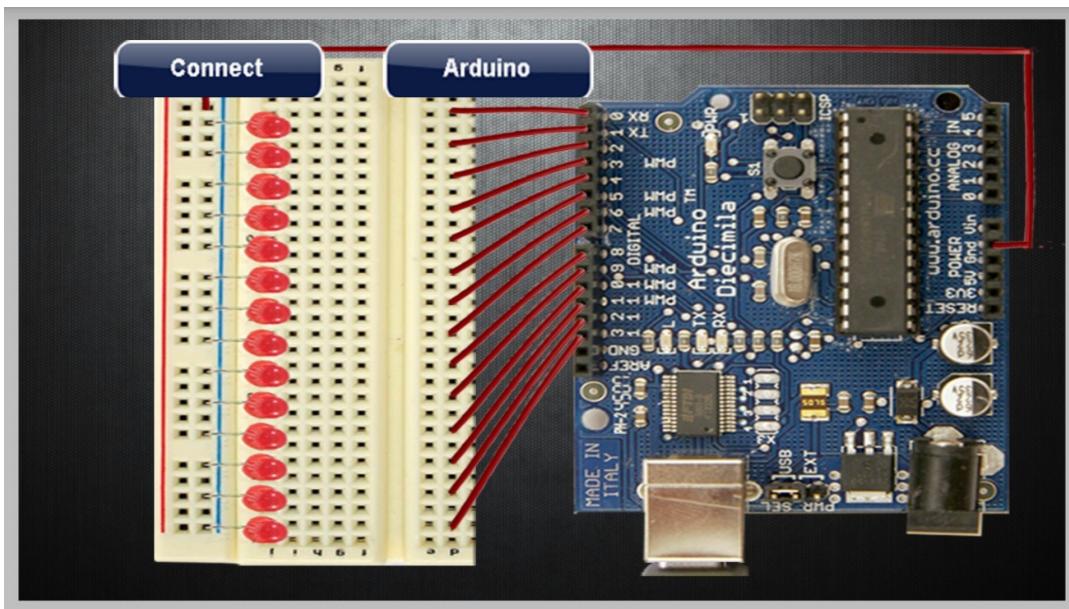


Figura 4.34: Tela principal do simulador *Arduino Simulator Mini Free*

Fonte: Autor

Durante a simulação, o circuito é alterado no *protoboard* de forma a fornecer o *hardware* adequado para simular o código. A figura 4.35 mostra a interface para simulação do projeto *Blink*.

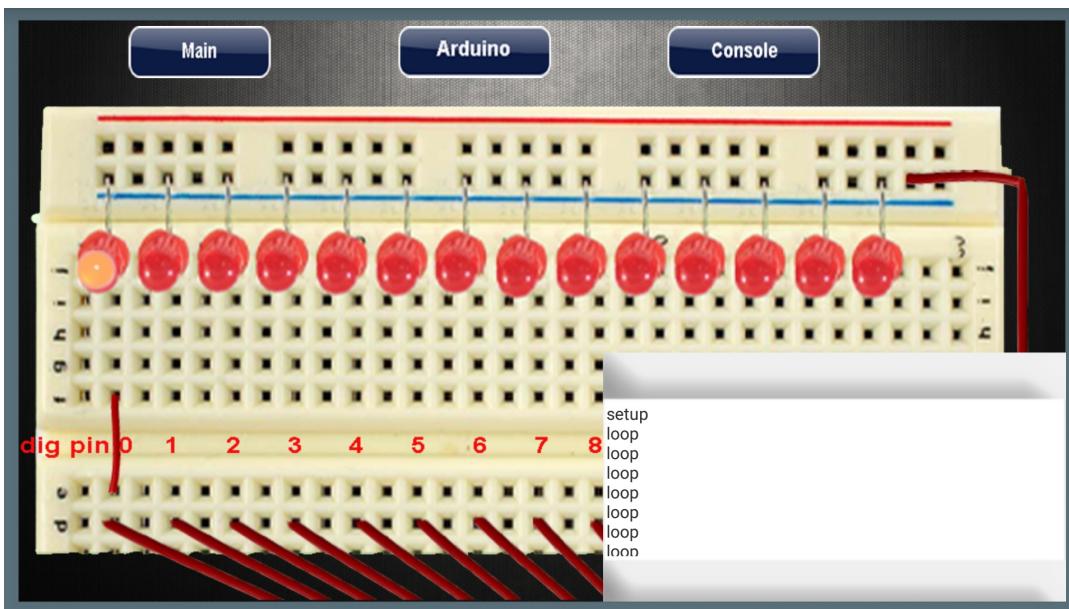
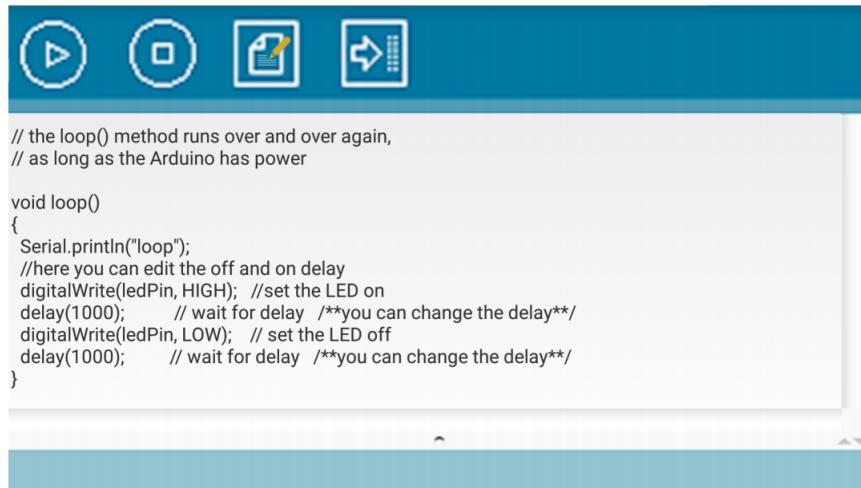


Figura 4.35: Tela de simulação do projeto *Blink*. O monitor serial no canto inferior direito pode ser escondido por meio do botão "Console"

Fonte: Autor

Também para a visualização do código, o simulador busca uma interface que lembra a IDE do Arduino, como mostra a figura 4.36.



```
// the loop() method runs over and over again,
// as long as the Arduino has power

void loop()
{
    Serial.println("loop");
    //here you can edit the off and on delay
    digitalWrite(ledPin, HIGH); //set the LED on
    delay(1000); // wait for delay /*you can change the delay*/
    digitalWrite(ledPin, LOW); // set the LED off
    delay(1000); // wait for delay /*you can change the delay*/
}
```

Figura 4.36: Tela edição do código-fonte a ser simulado.

