DISCIPLINA EA801

Laboratório de Projetos de Sistemas Embarcados

Metodologia de Projeto e Ambiente de Desenvolvimento de *Software*

Profs. Fabiano Fruett, Antonio Augusto Fasolo Quevedo e Wu Shin-Ting

FEEC / UNICAMP

Editado em julho de 2025 com suporte de Chatgpt, Gemini e NotebookLM

INTRODUÇÃO

Os sistemas embarcados são a base tecnológica de inúmeras inovações que moldam nosso dia a dia, desde *smartphones* e carros autônomos até dispositivos médicos avançados e a automação industrial. Caracterizados por executar funções específicas com alta confiabilidade e, muitas vezes,

em tempo real, esses sistemas operam sob restrições de *hardware*, demandando integração precisa de sensores e atuadores, comunicação eficaz com o ambiente físico e um gerenciamento rigoroso de energia e desempenho. A confiabilidade é primordial, pois falhas podem ter sérias consequências, e o baixo consumo de energia frequentemente possibilita mobilidade e operação por bateria.

Este roteiro serve como um guia prático para a metodologia de projetos em sistemas embarcados, apresentando diferentes tecnologias disponíveis. Nosso foco principal será em microcontroladores, explorando sua arquitetura interna, as melhores práticas de programação, a importância da documentação técnica e a eficiência do versionamento de código.

DESAFIO

O objetivo é desenvolver um sistema embarcado funcional, utilizando o *kit* BitDogLab (BDL) e sua placa RP2040 como base. A equipe deverá identificar um problema (real ou hipotético) e propor uma solução focada em microcontroladores. Em seguida, será preciso realizar uma análise técnica para selecionar os periféricos mais adequados disponíveis no *kit* BDL, implementando **um protótipo em Python** usando a Thonny IDE.

Os recursos disponíveis são:

- *Kit* de Desenvolvimento BDL (com placa RP2040 e diversos periféricos).
- Documentação técnica do microcontrolador (*datasheet*, Raspberry Pi Pico-series Python SDK) e dos periféricos (*datasheets*).
- Ambientes de desenvolvimento integrados (IDEs) e *toolchains* relevantes: IDE Thonny para RP2040.
- Exemplos de programação em Python para os *firmwares* dos componentes do BitDogLab.
- Apoio do professor para dúvidas técnicas e metodológicas.

O projeto será dividido nas seguintes fases:

1. Fase de Conceituação e Requisitos
    - Definição do Problema: A equipe deverá identificar e descrever um problema que possa ser solucionado por um sistema embarcado. O problema deve ser claro e permitir a aplicação dos conceitos estudados.
    - Levantamento de Requisitos: Com base no problema, a equipe definirá os requisitos funcionais (o que o sistema deve fazer) e requisitos não-funcionais (desempenho, custo, tamanho, consumo de energia, etc.).
    - Justificativa Técnica: Elaborar uma breve justificativa da relevância e viabilidade técnica da solução proposta.
2. Fase de Análise Técnica e Seleção de *Hardware*:
    - Seleção da Placa de Desenvolvimento:
        - Análise Comparativa: Avaliar tecnicamente as placas RP2040 (Raspberry Pi Pico/RP2040-based boards) e Black Pill (STM32F4xx) disponíveis, considerando:
            - Arquitetura do Microcontrolador: Cortex-M0+ (RP2040) vs. Cortex-M4 (Black Pill).
            - Recursos de *Hardware*: Velocidade de *clock*, quantidade de memória Flash e RAM, número de GPIOs, tipos de periféricos internos (UARTs, SPIs, I2Cs, ADCs, *Timers*, PWMs), presença de FPU (do inglês *Floating Point Unit*).
            - Capacidades Específicas: PIO (RP2040) vs. DMA avançado (Black Pill), desempenho para cálculos específicos.

        - Ambiente de Desenvolvimento/*Toolchain*: Facilidade de uso com IDEs e compiladores.
        - Consumo de Energia: Comparar as eficiências para a aplicação proposta.
        - Custo-benefício em relação aos requisitos do projeto.
    - Decisão Justificada: Apresentar a escolha da placa RP2040 com uma justificativa baseada nas características técnicas levantadas e nos requisitos do projeto.
- Seleção de Periféricos do *Kit* BDL:
    - Com base nos requisitos definidos, a equipe deverá identificar e selecionar os periféricos necessários do *kit* BDL.
    - Justificativa para cada periférico: Explicar por que cada periférico selecionado é o mais adequado tecnicamente para cumprir sua função no sistema, considerando suas especificações (precisão do sensor, capacidade de corrente, tipo de interface de comunicação, etc.).

3. Fase de Implementação e Testes
    - Desenvolvimento de *Software***:** Implementar o *firmware* do sistema no microcontrolador selecionado, utilizando a linguagem MicroPython. O código deve ser bem estruturado, modularizado e utilize a sintaxe Doxygen para a documentação dos programas Python.
    - Integração *Hardware-Software***:** Acoplar a placa de desenvolvimento no *kit* BDL e realizar a integração física e lógica.
    - Testes**:** Realizar testes de unidade, integração e validação do sistema para garantir que todos os requisitos funcionais e não-funcionais sejam atendidos. Isso abrange testar a leitura de sensores, o acionamento de atuadores e a lógica de controle, quando aplicável.
4. Entregas do Projeto
    - Documento de Projeto (Relatório no arquivo README.md do repositório do projeto no GitLab seguindo o modelo, e o *link* do projeto no Moodle).
    - Código-Fonte:
        - Código completo do *firmware*, organizado e comentado, no repositório de controle de versão do GitLab da Unicamp/GitHub.
    - Demonstração do Protótipo:
        - Apresentação do sistema funcionando no *kit* BDL na aula do dia de entrega do projeto.

Avaliação

O projeto será avaliado com base na:

- qualidade da análise técnica e justificativa das escolhas de *hardware*.
- funcionalidade da solução implementada.
- qualidade do código (estrutura, clareza, documentação).
- capacidade de aplicar os conceitos de sistemas embarcados.
- organização e clareza do relatório final e da apresentação.

FUNDAMENTOS TEÓRICOS

Esta seção apresenta os fundamentos teóricos essenciais para o projeto de sistemas embarcados, com ênfase no processo de desenvolvimento de *software*, com o objetivo de fornecer uma base para compreender os aspectos teóricos e práticos do desenvolvimento de *software* em sistemas embarcados modernos. Iniciamos com a metodologia de projeto, abordando as etapas típicas de especificação, modelagem, implementação e testes. Em seguida, exploramos as características dos

sistemas embarcados baseados em microcontroladores, destacando seu papel em aplicações de controle, automação, instrumentação e dispositivos inteligentes.

A seção também contempla a arquitetura interna dos microcontroladores, descrevendo seus principais blocos funcionais, como a CPU, memórias, barramentos, registradores e periféricos internos, e sua relação com o *software* embarcado. Por fim, discutimos os principais conceitos de programação embarcada, incluindo linguagens utilizadas, manipulação direta de registradores, uso de interrupções, técnicas de controle de fluxo e interação com periféricos.

Metodologia de projeto

A maior parte do conteúdo desta seção foi sintetizada do artigo elaborado pelo Prof. Cugnasca.

O desenvolvimento de sistemas embarcados tornou-se uma área essencial da engenharia moderna. Impulsionado pela miniaturização, aumento de capacidade e redução de custos dos eletrônicos, esses sistemas predominam em novas soluções, de eletrodomésticos inteligentes a veículos autônomos, integrando interfaces, dispositivos eletromecânicos, sensores e atuadores. A crescente demanda exige metodologias de projeto adequadas para garantir que os produtos atendam aos requisitos.

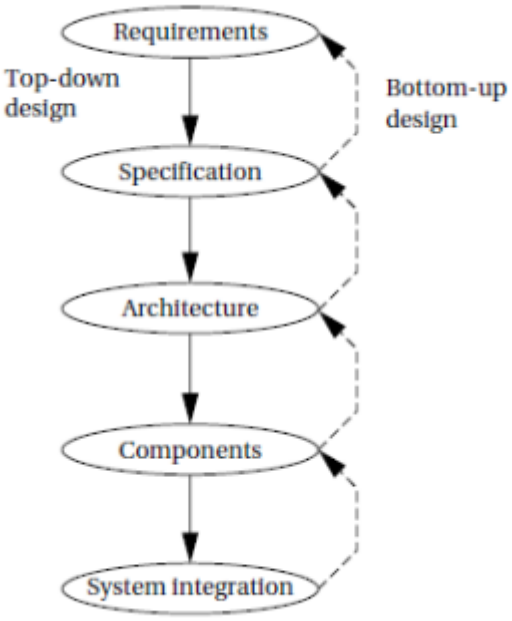
Sistemas embarcados se diferenciam de computadores de uso geral por suas características distintas, sendo projetados, muitas vezes com recursos de *hardware* limitados, para:

- Funções Dedicadas: Projetados para uma ou poucas funções específicas, configuráveis, mas não alteráveis pelo usuário final.
- Interação Física: Frequentemente se conectam ao ambiente via sensores e/ou atuadores.
- Robustez e Confiabilidade: Essenciais para resistir a condições adversas de uso e ambiente. Uma falha pode ter alto impacto, exigindo alta disponibilidade e capacidade de recuperação automática.
- Eficiência de Recursos: Geralmente operam com memória e velocidade de processamento limitadas, buscando minimizar componentes e otimizar consumo de energia para mobilidade e bateria.
- Interface Simplificada: A interface de usuário é, muitas vezes, limitada ou inexistente (botões, LEDs).

Para atender aos requisitos complexos desses sistemas, é fundamental empregar uma metodologia de desenvolvimento adequada. Uma boa metodologia define os passos desde a concepção até a implementação, teste e validação, facilitando a documentação técnica e a evolução futura do projeto. Ela também permite o uso de ferramentas de projeto modernas, o que, por sua vez, reduz tempo e custos, além de melhorar a qualidade. Uma metodologia eficaz também favorece a modularização e o trabalho colaborativo. O modelo de projeto de sistemas embarcados apresentado pelo Prof. Cugnasca engloba as seguintes fases:

- Conceituação: Fase inicial onde se define o projeto a partir das informações do cliente, com uma descrição clara e sem ambiguidades. É essencial identificar e eliminar “condicionantes fictícias” ou requisitos irrealistas que possam comprometer o projeto.
- Requisitos: Definição dos requisitos funcionais (o que o sistema fará) e não-funcionais (como desempenho, custo, tamanho, peso e consumo de energia).

- Especificação: Formalização dos requisitos em uma linguagem sem ambiguidades, validada pelo solicitante.
- Projeto (*Design*): Descrição funcional (arquitetura) do sistema via diagramas de blocos (componentes, módulos, conexões, funções), que após refinamentos, são implementados por componentes reais.
- Implementação: Dimensionamento e aplicação dos dispositivos eletrônicos. Componentes de *software* e *hardware* devem seguir interfaces padronizadas. Testes isolados de componentes são recomendados.
- Teste: Fase crucial para a descoberta de defeitos. A complexidade da integração pode ser minimizada com metodologias e gerenciamento de projeto adequados.



Dando continuidade à importância da estruturação do projeto, diversas abordagens e modelos podem ser adotados para guiar o desenvolvimento:

- Abordagem cascata (*Waterfall*): Linear e com poucas interações, mais útil em projetos de *hardware* que utilizam componentes de alta integração.
- Modelo espiral: Incorpora detalhamento sucessivo e prototipagem, adicionando recursos gradualmente com base em testes e aprendizado, permitindo o refinamento contínuo.
- Refinamentos sucessivos: Ideal quando a especificação do problema não é totalmente clara. Envolve interação constante entre o abstrato (especificação) e o concreto (protótipo), onde um protótipo inicial é gradualmente aprimorado com novos recursos baseados em testes, até a especificação final.

Para a robustez de qualquer sistema, independentemente do modelo de desenvolvimento, certas práticas são indispensáveis. Entre elas, destacam-se os métodos formais de especificação e validação. Eles usam linguagens e técnicas matemáticas para descrever o comportamento do sistema de forma precisa e sem ambiguidades. Isso permite verificar a correção do projeto em fases iniciais, identificando falhas conceituais antes da implementação de sistemas de alta criticidade em áreas como aeroespacial, médica e automotiva, onde uma falha pode ser catastrófica. Em vez de descrições em linguagem natural, que são ambíguas, os métodos formais empregam uma notação matemática rigorosa para expressar propriedades e o comportamento esperado. Exemplos incluem VDM (do inglês *Vienna Development Method*), uma linguagem completa para descrever sistemas por funções matemáticas, e a combinação de UML (do inglês *Unified Modeling Language*) com

OCL (do inglês *Object Constraint Language*), onde a OCL adiciona precisão formal às restrições dos modelos visuais da UML.

Complementarmente, a fase de testes inclui testes de unidade, que validam o funcionamento correto de componentes individuais (sejam módulos de *software* ou circuitos de *hardware*); testes de integração, que verificam a interação entre diferentes partes do sistema quando combinadas; e testes de validação em campo. Estes últimos simulam ou ocorrem no ambiente real de uso do produto, confirmando que o sistema atende aos requisitos funcionais e não-funcionais sob condições operacionais reais, garantindo a sua robustez e confiabilidade final.

No contexto do desenvolvimento de sistemas embarcados, um conceito financeiro e estratégico de grande importância é o NRE (do inglês *Non-Recurring Engineering*), ou Engenharia Não Recorrente. NRE se refere aos custos únicos e iniciais de engenharia necessários para projetar, desenvolver, prototipar e validar um sistema embarcado ou seus componentes. São despesas que ocorrem apenas uma vez para a criação do *design* original e do processo de fabricação, não se repetindo quando o produto é fabricado em volume.

O NRE é fundamental para a estimativa do custo total de um produto, pois representa o investimento inicial para tirar uma ideia do papel e transformá-la em um produto fabricável. Ele inclui custos como:

- *Design* do *Hardware*: Esquemáticos, *layout* de PCB, seleção de componentes.
- Desenvolvimento de *Firmware*/*Software*: Programação do microcontrolador, *drivers*, otimização de código.
- Prototipagem: Fabricação de unidades de teste.
- Testes e Validação: Engenharia de testes para garantir a funcionalidade, robustez e conformidade.
- Ferramental e Moldes: Criação de ferramentas específicas para a linha de produção (se aplicável).
- Certificações: Obtenção de aprovações regulatórias.

Para calcular o custo final por unidade de um sistema embarcado, esses custos de NRE são amortizados ao longo do volume total de produção. Por exemplo, se o NRE de um projeto for $100.000 e a empresa planejar fabricar 10.000 unidades, $10 do NRE são “embutidos” no custo de cada unidade. Isso significa que produtos com baixo volume de produção terão um custo unitário proporcionalmente mais alto devido ao impacto do NRE. Entender o NRE ajuda na análise da viabilidade financeira, precificação e tomada de decisões estratégicas de um projeto de sistema embarcado.

Por fim, a escolha da tecnologia de implementação também desempenha um papel decisivo no sucesso do projeto:

- Microcontroladores: É a escolha natural para a maioria dos sistemas embarcados. Permitem implementações com poucos componentes, são bem suportados por ferramentas de desenvolvimento (compiladores, simuladores, bibliotecas), e são uma tecnologia consolidada. Oferecem escalonamento, sincronização, interrupções e baixa latência para aplicações em tempo real, além de serem compactos, de baixo consumo e baixo custo. O NRE associado tende a ser menor comparado a soluções mais complexas, dada a vasta disponibilidade de ferramentas e exemplos.
- Microprocessadores: Permitem soluções mais complexas e poderosas, com capacidade para paralelismo. No entanto, são mais caros, consomem mais energia e podem ser menos determinísticos devido ao compartilhamento de recursos, tornando-os menos ideais para aplicações críticas em tempo real. O NRE aqui pode ser mais elevado devido à

complexidade de *design* de hardware (ex: memórias externas, interfaces de alta velocidade) e *software* (sistemas operacionais completos).

- FPGAs (do inglês *Field Programmable Gate Array*): Circuitos integrados configuráveis pelo usuário via linguagens de descrição de hardware (VHDL, Verilog). Projetos com FPGAs seguem a metodologia *Hardware-Software Codesign*, com especificação e teste em alto nível. Vantagens incluem facilidade de simulação e teste, sendo uma boa alternativa para sistemas críticos, mas seu custo pode ser proibitivo para pequenos projetos, sendo mais viável para produtos de larga escala. No entanto, o NRE para projetos com FPGAs é tipicamente muito alto, devido à complexidade do *design* em nível de *hardware* e à necessidade de engenheiros especializados. Seu custo total pode ser proibitivo para pequenos projetos, sendo mais viável para produtos de larga escala onde o NRE se dilui em muitas unidades.
- ASICs (do inglês *Application-Specific Integrated Circuit*): São circuitos integrados projetados e fabricados sob medida para uma função ou conjunto de funções muito específicas. Diferentemente de microcontroladores ou microprocessadores de uso geral, são otimizados para executar exclusivamente as tarefas para as quais foram criados, alcançando o mais alto desempenho, menor consumo de energia e menor tamanho físico para aquela aplicação dedicada. No entanto, o NRE para ASICs é extremamente alto, envolvendo custos significativos de *design*, verificação, fabricação de máscaras e testes rigorosos, que podem chegar a milhões de dólares. Devido a esse investimento inicial massivo em NRE, a fabricação de ASICs só é economicamente viável para volumes de produção muito elevados, onde o NRE é diluído em um grande número de unidades, resultando em um custo unitário muito baixo por *chip*. O ciclo de desenvolvimento é longo, e erros de *design* são caríssimos de corrigir.
- Plataformas de Prototipagem (Arduino, Raspberry Pi, Bluepill, Black Pill, Nucleo): Muito usadas para prototipagem rápida e projetos não-comerciais inicialmente. Oferecem baixo custo, vasta gama de módulos, ferramentas de desenvolvimento e forte suporte de comunidades. Permitem validar um protótipo antes de otimizar para uma implementação mais econômica com placas dedicadas. Embora o custo de *desenvolvimento inicial* (o "NRE" na fase de prototipagem) usando essas plataformas seja baixo e rápido, a transição para um produto comercial escalável a partir delas pode exigir um NRE adicional significativo para a otimização de *hardware*, redução de custo unitário e certificações.

O desenvolvimento de sistemas embarcados é uma área em constante evolução que exige uma abordagem estruturada. A escolha e aplicação de metodologias de projeto adequadas e das tecnologias de *hardware* e *software* mais apropriadas são essenciais para garantir que os produtos atendam aos requisitos de qualidade, custo e prazo, otimizando o aproveitamento das potencialidades tecnológicas e dos recursos humanos envolvidos.

Sistemas embarcados baseados em microcontroladores

No cerne desses sistemas embarcados estão os microcontroladores, dispositivos versáteis que integram em uma única pastilha uma unidade de processamento, memória e diversos periféricos configuráveis por *software*.

A evolução dos microcontroladores revolucionou o projeto de sistemas embarcados. Funções antes executadas por circuitos dedicados. como conversores analógico-digitais (AD/DA), interfaces seriais (UART, SPI, I2C), temporizadores e geradores de PWM, agora são implementadas via *software*, configurando módulos internos do próprio microcontrolador. Essa flexibilidade não só reduz a complexidade do *hardware*, o custo e o tempo de desenvolvimento, mas também eleva a programação ao centro do projeto.

Contudo, essa integração intensa entre *hardware* e *software* cria novos desafios. Embora o desenvolvimento deva ser unificado, as ferramentas e conhecimentos para o *software* embarcado são bem diferentes daqueles usados no projeto de *hardware*. Durante a fase de desenvolvimento do *software* embarcado, utiliza-se um conjunto de ferramentas, tanto de *software* quanto de apoio em *hardware*, que não fazem parte do produto final, mas são fundamentais para a criação, teste e depuração do código. Entre as principais ferramentas estão:

- Ambientes de desenvolvimento integrados (em inglês, *Integrated Development Environments* – IDEs), como CodeWarrior, STM32CubeIDE, Visual Studio Code, MPLAB X, uVision4-Keil e outros;
- Compiladores e *toolchains* específicos para a arquitetura do microcontrolador;
- Depuradores e simuladores para testes controlados e análise de comportamento;
- Placas de desenvolvimento e *kits* de avaliação, usados para prototipagem rápida com microcontroladores reais;
- Gravadores e depuradores físicos (padrão de teste e depuração JTAG ou protocolo SWD), que permitem a gravação do *firmware* e o rastreamento dos sinais em tempo real;
- Sistemas de versionamento de *software* (como Git ou SVN), para rastrear mudanças no código, facilitar a colaboração em equipe e gerenciar diferentes versões do projeto com segurança;
- Osciloscópios, analisadores lógicos e geradores de sinais, para inspeção de sinais elétricos e depuração de interfaces;
- Fontes de alimentação programáveis e multímetros, para avaliação das condições elétricas do sistema.

Essas ferramentas são empregadas principalmente nas fases iniciais do projeto, durante a prototipagem e o desenvolvimento funcional do *software*.

Quando o objetivo é levar o sistema embarcado ao mercado como um produto comercial finalizado, é necessário o desenvolvimento de um *hardware* dedicado. Nessa etapa, são projetados e implementados circuitos sob medida, considerando as especificações de uso, custo, desempenho, consumo e confiabilidade. Para isso, utilizam-se ferramentas especializadas de projeto eletrônico, como:

- *Softwares* de captura esquemática e *layout* de PCB (do inglês *Printed Circuit Board*), como Altium *Designer*, KiCad, Eagle, OrCAD e EasyEDA;
- Simuladores de circuitos analógicos e digitais, como LTspice, Proteus e Multisim, usados para validar funcionalidade antes da produção;
- *Softwares* de modelagem e simulação térmica e eletromagnética, para garantir a integridade do sinal e dissipação de calor;
- Ferramentas de geração de arquivos de fabricação (Gerber) e visualização 3D do circuito impresso;
- Equipamentos de bancada para testes do protótipo físico, como estações de solda, microscópios, câmeras térmicas e analisadores de potência.

Essas ferramentas possibilitam a criação de placas de circuito impresso com circuitos de interface, fontes de alimentação, circuitos de *clock*, sistemas de proteção, entre outros componentes essenciais que compõem o produto final. O desenvolvimento dessa etapa exige conhecimento em eletrônica

analógica e digital, além de integração com requisitos mecânicos, térmicos e normativos. A seleção e o uso corretos dessas ferramentas de desenvolvimento são fundamentais para garantir que o sistema embarcado seja funcional, confiável e eficiente.

Arquitetura de microcontroladores

A arquitetura de um microcontrolador (em inglês, *microcontroller unit* – MCU) é a base sobre a qual seus componentes interagem para processar dados e controlar sistemas. Compreender essa estrutura interna permite não apenas a programação eficiente de sistemas embarcados, mas também a otimização do código e o aproveitamento máximo dos recursos, resultando em sistemas mais eficientes e robustos. O cérebro do microcontrolador é a Unidade Central de Processamento (CPU), ou o núcleo. Ela é responsável por decodificar as instruções recebidas da memória, executá-las (realizando cálculos, transferências de dados, etc.) e controlar o fluxo das operações, determinando o próximo passo com base na programação.

A CPU se comunica com os demais módulos do microcontrolador, incluindo as memórias, através de barramentos. Os microcontroladores utilizam diferentes tipos de memória, cada uma com funções específicas. A RAM (do inglês *Random Access Memory*) armazena dados temporários durante a execução do programa, como variáveis e a pilha, perdendo seus dados quando o microcontrolador é desligado (volátil). A ROM (do inglês *Read-Only Memory*)/Flash guarda o código do programa que a CPU executa, sendo não volátil e mantendo os dados sem energia; a memória Flash, em particular, permite regravação para atualização de *firmware*. Há também a EEPROM (do inglês *Electrically Erasable Programmable Read-Only Memory*), uma memória não volátil que permite leitura, gravação e apagamento elétrico, ideal para armazenar configurações ou dados críticos que não podem ser perdidos.

Para a comunicação interna entre a CPU, as memórias e os diversos periféricos, os microcontroladores contam com um sistema de barramentos. Tradicionalmente, distinguimos três tipos principais: o barramento de dados, que transporta as informações a serem processadas; o barramento de endereços, que especifica a localização exata dos dados na memória ou dos periféricos a serem acessados; e o barramento de controle, que gerencia o fluxo de dados e sinaliza as operações de forma ordenada e sincronizada. No entanto, em microcontroladores mais modernos e de alto desempenho, a crescente complexidade e a necessidade de lidar com múltiplos dispositivos que operam em velocidades muito diferentes levaram à evolução desses sistemas de comunicação. Em vez de um único conjunto compartilhado de barramentos, é comum encontrar arquiteturas de barramento mais avançadas, como as matrizes de barramentos (em inglês, *bus matrix*) ou barramentos multicamadas (em inglês, *multi-layer bus*).

Essas arquiteturas permitem que múltiplos “mestres” (como a CPU, controladores DMA ou alguns periféricos avançados) acessem simultaneamente diferentes “escravos” (como módulos de memória Flash, SRAM, ou periféricos de alta velocidade) através de barramentos dedicados ou caminhos paralelos. Com uma matriz de barramentos, a CPU pode estar acessando a memória Flash enquanto um controlador DMA (do inglês *Direct Memory Access*) transfere dados entre um periférico e a SRAM, tudo de forma concorrente. Essa abordagem de múltiplos barramentos e interconexões complexas assegura que dispositivos de diferentes velocidades possam operar em sua máxima eficiência, otimizando o fluxo de dados e o desempenho geral do microcontrolador em processamento paralelo.

