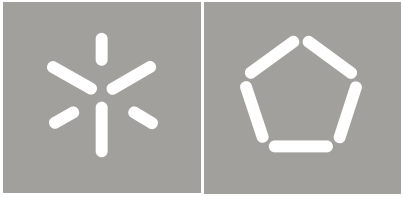
Universidade do Minho
Escola de Engenharia

Diogo Emanuel da Costa Lima

Framework Generativa para *Edge Devices*

Outubro de 2015

UMinho | 2015 Diogo Emanuel da Costa Lima *Framework* Generativa para *Edge Devices*



Universidade do Minho
Escola de Engenharia

Diogo Emanuel da Costa Lima

Framework Generativa para *Edge Devices*

Tese de Mestrado
Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Nuno Filipe Gomes Cardoso

Outubro de 2015

Declaração

Nome: Diogo Emanuel da Costa Lima
Endereço Eletrónico: ogoid.15@gmail.com
Telemóvel: +351 967704166
Bilhete de Identidade/Cartão do Cidadão: 13619522
Título da Dissertação: *Framework* Generativa para *Edge Devices*

Orientador: Professor Doutor Nuno Filipe Gomes Cardoso

Ano de conclusão: 2015
Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores

Universidade do Minho, _____/_____/_____

Assinatura:_____

Agradecimentos

O sucesso na elaboração desta dissertação, não se deve só a mim, mas a um conjunto de pessoas que de alguma forma contribuíram para que a mesma se tornasse uma realidade, às quais queria conceder sinceros agradecimentos. Abaixo são dirigidas algumas palavras a essas pessoas, não seguindo nenhuma ordem de importância, isto porque, todas elas foram importantes à sua maneira e de grande valor.

As primeiras palavras de agradecimento vão para o meu orientador e amigo Nuno Cardoso, pela confiança depositada em mim, pelo seu primoroso apoio prestado e os seus ensinamentos durante este percurso. Aos professores Adriano Tavares e Jorge Cabral por todas as sugestões e disponibilidade concedida.

Ao grupo de investigação ESRG (*Embedded Systems Research Group*) que muito bem me acolheu e me proporcionou todas as condições para a elaboração da dissertação, em especial, aos investigadores Tiago Gomes e João Brito pela disponibilidade e ajuda.

A todos os amigos de curso, pelo companheirismo, união e entreajuda durante todo o percurso académico, que fizeram com que se tornasse um percurso mais fácil de percorrer. Em especial, agradeço ao meu amigo Emanuel Ribeiro, que infelizmente não se encontra entre nós, mas vai ter sempre um lugar presente no coração, pelo excelente companheiro e amigo que foi e continuará a o ser.

Também não poderia deixar de agradecer aos meus pais, José Lima e Alice Costa, que sempre me apoiaram com o seu carinho durante todo o meu percurso, também aos meus irmãos e restante família, por todos os ensinamentos e educação que me levaram a ser o que sou hoje.

À minha namorada Sara, pelo sólido apoio contemplado e pela compreensão durante os longos e contínuos momentos de ausência devido à concentração na elaboração da dissertação.

A todos o meu muito obrigado!

Resumo

Nos dias de hoje, as empresas de *software* tentando captar o maior número de clientes, lançam para o mercado uma vasta gama de produtos. Metas temporais têm que ser cumpridas pela empresa aquando do desenvolvimento de um novo produto, de forma a evitar que empresas concorrentes cheguem primeiro ao mercado com um produto de especificações semelhantes.

Todavia, o desenvolvimento de um produto em tempo reduzido pode levar a imperfeições no mesmo. E para atenuar este problema, as empresas são obrigadas a adotar novos processos que reduzam o *time-to-market* sem afetar a qualidade do produto. Evitando assim, que o produto entre no mercado com problemas que poderiam resultar em prejuízos, como: muitos recursos humanos alocados à manutenção pós venda do produto, perda de clientes ou até indemnizações aos clientes lesados.

Normalmente, os produtos desenvolvidos por uma empresa pertencem à mesma família de produtos, onde a base dos diversos produtos é comum, só variando algumas partes dos mesmos, para adaptar os produtos aos diferentes níveis de exigência dos clientes. Desta forma, partes de um produto que já se encontra desenvolvido, testado e a ser comercializado podem ser reutilizadas para o desenvolvimento de novos produtos, acelerando assim o processo de desenvolvimento.

Um bom exemplo de família de produtos onde se verifica o padrão apresentado anteriormente são as redes de sensores sem fios (WSNs). Estes tipos de redes são hoje em dia amplamente utilizadas em diversas aplicações, tais como: domésticas, industriais, engenharia civil, militares, monitorização ambiental e saúde. Sendo a base da rede semelhante às várias aplicações, e só variam algumas configurações específicas, tipos de sensores e tipos de atuadores atribuídos aos nós da rede (*edge devices*).

Para acelerar o processo do desenvolvimento de aplicações que utilizam WSNs, é proposto nesta dissertação a elaboração de uma *framework* generativa, que permita obter um bom compromisso entre a gestão de variabilidade e o desempenho dos *edge devices*. Na fase de *design* da *framework* serão utilizadas técnicas de *Software Product Lines* (SPL) para identificar o grau de variabilidade das funcionalidades dos *edge devices*. E na fase de implementação será utilizado *template metaprogramming* em C++ para gerir essa variabilidade.

Palavras-Chave: *Framework* Generativa, WSN, *Template Metaprogramming*, C++.

Abstract

Nowadays, software companies trying to reach the largest possible number of customers, launch a wide range of products. Deadlines must be complied when developing a new product, in order to prevent that competitors arrive earlier to market with a product of similar specifications.

However, the development of a product in a short timeframe may lead to imperfections. In order to mitigate this problem, software houses are forced to adopt new processes that reduce time-to-market without affecting product quality. This is done in order, to prevent the product entering the market with bugs that could result in losses, such as: many human resources allocated to the after-sales maintenance of the product, loss of customers or even legal damages.

Usually, the products developed by a company belong to the same family of products, with a common base to all the products, varying some parts to adapt the products to different demand levels. Thus, parts of a product which were already developed, tested and marketed can be reused for the development of new products, thus speeding up the development process.

A good example of a family of products where there is the pattern previously presented are wireless sensor networks (WSNs). These types of networks are nowadays widely used in various applications, such as: domestic, industrial, civil engineering, military, environmental monitoring and health. Being the base of the network common to the various applications and vary only some specific configurations, sensor types and actuators assigned to the nodes in the network (edge devices).

To accelerate the development of applications using WSNs, it is proposed in this dissertation the development of a generative framework, enable to obtain a good compromise between the variability management and edge device's performance. In the design phase of the framework, Software Product Lines techniques will be used to identify the degree of variability of the functionalities of edge devices. In the framework's implementation phase template metaprogramming in C++ will be used to manage this variability.

Keywords: Generative Framework, WSN, Template Metaprogramming, C++.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Acrónimos

ADC	***A**nalog-to-**D**igital **C**onverter*
ADEOS	***A** **D**ecent **E**mbedded **O**perating **S**ystem*
AES	***A**dvanced **E**ncryption **S**tandard*
AOP	***A**spect **O**riented **P**rogramming*
API	***A**pplication **P**rogramming **I**nterface*
AVR	***A**dvanced **V**irtual **R**ISC*
bps	***b**its **p**er **s**econd*
CISC	***C**omplex **I**nstruction **S**et **C**omputer*
CPU	***C**entral **P**rocessing **U**nit*
DC	***D**uty **C**ycle*
DMA	***D**irect **M**emory **A**ccess*
DSM	***D**omain-**S**pecific **M**odel*
EC++	***E**mbedded **C++***
EEC++	***E**xtended **E**mbedded **C++***
ESRG	***E**mbedded **S**ystems **R**esearch **G**roup*
GCC	***G**NU **C**ompiler **C**ollection*
GNU	***G**nu's **N**ot **U**nix*
GPIO	***G**eneral **P**urpose **I**nput/**O**utput*
GPS	***G**lobal **P**osition **S**ystem*
HAL	***H**ardware **A**bstraction **L**ayer*
HTML	***H**yper **T**ext **M**arkup **L**anguage*
JTAG	***J**oint **T**est **A**ction **G**roup*
LED	***L**ight **E**mitting **D**iode*
LOC	***L**ines **O**f **C**ode*
IAR	***I**ngenjörsfirman **A**nders **R**undgren*
IDE	***I**ntegrated **D**evelopment **E**nvironment*
IEEE	***I**nstitute of **E**lectrical and **E**lectronics **E**ngineers*

ISR	***I**nterrupt **S**ervice **R**outine*
MAC	***M**edium **A**ccess **C**ontrol*
MP3	***MP**EG audio layer **3***
MPL	***M**eta**P**rogramming **L**ibrary*
MSXSL	***M**icro**S**oft e**X**tensible **S**tylesheet **L**anguage*
nesC	***n**etwork **e**mbedded **s**ystems **C***
NOM	***N**umber **O**f **M**odules*
OOP	***O**bject **O**riented **P**rogramming*
OS	***O**perating **S**ystem*
OSAL	***O**perating **S**ystem **A**bstraction **L**ayer*
OSI	***O**pen **S**ystems **I**nterconnection*
PDF	***P**ortable **D**ocument **F**ormat*
RAM	***R**andom **A**ccess **M**emory*
RF	***R**adio **F**requency*
RISC	***R**educed **I**nstruction **S**et **C**omputer*
ROM	***R**ead **O**nly **M**emory*
RTOS	***R**eal **T**ime **O**perative **S**ystem*
RTTI	***R**un-**T**ime **T**ype **I**nformation*
SFR	***S**pecial **F**unction **R**egisters*
SMS	***S**hort **M**essage **S**ervice*
SOC	***S**ystem **O**n **C**hip*
SPL	***S**oftware **P**roduct **L**ines*
SPI	***S**erial **P**eripheral **I**nterface*
STL	***S**tandard **T**emplate **L**ibrary*
STM	***S**GS-**T**hompson **M**icroelectronics*
TIMAC	***T**exas **I**nstruments **M**edium **A**ccess **C**ontrol*
TMP	***T**emplate **M**eta**P**rogramming*
UART	***U**niversal **A**synchronous **R**eceiver **T**ransmitter*
USB	***U**niversal **S**erial **B**us*
VDD	***V**oltage **D**rain **D**rain*
W3C	***W**orld **W**ide **W**eb **C**onsortium*
WSN	***W**ireless **S**ensor **N**etwork*
XML	*e**X**tensible **M**arkup **L**anguage*
XSLT	*e**X**tensible **S**tylesheet **L**anguage for **T**ransformation*

Capítulo 1

Introdução

Neste capítulo é contextualizado o âmbito da dissertação (1.1), seguidamente demonstrada a sua motivação e objetivos (1.2) e por último, apresentada a organização da dissertação (1.3).

1.1 Contextualização

As redes de sensores sem fios (WSNs) são identificadas como uma das mais importantes descobertas do século XXI [11]. Estas são utilizadas nos dias de hoje em diversas aplicações, tais como: domésticas, industriais, engenharia civil, militares, monitorização ambiental e saúde [12]. Normalmente, a sua principal função é a monitorização de fenómenos físicos associados à aplicação onde são inseridas. Cada tipo de aplicação exige a utilização de sensores dos mais variados tipos, tais como sensores de: aceleração, movimento, temperatura, humidade, campos magnéticos, radiação, *stress* mecânico, vento, entre outros.

As redes de sensores sem fios são compostas por vários nós (*edge devices*) que cooperam entre si para enviar informação desde o sensor até a uma central composta por um sistema informático. Os *edge devices* são dispositivos eletrónicos de pequenas dimensões, normalmente alimentados por baterias, compostos por um *transceiver* RF e um microcontrolador, podendo ser conectados aos mesmos um ou mais sensores dos tipos apresentados anteriormente.

Este tipo de redes são normalmente utilizadas para monitorização de fenómenos físicos em locais remotos e/ou de difícil acesso por meios humanos, sendo difícil a sua manutenção. Isto exige um grau elevado de preocupação no desenho e implementação, tanto a nível da gestão da bateria, como ao nível de evitar erros de *software* que

poderão comprometer o funcionamento dos *edge devices*.

O aumento da autonomia da bateria, normalmente é conseguido com utilização de técnicas de *Energy Harvesting*, através de geradores: fotovoltaicos, termoelétricos, microturbina ou piezoelétricos. Contudo, este tipo de fonte de energia extra, nem sempre é possível de aplicar, tanto a nível de incrementar o preço dos dispositivos, como exige um posicionamento adequado para captação da energia. Normalmente, isto não é possível visto que os *edge devices*, muitas das vezes são lançados aleatoriamente para o meio a monitorizar, através de helicópteros ou avionetas. Então, exige preocupações a nível de gestão do consumo de bateria, não só através da utilização de *hardware* que apresente baixos consumos, como também de um grau elevado de ajuste do *software*, mais propriamente dos protocolos de rede, pois é normalmente a parte que apresenta maior consumo energético. Este ajuste consiste na colocação do módulo em *sleep*, sempre que possível, e não comprometendo o funcionamento da rede.

Devido ao grande leque de aplicações em que as WSNs podem ser utilizadas, os *edge devices* apresentam um elevado grau de variabilidade do *software*. Esta variabilidade está ao nível das diferentes configurações do protocolo de comunicação, da utilização dos diferentes periféricos do microcontrolador para interação com a variedade de sensores/atuadores a utilizar em cada aplicação e ao nível das tarefas de sistema operativo que podem ser criadas para executar as funções associadas à aplicação em que estão inseridas. O elevado grau de variabilidade torna impensável a conceção *from scratch* de todo o *software* aquando o desenvolvimento de uma nova aplicação, isto porque aumentaria o *time to market* e a probabilidade do *software* conter erros que incorreriam em prejuízos à empresa que desenvolve este tipo de produtos.

Para resolver este problema é vantajoso utilizar uma *framework*, que facilite o desenvolvimento de aplicações para redes de sensores sem fios, explorando o que é comum às diferentes aplicações e fornecendo bibliotecas que permitam ser reutilizadas durante o desenvolvimento de novas aplicações para *edge devices*. Todavia, os *edge devices* possuem microcontroladores de poucos recursos, tanto a nível de memória de código/dados, como de processamento. Assim, estas bibliotecas deverão ser implementadas com mecanismos de programação que adicionem o menor *overhead*, em comparação ao código desenvolvido *from scratch*.

A programação deste tipo de dispositivos é normalmente efetuada através de linguagens de programação C/C++, mas maioritariamente em linguagem de progra-

mação C. Os mecanismos de gestão de variabilidade mais conhecidos e utilizados pela comunidade científica são: compilação condicional, polimorfismo dinâmico, programação orientada a aspetos e *template metaprogramming*.

A compilação condicional utiliza as potencialidades do pré-processador C/C++, para apresentar ou ocultar código ao compilador através de diretivas como o *#ifdef* e o *#endif*. Este mecanismo, tem a vantagem de não introduzir *overhead* em tempo de execução ao código, contudo tem a desvantagem de aumentar a complexidade de compreensão e manutenção do código.

O polimorfismo dinâmico é um mecanismo disponibilizado por linguagens orientadas a objetos, incluíndo a linguagem C++, que pode ser utilizado como mecanismo de gestão da variabilidade. Para isso é criada uma classe abstrata, contendo funções virtuais. A implementação das funções virtuais é efetuada nas classes que herdam da classe abstrata. Assim, é possível gerir a variabilidade criando várias classes que herdam da classe base, e implementando cada uma o seu comportamento para cada ponto de variabilidade. Este mecanismo torna o código bastante fácil de ler e de fácil manutenção, contudo tem como grande desvantagem a inserção de *overhead* em tempo de execução.

A Programação Orientada a Aspetos (AOP), é uma nova abordagem de programação e funciona como uma espécie de um pré-processador que injeta código pelas classes ou funções, tal como especificado pelo programador, através de métodos fornecidos pela linguagem (*aspect, pointcut e advice*). Posteriormente, o código é compilado pelo compilador da linguagem base. Este mecanismo tem como vantagem fornecer uma fácil separação de conceitos, ainda assim, apresenta a desvantagem de obrigar à utilização de ferramentas externas para além do compilador C/C++ e obrigar os programadores a entenderem novos conceitos de programação.

O C++ *template metaprogramming* (TMP) utiliza as potencialidades das *templates* em C++ para gerar e manipular código em tempo de compilação. O TMP permite estender as capacidades do compilador C++, fazendo com que este opere como um interpretador [4]. Este mecanismo de gestão de variabilidade tem como vantagem permitir gerir a variabilidade do código em tempo de compilação, não introduzindo *overhead* em tempo de execução, e apresenta um código bastante modular, trazendo consigo todas as vantagens do paradigma de programação orientada a objetos, ao nível da gestão do código. Embora que em certas situações, o código se possa tornar de difícil compreensão, pelo simples facto de ainda não ser possível fazer *debug* de *templates* nos compiladores atuais.

Devido às vantagens que apresenta em relação aos outros mecanismos existentes, o TMP será utilizado nesta dissertação para implementação da *framework* de desenvolvimento de aplicações para *edge devices* em WSNs. Na fase de *design* serão utilizadas técnicas de engenharia SPL, mais concretamente diagramas de funcionalidades para identificar o grau de variabilidade de um *edge device* e diagramas de classes para identificar os métodos, atributos e relações entre as classes contidas nas bibliotecas da *framework*.

1.2 Motivação e Objetivos

Em termos de motivação pessoal na escolha do tema de dissertação, esta teve em consideração que se inserisse numa área de particular interesse. Tema que possa ser aplicado nos dias de hoje, e que de alguma forma contribua para a criação/melhoria de algum produto, tecnologia ou serviço. E com isso aprender e melhorar aptidões que sejam úteis à integração no mercado de trabalho.

Neste sentido, surgiu a oportunidade de criar uma *framework* generativa para *edge devices*. *Framework* essa que facilitará o desenvolvimento de *edge devices* aplicados a redes de sensores sem fios (WSNs), acelerando o processo de elaboração deste tipo de dispositivos, e consequentemente reduzir o custo associado ao seu desenvolvimento. A elaboração desta *framework* ajudará na consolidação de conhecimentos sobre poderosos mecanismos de programação C++, como *template metaprogramming*, associados com a aprendizagem de técnicas de engenharia de *Software Product Lines*.

Esta dissertação tem como principal objetivo o desenvolvimento de uma *framework* generativa baseada em *template metaprogramming* que facilite o desenvolvimento de novos tipos de *edge devices* aplicados a redes de sensores sem fios. Permitindo obter um bom compromisso entre a gestão da variabilidade e o desempenho do código dos *edge devices*.

A *framework* a desenvolver será baseada no *software* TIMAC da *Texas Instruments* e terá como objetivo a conversão de algumas partes do mesmo para bibliotecas *template metaprogramming* em C++, sendo no final efetuados casos de teste de comparação, ao nível do desempenho das duas versões e obtidas conclusões acerca disso. Os casos de teste a realizar incidirão sobre o número de linhas de código, o número de módulos, o consumo de memória de código, o consumo de memória de dados e os tempos de execução da aplicação.

A *framework* proposta terá que aumentar a modularidade do código original

da TIMAC, bem como tentar evitar ao máximo a repetição de código, de forma a permitir uma fácil utilização e manutenção do mesmo.

1.3 Organização da Dissertação

Esta dissertação encontra-se estruturada em sete capítulos e um anexo. No capítulo 1, é contextualizado o âmbito da dissertação e descrita qual a motivação e os objetivos do seu desenvolvimento.

O capítulo 2, proporciona uma visão geral sobre como se encontram os temas abordados nesta dissertação, mais concretamente sobre: as redes de sensores sem fios, *frameworks* aplicadas a essas redes, engenharia de *Software Product Lines* (SPL) e no final, demonstrados os mecanismos de gestão de variabilidade ao nível da programação que podem ser utilizados para a implementação da *framework*.

O capítulo 3, apresenta uma visão geral sobre todos os componentes pertencentes à base de implementação do sistema, mais precisamente: a descrição das características do SOC (*System On Chip*) utilizado; a demonstração da pilha de *software* TIMAC em que se baseia a *framework* desenvolvida; o ambiente de desenvolvimento utilizado na implementação e testes ao código; a técnica de *template metaprogramming*, apresentando a biblioteca *Boost.MPL* que foi utilizada como base no desenvolvimento da *framework* para *edge devices*; e no final, é demonstrada a linguagem XML e XSLT, que possibilitam a geração de código respeitante à configuração da *framework*.

O capítulo 4, apresenta uma comparação entre a linguagem C e a linguagem C++, verificando se há custos no desempenho associados à utilização de linguagem C++ em substituição da linguagem C na implementação da *framework*. As comparações incidirão sobre algumas das funcionalidades da linguagem C++, tais como: referências, classes, construtores/destrutores, herança única, *overload* de operadores, *templates* e funções *inline*.

O capítulo 5, descreve como foi modelada e implementada a *framework*. Primeiro, é demonstrado como foi compatibilizada a biblioteca *Boost.MPL* com o compilador usado de forma a ser utilizada como base na implementação da *framework*. Por fim, é demonstrada a modelação e implementação da *framework*.

O capítulo 6, apresenta os resultados experimentais dos testes realizados à *framework*. Os testes incidirão sobre a comparação entre as duas versões de implementação da gestão da variabilidade do código da *framework* através de programação

condicional em linguagem C e de *template metaprogramming* em linguagem C++. Aplicando dois tipos de métricas: as métricas ao nível da gestão de código e as métricas ao nível do desempenho do código gerado pelas *frameworks*.

O capítulo 7, apresenta as conclusões retiradas após o desenvolvimento da dissertação, e enumera algumas sugestões de melhoria de forma a dar seguimento, ao projeto desenvolvido, como trabalho futuro.

Por último, o anexo A, apresenta os ficheiros XSLT referentes à geração do código de configuração da *framework*.

Capítulo 2

Estado da Arte

Neste capítulo, é dada uma visão geral sobre como se encontram os temas abordados nesta dissertação. Primeiramente, na secção 2.1 é disponibilizada uma visão geral do que são redes de sensores sem fios (WSNs), quais os seus cenários de aplicação (secção 2.1.1) e quais os pontos de variabilidade dos *edge devices* que constituem as WSNs (secção 2.1.2).

De seguida, na secção 2.2, é apresentada a definição do termo *framework*, indicando a que se destina e em que casos pode ser utilizada, seguido de uma demonstração de dois exemplos propostos por outros autores de *frameworks* aplicadas às WSNs (secção 2.2.1). No final da secção 2.2, serão discutidos os dois exemplos mostrados e qual será a abordagem a seguir nesta dissertação (secção 2.2.2).

Na secção 2.3, é apresentado o conceito de desenvolvimento de *Software Product Lines* (SPL), mostrando a forma de como se representa a variabilidade de um sistema através de diagramas de funcionalidades (secção 2.3.1).

A secção 2.4 é a última deste capítulo. É nela que são expostos os principais mecanismos de gestão de variabilidade do código ao nível da linguagem de programação C++, mecanismos esses, que possibilitam a criação da *framework* para WSNs. Os mecanismos apresentados são: Compilação Condicional (secção 2.4.1), Polimorfismo Dinâmico (secção 2.4.2), Programação Orientada a Aspetos (secção 2.4.3) e *Template Metaprogramming* (secção 2.4.4). De seguida, é feita uma comparação entre os mecanismos de gestão de variabilidade apresentados anteriormente (secção 2.4.5) e no final, são exibidos trabalhos de outros autores que utilizam este tipo de mecanismos, em aplicações de WSNs ou do género (secção 2.4.6).

2.1 *Wireless Sensor Network* (WSN)

As *Wireless Sensor Networks* (WSNs), em português, redes de sensores sem fios, são sistemas distribuídos tipicamente constituídos por dispositivos embebidos. Cada um equipado com uma unidade de processamento, um *transceiver wireless*, sensores e/ou atuadores [13].

Os dispositivos presentes numa WSN têm recursos limitados de memória de código/dados e processamento. Estes dispositivos, geralmente, têm que apresentar baixo consumo energético, já que normalmente são alimentados por baterias que devem durar meses ou até anos.

A programação de dispositivos pertencentes a WSNs exige alguns cuidados, para tentar maximizar o aproveitamento dos baixos recursos que estes dispositivos apresentam.

2.1.1 Cenários de Aplicação

Tal como foi dito anteriormente, as WSNs têm normalmente como objetivo a monitorização de algum fenómeno. Sendo nos dias de hoje aplicadas a uma ampla gama de cenários (expressos na figura 2.1), tais como [12]:

- **Aplicações militares:** monitorização de forças aliadas/opostas, vigilância, deteção de ataques nucleares, biológicos e químicos;
- **Aplicações ambientais:** acompanhamento de animais selvagens, monitorização das condições meteorológicas, agricultura de precisão e deteção de incêndios florestais;
- **Aplicações da saúde:** monitorização dos sinais vitais dos pacientes, deteção de quedas e administração de medicamentos;
- **Aplicações domésticas:** controlos de luminosidade/temperatura, deteção de incêndios, eletrodomésticos inteligentes e controlos remotos;
- **Aplicações industriais:** monitorização de qualidade dos produtos, comunicação com robôs móveis e monitorização de máquinas;
- **Aplicações de engenharia civil:** deteção de colisões de veículos e monitorização de estruturas como pontes, prédios, túneis, estradas, barragens e caminhos de ferro.

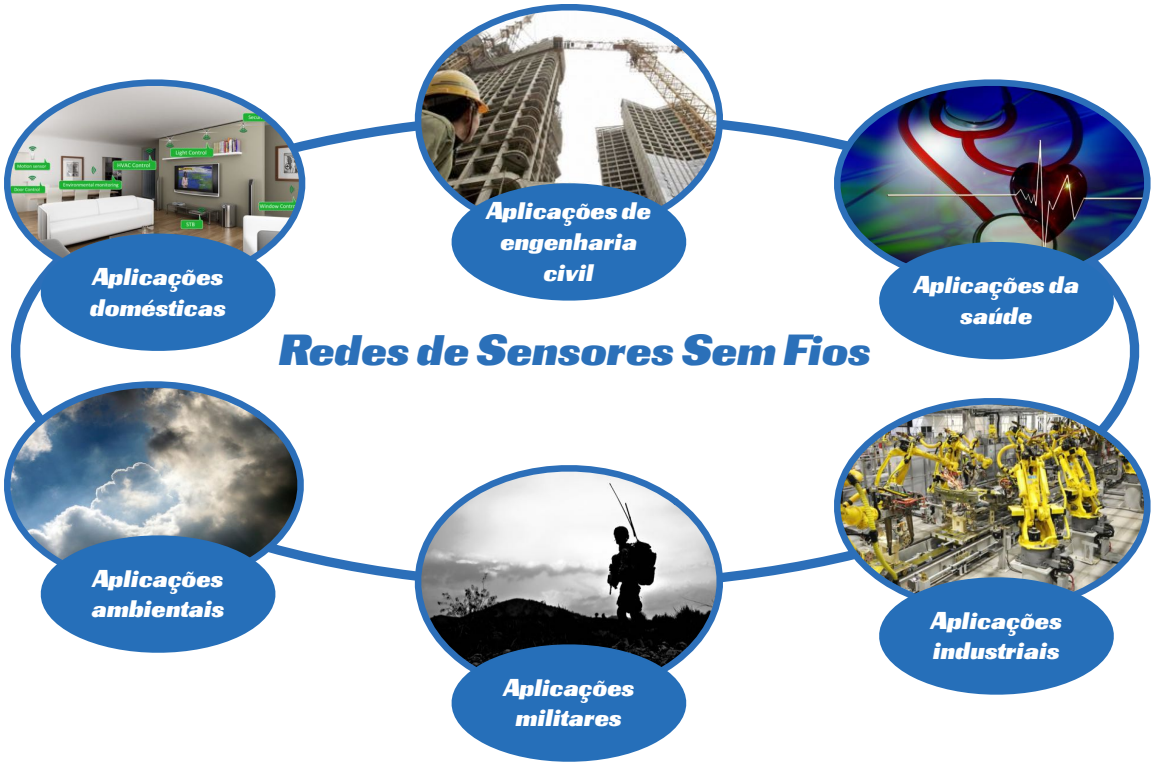


Figura 2.1: Cenários de aplicação das WSNs

2.1.2 Variabilidade dos *Edge Devices*

Os *edge devices* são conhecidos normalmente pelos nós que formam uma rede de sensores sem fios. A figura 2.2, apresenta uma rede de sensores sem fios composta por quatro *edge devices*, que desempenham um papel diferente na rede, tendo acoplados aos mesmos diferentes sensores e atuadores.

As redes de sensores sem fios são aplicadas sobre um grande número de cenários, tal como foi mostrado na secção 2.1.1. Cada cenário exige um nível de configuração da rede adaptado às condições do mesmo, bem como a utilização de diferentes sensores e atuadores conectados aos *edge devices*. Mesmo dentro do mesmo cenário de aplicação, os vários *edge devices* que formam a rede, poderão desempenhar diferentes funções, requerendo um nível diferente de configuração, bem como ter diferentes sensores e atuadores conectados ao mesmo. É esse nível de variabilidade, que se pretende gerir através do desenvolvimento de uma *framework* generativa.

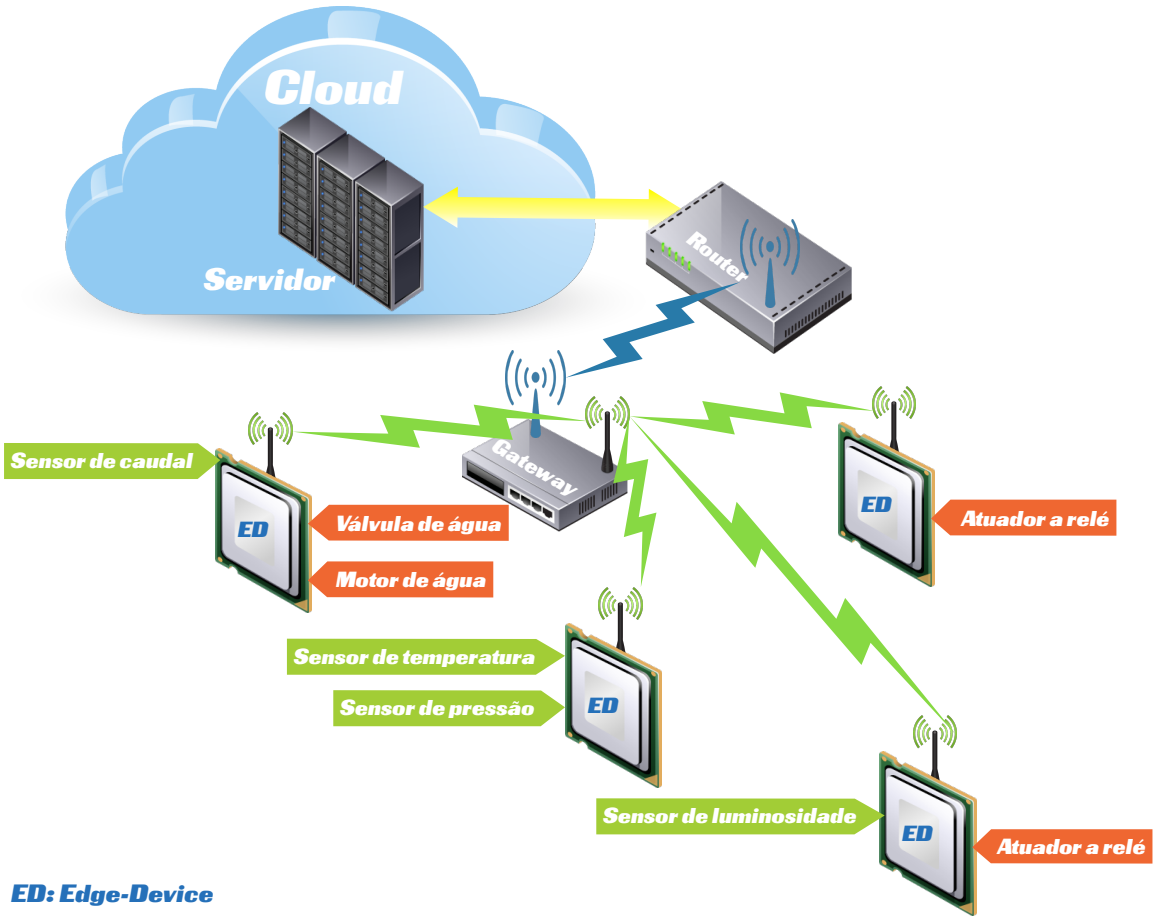


Figura 2.2: Variabilidade dos *edge devices*

2.2 *Framework*

Uma *framework* de *software* é uma plataforma que facilita o desenvolvimento de aplicações de *software* [14]. Esta, fornece uma base na qual os programadores podem desenvolver aplicações para uma plataforma específica. Agilizando assim, o processo de desenvolvimento de uma aplicação para a plataforma em questão.

A *framework* explora o que é comum no desenvolvimento de aplicações para o mesmo domínio do problema, e fornece mecanismos (classes e funções) que suportam a parte comum da aplicação. O seu desenvolvimento só é possível quando existem pontos comuns em várias aplicações, caso contrário o conceito *framework* não se aplica.

Na figura 2.3 são mostrados dois cenários diferentes, cenário A e cenário B.

Tanto no cenário A como no cenário B, estão representadas três aplicações diferentes simbolizadas pelos respetivos círculos.

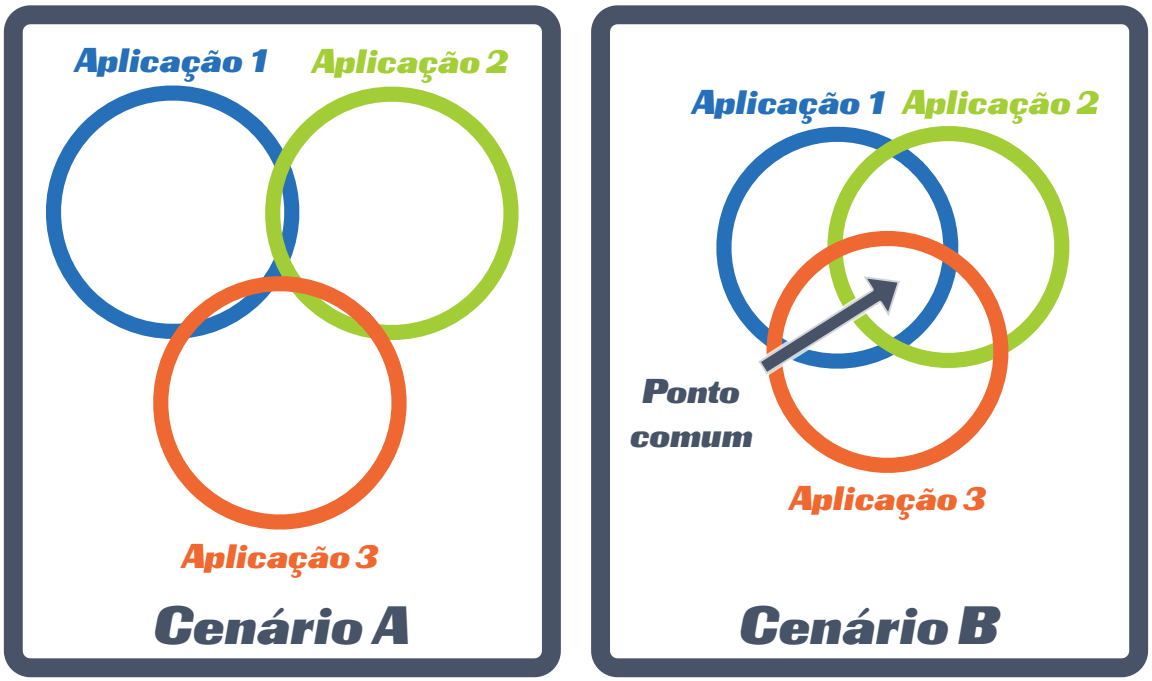


Figura 2.3: Em que casos pode ser desenvolvida uma *framework*

No cenário A, pode-se verificar que não há interseção dos três círculos, isto significa que não há pontos comuns entre as três aplicações. Assim sendo, não é possível criar uma *framework* para desenvolver aplicações neste cenário.

Já no cenário B, a interseção dos três círculos apresenta um ponto comum. Neste cenário, ao contrário do cenário anterior, é aplicável uma *framework*, para facilitar e consequentemente acelerar o desenvolvimento de aplicações que partilham o mesmo ponto comum.

2.2.1 *Frameworks* Aplicadas a WSNs

Nesta secção são apresentadas duas abordagens diferentes de *frameworks* para desenvolvimento de aplicações para WSNs, propostas por outros autores.

A Domain-Specific Modeling for Dynamically Reconfigurable Environmental Sensing Applications

Fajar et al. [1], em 2012, referem que a importância da reconfiguração nas WSNs tem levado muitos investigadores a desenvolver diferentes formas para redução dos custos na manutenção do *software* dos nós sensores. Contudo, a falta de utilização de metodologias de engenharia de *software* em estudos anteriores, pode causar muito esforço quando se pretende introduzir novas funcionalidades.

Foi proposto um modelo com alto nível de abstração que suporta a reconfiguração de aplicações de monitorização ambiental, permitindo aos programadores projetar as suas aplicações de forma flexível e reutilizável. O modelo proposto aumenta a produtividade em 6 a 8 vezes, comparado com a programação manual (sem suporte de uma *framework*).

Os autores basearam-se na metodologia de engenharia *Domain-Specific Model* (DSM) para solucionar o problema apresentado. Pois, segundo eles a utilização de DSM permite um alto nível de abstração da plataforma alvo, possibilitando a programação de WSN por pessoas que tenham pouca ou nenhuma experiência na programação deste tipo de dispositivos.

Figura 2.4: Arquitetura básica da DSM [1]

Na figura 2.4 está presente a arquitetura de uma DSM, onde os programadores modelam como pretendem o funcionamento do sistema numa linguagem mais intuitiva, neste caso linguagem gráfica. Depois de terminarem toda a definição da aplicação na linguagem gráfica, é executado um gerador que emite o código para a plataforma alvo.

Para separação de conceitos, os autores resolveram dividir o modelo em três níveis de abstração: nível lógico, nível físico e nível de alocação de tarefas. Na figura 2.5 é visível um exemplo onde se verifica essa divisão.

O nível lógico fornece os componentes necessários para modelação de tarefas básicas, tais como, amostragem de sensores e envio/receção de dados. Para além disso, também disponibiliza mecanismos que permitem definir a relação entre si. O nível físico é onde são definidos os componentes físicos necessários para execução das funções definidas no nível lógico. E por último, o nível de alocação de tarefas, permite

associar as tarefas implementadas no nível lógico aos elementos implementados no nível físico.

Como plataforma alvo os autores optaram pelo sistema operativo tinyOS [15] programado a partir de nesC [16].

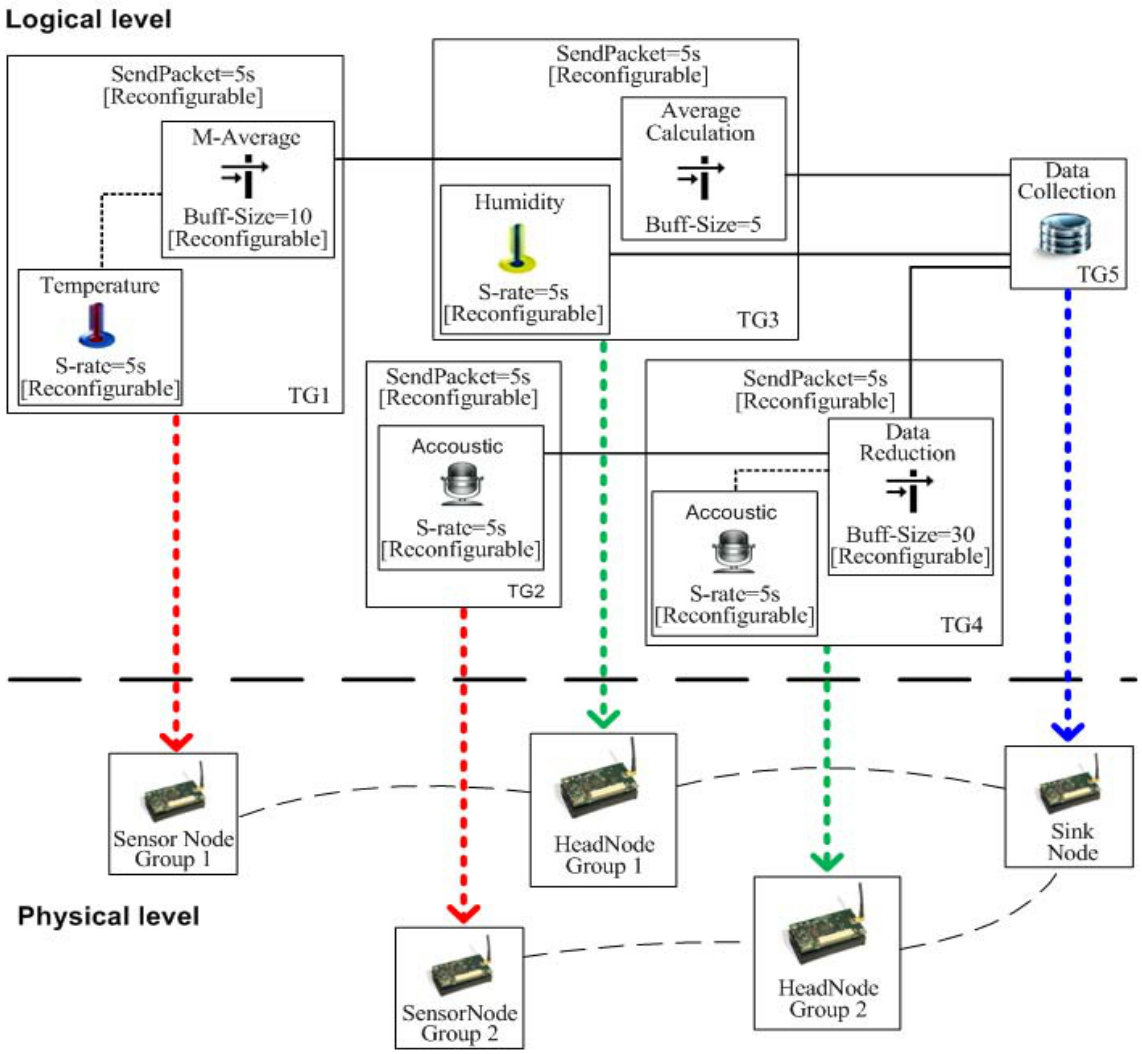


Figura 2.5: Exemplo demonstrativo da divisão do modelo em níveis [1]

An Extended Framework for the Development of WSN Applications

Pomante et al. [2], em 2010, afirmaram que a perceção do comportamento real de uma aplicação antes da sua implementação, permite aos engenheiros reduzir o tempo e custo de desenvolvimento. Esta afirmação aplica-se também ao desenvolvimento de

WSNs, cuja evolução depende fortemente do número de nós na sua configuração e na quantidade de tráfego. O qual, implica a necessidade de metodologias e ferramentas que assegurem que o produto final satisfaça as especificações e reduzam o tempo de desenvolvimento.

Assim, é sugerida uma *framework* na qual os *designers* podem criar componentes em qualquer nível da pilha de protocolo do modelo OSI, simular o funcionamento num dispositivo *host*, ajustar os parâmetros ideais e, no final, gerar automaticamente o código para a plataforma alvo.

A *framework* é baseada na ferramenta *simulink* [17] da *Mathworks* [18], e permite modelar aplicações para WSNs a partir de gráficos de estados e diagramas de blocos. Após a modelação é possível fazer verificações funcionais, verificação *hardware-in-the-loop* e gerar automaticamente código para a plataforma alvo. Contudo, a ferramenta *simulink* é do tipo *time-driven*, isto significa que lida com simulações periódicas. Enquanto que as WSNs são do tipo *event-based*, uma vez que, normalmente, os pacotes são enviados de forma assíncrona, pois dependem da ocorrência de eventos externos, das condições de trafego, do roteamento, entre outras causas. Para colmatar esse problema, os autores optaram por utilizar a biblioteca *SimEvents* [19] para *simulink*, que permite a simulação discreta de modelos do tipo *event-based*.

A solução de *framework* proposta, permite a geração automática de código para diferentes plataformas de nós sensores e diferentes sistemas operativos. Esta fornece extensões sobre a forma de blocos de bibliotecas, que podem ser reutilizados para desenvolver qualquer aplicação de WSN, podendo modelar com esses blocos o comportamento da mesma.

Na figura 2.6 está presente a interface de modelação de uma WSN na *framework* proposta pelos autores. Nesta figura estão presentes duas janelas, (a) e (b).

Em (a) está presente o modelo *Top-level*, onde é visível a criação de uma WSN com 5 nós, cada um representado pelo seu bloco. O bloco *Shared Buffer*, que simula o meio de comunicação entre os nós pertencente à WSN. O bloco *plots*, que é utilizado para apresentar dados de simulação em gráficos ou outro tipo de ferramenta disponibilizada pela *MathWorks*. E os blocos de configuração de parâmetros, que permitem definir parâmetros, tais como: *Transmission Bit Rate* e *Packet Error Rate*.

Em (b) é mostrado o interior de um bloco de nó sensor. Este é constituído por um bloco *Packed Builder Subsystem*, um bloco *Packed Splitter Subsystem* e um bloco *Channel Subsystem*, entre outros blocos de apoio. O bloco principal é o bloco *SOFTWARE* e é o único bloco do modelo usado para gerar código para a plataforma

alvo.

Segundo os autores, a vantagem de usar a *framework* desenvolvida por eles, em oposição à escrita manual de código para aplicações em WSNs, é que uma vez modelado e simulado o comportamento, o código pode ser gerado automaticamente e a aplicação executará na plataforma alvo em poucos minutos.

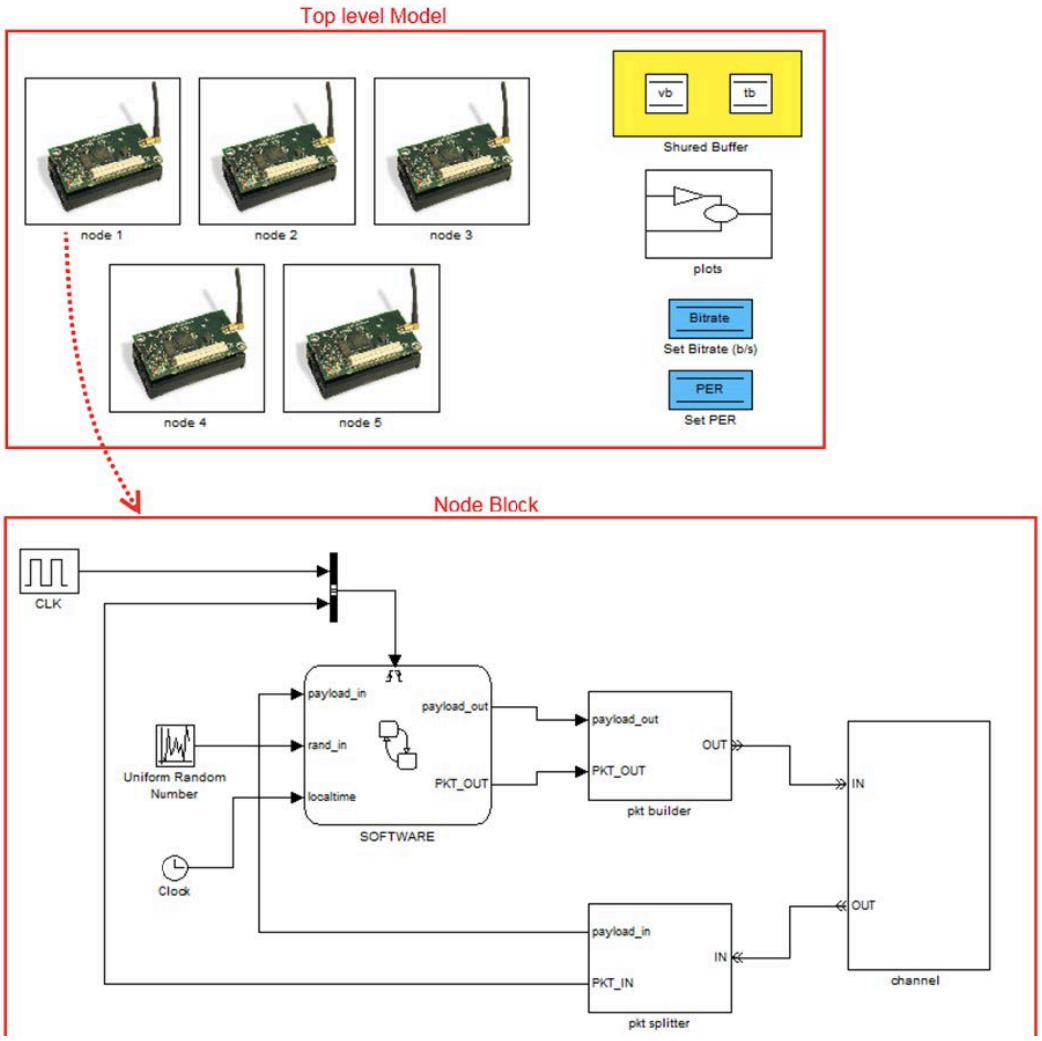


Figura 2.6: Modelo *Top-level* e bloco de um nó [2]

2.2.2 Abordagem a Seguir Nesta Dissertação

Os dois exemplos de *frameworks* para WSNs apresentados na secção 2.2.1, apresentam um elevado nível de abstração no desenvolvimento de aplicações para WSNs,

permitindo aos programadores implementar facilmente as aplicações, sem a necessidade de possuir grandes conhecimentos de programação, nem conhecimentos da plataforma alvo.

O programador apenas modela o sistema através de blocos gráficos ligados entre si, e no final dessa modelação, o código é gerado automaticamente para a plataforma alvo. O código gerado (C/C++) é injetado num ficheiro e posteriormente esse ficheiro é compilado para a plataforma alvo através de um compilador *standard* existente no mercado (GCC [20], IAR [21], KEIL [22], entre outros).