Fonte: Autor

O realismo buscado pelo *Arduino Simulator Mini Free* pode ajudar o usuário que deseja, além de realizar a simulação, fazer a montagem posterior do circuito, já que o simulador mostra todo o esquemático necessário para executar o projeto.

Além disso, o *design* proposto minimiza as dúvidas quanto ao seu uso. Ainda que não haja instruções, os elementos utilizados são bem conhecidos daqueles que já utilizam a IDE do Arduino e os botões possuem identificadores bastante claros quanto sua função, de modo que, muito provavelmente, apenas um usuário iniciante pode vir a ter alguma dificuldade em utilizar este aplicativo.

4.5.2 Funcionalidades oferecidas

Cada aplicativo oferece um conjunto de funcionalidades e possibilidades de uso diferentes para o usuário.

O projeto SOFIA é focado na simulação do Arduino UNO e procura fornecer ao usuário um conjunto de ferramentas para que se possa, além de simular o funcionamento do código, obter informações a respeito de sua execução. Desta forma, o sistema oferece a possibilidade de medição de frequência, *duty cicle*, memória, além de permitir a configuração manual de uma referência externa de tensão para o conversor A/D. Também, o monitor serial integrado dá ao usuário uma maior possibilidade na depuração do código, permitindo o envio de mensagens de dentro do código para visualização externa.

Além disso, o projeto SOFIA busca ser um simulador genérico, permitindo que o usuário entre com qualquer código hexadecimal gerado, independente de sua fonte. O usuário também é livre para testar diversas possibilidades de ligação entre entradas e saídas, podendo mesmo interligar múltiplas

entradas em um mesmo pino; ligar uma entrada analógica em um pino digital e ver o efeito que níveis indefinidos de tensão podem gerar; conta com aviso em caso de curto-circuito, etc. A ideia principal é que, apesar de ser um simulador, as possibilidades de uso se aproximem do que o usuário encontrará em uma montagem eletrônica real.

Já o projeto *BoardMicro - AVR Simulator* é mais focado no uso do *display TFT*, fazendo com que o usuário explore este recurso para obter resultados de sua simulação, já que apenas a visualização dos PORTs pode não ser o suficiente para obter informações.

Para interação com a simulação, o simulador utiliza a leitura do acelerômetro do *smartphone* como entrada analógica e permite também o envio de comandos do GDB (*GNU Debugger*). A versão *web* permite ainda a medição de parâmetros como o estado dos registradores, pilha, SDRAM e a velocidade do sistema, também permitindo controlar a velocidade de simulação.

Como dito anteriormente, o aplicativo não possui botões ou menus e a navegação por toques não é clara, mas é por meio dela é possível abrir o menu para carregar um código hexadecimal e acessar o *prompt* do GDB. Um programa a ser simulado pode tanto ser escolhido de um banco de exemplos (com 4 códigos disponíveis no aplicativo) ou carregados pelo usuário a partir de um repositório do *Dropbox*.

O *AndMCU* fornece um *kit* de desenvolvimento ao usuário e as possibilidades de interação com o aplicativo não vai muito além das possibilidades deste *hardware* virtual (com exceção para o uso do sensor de luminosidade do *smartphone* para entrada analógica). A única interação do usuário que é externo à este *kit* é na escolha do código a ser simulado ao iniciar o aplicativo.

Apesar disso, o sistema permite que diversas configurações sejam feitas no código em *assembly* que será simulado. Entre as possibilidades estão o controle de velocidade de simulação, modo de simulação passo a passo, desabilitar os *DIP Switches*, escrever conteúdo da memória em arquivo, entre outras possibilidades descritas na documentação.

O simulador fornece, assim como o *BoardMicro - AVR Simulator*, um banco contendo 10 exemplos já codificados e um arquivo *template* para servir de base para novos códigos. O usuário que desejar simular um código diferente deve adicioná-lo na pasta `/sdcard/AndMCU` do *smartphone*.

E finalmente, o *Arduino Simulator Mini Free* se mostrou-se o mais limitado em termos de funcionalidade. O simulador fornece 5 códigos exemplo para o usuário, que pode realizar modificações apenas de determinados parâmetros, como alterar o pino de saída, tempo de *delay* e mensagem do *display LCD*, não sendo possível criar um código customizado para a simulação.

O usuário também não pode fornecer dados de entrada durante a simulação, podendo apenas visualizar o funcionamento do circuito. Isso acaba invalidando um dos códigos exemplo, que requer uma entrada analógica em um circuito utilizando LDR (*Light Dependent Resistor*).

O *Arduino Simulator Mini Free* possui duas outras versões pagas na *Amazon Store*, com diferentes projetos e a possibilidade de edição do circuito eletrônico em uma delas (segundo a descrição). No entanto, nenhuma destas versões foi testada.

4.5.3 Requisitos do sistema e consumo de recursos

4.5.3.1 Sistema Operacional

Todos os aplicativos suportam o sistema Android, com o *Arduino Simulator Mini Free* disponível também para IOs (sob os nomes *Arduino Simulator 2X - Learn and DIY Safely* e *Arduino Simulator - Full Pack 2x*). A tabela 4.3 indica qual a versão mínima do Android é requerida por cada aplicativo. Este valor foi obtido a partir da descrição do aplicativo nas lojas *Google Play Store* e *Amazon Store*.

Tabela 4.3: Versão do Android requerida por cada aplicativo

Aplicativo	Versão do mínima
Arduino Simulator Mini Free	2.2
AndMCU	2.3.3
BoardMicro - AVR Simulator	4.0
SOFIA	5.0

Ao criar um projeto no *Android Studio*, pode-se verificar qual a distribuição de usuários em cada versão do Android. A figura 4.37 apresenta esta distribuição e por ela, pode-se observar que, com exceção do projeto SOFIA, todos os simuladores podem ser executados em 100% dos dispositivos Android ativos no momento.

4.5.3.2 Tamanho

O gráfico mostrado na figura 4.38 mostra o tamanho do arquivo APK (*Android Package Kit*) para cada aplicativo.

Pode-se observar que o *Arduino Simulator Mini Free* possui um tamanho muito superior aos demais, resultado este um tanto inesperado, uma vez que a versão *Full Pack* deste aplicativo possui um tamanho menor de apenas 9MB.