Um microcontrolador se destaca pela integração com uma variedade de periféricos internos que expandem suas funcionalidades e permitem a interação com o mundo externo. Entre os mais comuns, temos os GPIO (do inglês *General Purpose Input/Output*) para controle de pinos digitais, conversores AD/DA (analógico-digital e digital-analógico) para interagir com sinais do mundo real, Temporizadores (em inglês, *timers*) e PWM para gerar sinais temporizados e controlar dispositivos como motores e iluminação, e diversas opções de comunicação serial como UART, SPI, I2C e CAN. Essa capacidade se estende à troca de dados com periféricos externos ou outros sistemas, utilizando protocolos seriais síncronos (SPI e I2C) e assíncronos (UART), redes embarcadas (CAN, LIN, RS-485) e protocolos industriais e embarcados (Modbus, I2S, USB, BLE, entre outros).

Internamente à CPU, os registradores são pequenas áreas de memória que armazenam dados temporários, proporcionando acesso rápido e eficiente durante o processamento. A pilha (em inglês, *stack*) é uma área especial da memória RAM usada para guardar dados temporários, como variáveis locais e endereços de retorno de funções, crescendo e encolhendo conforme o fluxo do programa. Os microcontroladores operam em diferentes modos de operação, como o modo de usuário para execução normal do código e o modo de interrupção para lidar com eventos de alta prioridade, cada um com suas próprias permissões e características de acesso.

Finalmente, a capacidade de um microcontrolador é definida por seu repertório de instruções, que é o conjunto de comandos que a CPU pode entender e executar. Esse repertório inclui operações aritméticas, lógicas, de controle de fluxo e manipulação de dados. Cada instrução requer um ou mais ciclos de máquina para ser executada, sendo o ciclo de máquina o tempo básico que a CPU leva para completar uma operação fundamental. O número de ciclos necessários depende da complexidade da instrução e da arquitetura específica do microcontrolador.

Um exemplo de arquitetura é a arquitetura ARM (do inglês *Advanced RISC Machine*), amplamente utilizada em microcontroladores modernos. A ARM segue uma arquitetura *Load-Store*, que a distingue de arquiteturas mais antigas. Nessa arquitetura, a CPU só pode operar dados que estão em seus registradores de trabalho. As operações de memória (leitura e escrita) são feitas por instruções específicas de carga (em inglês, *load*) para trazer dados da memória para os registradores, e de armazenamento (em inglês, *store*) para mover dados dos registradores para a memória. Todas as outras operações (aritméticas, lógicas, etc.) são realizadas exclusivamente nos dados contidos nos registradores.

Essa abordagem *load-store* contribui significativamente para a previsibilidade do número de ciclos de *clock* para cada instrução. Como as operações de processamento são desacopladas do acesso à memória (que pode ser mais lento e variável), os tempos de execução das instruções aritméticas e lógicas nos registradores são geralmente fixos e conhecidos. Isso é crucial para aplicações de tempo real, onde a previsibilidade e o determinismo são essenciais. Ao eliminar a necessidade de operações complexas que acessam a memória diretamente durante o processamento, a arquitetura ARM permite que os desenvolvedores calculem com maior precisão o tempo que uma determinada sequência de código levará para ser executada, otimizando o desempenho e garantindo o cumprimento de prazos rigorosos.

Programação embarcada

Para que um microcontrolador opere corretamente, o desenvolvedor precisa dominar a configuração dos periféricos e sua interligação com o sistema, coordenando seus sinais via *software*. É assim que sensores são lidos, atuadores acionados e a lógica do sistema implementada. A programação embarcada exige uma abordagem específica, pois está diretamente ligada ao *hardware* e precisa atender a requisitos de tempo real e eficiência de recursos. O *software* em sistemas embarcados é uma área altamente especializada, demandando conhecimento em linguagens de baixo nível, manipulação direta de *hardware* e técnicas de controle de tempo e eventos. As linguagens mais comuns para o desenvolvimento de *software* embarcado são:

- **C**: É a linguagem predominante na programação de microcontroladores devido ao seu excelente equilíbrio entre controle de *hardware* e portabilidade. Permite manipulação direta de registradores e consumo mínimo de memória e processamento.
- **C++**: Também muito utilizada, especialmente quando o projeto exige abstrações mais complexas, como classes, encapsulamento e reutilização de código. Ainda assim, o uso de C++ em sistemas embarcados costuma ser restrito a subconjuntos da linguagem, com cuidado especial para não comprometer o desempenho ou aumentar o uso de memória.
- ***Assembly***: Usada em casos onde o controle fino do *hardware* e a otimização extrema são necessárias, como em trechos críticos de tempo ou de inicialização. No entanto, devido à sua baixa legibilidade e manutenção difícil, seu uso hoje é limitado a partes específicas do sistema.

Nos últimos anos, microcontroladores modernos começaram a oferecer suporte a linguagens de alto nível como **Python**, principalmente através de ambientes como MicroPython e CircuitPython. Essa abordagem se mostra bastante útil para prototipagem rápida, propósitos educacionais e o desenvolvimento de aplicações de baixa complexidade. No entanto, é importante entender que o uso de Python em ambientes embarcados vem com restrições importantes. Seu desempenho é inerentemente limitado, pois, sendo uma linguagem interpretada, Python acarreta uma sobrecarga considerável quando comparado ao código compilado em C. Isso ocorre porque, em C, o código é diretamente compilado para as instruções de máquina específicas da arquitetura do microcontrolador. Todas as instruções necessárias já estão integradas no código executável final, pronto para ser lido e processado diretamente pelo *hardware*.

Já em Python, o processo é diferente. Seu código-fonte não é convertido diretamente para instruções de máquina. Em vez disso, ele é transformado em um formato intermediário chamado *bytecode*. Para que o microcontrolador possa executar esse *bytecode*, ele precisa primeiro carregar um *software* interpretador (como carregar o MicroPython no Raspberry PI Pico) na sua memória. É esse interpretador que, em tempo de execução, lê linha por linha do *bytecode* e o traduz para as instruções de máquina correspondentes, que então são executadas pelo processador. Python atua como um "sistema operacional" simplificado que gerencia a interação com o *hardware* para o desenvolvedor. Esse processo de "tradução em tempo real" pelo interpretador é o que gera a sobrecarga. Cada instrução Python requer etapas adicionais de interpretação antes de ser executada, o que consome mais ciclos de processamento e, consequentemente, torna a execução mais lenta e menos eficiente em termos de uso de recursos quando comparada a um programa C compilado diretamente para o *hardware*.

Além disso, o controle de tempo crítico é deficiente, uma vez que Python não oferece o acesso granular e determinístico ao *hardware* que é essencial para aplicações de tempo real. Há também limitações em relação a interrupções e sinais paralelos; muitos microcontroladores com suporte a Python ainda lutam para processar interrupções com latência mínima ou sinais em paralelo de forma adequada, como é exigido na manipulação de barramentos SPI em alta velocidade ou na geração precisa de sinais PWM. Por fim, o consumo de memória é um fator determinante: Python demanda um espaço significativamente maior de memória RAM e Flash, o que pode inviabilizar seu uso em microcontroladores mais modestos, que dispõem de recursos limitados. Por todas essas razões, C e C++ continuam sendo as linguagens predominantes e preferenciais no desenvolvimento de sistemas embarcados, especialmente para aplicações industriais e de missão crítica, onde desempenho, determinismo e eficiência de recursos são inegociáveis.

Uma característica singular da programação embarcada é o acesso direto aos registradores de *hardware* do microcontrolador. É por meio desses registradores que o *software* pode configurar periféricos internos, controlar pinos de entrada/saída (GPIOs), iniciar conversões analógicas ou configurar temporizadores, entre muitas outras tarefas. Essa manipulação, frequentemente realizada via endereços de memória mapeados, exige um conhecimento detalhado do *datasheet* do microcontrolador. Além disso, a programação embarcada lida constantemente com eventos assíncronos, tanto externos (como um botão sendo pressionado ou a chegada de dados em uma porta serial) quanto internos (como o estouro de um temporizador). A maioria dos "eventos" no sentido de ações discretas, como "aconteceu algo", são representados por transições entre níveis (bordas). No entanto, a manutenção de um nível de sinal também é utilizada para indicar estados, condições ou para permitir que outras operações ocorram, podendo, nesse contexto mais amplo, ser interpretada como parte do sistema de eventos ou disparos (em inglês, *triggers*) para certas lógicas.

Para reagir rapidamente a eventos externos ou internos, interrupções e exceções são mecanismos fundamentais em sistemas embarcados. Exceções ocorrem de forma síncrona com os sinais de relógio e têm seu tratamento pré-definido pelo processador, exigindo pouca ou nenhuma intervenção direta do programador. Já as interrupções são geradas de forma assíncrona, normalmente por periféricos ou eventos externos, e permitem que o processador suspenda temporariamente o fluxo principal do programa para atender a esses eventos. Para utilizar as interrupções corretamente, o programador deve configurar os vetores de interrupção, identificar suas fontes e implementar as rotinas de tratamento correspondentes, conhecidas como ISRs (do inglês *Interrupt Service Routines*).

Contudo, nem todas as aplicações exigem resposta imediata a eventos. Em sistemas nos quais o tempo de resposta não é crítico, ou quando se busca uma estrutura de controle mais previsível, é comum empregar abordagens baseadas em laços principais com *polling* ou máquinas de estados. Nessas estratégias, o programa verifica periodicamente o estado dos dispositivos e executa ações conforme necessário, sem depender exclusivamente de interrupções. Essas abordagens são úteis para manter o controle total do fluxo do programa, reduzindo complexidade e facilitando a depuração, especialmente em sistemas embarcados com recursos limitados.

Independentemente da técnica utilizada para controle de eventos, o *software* embarcado pode ser desenvolvido em duas arquiteturas principais: *bare-metal* ou com o uso de um RTOS (do inglês *Real-Time Operating System*).

Na abordagem *bare-metal*, o código é executado diretamente sobre o *hardware*, sem camada intermediária de sistema operacional. Isso oferece alto controle, baixo *overhead* e previsibilidade temporal, características fundamentais em aplicações com requisitos rígidos de tempo real. No entanto, programar em *bare-metal* pode ser desafiador em termos de escalabilidade e manutenção, principalmente à medida que o sistema cresce em complexidade.

Para facilitar o desenvolvimento *bare-metal* e tornar o código mais modular e reutilizável, é comum adotar níveis de abstração, mesmo sem um sistema operacional. Ferramentas como CMSIS (do inglês *Common Microcontroller Software Interface Standard*), um padrão criado pela ARM para microcontroladores baseados na arquitetura Cortex-M, define uma camada de abstração que simplifica significativamente o desenvolvimento de *software*. Além disso, HALs (do inglês *Hardware Abstraction Layers*), bibliotecas de periféricos fornecidas pelos fabricantes (como as da STMicroelectronics ou Microchip) e geradores automáticos de código (por exemplo, STM32CubeMX, MPLAB X MCC ou Atmel START) permitem encapsular o acesso direto ao *hardware*. Com isso, o programador pode concentrar-se mais na lógica da aplicação do que nos detalhes de configuração de registradores, aumentando a produtividade e reduzindo a incidência de erros.

Já em sistemas mais complexos, que exigem gerenciamento de múltiplas tarefas concorrentes, sincronização entre processos ou suporte a múltiplos periféricos com diferentes prioridades, o uso de um RTOS se torna mais adequado. Um RTOS fornece mecanismos como escalonamento de tarefas, semáforos, temporizadores e filas de mensagens, permitindo uma estrutura de *software* mais organizada e escalável. Entre os RTOS mais populares estão o FreeRTOS, Zephyr e embOS, amplamente utilizados em aplicações industriais, médicas e de consumo.

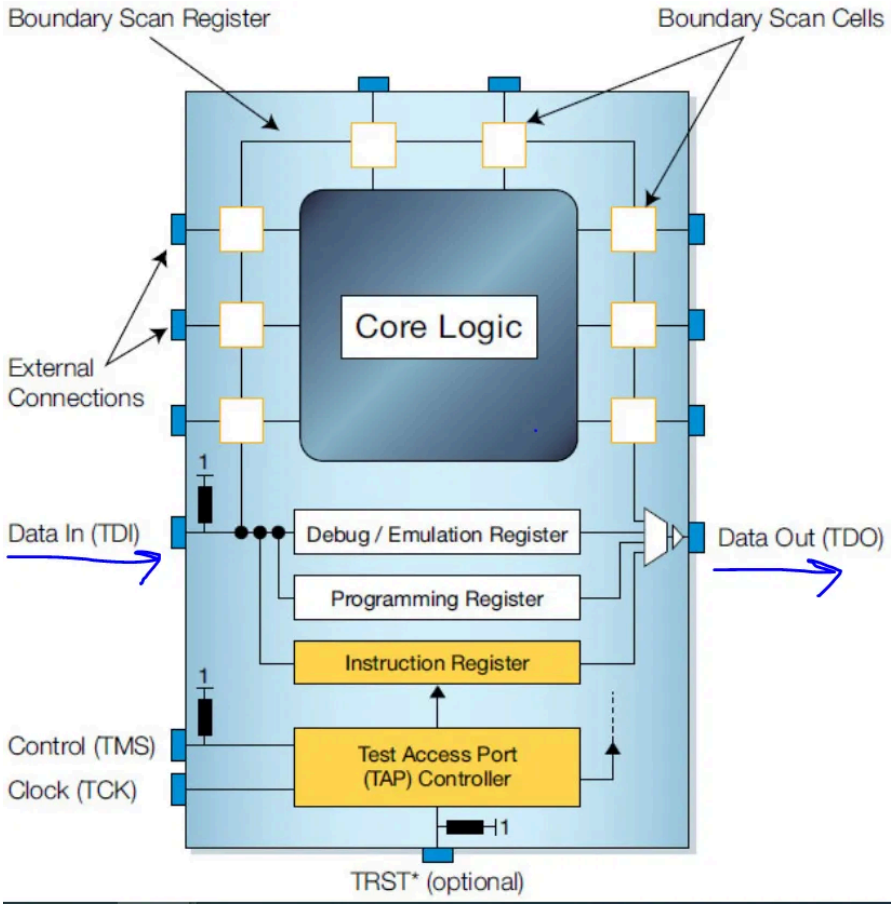
A escolha entre *bare-metal* e RTOS deve considerar a complexidade do sistema, os requisitos de tempo real, a necessidade de modularidade e a expectativa de manutenção futura. Sistemas simples com funções bem definidas podem operar de forma eficiente em *bare-metal*, com ou sem abstrações. Já projetos mais dinâmicos, com múltiplas interações simultâneas ou necessidades de atualização recorrente, tendem a se beneficiar de uma arquitetura baseada em RTOS.

Depuração embarcada

Em desenvolvimento de projetos de sistemas embarcados, a depuração do *hardware* e *software* é fundamental para garantir o correto funcionamento do produto final. Depuração se refere ao processo de identificar, analisar e corrigir erros ou falhas no código ou no funcionamento do sistema durante o desenvolvimento. Para isso, utilizam-se protocolos específicos que permitem o acesso direto aos microcontroladores, possibilitando a programação, testes e monitoramento durante o desenvolvimento. Dois dos protocolos mais importantes e amplamente utilizados para essa finalidade são o JTAG e o SWD.

O JTAG (do inglês *Joint Test Action Group*) é um padrão consolidado que surgiu inicialmente para facilitar o teste de placas eletrônicas por meio do chamado *boundary-scan*. *Boundary-scan* é uma técnica que permite testar as conexões entre componentes em uma placa eletrônica sem a necessidade de acesso físico direto a cada ponto, utilizando células de teste posicionadas ao redor dos “limites” (em inglês, *boundary*) de cada circuito integrado. Essas células formam uma cadeia que pode ser acionada via JTAG para verificar se as ligações estão corretas. Com o tempo, esse protocolo passou a ser também utilizado para programação e depuração de microcontroladores e microprocessadores. Ele funciona através de múltiplos pinos, normalmente entre quatro e cinco

linhas principais, que cumprem funções específicas na comunicação com o dispositivo: o TCK (do inglês *Test Clock*) fornece o sinal de relógio para sincronizar a troca de dados; o TMS (do inglês *Test Mode Select*) controla o estado da máquina de estados do protocolo; o TDI (do inglês *Test Data In*) é a linha pela qual os dados são enviados para o dispositivo; o TDO (do inglês *Test Data Out*) é a linha pela qual os dados são recebidos do dispositivo; e, opcionalmente, o TRST (do inglês *Test Reset*) pode ser usado para resetar a interface de teste. Essa estrutura multipino permite uma comunicação paralela e organizada, facilitando testes detalhados e acesso a múltiplos níveis internos do circuito.



Fonte: Medium.

Por ser um padrão IEEE, o JTAG é amplamente compatível com diversas arquiteturas e permite realizar testes complexos e detalhados, incluindo a capacidade de testar interconexões físicas entre componentes via boundary-scan. Porém, sua complexidade e o número relativamente alto de pinos exigidos podem ser limitadores em projetos compactos.

Visando uma alternativa mais simples e eficiente, especialmente para microcontroladores ARM Cortex-M, foi desenvolvido o protocolo SWD (do inglês *Serial Wire Debug*). Diferentemente do JTAG, o SWD utiliza apenas duas linhas principais para comunicação, uma linha de *clock* (SWCLK) e uma linha bidirecional de dados (SWDIO). Essa redução no número de pinos é uma grande vantagem em dispositivos onde a economia de espaço e custos é crítica. Como compromisso para simplificar a interface e reduzir os pinos, o SWD oferece um conjunto funcional focado em programação e depuração do núcleo do processador, mas não inclui algumas das funcionalidades mais abrangentes do JTAG, como o teste físico das interconexões via *boundary-scan*. O SWD opera de forma serial síncrona, transmitindo comandos e dados em pacotes bem definidos que permitem programar a memória, acessar registradores internos, e controlar o fluxo de depuração, como a inserção de *breakpoints* e a execução passo a passo do código.

Ambos os protocolos têm o mesmo objetivo básico: fornecer um canal direto de comunicação entre a ferramenta de desenvolvimento e o microcontrolador, possibilitando controle profundo sobre o funcionamento interno do sistema embarcado. Enquanto o JTAG é um protocolo mais universal e tradicional, o SWD é uma solução moderna, otimizada para os microcontroladores ARM, que combina simplicidade, menor uso de pinos e alta eficiência na operação de depuração. Assim, em projetos atuais que envolvem microcontroladores ARM Cortex-M, o SWD se tornou a interface preferida para depuração embarcada, facilitando o desenvolvimento de sistemas mais compactos e com maior controle durante o ciclo de vida do produto.

Geração de documentação em projetos embarcados

No desenvolvimento de sistemas embarcados, onde coexistem componentes de *hardware* e *software*, a documentação técnica é uma ferramenta essencial. Ela não apenas auxilia na implementação, mas é indispensável para manutenção, reutilização, colaboração entre áreas distintas do desenvolvimento e escalabilidade do projeto ao longo do tempo. Embora geradores automáticos de documentação não estejam diretamente envolvidos na criação funcional do sistema, seu papel na preservação da qualidade do projeto é fundamental. Assim, ao selecionar as ferramentas que irão compor o ambiente de desenvolvimento de um sistema embarcado, é fundamental incluir soluções que favoreçam a geração de documentação desde as fases iniciais.

No caso do *software*, essas ferramentas permitem a geração de documentação técnica diretamente a partir de comentários estruturados no código-fonte. Essa documentação automatizada pode abranger diversas camadas informativas:

- **Documentação de API**: detalha funções, métodos, parâmetros e valores de retorno, servindo como referência para desenvolvedores.
- **Documentação de Código-Fonte**: oferece uma visão de alto nível da estrutura do sistema, incluindo módulos, classes e suas inter-relações.
- **Documentação de Fluxo de Controle**: representa o comportamento lógico do sistema por meio de descrições e diagramas que explicam como os módulos se conectam e interagem.
- **Documentação de *Layout* de Memória**: útil em sistemas de baixo nível, descreve a alocação e organização de variáveis e estruturas na memória.

Essas camadas não só promovem uma visão mais clara do projeto como também facilitam a colaboração entre desenvolvedores, reduzem o tempo de aprendizagem para novos membros da equipe e aumentam a eficiência na identificação de erros ou falhas. Ferramentas maduras como o Doxygen permitem a geração de documentação técnica diretamente a partir de comentários estruturados no código-fonte, tanto para C/C++ quanto para Python.

Por outro lado, no desenvolvimento de *hardware*, a geração automática de documentação ainda enfrenta desafios mais técnicos do que simplesmente a diversidade de linguagens. Embora existam poucas linguagens de descrição de *hardware* amplamente utilizadas, como VHDL e Verilog, a dificuldade está na natureza própria dessas linguagens e do desenvolvimento em nível físico. As descrições em HDL (do inglês *Hardware Description Language*) envolvem comportamentos concorrentes, restrições temporais e níveis de abstração muito próximos do *hardware* real, o que torna mais complexo extrair, de forma automatizada, informações organizadas e compreensíveis. Além disso, há menos padronização nas convenções de documentação e menor maturidade nas ferramentas voltadas a esse fim. Ainda assim, o campo vem evoluindo, com esforços crescentes da

comunidade de EDA (do inglês *Electronic Design Automation*) para oferecer suporte à documentação automatizada e à visualização estrutural de projetos baseados em HDL.

Além disso, práticas da engenharia de *software* têm sido gradualmente adaptadas ao mundo do *hardware*. A utilização de diagramas da UML, por exemplo, tem se mostrado relevante na documentação e modelagem de sistemas embarcados:

- Diagramas de Componentes são usados para representar arquiteturas de alto nível, onde módulos de *hardware* (como processadores, controladores e periféricos) são modelados como componentes interconectados.
- Diagramas de Pacotes podem representar agrupamentos lógicos de módulos ou subsistemas, facilitando a organização de projetos complexos.
- Diagramas de Implementação (em inglês, *Deployment*) são aplicados para mostrar a alocação de funcionalidades em dispositivos físicos, como FPGAs, microcontroladores ou unidades de memória.
- Diagramas de Máquina de Estados são particularmente relevantes para o domínio de *hardware*, já que muitas funções de controle digital são implementadas por meio de máquinas de estados finitos. Esses diagramas descrevem o comportamento sequencial de módulos, mostrando os estados possíveis e as transições entre eles com base em condições e eventos.

Essa abordagem é especialmente útil em projetos que seguem metodologias de *co-design hardware/software*, em que a integração entre os domínios é essencial. Ferramentas como a SysML, considerada como um dialeto da UML voltado para engenharia de sistemas, também têm ganhado espaço por permitir uma representação mais completa e coesa dos aspectos funcionais, estruturais e físicos de sistemas embarcados. No entanto, apesar da relevância dos diagramas UML/SysML para representar sistemas embarcados, a geração automática desses diagramas a partir de descrições HDL ainda é limitada. Em muitos casos, a criação dos diagramas é feita manualmente, ou depende de ferramentas especializadas com suporte parcial. Isso limita sua adoção em projetos maiores ou mais dinâmicos, mas também destaca uma oportunidade para o desenvolvimento de soluções que integrem melhor documentação e automação no fluxo de co-projeto de *software* e *hardware*.

Versionamento de projetos

Um sistema de controle de versões (em inglês, *Version Control System – VCS*) é uma ferramenta essencial no desenvolvimento de *software*, responsável por gerenciar as diferentes versões dos arquivos que compõem um projeto. Esses sistemas são fundamentais para garantir a estabilidade, rastreabilidade e qualidade do projeto, tanto em desenvolvimento individual quanto colaborativo. Eles monitoram alterações feitas ao longo do tempo, permitindo que os desenvolvedores colaborem de forma eficiente. A cada operação conhecida como *commit* (confirmação), é criada uma nova versão única e rastreável que incorpora todas as modificações realizadas nos arquivos.

