UNIVERSIDADE FEDERAL DE PERNAMBUCO

Universidade Federal de Pernambuco

Centro de Tecnologia e Geociências

Departamento de Eletrônica e Sistemas

Graduação em Engenharia Eletrônica

Railton Silva Rocha Junior

Sistema embarcado para automação da irrigação por pulsos acionada por válvulas solenoides tipo latching

Recife

Dezembro de 2022

Railton Silva Rocha Junior

Sistema embarcado para automação da irrigação por pulsos acionada por válvulas solenoides tipo latching

Trabalho de Conclusão apresentado ao Curso de Graduação em Engenharia Eletrônica do Departamento de Eletrônica e Sistemas da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia Eletrônica.

Banca Examinadora

gov.br Documento assinado digitalmente
GUILHERME NUNES MELO
Data: 14/12/2022 16:10:24-0300
Verifique em https://verificador.iti.br

Prof. Guilherme Nunes Melo, Dr.

gov.br Documento assinado digitalmente
JOSE SAMPAIO DE LEMOS NETO
Data: 09/12/2022 21:48:40-0300
Verifique em https://verificador.iti.br

Prof. José Sampaio de Lemos Neto, Dr.

gov.br Documento assinado digitalmente
MANASSES MESQUITA DA SILVA
Data: 14/12/2022 09:03:24-0300
Verifique em https://verificador.iti.br

Prof. Manassés Mesquita da Silva, Dr.

Recife

Dezembro de 2022

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Rocha Junior, Railton Silva.
Sistema embarcado para automação da irrigação por pulsos acionada por válvulas solenoides tipo latching / Railton Silva Rocha Junior. - Recife, 2022.
87 : il., tab.

Orientador(a): Guilherme Nunes Melo
Cooorientador(a): Manassés Mesquita da Silva
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Tecnologia e Geociências, Engenharia Eletrônica - Bacharelado, 2022.
Inclui referências, apêndices.

1. Sistemas embarcados. 2. Automatização. 3. Irrigação pulsada. 4. Eficiência de irrigação. I. Melo, Guilherme Nunes. (Orientação). II. Silva, Manassés Mesquita da. (Coorientação). III. Título.

620 CDD (22.ed.)

"Buscai em primeiro lugar o Reino de Deus e a sua justiça e todas estas coisas vos serão dadas em acréscimo."

São Mateus 6, 33

Para José Geraldo, meu herói,

avô e pai

Agradecimentos

À Deus Pai, Filho e Espírito Santo, que me deu a vida, o propósito de minha existência e as forças necessárias para a caminhada. E à família que Ele me deu no Céu, que não cansa de interceder por mim, Nossa Senhora, São Josemaria Escrivá e São Francisco de Sales, Maria Lyvia e vovô Geraldo.

À minha família que sempre apostou na minha formação e fez sacrifícios inomináveis para que eu tivesse uma boa educação. Sou infinitamente grato à minha mãe Alexsandra, às minhas tias Andréa e Adriana, ao meu tio Glauber, à minha avó Laudicéia e ao meu pai Railton.

Sou grato à minha futura esposa, Girlayne Norberto, por todo amor, apoio, motivação e paciência nessa jornada, que nesse tempo foi verdadeiro bálsamo em minhas feridas. E também à minha sogra, Suely, por sempre cuidar de mim como uma mãe.

Agradeço aos meus professores, sempre me inspirando e me estimulando a ir mais longe, especialmente ao meu orientador Prof. Guilherme e ao Prof. Manassés, por todo apoio e paciência nessa trajetória. Agradeço também a Karla, a Rodrigo e à familia InsoleON por tornarem o projeto possível com todo apoio e dedicação.

E a todos os meus amigos, que me fortaleceram e me sustentaram, em especial aos amigos de graduação, Breno, Daniel Lucas, Farias, Pedro e Thales.

Resumo do Trabalho de Conclusão de Curso apresentado ao DES/UFPE como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Eletrônica (Eng.)

Sistema embarcado para automação da irrigação por pulsos acionada por válvulas solenoides tipo latching

Railton Silva Rocha Junior

Orientador(a): Guilherme Nunes Melo Dr.

Programa: Engenharia Eletrônica

A agricultura moderna tem como principal objetivo o aumento da produtividade combinado ao uso racional dos recursos naturais. Nesse contexto, as técnicas de irrigação de precisão vêm sendo aplicadas, dentre as quais destaca-se a técnica de irrigação pulsada por gotejamento, cuja economia de água tem sido atestada por diversos autores na literatura. A utilização de novas tecnologias aplicadas a produção agrícola, como os sistemas embarcados, permitem a automatização de diversas tarefas, antes realizadas de forma manual. Portanto, como forma de auxiliar o desenvolvimento da irrigação pulsada, este trabalho apresenta um sistema embarcado, implementado em *hardware* e *software*, para atuação independente, simultânea e cronometrada de até dez válvulas solenoides do tipo *latching*, a partir da configuração dos parâmetros de irrigação definidos por um operador. Os solenoides *latching* proporcionam maior economia de energia, pois, ao contrário dos solenoides convencionais que permanecem energizados durante o tempo de irrigação, o *latching* consome energia apenas na comutação do estado. Dessa forma, duas placas de circuito impresso foram confeccionadas, a placa de controle e a placa de interface com o usuário, permitindo a configuração independente e o acionamento simultâneo das dez válvulas solenoides. Além disso, o sistema implementado apresenta uma solução versátil e didática, a fim de possibilitar um aprimoramento de suas funcionalidades.

Palavras-chave: Sistemas embarcados, Automatização, Irrigação Pulsada, Eficiência de irrigação .

Abstract of Course Conclusion Work presented to DES/UFPE as a partial fulfillment of the requirements for the degree of Bachelor of Electronic Engineering (Eng.)

Embedded system for automation of pulse irrigation activated by latching solenoid valves

Railton Silva Rocha Junior

Advisor: Guilherme Nunes Melo Dr.

Course: Electronic Engineering

The main objective of modern agriculture is to increase productivity combined with the rational use of natural resources. In this context, microirrigation techniques have been applied, among which the pulsed drip irrigation technique stands out, whose water savings have been attested by several authors in the literature. The use of new technologies applied to agricultural production, such as embedded systems, allow for the automation of various tasks, which were previously performed manually. Therefore, as a way to help the development of pulsed irrigation, this work presents an embedded system, implemented in hardware and software, for independent, simultaneous and timed operation of up to ten solenoid valves of the latching type, based on the configuration of the defined irrigation parameters by an operator. Latching solenoids provide greater energy savings, because, unlike conventional solenoids that remain energized during the irrigation time, latching consumes energy only when switching the state. In this way, two printed circuit boards were made, the control board and the user interface board, allowing the independent configuration and the simultaneous actuation of the ten solenoid valves. In addition, the implemented system presents a versatile and didactic solution, in order to enable an improvement of its functionalities.

Keywords: Embedded Systems; Automation; Pulsed Irrigation; Irrigation Efficiency

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Abreviações

CI	Circuito Integrado
EECAC	Estação Experimental de Cana-de-Açúcar
GPIO	Porta de Entrada e Saída
IDE	Ambiente de Desenvolvimento Integrado
LCD	*Display* de Cristal Líquido
NVS	Memória Não Volátil
PCB	Placa de Circuito Impresso
PIPO	Entrada Paralela, Saída Paralela
PISO	Entrada Paralela, Saída Serial
RTC	Relógio de Tempo Real
SISO	Entrada Serial, Saída Serial
SIPO	Entrada Serial, Saída Paralela
SR	Registrador de Deslocamento
UFRPE	Universidade Federal Rural de Pernambuco

Capítulo 1

Introdução

1.1 Contextualização e Motivação

A agricultura moderna tem como principal objetivo o aumento da produtividade combinado ao uso racional dos recursos naturais. Portanto, tem-se buscado desenvolver novas tecnologias para aumentar a produção, reduzir os custos e melhorar de forma sustentável a qualidade dos produtos (ALMEIDA *et al.*, 2015). Embora os métodos de irrigação tenham se desenvolvido nas últimas décadas (BUSATO *et al.*, 2011), ainda é necessário desenvolver manejos de irrigação que proporcionem cada vez maior racionalização do uso de água e do consumo de energia (ANDRADE, 2021).

Nesse contexto, as técnicas de irrigação de precisão (ou, em inglês, *microirrigation*) vêm sendo aplicadas, dentre as quais destaca-se a técnica de irrigação pulsada por gotejamento (AYARS e PHENE, 2007). Segundo ALMEIDA *et al.* (2015), a técnica de irrigação por pulsos consiste na aplicação da lâmina de irrigação de forma fracionada, em uma série de ciclos de irrigação *on-off* (do inglês, liga e desliga), até que se tenha aplicado completamente a lâmina desejada, em que cada evento contém uma fase de irrigação e uma de repouso.

A economia de água promovida pela irrigação por pulsos, quando associada a técnica de gotejamento, tem sido atestada por diversos autores na literatura, como expõem os resultados de ASSOULINE *et al.* (2006) com a plantação de pimenta

em Israel, EL-ABEDIN (2006) com uma cultura de milho no Egito, WARNER *et al.* (2009) com tomates nos Estados Unidos e ALMEIDA *et al.* (2015) com uma plantação de alface-americana no Brasil. Entretanto, ainda são poucos os dados de estudo da eficiência da irrigação pulsada para as principais culturas brasileiras, como a cana de açúcar, sendo necessário um amplo esforço de pesquisa nesse meio (ANDRADE, 2021).

A operação da irrigação pulsada pode ser desafiadora se feita de forma manual, uma vez que é necessária a atuação periódica para iniciar ou pausar a irrigação em intervalos de tempo específicos, a fim de obter a lâmina de irrigação desejada, tornando o processo demorado e dependente de uma mão de obra dedicada. Segundo AYARS e PHENE (2007), para o potencial completo da irrigação de precisão ser alcançado, a automação é necessária, sobretudo na aplicação do volume necessário de água e na cronometragem precisa dessa aplicação.

Como etapa inicial do processo de automação, um controle em malha aberta pode ser utilizado para configurar a frequência e a duração da irrigação (AYARS e PHENE 2007). A utilização de novas tecnologias aplicadas na produção agrícola, como os sistemas embarcados, permitem a automatização de diversas tarefas manuais. (SANTOS, 2008).

De maneira geral, o sistema embarcado é constituído por um microprocessador ou microcontrolador para executar o *software* e alguns outros dispositivos de *hardware* como, processadores de sinal digital (DSP) e entradas e saídas periféricas. O *hardware* e *software* trabalham juntos para atingir alguma funcionalidade em um sistema maior, por exemplo, o controle de vôo de um avião, controle de navegação de um veículo, ou um dispositivo associado à rede para automação residencial (LEE e HSIUNG, 2004). Ainda segundo LEE e HSIUNG (2004), a utilização de sistemas embarcados tem tornado viável a implementação de uma grande variedade de equipamentos, permitindo um controle mais acessível e configurações flexíveis.

Dentre os vários dispositivos que propiciam a automatização da irrigação, destacam-se as válvulas solenoides. As válvulas solenoides são essenciais por permitirem uma

interface dos sistemas hidráulicos com os sistemas elétricos, visto que possibilitam o controle da vazão de água a partir de comandos elétricos (RIBEIRO, 1999).

Porém, a utilização desses dispositivos aumenta o consumo de energia da irrigação. Portanto, a utilização de válvulas mais eficientes, por exemplo, as válvulas solenoides do tipo *latching*, é fundamental para tornar a irrigação possível para sistemas com restrições energéticas, como os sistemas alimentados por bateria (VASCONCELOS, 2013).

As válvulas solenoides do tipo *latching* proporcionam maior economia de energia, pois, diferentemente das válvulas solenoides tradicionais, que ficam ativas durante todo o tempo da irrigação, o solenoide *latching* consome energia apenas na comutação do estado. Normalmente um pulso de tensão com duração na ordem de dezenas de millissegundos é utilizado para realizar a comutação. Entretanto, para garantir que os solenoides operem de maneira adequada, a aplicação correta do pulso deve ser respeitada e os circuitos eletrônicos de controle devem ser bem dimensionados (BERMAD, 2022).

1.2 Objetivo Geral

Desenvolver um sistema embarcado em *hardware* e *software* aplicado à irrigação pulsada para atuar de forma cronometrada e independentemente configurável em dez válvulas solenoides *latching*, de modo a substituir o processo de acionamento manual pelo operador. O sistema ainda deve implementar uma solução em *hardware* de forma versátil e didática, que possibilite o aprimoramento posterior das funcionalidades em trabalhos futuros.

1.2.1 Objetivos específicos

- Desenvolver o módulo em *hardware* utilizando um microcontrolador integrando uma interface de configuração para o operador, entradas de referência de tempo real e acionamento para controle de válvulas solenoides do tipo *latching*;

- Desenvolver o sistema visando a utilização em ambientes com falta de energia elétrica e que possa ser alimentado por baterias;
- Criar a solução com módulos e dispositivos de fácil aquisição para garantir a possibilidade de réplica do sistema;
- Implementar *software* embarcado para controle do sistema e para servir de referência didática de programação do módulo desenvolvido utilizando uma interface de programação de fácil acesso;
- Testar e validar o sistema de controle em conjunto com o arranjo hidráulico da irrigação.

1.3 Organização do TCC

O conteúdo deste TCC está dividido em cinco capítulos e um apêndice. As referências utilizadas são apresentadas nas páginas finais do trabalho.

Capítulo 2. Os diversos conceitos abordados neste trabalho são apresentados, como as máquinas de estado, os registradores de deslocamento, o funcionamento das válvulas solenoides, bem como as estratégias para ativação dessa família de dispositivos com o uso de circuito ponte H. Além disso, dispositivos importantes para o sistema são discutidos e explanados, como é o caso do RTC, microcontrolador, LCD e o protocolo de comunicação desses dispositivos, o I^2C.

Capítulo 3. As etapas de implementação do *hardware* e do *software* são abordadas para o desenvolvimento da solução, explorando os conceitos e as decisões tomadas para a execução do projeto de maneira assertiva.

Capítulo 4. A implementação do sistema é validada e testada, tanto na fase de prototipação quanto na versão final com a placa confeccionada, além disso as decisões são avaliadas à luz dos requisitos do sistema.

Capítulo 5. As considerações finais referentes ao trabalho desenvolvido são expostas, além disso, são apresentadas as dificuldades do trabalho e possíveis melhorias em trabalhos futuros.

Capítulo 2

Fundamentação Teórica

No presente capítulo são descritos os conceitos utilizados para o desenvolvimento do sistema como um todo. É discutida a programação do sistema implementado, como o tipo particular de grafo, chamado de máquina de estados finitos, o IDE (*Integrated Development Environment*, do inglês, ambiente de desenvolvimento integrado) do Arduino, bem como a programação para microcontroladores e a comunicação com a interface I^2C. Além disso, são apresentados especificações técnicas e aspectos teóricos do funcionamento do *hardware* utilizado, como os conceitos de pontes H, registradores de deslocamento (do inglês, *shift register*), válvula solenoide, *display LCD*, reguladores de tensão positiva fixa e ajustáveis, o funcionamento do RTC (*real-time clock*, ou em português, relógio de tempo real) e por fim o funcionamento do microcontrolador utilizado, o ESP32.

2.1 Máquina de estados finitos

As máquinas de estados finitos, ou simplesmente máquinas de estados, são um modelo matemático de computação amplamente utilizado em diversos sistemas. Uma máquina de estados é, por definição, um sistema que consiste em uma coleção finita de estados $K = \{K_0, ..., K_N\}$, que sucessivamente aceita uma sequência de entradas de um conjunto finito $A = \{a_0, ..., a_N\}$, produzindo uma sequência de saídas de um conjunto $B = \{b_0, ..., b_N\}$. Além disso, existe uma função de saída μ, que

computa a saída atual como uma função das entradas e estados presentes, e uma função δ, que computa o próximo estado a partir das entradas atuais e do estado presente. Assim, matematicamente, uma máquina de estados pode ser definida como $\mu(K_n, a_n) = b_n$, enquanto $\delta(K_n, a_n) = K_{n+1}$. Em linhas gerais, uma máquina de estados finitos pode ser definida como um conjunto de estados com ligações entre si, que produzem saídas a partir das entradas recebidas e a depender do estado em que se encontram (GOLOMB, 2017).

Uma máquina de estados pode ser completamente especificada a partir de uma tabela que indica a saída e o próximo estado correspondente para quaisquer combinações de estado e entrada. Por exemplo, considera-se uma máquina com três estados distintos, R, S e T, que aceita como entrada os valores 0 e 1, e produz como saída os símbolos a, b e c. A Tabela 2.1 apresenta o próximo estado e a saída correspondente a partir da entrada recebida, por exemplo, se a máquina estiver no estado S, caso seja recebido o valor 0, o símbolo b é escrito na saída e o sistema faz a transição para o estado R, por outro lado, caso seja recebido o valor 1, o símbolo c é escrito na saída e o sistema permanece no estado S (GOLOMB, 2017).

	Entradas	
Estados	0	1
R	S, a	T, b
S	R, b	S, c
T	T, a	R, a

Tabela 2.1: Representação de tabela de uma máquina de estados finitos (Fonte: Adaptado de GOLOMB, 2017).

Portanto, dado um estado inicial e uma sequência de entrada arbitrária, por exemplo, estado inicial S e entrada 01101, é possível seguir a sequência da máquina de estados, conforme a Tabela 2.2, e encontrar o resultado da sequência de saída como *bbaac* e o estado final S (GOLOMB, 2017).

Uma alternativa à representação por tabela é o método de diagrama de estados, em que os estados são representados a partir de nós, as entradas são representadas

Tempo	Entrada	Estado	Saída
1	0	S	b
2	1	R	b
3	1	T	a
4	0	R	a
5	1	S	c
6		S	

Tabela 2.2: Comportamento sequencial da máquina de estados finitos (Fonte: Adaptado de GOLOMB, 2017).

por setas orientadas de um nó a outro, de acordo com o próximo estado correspondente, e as saídas são representadas entre parênteses, ao lado de cada entrada. A Figura 2.1 exibe a máquina da Tabela 2.1 na forma de diagrama de estados. Nota-se que caso a máquina esteja no estado S e receba o valor 0, a seta elucida que o próximo estado é o R e a saída é b, por outro lado, caso receba o valor 1, a seta indica que o próximo estado é o S e a saída é c, ou seja, resultado equivalente ao obtido anteriormente (GOLOMB, 2017).

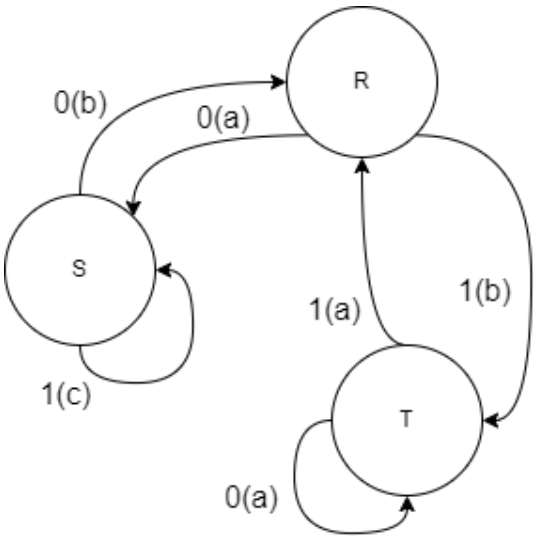


Figura 2.1: Diagrama de estados de uma máquina de estados finitos (Fonte: Adaptado de GOLOMB, 2017).

Sendo assim, segundo GOLOMB (2017), ambas representações especificam completamente uma máquina de estados finitos, porém a notação de diagrama de estados será preferencialmente adotada neste trabalho por permitir identificar de maneira

visual o comportamento da máquina de estados, a partir de uma sequência de entrada.

2.2 Registrador de deslocamento

Uma vez que a quantidade de portas de entrada e saída (GPIOs) disponíveis em um microcontrolador são limitadas, a utilização de dispositivos periféricos que permitam uma quantidade maior de saídas é crucial. Surge então a motivação de utilizar o registrador de deslocamento como parte integrante deste projeto.

Um registrador de deslocamento, ou *shift register* (*SR*), como é conhecido por seu termo inglês, é um registrador de *n-bits* com capacidade de deslocar a posição do dado armazenado em um *bit* a cada pulso de clock. É possível construir tais registradores utilizando uma coleção de dois ou mais *flip-flops* do tipo D sincronizados com o mesmo sinal de *clock* (WAKERLY, 2018).

Existem quatro tipos básicos de SR que são nomeados a depender da forma como estão organizadas suas entradas (ou em inglês, *in*) e saídas (ou em inglês, *out*), que podem ser paralelas (ou em inglês, *parallel*) ou seriais (ou em inglês, *serial*). Sendo assim, os possíveis tipos são: *serial-in*, *serial-out* (*SISO*); *serial-in*, *parallel-out* (*SIPO*); *parallel-in*, *parallel-out* (*PISO*); *parallel-in*, *parallel-out* (*PIPO*). As configurações de interesse neste trabalho são *SISO* e *SIPO*. Na configuração *SISO*, os *bits* são introduzidos na entrada serial e são movidos para o próximo registrador a cada pulso de *clock*, de modo que somente ao total de n pulsos de *clock* o *bit* é obtido na saída (FÉ, 2017). Uma implementação de um *SR* na configuração *SISO* pode ser vista na Figura 2.2.

No tipo de registrador de deslocamento *SIPO*, os *bits* também são introduzidos na entrada serial, como no *SISO*, entretanto cada um dos registradores internos apresenta uma saída. Dessa forma, com apenas uma entrada serial é possível utilizar n saídas, sendo uma das principais funções dessa configuração a conversão serial-paralelo de dados (WAKERLY, 2018). A Figura 2.3 exibe uma implementação utilizando *flip-flops* do tipo D, em que as saídas Q_1 a Q_n são as n saídas disponíveis

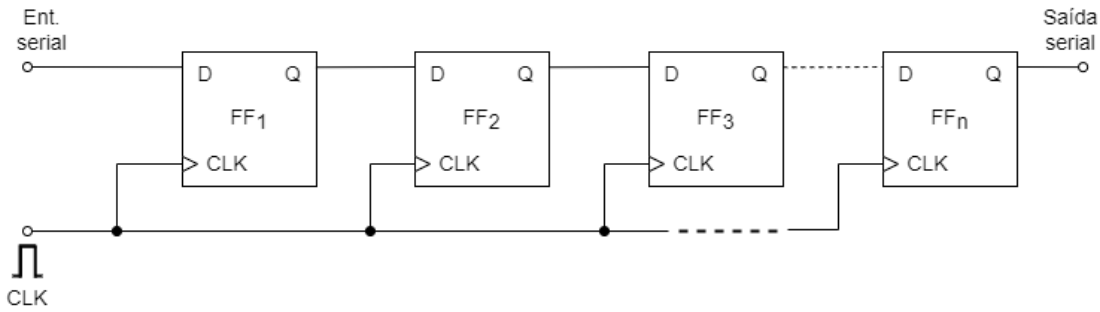


Figura 2.2: Estrutura de um registrador de deslocamento do tipo SISO (Fonte: Adaptado de WAKERLY, 2018).

nesse tipo de *SR*.

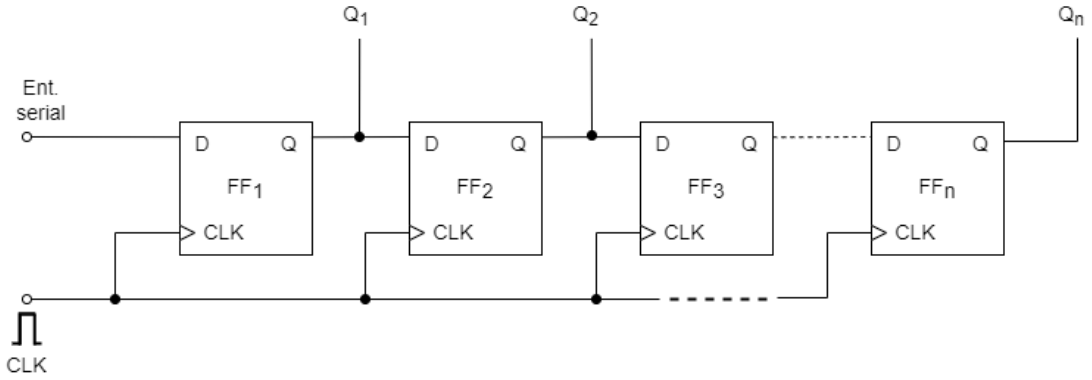
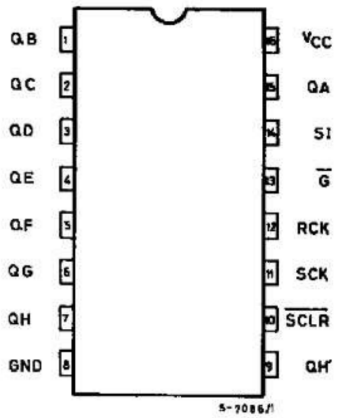


Figura 2.3: Estrutura de um registrador de deslocamento do tipo SIPO (Fonte: Adaptado de WAKERLY, 2018).

2.2.1 Registrador de deslocamento de 8-bits

Uma ampla gama de circuitos integrados (*CI*) implementam o circuito digital do *SR* nas várias configurações apresentadas e com diferentes valores de n, entretanto, o *CI 74HC595* é utilizado neste projeto, por disponibilizar as configurações *SISO* e *SIPO*. O dispositivo consiste em um registrador de deslocamento de *8-bits* amplamente disponível no mercado. A Figura 2.4 exibe a pinagem deste dispositivo, bem como uma breve descrição de cada pino (STMICROELECTRONICS, 1994).

Os pinos GND e V_{cc} são respectivamente o terra e a tensão de alimentação positiva, e devem ser mantidos fixos para garantir o correto funcionamento do dispositivo e referenciar o nível lógico do circuito como um todo. Sendo assim, um nível lógico



PIN DESCRIPTION

PIN No	SYMBOL	NAME AND FUNCTION
1, 2, 3, 4, 5, 6, 7, 15	QA to QH	Data Outputs
9	QH'	Serial Data Outputs
10	$\overline{\text{SCLR}}$	Shift Register Clear Input
11	SCK	Shift Register Clock Input
13	$\overline{\text{G}}$	Output Enable Input
14	SI	Serial Data Input
12	RCK	Storage Register Clock Input
8	GND	Ground (0V)
16	V_{CC}	Positive Supply Voltage

Figura 2.4: Descrição dos pinos do registrador de deslocamento 74HC595 (Fonte: STMICROELECTRONICS, 1994)).

baixo (ou em inglês *Low*, ou simplesmente *L*) deve estar próximo do valor de tensão do sinal *GND*, enquanto um nível lógico alto (ou em inglês *High*, ou simplesmente *H*) deve estar próximo do valor de tensão do sinal V_{cc}. Os pinos de controle são: *SI*, que é a entrada serial do dispositivo; *SCK*, que é o relógio da entrada do sistema; $\overline{SCLR}$, que é a entrada de limpeza das saídas; *RCK*, que é o *clock* de registro de armazenamento; e $\overline{G}$, que é o pino de habilitação das saídas (STMICROELECTRONICS, 1994).

Sempre que um nível lógico é adicionado em *SI* e um pulso de *clock* é aplicado no pino *SCK*, a entrada é deslocada para o próximo registro, e os demais registros também são deslocados internamente no *SR*. Entretanto, para que essa informação seja refletida nas saídas, i.e. nos pinos Q_A a Q_H, faz-se necessário aplicar um pulso positivo no pino *RCK*, dessa forma os dados internos do *SR* são colocados nos registradores de armazenamento. Ao aplicar um nível lógico baixo no pino $\overline{SCLR}$, o dispositivo e todas as suas saídas vão para nível lógico baixo. Os pinos de saída, por sua vez, apenas são habilitados se o pino $\overline{G}$ estiver em nível lógico baixo (STMICROELECTRONICS, 1994). A Figura 2.5 apresenta a tabela verdade que resume o funcionamento do dispositivo.

É possível observar que, nativamente, o dispositivo apresenta a configuração

INPUTS					OUTPUT
SI	SCK	$\overline{SCLR}$	RCK	$\overline{G}$	
X	X	X	X	H	QA THRU QH OUTPUTS DISABLE
X	X	X	X	L	QA THRU QH OUTPUTS ENABLE
X	X	L	X	X	SHIFT REGISTER IS CLEARED
L	⌐	H	X	X	FIRST STAGE OF S.R. BECOMES "L" OTHER STAGES STORE THE DATA OF PREVIOUS STAGE, RESPECTIVELY
H	⌐	H	X	X	FIRST STAGE OF S.R. BECOMES "H" OTHER STAGES STORE THE DATA OF PREVIOUS STAGE, RESPECTIVELY
X	¬	H	X	X	STATE OF S.R IS NOT CHANGED
X	X	X	⌐	X	S.R. DATA IS STORED INTO STORAGE REGISTER
X	X	X	¬	X	STORAGE REGISTER STATE IS NOT CHANGED

X: DON'T CARE

Figura 2.5: Tabela verdade do CI registrador de deslocamento 74HC595. (Fonte: STMICROELECTRONICS, 1994)

SIPO como predominante, ou seja, os valores são aplicados na entrada serial e recolhidos na saída paralela do dispositivo. Entretanto, o pino $Q_{H'}$ é utilizado para disponibilizar a saída serial do *SR*, ou seja, funcionando também na configuração *SISO*. Assim, um *CI* registrador de deslocamento pode aplicar sua saída serial na entrada serial de outro registrador, possibilitando o cascateamento de vários registradores de deslocamento (STMICROELECTRONICS, 1994).

2.2.2 Expansão de portas digitais do microcontrolador

Se os pinos $\overline{SCLR}$ e $\overline{G}$ são mantidos com níveis lógicos alto e baixo, respectivamente, o dispositivo permanece sempre com as saídas habilitadas e dependentes apenas dos demais pinos de controle, nomeadamente, SI, SCK e RCK. Dessa forma, é possível controlar diversas saídas paralelas utilizando apenas três pinos do microcontrolador para proporcionar o nível lógico das entradas de controle dos *SRs* conectados em cascata. A Figura 2.6 mostra como é feita essa ligação com o cascateamento de quatro *SR* (STMICROELECTRONICS, 1994).

2.3 Válvulas solenoides

Uma válvula solenoide é formada a partir da combinação de dois dispositivos funcionais básicos: o solenoide e a válvula. De maneira geral, uma válvula é um

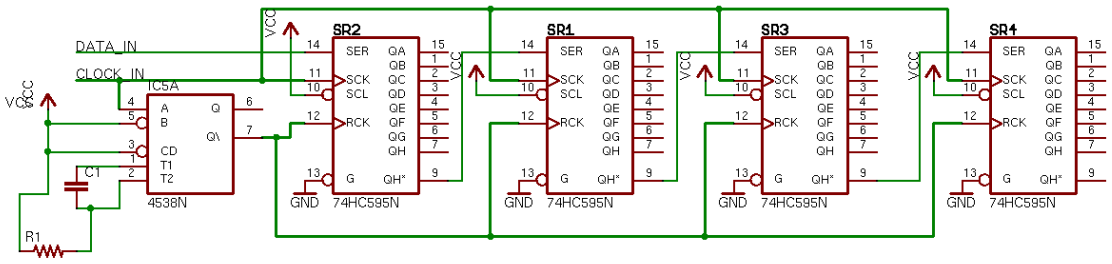


Figura 2.6: Ligação em cascata do registrador de deslocamento de 8-*bits*. (Fonte: STACKEXCHANGE, 2019)

dispositivo mecânico capaz de controlar o fluxo de um fluido em tubulações. O solenoide é uma bobina de fio que ao ser energizada eletricamente produz um campo magnético em seu interior, que provoca um movimento mecânico em um núcleo ferromagnético, colocado no centro do campo. Em uma operação convencional, quando a bobina é energizada o núcleo está em uma posição, quando desenergizada o núcleo está em outra posição. Portanto, uma válvula solenoide é um dispositivo construído para controlar a vazão do fluido a partir de uma bobina solenoide acionada eletricamente, que é acoplada ao corpo da válvula (RIBEIRO, 1999).

