
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE - UFCG
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA - CEEI
UNIDADE ACADÊMICA DE ENGENHARIA ELÉTRICA - UAEE

MARLEY LOBÃO DE SOUSA

# RELATÓRIO DE ESTÁGIO SUPERVISIONADO

## NÚCLEO VIRTUS - UFCG

**Campina Grande, PB**
**11 de março de 2022**

MARLEY LOBÃO DE SOUSA

# RELATÓRIO DE ESTÁGIO SUPERVISIONADO
# NÚCLEO VIRTUS - UFCG

Relatório de Estágio Supervisionado submetido à Universidade Federal de Campina Grande, como parte dos requisitos necessários para obtenção do grau de Bacharel em ciências no domínio da Engenharia Elétrica.

Professor Orientador: Danilo Freire de Souza Santos D.Sc
Professor Supervisor: Gutemberg Gonçalves dos Santos Júnior D.Sc

**Campina Grande, PB**
**11 de março de 2022**

MARLEY LOBÃO DE SOUSA

**RELATÓRIO DE ESTÁGIO SUPERVISIONADO**
**NÚCLEO VIRTUS - UFCG**

Relatório de Estágio Supervisionado submetido à Universidade Federal de Campina Grande, como parte dos requisitos necessários para obtenção do grau de Bacharel em ciências no domínio da Engenharia Elétrica.

Aprovado em: ____/____/____

---

Danilo Freire de Souza Santos D.Sc
Professor Orientador

---

Jaidilson Jó da Silva D.Sc
Professor Avaliador

**Campina Grande, PB**
**11 de março de 2022**

# AGRADECIMENTOS

Primeiramente agradeço a Deus, que em toda sua plenitude, me criou com inteligência e capacidade suficiente para estudar, compreender e admirar a sua bela natureza e seus processos físicos.

Deixo meus agradecimentos para meus pais, Marco e Raquel, minhas irmãs, Mirley e Maísa, e minhas tias, Cássia e Dalva, por todo carinho, incentivo e colaboração durante meu processo de crescimento pessoal e profissional durante minha graduação.

Também agradeço à minha namorada, Geovanna, por me fazer perceber meu próprio potencial, muito mais do que eu mesmo pude notar e sempre me fazer acreditar que posso alcançar o lugar desejado em minha área de atuação.

Deixo meus agradecimentos também a Lyang Medeiros, Murilo Santos, Breno Alencar, Samuel Cesarino, Matheus Morais, Mateus Pereira e Andresso da Silva, pois apesar de nosso período de formação não ter sido nada fácil, vocês fizeram com que a experiência fosse, sem dúvida, leve, feliz e diferenciada.

Agradeço ao professor Danilo Freire, pela sua orientação e dedicação em retirar todas minhas dúvidas em relação aos processos de estágio, bem como, pelos conhecimentos adquiridos ao longo das disciplinas e que foram aplicados durante o estágio a que se refere este relatório.

Deixo meus agradecimentos também, ao professor Gutemberg Júnior, por todo o apoio e supervisão fornecido desde a etapa de adaptação, até a compreensão e acompanhamento das atividades realizadas durante o estágio.

Por fim, agradeço aos profissionais que compõem o núcleo de pesquisa VIRTUS, que me direcionaram nessa etapa de aplicação do conhecimento que foi visto durante a graduação e todo aprendizado acerca do ambiente de trabalho que pertencem e da qual tive a experiência de participar. Em especial, a Gabriel Miranda, por nossa amizade e irmandade, assim como, todo auxílio durante o desenvolvimento dos desafios enfrentados no estágio, juntamente com as dificuldades que a vida acadêmica nos proporciona simultaneamente.

# Lista de Figuras

# Lista de Tabelas

# Lista de Algoritmos

# Lista de Símbolos e Abreviaturas

**UFCG** - Universidade Federal de Campina Grande

**CEEI** - Centro de Engenharia Elétrica e Informática

**DEE** - Departamento de Engenharia Elétrica

**Embedded** - Laboratório de Sistemas Embarcados e Computação Pervasiva

**PDI** - Pesquisa, Desenvolvimento e Inovação

**RISC** - *Reduced Instruction Set Computer*

**ARM** - *Advanced RISC Machine*

**CPU** - *Central Process Unit*

**GPIO** - *General Purpose Input/Output*

**RTOS** - *Real-Time Operating System*

**PLL** - *Phase Locked Loop*

**UART** - *Universal Asynchronous Receiver/Transmitter*

**Tx** - Pino Transmissor do UART

**Rx** - Pino Receptor do UART

**IDE** - *Integrated Development Environment*

**CCS** - *Code Composer Studio™*

# Sumário

# 1 INTRODUÇÃO

Para que o curso de Engenharia Elétrica da Universidade Federal de Campina Grande (UFCG) seja concluído, é necessário que o graduando realize um estágio obrigatório que pode ser na modalidade integrado ou supervisionado. A modalidade supervisionado difere apenas pela variedade de carga horária a ser cumprida. O estágio tem o intuito de que o aluno crie ligações entre a teoria que foi aprendida em sala de aula e a aplicação do conhecimento nas empresas, por meio da utilização dos mesmos em seus produtos que vão para o mercado.

Sendo assim, neste relatório são descritas as atividades que foram desenvolvidas pelo estudante de graduação em Engenharia Elétrica, Marley Lobão de Sousa, durante o estágio supervisionado que foi realizado no Núcleo VIRTUS - UFCG. O estágio foi cumprido no período compreendido entre 16 de novembro de 2021 e 25 de fevereiro de 2022, com carga horária de 20 horas semanais, totalizando 291 horas, sob a orientação do professor Danilo Freire de Souza Santos e a supervisão do professor Gutemberg Gonçalves dos Santos Júnior.

Devido as dificuldades relacionadas a pandemia de COVID-19, o estágio foi efetuado de forma remota, em cooperação com o Laboratório de Sistemas Embarcados e Computação Pervasiva (Embedded) da UFCG que forneceu os recursos necessários para desenvolvimento do projeto como um todo.

Este estágio fez parte da participação do aluno em um projeto de Pesquisa, Desenvolvimento e Inovação do Núcleo VIRTUS. As atividades realizadas foram relacionadas à desenvolvimento e implementação de *software* para realizar o controle e monitoramento de equipamentos e dispositivos por meio de sistemas embarcados. Para isso, foi realizado um treinamento de desenvolvimento ágil e versionamento de código para profissionalização e padronização do trabalho em equipe, e um estudo para melhor compreensão do microcontrolador a ser usado, bem como, da arquitetura do sistema a ser programado.

Os capítulos que se seguem possuem a seguinte organização e objetivo: o capítulo 2 corresponde à apresentação do laboratório em que foram alocadas as atividades e do projeto do estágio; no capítulo 3, são apresentados e discutidos os conceitos essenciais referentes as atividades desenvolvidas; no capítulo 4, são descritas as atividades desenvolvidas pelo aluno; e finalmente, no capítulo 5, serão tecidas as considerações finais a respeito das atividades; e no capítulo 6, constam as referências utilizadas.

# 2 APRESENTAÇÃO DO AMBIENTE DE TRABALHO E DO PROJETO

Nesta capítulo serão apresentados o Núcleo VIRTUS, o laboratório Embedded e a estrutura organizacional do projeto de pesquisa, desenvolvimento e inovação em que o estágio foi realizado.

## 2.1 NÚCLEO VIRTUS

O VIRTUS é o Núcleo de Pesquisa, Desenvolvimento e Inovação em Tecnologia da Informação, Comunicação e Automação, cujo prédio onde se localiza é mostrado na Figura 1 – um órgão suplementar da UFCG vinculado ao Centro de Engenharia Elétrica e Informática (CEEI). Tem por objetivo, criar novas opções de futuro por meio de projetos com empresas parceiras da indústria, nas mais diversas áreas de tecnologia da informação, comunicação e automação (VIRTUS, 2021).

Figura 1: Fotografia do Núcleo VIRTUS


Fonte: VIRTUS (2021)

Muitos profissionais do Departamento de Engenharia Elétrica (DEE) desenvolvem pesquisa neste núcleo, assim como alunos de diversas universidades, principalmente nas áreas de microeletrônica, sistemas embarcados e inteligência artificial.

## 2.2 LABORATÓRIO EMBEDDED

O Laboratório de Sistemas Embarcados e Computação Pervasiva (Embedded) é um dos laboratórios do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, Paraíba, Brasil. Na Figura 2 pode ser visto o edifício Gurdip Singh Deep, onde se localiza o laboratório.

Figura 2: Fotografia do Laboratório Embedded


Fonte: LIEC (2020)

No laboratório Embedded trabalham professores e pesquisadores do CEEI que desenvolvem projetos com alunos de nível de graduação e pós graduação, principalmente nas áreas de controle, sistemas embarcados e computação (Embedded Lab, 2019). Como já dito, as atividades foram realizadas com a cooperação deste laboratório, visto que, caso o estágio se caracterizasse como presencial, as instalações do mesmo seriam utilizadas.

## 2.3 PROJETO DE PESQUISA, DESENVOLVIMENTO E INOVAÇÃO (PDI)

O estagiário Marley Lobão de Sousa realizou as atividades em um projeto de cooperação técnico e científico entre o laboratório e uma empresa parceira do Núcleo VIRTUS. Com o intuito de que o aluno desenvolva habilidades profissionais como trabalho em equipe, planejamento de atividades, produção de relatórios técnicos, bem como aplique o conhecimento adquirido durante o curso de Engenharia Elétrica.

O objetivo do projeto de PDI executado no contexto desse estágio foi a realização de investigações, projeto e desenvolvimento de *software* para o controle e monitoramento de equipamentos e dispositivos remotos por meio de sistemas embarcados e protocolos para Internet das Coisas.

Neste capítulo foram apresentados o núcleo concedente e o laboratório em que foi realizado o estágio, assim como o projeto em que o aluno estava inserido. No próximo capítulo, são apresentados os conceitos e definições fundamentais para o desenvolvimento das atividades realizadas durante o estágio.

# 3 FUNDAMENTAÇÃO TEÓRICA

Para melhor compreensão das atividades realizadas no estágio e assim proporcionar o devido embasamento teórico, este capítulo apresenta os conteúdos relacionados as tecnologias, bem como, as ferramentas utilizadas, no desenvolvimento de *software* e de *firmware*, visando melhorar as capacidades técnicas adquiridas durante a graduação e que foram objetivo de aplicação na área de sistemas embarcados.

## 3.1 DESENVOLVIMENTO ÁGIL DE SISTEMAS

Nos dias atuais, ao se referir a gestão e inovação, as metodologias ágeis estão no topo da preferência dos gestores. Pois as mesmas tem proporcionado às equipes, uma melhor integração do trabalho, bem como uma visão geral do que foi feito, do que está sendo feito e das partes do trabalho que ainda são necessárias fazer para alcançar o objetivo final.

Por isso, os modelos ágeis são ferramentas bastante úteis para garantir uma maior chance de que o projeto não se perca de seus objetivos iniciais e de que os requisitos sejam atendidos com sucesso. O desenvolvimento de *software*, é apenas uma das áreas que tem se aproveitado muito de todas essas vantagens para melhorar seus processos de gestão e conquistar melhores resultados.

