UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ENGENHARIA DE SÃO CARLOS

DEPTO. DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

SOFIA - Um simulador Arduino baseado em Android

Autor: Kollins Gabriel Lima, n^o. USP 9012931

Orientador: Prof. Dr. Evandro Luis Linhari Rodrigues

São Carlos

2018

Kollins Gabriel Lima

SOFIA - Um simulador 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.

Resumo

Introduzido no mercado no ano de 2005, as placas de Arduino vem se popularizando cada vez mais devido à sua simplicidade e seu custo reduzido. 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 PC-Android, 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, consegue atender os objetivos propostos.

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 the 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 PC-Android 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, although still lacks in performance, it achieves its objectives.

Keywords: Arduino, Android, Simulator.

Lista de Figuras

Lista de Tabelas

Siglas

A/D	Analógico/Digital
CPU	*Central Process Unit* - Unidade Central de Processamento
CTC	*Clear Timer on Compare Match* - Limpar temporizador na correspondência de comparação
EEPROM	*Electrically-Erasable Programmable Read-Only Memory* - Memória apenas de leitura programável e apagável eletrônicamente.
E/S	Entrada/Saída
GVfs	*GNOME Virtual file system* - Sistema de arquivo virtual do GNOME
I^2C	*Inter-Integrated Circuit* - Circuito Inter-integrado
IDE	*Integrated Development Environment* - Ambiente de desenvolvimento integrado
MIPS	*Millions of Instructions Per Second* - Milhões de Instruções por Segundo
PC	*Program Counter* - Contador de Programa
PWM	*Pulse Width Modulation* - Modulação por Largura de Pulso
SDRAM	*Synchronous Dynamic Random-Access Memory* - Memória de acesso aleatório dinâmica síncrona
SPI	*Serial Peripheral Interface* - Protocolo Serial de Interface Periférica
TWI	*two-wire Serial Interface* - Interface serial de duas linhas
USART	*Universal Synchronous Asynchronous Receiver Transceiver* - Transmissor/Receptor Universal Síncrono e Assíncrono

Sumário

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]), desenvolvimento em conjunto com empresas interessadas (*Open Handset Alliance* - OHA) e o uso do Java para o desenvolvimento 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*), que oferece uma infinidade de aplicativos (de calculadoras a jogos com gráficos realistas), muitos deles disponibilizados gratuita-

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, Nwokorie [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 que serve como fonte de luz e permite o ajuste de intensidade do brilho por meio de botões, mostrando em um display LCD 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 Lin [8] que vai além do nível de aplicativo, fazendo modificações no kernel Linux para leitura de dados em um barramento CAN. Lin 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.

Procurando na *Google Play*, 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 ambientes de desenvolvimento integrado (IDE) 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 *MCU Prototype Board Simulator*,

que simula um kit de desenvolvimento (com botões, leds, LCD, etc.) e permite a execução de códigos assembly do microcontrolador 68705 da Motorola.

Quanto à simulação de placas de Arduino, apenas um aplicativo foi encontrado. O *CircSim Circuit Simulator* apresenta a interface de um simulador convencional de circuitos eletrônicos para PC. 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, pode-se citar ainda mais dois aplicativos com funções semelhantes à proposta neste trabalho. O primeiro, *BoardMicro - AVR Simulator*, não está diretamente ligado ao Arduino e destina-se à simulação do microcontrolador ATMega32U4 a partir de um arquivo hexadecimal fornecido pelo o usuário. Já o *Arduino Simulator Mini Free* é destinado à simulação do Arduino, no entanto possui limitações quanto ao código a ser simulado. Ambos os aplicativos estão disponíveis gratuitamente na *Amazon Store* e serão utilizados como referência para comparação com o presente trabalho, já que são os que mais se assemelham em funcionalidade.

Sendo assim, o presente trabalho surge também para preencher uma lacuna existente, dada a falta e a dificuldade em encontrar um aplicativo com as funcionalidades aqui propostas, além de, obviamente, oferecer mais uma possibilidade para explorar o Arduino, seja para estudantes, hobistas ou qualquer pessoa que se interesse pelo assunto.

1.1 Motivação

O Arduino é uma plataforma de *hardware* e *software* aberto, 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 - Internet das Coisas) 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 publico 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 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: Um dos mais completos encontrados. Possui recursos para montagem e simulação de circuitos eletrônicos analógicos e digitais, bem como desenvolvimento de layouts PCB. 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 à 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 entrada e saída [13].

- Autodesk Tinkercad: Certamente, a melhor ferramenta encontrada. É um ambiente de aprendizado online 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 *on-line*, sendo necessário fazer um cadastro para utilizá-la [14].

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

1.2 Objetivo

O objetivo principal do projeto é permitir a execução de programas desenvolvidos para a plataforma Arduino UNO em um aplicativo Android, permitindo assim a realização de testes sem a necessidade de uma placa de Arduino ou qualquer outro componente 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 modo automático.

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.

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, bem como algumas métricas de do projeto. Também é feita uma discussão a respeito dos resultados.

Capítulo 5: Conclui a respeito do trabalho desenvolvido até o momento e apresenta algumas perspectivas para a continuação do trabalho, bem como mostra o cronograma de atividades cumprido até o momento.

Capítulo 2

Embasamento Teórico

Neste capítulo será explicado o funcionamento e as características de cada módulo presente no microcontrolador do Arduino UNO (ATmega328P) e que foram implementados no simulador. Todas as informações foram retiradas da folha de dados do componente [15], exceto onde indicado.

2.1 Visão Geral

O ATmega328P é um microcontrolador RISC de 8-bits e arquitetura Harvard (memória de dados separada da memória de programa). Possui 28 pinos (encapsulamento PDIP), sendo 23 programáveis e pode tralhar 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;
- Conversor 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 (*2-wire Serial Interface*), compatível com I^2C da Philips;
- *Watchdog Timer* programável com oscilador separado;
- Entre outros.

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

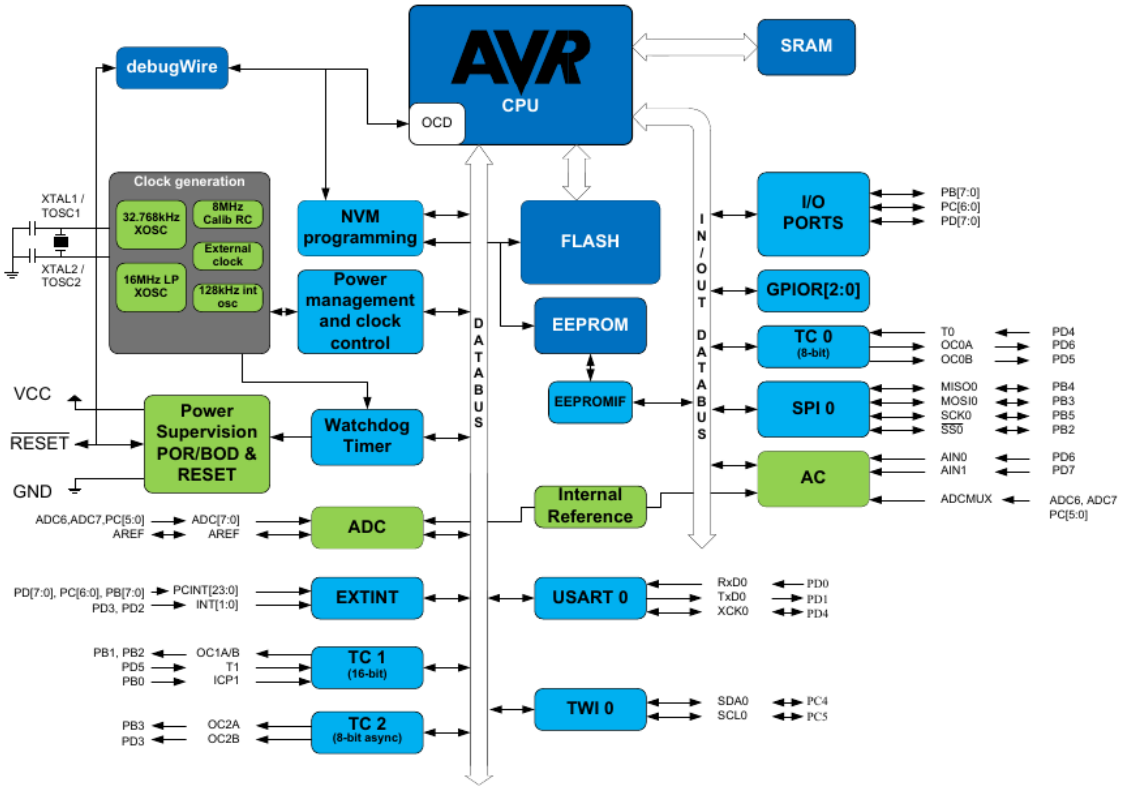


Figura 2.1: Diagrama de blocos da organização interna do ATmega328P

Fonte: Folha de dados ATmega328P

2.2 CPU

A CPU 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 de 14-bits; Registrador de *status* (8-bits) que armazenam 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 *pipeline* de um estágio o que, junto com a arquitetura Harvard, permite que o sistema atinja uma velocidade máxima de 1 MIPS/MHz.

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

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 *Watchdog Timer*
0x0E	TIMER2_COMPA	Comparação *Timer* 2 canal A
0x10	TIMER2_COMPB	Comparação *Timer* 2 canal B
0x12	TIMER2_OVF	Estouro do *Timer* 2
0x14	TIMER1_CAPT	Captura de evento *Timer* 1
0x16	TIMER1_COMPA	Comparação *Timer* 1 canal A
0x18	TIMER1_COMPB	Comparação *Timer* 1 canal B
0x1A	TIMER1_OVF	Estouro do *Timer* 1
0x1C	TIMER0_COMPA	Comparação *Timer* 0 canal A
0x1E	TIMER0_COMPB	Comparação *Timer* 0 canal B
0x20	TIMER0_OVF	Estouro do *Timer* 0
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

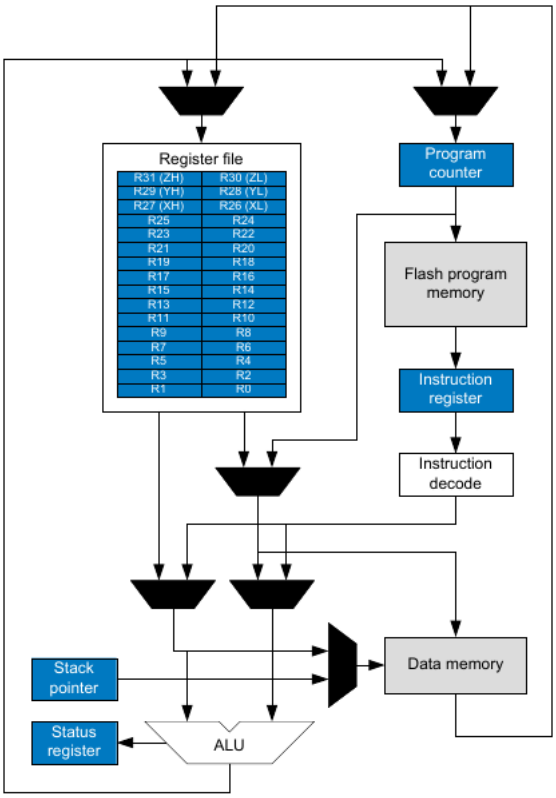


Figura 2.2: Diagrama de blocos da organização da CPU

Fonte: Folha de dados ATmega328P

Importante resaltar 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 envento e as disparadas por uma condição.

Quando as interrupções são disparadas por eventos, é habilitada uma *flag* indicando a ocorrência do evento. Se a interrupção estiver ativada para aquele evento, ela será tratada ou enfileirada para execução 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.

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 *flags* de modo que, se a condição for removida antes que a CPU possa tratar a interrupção correspondente, a interrupção não ocorrerá.

2.3 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.