ANDROID PLATFORM VERSION	API LEVEL	CUMULATIVE DISTRIBUTION
4.0 Ice Cream Sandwich	15	
4.1 Jelly Bean	16	99.2%
4.2 Jelly Bean	17	96.0%
4.3 Jelly Bean	18	91.4%
4.4 KitKat	19	90.1%
5.0 Lollipop	21	71.3%
5.1 Lollipop	22	62.6%
6.0 Marshmallow	23	39.3%
7.0 Nougat	24	8.1%
7.1 Nougat	25	1.5%

Figura 4.37: Distribuição acumulada de usuários Android. Como destacado, o simulador SOFIA poderá atingir pouco mais de 70% dos dispositivos Android ativos no momento, porém este número tende a aumentar com a modernização dos aparelhos

Fonte: *Android Studio*

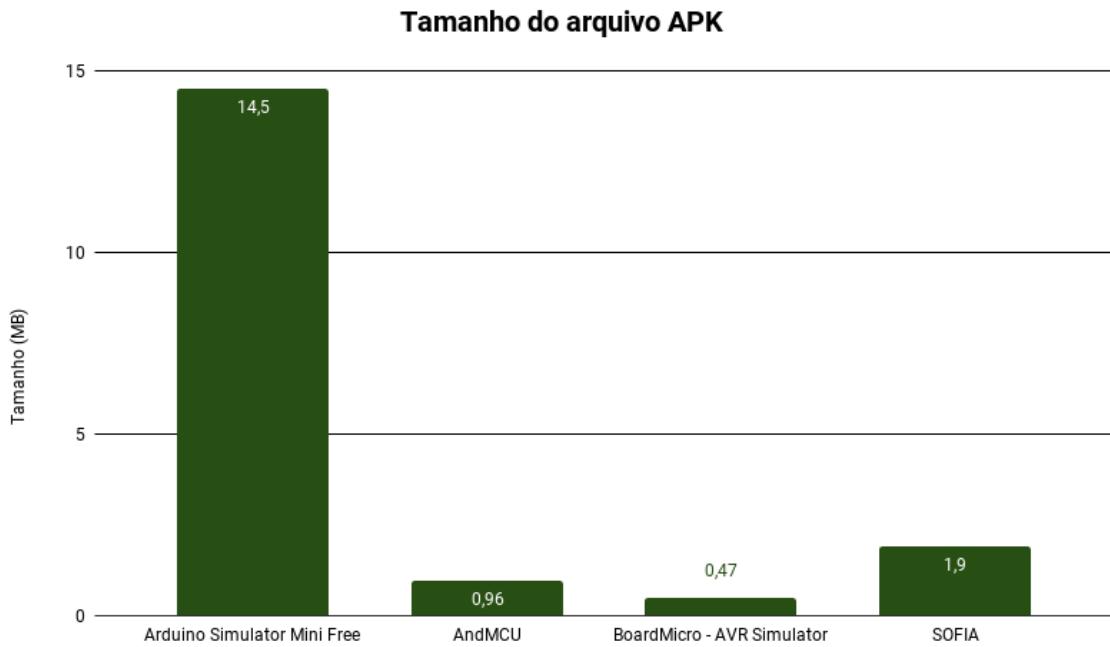


Figura 4.38: Tamanho do arquivo APK (MB) de cada aplicativo

Fonte: Autor

4.5.3.3 Uso de CPU

O uso de CPU foi medido utilizando um *shell* remoto acessado pelo ADB (*Android Debug Bridge*). O comando *top* pode ser utilizado de modo análogo ao *htop* dos sistemas Linux, e mostra diversas informações a respeito dos processos em execução, incluindo o uso de CPU. O gráfico da figura 4.39 apresenta a média obtida das medições, com indicação do desvio padrão. O aplicativo *AndMCU* foi testado em duas condições diferentes, para alta e baixa velocidade de simulação, definida pela diretriz *.speed*.

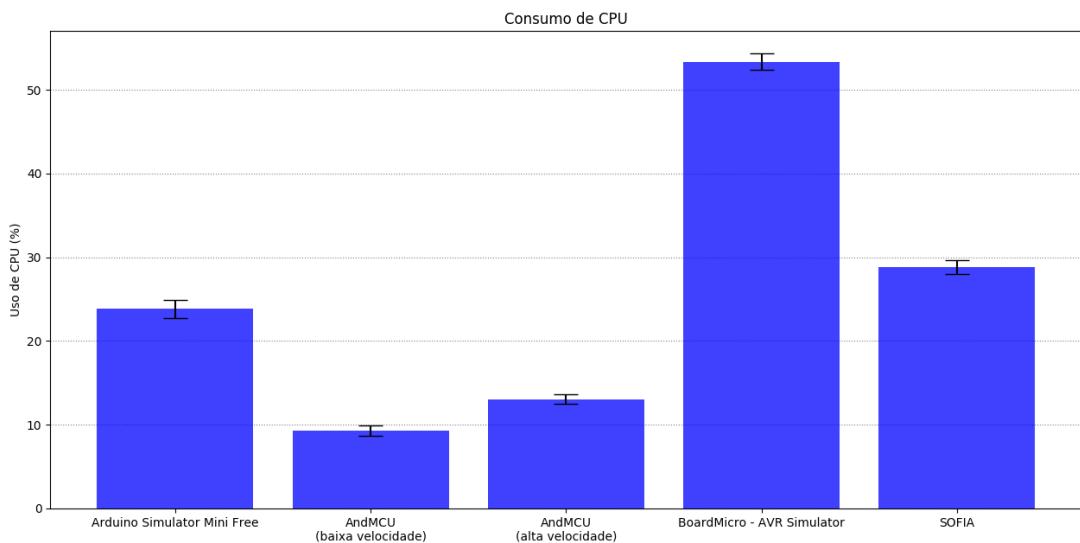


Figura 4.39: Consumo médio de CPU (%) de cada aplicativo. Para cada processo, foram coletadas 397 amostras

Fonte: Autor

É difícil comparar o resultado do *AndMCU* com os demais aplicativos pois, em todos eles, o código foi o mesmo (em C), a plataforma é a mesma (Arduino) e a frequência de intermitência é a mesma (500Hz), enquanto que o *AndMCU* está em outro contexto, com código em *assembly*, produzindo uma frequência de intermitência difícil de ser determinada. O que se observou, no entanto, foi que este aplicativo apresentou um consumo de CPU bastante reduzido em comparação aos demais, mesmo quando a velocidade de simulação foi configurada para a máxima possível.