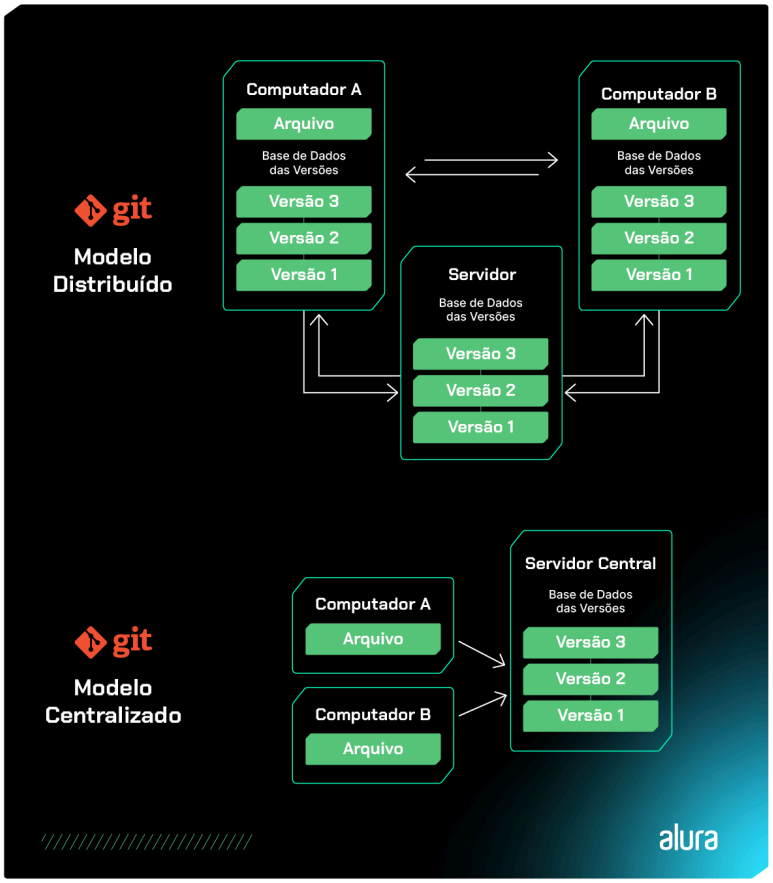
Além do *commit*, os sistemas de controle de versão incluem as seguintes funcionalidades:

- Registro e rastreamento de alterações ao longo do tempo;
- Criação de ramificações (em inglês, *branches*) para desenvolvimento paralelo;
- Mesclagem (em inglês, *merge*) de alterações ao tronco principal do projeto;

- Resolução de conflitos entre alterações simultâneas;
- Registro de autoria e tempo das modificações, com mensagens de *commit* associadas;
- Recuperação de versões anteriores;
- Suporte a múltiplos usuários com controle de concorrência;
- Utilização de *tags* para identificar versões específicas relevantes.

Essas funcionalidades, porém, dependem fortemente da capacidade dos sistemas de versionamento de analisar e comparar o conteúdo dos arquivos de forma eficiente. Como consequência, os sistemas de controle de versões operam de forma particularmente eficaz com arquivos de texto, como códigos-fonte em linguagens de programação ou descrição de *hardware* (HDLs como VHDL e Verilog). Por outro lado, arquivos binários ou visuais, como esquemáticos, imagens ou modelos gráficos, não oferecem a mesma granularidade para rastreamento de alterações, o que limita o aproveitamento pleno dos recursos de versionamento. Assim, para projetos de *hardware* digital, representações textuais como HDL são mais apropriadas ao uso de controle de versões do que representações esquemáticas.

Nesse contexto, é importante compreender as duas principais arquiteturas adotadas por sistemas de controle de versões: o modelo centralizado e o distribuído. Dentre os sistemas centralizados mais conhecidos estão o CVS (do inglês *Concurrent Versions System*) e o SVN (do inglês *Subversion*)**.** No modelo centralizado, há um repositório único que armazena todas as versões do código, e os usuários precisam se conectar a ele para obter ou enviar alterações. Isso impõe uma dependência de conectividade e reduz a flexibilidade em ambientes descentralizados.



Fonte: Alura

Já o Git[1] representa uma abordagem distribuída. Nesse modelo, cada desenvolvedor possui uma cópia completa do repositório, com todo o histórico de versões. Isso permite a realização de operações locais, como *commits*, comparações e reversões, mesmo sem conexão com um servidor central. As alterações podem ser sincronizadas posteriormente com outros repositórios remotos. Essa descentralização favorece a autonomia dos desenvolvedores, melhora o desempenho das operações locais e é especialmente vantajosa em equipes distribuídas ou com conectividade intermitente.

Cada sistema adota um mecanismo distinto de versionamento:

- O CVS registra apenas as diferenças (*deltas*) entre versões consecutivas;
- O SVN cria, sob o ponto de vista lógico, um *snapshot* completo do repositório a cada *commit*;
- O Git utiliza identificadores únicos baseados em *hashes* criptográficos do conteúdo e da mensagem do *commit*, garantindo integridade do histórico do projeto e rastreabilidade de cada alteração ao longo do tempo.

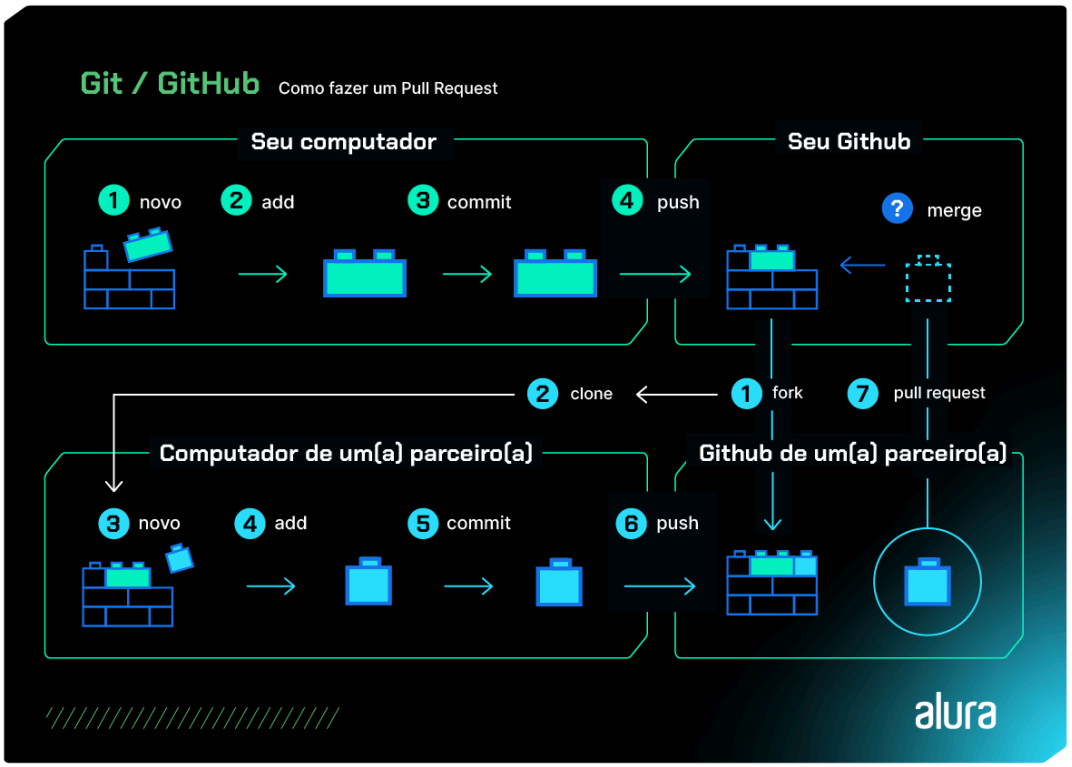
Diversas plataformas modernas expandem as funcionalidades do Git para apoiar o gerenciamento completo de projetos. Entre elas destacam-se o GitHub, o GitLab e o Bitbucket:

- O Bitbucket, mantido pela Atlassian, se integra estreitamente com ferramentas como Jira e Confluence, sendo uma escolha frequente em equipes que já utilizam esse ecossistema.
- O GitLab adota uma abordagem unificada, oferecendo ferramentas integradas para integração contínua, entrega contínua e monitoramento, cobrindo todo o ciclo de vida do desenvolvimento.
- O GitHub é amplamente reconhecido por sua interface acessível, vasta comunidade e forte presença em projetos de código aberto. Sua popularidade favoreceu a criação de uma ampla base de documentação, fóruns e tutoriais.

Uma análise comparativa anual entre GitHub e GitLab pode ser encontrada no *site* de Radix.

O seguinte diagrama demonstra o Git como um excelente suporte ao desenvolvimento colaborativo de projetos, especialmente em ambientes de código aberto ou em equipes distribuídas. Ele ilustra a possibilidade de se criar através da sua função *fork* uma cópia completa do repositório “Seu Github” para uma conta de usuário separada em plataformas como GitHub, GitLab ou Bitbucket e fazer modificações independentes. Os colaboradores do projeto original vão revisar suas alterações por meio da função *pull request*, onde será possível discutir elas por ali também. Se tudo estiver em ordem, eles poderão mesclar as alterações no projeto principal no “Seu Github”.

[1] “Git”" não é uma sigla, mas sim um termo coloquial derivado do inglês. Foi criado por Linus Torvalds em 2005 para desenvolver o *kernel* do sistema operacional Linux. O próprio Linus explicou que “git” é uma gíria britânica que significa “pessoa desagradável ou estúpida”, e ele escolheu esse nome em parte por brincadeira, expressando sua frustração com as limitações de outras ferramentas de controle de versões existentes na época.



Fonte: Alura.

No desenvolvimento de projetos em equipe, especialmente na área de Engenharia de Software, o uso de sistemas de controle de versão, como o Git, é fundamental para organizar o trabalho de forma eficiente e colaborativa. Uma prática comum nesses sistemas é o uso de uma *branch* principal, geralmente chamada de *main* (ou *master*, em projetos mais antigos). Essa *branch* representa a linha principal de desenvolvimento e deve conter a versão mais estável e consolidada do projeto.

A primeira versão da *main* não precisa ser um sistema completo, mas deve fornecer uma base mínima, funcional e bem estruturada para que os demais membros da equipe possam desenvolver novas funcionalidades de forma organizada. Essa versão inicial deve conter, por exemplo, a estrutura de pastas e arquivos do projeto, um arquivo README.md com as instruções iniciais, arquivos de configuração como o .gitignore e outras ferramentas básicas que serão utilizadas ao longo do desenvolvimento. Ela deve ser sólida o suficiente para evitar problemas futuros, mas não precisa incluir todas as funcionalidades finais do sistema. A responsabilidade por montar essa primeira versão da *branch* main geralmente é de um líder técnico ou de um pequeno grupo com experiência, que já tenha uma visão clara dos requisitos e da arquitetura do projeto. Essa equipe inicial define as bases e os padrões que os demais integrantes deverão seguir.

Com a *main* criada, cada membro da equipe deve clonar o repositório e criar uma nova *branch* a partir dela para desenvolver uma funcionalidade específica. Essas *branches* recebem nomes descritivos como *feature/login*, *bugfix/erro-cadastro* ou *hotfix/ajuste-deploy*. Após concluir o trabalho, o desenvolvedor deve abrir um *Pull Request* (PR), que é uma solicitação para integrar sua *branch* à *main*. Esse PR deve ser revisado por outro membro da equipe, garantindo a qualidade do código e a compatibilidade com o restante do projeto antes de ser incorporado.

É importante destacar que a *branch main* não deve ser modificada diretamente. Todas as alterações devem passar por *branches* separadas e ser integradas por meio de PRs aprovados. Além disso, boas práticas como atualizar constantemente sua *branch* com a *main*, padronizar os nomes das *branches* e mensagens de *commit*, e utilizar testes automatizados ajudam a manter o projeto organizado e facilitar a colaboração entre os membros da equipe. Esse processo garante que o trabalho em grupo seja mais rastreável, reduzindo conflitos de código e melhorando a qualidade final do produto desenvolvido.

Embora os sistemas de controle de versão tenham sido inicialmente concebidos para gerenciar arquivos de texto, como código-fonte, sua aplicação expandiu-se consideravelmente para o desenvolvimento de *hardware* digital. Isso é especialmente relevante quando o *hardware* é descrito por linguagens como VHDL ou Verilog, que utilizam arquivos textuais. Nesses cenários, os projetos de *hardware* se beneficiam das mesmas funcionalidades que o desenvolvimento de *software*, incluindo o controle de histórico detalhado, a criação de ramificações (em inglês *branches*) para experimentação e a colaboração simultânea entre equipes. Contudo, um desafio persistente em muitos projetos de sistemas embarcados é o tratamento de arquivos binários. Esquemas elétricos, layouts de placas de circuito impresso (PCBs) e arquivos de simulação, por exemplo, não são adequadamente manipulados por sistemas de versionamento tradicionais. Isso ocorre porque esses sistemas não conseguem inspecionar diretamente as mudanças internas desses arquivos nem resolver automaticamente os conflitos que surgem quando múltiplos colaboradores modificam o mesmo arquivo binário.

Na prática, para contornar essas limitações, equipes de desenvolvimento adotam estratégias como: representar modelos gráficos também em formatos textuais (quando disponíveis), manter artefatos derivados fora do versionamento direto, documentar cuidadosamente as dependências entre arquivos e empregar *scripts* automatizados para reconstruir componentes não textuais a partir de fontes versionadas. Além disso, é comum que se estabeleçam nomenclaturas coerentes e políticas internas para garantir a consistência entre as versões do *firmware* e do *hardware* correspondente. Mesmo sem suporte ideal a arquivos binários, o uso disciplinado de sistemas como o Git tem se consolidado como uma prática viável — e altamente recomendada — para projetos integrados de *hardware* e *software* em sistemas embarcados.

PLACAS DE DESENVOLVIMENTO

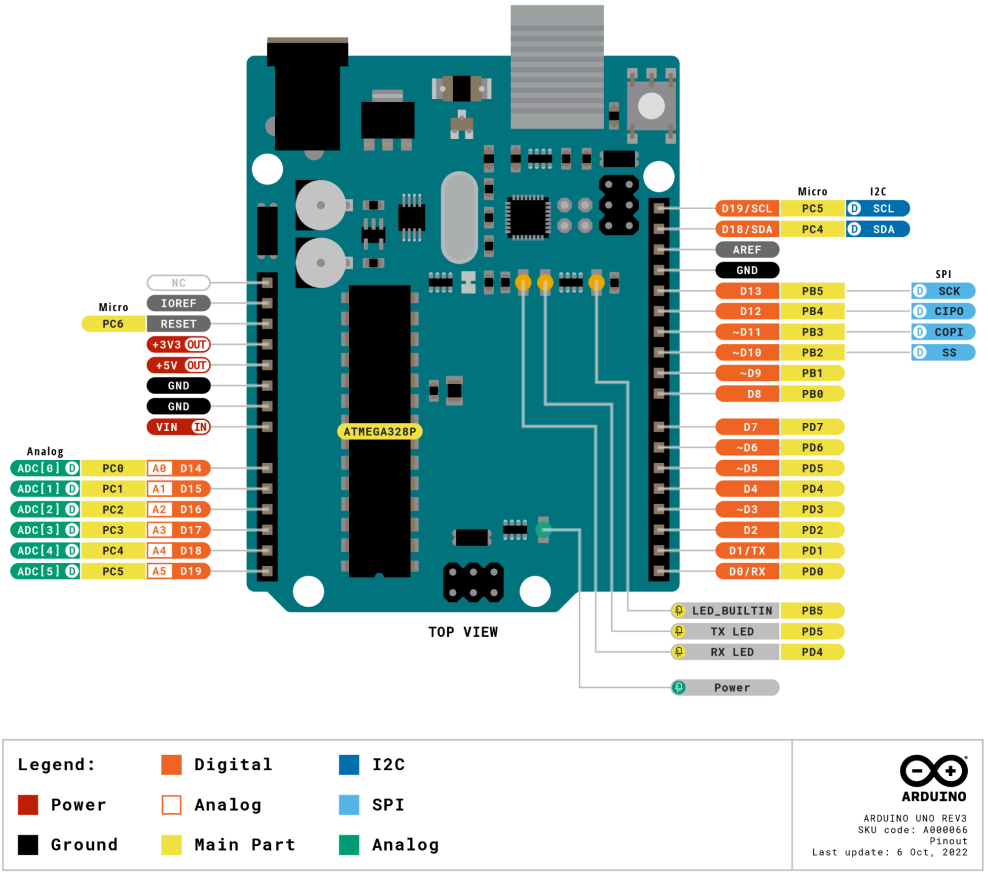
Para desenvolver projetos em sistemas embarcados, vamos usar primeiro as placas de desenvolvimento para prototipagem e testes de validação. Elas serão nosso laboratório prático para dar vida ao código e interagir com o *hardware*. Apresentaremos três plataformas populares, Arduino UNO (baseado em ATMega 328P), Black Pill (baseada em STM32) e Raspberry Pi Pico (baseado em RP2040), mas nosso foco principal será nas duas últimas.

Arduino UNO (ATMEGA328P)

O Arduino Uno é uma plataforma amplamente utilizada no ensino e em projetos de prototipagem rápida, oferecendo um bom equilíbrio entre simplicidade e funcionalidade, mas com limitações para aplicações mais complexas ou que demandem maior capacidade de processamento. Baseado no

microcontrolador ATmega328P, ele oferece uma arquitetura de 8 *bits* com um clock de 16 MHz, o que é suficiente para a maioria das aplicações embarcadas de baixa a média complexidade.

Em termos de memória, o Uno possui 32 kB de Flash para armazenamento de código, 2 kB de SRAM para dados voláteis e 1 kB de EEPROM para dados não voláteis. Embora essas capacidades limitem projetos com alto uso de dados, elas são adequadas para aplicações embarcadas básicas. Para a interação com o mundo analógico, destaca-se seu conversor ADC de 10 *bits* com 6 canais, permitindo a leitura de sensores como os de temperatura, luz ou potenciômetros. Já na comunicação serial, o Arduino Uno suporta UART, I2C e SPI, o que o torna apto a se comunicar com uma variedade de módulos como sensores digitais, *displays* e memórias externas. O microcontrolador integra três *timers* (dois de 8 *bits* e um de 16 *bits*), essenciais para controle de tempo, geração de PWM e contagem de eventos.



A carga de programa no Arduino Uno é feita via USB, utilizando um conversor serial integrado (ATmega16U2), o que permite o *upload* direto do código pela IDE Arduino, sem necessidade de um programador externo. No entanto, em termos de depuração física, o Uno não possui suporte nativo a depuração em tempo real via interfaces como JTAG ou SWD, que são comuns em microcontroladores mais avançados. A depuração típica é realizada por meio de mensagens no console serial, acessível pela própria IDE Arduino.

Embora o Microchip Studio (antigo Atmel Studio), IDE oficial e mais avançada da Microchip, ofereça recursos profissionais de depuração, como *breakpoints*, *watchpoints* e inspeção de registradores, esses só são plenamente utilizáveis em microcontroladores com suporte a interfaces

de depuração física, usando por exemplo JTAG ou debugWIRE). No caso do Arduino Uno, a depuração com Microchip Studio só seria viável se o microcontrolador ATmega328P for regravado com um *bootloader* compatível e conectado a um depurador externo, como o Atmel-ICE, utilizando a interface debugWIRE. Isso exige acesso físico ao pino RESET e cuidados na configuração do *fuse bit* correspondente, o que vai além do uso típico da placa Arduino.

A alimentação do Arduino Uno pode ser feita pela porta USB (5V) ou por uma fonte externa (7 a 12V), com regulação interna de tensão. A *toolchain* padrão é a IDE Arduino, que abstrai a complexidade da programação em C/C++ por meio de bibliotecas e funções simplificadas, embora também seja possível programar diretamente em C com o compilador AVR-GCC.

É importante notar que o Arduino Uno não executa Python diretamente devido às suas limitações de *hardware* e memória. No entanto, é possível usar Python em um computador para interagir com o Arduino via porta serial, utilizando bibliotecas como o pySerial. As funções dos periféricos em C são amplamente suportadas por bibliotecas da comunidade e pela própria IDE, com a possibilidade de acesso direto aos registradores para configurações mais avançadas. Por fim, o suporte a RTOS (do inglês *Real Time Operating System*) no Uno é bastante limitado devido ao seu *hardware* restrito, mas versões simplificadas como o FreeRTOS para AVR podem ser utilizadas, exigindo cuidado no uso de memória e processamento.

Black Pill (STM32F411)

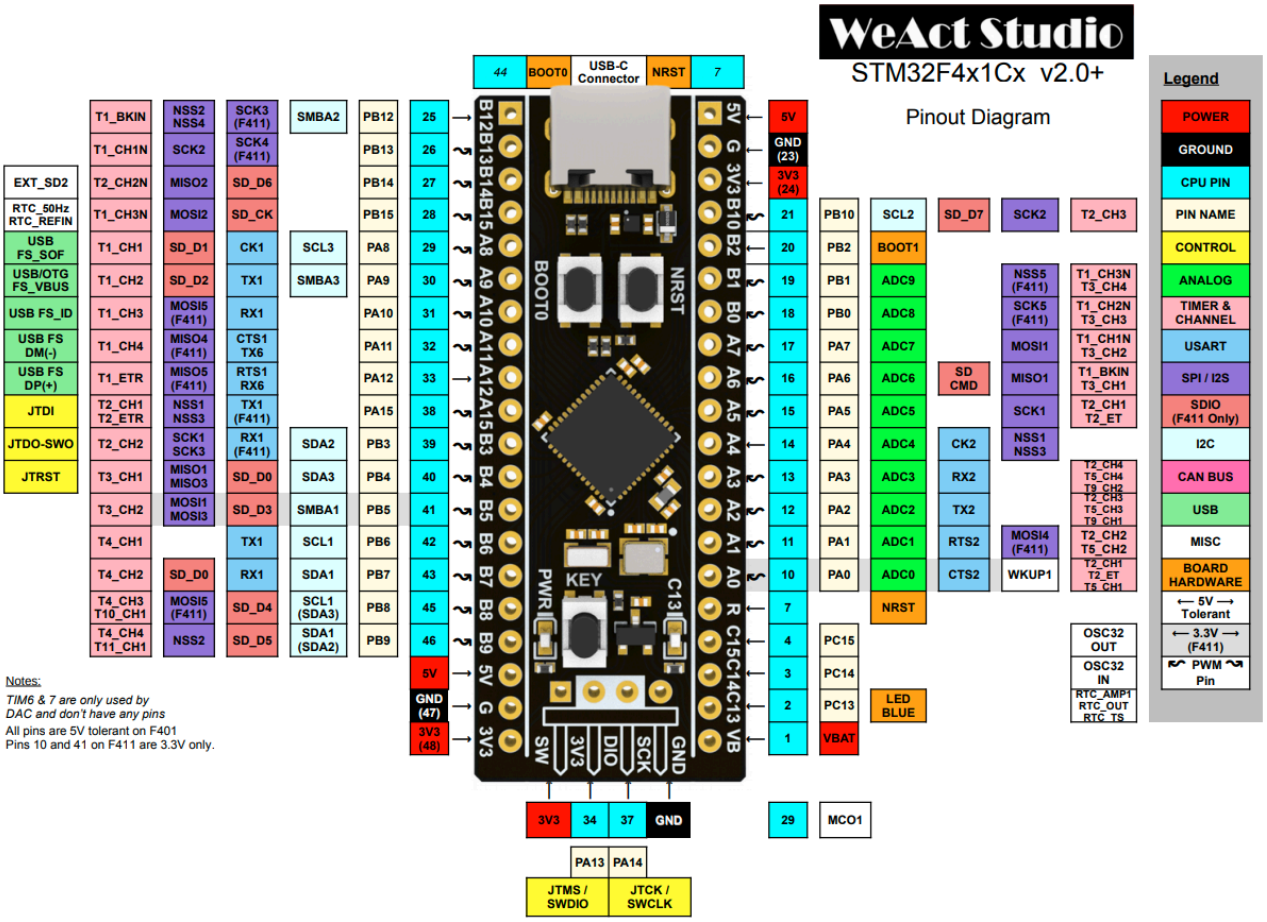
A Black Pill é uma placa de desenvolvimento popular baseada nos microcontroladores STM32F401CCU6 ou STM32F411CEU6, da STMicroelectronics. Ela se destaca pelo excelente custo-benefício e pelo poder de processamento superior, sendo ideal para projetos que demandam mais recursos e eficiência.

No seu núcleo, a Black Pill conta com um processador ARM Cortex-M4, que opera tipicamente a 100 MHz (no F401) ou 128 MHz (no F411). Sua arquitetura inclui uma Unidade de Ponto Flutuante (em inglês, *Floating Point Unit* – FPU), o que a torna perfeita para cálculos complexos e processamento de sinais. Em termos de memória, possui uma Flash interna de 256 KB (F401) ou 512 KB (F411) para o programa, e SRAM de 64 KB (F401) ou 128 KB (F411) para dados e pilha, tudo integrado no mesmo *chip*.

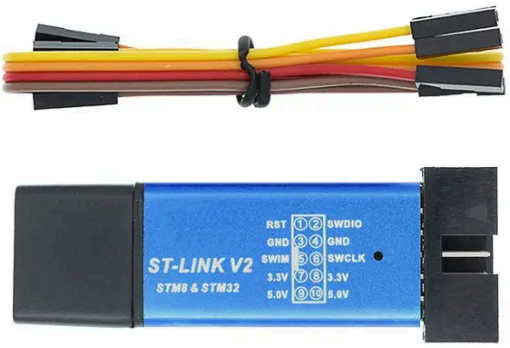
Para interagir com o ambiente, a placa dispõe de um ADC de 12 *bits* com múltiplos canais (até 16), capaz de ler diversos sinais analógicos. Em comunicação serial, a Black Pill é bem equipada com três UARTs, três SPIs e três I2Cs. Além disso, possui uma interface USB 2.0 *Full Speed Device* que permite sua atuação como diversos tipos de dispositivos USB. Por possuir um controlador USB *Full-Speed* (USB OTG FS), a porta USB-C pode ser configurada via *firmware* para operar como um Dispositivo de Classe de Comunicação (em inglês, *Communication Device Class* – CDC), apresentando-se ao computador como uma porta COM virtual para facilitar a comunicação serial e *flashing*. Para controle preciso de tempo e sinais complexos, a Black Pill oferece um rico conjunto de *timers* e múltiplos canais PWM.