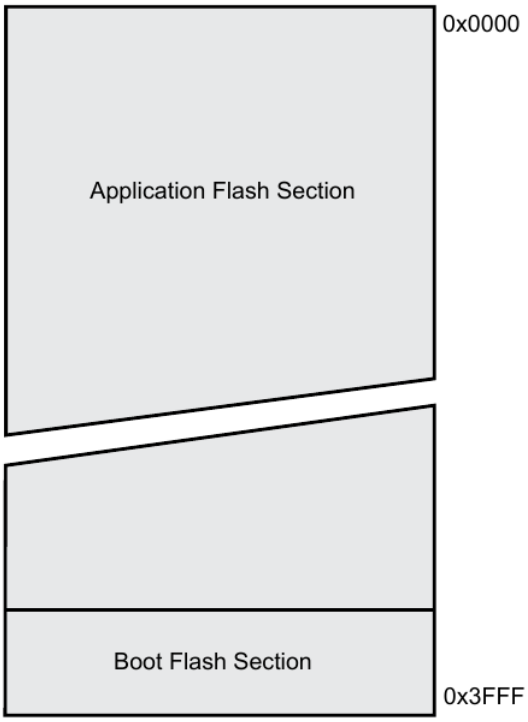


Figura 2.3: Memória de programa ATmega328P

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 [16]:

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

Início do registro	Contagem de bytes	Endereço	Tipo de registro	Dado	*Checksum*

- **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 dado 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 dado é carregado para os registradores CS e IP para os processadores 8086 e 80186
  - 04: Segmento estendido de memória Linear. O valor do campo dado 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).

2.4 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.

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;

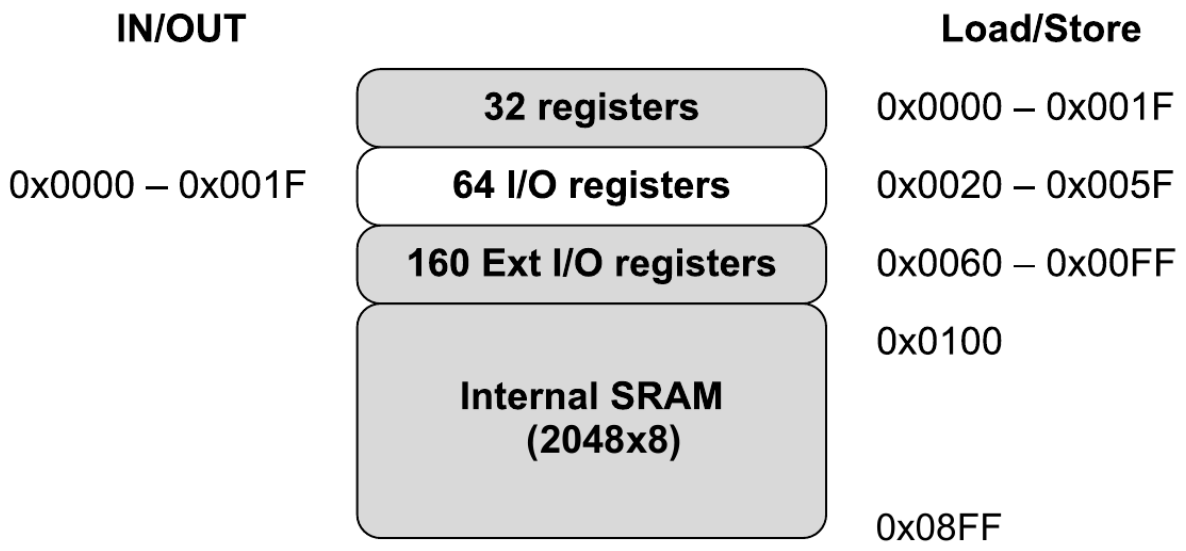


Figura 2.4: Memória de dados ATmega328P

Fonte: Folha de dados ATmega328P

- **Indireto com deslocamento**: Acesso à 63 endereços deslocados a partir do endereço base, dado pelo registrador 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 depois de serem utilizados como ponteiro para endereçamento.

2.5 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 entrada/saída (E/S) 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 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 PINxn, que é um registrador apenas de leitura responsável por armazenar o valor da entrada do pino. No entanto, é possível escrever um valor "1"(nível alto) neste registra-

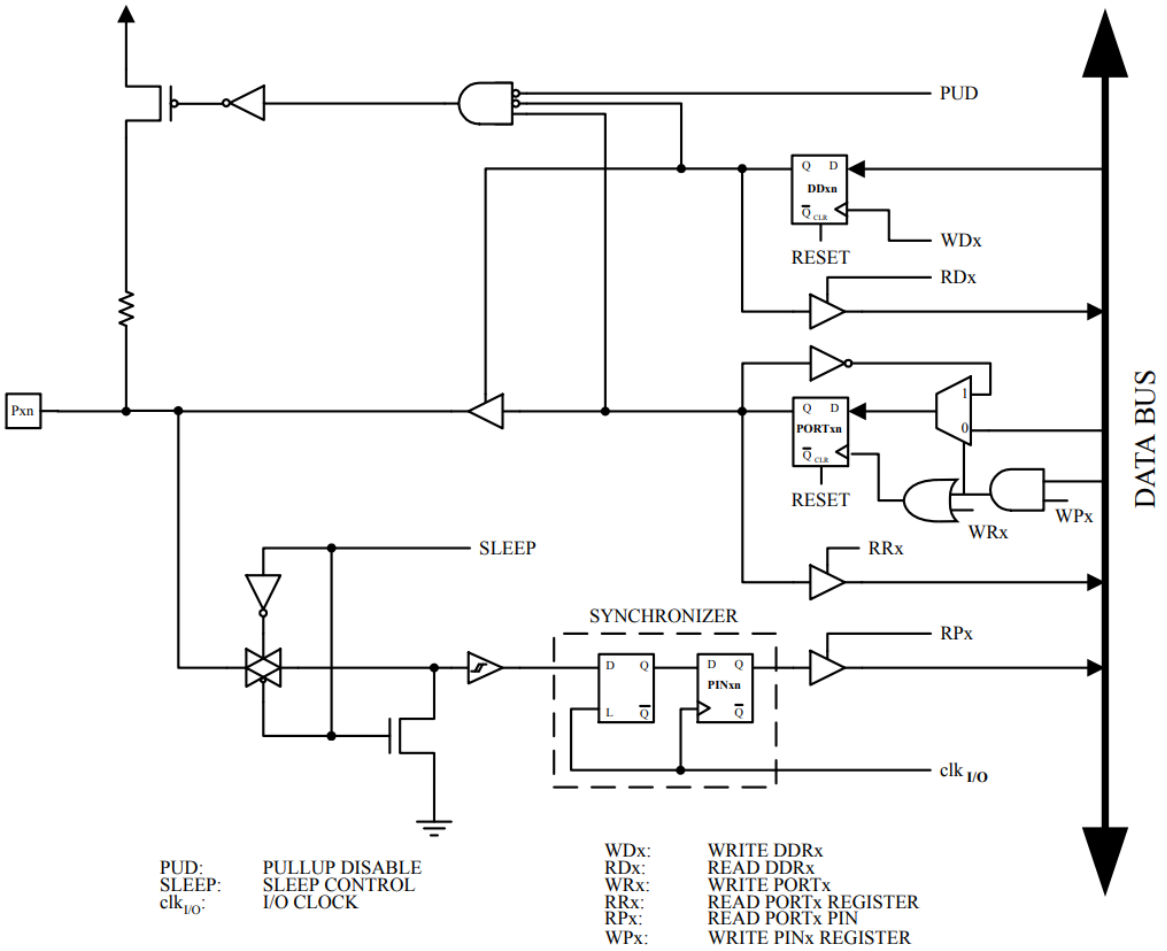


Figura 2.5: Organização do módulo de Entrada/Saída (E/S) do ATmega328P

Fonte: Folha de dados ATmega328P

dor. O efeito desta escrita será a inversão do valor contido no registrador PORTxn, 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.

2.6 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). Também por seu funcionamento assíncrono, 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, sobrescrevendo 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.

2.6.1 Modo Normal

No 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.

2.6.2 Modo CTC

O modo de funcionamento CTC 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 é mostrado na figura 2.9.