Quanto à operação, as válvulas solenoides são projetadas para atuarem na função de liga e desliga (*on-off*) e com a construção normalmente aberta ou normalmente fechada. A válvula normalmente fechada bloqueia o fluxo enquanto não há corrente aplicada à bobina e permite a vazão caso a bobina seja energizada. A válvula normalmente aberta tem o funcionamento inverso, assim permite a vazão do fluído quando a bobina está desenergizada e bloqueia o fluxo apenas quando é energizada (RIBEIRO, 1999).

Qualquer uma delas pode ser escolhida a depender do funcionamento do sistema, de modo a poupar energia no acionamento das válvulas, ou seja, se o sistema passa mais tempo permitindo a vazão do fluido, deve-se optar pela utilização de uma válvula normalmente aberta, gastando energia apenas no bloqueio do fluido, e vice-versa. (RIBEIRO, 1999).

As válvulas solenoides podem ser categorizadas em diferentes grupos de operação, dentre os quais destacam-se a operação (ou ação) direta e a operação indireta. No

dispositivo de ação direta o núcleo solenoide é mecanicamente ligado ao disco da válvula e abre ou fecha diretamente a válvula (RIBEIRO, 1999). A Figura 2.7 ilustra o funcionamento da válvula solenoide de ação direta normalmente fechada, em que a bobina está desenergizada na imagem da esquerda, bloqueando o fluxo do fluído, e energizada na imagem da direita, permitindo a passagem do fluido.

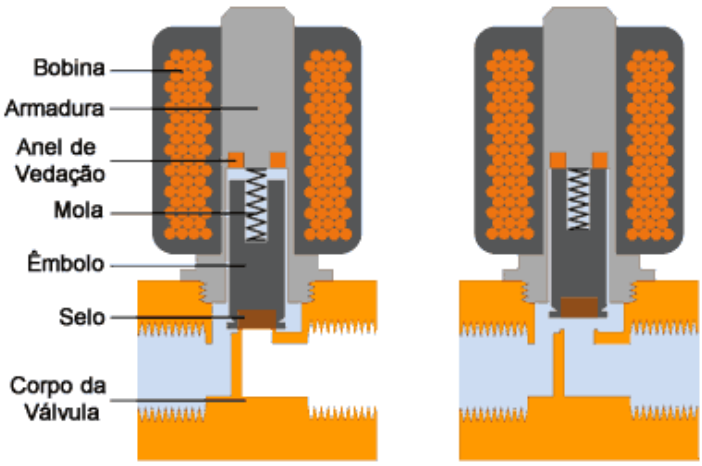


Figura 2.7: Válula solenoide de duas vias de operação direta, normalmente fechada (Fonte: SILVEIRA, 2017).

A Figura 2.8 exibe o funcionamento das válvulas solenoides operadas indiretamente. Cada parte componente da válvula solenoide é representada por uma letra: A, lado de entrada; B, diafragma; C, câmara de pressão; D, passagem de alívio de pressão; E, solenoide eletromecânico; F lado da saída.

As válvulas solenoides operadas indiretamente usam a pressão diferencial do meio incidente das portas da válvula para abrir e fechar. B é conectado ao corpo válvula por uma mola e separa A de F. Um pequeno orifício em B garante que o fluido possa fluir para C. A pressão gerada pelo fluido acima de B e a ação da mola asseguram que B permaneça fechado, bloqueando a vazão do fluido, enquanto E está bloqueando D (SILVEIRA, 2017).

A parte inferior da Figura 2.8 mostra a operação quando a válvula é ativada. Ao acionar o solenoide, D é liberado, fazendo com que a pressão em C fique menor, e consequentemente B seja levantado, permitindo a vazão do fluido. Portanto, a válvula de operação indireta pode ser utilizada em casos em que seja necessário

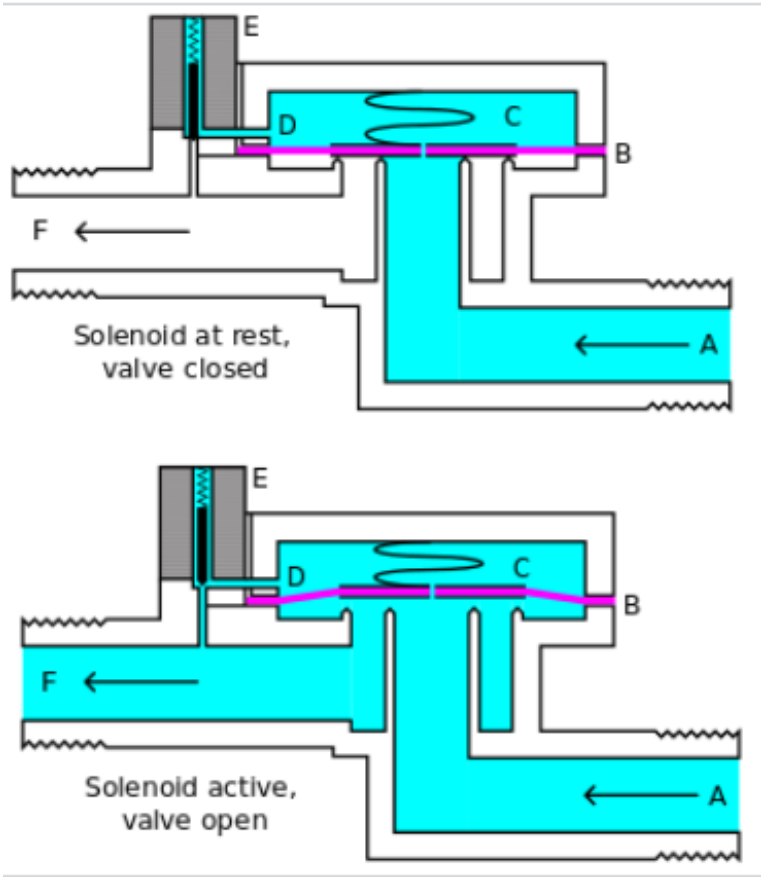


Figura 2.8: Atuação de válvula solenoide de ação indireta (Fonte: GONZÁLEZ, 2015).

uma grande taxa de fluxo e em que haja diferencial de pressão suficiente, como é o caso dos sistemas de irrigação (SILVEIRA, 2017).

2.3.1 Válvula solenoide *latching*

As válvulas tradicionais podem ser dispendiosas energeticamente, principalmente quando aplicadas a sistemas com limitações energéticas, como é o caso de sistemas alimentados com baterias ou painéis fotovoltaicos, por precisarem manter a bobina energizada durante toda a operação do sistema. Dessa forma, uma alternativa é a utilização das válvulas tipo *latching*, que consomem energia apenas na comutação da bobina de um estado para o outro, a partir da aplicação de um pulso de corrente contínua (VASCONCELOS, 2013).

Um solenoide do tipo *latching* é normalmente especificado, dentre outras carac-

terísticas, pela quantidade de fios, tensão de operação, resistência da bobina e tempo do pulso. A quantidade de fios, que podem ser 2 ou 3, define a característica da alimentação da bobina. Nesse caso, com dois fios, normalmente um positivo e outro negativo, o solenoide comuta para o estado aberto quando atuado com polarização positiva em seus terminais e para o estado fechado quando atuado com polarização negativa. Já com 3 fios, normalmente positivo, negativo e comum, é necessário colocar a referência conectada ao terminal comum e atuar com tensão positiva no terminal positivo para abrir a válvula, ou tensão negativa no terminal negativo para fechar (BERMAD, 2022).

A tensão de operação e a resistência definem, a partir da lei de Ohm, a corrente necessária para o funcionamento do dispositivo no momento do acionamento. Enquanto o tempo de pulso, normalmente em milissegundos, define o tempo que o pulso de tensão deve ser aplicado nos terminais do dispositivo para ocorrer a comutação de um estado para o outro (BERMAD, 2022).

2.4 Ponte H

A ponte H é um circuito elétrico comumente utilizado em conjunto com motores elétricos para controlar a velocidade e direção do motor, mas também podem ser utilizados com outras cargas, como o solenoide, para controlar o sentido da corrente em seus terminais, por meio dos mesmos princípios de funcionamento (PATSKO, 2006). Portanto, nesta seção é discutido o funcionamento da ponte H em conjunto com motores, para explorar assertivamente os conceitos de funcionamento do circuito, a fim de aplicá-los posteriormente ao comportamento do solenoide.

A maioria dos motores DC *(direct current*, ou traduzindo, corrente contínua) pode rotacionar em ambas as direções a depender de como a fonte de alimentação (por exemplo, uma bateria) está conectada aos terminais dele. Ou seja, para rotacionar o motor no sentido direto, os terminais positivo e negativo do motor devem ser conectados aos respectivos terminais positivo e negativo da bateria, enquanto que para rotacionar no sentido reverso, os terminais devem ser invertidos conectando o

positivo de um no negativo do outro (PATSKO, 2006).

A topologia básica da ponte H permite esse chaveamento. Conforme o nome sugere, quatro chaves mecânicas ou eletrônicas são posicionadas formando a letra "H", em que cada chave é disposta em um extremo e o motor ou solenoide é colocado no ramo central. A fonte de alimentação é representada pelo sinal positivo e a referência negativa é representada pelo sinal negativo (PATSKO, 2006). A Figura 2.9 exibe a topologia do circuito, em que cada chave S1, S2, S3 e S4, é simbolicamente representada como chaves mecânicas.

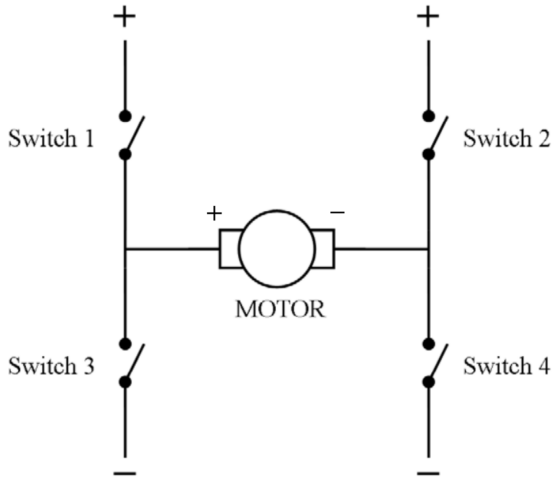


Figura 2.9: Topologia da ponte H. (Fonte: SIEBEN, 2003)

Para ligar o motor no sentido direto, as chaves S1 e S4 devem ser ligadas, permitindo que a corrente elétrica passe do terminal positivo para o terminal negativo do motor, conforme mostra a Figura 2.10. Por outro lado, para ligar o motor no sentido reverso, as chaves 2 e 3 devem ser ativadas, permitindo que a corrente elétrica flua do terminal negativo para o terminal positivo do motor, como visto na Figura 2.11. Entretanto, para não gerar um curto circuito na fonte de alimentação é necessário garantir que as chaves S1 e S3 ou S2 e S4 não estejam ativas simultaneamente, conforme exibido na Figura 2.12 (SIEBEN, 2003).

Os motores são frequentemente controlados por alguma forma de inteligência, como um microcontrolador, por exemplo, a fim de atingir seus objetivos mecânicos. Entretanto, embora o microcontrolador possa prover instruções para o motor, não

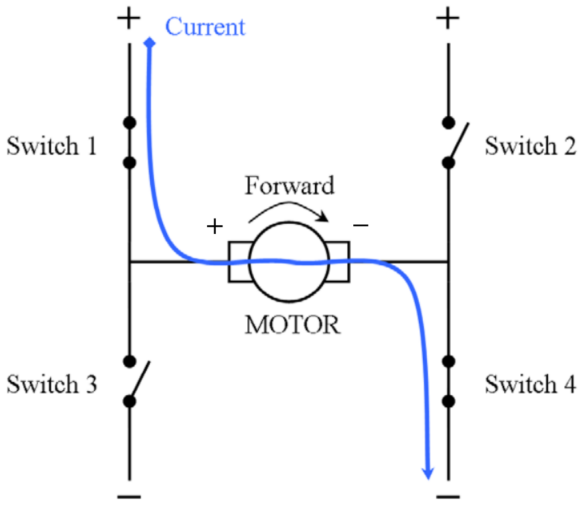


Figura 2.10: Ponte H ativa no sentido direto do motor (Fonte: SIEBEN, 2003).

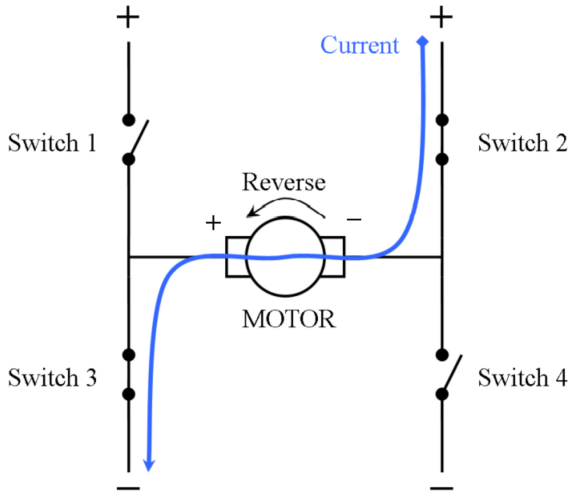


Figura 2.11: Ponte H ativa no sentido reverso do motor (Fonte: SIEBEN, 2003).

pode prover a potência necessária para acioná-los. Porém, ao substituir a chave mecânica pela eletrônica, utilizando um transistor bipolar de junção (TBJ), por exemplo, é possível utilizar a ponte H enviando um nível baixo de tensão na base do dispositivo para controlar um nível mais alto de tensão e ainda garantir o isolamento do controle (SIEBEN, 2003).

Comercialmente é possível encontrar CIs com pontes H que utilizam nível lógico TTL em suas entradas, que reconhece como nível lógico 0 as tensões entre 0 e 0,8 V e como nível lógico 1 as tensões entre 2,4 e 5 V, sendo assim compatível com as

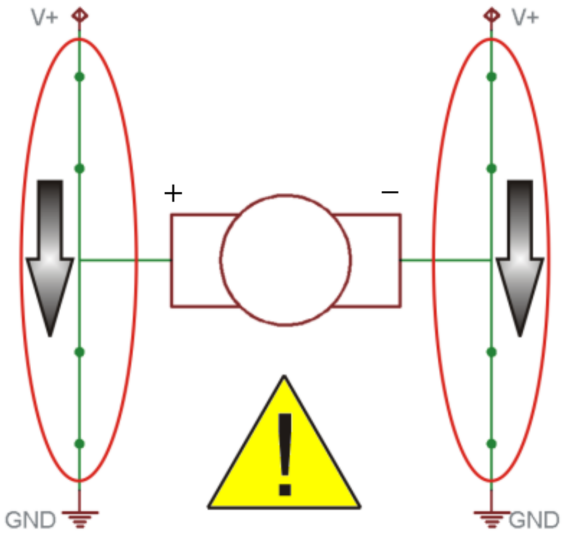


Figura 2.12: Caso proibido em uma ponte H (Fonte: PATSKO, 2006).

saídas dos microcontroladores que estão normalmente entre 3,3 e 5 V. A Figura 2.13 mostra a faixa de tensão do nível lógico da família TTL (BRAGA, 2022).

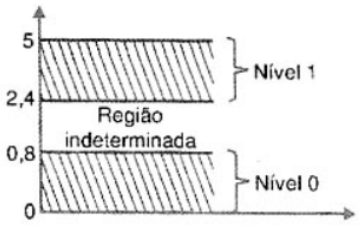


Figura 2.13: Faixas de tensão para o nível lógico TTL (Fonte: BRAGA,2022)

2.4.1 L298N

O CI L298N é uma ponte H de dois canais com tensão de alimentação de até 46 V e corrente de saída de até 2 A por canal, projetada para acionar cargas indutivas como relés, solenoides, motores de corrente contínua, entre outros. As entradas de controle são projetadas para funcionar com o nível lógico TTL (STMICROELECTRONICS, 2000). Um módulo que implementa o circuito básico de funcionamento do CI L298N pode ser encontrado facilmente no mercado e é apresentado na Figura 2.14

As entradas *IN1* e *IN2* são responsáveis por controlar as saídas *OUT1* e *OUT2*,

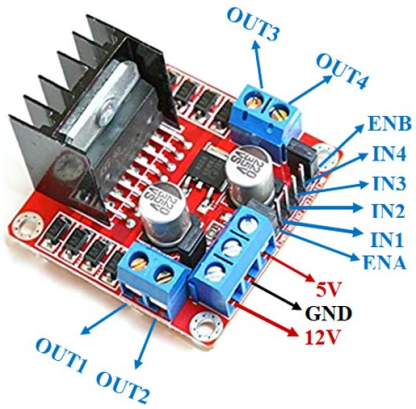


Figura 2.14: Módulo L298N (Fonte: COMPONENTS101, 2021)

enquanto o pino *ENA* permite o controle PWM para a saída, podendo regular o nível de tensão para a carga conectada às saídas. Analogamente, as entradas *IN3* e *IN4* são responsáveis por acionar as saídas *OUT3* e *OUT4* e o pino *ENB* faz o controle dessas saídas. A alimentação das saídas é feita a partir do pino de 12 V, enquanto o pino de 5 V é a saída do regulador de tensão, utilizando o CI 7805 (ver Seção 2.25) imbutido ao módulo, responsável por fazer a alimentação da parte lógica do CI L298N (COMPONENTS101, 2021).

2.5 Comunicação I^2C

Para a comunicação entre alguns dos dispositivos de entrada e saída do sistema implementado neste trabalho, utiliza-se a interface de comunicação I^2C. O barramento I^2C é uma interface bidirecional muito popular e poderosa utilizada para a comunicação entre um ou mais controladores, conhecidos como mestres, e um ou mais periféricos, chamados de escravos, empregando apenas dois fios para interface física da comunicação (VALDEZ e BECKER, 2018). A Figura 2.15 exemplifica um barramento I^2C em que um microcontrolador ou processador controla diversos dispositivos periféricos.

A camada física da comunicação consiste de duas linhas, chamadas de *serial clock* (SCL) e *serial data* (SDA), que se referem ao barramento de relógio e de

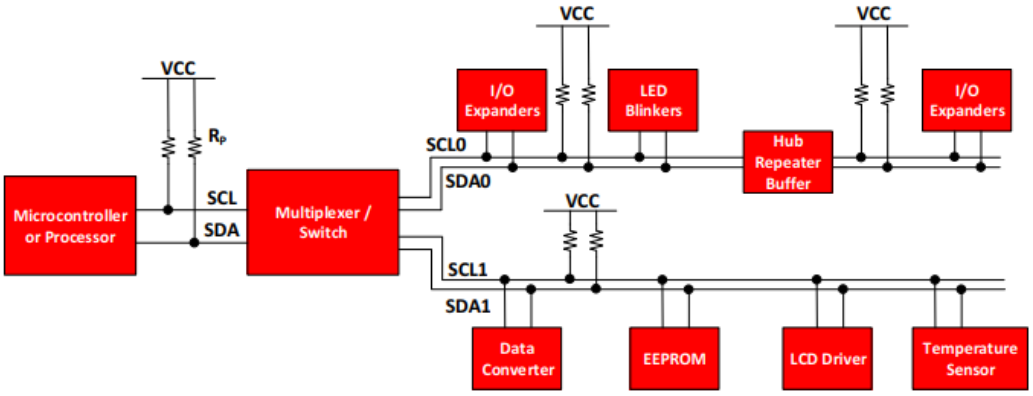


Figura 2.15: Exemplo de um barramento I^2C típico de um sistema embarcado. O microcontrolador representa o mestre I^2C que controla os expansores de entrada e saída, sensores variados, memória EEPROM, ADCs/DCAs, etc. (Fonte:VALDEZ e BECKER, 2018)

dados, respectivamente. Ambas devem ser conectadas a um resistor de *pull-up*[1] para garantirem um nível lógico alto de referência. Além disso, os dispositivos conectados ao barramento devem seguir a configuração de saída de dreno aberto (*open-drain*)[2], desse modo todos os dispositivos podem colocar o nível lógico para baixo, sem danificar os demais periféricos conectados(VALDEZ e BECKER, 2018).

As mensagens transmitidas são iniciadas com a condição START, que ocorre quando a linha SDA transiciona do nível alto para o nível baixo, enquanto a linha SCL ainda estiver em nível lógico alto, e são finalizadas com a condição STOP, que ocorre quando a linha SDA transiciona do nível lógico baixo para alto enquanto a linha SCL estiver em nível lógico alto. A Figura 2.16 exibe as condições de START e STOP.(VALDEZ e BECKER, 2018).

Para a transmissão dos *bits* de dados, a linha SDA mantém o estado estável enquanto o SCL está em nível lógico alto, por exemplo, se SDA for 0 no momento em que SCL estiver em 1, o valor do *bit* lido será 0. Ao final da mensagem o mestre aguarda um *bit* de reconhecimento (ACK), para garantir que o *byte* enviado foi

[1]Resistor utilizado para garantir um nível de tensão conhecido, especificamente um nível lógico alto quando se trata de um *pull-up*, ou um nível lógico baixo, quando se trata de um *pull-down* (SPARKFUN, 2022)

[2]Refere-se a um tipo de saída em que se pode forçar o barramento para uma tensão mais baixa, normalmente o terra, ou "soltar"o barramento para que a tensão volte para a referência fornecida por um resistor de *pull-up*(VALDEZ e BECKER, 2018)

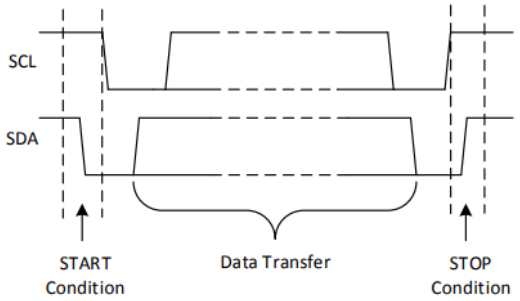


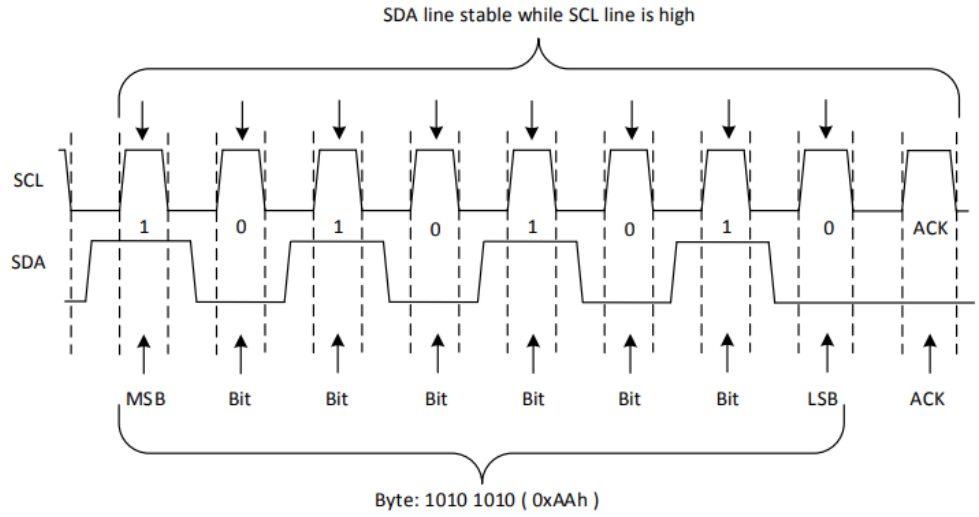
Figura 2.16: Condições de START e STOP da comunicação I^2C, nota-se as transições da linha SDA enquanto a linha SCL está em nível lógico alto (Fonte: VALDEZ e BECKER, 2018

recebido pelo escravo. Para isso o mestre põe a linha SDA para baixo e em seguida libera o controle, caso o sinal permaneça em nível lógico baixo, o escravo atuou e reconheceu os dados recebidos, caso contrário o sinal volta para alto, simbolizando um não reconhecimento (NACK) pelo escravo (VALDEZ e BECKER, 2018). As Figuras 2.17(a) e 2.17(b) mostram o envio de um *byte* com reconhecimento (ACK) e sem reconhecimento (NACK), respectivamente.

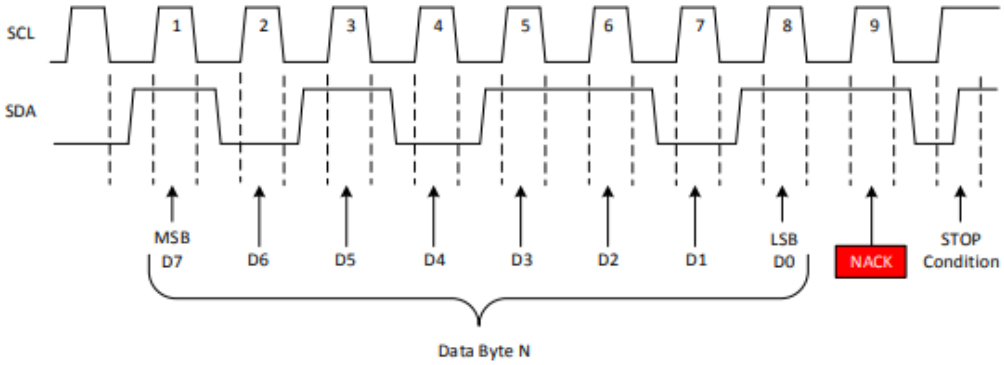
Em uma operação típica da interface, duas situações são possíveis: ou o mestre envia uma mensagem para o escravo ou o mestre lê dados do escravo. No primeiro caso, o mestre atua como transmissor e manda uma condição de START, envia o endereço do escravo que deve atender a mensagem, envia os dados ao escravo e finaliza a transmissão com uma condição de STOP (VALDEZ e BECKER, 2018).

Por outro lado, se o mestre desejar ler algum dado do escravo, inicialmente o mestre receptor envia a condição START e endereça ao escravo transmissor com um *bit* de leitura ($R/\overline{W} = 0$) ao final do endereço, em seguida requisita o registrador a ser lido no escravo, aguarda o dado transmitido pelo escravo e finaliza a comunicação, enviando a condição de STOP (VALDEZ e BECKER, 2018).

A comunicação I^2C é amplamente explorada neste trabalho como interface de comunicação e controle entre o microcontrolador e módulos periféricos, como por exemplo, o LCD e o RTC, a fim de diminuir a quantidade de pinos utilizados pelo microcontrolador.



(a) *Byte* transmitido com confirmação de recebimento.



(b) *Byte* transmitido sem confirmação de recebimento.

Figura 2.17: Transferência de um *byte* com confirmação de recebimento pelo escravo (bit ACK) e sem confirmação de recebimento (bit NACK).(Fonte:VALDEZ e BECKER, 2018

2.6 Módulo LCD

Segundo BARBACENA e FLEURY (1996), os módulos LCD (sigla em inglês para *display* de cristal líquido) são interfaces de saída muito úteis em sistemas microprocessados. Entre os mais comuns estão os LCDs do tipo caractere que podem ser especificados em número de linhas e colunas disponíveis, em que cada caractere é formado por uma matriz de 5x8 *pixels*. Os autores ainda apotam que a quantidade de pinos utilizados variam de 14 a 16 pinos e podem ser encontradas versões com *LED backlight* (iluminação de fundo), para facilitar leituras em lugares com baixa luminosidade.

Diversos autores apontam a utilidade dos *displays* para a construção de projetos de baixo custo, como é o caso de JACINTO (2020), que utilizou um módulo com dimensões de 20 colunas por 4 linhas (20x4), para exibição local de dados de temperatura e umidade e dados elétricos, como tensão, corrente e potência de um ambiente industrial. Já MANGUEIRA (2022) utilizou o *display* de 16x2 para exibição de dados de forças e deslocamentos em uma viga biapoiada, enquanto AKINWOLE (2020) fez a utilização para exibição dos níveis de água em um controle automático de água em um tanque.

Neste trabalho utiliza-se um *display* de dimensões 16x2, totalizando 32 caracteres, idêntico ao apresentado na Figura 2.18 e faz-se necessário a investigação dos pinos e conexões do dispositivo. A Tabela 2.3 exibe cada um dos pinos do dispositivo e faz uma breve descrição de suas funções.

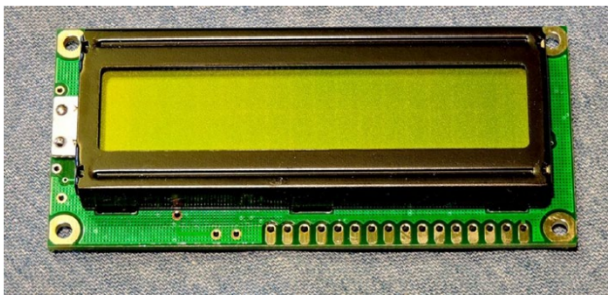


Figura 2.18: Módulo LCD de 16x2. (Fonte: GAY, 2017)

A necessidade de uma grande quantidade de portas de entrada e saída (I/O) do microcontrolador para atuar o *display* torna seu uso dificultoso (GAY, **?**), entretanto a utilização de um módulo de interface serial I^2C, segundo apontado por AKINWOLE (2020), reduz a complexidade do circuito e permite a diminuição de seis pinos de I/O para apenas dois pinos do barramento I^2C, SDA e SCL.

O módulo baseado no chip PCF8574 é apresentado na Figura 2.19, que é definido pelo fabricante como um expansor de portas de 8-*bits* para o barramento I^2C (INSTRUMENTS, 2015). Na parte inferior da placa mostrada na Figura 2.19, ainda é possível ver três espaços para conexões *jumpers*, marcados como $A0$, $A1$ e $A2$, que funcionam com um seletor de endereços I^2C. Caso nenhum fio seja instalado,

Pino	Função	Descrição
1	Alimentação	GND (terra)
2	Alimentação	VCC (+5 V)
3	V0	Tensão para ajuste de contraste
4	RS Seletor	1 - Dado, 0 - Instrução
5	R/W Seletor	1 - Leitura, 0 - Escrita
6	E Chip Select	1 ou (1 para 0) - Habilita, 0 - Desabilitado
7	B0 LSB	Barramento de dados
8	B1	
9	B2	
10	B3	
11	B4	
12	B5	
13	B6	
14	B7 MSB	
15	A	Anodo para LED backlight, se exister
16	B	Catodo para LED backlight, se exister

Tabela 2.3: Pinagem do módulo LCD 16x2 (Fonte: BARBACENA e FLEURY, 1996)

o módulo apresentará um endereço hexadecimal no barramento I^2C de 27 (GAY, 2017).

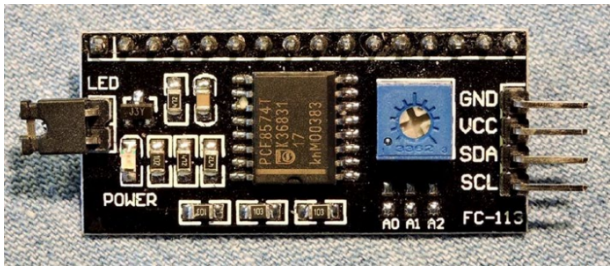


Figura 2.19: O módulo de interface serial I^2C com o chip PCF8574 no centro. (Fonte: GAY, 2017)

O pino 1 do cabeçalho de 16 pinos, na parte superior do módulo da Figura 2.19, encontra-se no lado direito, apesar da camada *silk* parecer indicar o contrário com a marcação do lado esquerdo. O mesmo cabeçalho é apresentado no diagrama elétrico do módulo exibido na Figura 2.20 nomeado como *JP1* e deve ser conectado ao módulo LCD. O diagrama elétrico ainda detalha as conexões internas entre o CI PCF8574, o cabeçalho e os pinos do barramento I^2C (GAY, 2017).