De acordo com Sutherland e Schwaber (2020), os criadores do Guia do Scrum (que também são os criadores desta metodologia, juntamente com Mike Beedle), é um *framework* leve que ajuda pessoas, times e organizações a gerar valor por meio de soluções adaptativas para problemas complexos.

Para isso, o time de desenvolvedores deve ser pequeno para ser ágil e grande para entregar um dado trabalho em um período de tempo definido pelos integrantes. Esse time, também denominado de *Scrum Team*, se divide basicamente em 3 partes, com responsabilidades diferentes, são eles: a equipe técnica, o *Product Owner* e o *Scrum Master* (Sutherland e Schwaber, 2020).

Nesse sentido, a equipe técnica é responsável pela criação dos planos a serem seguidos durante a *Sprint* (período de tempo definido para entrega de metas de trabalho), assim como adicionar características ao produto que permitam dar-lhe definições de conclusão, adaptar todos os dias o plano em relação ao que aponta a *Sprint*, bem como responsabilizar-se diante das decisões tomadas durante o processo.

Já o *Product Owner*, é basicamente o representante do cliente, ou seja, deve sempre indicar qual a meta do produto. Além disso, é responsável pelo gerenciamento eficaz do *Product Backlog* (basicamente a lista de tarefas a serem cumpridas pelos colaboradores), como criar seus itens e garantir que sejam claros.

E finalmente o *Scrum Master*, que é responsável por manter a cultura do Scrum conforme sua definição, com o objetivo de auxiliar os colaboradores a entender a metologia Scrum como um todo. Dessa maneira, o mesmo deve concentrar sua atenção na criação de incrementos de valor no produto, remover impedimentos ao progresso da equipe e garantir que todos os eventos ocorram de forma produtiva.

## 3.2 LINGUAGEM C E ASSEMBLY

O *Assembly* surgiu em meados dos anos 50, quando os computadores ainda eram movidos a válvula. Com a invenção do transistor, os computadores ficaram mais compactos e rápidos, e assim, surgiu a necessidade de se criar um novo modelo de programação. Esse modelo deveria ter leitura e escrita mais simples que cartões perfurados, mas mantendo o nível de programação próximo da linguagem de máquina. E foi assim que surgiram as linguagens de baixo nível e, consequentemente, a primeira linguagem de baixo nível, o *Assembly* (Andrade, 2014).

Programar em *Assembly* não é simples, visto que não é uma linguagem de boa portabilidade, pois há dependência da arquitetura de *hardware* para qual se direciona, como por exemplo, x86, *Advanced Reduced Instruction Set Computer Machine* (ARM), entre outras. No entanto, há algumas semelhanças como códigos mnemônicos, que são mais simples do que utilizar os comandos em linguagem de máquina, que seria diretamente os valores binários correspondentes a cada operação.

Agora, falando de linguagem C, temos um panorama diferente, pois se caracteriza como uma linguagem de programação de propósito geral que apresenta economia de expressão, fluxo de controle moderno e estruturas de dados, além de um rico conjunto de operadores. Apesar de ser conhecida por ser uma linguagem de alto nível, e não ser muito extensa, ainda possui algumas vantagens interessantes como ausência de restrições e generalidade, que a tornam mais apropriada do que outras linguagens mais populares e com maiores comunidades atualmente. C foi originalmente projetada e implementada no sistema operacional UNIX, por Dennis Ritchie. Além disso, esta linguagem de programação não é para nenhum *hardware* ou sistema em particular, garantindo excelente portabilidade. Dadas todas essas características, C é uma linguagem bastante expressiva e versátil (Kernighan e Ritchie, 1978).

Nos dias atuais, mesmo que as linguagens de montagem não sejam tão utilizadas em sistemas embarcados, o entendimento de como a construção e uso de cada instrução ainda é crucial para um bom desempenho do sistema e facilitar a compreensão necessária no desenvolvimento.

Em primeiro lugar, a montagem não é outra linguagem de programação. É uma interface de baixo nível, entre hardware e software. Ela proporciona uma melhor compreensão de como executar um programa. Também é necessário levar em consideração, que os programas de montagem podem, em diversos casos, ser executados mais rapidamente do que os programas C. Ocasionalmente, os compiladores estão passíveis de não poderem utilizar plenamente as características de *hardware* de um determinado processador, principalmente quando o processador fornece algumas operações específicas de que os compiladores não estão cientes. Sendo assim, frequentemente se faz necessário o uso de idiomas de montagem, para desenvolver alguma parte sensível do sistema desenvolvido com o objetivo de melhorar a velocidade de uma aplicação (Zhu, 2018).

Adentrando em questões relacionadas mais diretamente ao código, existem algumas operações que precisam ser, necessariamente, realizadas em linguagem de montagem pois simplesmente não há declaração equivalente na linguagem C. Em decorrência disso, os programas de montagem estão frequentemente nos códigos do núcleo dos sistemas operacionais para implementar tarefas de baixo nível, como exemplos, inicialização e programação de CPU (*Central Process Unit*).

## 3.3 PROTOCOLOS DE COMUNICAÇÃO SERIAL

A comunicação entre dispositivos pode ser feita de diversas formas. Uma delas é definida de forma onde possa se ter controle com o envio de instruções, representados por bits, a partir de um dos dispositivos, chamado de mestre, para outros que deverão executar as instruções recebidas, chamado de escravo.

Para isso, existem alguns protocolos, que possuem regras definidas, de forma que a transmissão e recepção de dados possa ser realizada de maneira a permitir a comunicação entre dois ou mais dispositivos em operação.

Os bits podem ser transmitidos de forma paralela ou serial. Na forma serial, os dados são transferidos bit a bit, um após o outro, assim é necessário apenas um fio que conecte fisicamente os dispositivos.

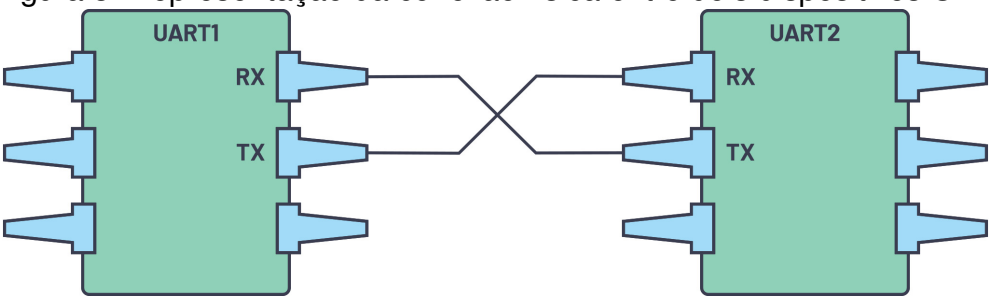
O protocolo UART, em português, Receptor-Transmissor Assíncrono Universal, será mostrado a fim de caracterizar um dos tipos de comunicação realizados pelo microcontrolador utilizado no estágio.

### 3.3.1 UART - UNIVERSAL ASYNCHRONOUS RECEIVER/TRANSMITTER

UART é um dos mais utilizados protocolos de comunicação entre dispositivos. Ao ser configurado corretamente, o dispositivo que opera segundo este protocolo, funciona com vários tipos diferentes de protocolos de comunicação que envolvem transmissão e recepção de dados em série (Peňa, 2020).

Na comunicação UART, dois dispositivos se comunicam diretamente um com o outro. O funcionamento se dá quando o transmissor converte dados paralelos de um dispositivo de controle para o formato serial, e então transmite-os em série para o receptor UART, que então converte os dados seriais de volta em dados paralelos para o outro dispositivo. Apenas dois fios são necessários para transmitir dados entre duas UARTs. Os dados fluem do pino Tx da UART transmissora para o pino Rx da UART receptora, conforme mostrado na Figura 3.

Figura 3: Representação da conexão física entre dois dispositivos UART




Fonte: Peňa (2020)

As UARTs transmitem os dados de forma assíncrona, ou seja, não há sinal de *clock* para sincronizar a saída de bits do transmissor para a amostragem de bits pelo receptor. Em vez de um sinal de *clock*, o transmissor adiciona bits de início e fim ao pacote de dados que está sendo transferido. Estes bits definem o início e o fim do pacote de dados para que o receptor saiba quando começar a ler os bits. Juntamente com os bits descritos, há ainda um bit de paridade, todos mostrados conforme sua posição na Figura 4. Este bit de paridade é uma forma do receptor perceber se algum dado mudou durante a transmissão, e assim identificar erros.

Figura 4: Representação do pacote no protocolo UART


Fonte: Peňa (2020)

Após o receptor ler o pacote de dados, ele conta o número de bits com o valor 1 e verifica se o total é um número par ou ímpar. Se o bit de paridade for um 0 (paridade par), os bits 1 no quadro de dados devem totalizar um número par. Se o bit de paridade for 1 (paridade ímpar), os bits 1 no quadro de dados devem totalizar um número ímpar.

Quando o receptor detecta um bit inicial, ele começa a ler os bits de entrada em uma frequência específica conhecida como *baud rate*. Essa frequência é uma medida da velocidade de transferência de dados, expressa em bits por segundo. Ambos os dispositivos devem operar aproximadamente com a mesma frequência. A taxa de transmissão entre o transmissor e receptor só pode diferir em cerca de 10%, antes que o tempo dos bits fique muito distante.

## 3.4 SISTEMAS EMBARCADOS

De acordo com Murti (2022), qualquer dispositivo que tenha poder de processamento embutido, assim como certa capacidade de entrada e saída, além de certa memória incorporada é um sistema embarcado.

Os primeiros microprocessadores surgiram em meados dos anos 70, e pouco a pouco, se tornaram a base dos sistemas computacionais, a medida que sua capacidade de processamento evoluiu. Atualmente, um número enorme de microprocessadores é usado na implementação de vários sistemas embarcados.

Há algumas especificidades dos embarcados que os distinguem de sistemas computacionais universais. Pois, um sistema embarcado específico executa um quantidade muito limitada de tarefas, além de monitorar e controlar alguns objetos físicos. Para alcançar esse intento, se faz necessária a conciliação das diferenças na forma de representar da informação, na velocidade do processamento da informação e a velocidade dos processos em um objeto físico observado (Barkalov et al., 2019).

Outro fator importante, é que muitos sistemas embarcados são móveis e precisam de baterias como fonte de alimentação. Portanto, para sistemas de características móveis e autônomas, é importante reduzir o máximo possível o consumo de energia. Os sistemas computacionais que atendem um propósito geral não possuem esses problemas. Seu principal objetivo é realizar vários cálculos o mais rápido possível.

Mesmo com essas limitações, os sistemas embarcados possuem certa complexidade na produção de *firmware*, que seja capaz de gerenciar todo o sistema de maneira eficiente. Portanto, o *hardware* utilizado será discutido para facilitar a compreensão dos códigos produzidos.

### 3.4.1 PROCESSADOR ARM CORTEX-M4F

A ARM é uma das empresas líderes em propriedade intelectual de semicondutores fundada em 1990, como *Advanced RISC Machine*. Ao longo dos anos, a ARM tem desenvolvido muitos projetos de processadores, um deles é o perfil Cortex M (ARM, 2022).

Este perfil utiliza o design da arquitetura ARMv6-M para seus processadores finais como Cortex-M0, Cortex-M0 plus e arquitetura ARMv7-M para seus projetos superiores, como Cortex-M3 e Cortex-M4. Os processadores são otimizados para aplicações sensíveis ao custo e à potência, como medição inteligente, sistemas de controle automotivo e industrial, e dispositivos com interface homem-máquina. O perfil consiste em Cortex-M0 e Cortex-M0 plus para potência ultra-baixa e Cortex-M3 e Cortex-M4 para aplicações de processamento digital de sinais.