A *toolchain* padrão para desenvolvimento em C/C++ para a Black Pill é o GCC para arquitetura ARM (arm-none-eabi-gcc). A STM32CubeIDE é o ambiente de desenvolvimento mais recomendado, pois já integra a maioria das ferramentas necessárias, exceto as funções do

STM32CubeProgrammer. Esse aplicativo é uma ferramenta essencial para carregar *firmwares* em formatos como .bin ou .hex que não são gerados diretamente pelo seu projeto C/C++ padrão (como é o caso do interpretador MicroPython), ou para realizar operações de baixo nível na Flash ou *Option Bytes*, especialmente se estiver usando o modo DFU (do inglês *Device Firmware Upgrade*) via USB-C. O console serial pode ser acessado de duas formas: através de uma UART externa, utilizando um conversor USB-Serial, ou diretamente pela porta USB-C configurada como uma VCP (do inglês *Virtual COM Port*). Muitas Black Pills vêm com um *bootloader* DFU pré-instalado na ROM. No entanto, para carregar o programa e realizar depuração avançada, a conexão é feita através dos pinos SWDIO/SWCLK, localizados no lado oposto do conector USB-C. Para essa finalidade, um *debugger* SWD externo (como o ST-Link v2 da STMicroelectronics e o J-Link da SEGGER) é essencial, utilizando o *software* OpenOCD, ou outras opções proprietárias, integrado na STM32CubeIDE para intermediar a comunicação com o microcontrolador.

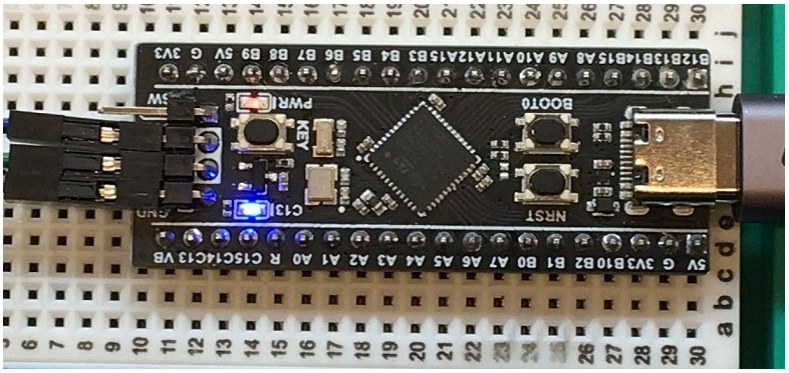


Fonte: Deepblue Mbedded.



Programador e depurador ST-LINK V2

A figura a seguir ilustra a alimentação e a conexão do microcontrolador ao STM32CubeProgrammer pela porta USB-C (lado direito), mostrando o método de *flashing* via DFU. Adicionalmente, ela demonstra a conexão com a STM32CubeIDE pelos pinos SWDIO (DIO), SWCLK (SCK) e GND (terra) (lado esquerdo), mediada por um programador e depurador externo. Essa última configuração é utilizada para a depuração em tempo real do código. A alimentação da placa é fornecida pela porta USB-C (5V). O pino de alimentação 3V3, localizado no lado oposto, permanece em aberto para esta configuração. A placa conta com um regulador de tensão interno que converte a tensão de entrada vinda da USB-C ou do pino 5V para os 3.3V que o microcontrolador STM32 e a maioria dos seus periféricos operam. **Jamais alimente a Black Pill pela porta USB-C e, ao mesmo tempo, pelo pino de 3.3V** (como o que pode vir do depurador SWD). Essa prática força o regulador de tensão *onboard* a lidar com duas fontes de 3.3V concorrentes, o que pode sobrecarregá-lo e causar danos permanentes à placa.



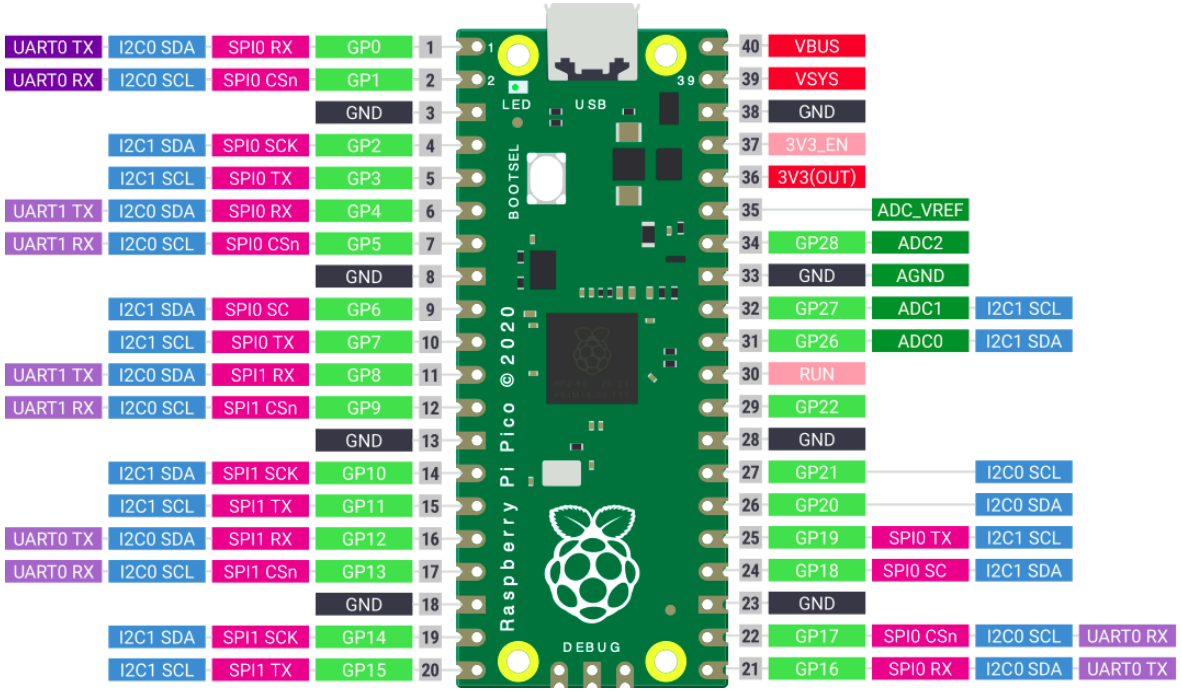
Fonte: Github.

A Black Pill também oferece suporte ao uso de Python, através das implementações MicroPython e CircuitPython. Para carregar o interpretador desses *bytecodes* Python no microcontrolador, precisa-se conectar o STM32CubeProgrammer com o microcontrolador pela porta USB-C. O processo é demonstrado em um video-tutorial que permite dispensar o uso de linha de comando na

gravação. Contudo, o controle total e otimizado de todos os periféricos é alcançado em C/C++ usando as bibliotecas e o SDK (do inglês *Software Development Kit*) da STMicroelectronics. Esse SDK inclui as bibliotecas CMSIS (do inglês *Cortex Microcontroller Software Interface Standard*), que fornece uma interface padronizada de *hardware*-abstração para processadores Arm Cortex-M, simplificando a programação de periféricos essenciais, e as bibliotecas HAL (do inglês *Hardware Abstraction Layer*), que oferecem uma camada de abstração mais granular e de alto nível para os periféricos específicos dos microcontroladores STM32. As funções dessas bibliotecas garantem o controle granular e o determinismo essenciais para aplicações críticas. Por fim, o suporte a RTOS como o FreeRTOS é robusto, o que, combinado com a arquitetura Cortex-M4 e a memória abundante, torna a Black Pill uma excelente plataforma para sistemas multitarefa com requisitos rigorosos de tempo real. Para facilitar o desenvolvimento de *firmwares* para RTOS na Black Pill, a plataforma STM32CubeF4 é amplamente utilizada, integrando o FreeRTOS e outras *middlewares* diretamente.

Para programação de STM32F411 a nível de *hardware*, consulte detalhes sobre registradores de configuração, controle, estado e dados na referência.

Raspberry Pi Pico (RP2040)



A Raspberry Pi Pico é uma placa de desenvolvimento compacta e eficiente, baseada no microcontrolador RP2040, projetado pela própria Raspberry Pi Foundation. Voltada para aplicações embarcadas, ela combina bom desempenho, baixo custo e grande flexibilidade. A alimentação pode ser feita diretamente pela porta USB (5V) ou por uma fonte externa entre 1,8 V e 5,5 V, conferindo flexibilidade em diferentes cenários de uso.

No núcleo do RP2040 há um processador dual-core ARM Cortex-M0+ com frequência de até 133 MHz, permitindo a execução paralela de tarefas. A placa inclui 264 KB de SRAM e utiliza uma

memória Flash externa de 2 MB para armazenar programas. Essa quantidade pode variar em versões alternativas.

Para entrada de sinais analógicos, a Pico oferece três canais ADC de 12 *bits*. A conectividade serial é ampla, com suporte a duas UARTs, duas SPIs e duas I2Cs, além de uma porta USB 1.1 que funciona tanto como *Device* quanto como *Host*, podendo a Pico atuar como um dispositivo USB quando conectada a um computador e um "mestre" ao se conectar a outros dispositivos USB (como um mouse, teclado ou pen drive USB) e controlá-los. Isso é uma capacidade mais avançada e nem todas as placas de desenvolvimento de baixo custo oferecem isso.

A placa também disponibiliza temporizadores, saídas PWM e outras funções essenciais para controle de tempo e geração de sinais. Um diferencial do RP2040 é o seu PIO (do inglês *Programmable Input/Output*), que consiste em um conjunto de "máquinas de estado" dedicadas e programáveis. É possível escrever pequenos programas em uma linguagem de *assembly* própria do PIO para que essas máquinas de estado controlem os pinos GPIO de forma extremamente precisa e em tempo real.

A Raspberry Pi Pico possui várias versões que se diferenciam principalmente pela conectividade, presença de *headers* (pinos GPIO) soldados e recursos de depuração. O Pico original é o mais básico, sem Wi-Fi ou *Bluetooth*, e geralmente vendido sem os pinos soldados. Já o Pico H é funcionalmente idêntico ao original, mas vem com os pinos soldados de fábrica e um conector SWD para depuração. O Pico W, por sua vez, adiciona conectividade Wi-Fi de 2.4 GHz graças a um *chip* da Infineon (CYW43439), mas o *Bluetooth* não está ativado. Ele pode ser identificado visualmente por uma antena metálica prateada no canto da placa. O Pico WH é a versão com Wi-Fi e também vem com *headers* soldados e o conector SWD, combinando os recursos do W e do H. Por fim, o Pico W2 é uma evolução do Pico W, lançado em 2024, que usa um *chip* de rádio atualizado (CYW43439B0) com suporte oficial a *Bluetooth* 5.2, além do Wi-Fi. Visualmente é semelhante ao Pico W, mas pode apresentar diferenças no formato da antena e traz a marcação "Pico W2" na parte de trás. Assim, é possível distinguir as versões pela presença ou ausência da antena, dos pinos soldados e pelas inscrições na placa.

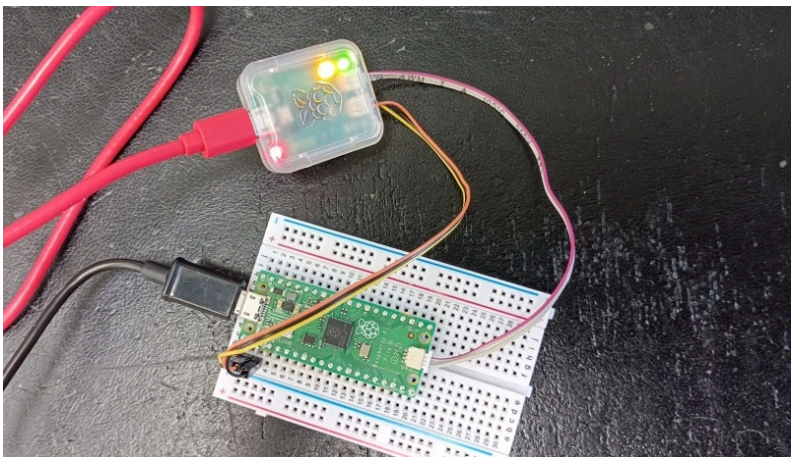
O desenvolvimento em C/C++ para o microcontrolador RP2040 é realizado utilizando o GCC para ARM (arm-none-eabi-gcc), gerenciado por meio do CMake, com base no SDK oficial da Raspberry Pi, que fornece bibliotecas, exemplos e abstrações de *hardware*. Essas abstrações de *hardware* simplificam o uso de interfaces como GPIO, SPI, I2C, UART, ADC e PWM. Além disso, há algumas bibliotecas auxiliares de alto nível, como suporte a USB via TinyUSB e multitarefa cooperativa, que facilitam o desenvolvimento de aplicações mais complexas. Porém, em comparação com as IDEs da STM, a abordagem do SDK do RP2040 é mais direta e manual, exigindo maior familiaridade com o *hardware* e a estrutura de projeto, pois a filosofia desse SDK tende a ser mais próxima do desenvolvimento *bare-metal* tradicional.

O ambiente de desenvolvimento típico envolve o uso de IDEs como o Visual Studio Code (VS Code), configurado com extensões para CMake, IntelliSense e integração com terminal, embora também seja possível usar outras IDEs como Eclipse, CLion ou o próprio terminal Linux/Windows com Make/CMake. O acesso ao console serial pode ser feito de duas formas. A primeira é através de uma das UARTs físicas do *chip*, sendo os pinos GP0 (TX) e GP1 (RX) tipicamente alocados para a UART0. A segunda alternativa é pela porta micro-USB, que pode ser configurada para operar

como uma interface serial virtual pelo seu controlador USB interno, funcionando de forma análoga ao Raspberry Pi Pico. Ambas as opções são extremamente úteis para visualizar *logs* e realizar testes.

A gravação do *firmware* no RP2040 é um processo simples e intuitivo: basta manter pressionado o botão BOOTSEL enquanto conecta a placa via micro-USB. Ao fazer isso, o microcontrolador entra no modo *USB Mass Storage*, apresentando a memória *Flash* interna ao computador como um disco removível. Nesse ponto, selecione manualmente o arquivo .uf2, ou outras extensões, que contém o *firmware* sem instruções de depuração e o arraste ou copie para o disco removível que o RP2040 aparece. Para a programação e depuração física, o RP2040 suporta nativamente a interface SWD. Isso permite a conexão de um depurador externo, como o *Raspberry Pi Debug Probe*, J-Link da SEGGER, *Black Magic Probe*, ou um *probe* com um *firmware* baseado no protocolo CMSIS-DAP de código aberto instalado, ligando-o aos três pinos DEBUG do microcontrolador.

A figura a seguir ilustra a conexão de um Raspberry Pi Pico ao computador, utilizando tanto a porta mini USB (cabo preto) quanto os três pinos DEBUG (cabo vermelho), SWDIO (do inglês *Serial Wire Data Input/Output*), SWCLK (do inglês *Serial Wire Clock*) e GND (do inglês *Ground*). A conexão via USB-C é multifuncional: ela alimenta a placa e permite a comunicação serial que uma aplicação possa necessitar, como exibir dados de sensores em um terminal. Paralelamente, no lado oposto da placa, os três pinos DEBUG, conectados a um Raspberry Pi Debug Probe, servem para controlar diretamente a execução do código, possibilitando a depuração em nível de *hardware*. Além disso, o próprio *Debug Probe* oferece um segundo conector de 3 pinos que viabiliza a comunicação serial do Pico alvo com um console serial no computador, utilizando um dos módulos UART do Pico.



Fonte: Elektormagazine.

O Raspberry Pi Pico não se limita ao desenvolvimento em C/C++; ele é totalmente compatível com linguagens interpretadas como MicroPython e CircuitPython. Nesses casos, o ambiente de desenvolvimento mais comum inclui o uso de Thonny IDE, que oferece um console interativo REPL (do inglês *Read-Eval-Print Loop*), *upload* automático de *scripts* e uma interface amigável para iniciantes. É possível escrever, testar e executar *scripts* diretamente na memória da placa, facilitando a prototipagem e o aprendizado. Contudo, como já comentado antes, para aplicações que exigem maior desempenho, controle preciso de tempo ou multitarefa, o uso de C/C++ é mais adequado.

Para programar o RP2040 em C/C++, é possível adotar dois níveis complementares de desenvolvimento. No nível mais próximo do *hardware*, busca-se maior controle e otimização por meio do acesso direto aos registradores de configuração, controle, estado e dados do microcontrolador. Essa abordagem, baseada na documentação de referência técnica, é essencial quando se deseja obter desempenho máximo ou realizar interações muito específicas com os periféricos. Por outro lado, para um desenvolvimento mais ágil e com maior nível de abstração, a Raspberry Pi Foundation oferece um SDK oficial para as placas da série Pico. Esse SDK inclui APIs de alto nível, disponíveis tanto para C quanto para Python, que facilitam o uso dos recursos do microcontrolador. Utilizando essas APIs, o desenvolvedor pode se concentrar na lógica da aplicação sem a necessidade de lidar diretamente com os registradores.

A documentação oficial do SDK em C/C++ está também disponível online e inclui uma lista completa das funções oferecidas. Além disso, o SDK dá suporte à programação paralela, aproveitando os dois núcleos do RP2040, e é compatível com sistemas operacionais em tempo real (RTOS), como o FreeRTOS, permitindo o desenvolvimento de aplicações com requisitos de tempo real e maior escalabilidade.

Raspberry Pi Pico vs. Black Pill

Segue-se uma tabela comparativa das principais características dos microcontroladores Raspberry Pi RP2040 e STM32F411. É importante notar que, para o RP2040, a coluna se refere à placa de desenvolvimento Raspberry Pi Pico (ou placas similares baseadas no RP2040), e não apenas ao microcontrolador. Isso porque o RP2040 em si não possui memória Flash interna; ele depende de uma memória Flash externa, que já vem integrada nas placas de desenvolvimento, influenciando diretamente a capacidade de armazenamento de código disponível ao usuário. Já para o STM32F411, as especificações listadas se referem ao microcontrolador em si, que possui memória *Flash* interna.

Característica	Raspberry PI Pico	*Black Pill*
Processamento	2 núcleos M0+, 133MHz max.	1 núcleo M4 com ULA ponto fixo, 100MHz max.
Memória	2MB FLASH externa, 264kB SRAM.	512kB FLASH interna, 128kB SRAM.
Periféricos analógicos	3 ADC 12 *bits* com 1 canalcada, sem DAC.	1 ADC 12 *bits* com 16 canais, sem DAC.
Periféricos de comunicação serial	2 UART, 2 SPI, 2 I2C, 1 USB 1.1 (*Device*/*Host*)	3 UART, 3 SPI, 3 I2C, 1 USB 2.0 FS (*Device*)
Timers	4 de 16 *bits*, PWM em módulo dedicado com 16 canais (configuráveis como 2 de 16 *bits*, 1 de 32 *bits*), RTC.	2 de 16 *bits*, 1 de 32 *bits*, RTC, PWM usando canais dos *timers* (máximo 13).
Carga de programa	Simples, cópia de arquivo ".bin" no *drive* virtual via USB.	Usa o ST-LINK (SWD), com *debug* integrado à IDE.

Depuração	Exige *debugger* externo SWD e instalação e configuração de OpenOCD.	Exige o ST-LINK (SWD), com *debug* integrado à IDE.
Alimentação	USB da Pico, ou bateria recarregável na BDL (seleção automática), carregada pela USB da Pico.	USB da Black Pill, ou ST-LINK (**Nunca conectar duas alimentações ao mesmo tempo!**). A bateria recarregável não pode ser carregada nem usada por causa da configuração da Black Pill.
Toolchain	Mais complexa (VS Code com SDK), exige vários plugins, apenas código em texto.	Integrada (Baseada em Eclipse), possui a CubeIDE, que permite gerar códigos de inicialização para os periféricos em interface gráfica.
Console serial	Porta COM virtual diretamente na mesma USB usada na carga de programa.	Permite implementar porta COM virtual na USB, separada do ST-LINK.
Suporte a Python	Micropython, cerca de 1.3 MB disponíveis para *scripts* Python, bibliotecas e arquivos de dados e 240 kB disponíveis para variáveis, objetos e execução do código Python.	CircuitPython, entre 192 KB e 256 KB disponíveis para *scripts* Python e bibliotecas e 96kB disponíveis para variáveis, objetos e execução do código Python.
Funções de periféricos em C	Extensão Pico SDK, que permite adicionar funções de alto nível para controle de periféricos, mas que deve ser adicionada ao VS Code; código de inicialização deve ser escrito pelo programador.	Ferramenta nativa da IDE (Cube), amplo suporte a periféricos com funções HAL (*Hardware Abstract Layer*) e geração de código de inicialização com interface gráfica.
Suporte a RTOS	Possível (FreeRTOS), com a instalação de extensão adicional; implementação de tarefas, semáforos, etc., em linhas de comando.	Possível, com o FreeRTOS integrado ao Cube, usando uma camada de abstração (CMSIS-RTOS); pode usar as funções do FreeRTOS diretamente com pequenas alterações; implementação de tarefas, semáforos, etc., na interface do Cube.

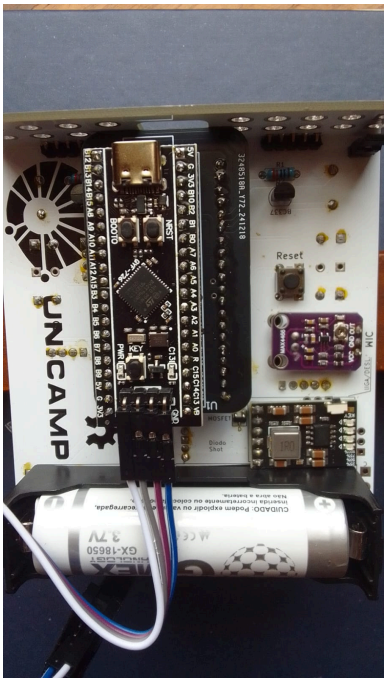
BitDogLab

A BitDogLab é uma placa de interface desenvolvida pela equipe do Prof. Fabiano Fruett para organizar e simplificar a conexão de diversos periféricos em projetos de sistemas embarcados.

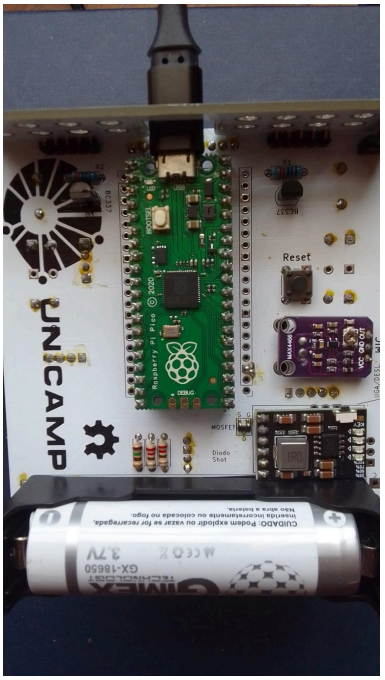
Inicialmente concebida para operar com microcontroladores RP2040 (presentes nas versões Raspberry Pi Pico, Pico H ou Pico W), a BitDogLab foi aprimorada em colaboração com o Prof. A. A. F. Quevedo. Essa evolução resultou em uma flexibilidade que permite acoplá-la, em uma de suas faces, tanto à Raspberry Pi Pico quanto à Black Pill. Dessa forma, a BitDogLab funciona como um *shield* versátil, compatível com ambas as plataformas, o que agiliza a prototipagem e a integração de componentes.

Na face da BitDogLab onde a placa de desenvolvimento proprietária é acoplada, ao lado da bateria, está localizado um gerenciador de energia. Esse componente permite carregar a bateria utilizando o mesmo conector mini USB empregado para programar o Raspberry Pi Pico. Além disso, ele gerencia o descarregamento da bateria e indica seu nível de carga através de quatro pequenos LEDs.

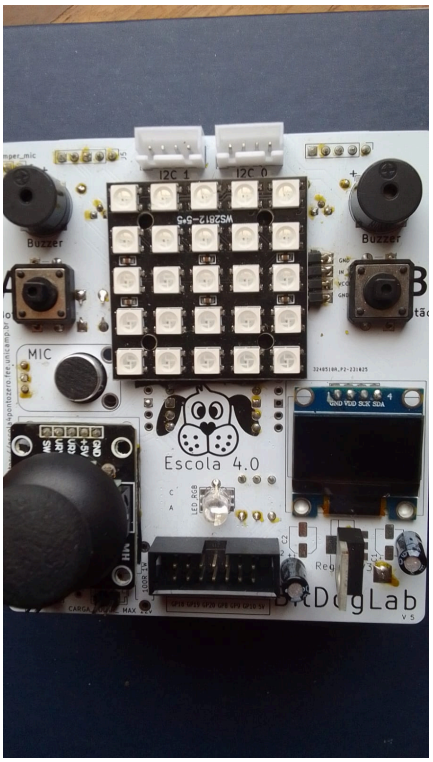
É importante notar, contudo, que não foi possível utilizar o pino de alimentação 3V3, localizado no lado oposto do conector USB-C da Black Pill, para carregar a bateria ou para alimentar o microcontrolador diretamente por ela. No caso da Black Pill, a alimentação do microcontrolador é sempre responsabilidade do computador ao qual está conectada.