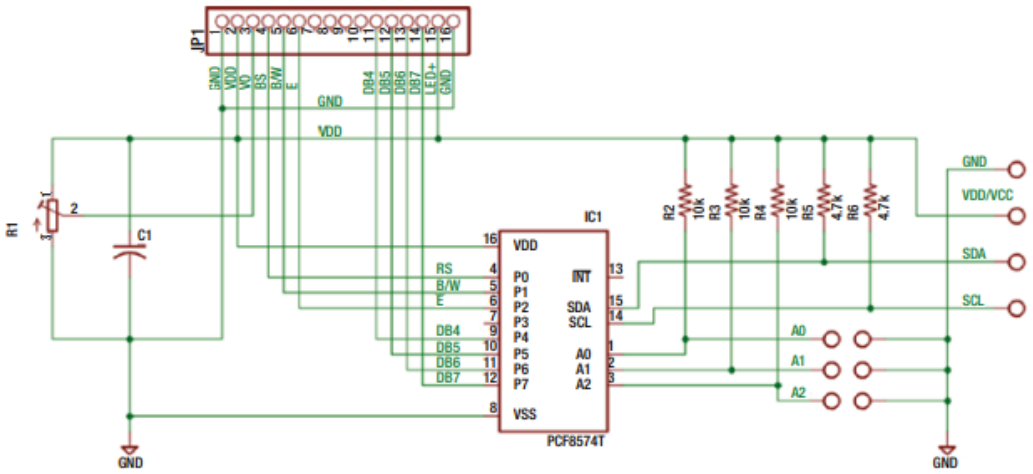


Figura 2.20: Diagrama elétrico do módulo I^2C com as conexões entre o CI PCF8574, o cabeçalho do módulo LCD e as conexões do barramento I^2C (SDA e SCL). (Fonte: Gay, 2017)

2.7 Relógio de tempo real DS3231

Um relógio de tempo real (RTC, sigla em inglês para *real-time clock*) é um dispositivo eletrônico projetado na forma de um circuito integrado para medir a passagem real do tempo, possibilitando uma referência precisa, mesmo quando a energia principal do sistema é desligada (HISTORY-COMPUTER,2022).

Diversos trabalhos utilizam alternativas para o controle do tempo real em conjunto com microcontroladores, como é o caso da utilização do módulo RTC DS3231, nos projetos de MNATI *et al.* (2021), em que o módulo é utilizado na construção de um termômetro sem contato e de baixo custo, e de KURIA *et al.* (2020), em um *datalogger* para monitoramento de temperatura e umidade.

O módulo DS3231, mostrado na Figura 2.21, é um CI de relógio de tempo real de 8 pinos, com suporte para manter informações de segundos, minutos, horas, dias, data, mês e ano. O ajuste das datas em meses com menos de 31 dias é feito de forma automática, inclusive para anos bissextos. O acesso aos registradores internos do dispostivo são disponibilizados por meio de uma interface I^2C bidirecional. O módulo também conta com um circuito comparador, que monitora o nível da tensão de alimentação V_{CC} para detectar falhas e chavear automaticamente para uma ba-

teria do modelo 2032 de 3 V, acoplada ao módulo (MAXIM, 2015).

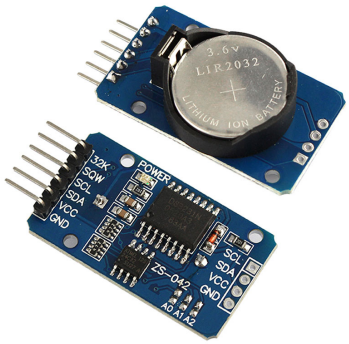


Figura 2.21: Módulo RTC DS3231 com a bateria do modelo 2032 de 3V acoplada e o CI com os cabeçalhos de pinos. (Fonte: ELECTROBIST, 2022

2.8 Microcontrolador ESP32

O ESP32 é um microcontrolador poderoso com WiFi e BLE (*Bluetooth Low Energy*) embutidos, projetado como uma solução para o desenvolvimento de dispositivos IoT (*Internet of Things*, ou internet das coisas, em português). O dispositivo no modelo ESP-WROOM-32 conta com componentes integrados como antena, oscilador e memória *flash*, facilitando a prototipação e salvando espaço na placa de circuito impresso (PCB, sigla do inglês para *Printed Circuit Board*). Há ainda versões de desenvolvimento, como o ESP32-DevKit, mostrado na Figura 2.22, que apresenta uma solução pronta para uso de testes e propósitos educacionais, além uma topologia amigável a uma matriz de contatos (MAIER *et al.*, 2017).

O ESP32 é um sistema com dois processadores Xtensa LX6 com arquitetura Harvard, memória ROM embutida de 448 kB, memória SRAM de 520 KB e duas memórias RTC de 8 kB e a memória externa pode chegar até 16 MB de memória *flash*. O microcontrolador ainda usa um relógio de até 160 MHz. Há quatro temporizadores de uso geral de 64-*bits*, com uma pré-escala de 16-*bits*, com valores na faixa de 2 a 65536. Cada temporizador usa um relógio de 80 MHz e pode contar

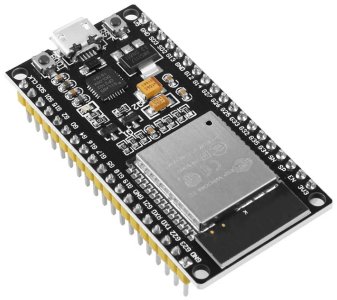


Figura 2.22: Módulo ESP32-DevKit. (Fonte: LEANTEC)

para cima, para baixo, pausar ou acionar eventos. (MAIER *et al.*, 2017).

O microcontrolador ainda conta com uma estrutura projetada para trabalhar com comunicação Wi-Fi e *Bluetooth Low Energy* (BLE). As entradas e saídas incluem dois ADCs (*Analog Digital Converters*) de 12-*bits*, somando 18 canais, dois DACs (*Digital Analog Converters*) de 8-*bits* para conversão de sinais digitais em saídas de tensão, dez do total de GPIOs são capazes de detectar variações de capacitância que podem ser usados como sensores de toque. Além disso, o dispositivo conta com diversas interfaces de comunicação como uma interface de *Ethernet*, três interfaces UARTs de até 5 Mbps, dois barramentos I^2C com frequências de 10 kHz até 10 MHz, pinos de entrada e saída com PWM (*Pulse Width Modulation*) (MAIER *et al.*, 2017).

A linguagem de programação usual do ESP32 é o C, uma vez que diversas bibliotecas são disponibilizadas em C. Entretanto, o microcontrolador pode ser facilmente programado em C++, por exemplo, utilizando bibliotecas do Arduino, com as devidas modificações que possam ser necessárias(MAIER *et al.*, 2017). Ainda é possível realizar a programação do ESP32 a partir da IDE do Arduino (Seção 2.8.2) após a configuração apropriada (SOUZA, 2022).

2.8.1 Linguagens de programação C/C++

C é uma linguagem de programação simples e pequena, podendo ser traduzida por compiladores também pequenos. Diversos aspectos podem ser associados à

linguagem, como os tipos e operações que estão próximas às máquinas reais, ao mesmo tempo em que a linguagem é suficientemente abstrada para que os detalhes da máquina não sejam um empecilho para a portabilidade entre sistemas(RITCHIE, 1993).

Segundo KORMANYOS (2021), C++ é uma linguagem de programação orientada a objetos moderna baseada em C, de modo que a maioria das construções de C estão em C++, salvo algumas poucas exceções. A utilização do C++ na programação de microcontroladores tem ganhado popularidade, uma vez que a comunidade de sistemas embarcados tem adquirido maturidade nos métodos de uso da linguagem.

Atualmente C e C++ permanecem relevantes, como aponta o levantamento do IEEE Spectrum's para 2022, em que as linguagens estão atrás somente da linguagem *Python*, porém apresentam o diferencial de serem mais velozes e flexíveis, sendo amplamente utilizadas para sistemas embarcados e computação de alto desempenho (CASS, 2022).

2.8.2 Plataforma Arduino

Arduino é uma plataforma de *hardware* e *software* de código aberto, criada inicialmente para artistas, *designers* e outros grupos que desejavam incorporar computação em suas criações, sem a necessidade de um amplo conhecimento em engenharia elétrica (BANZI e MICHAEL, 2022). Dessa forma, foi construído um ambiente intuitivo e prático de desenvolvimento, em que é possível utilizar uma versão simplificada do C++ para construir os códigos que podem ser facilmente carregados para o módulo, utilizando uma interface USB (BADAMASI, 2014). Segundo BOXALL (2013), um programa criado utilizando Arduino é comumente chamado de *sketch*.

O *software* utilizado para a programação do Arduino é chamado de Ambiente de Desenvolvimento Integrado (IDE, sigla em inglês para *Integrated Development Environment*) e promove um conjunto de ferramentas para programação do *hardware*(BADAMASI, 2014).

Segundo BOXALL(2013), a IDE pode ser dividida em três partes principais: a área de comando, a área editor de texto, e a área janela de mensagens. A área de comandos pode ser vista na parte superior da Figura 2.23 e conta com um menu de itens como: Arquivo, com opções para salvar, carregar e imprimir *sketches*, assim como um submenu de preferências; Editar, com opções de copiar, colar, entre outras funcionalidades de editores; *Sketch*, contém funções para verificação do programa antes de enviar para a placa, algumas pastas do *sketch* e opções de importação; Ferramentas, que contém uma variedade de funções para seleção do tipo de placa utilizada, detalhes da comunicação do dispositivo, a porta USB a ser utilizada, entre outros; Ajuda, que contém diversos *links* de interesse e a versão da IDE.

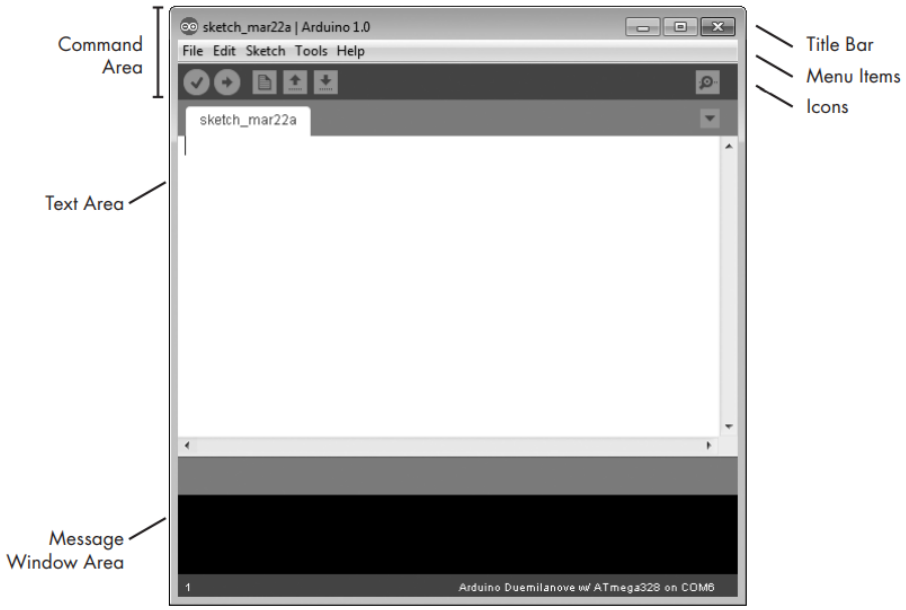


Figura 2.23: Ambiente de Desenvolvimento Integrado (IDE) do Arduino. (Fonte: BOXALL, 2013)

Dentre os ícones, localizados abaixo da barra de menu na Figura 2.23, podem ser visualizados da esquerda para direita: o ícone de Verificação, para verificação do código criado, compilando-o e verificando se é válido ou não; o ícone de Carregamento, que também verifica o código e envia para placa; Novo, para abrir um novo *sketch*; Abrir, para abrir um *sketch* salvo; Salvar, para salvar o *sketch* aberto; Monitor Serial, para abrir uma janela para o envio e recebimento de dados da placa

a partir da IDE (BOXALL, 2013).

A área de editor de texto, localizada na parte central da Figura 2.23, é o espaço em que se deve realizar a programação do dispositivo a partir dos códigos escritos. Sobretudo, na escrita do código utilizando o Arduino, há duas partes principais, o *setup*(configurações) e o *loop*. O *setup* é executado apenas uma vez na inicialização do programa, e deve ser utilizado para prover as condições iniciais das variáveis e das condições iniciais do sistema, O *loop*, por sua vez, trata-se de um laço infinito no qual o código é executado repetidamente (BADAMASI, 2014).

Por último, na parte inferior da Figura 2.23 está a área da janela de mensagens, que mostra as mensagens da IDE, como as informações de compilação, verificação, atualização de estado e erros nos *sketches*(BOXALL, 2013).

2.9 Regulador de tensão

Os circuitos elétricos e eletrônicos precisam de características elétricas, como tensão e corrente, específicas para o seu correto funcionamento. Tendo em vista que a maioria dos dispositivos eletrônicos são sensíveis a variação de tensão CC, necessitando de um valor constante para a operação, é comum a necessidade dos circuitos reguladores de tensão (BOYLESTAD e NASHELSKY, 2012). Ainda segundo BOYLESTAD e NASHELSKY (2012), a regulação é geralmente feita por um CI regulador de tensão, que recebe uma tensão CC e fornece um valor menor, garantindo uma tensão constante para a carga, independentemente das variações da tensão de entrada, ou mesmo variações da própria carga.

O fator de regulação de tensão (VR) é a variação da tensão CC de saída relativa às condições de trabalho quando se drena (com carga) ou não se drena (sem carga) corrente da fonte. O VR pode ser quantificado de acordo com a Equação 2.1, em que $\%VR$ é a regulação de tensão em porcentagem, V_{NL} é a tensão sem carga e V_{FL} é a tensão com carga máxima (BOYLESTAD e NASHELSKY, 2012).

$$\%VR = \frac{V_{NL} - V_{FL}}{V_{FL}} \times 100\% \tag{2.1}$$

Os CIs reguladores contêm, em um mesmo dispositivo, os circuitos da fonte de referência, o amplificador comparador, o dispositivo de controle e a proteção contra sobrecarga. O circuitos oferecem regulação para uma tensão positiva fixa, uma tensão negativa fixa ou uma tensão ajustável (BOYLESTAD e NASHELSKY, 2012). A Figura 2.24 exibe a conexão básica de um CI regulador de três terminais a uma carga.

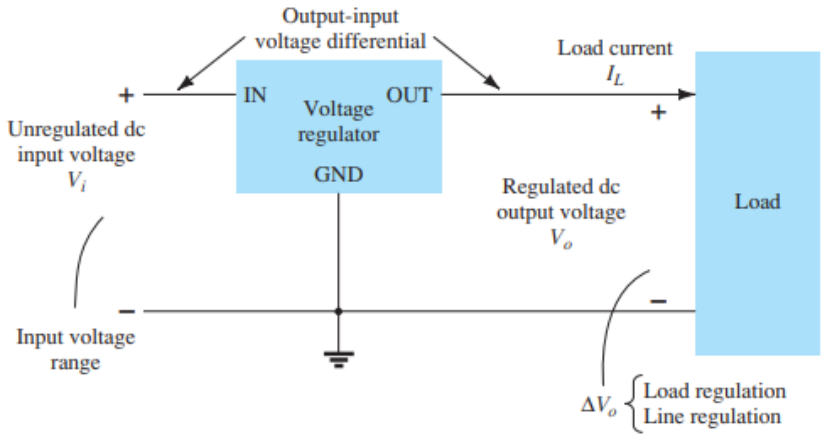


Figura 2.24: Estrutura básica para um regulador de três terminais. (Fonte: BOYLESTAD e NASHELSKY, 2012)

Para esse tipo de regulador, uma entrada de tensão CC não regulada, V_i, é aplicada a um terminal de entrada, uma tensão de saída CC regulada, V_o, é aplicada em um segundo terminal, enquanto o terceiro terminal é conectado ao terra (BOYLESTAD e NASHELSKY, 2012). Para a aplicação neste trabalho, vale destacar a operação dos reguladores de tensão positiva fixa e os de tensão ajustável.

2.9.1 Regulador de tensão positiva fixa

Uma família de CIs muito utilizados para a regulação de tensão fixa consiste na série 78 que possui dispositivos de saídas de 5 V até 24 V. O CI 78M05 possui saída

fixa de +5 V, uma tensão de *dropout* [1] típica de 1,7 V e uma corrente de saída de até 500 mA. A regulação de tensão na carga é normalmente de 25 mV, podendo chegar até 100 mV para correntes próximas do máximo (TOSHIBA, 1999). O circuito típico de aplicação é apresentado na Figura 2.25.

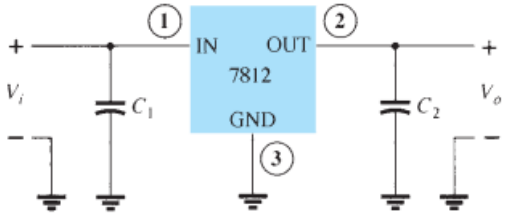


Figura 2.25: Circuito de conexões do CI da família 78MXX. (Fonte: BOYLESTAD e NASHELSKY, 2012)

2.9.2 Regulador de tensão ajustável

São chamados de reguladores de tensão ajustáveis, os reguladores de tensão disponíveis em configurações que permitem ao usuário ajustar a tensão de saída para um valor desejado (BOYLESTAD e NASHELSKY, 2012). O LM2596, por exemplo, permite o ajuste de tensão de 1,2 V a 37 V funcionando como um regulador abaixador (do inglês *step-down* ou *buck*) e fornece uma corrente de carga de até 3 A com excelentes valores de regulação de linha (variação da saída com relação às variações na entrada) e de carga. O regulador opera em uma frequência alta de chaveamento, em torno de 150 kHz, resultando em tamanhos menores nos componentes de filtragem, como o capacitor, do que se comparado com reguladores com frequências de chaveamento mais baixas (ONSEMI, 2008). Na Figura 2.26 é mostrado o CI com 5 pinos e encapsulamento do tipo TO-263.

O pino 1 do dispositivo, V_{in}, é a fonte de alimentação positiva do dispositivo, para uma correta regulação uma tensão de *dropout* em torno de 2 V deve ser respeitada para uma saída de 3 A. O pino 2 corresponde a saída regulada do dispositivo, enquanto o pino 3 é o pino de terra do circuito. O pino 4 é chamado de pino de

[1] Valor mínimo de tensão entre os terminais de entrada-saída que deve ser mantido para que o CI possa operar garantidamente como um regulador (BOYLESTAD e NASHELSKY, 2012).

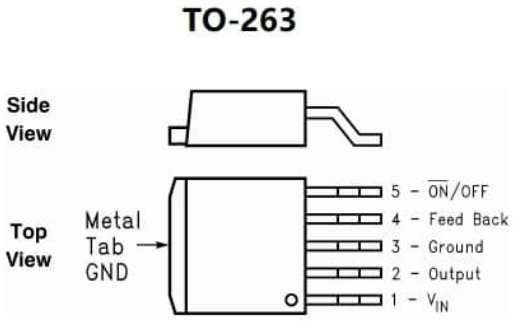
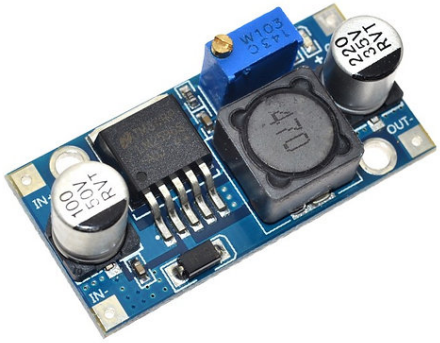


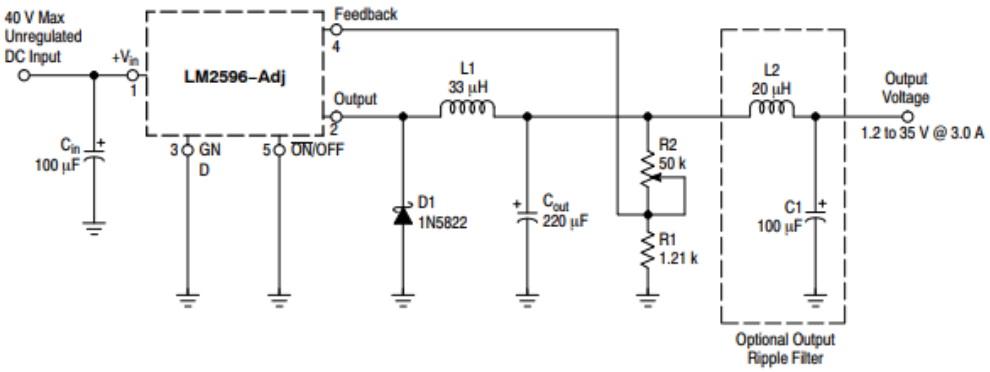
Figura 2.26: Pinagem do circuito LM2596 com encapsulamento TO-263. (Fonte: INSTRUMENTS, 2021)

feedback e está conectado diretamente ao amplificador de erro e a rede com resistores ajustáveis, R_1 e R_2, que possibilitam a configuração externa da tensão de saída. Por último, o pino 5 pode habilitar a saída do dispositivo, caso seja mantido em nível lógico baixo, ou desabilitar, caso seja mantido em nível lógico alto (ONSEMI, 2008).

Os módulos que implementam um regulador de tensão ajustável básico para esse dispositivo são comuns, como o módulo LM2596 utilizado em KHAING *et al.* e AMANPREET SINGH. A Figura 2.27(a) exibe o módulo, enquanto a Figura 2.27(b) exibe o diagrama elétrico do circuito implementado no módulo. É possível observar que um resistor variável, R_2, é colocado para realizar a excursão do sinal de saída.



(a) Módulo LM2596



(b) Circuito do módulo LM2596 com saída ajustável de 1,2 a 35 V e 3 A

Figura 2.27: Módulo implementado com LM2596 e seu diagrama elétrico. (Fonte: ONSEMI, 2008)

Capítulo 3

Desenvolvimento

3.1 Requisitos do sistema

Como parte essencial para o cumprimento dos objetivos geral e específicos do funcionamento do sistema, estabelece-se como primeiro requisito que ele deve funcionar com condições de falta de energia elétrica, uma vez que é possível encontrar no campo diversas instalações remotas, que muitas vezes não possuem rede concessionária de energia elétrica disponível, como é o caso da EECAC (Estação Experimental de Cana-de-Açúcar) da Universidade Federal Rural de Pernambuco (UFRPE). Dentre os dispositivos de maior consumo para o sistema, destaca-se a válvula solenoide, que comercialmente apresenta modelos de atuação contínua, que devem estar ligados durante todo o tempo de utilização, portanto, as válvulas solenoides do tipo *latching* são utilizadas como uma alternativa de baixo consumo.

Considerando a alimentação por baterias, o segundo requisito deriva do primeiro, a corrente instantânea (também conhecida como corrente *inrush*) do sistema não pode ser alta. Portanto, os dispositivos solenoides não podem ser atuados de modo simultâneo para não danificar os demais componentes sensíveis a uma corrente na ordem de ampères.

O terceiro requisito é a utilização de uma interface simples para configuração das funcionalidades e operação do sistema. A interface de usuário (UI, sigla para *User Interface*), com botões e um *display* LCD, concentra as informações de configuração

e operação em uma única UI e segue um fluxo de atuação intuitivo, de acordo com a operação utilizada em campo pelos usuários do sistema.

Além da interface de configuração, o quarto requisito consiste na indicação visual do estado de atuação de cada uma das válvulas para aferir quais estão ativas no tempo especificado. Para isso, utilizam-se LEDs para a indicação.

Por último, antes da irrigação é preciso garantir que a rede hidráulica esteja pressurizada corretamente. Para isso, é necessário a implementação de uma rotina de pressurização que possa ocorrer paralelamente a configuração da irrigação e que seja pausada a qualquer momento pelo operador ou quando ocorrer o início da irrigação.

3.2 Hardware

Para a implementação do sistema desejado, a etapa de proposição e validação da arquitetura do *hardware* eletrônico a ser utilizado tem suma importância, uma vez que o *hardware* tem uma característica permanente no produto final, diferentemente da implementação do *firmware/software* que pode ser modificada. Portanto, o *hardware* precisa ser bem definido, dimensionado e testado, tanto separadamente, quanto em conjunto com os outros sistemas, para garantir a implementação correta e duradoura da placa final.

Ao invés da aquisição de componentes isolados, diversos módulos são utilizados, a fim de garantir agilidade na construção do projeto, i.e., os módulos adquiridos podem ser utilizados tanto na etapa de prototipação e teste, quanto na placa final de circuito construída. O diagrama da solução proposta é exibido na Figura 3.1.

Em suma, o microcontrolador ESP32 é responsável por todo o controle do sistema, recebendo dados de entrada, por meio dos botões e do RTC, e acionando as saídas. Estas, por sua vez, podem ser divididas em interfaces com o usuário operador (LEDs e LCD) e atuadores, que compreendem o conjunto de saída: registradores de deslocamento, pontes H e válvulas solenoides.

Além disso, cada dispositivo possui uma tensão de alimentação específica para

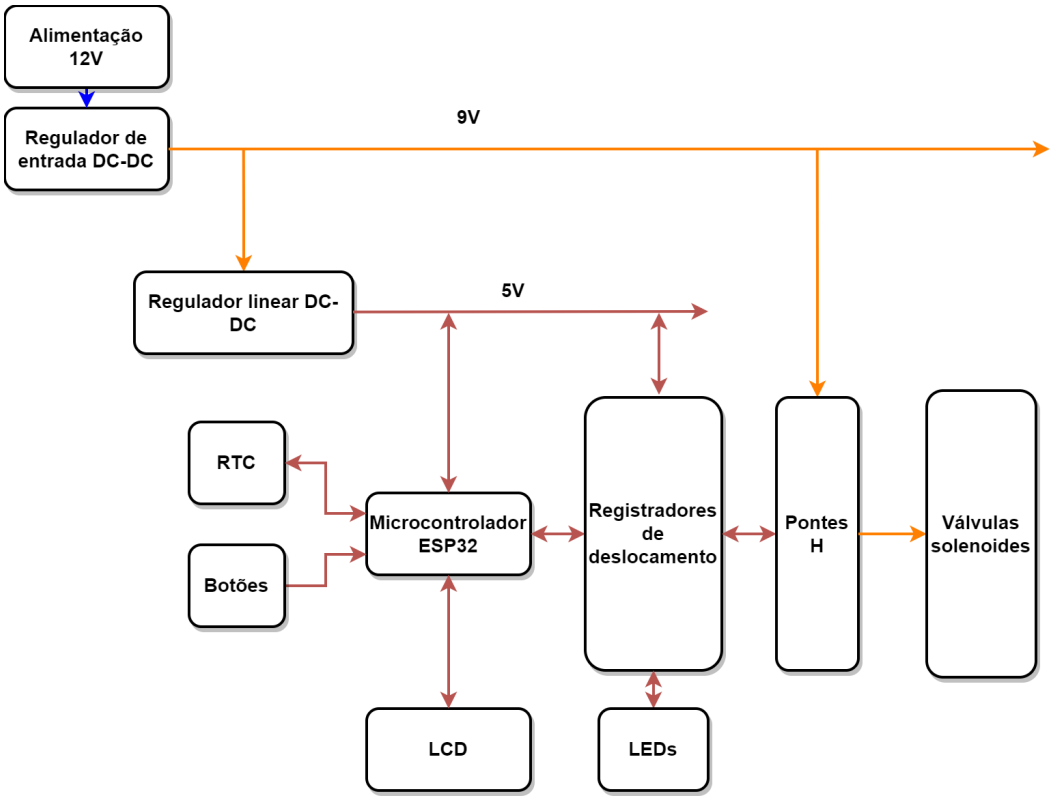


Figura 3.1: Arquitetura do *Hardware* proposta para o sistema implementado. (Fonte: o Autor)

operação, por isso, o sistema conta com dois reguladores. O primeiro consiste em um regulador de tensão ajustável abaixador, conforme descrito na Seção 2.9.2, que regula a tensão de entrada, que pode variar de 12 V até cerca de 37 V, para uma tensão constante de 9 V responsável pela alimentação das pontes H e consequentemente das válvulas solenoides. E o segundo consiste em um regulador linear encarregado de um outro estágio de conversão, transformando de 9 V para 5 V para alimentar os demais dispositivos.

Embora o microcontrolador tenha especial destaque no funcionamento do sistema, é necessário definir inicialmente os demais periféricos a fim de garantir sua operação e compatibilidade com o dispositivo de controle. Desse modo, o conjunto de saída é descrito inicialmente, começando pelas válvulas solenoides por serem o principal dispositivo de atuação do sistema. Posteriormente, a partir da corrente encontrada para o solenoide, pode-se definir a ponte H a ser utilizada e determinar a melhor forma de controlar os níveis lógicos de entrada desses dispositivos.

As demais entradas e os periféricos de interface com o usuário podem ser especificados independentemente, por estarem associados apenas por meio da lógica de programação e das interfaces de *hardware* presentes no microcontrolador. Uma vez que os demais componentes estão determinados, o microcontrolador pode ser avaliado no tocante à disponibilidade das entradas e saídas e das interfaces de comunicação com os módulos.

3.2.1 Especificação do solenoide

Tendo bem definidas as diferenças entre as válvulas solenoides explanadas na Seção 2.3, de acordo com os requisitos de alimentação elétrica do sistema, utiliza-se a válvula solenoide do tipo *latching*, com operação indireta por apresentar maior vazão e menor consumo de energia (uma vez que ela mantém o estado da última atuação, funcionando como uma espécie de memória mecânica para o sistema). Dessa forma, a energia gasta para a atuação da válvula ocorre apenas na comutação do estado.

Considerando, portanto, as dificuldades de operação com sinais negativos de tensão utilizando sistemas microcontrolados, opta-se pela utilização dos solenoides com 2 fios, permitindo a atuação com uma ponte H de maneira mais eficiente. Ou seja, com um sinal de controle sempre positivo em dois terminais da ponte H é possível controlar a polarização do solenoide a partir da combinação da entrada.

Para a especificação da tensão e da resistência da bobina do solenoide, o principal critério é a corrente instantânea na bobina no momento da comutação. Portanto, tendo em vista uma tensão de alimentação fixa, a resistência da bobina serve como critério de escolha, uma vez que quanto maior a resistência, menor também a corrente para uma mesma tensão.

Comercialmente é encontrado um solenoide da *RAIN-BIRD* de 9 V e aproximadamente 5 Ω de resistência, resultando, pela lei de Ohm, em uma corrente média de aproximadamente 1,8 A na ativação. O solenoide adquirido é mostrado na Figura 3.2.

Portanto, a ponte H responsável por acionar o solenoide deve suportar uma

Figura 3.2: solenoide *latching* da *RAIN-BIRD*. (Fonte: o Autor)

corrente de no mínimo 1,8 A. Supondo o acionamento das 10 válvulas de modo simultâneo, a corrente drenada da fonte pode alcançar um pico de 18 A, por isso, é necessário espaçar os pulsos de ativação dos solenoides, para garantir que o acionamento não ocorra de uma única vez. Segundo os fabricantes desses tipos de solenoides (Seção 2.3.1), o tempo de pulso mínimo para comutação é de 100 ms, dessa forma, um pulso de 200 ms é utilizado para garantir a comutação do estado do solenoide.