### 3.4.2 MICROCONTROLADOR TIVA™ TM4C123GH6PM

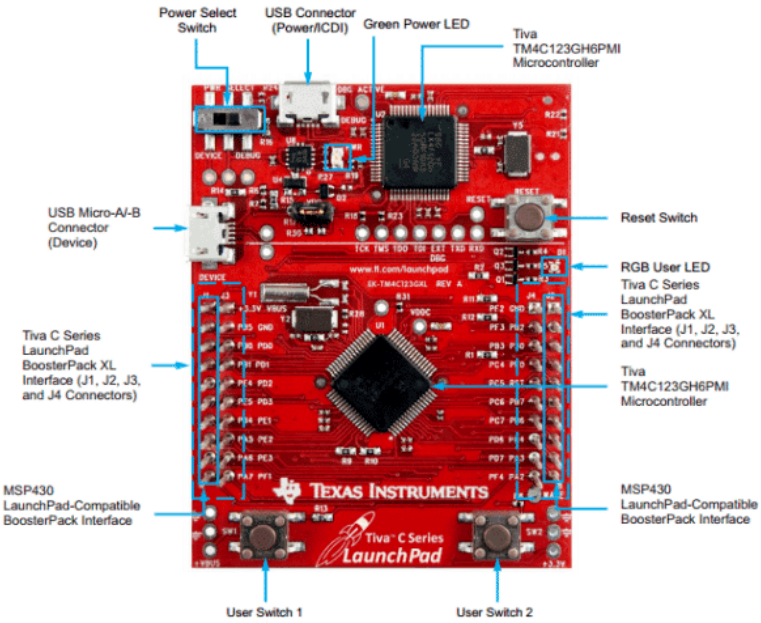
O microcontrolador utilizado no estágio foi a Tiva ™ TM4C123GH6PM, mostrado na Figura 5. O processador desse microcontrolador é um ARM Cortex-M4 que possui 32 bits com memória Flash de 256kB, frequência de *clock* máxima de 80MHz, *host* USB, módulo de hibernação, modulação por largura de pulso e uma ampla gama de outros periféricos. Um multiplexador interno permite que diferentes funções periféricas sejam atribuídas a cada um desses pinos de entrada e saída de propósito geral (GPIO) (Texas Instruments, 2014).

O ARM Cortex-M4F, inclui alguns componentes do sistema, como na Figura 6. Um desses componentes, é um temporizador de contagem regressiva de 24 bits, chamado ”SysTick”, que pode ser usado como um temporizador RTOS (*Real-Time Operating System*) ou como um contador simples. Também há um Controlador de Interrupção Vetorial Aninhado, que nada mais é, do que um controlador de interrupção incorporado que suporta processamento de interrupção de baixa latência.

Assim como um bloco de Controle de Sistema, que fornece informações de implementação e controle do sistema, incluindo configuração, monitoramento, atuação e relatório de exceções, e uma Unidade de Ponto Flutuante, dedicada a suportar a adição, subtração, multiplicação, divisão, algumas operações de precisão única, e de raiz quadrada. Ela também fornece conversões e operações de ponto flutuante.

Ainda na Figura 6, está ilustrado por meio de um diagrama de blocos de alto nível o microcontrolador Tiva™ TM4C123GH6PM. Podemos observar diversos periféricos acoplados por dois barramentos. Os blocos explorados neste estágio foram as GPIOs, controlador de relógio (por meio do controlador de sistema) e UART.

Figura 5: Imagem do Microcontrolador Tiva™ TM4C123GH6PM




Fonte: Texas Instruments (2020)

### 3.4.3 ENTRADAS E SAÍDAS DE PROPÓSITO GERAL

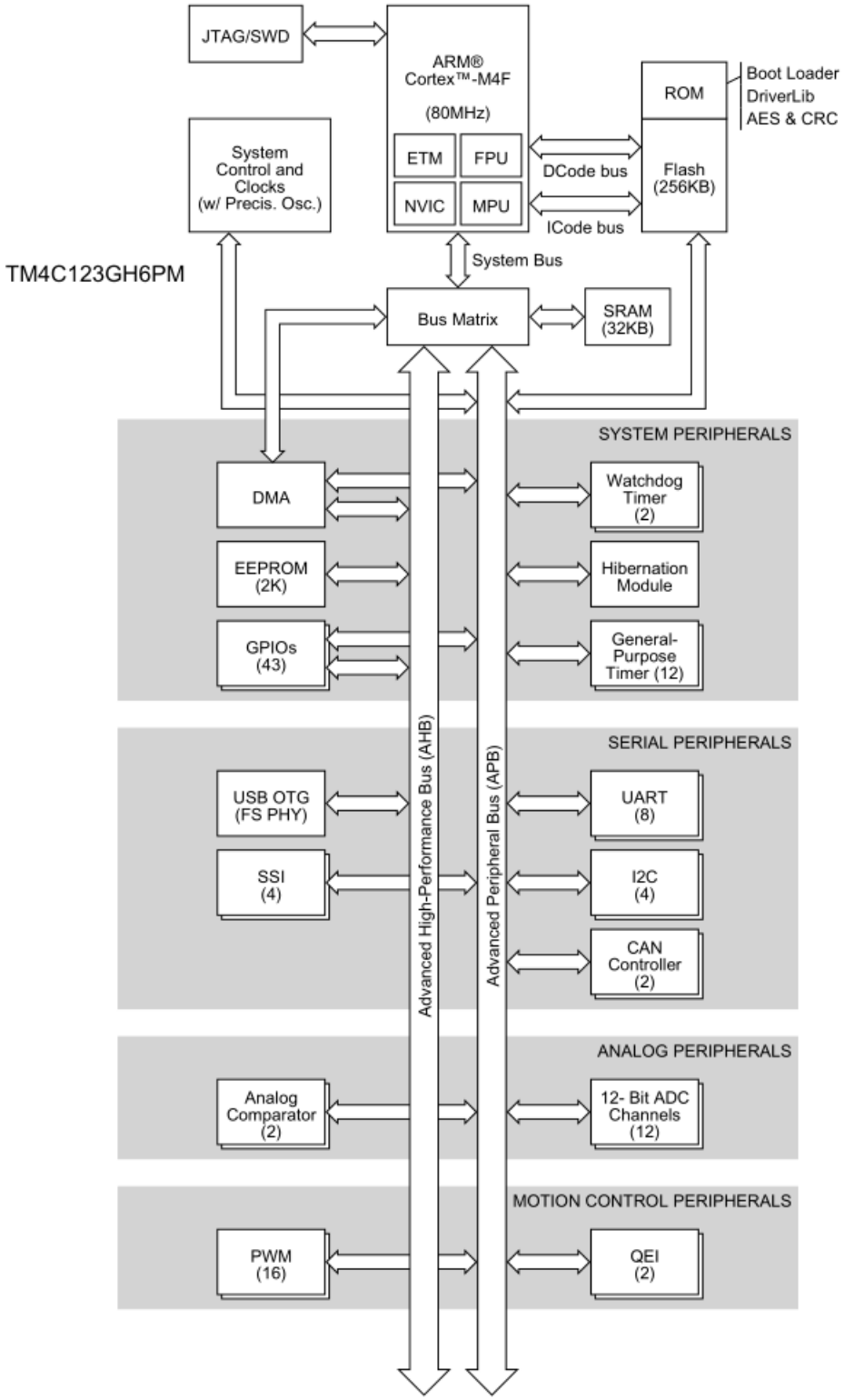
O módulo GPIO é composto de seis blocos físicos, cada um deles, correspondendo a uma porta individual (Porta A, Porta B, Porta C, Porta D, Porta E, Porta F). Esse módulo suporta até 43 pinos de entrada/saída programáveis, dependendo dos periféricos que estão sendo utilizados (Texas Instruments, 2014).

Dependendo da configuração, o módulo GPIO pode ter até 43 unidades ativas, em que as portas são acessadas através de um barramento periférico avançado. Ao utilizar mais de um pino, uma alta flexibilidade é alcançada, fazendo com que seja possível o uso de várias funções periféricas.

Uma funcionalidade particularmente útil, quando utilizada a programação em *Assembly*, é o mascaramento de bits em operações de leitura e escrita através de linhas de endereço. No que diz respeito as interrupções, o controle programável é capaz de ser acionado por borda de subida, queda ou ambas, bem como, pode ser configurado para estar sensível ao nível lógico escolhido.

Além disso, possui entrada tolerante a 5V e controle programável para configuração de pinos como baixas resistências para definir o nível lógico para alto ou baixo, correntes de 2, 4 e 8mA para comunicação digital, até 4 pinos para permitir 18mA em aplicações de alta corrente, controle da taxa de rotação para acionamento em 8 mA, operação em dreno aberto e também como entradas digitais.

Figura 6: Diagramas de blocos de alto nível da Tiva™ TM4C123GH6PM




Fonte: Texas Instruments (2014)