Por outro lado, o simulador *BoardMicro - AVR Simulator* apresentou um consumo muito maior que a média dos demais, resultado este que provavelmente está relacionado com sua velocidade de simulação, como mostra a seção 4.5.4, enquanto o projeto *SOFIA* e o *Arduino Simulator Mini Free* apresentaram resultados próximos da média.

Esta medida de CPU tem impacto direto no consumo de bateria de cada aplicativo. A figura 4.40 apresenta um gráfico indicando o consumo médio percentual de bateria para cada aplicativo em um período de 2 horas, onde cada aplicativo ficou em execução contínua durante 30 minutos (o aplicativo *AndMCU* foi utilizado em seu modo padrão, ou seja, baixa velocidade). Os resultados foram medidos com a ferramenta *Battery Historian* e pode-se observar que o gráfico apresenta as mesmas proporções ao apresentado na figura 4.39.

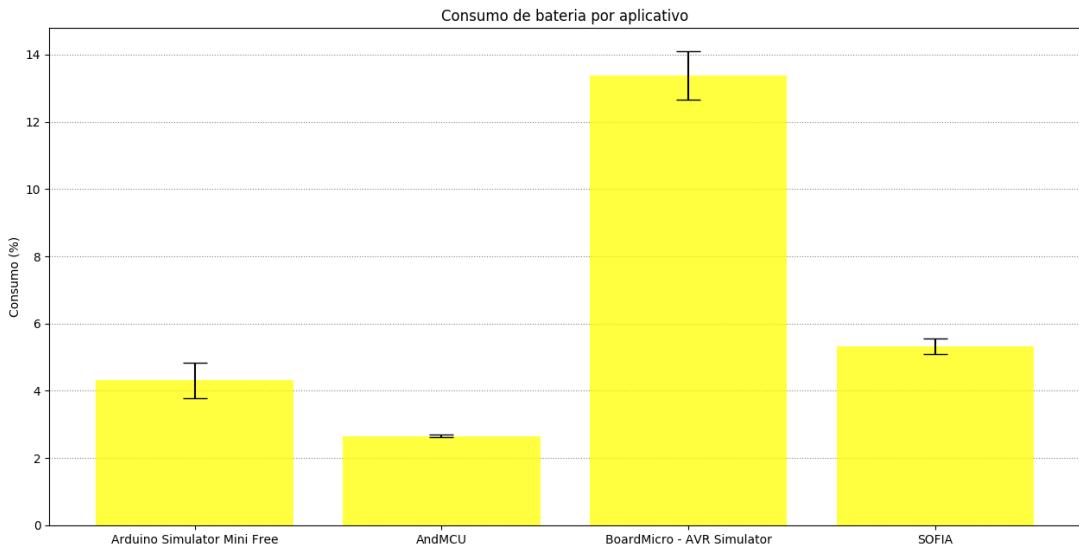


Figura 4.40: Consumo médio de Bateria (%) de cada aplicativo em um período de 2 horas, com cada aplicativo executando o projeto *Blink* continuamente por 30 minutos. Foram realizadas 5 medições para cada aplicativo.

Fonte: Autor

4.5.3.4 Uso de Memória

O uso de memória por cada aplicativo foi também medido do uso do *shell* remoto, com o comando *dumpsys meminfo*. Segundo a documentação desta instrução [24], o campo PSS (*Proportional Set Size*), que é a soma da memória privada (região de memória pertencente apenas ao aplicativo e liberada ao sistema quando este é encerrado) e da memória compartilhada (total de memória compartilhada utilizada pelo aplicativo, dividida pelo número de processos que compartilham a esta região de memória), fornece uma boa medida do "peso" deste processo na memória principal, e portanto, este foi o valor utilizado para comparar o uso de memória entre os aplicativos. O gráfico da figura 4.41 apresenta a média obtida das medições realizadas, com indicação do desvio padrão.

No quesito memória, embora o comentário a respeito do uso da estrutura *Enum* na seção 4.4, o projeto SOFIA se mostrou bastante competitivo em relação aos demais simuladores.

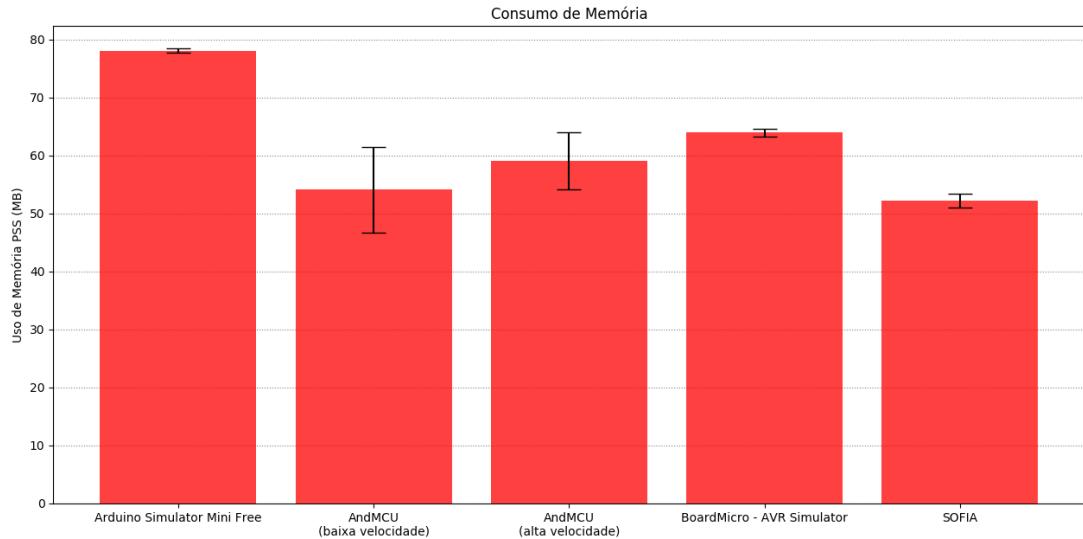


Figura 4.41: Consumo de memória (PSS em MB) de cada aplicativo. Foram realizadas 1000 medições para cada aplicativo

Fonte: Autor

4.5.4 Velocidade de simulação

A velocidade de simulação aqui apresentada é uma medida da percepção que o usuário tem do sistema, quando comparado com o tempo do mundo real.

É possível ter uma ideia clara da velocidade de simulação do SOFIA, uma vez que é apresentada a contagem do tempo simulado para o usuário. Este tempo simulado está reduzido em torno de 500x em comparação com o tempo real, já que são contados 2ms a cada segundo. Assim, o usuário percebe, na prática, um sistema com frequência de *clock* efetiva de 32kHz, uma vez que o *hardware* no qual o simulador se baseia é o Arduino UNO com frequência de 16MHz.