3.2.2 Especificação da ponte H

Obtendo a corrente necessária para ativação do solenoide, pode-se definir as especificações da ponte H para realizar a comutação. Desse modo, escolhe-se o módulo L298N, apresentado na Seção 2.4.1, que suporta uma corrente de saída de até 2 A por canal e apresenta diodos de proteção em sua construção, em caso de descarga da corrente indutiva do solenoide. O módulo conta com dois canais independentes, podendo ser utilizado para acionar dois dispositivos. Portanto, cinco módulos são utilizados para o acionamento das dez válvulas solenoides.

Como as tensões de entrada para acionamento da ponte H seguem o padrão da

$IN1$ (ou $IN3$)	$IN2$ (ou $IN4$)	$V_{OUT1} - V_{OUT2}$ (ou $V_{OUT3} - V_{OUT4}$)
0	0	$0V$
1	0	$+V_{CC}$
0	1	$-V_{CC}$
1	1	$0V$

Tabela 3.1: Resposta da saída de tensão do módulo L298N a partir dos níveis lógicos das entradas. (Fonte: o Autor)

família TTL, é possível utilizar as saídas do microcontrolador para fornecer os sinais de controle. Entretanto, tal abordagem consome vinte portas lógicas do microcontrolador para controle das dez válvulas solenoides. Diminuindo, assim, a quantidade de portas lógicas disponíveis do microcontrolador e dificultando o controle dos demais dispositivos. Sendo assim, um CI registrador de deslocamento é utilizado como um expansor de portas digitais (Seção 2.2.2).

3.2.3 Especificação do registrador de deslocamento

Os CIs registradores de deslocamento de 8-*bits* são utilizados em cascata como expansores de saídas digitais para o microcontrolador, assim, com apenas quatro dispositivos é possível obter 32 saídas digitais e utilizar inicialmente 3 pinos de saídas do microcontrolador como controle. Um pino de entrada serial (SI), um para o *clock* de entrada (SCK) e outro para controle de comutação da saída (RCK). Como os pinos de entrada e saída do SR operam com nível lógico TTL, uma tensão de 3.3 V na entrada, como é caso da saída do microcontrolador ESP32, é convertida em uma tensão de 5 V para a ponte H. Além disso, a corrente drenada pela entrada da ponte H também passa a ser fornecida pelo CI.

Apenas vinte portas são utilizadas para controle dos solenoides, restando doze portas da expansão para utilização em outras aplicações. Assim, outras dez portas são utilizadas para o acionamento dos LEDs indicadores de funcionamento dos solenoides presentes na interface de usuário. Desse modo, quando o comando de ativação é enviado para algum dos solenoides, a lógica de programação deve se encarregar de realizar o acionamento do LED correspondente, assim o operador poderá saber se o solenoide foi ativado ou não e conferir possíveis falhas.

Diante do exposto, o conjunto de saída conta com estes três principais dispositivos: o registrador de deslocamento, a ponte H e o solenoide. Em resumo, o microcontrolador comunica-se diretamente com os SR, colocando o valor de saída desejado em seus 32 registradores, a partir do controle de *SI* e *SCK*, e liberando a saída por meio do controle do pino *RCK*. Por sua vez, o nível lógico do SR é aplicado à entrada da ponte H, ativando a saída do solenoide. A Figura 3.3 exibe o diagrama dessa atuação, em que é possível ver o *bit32* com nível lógico 1, ativando o solenoide 1, e o *bit2* conectado ao LED9 também com nível lógico alto, indicando que o solenoide 9 está ativado.

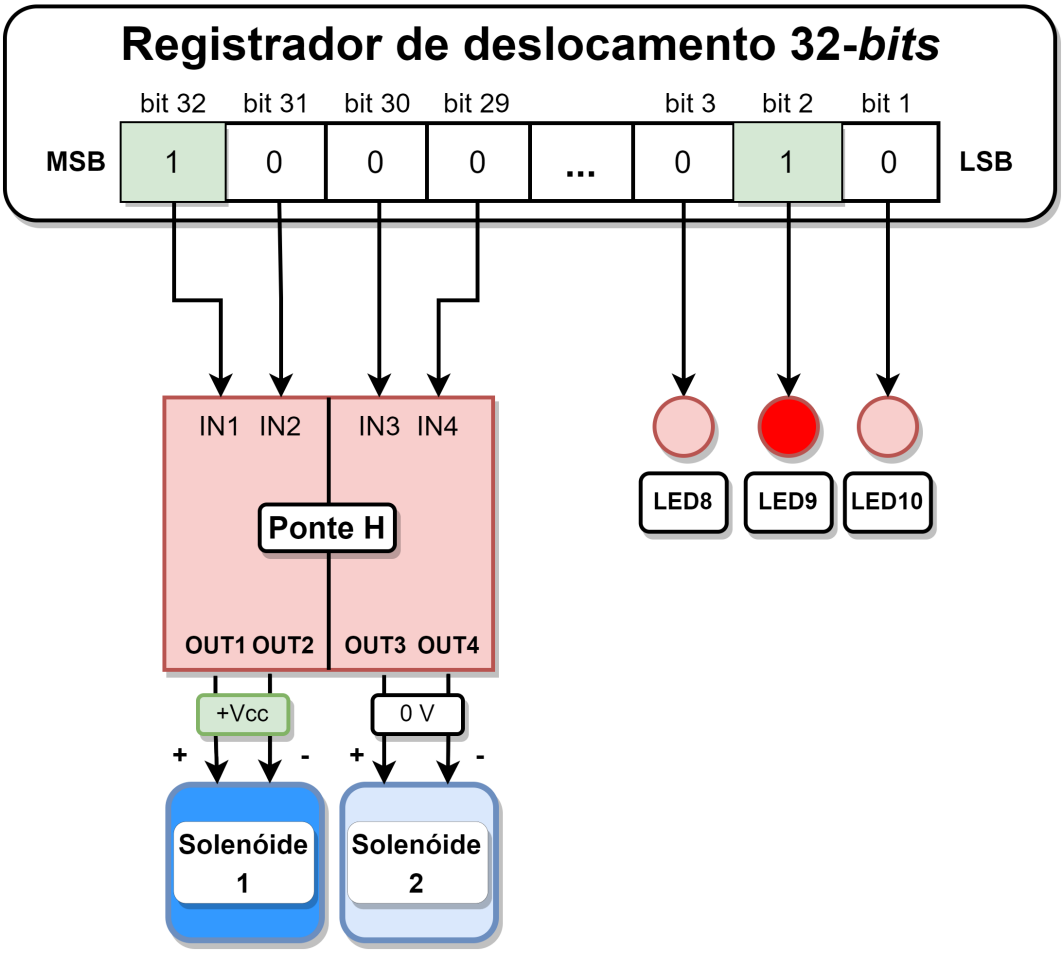


Figura 3.3: Diagrama de atuação dos solenoides a partir do comando enviado ao SR. (Fonte: o Autor)

3.2.4 Periféricos de interface com o usuário

Os periféricos que fazem a interface com o usuário operador do sistema são constituídos pelo *display* LCD, os botões e os LEDs indicadores de ativação das válvulas solenoides. Os botões e o LCD constituem a interface para configuração das rotinas de irrigação que o operador deseja programar.

Quatro botões são utilizados na placa, e sua funcionalidade varia dependendo do estado em que o sistema encontra-se na rotina de configuração, entretanto o funcionamento do ponto de vista elétrico é o mesmo. Um dos pinos dos botões está ligado diretamente na entrada digital do microcontrolador, configurada como entrada *pull-up*, enquanto o pino oposto é mantido em *GND*. Assim, quando o botão está em repouso, o nível lógico na entrada é naturalmente alto, e quando o botão é pressionado, um nível lógico baixo é enviado para o microcontrolador.

Para o controle do módulo LCD, utilizando a interface I^2C, apenas os pinos *SDA* e *SCL* são conectados ao microcontrolador. O módulo de interface I^2C também é conectado à alimentação de 5 V e ao *GND*, alimentando automaticamente o *display*.

3.2.5 Relógio de tempo real

A comunicação com o RTC escolhido, o DS3231, é feita a partir dos pinos de *SDA* e *SCL*, compartilhando a comunicação com o módulo LCD, como permite o protocolo de comunicação I^2C. Assim, é possível economizar GPIOs, concentrando os módulos em apenas um barramento. O módulo também é conectado à alimentação de 5 V e ao *GND*, possibilitando a sua alimentação enquanto o sistema estiver energizado. Para obter a referência de tempo, mesmo que o dispositivo esteja desligado, uma bateria de 3 V é acoplada ao módulo.

3.2.6 Microcontrolador

A Figura 3.4 mostra o diagrama elétrico do microcontrolador ESP32, com todos os pinos que são utilizados, nomeados por rótulos. As portas P22 e P21 são conectadas aos pinos do barramento I^2C, SCL e SDA, respectivamente. Os botões

esquerdo (BTN_LEFT), direito (BTN_RIGHT) e de configuração (BTN_CONFIG), são conectados respectivamente aos pinos P4, P17 e P18.

Os pinos de controle dos registradores de deslocamento são P25 com o pino de controle da entrada serial (DATA_PIN), P26 com o pino de controle do *clock* (CLOCK_PIN), P27 com o pino de controle da saída (LATCH_PIN) e, por último, o pino P16 utilizado para controle da habilitação da saída dos registradores (ESP_OE). Além disso, estão disponíveis, para utilização em projetos futuros, um outro botão (BTN_TBD1) e a interface do barramento SPI, para controle de um módulo de cartão SD, com os rótulos SPI_SCK, SPI_MISO, SPI_MOSI e SPI_CS.

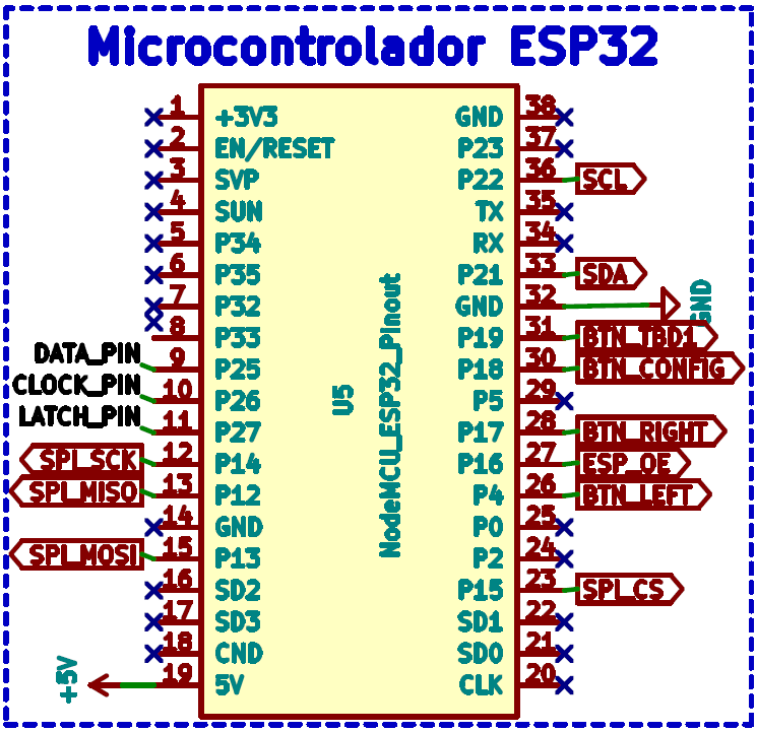


Figura 3.4: Diagrama elétrico do microcontrolador ESP32.(Fonte: o Autor)

3.2.7 Diagrama elétrico do Circuito

O diagrama elétrico do circuito reúne todas as implementações de *hardware* discutidas. Juntamente com o diagrama do ESP32, apresentado na Figura 3.4, podem-se destacar as ligações do conjunto de saída, as conexões com os módulos periféricos e a placa de interface desenvolvida separadamente.

O circuito do conjunto de saída para os registradores é apresentado na Figura 3.5, utilizando apenas um registrador capaz de controlar duas pontes H e, consequentemente, quatro válvulas solenoides. O primeiro registrador de deslocamento apresentado faz a conexão com o segundo por meio da saída serial, com trilha nomeada de SER2, o que ocorre sucessivamente até o quarto registrador de deslocamento utilizado.

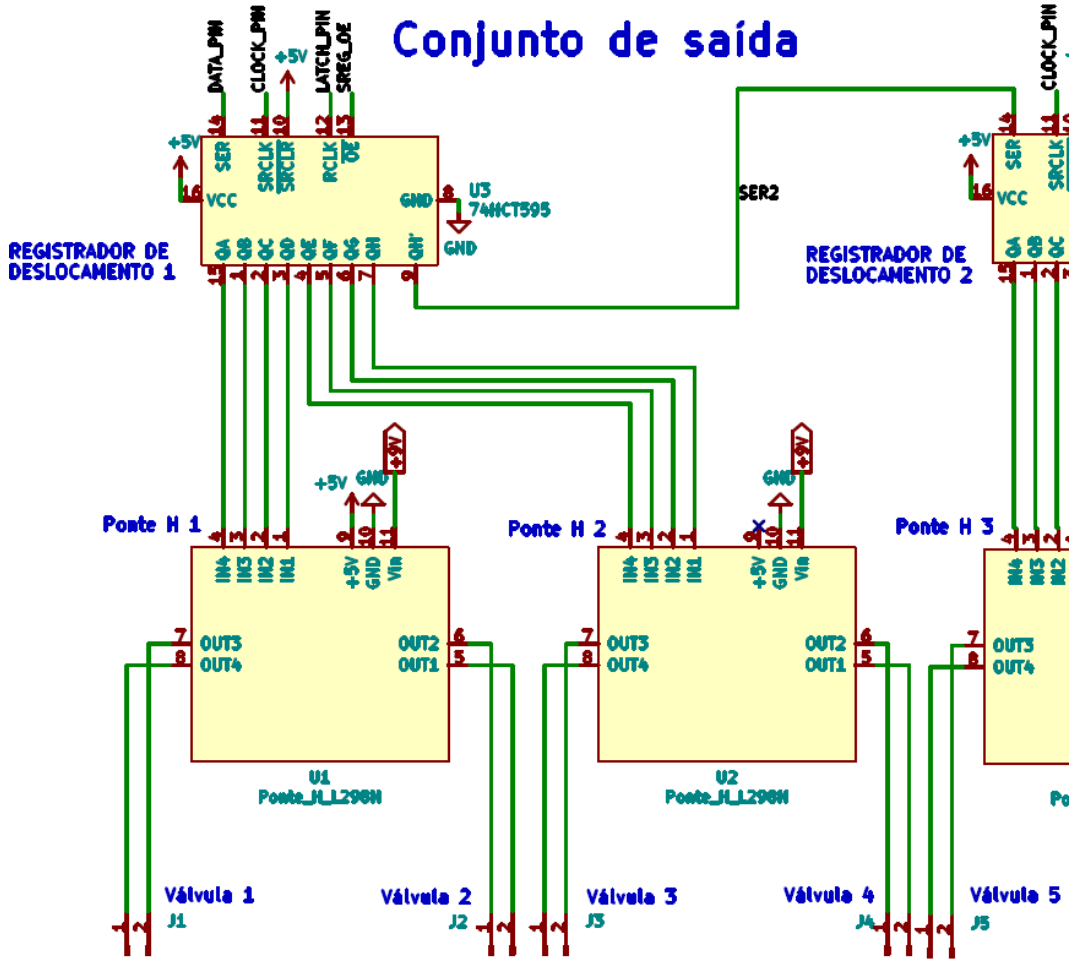


Figura 3.5: Conjunto de saída do primeiro registrador de deslocamento. (Fonte: o Autor)

O terceiro e quarto registradores de deslocamento apresentam, nas dez últimas saídas, os LEDs responsáveis pela indicação da atuação das válvulas, as conexões são mostradas na Figura 3.6.

Para evitar comandos espúrios nos SR, provocados pela flutuação de tensão durante a energização do circuito, utiliza-se a chave eletrônica com um TBJ (Transistor

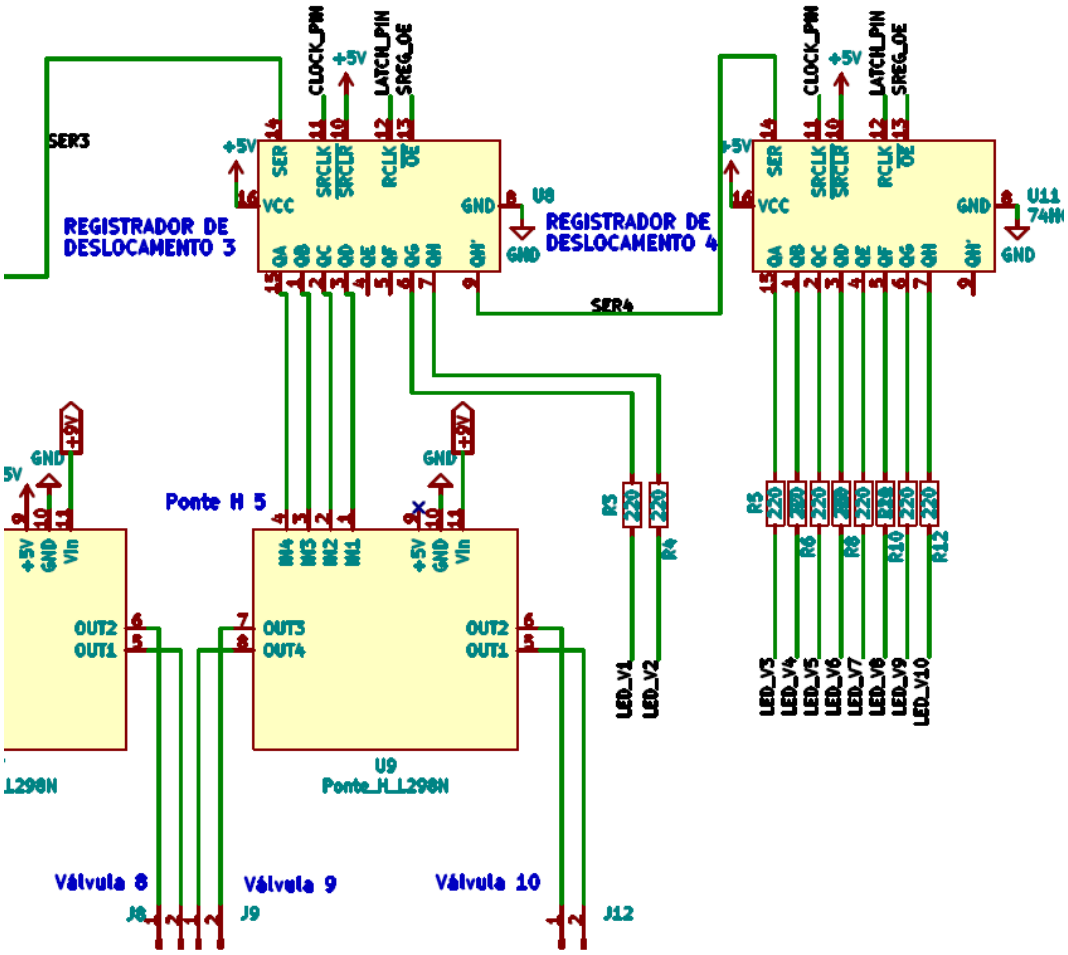


Figura 3.6: Conjunto de saída dos LEDs com o terceiro e quarto registrador de deslocamento.(Fonte: o Autor)

Bipolar de Junção) mostrada na Figura 3.7.

A chave funciona como um circuito de negação. Assim, quando o circuito é energizado, a saída do ESP32, ESP_OE, é imediatamente colocada para nível lógico baixo, levando o ponto SREG_OE para nível lógico alto, visto que o transistor está na região de corte. Garantindo, portanto, que as saídas dos registradores de deslocamento não são ativadas. A ativação apenas acontece quando o comando de ativação ocorrer, o que leva o nível lógico de ESP_OE para alto, polarizando o transistor para a região de saturação e consequentemente coloca SREG_OE para nível lógico baixo, habilitando as saídas dos registradores de deslocamento.

A ligação com os módulos periféricos acontece por meio dos conectores colocados no circuito para posicionar os módulos na placa final. A Figura 3.8 exibe os conecto-

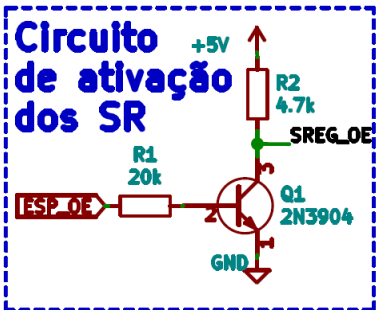


Figura 3.7: Circuito de ativação dos registradores de deslocamento.(Fonte: o Autor)

res dos periféricos com os rótulos que estão conectados ao microcontrolador, são eles: o conector do módulo RTC, que utiliza os pinos de SDA e SCL para comunicação no barramento I^2C; o conector de 20 pinos do circuito de interface, que tem as conexões com os sinais para os LEDs indicadores, para os botões e para o barramento I^2C para ativação do LCD (as conexões são convenientemente arranjadas de modo que a placa de interface possa ser confeccionada com o menor número de vias possível) e o conector para o módulo do cartão Micro SD, possibilitando o uso de um sistema de armazenamento mais extenso em projetos futuros.

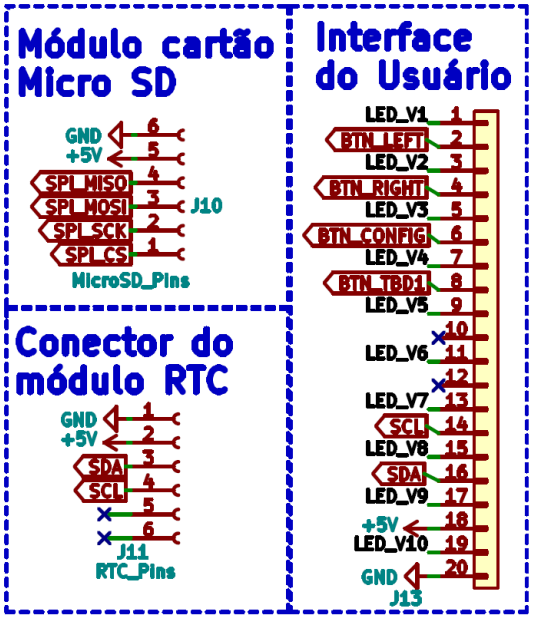


Figura 3.8: Conectores dos módulos periféricos.(Fonte: o Autor)

A placa de interface do usuário (confeccionada separadamente) recebe as co-

nexões com a placa de controle por meio de um conector de 20 pinos. O diagrama elétrico do circuito desenvolvido é apresentado na Figura 3.9. O circuito interliga-se com a placa principal por meio do conector, assim o conjunto I^2C e LCD realiza as ligações necessárias para conectar os módulos para controle do *display*, por meio do barramento I^2C. Além disso, os botões e os LEDs são conectados para respectivamente enviar e receber sinais da placa principal.

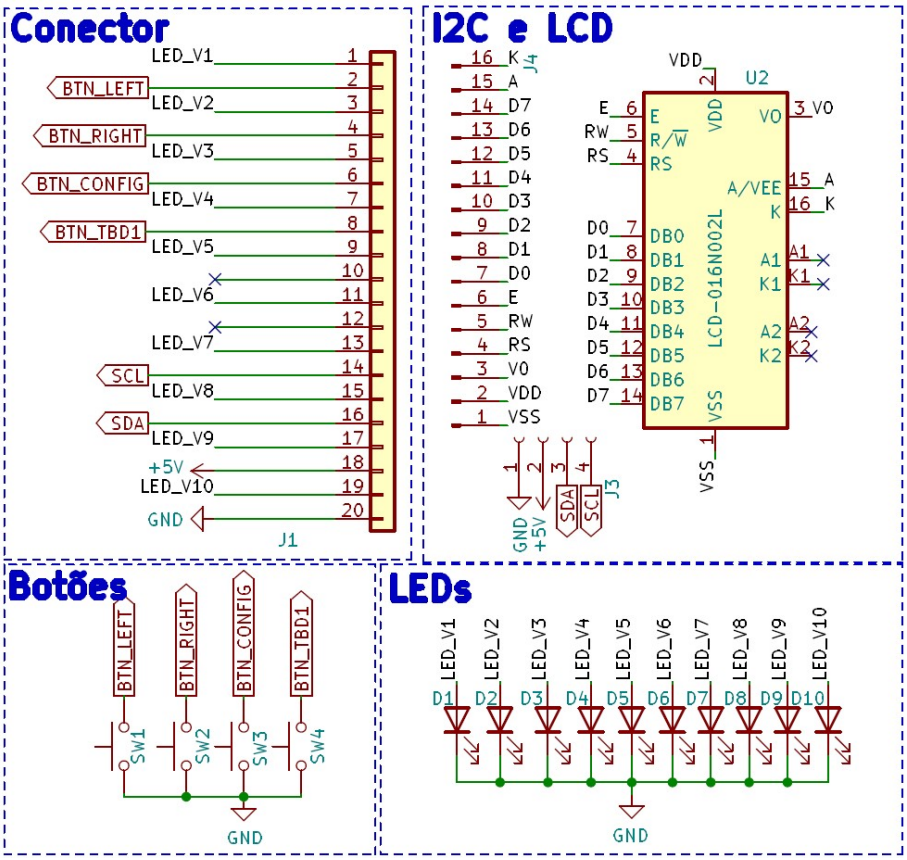


Figura 3.9: Circuito de Interface do Usuário.(Fonte: o Autor)

Por fim, encontra-se também na placa principal, o módulo responsável pela alimentação do sistema, que é aprensentado na Figura 3.10. O circuito conta com o conector J16, que recebe a alimentação geral do sistema, passa pelo regulador e é conectado a uma chave responsável por controlar a alimentação principal de 9 V do sistema, assim, o sistema é ligado apenas quando a chave SW1 for acionada.

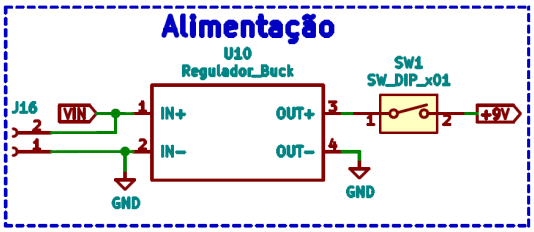


Figura 3.10: Circuito de alimentação do sistema.(Fonte: o Autor)

3.3 Software

Na construção do *software* várias rotinas são usadas para controle dos estados do sistema como um todo e da atuação dos periféricos. Nesta seção, as configurações iniciais são apresentadas para cada uma das rotinas, juntamente com os blocos de código ou máquinas de estados, que explicitam o funcionamento do sistema e as decisões tomadas. Como forma de modularizar o sistema, uma biblioteca que implementa a classe `ValvulaUtil` é criada, contendo diversos métodos que compõem o funcionamento do sistema, sobretudo na exibição das telas do LCD para o usuário e nas operações utilizando as válvulas. Algumas implementações importantes são discutidas nas próximas seções e o código completo está disponível no Apêndice A.

3.3.1 Rotina de exibição das telas do LCD

A biblioteca do Arduino, `LiquidCrystal_I2C.h`, é utilizada para a comunicação com o módulo LCD. A biblioteca apresenta uma interface de fácil inicialização, criando um objeto do tipo `LiquidCrystal_I2C`, apresentado no Código 3.1, que recebe o endereço do módulo no barramento I^2C, o valor hexadecimal $0x27$, e a quantidade de colunas e linhas do *display* utilizado, nesse caso 16 e 2 respectivamente. Com o objeto `lcd` instanciado é possível iniciá-lo com o método `.begin()`.

```
//Construtor
LiquidCrystal_I2C lcd(0x27, 16, 2);
//Inicialização
lcd.begin();
```

Código 3.1: Construtor do objeto `LiquidCrystal_I2C`. (Fonte: o Autor)

Os demais métodos para comunicação direta com LCD estão concentrados em um método auxiliar criado e denominado de `.imprimeLCD()`, da classe `ValvulaUtil`, que recebe a primeira e a segunda linha que se deseja imprimir, bem como a ação de limpar ou não o que estava escrito anteriormente. O método `.clear()` é responvável por limpar a tela, enquanto o método `.setCursor(coluna, linha)` é responsável por mover o cursor para a posição determinada pelos argumentos `coluna` e `linha`. Por último, o método `.print()` faz a impressão da *string* no *display*. O fluxo da função é apresentado no Código 3.2.

```
void ValvulaUtil::imprimeLCD(const char* primeiraLinha,
    const char* segundaLinha,
    bool limpa = true)
{
     if (limpa) {
      lcd.clear();
     }
     else
     {
      lcd.setCursor(0, 0);
     }
     lcd.print(primeiraLinha);
     lcd.setCursor(0, 1);
     lcd.print(segundaLinha);
}
```

Código 3.2: Método auxiliar `.imprimeLCD()` da classe `ValvulaUtil`. (Fonte: o Autor)

Com a execução do código, todas as linhas do LCD podem ser ocupadas pelos valores determinados nos argumentos de entrada, dessa forma, as diferentes telas da interface do usuário são exibidas no *display* utilizando o método `.imprimeLCD()`. Um método para cada uma das telas de interface está disponível, permitindo que a tela possa ser facilmente utilizada no código e exibida para o usuário.

3.3.2 Utilização do relógio de tempo real

Para a programação com o módulo RTC utiliza-se a biblioteca do Arduino, `Rtc_by_Makuna`, que apresenta uma série de métodos disponíveis para a utilização do dispositivo. Internamente a biblioteca seleciona o endereço padrão do módulo no barramento I^2C para a comunicação como o valor hexadecimal $0x68$, não chocando, portanto, com o valor definido para o LCD. O Código 3.3 descreve o passo a passo para fazer a configuração inicial do módulo.

```
//Para obter o tempo de compilação
RtcDateTime compiled = RtcDateTime(__DATE__, __TIME__);
compiled+=15;
RtcDateTime initialConfig = Rtc.GetDateTime();
if (initialConfig < compiled)
{
    Rtc.SetDateTime(compiled);
}
```

Código 3.3: Configuração inicial do relógio de tempo real. (Fonte: o Autor)

Primeiramente, é necessário sincronizar o módulo com o tempo real, para isso o tempo de compilação é obtido utilizando as definições `__DATE__` e `__TIME__` que recuperam a data e hora exatas em que a compilação ocorreu. A biblioteca aceita esses parâmetros no construtor do objeto `RtcDateTime` e permite que o valor criado seja somado com um inteiro, por meio de uma sobrecarga de operador. Assim o tempo é somado a um valor de compensação equivalente ao tempo de gravação, em torno de 15 segundos. O método `.GetDateTime()` comunica-se com o módulo e recupera o valor configurado, caso o relógio esteja atrasado com relação ao novo valor compilado, então o relógio é atualizado para o valor correto com o método `.SetDateTime()`.

Enquanto a bateria estiver carregada, o módulo mantém a hora e a data atualizadas, assim, durante os demais fluxos de código é possível utilizar apenas o método `.GetDateTime()` para recuperar a referência de tempo.

3.3.3 Ativação dos registradores de deslocamento

Para atuação no conjunto de saída, apenas os registradores de deslocamento precisam ser controlados pelo microcontrolador. A função `shiftOut()`, nativa do Arduino, auxilia na comunicação com o CI *74HC595*, apresentado na Seção 2.2.1. A assinatura da função é exibida no Código 3.4.

```
void shiftOut(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder,
          uint8_t val)
```

Código 3.4: Assinatura da função `shiftOut()`. (Fonte: o Autor)

O argumento `dataPin` corresponde ao pino de dados do microcontrolador, que deve ser conectado ao pino *SI* do registrador de deslocamento, o argumento `clockPin` é determinado pelo pino de *clock*, que deve ser conectado ao pino *SCK* do 74HC595. Já o parâmetro `bitOrder` representa a ordem que o valor deve ser enviado para a saída e recebe os valores `MSBFIRST`, quando o *bit* mais significativo deve ser o primeiro a ser enviado, ou `LSBFIRST`, quando o *bit* menos significativo deve ser enviado primeiro.