No entanto, um nível de abstração tão elevado nem sempre é bom, porque, além de limitar a utilização máxima dos recursos existentes na plataforma alvo, também obriga os programadores habituados a programar diretamente sobre a plataforma (em C/C++), a mudarem drasticamente a forma de programar e a instalarem ferramentas extras para além do compilador.

Nesta dissertação foi desenvolvida uma *framework* com um nível de abstração menor que as apresentadas na secção 2.2.1. *Framework* essa, indicada para um tipo de utilizador acostumado a programar este tipo de sistemas através de uma linguagem de programação convencional, tipo C ou C++.

Ao implementar a *framework*, utilizaram-se técnicas de Engenharia de *Software Product Lines* (SPL).

2.3 *Software Product Lines* (SPL)

Software Product Lines (SPL) é um portfólio de sistemas baseados em *software* semelhante e produtos produzidos a partir de um conjunto partilhado de recursos, utilizando um meio comum de produção [23]. Ao considerar um portfólio de produtos como uma única entidade a ser gerida, ao contrário de gerir individualmente uma infinidade de produtos, permite às organizações: diminuir o custo de desenvolvimento do produto, diminuir o *time-to-market* e aumentar a qualidade do produto [24]. Ou seja, a engenharia SPL não se foca só no desenvolvimento de um produto em particular, mas sim numa família de produtos, identificando o que é comum e variável dentro da gama de produtos pertencentes à família de produtos em questão. Desta forma, é possível reutilizar o *software* entre os vários produtos.

O gráfico da figura 2.7, apresenta quais são os custos e benefícios da utilização de Engenharia SPL, em comparação com Engenharia focada num só produto.

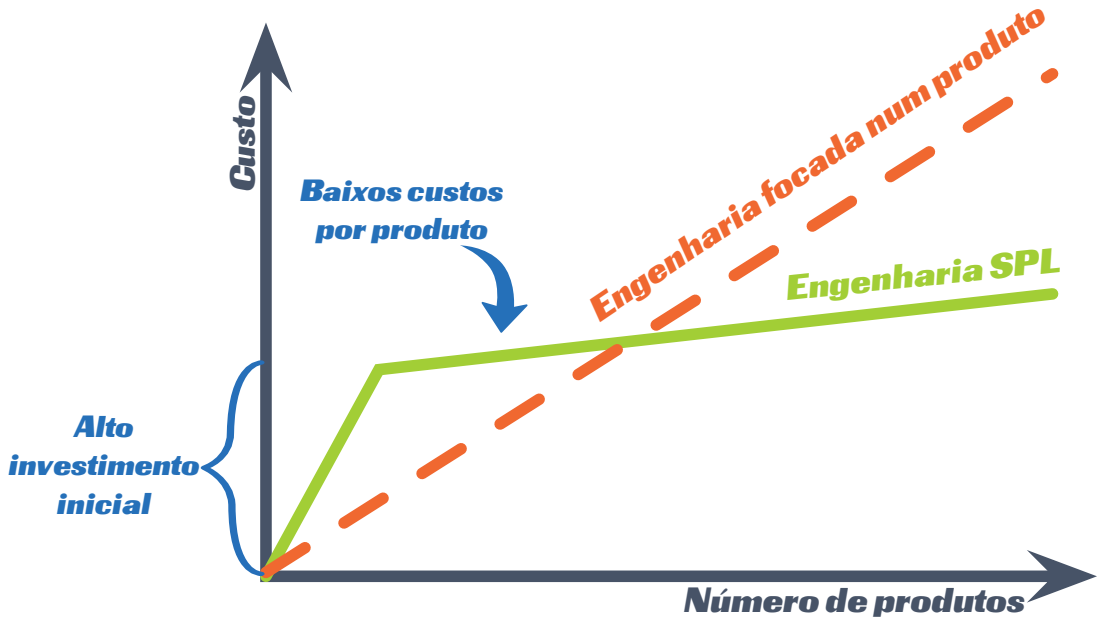


Figura 2.7: Custos/Benefícios da Engenharia SPL. Baseado em [3]

Ao analisar o gráfico, verifica-se que a engenharia SPL exige um alto investimento inicial, pois obriga a olhar para uma família de produtos, percebendo o que é comum a cada produto e o que difere os vários produtos dessa família. No final, é desenvolvida uma *framework* que gere esse grau de variabilidade e facilita o desenvolvimento de novos produtos. Todo esse processo demora tempo a ser implementado, mas depois de implementado, diminui o custo de desenvolvimento de novos produtos.

Observando a curva que representa a engenharia focada a um só produto, pode-se verificar que este tipo de engenharia não exige tantos custos iniciais no desenvolvimento do primeiro produto. E além disso, o tempo de desenvolvimento para os próximos produtos é bastante superior em relação à utilização de engenharia SPL.

Com isto, pode-se concluir que a utilização de engenharia SPL traz vantagens significativas a longo prazo em relação à engenharia focada num só produto.

2.3.1 Diagrama de Funcionalidades

Um diagrama de funcionalidades é uma representação gráfica utilizada em SPL, que fornece uma notação simples e intuitiva, para representar a variabilidade dos vários produtos pertencentes a uma família de produtos.

Com um diagrama de funcionalidades, é possível distinguir quais as funcionalidades obrigatórias em todos os produtos, e as funcionalidades opcionais que podem ou não existir nos distintos produtos pertencentes à mesma família.

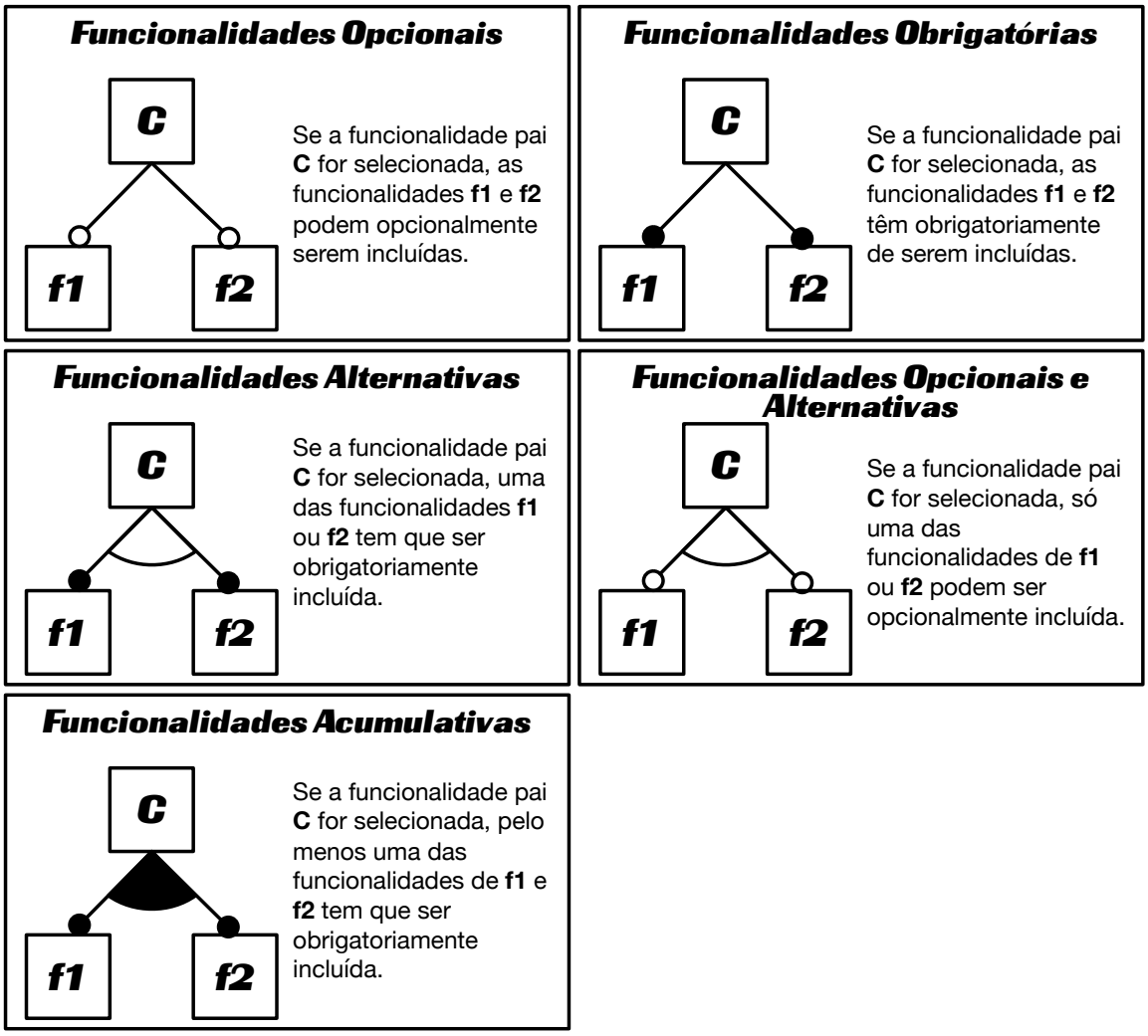


Figura 2.8: Tipos de funcionalidades. Baseado em [4]

Na figura 2.8 estão presentes os tipos de funcionalidades mais usadas no desenvolvimento de um diagrama de funcionalidades [4]. Os tipos de funcionalidades são:

- **Funcionalidades opcionais:** este tipo de funcionalidades, como o próprio nome indica, são funcionalidades que podem ou não ser incluídas caso a funcionalidade pai seja selecionada. Uma funcionalidade opcional é representada por

um segmento de reta, terminando numa circunferência sem preenchimento;

- **Funcionalidades obrigatórias:** são funcionalidades que têm que ser incluídas obrigatoriamente caso a funcionalidade pai seja selecionada. Uma funcionalidade obrigatória é representada por um segmento de reta, terminando numa circunferência com preenchimento;

- **Funcionalidades alternativas:** indica que uma, e só uma, das funcionalidades filha tem obrigatoriamente que ser incluída, caso a funcionalidade pai seja selecionada. Este tipo de funcionalidade é representado por segmentos de retas terminados em circunferências com preenchimento. Os segmentos de reta são sobrepostos por um arco sem preenchimento;

- **Funcionalidade opcionais e alternativas:** indica que uma, e só uma, das funcionalidades filha pode ser opcionalmente incluída caso a funcionalidade pai seja selecionada. Este tipo de funcionalidade é representado por segmentos de retas terminados em circunferências sem preenchimento. Os segmentos de reta são sobrepostos por um arco sem preenchimento;

- **Funcionalidade acumulativa:** indica que pelo menos uma das funcionalidades filha tem que ser incluída caso a funcionalidade pai seja selecionada. Este tipo de funcionalidade é representado por segmentos de retas terminados em circunferências com preenchimento. Os segmentos de reta são sobrepostos por um arco com preenchimento.

Na figura 2.9, é apresentado um cenário de aplicação de um diagrama de funcionalidades, no qual tenta abranger os vários tipos de funcionalidades. O diagrama de funcionalidades exposto, mostra a variabilidade de funcionalidades presentes nos telemóveis.

O telemóvel tem obrigatoriamente de possuir mecanismo de acessibilidade, ecrã, microfone, altifalante, e algumas funcionalidades. Assim todos esses elementos são representados por funcionalidades obrigatórias.

O mecanismo de acessibilidade presente no telemóvel é obrigatório e pode ser através de um teclado, através de uma película *touch* ou através de ambos. Sendo assim representados por funcionalidades acumulativas.

O ecrã do telemóvel também é obrigatório e pode ser a cores ou a preto e branco. Desta forma são representados por funcionalidades alternativas.

As funcionalidades são divididas em dois tipos: funcionalidades básicas e funcionalidades extra.

As funcionalidades básicas são todas elas obrigatórias, ou seja, todos os telemóveis têm que possibilitar fazer/receber chamadas e enviar/receber SMSs. Assim, são representadas por funcionalidades obrigatórias.

As funcionalidades extras, todas elas são facultativas. Opcionalmente, o telemóvel pode ter a função de lanterna, MP3, GPS ou câmara. Sendo assim representadas por funcionalidades opcionais. Em relação à câmara, esta só pode ser selecionada se o telemóvel possuir um ecrã a cores.

Se for escolhida a funcionalidade MP3, pode ser fornecido com o telemóvel uns *earPhones* ou uns *headPhones*. Apenas uma, ou até mesmo nenhuma, das opções é aceite. Sendo estas funcionalidades opcionais alternativas.

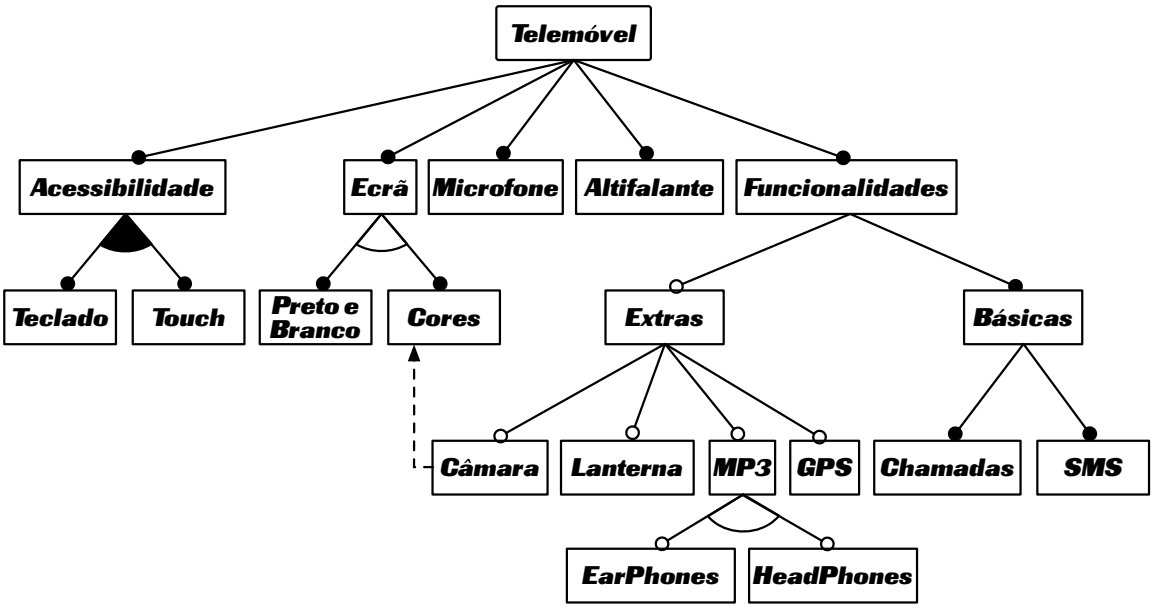


Figura 2.9: Diagrama de funcionalidades de um telemóvel

2.4 Mecanismos de Gestão de Variabilidade

Para gestão da variabilidade do código de uma família de produtos, têm que ser exploradas as técnicas oferecidas pelas linguagens de programação, que melhor desempenham o papel na gestão da variabilidade. Ou seja, deve ser explorada uma técnica que seja de fácil utilização, de fácil manutenção do código, permita a separação

de conceitos, evite a utilização de ferramentas extras para além do compilador e provoque o menor *overhead* possível no código produzido.

Nos subcapítulos seguintes são mostrados os principais mecanismos de gestão de variabilidade utilizados pela comunidade científica [4]. Em cada um dos mecanismos apresentados, é mostrado um pequeno exemplo de código, de como pode ser gerida a variabilidade da leitura de dados de diferentes sensores (sensor X e sensor Y) através do mecanismo em questão.

2.4.1 Compilação Condicional

A compilação condicional é obtida através das potencialidades das diretivas de pré-processamento existentes nas linguagens de programação C/C++, tais como: *#ifdef*, *#else*, *#endif* e *#elseif*. Com estas diretivas é possível definir qual será o código que irá ser apresentado ao compilador para que este efetue a sua compilação.

Este mecanismo não introduz *overhead* em tempo de execução, visto que a variabilidade é toda gerida em tempo de compilação. No entanto, o código desenvolvido através deste mecanismo fica disperso, tornando-o difícil de compreender, e de difícil manutenção [13].

A compilação condicional é um dos mecanismos de gestão de variabilidade mais utilizados pela comunidade científica e pela indústria [4]. Alguns dos projetos que usam intensivamente compilação condicional são: o sistema operativo Linux, o sistema operativo freeRTOS [25] e a TIMAC [26].

No código seguinte é mostrado como é gerida a variabilidade da classe *cSensor* através do mecanismo de compilação condicional:

Código 2.1: Mecanismo de compilação condicional - Criação da classe

```
class cSensor{
public:
  bool read(int &value){
  #ifdef _SENSOR_X_
    /* Código para o sensor X */
  #endif
  #ifdef _SENSOR_Y_
    /* Código para o sensor Y */
  #endif
    return true;
  }
};
```

A classe *cSensor* permite ler dados do sensor X ou do sensor Y. O código específico de cada sensor só é apresentado ao compilador, caso a macro de pré-

processamento do sensor em questão estiver definida. O código 2.1 corresponde à criação da classe. Para efetuar a leitura do sensor X, tem que ser definida a macro de pré-processamento *_SENSOR_X_*. Caso contrário, se for pretendido efetuar a leitura do sensor Y, tem que ser definida a macro de pré-processamento *_SENSOR_Y_*.

Código 2.2: Mecanismo de compilação condicional - Instanciação

```
#define _SENSOR_X_
#undef _SENSOR_Y_

int i_sensor_value = 0;
cSensor sensor;

sensor.read(i_sensor_value);
```

No código 2.2, é mostrado como se efetua a instanciação da classe *cSensor*, para efetuar a leitura de dados do sensor X.

De forma a compilar só o código de leitura correspondente ao sensor X, é definida a macro de pré-processamento *_SENSOR_X_* (*#define _SENSOR_X_*). Para evitar anteriores definições da macro de pré-processamento *_SENSOR_Y_* que iria provocar também a compilação do código de leitura relativo ao sensor Y, foram removidas todas as definições dessa macro (*#undef _SENSOR_Y_*).

De seguida, é criada uma variável que regista o valor lido do sensor (*int i_sensor_value*). E em seguida, criado um objeto do tipo *cSensor* (*cSensor sensor*). E chamado o método que efetua a leitura do sensor (*sensor.read(i_sensor_value)*).

2.4.2 Polimorfismo Dinâmico

O polimorfismo dinâmico é um mecanismo disponibilizado por linguagens orientadas a objetos, incluíndo a linguagem C++.

Para a gestão da variabilidade através de polimorfismo dinâmico, é criada uma classe abstrata, contendo funções virtuais. A implementação das funções virtuais é efetuada nas classes que herdam da classe abstrata. Desta forma, é possível gerir a variabilidade criando várias classes que herdam da classe base, sendo criada uma para cada ponto de variabilidade. Estas classes implementam nas funções virtuais correspondentes o seu comportamento.

O polimorfismo dinâmico tem a vantagem de oferecer uma gestão de variabilidade bastante limpa, pois separa os vários pontos de variabilidade por classes diferentes. Desta forma, torna o código bastante fácil de ler e de fácil manutenção.

Contudo, o polimorfismo dinâmico não é utilizado em sistemas embebidos de poucos recursos, como WSNs, porque a gestão da variabilidade é feita em tempo de execução, o que afeta o desempenho do dispositivo ao nível do consumo de memória RAM e processamento extra. Este mecanismo deve ser somente utilizado quando não é possível gerir a variabilidade em tempo de compilação.

Código 2.3: Mecanismo de polimorfismo dinâmico - Criação das classes

```
/* Classe abstrata */
class cSensor{
public:
  virtual bool read(int &value) = 0;
};

/* Classe sensor X */
class cX_Sensor : public cSensor{
public:
  bool read(int &value){
    /* Código para o sensor X */
  }
};

/* Classe sensor Y */
class cY_Sensor : public cSensor{
public:
  bool read(int &value){
  /* Código para o sensor Y */
  }
};
```

No código 2.3 é criada a classe abstrata *cSensor*. Nesta classe é criado o método virtual *read* que possibilita a leitura dos sensores.

Também são criadas duas classes que herdam de *cSensor* e implementam a função *read* para a leitura do respetivo sensor. A classe *cX_Sensor* implementa a função de leitura do sensor X e a classe *cY_Sensor* implementa a função de leitura do sensor Y.

Código 2.4: Mecanismo de polimorfismo dinâmico - Instanciação

```
int i_sensor_value = 0;
cSensor *pSensor = new cX_Sensor();

pSensor->read(i_sensor_value);
```

O código 2.4 apresenta a forma como devem ser instanciadas as classes criadas anteriormente para fazer a leitura de dados do sensor X.

Tal como na programação condicional, é criada uma variável para onde é lido o valor do sensor (*int i_sensor_value*). De seguida, é criado um apontador para um

objeto do tipo *cSensor* e atribuído um apontador para um objeto do tipo *cX_Sensor* (*cSensor *pSensor = new cX_Sensor()*). Por fim, é chamado o método que efetua a leitura do sensor (*pSensor->read(i_sensor_value)*).

2.4.3 Programação Orientada a Aspetos

A programação orientada a aspetos (AOP) foi criada por Kiczales et al. [27], em 1997, e visa a melhorar a separação de conceitos em *software*, proporcionando melhores meios para a decomposição de conceitos de *software* em módulos independentes. A AOP tenta resolver um dos principais defeitos da programação orientada a objetos (OOP). A impossibilidade de evitar o espalhamento de código por várias classes ("*crosscutting concerns*") para implementar uma nova funcionalidade, tais como, inserir *logs* de erros ou inserir mecanismos de segurança [4]. Este tipo de programação oferece meios para encapsular os "*crosscutting concerns*" em módulos separados, chamados aspetos [5].

A AOP funciona como uma espécie de um pré-processador, que injeta código pelas classes ou funções, tal como especificado pelo programador através de métodos fornecidos pela linguagem (*aspect*, *pointcut* e *advice*). Depois disso, o código pode ser normalmente compilado pelo compilador da linguagem base (neste caso específico o compilador de C++).

Contudo, a AOP tem duas desvantagens: impõe a utilização de ferramentas externas que são executadas antes do compilador (*AspectC++ weaver*) e obriga os programadores a entenderem novos conceitos de programação [4].

Já existem algumas aplicações de AOP na área de WSNs, uma delas é referida em "*Developing Embedded Software Product Lines with AspectC++*", onde Lohmann and Spinczyk [28], em 2006, propuseram a gestão da variabilidade de uma central meteorológica através de *AspectC++* [29].

Nos códigos seguintes é apresentada uma forma de resolver a variabilidade da leitura de diferentes sensores através de AOP aplicado a linguagem C++ (*AspectC++*).

Código 2.5: Mecanismo de programação orientada a aspetos - Criação da classe

```
/* Class Sensor */
class cSensor
{
public:
  static bool read(int &value){
    /* O código será injetado aqui pelo aspeto */
  }
};
```

O código 2.5 mostra a criação da classe *cSensor*. Neste caso, foi criada uma classe estática, na qual apenas contém o código que é comum aos dois sensores. O código que é diferente para cada um dos sensores será injetado através de AOP.

Código 2.6: Mecanismo de programação orientada a aspetos - Criação dos aspetos

```
/* Aspeto Sensor X */
aspect X_Sensor {
  advice execution("(% cSensor::read(...)"):around(){
    /* Código para o sensor X */
  }
};

/* Aspeto Sensor Y */
aspect Y_Sensor {
  advice execution("(% cSensor::read(...)"):around(){
    /* Código para o sensor Y */
  }
};
```

No código 2.6 é apresentada a criação de dois aspetos, um para cada um dos tipos de sensores existentes (sensor X e sensor Y). Para entender o funcionamento do aspeto, pode-se observar o aspeto que implementa o código específico para o sensor X (*aspect X_Sensor*), que é idêntico ao aspeto para o sensor Y.

Para possibilitar a injeção de código nos pontos necessários, o *AspectC++* fornece o mecanismo de *advice*. Os pontos onde se pretende injetar o código (*pointcut*) são indicados através da utilização da função *execution*. Esta função recebe como parâmetro uma expressão regular (*% cSensor::read(...)*), que indica para procurar o método *read* pertencente à classe cSensor (*cSensor::read*), não interessando qual é o retorno (*%*), nem quais são os parâmetros de entrada (*(...)*). A palavra *around* indica que o código presente entre chavetas (*{}*), é injetado exatamente onde foi encontrado um *pointcut*, dentro da função *read* da classe cSensor.

Código 2.7: Mecanismo de programação orientada a aspetos - Instanciação

```
/* Class Sensor */
int i_sensor_value = 0;

cSensor::read(i_sensor_value);
```

No código 2.7 é mostrado como se efetua a instanciação e leitura do valor do sensor. Neste mecanismo (AOP), não é necessário especificar na instanciação qual o sensor que se pretende efetuar a leitura, visto que ao executar o *AspectC++ weaver*, este injeta o código referente ao sensor selecionado no método *read* da classe *cSensor*.

Neste mecanismo é na mesma criada a variável para onde se pretende efetuar a leitura do valor do sensor (*int i_sensor_value = 0*). Como a classe *cSensor* é estática, a função de leitura é chamada sem a criação de um objeto (*cSensor::read(i_sensor_value)*).

2.4.4 *Template Metaprogramming*

A metaprogramação, literalmente, significa "programa que cria outros programas", ou seja, um programa que manipula código [30]. Este mecanismo introduz mais produtividade no desenvolvimento de programas, pois evita que partes do código sejam escritas manualmente.

C++ *template metaprogramming* (TMP) utiliza as potencialidades das *templates* em C++ para gerar e manipular código em tempo de compilação. Esta técnica de programação foi descoberta por acidente por Erwin Unruh, em 1994 [31].

O TMP permite estender as capacidades do compilador C++, fazendo com que este aja como um interpretador [4]. Isto permite gerir a variabilidade do código em tempo de compilação, não introduzindo qualquer *overhead* do código em tempo de execução.

O TMP é uma linguagem funcional, ao contrário do C++ *standard* que é imperativo. Tornando-se assim uma programação de complexa aprendizagem, pois obriga a mudar a forma dos programadores pensarem e desenvolverem os seus algoritmos.

O TMP ainda não é muito aplicado em sistemas embebidos de poucos recursos, como é o caso das WSNs. O TMP aplicado na área das WSNs é mencionado em "*C-MAC: a Configurable Medium Access Control Protocol for Sensor Networks*", onde Steiner et al. [32], em 2010, propuseram a gestão da camada MAC da pilha protocolar de uma WSN através de TMP. Noutro tipo de sistemas embebidos, mas também com recursos muito limitados, o TMP é citado em "*Exploiting Template Metaprogramming to Customize an Object-Oriented Operating System*", onde Pinto et al. [6], em 2013, efetuaram a gestão da variabilidade do sistema operativo ADEOS [33] com TMP e testaram a sua execução num microcontrolador 8051. Em sistemas embebidos com mais recursos que os dois exemplos apresentados anteriormente, o TMP é referido em "*Use of template metaprogramming to address the heterogeneity of Video Surveillance Systems*", onde Cardoso et al. [34], em 2012, propuseram a gestão da variabilidade de sistemas de vigilância através de TMP.

Nos códigos 2.8 e 2.9, é demonstrado como é possível gerir a variabilidade da leitura de dois sensores diferentes através de TMP.

Código 2.8: Mecanismo de *template metaprogramming* - Criação das estruturas

```
/* Estrutura sensor X */
struct SENSOR_X{
  static bool read(int &value){
    /* Código para o sensor X */
      return true;
  }
};

/* Estrutura sensor Y */
struct SENSOR_Y{
  static bool read(int &value){
    /* Código para o sensor Y */
    return true;
  }
};
```

No código 2.8 são criadas duas estruturas de dados, uma para cada tipo de sensor (sensor X e Y). Ambas as estruturas implementam o método *read* específico para o sensor em questão.

Código 2.9: Mecanismo de *template metaprogramming* - Criação da classe

```
/* Classe Sensor */
template<typename T>
class cSensor{
public:
  static bool read(int &value){
    return  mpl::if_< is_same< T, SENSOR_X >,
        SENSOR_X, SENSOR_Y >::type::read(value);
    }
};
```

No código 2.9, é criada a classe *cSensor*, com o método *read*. Para fazer a escolha de qual a estrutura de dados que será utilizada para ler o sensor, é utilizada a meta-função *if_*, onde o primeiro parâmetro de entrada é do tipo booleano. Caso o primeiro parâmetro de entrada seja verdadeiro é utilizada a estrutura disposta no segundo parâmetro de entrada do *if_*, caso seja falso é utilizada a estrutura disposta no terceiro parâmetro de entrada do *if_*.

O valor do primeiro parâmetro de entrada do *if_* é obtido através da meta-função *is_same_*. Esta meta-função, compara o tipo de dados recebido através da *template* T com o tipo de dados da estrutura que implementa a leitura do sensor X (*SENSOR_X*). Caso sejam iguais retorna verdadeiro, caso contrário retorna falso. Tanto a meta-função *if_* como a meta-função *is_same_* são disponibilizadas pela biblioteca *Boost.MPL* [35].

Código 2.10: Mecanismo de *template metaprogramming* - Instanciação

```
int i_sensor_value = 0;
cSensor<SENSOR_X>::read(i_sensor_value);
```

A instanciação da classe *cSensor* para a leitura do valor do sensor é trivial, e idêntica à utilização dos componentes da biblioteca STL do C++ [36]. A instanciação pode ser visualizada no código 2.10.

2.4.5 Comparação dos Mecanismos de Gestão de Variabilidade

Nas secções anteriores foram apresentados os principais mecanismos de gestão de variabilidade. Nesta secção é apresentada uma comparação entre esses mecanismos.

A comparação pode ser visualizada através da tabela 2.1, que mostra as vantagens e desvantagens dos quatro métodos de gestão da variabilidade descritos nas secções anteriores.

Tabela 2.1: Comparação dos mecanismos de gestão de variabilidade. Baseado em [4]

Mecanismo	Vantagens	Desvantagens
Compilação Condicional	- Bem conhecido; - Bom desempenho; - Alta granularidade.	- Difícil compreensão; - Difícil manutenção.
Polimorfismo Dinâmico	- Bem conhecido; - Fácil utilização; - Dinâmico.	- *Overhead* de desempenho em tempo de execução.
Programação Orientada a Aspetos	- Bom desempenho.	- Necessita de ferramentas externas; - Aprendizagem de nova linguagem; - *Debugging*.
Template Metaprogramming	- Bom desempenho; - Linguagem funcional; - Alta granularidade.	- Menos conhecido e suportado; - Complexo; - *Debugging*.

As aplicações contidas nos nós das WSNs são normalmente estáticas, ou seja, não necessitam de se adaptar a novas funcionalidades em tempo de execução. Tais como, inserir novos sensores, inserir novos atuadores e configurar a pilha protocolar.

Todas as funcionalidades de um nó de uma WSN são conhecidas à priori. Com isso, em tempo de compilação, podem ser resolvidas a maioria das variabilidades do sistema. Por fim, é gerado um executável que só contém as funcionalidades necessárias para a aplicação em concreto, que o nó da WSN irá desempenhar.

Assim, o polimorfismo dinâmico é removido da lista de possíveis mecanismos de gestão da variabilidade para aplicação em WSNs, uma vez que este resolve a variabilidade em tempo de execução, acrescentando *overhead* ao sistema e prejudicando o rendimento.

A compilação condicional é um método que consegue resolver toda a variabilidade de sistemas em tempo de compilação, sendo utilizado em abundância pela indústria e pela comunidade científica. Todavia, o código é de difícil compreensão e manutenção. Por isso, também não foi uma abordagem a seguir nesta dissertação.

2.4.6 Aplicação dos Mecanismos de Gestão de Variabilidade

Nesta secção são apresentadas algumas aplicações dos mecanismos de gestão de variabilidade em sistemas embebidos de poucos recursos. Apenas são apresentadas aplicações que utilizem AOP ou TMP. Aplicações com os outros mecanismos não são abordadas, visto que o polimorfismo dinâmico não é aplicado neste tipo de sistemas, e a compilação condicional é utilizada com abundância, muito conhecida, mas de difícil compreensão e manutenção.

Programação Orientada a Aspetos:

Nesta secção é mostrada uma aplicação, que utiliza programação orientada a aspetos para gerir a variabilidade de aplicações suportadas por sistemas embebidos com poucos recursos.

"*Developing Embedded Software Product Lines with AspectC++*": Neste artigo, Lohmann and Spinczyk [5], em 2006, efetuaram a comparação entre três mecanismos de gestão da variabilidade de código: Programação Condicional em linguagem C, Programação Orientada a Objetos (OOP) em linguagem C++ e Programação Orientada a Aspetos (AOP) em linguagem AspectC++. No artigo, demonstram que a AOP tem grandes potencialidades relativamente aos outros mecanismos, no desen-

volvimento de aplicações de sistemas embebidos, com recursos limitados de memória de código/dados e processamento.

Os autores seguem a metodologia SPL (o diagrama de funcionalidades está presente na figura 2.10) para criar (*from scratch*) uma central meteorológica, preparada para correr num pequeno microcontrolador de 8 *bits* (AVR ATmega). Os objetivos deste exemplo, são: demonstrar que a AOP é aplicável a sistemas embebidos, em geral; comprovar que é acessível, no que diz respeito ao consumo de recursos do código gerado; e por fim, mostrar que é utilizado nos dias de hoje, uma vez que, são disponibilizadas ferramentas de desenvolvimento no mercado, com um alto nível de maturidade.

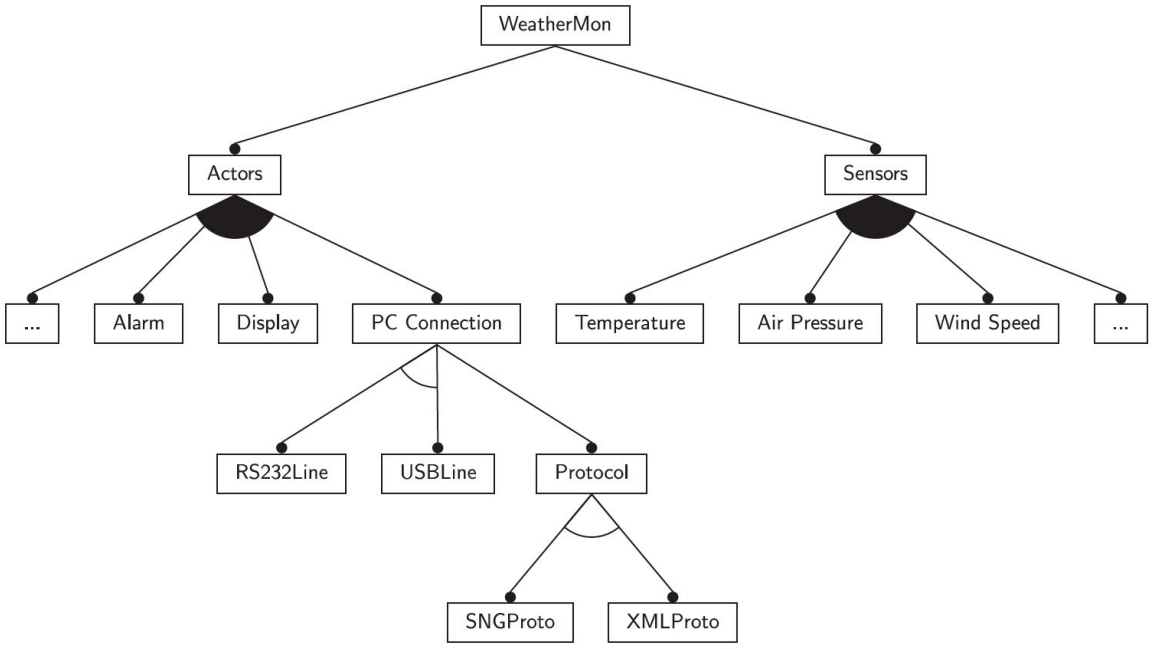


Figura 2.10: Diagrama de funcionalidades da estação meteorológica [5]

A figura 2.11, apresenta um gráfico comparativo entre os três mecanismos de gestão da variabilidade de código: Programação Condicional em linguagem C, OOP em linguagem C++ e AOP em linguagem AspectC++. Os gráficos mostram o consumo de memória RAM e *Flash* em distintas configurações da estação meteorológica. Nestes gráficos, pode ser comprovado que, claramente, a OOP consome bastante mais memória comparativamente às outras duas abordagens. Em relação à comparação entre o AOP e a programação condicional em C, o segundo apresenta melhores

resultados. Contudo, a AOP é uma programação mais legível que a programação condicional.

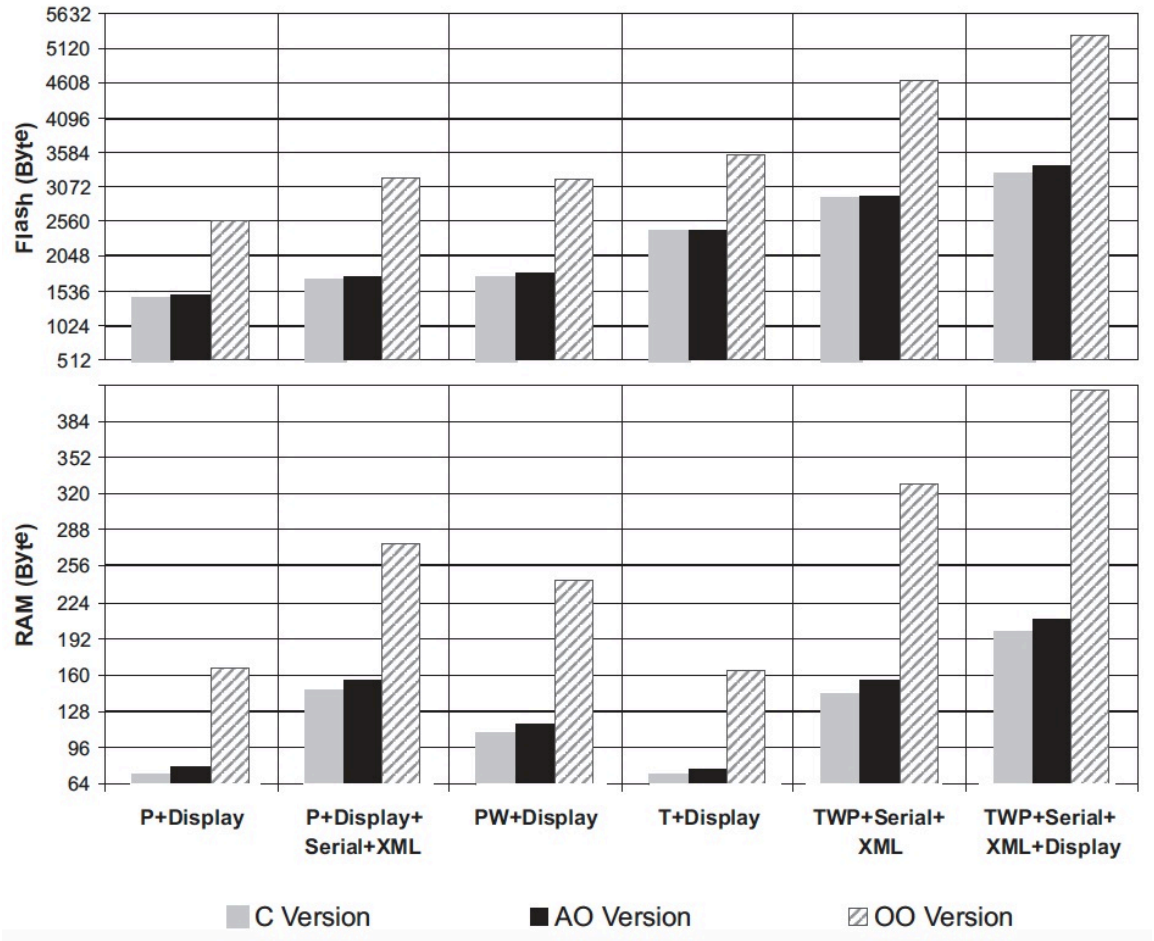


Figura 2.11: Comparação do *footprint* entre programação condicional, OOP e AOP [5]

***Template Metaprogramming*:**

Nesta secção são mostradas duas aplicações que utilizam *template metaprogramming* para gerir a variabilidade de aplicações suportadas por sistemas embebidos com poucos recursos.

"C-MAC: a Configurable Medium Access Control Protocol for Sensor Networks": Steiner et al. [32], em 2010, desenvolveram uma camada MAC (*Medium Access Control*) para a pilha de WSN (C-MAC) que é altamente configurável. Com

a C-MAC os programadores podem instanciar protocolos MAC que satisfaçam as necessidades da sua aplicação. Para conseguirem essa grande liberdade de configuração, o C-MAC foi escrito recorrendo a técnicas de meta-programação estática C++, pois estas técnicas não comprometem o tamanho do código gerado nem a *performance* da aplicação.

Segundo os autores, as WSNs estão altamente dependentes de um eficiente protocolo MAC que faça uso dos poucos recursos presentes numa WSN como: baixa largura de banda e baixos recursos de energia, mas também limitações de memória e processamento dos nós da rede. O C-MAC dá liberdade aos programadores para configurar muitos parâmetros de comunicação, ajustando o protocolo às necessidades das suas aplicações sem comprometer as limitações apresentadas anteriormente.

Os autores comparam o *C-MAC* com o *IEEE 802.15.4 MAC ZigBeeNet* produzido pela empresa *Meshnetics*, e afirmam que a implementação em meta-programação C++ apresenta funcionalidades equivalentes com baixo *footprint*. As técnicas de meta-programação estática utilizadas foram: *templates*, funções *inline* e *inline assembly*. Uma vez que, não afetam a *performance* nem o tamanho do código produzido.

Exploiting Template Metaprogramming to Customize an Object-Oriented Operating System: Pinto et al. [6], em 2013, indicam que hoje em dia, o crescimento da complexidade dos sistemas embebidos exige configurabilidade, variabilidade e reutilização. A compilação condicional e programação orientada a objetos são as duas abordagens mais utilizadas para gestão da variabilidade. Enquanto que a primeira aumenta a complexidade da manutenção do código, a segunda insere modularidade e adaptabilidade necessária para simplificar o desenvolvimento de *software* reutilizável e customizável, contudo, deteriora a *performance* e o consumo de memória.

Como alternativa aos dois métodos apresentados, os autores propõem a utilização de C++ *template metaprogramming*, pois alegam que o TMP não afeta a *performance* do sistema, nem consome muitos recursos de memória. Isto porque, a gestão da variabilidade do TMP é feita estaticamente, gerando apenas as funcionalidades desejadas, garantindo assim, que o código é otimizado e ajustado aos requisitos das aplicações e recursos de *hardware*.

Como cenário de teste, os autores aplicaram o mecanismo de TMP à gestão de variabilidade de um sistema operativo em tempo real (RTOS) orientado a objetos (ADEOS [33]).

De modo a expressar o grau de variabilidade do RTOS ADEOS, os autores optaram por utilizar um diagrama de funcionalidades, pois afirmam que permite expressar graficamente a variabilidade do sistema, desconsiderando o mecanismo de implementação. Esse diagrama pode ser visualizado na figura 2.12.

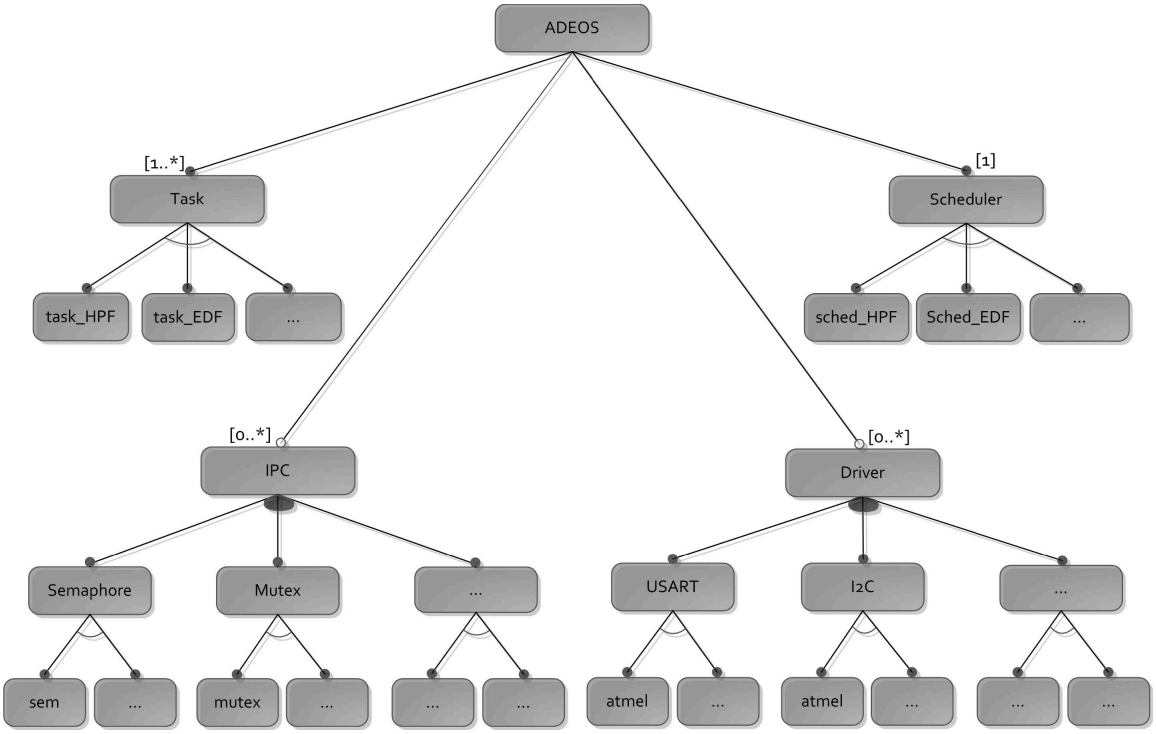


Figura 2.12: Diagrama de funcionalidades do RTOS ADEOS [6]

A plataforma alvo utilizada para correr o RTOS ADEOS foi um microcontrolador de poucos recursos, um Atmel 8051 (AT89C51). As ferramentas de programação utilizada foram as contidas no pacote do *IAR Embedded Workbench 8051* (compilador e *debugger*). O *debugger* foi utilizado para obtenção das informações de execução da aplicação.

Para avaliar o desempenho da implementação, os autores fizeram a comparação entre uma gestão de variabilidade implementada em polimorfismo dinâmico com uma implementação em TMP.

Segundo as simulações realizadas pelos autores, através de TMP foi conseguido um tempo de execução de menos 20% e um consumo de menos 40% de memória de código em relação à implementação com polimorfismo dinâmico. Contudo, a implementação em TMP apresenta mais 8% de linhas de código e mais 41% de classes

que a implementação em polimorfismo dinâmico. O que significa que a implementação em TMP permite um grau de modularidade e encapsulamento mais alto que o código implementado em polimorfismo dinâmico, possibilitando melhor gestão, manutenção e reutilização do código.

Com a comparação dos resultados, os autores concluíram que a abordagem TMP apresenta vantagens em termos de *performance* e consumo de memória, comparando com a abordagem da programação orientada a objetos tradicional.

2.5 Conclusões

Este capítulo apresentou uma visão geral sobre os temas abordados nesta dissertação, demonstrando as ferramentas e os conceitos relacionados com o tema, permitindo assim definir o rumo ao projeto. A visão incidiu sobre: (i) os cenários de aplicação e a variabilidade das redes de sensores sem fios; (ii) o termo *framework* e alguns exemplos de *frameworks* de gestão de WSNs implementadas por terceiros; (iii) o conceito de engenharia SPL e os seus diagramas de funcionalidades; (iv) os mecanismos de gestão de variabilidade ao nível da linguagem de programação C++, demonstrando as vantagens e desvantagens de cada um. Por fim, são apresentados trabalhos de autores que aplicaram algumas dessas técnicas de programação em aplicações de WSNs ou do género.

Com a revisão do estado da arte decidiu-se dois pontos-chave do rumo aplicado à dissertação. O primeiro foi o desenvolvimento de uma *framework* com um grau de abstração não muito elevado, de forma a não obrigar programadores habituados a programar diretamente sobre a plataforma (em C/C++) a mudarem drasticamente a forma de programar e a instalarem ferramentas extras para além do compilador. O segundo foi a escolha de *template metaprogramming* em linguagem C++ como forma de gerir a variabilidade da *framework* em tempo de compilação, visto que esta apresenta vantagens evidentes em relação aos outros mecanismos de gestão da variabilidade existentes.

Capítulo 3

Especificação do Sistema

Este capítulo apresenta uma visão geral sobre todos os componentes pertencentes à base de implementação do sistema. Primeiramente, é descrito o SOC de comunicação utilizado (secção 3.1), CC2530, na qual são apresentadas as características e os periféricos do microcontrolador presente nesse SOC. De seguida, é apresentada a pilha de *software* TIMAC (secção 3.2), descrevendo as camadas constituintes da mesma. Seguidamente, é apresentado o ambiente de desenvolvimento, *IAR Workbench* para 8051 (secção 3.3), utilizado para a edição, compilação e análise do código. Posteriormente, é exposta a técnica de *template metaprogramming* (secção 3.4), apresentando um exemplo que demonstra a sua utilização e as suas vantagens, e por fim é exibida a *framework Boost.MPL* que foi utilizada no apoio à programação TMP. No final, é apresentada a linguagem XML e XSLT (secção 3.5), que possibilita a geração de código respeitante à configuração da *framework*.

3.1 CC2530

O CC2530 [7], representado na figura 3.1, é um *System On Chip* (SOC) desenhado pela *Texas Instruments* para funcionar sobre o protocolo de comunicação IEEE 802.15.4, sendo este de baixas taxas de transmissão e de baixo custo, onde estão inseridas as redes de sensores sem fios (WSNs). Este dispositivo apresenta baixos consumos de energia, o que o torna uma excelente escolha no desenvolvimento de aplicações que assim o exijam. O SOC possui integrado um *transceiver* RF e um microcontrolador

Figura 3.1: SOC CC2530, retirada de [7]

da família 8051, que é descrito na secção seguinte (3.1.1).