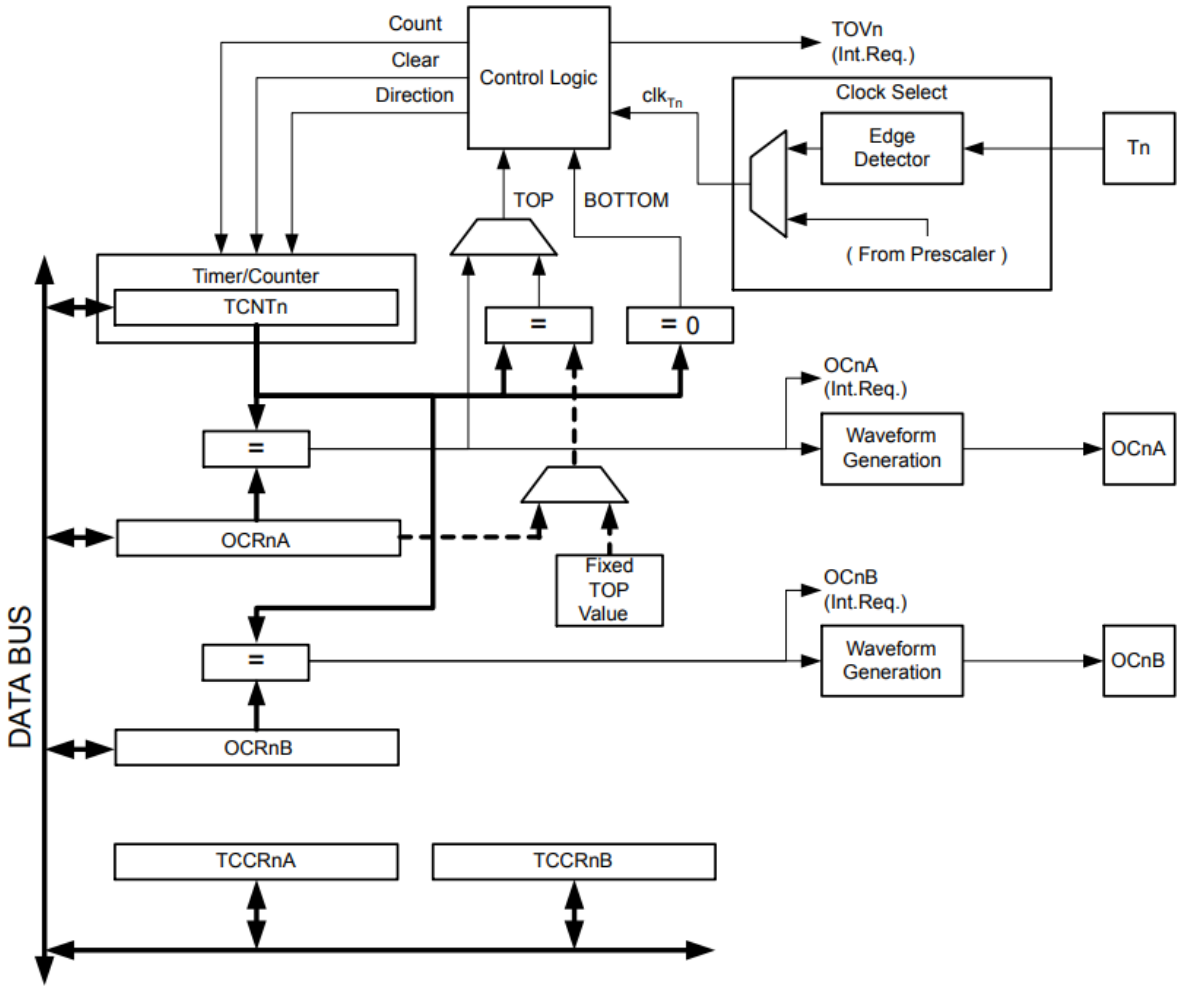


Figura 2.6: Organização do módulo de *Timer 0/2* do ATmega328P

Fonte: Folha de dados ATmega328P

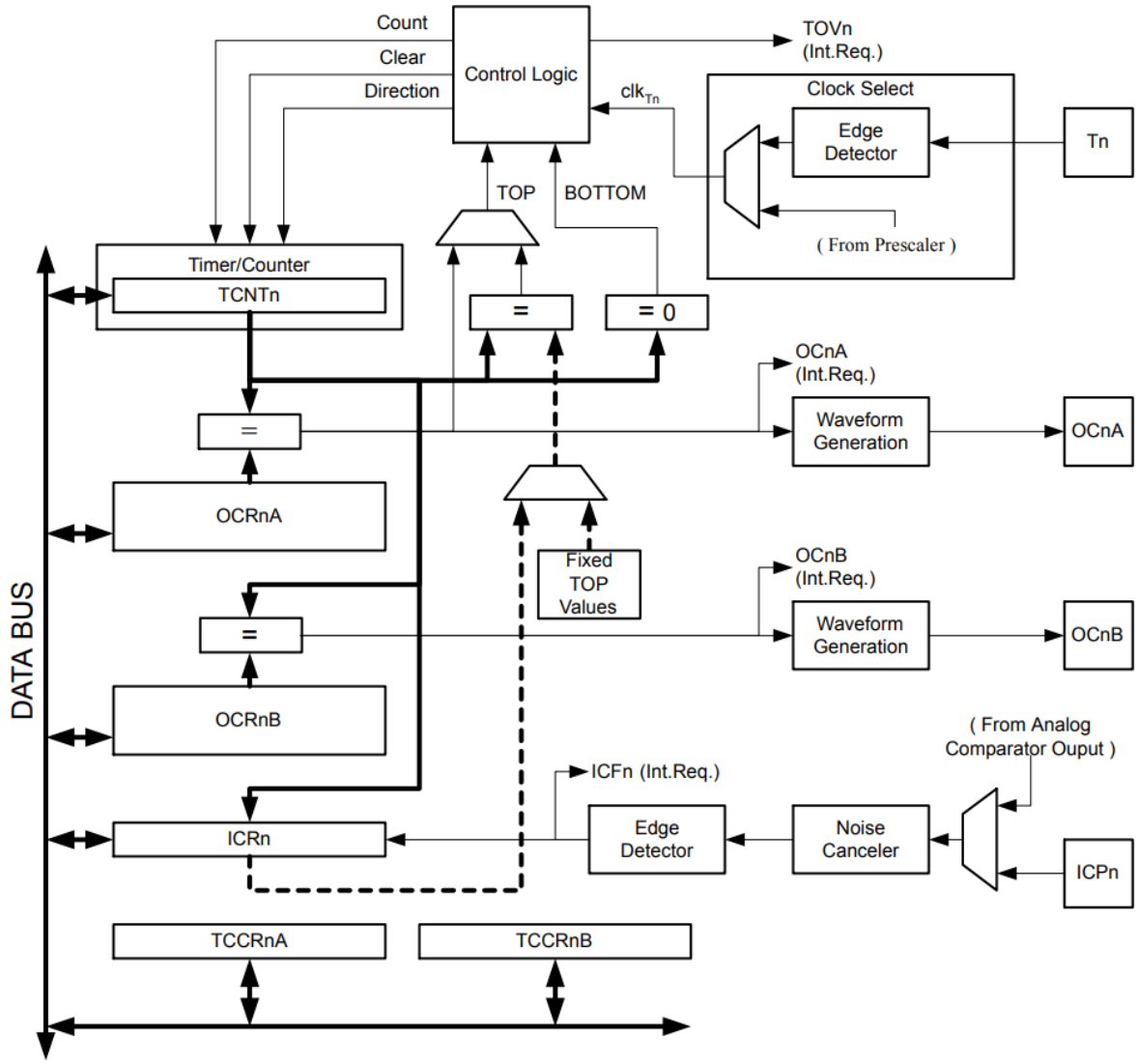


Figura 2.7: Organização do módulo de *Timer 1* do ATmega328P. Para contar em 16-bits, os registradores TCNTn, ICRn, OCRnA e OCRnB são divididos em dois registradores de 8-bits (*Low* e *High*).

Fonte: Folha de dados ATmega328P

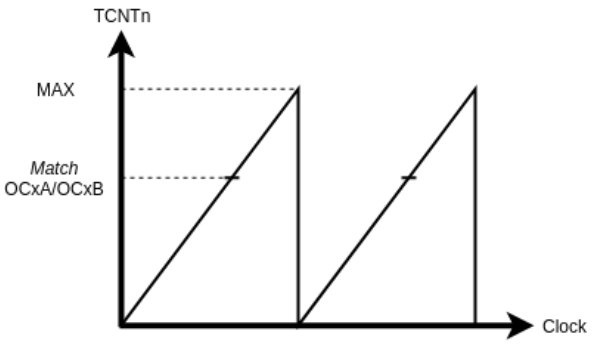


Figura 2.8: Diagrama de funcionamento do modo normal

Fonte: Autor

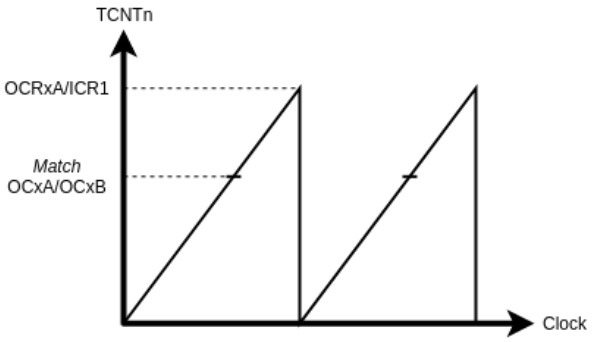


Figura 2.9: Diagrama de funcionamento do modo CTC

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.

2.6.3 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 *math* 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 *math* 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.

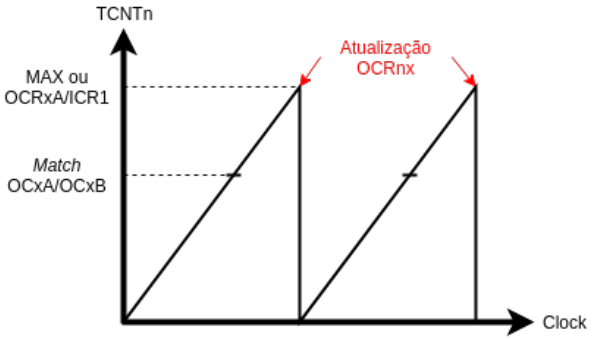


Figura 2.10: Diagrama de funcionamento do modo *fast PWM*

Fonte: Autor

2.6.4 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 atinga 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 *math* com OCRxA/OCRxB (na contagem progressiva) e nível alto em caso de *math* (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 em caso de *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 que a contagem atinge o valor máximo. Importante resaltar 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.

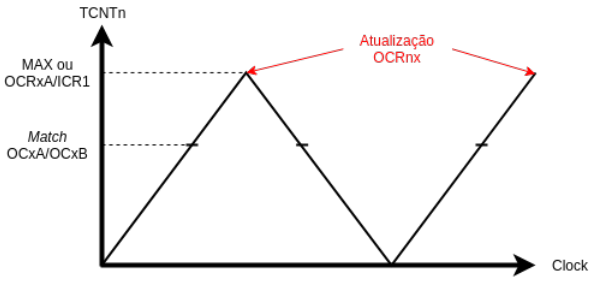


Figura 2.11: Diagrama de funcionamento do modo PWM com Correção de Fase

Fonte: Autor

2.6.5 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. Com isso, pode-se garantir a simetria dos pulsos gerados.

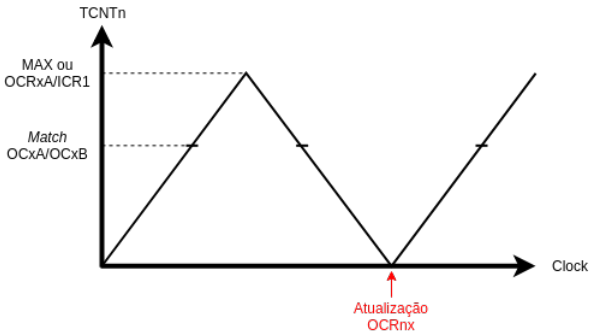


Figura 2.12: Diagrama de funcionamento do modo PWM com Correção de Fase e Frequência

Fonte: Autor

2.6.6 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.

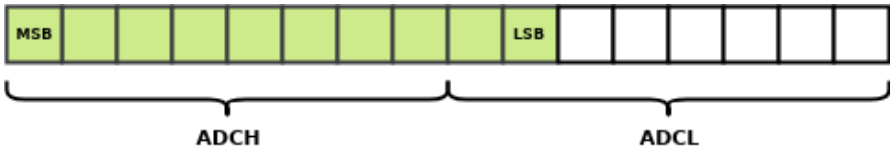
2.7 Conversor A/D

O ATmega328P possui um conversor analógico/digital (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 a entrada 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.

(a) Alinhamento à esquerda



(b) Alinhamento à direita

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.

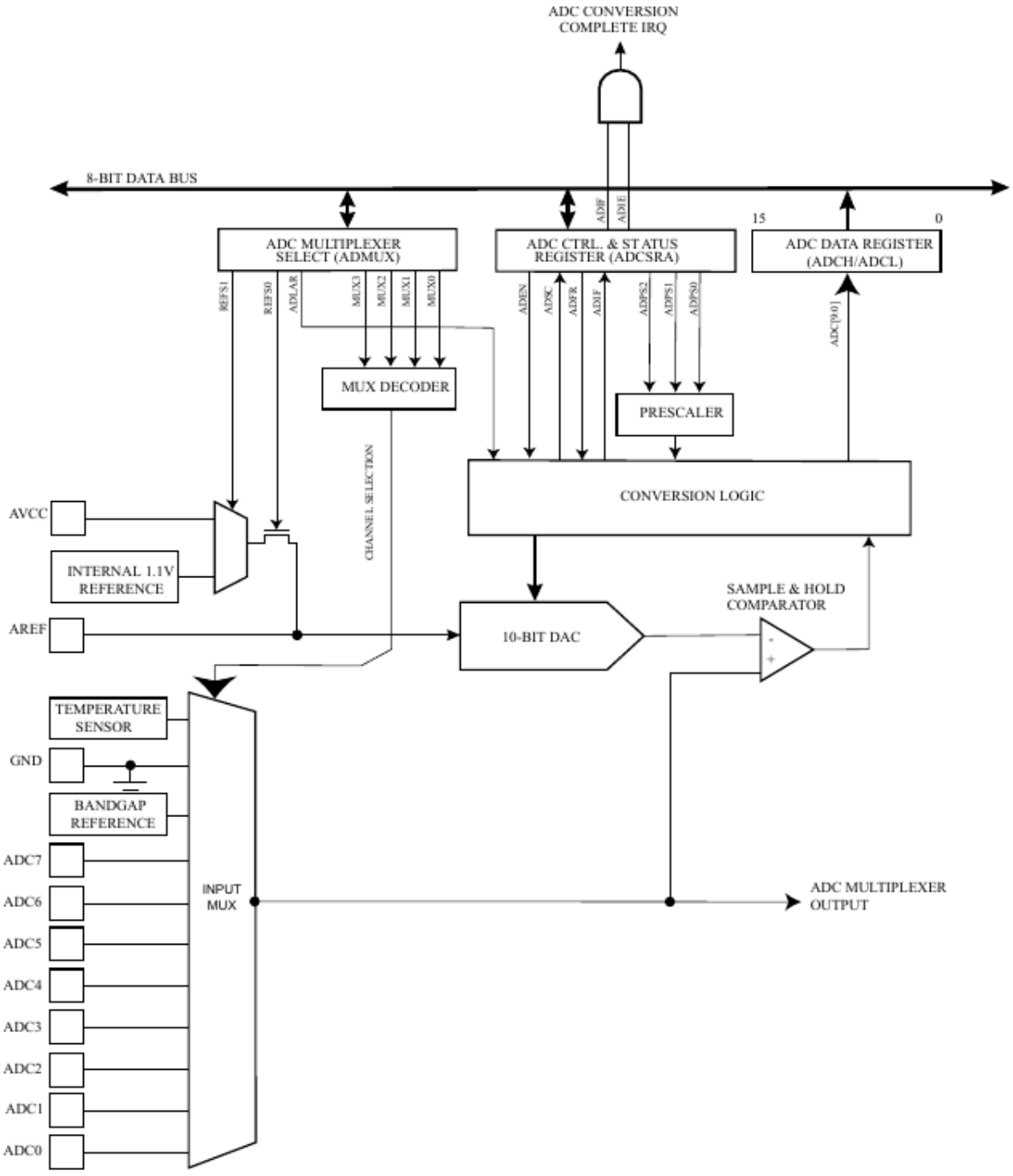


Figura 2.14: Organização do conversor A/D

Fonte: Folha de dados ATmega328P

$$ADC = \frac{V_{in} * 1024}{V_{ref}} \tag{2.1}$$

Valores de entrada superiores à V_{ref} terão valores convertidos próximos à 0x3FF (máximo valor 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 do conversor é feita 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 *Free Run*
001	Saída do comparador analógico
010	Interrupção externa 0
011	*Math* A *Timer* 0
100	*Overflow Timer 0*
101	*Math* B *Timer* 1
110	*Overflow Timer 1*
111	Captura de evento *Timer* 1

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°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.