BitDogLab com Black Pill



BitDogLab com Raspberry Pi Pico



Vista frontal de BitDogLab

Os periféricos embarcados na BitDogLab, todos de montagem em superfície (em inglês, *Surface Mount Device* – SMD) são:

- LED RGB SMD 5050/LED RGB com catodo comum 5mm em conformidade com a Diretiva de Restrição de Substâncias Perigosas (em inglês, *Restriction of Hazardous Substances* – RoHS).
- Matriz de 5x5 de LEDs RGB: SMD5050 WS2812.
- *Display* OLED I2C 0.96 polegadas 128×64: *driver* SSD 1306.
- Microfone com amplificador de áudio: CMA-4544PF-W.
- *Joystick* analógico 13x13mm: KY-023.
- Botões: chaves táteis 12 x 12 x 7.5 mm.
- *Buzzers* passivos: 80dB Externally Driven Magnetic 2.7kHz SMD, 8.5×8.5mm Buzzers RoHS.
- Conectores de expansão I2C compatíveis com RoHS**.**
- Conector de expansão de 14 pinos.
- Conector para painel solar.
- Conector para bateria externa.
- Painel compatível com garras jacaré ou parafusos.
- Circuito de gerenciamento de energia: IP5306.
- Bateria Li-Ion 18.650 3.800 mAh 3,7V recarregável.

A lista completa de materiais necessários para montagem de um *kit* BitDogLab está disponível no Google Drive. Com base na Tabela 9 (Pinagem) do *datasheet* de STM32F411 e no esquemático de BitDogLab, os pinos do RP2040 foram mapeados pelo Prof. Quevedo nos de STM32F411 para cada periférico montado na BitDogLab. Essa equivalência é detalhada na tabela a seguir.

BitDogLab	Pino no RP2040 (v 6.0)	Sinal do periférico	Pino no STM32F4	STM32F411
I2C0	GP0	SDA	PB7	I2C1
	GP01	SCL	PB6	
I2C1	GP02	SDA	PB8	I2C3
	GP03	SCL	PA8	
Conector de expansão (IDC)	GP08	IDC4	PB3	GPIO / UART1_RX
	GP28	IDC5	PB0	GPIO / ADC8 / MIC
	GP09	IDC6	PB4	GPIO / SPI1_MISO / SPI3_MISO
	GP04	IDC8	PA9	GPIO / UART1_TX
	GP17	IDC9	PC13	GPIO
	GP20	IDC10	PA0	GPIO / ADC0
	GP16	IDC11	PB14	GPIO / SPI2_MISO
	GP19	IDC12	PB15	GPIO / SPI2 MOSI
	GP18	IDC14	PB13	GPIO / SPI2 SCK
Microfone	GP28	Microfone	PB0	ADC8
2 botões	GP05	Botão A	PB12	GPIO / IRQ12
	GP06	Botão B	PA15	GPIO / IRQ15
2 *buzzers*	GP21	*Buzzer* A	PA1	GPIO / Timer2_CH2 /Timer5_CH2
	GP10	*Buzzer* B	PB5	GPIO / Timer3_CH2
Matriz de LEDs	GP07	Data In	PA10	SPI5_MOSI / Timer1_CH3
LED RGB	GP12	Anodo Red	PA2	GPIO / Timer2_CH3 /Timer5_CH3 / Timer9_CH1
	GP13	Anodo Green	PA7	GPIO / Timer1_CH1N /Timer3_CH2

	GP11	Anodo Blue	PA5	GPIO / Timer2_CH1
Display OLED	GP14	SDA	PB9	I2C2
	GP15	SCL	PB10	
Joystick	GP22	Switch	PA3	GPIO / IRQ3
	GP26	VRx	PA6	ADC6
	GP27	VRy	PB1	ADC9

Uma introdução prática a esses componentes, com os *firmwares* codificados em Python na Thonny IDE, está disponível no *site* da BitDogLab. Todos os materiais referentes ao projeto **BitDogLab** podem ser encontrados no repositório GitHub.

AMBIENTES DE DESENVOLVIMENTO INTEGRADO

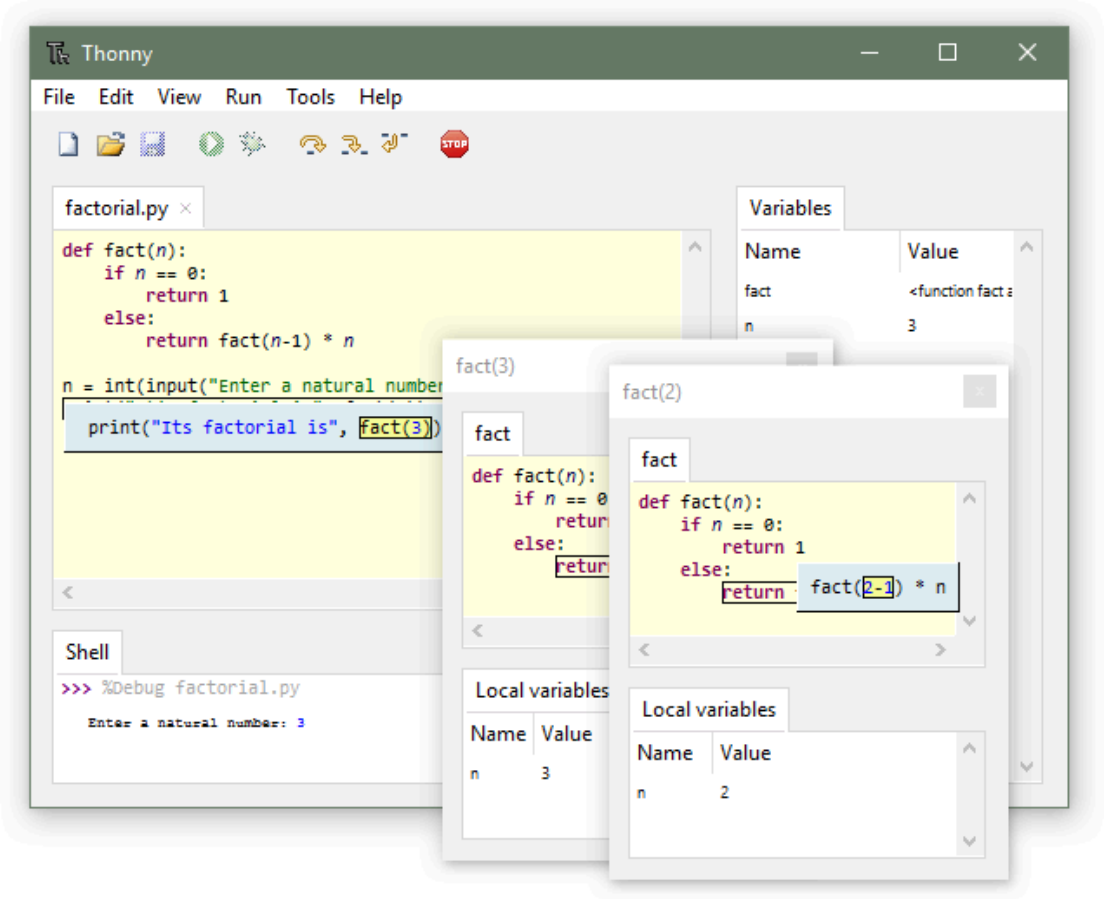
Um Ambiente de Desenvolvimento Integrado (IDE) é uma ferramenta essencial que unifica diversas funcionalidades de *software*, simplificando drasticamente o processo de programação e depuração. Para o desenvolvimento de *firmwares* em sistemas embarcados, a relevância de uma IDE é ainda maior: ela integra editores de código, compiladores, depuradores e, muitas vezes, ferramentas de configuração de *hardware* e *upload*, de documentação e versionamento, tudo em uma única interface coesa.

Tecnicamente, uma IDE acelera o ciclo de desenvolvimento ao automatizar tarefas repetitivas, como a compilação e o *flashing* do código no microcontrolador. Além disso, fornece recursos avançados de depuração, como *breakpoints* e inspeção de variáveis, que são cruciais para identificar e corrigir erros em tempo real no *hardware*. A completude das funções integradas em uma IDE pode ser um fator decisivo na escolha de um microcontrolador específico. Uma IDE bem suportada para uma determinada família de microcontroladores (como o STM32CubeIDE para STM32 ou a Arduino IDE para AVR) oferece uma experiência de desenvolvimento muito mais fluida e produtiva, reduzindo a curva de aprendizado e os desafios de integração de ferramentas avulsas.

Para o Raspberry Pi Pico, o Visual Studio Code (VS Code) é a IDE mais recomendada, especialmente para programação em C/C++. Sua grande flexibilidade e o vasto ecossistema de extensões (como C/C++ Extension Pack, CMake Tools e extensões de depuração) permitem uma integração perfeita com a *toolchain* do Pico (GCC para ARM e CMake), oferecendo depuração e um ambiente completo. Embora o Eclipse também possa ser configurado para o desenvolvimento com o RP2040, sua complexidade de configuração inicial, curva de aprendizado mais íngreme e maior consumo de recursos o tornam menos popular em comparação com a simplicidade e a adoção crescente do VS Code na comunidade.

Thonny IDE

A Thonny IDE é um ambiente de desenvolvimento integrado (IDE) leve e fácil de usar, especialmente popular entre iniciantes em Python e no universo de microcontroladores que suportam MicroPython, como o Raspberry Pi Pico. Sua simplicidade e interface intuitiva a tornam uma ferramenta excelente para aprender e prototipar rapidamente.



Fonte: Thonny.org

Instalar o Thonny é bastante direto: basta baixar o instalador para o seu sistema operacional (Windows, macOS, Linux) do *site* oficial e seguir os passos de instalação padrão. Uma vez instalado, a configuração para o Raspberry Pi Pico é igualmente simples. Navegue até "Ferramentas" -> "Opções" -> "Interpretador". Lá, selecione "MicroPython (Raspberry Pi Pico)" e a porta serial à qual seu Pico está conectado. O Thonny pode, inclusive, instalar automaticamente o *firmware* MicroPython no seu Pico, se necessário, facilitando ainda mais o primeiro uso.

Após a instalação da Thonny IDE, a implementação de *firmwares* em Python para o Raspberry Pi Pico se torna bastante fácil. Sua interface limpa e descomplicada permite que o desenvolvedor se concentre na programação, sem a necessidade de configurações complexas. A interface de linha de comando interativa REPL (do inglês *Read-Eval-Print Loop*) integrada oferece uma interação direta e em tempo real com o MicroPython executando no Pico, permitindo testar pequenos trechos de código e depurar interativamente ao ver resultados instantaneamente. O *upload* do código é simplificado, bastando um clique para enviar o *script* Python para a placa. Além disso, o Thonny facilita o desenvolvimento com a visualização de variáveis durante a execução e o gerenciamento

de arquivos diretamente no sistema de arquivos do Pico, otimizando a organização do projeto. Para uma introdução prática, acesse o tutorial simples de primeiros passos com a Thonny IDE para Raspberry Pi Pico no site de Embarcados.

Apesar das suas muitas vantagens, o Thonny apresenta restrições importantes no desenvolvimento de *firmwares* complexos para microcontroladores como o Raspberry Pi Pico. Ao executar código Python no Pico via Thonny, o interpretador MicroPython, já presente no microcontrolador, traduz os *bytecode*s Python para instruções nativas em tempo real. Essa tradução em tempo de execução impõe limitações cruciais na depuração. O Thonny não oferece depuração em nível de *hardware*, como ocorre com ferramentas C/C++ que utilizam interfaces como JTAG/SWD, junto com GDB e OpenOCD. Isso significa que ele não tem acesso direto aos registradores internos do microcontrolador nem à capacidade de parar a CPU em qualquer instrução de máquina. A depuração que o Thonny proporciona é mais lógica ou simbólica, focada no contexto do interpretador Python. Pode-se definir *breakpoints* no seu código Python e inspecionar variáveis no nível do *script*, mas sem visibilidade sobre o que o microcontrolador está realmente fazendo em termos de instruções de máquina ou como o MicroPython gerencia internamente os recursos. Por consequência, não é possível depurar *drivers* de periféricos ou código de baixo nível em C que faça parte do *firmware* do MicroPython, nem otimizar o uso de recursos em nível de *hardware* com essa ferramenta.

Além das restrições de depuração, o Thonny, por ser uma IDE mais simples e focada no aprendizado, também apresenta limitações em outras áreas relevantes para o desenvolvimento de *software* profissional. No que diz respeito ao suporte a documentação, embora permita comentários no código, não oferece ferramentas avançadas para a geração automática de documentação a partir do código-fonte, o que é essencial para projetos maiores. Da mesma forma, em relação ao versionamento de códigos, o Thonny não possui integração nativa ou robusta com sistemas como o Git.

Embora o Thonny seja uma excelente ferramenta para prototipagem rápida e fins educacionais com MicroPython, sua simplicidade inerente acarreta limitações significativas em depuração em nível de *hardware*, documentação formal e integração com sistemas de controle de versão. Para projetos mais ambiciosos, especialmente em C/C++, essas lacunas podem exigir o uso de outras ferramentas. Isso significa recorrer a *debuggers* físicos que utilizam SWD/JTAG, gerenciar a documentação e o versionamento do código manualmente, ou então mudar para IDEs mais completas que ofereçam todas essas funcionalidades integradas.

Visual Studio Code

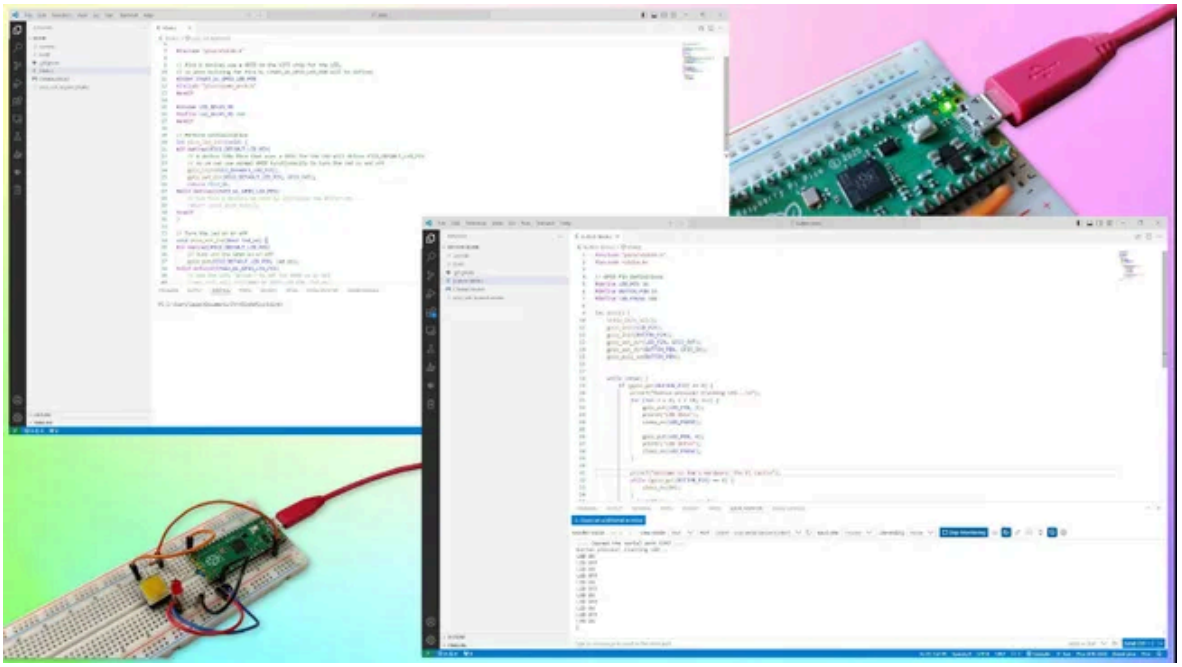
O Visual Studio Code (VS Code), um ambiente de desenvolvimento integrado multiplataforma e gratuito da Microsoft (disponível para Windows, MacOS e Linux), tem uma arquitetura baseada em extensões. Essa arquitetura é a chave para a sua flexibilidade. Sua natureza *open-source*, com código no GitHub, permite que a comunidade contribua com seu desenvolvimento, resultando em uma vasta gama de extensões e funcionalidades. Ele pode ser facilmente transformado em uma poderosa IDE para sistemas embarcados, adaptando-se às necessidades específicas do desenvolvimento de *firmwares*. Isso significa que ele abrange desde a escrita do código até a depuração física no microcontrolador, suportando diversas linguagens de programação (como

Python, Ruby, C/C++) e uma variedade de placas de desenvolvimento, incluindo o Raspberry Pi Pico.

Originalmente, o VS Code já se destaca como um promissor ambiente de suporte ao desenvolvimento de *software*, oferecendo um conjunto de funcionalidades essenciais:

- Edição Inteligente de Código: Com recursos como realce de sintaxe, autocompletar (IntelliSense para C/C++), refatoração de código e formatação automática, ele acelera a escrita e a manutenção do código.

- Versionamento Nativo: A integração profunda com o Git é um dos seus maiores trunfos. Sem a necessidade de *plugins* adicionais para as operações básicas, o VS Code permite gerenciar *commits*, *branches*, *pull*/*push* e resolver conflitos de forma visual e intuitiva, essencial para o trabalho em equipe.



Fonte: Tomshardware.

Embora não tenha uma ferramenta nativa como o Doxygen, extensões podem ser adicionadas para auxiliar na geração de documentação a partir de comentários estruturados, promovendo clareza no projeto e facilitando a colaboração. Para adicionar Doxygen ao VS Code, dois passos são necessários. Primeiro, instale a ferramenta Doxygen, baixando-a do site oficial e garantindo que esteja no PATH do sistema. Em seguida, instale uma extensão Doxygen no VS Code diretamente do Marketplace de Extensões, buscando por "Doxygen" para encontrar opções populares que auxiliarão na geração de comentários e na execução da ferramenta dentro da IDE. As mais comuns e úteis são Doxygen Documentation Generator e Doxygen Runner.

A flexibilidade do VS Code para sistemas embarcados reside em como ele expande suas capacidades. Com a instalação das extensões corretas (também detalhada em *datasheet* da Raspberry Pi Pico), ele se transforma em uma IDE completa para desenvolvimento de *firmwares*:

- Compilação e *Flashing*: Extensões como CMake Tools e integrações específicas para *toolchains* (como o GCC para ARM) permitem configurar, compilar e carregar o *firmware* no microcontrolador diretamente do ambiente.

- Programação de Microcontrolador: Ele se conecta a sistemas de *build* e *flash* que interagem com o *hardware* alvo, automatizando o processo de gravação do código.

- Depuração Física de *Firmwares*: Extensões como Cortex-Debug ou Native Debug se integram ao GDB (GNU *Debugger*) e a ferramentas como o OpenOCD[2]. Isso permite que o VS Code se comunique com *debuggers* de *hardware* externos (como o Raspberry Pi Debug Probe, ST-Link ou J-Link) via interfaces como SWD (do inglês *Serial Wire Debug*). Com isso, o desenvolvedor ganha controle total sobre a execução do *firmware* no microcontrolador, podendo definir *breakpoints*, executar o código passo a passo, inspecionar variáveis em tempo real e monitorar registradores.

Além disso, é necessário ter um console serial no VS Code para interagir com microcontroladores, permitindo visualizar mensagens de depuração (*logs*), receber dados de sensores ou enviar comandos para a placa. Embora o VS Code não venha com um console serial "nativo" como IDEs mais específicas de embarcados (tipo Arduino IDE ou Thonny IDE), é fácil adicionar essa funcionalidade por meio de extensões. Basta ir à aba de Extensões (Ctrl+Shift+X), procurar por "Serial Monitor" ou "Serial Port", instalar a extensão desejada e, em seguida, configurá-la pela Paleta de Comandos (Ctrl+Shift+P), selecionando a porta serial correta e definindo a velocidade de comunicação.

No VS Code, a visualização de dados e códigos é organizada de forma flexível e modular, permitindo que você personalize seu ambiente de trabalho. O conceito central é o *workspace*, que representa a pasta (ou pastas) de um projeto, onde todos os arquivos e configurações pertinentes são armazenados. Dentro desse *workspace*, o VS Code oferece diversas visualizações (em inglês, *views*) e painéis (em inglês, *panels*) que podem ser abertos, fechados e redimensionados conforme sua necessidade.

As visualizações, localizadas geralmente na barra lateral esquerda, são dedicadas a diferentes aspectos do desenvolvimento. Por exemplo, a visão "Explorer" exibe a estrutura de arquivos e pastas do seu projeto, enquanto a "Source Control" gerencia as operações Git, e a "Run and Debug" oferece controles para a depuração. Já os painéis, que tipicamente ficam na parte inferior da janela, abrigam ferramentas como o Terminal integrado, que permite executar comandos de linha; a aba "Problems", que lista erros e avisos de compilação; a "Output", para saídas de processos; e o "Debug Console", para interações durante a depuração.

Essa organização modular, onde se pode arrastar e soltar painéis, dividir o editor em múltiplas colunas ou linhas para visualizar vários arquivos simultaneamente, e alternar rapidamente entre as diversas visualizações, proporciona um controle granular sobre como os códigos e dados são apresentados, otimizando o fluxo de trabalho para diferentes tarefas de desenvolvimento. Ao longo do desenvolvimento, o VS Code distingue e gerencia diferentes "contextos" de trabalho que podem

[2] Quando um servidor GDB é integrado diretamente no *firmware* do *probe* de depuração, como é o caso do *Black Magic Probe* (BMP), não é necessário instalar OpenOCD, ou aplicativos equivalentes, no computador hospedeiro.

ser acessados através de diferentes visualizações e painéis organizados para uma finalidade específica, proporcionando uma experiência de desenvolvimento coesa:

- Contexto de Edição e Código Fonte: O editor principal e a visão de "Controle do Código-Fonte" (Git) são onde a maior parte do trabalho de escrita e versionamento acontece, com realce de sintaxe, autocompletar, e ferramentas de *diff*.

- Contexto de *Build*: Integrado com o sistema de *build* (ex: CMake), permite compilar o projeto, visualizar erros e avisos de compilação em um terminal dedicado.

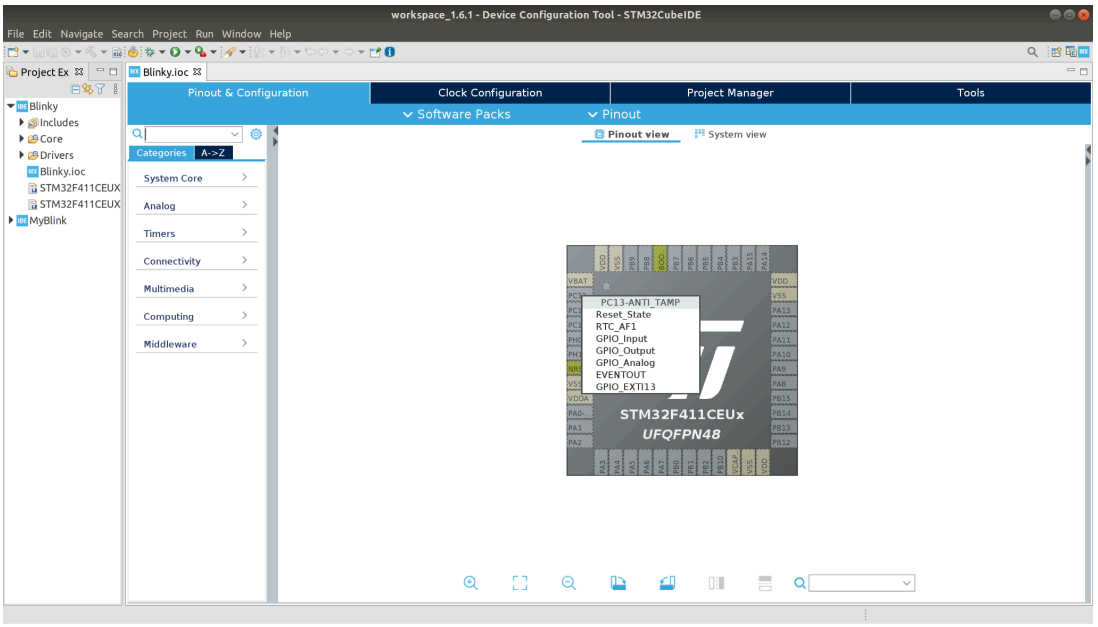
- Contexto de Depuração: Ativado quando um processo de depuração é iniciado, oferece painéis para *breakpoints*, variáveis, registradores e a pilha de chamadas, além do controle de execução (pausar, continuar, passo a passo). Esta visão interage diretamente com o *debugger* de *hardware*.

- Contexto de Terminal/Console Serial: Através de terminais integrados, o VS Code permite executar comandos de linha (como OpenOCD) e, com extensões de monitor serial, visualizar a saída do console serial do microcontrolador em tempo real, para *logs* e depuração básica da aplicação.

A capacidade de reunir todas as etapas do desenvolvimento embarcado em uma única IDE, com a flexibilidade de adicionar funcionalidades conforme a necessidade, faz do VS Code uma escolha extremamente poderosa e popular para o desenvolvimento de *firmwares* em C/C++.