3.1.1 Microcontrolador 8051

O 8051 é um microcontrolador de 8 *bits*, lançado pela Intel em 1980 [37]. O microcontrolador implementa uma arquitetura do tipo *Harvard*, fazendo a distinção física entre a memória de instruções ou código e a memória de dados. O conjunto de instruções do microcontrolador é do tipo CISC (*Complex Instruction Set Computer*).

Vários fabricantes adotaram a arquitetura 8051, levando cada fabricante a apresentar a sua versão do microcontrolador, com diferentes características ao nível do tamanho das memórias e da variedade de periféricos. Nos tópicos seguintes, são apresentadas as características e os periféricos contidos no microcontrolador 8051 do SOC CC2530 da *Texas Instruments* [38].

3.1.2 Características do Microcontrolador

O microcontrolador dispõe de 256 *KBytes* de memória *flash* utilizada para armazenar as instruções (memória de código). A memória de dados, está dividida em duas: a memória de dados externa (XRAM), onde são disponibilizados 4 *KBytes* de memória; e a memória de dados interna (RAM), que possui 256 *Bytes* de tamanho. A memória de dados interna está dividida em dois blocos de 128 *Bytes*, em que as posições de memória do bloco inferior podem ser acedidas de forma direta e indireta (DATA) e as posições de memória do bloco superior, só podem ser acedidas de forma indireta (IDATA). A memória IDATA só pode ser acedida indiretamente porque nessas posições de memória estão mapeados os registos SFR (*Special Function Registers*), e esses é que são acedidos de forma direta. Os SFR são registos especiais que possibilitam a configuração e acesso aos periféricos do microcontrolador. Nas posições de memória destinada aos SFRs, todos os registos que estão mapeados em posições de memória divisíveis por 8 são registos endereçáveis ao *bit*, todos os outros só são endereçáveis ao *Byte*.

O microcontrolador dispõe também de quatro bancos de registos de uso geral, em que cada banco possui 8 registos com um *Byte* de tamanho cada (R0 a R7). Estes bancos de registos estão mapeados nas primeiras posições de memória da memória de dados (RAM).

Além dos registos de SFR, este microcontrolador dispõe também de um espaço de memória destinada a armazenar mais registos especiais, esses registos têm o nome

de XREG e estão mapeados na memória externa. Tal como os registos do SFR, os registos do XREG são utilizados para configuração e acesso aos periféricos do microcontrolador.

3.1.3 Periféricos do Microcontrolador

A *Texas Instruments* inseriu no microcontrolador 8051 utilizado no SOC CC2530 uma ampla variedade de periféricos, tais como:

- **ADC:** contém como periférico um ADC (*Analog-to-Digital Converter*) *sigma delta* de oito canais. Este ADC tem uma resolução de 7 *bits* com uma largura de banda de 30KHz e uma resolução de 12 *bits* com uma largura de banda de 4KHz;

- **Amplificador operacional:** possui um amplificador operacional que pode ser utilizado para *buffer* ou para dar ganho à entrada do ADC;

- **Comparador analógico:** dispõe de um comparador analógico de duas entradas;

- **DMA:** contém um DMA (*Direct Memory Access*) de 5 canais;

- ***Encriptador/Desencriptador de AES:*** possui um módulo que efetua a encriptação e desencriptação de chaves AES (*Advanced Encryption Standard*) de 128 *bits*, utilizado na segurança do protocolo MAC;

- **Gerador de números aleatórios:** contém um módulo gerador de números aleatórios de 16 *bits*;

- **GPIOs:** dispõe de 21 pinos de uso geral, que podem ser configurados tanto ao nível da direção do fluxo do sinal, que pode ser de entrada ou de saída, como ao nível de conter, ou não, resistência interna de *pull-up* ou *pull-down*. Estes 21 pinos estão divididos por três portos: o porto P0, o porto P1 e o porto P2. O porto P0, bem como o porto P1, contém oito pinos e o porto P2 só contém 5 pinos;

- **Interface de *Debug*:** contém uma interface que permite efetuar depuração em tempo real ao *software*;

- **Timers:** dispõe de 5 temporizadores, nos quais: um é de 16 *bits* (*Timer 1*), dois são de 8 *bits* (*Timer 3* e *Timer 4*) e os dois últimos são utilizados para o sistema, um utilizado no modo de *sleep* (*Sleep Timer*) e o outro utilizado pela camada MAC (*Timer 2*);
- **USARTs:** possui duas portas série, USART0 e USART1, que podem ser utilizadas ambas como SPI *master* ou *slave*;
- **Vetor de interrupções:** contém um vetor de 18 interrupções configuráveis;
- ***Watchdog timer*:** possui um temporizador de *watchdog* que permite efetuar o *reset* ao microcontrolador caso o *software* bloqueie.

3.2 TIMAC

TIMAC é uma pilha de *software* disponibilizada pela *Texas Instruments* que implementa a camada de acesso ao meio (MAC) no protocolo IEEE 802.15.4 [26]. Esta pilha de *software* suporta múltiplas plataformas de *hardware*, entre as quais o CC2530. A TIMAC contém uma base de código que acelera o desenvolvimento de aplicações para WSNs. As bibliotecas da TIMAC apenas podem ser compiladas pelo compilador IAR para 8051, visto que existem algumas partes do código que estão em bibliotecas de código binário compilado pelo compilador IAR, como é o caso da camada MAC, que está numa biblioteca binária para ocultar os detalhes da sua implementação.

O código da TIMAC é praticamente todo implementado em linguagem C, tirando algumas pequenas exceções que utilizam linguagem *assembly*. A gestão da variabilidade das várias funcionalidades que compõem toda a pilha de *software* é efetuada em tempo de compilação, utilizando o mecanismo de compilação condicional (o funcionamento deste mecanismo foi demonstrado na secção 2.4.1), através das diretivas do pré-processador. Este mecanismo de gestão de variabilidade apresenta bom desempenho no código gerado, visto que a variabilidade é toda gerida em tempo de compilação. Contudo, apresenta um código de difícil leitura, e consequentemente de difícil manutenção, devido às enumeras diretivas de pré-processamento que existem espalhadas pelas bibliotecas de código (*#ifdef, #else, #endif e #elseif*).

A TIMAC contém um conjunto de camadas de abstração que implementam o acesso ao meio físico. A figura 3.2 apresenta um esquema representativo dessas camadas.

Figura 3.2: Pilha de *software* TIMAC

As camadas existentes na TIMAC são quatro: a camada MAC (representada a vermelho), que implementa o acesso ao meio no protocolo IEEE 802.15.4; a camada HAL (representada a verde), que implementa a abstração do *hardware*; a camada OSAL (representada a laranja), que implementa a abstração do sistema operativo; e por último, a camada de aplicações (representada a azul), que facilita a criação de tarefas do sistema operativo.

Esta dissertação apenas se focou nas camadas HAL e OSAL, onde a camada de aplicações foi associada internamente à camada OSAL. As subsecções seguintes apresentam a descrição dessas camadas.

3.2.1 Camada HAL

A camada HAL efetua a abstração do *hardware* presente no microcontrolador e na placa de desenvolvimento. Esta camada disponibiliza vários módulos que facilitam o acesso ao *hardware*, tais como:

- **ADC:** permite a gestão do ADC presente no microcontrolador e dos seus oito

canais;

- **DMA:** concede a gestão do DMA e dos seus cinco canais;
- **KEY:** proporciona a gestão dos botões, *switches* e *joystick* que a placa de desenvolvimento possui;
- **LED:** permite efetuar a gestão dos LEDs presentes na placa de desenvolvimento, apresenta funcionalidades como ligar/desligar LEDs e colocar LEDs a piscar;
- **USART:** permite fazer a gestão das portas série presentes no microcontrolador.

3.2.2 Camada OSAL

A camada OSAL faz a abstração do sistema operativo. Esta camada contém vários módulos que propiciam a gestão de todo o sistema operativo, tais como:

- **Eventos:** permite a criação de eventos, que são utilizados para sincronização entre tarefas;
- **Gestão de potência e consumos:** efetua a gestão do consumo do microcontrolador, colocando-o em *sleep* sempre que possível;
- **Interrupções:** permite ativar e desativar as interrupções do microcontrolador;
- **Memória:** faz a gestão da memória de alocação dinâmica;
- **Mensagens:** fornece mecanismos de gestão de mensagens, que podem ser utilizadas para comunicação de tarefa para tarefa, ou de ISRs para tarefa;
- **Sistema:** efetua a inicialização do sistema operativo e a gestão das tarefas criadas;
- **Temporizadores:** fornece mecanismos de gestão de temporizadores utilizados para despoletar eventos num tempo predefinido pelo utilizador;
- **Tarefas:** disponibiliza mecanismos que permitem criar e adicionar tarefas ao sistema operativo.

3.3 IAR *Workbench* para 8051

O IAR *Workbench* para 8051 [8] é desenvolvido pela empresa *IAR Systems* e fornece um conjunto de ferramentas que permitem a programação do microcontrolador 8051. A figura 3.3 apresenta um esquema sobre essas ferramentas.

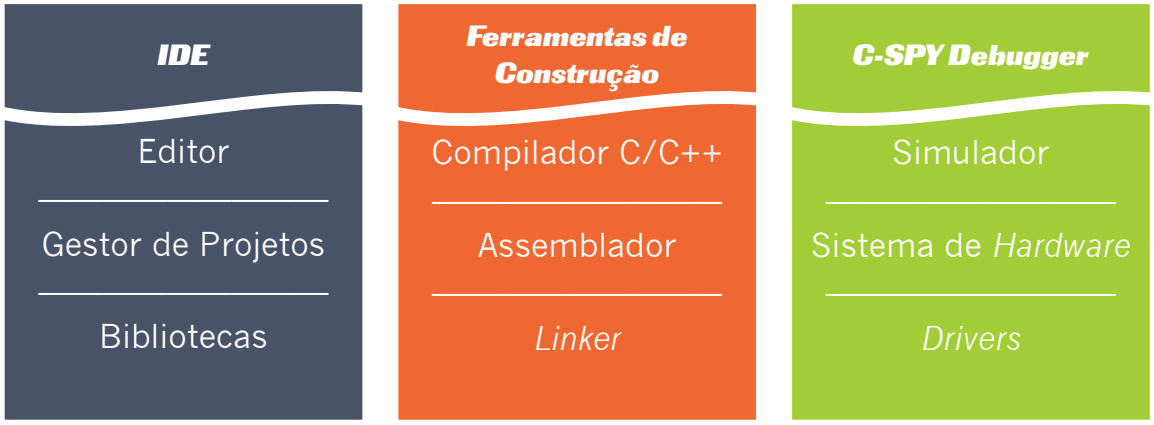


Figura 3.3: Ferramentas do IAR, baseado em [8]

As ferramentas do IAR *Workbench* estão divididas em três grupos:

- **IDE:** é um ambiente de desenvolvimento integrado que contém ferramentas de gestão de projetos e edição de código. Possui também os ficheiros de configuração para diferentes dispositivos, de diversos fabricantes e as bibliotecas de apoio à programação desses dispositivos;

- **Ferramentas de construção:** em inglês *build tools*, são ferramentas que permitem gerar o código binário para o microcontrolador. Estas são compostas por um compilador de C/C++ com um alto nível de otimização, um assemblador e um *linker*;

- **C-SPY *Debugger*:** é uma ferramenta que permite efetuar a depuração do código, que pode ser executada de duas formas: através do simulador ou diretamente na placa através de uma interface *JTAG*.

3.3.1 Compilador IAR C/C++ para 8051

O compilador IAR para 8051 [39] suporta duas linguagens de programação: linguagem C e linguagem C++. Tal como foi enunciado anteriormente, a linguagem escolhida no desenvolvimento da *framework* é o C++. A IAR fornece duas versões do compilador para a linguagem C++: uma com as funcionalidades básicas da linguagem, com o nome *Embedded* C++ (EC++); e outra com um número mais alargado de funcionalidades, com o nome *Extended Embedded* C++ (EEC++). A tabela 3.1 demonstra quais as funcionalidades da linguagem C++ suportadas, pelas versões do compilador descritas anteriormente.

Compatibilidade das funcionalidades C++:

Tabela 3.1: Compatibilidade das funcionalidades C++ no IAR para 8051

Funcionalidades	**EC++**[1]	**EEC++**[1]
Atributo *mutable*	✗	✓
Classes	✓	✓
Exceções	✗	✗
Funções *inline*	✓	✓
Herança múltipla	✗	✗
Herança única	✓	✓
Herança virtual	✗	✗
Namespaces	✗	✓
Novos operadores de *cast*	✗	✓
Operadores *new* e *delete*	✓	✓
Overload de operadores	✓	✓
Polimorfismo	✓	✓
RTTI	✗	✗
STL	✗	✓
Templates	✓	✓
Templates (suporte total)	✗	✓

[1] ✓- Possui compatibilidade | ✗- Não Possui compatibilidade

A versão EC++ dá suporte às seguintes funcionalidades básicas da linguagem C++: classes, funções *inline*, herança única, operadores *new* e *delete*, *overload* de operadores, polimorfismo e *templates* básicas. A versão EEC++ dá suporte às funcionalidades da versão EC++ e ainda contempla mais funcionalidades extra, tais como: atributos *mutables*, *namespaces*, novos operadores de *cast* (*dynamic_cast*, *static_cast*, *reinterpret_cast* e *const_cast*), STL e suporte total às *templates*. Contudo, há funcionalidades do C++ que não são suportadas pela versão EC++ nem pela versão EEC++, estas são: exceções, herança múltipla, herança virtual e RTTI.

Analisando as compatibilidades das duas versões, a versão escolhida para compilação da *framework* desenvolvida foi a EEC++, visto que dá suporte a um número mais alargado de funcionalidades, mas principalmente porque oferece o suporte total às *templates*, que é essencial para a aplicação das técnicas de *template metaprogramming*.

Níveis de otimização:

O compilador IAR fornece vários níveis de otimização, que podem ser selecionados pelo utilizador, segundo as necessidades da aplicação a implementar:

- ***None*:** não é aplicado nenhuma otimização ao código, contudo, é o que tem melhor suporte de depuração de código;
- ***Low*:** aplica uma otimização de nível baixo;
- ***Medium*:** consiste numa otimização de nível médio;
- ***High*:** é uma otimização de nível alto, o qual divide-se em três subníveis, que são os seguintes:
  - ***Balanced*:** efetua um compromisso entre o espaço de memória ocupado pela aplicação e a velocidade de processamento da aplicação;
  - ***Size*:** otimização mais direcionada para a redução dos consumos de memória de código e de dados pela aplicação;
  - ***Speed*:** otimização mais direcionada para a redução dos tempos de processamento da aplicação.

3.4 TMP

A metaprogramação literalmente significa "programa que cria outros programas" [30]. Enquanto um programa é um conjunto ordenado de instruções que criam e modificam dados, um metaprograma é um conjunto ordenado de instruções que criam e modificam programas [9].

O *template metaprogramming*, foi descoberto por acidente por Erwin Unruh em 1994, e é uma linguagem funcional, ao contrário do C++ *standard* que é imperativo. Tornando-se assim uma programação de complexa aprendizagem, pois obriga a mudar a forma dos programadores pensarem e desenvolverem os seus algoritmos.

C++ *template metaprogramming* (TMP) estende as capacidades do compilador C++, fazendo com que este aja como um interpretador [4]. Este feito é conseguido utilizando as potencialidades das *templates* em C++ para gerar e manipular código em tempo de compilação. A figura 3.4 apresenta o processo de compilação de um programa em linguagem C++.

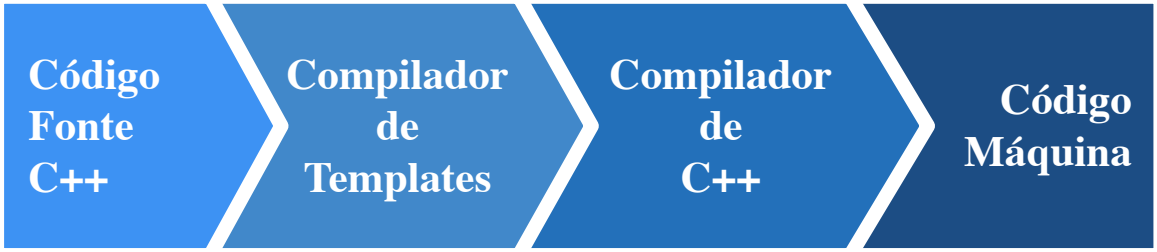


Figura 3.4: Processo de compilação das *templates*, baseado em [9]

As *templates* são uma funcionalidade do C++ que permitem a criação de código genérico. As *templates* evitam a repetição de código quando se pretende implementar ações iguais, sobre tipos de dados diferentes. Antes da fase de compilação normal é efetuada uma compilação ao nível das *templates*, que cria os elementos específicos por cada instanciação das *templates*, com diferentes tipos de dados. Esse código específico é posteriormente apresentado ao compilador de C++ que gera o código máquina a partir do mesmo. O TMP aproveita as potencialidades das *templates* e do seu compilador para manipular código em tempo de compilação. Desta forma, é possível resolver todas as ações que podem ser resolvidas em tempo de compilação, evitando que essas ações afetem o desempenho do programa em tempo de execução, gerando

assim um código otimizado. A secção 3.4.1 apresenta um exemplo de otimização do cálculo do fatorial.

3.4.1 Fatorial

O cálculo do fatorial de um número constante é um bom exemplo que demonstra as potencialidades da utilização de *template metaprogramming*. O código 3.1 apresenta uma forma de como implementar o cálculo do fatorial do número quatro, de forma dinâmica (em tempo de execução), utilizando uma função.

Cálculo em tempo de execução:

Código 3.1: Cálculo do fatorial - Tempo de execução

```
unsigned char Factorial(unsigned char N)
{
  if(N)
  {
    return Factorial(N-1) * N;
  }
  return 1;
};

int main()
{
  unsigned char fact = Factorial(4);
  return 0;
}
```

A implementação do cálculo do fatorial de um número pode ser efetuada de forma dinâmica, através de uma função iterativa ou recursiva. Neste caso, em específico, foi implementada a função fatorial utilizando a forma recursiva. A função que implementa o fatorial é a *Factorial*, e chama-se recursivamente a si mesma até o valor passado como parâmetro de entrada ser 0 (zero). A tabela 3.2 apresenta o código *assembly* que o compilador gera a partir do código demonstrado anteriormente.

Tabela 3.2: Código *assembly* gerado no cálculo do fatorial - Tempo de execução

Função main		**Função Factorial**	
unsigned char fact = Factorial(4)		*unsigned char Factorial(unsigned char N){*	
main:		Factorial:	
MOV	R1, #0x04	MOV	A, R6
LCALL	Factorial	PUSH	A

```
    MOV     A, R1
    MOV     R0, A
  return 0;
    MOV     R2, #0x00
    MOV     R3, #0x00
    RET
```

```
    MOV     A, R1
    MOV     R6, A
  if(N){
    MOV     A, R6
    JZ      0x00B2
  return Factorial(N-1) * N;
    MOV     A, R6
    MOV     B, A
    PUSH    B
    MOV     A, #0xFF
    ADD     A, R6
    MOV     R1, A
    LCALL   Factorial
    MOV     A, R1
    POP     B
    MUL     AB
    MOV     R1, A
    SJMP    0x00B4
  return 1;
    MOV     R1, #0x01
    POP     A
    MOV     R6, A
    RET
```

Neste caso, o cálculo do fatorial é todo efetuado em tempo de execução. Na tabela é apresentado do lado esquerdo, o código *assembly* da função *main* e do lado direito, o código da função *Factorial*. Esta implementação não apresenta um código otimizado, afetando o desempenho do programa em três aspetos: no consumo de memória de código, devido ao número de instruções utilizadas para o cálculo do fatorial; no consumo de memória de dados, devido ao crescimento da pilha na chamada da função recursiva; e nos tempos de execução do programa, no cálculo do valor do fatorial.

Cálculo em tempo de compilação:

Como o fatorial do número quatro é um número constante (*4! = 24*), era interessante que o seu cálculo fosse efetuado em tempo de compilação, evitando que as

operações do seu cálculo afetem o desempenho do programa durante a sua execução. Com a utilização de TMP é possível obter esse feito, e o código 3.2 apresenta a sua implementação.

Código 3.2: Cálculo do fatorial - Tempo de compilação

```
template <int N>
struct Factorial
{
  enum
  {
      value = Factorial<N-1>::value * N
  };
};

template <>
struct Factorial<0>
{
  enum
  {
      value = 1
  };
};

int main()
{
  unsigned char fact = Factorial<4>::value;
  return 0;
}
```

Enquanto que na versão da implementação de forma dinâmica é utilizada uma função, na implementação de forma estática com TMP é utilizada uma meta-função. A meta-função é a forma de criar funções em tempo de compilação. Os parâmetros de entrada de uma meta-função são obtidos através de *templates* e o retorno através de um tipo de dados constante, neste caso o *enum*.

A implementação consiste em criar uma meta-função de nome *Fatorial*, a qual é implementada através de duas *templates*: uma *template* genérica, que chama recursivamente a meta-função *Factorial* e uma *template* especializada, que efetua a condição de paragem quando for chamada a meta-função *Fatorial* com o valor zero. A figura 3.5 esquematiza como é efetuado o cálculo do fatorial em tempo de compilação através de TMP.

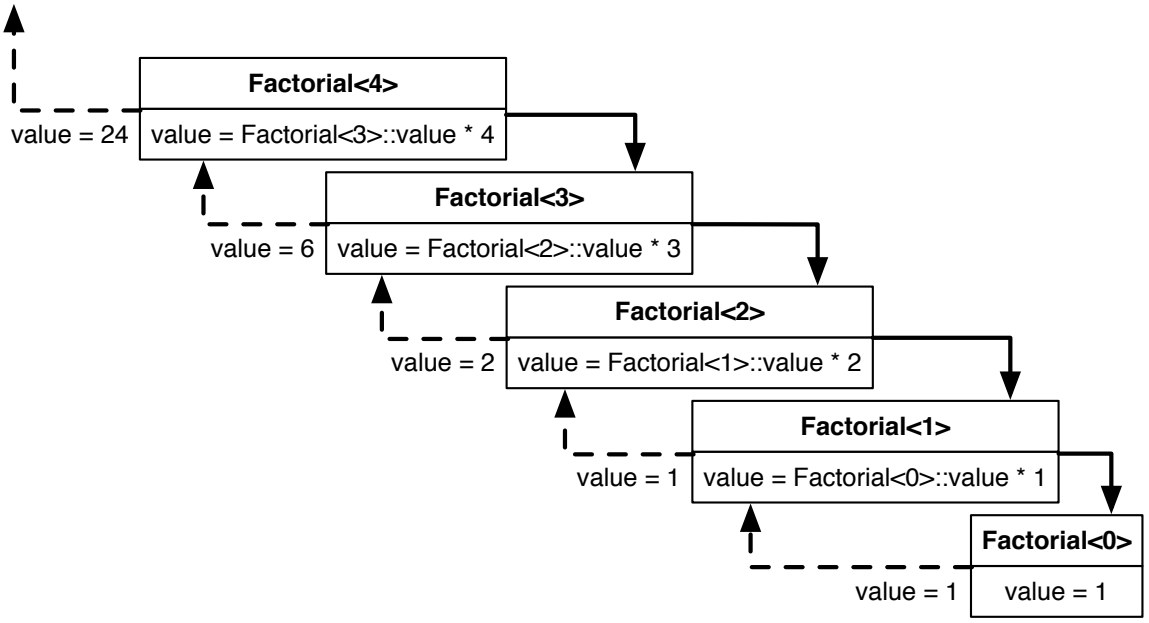


Figura 3.5: Iterações necessárias no cálculo do fatorial - Tempo de compilação

O cálculo do fatorial é efetuado em cinco iterações, que são as seguintes:

1ª Iteração: é efetuada a chamada da meta-função *Fatorial* com o valor 4 (quatro) passado como parâmetro de entrada de *template*. Como esse valor é diferente de 0 (zero), é utilizada a *template* genérica, que atribui à constante *value*, a multiplicação entre o valor 4 (quatro) e o *value* resultante da chamada recursiva à meta-função *Fatorial* com o valor 3 (três), passado como parâmetro de entrada de *template*.

2ª Iteração: é efetuada a chamada da meta-função *Fatorial* com o valor 3 (três) passado como parâmetro de entrada de *template*. Como esse valor é diferente de 0 (zero), é utilizada a *template* genérica, que atribui à constante *value*, a multiplicação entre o valor 3 (três) e o *value* resultante da chamada recursiva à meta-função *Fatorial* com o valor 2 (dois), passado como parâmetro de entrada de *template*.

3ª Iteração: é efetuada a chamada da meta-função *Fatorial* com o valor 2 (dois) passado como parâmetro de entrada de *template*. Como esse valor é diferente de 0 (zero), é utilizada a *template* genérica, que atribui à constante *value*, a multiplicação entre o valor 2 (dois) e o *value* resultante da chamada recursiva à

meta-função *Fatorial* com o valor 1 (um), passado como parâmetro de entrada de *template*.

4ª Iteração: é efetuada a chamada da meta-função *Fatorial* com o valor 1 (um) passado como parâmetro de entrada de *template*. Como esse valor é diferente de 0 (zero), é utilizada a *template* genérica, que atribui à constante *value*, a multiplicação entre o valor 1 (um) e o *value* resultante da chamada recursiva à meta-função *Fatorial* com o valor 0 (zero), passado como parâmetro de entrada de *template*.

5ª Iteração: é efetuada a chamada da meta-função *Fatorial* com o valor 0 (zero) passado como parâmetro de entrada de *template*. Como esse valor é igual a 0 (zero), é utilizada a *template* especializada, que atribui à constante *value*, o valor 1 (um).

No final de todas as iterações, é obtido em tempo de compilação, o valor do fatorial do número 4 (quatro), que é o valor constante 24 (vinte e quatro). A tabela 3.3 apresenta o código *assembly* gerado na compilação do código 3.2.

Tabela 3.3: Código *assembly* gerado no cálculo do fatorial - Tempo de compilação

Função main
unsigned char fact = Factorial<4>::value;
main:
MOV R0, #0x18
return 0;
MOV R2, #0x00
MOV R3, #0x00
RET

Como se pode verificar, à variável *fact* (contida no registo R0) é atribuído diretamente o valor constante 24 (24 em decimal = 0x18 em hexadecimal). Desta forma, pode-se concluir, que o cálculo do fatorial foi efetuado em tempo de compilação, melhorando assim o desempenho do programa em tempo de execução.

3.4.2 *Boost.MPL*

Como a TMP não é um tipo de programação muito trivial, a utilização de uma biblioteca de *metaprogramming* para auxílio da programação, traz muitas vantagens, em comparação com o desenvolvimento de raiz (*from scratch*) [30]. Estas vantagens são: qualidade, uma vez que apresenta um código mais limpo e fácil de entender; reutilização, pois contém vários componentes genéricos que podem ser reutilizados as vezes que forem necessários; portabilidade, porque dá suporte a vários compiladores; e por último, produtividade, uma vez que acelera o processo de desenvolvimento de código TMP.

A *Boost.MPL* é uma *framework* de alto nível, que fornece vários componentes de *template metaprogramming* de uso geral [35], tais como: sequências, iteradores, algoritmos, meta-funções e tipos de dados. A descrição desses componentes é efetuada nos tópicos seguintes:

- **Sequências:** são componentes que permitem agrupar sequências de tipos de dados em tempo de compilação, como por exemplo: vetores, listas ligadas e *queues*;

- **Iteradores:** permitem iterar sobre elementos pertencentes a uma sequência de tipos de dados em tempo de compilação, como por exemplo: ir para o próximo elemento da sequência e ir para o elemento anterior da sequência;

- **Algoritmos:** permitem o processamento de dados contidos nas sequências de tipos de dados em tempo de compilação, como por exemplo: procura de elementos e contagem de elementos;

- **Meta-funcões:** são funções utilizadas para manipulação de dados em tempo de compilação, como por exemplo: comparações de tipos de dados, seleção de tipos de dados, operadores lógicos e operadores *bitwize*;

- **Tipos de dados:** são tipos de dados equivalentes ao *standard* da linguagem C++, mas utilizados para manipulação em tempo de compilação, tais como: booleanos, inteiros, longos, entre outros.

3.5 Geração de código

Como o TMP não é uma linguagem de fácil leitura e escrita, recorreu-se às linguagens XML e XSLT para facilitar a configuração da *framework*, bem como permitir futuramente a criação de uma interface gráfica de manipulação da mesma. Isto porque, a linguagem XML é um *standard* bastante utilizado a nível científico e empresarial, existindo um vasto leque de ferramentas compatíveis com a mesma, que podem ser utilizadas para desenvolver de forma trivial a interface gráfica.

3.5.1 XML

O XML (*Extensible Markup Language*) [40] é uma simples e flexível linguagem de marcação, padronizada pela W3C (*World Wide Web Consortium*) [41] e, inicialmente desenvolvida para facilitar a partilha de informação através da internet. Hoje em dia, este padrão foi estendido a um número mais alargado de aplicações, sendo um padrão não só utilizado a nível *web*, mas também a nível local, empregue, por exemplo, na criação de ficheiros de configuração. Esta linguagem utiliza marcações (*tags*), que permitem estruturar informação de forma organizada e hierárquica, sendo uma linguagem de fácil leitura e escrita.

3.5.2 XSLT

O XSLT (*eXtensible Stylesheet Language for Transformation*) [42] é uma linguagem de transformação, padronizada também pela W3C, utilizada para definir ações de transformação aplicadas a informação contida em ficheiros XML, que transformam essa informação num formato de dados definido por essas ações. Ou seja, pode-se através de XSLT, definir ações que transformem os dados contidos num ficheiro XML, num ficheiro de formato HTML, PDF ou outro. Tal como o XML, o XSLT recorre a *tags*, mas desta vez para criar ações de transformação.

3.5.3 Processador de XSLT

O processador de XSLT é um *software* que executa o processamento de ficheiros XML e XSLT. A figura 3.6, apresenta a forma como é feito esse processamento.

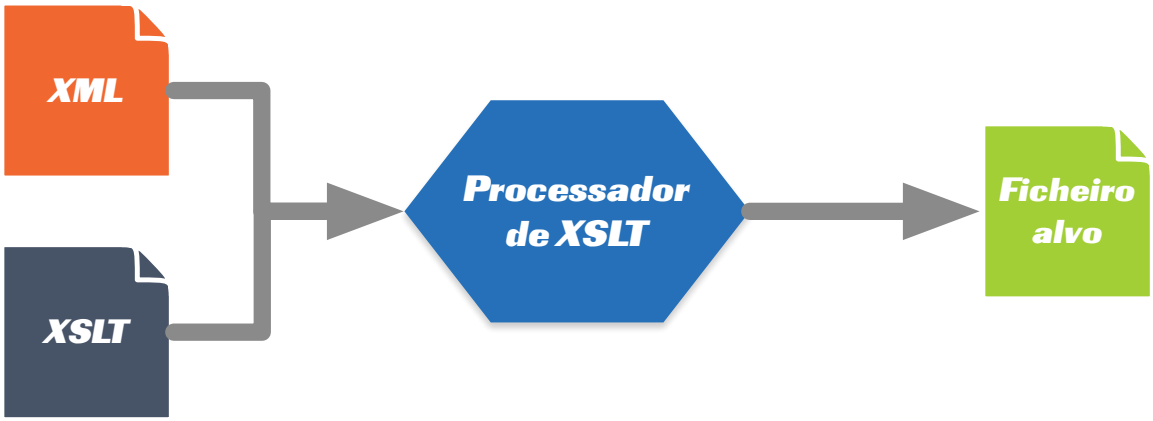


Figura 3.6: Processamento de XSLT

O processador de XSLT, recebe como entrada um ficheiro XML e um ficheiro XSLT. O ficheiro XML contém informação fonte estruturada sobre a forma de *tags*. O ficheiro XSLT possui a definição das transformações, também sobre a forma de *tags*, aplicadas à informação contida no ficheiro XML, de forma a gerar um ficheiro alvo com as formatações definidas pelas transformações. O processador utilizado nesta dissertação foi o MSXSL (*MicroSoft eXtensible Stylesheet Language*) [43], disponibilizado pela *Microsoft* [44].

3.6 Conclusões

Este capítulo apresentou uma visão geral sobre todos os componentes base utilizados no desenvolvimento da *framework*, tais como: SOC de comunicação CC2530, a pilha de *software* TIMAC, a plataforma de desenvolvimento *IAR Workbench* para 8051, a técnica de *template metaprogramming* e, no final, a geração de código através de XML e XSLT. Essa visão permitiu dar a conhecer as principais características e funcionalidades desses componentes, e com isso, verificar as suas vantagens e limitações a ter em consideração durante a modelação e implementação da *framework*.

Capítulo 4

C vs C++

Este capítulo apresenta uma comparação entre a linguagem C e a linguagem C++, na qual tenta desmistificar a ideia que o C++ é uma linguagem que afeta o desempenho do programa gerado, em relação à implementação ao equivalente em C. O capítulo expõe as várias funcionalidades do C++, sendo apresentado, em cada uma delas, um exemplo da sua utilização. E ainda, é exibida a forma equivalente de implementar a mesma funcionalidade em linguagem C, sendo por fim comparado o código *assembly* gerado nas duas implementações.

As funcionalidades apresentadas são as seguintes: referências (secção 4.1), classes (secção 4.2), construtores/destrutores (secção 4.3), herança única (secção 4.4), *overload* de operadores (secção 4.5), *templates* (secção 4.6) e funções *inline* (secção 4.7). O C++ possui mais funcionalidades que as enumeradas anteriormente, tais como: bibliotecas STL, RTTI, exceções, funções virtuais e herança múltipla. No entanto, estas afetam significativamente o desempenho do programa gerado e algumas delas, não são suportadas pelo compilador utilizado.

Os exemplos de programas demonstrados durante este capítulo, são compilados através do compilador IAR para 8051, com as otimizações todas desativadas.

4.1 Referências

As referências foram adicionadas ao C++, e permitem de forma mais segura criar variáveis que armazenam endereços de memória. Estas são muito semelhantes aos apontadores, mas enquanto que nos apontadores é possível alterar o valor do endereço de memória a qualquer momento no programa, com as referências só é possível fazê-lo no momento da sua inicialização. A declaração de uma referência é

efetuada através do operador *&*, que diferencia de um apontador que é declarado com o operador ***.

4.1.1 Implementação em C++

Um pequeno exemplo de manipulação de variáveis através de referências está presente no código 4.1.

Código 4.1: Referências em C++

```
void setValue(unsigned char &value){
  value = 10;
}

int main () {
  unsigned char val;
  setValue(val);
  return 0;
}
```

A função *setValue* tem um único parâmetro de entrada, neste caso, uma referência, *&value*, do tipo *unsigned char* (*unsigned char &value*), e a sua utilização atribui o valor 10 à variável, que é passada como parâmetro de entrada.

Na chamada da função, o parâmetro de entrada passado por referência é feito sem a utilização de nenhum operador (*setValue (val)*), estando o compilador encarregue de atribuir ao parâmetro de entrada, o endereço da variável, em vez do próprio valor da variável. Após a execução da função *setValue*, a variável *val* irá armazenar o valor 10.

4.1.2 Implementação em C

A implementação em C de uma referência pode ser efetuada através de apontadores. O código 4.2 apresenta a sua implementação.

Código 4.2: Referências em C

```
void setValue(unsigned char * const value){
  *value = 10;
}

int main () {
  unsigned char val;
    setValue(&val);
    return 0;
}
```

Tal como na implementação em C++, existe uma função *setValue*, que atribui o valor 10 à variável passada como parâmetro de entrada. No entanto, neste caso, a referência do parâmetro de entrada da função é substituída por um apontador constante, **const value*, do tipo *unsigned char* (*unsigned char *const value*). O apontador é constante de modo a limitar que a escrita seja apenas possível na passagem do parâmetro, no momento da chamada da função, tornando-se assim, equivalente à implementação com referências.

Na chamada da função, ao contrário da referência, tem que ser colocado o operador *&* antes da variável, indicando ao compilador que tem que atribuir ao parâmetro de entrada, o endereço de memória de variável e não o seu valor.

4.1.3 Comparações Entre as Duas Implementações

A tabela 4.1, disponibiliza o código *assembly* da chamada da função *setValue* gerado pelo compilador. E a tabela 4.2, disponibiliza o código *assembly* da própria função. O código *assembly* nas duas implementações (C e C++) é apresentado lado a lado de modo a facilitar a sua comparação.

Tabela 4.1: Referências - Código *assembly* gerado na chamada de função

C		C++	
setValue(&val);		*setValue(val);*	
MOV	R2, XSP(L)	MOV	R2, XSP(L)
MOV	R3, XSP(H)	MOV	R3, XSP(H)
LCALL	setValue	LCALL	setValue

Tabela 4.2: Referências - Código *assembly* gerado na função

C		C++	
*void setValue(unsigned char *value){*		*void setValue(unsigned char &value){*	
setValue:		setValue:	
PUSH	DPL	PUSH	DPL
PUSH	DPH	PUSH	DPH
**value = 10;*		**value = 10;*	
MOV	DPL, R2	MOV	DPL, R2

```
    MOV     DPH, R3                 MOV     DPH, R3
    MOV     A, #0x0A                MOV     A, #0x0A
    MOVX    @DPTR, A                MOVX    @DPTR, A
}                                   }
    POP     DPH                     POP     DPH
    POP     DPL                     POP     DPL
    RET                             RET
```

Analisando o código *assembly* gerado pelo compilador, pode-se verificar que é idêntico nas duas implementações (C e C++), tanto na chamada de função como na própria função.

A tabela 4.3 apresenta os valores de consumos de memória das duas implementações.

Tabela 4.3: Referências - Comparação

Memória	**C**	**C++**
CODE(*bytes*)	256	256
DATA(*bytes*)	11	11
XDATA(*bytes*)	3839	3839
IDATA(*bytes*)	64	64
BIT(*bits*)	8	8

Comprova-se assim, que a versão em C++ tem consumos idênticos à versão em C. No entanto, a implementação em C++ possui um código mais percetível e menos suscetível a cometer erros.

4.2 Classes

O C++ apresentou um novo paradigma de programação, o paradigma de programação orientada a objetos. O paradigma de programação orientada a objetos criou um novo conceito, o encapsulamento dos dados. O encapsulamento, fornece a capacidade de limitar o acesso externo a dados que são internos ao próprio objeto, e de permitir a implementação de funções membro (métodos), as quais acedem diretamente aos atributos do objeto através do apontador *this*.

Em C++, a única diferença entre uma classe e uma estrutura, é que na classe, por defeito, os membros (métodos e atributos) são privados, contrariamente aos da estrutura, que por defeito, são públicos.

4.2.1 Implementação em C++

O código 4.3 mostra a criação de uma classe em C++.

Código 4.3: Classes em C++

```
class Rectangle {
    unsigned char width, height;
  public:
    void setValues (unsigned char, unsigned char);
};

void Rectangle::setValues (unsigned char x, unsigned char y){
  this->width = x;
  this->height = y;
}

int main () {
  Rectangle rect;
  rect.setValues(3, 4);

  return 0;
}
```

A classe implementa o objeto *Rectangle* que tem dois atributos do tipo *unsigned char*: a largura (*width*) e a altura (*height*). Os atributos são privados (por defeito, os membros de uma classe são privados), assim não podem ser acedidos externamente de forma direta.

Esta classe possui o método *setValues* que permite alterar os valores dos atributos *width* e *height*. Este método é público pois está definido depois da palavra-chave *public:*. O acesso aos dois atributos do objeto é feito através do apontador *this*, que é um apontador disponibilizado pela própria linguagem C++ e contém o endereço de memória onde está armazenado o objeto. Devido ao encapsulamento, o acesso aos métodos da classe é feito a partir do próprio objeto (*rect.setValues(3, 4)*).

4.2.2 Implementação em C

O C não é uma linguagem orientada a objetos, o elemento do C (*statement*) mais parecido com uma classe é a estrutura de dados (*struct*), mas com a limitação de não permitir definir restrições de acesso às variáveis, bem como, não ser possível criar funções membro. O código 4.4 mostra como criar o conceito de objeto em C.

Código 4.4: Classes em C

```
struct Rectangle {
    unsigned char width, height;
};

void Rectangle_setValues (struct Rectangle *this_, unsigned char x,
    unsigned char y) {
  this_->width = x;
  this_->height = y;
}

int main () {
  struct Rectangle rect;
  Rectangle_setValues(&rect,3,4);

  return 0;
}
```

Nesta implementação, a classe é substituída por uma estrutura de dados (*struct*), que tem os mesmos atributos que a implementação em C++ (*width* e *height*). Contudo, os atributos podem ser acedidos livremente, uma vez que em C não é possível a restrição de acesso às variáveis da estrutura.

Como em C, não é possível criar funções membro, a função *Rectangle_setValues* é criada externamente à própria estrutura, e além de receber como parâmetros de entrada os valores a atribuir às dimensões do retângulo, também recebe como primeiro parâmetro de entrada um apontador para um elemento do tipo da estrutura *Rectangle* (struct Rectangle *this_). Este parâmetro de entrada tem o nome *this_* e implementa o apontador *this*, que em C++ é intrínseco à própria linguagem.

No momento da chamada da função *Rectangle_setValues*, tem que ser fornecido o endereço de memória onde estão armazenadas as dimensões do retângulo (*Rectangle_setValues(&rect, 3, 4)*).

4.2.3 Comparações Entre as Duas Implementações

A tabela 4.4 disponibiliza o código *assemly* da chamada de função, gerado pelo compilador, nas implementações em C e C++.

Tabela 4.4: Classes - Código *assembly* gerado na chamada de função

C	C++
Rectangle_setValues(&rect, 3, 4);	*rect.setValues(3, 4);*
MOV R4, #0x04	MOV R4, #0x04

MOV	R1, #0x03	MOV	R1, #0x03
MOV	R2, XSP(L)	MOV	R2, XSP(L)
MOV	R3, XSP(H)	MOV	R3, XSP(H)
LCALL	Rectangle_setValues	LCALL	setValues

Comparando o código das duas implementações pode-se confirmar que é idêntico. O apontador *this* é guardado nos registos R2 e R3 e os dois parâmetros de entrada nos registos R1 e R4.

A tabela 4.5 apresenta os valores de consumos de memória das duas implementações.

Tabela 4.5: Classes - Comparação

Memória	**C**	**C++**
CODE(*bytes*)	274	274
DATA(*bytes*)	11	11
XDATA(*bytes*)	3839	3839
IDATA(*bytes*)	64	64
BIT(*bits*)	8	8

A análise da tabela comprova que as duas implementações consomem exatamente os mesmos recursos de memória. Contudo, a implementação em C++ é mais percetível que a implementação em C, e além disso, permite restringir o acesso aos membros.

4.3 Construtores e Destrutores

O construtor C++ é uma função membro, que é automaticamente chamada quando é criado um novo objeto, servindo este, por exemplo, para iniciar os atributos do objeto. De forma equivalente, também, é disponibilizado o destrutor, que é uma função membro chamada sempre que o objeto é libertado de memória.

4.3.1 Implementação em C++

O código 4.5 apresenta implementação de um construtor em C++.

Código 4.5: Construtores em C++

```
class Rectangle {
    unsigned char width, height;
  public:
    Rectangle (unsigned char, unsigned char);
};

Rectangle::Rectangle (unsigned char x, unsigned char y) {
  this->width = x;
  this->height = y;
}

int main () {
  Rectangle rect(3, 4);

  return 0;
}
```

A classe implementa o objeto *Rectangle* que tem dois atributos do tipo *unsigned char*: a largura (*width*) e a altura (*height*). Os atributos são privados (por defeito, os membros de uma classe são privados), assim não podem ser acedidos externamente de forma direta.

O construtor da classe é implementado através do método *Rectangle*, o qual tem o mesmo nome da classe e não define dados de retorno. No momento da definição do objeto, o construtor é chamado implicitamente (*Rectangle rect(3 , 4)*).

4.3.2 Implementação em C

O C, como não é uma linguagem orientada a objetos, não tem o conceito de construtor e destrutor. Devido a isso, não existe forma do compilador chamar as funções do construtor ou destrutor, essa chamada tem que ser implementada pelo programador.

O código 4.6 apresenta a forma equivalente de criar um construtor em C.

Código 4.6: Construtores em C

```
struct Rectangle {
    unsigned char width, height;
};

void Rectangle_Constructor (struct Rectangle *this_, unsigned char x,
    unsigned char y) {
  this_->width = x;
  this_->height = y;
}

int main () {
```

```
    struct Rectangle rect;
    Rectangle_Constructor(&rect,3,4);

    return 0;
}
```

O construtor é implementado como uma função normal, que implementa o equivalente a um método em C++ (ler secção 4.2.2), em que o primeiro parâmetro de entrada da função implementa o apontador *this* (*this_*), equivalente ao C++. Ao contrário do C++, em C tem que ser chamado o construtor manualmente (*Rectangle_Constructor(&rect,3,4)*).

4.3.3 Comparações Entre as Duas Implementações

O código *assembly* gerado pelo compilador, na chamada da função construtor nas duas implementações, pode ser visualizado na tabela 4.6.

Tabela 4.6: Construtor - Código *assembly* gerado na chamada de função

C		**C++**	
Rectangle_Constructor(&rect, 3, 4);		*Rectangle rect(3, 4);*	
MOV	R4, #0x04	MOV	R4, #0x04
MOV	R1, #0x03	MOV	R1, #0x03
MOV	R2, XSP(L)	MOV	R2, XSP(L)
MOV	R3, XSP(H)	MOV	R3, XSP(H)
LCALL	Rectangle_Constructor	LCALL	Rectangle

Analisando o código *assembly* gerado, pode-se constatar que é igual tanto na implementação em C como em C++, e idêntico ao código *assembly* de uma chamada de função normal (ver secção 4.5).

A tabela 4.7 apresenta os valores de consumos de memória das duas implementações.

Tabela 4.7: Construtores - Comparação

Memória	C	C++
CODE(*bytes*)	274	274
DATA(*bytes*)	11	11
XDATA(*bytes*)	3839	3839
IDATA(*bytes*)	64	64
BIT(*bits*)	8	8

Os dados da tabela comprovam, mais uma vez, que a implementação em C++ não acresce o consumo de memória em relação à implementação em C. Desta forma, pode-se verificar que a implementação em C++, traz a vantagem de apresentar menor linhas de código e evita que o programador, se possa esquecer de chamar o construtor e o destrutor, pois este é chamado implicitamente pelo compilador C++.

4.4 Herança Única

A herança é um mecanismo introduzido nas linguagens orientadas a objetos que o C++ também adotou, que permite que uma classe herde os métodos e atributos de outra classe (herança única). Se, por exemplo, forem criadas duas classes que contêm alguns dos métodos e atributos iguais, deve ser utilizada herança. Para isso, é implementada uma classe base que contém os métodos e os atributos comuns, e duas classes que herdam da classe base, e implementam os métodos que não são comuns.

4.4.1 Implementação em C++

O código 4.7 apresenta um exemplo de uma implementação com a utilização de herança única em C++.

Código 4.7: Herança em C++

```
class Rectangle {
  unsigned char width, height;
public:
  Rectangle(unsigned char x, unsigned char y): width(x), height(y){}
};

struct Parallelepiped : Rectangle {
  unsigned char depth;
```

```
  Parallelepiped(unsigned char x, unsigned char y, unsigned char z) :
      Rectangle(x, y), depth(z){}
};

int main () {
  Parallelepiped parallelepiped(2, 3, 4);
  return 0;
}
```

Para exemplificar a utilização de herança única, é implementada uma nova classe chamada *Parallelepiped*, que implementa o objeto paralelepípedo (em português). Um paralelepípedo é um objeto tridimensional, constituído pelos atributos altura (*height*), largura (*width*) e profundidade (*depth*). As duas dimensões, altura (*height*) e largura (*width*) são herdadas da classe que implementa o retângulo (*Rectangle*), e a classe *Parallelepiped* apenas implementa o atributo profundidade (*depth*).

4.4.2 Implementação em C

A forma equivalente de implementar o conceito de herança única em C é apresentada no código 4.8.

Código 4.8: Herança em C

```
struct Rectangle {
  unsigned char width, height;
};

void Rectangle_Constructor(struct Rectangle *this_, unsigned char x,
    unsigned char y){
  this_->width = x;
  this_->height = y;
}

struct Parallelepiped{
  struct Rectangle rectangle;
  unsigned char depth;
};

void Parallelepiped_Constructor(struct Parallelepiped *this_,
    unsigned char x, unsigned char y, unsigned char z){
  Rectangle_Constructor((struct Rectangle *)this_, x, y);
  this_->depth = z;
}

int main () {
  Parallelepiped parallelepiped;
  Parallelepiped_Constructor(&parallelepiped, 2, 3, 4);
  return 0;
}
```

Para implementar o equivalente à classe *Parallelepiped*, é implementada uma estrutura de dados com o mesmo nome, que define uma variável do tipo *Rectangle* seguida de uma outra do tipo *unsigned char*, que implementa o atributo profundidade (*depth*). A função que implementa o construtor do paralelepípedo, inicialmente efetua a chamada ao construtor do retângulo e seguidamente, inicia a variável do atributo profundidade (*depth*).

4.4.3 Comparações Entre as Duas Implementações

A tabela 4.8 apresenta o código *assembly* gerado no construtor do objeto *Parallelepiped* para as duas implementações (C e C++).