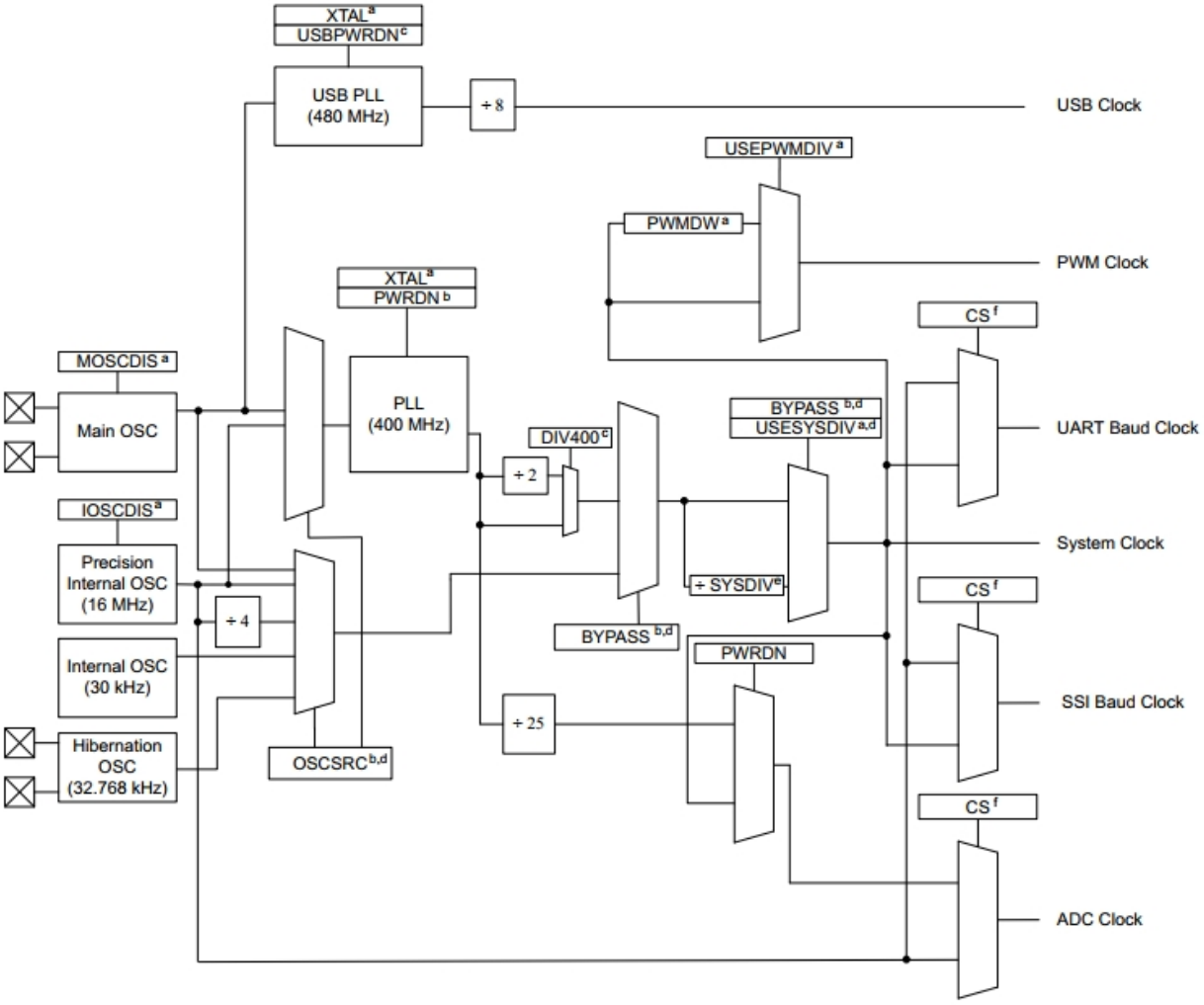
### 3.4.4 CONTROLADOR DE RELÓGIO

O controlador de sistema determina o controle dos *clocks*. Como foram utilizados conceitos de interrupção e *timers*, esta unidade será mostrada de forma independente.

Existem várias fontes de relógio para uso nesse microcontrolador, são elas, o oscilador interno de precisão, o oscilador principal, o oscilador interno de baixa frequência e o módulo de hibernação. Para o caso de uso do estágio, foi utilizado apenas o oscilador interno de precisão.

O oscilador interno de precisão, na Figura 7, é uma fonte de *clock* que não requer o uso de nenhum componente externo e fornece uma frequência de 16MHz com ±1% de precisão com calibração e ±3%, de acordo com a temperatura. Esse oscilador, permite um custo reduzido do sistema em aplicações que requerem uma fonte de relógio precisa.

Figura 7: Diagrama de blocos e componentes da árvore de relógio




Fonte: Texas Instruments (2014)

Ao acionar esse oscilador, é possível calibrar a saída para diversos valores de frequência, inclusive para 80MHz, utilizando fatores multiplicativos na saída do PLL (*Phase Locked Loop*) de 400MHz, no diagrama de blocos da Figura 7, que é o bloco que garante a elevada precisão no *clock* fornecido na saída.

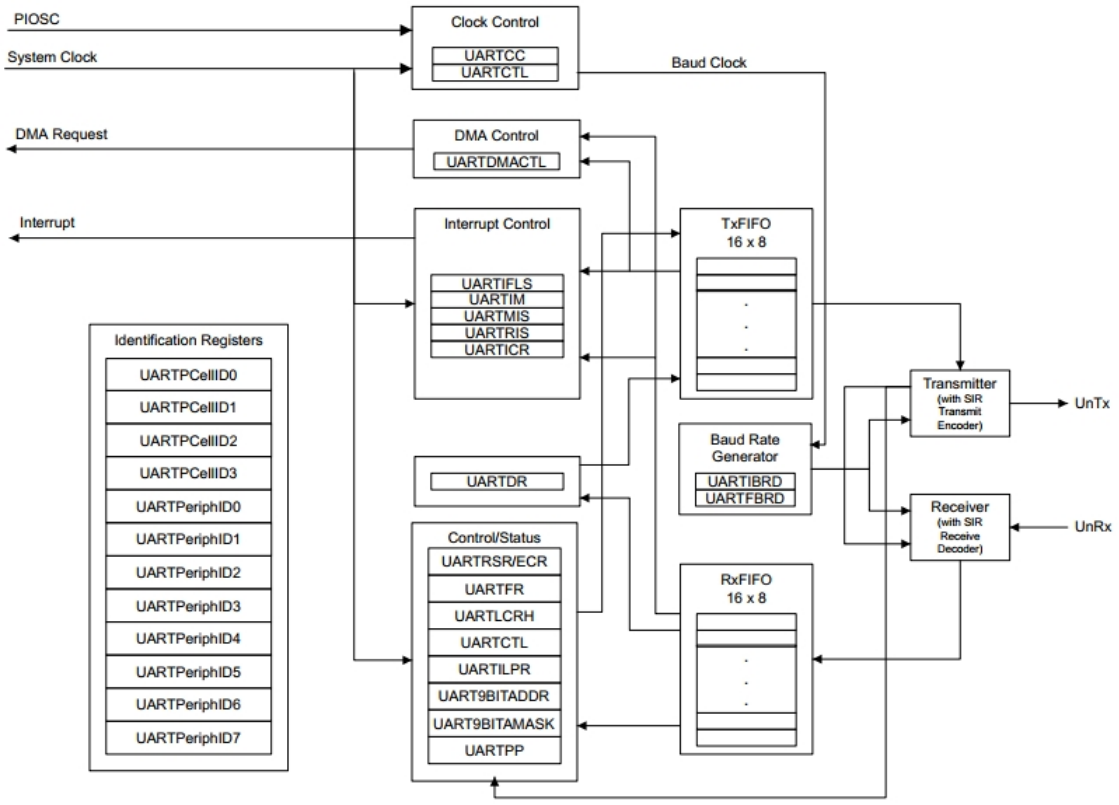
Independentemente do fato desse oscilador ser a fonte para o relógio do sistema, a sua configuração pode ser alterada para gerar o *clock* para o conversor analógico-digital, para o *baud rate* dos módulos UART, assim como em outros periféricos.

### 3.4.5 UART

O controlador TM4C123GH6PM inclui oito receptores/transmissores assíncronos universais (UART) com gerador de *baud rate* programável permitindo velocidades de até 5 Mbps para velocidade regular e 10 Mbps para alta velocidade.

Na Figura 8, podemos verificar como o sinal de *clock* chega ao módulo UART, até a geração do *baud rate* necessário para o funcionamento da leitura, e assim a saída de transmissão e entrada de recepção possam passar ou receber os dados presentes nas filas de armazenamento 16x8.

Figura 8: Diagrama de blocos do módulo UART




Fonte: Texas Instruments (2014)

Há um separação entre as filas (*First-In-First-Out*) da transmissão (Tx) e recepção (Rx) com capacidade de 16x8 para reduzir a carga de operação de interrupção da CPU. Já os bits são de comunicação assíncrona padrão para início, parada e paridade.

Esse módulo possui como características da interface serial totalmente programável, a capacidade de 5, 6, 7, ou 8 bits de dados, geração/detecção de bits pares, sem paridade e seleção de 1 ou 2 bits de parada.

Neste capítulo, foram discutidos os conceitos essenciais relacionados a metodologia ágil Scrum, protocolo de comunicação serial UART e o sistema embarcado utilizado durante o estágio. No próximo capítulo, serão apresentadas as atividades desenvolvidas de acordo com os assuntos já abordados.

# 4 ATIVIDADES REALIZADAS

Neste capítulo serão apresentadas as principais atividades desenvolvidas durante o estágio. O estagiário foi integrado a uma equipe técnica, que se utilizou de um processo de desenvolvimento ágil de sistemas para planejamento e execução das tarefas.

Portanto, com o intuito de alcançar o objetivo definido na seção 2.3, um fluxo de atividades foi realizado de modo a capacitar o estagiário nas tecnologias alvo do projeto. Para tanto, foram realizadas as atividades de aprimoramento das técnicas de programação por meio de boas práticas, tais como, revisão e validação dos códigos produzidos, leitura e modificação do valor de registradores para acesso as entradas e saídas de propósito geral, assim como, o uso da comunicação serial UART, para que o microcontrolador se comunicasse com outros dispositivos, que poderiam ser acessados remotamente, e assim possibilitar o acionamento e monitoramento de periféricos.

## 4.1 PROCESSOS DE GESTÃO E DESENVOLVIMENTO ÁGIL

Durante o período do estágio, foi utilizado da metodologia Scrum, no qual seus fundamentos já foram introduzidos na seção 3.1, para melhor gerenciamento de desenvolvimento do projeto. A equipe de Scrum foi composta por doze integrantes, dos quais, dez pessoas constituíram a equipe técnica, contando com o estagiário. Ademais, havia um *Scrum Master* e um *Product Owner*.

Os *Product Backlogs* foram criados pela equipe técnica em conjunto com o *Scrum Master*, de forma a realizar um treinamento que fosse capaz de desenvolver habilidades no estagiário de maneira adaptada as demandas do projeto em questão. Assim, com as macro-atividades criadas e as horas dedicadas definidas, foi dado início as *Sprints*.

Essas *Sprints* tinham duração de duas semanas, com início na segunda-feira, logo após o planejamento das atividades de todos os colaboradores do time e das suas respectivas horas de trabalho dedicadas para cada tarefa. Por meio de uma plataforma interna, foram organizadas as atividades que iriam ser realizadas de acordo com o tempo hábil e nível adquirido pelo estagiário para realização da atividade no período da *Sprint*.

As tarefas a serem realizadas foram dispostas na plataforma e inicializada com o estado de *TO DO*. A partir do momento que era dado o início da tarefa, o seu estado passava a ser *ON GOING* e permanecendo assim até sua finalização. Ao concluir uma tarefa, esta era posta em espera, como em *REVIEW*, para análise de outros colaboradores do time com eventuais sugestões para correção. Finalmente, ao ser constatada a conclusão com sucesso da tarefa, o estado de *DONE* deveria ser atribuído.

Para acompanhamento durante as *Sprints*, foram realizadas reuniões diárias que tinham duração de aproximadamente 15 minutos, com limite máximo de 30 minutos, para evitar o bloqueio do time no cumprimento das atividades atribuídas. Nessas reuniões, eram reportados ao *Scrum Master*, o andamento do projeto, inclusive os sucessos e impedimentos encontrados durante o desenvolvimento.

Ao final de cada mês, também aconteciam reuniões com o *Product Owner*, com o objetivo de relatar o quão produtiva foi a *Sprint*, bem como, receber auxílio referente a relevância das atividades desenvolvidas para o contexto de sistemas embarcados.

## 4.2 APRIMORAMENTO NA IMPLEMENTAÇÃO DE SOFTWARE

Inicialmente, a tarefa proposta foi a implementação de um código em linguagem C, capaz de organizar dados que seriam transmitidos por meio de um barramento. Para realizar a transmissão no formato desejado, ao receber 32 bits, os mesmos deveriam ser separados em um número em representação de ponto flutuante, e assim informar na saída a decodificação para decimal.

A função principal (*main*) se encontra no Algoritmo 1. Podemos visualizar nessa função, primeiramente nas linhas de 11 a 13, a extração do conjunto de bytes, de acordo com a precisão simples do padrão IEEE754, ou seja, um bit de sinal, 8 bits de expoente e 23 bits para a mantissa, com o auxílio dos operadores de deslocamento ($<<$ e $>>$).