Por último, o argumento `val` corresponde ao valor de *8-bits*, que deve ser enviado para a saída. Para liberar a saída, quando a configuração é finalizada, o pino de controle `latchPin` deve ser conectado ao pino *RCK* do SR e habilitado com um pulso positivo por fora da função `shiftOut()`.

Entretanto, com a concatenação dos quatro registradores de deslocamento, um valor de 32-*bits* deve ser enviado para a saída. Por isso, uma função denominada de `sendToShiftRegister()` é criada para controlar a escrita dos *bits* e é exibida no Código 3.5.

```
void sendToShiftRegister(uint32_t output){
    uint8_t outputByte;
    digitalWrite(latchPin, LOW);
    digitalWrite(clockPin, LOW);
    //Carrega o dado
    for (int i = 0; i < REGISTER_NUMBER; i++){
```

```
        outputByte = (uint8_t)(output>>(i*8));
        shiftOut(dataPin, clockPin, LSBFIRST, outputByte);
    }
    //Libera pra o registrador de saida
    digitalWrite(latchPin, HIGH);
    delayMicroseconds(500); //Pausa por 0,5 ms para enviar o dado
    digitalWrite(latchPin, LOW);
}
```

Código 3.5: Função de controle dos SR conectados em cascata. (Fonte: o Autor)

O parâmetro `output` corresponde a saída de 32-*bits* a ser configurada. Inicialmente as variáveis de controle do SR, `latchPin` e `clockPin`, são colocadas em nível lógico baixo. Então, a saída de 32-*bits* é fracionada em blocos de 8-*bits* e enviada sequencialmente por meio da função `shiftOut()`. Por fim, a saída é habilitada com um nível lógico alto, escrito no pino `latchPin` por um intervalo de 500 µs.

3.3.4 Tipo de representação da válvula

A configuração do acionamento de cada válvula pode ser representada a partir de três variáveis: tempo total de aplicação da lâmina de irrigação (`tempoIrrigacao`), número de pulsos da irrigação (`numPulsos`) e intervalo entre os pulsos (`interPulsos`). Com isso é possível armazenar a configuração de cada uma das válvulas e obter os eventos de irrigação para a aplicação da lâmina de irrigação individualmente. O Código 3.6 mostra a definição da estrutura (tipo *struct* da linguagem C) `valvula_t`. Um operador de atribuição (`operator=`) é definido para facilitar a transferência de dados entre as variáveis do tipo `valvula_t`.

```
typedef struct{
        uint16_t tempoIrrigacao;
        uint8_t numPulsos;
        uint8_t interPulsos;
        void operator= (valvula_t v);
} valvula_t;
```

Código 3.6: *Struct* `valvula_t` para configuração da válvula. (Fonte: o Autor)

A configuração de cada válvula é armazenada em um vetor de `valvula_t`, alocado estaticamente com o número máximo de válvulas configuradas, ou seja, dez para este projeto. Para garantir que a configuração permaneça, mesmo que o microcontrolador desligue, implementa-se o salvamento desses dados na memória não volátil (NVS, sigla do inglês para *Non-Volatile Storage*) do microcontrolador ESP32. Para isso, utiliza-se a biblioteca `Preferences`, que salva os dados em seções chamadas de *"namespace"* dentro da NVS, em que cada seção guarda um conjunto de pares chave-valor.

O método `.begin()`, da classe `Preferences`, recebe uma *string* com o nome que se deseja dar à seção. Uma vez iniciada, é possível recuperar ou salvar dados, para isso são utilizados os métodos `.getBytes()` e `.putBytes()`, respectivamente, passando como um dos argumentos o tamanho do tipo `valvula_t`. A Tabela 3.2 mostra a estrutura da seção de memória e como os dados são armazenados.

namespace: valvulas			
Chave	**Valor** (*struct* **valvula_t**)		
	2 *bytes*: Tempo de Irrigação	1 *byte*: Número de Pulsos	1 *byte*: Intervalo entre Pulsos
valvula_1	300	5	10
valvula_2	300	1	0
..	...	...	...
valvula_10	200	4	50

Tabela 3.2: Organização da seção de memória da NVS com os dados armazenados. (Fonte: o Autor)

Dois métodos para a classe `ValvulaUtil` são criados para encapsular os métodos da biblioteca `Preferences` para esta aplicação, o método `.salvaValvulaFlash()` e o método `.recuperaValvulaFlash()`. Os métodos recebem como parâmetros o índice da válvula (`indexValv`) e a cópia de uma váriável do tipo `valvula_t`, para o caso do salvamento, ou uma referência do tipo `valvula_t` que permita modificação na função, para o caso da recuperação. Ambas as funções retornam um `bool` com valor `true` em caso de sucesso ou `false` em caso de erro. As declarações das funções

são apresentadas no Código 3.7.

```
bool ValvulaUtil::salvaValvulaFlash(uint8_t indexValv, valvula_t valv);

bool ValvulaUtil::recuperaValvulaFlash(uint8_t indexValv, valvula_t& valv);
```

Código 3.7: Métodos para salvar e recuperar a configuração das válvulas na memória não volátil. (Fonte: o Autor)

3.3.5 Modelagem dos eventos de irrigação

Para modelar os eventos de irrigação a *struct* `evento_irrigação_t` é criada. A declaração da *struct* é exibida no Código 3.8.

```
typedef struct{
        uint8_t indexValvula;
        bool acao;
        uint32_t horaProgramada;
        bool foiAtendido;
        void operator= (evento_irrigacao_t e);
} evento_irrigacao_t;
```

Código 3.8: *Struct* `evento_irrigação_t` para armazenar os eventos de irrigação. (Fonte: o Autor)

A estrutura apresenta o campo `indexValvula`, para identificação da válvula que se deseja atuar. O campo `acao` informa se o evento é de desligamento, caso tenha o valor `false`, ou de acionamento, caso tenha o valor `true`. O atributo `horaProgramada` define o momento de atuação do evento, em segundos, desde uma data de referência, nesse caso, 01/01/2000. Por último, o campo `foiAtendido` explicita se o evento já ocorreu, caso o valor seja `true`, ou se ainda não ocorreu, caso o valor seja `false`. A *struct* também apresenta a sobrecarga do operador de atribuição (`operator=`), para facilitar a transferência dos atributos entre os eventos.

3.3.6 Modelagem da geração dos eventos de irrigação a partir da configuração da válvula

Com as válvulas configuradas, é possível gerar todos os eventos de irrigação, agendando os acionamentos ou desligamentos das válvulas, para o tempo determinado. Cada válvula, necessariamente, deve ter a quantidade de eventos igual ao dobro do número de pulsos (NP), pois cada pulso precisa de um evento para ligar e outro para desligar. A Equação 3.1 mostra a relação

$$Número\ Eventos \equiv NE = 2 \times NP. \tag{3.1}$$

Por outro lado, o tempo de irrigação para cada pulso pode ser obtido por meio da razão entre o tempo total de irrigação e o número de pulsos que se deseja atuar, conforme a Equação 3.2:

$$Tempo\ do\ pulso \equiv TP = \frac{TI}{NP}. \tag{3.2}$$

Por exemplo, considere um horário de início às 14:00 e uma configuração de válvula qualquer, com tempo total de irrigação de 300 minutos, número de pulsos igual a 5 e intervalo entre os pulsos de 10 minutos. Utilizando-se as Equações 3.1 e 3.2, chega-se a um total de 10 eventos e 60 minutos de tempo do pulso. A visão geral dos eventos do exemplo é apresentada na Figura 3.11.

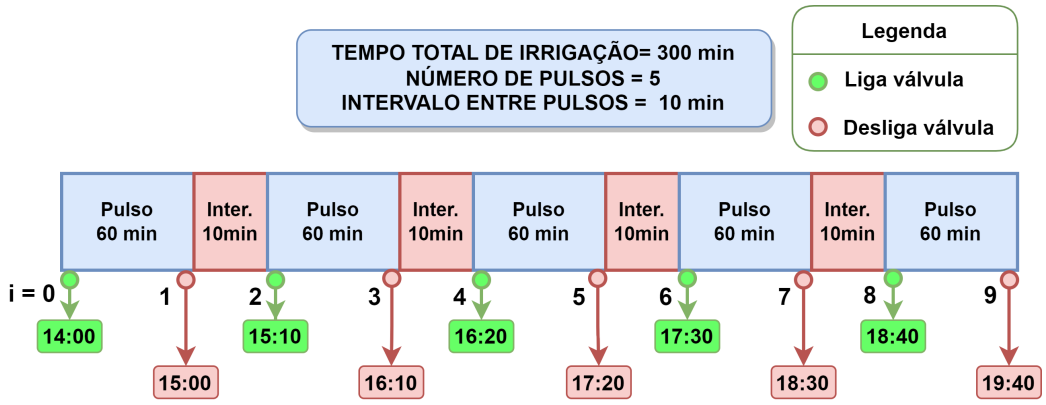


Figura 3.11: Exemplo de eventos de irrigação para uma configuração específica. (Fonte: o Autor)

Observa-se, portanto, uma sequência dos eventos de ativação e desligamento dos solenoides. Tomando o primeiro evento como o evento de número 0, é possível extrair que os eventos pares (0, 2, 4, ...) são de ativação das válvulas, enquanto os eventos ímpares (1, 3, 5, ...) são de desativação. Considerando arbitrariamente o evento de ativação $i = 6$, verifica-se que ocorrem três pulsos e três intervalos, enquanto no evento de desativação com $i = 7$, ocorreram quatro pulsos e três intervalos. Dessa forma, a partir do padrão dos eventos, é possivel chegar a Equação 3.3:

$$Horário\ do\ Evento\ i \equiv H_i = I + \left\lfloor \frac{i}{2} \right\rfloor \times (TP + IP) + (k) \times TP, \tag{3.3}$$

em que, $i \equiv k \bmod 2$ e I é o *tempo de início*.

O valor de k é definido como o resto da divisão de i por 2, distinguindo entre os eventos pares e ímpares, enquanto I é definido como o tempo de início da irrigação. Ao tomar todos os valores em segundos, e considerando I com relação à referência do RTC, é possível obter o horário do evento em segundos, H_i, com relação à referência, o que torna fácil a ordenação dos eventos.

No programa, a Equação 3.3 é utilizada para gerar os eventos de cada uma das válvulas a partir do método `.agendamentoAPartirDaValvula()`, cuja declaração é mostrada no Código 3.9.

```
void ValvulaUtil::agendamentoAPartirDaValvula(
    uint8_t index,
    valvula_t valvula,
    uint32_t tempoInicio,
    evento_irrigacao_t* agenda);
```

Código 3.9: Método para gerar os eventos de irrigação a partir dos parâmetros de configuração das válvulas. (Fonte: o Autor)

Para gerar todos os eventos de irrigação, o método recebe o índice da válvula por meio do parâmetro `index`, as configurações definidas para a válvula especificada por meio do parâmetro `valvula` do tipo `valvula_t`, o tempo de início do agendamento,

em segundos, como um inteiro de 32-*bits*, com o parâmetro `tempoInicio`. Por fim, o parâmetro `agenda`, que é um ponteiro para um vetor de `evento_irrigacao_t`, com todos os eventos que devem ser atendidos na irrigação, para os parâmetros definidos de cada válvula.

O Código 3.10 exibe como é definido o laço principal utilizado para preencher o vetor `agenda`, para cada `evento_irrigacao_t`, bem como a implementação da Equação 3.3 utilizando a linguagem C.

```
uint8_t numEventos = s_valvula.numPulsos*2;
uint32_t tempoDoPulso = (s_valvula.tempoIrrigacao*MINUTES_IN_SECONDS)/s_valvula.numPulsos;
for (int i=0; i<numEventos; i++) {
  agenda[i].indexValvula = index;
  agenda[i].foiAtendido = false;
  agenda[i].horaProgramada = tempoInicio;
  uint8_t resto = i%2;
  //Se for ímpar o evento deve ser de desativação (false)
  //Se for par o evento deve ser de ativação (true)
  s_agenda[i].acao = (resto == 1)? false: true;
  s_agenda[i].horaProgramada +=
  ((i/2)*(tempoDoPulso
      + s_valvula.interPulsos*MINUTES_IN_SECONDS) + resto*(tempoDoPulso);
}
```

Código 3.10: Implementação do laço principal do método `.agendamentoAPartirDaValvula()`. (Fonte: o Autor)

3.3.7 Rotina de aquisição dos dados de entrada

Para obter as informações de entrada dos botões, é utilizada a biblioteca `ezButton.h`, que apresenta diversas funcionalidades necessárias para a aplicação neste projeto. Inicialmente um objeto do tipo `ezButton` deve ser instanciado, passando a porta lógica utilizada como argumento para o construtor. Como configuração inicial é necessário definir o tempo de *debounce*[1], com o método `.setDebounceTime(TEMPO)`.

[1]É o tempo que o programa utiliza para distinguir um comando dado pelo usuário, de uma trepidação da chave mecânica.

O método `.loop()` deve ser utilizado em conjunto com a função `loop()` do Arduino, para fazer a atualização periódica do estado dos botões, assim, o método `.isPressed()` pode ser utilizado para verificar se o botão foi ou não pressionado desde a última verificação. O Código 3.11 apresenta a utilização resumida desses métodos.

```
//Inicialização do objeto
ezButton botao(PINO_BOTAO);
//Configuração inicial
void setup() {
 //...
 botao.setDebounceTime(TEMPO_DE_DEBOUNCE);
 //...
}
//Operação periódica
void loop() {
 botao.loop();
 if(botao.isPressed()){
  //Ação a ser realizada
 }
 //...
}
```

Código 3.11: Métodos utilizados para controle do botão. (Fonte: o Autor)

Embora quatro botões tenham sido colocados na placa final, apenas três botões são utilizados no projeto. O quarto botão é colocado para utilização futura. O primeiro botão é nomeado como `btnLeft`, fazendo referência a botão esquerdo, o segundo como `btnRight`, referindo-se ao botão direito, e o terceiro como `btnConfig`, que é o botão de configuração.

Sempre que os botões são pressionados, uma *flag* global é levantada no sistema para que seja feito o devido tratamento em outra parte do programa, normalmente controlando a mudança de estado ou acionando alguma funcionalidade. Assim, a função dos botões varia de acordo com o estado em que o sistema encontra-se. Porém, de maneira geral, o `btnLeft` realiza a ação de transicionar o fluxo das telas

para a esquerda, o `btnRight` altera o fluxo das telas para a direita e, por fim, o `btnConfig` exibe configurações e confirma ações do usuário.

3.3.8 Estados do sistema

Os estados do sistema, por vezes, tornam-se complicados, uma vez que cada etapa precisa de uma série de configurações para garantir o pleno funcionamento do sistema. Por isso, dois tratamentos são criados para cada estado, o primeiro consiste no tratamento das condições de transição entre um estado e outro, e o segundo consiste das ações tomadas.

O tratamento das transições leva em consideração os botões pressionados ou a contagem de tempo em determinado estado, verificando periodicamente o estado do sistema na função `transicoesDeEstado()`, disposta na função `loop()` do Arduino.

As ações dos estados são tomadas também de forma periódica, porém com frequência mais baixa, a partir de um temporizador de 500 ms configurado externamente, assim, somente quando o temporizador incrementa uma variável de controle durante a interrupção é que o sistema realiza as ações do estado, a partir da função `acoesDosEstados()`. Na Figura 3.12 é possível ver o diagrama de estados do sistema de forma simplificada, em que cada estado pode agrupar outros estados intermediários internamente.

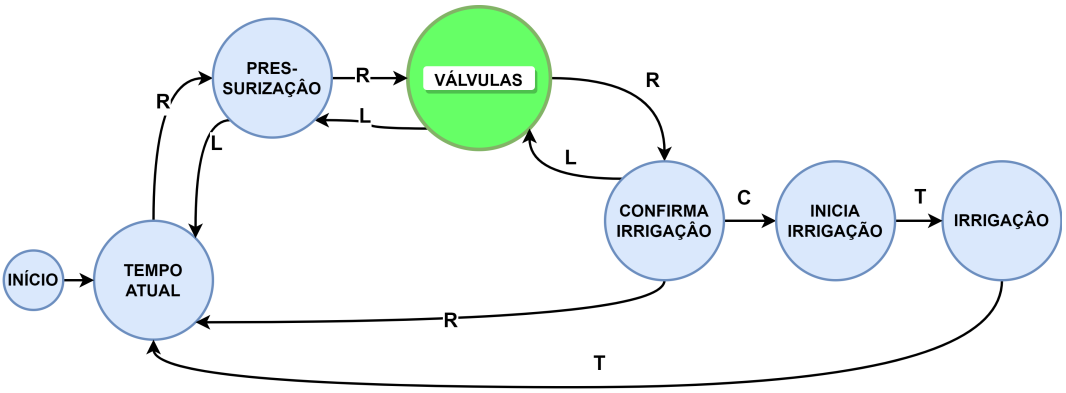


Figura 3.12: Máquina de estados simplificada do sistema. (Fonte: o Autor)

De modo geral, o comportamento dos estados varia com a atuação de algum dos botões, exibidos no diagrama como botão esquerdo (**L**), botão direito (**R**) e botão

de configuração (**C**) ou através do tempo (**T**). Os estados apresentados em azul são estados completos, enquanto os estados em verde contêm estados intermediários.

O primeiro estado exibido ao ligar o sistema é o `TEMPO ATUAL`. Ao apertar **R**, o operador é levado ao estado de `PRESSURIZAÇÃO`, em que o usuário pode iniciar ou pausar a pressurização do sistema. Em seguida, prosseguindo para a direita, o estado das `VÁLVULAS` é alcançado, nesse superestado, as válvulas são exibidas e podem ser configuradas individualmente.

Ao continuar apertando **R**, chega-se ao estado `CONFIRMA IRRIGAÇÃO`, em que é possível iniciar a `IRRIGAÇÃO` propriamente, apertando o botão **C**, ou retornar ao estado `TEMPO ATUAL` apertando **R** novamente. Ainda é possível navegar no sentido contrário entre os estados apertando o botão **L**.

3.3.9 Estado TEMPO ATUAL

Neste estado, o sistema recupera o valor do RTC e utiliza o método da classe `ValvulaUtil`, chamado de `.telaDataHora()`, para exibir a data e a hora no LCD, informando o operador da referência de tempo do sistema. O método recebe como parâmetros os dados de dia, mês, ano, hora, minuto e segundo, para construir a tela do LCD, mostrada na Figura 3.13.

Figura 3.13: Tela de data e hora do LCD. (Fonte: o Autor)

3.3.10 Estado PRESSURIZAÇÃO

No estado de pressurização, o sistema exibe a mensagem "*Iniciar Pressurização?*" no *display*, conforme exibido na Figura 3.14. Ao pressionar o botão **C**, o sistema inicia a pressurização, enviando o comando de acionamento para todas as válvulas e ativando um temporizador que conta o tempo de pressurização na tela conforme

exibe a Figura 3.15. Caso o sistema já tenha sido iniciado, um novo comando no botão **C** desativa todas as válvulas e para o temporizador. Uma vez iniciado o processo de pressurização, o operador pode continuar navegando entre as demais telas para concluir a configuração das válvulas, antes de iniciar a irrigação propriamente.

Figura 3.14: Tela para iniciar pressurização. (Fonte: o Autor)

Figura 3.15: Tela de pressurização. (Fonte: o Autor)

3.3.11 Estado VÁLVULAS

O estado `VÁLVULAS`, apresentado na Figura 3.16, é composto por diversos estados intermediários. O primeiro consiste da exibição dos parâmetros configurados nas válvulas (EXIBE VÁLVULAS). Os parâmetros são exibidos conforme a tela do LCD apresentada na Figura 3.17. A tela contém o índice da válvula, o parâmetro `TI` que corresponde ao tempo total de irrigação em minutos, o parâmetro `NP` que corresponde ao número de pulsos da irrigação, e por fim, o parâmetro `IP` que corresponde ao intervalo entre pulsos em minutos.

Ao apertar os botões **R** e **L** é possível aumentar ou diminuir o índice da válvula que está sendo exibida. Caso seja pressionado o botão **L** na válvula 1, o sistema volta para o estado da `PRESSURIZAÇÃO`. Por outro lado, caso o botão **R** seja pressionado enquanto estiver exibindo a válvula 10, o sistema avança para o estado `CONFIRMA IRRIGAÇÃO`. Para realizar a configuração individual das válvulas, o botão **C** deve ser pressionado enquanto a tela de exibição da válvula de interesse estiver aparecendo, iniciando o subestado `CONFIG. VALV. X`. O valor de X é determinado

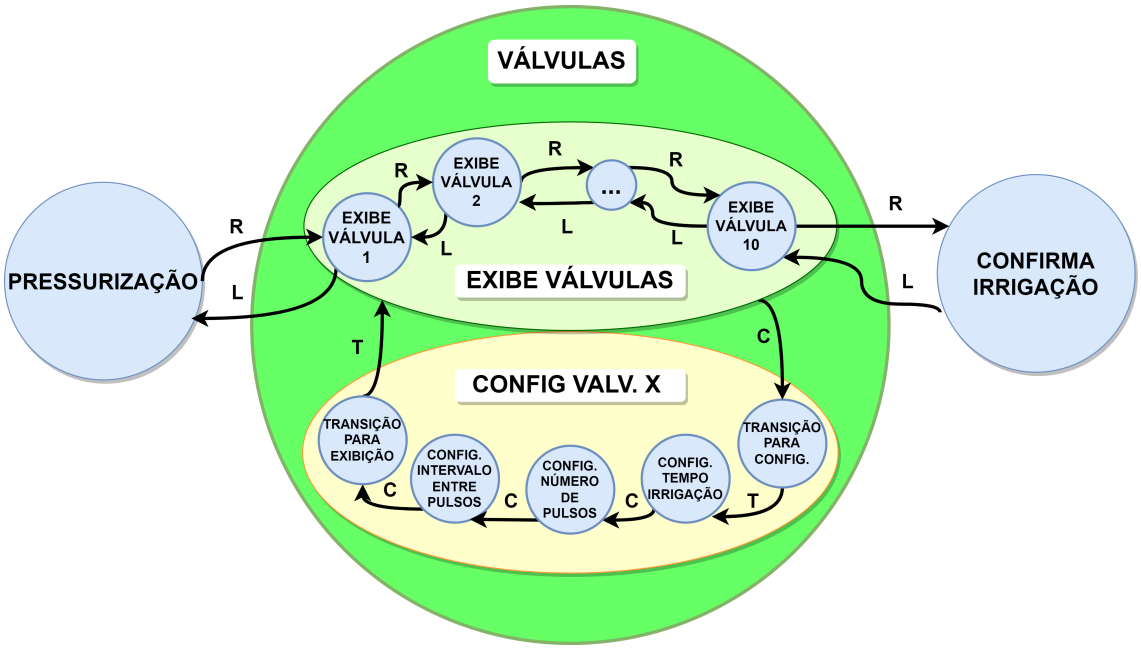


Figura 3.16: Máquina de estados VÁLVULAS com estados intermediários internos. (Fonte: o Autor)

Figura 3.17: Tela exibição das válvulas. (Fonte: o Autor)

pelo índice da válvula que estava sendo exibida no momento da configuração.

Ao entrar no modo de configuração, o sistema mostra uma tela de transição, conforme a Figura 3.18, informando qual válvula está sendo configurada por dois segundos, representado no diagrama de estados por **T**.

Figura 3.18: Tela exibida no estado de transição para configuração, TRANSIÇÃO PARA CONFIG. (Fonte: o Autor)

Em seguida a tela de configuração do tempo de irrigação da Figura 3.19(a) é apresentada. Na configuração dos parâmetros, os botões **L** e **R** passam a funcionar, respectivamente, diminuindo e aumentando os valores. Uma vez selecionado

o valor desejado, o botão **C** deve ser pressionado para definir a configuração e o próximo parâmetro é exibido para a configuração. As telas de configuração dos demais parâmetros são mostradas nas Figuras 3.19(b), 3.19(c).

(a) Tela exibida no estado de configuração do tempo de irrigação, CONFIG. TEMPO IRRIGAÇÃO.

(b) Tela exibida no estado de configuração do número de pulsos, CONFIG. NÚMERO DE PULSOS.

(c) Tela exibida no estado de configuração do intervalo entre pulsos, CONFIG. INTERVALO ENTRE PULSOS.

Figura 3.19: Telas exibidas no estado `CONFIG. VALV. X`. (Fonte: o Autor)

Finalizada a configuração de todos os parâmetros, o sistema faz o salvamento dos novos valores na memória NVS e mostra a tela de transição para exibição da Figura 3.20 por dois segundos, em seguida a tela de exibição das válvulas, mostrada na Figura 3.17, é apresentada novamente ao usuário com os valores dos parâmetros atualizados.

Figura 3.20: Tela mostrada no estado de transição para exibição, TRANSIÇÃO PARA EXIBIÇÃO. (Fonte: o Autor)

3.3.12 Estado CONFIRMA IRRIGAÇÃO

O estado CONFIRMA IRRIGAÇÃO exibe para o usuário a tela apresentada na Figura 3.21. O operador pode voltar a navegação para revisar alguma configuração

com os botões **L** e **R**, ou pode iniciar a irrigação com as configurações definidas apertando o botão **C**.

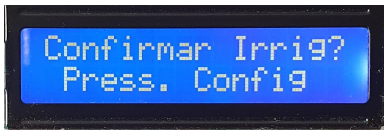


Figura 3.21: Tela do estado CONFIRMA IRRIGAÇÃO. (Fonte: o Autor)

Uma vez iniciada a irrigação, o sistema entra em um estado intermediário, chamado de INICIA IRRIGACAO, em que os eventos de irrigação são gerados. O método descrito na Seção 3.3.6, `.agendamentoAPartirDaValvula()`, é usado para gerar os eventos de todas as válvulas. Para isso, um vetor estático do tipo `eventos_irrigacao_t`, chamado de `evento_buffer`, é criado para ser passado como parâmetro para o método.

Cada válvula é então enviada como parâmetro com um laço `for` e o vetor é preenchido sequencialmente. O Código 3.12 exibe o preenchimento do vetor de eventos a partir da iteração dos índices das válvulas e da posição em que os dados são colocados na variável `evento_buffer`. Além disso o tempo de início é determinado pelo valor atual da hora, `now`, somado a um tempo de compensação antes da ativação dos eventos.

```
int posicaoDaAgenda=0;
for(int i = 0; i < NUM_VALVULAS; i++){
    if(i == 0){
        posicaoDaAgenda += 0;
    } else{
        posicaoDaAgenda += (valvulas[i-1].numPulsos)*2;
    }
    vUtil.agendamentoAPartirDaValvula(i+1, valvulas[i],
        now + IRRIG_OFFSET, //Início da irrigação
        evento_buffer + posicaoDaAgenda);
}
```

Código 3.12: Preenchimento do vetor de eventos. (Fonte: o Autor)

Assim, os eventos de irrigação do vetor ficam ordenados de forma crescente, segundo os índices das válvulas. Entretanto, devido a ordem de ativação dos eventos levar o tempo em consideração, torna-se necessário fazer o ordenamento dos eventos com relação a hora programada.

Para realizar a ordenação com relação ao tempo, a função `sort()` é utilizada a partir de uma biblioteca chamada `Arduino_Helpers.h`, que faz algumas implementações padrão do C++ para o ambiente do Arduino. A função recebe como parâmetros de entrada as posições do primeiro e do último elemento do vetor e uma função de comparação personalizada. O usuário da biblioteca deve implementar a função de comparação para a sua aplicação, desde que retorne valores do tipo `bool` e garanta o retorno `true`, somente se o primeiro argumento for menor do que o segundo. O Código 3.13 exibe a implementação da função de comparação, `compareFunc()` e a utilização da função `sort()`, com o vetor de armazenamento dos eventos, o `evento_buffer`.

```
/*Funcao para realizar a comparação entre dois eventos no sort*/
bool compareFunc(evento_irrigacao_t a, evento_irrigacao_t b){
    if(a.horaProgramada == b.horaProgramada) {
        return a.indexValvula < b.indexValvula;
    }
    return a.horaProgramada < b.horaProgramada;
}
std::sort(evento_buffer, evento_buffer + totalEventos, compareFunc);
```

Código 3.13: Utilização da função `sort()` para ordemento dos eventos de atuação da irrigação. (Fonte: o Autor)

Durante a execução dessas tarefas, o texto mostrado na Figura 3.22 é exibido no LCD. Tendo em vista a finalização da geração de todos os eventos de irrigação, torna-se possível executar o estado de IRRIGAÇÃO propriamente.

3.3.13 Estado IRRIGAÇÃO

Com todos os eventos ordenados, o estado IRRIGAÇÃO fica responsável por gerenciar o acionamento das válvulas solenoides no tempo correto, comparando o

Figura 3.22: Tela do estado INICIA IRRIGAÇÃO. (Fonte: o Autor)

horário atual com o horário programado no próximo evento e executando-o, caso seja maior. A variável `contadorEventos` é utilizada para identificar o evento de ativação atual no programa, assim a variável realiza a excursão nos índices do vetor `evento_buffer`, da posição 0 até a posição `totalEventos-1`. O horário da execução do programa é obtido por meio do método `.GetDateTime()`. A implementação é exibida no Código 3.14.

```
if(contadorEventos < totalEventos){
    vUtil.telaIrrigacao(contadorEventos, totalEventos);
    RtcDateTime actual = Rtc.GetDateTime();
    if(actual > evento_buffer[contadorEventos].horaProgramada){
        if(trataEventoIrrigacao(&evento_buffer[contadorEventos],
            &flagAtuandoValvula)){
            contadorEventos++;
        }
    }
}
```

Código 3.14: Implementação da execução dos eventos de irrigação. (Fonte: o Autor)

Para garantir que apenas uma das válvulas é atuada por vez, a função `trataEventoIrrigacao()` é executada somente se o sistema estiver disponível para atuar, condição representada pela variável `flagAtuandoValvula` com valor `false`. Se o sistema atuar a válvula com sucesso, a variável `contadorEventos` é incrementada para atuar no próximo evento.

A implementação da função `trataEventoIrrigacao()` é exibida no Código 3.15 e é responsável pela ativação ou desativação dos solenoides e pela atualização do estado dos LEDs. Para isso, um inteiro de 32-*bits* é utilizado, em que cada *bit*

representa uma saída dos registradores de deslocamento em cascata, conforme a Figura 3.3 discutida anteriormente.

```
bool trataEventoIrrigacao(evento_irrigacao_t* e, bool flagAtuando){
  if(!flagAtuando){ //Não realiza ação se já estiver atuando
    uint32_t aux = e->acao? 0x80000000: 0x40000000;
    aux = (aux >> ((e->indexValvula - 1)*2));
    aux += outputIndicationMask(e->acao, e->indexValvula);
    sendToShiftRegister(aux);
    *flagAtuando = true;
    timerWrite(timerOutputCtrl, 0);
    timerAlarmEnable(timerOutputCtrl);
    return true;
  }
  return false;
}
```

Código 3.15: Implementação da função `trataEventoIrrigacao()`. (Fonte: o Autor)

Para ativação do solenoide, a sequência binária $0b10$ deve ser enviada para o registrador de deslocamento, enquanto para desativação, a sequência $0b01$ deve ser enviada. As máscaras hexadecimais de 32-*bits*, $0x80000000$ e $0x40000000$, são utilizadas para este fim lógica, uma vez que o *nibble* mais significativo apresenta a sequência de ativação e de desativação, respectivamente, com os valores binários $0b1000$ e $0b0100$. Portanto, realizar um deslocamento para a direita de 2 *bits* nesses números permite enviar o comando de controle para qualquer uma das 10 válvulas configuradas.