STM32CubeIDE

A STM32CubeIDE é o ambiente de desenvolvimento de *software* desenvolvido pela STMicroelectronics para os microcontroladores por ela fabricados. Essa IDE é baseado na plataforma Eclipse, um ambiente de desenvolvimento multi-linguagem, gratuito e de código aberto, lançado sob os termos da *Eclipse Public License*. O Eclipse tem se consolidado como padrão em empresas que desenvolvem projetos de *software*, graças à sua arquitetura modular baseada em plug-ins. Esses plug-ins permitem adicionar ou expandir funcionalidades sem alterar o núcleo do sistema. É possível, por exemplo, adicionar novas linguagens de programação, ferramentas de teste, depuração e suporte a famílias específicas de microcontroladores.



Fonte: DanielleThurow.

Um conceito central no Eclipse é o *workspace* (espaço de trabalho), que organiza os projetos e configurações do usuário. Cada desenvolvedor pode ter seu próprio workspace, com múltiplos projetos, e a IDE permite importar e exportar projetos ou workspaces completos, facilitando a colaboração e a continuidade do trabalho. Outro conceito importante é o de perspectiva, que organiza janelas e ferramentas conforme a tarefa em execução. Neste curso, utilizaremos três perspectivas principais:

- Inicialização: configuração dos periféricos do microcontrolador e parâmetros de inicialização.
- Programação: edição de código, estrutura de arquivos, mensagens de compilação e navegação no projeto.
- Depuração: execução passo-a-passo, visualização de variáveis, código assembly e comunicação via terminal serial integrado.

A STM32CubeIDE, por ser baseado no Eclipse, oferece suporte nativo ao sistema de controle de versões Git através do *plug-in* EGit, já integrado na maioria das suas distribuições. Com o EGit, pode-se inicializar um repositório local para um projeto, clonar repositórios remotos (como GitHub ou GitLab), e realizar operações como *commit*, *push*, *pull* e *merge* diretamente na IDE. Além disso, a visualização do histórico de alterações e a resolução de conflitos de código contam com suporte gráfico, facilitando o processo. Para acessar esses recursos na STM32CubeIDE, basta clicar com o botão direito no projeto, selecionar *Team* > *Share Project*, escolher Git como sistema de versionamento e configurar um repositório local ou conectar-se a um remoto. Posteriormente, a perspectiva Git (acessível via *Window* > *Perspective* > *Open Perspective* > *Other...* > *Git*) centraliza todas as ferramentas de versionamento.

Para automatizar a geração de documentação e ter um console serial para interagir com o microcontrolador, é necessário instalar dois *plugins* específicos na STM32CubeIDE:

- Eclox (integração com Doxygen): O Eclox é um front-end para o gerador de documentação Doxygen. Ele permite criar documentação de código diretamente no Eclipse por meio de uma interface gráfica simples. Com isso, os desenvolvedores poderão documentar funções, estruturas e arquivos do projeto, gerando documentação técnica automatizada — prática fundamental em ambientes profissionais e acadêmicos.
- TM Terminal (terminal serial integrado): O TM Terminal é um terminal de linha de comando que pode ser embutido no Eclipse. Ele permite a comunicação serial com o microcontrolador diretamente na IDE, facilitando a depuração de aplicações que utilizam UART, por exemplo. Assim, não é mais necessário alternar entre a IDE e programas externos como PuTTY ou TeraTerm.

A instalação de *plugins* como o Eclox e o TM Terminal na STM32CubeIDE é um processo simples, realizado através do *Eclipse Marketplace*. Para isso, acesse o *menu Help* na barra superior da STM32CubeIDE e selecione *Eclipse Marketplace*. Na caixa de busca, digite os nomes dos *plugins* desejados, Eclox e TM Terminal. Em seguida, clique em *Install* e siga as instruções para concluir a instalação.

Seguindo o passo a passo apresentado no tutorial elaborado pelo Prof. Quevedo, consegue-se criar um projeto, configurar os registradores por um editor gráfico e gerar os códigos em C usando STM32CubeMX dentro da STM32CubeIDE.

FERRAMENTAS PARA DESENVOLVIMENTO DE PROJETOS

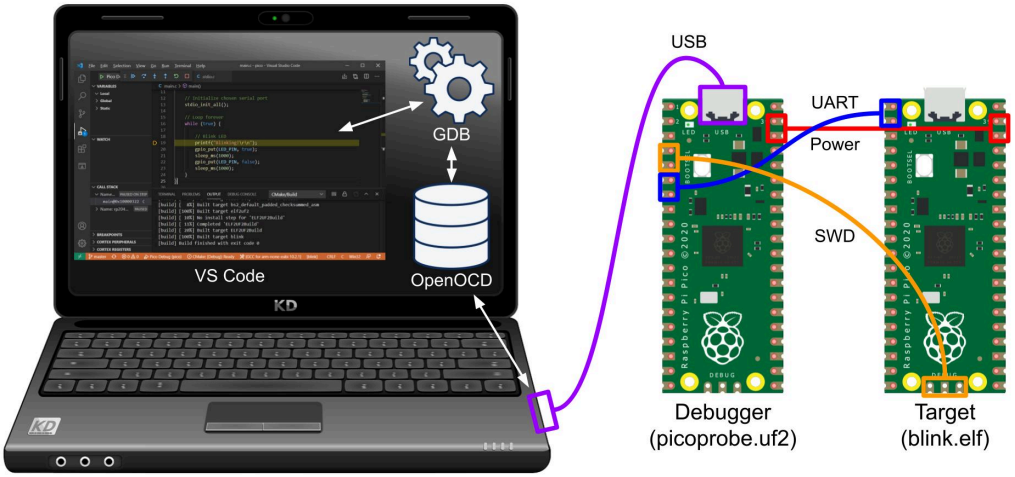
Nesta seção, vamos aprofundar nas ferramentas essenciais que você utilizará para desenvolver seus projetos neste semestre. Focaremos nos detalhes práticos de como construir, configurar e empregar cada uma dessas ferramentas em seu fluxo de trabalho. Nosso objetivo é proporcionar um primeiro contato prático e direto, facilitando a implementação de projetos complexos, desde a depuração de *hardware* até a documentação do código e o versionamento de todas as suas alterações.

Probes de depuração DIY

Uma alternativa de baixo custo para a depuração de sistemas embarcados é o uso de um segundo microcontrolador configurado como um *debug probe* DIY (do inglês *Do It Yourself*). Nessa configuração, esse microcontrolador atua como uma ponte física entre os pinos DEBUG do microcontrolador-alvo, como um Raspberry Pi Pico, e a porta USB do computador. Essa solução funciona como uma substituição acessível a *debug probes* comerciais, como o Debug Probe da Raspberry Pi Foundation, o J-Link da SEGGER ou o ULINK da Keil. Embora ofereça apenas recursos básicos, elimina a necessidade de investir em *hardware* dedicado mais caro.

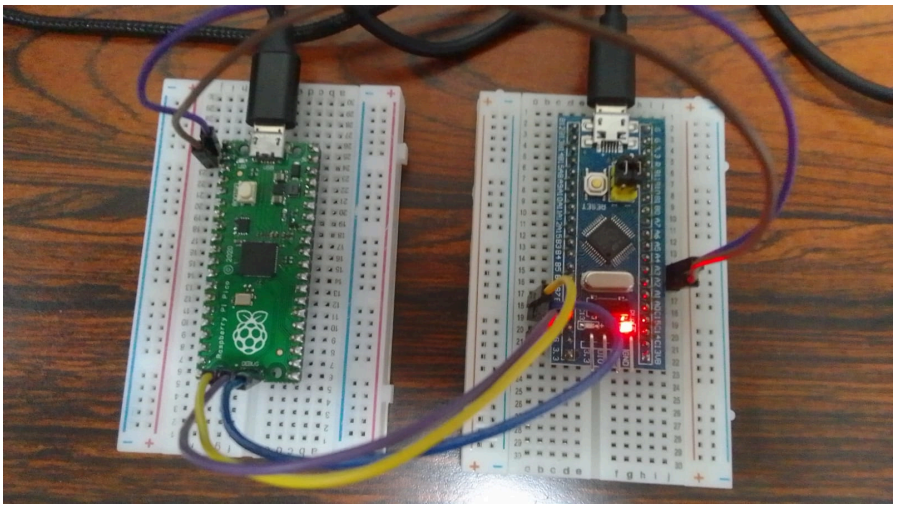
Nesse arranjo DIY, o microcontrolador-probe (que pode ser outra Raspberry Pi Pico, uma Blue Pill ou outro STM32) não executa o *firmware* em desenvolvimento. Em vez disso, ele é gravado com um *firmware* especial que implementa o protocolo CMSIS-DAP, transformando-o em um adaptador USB-para-SWD. Como exemplo, a figura que se segue ilustra um Raspberry Pi Pico configurado

como *probe*, com o *firmware* picoprope.uf2 (baseado em CMSIS-DAP) carregado. Ele se conecta à porta USB do computador de um lado e, do outro, aos três pinos DEBUG do Raspberry Pi Pico alvo (SWCLK, GND e SWDIO). Essa conexão é realizada através dos pinos GPIO específicos do microcontrolador-probe configurados para SWD: GP2 (SWCLK), GND e GP3 (SWDIO). Adicionalmente, para que o microcontrolador-alvo possa se comunicar com o console serial do computador, deve-se conectar os pinos GP4 e GP5 (TX e RX do UART1 do probe) com os pinos GP1 e GP0 (RX e TX do UART0 do microcontrolador-alvo), respectivamente.



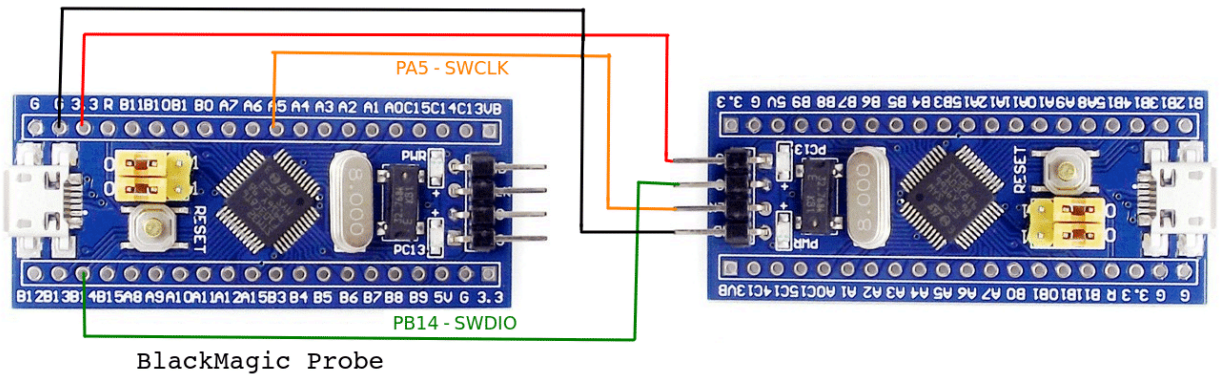
Fonte: Maker.io.

DAPLink é um *firmware* de código aberto, criado pela ARM mbed, que implementa o protocolo CMSIS-DAP. Seu repositório oficial não só oferece o código-fonte desse *firmware*, mas também disponibiliza funcionalidades práticas como programação via "arrastar e soltar" (*Mass Storage Device*), uma porta serial virtual da CDC (do inglês *Communication Device Class*) e uma interface de depuração HID (do inglês *Human Interface Device*). O DAPLink original da ARM mbed é otimizado para microcontroladores NXP e outros frequentemente encontrados em placas de desenvolvimento. No entanto, para placas como a Blue Pill (que utiliza o STM32F103C8T6 da STMicroelectronics), é necessário buscar uma implementação ou "fork" do DAPLink que seja especificamente compatível com o STM32F103. Um exemplo notável é o projeto dap42 no GitHub, amplamente conhecido por oferecer suporte explícito à Blue Pill. A figura a seguir demonstra a conexão de uma Raspberry Pi Pico a um computador, utilizando uma Blue Pill como *probe* DAPLink com o *firmware* também disponível no GitHub. Nesse arranjo, o *firmware* configura os pinos PB9, PB8, PA2 e PA3 da Blue Pill para os sinais SWDIO, SWCLK, TX e RX, respectivamente. É essencial que cada microcontrolador tenha sua própria alimentação, embora o pino VBUS da Raspberry Pi Pico possa, opcionalmente, ser alimentado diretamente pela saída de 5V da Blue Pill para maior conveniência.



No lado do computador, é instalado o OpenOCD (do inglês *Open On-Chip Debugger*). O OpenOCD é um servidor de depuração que atua como um intermediário: ele traduz os comandos de depuração da IDE (como VS Code ou Eclipse, que usam o protocolo GDB) para um protocolo de depuração de baixo nível (como o CMSIS-DAP ou o protocolo proprietário no ST-Link) e então se comunica com o microcontrolador-*probe* via USB, usando o protocolo CMSIS-DAP, por exemplo. Por fim, o microcontrolador-*probe* converte esses comandos para a interface de depuração de *hardware* (como SWD ou JTAG) e os envia ao microcontrolador alvo para depuração. Dessa forma, é possível configurar *breakpoints*, inspecionar variáveis em tempo real e realizar a execução do código passo a passo, facilitando a identificação e correção de erros.

Podemos ainda transformar uma Blue Pill em um *Black Magic Probe* (BMP) apenas carregando o *firmware* BMP nela; o processo de geração desse *firmware* é detalhado neste artigo. O BMP é compatível com o RP2040. O grande diferencial do BMP, comparado a outros *firmwares* baseados em CMSIS-DAP, como DAPLink, é a integração de um servidor GDB diretamente no próprio dispositivo. Isso elimina a necessidade de instalar o OpenOCD ou qualquer *software* similar no computador hospedeiro. Um cliente GDB para microcontroladores instalado no computador já é suficiente para iniciar a depuração. Além disso, o BMP atua como ponte serial entre o alvo e o console, usando os pinos PA3 (RX) e PA2 (TX) do módulo UART2 para se conectar, por exemplo, aos pinos TX e RX de um módulo UART do Blue Pill alvo.



Fonte: Embarcados.

MicroPython para Pico

O MicroPython para o Pico é uma porta de entrada para o mundo da eletrônica e programação embarcada. Ele é um interpretador da linguagem Python 3, especificamente desenvolvido para o microcontrolador RP2040, o cérebro por trás da placa Raspberry Pi Pico. Em vez de exigir linguagens de baixo nível, como C/C++, o MicroPython atua como uma ponte, permitindo que se use Python para interagir diretamente com os recursos de *hardware* do *chip*. Isso torna a prototipagem mais rápida e a curva de aprendizado muito mais suave, abrindo as portas para entusiastas que desejam criar projetos de eletrônica de forma eficiente e descomplicada.

Uma das principais vantagens do uso do MicroPython em microcontroladores como o Pico é sua biblioteca nativa chamada `machine`. Essa biblioteca funciona como uma interface direta com os recursos físicos do microcontrolador, permitindo que o programador acesse e controle funcionalidades de *hardware* de forma prática e eficiente, utilizando comandos simples da linguagem Python. Cada componente de *hardware*, como pinos GPIO, barramentos de comunicação, conversores analógico-digitais, temporizadores, entre outros, é representado por uma classe específica dentro dessa biblioteca. Dessa forma, o MicroPython abstrai a complexidade da manipulação direta de registradores, sem abrir mão do controle de baixo nível necessário em projetos embarcados.

A classe mais básica da biblioteca `machine` é a classe `Pin`. Ela representa os pinos de propósito geral (GPIO) do microcontrolador, permitindo configurá-los como entradas ou saídas digitais. Essa classe é diretamente responsável por acessar e manipular os registradores de GPIO do *chip*. Por exemplo, para configurar um pino como saída, basta instanciá-lo com o número correspondente ao GPIO físico: `Pin(25, Pin.OUT)` configura o GPIO25 como saída, o qual, no caso do Pico, acende o LED *on-board*. Além disso, essa mesma classe `Pin` é utilizada por outras classes, como `I2C`, `SPI` e `PWM`, que exigem a definição dos pinos que serão usados em suas operações.

As classes `I2C` e `SPI` são utilizadas para comunicação serial com dispositivos externos, como sensores, displays e outros microcontroladores. Essas interfaces exigem que determinados pinos físicos sejam atribuídos a funções específicas, como SDA e SCL para I2C, ou MOSI, MISO e SCK para SPI. No MicroPython, isso é feito passando objetos `Pin` como parâmetros no momento da criação da interface. Por exemplo, `I2C(0, scl=Pin(17), sda=Pin(16))` configura a interface I2C 0 utilizando os GPIOs 17 e 16. Essas classes dependem da classe `Pin` para definir quais pinos físicos serão usados na comunicação, evidenciando a interdependência entre os módulos da biblioteca `machine`.

Outro recurso importante é a geração de sinais PWM (modulação por largura de pulso), implementada pela classe `PWM`. Esse recurso é amplamente utilizado no controle de motores, brilho de LEDs e servomecanismos. A configuração de PWM também parte da definição de um pino específico, como em `PWM(Pin(15))`, onde o GPIO 15 é configurado para gerar um sinal PWM. O ciclo de trabalho e a frequência do sinal podem ser ajustados facilmente por métodos dessa classe, o que oferece grande flexibilidade em aplicações de controle.

A leitura de sinais analógicos é feita por meio da classe `ADC`, que permite acessar os conversores analógico-digitais do microcontrolador. Essa classe também utiliza diretamente os GPIOs dedicados a funções analógicas. Um exemplo seria `ADC(26)`, que configura o GPIO 26 como entrada analógica. A leitura de valores pode ser feita com métodos como `read_u16()`, retornando a representação digital de uma tensão de entrada.

Temporizadores são controlados pela classe `Timer`, que permite executar funções de forma periódica ou após um determinado intervalo de tempo. Eles são especialmente úteis para aplicações que exigem tarefas temporizadas com precisão, como geração de interrupções ou delays não bloqueantes. Diferentemente das outras classes, os temporizadores não requerem a configuração de pinos, pois operam internamente no microcontrolador.

A biblioteca `machine` do MicroPython ainda se destaca por oferecer duas formas de gerenciar protocolos de comunicação, atendendo a diferentes necessidades de flexibilidade e desempenho. A comunicação por *hardware* utiliza os circuitos dedicados no *chip*, como os controladores I2C e SPI. Por serem otimizados em nível de *hardware*, esses protocolos são a opção mais rápida e eficiente. No entanto, o Pico tem um número limitado desses controladores, dois para I2C e dois para SPI. Cada um está vinculado a um conjunto fixo de pinos GPIO. Por exemplo, o controlador I2C0 só funcionará com os pinos designados para ele. Tentar usar um pino que não faça parte desse conjunto resultará em um erro, pois o MicroPython não conseguirá encontrar o periférico de *hardware* correspondente.

Em contraste, os protocolos por *software*, conhecidos como *bit-banging*, "emulam" um protocolo de comunicação usando apenas os pinos de propósito geral (GPIO) da placa. No MicroPython, as classes de comunicação via *software* tem o prefixo `Soft`, como nas classes `SoftI2C` e `SoftSPI`. Como essas classes não depende de periféricos de *hardware*, ela oferece uma flexibilidade muito maior: pode-se usar praticamente qualquer pino GPIO disponível para as linhas de comunicação. A principal desvantagem é que essa abordagem é menos eficiente e pode ser mais lenta, pois o processador precisa gerenciar cada *bit* da comunicação manualmente.

Um aspecto essencial na utilização dos pinos do Pico é a chamada **multiplexação de pinos**. Isso significa que um mesmo pino físico pode desempenhar diferentes funções, como GPIO, PWM, ADC, I2C, SPI, UART, entre outras, dependendo de como ele é configurado no código. Por padrão, um pino é inicializado como GPIO, mas ao ser instanciado dentro de uma classe como `PWM` ou `I2C`, sua função interna muda automaticamente. Por exemplo, o pino físico GP0 pode funcionar como saída digital, como SDA para I2C, ou como saída PWM, dependendo da forma como foi instanciado. Essa flexibilidade é possível porque o microcontrolador RP2040 do Pico possui um sistema interno de seleção de função para cada pino, controlado pelos registradores do chip. O MicroPython cuida da configuração correta desses registradores de forma automática ao instanciar cada objeto da biblioteca `machine`.

No entanto, é importante que o programador tenha atenção ao mapa de pinos e às funções alternativas disponíveis para cada GPIO, pois a tentativa de usar o mesmo pino para múltiplas funções simultaneamente pode gerar conflitos. Para isso, recomenda-se sempre consultar o pinout

oficial do Raspberry Pi Pico e a referência rápida do MicroPython para o RP2040, que traz exemplos práticos e explicações detalhadas sobre cada classe e método.

Documentação de *firmware* com Doxygen

Manter a documentação atualizada e precisa é um desafio, mas ferramentas como o Doxygen simplificam esse processo, automatizando a geração de documentos a partir de comentários em seu próprio código. O Doxygen é uma ferramenta de documentação predominantemente da interface de código-fonte amplamente utilizada que gera documentação em diversos formatos (HTML, PDF via LaTeX, XML, etc.) a partir de blocos de comentários especiais dentro do seu código. O Doxygen não apenas extrai descrições, mas também informações sobre a estrutura do código, como dependências de arquivos, hierarquias de classes e fluxos de chamadas de funções.

Para documentar um arquivo de código em C, deve-se adicionar um bloco de comentário no cabeçalho do arquivo. Este bloco geralmente começa com `/**` ou `/*!` e pode incluir *tags* como `\file` ou `@file` para descrever o propósito do arquivo, autor, data e uma breve descrição (`\brief`) seguida por detalhes mais extensos. Por exemplo:

```
/**
 * @file main.c
 * @brief Este arquivo contém a lógica principal do sistema de controle de
motor.
 * @author Seu Nome
 * @date 2024-07-17
 * @details Este módulo gerencia a inicialização do hardware, o loop principal
 * e as interrupções do sistema.
 */
```

Para documentar variáveis de qualquer tipo de dado, o ideal é posicionar um bloco de comentário Doxygen imediatamente acima da sua declaração. Dentro desse bloco, recomenda-se descrever claramente o propósito da variável, suas unidades (se aplicável), e quaisquer restrições ou faixas de valores que ela possa assumir.

```
/**
 * @var system_state
 * @brief Variável global que armazena o estado atual do sistema.
 * @details Os valores possíveis incluem STATE_IDLE, STATE_RUNNING, STATE_ERROR.
 */
volatile uint8_t system_state;
```

A documentação da interface de funções é onde o Doxygen realmente se destaca. Ele descreve o que a função faz, seus parâmetros e o que a função retorna. *Tags* comuns incluem `\brief` (breve descrição), `\param` (para cada parâmetro, explicando seu propósito e, se for o caso, suas unidades ou faixas de valores), e `\return` (para descrever o valor de retorno da função).

```
/**
 * @brief Inicializa os periféricos GPIO para controle do motor.
 * @param[in] port_config Estrutura com as configurações de porta (ex: GPIOA,
GPIOB).
 * @param[in] pin_mask Máscara de bits para os pinos a serem configurados.
 * @return void
```

```
 * @details Esta função configura os pinos especificados como saídas push-pull
com velocidade alta,
 * ativando os clocks necessários.
 */
void init_motor_gpios(GPIO_TypeDef *port_config, uint16_t pin_mask);
```

O controle sobre o resultado da documentação gerada pelo Doxygen é exercido principalmente por meio do seu arquivo de configuração, o Doxyfile. Para começar, deve-se gerar um Doxyfile padrão diretamente no terminal utilizando o comando `doxygen -g`. Este arquivo de texto permite um controle detalhado de praticamente todos os aspectos do processo de geração da documentação. Para definir o formato dos documentos de saída, basta ajustar as tags relevantes; por exemplo, configure `GENERATE_HTML = YES` para obter saída em HTML, `GENERATE_LATEX = YES` para gerar arquivos LaTeX (que podem ser subsequentemente compilados para PDF), e `GENERATE_RTF = NO` se não desejar o formato RTF. Além disso, o Doxyfile permite que você customize o nome do projeto (`PROJECT_NAME`), especifique o diretório onde a documentação será gerada (`OUTPUT_DIRECTORY`), e indique os arquivos de entrada (`INPUT`) dos quais o Doxygen extrairá os comentários para a documentação.

Uma característica do Doxygen é sua capacidade de integrar aplicativos de terceiros para a geração de diagramas, o que é útil para visualizar arquiteturas de *software* e fluxos de comunicação em projetos. É importante notar que esses aplicativos precisam ser instalados separadamente. Para evitar problemas de caminho e execução, recomenda-se instalá-los em pastas cujos nomes não contenham caracteres especiais ou espaços em branco. Entre essas ferramentas, destacam-se: Graphviz (DOT) e Mscgen. O Graphviz (DOT) gera uma variedade de diagramas, incluindo gráficos de dependência de inclusão, gráficos de chamadas de funções, diagramas de herança de classes e gráficos de colaboração. Para habilitar essa funcionalidade, deve-se definir `HAVE_DOT = YES` no seu Doxyfile e, opcionalmente, especificar o caminho para o executável do Graphviz (por exemplo, `DOT_PATH = /caminho/para/bin/dot`) caso ele não esteja no PATH do sistema. Por sua vez, Mscgen cria diagramas de sequência de mensagens (em inglês, *Message Sequence Charts* – MSC), que são excelentes para ilustrar interações entre diferentes componentes ou entidades do sistema. Ao inserir o código MSC diretamente nos comentários do Doxygen, ele será renderizado na documentação final. No Doxyfile, certifique-se de que a variável `MSCGEN_PATH` aponte corretamente para o executável do mscgen, e que opções como `CALL_GRAPH = YES` ou outras configurações de gráfico relevantes estejam ativadas, dependendo do tipo de diagrama desejado. A sintaxe para incorporar um diagrama MSC diretamente em seu comentário Doxygen é `@msc ... @endmsc`.