Para os demais simuladores, a velocidade de simulação foi medida alterando o projeto *Blink* para que a intermitência fosse de 1s. O *Arduino Simulator Mini Free* se mostrou bastante fiel ao tempo real, exibindo, de fato, uma intermitência de 1s, fazendo com que o usuário não perceba a diferença deste com um sistema real. Já o simulador *BoardMicro - AVR Simulator* se mostra acelerado, com uma frequência de intermitência 4x maior que a esperada.

Para o projeto *AndMCU*, não foi possível ter uma ideia clara de sua real velocidade de simulação pois, como já mencionado anteriormente, este aplicativo está em outro contexto do qual não é bem determinado/conhecido como no caso do Arduino. No entanto, é possível ler na documentação do projeto que o sistema possui uma velocidade de simulação adequada para que os resultados possam ser observados por estudantes, ou, como o autor coloca em outras palavras, "não se deve esperar deste

simulador uma velocidade elevada como o de um emulador de CPU"(em tradução live) [23].

4.5.5 Documentação

O projeto SOFIA conta com uma documentação *on-line* (em inglês) hospedada no *Gitbook*. Esta documentação pode ser acessada a partir do repositório do *Github* ou de dentro do próprio aplicativo, nos menus de ajuda e de informações do projeto.

A documentação do projeto SOFIA é dividido em 3 partes. A primeira mostra uma visão geral, apresentando uma breve descrição do que é o projeto, seus objetivos, público alvo, etc. A segunda parte é um manual do usuário e descreve, passo-a-passo, quais os procedimentos para baixar e instalar o simulador (bem como a versão modificada da IDE do Arduino) e como utilizá-lo, detalhando todos os seus recursos. Por fim, a última parte apresenta uma visão mais voltada ao desenvolvimento do projeto, com diagramas, métricas de *software* e comentários a respeito do código desenvolvido. O *link* para acessar a documentação do projeto SOFIA é mostrado abaixo:

- **Documentação SOFIA:**

- <https://project-sofia.gitbook.io/project/>

O aplicativo *AndMCU* também conta com duas páginas de documentação *on-line* (também em inglês) hospedadas no *Google Sites* e que podem ser acessadas a partir da página de *download* do aplicativo na *Google Play Store*.

Essa documentação apresenta as características e funcionalidades do sistema; o conjunto de instruções do simulador, bem como algumas diretrizes em *assembly* que podem ser utilizadas; detalham a organização do processador que foi implementado, mostrando seu mapa de memória, esquema de ligação das entradas e saída e uma explicação sobre a CPU e seus registradores; apresenta o *hardware* virtual; entre outras informações. Abaixo é mostrado o endereço eletrônico onde este material pode ser acessado

- **Documentação AndMCU:**

- <https://sites.google.com/site/hkonstas/android-stuff/andmcu>
 - <https://sites.google.com/site/hkonstas/android-stuff/andmcu-andmcu-documentation>

Pouca informação foi encontrada para os projetos *BoardMicro - AVR Simulator* e *Arduino Simulator Mini Free*, além das descrições presentes nas páginas de *download* destes aplicativos. O projeto *BoardMicro - AVR Simulator* conta apenas com um breve arquivo *README* em seu repositório no *Github*, disponível no *link* abaixo:

- **Repositório do projeto *BoardMicro - AVR Simulator*:**

– <https://github.com/blakewford/boardmicro>

Quanto ao *Arduino Simulator Mini Free*, um vídeo disponível no *YouTube* foi o único material encontrado a respeito da ferramenta. O vídeo mostra a versão do aplicativo para IOs e pode ser acessado pelo *link* a seguir:

- **Vídeo informativo do aplicativo *Arduino Simulator Mini Free*:**

– <https://www.youtube.com/watch?v=LJxdy6FHVGg>

4.5.6 Disponibilidade

Todos os aplicativos apresentados podem ser obtidos gratuitamente na internet. O projeto *AndMCU* pode ser baixado na *Google Play Store*, enquanto o *Arduino Simulator Mini Free* está disponível na *Amazon Appstore*. O simulador *BoardMicro - AVR Simulator* pode ser adquirido em ambas, *Google Play Store* e *Amazon Appstore*, além de ser um projeto de código aberto disponível no *Github*. O projeto *SOFIA*, por ainda estar em fase experimental, pode ser baixado apenas na forma de código-fonte pelo *Github*, ou na forma de APK no site do projeto no *Gitbook*, apresentado na seção 4.6, devendo ser disponibilizado também na *Google Play Store* posteriormente. Além disso, o projeto *SOFIA* é o único com uma versão em português, se adaptando automaticamente caso o usuário esteja com esta língua ativa no sistema Android.

O simulador *Arduino Simulator Mini Free* possui ainda duas versões pagas para Android. Uma delas se chama *Arduino Simulator Mini*, e custa R\$ 2,35. A outra se chama *Arduino Simulator DIY Safely*, e possui o preço de R\$ 4,44. Embora não fique claro quais as diferenças entre cada versão olhando apenas para a descrição dos aplicativos, pode-se notar uma grande diferença nas permissões que cada aplicativo requer para funcionar. A versão gratuita requer uma quantidade enorme de permissões, tais como acesso ao GPS, leitura do histórico de navegação e favoritos do navegador *web*, entre outras que não estão relacionadas com sua funcionalidade, chegando a ser reconhecida como uma ameaça pelo antivírus *AVG Pro*. O *Arduino Simulator Mini* requer permissões para gravar áudio, acessar o cartão SD, enquanto que o *Arduino Simulator DIY Safely* apenas exige a permissão para acessar o recurso de vibração do *smartphone*.

Vale mencionar também que o *Arduino Simulator Mini Free* é o único dos simuladores testados a conter propagandas, que são exibidas a qualquer momento durante o uso do sistema Android e não apenas dentro do aplicativo.

4.6 Código

O código-fonte do projeto foi disponibilizado em um repositório *on-line* no *Github* sob a licença *Apache 2.0*. As modificações feitas na IDE do Arduino também estão disponíveis no *Github* sob a mesma licença do projeto original (GPL v2.0).

- **Projeto SOFIA:** <https://github.com/kollinslima/ProjectSOFIA>
- **Arduino IDE:** <https://github.com/kollinslima/Arduino/tree/android>

Capítulo 5

Conclusão

Nesta monografia foi apresentado o projeto SOFIA, um simulador do Arduino UNO criado para Android.