Além do comando da válvula, é necessário reforçar o comando dos LEDs. Para isso, a função `outputIndicationMask()` cria uma máscara que permite manter o estados dos LEDs dos 10 *bits* menos significativos do SR independentemente da atuação das válvulas. Desse modo, a máscara é somada à variável de ativação e enviada para o registrador de deslocamento por meio da função `sendToShiftRegister()`.

Uma vez enviado o comando, ainda é preciso garantir o tempo necessário do pulso para a comutação das válvulas. Para isso, um alarme é habilitado, utili-

zando o `timer` do microcontrolador. A `flagAtuandoValvula` é colocada com valor lógico `false` para bloquear a atuação de novas válvulas até a finalização da atuação. Passados aproximadamente 200 ms, que é o tempo configurado no `timer`, a função `clearOutputControl()` realiza a limpeza das saídas do registrador de deslocamento para os solenoides e desabilita o `timer`, até a próxima atuação. Quando finalizados todos os eventos, o sistema volta para o estado inicial de exibição do tempo e configuração das válvulas.

Capítulo 4

Resultados

4.1 Confecção da placa de circuito impresso

Após todos os testes de validação do funcionamento do sistema, uma placa de circuito impresso (PCB, do inglês *Printed Circuit Board*) é apresentada, utilizando o *software* de prototipação de circuito *KiCAD*. O diagrama de circuito elétrico, discutido no Capítulo 3, é obtido utilizando a ferramenta de prototipação para produzir as duas placas: a placa principal, responsável pelo controle e acionamento das válvulas, de tamanho aproximado de 15 cm $\times$ 22 cm; e a placa de interface do usuário, com dimensões de aproximadamente 9 cm $\times$ 15 cm.

As placas são confeccionadas utilizando as dependências do Departamento de Engenharia Elétrica (DEE) da Universidade Federal de Pernambuco, mais especificamente no GEPAE (Grupo de Eletrônica de Potência e Acionamentos Elétricos), que conta com uma CNC de 3 eixos capaz de usinar PCBs e outros materiais. A Figura 4.1 exibe a confecção da placa principal.

Com a finalização da confecção das placas, os componentes e módulos são soldados no circuito. Alguns módulos necessitam de pequenas adaptações, como o ajuste dos conectores para permitir a soldagem e fixação na placa. A Figura 4.2 exibe a placa principal e a placa de interface confeccionadas.

Para garantir a integridade das placas desenvolvidas, também é confeccionada uma caixa em acrílico para proteger e isolar o circuito. A caixa possui a tampa supe-

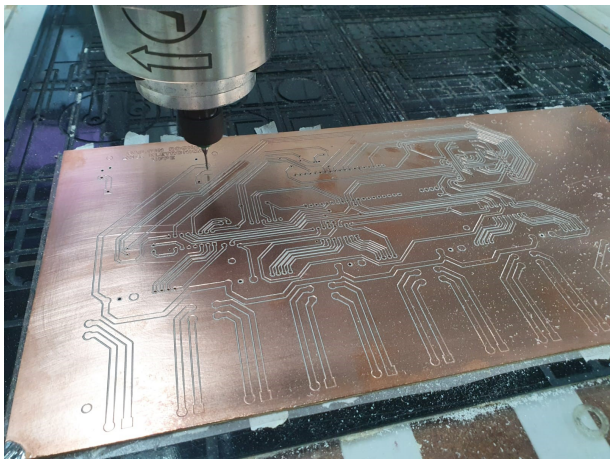


Figura 4.1: Utilização da CNC para usinagem da PCB. (Fonte: o Autor)

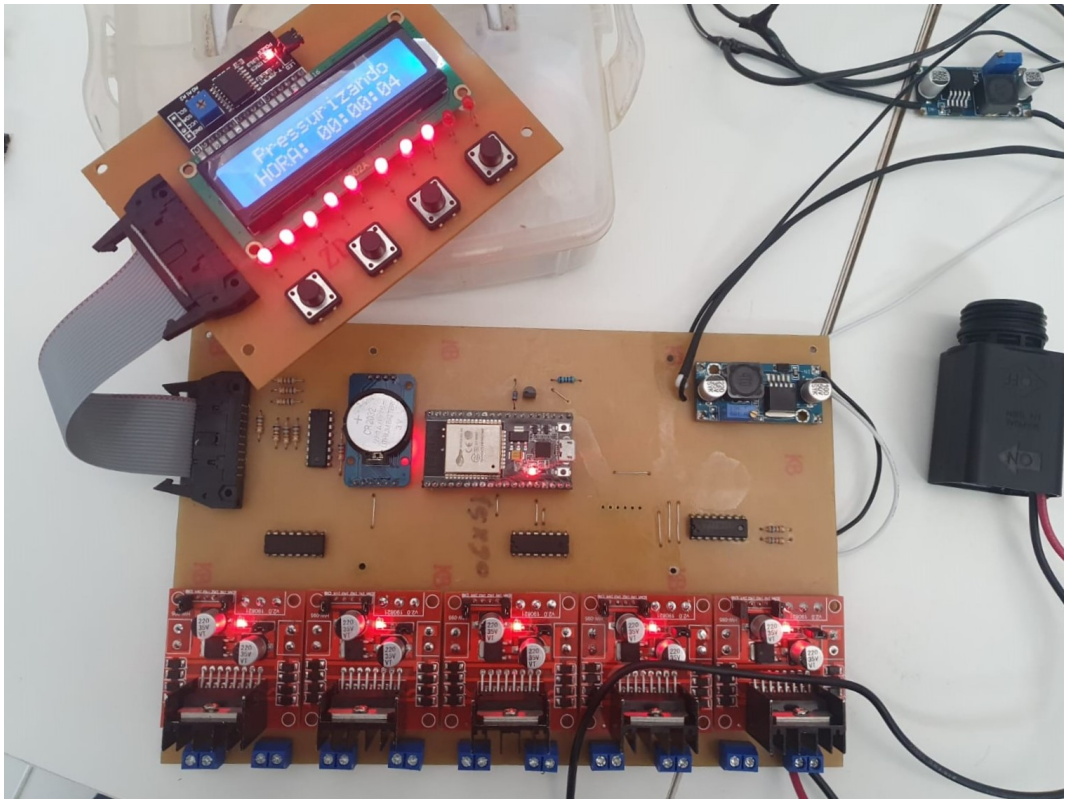


Figura 4.2: Placa principal e de interface do usuário confeccionadas. (Fonte: o Autor)

rior removível e fixada com parafusos colocados na parte inferior. Entretanto, para aumentar a durabilidade do *hardware* e permitir sua melhoria em projetos futuros uma interface USB, que permite se comunicar e programar o microcontrolador, está disponível na parte externa da caixa. Desse modo, é possível atualizar o *firmware*, sem comprometer a integridade do *hardware*. O invólucro desenvolvido para com-

portar o sistema é apresentado na Figura 4.3.

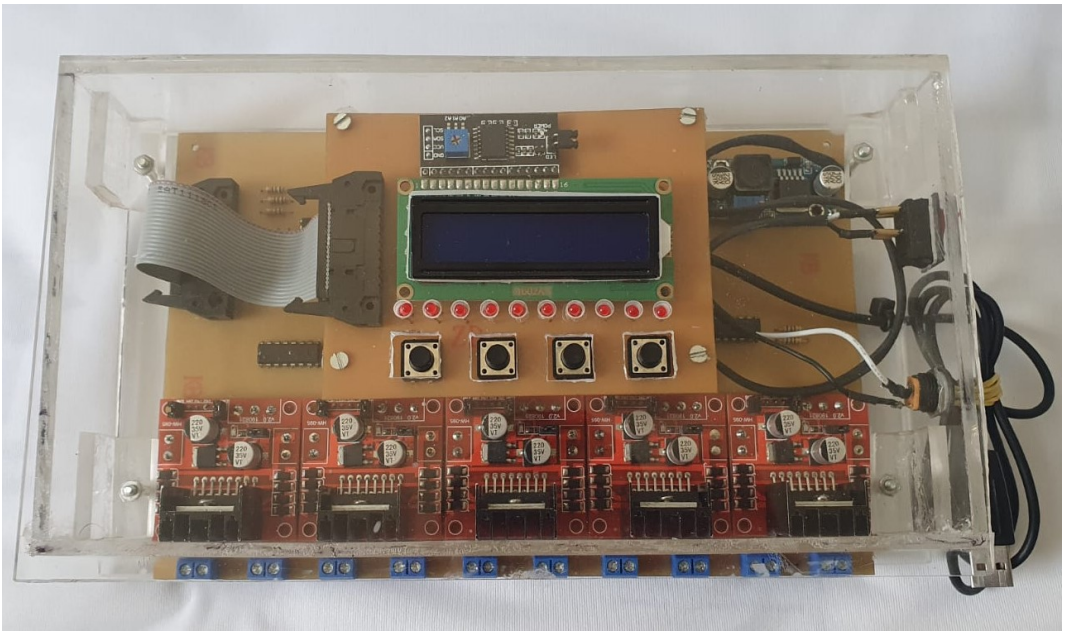


Figura 4.3: Caixa de acrílico desenvolvida para comportar o *harware*. (Fonte: o Autor)

4.2 Validação do sistema desenvolvido

A validação do funcionamento do sistema é realizada de maneira iterada, tanto na etapa de desenvolvimento do protótipo, com a matriz de contatos, quanto com a utilização da placa confeccionada. Inicialmente são realizados testes de acionamentos dos solenoides em um ambiente controlado, para verificação do comportamento do sistema e suas características elétricas, sobretudo a corrente de consumo. Com a comprovação da ativação dos solenoides no laboratório, são realizados testes com o arranjo hidráulico, ainda com o circuito na matriz de contatos. Por fim, após a confecção das placas e da caixa dde acrílico, o circuito é testado novamente utilizando a malha hidráulica para validação.

4.2.1 Testes de acionamento dos solenoides e corrente do sistema

O acionamento dos solenoides é testado ainda na fase de desenvolvimento, utilizando matrizes de contato, a fim de validar a corrente consumida pelo circuito durante a atuação no dispositivo. A montagem da Figura 4.4 é utilizada para testes de corrente.

Figura 4.4: Montagem para teste de consumo de corrente com o solenoide (Fonte: o Autor)

Um multímetro é colocado em série com a entrada de alimentação do circuito para medição da corrente de entrada, enquanto um osciloscópio é colocado com uma garra de corrente para avaliar a forma de onda da corrente. Durante a operação com os dispositivos ligados, obtém-se uma corrente média de aproximadamente 230 mA com o multímetro, variando de 173 mA até 343 mA segundo resultado obtido no osciloscópio.

Em outro teste, um multímetro digital Fluke, com captura de valores máximos e mínimos, é utilizado com o objetivo de obter a máxima corrente durante a ativação do solenoide. Para realização do teste, um solenoide é conectado ao primeiro canal de ativação por meio de uma ponte H, em seguida, um comando de pressurização é enviado acionando a válvula conectada, a fim de capturar a corrente de pico com o multímetro. A corrente obtida é de 1,44 A conforme exibido na Figura 4.5 e a comutação do solenoide é constatada a partir do som característico produzido pelo

dispositivo.

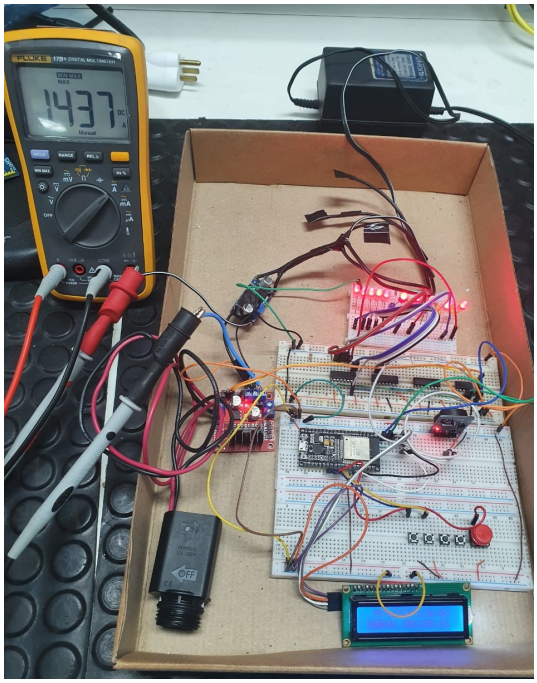


Figura 4.5: Montagem para teste de corrente máxima com solenoide conectado. (Fonte: o Autor)

Na mesma montagem da Figura 4.5, dois canais do osciloscópio Yokogawa são utilizados para capturar as formas de onda das tensões de acionamento, tanto do solenoide quanto do sinal enviado para o SR. A Figura 4.6 exibe a tela do osciloscópio, em que a tensão do SR durante a ativação pode ser visualizada pelo cursor 2, com valor de 4,4 V. Em fase com a subida da tensão do SR, pode-se observar a subida da tensão do solenoide, marcada pelo cursor 1 em torno de 6,4 V.

A diferença entre a tensão observada no solenoide e a tensão de entrada de 9 V está na queda de tensão presente na ponte H, que segundo a folha de dados, pode ter valor máximo de 4,9 V para uma corrente em torno de 1,8 A, assim, a queda de tensão observada foi de 2,6 V. Entretanto, a discrepância não provocou falha no funcionamento do dispositivo e permitiu que a corrente de ativação fosse menor, conforme observado na medição.

A duração do pulso pode ser vista na Figura 4.7(a), em que são apresentados os pulsos do primeiro solenoide e o sinal de ativação do SR para o segundo solenoide. Conforme apresentado, cada possui duração de aproximadamente 200 ms e,

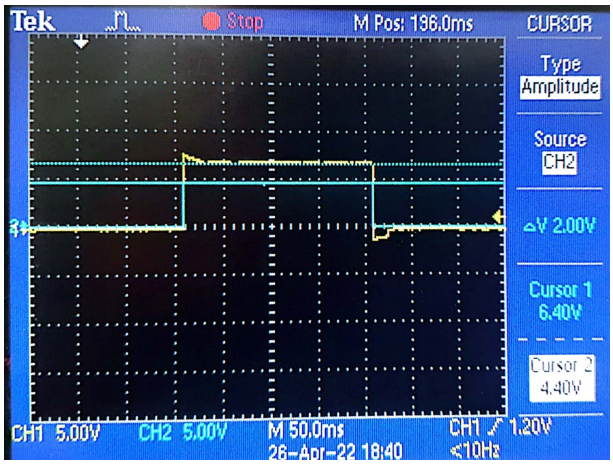


Figura 4.6: Tensões do solenoide e do registrador de deslocamento na ativação. (Fonte: o Autor)

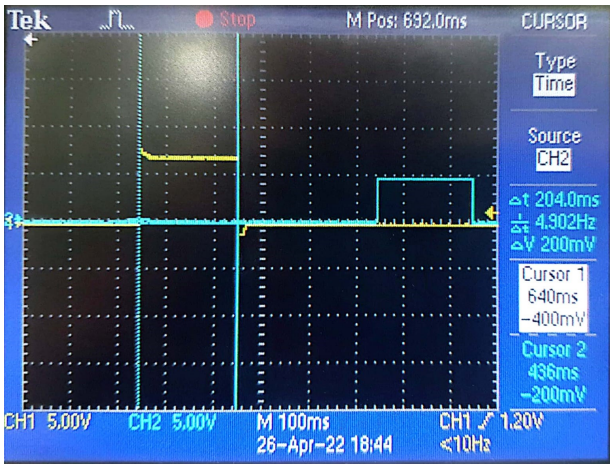
de acordo com a Figura 4.7(b), um espaçamento de 300 ms. Ambas características estão definidas, para garantir, respectivamente, a duração de pulso suficiente para a comutação correta da válvula e o espaçamento entre os pulsos, ativando apenas uma válvula por vez.

4.2.2 Teste de acionamento em campo

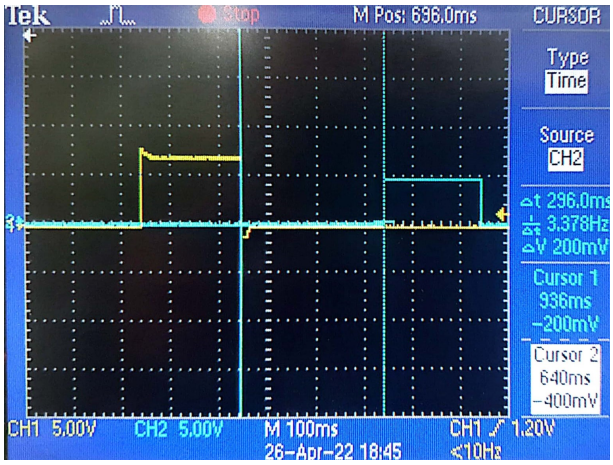
Um arranjo hidráulico com dez válvulas solenoides é montado nas dependências do Laboratório de Hidráulica da UFRPE para validação do circuito. Inicialmente o circuito proposto é montado utilizando algumas matrizes de contatos e cabos de prototipação, chamados comumente de *jumpers*, para verificar o acionamento correto das válvulas. O circuito é apresentado na Figura 4.8.

Posteriormente, com as placas já confeccionadas, os circuitos são testados novamente. O circuito com a placa definitiva é apresentado na Figura 4.9, em que é possível ver o arranjo hidráulico utilizado. Em ambos os casos, uma bomba hidráulica é conectada à parte inferior de uma caixa d'água e ativada, pressurizando o sistema e permitindo o retorno somente pelos ramos com as válvulas presentes.

Para realizar a verificação indivual do acionamento e desativamento das válvulas, os registros manuais abaixo de cada válvula são fechados, excetuando a válvula de teste. Então, o circuito é acionado e desativado utilizando o estado de pressurização



(a) Pulso aplicado para ativação do solenoide.



(b) Intervalo entre as ativações

Figura 4.7: Formas de onda da tensão de saída da válvula solenoide 1 e da tensão de controle da válvula 2. (Fonte: o Autor)

que habilita ou desabilita todas as válvulas, a fim de verificar a comutação da válvula de teste. O processo é repetido para cada uma das válvulas solenoides, validando a comutação correta dos dispositivos.

Para testar o estado de irrigação propriamente dito, uma configuração com valores reduzidos de tempo é escolhida para cada válvula e o tempo de cada ativação e desativação é conferido a partir de um cronômetro até o fim da execução de todos os eventos de irrigação. Também são testadas outras rotinas de irrigação com tempos diferentes, a fim de conferir o acionamento correto do solenoide e validar o tempo correto de atuação de cada evento. Desse modo, todas as funcionalidades do circuito são testadas e obteve-se resultados satisfatórios, realizando o acionamento

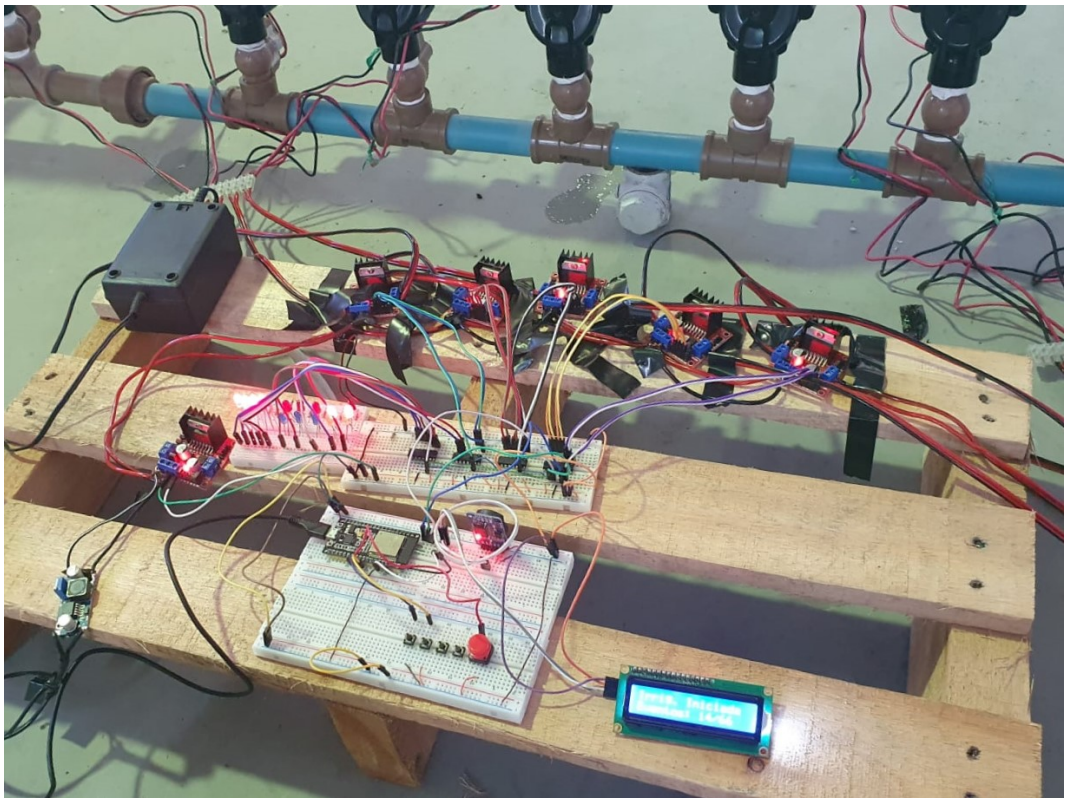


Figura 4.8: Experimento com malha hidráulica para validação do protótipo do projeto montado com matrizes de contatos. (Fonte: o Autor)

ou desativamento das válvulas de acordo com a irrigação programada e executando cada um dos eventos em tempo preciso.

Portanto, o *hardware* e o *software* desenvolvidos para a aplicação, após os testes e validações, garantem que o sistema tenha baixo consumo energético, utilizando válvulas *latching* e alimentação elétrica por bateria. Além disso, os módulos utilizados e descritos anteriormente podem ser facilmente encontrados, possibilitando a réplica do sistema.

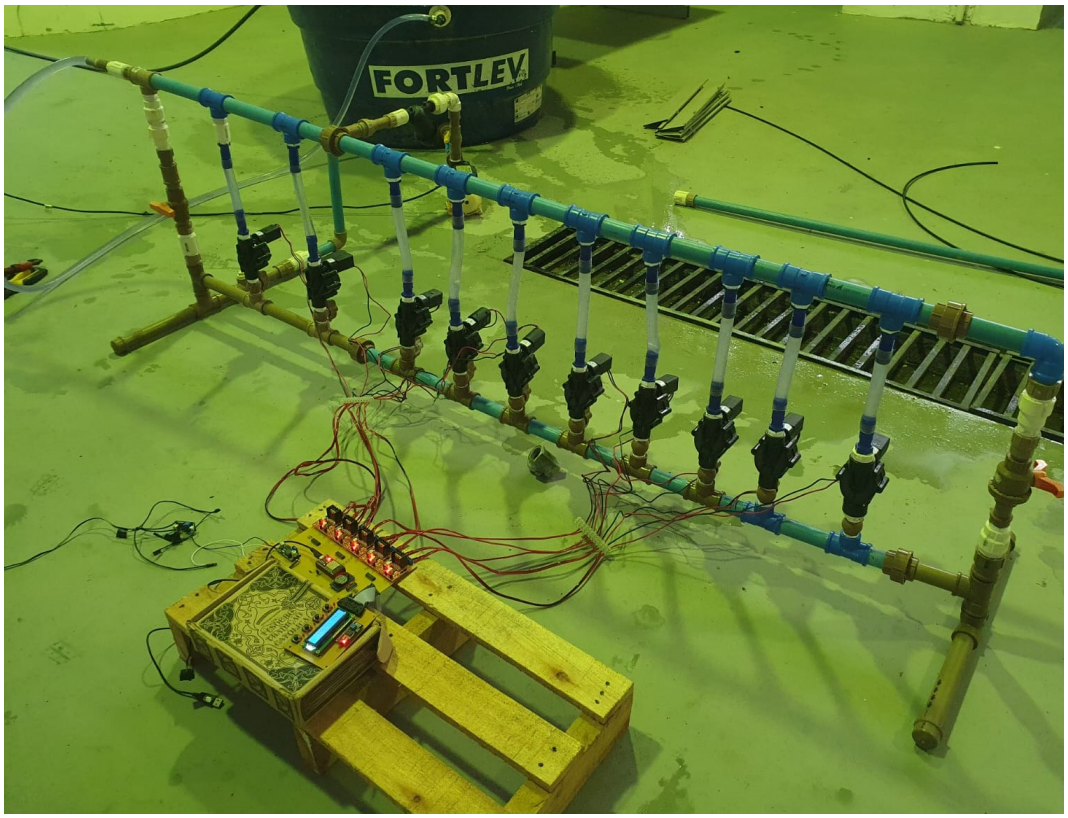


Figura 4.9: Experimento com malha hidráulica para validação das placas confeccionadas. (Fonte: o Autor)

Capítulo 5

Considerações finais

5.1 Conclusões

Neste trabalho apresenta-se um sistema embarcado, implementado em *hardware* e *software*, alimentado por bateria, para controle e acionamento simultâneo de dez válvulas solenóides do tipo *latching*, aplicadas à irrigação pulsada. Com a confecção de duas placas de circuito impresso, uma de controle e uma de interface para o usuário, o sistema desenvolvido pode ser configurado e habilitado por um operador, a fim de substituir o processo de acionamento manual das válvulas durante a execução da irrigação pulsada.

Os componentes e módulos eletrônicos podem ser facilmente adquiridos no mercado, permitindo a réplica do sistema e servindo como base para outras implementações. O projeto apresenta alternativas para a escolha do *hardware* para acionamento independente e simultâneo dos solenoides do tipo *latching*, utilizando poucas portas do microcontrolador e respeitando às limitações elétricas do circuito. Além disso, a caixa de acrílico permite a fácil identificação dos dispositivos utilizados, garantindo assim, uma característica didática para o sistema desenvolvido.

A plataforma Arduino é utilizada para a implementação do *software*, por ser uma tecnologia difundida e de fácil acesso, permitindo que outros estudantes possam sentir-se confortáveis em fazer modificações neste trabalho, mesmo que não sejam da área de eletrônica.

O sistema implementado cumpre os objetivos e requisitos propostos por meio do desenvolvimento de uma solução completa, cuidadosamente testada e validada nas etapas de desenvolvimento e de prototipação utilizando um arranjo hidráulico de irrigação com dez válvulas solenóides do tipo *latching*.

5.2 Dificuldades encontradas

Durante a elaboração do projeto diversos desafios foram encontrados e posteriormente superados. A especificação das válvulas solenóides do tipo *latching* precisou ser alterada devido a indisponibilidade do fornecedor para um modelo de 12 V inicialmente especificado. Entretanto, com o ajuste do potenciômetro do regulador de entrada, foi possível mudar a alimentação para 9 V. Outro desafio foi a elaboração de um circuito para a atuação de várias válvulas, gerando um circuito maior e mais complexo. Assim, a dificuldade inerente na confecção de uma placa de circuito extensa foi garantir a qualidade da usinagem uniforme da placa, devido ao desgate da mesa de sacrifício da CNC e a falta de experiência no seu manuseio. Portanto, diversas correções foram necessárias para retirar as imperfeições da impressão que estavam provocando curto em algumas partes do circuito.

5.3 Trabalhos futuros

O sistema desenvolvido foi pensado para permitir a execução de diversas melhorias de funcionalidade, modificando apenas o *firmware* do dispositivo. Portanto, como visões futuras para o projeto, podem ser desenvolvidos:

- Um aplicativo para a utilização em conjunto com o ESP32, que permite a comunicação WiFi e BLE, para configuração das válvulas e monitoramento da execução dos eventos de irrigação em tempo real;
- A utilização do módulo SD em conjunto com o sistema para armazenar os dados de atuação das válvulas e as rotinas executadas;

- A elaboração de otimizações energéticas do sistema, além da utilização das válvulas *latching*, para permitir uma maior durabilidade das baterias utilizadas;
- A criação de um sistema *web* para ativação remota das rotinas de irrigação;
- Agregar uma rede de sensores sem fio capaz de se comunicar utilizando WiFi ou BLE, para implementar um controle em malha fechada da irrigação.

Referências Bibliográficas

[1] GOLOMB, S. W. *Shift register sequences: secure and limited-access code generators, efficiency code generators, prescribed property generators, mathematical models*. World Scientific, 2017.

[2] WAKERLY, J. *Digital Design: Principles and Practices*. Pearson, 2018. ISBN: 013446009X; 9780134460093.

[3] STMICROELECTRONICS. "8 bit shift register with output latches 3 state". 1994. Disponível em: <https://pdf1.alldatasheet.com/datasheet-pdf/view/23121/STMICROELECTRONICS/74HC595.html>. Acesso em: 05 out. 2022.

[4] STACKEXCHANGE. "Cascade shift registers driven by single-cycle microcontroller". 2019. Disponível em: <https://electronics.stackexchange.com/questions/417341/cascade-shift-registers-driven-by-single-cycle-microcontroller>. Acesso em: 05 out. 2022.

[5] SILVEIRA, C. B. "Como Funciona a Válvula Solenoide e Quais os Tipos?" 2017. Disponível em: <https://www.citisystems.com.br/valvula-solenoide/>. Acesso em: 05 out. 2022.

[6] GONZÁLEZ, J. J. C. "Domótica aplicada al monitoreo de llaves de Agua". 2015. Disponível em: <https://www.gestiopolis.com/domotica-aplicada-al-monitoreo-de-llaves-de-agua/>. Acesso em: 05 out. 2022.

[7] SIEBEN, V. "A High Power H-Bridge". 2003.

[8] PATSKO, L. F. "Tutorial Montagem da Ponte H". 2006. Disponível em: <https://www.robocore.net/upload/attachments/ponte_h_590.pdf>. Acesso em: 05 out. 2022.

[9] BRAGA, N. C. “Conheça a família TTL (MEC082)”. 2022. Disponível em: <https://www.newtoncbraga.com.br/index.php/robotica/3790>. Acesso em: 05 out. 2022.

[10] COMPONENTS101. “L298N Motor Driver Module”. 2021. Disponível em: <https://components101.com/modules/l293n-motor-driver-module>. Acesso em: 05 out. 2022.

[11] VALDEZ, J., BECKER, J. “Understanding the I2C Bus”, *Texas Instruments, Texas*, 2018.

[12] GAY, W. “I2C LCD Displays”. In: *Custom Raspberry Pi Interfaces*, Springer, pp. 35–54, 2017.

[13] ELECTROBIST. “DS3231 Precision RTC Real Time Clock Module”. 2022. Disponível em: <https://electrobist.com/product/ds3231-precision-rtc-real-time-clock-module/>. Acesso em: 05 out. 2022.

[14] LEANTEC. “PLACA DE DESARROLLO NODEMCU ESP32 WIFI + BLUETOOTH ESP WROOM 32 38 PINES”. 2022. Disponível em: <https://leantec.es/wp-content/uploads/2020/02/0815-00.jpg>. Acesso em: 05 out. 2022.

[15] BOXALL, J. *Arduino workshop: a hands-on introduction with 65 projects*. No Starch Press, 2013. ISBN: 1593274483; 9781593274481; 1593275250; 9781593275259.

[16] BOYLESTAD, R. L., NASHELSKY, L. *Electronic devices and circuit theory*. Prentice Hall, 2012.

[17] INSTRUMENTS, T. “LM2596 SIMPLE SWITCHER® Power Converter 150-kHz 3-A Step-Down Voltage Regulator”. 2021. Disponível em: <https://www.ti.com/lit/ds/symlink/lm2596.pdf?ts=1664575861792&ref_url=https%253A%252F%252Fwww.google.at%252F>. Acesso em: 05 out. 2022.