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

Onde T_{OS} é um valor inserido na EEPROM de cada componente como parte dos testes em produção e k é um valor a ser determinado na calibração. (Na prática, utiliza-se T_{OS} = 324,31 e k = 1,22 [17])

2.8 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.

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. 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.

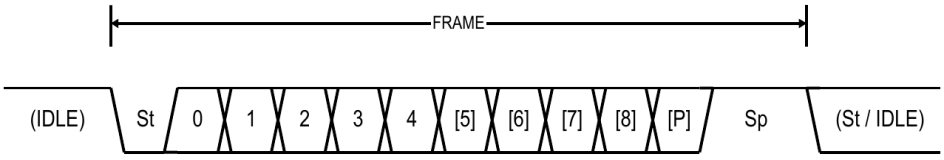


Figura 2.16: Formato de um *frame* transmitido pela USART.

Fonte: Folha de dados ATmega328P

Durante a transmissão e a recepção, a USART usa um registrador auxiliar UDRn (como mostrado na figura 2.15). Este registrador na verdade é 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 ser transmitido) 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 (*High* e *Low*) e pode ser calculada pela equação 2.3 (para o modo assíncrono).

$$BAUD = \frac{f_{osc}}{16(UBRRn+1)} \tag{2.3}$$

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.

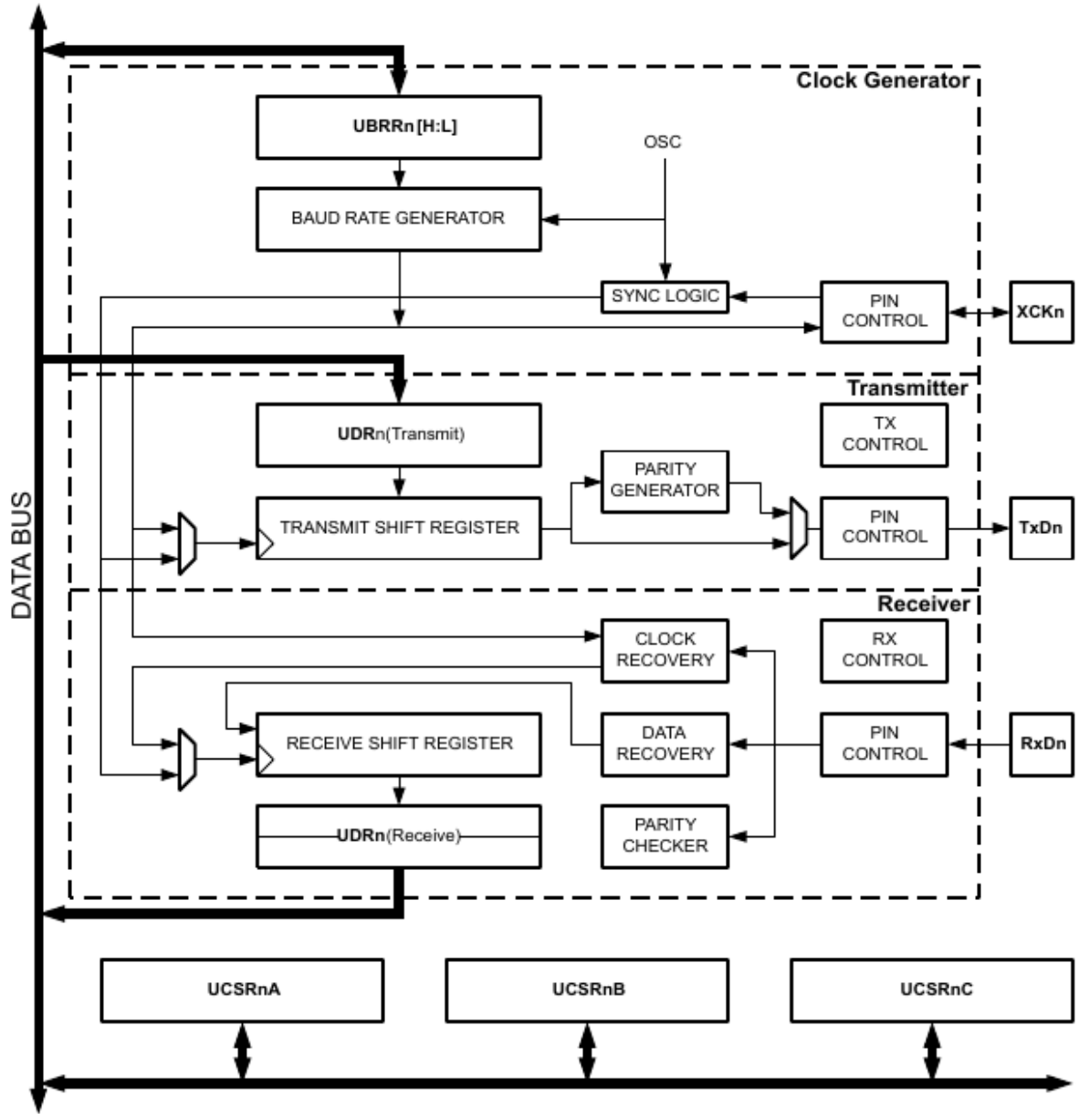


Figura 2.15: Organização da USART

Fonte: Folha de dados ATmega328P

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)

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* e *Draw.io*.
- Foi utilizado o método ágil *Scrum* para a construção do sistema. A plataforma *Taiga* foi utilizada para organização e planejamento dos *Sprints*.
- Para controle de versão foi utilizado o *Git* sincronizado à um repositório *on-line* no *Github*.
- Para documentação do projeto, foi utilizada uma página do *Gitbook*.
- Para o realizar modificações no código da IDE do Arduino, foi utilizado o *InteliJ IDEA* para escrever o código e o *Apache Ant* para a compilação. Também foi utilizado o *Inkscape* para a alteração no *design* (inserção do botão "Android").
- Para o desenvolvimento *mobile*, foi utilizada a IDE *Android Studio*.
- Para criar testes de unidade, foi utilizado o *JUnit4* em conjunto com o *PowerMock* (ambos utilizados como *plugin* do *Gradle*).
- O *SonarQube* foi utilizado para fazer a analise estática do código do simulador. Em conjunto, foi utilizado o *JaCoCo* para obter medidas de cobertura de código.

- Para montar códigos *Assembly* escritos para o ATmega328P, foi utilizado o *AVRA*.
- 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, no qual se fez uso também de um osciloscópio InfiniiVision DSOX2002A, da Keysight.

3.2 Método

3.2.1 Desenvolvimento na IDE do Arduino

A primeira parte do desenvolvimento ocorreu na IDE do Arduino. Não houve muito desenvolvimento nela além da 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.

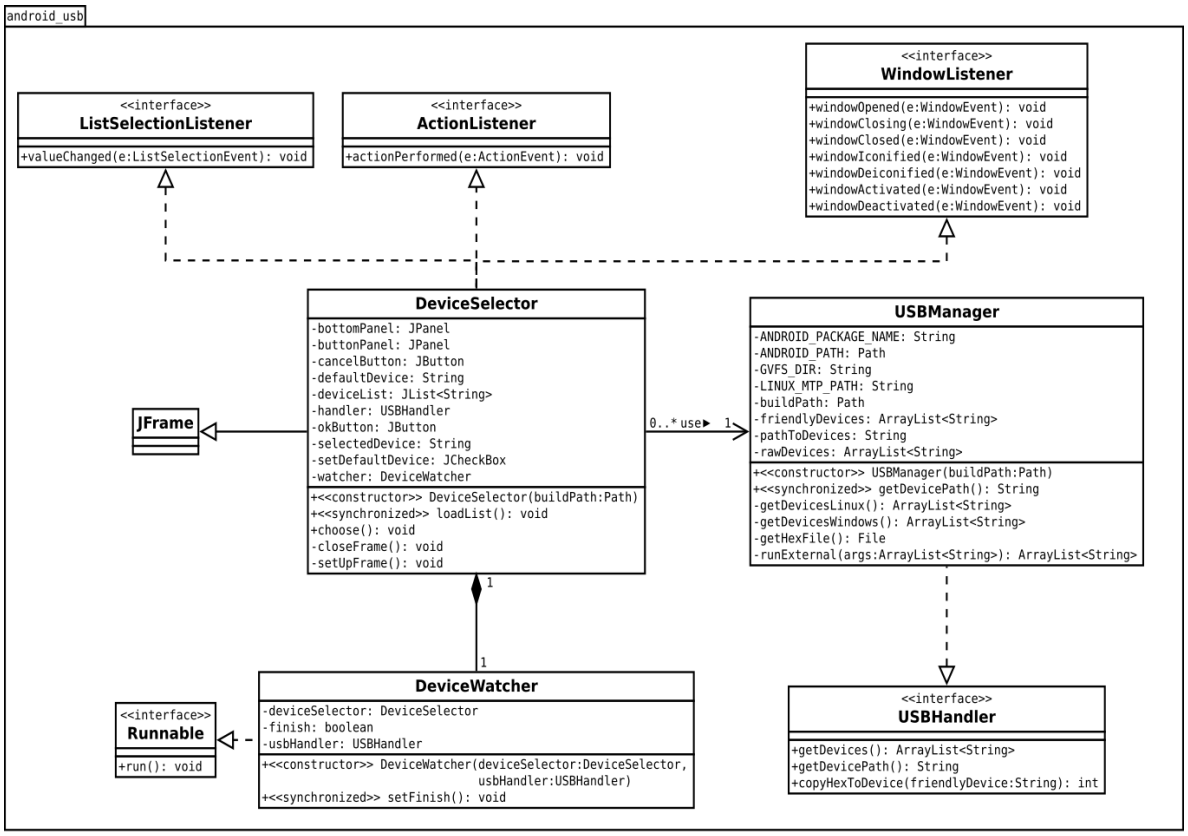


Figura 3.1: Diagrama de classes das modificações realizadas na IDE do Arduino

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 montados 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 são montados por meio do *GNOME Virtual file system* (GVfs). Utilizando a variável de ambiente *$XDG_RUNTIME_DIR* é posível acessar o dispositivo como uma pasta no sistema de arquivos. Desta forma, o arquivo .hex 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 .hex 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 comandos fazem parte do pacote básico de instalação das principais distribuições Linux.)

3.2.2 Desenvolvimento Android

3.2.2.1 Arquitetura

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.

Tudo tem início na classe *UCModule*. 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. Os módulos de CPU, *Timers*, conversor A/D são iniciados em uma nova *thread*, enquanto o módulo de interrupção é inicializado estaticamente.