Posteriormente, são corrigidos com o bit 1 mais significativo escondido na mantissa (linha 15), a soma do número bias no expoente (linha 16), nesse caso 0x7F, e o sinal de cada número (linha 17), caso seja um valor negativo. Assim, finalmente obtemos o valor decodificado para ponto flutuante em base decimal.

A função ”double converter_mant_bin_dec(uint32_t);”, da linha 24 até a 36, converte os bits da mantissa de binário na representação de ponto flutuante de precisão simples, para decimal, usando a posição dos bits como expoente na potência de dois na linha 31 e pelo decremento da variável ”sequencial” na linha 32, com o objetivo de dar início a conversão do valor informado.

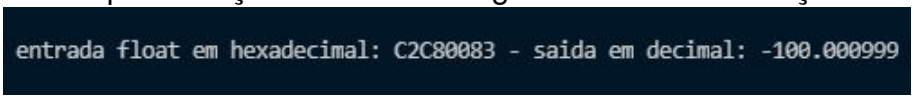
Entretanto, ao se analisar o código do Algoritmo 1, se percebeu alguns problemas relacionados com a aplicação a qual o mesmo se destina. Primeiramente, o código possui operações que demandam bastante processamento, como multiplicação por diversas potências de 2. Também não foi levado em consideração que o processador pode operar com diferentes representações numéricas.

Finalmente, na Figura 9, podemos ver a entrada dos bits, em formato hexadecimal e a saída do programa, produzida pela linha 19, em decimal.

```c
#include <stdio.h>
#include <math.h>
#include <stdint.h>

double converter_mant_bin_dec(uint32_t);

int main(){
 uint32_t array_bytes = 0xC2C80083, mantissa, expoente, sinal;
 double number;

 sinal = array_bytes >> 31;
 expoente = (array_bytes << 1) >> 24;
 mantissa = (array_bytes << 9) >> 9;

 number = (1 + converter_mant_bin_dec(mantissa));
 number *= pow(2, (expoente - 0x7F));
 if(sinal) number *= -1;

 printf("\nentrada float em hexadecimal: %X - saida em decimal: %lf\n",
    array_bytes,number);

 return 0;
}

double converter_mant_bin_dec(uint32_t valor){
 short int resto, sequencial = 23;
 double valor_decimal = 0;

 while (valor != 0){
  resto = valor % 2;
  valor /= 2;
  valor_decimal += resto * pow(2, -1*sequencial);
  --sequencial;
 }

 return valor_decimal;
}
```

Algoritmo 1: Decodificador de precisão simples IEEE754 para base decimal

Figura 9: Representação da saída do algoritmo de decodificação IEEE754




Fonte: Autoria Própria

Para melhorar esse código, foi desenvolvido o Algoritmo 2, que se utiliza do conceito de compartilhamento de memória promovido pela estrutura de dados ”union data”, na linha 5, para reduzir o uso de memória utilizado. Pois em apenas um valor de ponto flutuante há 32 bits (4 bytes), enquanto em um tipo char há 8 bits (1 byte). Dessa forma, é possível manipular cada byte por vez, como nas linhas 12 a 16, e organizar melhor os dados.

```
1  #include <stdio.h>
2
3  typedef union data data_t;
4
5  union data {
6   float f;
7   char c[4];
8  };
9
10 int main() {
11  data_t dados;
12
13  dados.c[3] = 0xC2;
14  dados.c[2] = 0xC8;
15  dados.c[1] = 0x00;
16  dados.c[0] = 0x83;
17
18  printf("\nvalor convertido: %f\n", dados.f);
19
20  return 0;
21 }
```

Algoritmo 2: Otimização de decodificador de precisão simples IEEE754

Além disso, não há uso de operações matemáticas explícitas na conversão. Apenas o melhor uso da capacidade do processador de utilizar diferentes representações. Todas as vantagens citadas são de fundamental importância, dado a pequena disponibilidade de recursos em sistemas embarcados.

Na Figura 10, por meio da linha 18, vemos a saída do programa otimizado, que é idêntica a primeira implementação, em termos de precisão, porém muito mais eficiente, principalmente sendo executado em um microcontrolador, por exemplo.

Figura 10: Representação da saída do algoritmo otimizado para decodificação IEEE754


Fonte: Autoria Própria

Na Tabela 1, podemos comparar o tempo de execução de cada algoritmo. Nessa análise tempo, o algoritmo otimizado leva apenas 66,06% do que o primeiro código, o que é bastante significativo quando se está trabalhando com sistemas embarcados que devem possuir respostas rápidas aos estímulos do ambiente.

Tabela 1: Tempo de execução para cada implementação do decodificador IEEE754

| Algoritmo | Tempo de Execução (ms) |
|---|---|
| 1 | 439 |
| 2 | 290 |

Fonte: Autoria Própria

## 4.3 IMPLEMENTAÇÃO DE SOFTWARE PARA ESTUDO E VALIDAÇÃO DE COMUNICAÇÃO SERIAL UART

Uma outra atividade, foi escrever um programa, na linguagem de programação C, que empacotasse e desempacotasse um conjunto de 16 bytes, de modo semelhante à comunicação por protocolo UART, que foi mostrado na seção 3.3.1.

O Algoritmo 3 mostra a implementação deste protocolo.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5
6 #define DATA_SIZE 3 //Quantidade de dados na mensagem
7 #define BYTE_SIZE 16 //Bytes na mensagem
```

```
8  #define INITIAL_CONDITION_BYTE 0xAA //Byte padronizado no comeco
9  #define STOP_CONDITION_BYTE 0x55 //Byte padronizado no final
10
11 #define FIRST_DATA_BYTE_ADDRESS 0 //Indexador do byte inicial
12 #define ADDRESS_FRAME 1 //Indexador do byte de id da mensagem
13 #define DEVICE_ADDRESS 2 //Indexador do byte de id do dispositivo
14 #define FIRST_DATA_ADDRESS 3 //Indexador do primeiro byte de dados
15 #define FINAL_DATA_ADDRESS 14 //Indexador do ultimo byte de dados
16 #define END_DATA_ADDRESS  15 // Indexador do byte final
17
18 typedef union data data_t; //Dados inclusos ou recuperados das
      mensagens
19
20 union data {
21  float f;
22  char c[4];
23 };
24
25 typedef enum data_message_id data_message_id_t; //Identificadores das
      mensagens
26
27 enum data_message_id {
28  REQUEST_DATA_UPDATE = 0,
29  RESPONSE_DATA_UPDATE,
30  CONF_DATA
31 };
32
33 typedef struct data_message data_message_t; //Pacote de dados
34
35 struct data_message {
36  data_message_id_t data_message_id;
37  uint8_t device_id;
38  data_t data[DATA_SIZE];
39  uint8_t byte_buffer[BYTE_SIZE];
40 };
41
42 bool data_message_unpack(data_message_t *data_message);
43 //funcao que desempacota os dados no array
```

```
44
45 void data_message_pack(data_message_t *data_message, uint8_t id,
       data_message_id_t data_message_id, data_t *data);
46 //funcao que empacota os dados no array
47
48 void data_message_print(data_message_t data_message);
49 //funcao que exibe a mensagem
50
51 int main(){
52  data_t dados[DATA_SIZE];
53  data_message_id_t mensagem_id;
54  data_message_t mensagem_dados;
55  uint8_t device_id;
56
57  //Identificacao do tipo de mensagem e do dispositivo para o
        empacotamento
58  mensagem_id = RESPONSE_DATA_UPDATE;
59  device_id = 0x1;
60
61  //Preparando dados para o empacotamento
62  for (uint8_t dado = 0; dado < DATA_SIZE; dado++){
63   dados[dado].c[0] = 0xB4; dados[dado].c[1] = 0x11;
64   dados[dado].c[2] = 0x85; dados[dado].c[3] = 0x1D;
65  }
66
67  //Realizando o empacotamento
68  printf("\nEmpacotando dados...\n");
69  data_message_pack(&mensagem_dados, device_id, mensagem_id, dados);
70  data_message_print(mensagem_dados);
71
72  //Realizando o desempacotamento e sua validacao
73  printf("\nDesempacotando dados...\n");
74  if(data_message_unpack(&mensagem_dados)){
75   printf("ID do circuito: 0x%X - ID do pacote: 0x%X",mensagem_dados.
        device_id,mensagem_dados.data_message_id);
76
77   for (uint8_t dado = 0; dado < DATA_SIZE; dado++)
78    printf(" - Dado[%d]: 0x%X",dado+1,mensagem_dados.data[dado]);
```

```
79
80   printf("\n\nPACOTE VALIDO\n");
81  }
82
83  else printf("\nPACOTE INVALIDO!\n");
84
85  return (EXIT_SUCCESS);
86 }
87
88 bool data_message_unpack(data_message_t *data_message){
89  uint8_t byte_pacote = DATA_IN_PACKAGE_END_BYTE_ADDRESS;
90
91  //Teste de validacao da mensagem com o primeiro e o ultimo byte
92  if(data_message->byte_buffer[FIRST_DATA_BYTE_ADDRESS] ==
       INITIAL_CONDITION_BYTE && data_message->byte_buffer[
       END_DATA_ADDRESS] == STOP_CONDITION_BYTE){
93
94   //Recuperando os bytes de identificacao do dispositivo e da mensagem
95   data_message->device_id = data_message->byte_buffer[ADDRESS_FRAME];
96   data_message->data_message_id = data_message->byte_buffer[
        DEVICE_ADDRESS];
97
98   //Recuperando os bytes de dados da mensagem
99   for (uint8_t dado = 0; dado < DATA_SIZE; dado++){
100   for (uint8_t byte_dado = 0; byte_dado <= DATA_SIZE; byte_dado++){
101    data_message->data[dado].c[byte_dado] = data_message->byte_buffer[
          byte_message-byte_dado];
102   }
103   byte_pacote-=4;
104  }
105
106  return true;
107
108 } else {
109  return false;
110 }
111 }
112
```