Pode-se dizer que o sistema desenvolvido atende aos objetivos propostos: o sistema é capaz de executar códigos escritos para o Arduino UNO (ATmega328P) diretamente no dispositivo Android, bem como permitir que o usuário interaja com o sistema por meio de sinais de entrada (digital e analógico) ou fazendo medições dos estados dos pinos digitais. O simulador também conta com um monitor serial e com recursos para depuração do código simulado, tais como frequencímetro e mapa de memória. O usuário tem ainda a disposição uma IDE Arduino que foi modificada para facilitar o processo de transferência de códigos entre o computador e simulador.

No início do projeto, a arquitetura projetada e as técnicas de desenvolvimento estavam muito atreladas à experiência que se tinha com aplicações *desktop*, o que contribuiu para que muitas das escolhas de implementação não resultarem na melhor solução do problema em um sistema *mobile*. Na seção 4.4, por exemplo, mostrou-se o problema relacionado com a estrutura *Enum*, e posteriormente descobriu-se que esta não é uma estrutura recomendada para se utilizar no Android.

No entanto, o principal problema encontrado na arquitetura projetada, que veio a impactar seriamente o desempenho do simulador, foi o uso de *threads*. *Thread* é um recurso importante e muito poderoso e seu uso no Android é muitas vezes inevitável, já que o sistema não permite que operações de E/S sejam realizadas na *thread* principal. No entanto, a arquitetura original contava com 6 *threads* permanentes e diversas *AsyncTasks* criadas dinamicamente, o que tornava o sistema cerca de 10.000x mais lento que o atual mostrado neste trabalho (e não contava com todas as funcionalidades desenvolvidas). Aos poucos, esta estratégia inicial foi sendo substituída por uma abordagem de *thread* única, mas ainda existem trechos de código residual que não puderam ser refatorados.

Outro ponto importante que se pode concluir é a respeito dos testes realizados. O desenvolvimento

de testes de unidade automatizados foi uma atividade que consumiu bastante tempo (um dos motivos foi a inexperiência do autor com as ferramentas utilizadas), porém ajudaram a identificar defeitos que seriam difíceis de depurar caso não tivessem sido corrigidos, mostrando sua importância para a qualidade do *software* em desenvolvimento.

Em comparação com outros aplicativos, como apresentado na seção 4.5, o simulador desenvolvido apresenta bons resultados, fazendo deste uma alternativa viável, principalmente por oferecer alguns recursos que não estão presentes em outros simuladores (como ferramentas de depuração).

Por fim, conclui-se que este projeto teve grande importância para o aprendizado de novas tecnologias, principalmente no que diz respeito ao desenvolvimento Android, programação para Arduino, organização e arquitetura do microcontrolador ATmega328P e teste de *software*, além, obviamente, da experiência em se desenvolver um projeto de *software* completo, usando uma metodologia ágil, fazendo a especificação de requisitos e tomando várias decisões de projeto que tiveram grande impacto no produto final.

5.1 Trabalhos futuros

Muito ainda pode ser feito para melhorar o aplicativo. Como mencionado anteriormente, ainda existe código residual que não pode ser refatorado de uma arquitetura que não se mostrou eficiente, o que causa impacto no desempenho do sistema. Portanto, uma revisão da arquitetura e a adequação do código, buscando estratégias de implementação, algoritmos e estruturas de dados que sejam mais eficientes no Android é certamente uma tarefa importante para a continuidade do trabalho e para as próximas versões.

Além disso, como também foi apontado, a atividade de teste se mostrou importante para a qualidade do código. Outras técnicas de teste, como teste de interface, teste de mutação, etc., não foram aplicados ao projeto nesta versão e podem contribuir ainda mais para tornar o sistema mais confiável.

Também, a interface do aplicativo, embora tenha sido planejada para se fosse simples e o mais intuitivo possível, ainda tem muito o que melhorar e é algo a ser pensado junto a revisão da arquitetura.

Em termos de funcionalidades, ainda há vários módulos do Arduino que não foram implementados: modos de hibernação, EEPROM, comparador analógico, SPI, etc. Estas funcionalidades, junto ao suporte de novas placas, trariam mais utilidade ao simulador e o tornaria mais completo.

O mapa de memória também é uma funcionalidade que pode ser revista em termos de separação de informações, ou seja, mostrando diferentes abas com informações a respeito da pilha, registradores e RAM externa, por exemplo, facilitaria a leitura das diferentes regiões de memória e seu conteúdo. Pensando nisso também, o valor do PC (que não está mapeado em memória) também pode ser uma

informação relevante para o usuário. Se utilizado em conjunto com um recurso de execução passo a passo do código, pode fazer do simulador uma ferramenta bastante útil para aqueles usuários mais interessados no estudo do processador em um nível mais baixo.

Outra funcionalidade interessante a ser implementada é a geração de arquivos de *log*. Este recurso poderia ser integrado, por exemplo, no frequencímetro, dando a possibilidade de gravação da forma de onda que está sendo gerada para que o usuário possa visualizá-la em algum programa externo (como o *gnuplot*), ou ainda, pensar na criação de um osciloscópio virtual. Esta é uma outra funcionalidade que traria o simulador para um nível mais acadêmico.

Por fim, um recurso importante, observado nos outros aplicativos, mas que não foi implementado nesta versão são os exemplos pré-compilados já embutidos no simulador. Este recurso daria a possibilidade de um usuário testar o simulador assim que o *download* fosse concluído, não dependendo de nenhum outro *software* ou aplicativo para gerar os códigos mais comuns, como o projeto *Blink* que foi utilizado durante os testes.