A classe *UCModule* possui uma extensão que é a classe *UCModule_View*. Esta também é inicializada em uma *thread* e 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*). É

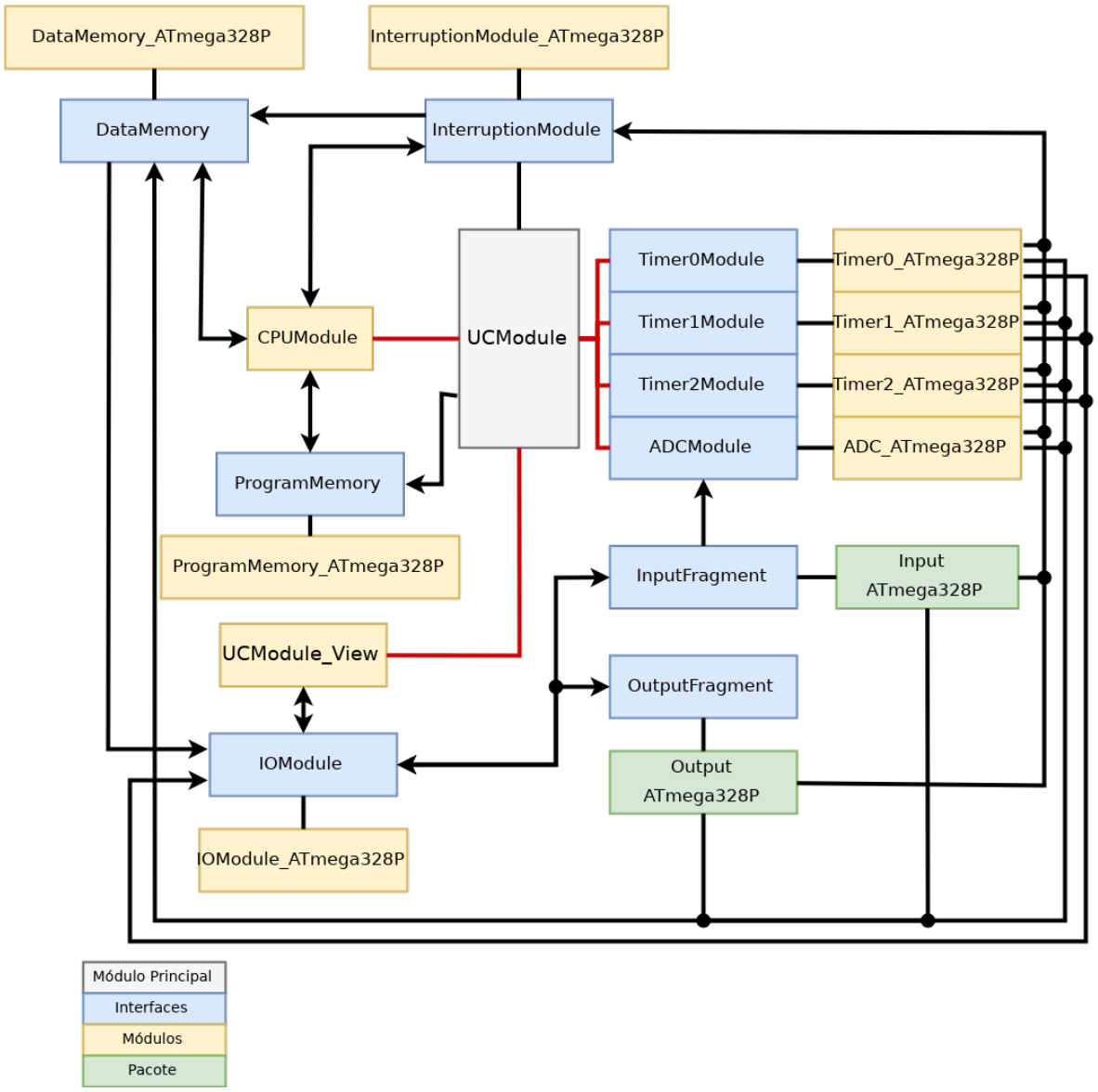


Figura 3.2: Arquitetura do simulador

Fonte: Autor

na *UCModule_View* que os módulos de entrada e saída são inicializados.

O módulo de entrada e saída 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. É nesta classe também que são verificados e tratados curto-circuito.

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 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.

A memória de dados (*DataMemory*) é o módulo responsável por armazenar todas as informações dos registradores e da memória SDRAM externa. Trata-se de uma classe cujos métodos de leitura e escrita estão sincronizados para evitar conflitos, já que ela é acessada de diferentes módulos rodando de maneira concorrente. Além disso, a memória de dados notifica a classe *IOModule* em caso de alteração nos registradores de E/S (PINx, PORTx e DDRx).

O módulo de interrupção (*InterruptionModule*) é responsável por receber requisições de interrupção dos módulos de *Timer*, E/S e conversor A/D, organizando-as em ordem de prioridade. Ela também é responsável por armazenar os endereços de desvio das interrupções e fornecer à CPU para a execução da rotina de interrupção. Este módulo pode ser acessado estaticamente por todas as classes do projeto, por meio da classe *UCModule*.

Os módulos de *Timer* (*TimerxModule*) são executados continuamente em *threads* próprias. Elas fornecem todos os modos de funcionamento dos apresentados na seção 2.6, com exceção do funcionamento assíncrono para o *Timer* 2.

O conversor A/D (*ADCModule*), assim como os *Timers*, executa continuamente em uma *thead* e apresenta todos os módulos de funcionamento descritos na seção 2.7, com exceção do sensor de temperatura e do disparo pela saída do comparador analógico (já que este módulo não foi implementado).

A figura 3.2 mostra que as comunicações ocorrem sempre por meio de interfaces (exceto para envio de dados, que precisam partir dos módulos). Esta configuração isola toda a parte de controle do funcionamento dos módulos específicos do ATmega328P. Isso 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).

3.2.2.2 Estratégias de implementação

Uma vez apresentada a arquitetura do sistema, serão detalhadas a seguir as principais estratégias de implementação utilizadas no simulador.

3.2.2.2.1 *Clock*

Não foi implementada uma fonte de *clock* para o funcionamento do simulador. Toda a sincronia dos módulos é feita por meio de variáveis de condição.

Cada *thread* (correspondente à um módulo) executa sua função dentro de uma estrutura de repetição. A cada iteração, a *thread* é bloqueada em uma variável de condição e fica a espera dos demais módulos. Quando todos os módulos terminarem suas tarefas referentes aquele ciclo de *clock*, o último módulo envia uma mensagem ao módulo principal, que envia um sinal a cada *thread*, desbloqueando-as.

Esta estratégia mantém a sincronização ao mesmo tempo que não limita a velocidade de execução. Um ponto negativo desta abordagem é que a velocidade de execução do programa não é fixa, podendo acelerar ou desacelerar em função da carga no sistema (externo ao aplicativo).

Também referente ao *clock*, foi preservado o número correto de ciclos que cada instrução da CPU leva para executar, bem como o número de ciclos para a conversão A/D (para o conversor A/D, foi adotado um número fixo de 13 ciclos de clock, apesar de este número variar conforme o modo de operação e para a primeira conversão).

3.2.2.2.2 Memória de Programa

Como dito na seção 2.3, a memória de programa do ATmega328P é organizada em 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 lê 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*.

3.2.2.2.3 Memória de Dados

Alguns registradores são tratados de maneira diferentes no ATmega328P. Pode-se citar, por exemplo, o registrador PINx. Este, como explicado na seção 2.5, apesar de ser um registrador de leitura, permite também a escrita de um valor. Esta escrita, no entanto, 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.

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.

3.2.2.2.4 Leitura e Escrita 16-bits

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 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.

3.2.2.2.5 Decodificação de instruções

As instruções da arquitetura trabalhada não fornece campos com valores fixos para *opcode*, operadores, etc., o que dificulta a decodificação das instruções.

Pensando em velocidade e também manutenibilidade, a estratégia adotada para a decodificação das instruções foi a utilização de um banco de dados com as instruções já 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 (apenas o vetor de instruções consome 128kB de memória), esta solução evita o uso de condicionais para decodificar instruções, o que proporciona um ganho de desempenho e torna o código mais legível.

3.2.2.2.6 *Prescaler*

O uso de *prescaler* foi restringido apenas aos *Timers*, já que este é um recurso importante para controlar a temporização. Para o conversor A/D e para o sistema em geral o *prescaler* foi desabilitado pois, para o simulador, seu efeito não é relevante, além de este prejudicar bastante o desempenho do simulador.

3.2.2.2.7 Entrada e Saída

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, será adotado 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 for habilitado).

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.

4.1 Arduino IDE

A figura 4.1 mostra o 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.

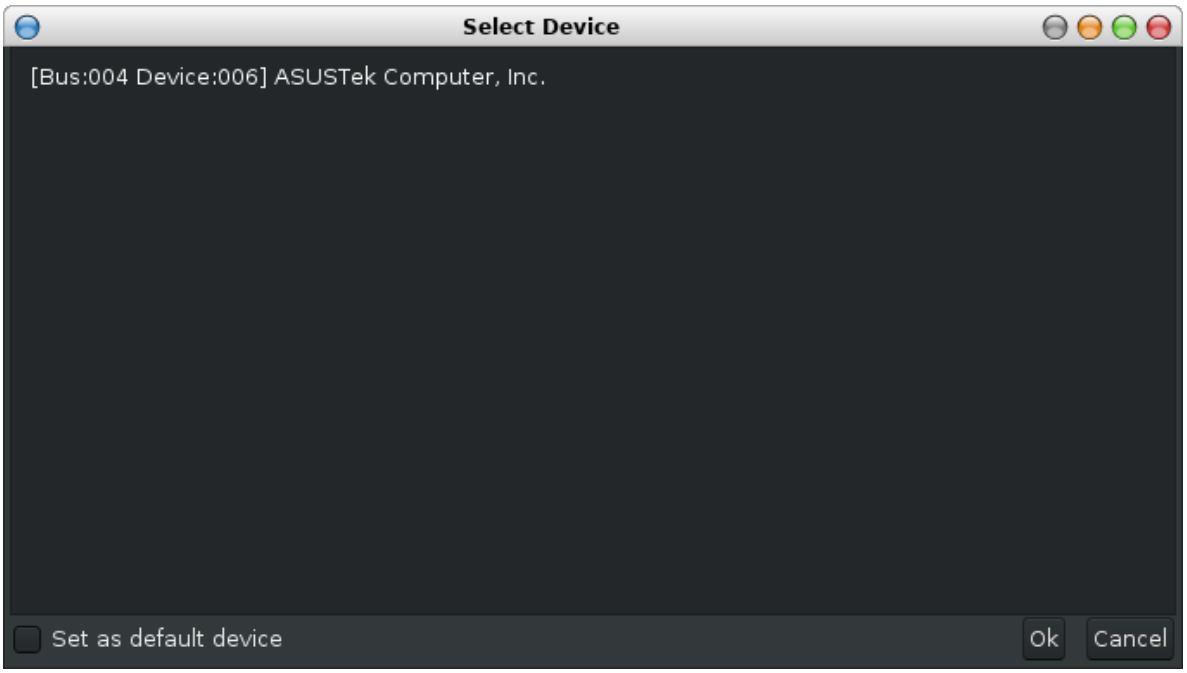


Figura 4.2: Seletor de dispositivos.

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.

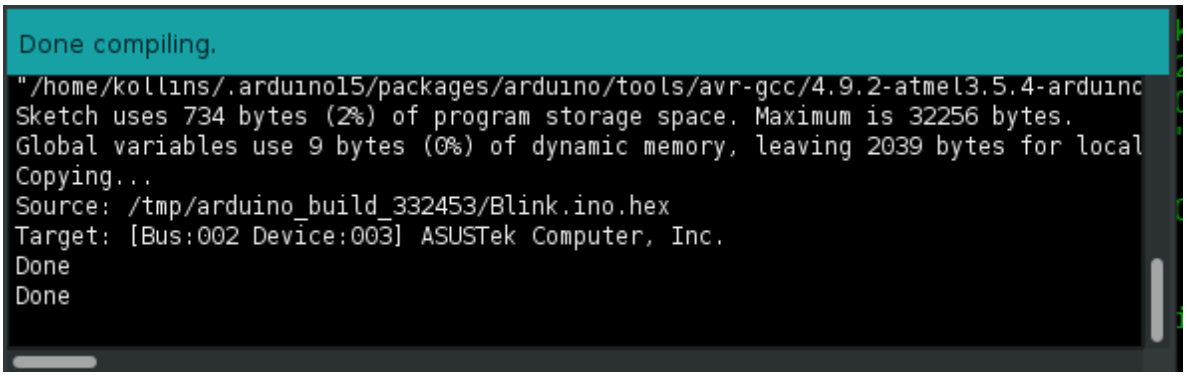


Figura 4.3: Cópia do arquivo realizada com sucesso para o *smartphone*

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.