```
113 void data_message_pack(data_message_t *data_message, uint8_t device_id,
        data_message_id_t data_message_id, data_t *data){
114  uint8_t byte_pacote = DATA_IN_PACKAGE_BEGIN_BYTE_ADDRESS;
115
116  //Inserindo os bytes de inicio, identificacao do dispositivo e da
        mensagem
117  data_message->byte_buffer[FIRST_DATA_BYTE_ADDRESS] =
        INITIAL_CONDITION_BYTE;
118  data_message->byte_buffer[ADDRESS_FRAME] = data_message_id;
119  data_message->byte_buffer[DEVICE_ADDRESS] = device_id;
120
121  //Inserindo os bytes de dados na mensagem
122  for (uint8_t dado = 0; dado < DATA_SIZE; dado++){
123   for (uint8_t byte_dado = 0; byte_dado <= DATA_SIZE; byte_dado++){
124    data_message->byte_buffer[byte_mensagem+byte_dado] = data[dado].c[
          byte_dado];
125   }
126   byte_pacote+=4;
127  }
128
129  //Inserindo o byte final na mensagem
130  data_message->byte_buffer[END_DATA_ADDRESS] = STOP_CONDITION_BYTE;
131 }
132
133 void data_message_print(data_message_t data_message){
134  printf("[ ");
135  for (uint8_t byte_mensagem = 0; byte_mensagem < BYTE_SIZE;
        byte_mensagem++){
136   printf("0x%X ",data_message.byte_buffer[byte_mensagem]);
137  }
138  printf("]\n");
139 }
```

Algoritmo 3: Implementação do modo de comunicação serial UART

A mensagem se chama ”data_message_t” (definida na linha 33) e contém como propriedades, ”data_message_id”, como identificador do pacote, que pode ser ”REQUEST_DATA_UPDATE”, ”RESPONSE_DATA_UPDATE” ou ”CONF_DATA”, nas linhas 28, 29 e 30, respectivamente.

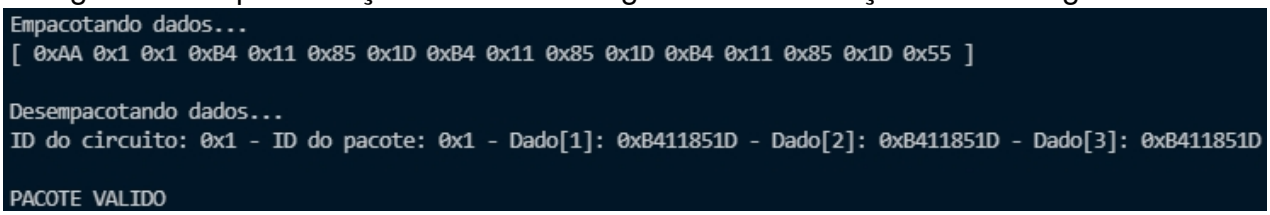
Assim como, também há ”device_id”, como identificador do dispositivo (linha 37), que pode ser um valor entre 0 e 255, ”data” (linha 38), como conjunto de dados (3 dados de 32 bits cada) a serem empacotados ou desempacotados, ”byte_buffer” (linha 39), como conjunto de 16 bytes a serem desempacotados ou após o empacotamento.

A conformação dos bytes deve ser a seguinte: o primeiro byte deve ser o de identificação do início da mensagem [0xAA], o segundo o id do dispositivo, o terceiro o id do tipo de pacote, do quarto ao sétimo os bytes do primeiro dado, do oitavo ao décimo primeiro o do segundo dado, do décimo segundo ao décimo quinto o do terceiro e o do décimo sexto o byte de identificação fim da mensagem [0x55]. Portanto, a mensagem deve possuir a seguinte forma: [INITIAL_CONDITION_BYTE, ADDRESS_FRAME, DEVICE_ADDRESS, data, STOP_CONDITION_BYTE].

No programa, há a função ”bool data_message_unpack(data_message_t *data_message);”, na linha 88, que utiliza o vetor de bytes ”byte_buffer” para preencher as variáveis ”data_message_id”, ”device_id” e ”data” e retorna verdadeiro caso o pacote seja válido. Também deve existir a função ”void data_message_pack(data_message_t *data_message, uint8_t device_id, data_message_id_t data_message_id, data_t *data);”, na linha 113, para preencher o ”byte_buffer”.

Os resultados são apresentados com a função ”void data_message_print(data_message_t data_message);”, na linha 133. Na Figura 11, está a saída do programa com a mensagem empacotada e desempacotada, bem como, a validade da mensagem.

Figura 11: Representação da saída do algoritmo de formação da mensagem UART




Fonte: Autoria Própria

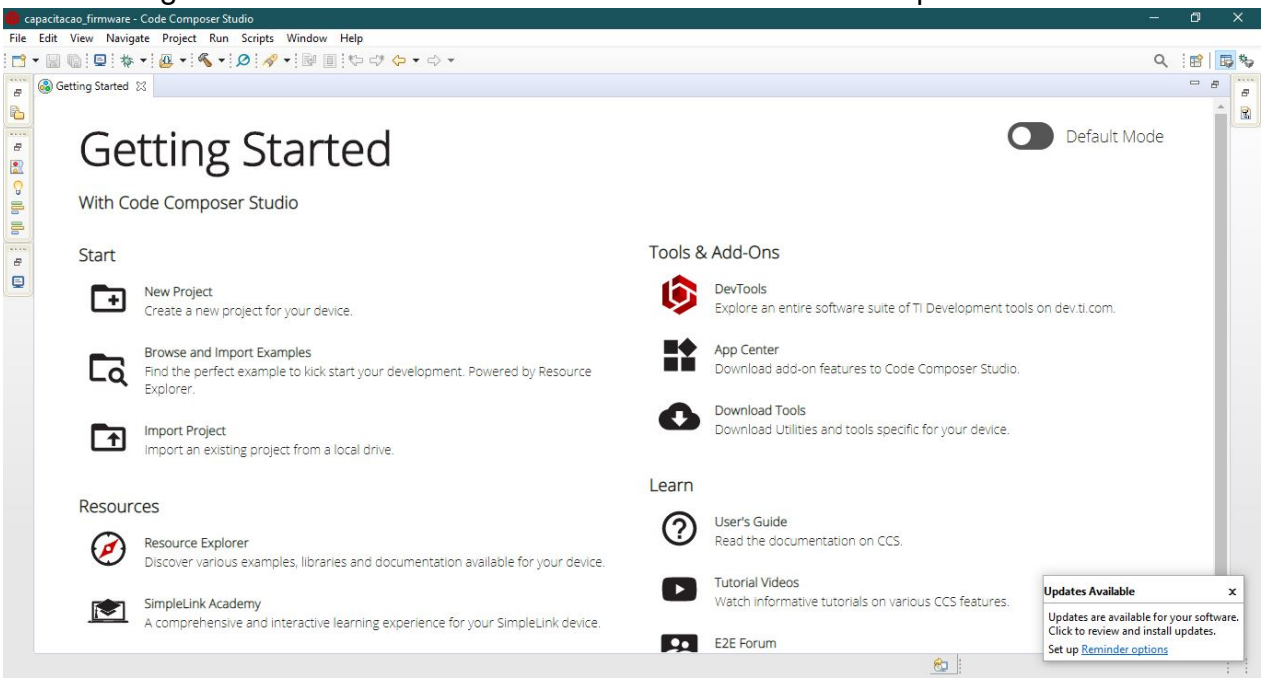
## 4.4 IMPLANTAÇÃO E USO DE AMBIENTE DE DESENVOLVIMENTO PARA SISTEMAS EMBARCADOS

Para realizar as atividades do estágio, foi utilizado como ambiente de desenvolvimento para sistemas embarcados o Code Composer Studio™ que é um ambiente de desenvolvimento integrado (IDE) que suporta os microcontroladores da Texas Instruments e portfólios de processadores incorporados.

O *software* Code Composer Studio™ , também conhecido como CCS, compreende um conjunto de ferramentas utilizadas para desenvolver e depurar aplicações referentes a sistemas embarcados. O *software* inclui um compilador otimizado de C/C++, editor de código fonte, ambiente de construção de projetos, depurador, perfilador entre outros recursos (Texas Instruments, 2021).

A janela inicial do ambiente de desenvolvimento do CCS é mostrada na Figura 12. Nessa janela, é possível criar um novo projeto, importar e navegar por exemplos iniciais de como produzir códigos nessa IDE, importar projetos já existentes, assim como ter acesso a documentação, *datasheets*, vídeos, fóruns e até *downloads* de materiais, inclusive alguns deles usados pelo estagiário, para facilitar a produção de código, e que ainda serão mostrados neste relatório.

Figura 12: Ambiente de desenvolvimento no Code Composer Studio™

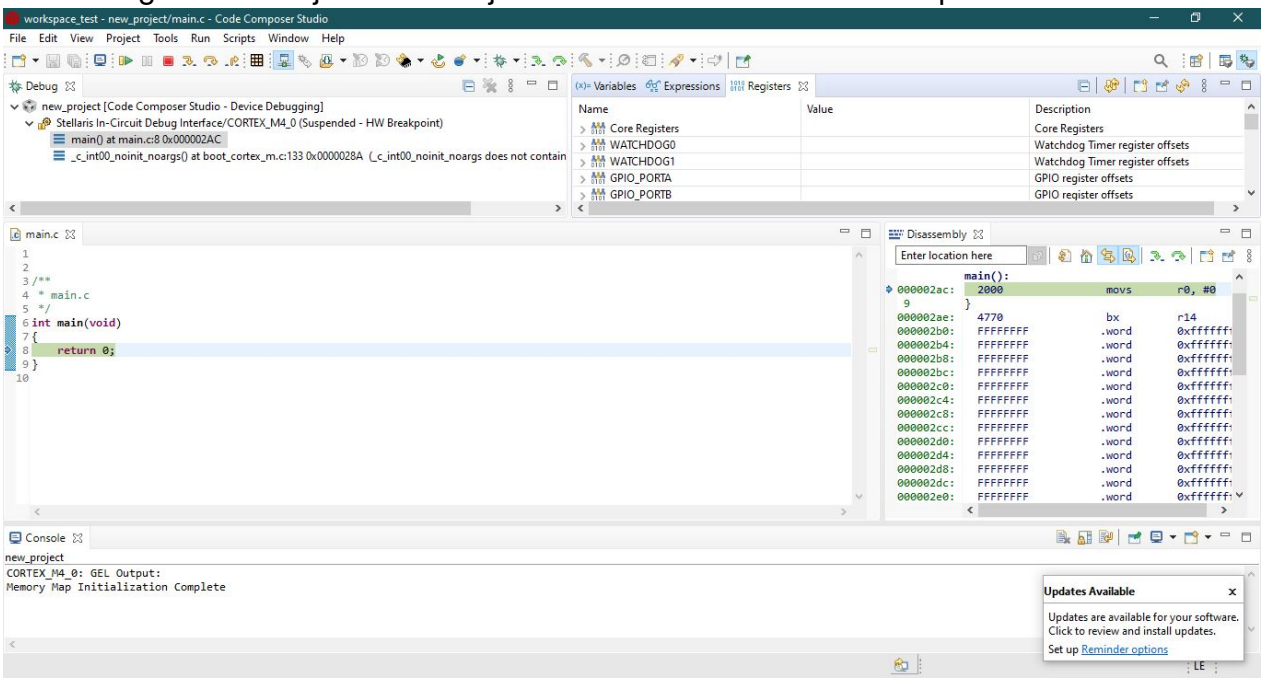



Fonte: Autoria Própria

Também já é possível observar as opções de *Debug* e *Build*, que são etapas no desenvolvimento para validação e experimentação no microcontrolador.

Na Figura 13, podemos visualizar um projeto criado em linguagem C, e as janelas auxiliares de *console* e *disassemble*, que auxiliam a identificar problemas nos programas e a executar paulatinamente os comandos digitados, respectivamente. Dessa maneira, o estagiário conseguiu desenvolver as tarefas relativas ao microcontrolador usado e explorar seus recursos.

Figura 13: Projeto inicial e janelas auxiliares no Code Composer Studio™




Fonte: Autoria Própria

## 4.5 CONFIGURAÇÃO PARA CONTROLE E MONITORAMENTO COM O USO DE PINOS DE ENTRADA E SAÍDA DE PROPÓSITO GERAL

Na primeira atividade implementada, de fato, no sistema embarcado, foi utilizada a linguagem *Assembly*, por meio do função "__asm();" para produção de *firmware*, com a intenção de entender o funcionamento do processador da Tiva C TM4C123GH6PM ao executar as instruções de acordo com a arquitetura ARM. Portanto, foi realizada a configuração dos registradores do microcontrolador, para que os diodos emissores de luz pudessem ser ligados, assim como qualquer periférico ligado aos pinos digitais da placa.

No Algotritmo 4, podemos visualizar o acesso e configuração ao registrador de controle do sistema SYSCT_RCGCGPIO pelo endereço 0x400FE608, entre as linhas 2 e 7, permitindo que a porta F entre em modo de execução após o registrador R1 receber o valor de 0x20.