Tabela 4.8: Herança - Código *assembly* gerado no construtor da classe *Parallelepiped*

C		C++	
void Parallelepiped_Construtor(){		*Parallelepiped(){*	
Parallelepiped_Constructor:		Parallelepiped:	
MOV	A, #0xF6	MOV	A, #0xF6
LCALL	?FUNC_ENTER_XDATA	LCALL	?FUNC_ENTER_XDATA
MOV	A, R2	MOV	A, R2
MOV	R6, A	MOV	R6, A
MOV	A, R3	MOV	A, R3
MOV	R7, A	MOV	R7, A
MOV	V0, R1	MOV	V0, R1
MOV	V1, R4	MOV	V1, R4
MOV	V2, R5	MOV	V3, R5
Rectangle_Constructor(this_, x, y)		*Rectangle(x, y)*	
MOV	R4, V1	MOV	R4, V1
MOV	R1, V0	MOV	R1, V0
MOV	A, R6	MOV	A, R6
MOV	R2, A	MOV	R2, A
MOV	A, R7	MOV	A, R7
MOV	R3, A	MOV	R3, A
LCALL	Rectangle_Constructor	LCALL	Rectangle
this_->dept = z		*dept(z)*	

```
MOV     A, V2                   MOV     A, V2
PUSH    A                       PUSH    A
MOV     DPL, R6                 MOV     DPL, R6
MOV     DPH, R7                 MOV     DPH, R7
INC     DPTR                    INC     DPTR
INC     DPTR                    INC     DPTR
POP     A                       POP     A
MOVX    @DPTR, A                MOVX    @DPTR, A
MOV     R7, #0x03               MOV     A, R6
LJMP    ?FUNC_LEAVE_XDATA       MOV     R2, A
                                MOV     A, R7
                                MOV     R3, A
                                MOV     R7, #0x03
                                LJMP    ?FUNC_LEAVE_XDATA
```

Comparando o código *assembly* das duas implementações, pode-se constatar que é muito idêntico. Apenas existe um pequeno acréscimo de quatro instruções *MOV* (*MOV A, R6; MOV R2, A; MOV A, R7; MOV R3, A*) no epílogo da função da implementação em C++. Por algum motivo, o compilador C++ retorna o apontador *this* no final do construtor da classe *Parallelepiped*, sendo essas instruções extra respeitantes a esse retorno.

O retorno é depois descartado, pelo que essas instruções extra são desnecessárias pelo menos para este exemplo de implementação. Talvez essas instruções sejam retiradas quando se aumenta o nível de otimização do compilador (as otimizações do compilador encontram-se desativadas). A tabela 4.9 apresenta os valores de consumos de memória para as duas implementações.

Tabela 4.9: Herança - Comparação

Memória	**C**	**C++**
CODE(*bytes*)	384	388
DATA(*bytes*)	13	13
XDATA(*bytes*)	3839	3839
IDATA(*bytes*)	64	64
BIT(*bits*)	8	8

Analisando a tabela, pode-se comprovar, que a versão em C++ apenas acresce na memória de código quatro *bytes*. Estes *bytes* extra correspondem às quatro instruções *MOV* que são adicionados no epílogo para retorno do apontador *this*.

Contudo, esse acréscimo, para além de ser inútil para este tipo de implementação, é pouco relevante e será removido quando as otimizações do compilador estiverem ativadas.

A versão em C++ possui menos linhas de código, apresentando, assim, uma implementação mais agradável e percetível que a implementação equivalente em C.

4.5 *Overload* de Operadores

Em C e C++ existem diversos operadores nativos para aplicar ações sobre variáveis (+=, +, *, -, /, etc). O C++ introduziu o *overload* de operadores como forma de possibilitar ao programador implementar as ações efetuadas pelos operadores *standard* da linguagem, sobre objetos de classes criadas pelo próprio programador.

4.5.1 Implementação em C++

O código 4.9 apresenta um exemplo de como é possível implementar o *overload* de um operador.

Código 4.9: *Overload* de Operadores em C++

```
class Rectangle {
    unsigned char width, height;
  public:
    void setValues (unsigned char, unsigned char);
    Rectangle operator +(const Rectangle &a);
};

void Rectangle::setValues (unsigned char x, unsigned char y){
  this->width = x;
  this->height = y;
}

Rectangle Rectangle::operator +(const Rectangle &a){
  Rectangle rect;
  rect.width = this->width + a.width;
  rect.height = this->height + a.height;
  return rect;
}

int main () {
  Rectangle rect_a, rect_b, rect_c;
```

```
    rect_a.setValues(3, 4);
    rect_b.setValues(4, 5);
    rect_c = rect_a + rect_b;
    return 0;
}
```

No código apresentado é possível visualizar o *overload* do operador +, para fazer a soma das dimensões de dois retângulos (*height* e *width*).

A implementação do *overload* de um operador é do género da implementação de um método normal numa classe, apenas é substituído o nome do método pela palavra-chave *operator*, seguida do operador em questão, neste caso, o operador + (*Rectangle Rectangle::operator +(const Rectangle &a)*).

O método criado, recebe como parâmetro de entrada, uma referência para um objeto do tipo *Rectangle*, e retorna outro objeto do mesmo tipo. O valor de retorno será a soma das dimensões do objeto atual, com as dimensões do objeto *Rectangle* recebido como parâmetro de entrada.

4.5.2 Implementação em C

Em C, não existe o conceito de *overload* de operadores. O código 4.10 mostra uma forma equivalente de implementar os operadores em C.

Código 4.10: *Overload* de Operadores em C

```
struct Rectangle {
    unsigned char width, height;
};

void Rectangle_setValues (struct Rectangle *this_, unsigned char x,
    unsigned char y) {
  this_->width = x;
  this_->height = y;
}

Rectangle Rectangle_ADD(struct Rectangle *this_, const Rectangle *a){
  Rectangle rect;
  rect.width = this_->width + a->width;
  rect.height = this_->height + a->height;
  return rect;
}

int main () {
  Rectangle rect_a, rect_b, rect_c;
  Rectangle_setValues(&rect_a, 3, 4);
  Rectangle_setValues(&rect_b, 4, 5);
  rect_c = Rectangle_ADD(&rect_a, &rect_b);
  return 0;
}
```

Em C, a forma de implementar operadores é através de funções normais. É implementada a função *Rectangle_ADD* que recebe como parâmetro de entrada o apontador para duas variáveis do tipo *Rectangle* e retorna uma variável do tipo *Rectangle* com a soma das dimensões dos retângulos, recebidos como parâmetros de entrada.

4.5.3 Comparações Entre as Duas Implementações

A tabela 4.10 apresenta, o código *assembly* gerado na chamada da função que implementa o operador, e a tabela 4.11 mostra o código *assembly* gerado pela própria função.

Tabela 4.10: *Overload* de Operadores - Código *assembly* gerado na chamada de função

C		C++	
rect_c = Rectangle_ADD (&rect_a, &rect_b);		*rect_c = rect_a + rect_b;*	
MOV	R4, #0x04	MOV	R4, #0x04
LCALL	?XSTACK_DISP100_8	LCALL	?XSTACK_DISP100_8
MOV	V0, R0	MOV	V0, R0
MOV	V1, R1	MOV	V1, R1
MOV	R0, #0x08	MOV	R0, #0x08
LCALL	?PUSH_XSTACK_I_TWO	LCALL	?PUSH_XSTACK_I_TWO
MOV	A, #0x02	MOV	A, #0x02
LCALL	?XSTACK_DISP102_8	LCALL	?XSTACK_DISP102_8
MOV	A, #0x04	MOV	A, #0x04
LCALL	?XSTACK_DISP101_8	LCALL	?XSTACK_DISP101_8
LCALL	Rectangle_ADD	LCALL	operator+
MOV	A, #0x02	MOV	A, #0x02
LCALL	?DEALLOC_XSTACK8	LCALL	?DEALLOC_XSTACK8

Tabela 4.11: *Overload* de Operadores - Código *assembly* gerado na função

C	C++

```
Rectangle Rectangle_ADD
(struct Rectangle *this_,
const Rectangle *a)
Rectangle_ADD:
 MOV     A, #0xF8
 LCALL   ?FUNC_ENTER_XDATA
 MOV     A, #0xFE
 LCALL   ?ALLOC_XSTACK8
 MOV     A, R4
 MOV     R0, A
 MOV     A, R5
 MOV     R1, A
 MOV     A, #0x0A
 LCALL   ?XSTACK_DISP0_8
 MOVX    A, @DPTR
 MOV     R4, A
 INC     DPTR
 MOVX    A, @DPTR
 MOV     R5, A
rect.width = this_->width + a->width;
 MOV     DPL, R0
 MOV     DPH, R1
 MOVX    A, @DPTR
 MOV     R6, A
 MOV     DPL, R2
 MOV     DPH, R3
 MOVX    A, @DPTR
 ADD     A, R6
 PUSH    A
 MOV     DPL, XSP(L)
 MOV     DPH, XSP(H)
 POP     A
 MOVX    @DPTR, A
rect.height = this_->height + a->height;
```

```
Rectangle Rectangle::operator
+(const Rectangle &a)
operator+:
 MOV     A, #0xF8
 LCALL   ?FUNC_ENTER_XDATA
 MOV     A, #0xFE
 LCALL   ?ALLOC_XSTACK8
 MOV     A, R4
 MOV     R0, A
 MOV     A, R5
 MOV     R1, A
 MOV     A, #0x0A
 LCALL   ?XSTACK_DISP0_8
 MOVX    A, @DPTR
 MOV     R4, A
 INC     DPTR
 MOVX    A, @DPTR
 MOV     R5, A
rect.width = this->width + a.width;
 MOV     DPL, R0
 MOV     DPH, R1
 MOVX    A, @DPTR
 MOV     R6, A
 MOV     DPL, R2
 MOV     DPH, R3
 MOVX    A, @DPTR
 ADD     A, R6
 PUSH    A
 MOV     DPL, XSP(L)
 MOV     DPH, XSP(H)
 POP     A
 MOVX    @DPTR, A
rect.height = this->height + a.height;
```

Analisando o código *assembly* gerado, pode-se verificar que é idêntico tanto na implementação em C como na implementação C++. A tabela 4.12 apresenta os consumos de memória das duas versões.

Tabela 4.12: *Overload* de Operadores - Comparação

Memória	C	C++
CODE(*bytes*)	602	602
DATA(*bytes*)	12	12
XDATA(*bytes*)	3839	3839
IDATA(*bytes*)	64	64
BIT(*bits*)	8	8

Analisando os valores da tabela, pode-se confirmar, novamente, que as duas implementações consomem exatamente os mesmos recursos de memória, contudo, a versão em C++ apresenta uma implementação mais elegante que a sua equivalente em C.

4.6 *Templates*

As *Templates* foram introduzidas no C++ como forma de estabelecer código genérico, possibilitando evitar a repetição de código, quando se pretende criar classes de ações idênticas, mas em tipos de dados diferentes. O compilador é que se encarrega de gerar código para os diferentes tipos de dados, em vez do programador ter que implementar código para cada um dos tipos de dados.

4.6.1 Implementação em C++

O código 4.11 apresenta uma forma de utilizar a potencialidade das *templates* C++ para tornar o código genérico.

Código 4.11: *Templates* em C++

```
template <typename T>
class Rectangle {
    T width, height;
  public:
    Rectangle (T x, T y){
        this->width = x;
        this->height = y;
```

```
    }
};

int main () {
  Rectangle<unsigned char> rect_uchar(3, 4);
  Rectangle<unsigned int> rect_uint(3, 4);
  return 0;
}
```

Como exemplo da utilização de *templates*, é implementada uma classe *Rectangle*, em que o tipo de dados das dimensões (*height* e *width*) pode ser configurável. A classe recebe como parâmetro de entrada de *template T*, o tipo de dados pretendido para as dimensões do retângulo (*template <typename T>*). No momento da compilação, o compilador substitui em todos sítios onde está presente o parâmetro *template T*, pelo tipo de dados contido.

Neste exemplo, é instanciada a classe *Rectangle* com dois tipos de dados diferentes (*unsigned char* e *unsigned int*). À medida que se instancia a classe com novos tipos de dados, o compilador gera código para todos os métodos da classe, ajustados ao novo tipo de dados.

4.6.2 Implementação em C

O código 4.12 apresenta uma forma de implementar em C o código equivalente à implementação em C++.

Código 4.12: *Templates* em C

```
struct Rectangle_uchar {
    unsigned char width, height;
};

void Rectangle_Constructor_uchar (struct Rectangle_uchar *this_,
    unsigned char x, unsigned char y) {
  this_->width = x;
  this_->height = y;
}

struct Rectangle_uint {
    unsigned int width, height;
};

void Rectangle_Constructor_uint (struct Rectangle_uint *this_,
    unsigned int x, unsigned int y) {
  this_->width = x;
  this_->height = y;
}

int main () {
  struct Rectangle_uchar rect_uchar;
```

```
    struct Rectangle_uint rect_uint;
    Rectangle_Constructor_uchar(&rect_uchar,3,4);
    Rectangle_Constructor_uint(&rect_uint,3,4);

    return 0;
}
```

Para implementar o equivalente em C, tem que ser desenvolvido o código, para cada um dos tipos de dados necessários, neste caso, para o tipo de dados *unsigned char* e *unsigned int*. Sendo isso referente, tanto a nível da estrutura de dados (*void Rectangle_Constructor_uchar* e *void Rectangle_Constructor_uint*), como para o método que desempenha o papel de construtor (*void Rectangle_Constructor_uchar* e *void Rectangle_Constructor_uint*).

4.6.3 Comparações Entre as Duas Implementações

A tabela 4.13, apresenta o código *assembly* gerado pelo compilador na função construtor da classe *Rectangle*, com dimensões do tipo *unsigned char*.

Tabela 4.13: *Templates* - Código *assembly* gerado na função

C		C++	
Rectangle_Constructor_uchar:		Rectangle:	
PUSH	DPL	PUSH	DPL
PUSH	DPH	PUSH	DPH
this_->width = x;		*this->width = x;*	
MOV	A, R1	MOV	A, R1
PUSH	A	PUSH	A
MOV	DPL, R2	MOV	DPL, R2
MOV	DPH, R3	MOV	DPH, R3
POP	A	POP	A
MOVX	@DPTR, A	MOVX	@DPTR, A
this_->height = y;		*this->height = y;*	
MOV	A, R4	MOV	A, R4
PUSH	A	PUSH	A
MOV	DPL, R2	MOV	DPL, R2
MOV	DPH, R3	MOV	DPH, R3
INC	DPTR	INC	DPTR

```
POP     A                    POP     A
MOVX    @DPTR, A             MOVX    @DPTR, A
POP     DPH                  POP     DPH
POP     DPL                  POP     DPL
RET                          RET
```

Analisando o código *assembly*, verifica-se que o mesmo é idêntico para as duas implementações, isto, porque as *templates* são resolvidas ao nível do *front-end* do compilador, não afetando o desempenho do código produzido.

A tabela 4.14 apresenta os valores dos consumos de memória provocados pelas duas implementações.

Tabela 4.14: *Templates* - Comparação

Memória	**C**	**C++**
CODE(*bytes*)	481	481
DATA(*bytes*)	12	12
XDATA(*bytes*)	3839	3839
IDATA(*bytes*)	64	64
BIT(*bits*)	8	8

A análise da tabela comprova, novamente, que os *templates* não afetam o desempenho do código gerado.

A implementação em C++ com *templates*, apresenta um número de linhas de código muito menor que a implementação em C e possibilita aumentar a facilidade da manutenção de código, uma vez que este é genérico para vários tipos de dados.

4.7 Funções *Inline*

As funções *inline* foram adicionadas ao C++ para evitar o *overhead* provocado pelas chamadas de funções (*calls*). Assim, o código da função é injetado diretamente no local onde é efetuada a chamada à função.

Em C, esse problema é resolvido através da utilização de *macros* do pré-processador, no entanto, essa forma de programação é pouco segura e de difícil depuração de erros. Enquanto, as *macros* são resolvidas pelo pré-processador, as funções *inline* são

resolvidas pelo próprio compilador. Neste caso, em função das configurações de otimização, o compilador avalia se faz a injeção de código ou a trata como uma função normal.

4.7.1 Implementação em C++

O código 4.13 apresenta uma forma de implementar uma função *inline* em C++.

Código 4.13: Funções *Inline* em C++

```
inline unsigned char sum(unsigned char a, unsigned char b) {
  return a + b;
}

int main () {
  return sum(3,4);
}
```

No exemplo de código é implementada uma função *inline sum*, que recebe como parâmetro de entrada duas variáveis do tipo *unsigned char*, e retorna a soma das mesmas. A implementação de uma função *inline* é muito idêntica a uma função normal, apenas é precedida pela palavra-chave *inline*, antes do tipo de dados de retorno (*inline unsigned char sum*). A chamada de uma função *inline* é idêntica a uma função normal (*sum(3,4)*).

4.7.2 Implementação em C

O código 4.14 apresenta a forma equivalente de implementar uma função *inline* em linguagem C.

Código 4.14: Funções *Inline* em C

```
#define sum(a, b) ((a) + (b))

int main () {
  return sum(3,4);
}
```

Tal como foi referido anteriormente, a forma de implementar uma função *inline* em C é através de *macros* de pré-processamento. Para isso, é definida uma *macro* com parâmetros, que recebe as duas variáveis a somar (*#define sum(a, b) ((a) + (b))*). Para evitar problemas com precedência de operadores, são adicionados parênteses extra durante o cálculo da soma, uma vez que o pré-processador, apenas se limita a

substituir texto contido depois do *define*, em todas as partes do código onde é utilizada a *macro*. A chamada da *macro* é idêntica à implementação em C++ (*sum(3,4)*).

4.7.3 Comparações Entre as Duas Implementações

A tabela 4.15 apresenta o código *assembly* gerado pelo compilador, para as duas implementações (C e C++).

Tabela 4.15: Funções *Inline* - Código *assembly* gerado

C		C++	
return SUM(3, 4)		return sum(3, 4)	
main:		main:	
MOV	R2, #0x07	MOV	R2, #0x07
MOV	R3, #0x00	MOV	R3, #0x00

Analisando a tabela, pode-se verificar, que o código é idêntico para as duas implementações. O compilador não gera nenhuma instrução *call*, e como é a soma de duas constantes (3 + 4 = 0x07), atribui diretamente aos registos de retorno do programa (R2 e R3), o valor da soma.

A tabela 4.16 apresenta os valores de consumos de memória das duas implementações.

Tabela 4.16: Funções *Inline* - Comparação

Memória	C	C++
CODE(*bytes*)	196	196
DATA(*bytes*)	11	11
XDATA(*bytes*)	3839	3839
IDATA(*bytes*)	64	64
BIT(*bits*)	8	8

A análise dos valores da tabela vem comprovar, novamente, que a implementação em C++ não acrescenta nenhum *overhead* em relação à implementação em C. O que prova que devem ser utilizadas funções *inline* em alternativa a *macros* de pré-processamento, isto porque, as funções *inline* apresentam uma implementação

mais segura, de fácil depuração de erros e mais elegante, que a implementação com *macros*.

4.8 Conclusões

Neste capítulo, foram apresentados testes de comparação, entre algumas das funcionalidades da linguagem C++, com a implementação equivalente em linguagem C. Sendo o C++ uma linguagem de programação orientada a objetos, o programador beneficia das vantagens deste novo paradigma de programação, como o encapsulamento de dados, a facilidade de esconder partes da implementação ao exterior dos objetos, a herança que permite estruturar melhor os dados e o polimorfismo, neste caso o polimorfismo paramétrico (*templates*). As vantagens da utilização de C++ são evidentes e permitem obter uma implementação mais elegante, com um menor número de linhas de código, o que facilita a análise e manutenção do código implementado. Além disso, é uma linguagem menos suscetível a erros, porque o compilador de C++ faz uma verificação de tipo de dados mais intensiva que o compilador de C.

Foram apresentadas as funcionalidades que não afetam o desempenho do programa em relação à implementação equivalente em linguagem C, tais como: referências, classes, construtores/destrutores, herança única, *overload* de operadores, *templates* e funções *inline*. Contudo, o C++ possui mais funcionalidades para além das descritas, tais como, bibliotecas STL, RTTI, exceções, funções virtuais e herança múltipla. No entanto, estas são funcionalidades que afetam significativamente o desempenho do programa gerado, e não são suportadas pelo compilador IAR para 8051.

Os resultados dos testes de comparação entre as duas linguagens, vieram comprovar que na linguagem C++, e ao contrário do que muitos pensam, muitas das funcionalidades não introduzem qualquer acréscimo no desempenho do programa e nos consumos de memória do código binário gerado, em relação à implementação equivalente em linguagem C. Desta forma, torna-se uma linguagem de programação viável, para o desenvolvimento de código a ser executado em sistemas embebidos de poucos recursos, tal como o sistema embebido apresentado nesta dissertação.

Capítulo 5

Implementação do Sistema

Este capítulo descreve como foi modelada e implementada a *framework* generativa. Primeiramente (secção 5.1), é descrito como foi exportada a biblioteca *Boost.MPL*, de forma a ser compatível com o compilador IAR para 8051. A biblioteca *Boost.MPL* foi utilizada como base na implementação da *framework* generativa. Seguidamente (secção 5.2), é apresentada a forma como foi modelada e implementada a camada HAL (*Hardware Abstraction Layer*). E ainda, quais os módulos da camada HAL que foram convertidos para C++ e como foram aplicadas as técnicas de *template metaprogramming* para gerir a variabilidade de código da mesma. No final (secção 5.3), é apresentada a modelação e a implementação da *framework* generativa, no que diz respeito à gestão da variabilidade de código da camada OSAL (*Operating System Abstraction Layer*).

5.1 Exportação da Biblioteca *Boost.MPL*

A biblioteca *Boost.MPL* foi desenvolvida com intuito de oferecer compatibilidade a uma ampla gama de compiladores, ou seja, foi desenvolvida de forma a não serem necessários ajustes ao código da mesma por parte do programador, para que compile e execute corretamente. No entanto, nem todos os compiladores concedem total compatibilidade com a biblioteca. A tabela 5.1 apresenta o grau de compatibilidade de uma lista de compiladores.

Tabela 5.1: Compatibilidade da biblioteca *Boost.MPL* [10]

Compilador	Versões	Categoria
Borland C++	5.6.4	B
Borland C++	5.6.1	C
Comeau C/C++	4.2.45, 4.3.3	A
Compaq C++ (Tru64 UNIX)	6.5	A
GCC	3.2.2, 3.3.1, 3.4	A
GCC	2.95.3	B
HP aCC	3.55	C
Intel C++	7.1, 8.0	A
Metrowerks CodeWarrior	8.3, 9.2	A
Microsoft Visual C++	7.1	A
Microsoft Visual C++	6.0 sp5, 7.0	B
SGI MIPSpro	7.3	B
Sun CC	5.6	C

Os programadores da biblioteca *Boost.MPL* dividiram o grau de compatibilidade do código da biblioteca em três categorias: categoria A, categoria B e categoria C. Os compiladores pertencentes à categoria A, apresentam um grau de compatibilidade elevado com a biblioteca, isto é, permitem compilar e executar o código da mesma sem a necessidade de alterações por parte do utilizador. Os compiladores referentes à categoria B, também dão suporte à biblioteca, contudo, têm que ser efetuadas algumas alterações por parte do utilizador, para que o código desta compile e execute corretamente. Por último, os compiladores pertencentes à categoria C, não suportam a maior parte das funcionalidades da biblioteca, por isso, não é aconselhada a sua utilização.

Tal como é possível visualizar na tabela 5.1, o compilador utilizado nesta dissertação, o IAR para o 8051, não está presente. Isto deve-se ao facto de não terem sido realizados testes de compatibilidade do código da biblioteca com o compilador IAR para 8051, por parte dos programadores da biblioteca. Por isso, não é garantida a compatibilidade da biblioteca *Boost.MPL* com o compilador IAR para 8051.

Depois de realizados testes de compatibilidade da biblioteca no compilador IAR para 8051, verificou-se que este não oferecia compatibilidade de forma direta com a biblioteca, ou seja, não era possível a utilização da biblioteca fazendo somente a in-

clusão da mesma num projeto. Isto porque, muito do código pertencente à biblioteca é gerado antes da sua compilação através do pré-processador, e essa geração não é compatível com o pré-processador do compilador IAR. Este problema levou à necessidade de obter uma solução para tornar a biblioteca compatível com o compilador utilizado.

A solução encontrada consistiu em efetuar a exportação dos ficheiros gerados por um pré-processador de um compilador compatível e, de seguida, testar se depois dessa exportação eram compatíveis com o compilador IAR. Essa exportação realizou-se através do compilador *Microsoft Visual Studio C++* [45], uma vez que este oferece total compatibilidade com a biblioteca *Boost.MPL*, tal como foi apresentado na tabela 5.1.

O processo de exportação da biblioteca é um trabalho moroso, visto que esta disponibiliza muitas funcionalidades e, consequentemente, muitos ficheiros associados a cada uma dessas funcionalidades. Assim, apenas foram exportadas as funcionalidades necessárias para a utilização no projeto. A exportação foi executada de forma progressiva, sendo exportadas novas funcionalidades à medida que eram necessárias.

As funcionalidades exportadas foram divididas em cinco grupos: Estruturas de Condição, Operadores, Tipos de Dados, Contentores e Iteradores, que se encontram descritos nas subsecções abaixo.

5.1.1 Estruturas de Condição

O grupo Estruturas de Condição (*Conditionals*, em inglês) apresenta meta-funções que permitem implementar condições seletivas como, por exemplo, condições *IF*. A tabela 5.2 apresenta as meta-funções que foram extraídas da biblioteca *Boost.MPL* para este grupo.

Tabela 5.2: MPL - Estruturas de Condição

Nome	Meta-função	Descrição
eval_if.hpp	eval_if	Avalia uma das duas *nullary-metafunction* (condição com tipos MPL)
	eval_if_c	Avalia uma das duas *nullary-metafunction* (condição com tipos C *runtime*)
if.hpp	if_	If (condição com tipos MPL)
	if_c	If (condição com tipos C *runtime*)

is_same.hpp	is_same	Compara dois tipos de dados

5.1.2 Operadores

O grupo Operadores (*Operators*, em inglês) disponibiliza meta-funções que permitem efetuar operações sobre dados de forma estática, ou seja, ao nível da compilação. A tabela 5.3 apresenta a lista de operadores disponibilizados na biblioteca *Boost.MPL* que foram exportados.

Tabela 5.3: MPL - Operadores

Nome	Meta-função	Descrição
and.hpp	and_	E lógico (&&)
equal_to.hpp	equal_to	Igual (==)
greater_equal.hpp	greater_equal	Maior e igual (>=)
greater.hpp	greater	Maior (>)
less_equal.hpp	less_equal	Menor e igual (<=)
less.hpp	less_	Menor (<)
not_equal_to.hpp	not_equal_to	Diferente (!=)
not.hpp	not_	Negação lógica (!)
or.hpp	or_	Ou lógico (||)
shift_left.hpp	shift_left	Deslocar à esquerda («)
shift_right.hpp	shift_right	Deslocar à direita (»)

5.1.3 Tipos de Dados

O grupo Tipos de Dados (*Types*, em inglês) apresenta meta-funções que representam tipos de dados estáticos utilizados na biblioteca *Boost.MPL*. Os tipos de dados que foram exportados estão presentes na tabela 5.4.

Tabela 5.4: MPL - Tipos de Dados

Nome	Meta-função	Descrição
bool.hpp	bool_	Booleano MPL
char.hpp	char_	*Char* MPL
empty_base.hpp	empty_base	Classe vazia
int.hpp	int_	*Int* MPL
integral_c.hpp	integral_c	Tipo constante genérico
integral_constant.hpp	integral_constant	Constante do tipo MPL
long.hpp	long_	*Long* MPL
pair.hpp	pair	Par de dois tipos
size_t.hpp	size_t_	*Size_t* MPL
void.hpp	void_	*Void* MPL

5.1.4 Contentores

O grupo Contentores (*Containers*, em inglês) disponibiliza meta-funções que permitem criar sequências de tipos de dados. Embora a biblioteca *Boost.MPL* disponibilize outros tipos de contentores, neste projeto em específico, apenas foi necessário exportar o contentor *Vector*, tal como é possível visualizar na tabela 5.5.

Tabela 5.5: MPL - Contentores

Nome	Meta-função	Descrição
vector.hpp	vector	Vetor MPL

5.1.5 Iteradores

O grupo Iteradores (*Iterators*, em inglês) disponibiliza meta-funções que permitem efetuar operações sobre os contentores, tais como, procura de elementos, acesso aos elementos e formas de percorrer os elementos. As meta-funções que foram exportadas estão listadas na tabela 5.6.

Tabela 5.6: MPL - Iteradores

Nome	Meta-função	Descrição
advance.hpp	advance	Avançar elementos no contentor
at.hpp	at	Aceder ao elemento de um índice do contentor
begin_end.hpp	begin	Elemento inicial de um contentor
	end	Elemento final de um contentor
contains.hpp	contains	Verificar se um contentor contém o elemento
deref.hpp	deref	Desreferenciação de um iterador
find.hpp	find	Procurar a primeira ocorrência de um elemento T num contentor
find_if.hpp	find_if	Procurar a primeira ocorrência de um elemento T num contentor que satisfaça uma dada condição.
for_each.hpp	for_each	Percorrer elementos do contentor
next_prior.hpp	next	Elemento seguinte do contentor
	prior	Elemento anterior do contentor
size.hpp	size_	Tamanho de um contentor MPL (vetor, lista, etc)

5.2 Implementação da Camada HAL

A camada HAL (*Hardware Abstraction Layer*) é fornecida pela TIMAC da *Texas Instruments*, e tem a função de criar uma camada de abstração de acesso ao *hardware* presente no microcontrolador e na placa de desenvolvimento. Esta camada fornece um conjunto de bibliotecas em linguagem C que facilitam o acesso ao *hardware*.

A HAL implementada pela *Texas Instruments* fornece bibliotecas de abstração para todo o *hardware* presente no microcontrolador e para a placa de desenvolvimento, no entanto, nesta dissertação, para demonstração do conceito, apenas foi feita a transformação de algumas bibliotecas para *template metaprogramming*, tais como:

- **Registos do SFR:** (secção 5.2.1), que implementa um *wrapper* de acesso aos registos do SFR;

- **Portos de Entrada/Saída:** (secção 5.2.2), que implementa o módulo de gestão dos portos de Entrada/Saída;

- **LEDs:** (secção 5.2.3), que implementa o módulo de gestão dos LEDs presentes na placa de desenvolvimento;

- **Interrupções:** (secção 5.2.4), que permite gerir as interrupções do microcontrolador;

- **ADC:** (secção 5.2.5), que implementa o módulo de gestão do ADC que o microcontrolador possui;

- **DMA:** (secção 5.2.6), que implementa o módulo de gestão do DMA;

- **Portas Série:** (secção 5.2.7), que implementa o módulo de gestão das portas série.

Por último, também foi implementado em TMP o módulo de configuração e inicialização de todos os módulos listados anteriormente (secção 5.2.8). As próximas secções descrevem a forma como foram modelados e implementados cada um dos módulos descritos anteriormente.

5.2.1 Registos do SFR

Implementação do acesso aos registos do SFR:

***Framework* da *Texas Instruments*:**

O compilador IAR para 8051 fornece bibliotecas de acesso aos registos do SFR do micro 8051. Esse acesso é feito através da utilização de um tipo de dados especial, iniciado pela palavra-chave *___sfr*, que indica que está mapeada no espaço de memória reservado aos registos do SFR. Também é indicado que a memória não deve ser inicializada a zero pelo compilador, através da palavra-chave *___no_init*. O endereço de memória onde é mapeado o registo, é efetuado no final da definição e precedido do caractere @.

O código 5.1, define a forma como o compilador IAR implementa o acesso aos registos do SFR.

Código 5.1: Estruturas especiais que implementam o mapeamento dos registos do SFR

```
#define SFR(name,addr) \
__sfr __no_init volatile unsigned char name @ addr;

#define SFRBIT(name, addr, bit7, bit6, bit5, bit4, bit3, bit2, bit1,
    bit0) \
__sfr __no_init volatile union \
{                              \
  unsigned char name;          \
  struct {                     \
    unsigned char bit0 : 1;    \
    unsigned char bit1 : 1;    \
    unsigned char bit2 : 1;    \
    unsigned char bit3 : 1;    \
    unsigned char bit4 : 1;    \
    unsigned char bit5 : 1;    \
    unsigned char bit6 : 1;    \
    unsigned char bit7 : 1;    \
  };                           \
} @ addr;
```

Analisando o código, pode-se constatar que são implementadas duas macros do pré-processador com parâmetros: (i) a macro *SFR* que implementa o acesso aos registos endereçáveis ao *byte*, e recebe como parâmetros o nome do registo (*name*) e o endereço do registo (*addr*); e (ii) a macro *SFRBIT* que implementa o acesso aos registos endereçáveis ao *bit*, e recebe como parâmetros o nome do registo do *byte* (*name*), o endereço do registo (*addr*), e seguido do nome dos oito *bits* (*bit7* a *bit0*) pertencentes ao registo.

As macros foram criadas de modo a evitar a repetição de código na implementação do acesso a todos os registos do SFR. O código 5.2 apresenta como são instanciadas as macros para implementar o acesso aos registos.

Código 5.2: Mapeamento dos registos do SFR

```
/* Port 0 */
SFRBIT( P0, 0x80, P0_7, P0_6, P0_5, P0_4, P0_3, P0_2, P0_1, P0_0 )
SFR( SP,    0x81  )   /*  Stack Pointer */
SFR( DPL0,  0x82  )   /*  Data Pointer 0 Low Byte */
SFR( DPH0,  0x83  )   /*  Data Pointer 0 High Byte */
SFR( DPL1,  0x84  )   /*  Data Pointer 1 Low Byte */
SFR( DPH1,  0x85  )   /*  Data Pointer 1 High Byte */
SFR( U0CSR, 0x86  )   /*  USART 0 Control and Status */
SFR( PCON,  0x87  )   /*  Power Mode Control */
```

No microcontrolador 8051, os registos do SFR estão mapeados na memória RAM interna, nas posições de memória que vão de *0x80* a *0xFF*. Apenas os registos

que estão mapeados nas posições de memória terminadas em *0x0* e *0x8*, é que são endereçáveis ao *bit*. Então, o acesso a esses registos (endereçáveis ao *bit*) é implementado através da instanciação da macro *SFRBIT* e os outros (não endereçáveis ao *bit*) pela macro *SFR*.

***Framework* Generativa:**

O mapeamento dos registos do SFR que a IAR implementou não é reconhecido como *typenames*, logo, é difícil a sua utilização direta em *template metaprogramming*. Esse problema levou à necessidade de criar uma forma de tornar o acesso aos registos do SFR através de elementos do tipo *typename*. A figura 5.1 mostra o diagrama de classes de como foi implementado esse acesso.

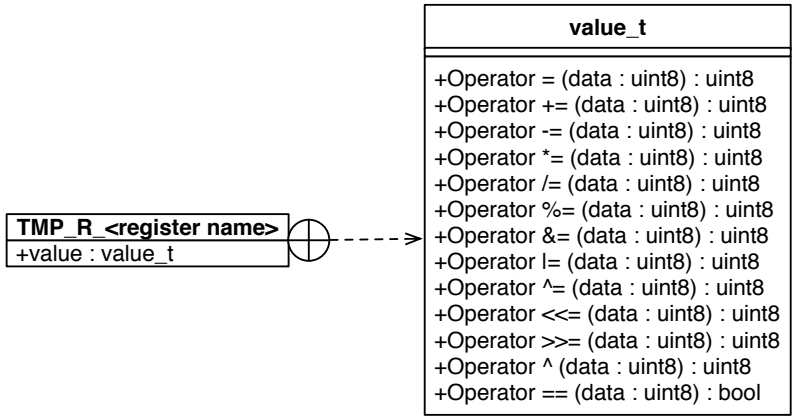


Figura 5.1: Diagrama de classes - Estrutura que implementa o *wrapper* aos registos do SFR

O acesso foi obtido através de uma classe que implementa um *wrapper* aos registos. A classe tem o nome *TMP_R_<register_name>*, em que o *<register_name>* é substituído pelo nome de cada registo do SFR. A classe declara a variável estática *value* com o tipo de dados *value_t*, em que esse tipo de dados é definido numa estrutura de dados privada e interna à própria classe. A estrutura de dados *value_t* implementa o *overload* dos operadores, de forma a ser possível manipular o registo do SFR através dos operadores normais da linguagem C++. O código 5.3, apresenta a implementação dos *wrappers* de acesso aos registos do SFR.

Código 5.3: Estrutura que implementa o *wrapper* aos registos do SFR

```
#define TMP_SFR_GEN(REG) \
```

```
class TMP_R_ ## REG \
{ \
 protected: \
    struct value_t { \
        TMP_GEN_OPERATORS_ALL(REG) \
        inline uint8 operator()(){ \
          return REG; \
        } \
    }; \
public: \
    static value_t value; \
};

#define TMP_SFRBIT_GEN(REG, BIT7, BIT6, BIT5, BIT4, BIT3, BIT2, BIT1,
    BIT0) \
class TMP_R_ ## REG { \
protected: \
  struct value_t { \
    TMP_GEN_OPERATORS_ALL(REG) \
    inline uint8 operator()(){ \
      return REG; \
    } \
  }; \
public: \
  static value_t value; \
}; \
TMP_SFR_GEN(BIT7) \
TMP_SFR_GEN(BIT6) \
TMP_SFR_GEN(BIT5) \
TMP_SFR_GEN(BIT4) \
TMP_SFR_GEN(BIT3) \
TMP_SFR_GEN(BIT2) \
TMP_SFR_GEN(BIT1) \
TMP_SFR_GEN(BIT0)
```

Para evitar repetição de código, foram utilizadas as potencialidades do pré-processador para gerar código. Ao analisar o código, pode-se verificar que são implementadas duas macros com parâmetros: (i) a macro denominada por *TMP_SFR_GEN*, que implementa o *wrapper* dos registos endereçáveis ao *byte*, e recebe como parâmetro o nome do registo do SFR (*REG*); e (ii) a macro denominada por *TMP_SFRBIT_GEN*, que implementa o *wrapper* dos registos endereçáveis ao *bit*, e recebe como parâmetro o nome do registo do SFR, seguido dos nomes dos oito *bits* pertencentes a esse registo (*BIT0* a *BIT7*).

O código 5.4, apresenta como são instanciadas as macros para implementar os *wrappers* de acesso aos registos.

Código 5.4: Implementação dos *wrappers* dos registos do SFR

```
/* Port 0 */
TMP_SFRBIT_GEN( P0, P0_7, P0_6, P0_5, P0_4, P0_3, P0_2, P0_1, P0_0 )
TMP_SFR_GEN(SP)          /* Stack Pointer */
```

```
TMP_SFR_GEN(DPL0)       /*  Data Pointer 0 Low Byte */
TMP_SFR_GEN(DPH0)       /*  Data Pointer 0 High Byte */
TMP_SFR_GEN(DPL1)       /*  Data Pointer 1 Low Byte */
TMP_SFR_GEN(DPH1)       /*  Data Pointer 1 High Byte */
TMP_SFR_GEN(U0CSR)      /*  USART 0 Control and Status */
TMP_SFR_GEN(PCON)       /*  Power Mode Control */
```

A instanciação das macros que criam os *wrappers* aos registos do SFR, é muito idêntica à instanciação das macros criadas pelo compilador IAR para acesso aos registos do SFR, tal como apresentado anteriormente. Por cada registo é chamada a macro correspondente, a *TMP_SFR_GEN* no caso do registo ser endereçável ao *byte* ou a *TMP_SFRBIT_GEN* se for endereçável ao *bit*.

Acesso aos registos do SFR:

***Framework* da *Texas Instruments*:**

O código 5.5 apresenta um exemplo de como é efetuado o acesso aos registos do SFR com a *Framework* da *Texas Instruments*.

Código 5.5: Acesso aos registos do SFR - *Framework* da *Texas Instruments*

```
TCON = 0;
```

Tal como se pode verificar, o acesso é efetuado de forma direta, como se fosse uma variável normal. Neste exemplo, é apresentado como é atribuído o valor zero (0) ao registo *TCON*, sendo essa escrita efetuada através do operador de atribuição (=).

***Framework* Generativa:**

O código 5.6 apresenta um exemplo de como é efetuado o acesso aos registos na *Framework* Generativa.

Código 5.6: Acesso aos registos do SFR - *Framework* Generativa

```
TMP_R_TCON::value = 0;
```

O acesso é feito de forma muito idêntica à *Framework* da *Texas Instruments*. O nome do registo difere apenas de ser precedido pela palavra-chave *TMP_R_*. E a atribuição, em vez de ser feita diretamente sobre o nome, é efetuada à variável interna estática *value*, da seguinte forma: *TMP_R_TCON::value = 0*.

5.2.2 Portos de Entrada/Saída

Para aceder aos portos de entrada/saída do microcontrolador, foi desenvolvido um módulo com o nome *Chal_IO*. Este módulo permite ao utilizador efetuar operações nos portos e nos pinos correspondentes de cada porto. As operações que são possíveis realizar, são: operações de leitura e escrita nos portos e nos pinos, definir se os pinos são de entrada ou de saída, selecionar se o pino é de uso geral ou um pino ligado a um periférico e configurar se os pinos têm resistência interna de *pull-up* ou *pull-down*.

Diagrama de funcionalidades:

A figura 5.2 apresenta o diagrama de funcionalidades do módulo *Chal_IO*.

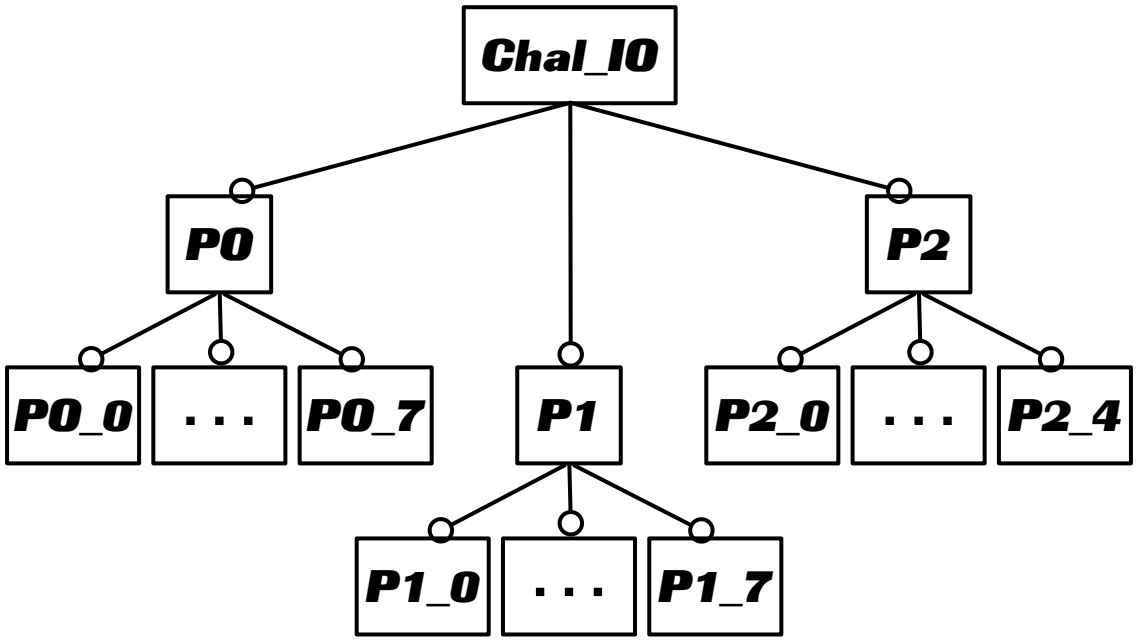


Figura 5.2: Diagrama de funcionalidades - Módulo de gestão dos portos de entrada/saída

Analisando o diagrama de funcionalidades, verifica-se que o microcontrolador dispõe de três portos de entrada e saída (*P0*, *P1* e *P2*). Cada um dos portos possui oito pinos (*P<n>_0* a *P<n>_7*, em que *<n>* representa o número do porto), exceto o porto dois (*P2*) que, apenas, contém cinco pinos (*P2_0* a *P2_4*).

Implementação do acesso aos pinos de entrada/saída:

A figura 5.3 apresenta o diagrama de classes de como foi implementado o acesso de leitura e escrita nos pinos.

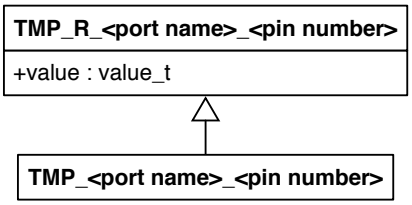


Figura 5.3: Diagrama de classes - Acesso aos pinos de entrada/saída

O diagrama de classes é genérico, e representa a implementação do acesso a um único pino, em que o *<port name>* é substituído pelo nome do porto a que o pino pertence, e o *<pin number>* é substituído pelo número do pino a implementar.

Observando o diagrama de classes, pode-se constatar que o acesso aos pinos é realizado através de uma classe com o nome *TMP_<port name>_<pin number>*. Como os pinos estão mapeados nos registos do SFR, a classe *TMP_<port name>_<pin number>* apenas herda da classe que implementa o acesso ao registo SFR do porto correspondente. Ou seja, herda da classe *TMP_R_<port name>_<pin number>* que foi apresentada na secção 5.2.1.

O código 5.7 demonstra como foi implementado o acesso aos pinos, respeitando a modelação efetuada no diagrama de classes anterior.

Código 5.7: Implementação do acesso aos pinos de entrada/saída

```
#define TMP_PINS_GEN(PORT, PIN) \
class TMP_ ## PORT ## _ ## PIN : public TMP_R_ ## PORT ## _ ## PIN \
{ \
};
```

Para evitar a repetição de código, recorreu-se à implementação da macro com parâmetros, *TMP_PINS_GEN*. A macro recebe como parâmetros: o nome do porto (*PORT*) e o número do pino (*PIN*). A macro implementa a classe *TMP_<port name>_<pin number>* (*TMP_ ## PORT ## _ ## PIN*) que herda da classe *TMP_R_<port name>_<pin number>* (*TMP_R_ ## PORT ## _ ## PIN*). Para todos os pinos é efetuada a chamada da macro *TMP_PINS_GEN*.

Implementação do acesso aos portos de entrada/saída:

A figura 5.4 apresenta o diagrama de classes da implementação do acesso aos portos de entrada/saída.

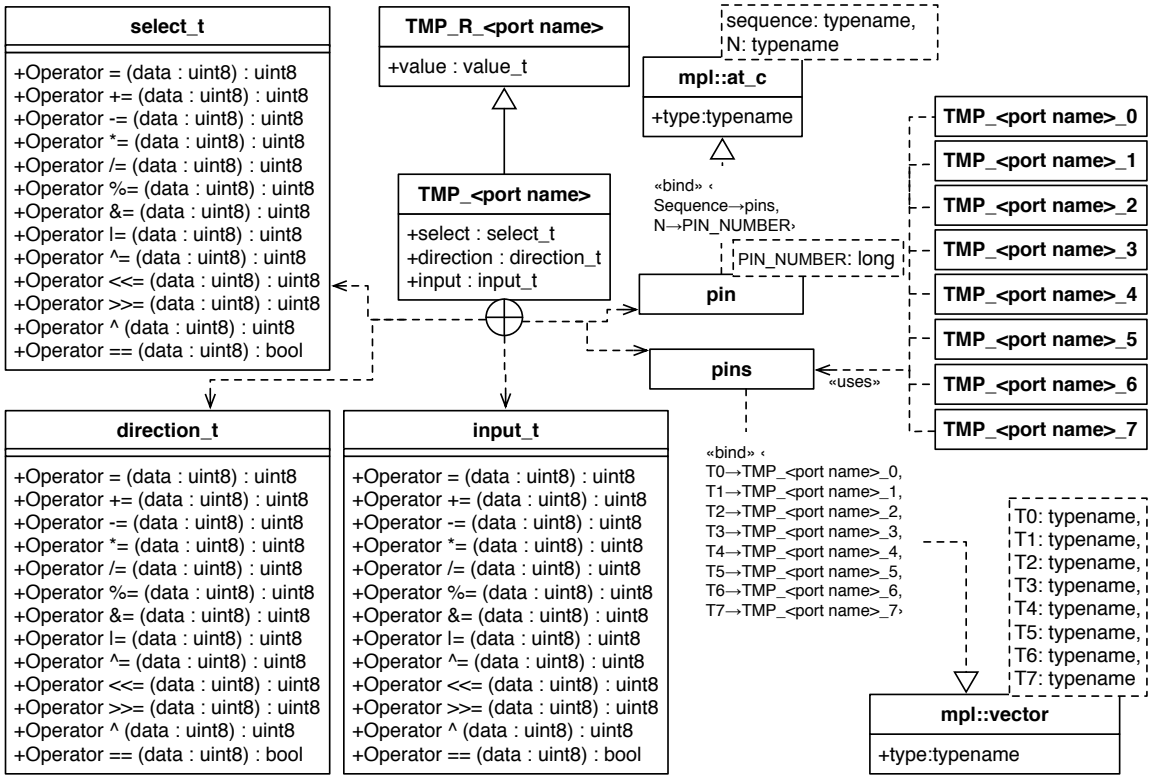


Figura 5.4: Diagrama de classes - Acesso aos portos de entrada/saída

Este diagrama de classes também é genérico, e representa a implementação do acesso a um único porto, em que o *<port name>* é substituído pelo nome do porto a implementar.

O acesso aos portos de entrada/saída é implementado pela classe *TMP_<port name>*, a qual herda da classe *TMP_R_<port name>*, que implementa o acesso ao registo do SFR correspondente ao porto. Esta classe contém três atributos públicos e estáticos: *select*, *direction* e *input*. O atributo *select* permite selecionar se os pinos pertencentes ao porto funcionam em acesso geral ou acesso por periférico. O atributo *direction* possibilita selecionar se os pinos do porto são de entrada ou de saída. E por último, o atributo *input* permite selecionar se os pinos do porto têm uma resistência interna de *pull-up* ou *pull-down*. Cada um desses atributos é do tipo de uma estrutura de dados interna que implementa o *overload* dos operadores do C++,

para manipulação dos registos SFR associados aos atributos (*select_t* para o *select*, *direction_t* para o *direction* e *input_t* para o *input*).