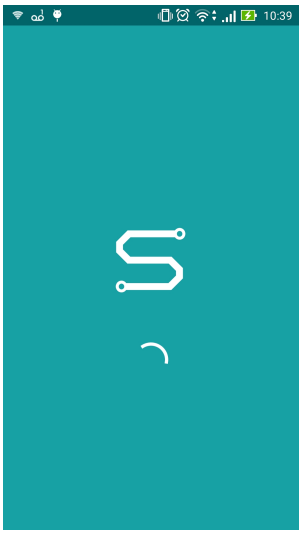


Figura 4.4: *Splash Screen* exibida ao abrir o simulador.

Fonte: Autor

O usuário é então redirecionado para a tela inicial do simulador, mostrada na figura 4.5

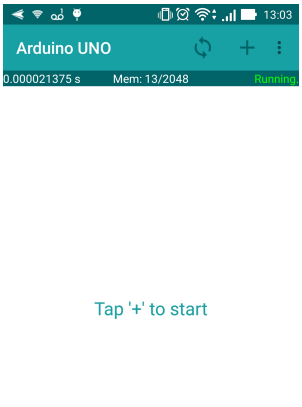


Figura 4.5: Tela inicial do simulador.

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, como mostra a figura 4.6. Nesta *toolbar*, o usuário tem acesso às seguintes opções:

- Modelo simulado: Mostra qual placa de Arduino está sendo utilizada para a simulação;
- Botão *Reset*: Permite o *reset* manual do sistema.
- Adicionar Entrada/Saída: Permite adicionar uma entrada/saída digital ou uma entrada analógica.

Figura 4.6: *Toolbar* do sistema

Fonte: Autor

- Outras opções: Contém funções adicionais para importar código do *smartphone*, remover todas as entradas/saídas e acesso à informações do projeto.

Além disso, o usuário pode visualizar:

- Tempo simulado: Exibe tempo simulado do sistema, baseado no cristal de 16MHz e atualizado a cada pulso de *clock* interno do sistema.
- Medidor de memória: Mostra o consumo de memória (em bytes) do programa em execução.
- Status: Mostra o *status* da simulação.

Caso não seja encontrado um arquivo hexadecimal ou ocorra alguma falha na abertura, o usuário deve ver uma mensagem na barra de *status* informando o problema, como mostrado na figura 4.7.

4.3 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 4.8. 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 4.8 mostra os três estados possíveis para uma saída no simulador.

Uma entrada pode ser digital ou analógica. Uma entrada digital é mostrada na figura 4.9. Pode-se observar que existem 4 elementos em uma entrada digital, são eles (da esquerda para a direita):

Figura 4.7: Falha ao abrir arquivo hexadecimal

Fonte: Autor

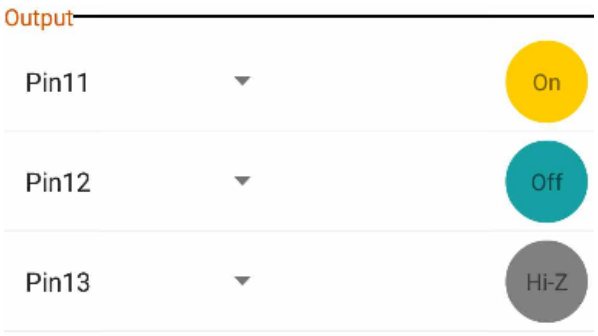


Figura 4.8: Saídas digitais do simulador

Fonte: Autor

Figura 4.9: Entradas digitais do simulador.

Fonte: Autor

- Botão: por onde o usuário envia sinais ao o sistema.
- Seletor de modo: define a operação do botão, podendo ser:
  - Push-GND: envia nível baixo se pressionado, indefinido caso contrário (é a opção default).
  - 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 (conforme mostra a figura 4.10) e a simulação para, podendo ser reiniciada manualmente.

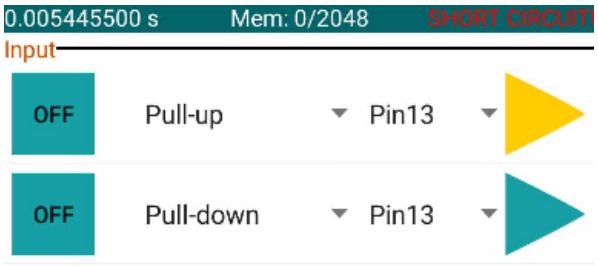


Figura 4.10: Condição de cuito-circuito entre entradas.

Fonte: Autor

Uma entrada analógica é mostrada na figura 4.11. Ela possui um seletor de pino, uma barra deslizante e um voltímetro, indicando o valor de tensão enviado ao Arduino. Assim como na entrada digital, nenhuma entrada está selecionada por padrão.

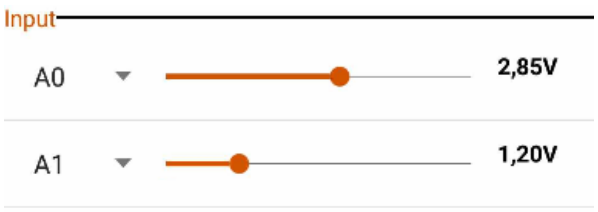


Figura 4.11: Entradas analógicas do simulador.

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 4.12. Alternativamente, pode-se utilizar a opção "*Clear I/O*"da *toolbar* para remover todas as entradas e saídas.

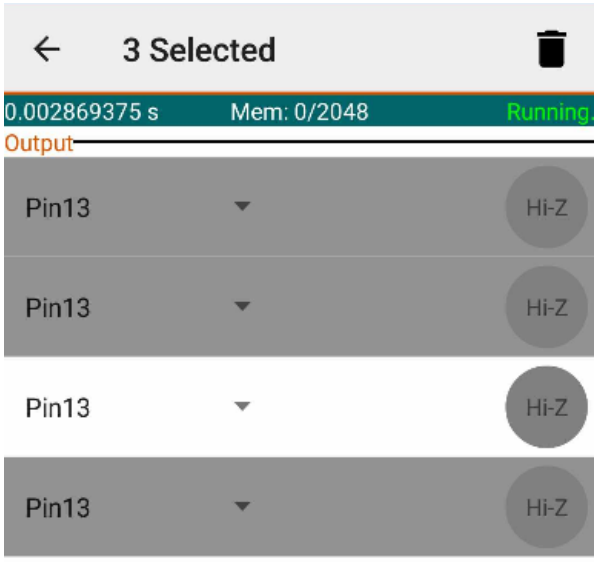


Figura 4.12: Remoção manual de pinos de saída.

Fonte: Autor

4.4 Métricas de *Software*

4.4.1 Cobertura

Foram criados testes de unidade para os módulos de CPU e memória de dados. Estes módulos foram escolhidos por serem os mais importantes para o funcionamento da simulação.

No total, foram criados 356 casos de teste cobrindo todas as instruções da CPU e as operações de leitura e escrita na memória (considerando todos os casos especiais). O foco dos testes foi a verificação de valores limite (operações envolvendo *overflow* ou *underflow*) e verificação das *flags* do registrador SREG.

As figuras 4.13, 4.14 e 4.15 mostram os resultados de cobertura dos testes para o projeto, CPU e memória de dados respectivamente. Como se pode ver pela figura 4.13, os casos de teste cobrem cerca de 19% de todo o projeto e a CPU é coberta quase que totalmente, como mostra a figura 4.14.

Figura 4.13: Cobertura dos testes para o projeto.

Fonte: Autor

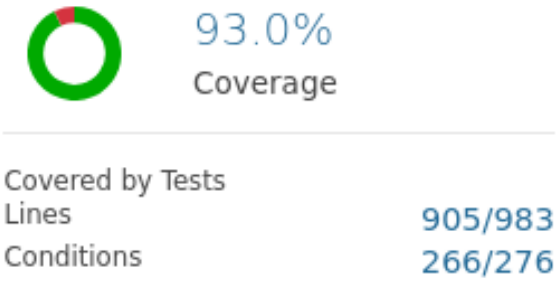


Figura 4.14: Cobertura dos testes para o módulo de CPU.

Fonte: Autor

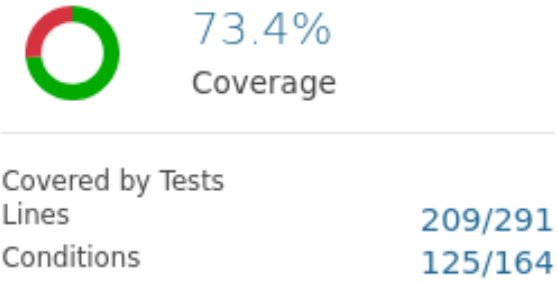


Figura 4.15: Cobertura dos testes para o módulo de memória de dados.

Fonte: Autor

4.4.2 Análise Estática

Com o uso do SonarQube, foi realizada a analise estática do código com o objetivo de encontrar pontos de melhoria no código. 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 4.16 mostra um gráfico relacionando cada classe do projeto com a quantidade de *bugs* encontrados (tempo para correção dos defeitos). Pode-se observar que uma classe se destaca pela sua quantidade de *bugs*, esta classe é a *DataMemory_ATmega328P*.

É importante ressaltar que, por ser uma ferramenta de análise estática, muitos problemas apontados

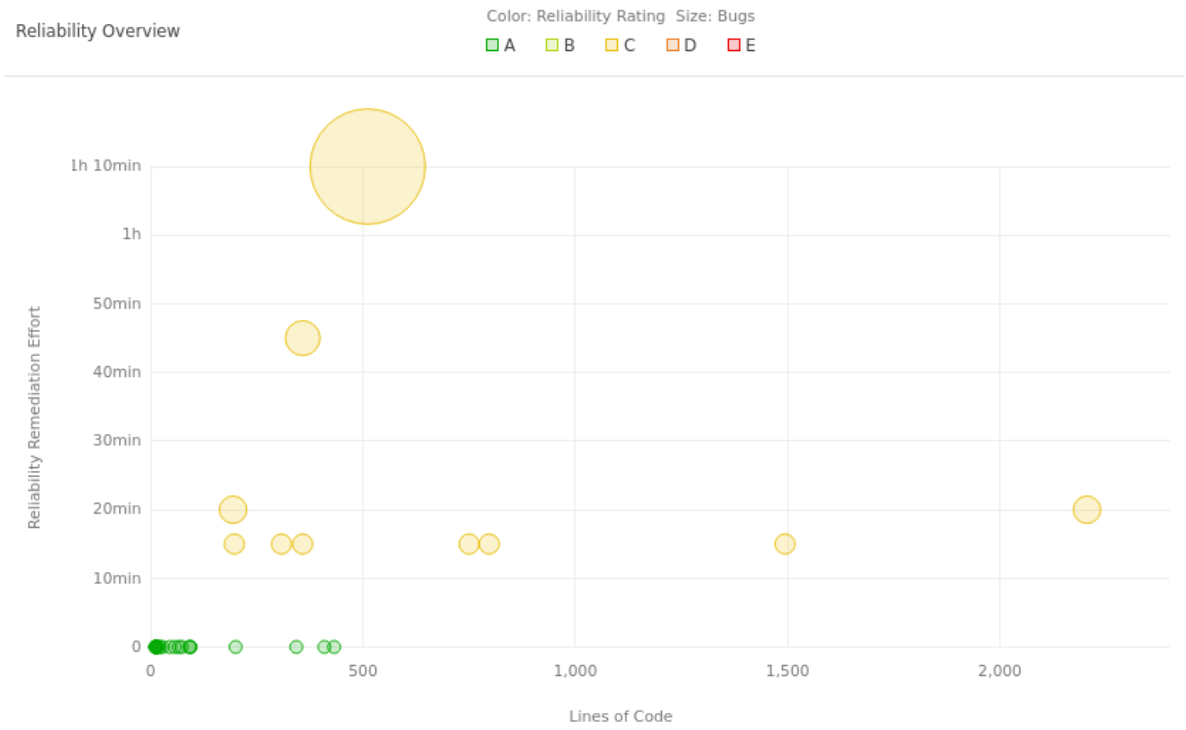


Figura 4.16: Gráfico linhas de código x esforço para resolução dos *bugs*.

Fonte: Autor

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*, mas mesmo assim o SonarQube não os reconhece) e quanto à conversão de dados de *int* para *byte* (os casos de teste criados com o JUnit4 estão testando estas conversões). Devido à estes *bugs*, o sistema foi classificado com o *ranking* D em termos de confiabilidade.