Referências

- [1] Olhar Digital. Microsoft relembra produtos que eram sucesso na época do XP. Site. Disponível em: <https://olhardigital.com.br/noticia/microsoft-relembra-produtos-que-eram-sucesso-na-epoca-do-xp/40602>, 2014. Acesso: 11 de Março de 2018.
- [2] Tecmundo. Samsung Galaxy S9. Site. Disponível em: <https://comparador.tecmundo.com.br/samsung-galaxy-s9/>. Acesso: 18 de Março de 2018.
- [3] LECHETA, Ricardo R. *Google Android: Aprenda a criar aplicações para dispositivos móveis com o Android SDK*. Livro. Novatec Editora, São Paulo, Brasil. 4º Edição, 2015.
- [4] Documentação Android. Content license. Site. Disponível em: <https://source.android.com/setup/licenses>. Acesso: 18 de Março de 2018.
- [5] DEITEL, P., DEITEL, H., and DEITEL, A. *Android for Programmers: An App-Driven Approach*. Livro. Prentice Hall Press, Upper Saddle River, NJ, USA. 1º Edição, 2012.
- [6] JUNIOR, José Ernesto Almas de Jesus. Implementação de um osciloscópio de baixo custo com exibição gráfica em aplicativo para android. Trabalho de Conclusão de Curso. Universidade de São Paulo. São Carlos, 2016. Disponível em <http://www.tcc.sc.usp.br/tce/disponiveis/97/970010/tce-04012017-163919/?&lang=br>. Acesso: 20 de Novembro de 2018.
- [7] EBERENDU, A. C., OMAIYE, B. O., and NOWOKORIE, E. C. Using android application to turn smart device into digital microscope on arduino and window platform. In *2017 IEEE 3rd International Conference on Electro-Technology for National Development (NIGERCON)*, pages 508–513, 2017. Disponível em: <https://ieeexplore.ieee.org/document/8281919>. Acesso: 20 de Novembro de 2018.
- [8] TENG, H. F., WANG, M. J., and LIN, C. M. An implementation of android-based mobile virtual instrument for telematics applications. In *2011 Second International Conference on Innovations*

- in Bio-inspired Computing and Applications*, pages 306–308, 2011. Disponível em: <https://ieeexplore.ieee.org/document/6118785>. Acesso: 20 de Novembro de 2018.
- [9] OVERMAN, J., POOL, L., KREUK, L., and KETEL T. Arduino - The Open-Source IDE. Site. Disponível em: <https://delftswa.gitbooks.io/desosa-2017/content/arduino/chapter.html>, Acesso: 08 de Abril de 2018.
- [10] Labcenter Electronics. Create Your Package. Site. Disponível em: <https://www.labcenter.com/buy-vsm/>. Acesso: 15 de Abril de 2018.
- [11] VirtualBreadboard. VBB Software Licenses. Site. Disponível em: <http://www.virtualbreadboard.com/DocView.html?doc=WebShop/WebShop>. Acesso: 15 de Abril de 2018.
- [12] Simuino. Arduino UNO/MEGA Simulator. Site. Disponível em: <http://web.simuino.com/home-1>. Acesso: 15 de Abril de 2018.
- [13] HUANG, Stanley. CodeBlocks Arduino IDE. Site. Disponível em: <http://arduinodev.com/codeblocks/>. Acesso: 15 de Abril de 2018.
- [14] Autodesk. O Tinkercad é um aplicativo simples e on-line de projeto e impressão 3d para todos os usuários. Site. Disponível em: <https://www.tinkercad.com/>. Acesso: 15 de Abril de 2018.
- [15] SIMMONS, Stan. Simulator Download. Site. Disponível em: <https://www.sites.google.com/site/unoardusim/services>. Acesso: 20 de Novembro de 2018.
- [16] Atmel. ATmega328/P datasheet complete. Folha de Dados. Disponível em: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf. Acesso: 25 de Maio de 2018.
- [17] Intel. Hexadecimal Object File Format Specification. Relatório técnico. Disponível em: https://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2012/ads264_mws228/Final%20Report/Final%20Report/Intel%20HEX%20Standard.pdf. Acesso: 25 de Maio de 2018.
- [18] Arduino. Internal Temperature Sensor. Site. Disponível em: <https://playground.arduino.cc/Main/InternalTemperatureSensor>. Acesso: 27 de Maio de 2018.
- [19] PRESSMAN, Roger. *Engenharia de Software: Uma abordagem profissional*. Livro. AMGH Editora Ltda, Porto Alegre, Brasil. 7º Edição, 2011.

- [20] NETO, Rodolpho Ugolini. Ferramenta SCRUM gratuita (free). Site. Disponível em: https://www.ibm.com/developerworks/community/blogs/rationalbrasil/entry/ferramenta_srum_gratu_c3_adta_free96?lang=en. Acesso: 30 de Outubro de 2018.
- [21] SUTHERLAND, Jeff. *SCRUM: A arte de fazer o dobro de trabalho na metade do tempo*. Livro. Leya, São Paulo, Brasil, 2014.
- [22] DELAMARO, M. E., MALDONADO, José Carlos, and JINO, Mario. *Introdução ao Teste de Software*. Livro. Campus, Rio de Janeiro, Brasil, 2007.
- [23] KONSTAS, Harry. AndMCU Documentation. Site. Disponível em: <https://sites.google.com/site/hkonstas/android-stuff/andmcu/andmcu-documentation>. Acesso: 04 de Agosto de 2018.
- [24] Documentação Android. dumpsys. Site. Disponível em: <https://developer.android.com/studio/command-line/dumpsys#meminfo>. Acesso: 06 de Agosto de 2018.

Apêndice A

Projeto *Blink*

```

1  /*
2   *      Blink
3
4   *      Turns an LED on for one millisecond, then off for one millisecond, repeatedly.
5
6   *      This code was modified by Kollins Lima (August 01, 2018).
7
8   *      The original code is available at:
9   *      http://www.arduino.cc/en/Tutorial/Blink
10 */
11
12 int output_pin = LED_BUILTIN;
13
14 // the setup function runs once when you press reset or power the board
15 void setup() {
16     // initialize digital pin LED_BUILTIN as an output.
17     pinMode(output_pin, OUTPUT);
18 }
19 // the loop function runs over and over again forever
20 void loop() {
21     digitalWrite(output_pin, HIGH);    // turn the LED on (HIGH is the voltage level)
22     delay(1);                      // wait for a millisecond
23     digitalWrite(output_pin, LOW);    // turn the LED off by making the voltage LOW
24     delay(1);                      // wait for a millisecond
25 }
```

Código A.1: Projeto *Blink*, utilizado para teste manual do frequencímetro, módulo de saída e como projeto base para comparação entre aplicativos.

Apêndice B

Projeto *Blink* em assembly

```

1 ; Blink Led - AndMCU
2 ; Kollins Lima - August 06, 2018
3
4 .autorun
5
6 .dump
7
8 ;. speed 1 ; Low Speed
9 ;. speed 255 ; High Speed
10
11 PORTA    equ 0
12 PORTB    equ 0x1
13 PORTC    equ 0x2
14 DDRA     equ 0x3
15 DDRB     equ 0x4
16 DDRC     equ 0x5
17
18 TIMER    equ 0xa
19 CONTROL  equ 0xb
20
21 .lcd "Blink Led"
22
23 init:
24
25     lda #255
26     sta DDRA ; port A output
27     lda #15
28     sta DDRC
29
30 start:
31
32 loop:
33
34     incx
35     stx PORTA
36
37     decx
38     stx PORTA
39
40     jmp loop
41 end

```

Código B.1: Projeto *Blink* em assembly para ser utilizado no simulador *AndMCU*.