A classe *TMP_<port name>*, também, contém internamente a declaração de um vetor MPL estático, com o nome *pins*. Este vetor é constituído por os tipos de dados que dão acesso aos oito pinos do porto (*TMP_<port name>_0* a *TMP_<port name>_7*). Para aceder a cada pino de um porto, através do número do pino, é implementada internamente uma meta-função de nome *pin*, que recebe como parâmetro de entrada de *template* o número do pino a aceder (*PIN_NUMBER: long*). Esta meta-função herda da meta-função *at_c* do MPL.

O código 5.8 apresenta como foi implementado o acesso aos portos de entrada/-saída.

Código 5.8: Implementação do acesso aos portos de entrada/saída

```
#define TMP_PORTS_CLASS_GEN(PORT, ...) \
class TMP_ ## PORT : public TMP_R_ ## PORT { \
  typedef mpl::vector<__VA_ARGS__> pins; \
    \
  struct direction_t { \
    TMP_GEN_OPERATORS_ALL(PORT ## DIR) \
    inline uint8 operator()(){ \
      return PORT ## DIR; \
    } \
  }; \
  struct input_t { \
    TMP_GEN_OPERATORS_ALL(PORT ## INP) \
    inline uint8 operator()(){ \
      return PORT ## INP; \
    } \
  }; \
  struct select_t { \
    TMP_GEN_OPERATORS_ALL(PORT ## SEL) \
    inline uint8 operator()(){ \
      return PORT ## SEL; \
    } \
  }; \
public: \
  static direction_t direction; \
  static input_t input; \
  static select_t select; \
  template <long PIN_NUMBER> \
  struct pin : mpl::at_c<pins, PIN_NUMBER>::type{}; \
};
```

Analisando o código, verifica-se que é utilizada uma macro, com parâmetros, para gerar de forma genérica o acesso aos portos, evitando a repetição de código. A macro tem o nome *TMP_PORTS_CLASS_GEN*, e recebe como parâmetros, o nome do porto (*PORT*), seguida da lista de pinos pertencentes ao porto em questão

(lista passada como parâmetro através de uma *variadic macro*).

A macro implementa a classe *TMP_<port name>* (*class TMP_ ## PORT*) que herda da classe *TMP_R_<port name>* (class TMP_R_ ## PORT). Internamente à classe, e de forma privada, é declarado o vetor MPL *pins* que contém a lista de pinos pertencentes ao porto. Também de forma privada, são implementadas as estruturas que implementam o *overload* dos operadores, para possibilitar a manipulação dos registos de configuração dos portos (*direction_t*, *input_t* e *select_t*).

De forma pública, são declaradas as três variáveis estáticas que dão acesso às configurações dos registos que dizem respeito ao porto (*direction*, *input* e *select*). Por fim, também de forma pública, é implementada a meta-função *pin* que dá acesso aos pinos pertencentes ao porto, através do seu número (*PIN_NUMBER*), a qual herda da meta-função *mpl::at_c*.

Implementação do módulo que engloba o acesso a todos os portos e pinos de entrada/saída:

A figura 5.5 apresenta a classe *TMP_PORTS*, que engloba o acesso a todos os pinos e portos de entrada/saída.

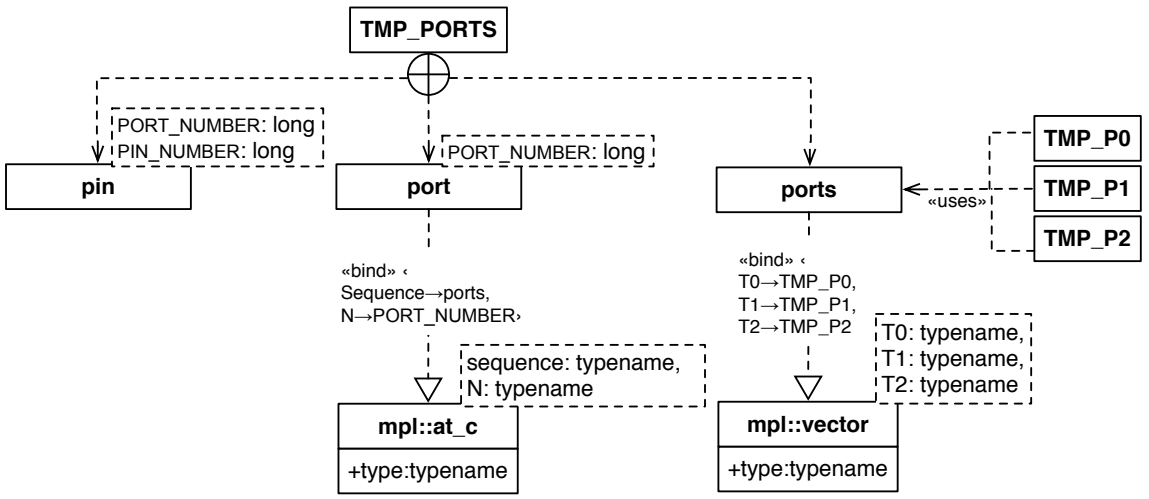


Figura 5.5: Diagrama de classes - Acesso aos portos e pinos de entrada/saída

A classe *TMP_PORTS* define internamente um vetor MPL com o nome *ports*, que contém as estruturas de acesso a todos os portos da placa (*TMP_P0* a *TMP_P2*). Além disso, esta classe, possui também internamente a meta-função *pin* que dá acesso

aos pinos do porto, quando fornecido como parâmetro de entrada de *template* o número do porto (*PORT_NUMBER*) e o número do pino (*PIN_NUMBER*) a aceder.

O código 5.9 apresenta a implementação da classe *TMP_PORTS*.

Código 5.9: Implementação do acesso aos portos e pinos de entrada/saída

```
class TMP_PORTS {
  typedef mpl::vector<TMP_P0, TMP_P1, TMP_P2> ports;

public:
  template <long PORT_NUMBER>
  struct port:mpl::at_c<ports, PORT_NUMBER>::type{};

  template <long PORT_NUMBER, long PIN_NUMBER>
  struct pin:port<PORT_NUMBER>::template pin<PIN_NUMBER>{};
};
```

A classe *TMP_PORTS* é composta por um vetor MPL, com o nome *ports* e de acesso privado, com todas as estruturas de acesso aos portos da placa (*TMP_P0*, *TMP_P1* e *TMP_P2*). Também define duas meta-funções: a meta-função *port*, que dá acesso aos portos e a meta-função *pin*, que dá acesso aos pinos do microcontrolador.

Acesso aos portos e pinos de entrada/saída:

***Framework* da *Texas Instruments*:**

O código 5.10 mostra como é possível efetuar o acesso aos portos através da *framework* da *Texas Instruments*.

Código 5.10: Acesso aos portos e pinos de entrada/saída - *Framework* da *Texas Instruments*

```
P0=0;
P0DIR = 0;
P0SEL = 0;

P0_1=0;
```

O acesso aos portos, pinos e registos de configuração dos portos, é efetuado de forma direta, tal como se tratasse de uma variável normal.

***Framework* Generativa:**

O código 5.11 apresenta como é feito o acesso aos portos e pinos através da *Framework* Generativa.

Código 5.11: Acesso aos portos e pinos de entrada/saída - *Framework* Generativa

```
TMP_PORTS::port<0>::value=0;
TMP_PORTS::port<0>::direction=0;
TMP_PORTS::port<0>::select=0;

TMP_PORTS::port<0>::pin<1>::value=0;
TMP_PORTS::pin<0,1>::value=0;
```

O acesso é efetuado através da instanciação da classe *TMP_PORTS*. Caso se pretenda aceder a um porto, tem que ser chamada a meta-função *port*, e passado como parâmetro de entrada de *template* o número do porto a aceder (*TMP_PORTS::port<0>*). Após aceder ao porto, pode ser alterado o seu valor (*TMP_PORTS::port<0>::value*), ou configurar o seu funcionamento, através das variáveis *direction*, *select* e *input*.

O acesso aos pinos é efetuado, de forma idêntica, através da instanciação da classe *TMP_PORTS*, o qual pode ser feito de duas formas: (i) através da utilização da meta-função *pin*, passando como parâmetros de entrada de *template* o número do porto e o número do pino a aceder (*TMP_PORTS::pin<0,1>::value*); ou (ii) acedendo primeiro ao porto através da meta-função *port* e, posteriormente, acedendo à meta-função *pin* que cada porto possui para aceder aos seus pinos (*TMP_PORTS::port<0>::pin<1>::value*).

5.2.3 LEDs

Para aceder aos LEDs da placa de desenvolvimento, foi implementado um módulo com o nome *Chal_led*. Este módulo, permite configurar e ligar/desligar todos os LEDs da placa de desenvolvimento.

Diagrama de funcionalidades:

A figura 5.6 apresenta o diagrama de funcionalidades do módulo que implementa a gestão dos LEDs da placa.

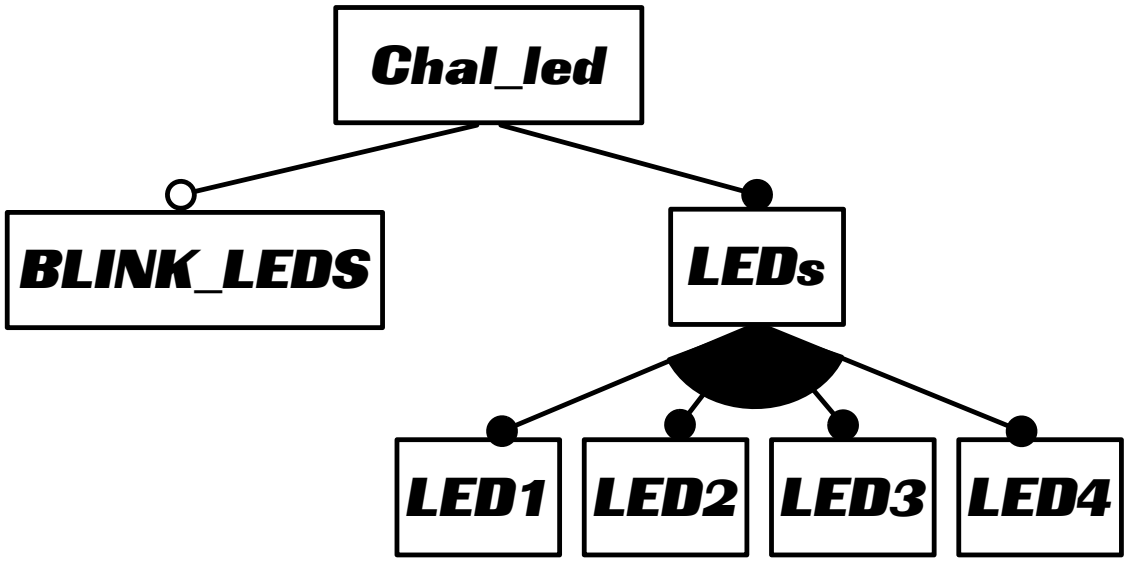


Figura 5.6: Diagrama de funcionalidades - Módulo de gestão dos LEDs

Analisando o diagrama de funcionalidades, pode-se verificar que se for selecionado o módulo de gestão dos LEDs, tem que ser ativado no mínimo um LED, no entanto, estão disponíveis quatro LEDs que podem ser ativados. Existe, também, a funcionalidade *BLINK_LEDS* que é opcional, e pode ser ativada ou não, dependendo das necessidades da aplicação. A ativação dessa funcionalidade possibilita configurar os LEDs de forma a piscarem com uma frequência de valor estabelecido pelo utilizador.

Implementação do mapeamento dos LEDs

A figura 5.7 exibe o diagrama de classes de como foi implementado o mapeamento dos LEDs nos portos do microcontrolador.

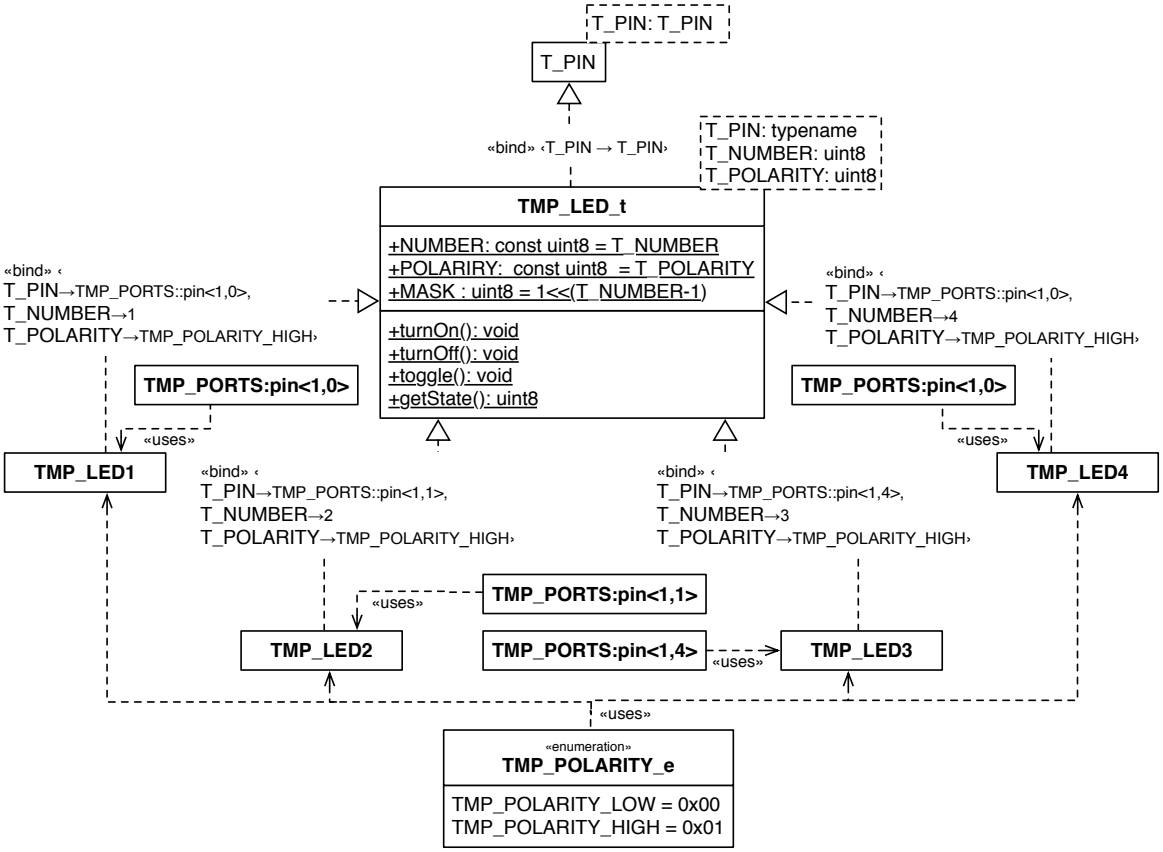


Figura 5.7: Diagrama de classes - Mapeamento dos LEDs

A classe *TMP_LED_t* implementa a estrutura de manipulação de um LED. A classe recebe como parâmetros de entrada de *template*, a estrutura de dados do pino ao qual o LED está ligado (*T_PIN*), o número do LED (*T_NUMBER*) e, por último, a polaridade de ativação do LED (*T_POLARITY*). Como cada um dos LEDs está ligado a um pino, a classe *TMP_LED_t* herda todos os membros do pino, passado como parâmetro de entrada de *template PIN*. A classe *TMP_LED_t*, também, dispõe de alguns métodos de manipulação do LED: o método que permite ligar o LED (*turnOn*), o método que possibilita desligar o LED (*turnOff*), o método que permite alternar o estado do LED (*toggle*) e o método que permite obter o estado do LED (*getState*).

As estruturas de dados de acesso aos LEDs são implementadas com o recurso à classe *template TMP_LED_t*. Por cada LED, é instanciada a classe com os parâmetros de entrada de *template*, devidamente configurados para esse LED. Por exemplo, para criar a estrutura de dados de acesso ao *LED1*, é instanciada a classe *template*

TMP_LED_t com o pino *P1_0* no parâmetro *PIN* (*TMP_PORTS::pin<1,0>*), o valor *1* no parâmetro *T_NUMBER* e, por último, o valor *TMP_POLARITY_HIGH* no parâmetro *T_POLARITY*.

O código 5.12 apresenta a implementação do diagrama de classes anterior.

Código 5.12: Mapeamento dos LEDs

```
template <typename T_PIN, uint8 T_NUMBER, uint8 T_POLARITY>
struct TMP_LED_t : T_PIN
{
  static const uint8 NUMBER = T_NUMBER;
  static const uint8 POLARIRY = T_POLARITY;
  static const uint8 MASK = mpl::shift_left< mpl::integral_c< uint8,
     1 >, mpl::integral_c< uint8, T_NUMBER - 1 > >::value;

  static inline void turnOn();
  static inline void turnOff();
  static inline void toggle();
  static inline uint8 getState();
};

typedef TMP_LED_t<TMP_PORTS::pin<1,0>,1,TMP_POLARITY_HIGH> TMP_LED1;
typedef TMP_LED_t<TMP_PORTS::pin<1,1>,2,TMP_POLARITY_HIGH> TMP_LED2;
typedef TMP_LED_t<TMP_PORTS::pin<1,4>,3,TMP_POLARITY_HIGH> TMP_LED3;
typedef TMP_LED_t<TMP_PORTS::pin<1,0>,4,TMP_POLARITY_HIGH> TMP_LED4;
```

Examinando o código, verifica-se que a classe *TMP_LED_t* herda da classe passada no parâmetro de entrada de *template T_PIN*. Isto devido ao facto de um LED estar mapeado diretamente num único pino de entrada/saída. A classe *TMP_LED_t* contém três atributos estáticos e constantes: o *NUMBER* que guarda o número do LED contido no parâmetro de entrada de *template T_NUMBER*; o *POLARITY* que guarda o tipo de polaridade do LED, contido no parâmetro de entrada de *template T_POLARITY*; e o atributo *MASK* que contém uma máscara de *bits*, para acesso ao pino que o LED está ligado no porto.

Tal como foi demonstrado no diagrama de classes, a classe *template TMP_LED_t* também possui quatro métodos de manipulação do estado do LED (*turnOn*, *turnOff*, *toggle*, *getState*).

Neste código também é demonstrado o mapeamento dos quatro LEDs (*TMP_LED1*, *TMP_LED2*, *TMP_LED3* e *TMP_LED4*), recorrendo à instanciação da classe *template TMP_LED_t*.

Implementação da configuração do módulo de gestão dos LEDs

A figura 5.8 apresenta o diagrama de classes de como foi implementada a configuração do módulo de gestão dos LEDs.

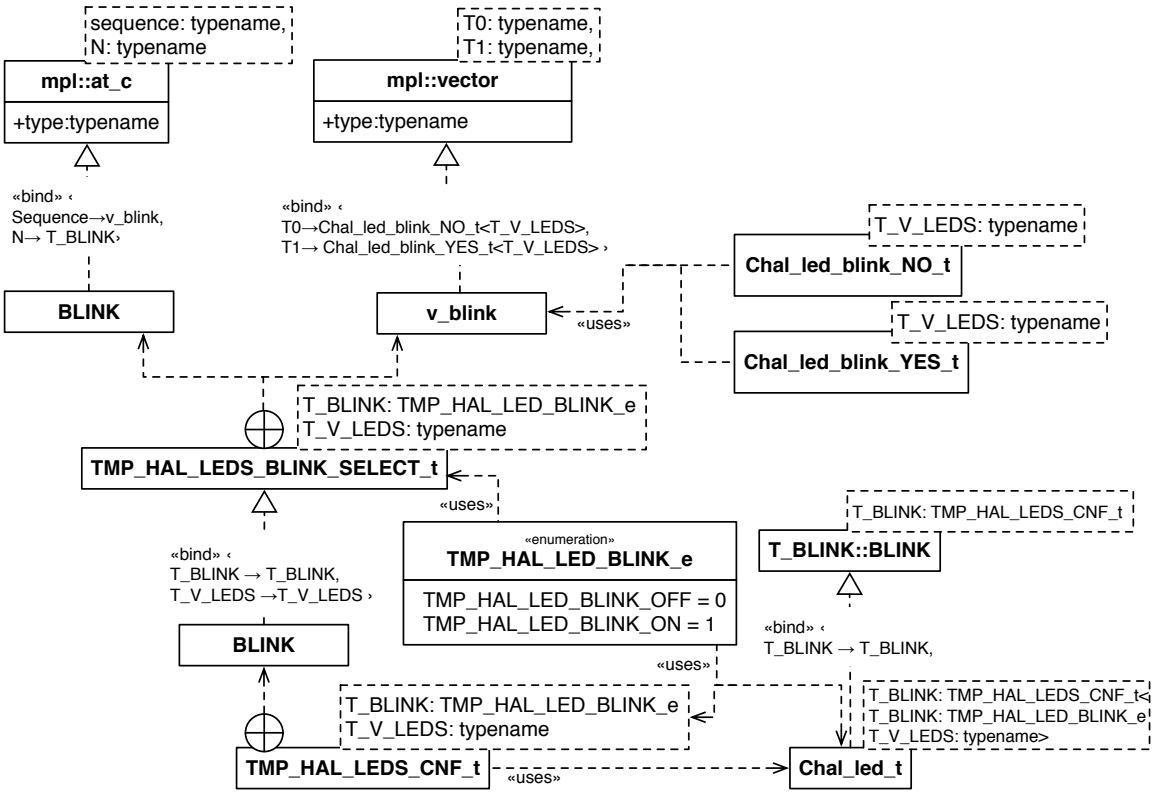


Figura 5.8: Diagrama de classes - Configuração do módulo de gestão dos LEDs

O módulo de manipulação dos LEDs é implementado pela classe *template* *Chal_led_t*, a qual recebe como parâmetro de entrada de *template* a estrutura *TMP_HAL_LEDS_CNF_t*. Esta é uma estrutura de configuração, que permite ao utilizador definir quais os LEDs que pretende utilizar, atribuindo ao parâmetro de entrada de *template T_V_LEDS*, um vetor MPL com a lista de LEDs. Também na mesma estrutura (*TMP_HAL_LEDS_CNF_t*) é possível ativar/desativar o modo *Blink*, através do parâmetro de entrada de *template T_BLINK*.

A seleção do modo *Blink* é efetuada através da implementação da meta-função *TMP_HAL_LEDS_BLINK_SELECT*. Esta meta-função recorre às meta-funções *at_c* e *vector* do MPL para implementar essa escolha.

A classe *template Chal_led_t* herda da classe *Chal_led_blink_NO_t* ou da

classe *Chal_led_blink_YES_t*, dependendo se o modo *Blink* foi definido como inativo ou ativo, respetivamente.

O código 5.13 é referente à implementação do diagrama de classes anterior.

Código 5.13: Configuração do módulo de gestão dos LEDs

```
template <TMP_HAL_LED_BLINK_e T_BLINK, typename T_V_LEDS>
class TMP_HAL_LED_BLINK_SELECT_t {
  static_assert(T_BLINK <= TMP_HAL_LED_BLINK_ON, "Invalid Blink Mode"
      );
  typedef mpl::vector< Chal_led_blink_NO_t< T_V_LEDS >,
                       Chal_led_blink_YES_t< T_V_LEDS > > v_blink;
public:
  struct BLINK : public mpl::at_c<v_blink, static_cast<long>(T_BLINK)
      >::type{};
};

template <TMP_HAL_LED_BLINK_e T_BLINK, typename T_V_LEDS>
struct TMP_HAL_LEDS_CNF_t
{
  struct BLINK : public TMP_HAL_LED_BLINK_SELECT_t< T_BLINK, T_V_LEDS
      >::BLINK {};
};

template <typename T_BLINK>
class Chal_led_t : public T_BLINK::BLINK {};
```

A meta-função *TMP_HAL_LED_BLINK_SELECT_t* implementa a seleção do modo *Blink*. Esta possui internamente, e de forma privada, um vetor MPL com o nome *v_blink*, que contém dois elementos: a classe que implementa o módulo de gestão dos LEDs com a opção *Blink* inativa (*Chal_led_blink_NO_t*) e a classe com a opção *Blink* ativa (*Chal_led_blink_YES_t*). A classe selecionada pode ser acedida através do tipo de dados *BLINK*, que herda os membros da classe selecionada, através da utilização da meta-função *mpl::at_c* do MPL.

Se a meta-função *TMP_HAL_LED_BLINK_SELECT_t* for instanciada com uma configuração inválida, passada no parâmetro de entrada de *template T_BLINK* (configuração que não faz parte da enumeração *TMP_HAL_LED_BLINK_e*), é apresentado o erro de compilação "*Invalid Blink Mode*". Este erro de compilação é mostrado ao utilizador através da função *static_assert()*.

Implementação do módulo de gestão dos LEDs

O diagrama de classes apresentado na figura 5.9, diz respeito à implementação do módulo de gestão dos LEDs.

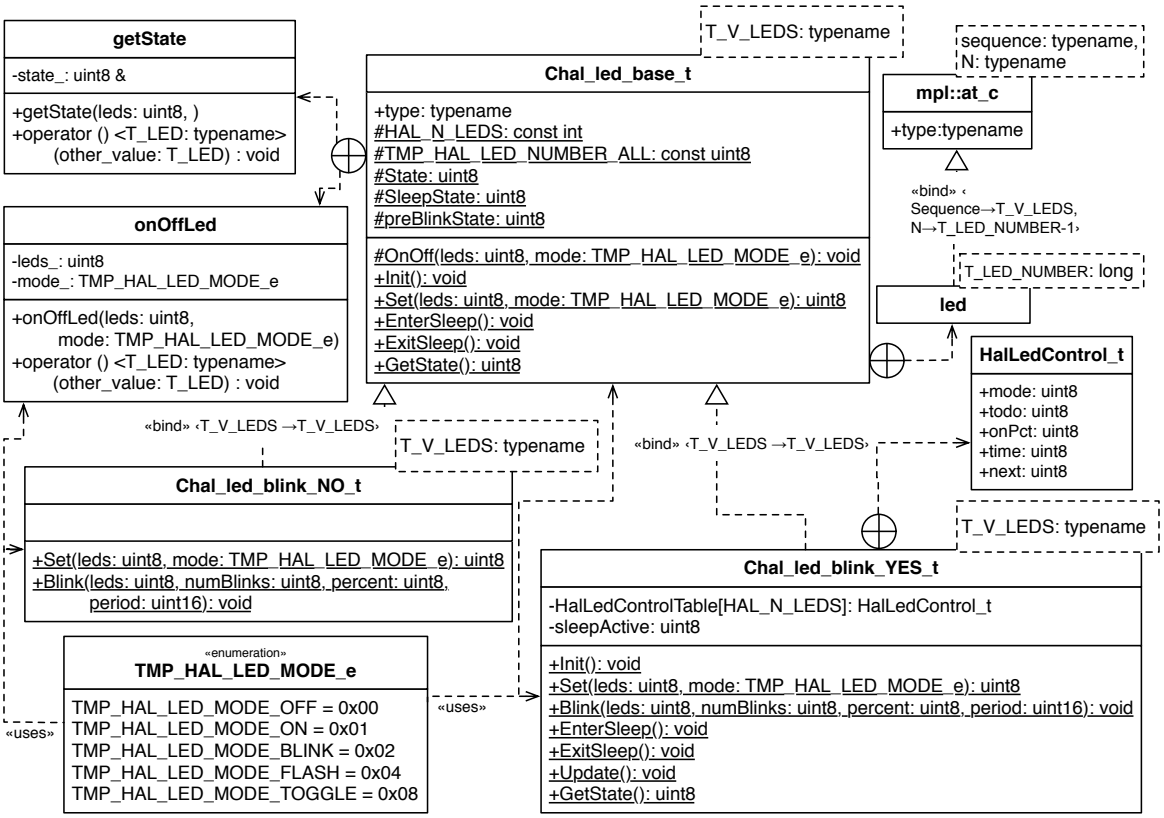


Figura 5.9: Diagrama de classes - Módulo de gestão dos LEDs

No diagrama de classes é possível visualizar a classe *template Chal_led_base_t*. Esta classe *template* implementa tudo o que é genérico, ou seja, tudo o que não depende se o modo *Blink* está ativo ou inativo. Tudo o que depende do modo *Blink* é implementado em classes que herdam da classe *Chal_led_base_t*. Assim, o modo *Blink* inativo é implementado com a classe *Chal_led_blink_NO_t* e o modo *Blink* ativo é implementado com a classe *Chal_led_blink_YES_t*.

A classe *template Chal_led_base_t* define internamente a meta-função *led*, que através da meta-função *at_c* do MPL, fornece o acesso ao LED correspondente ao índice passado como parâmetro de entrada de *template T_LED_NUMBER*. A meta-função *at_c* acede ao vetor de LEDs passado como parâmetro de entrada de *template* da classe *T_V_LEDS*.

A estrutura *getState* implementa um *functor* utilizado para aceder ao estado dos LEDs. O *functor* é passado como parâmetro de entrada a um *for_each* MPL, que percorre todos os elementos do vetor dos LEDs (*T_V_LEDS*), chamando o *functor*

para cada um deles.

A estrutura *onOffLed* implementa um *functor* utilizado para ativar ou desativar os LEDs. Tal como acontece na estrutura *getState*, esse *functor* é passado como parâmetro de entrada a um *for_each* MPL, que percorre todos os elementos do vetor dos LEDs (*T_V_LEDS*), chamando o *functor* para cada um deles. O código 5.14 apresenta a implementação desse *functor*.

Código 5.14: *Functor* de ativação/inativação dos LEDs

```
struct onOffLed{
    onOffLed(uint8 leds, TMP_HAL_LED_MODE_e mode) { leds_ = leds;
        mode_ = mode;}

  template <typename T_LED>
    inline void operator()(T_LED other_value) const{
      if (leds_ & mpl::shift_left< mpl::integral_c< uint8, 1 >,  mpl
          ::integral_c< uint8, T_LED::NUMBER - 1 > >::value){
           if (mode_ == TMP_HAL_LED_BLINK_ON){
             T_LED::turnOn();
           }else{
             T_LED::turnOff();
           }
         }
     }
     private:
       uint8 leds_;
       TMP_HAL_LED_MODE_e mode_;
  };

static void OnOff (uint8 leds, TMP_HAL_LED_MODE_e mode){
  mpl::for_each<T_V_LEDS>(onOffLed(leds, mode));
...
}
```

O *functor* é implementado pela estrutura de dados *onOffLed*, em que o construtor dessa estrutura recebe como parâmetros de entrada, a máscara de escolha dos LEDs (*leds*) e o tipo de ação sobre o *led* (*mode*). Essa ação pode ser: desligar (*TMP_HAL_LED_BLINK_OFF*) ou ligar o LED (*TMP_HAL_LED_BLINK_ON*).

Além disso, é implementado o *overload* do operador (*()*) que recebe como parâmetro de entrada a estrutura de acesso ao LED (*T_LED*), o qual é atribuído de forma automática pelo *for_each* do MPL. A função, que implementa o operador, verifica se a máscara de *bits* passada como parâmetro de entrada (*led*), tem ativo o *bit* correspondente ao LED atual (*T_LED*). Em caso afirmativo, chama a função de mudança de estado do LED, ou seja, chama a função *T_LED::turnOn()*, caso o *mode* esteja definido como *TMP_HAL_LED_BLINK_ON* ou a função *T_LED::turnOff()*, caso contrário.

A função *OnOff* é um método da classe *template Chal_led_base_t*. Esta função permite ligar/desligar LEDs e recebe, também, como parâmetros de entrada, a máscara de escolha dos LEDs (*leds*) e o tipo de ação sobre o *led* (*mode*). Esta função utiliza o *for_each* do MPL para gerar para cada LED, contido no vetor MPL *T_V_LEDS*, a verificação se o *bit* incluído na máscara *leds* é referente ao LED atual, e em função disso desliga/liga o LED dependendo do que foi configurado na variável *mode*. O *for_each* recebe, como parâmetro de entrada de *template*, o vetor MPL de LEDs *T_V_LEDS* e como parâmetro de entrada de função o *functor onOffLed*.

A utilização do *for_each* evita a repetição de código para implementar a função *OnOff* em cada um dos LEDs. Esta permite de forma estática, gerar o código de comparação para cada um dos LEDs. O código 5.15 apresenta como ficaria o código se não fosse utilizado o *for_each*.

Código 5.15: Função *OnOff* sem a utilização do *for_each*

```
static void OnOff (uint8 leds, TMP_HAL_LED_MODE_e mode){
 if (leds & 1<<0){
    if (mode == TMP_HAL_LED_BLINK_ON){
      TMP_LED1::turnOn();
    }else{
      TMP_LED1::turnOff();
    }
  }
  if (leds & 1<<1){
    if (mode == TMP_HAL_LED_BLINK_ON){
      TMP_LED2::turnOn();
    }else{
      TMP_LED2::turnOff();
    }
  }
  if (leds & 1<<2){
    if (mode == TMP_HAL_LED_BLINK_ON){
      TMP_LED3::turnOn();
    }else{
      TMP_LED3::turnOff();
    }
  }
  if (leds & 1<<3){
    if (mode == TMP_HAL_LED_BLINK_ON){
      TMP_LED4::turnOn();
    }else{
      TMP_LED4::turnOff();
    }
  }
 ...
}
```

Analisando o código, pode-se comprovar que a não utilização do *for_each* obriga à repetição de código. Para cada LED, tem que ser implementada uma condição de

comparação do número do LED com a máscara de *bits* (*leds & 1«0*) e implementada a chamada às funções de ligar/desligar o LED.

A utilização do *for_each*, para além da vantagem de evitar a repetição de código, permite tornar o número de LEDs escalável. Ou seja, gera apenas código para os LEDs contidos no vetor MPL *T_V_LEDS*.

Configuração do módulo de gestão dos LEDs

Framework da *Texas Instruments*:

No código 5.16 é apresentado como é configurado o módulo de gestão dos LEDs com a *Framework* da *Texas Instruments*.

Código 5.16: Configuração do módulo de gestão dos LEDs - *Framework* da *Texas Instruments*

```
/* Set to TRUE enable LED usage, FALSE disable it */
#ifndef HAL_LED
#define HAL_LED TRUE
#endif
#if (!defined BLINK_LEDS) && (HAL_LED == TRUE)
#define BLINK_LEDS
#endif
```

A ativação do módulo de gestão dos LEDs é feita através da definição da macro de pré-processamento *HAL_LED*. A ativação do modo *Blink* é efetuada pela definição da macro de pré-processamento *BLINK_LEDS*.

Framework Generativa:

O código 5.17 apresenta como é configurado o módulo de gestão dos LEDs com a *Framework* Generativa.

Código 5.17: Configuração do módulo de gestão dos LEDs - *Framework* Generativa

```
typedef mpl::vector< TMP_LED1, TMP_LED2, TMP_LED3, TMP_LED4 >
    V_HAL_LEDS;

typedef TMP_HAL_LEDS_CNF_t< TMP_HAL_LED_BLINK_ON, V_HAL_LEDS >
    LEDS_CNF;

typedef Chal_led_t< LEDS_CNF > Chal_led;
```

Para configurar o módulo de gestão dos LEDs, tem que ser definido um vetor MPL com os LEDs que se pretendem utilizar (*V_HAL_LEDS*). Na *Framework* da *Texas Instruments* não é possível definir essa escolha, sendo gerado o código para todos os LEDs da placa, caso estes sejam utilizados ou não.

De seguida, é definida a estrutura de configuração *TMP_HAL_LEDS_CNF_t* com o nome *LEDS_CNF* (utilizando o *typedef*), em que é fornecido como primeiro parâmetro de entrada de *template* a escolha do modo *Blink*. Neste caso, foi definido como ativo recorrendo à enumeração *TMP_HAL_LED_BLINK_ON*. E como segundo parâmetro de entrada de *template* da estrutura é fornecido o vetor *V_HAL_LEDS* criado anteriormente.

Para finalizar, é configurada a classe *template Chal_led_t*, definindo como parâmetro de entrada de *template* a estrutura de configuração definida anteriormente (*LEDS_CNF*). Desta configuração, e recorrendo a um *typedef*, é obtida a classe de manipulação dos LEDs da placa, denominada por *Chal_led*.

5.2.4 Interrupções

Para manipular as interrupções do microcontrolador, é disponibilizado um módulo com o nome *Chal_interrupt*. O módulo permite ativar as interrupções e atribuir código às rotinas de serviço à interrupção das mesmas.

Diagrama de funcionalidades:

A figura 5.10 apresenta o diagrama de funcionalidades do módulo de gestão das interrupções.

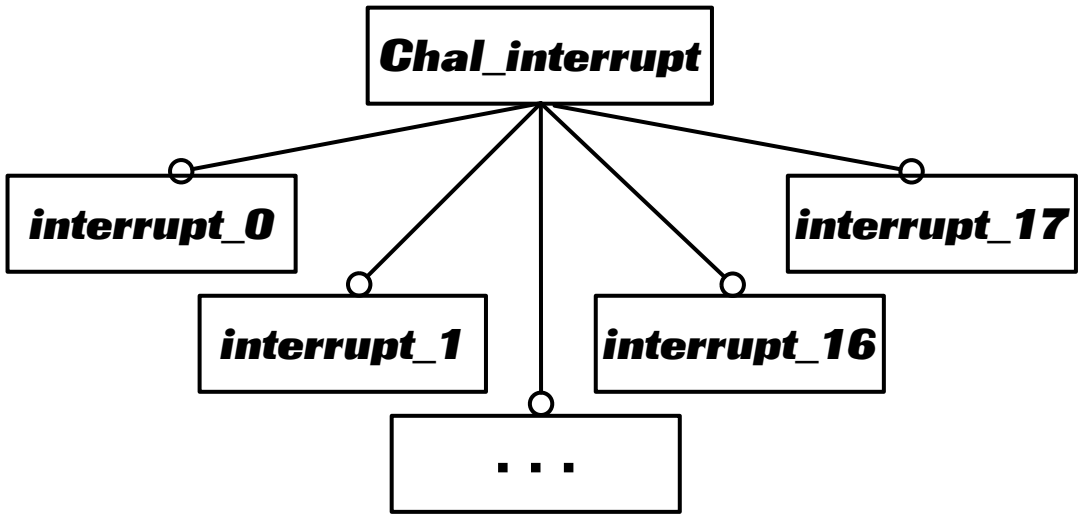


Figura 5.10: Diagrama de funcionalidades - Módulo de gestão das interrupções

Analisando o diagrama de funcionalidades, pode-se verificar que o microcontrolador dispõe de dezoito interrupções que podem ser utilizadas pelo utilizador de

forma opcional.

Implementação do módulo de gestão das interrupções

A figura 5.11 ilustra o diagrama de classes do módulo de gestão das interrupções.

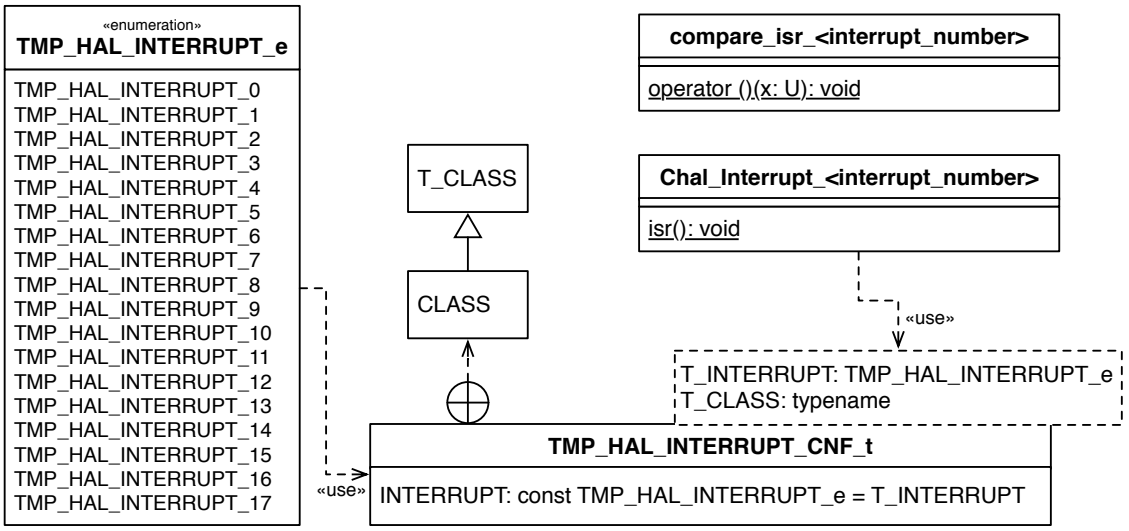


Figura 5.11: Diagrama de classes - Módulo de gestão das interrupções

O diagrama de classes define a classe *template* *TMP_HAL_INTERRUPT_CNF_t* que permite configurar uma interrupção. A classe *template* recebe dois parâmetros de entrada de *template*: o parâmetro de entrada *T_INTERRUPT*, que indica o número da interrupção a configurar (obtido através da enumeração *TMP_HAL_INTERRUPT_e*) e o parâmetro de entrada *T_CLASS*, que recebe uma classe do tipo *Chal_Interrupt_<interrupt_number>*.

A classe *Chal_Interrupt_<interrupt_number>* contém um método com o nome *isr*, o qual efetua a chamada da função que implementa a *ISR* (rotina de serviço à interrupção) de uma interrupção. Existe uma classe deste tipo para cada uma das interrupções, em que o *<interrupt_number>* contido no nome da classe é substituído pelo número da interrupção correspondente. O código 5.18 apresenta a implementação das várias classes *Chal_Interrupt_<interrupt_number>*.

Código 5.18: Implementação do módulo de gestão das interrupções

```
#define TMP_INTERRUPTS_GEN(INT_NUM, ISR) \
template < typename T_CLASS > \
```

```
struct Chal_Interrupt_ ## INT_NUM \
{ \
  inline static void isr() \
  { \
    ISR ## ;\
  } \
};

TMP_INTERRUPTS_GEN(2, T_CLASS::RxIsr())
TMP_INTERRUPTS_GEN(3, T_CLASS::RxIsr())
TMP_INTERRUPTS_GEN(7, T_CLASS::TxIsr())
TMP_INTERRUPTS_GEN(8, T_CLASS::Isr())
TMP_INTERRUPTS_GEN(14, T_CLASS::TxIsr())
```

Analisando o código apresentado, pode-se constatar que todas as classes *Chal_Interrupt_<interrupt_number>* são geradas por uma macro com parâmetros, denominada por *TMP_INTERRUPTS_GEN*. A classe gerada implementa o método *inline isr* para evitar que o compilador gere uma chamada de função (*call*).

Configuração do módulo de gestão das interrupções

***Framework* da *Texas Instruments*:**

O código 5.19 apresenta como são configuradas as interrupções na *Framework* da *Texas Instruments*.

Código 5.19: Configuração do módulo de gestão das interrupções - *Framework* da *Texas Instruments*

```
#define HAL_UART_ISR = 1

#if HAL_UART_ISR
#include "_hal_uart_isr.c"
#endif
```

As interrupções são ativadas através da definição de macros de pré-processamento. Por exemplo, para ativar a interrupção da porta série em modo ISR, tem que ser definida a macro *HAL_UART_ISR*.

***Framework* Generativa:**

A configuração das interrupções com a *Framework* Generativa é apresentada no código 5.20.

Código 5.20: Configuração do módulo de gestão das interrupções - *Framework* Generativa

```
typedef TMP_HAL_INTERRUPT_CNF_t<TMP_HAL_INTERRUPT_3, Chal_uart >
    INTERRUPT_3_CNF;
```

```
typedef TMP_HAL_INTERRUPT_CNF_t<TMP_HAL_INTERRUPT_14, Chal_uart >
    INTERRUPT_14_CNF;

typedef mpl::vector<INTERRUPT_3_CNF, INTERRUPT_14_CNF> V_INTERRUPTS;
```

Para configurar cada uma das interrupções, é utilizada a estrutura de configuração *TMP_HAL_INTERRUPT_CNF_t*, sendo depois preenchido um vetor MPL de nome *V_INTERRUPTS* com essas estruturas de configuração. Esse vetor é utilizado para gerar o código das ISRs.

5.2.5 ADC

Para manipular o ADC presente no microcontrolador é disponibilizado um módulo com o nome *Chal_ADC*. Com este módulo é possível configurar e utilizar o ADC e os seus canais.

Diagrama de funcionalidades:

Na figura 5.12 é apresentado o diagrama de funcionalidades do módulo de gestão do ADC.

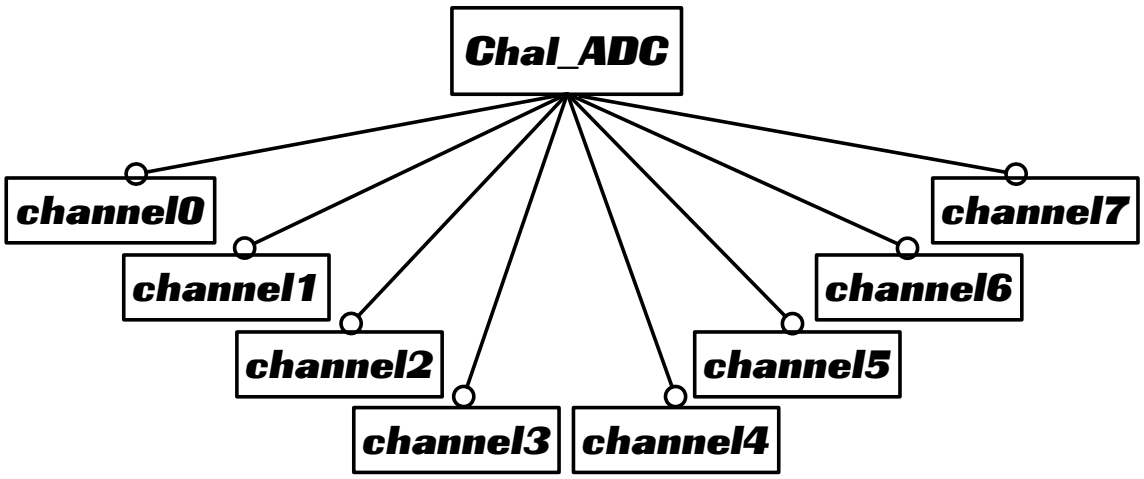


Figura 5.12: Diagrama de funcionalidades - Módulo de gestão do ADC

O diagrama demonstra que o ADC, representado pela classe *Chal_ADC*, é composto por oito canais (*channel0* a *channel7*). Estes canais podem ser selecionados opcionalmente pelo utilizador.

Implementação da configuração do módulo de gestão do ADC:

A figura 5.13 apresenta o diagrama de classes da estrutura de configuração do módulo ADC.

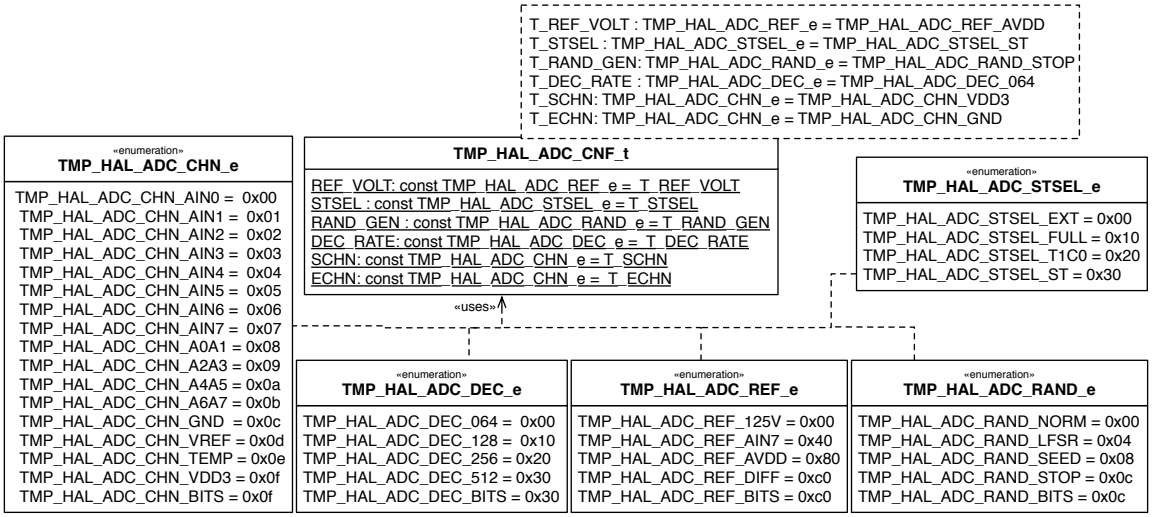


Figura 5.13: Diagrama de classes - Configuração do ADC

A classe *template* *TMP_HAL_ADC_CNF_t* permite efetuar a configuração do módulo ADC. Esta classe recebe seis parâmetros de entrada de *template*, que permitem definir as várias configurações do mesmo. Para cada uma das configurações, existe uma enumeração com as várias opções de escolha (*TMP_HAL_ADC_RAND_e*, *TMP_HAL_ADC_CHN_e*, *TMP_HAL_ADC_DEC_e*, *TMP_HAL_ADC_REF_e*, *TMP_HAL_ADC_STSEL_e*).

O código 5.21 apresenta como foi implementada a classe *TMP_HAL_ADC_CNF_t*.