[18] ONSEMI. “LM2596 3.0 A, Step-Down Switching Regulator”. 2008. Disponível em: <https://www.onsemi.com/pdf/datasheet/lm2596-d.pdf>. Acesso em: 05 out. 2022.

[19] BARBACENA, I. L., FLEURY, C. A. “Display lcd”. 1996. Disponível em: <https://wiki.sj.ifsc.edu.br/images/9/97/DsiplayLcd-unicamp.pdf>. Acesso em: 05 out. 2022.

[20] ALMEIDA, W. F. D., LIMA, L. A., PEREIRA, G. M. "Drip pulses and soil mulching effect on american crisphead lettuce yield", *Engenharia Agrícola*, v. 35, pp. 1009–1018, 2015.

[21] BUSATO, C. C. M., SOARES, A. A., SEDIYAMA, G. C., et al. "Manejo da irrigação e fertirrigação com nitrogênio sobre as características químicas da videira'Niágara Rosada'", *Ciência Rural*, v. 41, pp. 1183–1188, 2011.

[22] ANDRADE, W. J. M. *Efeito do gotejamento contínuo e pulsado e lâminas de irrigação sobre aspectos fisiológicos, acúmulo e exportação de nutrientes na cana-de-açúcar*. Dissertação, Universiade Federal Rural de Pernambuco, Recife, PE, 2021.

[23] AYARS, J. E., PHENE, C. J. "Automation". In: *Developments in Agricultural Engineering*, v. 13, Elsevier, pp. 259–284, 2007.

[24] ASSOULINE, S., MÖLLER, M., COHEN, S., et al. "Soil-plant system response to pulsed drip irrigation and salinity: Bell pepper case study", *Soil Science Society of America Journal*, v. 70, n. 5, pp. 1556–1568, 2006.

[25] EL-ABEDIN, T. Z. "Effect of pulse drip irrigation on soil moisture distribution and maize production in clay soil", *New Trends in Agricultural Engineering*, v. 22, pp. 1032–1050, 2006.

[26] WARNER, R., HOFFMAN, O., WILHOIT, J. "The effects of pulsing drip irrigation on tomato yield and quality in Kentucky", *Fruit and Vegetable Crop Research Report*, v. 1, pp. 39–40, 2009.

[27] SANTOS, C. C. D. *Sistema de sensoriamento remoto de umidade e temperatura do solo para irrigação de precisão*. Dissertação (mestrado), Universidade Federal do Ceará. Departamento de Engenharia Elétrica, Fortaleza, CE, 2008.

[28] LEE, T.-Y., HSIUNG, P.-A. "Embedded software synthesis and prototyping", *IEEE Transactions on Consumer Electronics*, v. 50, n. 1, pp. 386–392, 2004.

[29] RIBEIRO, M. A. "Instrumentação", *Tek Treinamentos LTDA*, v. 16, 1999.

[30] VASCONCELOS, H. S. *Automação de sistema de irrigação em malha fechada utilizando rede sem fio de sensores capacitivos de umidade do solo*. Dissertação (mestrado), Universidade Federal do Ceará, Centro de Ciências Agrárias, Departamento de Engenharia Agrícola, Programa de Pós-Graduação em Engenharia Agrícola, Fortaleza, CE, 2013.

[31] BERMAD. “Magnectic latch solenoid actuator”. 2022. Disponível em: <https://catalog.bermad.com/BERMAD%20Assets/Irrigation/Solenoids/IR-SOLENOID-S-392T-2W/IR_Accessories-Solenoid-S-392T-2W_Product-Page_English_2-2020_XSB.pdf>. Acesso em: 05 out. 2022.

[32] FÉ, B. O. C. D. *Roteador nanoeletrônico para redes-em-chip baseado em transistores monoelétron.* dissertação, Universidade de Brasília, Brasília, DF, 2017.

[33] STMICROELECTRONICS. “Dual full-bridge driver”. 2000. Disponível em: <https://pdf1.alldatasheet.com/datasheet-pdf/view/22440/STMICROELECTRONICS/L298N.html>. Acesso em: 05 out. 2022.

[34] SPARKFUN. “Pull-up Resistors”. 2022. Disponível em: <https://learn.sparkfun.com/tutorials/pull-up-resistors/all>. Acesso em: 05 out. 2022.

[35] JACINTO, M. P. *JR-Control: dispositivo para monitoramento de máquinas industriais com controle de temperatura ambiente.* Trabalho de conclusão de curso, 2020.

[36] MANGUEIRA, J. I. S. *Desenvolvimento de um sistema de aquisição de dados para experimentação com modelos de vigas retas.* Trabalho de conclusão de curso, 2022.

[37] AKINWOLE, O. “Design, simulation and implementation of an Arduino microcontroller based automatic water level controller with I2C LCD display”, *International Journal of Advances in Applied Sciences (IJAAS)*, v. 9, n. 2, pp. 77–84, 2020.

[38] INSTRUMENTS, T. “PCF8574 Remote 8-Bit I/O Expander for I2C Bus”. 2015. Disponível em: <https://www.ti.com/lit/ds/symlink/pcf8574.pdf?ts=1664869729510&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FPCF8574>. Acesso em: 05 out. 2022.

[39] HISTORY-COMPUTER. “What is a Real-Time Clock (RTC) and Why are They Important in Computing?” 2022. Disponível em: <https://history-computer.com/real-time-clock/>. Acesso em: 05 out. 2022.

[40] MNATI, M. J., CHISAB, R. F., AL-RAWI, A. M., et al. “An open-source non-contact thermometer using low-cost electronic components”, *HardwareX*, v. 9, pp. e00183, 2021.

[41] KURIA, K. P., ROBINSON, O. O., GABRIEL, M., et al. "Monitoring temperature and humidity using arduino nano and module-DHT11 sensor with real time DS3231 data logger and LCD display", *Health Hyg*, v. 6, n. 7, pp. 8, 2020.

[42] MAXIM. "Extremely Accurate I2C-Integrated RTC/TCXO/Crystal Typical Operating Circuit". 2015. Disponível em: <https://electrosome.com/datasheet/ds3231-rtc/>. Acesso em: 05 out. 2022.

[43] MAIER, A., SHARP, A., VAGAPOV, Y. "Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things". In: *2017 Internet Technologies and Applications (ITA)*, pp. 143–148. IEEE, 2017.

[44] SOUZA, F. "Como programar o ESP32 na Arduino IDE?" 2022. Disponível em: <https://embarcados.com.br/como-programar-o-esp32-na-arduino-ide/>. Acesso em: 05 out. 2022.

[45] RITCHIE, D. M. "The development of the C language", *ACM Sigplan Notices*, v. 28, n. 3, pp. 201–208, 1993.

[46] KORMANYOS, C. *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming*. Springer, 2021. ISBN: 3662629968; 9783662629963; 366262995X; 9783662629956.

[47] CASS, S. "Top Programming Languages 2022 Python's still No. 1, but employers love to see SQL skills". 2022. Disponível em: <https://spectrum.ieee.org/top-programming-languages-2022/ieee-spectrums-top-programming-languages-2022>. Acesso em: 05 out. 2022.

[48] BANZI, M., MICHAEL, S. *Make: Getting Started with Arduino: The Open Source Electronics Prototyping Platform, 4th Edition*. Make Community, 2022. ISBN: 9781680456936; 1680456938.

[49] BADAMASI, Y. A. "The working principle of an Arduino". In: *2014 11th international conference on electronics, computer and computation (ICECCO)*, pp. 1–4. IEEE, 2014.

[50] TOSHIBA. "0.5A Three terminal positive voltage regulators". 1999. Disponível em: <https://pdf1.alldatasheet.com/datasheet-pdf/view/31368/TOSHIBA/78M05.html>. Acesso em: 05 out. 2022.

[51] KHAING, K. K., NWET, T. T., NAING, T. "DANCING ROBOT USING ARDUINO", *Journal of Research and Applications*, v. 01, n. 1, 2019.

[52] AMANPREET SINGH, ADARSH SACHAN, K. G. G. K. H. K. S. A. S. "DANCING ROBOT USING ARDUINO", *International Research Journal of Modernization in Engineering Technology and Science*, 2022. Disponível em: <https://www.irjmets.com/uploadedfiles/paper/issue_5_may_2022/22792/final/fin_irjmets1652445748.pdf>. Acesso em: 05 out. 2022.

Apêndice A

Código Fonte

A.1 Arquivo *ValvulaUtil.h*

```
1  // ValvulaUtil.h - Biblioteca utilitária para valvulas para elaboração do TCC
2  // Autor: Railton Silva Rocha Junior
3  // Github: https://github.com/rsrj/
4  #ifndef ValvulaUtil_H
5  #define ValvulaUtil_H
6
7  #include <Arduino.h>
8  #include <Preferences.h>
9  #include "LiquidCrystal_I2C.h"
10
11 #define SIZE_TYPE_VALV sizeof(valvula_t)
12 #define countof(a) (sizeof(a) / sizeof(a[0]))
13
14 #define MINUTES_IN_SECONDS 60
15 #define HOUR_IN_SECONDS MINUTES_IN_SECONDS*60
16 #define DAY_IN_SECONDS HOUR_IN_SECONDS*24
17
18 #define TEMPO_PRESSURIZACAO 4 //tempo de pressurizacao em minutos
19
20 typedef struct valvula_t{
21         uint16_t tempoIrrigacao;
22         uint8_t numPulsos;
23         uint8_t interPulsos;
24         void operator= (valvula_t v);
25 } valvula_t;
26
27 /*Variavel responsavel por armazenar os eventos de irrigacao*/
28 typedef struct evento_irrigacao_t{
29         uint8_t indexValvula;
30         bool acao; //on - true, off - false
```

```
31          uint32_t horaProgramada;
32          bool foiAtendido;  //true caso já tenha sido executado, false caso ainda
    ↪  precise executar
33          void operator= (evento_irrigacao_t e);
34  } evento_irrigacao_t;
35
36  /*Estados para que cuidam do que será exibido no LCD*/
37  typedef enum{
38    LCD_EXIBE,
39    TRANS_TO_CONFIG,          //Marca a transição de estado da exibição para a
    ↪  configuração
40    LCD_CONFIG,
41    TRANS_TO_EXIBE,          //Marca a transição de estado da configuração para a
    ↪  exibição
42    TEMPO_EXIBE,
43    TEMPO_CONFIG,
44    CONFIRM_IRRIG,
45    INIT_IRRIG,
46    IRRIGATION,
47    INIT_PRESSUR,
48    PRESSURIZATION,
49    STOP_PRESSUR,
50  }STATES_LCD;
51
52  class ValvulaUtil{
53          protected:
54                  bool _started_;
55          public:
56                  /*Construtores e destrutores da classe utilitária válvula útil*/
57                  ValvulaUtil();
58                  ValvulaUtil(uint8_t maxNumValv, LiquidCrystal_I2C& lcd);
59                  ~ValvulaUtil();
60
61                  /*Inicia o objeto ValvulaUtil*/
62                  bool begin();
63                  /*Finaliza o objeto ValvulaUtil*/
64                  void end();
65                  /* Faz a persistência da válvula para a memória,
66                  *  verifica o valor armazenado na memoria, caso os mesmos dados
67                  *  ja estejam na memoria, nao faz o salvamento para poupar a
    ↪  FLASH
68                  *  Retorno: retorna true caso tenha realizado o salvamento com
    ↪  êxito
69                  *  retorna false caso contrário. */
70                  bool salvaValvulaFlash(uint8_t indexValv, valvula_t valv);
71
72
73                  /* Faz a leitura da válvula na memória, e armazena o valor na
```

```
74                  *  estrutura passada como referência.
75                  *  Retorno: retorna o true caso tenha conseguido fazer o
    ↪ salvamento com sucesso
76                  *  retorna false caso contrário.*/
77                  bool recuperaValvulaFlash(uint8_t indexValv, valvula_t& valv);
78
79                  /** Descrição: método auxiliar que faz a impressao no LCD */
80                  void imprimeLCD(const char* primeiraLinha, const char*
    ↪ segundaLinha, bool limpa);
81
82                  /** Faz a impressao da tela de transicao de exibicao no LCD */
83                  void telaValvula(int index, valvula_t& valvula, bool limpa);
84
85                  /** Faz a impressao da tela de configuracao de tempo de irrigacao
    ↪ */
86                  void telaConfigTempIrrig(int index, int tempoIrrigacao);
87
88                  /** Faz a impressao da tela de configuracao de tempo de irrigacao
    ↪ */
89                  void telaConfigNumPulsos(int index, int numeroPulsos);
90
91                  /** Faz a impressao da tela de transicao de exibicao no LCD */
92                  void telaConfigInterPulsos(int index, int intervaloPulsos);
93
94                  /** Faz a impressao da tela de transicao configuracao no LCD */
95                  void telaTransicaoConfig(int index);
96
97                  /** Faz a impressao da tela de transicao de exibicao no LCD */
98                  void telaTransicaoExib(int index);
99
100                 /** Faz a impressao da tela de transicao de irrigacao no LCD */
101                 void telaConfirmaIrrig();
102
103                 /** Faz a impressao da tela de inicio da irrigacao no LCD */
104                 void telaInitIrrig();
105
106                  /** Faz a impressao da tela de quando a irrigacao nao e possivel
    ↪ */
107                 void telaIrrigNaoPossivel();
108
109                 /** Faz a impressao da tela de quando a irrigacao foi iniciada
    ↪ */
110                 void telaIrrigacao(int contador, int total);
111
112                 /** Faz a impressao da tela de antes do início da
    ↪ pressurizacao*/
113                 void telaDataHora(const uint8_t& rtcDay, const uint8_t& rtcMonth,
    ↪ const uint16_t& rtcYear,
```

```
114                         const uint8_t& rtcHour, const uint8_t& rtcMinute, const
    ↪  uint8_t& rtcSecond);
115
116                 /** Faz a impressao da tela com o tempo de pressurização*/
117                 void telaInicioPressurizacao();
118
119                 /** Faz a impressao da tela de relogio no LCD*/
120                 void telaPressurizacao(const uint8_t& rtcHour, const uint8_t&
    ↪  rtcMinute,const uint8_t& rtcSecond);
121
122                 /* Função que converte uma configuração de válvula recebida e
    ↪  transforma em um vetor de agendamentos
123                 A quantidade de eventos de ativação/desativação da válvula é o
    ↪  dobro do NP
124                 (cada pulso gera um evento de ativação no início e de desativação
    ↪  no final) */
125                 void agendamentoAPartirDaValvula (uint8_t index, valvula_t
    ↪  valvula, uint32_t tempoInicio, evento_irrigacao_t* agenda);
126
127                 /*Função que verifica o tamanho total necessário para a agenda
    ↪  */
128                 int numeroTotalEventos(valvula_t *valvulas);
129         private:
130                 Preferences _prefs;
131                 uint8_t _maxNumValv;
132                 LiquidCrystal_I2C _lcd;
133 };
134
135 #endif
136
137
```

A.2 Arquivo *ValvulaUtil.cpp*

```
1  // ValvulaUtil.cpp - Biblioteca utilitária para valvulas para elaboração do TCC
2  // Autor: Railton Silva Rocha Junior
3  // Github: https://github.com/rsrj/
4
5  #include "ValvulaUtil.h"
6
7  void evento_irrigacao_t::operator= (evento_irrigacao_t e) {
8          indexValvula = e.indexValvula;
9          acao = e.acao; //on - true, off - false
10         horaProgramada = e.horaProgramada;
11         foiAtendido = e.foiAtendido;
12 }
13
```

```
14  void valvula_t::operator= (valvula_t v) {
15          tempoIrrigacao = v.tempoIrrigacao;
16          numPulsos = v.numPulsos;
17          interPulsos = v.interPulsos;
18  }
19
20  ValvulaUtil::ValvulaUtil(uint8_t maxNumValv, LiquidCrystal_I2C& lcd)
21          :_started_(false),
22          _maxNumValv(maxNumValv),
23          _lcd(lcd)
24  {}
25
26  ValvulaUtil::~ValvulaUtil(){
27          end();
28  }
29
30  bool ValvulaUtil::begin(){
31          if(_started_){
32          return false;
33      }
34          if(!_prefs.begin("valvulas")){
35                  return false;
36          }
37          _lcd.begin();
38          _started_ = true;
39          return true;
40  }
41
42  void ValvulaUtil::end(){
43          if(!_started_){
44                  return;
45          }
46          _prefs.end();
47          _started_ = false;
48  }
49
50
51   bool ValvulaUtil::salvaValvulaFlash(uint8_t indexValv, valvula_t valv){
52          if(indexValv <= 0 || indexValv>_maxNumValv){
53                  return false;
54          }
55          //Constroi a string key que será usada para armazenar na Flash
56      char keyValv[15];
57      sprintf(keyValv,"valvula_%d", indexValv);
58
59      char buffRec[SIZE_TYPE_VALV];
60
61      //Se retornar valor a chave existe e o valor salvo na Flash foi recuperado,
    ↪  verifica se são iguais
```

```
62      if(_prefs.getBytes(keyValv, buffRec, SIZE_TYPE_VALV)){
63                  if((valv.tempoIrrigacao ==
    ↪   ((uint16_t)(buffRec[0]<<8)+buffRec[1]))
64                          &&(valv.numPulsos == buffRec[2])
65                          &&(valv.interPulsos == buffRec[3])){
66                          Serial.printf("\nDados iguais, nao faz salvamento");
67                          return true; //Dados iguais, não é necessário realizar
    ↪   novo salvamento;
68        }
69      }
70      /*Cria buffer para salvamento*/
71      char buffer[SIZE_TYPE_VALV] = {(uint8_t)((valv.tempoIrrigacao)>>8), //upper
    ↪   byte do uint16_t
72                  (uint8_t) valv.tempoIrrigacao,
73                  (uint8_t) valv.numPulsos,
74                  (uint8_t) valv.interPulsos};
75      /*Persiste buffer na memória caso não consiga retorna falso*/
76      if(!(_prefs.putBytes(keyValv, buffer, SIZE_TYPE_VALV))){
77                  Serial.printf("\nNao foi possivel salvar.");
78          return false;
79      }
80      Serial.printf("\nFaz o salvamento");
81      return true;
82  }
83
84  bool ValvulaUtil::recuperaValvulaFlash(uint8_t indexValv, valvula_t& valv){
85
86          if (indexValv <= 0 || indexValv>_maxNumValv) {
87                  return false; //Valida se o índice está no intervalo válido
88          }
89          //Constroi a string key que será usada para armazenar na Flash
90          char keyValv[20];
91          sprintf(keyValv,"valvula_%d", indexValv);
92
93          char buff[SIZE_TYPE_VALV];
94
95          if (_prefs.getBytes(keyValv, buff, SIZE_TYPE_VALV)) {
96                  valv.tempoIrrigacao = (((uint16_t)buff[0])<<8) + buff[1];
97                  valv.numPulsos = buff[2];
98                  valv.interPulsos = buff[3];
99
100                 return true;
101         }
102
103         return false;
104 }
105
106 void ValvulaUtil::imprimeLCD(const char* primeiraLinha, const char* segundaLinha,
    ↪   bool limpa = true){
```

```
107         if (limpa) {
108                 _lcd.clear();
109         } else {
110                 _lcd.setCursor(0, 0);
111         }
112         _lcd.print(primeiraLinha);
113         _lcd.setCursor(0, 1);
114         _lcd.print(segundaLinha);
115 }
116
117 void ValvulaUtil::telaValvula(int index, valvula_t& valvula, bool limpa = true){
118         char primeiraLinha[17];
119         char segundaLinha[17];
120
121         snprintf_P(primeiraLinha,
122                 countof(primeiraLinha),
123                 PSTR("V.%03d TI:%4dmin"),
124                 index,
125                 valvula.tempoIrrigacao);
126         snprintf_P(segundaLinha,
127                 countof(segundaLinha),
128                 PSTR("NP:%2d IP:%4dmin"),
129                 valvula.numPulsos,
130                 valvula.interPulsos);
131
132         imprimeLCD(primeiraLinha, segundaLinha, limpa);
133 }
134
135 void ValvulaUtil::telaConfigTempIrrig(int index, int tempoIrrigacao){
136         char primeiraLinha[17];
137         char segundaLinha[17];
138
139         snprintf_P(primeiraLinha,
140                 countof(primeiraLinha),
141                 PSTR("V.%03d TempoIrrig"),
142                 index);
143         snprintf_P(segundaLinha,
144                 countof(segundaLinha),
145                 PSTR("CONF.TI: %4dmin"),
146                 tempoIrrigacao);
147
148         imprimeLCD(primeiraLinha, segundaLinha, false);
149 }
150
151 void ValvulaUtil::telaConfigNumPulsos(int index, int numeroPulsos){
152         char primeiraLinha[17];
153         char segundaLinha[17];
154
```

```
155         snprintf_P(primeiraLinha,
156                 countof(primeiraLinha),
157                 PSTR("V.%03d NumPulsos"),
158                 index);
159         snprintf_P(segundaLinha,
160                 countof(segundaLinha),
161                 PSTR("CONF.NP: %4d"),
162                 numeroPulsos);
163 
164         imprimeLCD(primeiraLinha, segundaLinha, false);
165 }
166 
167 void ValvulaUtil::telaConfigInterPulsos(int index, int intervaloPulsos){
168         char primeiraLinha[17];
169         char segundaLinha[17];
170 
171         snprintf_P(primeiraLinha,
172                 countof(primeiraLinha),
173                 PSTR("V.%03d InterPulso"),
174                 index);
175         snprintf_P(segundaLinha,
176                 countof(segundaLinha),
177                 PSTR("CONF.IP: %4dmin"),
178                 intervaloPulsos);
179 
180         imprimeLCD(primeiraLinha, segundaLinha, false);
181 }
182 
183 void ValvulaUtil::telaTransicaoConfig(int index){
184         char primeiraLinha[17];
185         char segundaLinha[17];
186 
187         snprintf_P(primeiraLinha,
188                 countof(primeiraLinha),
189                 PSTR("     Config."));
190         snprintf_P(segundaLinha,
191                 countof(segundaLinha),
192                 PSTR("    Valvula %d"),
193                 index);
194 
195         imprimeLCD(primeiraLinha, segundaLinha, false);
196 }
197 
198 void ValvulaUtil::telaTransicaoExib(int index){
199         char primeiraLinha[17];
200         char segundaLinha[17];
201         snprintf_P(primeiraLinha,
202                 countof(primeiraLinha),
```

```
203                 PSTR(" Config. Valv %d "),
204                 index);
205         snprintf_P(segundaLinha,
206                 countof(segundaLinha),
207                 PSTR("   Concluida   "),
208                 index);
209
210         imprimeLCD(primeiraLinha, segundaLinha, false);
211 }
212
213 void ValvulaUtil::telaConfirmaIrrig(){
214         char primeiraLinha[17];
215         char segundaLinha[17];
216         snprintf_P(primeiraLinha,
217                 countof(primeiraLinha),
218                 PSTR("Confirmar Irrig?"));
219         snprintf_P(segundaLinha,
220                 countof(segundaLinha),
221                 PSTR(" Press. Config  "));
222
223         imprimeLCD(primeiraLinha, segundaLinha, false);
224 }
225
226 void ValvulaUtil::telaInitIrrig(){
227         char primeiraLinha[17];
228         char segundaLinha[17];
229         snprintf_P(primeiraLinha,
230                 countof(primeiraLinha),
231                 PSTR("   Iniciando    "));
232         snprintf_P(segundaLinha,
233                 countof(segundaLinha),
234                 PSTR("   Irrigacao    "));
235
236         imprimeLCD(primeiraLinha, segundaLinha, false);
237 }
238
239 void ValvulaUtil::telaIrrigNaoPossivel(){
240         char primeiraLinha[17];
241         char segundaLinha[17];
242         snprintf_P(primeiraLinha,
243                 countof(primeiraLinha),
244                 PSTR("   Irrigacao    "));
245         snprintf_P(segundaLinha,
246                 countof(segundaLinha),
247                 PSTR("  Nao possivel  "));
248
249         imprimeLCD(primeiraLinha, segundaLinha, false);
250 }
```

```
251
252 void ValvulaUtil::telaIrrigacao(int contador, int total){
253         char primeiraLinha[17];
254         char segundaLinha[17];
255         snprintf_P(primeiraLinha,
256                 countof(primeiraLinha),
257                 PSTR("Irrig. Iniciada "));
258         snprintf_P(segundaLinha,
259                 countof(segundaLinha),
260                 PSTR("Eventos: %2d/%2d"), contador, total );
261
262         imprimeLCD(primeiraLinha, segundaLinha, false);
263 }
264
265
266 void ValvulaUtil::telaInicioPressurizacao(){
267         char primeiraLinha[17];
268         char segundaLinha[17];
269         snprintf_P(primeiraLinha,
270                 countof(primeiraLinha),
271                 PSTR("     Iniciar    "));
272         snprintf_P(segundaLinha,
273                 countof(segundaLinha),
274                 PSTR(" Pressurizacao? "));
275
276         imprimeLCD(primeiraLinha, segundaLinha, false);
277 }
278
279 void ValvulaUtil::telaPressurizacao(const uint8_t& rtcHour,
280                 const uint8_t& rtcMinute,
281                 const uint8_t& rtcSecond){
282         char primeiraLinha[17];
283         char segundaLinha[17];
284         snprintf_P(primeiraLinha,
285                 countof(primeiraLinha),
286                 PSTR("  Pressurizando  "));
287         snprintf_P(segundaLinha,
288                         countof(segundaLinha),
289                         PSTR(" HORA: %02u:%02u:%02u "),
290                         rtcHour,
291                         rtcMinute,
292                         rtcSecond);
293
294         imprimeLCD(primeiraLinha, segundaLinha, false);
295 }
296
297
298 void ValvulaUtil::telaDataHora(const uint8_t& rtcDay,
```

```
299                 const uint8_t& rtcMonth,
300                 const uint16_t& rtcYear,
301                 const uint8_t& rtcHour,
302                 const uint8_t& rtcMinute,
303                 const uint8_t& rtcSecond){
304         char primeiraLinha[17];
305         char segundaLinha[17];
306         //snprintf_P lê a string de formato da FLASH, poupando a RAM
307         snprintf_P(primeiraLinha,
308                         countof(primeiraLinha),
309                         PSTR("DATA: %02u/%02u/%04u"),
310                         rtcDay,
311                         rtcMonth,
312                         rtcYear);
313         snprintf_P(segundaLinha,
314                         countof(segundaLinha),
315                         PSTR("HORA: %02u:%02u:%02u "),
316                         rtcHour,
317                         rtcMinute,
318                         rtcSecond);
319
320         imprimeLCD(primeiraLinha, segundaLinha, false);
321 }
322
323 void ValvulaUtil::agendamentoAPartirDaValvula (uint8_t index, valvula_t valvula,
    ↪  uint32_t tempoInicio, evento_irrigacao_t* agenda){
324   static uint8_t s_index;
325   static valvula_t s_valvula;
326   static uint32_t s_tempoInicio;
327   static evento_irrigacao_t* s_agenda;
328
329   s_index = index;
330   s_valvula = valvula;
331   s_tempoInicio = tempoInicio;
332   s_agenda = agenda;
333
334   if(s_valvula.numPulsos == 0){
335           return;
336   }
337   static uint8_t s_numEventos;
338   s_numEventos = s_valvula.numPulsos*2;
339   //evento_irrigacao_t agenda[numEventos];
340   static uint32_t s_tempoDoPulso;
341   s_tempoDoPulso =
    ↪  (s_valvula.tempoIrrigacao*MINUTES_IN_SECONDS)/s_valvula.numPulsos;
342   for (int i=0; i<s_numEventos; i++) {
343     s_agenda[i].indexValvula = s_index;
344     s_agenda[i].foiAtendido = false;
```

```
345       s_agenda[i].horaProgramada = s_tempoInicio;
346
347       uint8_t resto = i%2; //Se for par o evento deve ser de ativação (true)
348           uint32_t tempoPressurizacao = (i==0 || i==1)? 0 :
    ↪  (TEMPO_PRESSURIZACAO*MINUTES_IN_SECONDS);
349       s_agenda[i].acao = resto? false: true; //se resto for zero significa que é
    ↪  par, evento de ativação. Se resto for um significa que é impar, evento de
    ↪  desativação
350       s_agenda[i].horaProgramada += ((i/2)*(s_tempoDoPulso +
    ↪  s_valvula.interPulsos*MINUTES_IN_SECONDS) + resto*(s_tempoDoPulso) + (i/2 -
    ↪  !resto)*tempoPressurizacao);
351     }
352
353     s_index = 0;
354     s_valvula.numPulsos = 0;
355     s_valvula.tempoIrrigacao = 0;
356     s_valvula.interPulsos = 0;
357     s_tempoInicio = 0;
358     s_tempoDoPulso = 0;
359     s_numEventos = 0;
360     s_agenda = NULL;
361
362   }
363
364   int ValvulaUtil::numeroTotalEventos(valvula_t *valvulas){
365     int totalEventos = 0;
366
367     for (int i = 0; i < _maxNumValv; i++) {
368       totalEventos += (valvulas[i].numPulsos)*2;
369     }
370     return totalEventos;
371   }
```