Apêndice C

Projeto *Input to Output*

```

1  /*
2   *      Input to Output
3
4   *      Read signal from input and send it to output
5
6   *      This code was developed by Kollins Lima (August 05, 2018).
7 */
8
9 int input_pin = 8;
10 int output_pin = 9;
11 int val = 0;
12
13 void setup() {
14     pinMode(output_pin, OUTPUT);
15     pinMode(input_pin, INPUT);
16 }
17
18 void loop() {
19     val = digitalRead(input_pin);
20     digitalWrite(output_pin, val);
21 }
```

Código C.1: Projeto *Input to Output*, utilizado para teste manual do módulo de E/S.

Apêndice D

Projeto *Interrupt*

```

1  /*
2   *      Interrupt
3
4   *      Change state of output pin based on an interrupt event
5
6   *      This code was modified by Kollins Lima (August 05, 2018).
7
8   *      The original code is available at:
9   *      https://www.arduino.cc/reference/en/language/functions/external-interrupts/
10  *          attachInterrupt/
11
12 const byte ledPin = 9;
13 const byte interruptPin = 2;
14 byte state = LOW;
15
16 void setup() {
17     pinMode(ledPin, OUTPUT);
18     pinMode(interruptPin, INPUT_PULLUP);
19     attachInterrupt(digitalPinToInterruption(interruptPin), blink, CHANGE);
20 }
21
22 void loop() {
23     digitalWrite(ledPin, state);
24 }
25
26 void blink() {
27     state = !state;
28 }
```

Código D.1: Projeto *Interrupt*, utilizado para teste manual de interrupções externas e do resistor de *pull-up*.

Apêndice E

Projeto *Timer*

```

1  /*
2   *      Timer
3
4   *      Uses interruption caused by timer overflow to blink led
5
6   *      This code was developed by Kollins Lima (August 03, 2018).
7 */
8 const int ledPin = 9;
9
10 void setup()
11 {
12     pinMode(ledPin, OUTPUT);
13
14     // Timer configuration
15     /*
16         - Normal mode
17         - OC1A/OC1B disconnected
18         - No prescaling
19     */
20     TCCR1A = 0x00;
21     TCCR1B = 0x00;
22     TCCR1B |= (1<<CS10);
23
24     // Initial counting value
25     TCNT1 = 0x00FF;
26
27     // Enable overflow interruption
28     TIMSK1 |= (1 << TOIE1);
29 }
30
31 void loop()
32 {}
33
34 ISR(TIMER1_OVF_vect)
35 {
36     digitalWrite(ledPin, digitalRead(ledPin) ^ 1);
37     TCNT1 = 0x00FF;
38 }
```

Código E.1: Projeto *Timer*, utilizado para testar os temporizadores. O código mostrado está configurado para testar o *Timer1*.

Apêndice F

Projeto *Analog Input*

```
1 /*  
2      Analog Input  
3  
4      Read an analog value and display the result on PORTD (pin 0 – pin 7)  
5  
6      This code was developed by Kollins Lima (August 05, 2018).  
7 */  
8  
9 int sensorPin = A0;  
10 int sensorValue = 0;  
11  
12 void setup() {  
13     DDRD = 0xFF;  
14 }  
15  
16 void loop() {  
17     sensorValue = analogRead(sensorPin);  
18     PORTD = sensorValue;  
19 }
```

Código F.1: Projeto *Analog Input*, utilizado para testar o canal analógico.

Apêndice G

Projeto *Serial*

```

1  /*
2   *      Serial
3
4   *      Prints on the serial monitor whatever it receives as input.
5
6   *      This code was modified by Kollins Lima (August 05, 2018).
7
8   *      The original code is available at:
9   *      https://www.arduino.cc/en/Serial/Read
10 */
11
12 int incomingByte = 0;      // for incoming serial data
13
14 void setup() {
15     Serial.begin(9600);    // opens serial port, sets data rate to 9600 bps
16 }
17
18 void loop() {
19     // send data only when you receive data:
20     if (Serial.available() > 0) {
21         // read the incoming byte:
22         incomingByte = Serial.read();
23
24         // say what you got:
25         Serial.write(incomingByte);
26     }
27 }
```

Código G.1: Projeto *Serial*, utilizado para testar a USART e o monitor serial.

Apêndice H

Projeto *Analog Serial*

```

1  /*
2   * Analog Serial
3
4   * Print the decimal value of the analog conversion on serial monitor
5
6   * This code was modified by Kollins Lima (August 05, 2018).
7
8   * The original code is available at:
9   * http://www.arduino.cc/en/Tutorial/AnalogReadSerial
10 */
11
12 // the setup routine runs once when you press reset:
13 void setup() {
14     // initialize serial communication at 9600 bits per second:
15     Serial.begin(9600);
16     analogReference(EXTERNAL);
17 }
18
19 // the loop routine runs over and over again forever:
20 void loop() {
21     // read the input on analog pin 0:
22     int sensorValue = analogRead(A0);
23     // print out the value you read:
24     Serial.println(sensorValue);
25     delayMicroseconds(10);
26 }
```

Código H.1: Projeto *Analog Serial*, utilizado para testar a integração entre USART e canal analógico, bem como uma referência externa para o conversor A/D.

Apêndice I

Projeto *Memory Measure*

```

1  /*
2   *      Memory Measure
3
4   *      Print amount of free memory on the serial monitor.
5
6   *      This code was modified by Kollins Lima (August 05, 2018).
7
8   *      The original code can be found on the book "Arduino Cookbook", First Edition , by Mychael
9   *      Margolis .
9 */
10
11 void setup() {
12     Serial.begin(9600);
13 }
14
15 void loop() {
16     int test[100], i;
17
18     for(i = 0; i < 100; i++){
19         test[i] = 1;
20     }
21
22     Serial.print(memoryFree());
23     // print the free memory
24     Serial.print('\n');
25     delayMicroseconds(100);
26 }
27
28 // variables created by the build process when compiling the sketch
29 extern int __bss_end;
30 extern void *_brkval;
31
32 // function to return the amount of free RAM
33 int memoryFree(){
34     int freeValue;
35     if((int)_brkval == 0){
36         freeValue = ((int)&freeValue) - ((int)&__bss_end);
37     }
38     else{
39         freeValue = ((int)&freeValue) - ((int)_brkval);
40     }
41     return freeValue;
42 }
```

Código I.1: Projeto *Memory Measure*, utilizado para testar o medidor de memória.