Código 5.21: Implementação da configuração do módulo de gestão do ADC

```
template <
TMP_HAL_ADC_REF_e T_REF_VOLT = TMP_HAL_ADC_REF_AVDD,
TMP_HAL_ADC_STSEL_e T_STSEL = TMP_HAL_ADC_STSEL_ST,
TMP_HAL_ADC_RAND_e T_RAND_GEN = TMP_HAL_ADC_RAND_STOP,
TMP_HAL_ADC_DEC_e T_DEC_RATE = TMP_HAL_ADC_DEC_064,
TMP_HAL_ADC_CHN_e T_SCHN = TMP_HAL_ADC_CHN_VDD3,
TMP_HAL_ADC_CHN_e T_ECHN = TMP_HAL_ADC_CHN_GND
        >
struct TMP_HAL_ADC_CNF_t{
  static const TMP_HAL_ADC_REF_e REF_VOLT =  T_REF_VOLT;
  static const TMP_HAL_ADC_STSEL_e STSEL = T_STSEL;
  static const TMP_HAL_ADC_RAND_e RAND_GEN = T_RAND_GEN;
```

```
    static const TMP_HAL_ADC_DEC_e DEC_RATE =  T_DEC_RATE;
    static const TMP_HAL_ADC_CHN_e SCHN = T_SCHN;
    static const TMP_HAL_ADC_CHN_e ECHN =  T_ECHN;
};
```

Como se pode verificar, a classe *template TMP_HAL_ADC_CNF_t* tem seis parâmetros de entrada de *template*. Esses parâmetros de entrada são atribuídos a variáveis constantes estáticas, as quais serão acedidas pelo módulo de gestão do ADC para executar segundo as configurações desses parâmetros.

Implementação do módulo de gestão do ADC:

A figura 5.14 contém o diagrama de classes do módulo de gestão do ADC.

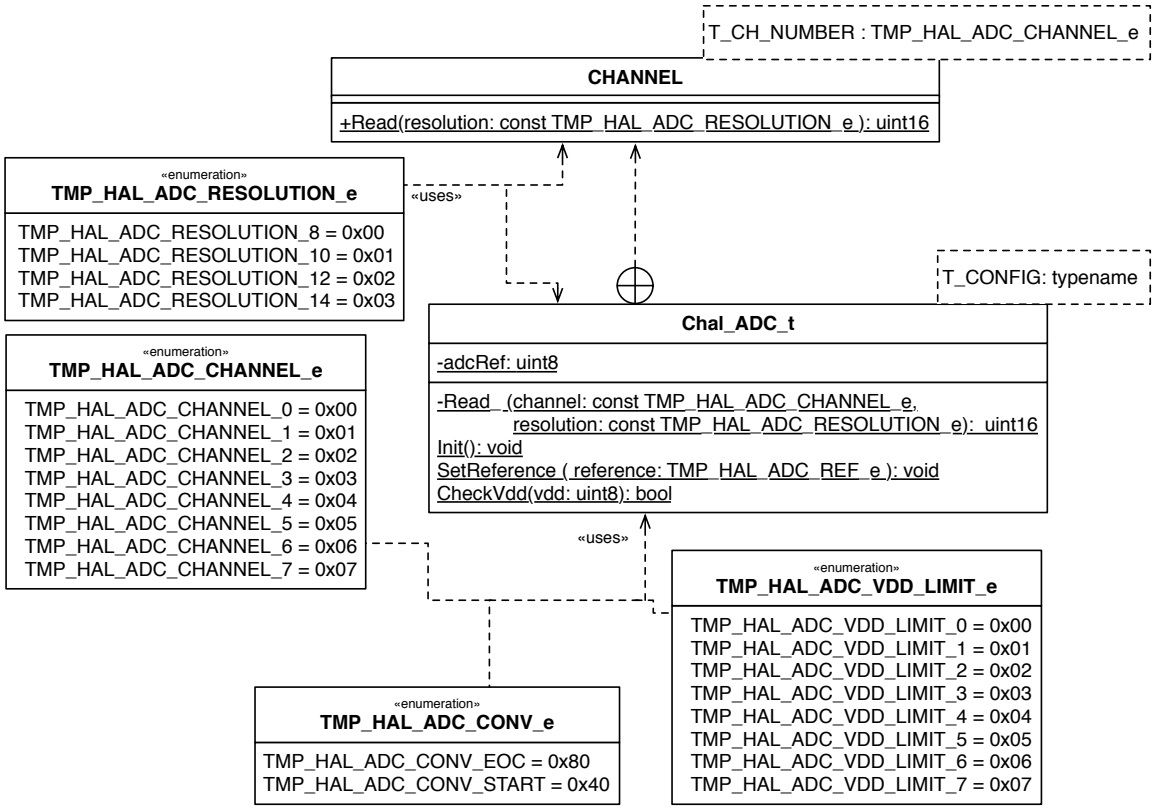


Figura 5.14: Diagrama de classes - Módulo de gestão do ADC

O módulo de gestão do ADC é implementado pela classe *template Chal_ADC_t*. Esta, recebe como parâmetro de entrada de *template* uma estrutura de configuração *TMP_HAL_ADC_CNF_t* que fornece as configurações aplicadas ao módulo.

A classe *template Chal_ADC_t* possui métodos para leitura (*Read_()*), iniciação (*init()*), alteração da referência (*SetReference()*) e verificação da tensão de VDD do ADC. Para além disso, também possui internamente uma meta-função denominada por *CHANNEL*. Esta meta-função possibilita o acesso aos canais do ADC, passando como parâmetro de entrada de *template* o número do canal a aceder (*T_CH_NUMBER*). Internamente à meta-função é disponibilizado o método *Read*, que permite fazer a leitura do valor de tensão num dado canal.

O código 5.22 apresenta a implementação da modelação do diagrama de classes anterior.

Código 5.22: Implementação do módulo de gestão do ADC

```
template <typename T_CONFIG>
class Chal_ADC_t {
  static uint8 adcRef;
  static uint16 Read_ (const TMP_HAL_ADC_CHANNEL_e channel, const
     TMP_HAL_ADC_RESOLUTION_e resolution);
public:
  inline static void Init (void);
  static void SetReference ( TMP_HAL_ADC_REF_e reference );
  static bool CheckVdd(uint8 vdd);

  template <TMP_HAL_ADC_CHANNEL_e T_CH_NUMBER>
  struct CHANNEL{
    static_assert(T_CH_NUMBER <= TMP_HAL_ADC_CHANNEL_7, "Invalid
       Channel\n");

    inline static uint16 Read( const TMP_HAL_ADC_RESOLUTION_e
       resolution ){
      return Read_(T_CH_NUMBER, resolution);
    }
  };
};
```

O método *Read_* da classe *template Chal_ADC_t* é privado, e a leitura do valor de um canal do ADC só pode ser efetuado através do método *Read* pertencente à meta-função *CHANNEL*. Isto para que seja validado em tempo de compilação, se o canal acedido é válido (canal de 0 a 7). Essa validação é efetuada pela função *static_assert* que emite um erro de compilação "*Invalid Channel*", caso a meta-função *CHANNEL* seja chamada com um *T_CH_NUMBER* inválido. O método *Read* da meta-função *CHANNEL* apenas se limita a fazer a chamada ao método *Read_* da classe *Chal_ADC_t*.

Configuração do módulo de gestão do ADC

***Framework* da *Texas Instruments*:**

O código 5.23 apresenta como é efetuada a configuração do módulo de gestão do ADC.

Código 5.23: Configuração do módulo de gestão do ADC - *Framework* da *Texas Instruments*

```
#define HAL_ADC_REF_VOLT   HAL_ADC_REF_AVDD
#define HAL_ADC_STSEL      HAL_ADC_STSEL_ST
#define HAL_ADC_RAND_GEN   HAL_ADC_RAND_STOP
#define HAL_ADC_DEC_RATE   HAL_ADC_DEC_064
```

A configuração é efetuada através da definição de macros de pré-processamento, definindo uma para cada valor de configuração.

***Framework* Generativa:**

No código 5.24 apresenta como se efetua a configuração do módulo ADC com a *Framework* Generativa.

Código 5.24: Configuração do módulo de gestão do ADC - *Framework* Generativa

```
typedef TMP_HAL_ADC_CNF_t< TMP_HAL_ADC_REF_AVDD,
                           TMP_HAL_ADC_STSEL_ST,
                           TMP_HAL_ADC_RAND_STOP,
                           TMP_HAL_ADC_DEC_064 > ADC_CNF;

typedef Chal_ADC_t< ADC_CNF > Chal_ADC;
```

A configuração é feita pela estrutura de configuração *TMP_HAL_ADC_CNF_t*, sendo esta passada como parâmetro de entrada de *template* à classe que implementa o módulo de gestão do ADC (*Chal_ADC_t*).

5.2.6 DMA

Para manipulação do DMA presente no microcontrolador é disponibilizado um módulo com o nome *Chal_DMA*. Com este módulo é possível aceder e configurar os vários canais disponibilizados pelo DMA.

Diagrama de funcionalidades:

O diagrama de funcionalidades presente na figura 5.15 apresenta a variabilidade do módulo de gestão do DMA.

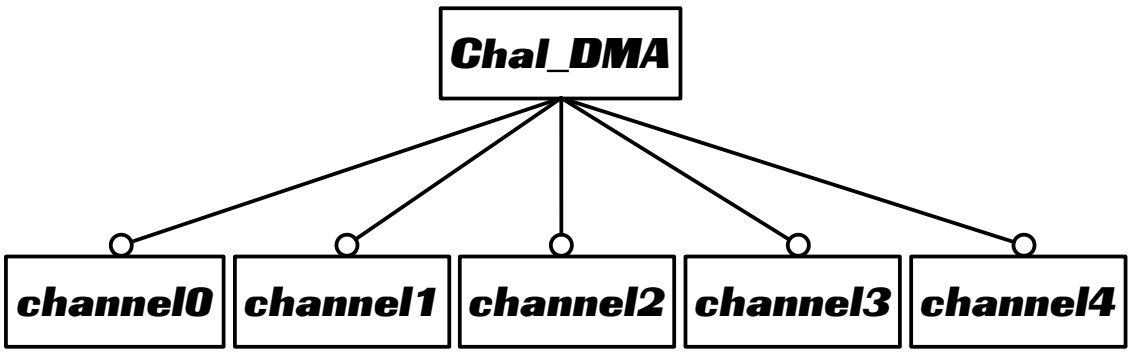


Figura 5.15: Diagrama de funcionalidades - Módulo de gestão do DMA

O DMA do microcontrolador possui cinco canais que podem ser acedidos e configurados de forma individual e opcional pelo utilizador.

Implementação do módulo de gestão do DMA:

Na figura 5.16 está presente o diagrama de classes da implementação do módulo de gestão do DMA.

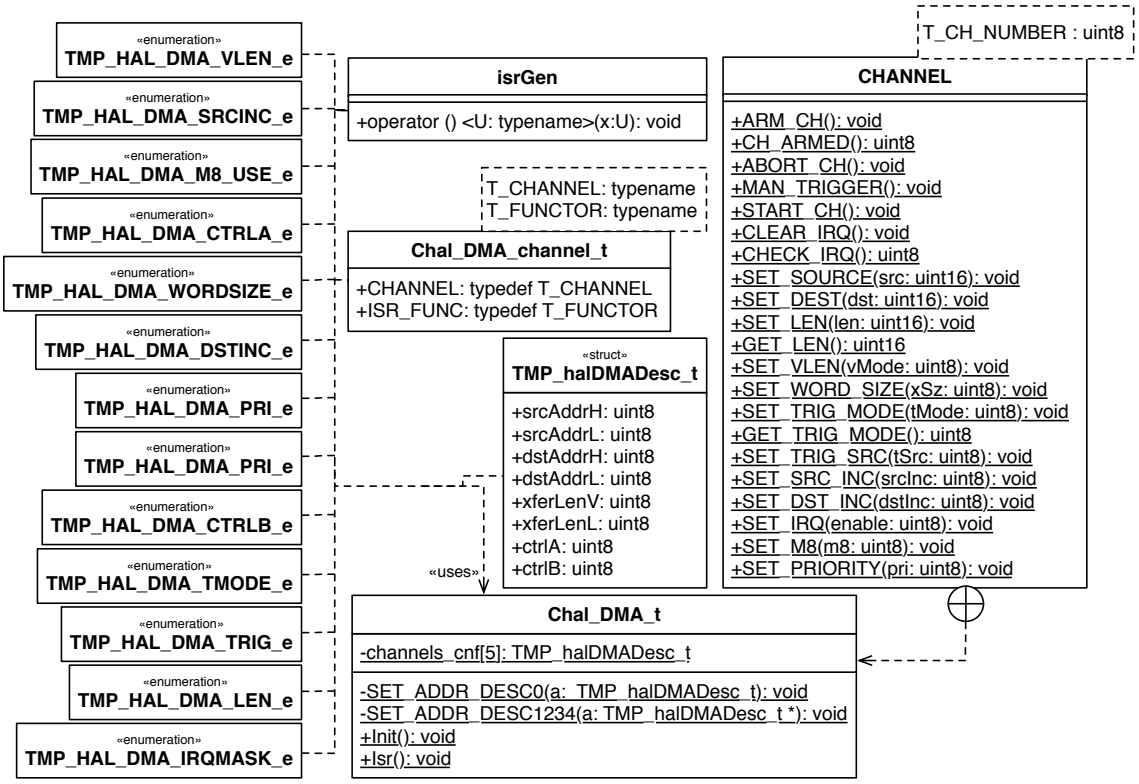


Figura 5.16: Diagrama de classes - Configuração do DMA

A classe *Chal_DMA_t* implementa o módulo de gestão do DMA. Esta classe contém o atributo privado *channels_cnf*, que é um *array* de cinco posições com a configuração de cada um dos canais do DMA.

Os métodos da classe *Chal_DMA_t* são quatro, dois privados e dois públicos. Os métodos privados são: o método *SET_ADDR_DESC0*, que permite definir o endereço da estrutura de configuração do primeiro canal (canal zero) e o método *SET_ADDR_DESC1234*, que permite definir o endereço da estrutura de configuração do resto dos canais (canal um a quatro). Os métodos públicos são: o método *init*, que permite a inicialização do DMA e o método *isr*, que implementa a rotina de serviço à interrupção do DMA.

Para além dos métodos e atributos descritos anteriormente, a classe *Chal_DMA_t* também possui uma meta-função com o nome *CHANNEL* que permite aceder aos vários canais do ADC. Esta meta-função recebe como parâmetro de entrada de *template* o número do canal a aceder (*T_CH_NUMBER*) e disponibiliza vários métodos que permitem manipular o canal selecionado.

A classe *template Chal_DMA_channel_t* presente no diagrama, permite efetuar a configuração de um canal. Esta, tem como parâmetros de entrada de *template*, o parâmetro de entrada *T_CHANNEL* que indica o canal a configurar e o parâmetro de entrada de *template T_FUNCTOR* que indica a classe do módulo que irá utilizar esse canal do DMA.

Neste diagrama de classes também está presente a classe *isrGen*, que implementa um *functor* que permite a geração da ISR do DMA.

Para cada uma das configurações do DMA, existe uma enumeração. A figura 5.17 apresenta todas essas enumerações mais detalhadamente.

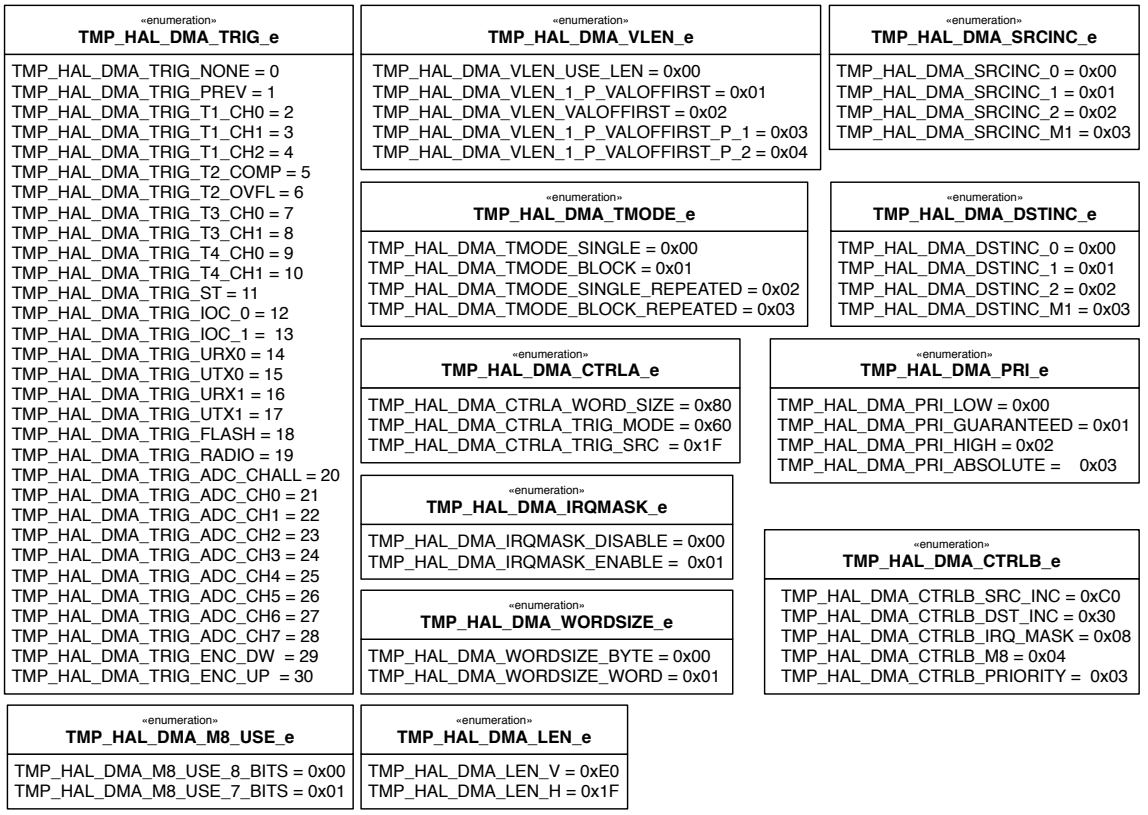


Figura 5.17: Diagrama de classes - *Enums* de configuração do DMA

O código 5.25 apresenta a implementação da ISR do módulo DMA.

Código 5.25: ISR do módulo de gestão do DMA

```
template <typename T_CHANNEL, typename T_FUNCTOR>
struct Chal_DMA_channel_t {
  typedef T_CHANNEL CHANNEL;
  typedef T_FUNCTOR ISR_FUNC;
};

struct isrGen{
  template< typename U >
  void operator()(U x){
    if (U::CHANNEL::CHECK_IRQ()){
      U::CHANNEL::CLEAR_IRQ();
      U::ISR_FUNC::Isr();
    }
  }
};

void Chal_DMA_t::Isr( void ){
...
  mpl::for_each<TMP_CONFIG::V_CHANNELS>( isrGen() );
...
}
```

O DMA do microcontrolador dispõe apenas de uma interrupção comum a todos os canais do mesmo. Para verificar qual dos canais despoletou a interrupção, existe uma *flag* para cada um dos canais que pode ser verificada. À medida que se pretenda utilizar mais canais no DMA, tem que ser adicionado código à ISR respeitante à verificação da *flag* e à execução do conteúdo da interrupção respeitante ao canal.

Para adicionar canais de forma escalável à ISR utilizou-se o *for-each* do MPL. O *for-each* percorre um vetor MPL que contém uma lista de configurações de canais, sendo estas efetuadas através da classe *Chal_DMA_channel_t*. Para cada um dos elementos do vetor MPL, é gerado o código contido no *functor isrGen*.

O *functor isrGen* implementa, por cada canal, a condição que verifica a *flag* de interrupção respeitante ao canal (*U::CHANNEL::CHECK_IRQ()*). E no interior dessa condição, gera a limpeza dessa *flag* (*U::CHANNEL::CLEAR_IRQ()*), seguida da chamada da função que implementa as ações da interrupção referente ao canal (*U::ISR_FUNC::Isr()*).

Configuração do módulo de gestão do DMA

***Framework* da *Texas Instruments*:**

O código 5.26 apresenta como é configurado o módulo de gestão do DMA recorrendo à *Framework* da *Texas Instruments*.

Código 5.26: Configuração do módulo de gestão do DMA - *Framework* da *Texas Instruments*

```
#define HAL_DMA = TRUE
#define HAL_UART_DMA = 1
```

A configuração do módulo de gestão do DMA é implementada também com recurso a definições de macros de pré-processamento. O código anterior é referente à ativação do DMA para a porta série em modo DMA.

***Framework* Generativa:**

No código 5.27 está presente a mesma configuração, mas com a *Framework* Generativa.

Código 5.27: Configuração do módulo de gestão do DMA - *Framework* Generativa

```
typedef Chal_DMA_channel_t< Chal_DMA_t::CHANNEL< HAL_DMA_CH_TX >,
    Chal_uart > uart_channel;

typedef mpl::vector< uart_channel > V_CHANNELS;
```

Com a *Framework* Generativa, a ativação do DMA para utilização de um canal na porta série é efetuado da seguinte maneira: primeiramente, e recorrendo à estrutura de configuração *Chal_DMA_channel_t*, é implementada a atribuição de um canal DMA (*Chal_DMA_t::CHANNEL<HAL_DMA_CH_TX>*) ao módulo que implementa a porta série (*Chal_uart*). E de seguida, é adicionada essa configuração a um vetor MPL com o nome *V_CHANNELS*.

5.2.7 Portas Série

Para manipulação das portas série disponíveis no microcontrolador é facultado um módulo com o nome *Chal_uart*. Com a utilização deste módulo é possível configurar e utilizar as duas portas série disponíveis no microcontrolador.

Diagrama de funcionalidades:

A figura 5.18 apresenta o diagrama de funcionalidades que esquematiza o grau de variabilidade do módulo de gestão das portas série.

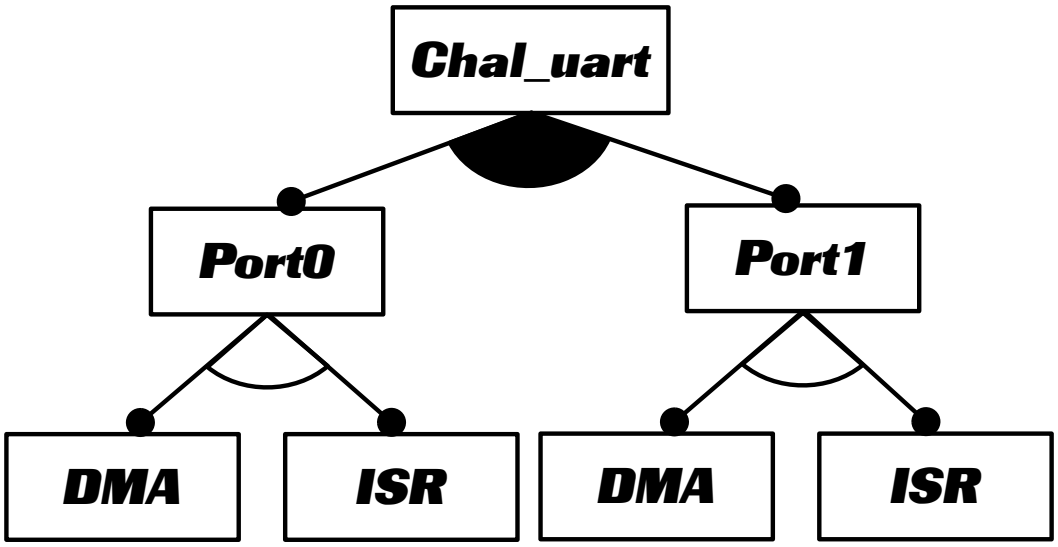


Figura 5.18: Diagrama de funcionalidades - Módulo de gestão das portas série

O diagrama mostra que, sendo o módulo de gestão das portas série (*Chal_uart)* selecionado, tem de ser selecionada pelo menos uma das portas série. As escolhas podem ser: a porta série zero (*Port0*) ou porta série um (*Port1*).

Se essa escolha incidir sobre a porta série zero (*Port0*) ou porta série um (*Port1*), tem que ser obrigatoriamente selecionado um, e só um, meio de comunicação com a mesma. Este pode ser, por DMA ou por ISR. (Nota: De salientar que foram implementados os dois meios de comunicação. No entanto, devido a terem implementações muito idênticas, neste relatório apenas é apresentada a implementação para o meio DMA).

Implementação das estruturas de configuração geral das portas série:

Implementação da seleção da porta série:

A figura 5.19 ilustra as estruturas de configuração geral dos portos da porta série.

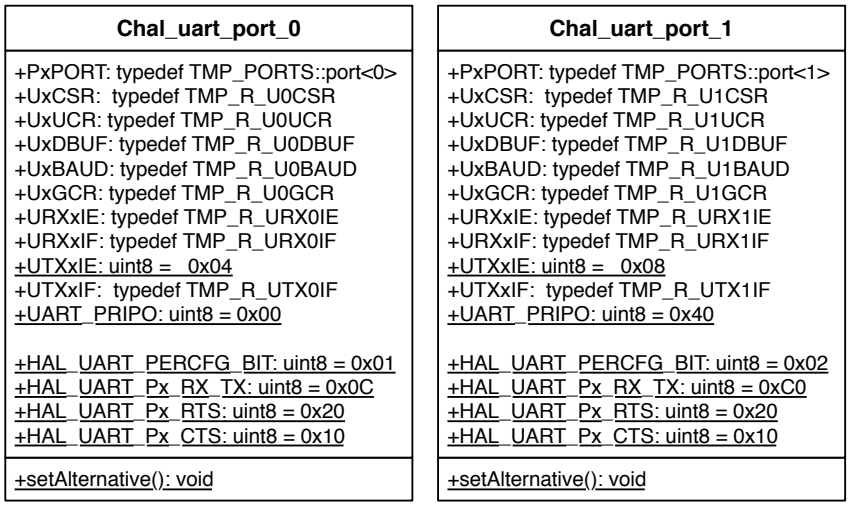


Figura 5.19: Diagrama de classes - Estruturas com valores de configuração geral para os diferentes portos

Para cada uma das portas série (*Port0* e *Port1*) foi implementada uma estrutura que contém todas as configurações respeitantes à porta em questão, tais como: o porto de entrada/saída onde está mapeada a porta série, as configurações e as estruturas de acesso aos registos SFR referentes ao módulo UART.

O código 5.28 apresenta a implementação da estrutura com os valores de configuração para a porta série zero (*Port0*).

Código 5.28: Estruturas com valores de configuração para o porto zero

```
struct Chal_uart_port_0{
  typedef TMP_PORTS::port<0>     PxPORT;
  typedef TMP_R_U0CSR            UxCSR;
  typedef TMP_R_U0UCR            UxUCR;
  typedef TMP_R_U0DBUF           UxDBUF;
  typedef TMP_R_U0BAUD           UxBAUD;
  typedef TMP_R_U0GCR            UxGCR;
  typedef TMP_R_URX0IE           URXxIE;
  typedef TMP_R_URX0IF           URXxIF;
  typedef TMP_R_UTX0IF           UTXxIF;

  static const uint8 UTXxIE =    0x04;
  static const uint8 UART_PRIPO = 0x00;
  static const uint8 HAL_UART_PERCFG_BIT = 0x01;
  static const uint8 HAL_UART_Px_RX_TX =   0x0C;
  static const uint8 HAL_UART_Px_RTS =     0x20;
  static const uint8 HAL_UART_Px_CTS =     0x10;

  inline static void setAlternative();
};
```

Os registos SFR de configuração da porta série são diferentes para cada uma das portas. A estrutura *Chal_uart_port_0* cria *typedefs* internos que dão um nome genérico a cada uma das estruturas que implementam o acesso ao registo do SFR referentes ao número da porta série em questão, neste caso porta série zero. Por exemplo, o registo *TMP_R_U0CSR* é acedido pelo nome *UxCSR*.

Os valores das configurações aplicadas aos registos SFR, para configurar a porta série em questão, são guardados em atributos constantes da estrutura.

Implementação da seleção dos *baud rates*:

Na figura 5.20 estão presentes as estruturas de configuração para os diversos *baud rates*.

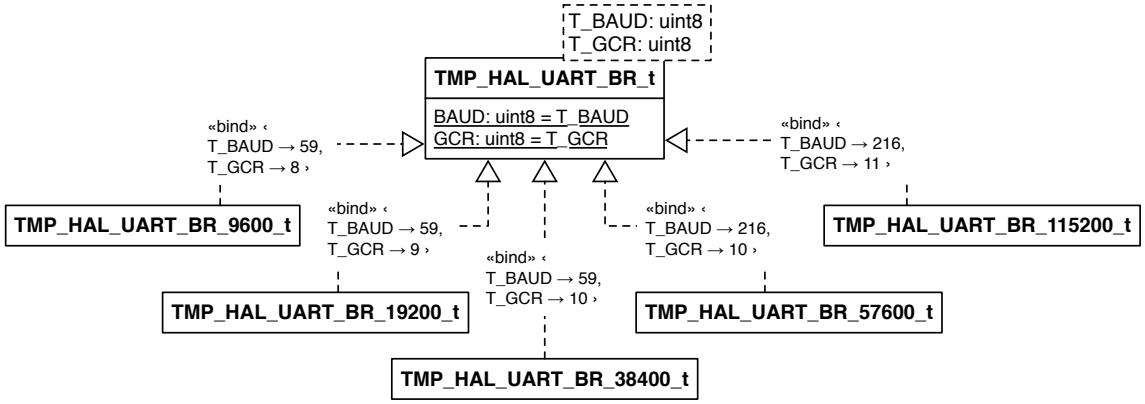


Figura 5.20: Diagrama de classes - Estruturas com valores de configuração geral para todos os *baud rates*

Para cada um dos *baud rates* disponíveis, existe uma estrutura de dados com os valores de configurações que devem ser aplicados nos registos do SFR para o *baud rate* em questão. As estruturas são criadas através da classe *template* *TMP_HAL_UART_BR_t*. Esta classe, recebe dois parâmetros de entrada de *template* respeitantes às configurações (*T_BAUD* e *T_GCR*) e atribui os valores desses dois parâmetros a dois atributos internos à classe (*BAUD* e *GCR*).

O código 5.29 apresenta a implementação das estruturas de configuração para os diversos *baud rates*, tal como foram modeladas no diagrama de classes.

Código 5.29: Estruturas com valores de configuração geral para todos os *baud rates*

```
template <uint8 T_BAUD, uint8 T_GCR>
```

```
struct TMP_HAL_UART_BR_t{
  static const uint8 BAUD = T_BAUD;
  static const uint8 GCR = T_GCR;
};

typedef TMP_HAL_UART_BR_t<59, 8> TMP_HAL_UART_BR_9600_t;
typedef TMP_HAL_UART_BR_t<59, 9> TMP_HAL_UART_BR_19200_t;
typedef TMP_HAL_UART_BR_t<59, 10> TMP_HAL_UART_BR_38400_t;
typedef TMP_HAL_UART_BR_t<216,10> TMP_HAL_UART_BR_57600_t;
typedef TMP_HAL_UART_BR_t<216,11> TMP_HAL_UART_BR_115200_t;
```

As estruturas são implementadas recorrendo ao *typedef*. Para cada um dos *baud rates* é criado um novo *typename* baseado na classe *template* *TMP_HAL_UART_BR_t*, atribuindo aos seus parâmetros de entrada de *template* as configurações para o *baud rate* em questão.

Implementação do módulo de gestão das portas série:

A figura 5.21 apresenta o diagrama de classes do módulo que implementa a gestão das portas série.

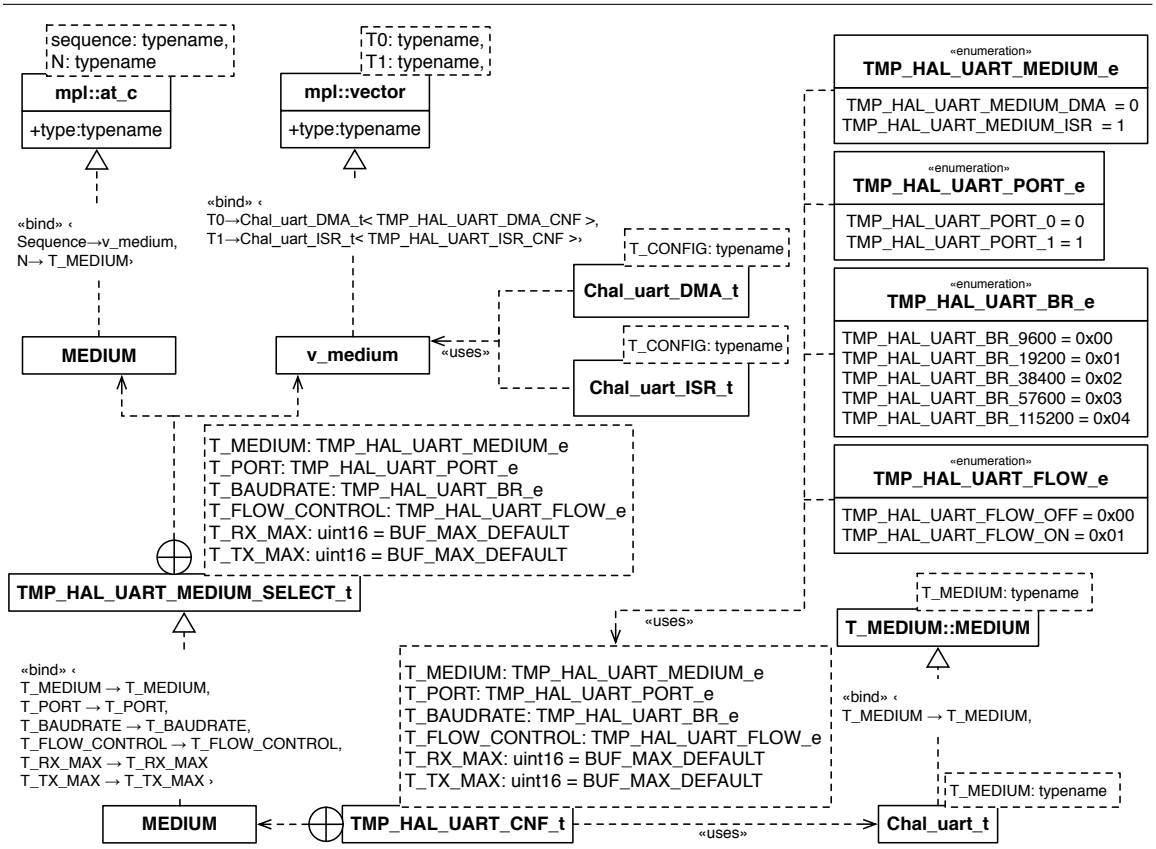


Figura 5.21: Diagrama de classes - Configuração geral da porta série

O módulo que permite fazer a gestão das portas série é implementado pela classe *template Chal_uart_t*. Esta, recebe como parâmetro de entrada de *template* as configurações baseadas na estrutura de configuração *TMP_HAL_UART_CNF_t*. Essa estrutura de configuração permite definir várias configurações, tais como: o meio de comunicação, através do seu parâmetro de entrada de *template T_MEDIUM*; o número da porta série a configurar, através do seu parâmetro de entrada de *template T_PORT*; o *baud rate* pelo parâmetro de entrada de *template T_BAUDRATE*; o controlo de fluxo através do parâmetro de entrada de *template T_FLOW_CONTROL* e por fim o tamanho dos *buffers* de receção e envio, através dos parâmetros de entrada de *template T_RX_MAX* e *T_TX_MAX*, respetivamente.

A escolha do meio de comunicação a utilizar, é implementado pela meta-função *TMP_HAL_UART_MEDIUM_SELECT_t*. Esta meta-função, recorre a um vetor MPL de nome *v_medium* e à meta-função *at_c* para fornecer a classe que implementa o meio de comunicação selecionado. Se o meio de comunicação por DMA for selecionado, é fornecida a classe *Chal_uart_DMA_t*, ou se o meio de comunicação por ISR for selecionado, é fornecida a classe *Chal_uart_ISR_t*. O código 5.30 apresenta uma parte da implementação da meta-função *TMP_HAL_UART_MEDIUM_SELECT_t*.

Código 5.30: Configuração geral da porta série

```
template <  TMP_HAL_UART_MEDIUM_e T_MEDIUM,
            TMP_HAL_UART_PORT_e T_PORT,
            TMP_HAL_UART_BR_e T_BAUDRATE,
            TMP_HAL_UART_FLOW_e T_FLOW_CONTROL,
            uint16 T_RX_MAX = BUF_MAX_DEFAULT,
            uint16 T_TX_MAX = BUF_MAX_DEFAULT >
class TMP_HAL_UART_MEDIUM_SELECT_t {
  static_assert(T_MEDIUM <= TMP_HAL_UART_MEDIUM_ISR, "Invalid Medium"
      );
...
};
```

Se for chamada a meta-função *TMP_HAL_UART_MEDIUM_SELECT_t* com a seleção de um meio de comunicação inválido, é mostrado o erro de compilação "*Invalid Medium*". A implementação desse erro é conseguida através da função *static_assert* que analisa o parâmetro de entrada de *template T_MEDIUM*.

Implementação das estruturas de configuração das portas série para o meio de comunicação por DMA:

Implementação da seleção da porta série:

A figura 5.22 apresenta o diagrama de classes da estrutura de seleção das portas

série para o meio de comunicação por DMA.

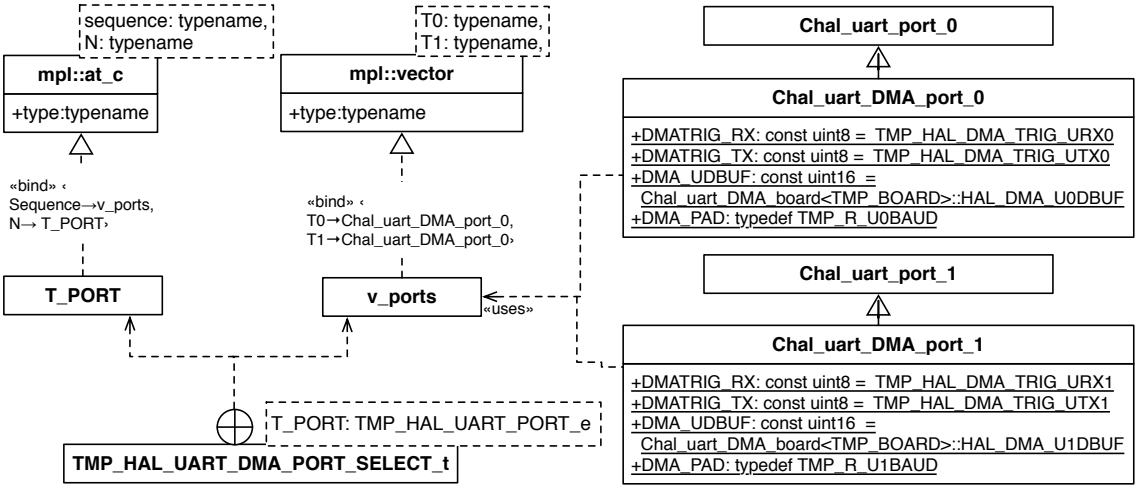


Figura 5.22: Diagrama de classes - Estruturas com valores de configuração no meio DMA para os diferentes portos

A seleção do número da porta série é implementada por uma meta-função de nome *TMP_HAL_UART_DMA_PORT_SELECT_t*. Esta meta-função fornece a estrutura que contém as configurações para a porta série, selecionada através do parâmetro de entrada de *template T_PORT*. A meta-função é implementada com recurso à meta-função MPL *at_c* e a um vetor MPL de nome *v_ports*.

O vetor *v_ports* contém as estruturas de configuração para as duas portas série. Estas são: *Chal_uart_DMA_port_0* referente à porta série zero e *Chal_uart_DMA_port_1* referente à porta série um. O código 5.31 apresenta a implementação da estrutura com os valores de configuração da porta série zero, para o meio de comunicação por DMA.

Código 5.31: Estruturas com valores de configuração no meio DMA para o porto zero

```
struct Chal_uart_DMA_port_0 : Chal_uart_port_0{
  static const uint8 DMATRIG_RX = TMP_HAL_DMA_TRIG_URX0;
  static const uint8 DMATRIG_TX = TMP_HAL_DMA_TRIG_UTX0;
  static const uint16 DMA_UDBUF = Chal_uart_DMA_board<TMP_BOARD>::
     HAL_DMA_U0DBUF;
  typedef TMP_R_U0BAUD DMA_PAD;
};

template <TMP_HAL_UART_PORT_e T_PORT>
class TMP_HAL_UART_DMA_PORT_SELECT_t {
```

```
    static_assert(T_PORT<=TMP_HAL_UART_PORT_1, "Invalid port");
...
};
```

A estrutura com os valores de configuração para o meio de comunicação por DMA, herda as configurações gerais para a porta série em questão. Ou seja, as configurações que não dependem do meio de comunicação. No código é apresentada a estrutura com os valores de configuração para a porta série zero, no meio comunicação por DMA. Esta é implementada pela estrutura de dados *Chal_uart_DMA_port_0* e herda as configurações gerais da porta série zero através da estrutura de dados *Chal_uart_port_0*.

Neste código também está presente a implementação da meta-função de seleção da porta série, *TMP_HAL_UART_DMA_PORT_SELECT_t*. Esta meta-função emite o erro de compilação "*Invalid port*" se ao parâmetro de entrada de *template* *T_PORT* for atribuído um número de porta série inválido.

Implementação da seleção dos *baud rates*:

A figura 5.23 apresenta o diagrama de classes das estruturas com valores de configuração no meio de comunicação por DMA para todos os *baud rates*.

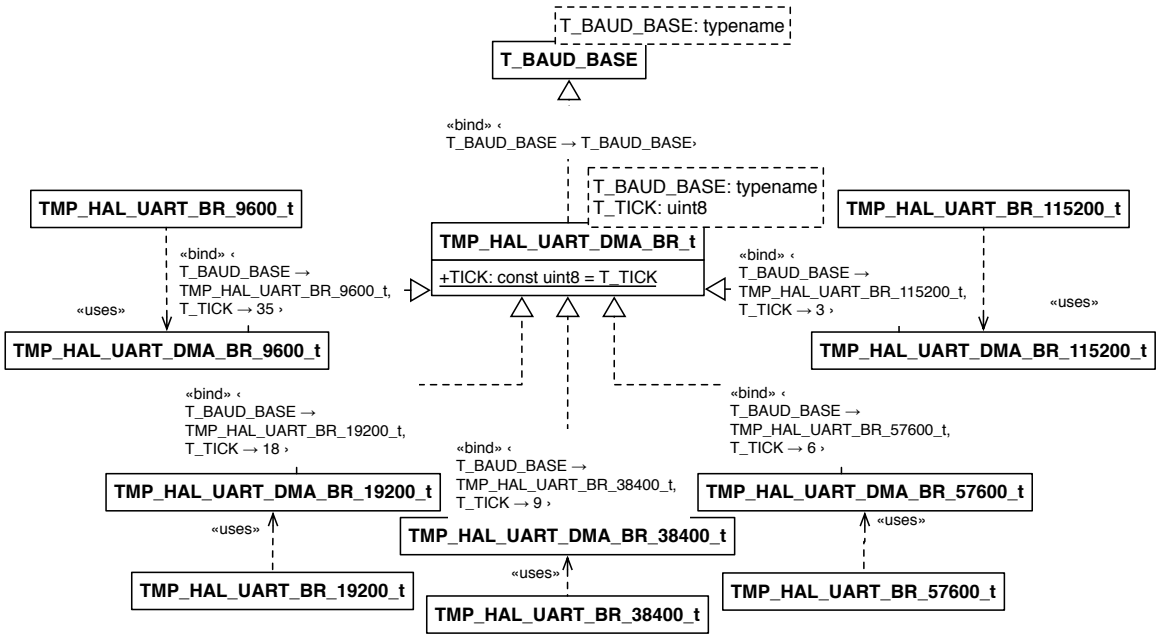


Figura 5.23: Diagrama de classes - Estruturas com valores de configuração no meio DMA para todos os *baud rates*

As estruturas, que contêm as configurações para os diferentes *baud rates* existentes no meio de comunicação por DMA, são implementadas com recurso à utilização da classe *template TMP_HAL_UART_DMA_BR_t*. Esta classe, recebe no parâmetro de entrada de *template T_BAUD_BASE*, a estrutura que implementa as configurações gerais do *baud rate* em questão, e um parâmetro de entrada *T_TICK*. A classe *template TMP_HAL_UART_DMA_BR_t* herda todos os membros da classe base (*T_BAUD_BASE*).

O código 5.32 é referente à implementação das estruturas que contêm os valores de configuração no meio de comunicação DMA para todos os *baud rates*.

Código 5.32: Estruturas com valores de configuração no meio DMA para todos os *baud rates*

```
template <typename T_BAUD_BASE, uint8 T_TICK>
struct TMP_HAL_UART_DMA_BR_t : public T_BAUD_BASE{
  static const uint8 TICK = T_TICK;
};

typedef TMP_HAL_UART_DMA_BR_t<TMP_HAL_UART_BR_9600_t, 35>
    TMP_HAL_UART_DMA_BR_9600_t;

typedef TMP_HAL_UART_DMA_BR_t<TMP_HAL_UART_BR_19200_t, 18 >
    TMP_HAL_UART_DMA_BR_19200_t;

typedef TMP_HAL_UART_DMA_BR_t<TMP_HAL_UART_BR_38400_t, 9>
    TMP_HAL_UART_DMA_BR_38400_t;

typedef TMP_HAL_UART_DMA_BR_t<TMP_HAL_UART_BR_57600_t, 6>
    TMP_HAL_UART_DMA_BR_57600_t;

typedef TMP_HAL_UART_DMA_BR_t<TMP_HAL_UART_BR_115200_t, 3>
    TMP_HAL_UART_DMA_BR_115200_t;
```

Para cada um dos *baud rates*, é implementado um novo *typename* recorrendo ao *typedef*. Esse *typename* é baseado na classe *template TMP_HAL_UART_DMA_BR_t* que recebe para cada um dos *baud rates* a estrutura que contém as configurações gerais para esse *baud rate* e o valor aplicado ao parâmetro de entrada *T_TICK*.

A figura 5.24 apresenta o diagrama de classes da meta-função que implementa a escolha do *baud rate*.

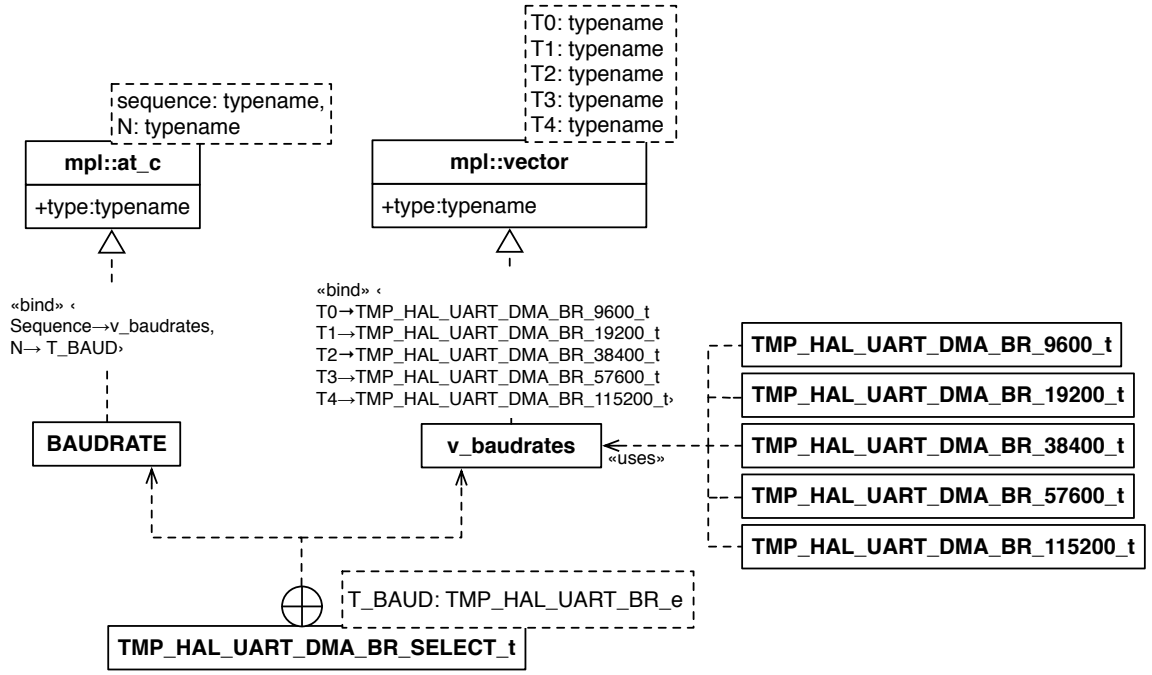


Figura 5.24: Diagrama de classes - Seleção do *baud rate* no meio DMA

A seleção do *baud rate* a utilizar, referente ao meio de comunicação DMA, é implementado pela meta-função *TMP_HAL_UART_DMA_BR_SELECT_t*. Esta, recebe como parâmetro de entrada de *template* o número do *baud rate* selecionado (*T_BAUD*). Com o recurso à utilização da meta-função *at_c* do MPL e de um vetor MPL de nome *v_baudrates*, é devolvida a estrutura que contém as configurações do *baud rate* selecionado. O código 5.33 apresenta a implementação da meta-função *TMP_HAL_UART_DMA_BR_SELECT_t*.

Código 5.33: Seleção do *baud rate* no meio DMA

```
template <TMP_HAL_UART_BR_e T_BAUD>
class TMP_HAL_UART_DMA_BR_SELECT_t {
  static_assert(T_BAUD <= TMP_HAL_UART_BR_115200, "Invalid Baudrate");
...};
```

Esta meta-função, emite o erro de compilação "*Invalid Baudrate*" se ao parâmetro de entrada de *template T_BAUD* for atribuído um número de *baud rate* inválido. Esse erro é emitido pela função *static_assert*.

Implementação da seleção do *flow control*:

A figura 5.25 apresenta o diagrama de classes da seleção do *flow control* no meio DMA.