Para aprimorar o uso do Doxygen na documentação dos projetos em C, pode-se encontrar mais detalhes e guias completos disponíveis *online*. É importante notar que o Doxygen não se restringe apenas ao C/C++. Ele é também uma ferramenta eficaz para documentar códigos em Python, por exemplo, usando uma sintaxe de comentários muito similar. A principal diferença reside na forma como os comentários são estruturados para documentar elementos específicos de Python, como classes, métodos e módulos, que se alinham melhor com as convenções de *docstrings* da linguagem. Contudo, os princípios de usar blocos especiais de comentários (`"""` ou `'''` em Python) e *tags*

como `@param` e `@return` permanecem os mesmos, garantindo a mesma capacidade de extrair descrições e gerar diagramas de dependência estrutural.

A integração do Doxygen com ambientes de desenvolvimento modernos é bastante eficiente. Existem extensões no VS Code Marketplace, como "Doxygen Documentation Generator" ou "Doxygen Runner", que facilitam a criação de blocos de comentários Doxygen automaticamente e permitem acionar a geração da documentação diretamente da IDE. Após configurar seu Doxyfile, você pode usar a extensão para executar o Doxygen e até mesmo visualizar a saída HTML dentro do próprio VS Code. Isso agiliza o ciclo de documentação, permitindo que o desenvolvedor mantenha o foco no código. Por ser baseado no Eclipse, a STM32CubeIDE não possui uma integração Doxygen tão nativa quanto o VS Code com suas extensões, mas é perfeitamente possível utilizá-lo. A abordagem mais comum é instalar o plugin eclox através de STM32CubeIDE Marketplace.

Versionamento com GitLab

É introduzido o uso do GitLab da Unicamp (disponível para todos os alunos em gitlab.unicamp.br). Esta plataforma não é apenas um repositório para guardar o código, mas um ecossistema completo que integra o controle de versão Git com funcionalidades de gerenciamento de projetos, pipelines de CI/CD (Integração Contínua/Entrega Contínua) e muito mais. A familiaridade com o GitLab da Unicamp, em conjunto com ambientes de desenvolvimento integrados como o Visual Studio Code (VS Code) e a STM32CubeIDE, capacita os estudantes a desenvolverem seus projetos de forma organizada e colaborativa.

Para aproveitar ao máximo o GitLab, é vital compreender os conceitos fundamentais do Git. O coração de todo projeto Git é o repositório, um diretório que armazena todas as versões e o histórico de um projeto. As alterações que você faz em seus arquivos não são salvas automaticamente no histórico; elas primeiro passam pela área de *staging*, um espaço intermediário onde você seleciona e agrupa as modificações que deseja registrar. O ato de registrar essas alterações agrupadas no histórico do repositório é chamado de *commit*. Cada *commit* é um "instantâneo" do seu projeto em um dado momento, sempre acompanhado de uma mensagem descritiva que explica o que foi modificado. Para trabalhar em equipe ou sincronizar o trabalho entre diferentes máquinas, entra em cena o repositório remoto, no nosso caso, o GitLab da Unicamp. Este repositório atua como a "fonte da verdade" do projeto, hospedando a versão principal e mais atualizada.

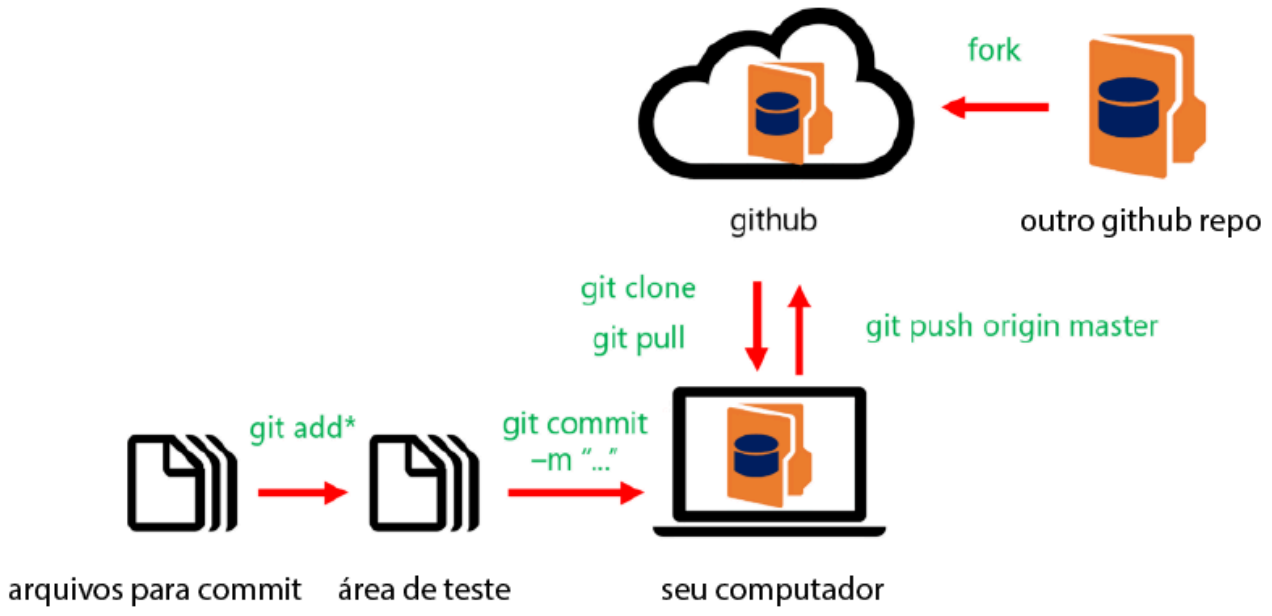
Com o repositório remoto em mente, dois comandos se tornam essenciais: *pull* e *push*. O comando *pull* é usado para baixar as alterações mais recentes do repositório remoto para o seu repositório local, garantindo que sua cópia do projeto esteja sempre atualizada com o trabalho dos colegas. Por sua vez, o comando *push* é a forma de enviar seus *commits* locais para o repositório remoto, compartilhando suas contribuições com a equipe. Dominar esses conceitos e operações é o primeiro passo para uma colaboração eficiente em qualquer projeto de engenharia.

Para começar a usar o Git, o primeiro passo é clonar um repositório existente. Isso cria uma cópia local do projeto no seu computador. No terminal, você usaria o comando `git clone <URL_do_repositorio>`, onde `<URL_do_repositorio>` seria o endereço do projeto no

gitlab.unicamp.br. Este endereço pode ser consultado na opção avançada da pasta que contém o projeto. Por exemplo: `git clone https://gitlab.unicamp.br/projeto.git`.

Depois de fazer alterações nos arquivos, precisa adicioná-los à área de *staging* dentro da pasta do projeto clonado usando `git add <nome_do_arquivo>` ou `git add .` para adicionar todas as modificações no diretório atual. Em seguida, faz-se "commit" dessas alterações com `git commit -m "mensagem descritiva do commit"`. Uma vez atualizado localmente, para enviar essas alterações para o GitLab da Unicamp, usa-se `git push origin main`, onde `origin` é o nome padrão que o Git dá ao repositório remoto principal quando o projeto é clonado pela primeira vez. Isso atualiza a versão do ramo principal `main` remota com as alterações feitas localmente em `main`. Para se manter atualizado com as alterações de outros colaboradores, antes de começar a trabalhar, é boa prática puxar as atualizações com `git pull origin main`.

No entanto, é recomendável e uma prática padrão no desenvolvimento de *software* e *hardware* digital trabalhar sempre em uma ramificação (em inglês *branch*) separada de `main`, realizando o *merge* com a *branch* `main` apenas quando seu trabalho estiver completamente validado. Essa abordagem é fundamental por diversas razões. Primeiramente, ela garante o isolamento do desenvolvimento: a *branch* `main` mantém-se estável e funcional, sem ser comprometida por alterações em andamento, permitindo que se experimente novas funcionalidades ou correções de *bugs* em um ambiente seguro. Se algo der errado, a *branch* pode ser descartada sem impactar o projeto principal. Além disso, essa prática otimiza a colaboração eficiente. Vários desenvolvedores podem trabalhar simultaneamente em funcionalidades distintas sem interferência direta. A criação de "*Merge Requests*" facilita a revisão de código, assegurando a qualidade, identificando *bugs* e promovendo o compartilhamento de conhecimento antes da integração na `main`. O controle de qualidade também é aprimorado, pois a *branch* de uma funcionalidade pode passar por testes automatizados (CI/CD) ou manuais sem bloquear o *pipeline* de entrega da `main`. Por fim, um histórico claro na `main`, onde cada *merge* representa uma funcionalidade ou correção validada, simplifica a auditoria e a compreensão da evolução do projeto.



Fonte: Dankicode.

Para implementar essa metodologia, use o seguinte fluxo de trabalho:

- Crie uma nova *branch*: Inicie seu trabalho criando uma nova ramificação para a funcionalidade ou correção que está desenvolvendo com `git checkout -b nome_da_branch_ramificada`.
- Desenvolva e faça commits: Realize suas alterações no código, adicione novos arquivos e faça *commits* frequentes para registrar progresso usando `git add .` e `git commit -m "Sua mensagem descritiva"`.
- Mantenha-se atualizado (opcional, mas recomendado): Periodicamente, incorpore as últimas atualizações da *branch* `main` para a *branch*, minimizando potenciais conflitos futuros. Para isso, pode-se alternar para a `main` (`git checkout main`), puxar as atualizações (`git pull origin main`), voltar para branch ramificada (`git checkout nome_da_branch_ramificada`) e então fazer o *merge* (`git merge main`) ou *rebase* (`git rebase main`) das mudanças.
- Envie atualizações da *branch* para o remoto: Compartilhe o progresso da sua ramificação com o repositório remoto utilizando `git push origin nome_da_branch_ramificada`.
- Crie um *merge request* (MR) no GitLab: No GitLab, abra um *merge request* da *branch* ramificada para a `main`, onde o código será revisado por outros membros da equipe e testes automáticos podem ser executados.
- Realize o *merge* na `main`: Após a validação, revisão e aprovação do MR pelos aprovadores elegíveis, a *branch* será incorporada à `main`. É uma boa prática então deletar a *branch* ramificada (local e remotamente) para manter o repositório organizado.

Para aprimorar o uso do Git e do GitLab nos projetos, pode-se encontrar mais detalhes e guias completos disponíveis *online*.

O Visual Studio Code (VS Code) é um ambiente de desenvolvimento amplamente utilizado, e sua integração com o Git e o GitLab é excelente. A extensão GitLab Workflow, disponível no *Marketplace* do VS Code, é uma versão oficial de GitLab para VS Code. Após instalar a extensão GitLab Workflow, é preciso configurá-la. Isso geralmente envolve autenticar-se com usuário e senha do `gitlab.unicamp.br`, ou usando um Token de Acesso Pessoal (em inglês *Personal Access Token* - PAT) que pode ser gerado nas configurações do perfil no GitLab (*Settings* > *Access Tokens*). Com a extensão configurada, pode-se visualizar e gerenciar *Issues*, *Merge Requests*, e o *status* dos *Pipelines* CI/CD diretamente na interface do VS Code, sem precisar alternar para o navegador. As operações básicas de *commit*, *pull* e *push* também podem ser realizadas através da interface visual do VS Code, na aba de Controle do Código-Fonte, tornando o processo mais intuitivo.

A STM32CubeIDE, uma ferramenta essencial para quem trabalha com microcontroladores STM32, oferece também uma excelente integração com o Git. Para isso, a melhor opção é o *plugin* EGit (do inglês *Git Integration for Eclipse*), que se pode encontrar e instalar diretamente pelo *Marketplace* da própria IDE. Para instalá-lo, é bem simples: vá em `Help` no menu superior, clique em `Eclipse`

`Marketplace...`, procure por EGit, e siga os passos para instalar. Após a instalação, é necessário reiniciar a STM32CubeIDE para que as mudanças tenham efeito. Uma vez instalado, o EGit proporciona suporte completo a todas as operações Git, como fazer *commits*, criar e gerenciar *branches*, realizar *push* e *pull* de alterações, e até mesmo resolver conflitos, tudo isso sem que se precise sair do ambiente da IDE para usar o terminal externo. Ele realmente simplifica o controle de versão dos projetos STM32.

Guia de uso do GitLab, Doxygen e Thonny no Windows

Este guia detalha o procedimento para alunos desenvolverem a primeira versão do *firmware* de seus projetos em Python, utilizando o GitLab da Unicamp em conjunto com a IDE Thonny e o Doxygen para documentação no sistema operacional Windows. Cada grupo, com no máximo dois alunos, trabalhará em um projeto definido por ele. É necessário que todos os membros tenham uma conta no GitLab da Unicamp.

O objetivo da primeira versão da `main` é estabelecer uma base mínima e funcional para o projeto, permitindo que a equipe colabore de forma organizada. Em vez de um sistema completo, essa versão inicial deve focar na estrutura essencial, como a hierarquia de pastas e arquivos, o `README.md` com instruções, o `.gitignore` e outras configurações fundamentais. Criar uma boa base desde o início previne problemas futuros, enquanto a equipe trabalha gradualmente nas funcionalidades completas.

Preparação do ambiente

Antes de iniciar, certifique-se de ter os seguintes *softwares* instalados:

- **Git for Windows:** Essencial para interagir com o GitLab via linha de comando. Faça o *download* em https://git-scm.com/download/win e siga as instruções de instalação.
- **Thonny IDE:** A IDE Python que será utilizada para desenvolver o firmware. Faça o download em https://thonny.org/ e instale-o.
- **Doxygen:** Ferramenta para gerar documentação a partir do código-fonte. Faça o *download* em http://www.doxygen.nl/download.html (baixe a versão doxygen-*-setup.exe) e siga as instruções de instalação. Certifique-se de que o Doxygen esteja no PATH do seu sistem ou que o caminho completo para o executável doxygen.exe seja conhecido.
- Graphviz (Opcional, mas recomendado para Doxygen): Para que o Doxygen gere diagramas (como gráficos de dependência de chamadas), será necessário o Graphviz. Faça o *download* em https://graphviz.org/download/ e instale-o.
- Mscgen (Opcional, mas recomendado para Doxygen): Para que o Doxygen gere diagramas de sequência de mensagens a partir de uma descrição em texto simples. Um endereço para *download* é https://www.mcternan.me.uk/mscgen/.

Ativação da conta GitLab da Unicamp

Qualquer aluno matriculado na Unicamp já possui uma conta no GitLab da Unicamp. Para ativá-la, basta seguir estes passos:

1. Acesse o GitLab da Unicamp: https://gitlab.unicamp.br.

2. Faça *login* com as credenciais da Unicamp (RA e senha do SIGA/DAC). Se for o primeiro acesso, o sistema pode pedir que algumas informações do perfil do usuário sejam completadas.
3. Preencha o perfil com o nome completo, e-mail institucional e outras informações solicitadas.

Com seu perfil completo, o acesso ao servidor será ativado, e o uso do GitLab é liberado. O preenchimento correto é importante para que seus *commits* sejam facilmente identificados.

Clonagem do repositório do projeto

O primeiro passo no desenvolvimento do projeto P1 é clonar o projeto do repositório do GitLab para o computador local.

1. Abra o `Git Bash` (instalado com o Git for Windows).
2. Navegue até o diretório onde deseja armazenar o projeto clonado (ex: cd C:/EA801/turma<letra_da_turma>/SeuRA/).
3. Para clonar o projeto (substitua a URL pelo *link* do projeto específico. Por exemplo, https://gitlab.unicamp.br/EA801_2025S2/tA/G1/P1):

```
git clone  https://gitlab.unicamp.br/EA801_2025S2/tA/G1/P1.git
```

4. Isso criará uma pasta com o nome do projeto P1 contendo todos os arquivos do repositório.
5. Acesse a pasta local do projeto:

```
cd P1
```

Configuração do .gitignore

Para garantir que apenas os arquivos necessários para o *firmware* e sua documentação sejam versionados, precisa-se configurar o arquivo .gitignore do projeto. Isso evitará que arquivos intermediários, temporários, de *build* ou de *cache* sejam incluídos no controle de versão.

1. Se está usando Windows e colabora com usuários de Linux/macOS, habilite a conversão automática de LF (do inglês *Line Feed*, ou \n) do Linux para a combinação de caracteres CRLF (do inglês *Carriage Return* + *Line Feed*, ou \r\n) do Windows para indicar o fim de uma linha.

```
git config --global core.autocrlf true
```

2. Acesse a pasta-raiz do projeto (P1) usando o explorador de arquivos. Clique com o botão direito do *mouse* na pasta para abrir um *menu* de contexto e selecione "Open Git Bash here" para abrir o terminal. No terminal, abra o novo arquivo `.gitignore` com a linha de comando:

```
nano .gitignore
```

   Após inserir o seguinte conteúdo no arquivo, pressione Ctrl + O para salvar e, em seguida, Ctrl + X para fechar o editor.

```
# Python
# Ignorar arquivos de cache e compilados do Python
__pycache__/
*.pyc
*.pyd
*.pyo

# Ignorar arquivos temporários e de log
*.tmp
*.log
*.swp
*~

# Ignorar arquivos específicos do Thonny (se gerados na raiz do projeto)
*.ini
*.bak
# Se o Thonny criar outras pastas de configuração na raiz, adicione-as
aqui.

# Ignorar pastas de ambiente virtual
venv/
.venv/

# Doxygen
# Ignorar o conteúdo de docs
docs/*
# mas não a pasta
!docs/.gitkeep

# Ignorar quaisquer outros arquivos não essenciais para o firmware ou
documentação final
# Exemplo: pasta de resultados de teste ou simulação
```

3. Adicione o arquivo .gitignore no Git com a linha de comando:

```
git add .gitignore
```

4. Registre as atualizações no repositório local com uma mensagem de *commit* que descreva claramente as alterações feitas:

```
git commit .gitignore -m “Adiciona .gitignore.”
```

Estruturação das pastas

Para este projeto, os arquivos e pastas são organizados em um nível de profundidade.

```
None
P1
├── .gitignore
├── meu_projeto.py
├── lib
├── Doxyfile
├── docs
└── README.md
```

- .gitignore: Como vimos, este arquivo informa ao Git quais arquivos e pastas ele deve ignorar e não incluir no repositório. Isso evita que arquivos temporários, de *cache* ou de compilação sejam rastreados.
- meu_projeto.py: É o código principal da sua aplicação. Ele contém a lógica central do sistema embarcado, como a leitura de sensores, o controle de atuadores ou a comunicação com outros dispositivos.
- lib: É usada para armazenar bibliotecas, módulos ou arquivos Python que não são nativos do MicroPython, mas são necessários para o projeto.
- Doxyfile: Este é o arquivo de configuração para o Doxygen, uma ferramenta de documentação. Ele contém todas as configurações necessárias para gerar documentação a partir dos comentários no código-fonte, criando manuais de referência para a aplicação projetada.
- docs/: Esta pasta armazena a documentação do projeto. É comum que as páginas geradas pelo Doxygen sejam salvas aqui. Além disso, a pasta pode conter manuais de usuário, diagramas de arquitetura e qualquer outro tipo de documentação adicional.
- README.md: É um documento de texto escrito em formato Markdown (`.md`). Ele funciona como a "página inicial" do projeto e é o primeiro arquivo que uma pessoa lê ao acessá-lo. Contém tipicamente o título e descrição do projeto, estrutura da pasta, componentes de *firmwares* e *hardware* necessários, configuração do projeto, incluindo como instalar as bibliotecas da pasta `libs` no Pico, guia de uso, créditos e licença.

Desenvolvimento do *firmware* com Thonny

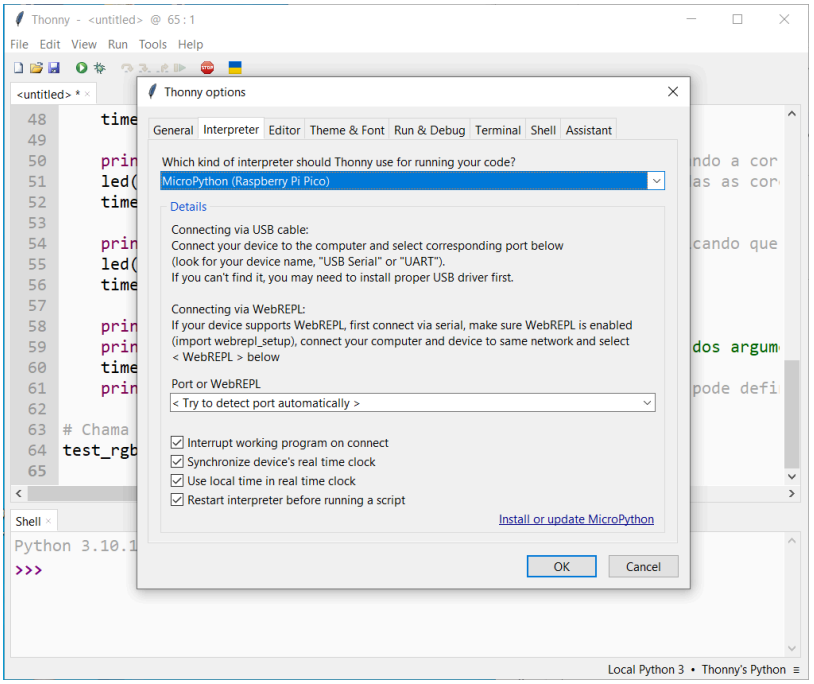
Passo a passo de desenvolvimento:

1. Crie a pasta `lib` na pasta-raiz do seu projeto e adicione-a ao Git. Essa pasta será usada para armazenar as bibliotecas de terceiros necessárias.

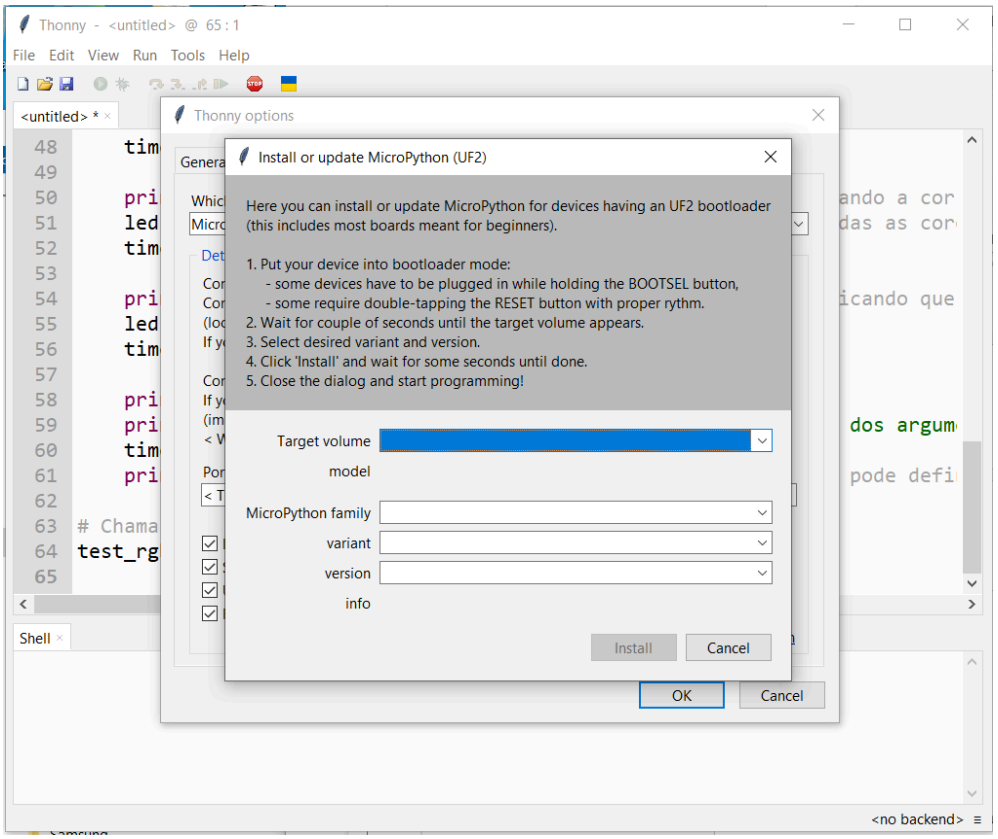
   ```
   mkdir lib
   touch lib/.gitkeep
   git add lib
   git commit lib -m "Adiciona a pasta vazia lib"
   ```

2. Abra a IDE Thonny.

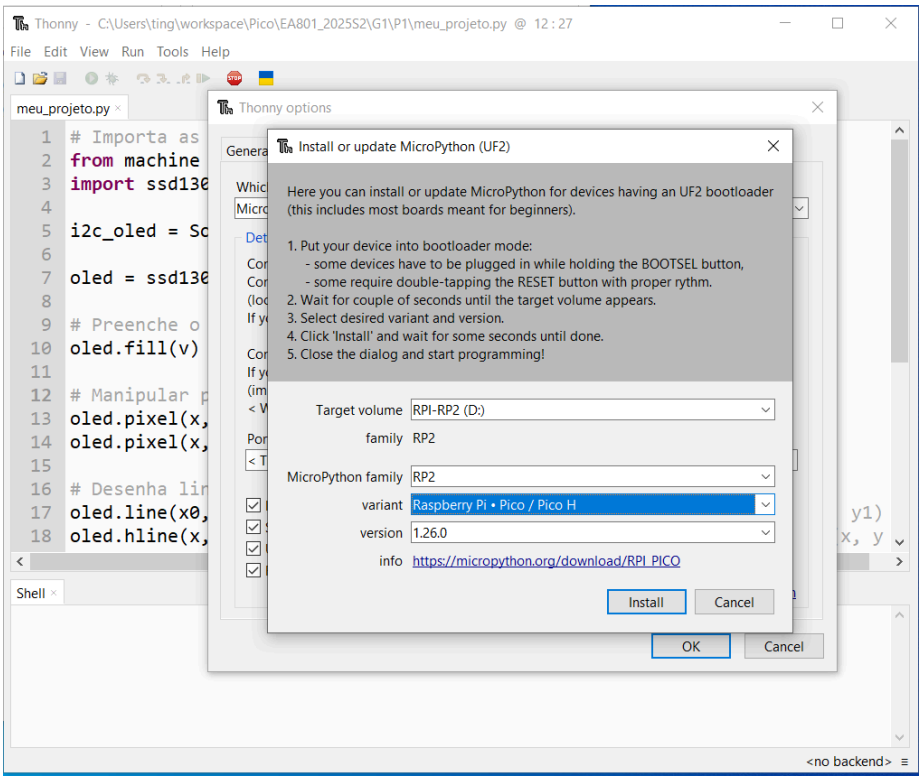
3. Se é a primeira vez que usa a sua placa Raspberry Pi Pico para programação de *firmware* em Python, vá para Run > Configure Interpreter.
4. Na janela aberta, selecione o *hardware* onde rodará o código em Python. Automaticamente, muda-se para uma tela em cujo canto inferior direito há um acesso para instalação/atualização do interpretador MicroPython.