```
1  int main(void){
2   __asm(" MOV R0, #0xE608;\n"
3    " MOVT R0,#0x400F;\n"
4    " LDR R1, [R0];\n"
5    " MOVW R1, #0x20;\n"
6    " STR R1, [R0];\n"
7   );
8
9   __asm(" MOV R0, #0x5400;\n"
10   " MOVT R0,#0x4002;\n"
11   " LDR R1, [R0];\n"
12   " MOVW R1, #0x04;\n"
13   " STR R1, [R0];\n"
14  );
15
16  __asm(" MOV R0, #0x551C;\n"
17   " MOVT R0,#0x4002;\n"
18   " LDR R1, [R0];\n"
19   " MOVW R1, #0x04;\n"
20   " STR R1, [R0];\n"
21  );
22
23  __asm(" MOV R0, #0x53FC;\n"
24   " MOVT R0,#0x4002;\n"
25   " LDR R1, [R0];\n"
26   " MOVW R1, #0x04;\n"
27   " STR R1, [R0];\n"
28  );
29
30   __asm(" MOV R0, #0x00;\n");
31
32  return 0;
33 }
```

Algoritmo 4: Ativação de saídas digitais

Note que, nas linhas 12, 19 e 26, o mesmo valor de 0x04 é colocado no registrador R1, que guarda endereços diferentes em cada uma delas. Isso acontece porque, a GPIO em objetivo, se encontra no pino 2, conforme pode ser consultado na Tabela 2, pois esse número indica qual bit deve ser alterado para acionar cada um dos dispositivos.

Tabela 2: Definição das entradas e saídas de propósito geral

| **Pino GPIO** | **Função do Pino** | **Dispositivo USB** |
|---|---|---|
| PF4 | GPIO | Chave 1 |
| PF0 | GPIO | Chave 2 |
| PF1 | GPIO | Diodo emissor de luz vermelha |
| PF2 | GPIO | Diodo emissor de luz azul |
| PF3 | GPIO | Diodo emissor de luz verde |

Fonte: Adaptada de Texas Instruments (2020)

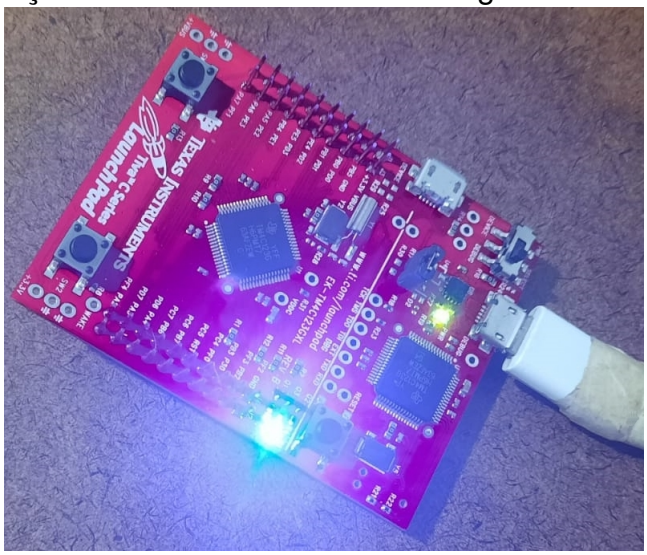
As chaves 1 e 2 apresentadas na tabela acima, também foram utilizadas como recurso de entrada de propósito geral em uma das atividades, assim como os demais diodos emissores de luz para acompanhamento da operação correta do microcontrolador.

Da linha 9 até a 14, o registrador GPIO_PORTF_DIR é configurado para definir a direção de trânsito dos dados (entrada ou saída), no endereco 0x40025400, para o pino PF2 (diodo emissor de luz azul), ou seja, como pino de saída. Para finalizar a configuração da saída digital, na linha 16 até a 21, o registrador GPIO_PORTF_DEN deve ser permitida para habilitação digital no endereço 0x4002551C.

Finalmente, para acessar e modificar o conteúdo, nas linhas 23 a 28, o registrador GPIO_PORTF_DATA com endereco 0x400253FC, deve receber o mesmo valor para assim polarizar diretamente o diodo emissor de luz azul.

O acionamento da saída digital pode ser vista na Figura 14. Dessa mesma maneira, equipamentos podem ser conectados a esse pino, para que seja realizado seu controle e monitoramento, por meio de medições com sensores nos pinos como entrada e ativação de atuadores nos pinos como saída.

Figura 14: Representação do acionamento de saída digital no diodo emissor de luz azul




Fonte: Autoria Própria

## 4.6 EXPERIMENTAÇÃO DE PROTOCOLO UART E INTERRUPÇÕES EM DIFERENTES CASOS DE USO

Por fim, foram apresentados ao estagiário, as bibliotecas e *drivers* fornecidos pela própria Texas Instruments, que podem facilitar o desenvolvimento ao se utilizar de funções preparadas para acionamento dos registradores para diversos usos, sem que o *data sheet* tenha que ser consultado a cada passo.

Para essa atividade, conceitos de interrupções e comunicação serial UART foram necessários para validação e experimentação do funcionamento da placa.

Portanto, fazendo uso das funções disponibilizadas, um conjunto de 16 bytes deveria ser enviado na UART1, por meio de uma interrupção disparada pelo *timer*. Além disso, ao pressionar o botão da chave 1 (consultar Tabela 2), uma interrupção de borda de descida é acionada para envio, pela UART0, dos ultimos 16 bytes recebidos.

Além disso, o diodo emissor de luz contido na Tiva deve ser usado, na cor vermelha, para indicar que o botão de chave 1 foi pressionado; na cor azul, para indicar o recebimento completo dos 16 bytes na UART1; e a cor verde para indicar o envio completo dos 16 bytes na UART0.

Primeiramente, no Algoritmo 5, são incluídas algumas bibliotecas padrão da linguagem C (linhas 1 a 3), assim como os arquivos de cabeçalho fornecidos pela Texas Instruments para uso das funções que auxiliam no desenvolvimento (linhas 5 a 14). Além disso, também são definidos a quantidade de bytes que será transmitido e recebido pela UART e o *baud rate* em 115200, nas linhas 16 e 17.

Em seguida, há a declaração das funções responsáveis pelas interrupções. A função ”void Timer0IntHandler(void);”, na linha 69, que é chamada quando há um estouro no Timer0, e assim, é responsável pelo recebimento de dados na UART1. Enquanto que, a função ”void Switch1IntHandler(void);”, na linha 109, é a função chamada na interrupção causada pelo botão da chave 1 para transmissão de dados via UART0. Também existe uma função chamada ”UARTSend”, na linha 103, que é responsável pelo envio de bytes para a UART.

Na linha 28, usando a função ”SysCtlClockSet” é possível se utilizar da arquitetura ARM para que, ao fazer uso do oscilador de precisão com o cristal em 16MHz, e esse sinal ao passar pelo PLL da Figura 7, possa possuir uma frequência de 200MHz. Após esse valor ser multiplicado por um fator de 2,5 os 80MHz de frequência são obtidos. Assim, é possível configurar o *clock* para o valor máximo alcançado pelo microcontrolador, visto que não houve limitações de projeto nesse sentido.

Para configurar a as entradas e saídas, a porta F é habilitada, assim como a definição dos diodos emissores de luz como saídas digitais e o botão da chave 1 como entrada digital, nas linhas 32 e 33, respectivamente. Com isso, há a permissão para a interrupção causada pelo botão, visualizada nas linhas 36 e 37, ao ocorrer uma borda de descida do acionamento.

Então, é feita também a configuração da UART0 e UART1, que pode ser observado nas linhas 49 e 50, aplicando o *baud rate* de 115200 que será utilizado, o comprimento de uma só palavra em 8 bits, incluindo apenas um bit de parada e sem bit de paridade, para este caso.

Finalmente, é necessário também configurar as opções de *timer*. O Timer0 é ativado como periférico e para operar de modo periódico (linhas 53 e 54). Além disso, na mesma linha 54, é importante notar que apenas a parte A é ativada, significando que 16 bits são usados para contagem, fazendo com que o disparo do timer aconteça de modo mais rápido.

O *delay* usado para avaliar o período do *timer* é então calculado na linha 56, considerando um *duty cycle* de 50 por cento, com a divisão por 2, e para estabelecer um período de disparo considerável para visualização, a divisão por 0,1 resultando em uma frequência 10 vezes menor que a capturada pela função ”SysCtlClockGet”.

Assim, para concluir há a habilitação da interrupção do Timer0A (linha 59) para quando houver estouro do contador, bem como das interrupções de modo geral ativando as demais que foram requeridas (linha 61). Para manter o sistema em operação, existe na linha 64 um *loop* infinito, logo as instruções necessárias serão efetuadas quando as interrupções forem disparadas.

```c
#include <stdint.h>
#include <stdbool.h>
#include <string.h>

#include "inc/tm4c123gh6pm.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"

#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "driverlib/gpio.h"
#include "driverlib/timer.h"
#include "driverlib/uart.h"
#include "driverlib/pin_map.h"

#define NUM_UART_DATA 16
#define BAUD_RATE 115200

uint8_t ui8DataTx[NUM_UART_DATA];
uint8_t ui8DataRx[NUM_UART_DATA];
uint32_t ui32index;

void Timer0IntHandler(void);
void Switch1IntHandler(void);
void UARTSend(uint32_t ui32UARTBase, const uint8_t *pui8Buffer,
   uint32_t ui32Count);

int main(void){
 SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ
    | SYSCTL_OSC_MAIN);

 //GPIO
 SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
 GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 |
    GPIO_PIN_3);
 GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, GPIO_PIN_4);

 IntEnable(INT_GPIOF);
```

```
36  GPIOIntEnable(GPIO_PORTF_BASE, GPIO_INT_PIN_4);
37  GPIOIntTypeSet(GPIO_PORTF_BASE, GPIO_PIN_4, GPIO_FALLING_EDGE);
38  IntMasterEnable();
39
40  //UART
41  SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
42  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
43  GPIOPinConfigure(GPIO_PA0_U0RX);
44  GPIOPinConfigure(GPIO_PA1_U0TX);
45  GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
46  SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);
47  UARTLoopbackEnable(UART1_BASE);
48
49  UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), BAUD_RATE, (
       UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
50  UARTConfigSetExpClk(UART1_BASE, SysCtlClockGet(), BAUD_RATE, (
       UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
51
52   //TIMER
53  SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
54  TimerConfigure(TIMER0_BASE, TIMER_CFG_A_PERIODIC);
55  uint32_t timer_period = 0;
56  timer_period = (SysCtlClockGet() / 0.1) / 2;
57  TimerLoadSet(TIMER0_BASE, TIMER_A, timer_period - 1);
58
59  IntEnable(INT_TIMER0A);
60  TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
61  IntMasterEnable();
62  TimerEnable(TIMER0_BASE, TIMER_A);
63
64  while(1){}
65
66  return 0;
67 }
68
69 void Timer0IntHandler(void){
70  TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
71
```