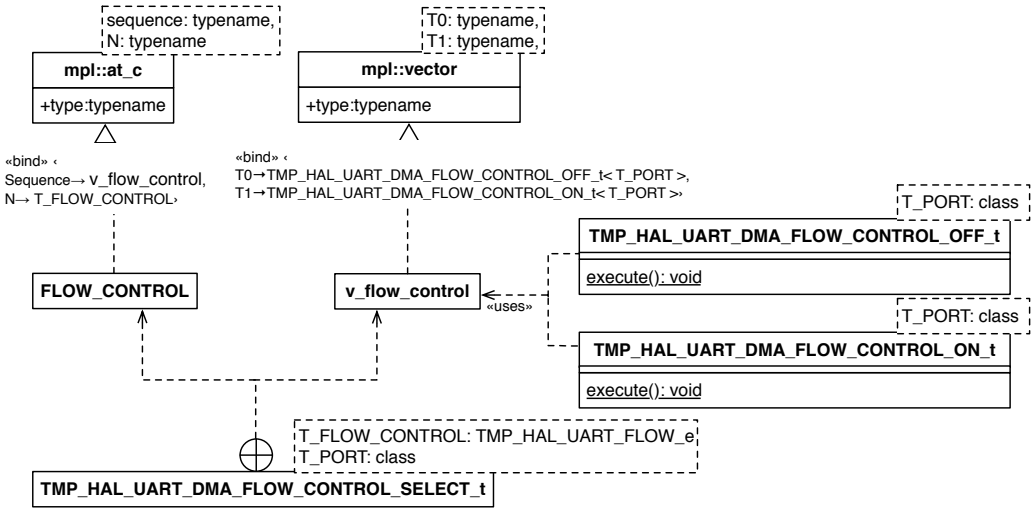


Figura 5.25: Diagrama de classes - Seleção do *flow control* no meio DMA

A meta-função *TMP_HAL_UART_DMA_FLOW_CONTROL_SELECT_t* permite selecionar o *flow control* a usar. Esta meta-função tem dois parâmetros de entrada de *template*: o parâmetro *T_FLOW_CONTROL* que permite definir se o *flow control* está ativo ou não e o parâmetro *T_PORT* que corresponde à estrutura que contém as configurações da porta série selecionada.

A implementação da meta-função é efetuada recorrendo à meta-função MPL *at_c* e a um vetor MPL com o nome *v_flow_control*. O vetor contém as duas estruturas que implementam o *flow control* ativo (*TMP_HAL_UART_DMA_FLOW_CONTROL_ON_t*) e o *flow control* inativo (*TMP_HAL_UART_DMA_FLOW_CONTROL_OFF_t*).

O código 5.34 demonstra parte da implementação das classes modeladas no diagrama de classes anterior.

Código 5.34: Seleção do *flow control* no meio DMA

```
template <class T_PORT>
struct TMP_HAL_UART_DMA_FLOW_CONTROL_ON_t{
  inline static void execute();
};

template <class T_PORT>
```

```
struct TMP_HAL_UART_DMA_FLOW_CONTROL_OFF_t{
  inline static void execute();
};

template <TMP_HAL_UART_FLOW_e T_FLOW_CONTROL, class T_PORT>
class TMP_HAL_UART_DMA_FLOW_CONTROL_SELECT_t {
  static_assert(T_FLOW_CONTROL <= TMP_HAL_UART_FLOW_ON, "Invalid flow
      control");
... };
```

Para cada uma das opções de escolha de *flow control* (ativo/inativo), existe uma estrutura de dados que contém um método de nome *execute*. Esse método, executa as ações necessárias para o tipo de *flow control* selecionado.

A meta-função *TMP_HAL_UART_DMA_FLOW_CONTROL_SELECT_t* emite o erro de compilação "*Invalid flow control*" caso o valor de seleção do *flow control*, fornecido no parâmetro de entrada de *template T_FLOW_CONTROL*, seja inválido.

Implementação da estrutura de configuração das portas série para o meio de comunicação por DMA:

A figura 5.26 contém o diagrama de classes referente à configuração da porta série para o meio de comunicação por DMA.

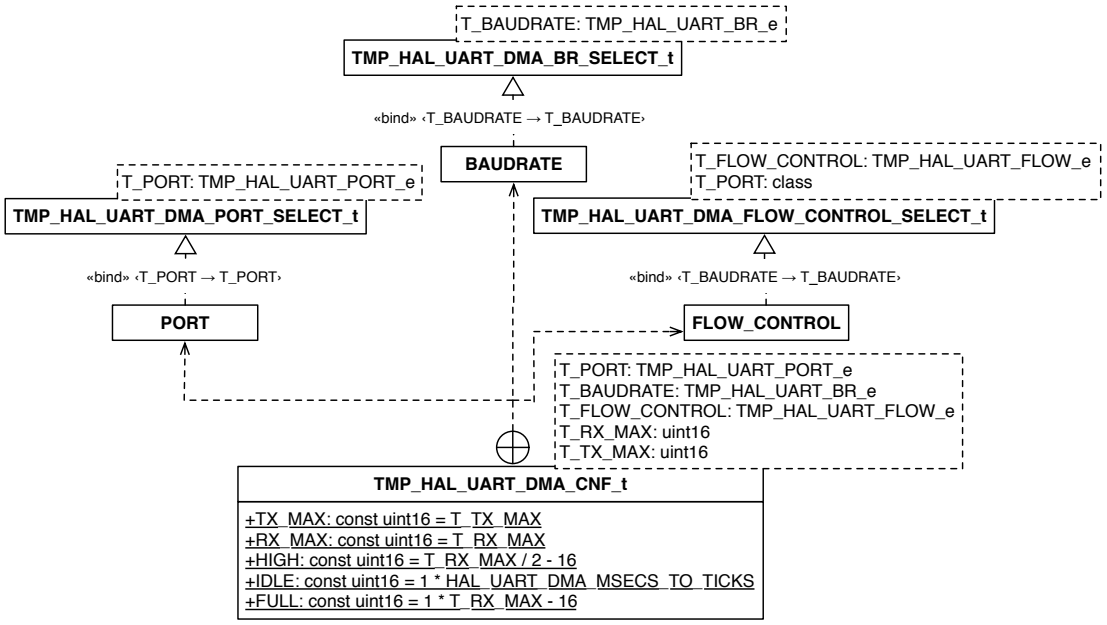


Figura 5.26: Diagrama de classes - Estrutura de configuração da porta série no meio DMA

A estrutura *TMP_HAL_UART_DMA_CNF_t*, permite configurar os parâmetros da porta série referentes ao meio de comunicação por DMA. Esses parâmetros são: escolha da porta série (*T_PORT*), escolha do *baud rate* (*T_BAUDRATE*), ativação e desativação do *flow control* (*T_FLOW_CONTROL*) e configuração do tamanho do *buffer* de leitura (*T_RX_MAX*) e de escrita (*T_TX_MAX*).

O código 5.35 é referente à implementação do diagrama de classes anterior.

Código 5.35: Implementação da configuração do módulo de gestão das portas série no meio DMA

```
template <  TMP_HAL_UART_PORT_e T_PORT,
            TMP_HAL_UART_BR_e T_BAUDRATE,
            TMP_HAL_UART_FLOW_e T_FLOW_CONTROL,
            uint16 T_RX_MAX,
            uint16 T_TX_MAX >
struct TMP_HAL_UART_DMA_CNF_t{
  struct PORT:TMP_HAL_UART_DMA_PORT_SELECT_t<T_PORT>::PORT{};

  struct BAUDRATE:
    TMP_HAL_UART_DMA_BR_SELECT_t<T_BAUDRATE>::BAUDRATE{};

  struct FLOW_CONTROL:
    TMP_HAL_UART_DMA_FLOW_CONTROL_SELECT_t
    <T_FLOW_CONTROL, PORT>::FLOW_CONTROL{};

  static const uint16 TX_MAX = T_TX_MAX;
  static const uint16 RX_MAX = T_RX_MAX;
  static const uint16 HIGH = T_RX_MAX / 2 - 16;
  static const uint16 IDLE = 1 * HAL_UART_DMA_MSECS_TO_TICKS;
  static const uint16 FULL = 1 * T_RX_MAX - 16;
};
```

A estrutura de configuração *TMP_HAL_UART_DMA_CNF_t* disponibiliza os elementos selecionados, pelos parâmetros de entrada de *template*, em *typenames*. Os parâmetros de entrada de *template* são: *typename PORT*, através da meta-função *TMP_HAL_UART_DMA_PORT_SELECT_t*; *typename BAUDRATE*, utilizando a meta-função *TMP_HAL_UART_DMA_BR_SELECT_t*; e por fim, o *typename* FLOW_CONTROL, recorrendo à utilização da meta-função *TMP_HAL_UART_DMA_FLOW_CONTROL_SELECT_t*. Além disso, esta estrutura disponibiliza os parâmetros de entrada que são numéricos nos atributos constantes: *TX_MAX*, *RX_MAX*, *HIGH*, *IDLE* e *FULL*.

Implementação do módulo de gestão das portas série para o meio de comunicação por DMA:

A figura 5.27 apresenta o diagrama de classes do módulo de gestão das portas

série, para o meio de comunicação por DMA.

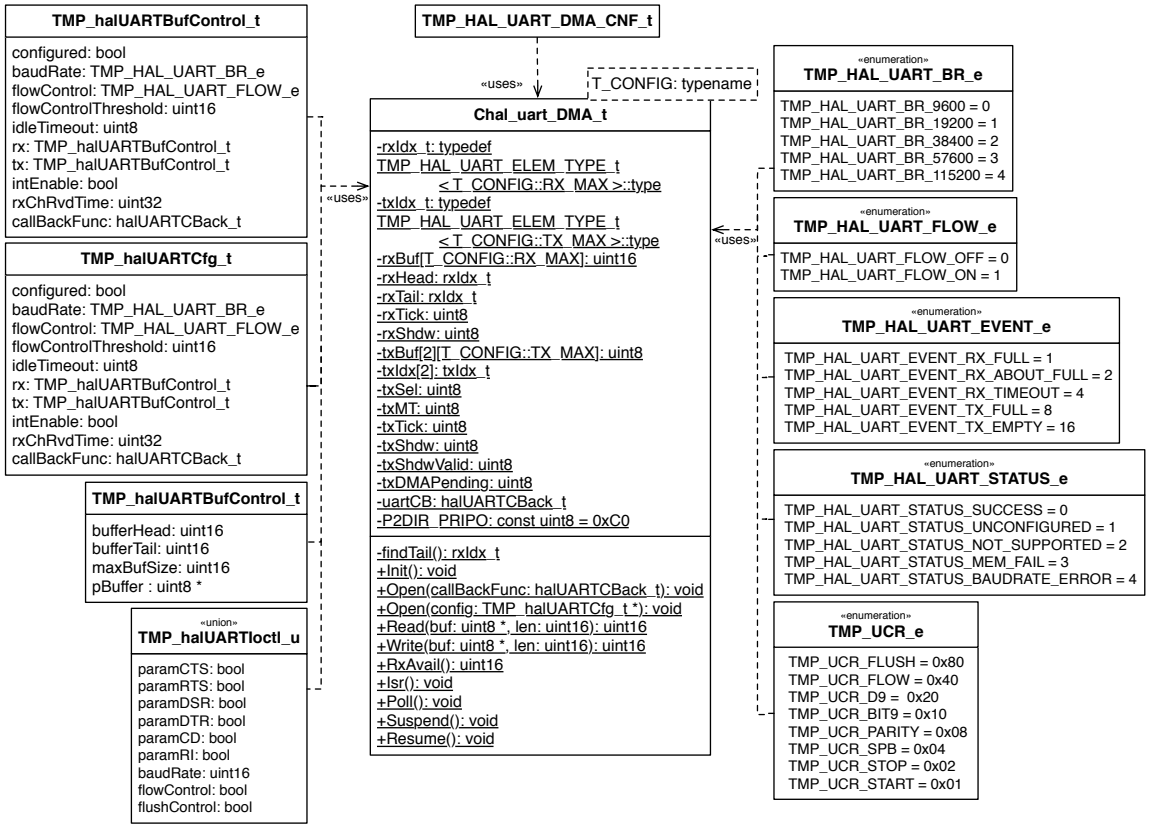


Figura 5.27: Diagrama de classes - Módulo de gestão das portas série no meio DMA

A classe *template Chal_uart_DMA_t* contém todos os métodos e atributos que implementam o módulo de gestão das portas série, para o meio de comunicação por DMA. Da lista de métodos que esta implementa, destacam-se a leitura, a escrita e a inicialização da porta série. A classe recebe como parâmetro de entrada de *template* uma estrutura de configuração *TMP_HAL_UART_DMA_CNF_t*, que contém as configurações a atribuir ao módulo definidas pelo utilizador.

Além disso, são também implementadas várias estruturas de dados e vários *enums* que dão apoio à classe *template Chal_uart_DMA_t*.

Configuração do módulo de gestão das portas série

Framework da *Texas Instruments*:

O código 5.36 apresenta como é configurado o módulo de gestão das portas série com a *Framework* da *Texas Instruments*.

Código 5.36: Configuração do módulo de gestão das portas série - *Framework* da *Texas Instruments*

```
#define HAL_UART_DMA = 1

halUARTCfg_t config;
config.baudRate              = HAL_UART_BR_115200;
config.flowControl           = HAL_UART_FLOW_OFF;

HalUARTOpenDMA(HAL_UART_PORT_0, &config);
```

Para ativar o módulo de gestão das portas série com a *Framework* da *Texas Instruments* é definida a macro de pré-processamento referente ao meio de comunicação e ao número da porta a utilizar. Neste exemplo, é ativada a porta série zero no meio de comunicação DMA, sendo definida a macro *HAL_UART_DMA* com o valor um (o valor um corresponde à porta série zero).

Por outro lado, a configuração dos parâmetros da porta série é efetuada em tempo de execução a partir da estrutura de dados *halUARTCfg_t*. Para isso, é definida uma variável com o tipo da estrutura de dados (*halUARTCfg_t config*), sendo esta preenchida com as configurações necessárias. No final, é chamada a função *HalUARTOpenDMA*, que recebe dois parâmetros de entrada: o número da porta a configurar, neste caso a porta série zero (HAL_UART_PORT_0) e o endereço da variável definida anteriormente (*&config*). A implementação da função *HalUARTOpenDMA* está presente no código 5.37.

Código 5.37: Validação de erros do módulo de gestão das portas série - *Framework* da *Texas Instruments*

```
static void HalUARTOpenDMA(halUARTCfg_t *config){
...
  if (config->baudRate == HAL_UART_BR_57600 ||
      config->baudRate == HAL_UART_BR_115200){
    UxBAUD = 216;
  }else{
    UxBAUD = 59;
  }

  switch (config->baudRate){
    case HAL_UART_BR_9600:
      UxGCR = 8;
      dmaCfg.txTick = 35;
      break;
    case HAL_UART_BR_19200:
      UxGCR = 9;
      dmaCfg.txTick = 18;
      break;
    case HAL_UART_BR_38400:
```

```
            UxGCR = 10;
            dmaCfg.txTick = 9;
            break;
          case HAL_UART_BR_57600:
            UxGCR = 10;
            dmaCfg.txTick = 6;
            break;
          default:
            // HAL_UART_BR_115200
            UxGCR = 11;
            dmaCfg.txTick = 3;
            break;
        }
    ...
    }
```

A função *HalUARTOpenDMA* efetua a seleção dos *baud rates* em tempo de execução, recorrendo aos *statements switch case* e *if* para selecionar a configuração das várias opções de *baud rate*. Nesta implementação, a validação que verifica se o *baud rate* é inválido é também implementada em tempo de execução. E caso o *baud rate* seja inválido, apenas se limita à atribuição de um *baud rate* por defeito, neste caso 115200bps.

***Framework* Generativa:**

O código 5.38, apresenta como é configurado o módulo de gestão das portas série com a *Framework* Generativa.

Código 5.38: Configuração do módulo de gestão das portas série - *Framework* Generativa

```
typedef TMP_HAL_UART_CNF_t<TMP_HAL_UART_MEDIUM_DMA,
        TMP_HAL_UART_PORT_0,
        TMP_HAL_UART_BR_115200,
        TMP_HAL_UART_FLOW_OFF > UART_CNF;
typedef Chal_uart_t< UART_CNF > Chal_uart;

TMP_CONFIG::Chal_uart::Open();
```

Na *Framework* Generativa, a configuração é efetuada através da utilização da estrutura de configuração *TMP_HAL_UART_CNF_t*. Esta estrutura, recebe quatro parâmetros de entrada de *template* obrigatórios: o meio de comunicação, a porta série, o *baud rate* e o *flow control*. Neste caso específico, o meio de comunicação é o DMA (*TMP_HAL_UART_MEDIUM_DMA*), a porta série escolhida é a zero (*TMP_HAL_UART_PORT_0*), o *baud rate* definido é 115200bps (TMP_HAL_UART_BR_115200) e o *flow control* está definido como inativo (TMP_HAL_UART_FLOW_OFF).

A estrutura de configuração é atribuída como parâmetro de entrada, ao módulo que implementa a gestão da porta série. De seguida, é chamado o método *Open*, que efetua a configuração da porta série, segundo os parâmetros definidos na estrutura de configuração. Neste caso, o método *Open* não tem qualquer parâmetro de entrada, isto porque, os valores de configuração já foram atribuídos de forma estática (em tempo de compilação), à classe que implementa o módulo de gestão das portas série (*Chal_uart_t*). A implementação da função *Open* é apresentada no código 5.39.

Código 5.39: Validação de erros do módulo de gestão das portas série - *Framework* Generativa

```
static void Open(){
...
    T_CONFIG::PORT::UxBAUD::value = T_CONFIG::BAUDRATE::BAUD;
    T_CONFIG::PORT::UxGCR::value = T_CONFIG::BAUDRATE::GCR;
    txTick = T_CONFIG::BAUDRATE::TICK;
...
}
```

Nesta implementação, não é efetuada a escolha das opções do *baud rate* em tempo de execução, contrariamente ao que acontece na *Framework* da *Texas Instruments*. A escolha é efetuada em tempo de compilação, usando a estrutura de configuração *TMP_HAL_UART_CNF_t*. Desta forma, a função *Open* apenas se limita a atribuir aos registos as configurações.

Na *Framework* Generativa, os parâmetros são todos validados em tempo de compilação, tal como mostrado anteriormente, durante a descrição da implementação do módulo. Desta forma, evita que hajam erros em tempo de execução, associados a configurações inválidas.

5.2.8 Ativação e Inicialização dos *Drivers*

***Framework* da *Texas Instruments*:**

O código 5.40 apresenta como é efetuada a ativação dos *drivers*, através da *Framework* da *Texas Instruments*.

Código 5.40: Ativação dos *drivers* - *Framework* da *Texas Instruments*

```
HAL_LED= TRUE
HAL_UART= TRUE
HAL_DMA = TRUE
HAL_ADC = TRUE
```

A ativação dos *drivers*, é efetuada através da definição de macros de pré-processamento. Para cada um dos *drivers* a ativar, é definida a macro correspondente a esse *driver*. As macros são depois utilizadas na função de inicialização dos *drivers*, para que esta só inicie os *drivers* ativados. A implementação da função que inicia os *drivers* está presente no código 5.41.

Código 5.41: Inicialização dos *drivers* - *Framework* da *Texas Instruments*

```
void HalDriverInit(void)
{
/* LED */
#if (defined HAL_LED) && (HAL_LED == TRUE)
  HalLedInit();
#endif

/* UART */
#if (defined HAL_UART) && (HAL_UART == TRUE)
  HalUARTInit();
#endif

/* DMA */
#if (defined HAL_DMA) && (HAL_DMA == TRUE)
  // Must be called before the init call to any module that uses DMA.
  HalDmaInit();
#endif

/* ADC */
#if (defined HAL_ADC) && (HAL_ADC == TRUE)
  HalAdcInit();
#endif
...
}
```

A função que inicia os *drivers* é a *HalDriverInit*. Esta função verifica, através de condições *#if* do pré-processador, se a macro referente ao *driver* foi definida. Em caso afirmativo, é apresentado ao compilador, o código de chamada da função *init* da *driver*.

***Framework* Generativa:**

O código 5.42 apresenta como é efetuada a configuração e ativação dos módulos que implementam os *drivers* da camada HAL com a *Framework* Generativa.

Código 5.42: Ativação dos *drivers* - *Framework* Generativa

```
struct TMP_CONFIG
{
/* Configuração do módulo de gestão dos LEDs */
  typedef mpl::vector< TMP_LED1, TMP_LED2, TMP_LED3, TMP_LED4 >
      V_HAL_LEDS;
```

```
    typedef TMP_HAL_LEDS_CNF_t<TMP_HAL_LED_BLINK_ON, V_HAL_LEDS >
        LEDS_CNF;
    typedef Chal_led_t< LEDS_CNF > Chal_led;

  /* Configuração do módulo de gestão das portas série */
    typedef TMP_HAL_UART_CNF_t< TMP_HAL_UART_MEDIUM_DMA,
                                TMP_HAL_UART_PORT_0,
                                TMP_HAL_UART_BR_115200,
                                TMP_HAL_UART_FLOW_OFF>UART_CNF;
    typedef Chal_uart_t< UART_CNF > Chal_uart;

  /* Configuração do módulo de gestão das Interrupções */
    typedef TMP_HAL_INTERRUPT_CNF_t<TMP_HAL_INTERRUPT_8, Chal_DMA_t >
        INTERRUPT_8_CNF;
    typedef mpl::vector< INTERRUPT_8_CNF > V_INTERRUPTS;

  /* Configuração do módulo de gestão do DMA */
    typedef Chal_DMA_t Chal_DMA;
    typedef Chal_DMA_channel_t< Chal_DMA_t::CHANNEL< HAL_DMA_CH_TX >,
        Chal_uart > uart_channel;
    typedef mpl::vector< uart_channel > V_CHANNELS;

  /* Configuração do módulo de gestão do ADC */
    typedef TMP_HAL_ADC_CNF_t< > ADC_CNF;
    typedef Chal_ADC_t< ADC_CNF > Chal_ADC;

  /* Ativação dos módulos */
    typedef mpl::vector< Chal_led, Chal_uart, Chal_DMA, Chal_ADC>
        V_HAL_DRIVERS;
};
```

Todas as configurações da camada HAL são definidas dentro da estrutura de dados *TMP_CONFIG*. Para cada um dos módulos, e tal como foi demonstrado anteriormente, existe uma estrutura de configuração, que é utilizada para configurar o módulo. De salientar que a configuração é toda efetuada em tempo de compilação.

Para cada uma das classes que implementam os módulos, é passada por parâmetro de entrada de *template* a estrutura de configuração. O *typedef* é utilizado para atribuir um nome ao módulo com essa configuração. Após a configuração de todos os módulos, estes são adicionados a um vetor MPL de nome *V_HAL_DRIVERS*. Este vetor irá conter todos os módulos que se pretendem inicializar, sendo utilizado pela função que faz a inicialização dos módulos. A implementação dessa função está presente no código 5.43.

Código 5.43: Inicialização dos *drivers* - *Framework* Generativa

```
struct driver_init
{
  template< typename U >
  void operator()(U x)
  {
```

```
        U::Init();
    }
};

struct HalDrivers
{
  #pragma inline=forced
  template <typename Vector>
  static inline void InitAll()
  {
    mpl::for_each<Vector>( driver_init() );
  }
};

void HalDriverInit (void)
{
    HalDrivers::InitAll<TMP_CONFIG::V_HAL_DRIVERS>();
}
```

A função que inicializa os módulos é a *HalDriverInit*, a qual chama o método *InitAll* da estrutura *HalDrivers*. O método *InitAll* é uma meta-função que recebe como parâmetro de entrada de *template*, o vetor de MPL *V_HAL_DRIVERS* contido na estrutura *TMP_CONFIG*. O método *InitAll*, utiliza o *for-each* do MPL para percorrer o vetor *V_HAL_DRIVERS* e gerar o código da chamada de todas as funções de inicialização (*init*), pertencentes a cada um dos módulos.

5.3 Implementação da Camada OSAL

A camada OSAL (*Operating System Abstraction Layer*) é fornecida pela TIMAC da *Texas Instruments*, e tem a função de criar uma camada de abstração na utilização do sistema operativo. Esta camada de abstração, permite que a utilização das APIs de manipulação do sistema operativo, sejam transparentes, independentemente do sistema operativo utilizado.

Contudo, para permitir um grau de otimização maior, o foco desta dissertação não incidiu apenas em modificar a camada de abstração do sistema operativo, como também, em aplicar as técnicas de *template metaprogramming* ao próprio sistema operativo.

Para os microcontroladores de poucos recursos, como é o caso do microcontrolador presente no *SOC CC2530*, a *Texas Instruments* utiliza um sistema operativo minimalista orientado a eventos. Nesta dissertação foram organizados os vários componentes do sistema operativo em classes C++ e, seguidamente, aplicadas técnicas de *template metaprogramming*, para gerir a variabilidade do sistema operativo.

Os componentes do sistema operativo que foram modificados são: Mensagens (secção 5.3.1), que implementa uma lista de mensagens para comunicação entre tarefas do OS; Temporizadores (secção 5.3.2), que possibilita a criação de temporizadores; Tarefas (secção 5.3.3), que permite a criação de tarefas; OSAL, que implementa a gestão do sistema operativo. Por fim, é demonstrado como é configurado o módulo OSAL (5.3.5).

Diagrama de funcionalidades:

A figura 5.28 apresenta o diagrama de funcionalidades da camada OSAL.

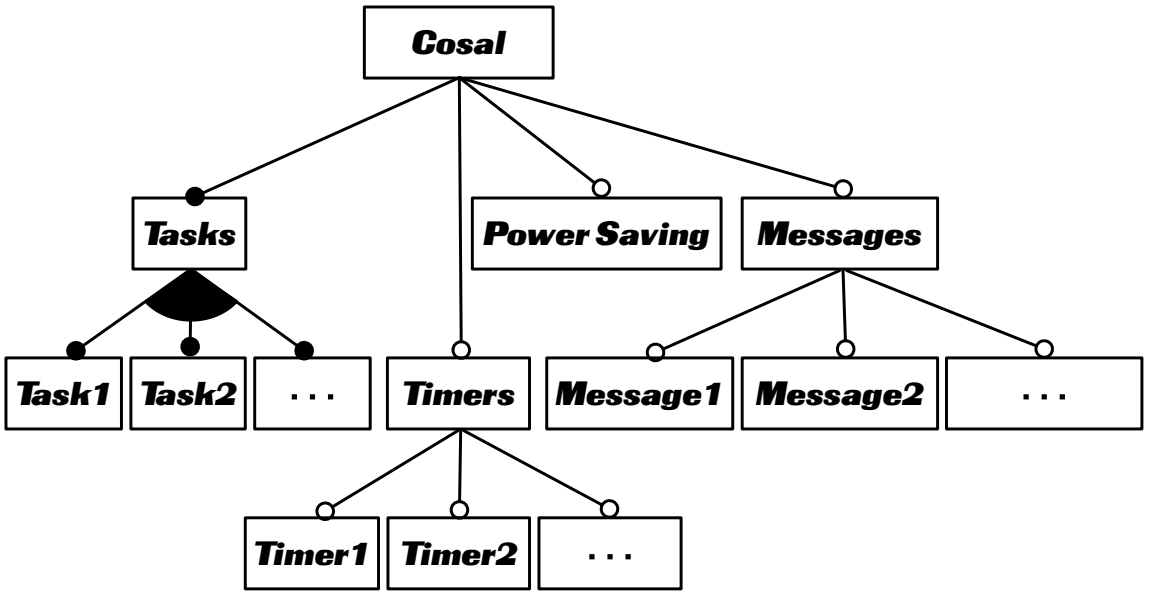


Figura 5.28: Diagrama de funcionalidades - Camada OSAL

A camada OSAL é representada pelo módulo *Cosal*. Neste módulo estão presentes quatro funcionalidades: as tarefas (*Tasks*), os temporizadores (*Timers*), o modo de *Power Saving* e as mensagens (*Messages*).

As tarefas (*Tasks*) são apresentadas como funcionalidades acumulativas, visto que pelo menos uma tarefa tem que ser criada, mas podem ser criadas mais que uma.

Tanto as mensagens (*Messages*) como os temporizadores (*Timers*) são apresentados como funcionalidades opcionais, isto é, podem ou não fazer parte das configurações do sistema.

A opção de *Power Saving* é apresentada como uma funcionalidade opcional, porque o modo de *power saving* pode ser ativado ou não.

5.3.1 Mensagens

As mensagens são um meio de sincronismo disponibilizado pelo sistema operativo, que permitem a troca de dados entre as várias tarefas. A figura 5.29 apresenta o diagrama de classes da implementação das mensagens.

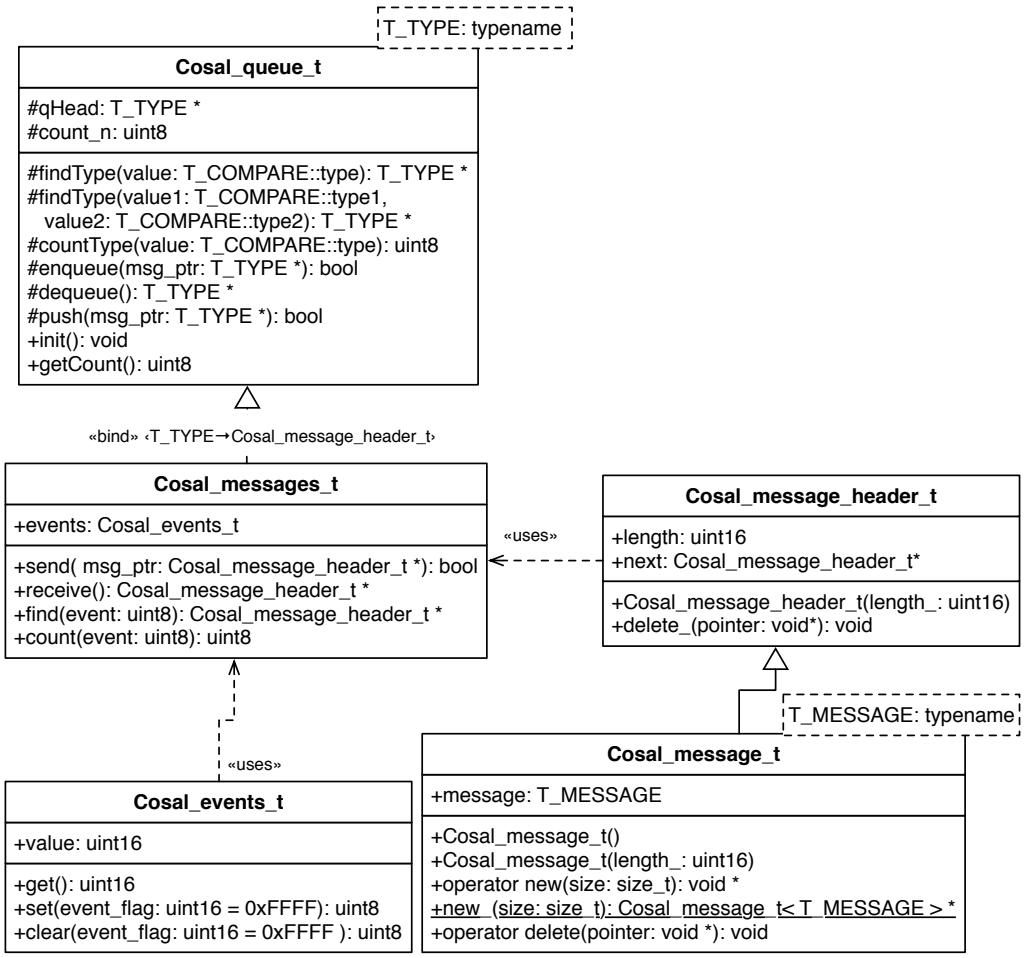


Figura 5.29: Diagrama de classes - Mensagens do OSAL

A lista de mensagens é implementada pela classe *Cosal_messages_t*, que fornece os seguintes métodos: envio de mensagens (*send*), receção de mensagens (*receive*), procura de mensagens (*find*) e contagem de mensagens (*count*).

A classe *Cosal_messages_t* herda os membros da classe *template* *Cosal_queue_t*, que implementa uma *queue* de dados. Na herança, é passada como parâmetro de entrada de *template* a estrutura *Cosal_message_header_t*. A estrutura

de dados *Cosal_message_header_t*, implementa um nó da lista ligada, que contém um atributo que aponta para o próximo nó da lista (*next*).

A mensagem é implementada pela estrutura de dados *Cosal_message_t*, que herda os membros da estrutura de dados *Cosal_message_header_t*, que implementa um nó da lista ligada. A estrutura de dados *Cosal_message_t* implementa uma mensagem genérica, e recebe como parâmetro de entrada de *template T_MESSAGE*, o tipo de mensagem específico que é pretendido inserir na lista de mensagens.

A classe que implementa a lista de mensagens (*Cosal_messages_t*), contém também o atributo *events*. Este atributo é do tipo *Cosal_events_t*, e implementa uma estrutura de dados com dezasseis *flags*, que permitem sinalizar os eventos que estão ativos.

5.3.2 Temporizadores

Os temporizadores são mecanismos disponibilizados pelo sistema operativo, que permitem definir a ativação de eventos, num intervalo de tempo predefinido pelo utilizador. Um temporizador é atribuído a uma única tarefa, e permite ativar a tarefa a qual foi atribuído, sempre que o valor de temporização chegue a zero. A figura 5.30 apresenta o diagrama de classes da implementação dos temporizadores.

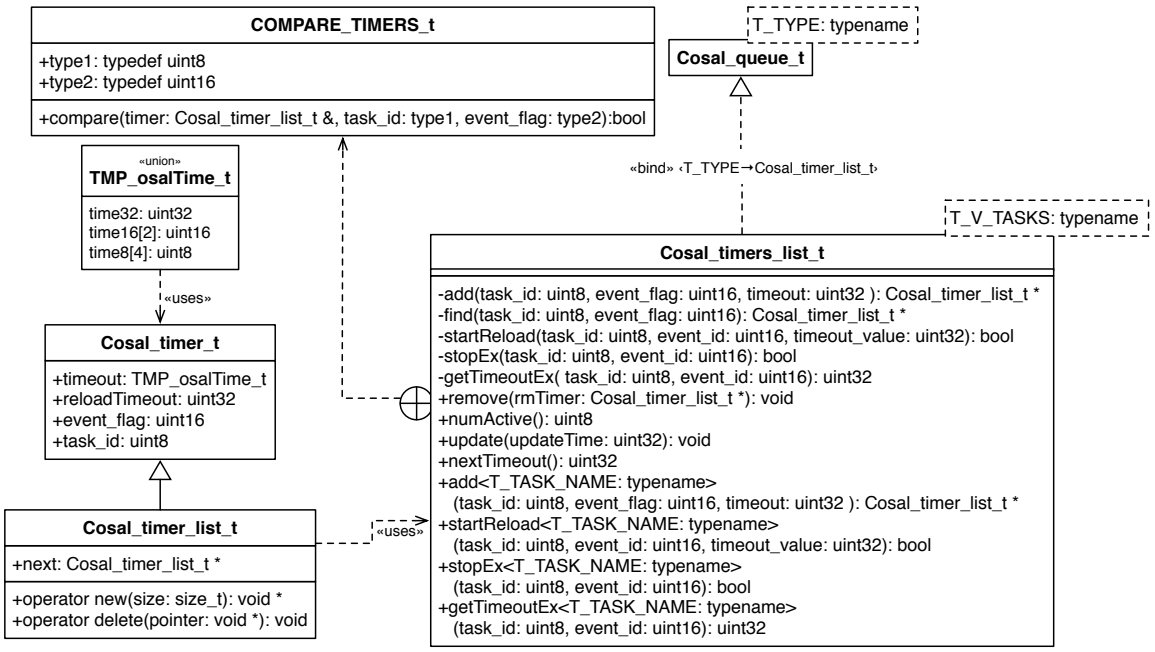


Figura 5.30: Diagrama de classes - Temporizadores do OSAL

O módulo que gere os temporizadores é implementado pela classe *Cosal_timers_list_t*, que contém métodos que permitem: adicionar um temporizador, inicializar um temporizador, parar um temporizador e remover um temporizador. Os vários temporizadores são armazenados numa lista ligada. Logo, a classe *Cosal_timers_list_t* herda os membros da classe que implementa uma *queue* (*Cosal_queue_t*), que é a mesma *queue* utilizada para as mensagens.

A classe *Cosal_timers_list_t* contém internamente a estrutura de dados *COMPARE_TIMERS_t*. A estrutura de dados *COMPARE_TIMERS_t*, é utilizada para ser fornecida como parâmetro de entrada de *template* às funções *template* da classe *Cosal_queue_t*, indicando qual a função de comparação. O código 5.44 apresenta a implementação da estrutura de dados *COMPARE_TIMERS_t*.

Código 5.44: Implementação da estrutura de comparação

```
struct COMPARE_TIMERS_t{
  typedef uint8 type1;
  typedef uint16 type2;
  inline static bool compare( Cosal_timer_list_t &timer,
                              type1 task_id,
                              type2 event_flag){
    return ( timer.event_flag==event_flag &&
             timer.task_id==task_id );
  }
};

Cosal_timer_list_t *find( uint8 task_id, uint16 event_flag ){
    return findType<COMPARE_TIMERS_t>(task_id, event_flag);
}
```

A estrutura *COMPARE_TIMERS_t* define o método *compare*. Este método recebe três parâmetros de entrada: uma referência para um temporizador (*Cosal_timer_list_t &timer*), o id da tarefa (*type1 task_id*) e uma *flag* de evento (*type2 event_flag*). O método *compare* retorna o resultado da comparação entre os valores dos dois últimos parâmetros de entrada (*task_id* e *event_flag*) e os valores presentes no temporizador *timer*.

O método *find* pertence a classe *Cosal_timers_list_t*, este método permite procurar se existe algum temporizador na lista de temporizadores, que coincida com o id da tarefa (*task_id*) e com a *flag* de eventos (*event_flag*). A procura é efetuada pelo método *findType* que pertence à *Cosal_queue_t*. Esse método é uma função *template* que recebe como parâmetros de entrada de função, a *task_id* e a *event_flag* e como parâmetro de entrada de *template* a estrutura *COMPARE_TIMERS_t*.

5.3.3 Tarefas

Uma das funções do sistema operativo é efetuar a gestão de múltiplas tarefas (*multi-task*). Desta forma, é possível dividir um programa em tarefas isoladas, que executam ações distintas entre si, permitindo assim, uma melhor organização do programa. A figura 5.31 apresenta o diagrama de classes da implementação das tarefas.

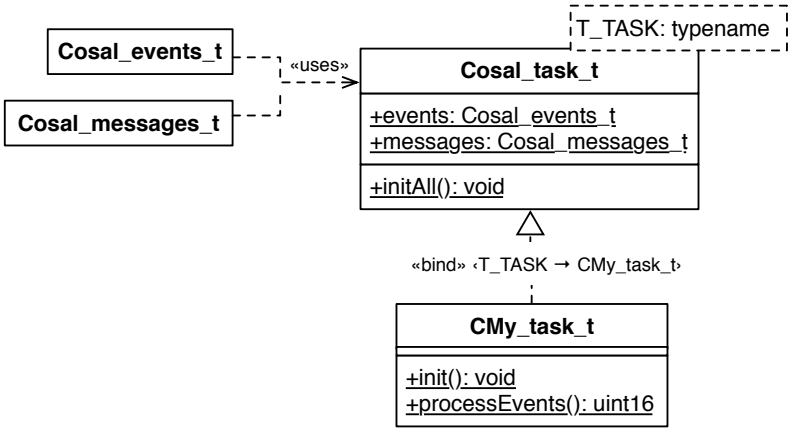


Figura 5.31: Diagrama de classes - Tarefas do OSAL

No diagrama de classes está representada a classe *template* *Cosal_task_t*. A classe implementa a base de todas as tarefas, ou seja, o que é comum às mesmas. Dessa forma, implementa o atributo *events*, que contém os eventos a processar pela tarefa, e o atributo *messages*, que contém a lista de mensagens recebidas pela tarefa. Para além dos atributos, também contém o método *initAll*, responsável pela inicialização da tarefa.

Todas as tarefas adicionadas pelo utilizador, são implementadas por uma classe que tem obrigatoriamente de herdar da classe *Cosal_task_t*, e na herança passa como atributo *template*, o nome da classe implementada. Outra exigência na implementação da classe que implementa a tarefa, é que esta tem que conter no mínimo dois métodos: o método *init* e o método *processEvents*. O método *init*, é responsável pela inicialização da tarefa, o qual é chamado pelo método *initAll* pertencente à classe *Cosal_task_t*, no momento da inicialização do sistema operativo. O método *processEvents*, é responsável pelo processamento dos eventos e mensagens da tarefa, o qual é chamado pelo gestor de tarefas sempre que existam eventos associados a essa tarefa

à espera de serem processados. No diagrama de classes está presente a implementação de uma tarefa de nome *CMy_task_t*. O código 5.45 mostra a criação da classe que implementa essa tarefa.

Código 5.45: Implementação de uma tarefa

```
class CMy_task_t : public Cosal_task_t<CMy_task_t> {
public:
  static void init();
  static uint16 processEvents();
};

void CMy_task_t::init(){
  //Inicialização da tarefa
}
uint16 CMy_task_t::processEvents(){
  if(events.get() & EVENT_MASK){
      //Ações de processamento do evento
  }
  return 0;
}
```

A tarefa *CMy_task_t* é implementada por uma classe com o mesmo nome. Esta classe, herda da classe *Cosal_task_t*, e durante a herança passa como atributo *template* à mesma a classe implementada (*CMy_task_t*).

A classe *CMy_task_t* implementa os dois métodos obrigatórios: o método *init*, onde é inserido código de inicialização da tarefa; e o método *processEvents*, que efetua o processamento dos eventos pertencentes a essa tarefa. O acesso aos eventos é efetuado pelo método *get* do atributo *events*.

5.3.4 OSAL

Implementação da estrutura de configuração do módulo de gestão da camada OSAL:

O módulo OSAL é configurado através de uma estrutura de configuração. A figura 5.32 apresenta como foi implementada essa estrutura.

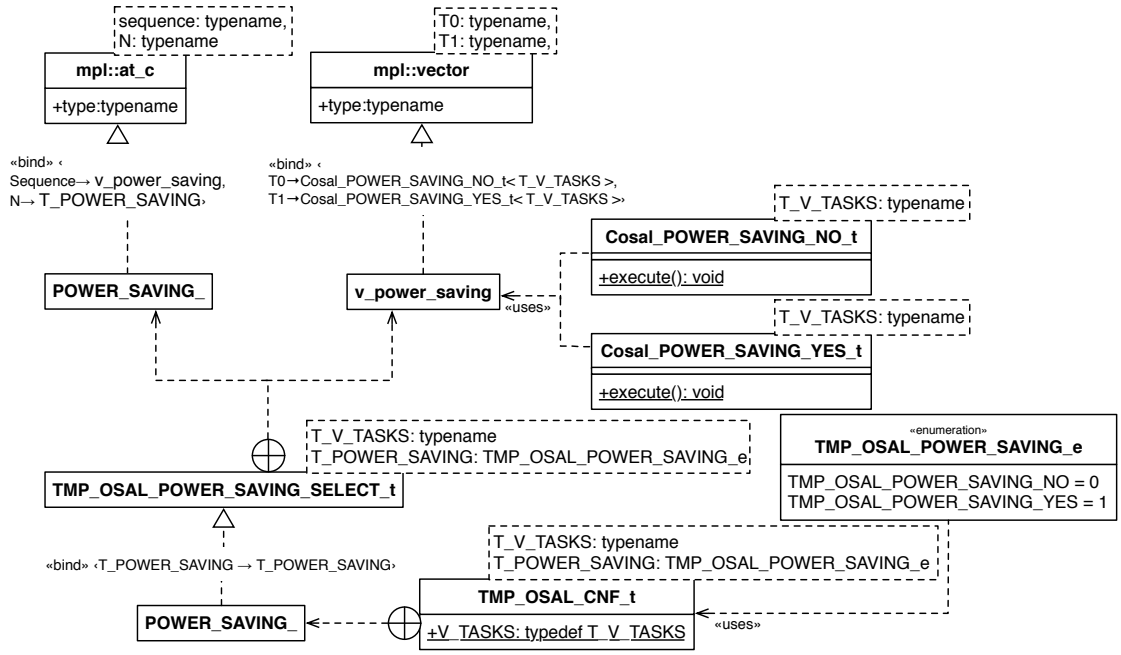


Figura 5.32: Diagrama de classes - Configuração do módulo OSAL

A estrutura de configuração é implementada pela classe *template* *TMP_OSAL_CNF_t*. A classe *template* recebe dois parâmetros de entrada de *template*: o parâmetro *T_V_TASKS*, que é um vetor MPL de tarefas, e o parâmetro *T_POWER_SAVING*, que permite definir se o modo de poupança de energia (*power saving*) está ativo ou não.

A seleção do modo de *Power Saving* é efetuada pela meta-função *TMP_OSAL_POWER_SAVING_SELECT_t*. Esta meta-função recorre à meta-função *at_c* e um vetor do MPL para fornecer a estrutura que implementa o modo selecionado. Caso o modo *Power Saving* esteja ativo, apresenta a estrutura *Cosal_POWER_SAVING_YES_t*, caso contrário, apresenta a estrutura *Cosal_POWER_SAVING_NO_t*. O código 5.46 apresenta a implementação da meta-função *TMP_OSAL_POWER_SAVING_SELECT_t*.

Código 5.46: Configuração do módulo OSAL

```
template < typename T_V_TASKS,
        TMP_OSAL_POWER_SAVING_e T_POWER_SAVING >
class TMP_OSAL_POWER_SAVING_SELECT_t {
  static_assert(T_POWER_SAVING <= TMP_OSAL_POWER_SAVING_YES, "Invalid
       Power Saving Mode");
};
```

A meta-função *TMP_OSAL_POWER_SAVING_SELECT_t* emite o erro de compilação "*Invalid Power Saving Mode*" caso seja chamada com uma configuração inválida no parâmetro de entrada *T_POWER_SAVING*.

Implementação do módulo de gestão da camada OSAL:

A figura 5.33 apresenta o diagrama de classes da implementação do módulo OSAL.

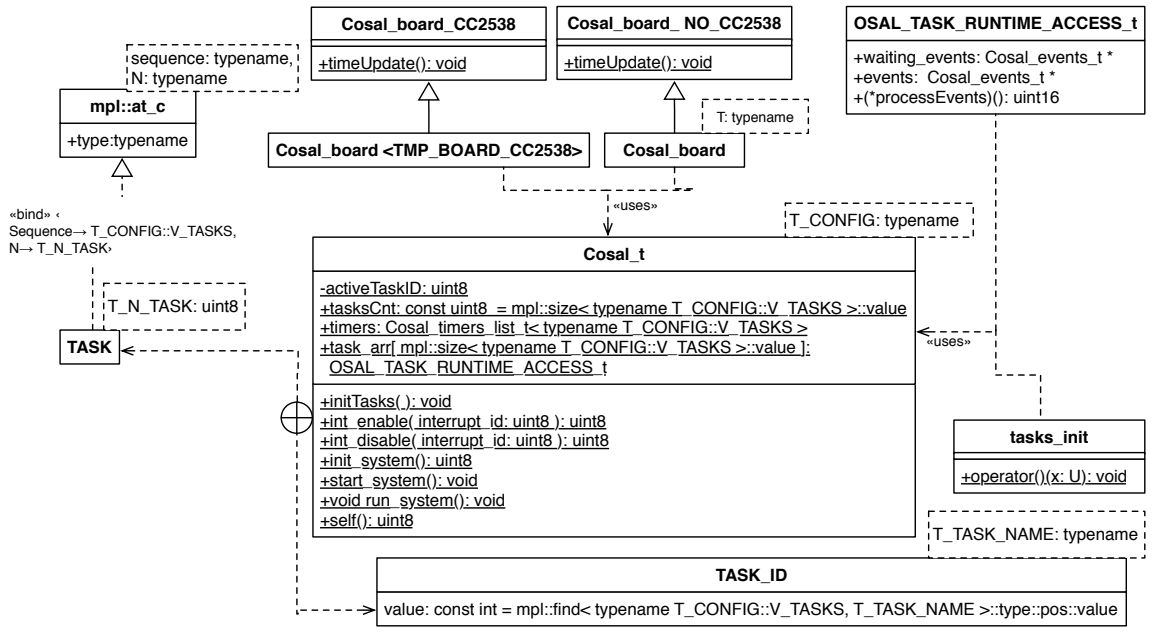


Figura 5.33: Diagrama de classes - Módulo OSAL

O módulo OSAL é implementado pela classe *Cosal_t*. Esta, é uma classe *template* que recebe como parâmetro de entrada, uma estrutura de configuração *TMP_OSAL_CNF_t*. A classe *Cosal_t* contém vários atributos: o atributo *activeTaskID*, que indica qual o id da tarefa em execução; o atributo *tasksCnt*, que indica o número de tarefas, o qual é obtido estaticamente (em tempo de compilação) através da meta-função *size* do MPL que faz a contagem do número de elementos do vetor *V_TASKS*; o atributo *timers*, que implementa o acesso aos temporizadores através da classe *Cosal_timers_list_t*; e por último, o atributo *task_arr*, que é um *array* do tipo *OSAL_TASK_RUNTIME_ACCESS_t*, de tamanho igual ao número de tarefas.

A estrutura de dados *OSAL_TASK_RUNTIME_ACCESS_t* implementa o acesso aos elementos de cada tarefa, que são necessários aceder pelo gestor de ta-

refas durante a execução do programa (tempo de execução). Esta estrutura contém três atributos: um apontador para os eventos da tarefa que estão em espera de ser processados (*waiting_events*), um apontador para os eventos que estão a ser processados atualmente (*events*) e, por último, um apontador para a função que processa os eventos de cada tarefa (*processEvents*).