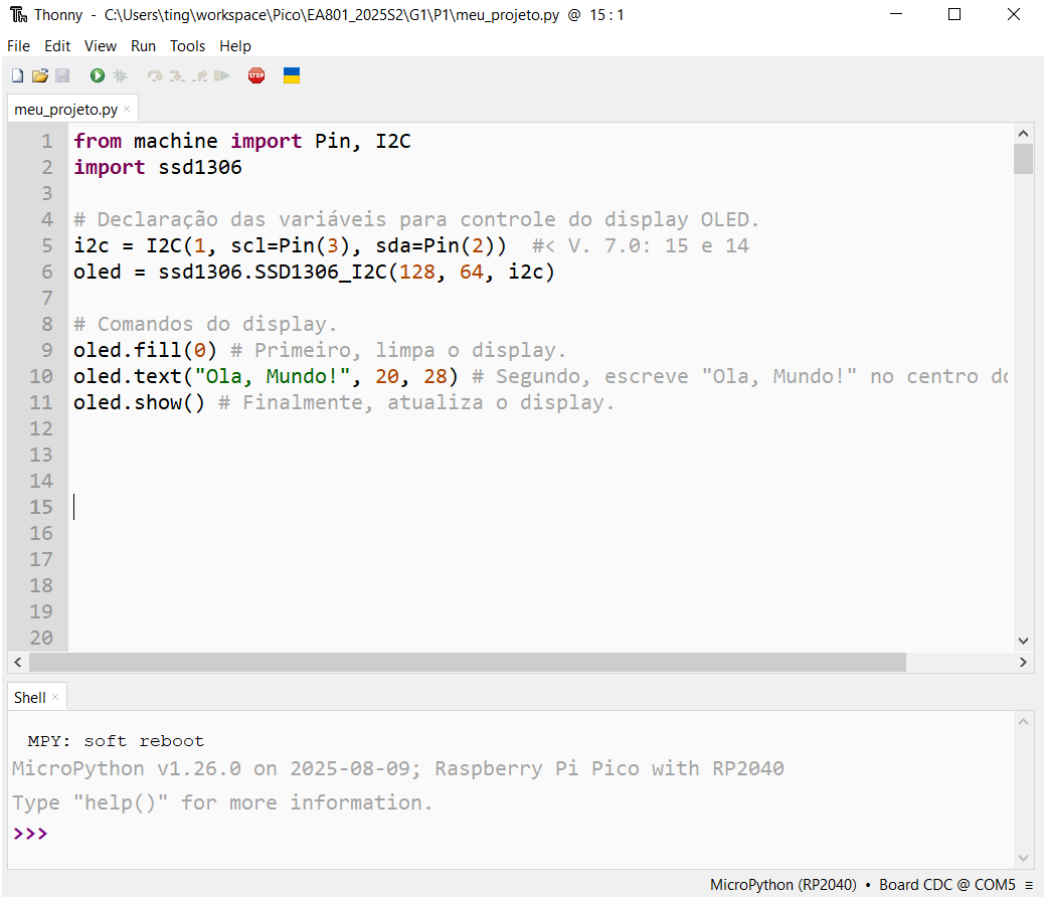
5. Siga o procedimento na nova tela para instalar ou atualizar o MicroPython.



Se o preenchimento automático dos campos “Target volume” e “MicroPython family” ocorrer, selecione “Raspberry Pi Pico/Pico H” no campo “variant”. O botão de Install será ativado. Clique nele para começar a instalação do MicroPython.



Ao concluir a instalação, o Shell da IDE Thonny exibirá o *prompt* do MicroPython. No canto inferior direito da janela, aparecerá a porta COM à qual o Pico está conectado. Caso essas informações não apareçam, clique no botão “Stop” (quadrado vermelho) na barra de ferramentas superior. Isso interrompe qualquer *script* em execução e, na maioria dos casos, reinicia o interpretador do MicroPython, permitindo que a conexão seja restabelecida corretamente.



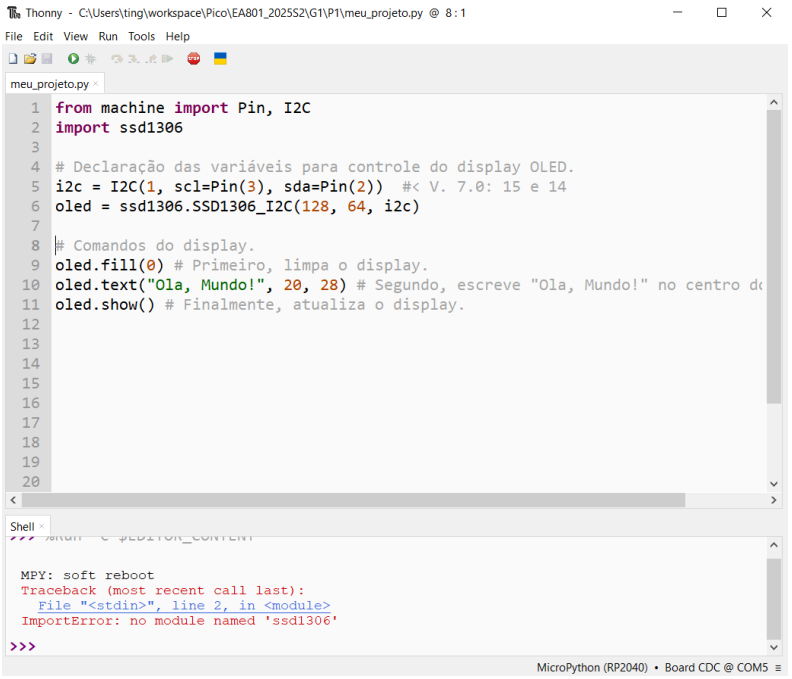
Para alguns sistemas, é preciso entrar no modo *bootloader* do Raspberry Pi Pico antes de abrir a IDE Thonny. Isso garante que o volume de destino (`target volume`) seja reconhecido. Também pode ser necessário reconectar o Pico mais de uma vez, mantendo o botão BOOTSEL pressionado até que ele entre no modo *bootloader*.

Dois tutoriais, da RoboCore e da Embarcados, explicam o passo-a-passo para preparar Raspberry Pi Pico para programação em MicroPython.

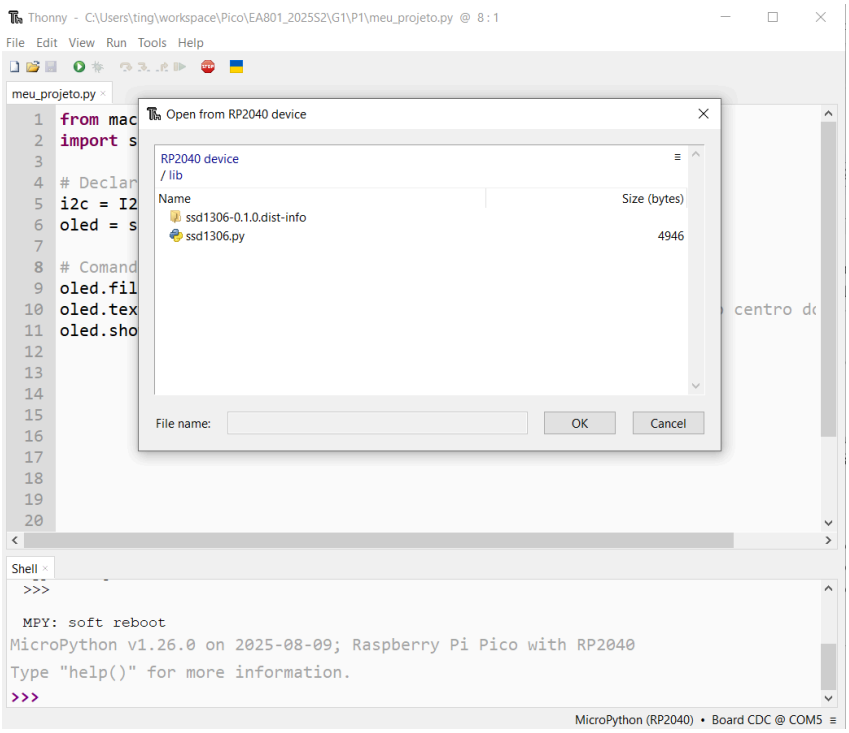
6. Com o *firmware* do interpretador instalado, vá em “File” (Arquivo) -> “New” (Novo) para abrir um novo projeto. Salve o programa com “Save as …” na pasta do projeto P1 nomeando o projeto com um nome de sua escolha, como meu_projeto.

   Para mais detalhes sobre as funções disponíveis no Thonny IDE, consulte este tutorial.

7. Escreva o código do projeto no Editor. Um bom exemplo é o *script* do curso *online* Introdução Prática à BitDogLab para exibir "Olá, Mundo!". Ele demonstra como controlar o periférico OLED do *shield* BitDogLab usando um *firmware* em MicroPython.
8. Para executar o *script*, clique em Run > "Run current script". Se a configuração estiver correta, a mensagem "Olá, Mundo!" será exibida no display OLED.
9. Se o Shell exibir um erro de importação relacionado ao módulo `ssd1306`, isso significa que a biblioteca não foi encontrada pelo interpretador MicroPython. Isso acontece porque a biblioteca `ssd1306` não é nativa e precisa ser instalada no Raspberry Pi Pico.



Para solucionar o problema, selecione Tools > “Manage packages …” e insira `ssd1306` no campo de busca da janela que se abrir. Selecione então o pacote “ssd1306” da lista “Search Results” e clique em “Install” para instalá-lo diretamente no Raspberry Pi Pico. Navegue até File > Open > RP2040 device. Lá, você encontrará o arquivo `ssd1306.py` na pasta `lib`, que é o diretório padrão do MicroPython para bibliotecas de terceiros.



Após a instalação, reexecute o *script* (Run > "Run current script"). O erro de importação desaparecerá e o *script* funcionará normalmente. Para evitar novas buscas pelo ssd1306, abra o arquivo na IDE (File > “Open …” > “RP2040 device” > lib > ssd1306.py) e salve-o

na pasta-raíz do computador. É uma boa prática fazer uma cópia da biblioteca na pasta `lib` do computador (File > "Save copy …" > This computer > navegue_até_pasta_lib > Salvar).

Como alternativa, pode-se baixar o arquivo `ssd1306.py` diretamente do repositório de BitDogLab para a pasta `lib` do projeto `P1`. Depois, para que a biblioteca funcione no Pico, salve uma cópia do arquivo no dispositivo através da IDE Thonny: vá em File > "Save copy…" > "RP2040 device". Isso garantirá que o arquivo seja salvo na pasta `lib` do microcontrolador.

10. Na pasta-raíz do projeto, adicione e registre localmente `ssd1306.py` e meu_projeto.py no Git.

```
git add lib/ssd1306.py meu_projeto.py
git commit meu_projeto.py lib/ssd1306.py -m "Adiciona meu_projeto.py e a biblioteca ssd1306.py"
```

Documentação com Doxygen

Para que o Doxygen gere a documentação de um projeto, é necessário seguir três passos principais:

1. Adicionar comentários ao código: Insira comentários de documentação no seu código Python, seguindo a sintaxe específica do Doxygen.
2. Configurar o Doxygen: Crie e personalize um arquivo de configuração (Doxyfile) para definir como a documentação será apresentada.
3. Definir a pasta de saída: Reserve uma pasta no seu projeto, como docs/, para armazenar os arquivos de documentação que o Doxygen irá gerar.

Segue-se o exemplo de um *docstring* (do inglês *documentation string*) no estilo Doxygen em Python:

```
#!/usr/bin/env python3
"""! @brief Exemplo simples para inicializar e exibir texto em um display OLED."""

##
# @file meu_projeto.py
# @brief O script demonstra como usar a biblioteca `ssd1306` em um Pico
# com MicroPython para interagir com um display OLED. Ele inicializa a interface
# I2C e exibe a mensagem "Ola, Mundo!". O I2C é inicializado usando os pinos
# dedicados do microcontrolador para essa função, o que garante um desempenho mais estável e eficiente do que o método de bit-banging.
# @author Wu Shin-Ting
# @date 12/08/2025

# Imports
```

```
"""! Importa as classes Pin e I2C da biblioteca machine."""
from machine import Pin, I2C
"""! Importa a biblioteca ssd1306 para controlar o display OLED."""
import ssd1306

##
# @brief Configura e inicializa a interface i2c.
"""! Declaração da interface I2C com os pinos de SDA e SCL."""
i2c = I2C(1, scl=Pin(3), sda=Pin(2))  #< V. 7.0: 15 e 14

##
# @brief Inicializa o objeto do display OLED com as dimensões de 128x64 e
a interface i2c.
oled = ssd1306.SSD1306_I2C(128, 64, i2c)

oled.fill(0) # Primeiro, limpa o display.
oled.text("Ola, Mundo!", 20, 28) # Segundo, escreve "Ola, Mundo!" no
centro do display.
oled.show() # Finalmente, atualiza o display.
```

Para mais informações sobre como documentar código Python com a sintaxe Doxygen, consulte o tutorial de John Woolsey e o manual de Doxygen, de autoria de Dimitri van Heesch.

Para configurar a documentação do projeto, crie um arquivo `Doxyfile` na pasta raiz com a seguinte linha de comando

```
doxygen -g Doxyfile
```

e ajuste os valores dos seguintes parâmetros.

```
None
PROJECT_NAME = "Nome do seu projeto"
INPUT = .
FILE_PATTERNS = *.py
OPTIMIZE_OUTPUT_FOR_C = NO
OUTPUT_DIRECTORY = ./docs
EXTRACT_ALL = YES
EXTRACT_PRIVATE = YES
JAVADOC_AUTOBRIEF = YES
EXCLUDE_PATTERNS = */__pycache__/*
RECURSIVE = YES
Users\ting
HAVE_DOT = YES
```

```
DOT_PATH = "caminho/para/o/dot/bin"
CALL_GRAPH = YES
CALLER_GRAPH = YES
GRAPH_METRICS = YES
CLASS_DIAGRAMS = YES
UML_LOOK = YES
HTML_OUTPUT = html
GENERATE_LATEX = NO
CREATE_SUBDIRS = YES
```

Em seguida, adicione o `Doxyfile` ao repositório Git. Para isso, execute no Git Bash a seguinte linha de comando na pasta raiz do projeto:

```
git add Doxyfile
```

Para que a estrutura do projeto seja compartilhada corretamente com a pasta `docs` já no lugar, precisamos adicionar um arquivo de marcador dentro dela, o Git não rastreia pastas vazias. A convenção padrão para isso é usar um arquivo vazio chamado `.gitkeep`. Ele sinaliza ao Git para rastrear a pasta, mesmo que não haja nenhum conteúdo gerado ainda. Pode-se fazer isso no Git Bash com as seguintes linhas de comando:

```
None
mkdir docs
touch docs/.gitkeep
git add docs/.gitkeep
```

Registramos ainda essas atualizações no repositório local com a linha de comando

```
git commit Doxyfile docs/.gitkeep -m "Adiciona Doxyfile e a pasta docs"
```

e geramos a documentação do código devidamente comentado com

```
doxygen Doxyfile
```

Isso criará a documentação HTML na pasta docs/html (que será ignorada pelo Git, conforme configurado no .gitignore). Pode-se abrir docs/html/index.html em um navegador para visualizar a documentação.

Por fim, para que as atualizações sejam compartilhadas com a equipe de projeto, elas devem ser enviadas para o repositório remoto. Execute o comando a seguir:

```
git push origin main
```

Com este procedimento, a versão 0.1 do projeto `P1` foi estabelecida na *branch* principal. Essa versão servirá como base para a equipe realizar refinamentos e aprimoramentos contínuos, visando o desenvolvimento do protótipo inicial.

Controle de Versão com Git

O vasto conjunto de comandos do Git pode parecer intimidante no início. Felizmente, para a maioria das tarefas de desenvolvimento de *firmware*, precisar-se-á dominar apenas um pequeno subconjunto desses comandos. Esta seção foca nos comandos mais comuns e essenciais do Git que serão usados frequentemente para colaborar, gerenciar suas alterações e manter seu projeto organizado. Com eles, pode-se adicionar novos arquivos, salvar o progresso, compartilhar o trabalho com um grupo e, se necessário, desfazer erros, garantindo um fluxo de trabalho suave e eficiente. No desenvolvimento local, pode-se adicionar e modificar arquivos à vontade na pasta de projeto.

1. **Verificar o Status:** Para ver quais arquivos foram modificados, adicionados ou excluídos:

```
git status
```

2. **Adicionar Arquivos para *Staging*:** Quando estiver satisfeito com as modificações em um ou mais arquivos, precisa-se "estagiá-los" para o próximo *commit*.
    - Para adicionar um arquivo específico:

```
git add seu_arquivo.py
```

    - Para adicionar todas as modificações no projeto (o `.gitignore` garantirá que apenas os arquivos corretos sejam adicionados):

```
git add .
```

3. **Realizar um *commit* (registrar localmente):** Depois de adicionar os arquivos, "commite" as mudanças, criando um ponto de salvamento na história do repositório local.

```
git commit -m "Mensagem descritiva do seu commit, ex: Adiciona funcao de controle de motor e docstrings Doxygen"
```

Dica: Escreva mensagens de *commit* claras e concisas que descrevem as mudanças realizadas.

4. **Atualizar a Versão Remota (*Push*):** Apenas quando estiver seguro das modificações conduzidas e quiser compartilhá-las com o repositório remoto (e com um colega de grupo), faz-se um *push*.

    - Antes de fazer o *push*, é uma boa prática fazer um *pull* para garantir que se tenha as últimas modificações do repositório remoto e evitar conflitos.

```
git pull origin main
```

- (Assumindo que main é o nome do *branch* principal. Verifique o nome do *branch* principal no GitLab, geralmente é main ou master). Resolva quaisquer conflitos que possam surgir durante o pull antes de prosseguir.
- Para enviar definitivamente as mudanças para o repositório remoto:

```
git push origin main
```

- (Substitua main pelo nome do *branch* principal se for diferente). O Git solicitará as credenciais da Unicamp. Pode-se configurar o **Git Credential Manager** para não ter que digitá-las toda vez:

```
git config --global credential.helper manager-core
```

- Na primeira vez que se faz o *push*, uma janela de *login* do *Git Credential Manager* pode aparecer, permitindo que sejam armazenadas as credenciais com segurança.

5. **Reversão de modificações:** É possível reverter tanto modificações locais quanto remotas.

   - **Reversão de modificações locais:** Pode-se reverter arquivos não commitados ou *commits* locais que ainda não foram enviados para o repositório remoto.

     i. Descartar modificações em um arquivo não "commitado" (cuidado, isso apagará suas mudanças):

     ```
     git checkout -- <caminho/do/arquivo>
     ```

     ii. Ou para descartar todas as modificações não "commitadas" em um diretório:

     ```
     git checkout -- <caminho/do/diretorio>/
     ```

     iii. Desfazer o último *commit* local (sem modificar os arquivos no diretório de trabalho):

     ```
     git reset HEAD~1
     ```

     Os arquivos do último *commit* voltarão para o estado de "*staged*". Pode-se então modificá-los ou descartá-los.

     iv. Desfazer o último *commit* local (e reverter os arquivos no diretório de trabalho):

     ```
     git reset --hard HEAD~1
     ```

     ATENÇÃO: Este comando remove permanentemente as alterações do último *commit* e as descartadas do diretório de trabalho. Use com extrema cautela!

   - **Reversão de Modificações Remotas (Evitar quando possível):** Reverter modificações que já foram enviadas para o repositório remoto (push) pode ter implicações para outros colaboradores. Evita-se este procedimento, pois poderá

afetar o desenvolvimento de um colega da equipe. **Se for absolutamente necessário, converse com a equipe antes**. A forma mais segura de reverter *commits* remotos é usando git revert. Ele cria um novo *commit* que desfaz as alterações de um *commit* anterior. Isso mantém o histórico do projeto linear e não reescreve o histórico.

i. Primeiro, identifique o *hash* do *commit* que se quer reverter (pode-se usar git log para ver o histórico de *commits* identificados pelos *hashes* abreviados):

```
git log --oneline
```

ii. Reverta o *commit* específico (para evitar ambiguidade, substitua hash_aberviado_do_commit pelo *hash* real, hash_do_commit):

```
git revert <hash_do_commit>
```

Use o seguinte comando para obter *hash* completo a partir do hash_abreviado_do_commit:

```
git rev-parse <hash_abreviado_do_commit>
```

iii. Isso abrirá um editor de texto para adicionar uma mensagem ao novo *commit* de reversão. Salve e feche. Em seguida, é necessário fazer um git push para enviar o *commit* de reversão para o repositório remoto.

6. **Ramificação do ramo principal/*main*/*master*:** Desenvolver novas funcionalidades ou correções de *bugs* de forma isolada, sem afetar a linha principal do projeto, git branch é o comando mais apropriado. Cenários de uso típico desse comando são

- Clonar repositório e trabalhar em uma *branch* diferente: clona um repositório URL, mas baixa apenas a branch especificada, economizando espaço e tempo de download, já que não carrega todo o histórico do projeto.

```
git clone -b nome-da-branch --single-branch URL
```

- Criar repositório com *branch* inicial personalizada: Esta opção já cria e define a **branch inicial** com o nome personalizado, diferente do padrão *main*/*master*.

```
git init -b nome-da-branch
```

- Renomear *branch* após clonar: Renomeia a *branch* atual, mudando-a de `main` para `nova-branch` no seu repositório local.

```
git branch -m main nova-branch
```

- Subir nova *branch* e torná-la principal no repositório remoto: Envia a `nova-branch` para o repositório remoto (`origin`). A opção `-u` (de *upstream*) configura essa nova *branch* como a principal, facilitando o uso do `git pull` e `git push` no futuro sem precisar especificar o nome da *branch*. Depois, é preciso

alterar a branch principal diretamente nas configurações do *site* para que ela se torne o padrão do projeto.

`git push -u origin nova-branch` + alterar no site

7. **Fusão de um *branch* com o *branch* principal:** A fusão de um *branch* com o *branch* principal é o processo de integrar as alterações feitas em um ambiente de desenvolvimento isolado (o *branch*) de volta ao código estável (*branch* principal) do projeto. O processo de fusão é simples e geralmente segue estes passos:

    - **Vá para o *branch* principal:** Primeiro, você precisa estar no *branch* principal que irá receber as alterações, que é tipicamente o `main` (ou `master`).

    ```
    git checkout main
    ```

    - **Atualize o *branch* principal:** É uma boa prática garantir que seu *branch* principal local esteja atualizado com a versão remota.

    ```
    git pull
    ```

    - **Faça a fusão:** Use o comando `git merge` para trazer as alterações do seu *branch* de desenvolvimento para o `main`.

    ```
    git merge nome-do-seu-branch
    ```

    Se não houver conflitos, a fusão é concluída automaticamente. Caso contrário, o Git irá pausar o processo e você precisará editar os arquivos para resolver os conflitos antes de finalizar a fusão com um novo *commit*.

    Essa fusão de um *branch* deve ser feita apenas quando as seguintes condições forem atendidas:

    - O trabalho está concluído: A nova funcionalidade ou correção de *bug* foi totalmente implementada.
    - O código foi testado: Todas as alterações foram exaustivamente testadas e não apresentam erros ou regressões.
    - A revisão de código foi aprovada: Em equipes, a fusão deve ser feita após a aprovação de um Pull Request (ou Merge Request), garantindo que o código foi revisado e validado por outros membros da equipe.