```
72  UARTSend(UART0_BASE, (uint8_t *)"\033[2J\033[1;1H", 10);
73  UARTSend(UART0_BASE, (uint8_t *)"\nComunicacao UART ->", strlen("\
        nComunicacao UART ->"));
74
75  //Bytes de dados a serem enviados pela UART
76  ui8DataTx[0] = 'E'; ui8DataTx[1] = 'S';
77  ui8DataTx[2] = 'T'; ui8DataTx[3] = '.';
78  ui8DataTx[4] = '-'; ui8DataTx[5] = 'V';
79  ui8DataTx[6] = 'I'; ui8DataTx[7] = 'R';
80  ui8DataTx[8] = 'T'; ui8DataTx[9] = 'U';
81  ui8DataTx[10] = 'S'; ui8DataTx[11] = '-';
82  ui8DataTx[12] = 'U'; ui8DataTx[13] = 'F';
83  ui8DataTx[14] = 'C'; ui8DataTx[15] = 'G';
84
85  for(ui32index = 0 ; ui32index < NUM_UART_DATA ; ui32index++){
86   UARTCharPut(UART1_BASE, ui8DataTx[ui32index]);
87  }
88
89  while(UARTBusy(UART1_BASE)){}
90
91  UARTSend(UART0_BASE, (uint8_t *)"\n\n\rReceiving data (UART1) ",
        strlen("\n\n\rReceiving data (UART1) "));
92
93  for(ui32index = 0 ; ui32index < NUM_UART_DATA ; ui32index++){
94   ui8DataRx[ui32index] = UARTCharGet(UART1_BASE);
95  }
96
97  //Piscando LED azul
98  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
99  SysCtlDelay(SysCtlClockGet() / 2); //delay
100 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
101 }
102
103 void UARTSend(uint32_t ui32UARTBase, const uint8_t *pui8Buffer,
        uint32_t ui32Count){
104  while(ui32Count--){
105   UARTCharPut(ui32UARTBase, *pui8Buffer++);
106  }
```

```
107 }
108
109 void Switch1IntHandler(void){
110  GPIOIntClear(GPIO_PORTF_BASE, GPIO_INT_PIN_4);
111
112  //Piscando LED vermelho
113  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
114  SysCtlDelay(SysCtlClockGet() / 5); //delay
115  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0);
116
117  UARTSend(UART0_BASE, (uint8_t *)"\n\rSending (UART0): ", strlen("\n\
     rSending (UART0): "));
118
119  UARTSend(UART0_BASE, (uint8_t*)ui8DataRx, NUM_UART_DATA);
120
121  //Piscando o LED verde
122  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);
123  SysCtlDelay(SysCtlClockGet() / 5); //delay
124  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0);
125  SysCtlDelay(SysCtlClockGet() / 2.5); //delay
126 }
```

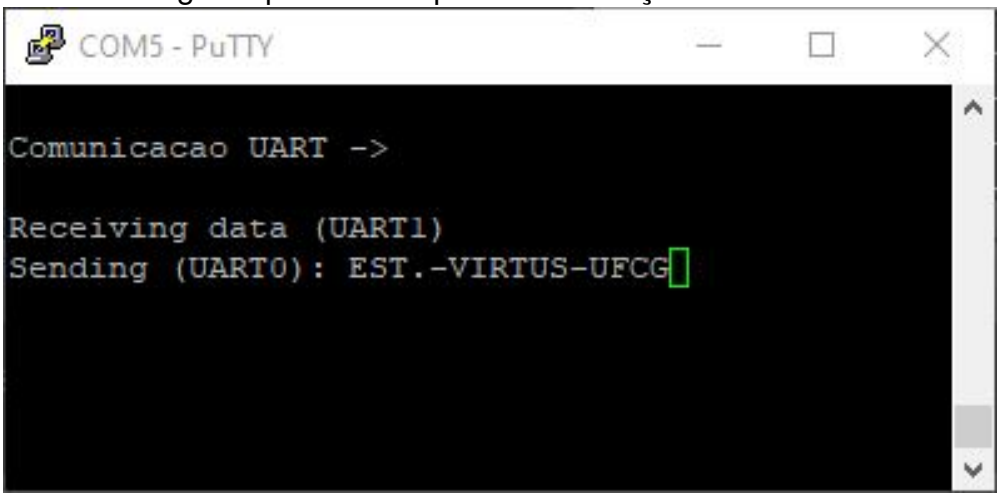
Algoritmo 5: Uso de UART e interupções para transmissão e recepção de dados

Os 16 bytes a serem enviados estão nas linhas 76 até a 83. Vale destacar que ao chamar cada função de interrupção, o primeiro comando a ser executado (linhas 70 e 110), serve para limpar a interrupção chamada e prepará-la para a próxima execução. A ativação dos diodos emissores de luz foram inseridas nas respectivas funções de interrupção, e estão indicadas por comentário no código, nas linhas 97, 112 e 121.

Ao carregar o Algoritmo 5 na placa, observamos no terminal do *software* PuTTY, que mostra o conteúdo da porta serial na Figura 15, primeiro a mensagem ”Comunicacao UART ->”, ocorrendo o envio e recebimento contínuo dos bytes na UART1, indicado pela mensagem: ”Receiving data (UART1)”. No momento em que o nível lógico do botão da Chave SW1 passa para baixo, há o envio dos dados pela UART0 e assim a mensagem ”Sending (UART0): EST.-VIRTUS-UFCG” é também mostrada no terminal.

Logo, podemos perceber que, cada caractere enviado pela UART possui 8 bits (1 byte), que são justamente os bits de dados no pacote (*data frame*) mostrado na Figura 3. Portanto, 16 caracteres foram enviados e recebidos via UART com sucesso utilizando essa mesma configuração de pacote.

Figura 15: Mensagem apresentada pela comunicação UART visualizada na serial




Fonte: Autoria Própria

Neste capítulo foram apresentadas as atividades realizadas durante o estágio supervisionado conforme descrito acima. No próximo capítulo, serão feitas as considerações finais relativas a relação entre os conteúdos estudados durante a graduação com o que foi desenvolvido no estágio, as habilidades desenvolvidas durante a realização das tarefas e o que possibilitou um bom desenvolvimento profissional para o estagiário.

# 5 CONSIDERAÇÕES FINAIS

Neste relatório foram descritas as principais atividades desenvolvidas em um projeto de Pesquisa, Desenvolvimento e Inovação, vinculado a UFCG pela disciplina de Estágio Supervisionado obrigatório no Núcleo VIRTUS, em cooperação com o Laboratório de Sistemas Embarcados e Computação Pervasiva (Embedded). O estagiário Marley Lobão de Sousa cumpriu a carga horária de 291 horas, ultrapassando a carga horária mínima para o estágio supervisionado matriculado (240 horas).

O estagiário utilizou conhecimentos que estiveram relacionados as disciplinas de introdução à programação, pelo uso da linguagem C para os códigos direcionados a sistemas embarcados, arquitetura de sistemas digitais, para entendimento da arquitetura e periféricos do microcontrolador e interação com a linguagem *Assembly*, e informática industrial pelo uso do conhecimento de cerimônias e atores de metodologias ágeis e boas práticas na engenharia de *software*.

Nas tarefas realizadas, o aluno esteve em uma equipe de que utilizou a metodologia de desenvolvimento ágil Scrum, realizou o estudo dirigido para arquitetura e comunicação em sistemas embarcados, investigou soluções que envolveram controle e monitoramento de entradas e saídas no microcontrolador, realizou a validação e experimentação dos códigos desenvolvidos, instalou e aprendeu a configurar e utilizar um ambiente de desenvolvimento direcionado para sistemas embarcados.

Por conta da pandemia, não foi possível realizar os testes com outro *hardware* acoplado, ou em conjunto com outro sistema. Mesmo com essas limitações, o estagiário conseguiu realizar experimentos e validações que ajudaram a visualizar os conceitos vistos na graduação, bem como a participar de uma equipe de trabalho e ver de perto o processo de desenvolvimento, além de conhecer a arquitetura ARM, acionamento de GPIOs, conceitos e utilização da comunicação serial UART.

Dessa maneira, foi possível colocar em prática diversos conceitos de engenharia elétrica adquiridos no curso de graduação e desenvolver o trabalho em equipe, planejamento de atividades, produção de relatórios técnicos com as tarefas realizadas, de forma que o estágio colaborou na aplicação e descoberta de novas tecnologias, aumentando as habilidades e complementando a formação do aluno.

# 6 REFERÊNCIAS


VIRTUS. Núcleo VIRTUS - UFCG.
Disponível em: https://www.virtus.ufcg.edu.br/, 2021. Online. Acesso em: Janeiro de 2022. Citado na página 2.

LIEC. Laboratório de Instrumentação Eletrônica e Controle.
Disponível em: http://liec.ufcg.edu.br/Fotos/, 2020. Online. Acesso em: Janeiro de 2022. Citado na página 3.

Embedded Lab. Laboratório de Sistemas Embarcados e Computação Pervasiva.
Disponível em: https://www.embedded.ufcg.edu.br/, 2019. Online. Acesso em: Março de 2022. Citado na página 3.

J. Sutherland and K. Schwaber. The Scrum Guide.
Disponível em: https://scrumguides.org/, 2020. Online. Acesso em: Janeiro de 2022. Citado na página 5.

E. Andrade. História da Computação: Um Pouco de Assembly. Jornal do Programa de Educação Tutorial de Ciências da Computação - UFCG.
Disponível em: www.dsc.ufcg.edu.br/~pet/jornal/maio2014/materias/historia_da_computacao.html, 2014. Online. Acesso em: Janeiro de 2022. Citado na página 6.

B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall Software Series, first edition, 1978. Citado na página 6.

Y. Zhu. *Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C*. E-Man Press LLC, third edition, 2018. ISBN 978-0-9826926-6-0. Citado na página 7.

E. Peña. UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter. Analog Devices.
Disponível em: https://www.analog.com/media/en/analog-dialogue/volume-54/number-4/uart-a-hardware-communication-protocol.pdf, 2020. Online. Acesso em: Fevereiro de 2022. Citado na página 8.

K. C. S. Murti. *Design Principles for Embedded Systems*. Springer - Transactions on Computer Systems and Networks, 2022. Citado na página 9.

A. Barkalov, L. Titarenko, and M. Mazurkiewicz. *Foundations of Embedded Systems*. Springer - Studies in Systems, Decision and Control 195, 2019. Citado na página 9.

ARM. 30 Years of ARM Innovation.
Disponível em: https://www.arm.com/company, 2022. Online. Acesso em: Março de 2022. Citado na página 10.

Texas Instruments. ARM® Cortex®-M4F Based MCU TM4C123G LaunchPad™ Evaluation Kit - Technical documentation: Data sheet.
Disponível em: https://www.ti.com/lit/pdf/spms376, 2014. Online. Acesso em: Fevereiro de 2022. Citado nas páginas 10, 11, 13 e 14.

Texas Instruments. ARM® Cortex®-M4F Based MCU TM4C123G LaunchPad™ Evaluation Kit - Technical documentation: User Guide.
Disponível em: https://www.ti.com/lit/pdf/spmu296, 2020. Online. Acesso em: Fevereiro de 2022. Citado nas páginas 11 e 29.

Texas Instruments. CCSTUDIO: Code Composer Studio™ integrated development environment (IDE).
Disponível em: https://www.ti.com/tool/CCSTUDIO, 2021. Online. Acesso em: Fevereiro de 2022. Citado na página 26.