A.3 Arquivo *Maquina_Estados_wLibrary.ino*

```
1   /**
2    * Maquina_Estados_wLibrary.ino
3    * Autor: Railton Silva Rocha Junior
4    * Github: https://github.com/rsrj/
5    * Descrição: Programa principal, responsável por unir as funções implementadas
    ↪   (Exibição LCD, RTC, Persistência, Maquinas de estados)
6    * em um único programa.
7    */
8
9   #include <Arduino_Helpers.h>
10  #include <AH/STL/algorithm>
11  #include <AH/STL/iterator>
```

```
12
13  #include <ezButton.h>
14  #include <Wire.h>
15  #include "LiquidCrystal_I2C.h"
16  #include <RtcDS3231.h>
17  #include <time.h>
18  #include "ValvulaUtil.h"
19
20
21  #define DEBOUNCE_TIME 50
22  #define NUM_VALVULAS 10
23  #define NUM_MAXIMO_EVENTOS 200
24  #define REGISTER_NUMBER 4
25  #define IRRIG_OFFSET 10
26
27  ezButton btnLeft(4);
28  ezButton btnRight(17);
29  ezButton btnConfig(18);// Reserved pins on schematic  4,    17,      18,     19
30                         //                            LEFT, RIGHT, CONFIG,   TO BE
31                         // Pinos para serem utilizados no futuro
32                         //   SPI
33                         //     12,       13,       14,     15
34                         //     SPI_MISO, SPI_MOSI, SPI_SCK, SPI_CS
35                         // Demais pinos utilizados
36                         //   SHIFT REGISTER
37                         //     16,   25,  26,    27
38                         //     OE, DATA, CLK, LATCH
39                         //   I2C
40                         //      21,  22
41                         //     SDA, SCL
42
43  RtcDS3231<TwoWire> Rtc(Wire);
44  LiquidCrystal_I2C lcd(0x27, 16, 2);
45  ValvulaUtil vUtil(NUM_VALVULAS, lcd);
46
47  /*CONFIGURAÇÃO DO TIMER*/
48  hw_timer_t *timer = NULL; //Variavel responsável por armazenar o timer
49  portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED; //Cria prioridade para a
    ↪  interrupção
50
51  /*CONFIGURAÇÃO DO TIMER DE CONTROLE DA SAÍDA*/
52  hw_timer_t *timerOutputCtrl = NULL; //Variavel responsável por armazenar o timer
53  portMUX_TYPE timerMuxOutputCtrl = portMUX_INITIALIZER_UNLOCKED; //Cria prioridade
    ↪  para a interrupção
54  volatile int countInterruptOutputCtrl; //trigger
55
56  /*************************************/
57  /************INTERRUPÇÃO**************/
```

```
58  /*************************************/
59  volatile int countInterrupt; //trigger
60
61  //Rotina de interrupção do timer
62  void IRAM_ATTR onTime(){
63    portENTER_CRITICAL_ISR(&timerMux);
64    countInterrupt++;
65    portEXIT_CRITICAL_ISR(&timerMux);
66  }
67
68
69  //Rotina de interrupção do timer responsavel pelo gerenciamento da ativacao da
    ↪  saída
70  void IRAM_ATTR onTimeOutputControl(){
71    portENTER_CRITICAL_ISR(&timerMuxOutputCtrl);
72    countInterruptOutputCtrl++;
73    portEXIT_CRITICAL_ISR(&timerMuxOutputCtrl);
74  }
75
76  int counter = 1;
77
78  /*Definicao dos pinos de controle do ShiftRegister*/
79  //Pino conectado ao pino 14(SI) do 74HC595
80  int dataPin = 25;
81  //Pino conectado ao pino 11(SCK) do 74HC595
82  int clockPin = 26;
83  //Pino conectado ao pino 12(RCK) do 74HC595
84  int latchPin = 27;
85
86  //Pino conectado ao pino 13(OE) do 74HC595 conectado ao resistor de pullup -
87  //Deve ser colocado para zero após ativação do ESP32
88  int outputEnablePin = 16;
89
90  /*Vetor para persistir válvulas na memória*/
91  valvula_t valvulas[NUM_VALVULAS];
92  //vetor para eventos de irrigacao
93  evento_irrigacao_t evento_buffer[NUM_MAXIMO_EVENTOS];
94  //vetor auxiliar para eventos de pressurizacao
95  evento_irrigacao_t eventoPressurValvulas[NUM_VALVULAS];
96  bool outputStates[NUM_VALVULAS]; //Indicador de saída inicializado com zero por
    ↪  default
97  int totalEventos = 0;
98
99  /*Variáveis de estado*/
100 STATES_LCD estadoAtual = TEMPO_EXIBE;
101 STATES_CONFIG_DATA_HORA estadoConfigDataHora = CONFIG_HORA;
102 uint8_t contaCircularValv = 1;
103 uint8_t contaConfig = 1;
```

```
104 uint8_t contaIntervalo = 1;
105 uint8_t multiplicadorRight = 1;
106 uint8_t multiplicadorLeft = 1;
107 uint8_t cadenciaDoMultiplicador = 0; //Variavel responsavel por segurar o avanco
    ↪  do multiplicador
108
109 int contadorEventos;
110
111 /*******************FLAGS************************/
112 bool flagBotaoLeft = false;
113 bool flagBotaoRight = false;
114 bool flagBotaoConfig = false;
115 bool flagAtuandoValve = false;
116 bool flagAlarmeProgramado = false;
117 bool flagIrrigNaoPossivel = false;
118 bool flagPressurization = false;
119 bool flagAtuandoPressur = false; //Flag para auxiliar na atuação da
    ↪  pressurizacao
120
121 /************************************************
122  ******************PROTOTIPOS********************
123  ************************************************/
124 /*Funcao para realizar a comparação entre dois eventos no sort*/
125 bool compareFunc(evento_irrigacao_t a, evento_irrigacao_t b){
126   if(a.horaProgramada == b.horaProgramada) {
127     return a.indexValvula < b.indexValvula;
128   }
129   return a.horaProgramada < b.horaProgramada;
130 }
131
132 void state_LCD_EXIBE(int index, valvula_t& valvula);
133 void state_TRANS_TO_CONFIG(uint8_t, uint8_t*);
134 void state_LCD_CONFIG(uint8_t);
135 void state_TRANS_TO_EXIBE(uint8_t, uint8_t*);
136 void state_TEMPO_EXIBE(RtcDateTime& now);
137 void state_TEMPO_CONFIG();
138 void state_CONFIRM_IRRIG();
139 void state_INIT_IRRIG(RtcDateTime&);
140 void state_IRRIGATION();
141 void state_INIT_PRESSUR();
142 void state_PRESSURIZATION(RtcDateTime&);
143
144 void acoesDosEstados();
145 void transicoesDeEstado();
146
147 //Funcoes auxiliares
148 void incrementaDecrementaParametro(valvula_t*, bool, uint8_t, uint8_t);
149 void gestaoParametrosValvulaConfig();
```

```
150 void printDateTime(const RtcDateTime& dt);
151
152 //Funcoes de controle do Shift Register
153 void sendToShiftRegister(uint32_t output);
154 bool trataEventoIrrigacao(evento_irrigacao_t* e, bool* flagAtuando);
155 bool clearOutputControl();
156
157 void setup() {
158   // set the 74HC595 Control pin as output
159   delay(1000);
160   pinMode(outputEnablePin, OUTPUT); //13(OE) do 74HC595
161   pinMode(latchPin, OUTPUT);     //11(SCK) do 74HC595
162   pinMode(clockPin, OUTPUT);     //14(SI) do 74HC5955
163   pinMode(dataPin, OUTPUT);      //12(RCK) do 74HC595
164   digitalWrite(dataPin,LOW);
165   digitalWrite(latchPin, LOW);
166   digitalWrite(clockPin, LOW);
167   delay(500);
168   sendToShiftRegister(0x00000000);
169   digitalWrite(outputEnablePin, HIGH); //Permite a ativação das saídas caso seja
    ↪  colocado em zero
170   //Desliga todas as saídas
171   sendToShiftRegister(0x00000000);
172
173   Serial.begin(57600);
174   delay(1000);
175
176   //Já inicia LCD e Preferences.
177   vUtil.begin();
178   Rtc.Begin();
179
180   btnLeft.setDebounceTime(DEBOUNCE_TIME);
181   btnRight.setDebounceTime(DEBOUNCE_TIME);
182   btnConfig.setDebounceTime(DEBOUNCE_TIME);
183
184   for(int i = 1; i<= NUM_VALVULAS; i++) {
185     if(vUtil.recuperaValvulaFlash(i, valvulas[i-1])){
186       Serial.printf("\nValores Recuperados da Flash %d: %d , %d, %d\n",i,
    ↪  valvulas[i-1].tempoIrrigacao, valvulas[i-1].numPulsos,
    ↪  valvulas[i-1].interPulsos);
187       delay(500);
188     } else {
189         Serial.printf("\nNao foi possivel recuperar valor da Flash para indice
    ↪  %d\n",i);
190       }
191   }
192
193   //Configuração do timer
```

```
194   timer = timerBegin(0, 80, true); //id do timer, prescaler, flag - rising or
      ↪  falling edge
195   timerAttachInterrupt(timer, &onTime, true);
196   //Configura o alarme para acionar a cada meio segundo
197   timerAlarmWrite(timer, 500000, true);
198   timerAlarmEnable(timer);
199
200   //Configuração do timer de controle da saida
201   timerOutputCtrl = timerBegin(1, 80, true); //id do timer, prescaler, flag -
      ↪  rising or falling edge
202   timerAttachInterrupt(timerOutputCtrl, &onTimeOutputControl, true);
203   //Configura o alarme para acionar a cada 200ms
204   timerAlarmWrite(timerOutputCtrl, 200000, true);
205   //timerAlarmEnable(timer1);
206
207   /*Para imprimir o tempo de compilação*/
208   RtcDateTime compiled = RtcDateTime(__DATE__, __TIME__);
209   compiled+=15;
210   RtcDateTime initialConfig = Rtc.GetDateTime();
211   if (initialConfig < compiled)
212   {
213       Rtc.SetDateTime(compiled);
214   }
215   //Para garantir que o DS3231 está corretamente configurado
216   Rtc.Enable32kHzPin(false);
217   Rtc.SetSquareWavePin(DS3231SquareWavePin_ModeNone);
218
219 }
220
221 void loop() {
222   /*Atualiza estado dos botões*/
223   botoes();
224   /*Controla a transição dos estados*/
225   transicoesDeEstado();
226
227   /*Entra periodicamente, a depender do timer, para realizar as ações dos
      ↪  estados*/
228   if(countInterrupt > 0) {
229     portENTER_CRITICAL(&timerMux);
230     countInterrupt--;
231     portEXIT_CRITICAL(&timerMux);
232
233     acoesDosEstados();
234   }
235
236   if(countInterruptOutputCtrl > 0) {
237     portENTER_CRITICAL(&timerMux);
238     countInterruptOutputCtrl--;
```

```
239       portEXIT_CRITICAL(&timerMux);
240
241       clearOutputControl(&flagAtuandoValve);
242     }
243
244   }
245
246   /*Funcao que verifica o estado dos botões e levanta uma flag quando os mesmos
      ↪  foram acionados*/
247   void botoes() {
248     btnLeft.loop();
249     btnRight.loop();
250     btnConfig.loop();
251
252     if(btnLeft.isPressed()){
253       flagBotaoLeft = true;
254     }
255
256     if(btnRight.isPressed()){
257       flagBotaoRight = true;
258     }
259
260     if(btnConfig.isPressed()){
261       flagBotaoConfig = true;
262     }
263   }
264
265   /* Responsável pelas transições de estado, verifica se houve mudança nos botões
      ↪  e
266   * faz as trocas de estados necessárias.
267   */
268   void transicoesDeEstado(){
269     switch(estadoAtual){
270
271       case LCD_EXIBE:
272         if (flagBotaoRight) {
273           flagBotaoRight = false;
274           contaCircularValv++;
275           if (contaCircularValv > NUM_VALVULAS) {
276             contaCircularValv = 1;
277             estadoAtual = CONFIRM_IRRIG;
278           }
279         }
280         if (flagBotaoLeft) {
281           flagBotaoLeft = false;
282           contaCircularValv--;
283           if (contaCircularValv <= 0) {
284             contaCircularValv = NUM_VALVULAS;
```

```
285             estadoAtual = INIT_PRESSUR;
286           }
287         }
288         if (flagBotaoConfig) {
289           flagBotaoConfig = false;
290           estadoAtual = TRANS_TO_CONFIG;
291         }
292       break;
293
294       case TRANS_TO_CONFIG:
295         //Estado responsável pela transição de um estado para o outro
296         if(contaIntervalo >= 4){
297           estadoAtual = LCD_CONFIG;
298           contaIntervalo = 1;
299           contaConfig = 1;
300         }
301         else{
302           estadoAtual = TRANS_TO_CONFIG;
303         }
304       break;
305
306       case LCD_CONFIG:
307
308         gestaoParametrosValvulaConfig();
309
310         if (flagBotaoConfig) {
311           flagBotaoConfig = false;
312           estadoAtual = LCD_CONFIG;
313           contaConfig++;
314           lcd.clear();
315           if(contaConfig > 3) {
316             contaConfig = 1;
317             estadoAtual = TRANS_TO_EXIBE;
318           }
319         }
320
321       break;
322
323       case TRANS_TO_EXIBE:
324         //Estado responsável pela transição de um estado para o outro
325         if(contaIntervalo >= 4){ //Transicao de 4*500ms = 2s
326           estadoAtual = LCD_EXIBE;
327           contaIntervalo = 1;
328         }
329         else{
330           estadoAtual = TRANS_TO_EXIBE;
331         }
332       break;
```

```
333
334     case TEMPO_EXIBE:
335       if (flagBotaoRight) {
336         flagBotaoRight = false;
337         estadoAtual = INIT_PRESSUR;
338       }
339       if (flagBotaoLeft) {
340         flagBotaoLeft = false;
341         estadoAtual = LCD_EXIBE;
342         contaCircularValv = NUM_VALVULAS;
343       }
344       if (flagBotaoConfig) {
345         flagBotaoConfig = false;
346         estadoAtual = TEMPO_EXIBE;//Deve alterar quando for implementada a rotina
    ↪  de configuracao de tempo
347       }
348     break;
349
350     case TEMPO_CONFIG:
351     break;
352
353     case CONFIRM_IRRIG:
354       if (flagBotaoRight) {
355         flagBotaoRight = false;
356         estadoAtual = TEMPO_EXIBE;
357       }
358       if (flagBotaoLeft) {
359         flagBotaoLeft = false;
360         estadoAtual = LCD_EXIBE;
361         contaCircularValv = NUM_VALVULAS;
362       }
363       if (flagBotaoConfig) {
364         flagBotaoConfig = false;
365         estadoAtual = INIT_IRRIG;
366       }
367     break;
368
369     case INIT_IRRIG:
370       //Estado responsavel por agendar os eventos de irrigacao, se for possivel
    ↪  agendar,
371       //Apos 5 segundos avança para o estado de irrigacao, se nao for exibe o
    ↪  "Irrigacao nao possivel" e
372       //Volta para o estado de exibicao de tempo
373       if(contaIntervalo >= 10){ //Transicao de 10*500ms = 5s
374         if(flagIrrigNaoPossivel || !flagAlarmeProgramado){
375           contaIntervalo = 1;
376           estadoAtual = TEMPO_EXIBE;
377         }
```

```
        else {
          contaIntervalo = 1;
          estadoAtual = IRRIGATION;
        }
      }
    break;

    case IRRIGATION:
      if(contadorEventos >= totalEventos){
        contadorEventos = 0;
        totalEventos = 0;
        flagAlarmeProgramado = false; //Finaliza a irrigacao
        estadoAtual = TEMPO_EXIBE;

      }
    break;

    case INIT_PRESSUR:
      if(flagPressurization){
        estadoAtual = PRESSURIZATION;
        break;
      }
      if (flagBotaoRight) {
        flagBotaoRight = false;
        estadoAtual = LCD_EXIBE;
        contaCircularValv = 1;
      }
      if (flagBotaoLeft) {
        flagBotaoLeft = false;
        estadoAtual = TEMPO_EXIBE;
      }
      if (flagBotaoConfig) {
        flagBotaoConfig = false;
        estadoAtual = PRESSURIZATION;
      }
    break;

    case PRESSURIZATION:
      if (flagBotaoRight) {
        flagBotaoRight = false;
        estadoAtual = LCD_EXIBE;
        contaCircularValv = 1;
      }
      if (flagBotaoLeft) {
        flagBotaoLeft = false;
        estadoAtual = TEMPO_EXIBE;
      }
      if (flagBotaoConfig) {
```

```
426           flagBotaoConfig = false;
427           estadoAtual = STOP_PRESSUR;
428         }
429       break;
430
431       case STOP_PRESSUR:
432         if(flagPressurization) {
433           break;
434         }
435         if (flagBotaoRight) {
436           flagBotaoRight = false;
437           estadoAtual = LCD_EXIBE;
438           contaCircularValv = 1;
439         }
440         if (flagBotaoLeft) {
441           flagBotaoLeft = false;
442           estadoAtual = TEMPO_EXIBE;
443         }
444         if (flagBotaoConfig) {
445           flagBotaoConfig = false;
446           estadoAtual = INIT_PRESSUR;
447         }
448       break;
449
450     }
451   }
452
453   /* Responsável pelas ações e exibicoes de cada estado
454   */
455   void acoesDosEstados(){
456     switch(estadoAtual){
457       case LCD_EXIBE:
458         state_LCD_EXIBE(contaCircularValv, valvulas[contaCircularValv-1]);
459       break;
460       case TRANS_TO_CONFIG:
461         state_TRANS_TO_CONFIG(contaCircularValv, &contaIntervalo);
462       break;
463       case LCD_CONFIG:
464         state_LCD_CONFIG(contaConfig);
465       break;
466       case TRANS_TO_EXIBE:
467         state_TRANS_TO_EXIBE(contaCircularValv, &contaIntervalo);
468       break;
469       case TEMPO_EXIBE:
470       {
471         RtcDateTime actual = Rtc.GetDateTime();
472         state_TEMPO_EXIBE(actual);
473         break;
```

```
474       }
475
476       case TEMPO_CONFIG:
477       break;
478
479       case CONFIRM_IRRIG:
480         state_CONFIRM_IRRIG();
481       break;
482
483       case INIT_IRRIG:
484       {
485         RtcDateTime actual = Rtc.GetDateTime();
486         state_INIT_IRRIG(actual);
487         contaIntervalo++;
488         break;
489       }
490
491       case IRRIGATION:
492         state_IRRIGATION();
493       break;
494
495       case INIT_PRESSUR:
496         state_INIT_PRESSUR();
497       break;
498
499       case PRESSURIZATION:
500       {
501         RtcDateTime actual = Rtc.GetDateTime();
502         state_PRESSURIZATION(actual);
503         break;
504       }
505
506       case STOP_PRESSUR:
507       {
508         if(flagPressurization){
509           flagPressurization = false;
510           RtcDateTime pressStop = Rtc.GetDateTime();
511           //Cria o agendamento de despressurizacao das valvulas
512           for(int i = 0; i < NUM_VALVULAS; i++){
513             eventoPressurValvulas[i].acao = false;
514             eventoPressurValvulas[i].foiAtendido = false;
515             eventoPressurValvulas[i].indexValvula = i+1;
516             eventoPressurValvulas[i].horaProgramada = pressStop;
517           }
518           contadorEventos = 0;
519           totalEventos = NUM_VALVULAS;
520           flagAtuandoPressur = true;
521         }
```

```
522
523       break;
524       }
525     }
526
527     //Rotina para tratamento dos eventos de pressurizacao ou despressurizacao
    ↪  iniciais
528     if (flagAtuandoPressur){
529       if(contadorEventos < totalEventos){
530         RtcDateTime actual = Rtc.GetDateTime();
531         if(actual > eventoPressurValvulas[contadorEventos].horaProgramada){
532           if(trataEventoIrrigacao(&eventoPressurValvulas[contadorEventos],
    ↪  &flagAtuandoValve)){
533             contadorEventos++; //Se o evento ocorreu com sucesso, então pode-se
    ↪  incrementar o contador
534           }
535         }
536       }
537       else{
538         flagAtuandoPressur = false; //Terminou a atuação da Pressurização
539         contadorEventos = 0;
540         totalEventos = 0;
541       }
542     }
543
544   }
545
546   /*************************************************************************/
547   /*****************IMPLEMENTACOES DE CADA ESTADO************************/
548   /*************************************************************************/
549
550   void state_LCD_EXIBE(int index, valvula_t& valvula){
551     vUtil.telaValvula(index, valvula, false);
552   }
553
554   void state_TRANS_TO_CONFIG(uint8_t index, uint8_t* conta){
555     if(*conta == 1){
556       lcd.clear();
557     }
558     *conta += 1; //Incrementa o valor apontado pelo ponteiro
559     vUtil.telaTransicaoConfig(index);
560   }
561
562   void state_LCD_CONFIG(uint8_t conta){
563     switch(conta){
564       case 1: //Configura tempo total de irrigação
565         vUtil.telaConfigTempIrrig(contaCircularValv,
    ↪  valvulas[contaCircularValv-1].tempoIrrigacao);
```

```
566         break;
567       case 2: //Configura número de pulsos
568         vUtil.telaConfigNumPulsos(contaCircularValv,
    ↪  valvulas[contaCircularValv-1].numPulsos);
569         break;
570       case 3: //Configura intervalo entre os pulsos
571         vUtil.telaConfigInterPulsos(contaCircularValv,
    ↪  valvulas[contaCircularValv-1].interPulsos);
572         break;
573     }
574   }
575   void state_TRANS_TO_EXIBE(uint8_t index, uint8_t* conta){
576     if(*conta == 1){
577       lcd.clear();
578       vUtil.salvaValvulaFlash(index, valvulas[index-1]);
579     }
580     *conta += 1; //Incrementa o valor apontado pelo ponteiro
581     vUtil.telaTransicaoExib(index);
582   }
583   void state_TEMPO_EXIBE(RtcDateTime& now){
584      vUtil.telaDataHora(now.Day(), now.Month(), now.Year(), now.Hour(),
    ↪  now.Minute(), now.Second());
585   }
586   void state_TEMPO_CONFIG(){}
587
588   //Aguarda o operador confirmar a operação
589   //Pergunta o tempo de pressurização
590   //Cria vetor de eventos a partir das válvulas configuradas
591   void state_CONFIRM_IRRIG(){
592     vUtil.telaConfirmaIrrig();
593   }
594   void state_INIT_IRRIG(RtcDateTime& now){
595
596     if(!flagIrrigNaoPossivel){
597       vUtil.telaInitIrrig();
598     }
599     else{
600       vUtil.telaIrrigNaoPossivel();
601     }
602
603     if(!flagAlarmeProgramado){
604       //Antes da execucao da rotina de configuracao dos eventos é necessário
    ↪  verificar se o numero total de eventos
605       //é menor que o máximo alocado.
606       totalEventos = vUtil.numeroTotalEventos(valvulas);
607       if(totalEventos >= NUM_MAXIMO_EVENTOS){
608         Serial.printf("Numero de eventos (%d) maior que o numero maximo permitido
    ↪  (%d)\n", totalEventos, NUM_MAXIMO_EVENTOS);
```

```
609         flagAlarmeProgramado = false;
610         flagIrrigNaoPossivel = true;
611         return;
612       }
613 
614       int posicaoDaAgenda=0;
615       for(int i = 0; i < NUM_VALVULAS; i++){ //Forma de colocar as válvulas no
    ↪  mesmo vetor de uma vez
616          if(i == 0){
617             posicaoDaAgenda += 0;
618          }else{
619             posicaoDaAgenda += (valvulas[i-1].numPulsos)*2;
620          }
621          vUtil.agendamentoAPartirDaValvula(i+1, valvulas[i], now + IRRIG_OFFSET,
    ↪  evento_buffer + posicaoDaAgenda); //Agenda para 10 segundos depois
622       }
623 
624       Serial.printf("BEFORE SORT\n");
625       for(int i = 0; i < totalEventos; i++){
626         Serial.printf("Evento: %d\n", i);
627         Serial.print("Hora do Alarme:");
    ↪  printDateTime(RtcDateTime(evento_buffer[i].horaProgramada));
628         Serial.printf("\nValvula: %d\n", evento_buffer[i].indexValvula);
629         evento_buffer[i].acao? Serial.printf("Acao: ligar\n"): Serial.printf("Acao:
    ↪  desligar\n");
630         Serial.print("Foi atendido:"); evento_buffer[i].foiAtendido?
    ↪  Serial.printf("sim\n"): Serial.printf("nao\n");
631       }
632       //Organiza o vetor agenda execucao dos eventos na ordem
633       std::sort(evento_buffer, evento_buffer + totalEventos, compareFunc);
634       flagAlarmeProgramado = true;
635       contadorEventos = 0; //Inicia do zero o contador de eventos
636 
637       Serial.printf("\n\nAFTER SORT\n");
638       for(int i = 0; i < totalEventos; i++){
639         Serial.printf("Evento: %d\n", i);
640         Serial.print("Hora do Alarme:");
    ↪  printDateTime(RtcDateTime(evento_buffer[i].horaProgramada));
641         Serial.printf("\nValvula: %d\n", evento_buffer[i].indexValvula);
642         evento_buffer[i].acao? Serial.printf("Acao: ligar\n"): Serial.printf("Acao:
    ↪  desligar\n");
643         Serial.print("Foi atendido:"); evento_buffer[i].foiAtendido?
    ↪  Serial.printf("sim\n"): Serial.printf("nao\n");
644       }
645     }
646   }
647   void state_IRRIGATION(){
648     if(contadorEventos < totalEventos){
```

```
649       vUtil.telaIrrigacao(contadorEventos, totalEventos);
650       RtcDateTime actual = Rtc.GetDateTime();
651       if(actual > evento_buffer[contadorEventos].horaProgramada){
652         if(trataEventoIrrigacao(&evento_buffer[contadorEventos],
      ↪  &flagAtuandoValve)){
653           contadorEventos++; //Se o evento ocorreu com sucesso, então pode-se
      ↪  incrementar o contador
654         }
655       }
656     }
657   }
658
659   /**
660    * Funcao responsavel pelas acoes do estado antes de iniciar a pressurizacao
661    */
662   void state_INIT_PRESSUR(){
663     vUtil.telaInicioPressurizacao();
664   }
665
666   /**
667    * Funcao responsavel pela exibicao do tempo da pressurizacao propriamente
668    */
669   void state_PRESSURIZATION(RtcDateTime& actual){
670     static RtcDateTime pressStart;
671     if(!flagPressurization){
672       flagPressurization = true;
673       pressStart = Rtc.GetDateTime();
674       //Cria o agendamento de pressurizacao das valvulas
675       for(int i = 0; i < NUM_VALVULAS; i++){
676         eventoPressurValvulas[i].acao = true;
677         eventoPressurValvulas[i].foiAtendido = false;
678         eventoPressurValvulas[i].indexValvula = i+1;
679         eventoPressurValvulas[i].horaProgramada = pressStart;
680       }
681       contadorEventos = 0;
682       totalEventos = NUM_VALVULAS;
683       flagAtuandoPressur = true;
684       //Deve enviar comandos para ligar as valvulas para a pressurizacao
685     }
686     RtcDateTime pressurTime = actual - pressStart;
687     vUtil.telaPressurizacao(pressurTime.Hour(), pressurTime.Minute(),
      ↪  pressurTime.Second());
688
689   }
690
691   /**************FUNÇÕES AUXILIARES*******************/
692
693   /** Incrementa(up == true) ou decrementa (up == false) o parâmetro da válvula em
      ↪  configuração atual
```

```
694   */
695  void incrementaDecrementaParametro(valvula_t* valvula, bool up, uint8_t
     ↪  multiplicador, uint8_t estadoConfig){
696      //Se up==true soma, caso contrário subtrai
697      int operador=0;
698      operador = up?multiplicador:(-1)*multiplicador;
699      switch(estadoConfig){
700        //Configura a válvula atual aumentando (ou diminuindo) o valor do parâmetro
     ↪  selecionado
701        case 1: //Configura tempo total de irrigação
702          valvula->tempoIrrigacao += operador;
703          break;
704        case 2: //Configura número de pulsos
705          valvula->numPulsos += operador;
706          break;
707        case 3: //Configura intervalo entre os pulsos
708          valvula->interPulsos += operador;
709          break;
710      }
711  }
712
713  /** Função auxiliar responsável por gerenciar o incremento ou decremento dos
     ↪  parametros de configuração das válvulas a partir
714   *  da gestão do tempo de pressão dos botões
715   */
716  void gestaoParametrosValvulaConfig (){
717
718    if (flagBotaoRight) {
719      if(btnRight.getState() != LOW){ //o botao foi solto
720        flagBotaoRight = false;
721        cadenciaDoMultiplicador = 0;
722        multiplicadorRight = 1;
723      }
724      else if(countInterrupt>0) {
725        incrementaDecrementaParametro(&valvulas[contaCircularValv-1], true,
     ↪  multiplicadorRight, contaConfig); //incrementa o parametro
726
727        cadenciaDoMultiplicador++;
728        if(cadenciaDoMultiplicador % multiplicadorRight == 0) {
729          cadenciaDoMultiplicador = 0;
730          multiplicadorRight++;
731        }
732        if(multiplicadorRight > 20){
733          multiplicadorRight = 20;
734        }
735      }
736    }
737    if (flagBotaoLeft) {
```

```
738       if(btnLeft.getState() != LOW){ //o botao foi solto
739         flagBotaoLeft = false;
740         cadenciaDoMultiplicador = 0;
741         multiplicadorLeft = 1;
742       }
743       else if(countInterrupt>0) {
744         incrementaDecrementaParametro(&valvulas[contaCircularValv-1], false,
      ↪  multiplicadorLeft, contaConfig); //decrementa o parametro
745         cadenciaDoMultiplicador++;
746         if(cadenciaDoMultiplicador%multiplicadorLeft == 0) {
747           cadenciaDoMultiplicador = 0;
748           multiplicadorLeft++;
749         }
750         if(multiplicadorLeft > 20){
751           multiplicadorLeft = 20;
752         }
753       }
754     }
755   }
756
757   /* Imprime data e hora na porta serial*/
758   void printDateTime(const RtcDateTime& dt)
759   {
760       char datestring[20];
761
762       snprintf_P(datestring,
763                countof(datestring),
764                PSTR("%02u/%02u/%04u %02u:%02u:%02u"),
765                dt.Month(),
766                dt.Day(),
767                dt.Year(),
768                dt.Hour(),
769                dt.Minute(),
770                dt.Second() );
771       Serial.print(datestring);
772   }
773
774
775   /*************************************************************/
776   /***************FUNÇÕES DE ATIVAÇÃO DAS SAÍDAS ****************/
777   /*************************************************************/
778   /* Funcao responsavel por enviar para a saída
779    *
780   */
781   void sendToShiftRegister(uint32_t output){
782     uint8_t outputByte;
783     digitalWrite(latchPin, LOW);
784     digitalWrite(clockPin, LOW);
```

```
785   //carrega o dado
786   for (int i = 0; i < REGISTER_NUMBER; i++){
787     outputByte = (uint8_t)(output>>(i*8));
788     Serial.printf("outputByte: %x , indice: %d\n", outputByte, i);
789     shiftOut(dataPin, clockPin, LSBFIRST, outputByte);
790   }
791   //libera pra o registrador de saida
792   digitalWrite(latchPin, HIGH);
793   delayMicroseconds(500); //Pausa por 0,5 ms para enviar o dado
794   digitalWrite(latchPin, LOW);
795 }
796
797 /*  Funcao responsável por gerenciar a ativação de uma válvula em um determinado
    ↪  evento
798  *  A função deve verificar a ação, se é de desligar ou de ligar
799  *  Enviar o comando especificado para a válvula especificada
800  *  Gerenciar para que o comando seja extinto após 200ms
801  *  Levanta flag de ocupado enquanto a ação completa não for atendida.
802  */
803
804 bool trataEventoIrrigacao(evento_irrigacao_t* e, bool* flagAtuando){
805   if(!(*flagAtuando)){ //Se nao estiver atuando em nenhuma valvula
806     uint32_t aux = e->acao? 0x80000000: 0x40000000; //O nibble mais significativo
    ↪  fica: 0b1000(8H) para a ativação ou 0b0100(4H) para a desativacao
807     aux = (aux >> ((e->indexValvula - 1)*2)); //Cada 2 bits ativa uma válvula,
    ↪  começando pelo MSB
808     aux += outputIndicationMask(e->acao, e->indexValvula); //define a nova acao e
    ↪  acrescenta na mascara de saida
809     sendToShiftRegister(aux);
810     *flagAtuando = true;
811     timerWrite(timerOutputCtrl, 0);
812     timerAlarmEnable(timerOutputCtrl);
813     return true;
814   }
815   return false;
816 }
817
818 /* Limpa a saída e desliga o timer*/
819 bool clearOutputControl(bool* flagAtuando){
820   if(*flagAtuando){
821     uint32_t aux = 0x00000000;
822     aux += outputIndicationMask(false, 0);
823     *flagAtuando = false;
824     timerAlarmDisable(timerOutputCtrl);
825     timerWrite(timerOutputCtrl, 0);
826     sendToShiftRegister(aux);
827   }
828 }
```

```
829
830 /**
831  * @brief Cria a máscara da ativação dos LEDs da saída. Recebe a válvula e o novo
    ↪  estado da mesma.
832  * Retorna uma máscara aditiva que deve ser somada a saída de 32 bits que será
    ↪  enviada ao shift register.
833  * Caso seja fora do escopo de índices das válvulas retorna o estado atual da
    ↪  saída como máscara sem
834  * alterar nenhum estado.
835  *
836  * @param action
837  * @param index
838  * @return uint32_t máscara aditiva calculada
839  */
840 uint32_t outputIndicationMask(bool action, uint8_t index){
841     if (index >= 1 && index <= NUM_VALVULAS){
842       outputStates[index - 1] = action; //Atualiza a indicação de saída para o
    ↪  índice.
843     }
844     uint32_t mask = 0;
845     for(uint8_t i = 0; i < NUM_VALVULAS; i++){
846         if(outputStates[i] == true){
847             uint32_t one = 1;
848             mask += one << (NUM_VALVULAS -1 - i);
849         }
850     }
851     return mask;
852 }
```