Outra métrica fornecida diz respeito à segurança do sistema e se baseia na quantidade de vulnerabilidades encontradas. A figura 4.17 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, esta é a *UCModule*.

Em termos de vulnerabilidades, o sistema apresenta melhores resultados em comparação aos *bugs*. Das que não foram resolvidas, sobraram vulnerabilidades relacionadas à manipulação de variáveis estáticas. Muitas destas, no entanto, são variáveis privadas, de forma que não existem grandes problemas relacionado com o acesso delas por classes externas. 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 compexidade cognitiva dos métodos e quanto ao uso devido/indevido de padrões de codificação (esses são chamados *Code Smells*). A figura 4.18 apresenta um gráfico relacionando cada classe do

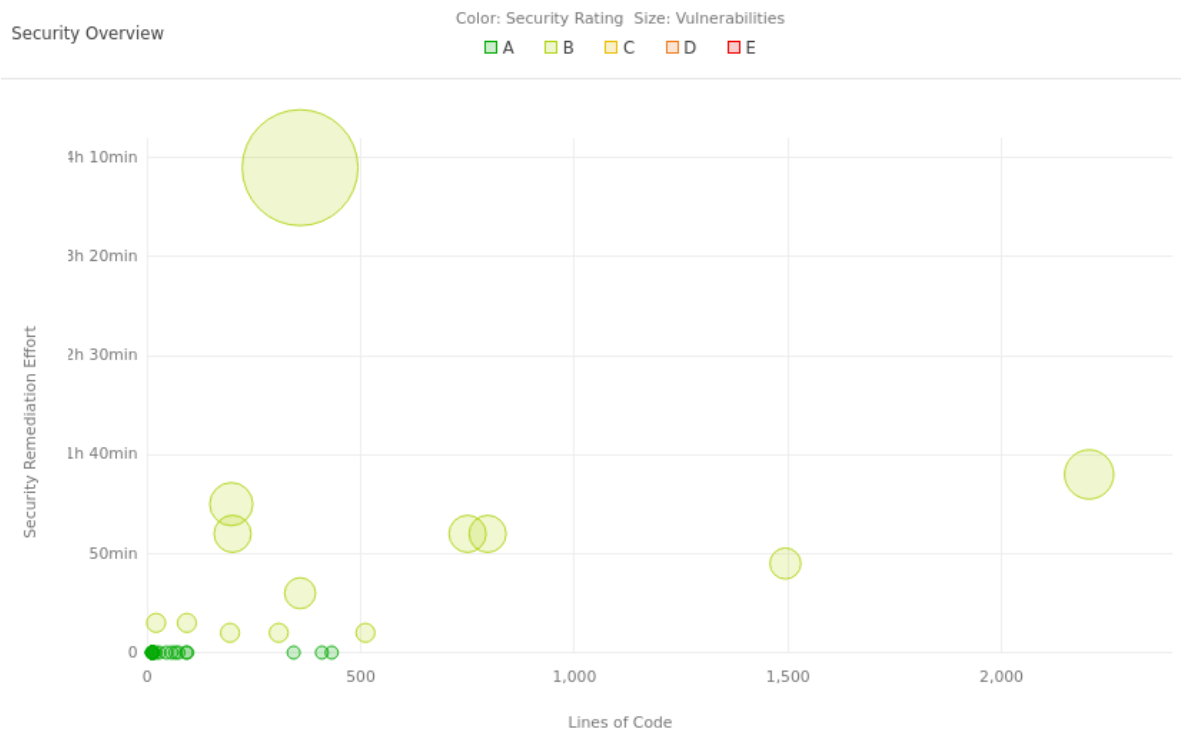


Figura 4.17: Gráfico linhas de código x esforço para resolução das vulnerabilidades.

Fonte: Autor

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*.

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, e portanto nenhuma sugestão desta seção foi aplicada no código até o momento. Apesar disso, o sistema foi classificado com *ranking* A em termos de manutenibilidade.

Também foi obtida uma métrica de código duplicado. A figura 4.19 apresenta um gráfico relacionando cada classe do projeto com a quantidade de linhas de código duplicadas.

As três classes com maior número de código duplicado são as classes referentes aos *Timers*. Os modos de operação dos *Timers* se 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). Desta forma, foram criados diferentes métodos, cada um com pequenas diferenças de modo a atender o modo de operação configurado.

Por ter um grande número de modos de operação, a classe com mais código duplicado (em destaque no gráfico) é a classe *Timer1_ATmega328P*.

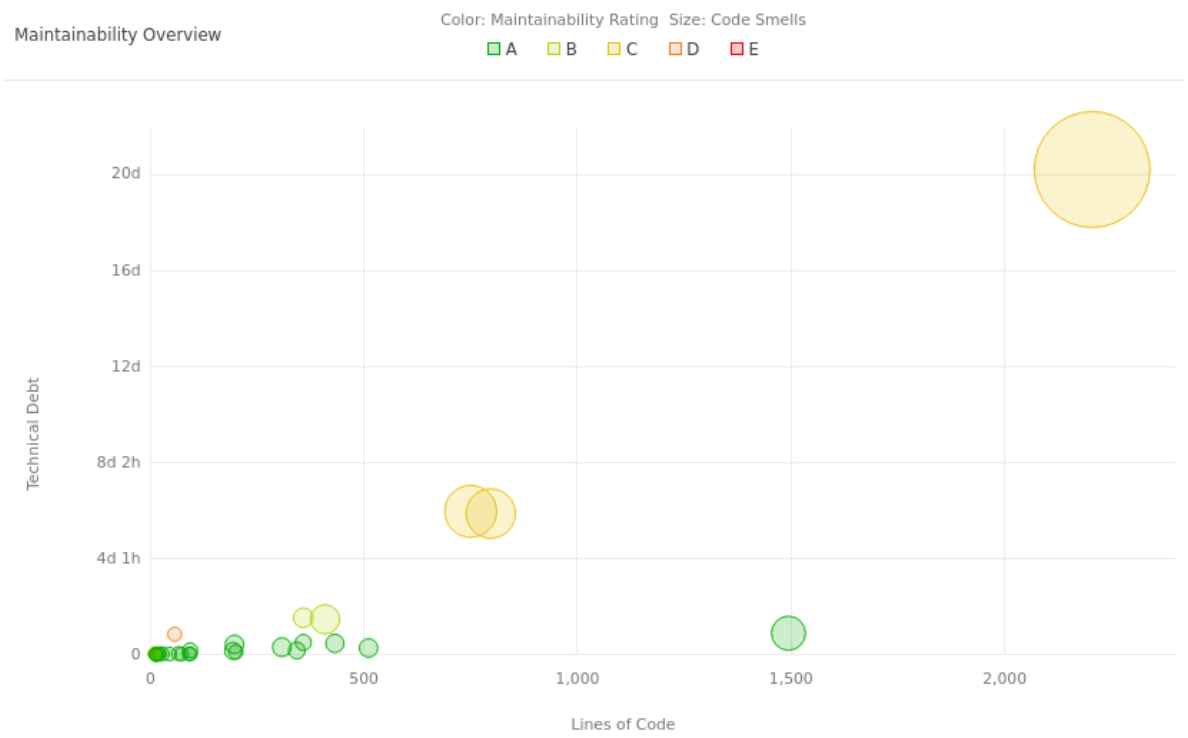


Figura 4.18: Gráfico linhas de código x esforço para resolução dos *Code Smells*.

Fonte: Autor

A última métrica importante de ser mencionada é a de complexidade ciclomática. Esta é uma media importante pois diz qual é o número mínimo de casos de teste necessários para que eles cubram todo o projeto. O valor obtido para complexidade ciclomática foi de 1.802.

4.4.3 *Profiling*

Foi utilizada a ferramenta de *profilling* do próprio Android Studio para avaliar o consumo de recursos do aplicativo e tentar identificar pontos de otimização. Para isso, utilizou-se o projeto *Blink* apresentado no Apêndice A. A figura 4.20 mostra o desempenho do sistema quanto ao uso de CPU e memória. Pode-se observar pela figura 4.20 que o uso de CPU fica entre 10 e 20%, enquanto que o uso de memória está entre 25MB e 35MB.

Olhando especificamente para a CPU, observou-se dois pontos críticos no sistema: um deles é o mecanismo de sincronização entre as *threads* e o outro é a atualização dos elementos visuais (tempo simulado e uso de memória) pela *UCModule_View*, como mostrado na figura 4.21.

O primeiro ponto identificado é de difícil otimização. As funções de *clock* mostradas na figura 4.21 fazem uso de *locks* para manter a sincronia entre as *threads* e este é um mecanismo necessário. O código 4.1 mostra o método de *clock* da CPU. Esta mesma estrutura se repete para o *clock* dos demais módulos.

Figura 4.19: Gráfico linhas de código x linhas de código duplicadas.

Fonte: Autor

```
1 public void clockCPU() {
2     clockLock.lock();
3     try {
4         cpuClockCondition.signal();
5     } finally {
6         clockLock.unlock();
7     }
8 }
```

Código 4.1: Método *clockCPU*.

Para poder otimizar este código, seria necessário repensar a maneira como os módulos são sincronizados, o que envolve uma mudança na arquitetura do simulador. Também seria possível pensar em uma maneira de desligar módulos que não estão sendo utilizados. O registrador PRR do ATmega328P permite desligar módulos para se obter uma economia de energia, no entando, esta funcionalidade ainda não foi implementada no simulador.

Quanto o segundo ponto crítico, a impressão dos textos de tempo simulado e uso de memória não afetam o funcionamento do sistema e, a princípio, poderiam ser removidos. Uma abordagem menos drástica seria diminuir a frequência de impressão para reduzir a quantidade de chamadas à função *setText*.

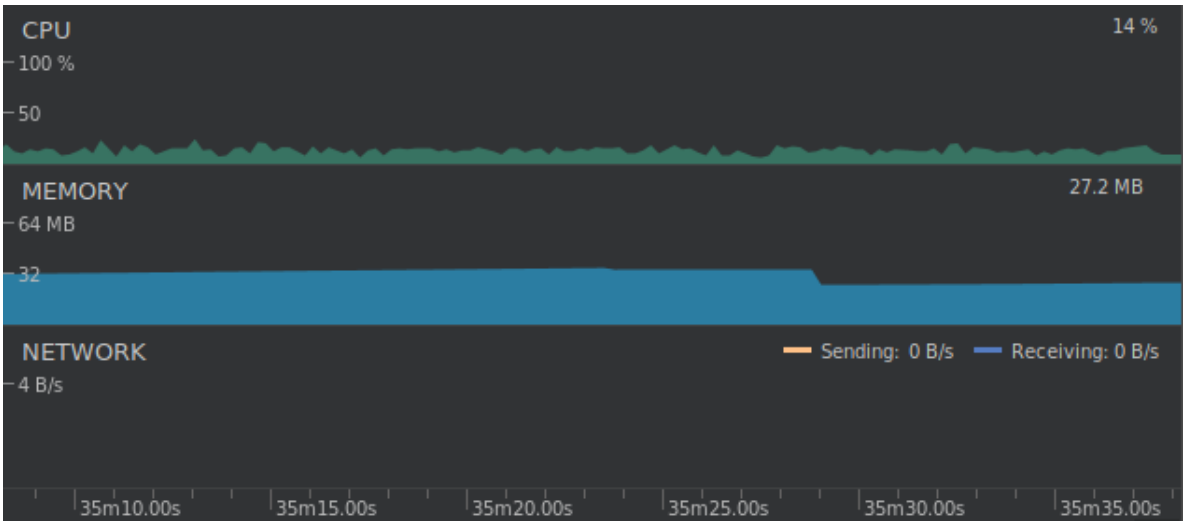


Figura 4.20: Uso de CPU e memória para o projeto *Blink*.

Fonte: Autor

Name	Total (µs)	%	Self (µs)	%	Children (µs)	%
main() ()	1,634,507	100.00	7,933	0.49	1,626,574	99.51
dispatchMessage() (android.os.Handler)	1,587,398	97.12	2,573	0.16	1,584,825	96.96
handleCallback() (android.os.Handler)	1,456,993	89.14	1,315	0.08	1,455,678	89.06
run() (android.view.Choreographer$FrameDisplayEventReceiver)	1,297,118	79.36	605	0.04	1,296,513	79.32
run() (com.example.kollins.sofia.UCModule_View$1)	158,560	9.70	2,340	0.14	156,220	9.56
setText() (android.widget.TextView)	155,122	9.49	1,033	0.06	154,089	9.43
access$200() (com.example.kollins.sofia.UCModule_View)	339	0.02	339	0.02	0	0.00
access$400() (com.example.kollins.sofia.UCModule_View)	274	0.02	274	0.02	0	0.00
access$100() (com.example.kollins.sofia.UCModule_View)	258	0.02	258	0.02	0	0.00
access$300() (com.example.kollins.sofia.UCModule_View)	227	0.01	227	0.01	0	0.00
handleMessage() (com.example.kollins.sofia.UCModule$UCHandler)	114,035	6.98	5,071	0.31	108,964	6.67
clockCPU() (com.example.kollins.sofia.CPUModule)	20,516	1.26	1,424	0.09	19,092	1.17
clockADC() (com.example.kollins.sofia.atmega328p.ADC_ATmega328P)	18,482	1.13	1,367	0.08	17,115	1.05
clockTimer0() (com.example.kollins.sofia.atmega328p.Timer0_ATmega328P)	17,774	1.09	1,304	0.08	16,470	1.01
clockTimer1() (com.example.kollins.sofia.atmega328p.Timer1_ATmega328P)	17,468	1.07	1,339	0.08	16,129	0.99
clockTimer2() (com.example.kollins.sofia.atmega328p.Timer2_ATmega328P)	17,244	1.05	1,265	0.08	15,979	0.98
clockUCView() (com.example.kollins.sofia.UCModule_View)	15,656	0.96	1,236	0.08	14,420	0.88

Figura 4.21: Tempo de uso da CPU (medido em uma janela de 5 minutos).

Fonte: Autor

No entanto, pode-se observar pela figura 4.21 que a maior parte do tempo é gasto com funções internas (para gerenciamento de interface gráfica). Os métodos de *clock* e a *UCModule_View* correspondem a apenas 17% do uso da CPU.

Também foi feito um *profilling* específico para o uso de memória. Os resultados são mostrados na figura 4.22.

O problema maior com memória é novamente a classe *UCModule_View*, que faz continuamente chamadas para o método *getString* a fim de formatar corretamente o tempo simulado na tela. Uma solução seria evitar o uso desta função, fazendo uma formatação manual da *string* ou utilizando soluções locais, que não envolvam o uso de arquivos de recurso do Android.

Também é possível observar que o módulo de CPU aparece nas primeiras posições da lista. Isso ocorre pois cada instrução utiliza variáveis locais para armazenar seus operadores. É interessante criar variáveis globais para serem utilizadas pelas instruções da CPU uma vez que seu uso é intenso.

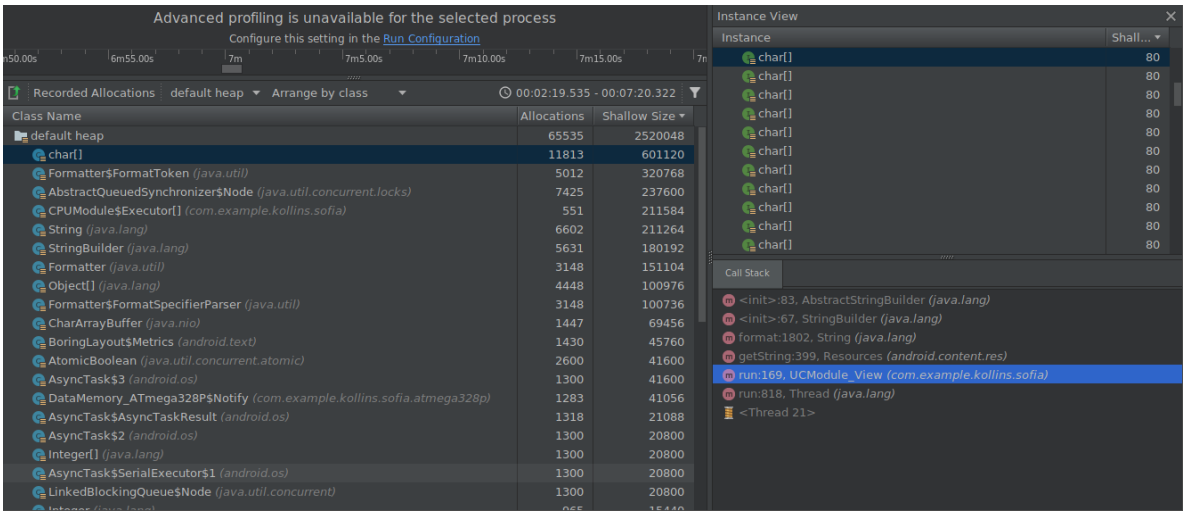


Figura 4.22: Consumo de memória (medido em uma janela de 5 minutos).

Fonte: Autor

4.4.4 Código e documentação

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 na internet.

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

Além do código, foi escrita uma documentação referente ao projeto no *Gitbook*, que pode ser acessada a partir do repositório do projeto ou no menu "*About*"do aplicativo.

- **Documentação:** `https://project-sofia.gitbook.io/project/`

Capítulo 5

Conclusão

Nesta monografia foi apresentado o projeto SOFIA, um emulador Arduino criado para Android.

Pode-se dizer que o sistema desenvolvido atende aos objetivos propostos mesmo ainda sendo uma versão parcial: 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 analógica) ou fazendo medições dos estados dos pinos digitais. O usuário também tem a disposição uma IDE modificada que o permite escrever códigos e transferi-los para o simulador facilmente.

Entre as limitações do sistema, destaca-se sua performance em termos de velocidade de simulação. A execução de códigos provindos da IDE ainda é lenta devido a quantidade de instruções *assembly* geradas pelo compilador e a capacidade de processamento do *smartphone*. Para o desenvolvimento, foi utilizado um *smartphone* ASUS ZenFone 2, com 4 núcleos de processamento (2,33GHz), 4GB de memória principal, 32GB de armazenamento, executando o Android 5.0, e frequência de *clock* efetiva observada neste sistema foi de aproximadamente 200Hz para o projeto *Blink* (uma redução de 80.000 vezes a velocidade do sistema real). Ainda assim, é possível utilizar o sistema para pequenos projetos (utilizando escalas de tempo reduzidas), o que ainda o torna útil para aplicações didáticas.

Em termos de arquitetura, o sistema buscou preservar os mesmos módulos e fluxos de comunicação existentes no *hardware* do microcontrolador, por motivos, principalmente, de facilidade de implementação e desempenho, o que acabou tornando a arquitetura do *software* um pouco mais complexa. No entanto, o sistema ainda está modular, o que é uma característica importante para sua expansão.

Quanto à qualidade do sistema, a análise estática do SonarQube não revelou grandes problemas no código e os principais módulos foram quase que totalmente cobertos por teste. Ainda é necessária a criação de muitos casos de teste para cobrir todo o projeto e garantir um nível maior de qualidade, estes estão sendo desenvolvidos em ordem de prioridade para os demais módulos.

Para a continuidade do projeto, deseja-se incluir nas próximas versões:

- **Instrumentos de medição:** Permitir que o usuário obtenha informações a respeito de frequência de oscilação, *duty cycle*, etc. nos pinos de saída.
- **Visualização da memória:** Apresentar ao usuário o estado dos registradores e da memória externa em tempo real de simulação.
- **Criação de novos módulos:** Vários módulos e funcionalidades do ATmega328P ainda não foram adicionados e devem ser incluídos para as próximas versões, tais como comparador analógico, memória EEPROM, etc.
- **Comunicação serial:** Entre os módulos não implementados, a USART é um dos principais e deve ser o primeiro a ser trabalhado. Espera-se também utilizar o monitor serial da própria IDE do Arduino (assim como ocorre nos sistema real) de modo a facilitar seu uso.

Além disso, serão estudadas maneiras de aumentar o desempenho da simulação, seja por meio de alterações internas no sistema ou com otimizações externas.

5.0.1 Cronograma de Atividades

A tabela 5.1 apresenta o cronograma executado até o momento em relação ao planejado.

Tabela 5.1: Cronograma

	03/18	04/18	05/18	06/18	07/18	08/18	09/18	10/18	11/18	12/18
01										
02										
03										
04										
05										
06										
07										
08										
09										
	Planejado									
	Executado									

1. Alteração do código fonte da IDE oficial do Arduino e integração com o simulador.

2. Planejamento do sistema como um todo, incluindo a definição dos elementos de interface, estrutura de classes, arquitetura e demais itens relacionados à organização interna do sistema;

3. Decodificação do arquivo hexadecimal gerado pelo compilador e execução das instruções *assembly*;

4. Desenvolvimento do módulo de entrada e saída digital;

5. Detecção e tratamento de interrupções;
6. Desenvolvimento do módulo de *timer* (8 e 16 bits);
7. Desenvolvimento do módulo ADC (*Analog to Digital Converter*);
8. Escrita da monografia (parcial);
9. Testes e validações

Referências

[1] Olhar Digital. Microsoft relembra produtos que eram sucesso na época do xp. https://olhardigital.com.br/noticia/microsoft-relembra-produtos-que-eram-sucesso-na-epoca-do-xp/40602, Acesso em: 11 de março de 2018.

[2] Tecmundo. Samsung galaxy s9. https://comparador.tecmundo.com.br/samsung-galaxy-s9/, Acesso em: 18 de março de 2018.

[3] R.R. Lecheta. *Google Android 4ª edição: Aprenda a criar aplicações para dispositivos móveis com o Android SDK*. Novatec Editora, 2015.

[4] Documentação Android. Content license. https://source.android.com/setup/licenses, Acesso em: 18 de março de 2018.

[5] Paul Deitel, Harvey Deitel, and Abbey Deitel. *Android for Programmers: An App-Driven Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2012.

[6] José Ernesto Almas de Jesus JUNIOR. Implementação de um osciloscópio de baixo custo com exibição gráfica em aplicativo para android, 2016.

[7] A. C. Eberendu, B. O. Omaiye, and E. C. Nwokorie. 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, Nov 2017.

[8] H. F. Teng, M. J. Wang, and C. M. Lin. 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, Dec 2011.

[9] J. Overman, L. Pool, L. Kreuk, and T. Ketel. Arduino - the open-source ide. https://delftswa.gitbooks.io/desosa-2017/content/arduino/chapter.html, Acesso em: 08 de abril de 2018.

[10] Labcenter Electronics. Create your package. https://www.labcenter.com/buy-vsm/, Acesso em: 15 de abril de 2018.

[11] VirtualBreadboard. Vbb software licenses. http://www.virtualbreadboard.com/DocView.html?doc=WebShop/WebShop, Acesso em: 15 de abril de 2018.

[12] Simuino. Arduino uno/mega simulator. http://web.simuino.com/home-1, Acesso em: 15 de abril de 2018.

[13] Stanley Huang. Codeblocks arduino ide. http://arduinodev.com/codeblocks/, Acesso em: 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. https://www.tinkercad.com/, Acesso em: 15 de abril de 2018.

[15] Atmel. Atmega328/p datasheet complete. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf, Acesso em: 25 de maio de 2018.

[16] Intel. Hexadecimal object file format specification. https://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2012/ads264_mws228/Final%20Report/Final%20Report/Intel%20HEX%20Standard.pdf, Acesso em: 25 de maio de 2018.

[17] Arduino. Internal temperature sensor. https://playground.arduino.cc/Main/InternalTemperatureSensor, Acesso em: 27 de maio de 2018.

Apêndice A

Projeto *Blink* utilizado para realização do *profilling*

```
/*
  Blink

  Turns an LED on for one microsecond, then off for one microsecond, repeatedly.

  This code was modified by Kollins Lima (May 25, 2018).

  The original code is available at:
  http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);  // turn the LED on (HIGH is the voltage level)
  delayMicroseconds(1);             // wait for a microsecond
  digitalWrite(LED_BUILTIN, LOW);   // turn the LED off by making the voltage LOW
  delayMicroseconds(1);             // wait for a microsecond
}
```

Código A.1: Projeto *Blink*.