Além dos atributos enumerados anteriormente, a classe *Cosal_t* contém também vários métodos: o método *initTasks*, que efetua a inicialização de todas as tarefas contidas no vetor MPL *V_TASKS*; o método *int_enable*, que ativa as interrupções do microcontrolador; o método *int_disable*, que desativa as interrupções do microcontrolador; o método *init_system*, que inicializa todo o sistema operativo; e por fim, os métodos *start_system* e *run_system*, que iniciam a execução das tarefas.

A classe *Cosal_t* implementa duas meta-funções: a meta-função *TASK_ID* e a meta-função *TASK*. A meta-função *TASK_ID* recebe como parâmetro de entrada de *template* o nome da tarefa (*T_TASK_NAME*), e devolve o seu ID. A meta-função *TASK*, faz o inverso, ou seja, recebe como parâmetro de entrada de *template* o ID da tarefa (*T_N_TASK*) e devolve a estrutura de acesso à tarefa com esse ID.

O código 5.47 apresenta como são inicializadas as tarefas do OSAL.

Código 5.47: Inicialização das tarefas do OSAL

```
struct tasks_init
{
  template< typename U >
  inline void operator()(U x)
  {
      U::initAll();
  }
};

inline static void initTasks( void )
{
  mpl::for_each< T_CONFIG::V_TASKS >( tasks_init() );
}
```

A inicialização das tarefas é implementada pelo método *initTasks*, pertencente à classe *Cosal_t*. Este método, utiliza o *for_each* do MPL para percorrer o vetor MPL de tarefas *V_TASKS*, e gerar a chamada da função *initAll*, pertencente a cada tarefa. A chamada às funções *initAll* é implementada pelo *functor tasks_init*.

5.3.5 Configuração do OSAL

***Framework* da *Texas Instruments*:**

O código 5.48 apresenta a configuração do OSAL com a *Framework* da *Texas Instruments*.

Código 5.48: Configuração do OSAL - *Framework* da *Texas Instruments*

```
#define POWER_SAVING

const pTaskEventHandlerFn tasksArr[] ={
  macEventLoop,
  MSA_ProcessEvent,
  My_task_ProcessEvent,
  Hal_ProcessEvent,
  ADC_task_ProcessEvent
};

void osalInitTasks( void ){
  uint8 taskID = 0;

  tasksEvents= (uint16*)osal_mem_alloc(sizeof(uint16)*tasksCnt);
  osal_memset(tasksEvents, 0, (sizeof(uint16)* tasksCnt));

  macTaskInit( taskID++ );
  MSA_Init( taskID++ );
  My_task_Init( taskID++ );
  Hal_Init( taskID++ );
  ADC_task_Init( taskID++ );
  My_task_Init( taskID );
}
```

Para ativar o modo de *power saving*, é definida a macro de pré-processamento *POWER_SAVING* (*#define POWER_SAVING*).

Por cada uma das tarefas implementadas têm que ser efetuados dois passos: primeiro, tem que ser adicionado ao *array tasksArr*, o apontador para a função que implementa o processamento de eventos para essa tarefa e por fim, também por cada uma das tarefas, tem que ser adicionada a função *init* de cada tarefa à função *osalInitTasks*.

***Framework* Generativa:**

Na configuração do OSAL foi acrescentado mais um nível de abstração em relação à configuração da camada HAL. Enquanto na camada HAL a configuração é efetuada diretamente em código C++, na camada OSAL, foi adicionada a linguagem de marcação XML para definir as configurações dessa camada.

Foi escolhida a linguagem XML, devido a ser um *standard* bastante utilizado. Isto leva a que exista no mercado um vasto leque de ferramentas de edição de linguagem XML, que podem ser utilizadas para adicionar à *framework* um nível de abstração ainda mais elevado que o existente, implementando, por exemplo, uma interface gráfica para configuração da *framework*.

O ficheiro de configuração XML da camada OSAL tem a estrutura apresentada no código 5.49.

Código 5.49: Configuração do OSAL - Ficheiro XML - *Framework* Generativa

```
<?xml version="1.0" encoding="UTF-8"?>
<osal power_saving="YES">
  <tasks>
    <task name="CmacEventLoop_task"/>
    <task name="CMSA_task"/>
    <task name="CMy_task"/>
    <task name="CHal_task"/>
    <task name="CADC_task"/>
  </tasks>
</osal>
```

As configurações do OSAL são definidas na *tag* principal *osal*, que define o atributo *power_saving*, que permite definir se o modo de *power saving* está ativo ou não. Neste caso específico, foi definido como ativo, ou seja, *power_saving="YES"*. Internamente à *tag osal* tem que ser definida a *tag tasks*, e internamente a essa *tag* é que são definidas as *tags* de definição das tarefas. Cada tarefa é definida através de uma *tag task*, a qual define o atributo *name* que define o nome da tarefa.

Para converter o formato XML para a linguagem C++ recorreu-se à linguagem de transformação XSLT. Esta linguagem define ações de transformação, que permitem gerar o formato de dados desejado, a partir dos dados lidos de um ficheiro XML, com extensão *.xml*. Para cada um dos ficheiros a gerar, é definido um ficheiro de extensão *.xsl* que implementa, através de XSLT, as ações de transformação aplicadas ao ficheiro XML para gerar o ficheiro desejado. No final, os ficheiros de extensão *.xml* e *.xsl* são processados por um programa processador de XLST, que gera um ficheiro com o formato definido no ficheiro de extensão *.xsl*, com os dados obtidos do ficheiro de extensão *.xml*. O processador XSLT utilizado foi o MSXSL, que é um processador implementado pela *Microsoft* [43].

O ficheiro de configuração XML mostrado anteriormente dá origem a dois ficheiros C++ gerados através do seu ficheiro XSLT correspondente (os códigos XSLT referentes aos dois ficheiros estão presentes no apêndice A). O primeiro ficheiro con-

tém a configuração do módulo OSAL, e o seu conteúdo é apresentado no código 5.50.

Código 5.50: Configuração do OSAL - Estrutura de configuração - *Framework* Generativa

```
#ifndef __COSAL_CONFIG_H__
#define __COSAL_CONFIG_H__

#include "containers/vector.hpp"

#include "CmacEventLoop_task.hpp"
#include "CMSA_task.hpp"
#include "CMy_task.hpp"
#include "CHal_task.hpp"
#include "CADC_task.hpp"

struct TMP_OSAL_CONFIG {
  typedef mpl::vector< CmacEventLoop_task_t, CMSA_task_t, CMy_task_t,
      CHal_task_t, CADC_task_t> V_OSAL_TASKS;
  typedef TMP_OSAL_CNF_t<V_OSAL_TASKS, TMP_OSAL_POWER_SAVING_YES>
     OSAL_CNF;
  typedef Cosal_t< OSAL_CNF > Cosal;
};

#endif /* __COSAL_CONFIG_H__ */
```

O ficheiro de configuração gerado contém no seu início a inclusão de todos os ficheiros *header file* (*.hpp*) que possuem o protótipo das classes que implementam cada tarefa.

Após isso, o ficheiro contém a estrutura de dados *TMP_OSAL_CONFIG*, que internamente, possui a configuração do módulo OSAL. Primeiramente, cria um vetor MPL constituído pelas classes que implementam as tarefas. De seguida, e recorrendo à estrutura de configuração *TMP_OSAL_CNF_t*, atribui ao primeiro parâmetro de entrada de *template* dessa estrutura, o vetor MPL criado anteriormente e ao segundo parâmetro de entrada, o enumerado que seleciona o modo *power saving* como ativo (*TMP_OSAL_POWER_SAVING_YES*). No final, configura o módulo OSAL através da estrutura de configuração criada anteriormente.

O segundo ficheiro é respeitante ao preenchimento do *array*, que contém as informações das tarefas que precisam de ser acedidas, pelo gestor de tarefas, durante a execução do programa (tempo de execução). O código 5.51 apresenta o código presente nesse ficheiro.

Código 5.51: Configuração do OSAL - *Array* de tarefas - *Framework* Generativa

```
#ifndef __COSAL_TASKS_INIT_H__
```

```
#define __COSAL_TASKS_INIT_H__

template < typename T_CONFIG >
OSAL_TASK_RUNTIME_ACCESS_t Cosal_t< T_CONFIG >::task_arr[ mpl::size<
    typename T_CONFIG::V_TASKS >::value ] = {
  {&T_CONFIG::V_TASKS::item0::messages.events,
     &T_CONFIG::V_TASKS::item0::events,
     T_CONFIG::V_TASKS::item0::processEvents },
  {&T_CONFIG::V_TASKS::item1::messages.events,
     &T_CONFIG::V_TASKS::item1::events,
     T_CONFIG::V_TASKS::item1::processEvents },
  {&T_CONFIG::V_TASKS::item2::messages.events,
     &T_CONFIG::V_TASKS::item2::events,
     T_CONFIG::V_TASKS::item2::processEvents },
  {&T_CONFIG::V_TASKS::item3::messages.events,
     &T_CONFIG::V_TASKS::item3::events,
     T_CONFIG::V_TASKS::item3::processEvents },
  {&T_CONFIG::V_TASKS::item4::messages.events,
     &T_CONFIG::V_TASKS::item4::events,
     T_CONFIG::V_TASKS::item4::processEvents }
};

#endif /* __COSAL_TASKS_INIT_H__ */
```

O ficheiro gerado contém a inicialização do *array task_arr* pertencente ao módulo OSAL. Este *array* possui, por cada uma das tarefas, os apontadores para os elementos que o gestor de tarefas necessita de aceder durante a execução do programa (tempo de execução). Esses elementos são: o apontador para os eventos à espera de serem processados pela tarefa (*&T_CONFIG::V_TASKS::item0::messages.events*), o apontador para os eventos que estão a ser processados atualmente pela tarefa (*&T_CONFIG::V_TASKS::item0::events*) e, por último, o apontador para o método que processa os eventos da tarefa (*T_CONFIG::V_TASKS::item0::processEvents*).

5.4 Conclusões

Este capítulo, apresentou como foi modelada e implementada a *framework* generativa. A implementação da mesma, teve como base a biblioteca TMP *Boost.MPL*. Como essa biblioteca não era compatível com o compilador utilizado (IAR para 8051), tiveram que ser exportadas as funcionalidades necessárias da mesma, de forma a se tornarem compatíveis, e assim, poderem ser utilizadas na implementação da *framework*.

Para modelar a *framework*, recorreu-se a diagramas de funcionalidade que demonstram o grau e a variabilidade de cada uma das partes do sistema, seguida da apresentação dos diagramas de classes da implementação que permite gerir essa varia-

bilidade, através de TMP. Para além dos diagramas, também é demonstrado algumas partes do código da implementação, de forma a clarificar alguns detalhes de implementação que não são visíveis nos diagramas de classes.

A *framework* implementada incidiu sobre a camada HAL e OSAL da TIMAC da *Texas Instruments*, e consistiu na conversão das bibliotecas, constituintes dessas camadas, de linguagem C para classes em linguagem C++, seguida da aplicação de técnicas de *template metaprogramming*, para gerir a variabilidade das funcionalidades presentes em cada uma das camadas, HAL e OSAL.

Capítulo 6

Resultados Experimentais

Esta dissertação apresentou uma alternativa à forma de como é feita a gestão da variabilidade do código da pilha de *software* TIMAC desenvolvida pela *Texas Instruments*, mais propriamente a gestão da sua camada HAL e OSAL. A *Texas Instruments* utilizou a linguagem C com compilação condicional para gerir essa variabilidade, enquanto nesta dissertação foi apresentada, como alternativa, a utilização da linguagem C++ com *template metaprogramming*.

Neste capítulo são apresentados os resultados experimentais das comparações efetuadas entre as duas implementações. As comparações dos resultados experimentais, basearam-se em dois tipos de métricas: métricas de gestão do código e métricas de desempenho do código. As métricas de gestão do código avaliam o número de linhas de código e de módulos de cada implementação. E as métricas de desempenho do código comparam as duas implementações ao nível do consumo de memória de código, consumo de memória de dados e dos tempos de execução.

6.1 Ambiente de Testes

Designa-se ambiente de testes, a plataforma sobre o qual decorreram os testes ao sistema implementado. A plataforma divide-se em dois grupos: o *hardware* e o *software*. O *hardware* indica os componentes físicos utilizados na elaboração dos testes. E o *software* indica os programas utilizados na conceção do código e nos testes do mesmo. As subsecções seguintes apresentam a descrição dos dois grupos enumerados anteriormente.

6.1.1 *Hardware*:

O *hardware* é composto por todos os componentes físicos utilizados na elaboração dos testes.

Placa de desenvolvimento:

Os testes foram realizados na placa de desenvolvimento CC2530DK, desenvolvida por elementos pertencentes ao grupo de investigação à qual esta dissertação está inserida, o ESRG (*Embedded Systems Research Group*). A tabela 6.1 apresenta a listagem dos componentes dessa placa.

Tabela 6.1: Componentes de *hardware* da placa de desenvolvimento CC2530DK

Componente	Referência
Módulo de comunicação	CC2530EM
SOC	CC2530F256
Microcontrolador	8051

A placa é composta por vários elementos físicos que facilitam a prototipagem de sistemas, dos quais se destacam: o interface USB com o computador, alguns botões e LEDs de utilizador e o barramento de acesso aos portos e pinos de interface. A figura 6.1 apresenta a imagem da placa de desenvolvimento CC2530DK.

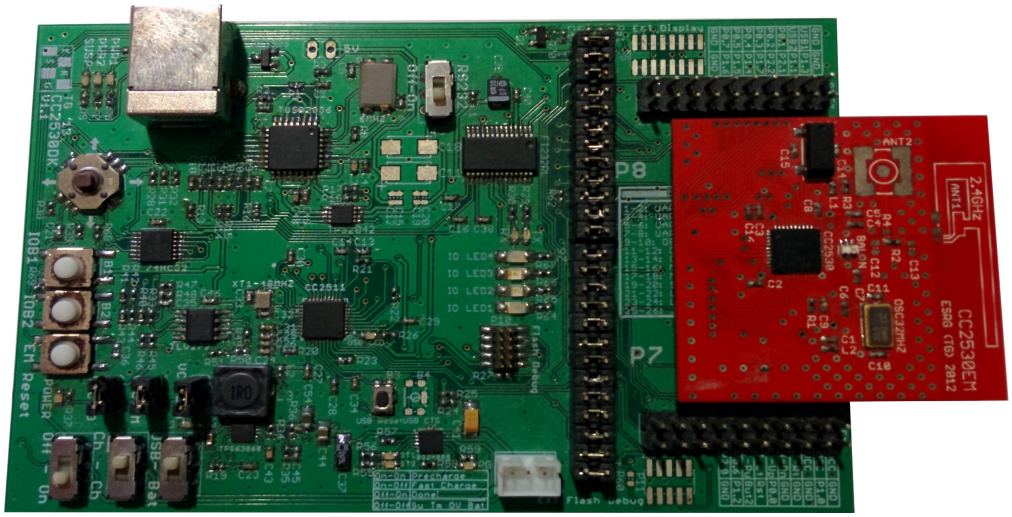


Figura 6.1: Placa de desenvolvimento CC2530DK

Além disso, a placa suporta vários módulos de comunicação, sendo apenas utilizado o módulo de comunicação CC2530EM, também este desenvolvido pelo ESRG. Este módulo possui integrado no mesmo os elementos necessários para a comunicação, tais como: a antena de comunicação e o SOC (*System On Chip*) de comunicação CC2530F256, que contém internamente um microcontrolador 8051. A tabela 6.2 apresenta um resumo das características desse microcontrolador.

Tabela 6.2: Características do microcontrolador

Característica		**Valor**
Arquitetura		8051
FLASH	(CODE)	256 *KBytes*
RAM	(DATA)	128 *Bytes*
	(IDATA)	128 *Bytes*
XRAM	(XDATA)	8 *KBytes*
Frequência do CPU		24 MHz

O microcontrolador possui 256 *KBytes* de memória FLASH, que é utilizada para armazenamento das instruções (CODE). Possui 256 *Bytes* de RAM, sendo 128 *Bytes* de acesso direto (DATA) e os outros 128 *Bytes* de acesso indireto (IDATA). Além da memória RAM, contém também memória RAM externa (XDATA), tendo esta 8 *KBytes* de tamanho. A frequência de relógio do CPU é de 24 MHz.

Osciloscópio:

Para obter os tempos de execução do *software*, foi utilizado o osciloscópio digital DS203 [46]. A imagem do osciloscópio está presente na figura 6.2.

Figura 6.2: Osciloscópio DS203

6.1.2 *Software*:

O *software* é composto por todos os programas utilizados no desenvolvimento do código e nos testes ao mesmo. A tabela 6.3 apresenta a plataforma de *software* utilizada.

Tabela 6.3: Componentes de *software* utilizados

IDE	**Versão**
IAR Embedded Workbench for 8051	8.30.2

Para desenvolver o código da *framework* foi utilizado o IDE *IAR Embedded Workbench for 8051* [47] (versão 8.30.2). O IDE IAR fornece um ambiente integrado de desenvolvimento, que facilita a edição, a compilação e a depuração do código. Do vasto leque de ferramentas disponibilizadas pelo IDE IAR, destacam-se: o compilador, o assemblador, o *linker* e o editor.

6.2 Métricas de Teste

Para comparar as duas implementações (TMP e C) utilizaram-se algumas métricas de teste e comparação. As métricas utilizadas estão divididas em dois grupos:

- **Gestão do código:** Indica a facilidade da gestão e manutenção do código implementado, esta é efetuada por duas análises, que são:
  - **LOC (*Lines Of Code*):** Contagem do número de linhas de código sem espaços nem comentários;
  - **NOM (*Numbers Of Modules*):** Contagem do número de módulos da *framework*, cada *.h* (linguagem C) ou *.hpp* (linguagem C++) corresponde a um módulo.
- **Desempenho do Código:** Indica o desempenho do código gerado pelo compilador, esse desempenho é analisado das seguintes formas:
  - **Consumo de memória de código (CODE):** Tamanho em *Bytes* da memória de código ocupada;
  - **Consumo de memória de dados (DATA e XDATA):** Tamanho em *Bytes* da memória de dados consumida;
  - **Tempos de execução:** Tempos gastos na execução das tarefas, avaliado através do *Duty-Cycle*.

6.3 Testes Realizados

Após serem apresentadas as métricas de comparação das duas implementações na secção anterior (secção 6.2), nesta secção são demonstrados os testes realizados nas duas implementações (C e TMP), com base nessas métricas.

6.3.1 Gestão de Código

As métricas de gestão de código fazem uma comparação ao nível da organização e manutenção do código da *framework*. Por cada uma das implementações (C e TMP) é efetuada a análise do número de linhas de código (LOC) de cada módulo e feita a contagem do número de módulos (NOM).

Número de linhas de código (LOC):

Camada HAL:

A tabela 6.4 apresenta os valores de contagem do número de linhas de código para cada módulo desenvolvido nas duas implementações (C e TMP) da camada HAL. Os módulos são: módulo de gestão dos LEDs da placa (LEDs), módulo de gestão do ADC do microcontrolador (ADC), módulo de gestão do DMA presente no microcontrolador (DMA) e módulo de gestão das portas série do microcontrolador (USART). No final da tabela 6.4 é apresentada a soma total do número de linhas de código da camada HAL.

Tabela 6.4: Número de linhas de código (LOC) - HAL

Módulos	**C**	**TMP**
LEDs	268	319
ADC	146	130
DMA	217	223
USART	926	866
Total	**1557**	**1538**

Analisando os resultados da tabela, pode-se concluir que o número de linhas de código total da camada HAL é de 1557 na implementação em C e de 1538 na implementação em TMP. A figura 6.3 apresenta o gráfico de comparação do total das linhas de código da camada HAL nas duas implementações.

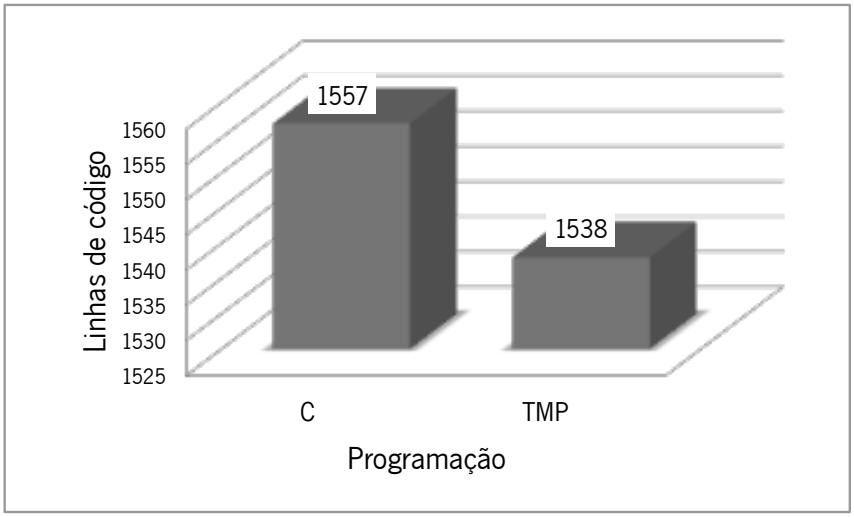


Figura 6.3: Gráfico - Número de linhas de código (LOC) - HAL

Analisando a tabela e o gráfico anterior, conclui-se que o número de linhas de código da camada HAL é ligeiramente inferior na implementação em TMP, que equivale a cerca de 1,22% menor que a implementação em C.

Camada OSAL:

A tabela 6.5 apresenta os valores de contagem do número de linhas de código para cada módulo desenvolvido nas duas implementações (C e TMP) da camada OSAL. Os módulos são: o módulo que implementa uma lista de mensagens para comunicação entre tarefas do OS (Mensagens), o módulo que possibilita a criação de temporizadores (Temporizadores), o módulo que permite a criação de tarefas (Tarefas) e o módulo que implementa a gestão do sistema operativo (OSAL). Além disso, a tabela 6.5 apresenta, no final, a soma total do número de linhas de código da camada OSAL.

Tabela 6.5: Número de linhas de código (LOC) - OSAL

Módulos	C	TMP
Mensagens	252	113
Temporizadores	236	320
Tarefas	—	16
OSAL	242	301
Total	**730**	**750**

Analisando os resultados da tabela 6.5, conclui-se que no total o número de linhas de código da camada OSAL na implementação em C é de 730, enquanto na implementação em TMP é de 750. A figura 6.4 apresenta o gráfico de comparação do total das linhas de código da camada OSAL nas duas implementações.

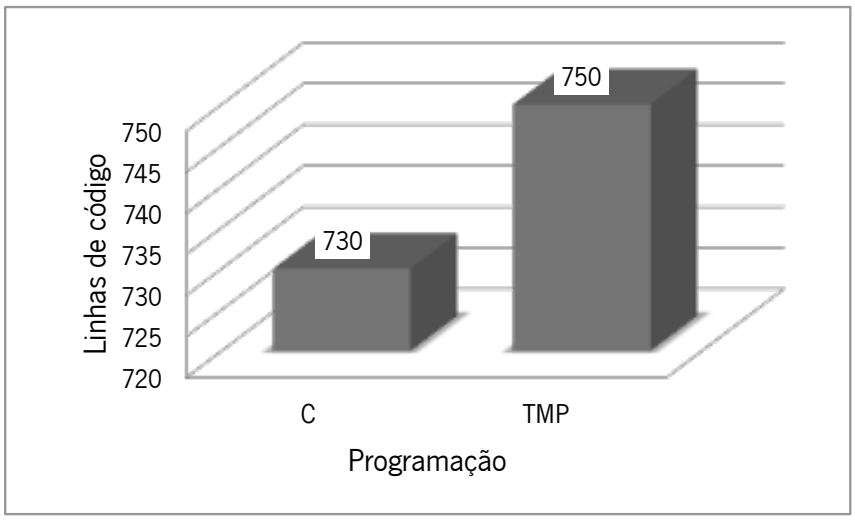


Figura 6.4: Gráfico - Número de linhas de código (LOC) - OSAL

Analisando a tabela e o gráfico anterior, pode-se verificar que o número de linhas de código da camada OSAL é ligeiramente superior na implementação em TMP, que equivale a cerca de 2,67% superior em relação à implementação em C.

Total das duas camadas:

A figura 6.5 apresenta o gráfico da soma das linhas de código das duas camadas: HAL e OSAL.

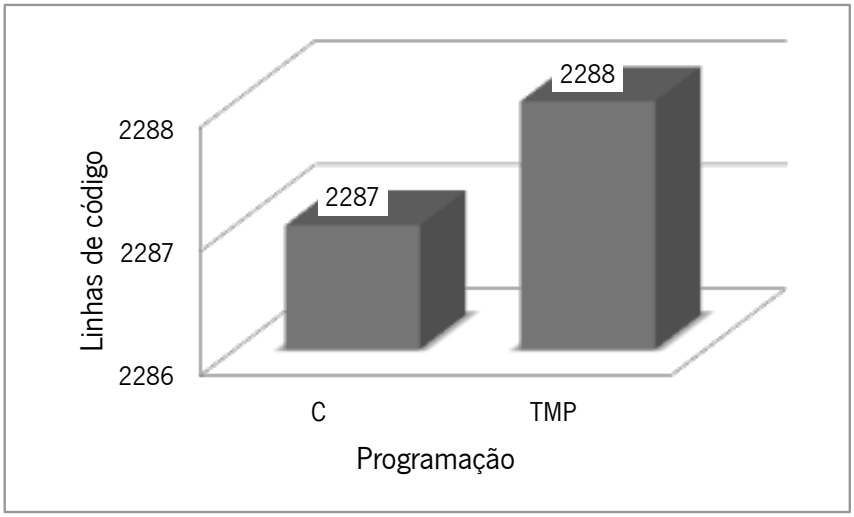


Figura 6.5: Gráfico - Número de linhas de código (LOC) - Total

Analisando o gráfico, conclui-se que o número de linhas de código das duas implementações é praticamente idêntico, apenas difere numa única linha de código a mais na implementação em TMP. Isto significa que a implementação em TMP não traz qualquer desvantagem no aumento das linhas de código em relação à implementação em C.

Número de módulos (NOM):

A tabela 6.6 apresenta a contagem do número de módulos das camadas HAL e OSAL nas duas implementações (C e TMP). No final da tabela, é apresentada a soma total dos módulos para cada uma das implementações. Cada módulo é identificado por um ficheiro *.h* em linguagem C ou *.hpp* em linguagem C++.

Tabela 6.6: Número de módulos (NOM)

Camadas	C	TMP
HAL	6	8
OSAL	2	7
Total	**8**	**15**

Analisando os resultados da tabela 6.6, pode-se constatar que tanto a camada HAL como a camada OSAL possui mais módulos na implementação em TMP, do

que na implementação em C. A camada HAL possui seis módulos na implementação em C e oito módulos na implementação em TMP. A camada OSAL possui dois módulos na implementação em C e sete módulos na implementação em TMP. O gráfico presente na figura 6.6, apresenta a comparação da soma total de módulos para as duas implementações.

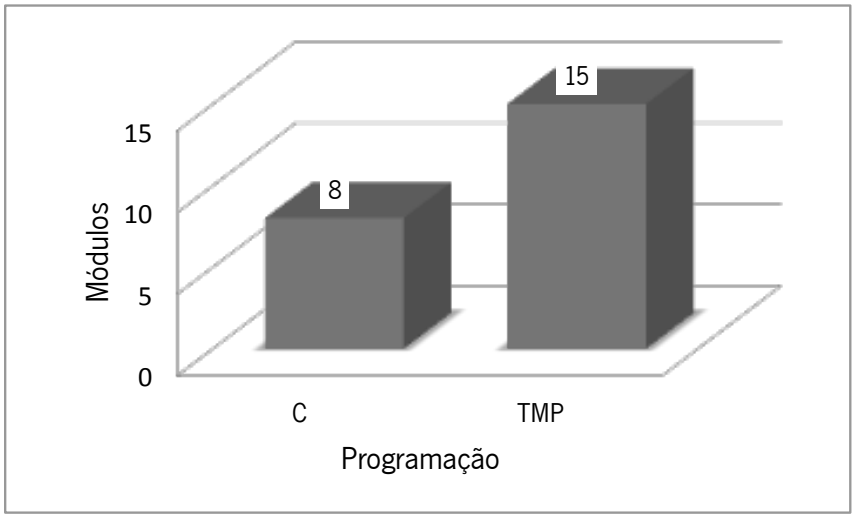


Figura 6.6: Gráfico - Número de módulos (NOM) - Total

No total, a implementação em C contém oito módulos, enquanto que a implementação em TMP possui quinze módulos. Isto significa que a implementação em TMP possui mais 46,67% de módulos que a implementação em C. Desta forma, é possível concluir que a implementação em TMP apresenta uma maior modularidade ao nível do código que a implementação em C e, como consequência, permite fazer uma melhor manutenção e gestão do código.

6.3.2 Desempenho do Código

As métricas de desempenho do código fazem uma comparação ao nível do *performance* do código gerado, verificando por cada uma das implementações as seguintes variáveis: o tamanho da memória de código (CODE) e da memória de dados (DATA) consumida e os tempos de execução do *software*.

Os testes de comparação foram efetuados para três níveis de otimização do compilador:

- ***Balanced*:** Otimização intermédia que efetua um compromisso entre o tamanho de memória ocupado e a velocidade de processamento;
- ***Size*:** Otimização ao nível do tamanho de memória de dados e de código;
- ***Speed*:** Otimização ao nível da velocidade de processamento da aplicação.

Como o grau de variabilidade do código das funcionalidades existentes nas duas *frameworks* é muito elevado, se fossem apresentados todos os casos de teste da variabilidade do código do sistema, faria com que este capítulo fosse demasiado extenso. Assim, decidiu-se apresentar apenas um caso de teste que abrangesse um grande número de funcionalidades, tanto ao nível da camada HAL como da camada OSAL. A tabela 6.7 apresenta a configuração das funcionalidades para a camada OSAL.

Tabela 6.7: Configuração das funcionalidades - OSAL

Funcionalidade	**Configuração**
Número de tarefas	5
Modo *Power Saving*	Ativo

A camada OSAL foi configurada com cinco tarefas e com o modo de poupança de energia (*Power Saving*) ativo.

A tabela 6.8 apresenta a configuração das funcionalidades para a camada HAL.

Tabela 6.8: Configuração das funcionalidades - HAL

Funcionalidade	**Configuração**
USART	
Número da porta série	0
Meio de comunicação	ISR
LEDs	
Número de LEDs	1
Modo *Blink*	Ativo
ADC	
Número de canais	1

A camada HAL foi configurada da seguinte forma: o módulo USART foi ativado com a configuração da porta série zero no meio de comunicação por ISR, o módulo LEDs foi ativado com apenas um LED e o modo *Blink* ativo e, por último, o módulo ADC foi ativado com apenas um canal ativo.

Consumo de memória de código (CODE):

A tabela 6.9 apresenta o consumo de memória de código provocado pelas duas implementações, C e TMP.

Tabela 6.9: Consumo de memória de código (CODE) em *Bytes*

	Balanced		***Size***		***Speed***	
Memórias	**C**	**TMP**	**C**	**TMP**	**C**	**TMP**
CODE	47557	47388	45532	45872	50592	51328

No nível de otimização *Balanced* do compilador, são ocupados 47557 *Bytes* de memória de código na implementação em C, enquanto na implementação em TMP 47388 *Bytes*. No nível de otimização *Size* do compilador, são ocupados 45532 *Bytes* de memória de código na implementação em C e 45872 *Bytes* na implementação em TMP. E por último, no nível de otimização *Speed* do compilador, são ocupados pela implementação em C 50592 *Bytes* de memória de código, enquanto que a implementação em TMP ocupa 51328 *Bytes*. O gráfico da figura 6.7 demonstra a comparação desses valores.

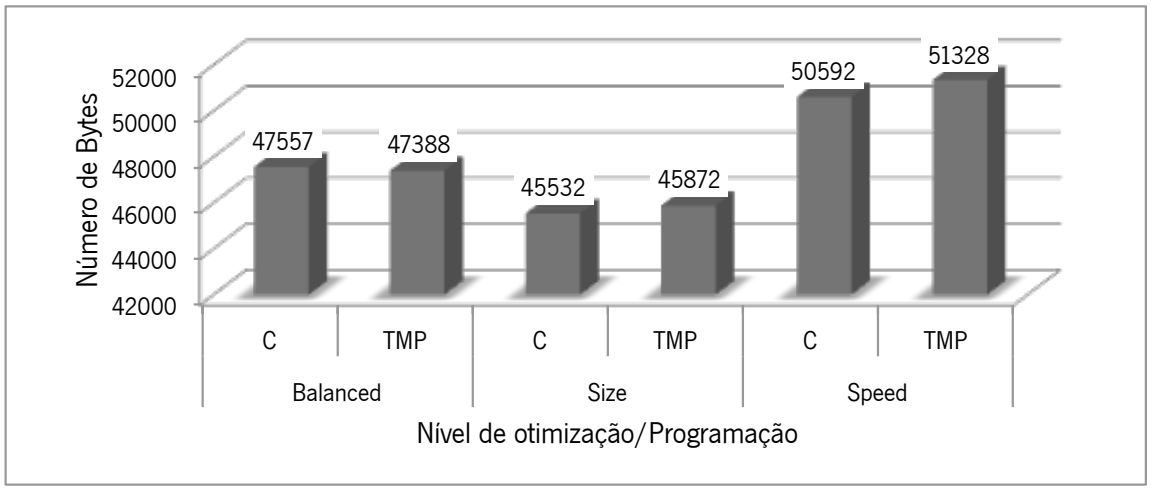


Figura 6.7: Gráfico - Consumo de memória de código (CODE) em *Bytes*

No nível de otimização *Balanced* do compilador, a implementação em TMP reduz em cerca de 0,36% o tamanho de memória de código ocupada em relação à implementação em C. No nível de otimização *Size* do compilador, a implementação em TMP aumenta em cerca de 0,74% a memória ocupada em relação à implementação em C. E por último, no nível de otimização *Speed* do compilador, a implementação em TMP ocupa cerca de mais 1,43% de memória de código que a implementação em C.

Analisando os valores obtidos, pode-se concluir que o TMP afeta um pouco os consumos de memória de código em relação à implementação em C, contudo são valores ligeiramente superiores nos níveis de otimização *Size* e *Speed*, e até inferiores na opção de otimização *Balanced*.

Consumo de Memória de Dados:

A tabela 6.10 apresenta os consumos provocados pelas duas implementações, ao nível da memória de dados interna (DATA) e externa (XDATA).

Tabela 6.10: Consumo de memória de dados (DATA) em *Bytes*

	Balanced		***Size***		***Speed***	
Memórias	**C**	**TMP**	**C**	**TMP**	**C**	**TMP**
DATA	90	86	90	86	90	86
XDATA	4016	4270	4016	4271	4016	4270
Total	**4106**	**4356**	**4106**	**4357**	**4106**	**4356**

A implementação em TMP apresenta sempre menores consumos de memória de dados interna (DATA) que a implementação em C, contudo, a nível de memória de dados externa (XDATA) a implementação em C apresenta menores consumos que a implementação em TMP. No total, e no nível otimização *Balanced* do compilador, a implementação em C apresenta 4106 *Bytes* consumidos de memória de dados, enquanto a implementação em TMP 4356 *Bytes*. No nível de otimização *Size* do compilador, a implementação em C consome 4106 *Bytes*, enquanto a implementação em TMP 4357 *Bytes*. Por último, no nível de otimização *Speed* do compilador, a implementação em C consome 4106 *Bytes* de memória de dados, enquanto a implementação em TMP 4356 *Bytes*. O gráfico da figura 6.8 apresenta a comparação dos valores totais de consumos de memória de dados.

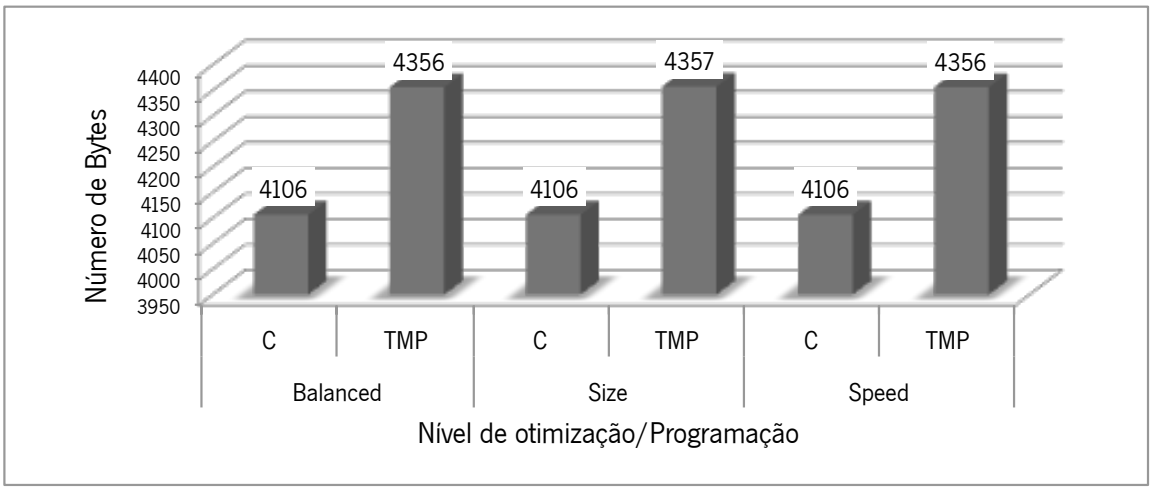


Figura 6.8: Gráfico - Consumo de memória de dados (DATA) total em *Bytes*

No nível de otimização *Balanced* do compilador, a implementação em C ocupa cerca de menos 5,74% de memória de dados que a implementação em TMP. No nível de otimização *Size* do compilador, a implementação em C consome cerca de menos 5,76% de memória de dados que a implementação em TMP. E por último, no nível de otimização *Speed* do compilador, a implementação em C consome cerca de menos 5,74% de memória de dados que a implementação em TMP.

Analisando os valores obtidos, pode-se concluir que o consumo de memória de dados é ligeiramente superior na implementação em TMP. Contudo, parte desse aumento pode ser justificado porque a implementação em C utiliza alocação de memória dinâmica na criação dos eventos, ao contrário da implementação em TMP que faz isso de forma estática.

Tempos de execução:

A tabela 6.11 apresenta os valores dos tempos de execução de cada uma das versões, C e TMP.

Tabela 6.11: Tempos de execução em *µsegundos*

	Balanced		***Size***		***Speed***	
Tempos	**C**	**TMP**	**C**	**TMP**	**C**	**TMP**
T*on*(µs)	484.3	445.5	500.7	475.2	465	393.2
T*off*(µs)	4878	4866	4874	4896	4843	4822
DC(%)	**9.03**	**8.39**	**9.32**	**8.85**	**8.76**	**7.54**

A tabela apresenta os valores do tempo em execução (T*on*), do tempo em *sleep* (T*off*) e do valor do *Duty-Cycle* (DC), que indica o ciclo de trabalho, calculado pela seguinte fórmula: $DC(\%) = \frac{Ton}{Ton+Toff} \cdot 100$. O tempo que demora a executar uma tarefa é dado pelo T*on*, e esse tempo, bem como o valor de *Duty-Cycle* (DC), é sempre inferior na implementação em TMP para os três níveis de otimização. No nível de otimização *Balanced* do compilador, o valor de DC na implementação em C é de 9,03% e na implementação em TMP de 8,39%. No nível de otimização *Size* do compilador, a implementação em C apresenta um DC de 9,32% enquanto que a implementação em TMP 8,85%. Por último, no nível de otimização *Speed* do compilador, a implementação em C contém um DC de 8,76%, enquanto a implementação em TPM um DC de 7,54%. A figura 6.9 apresenta a comparação dos valores de *Duty-Cycle*.

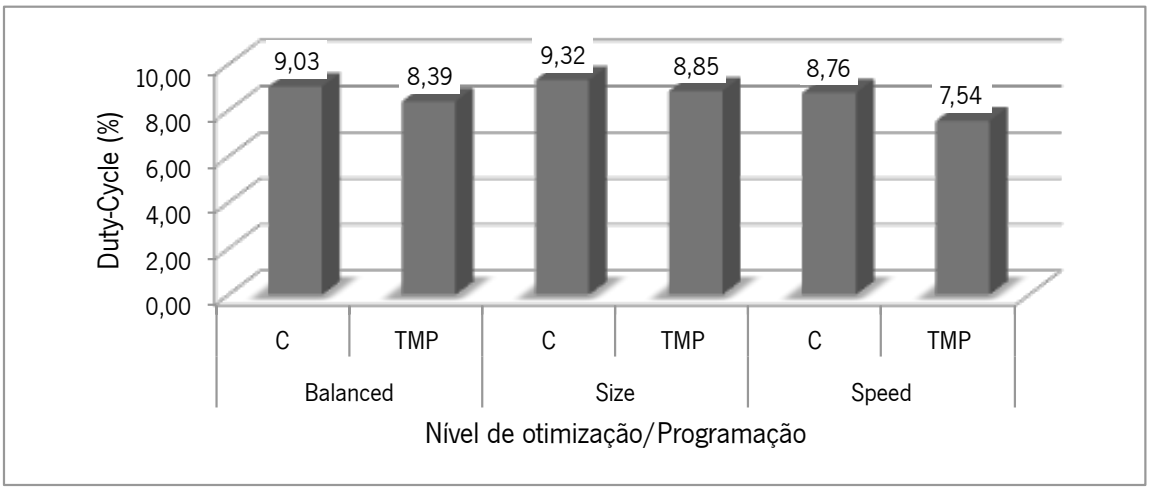


Figura 6.9: Gráfico - *Duty-Cycle* em percentagem

No nível de otimização *Balanced* do compilador, o DC da implementação em TMP é cerca de 7,09% menor que da implementação em C. No nível de otimiza-

ção *Size* do compilador, o DC da implementação em TMP é cerca de 5,04% menor que a implementação em C. E por último, no nível de otimização *Speed* do compilador, a implementação em TMP consegue obter cerca de 13,93% menos DC que a implementação em C.

Analisando os valores anteriores, pode-se concluir que a implementação em TMP consegue menores tempos de execução em relação à implementação em C. Isto porque o TMP reduz o número de instruções de saltos (*jumps*), relativas ao código binário gerado, em relação à implementação em C.

6.4 Conclusões

Este capítulo, apresentou os resultados experimentais das comparações efetuadas entre duas versões de implementação da gestão da variabilidade do código, da camada HAL e OSAL da TIMAC. As duas versões de implementação, onde foram efetuados os casos de teste, são: a implementação em linguagem C com programação condicional, desenvolvida pela *Texas Instruments* e a implementação em C++ com TMP, desenvolvida nesta dissertação.

Nessa análise foram utilizadas dois tipos de métricas: as métricas de gestão do código que avaliam o número de linhas de código e de módulos de cada implementação; e as métricas de desempenho do código, que comparam as duas implementações ao nível do consumo de memória de código, consumo de memória de dados e dos tempos de execução.

Ao analisar os resultados das métricas de gestão de código, pode-se concluir que a implementação em TMP não aumenta o número de linhas de código em relação à implementação em C e que oferece um grau de modularidade maior, permitindo, assim, uma melhor manutenção e gestão do código.

Por fim, ao analisar os resultados das métricas de desempenho do código gerado, pode-se concluir que a implementação em TMP afeta um pouco o consumo de memória, tanto a nível da memória de dados como da memória de código. Contudo, apresenta melhorias significativas nos tempos de execução. Logo, a implementação em TMP não afeta em muito o desempenho da aplicação de teste, comparativamente com a aplicação em C.

Assim, conclui-se que embora ao nível do desempenho do código gerado, a implementação em TMP apresente um aumento irrelevante no consumo de memória, comparativamente com a solução em C, no que diz respeito aos tempos de execução,

os papéis invertem-se, uma vez que os resultados são melhores com a implementação em TMP. Ao nível da gestão do código, é evidente que a implementação em TMP apresenta melhorias relativamente à implementação em C. Logo, num contexto geral, a implementação em TMP apresenta melhores resultados que a implementação em C.

Capítulo 7

Conclusões

Neste capítulo são apresentadas as conclusões retiradas após o desenvolvimento da dissertação (secção 7.1). E no final, enumeradas algumas sugestões de melhoria de forma a dar seguimento ao trabalho desenvolvido (secção 7.2).

7.1 Conclusão

Esta dissertação apresentou o desenvolvimento de uma *framework* generativa para *edge devices*, que incidiu sobre as camadas HAL (*Hardware Abstraction Layer*) e OSAL (*Operating System Abstraction Layer*) da pilha de *software* TIMAC para WSNs (*Wireless Sensor Networks*) da TI (*Texas Instruments*). As bibliotecas da TIMAC foram desenvolvidas em linguagem C e como mecanismo de gestão da variabilidade do código, utilizaram compilação condicional. Embora a compilação condicional apresente excelentes resultados ao nível do código executável gerado aquando da compilação, ao nível da gestão da variabilidade, apresenta um código de difícil manutenção e muito suscetível a erros. Isto acontece devido à existência de inúmeras diretivas de pré-processamento espalhadas pelo código das bibliotecas (*#ifdef, #else, #endif e #elseif*). Após análise dos vários mecanismos de gestão da variabilidade do código existentes, verificou-se que o *template metaprogramming* era uma boa alternativa ao mecanismo utilizado pela *Texas Instruments*, sendo escolhido como mecanismo de gestão de variabilidade utilizado para o desenvolvimento da *framework*.

A *framework* foi modelada seguindo técnicas de engenharia de SPL (*Software Product Lines*), recorrendo a diagramas de funcionalidades para esquematizar o grau de variabilidade dos componentes que compõem as camadas HAL e OSAL da TIMAC. Depois de representar individualmente os componentes em diagramas de classes, cada

um deles é implementado através de técnicas de *template metaprogramming*, sendo em muitos casos utilizados tipos de dados e meta-funções da biblioteca *Boost.MPL* (*MetaProgramming Library*).

Após a implementação da *framework*, foram realizados casos de teste para comparar as duas versões: a implementada nesta dissertação e a implementada pela própria TI. Os casos de teste foram efetuados seguindo dois tipos de métricas: as métricas de gestão de código e as métricas de desempenho do código. Depois de analisados os resultados dos casos de teste nas duas versões, concluiu-se que, ao nível da gestão do código, o TMP oferece um maior grau de modularidade, permitindo, assim, uma melhor manutenção e gestão do código. Em relação ao desempenho do código gerado, o TMP apresenta um ligeiro aumento nos consumos de memória de código e memória de dados, contudo, demonstra melhorias na diminuição dos tempos de execução. Logo, pode-se concluir que o C++ e o *template metaprogramming* são uma opção viável, relativamente à linguagem C e compilação condicional para gestão da variabilidade do código deste tipo de dispositivos.

Todos os objetivos propostos inicialmente foram alcançados. Ao nível pessoal, a elaboração desta dissertação, permitiu aprofundar conhecimentos sobre programação orientada a objetos, mais propriamente, sobre a linguagem de programação C++. Permitiu também, adquirir novos conhecimentos, tais como: os vários mecanismos de gestão da variabilidade de código, mais especificamente sobre *template metaprogramming*, diagrama de funcionalidades e técnicas de engenharia de *Software Product Lines*.

7.2 Trabalho Futuro

Os objetivos propostos inicialmente foram cumpridos, contudo, podem ser adicionadas novas funcionalidades de forma a melhorar e a expandir a *framework*, tais como:

- **Expandir a gestão da variabilidade à camada MAC da TIMAC:** esta dissertação teve como objetivo melhorar a forma como é gerida a variabilidade do código da camada HAL e OSAL da pilha de *software* TIMAC. Como trabalho futuro, é proposto estender a *framework* ao resto das camadas, ou seja, à camada MAC;

- **Aumentar o nível de abstração da *framework*:** desenvolver uma interface gráfica, que permite configurar e gerir toda a *framework*, num nível de abstração mais elevado. A interface gráfica baseia-se em modelos que manipulam ficheiros XML, os quais são processados por um processador de XSLT, gerando de forma automática os ficheiros de configuração C++;

- **Exportação de mais funcionalidades da biblioteca *Boost.MPL*:** ao utilizar novas funcionalidades da biblioteca *Boost.MPL*, é possível melhorar o processo de gestão da variabilidade do código, como por exemplo, aumentar o número de validações em tempo de compilação;

- **Alargar a compatibilidade com novas placas de desenvolvimento suportadas pela TIMAC:** a *framework* apresentada nesta dissertação, apenas é compatível com a placa de desenvolvimento CC2530DK [48]. Como trabalho futuro propõe-se expandir a compatibilidade da *framework* para outras placas de desenvolvimento, suportadas pela TIMAC;

- **Aplicar o modelo a placas de desenvolvimento de outros fabricantes:** como o modelo exposto aqui apresentou bons resultados, seria interessante aplicar este modelo a placas de outros fabricantes, que usam outro tipo de arquiteturas, tais como, as placas de desenvolvimento da STM [49], nomeadamente a STM32F4-Discovery [50].

Apêndice A

Ficheiros XSLT

Este anexo apresenta os códigos XSLT de geração dos ficheiros de configuração C++ do módulo OSAL da *framework*, estes códigos aplicam ações de transformação sobre os dados contidos no ficheiro XML apresentado na secção 5.3.5. Primeiramente (secção A.1) é apresentado o ficheiro que gera a estrutura de configuração do OSAL e de seguida (secção A.2) é apresentado o ficheiro que gera o *array* de tarefas do OSAL.

A.1 Geração da Estrutura de Configuração do OSAL

Código A.1: Geração da estrutura de configuração do OSAL

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
    Transform">
  <xsl:strip-space elements="*"/>
  <xsl:output method="text" encoding="iso-8859-1" indent="no"/>
  <xsl:template match="/">
    <xsl:text>#ifndef __COSAL_CONFIG_H__&#xA;</xsl:text>
    <xsl:text>#define __COSAL_CONFIG_H__&#xA;&#xA;</xsl:text>
    <xsl:text>#include "containers/vector.hpp"&#xA;&#xA;</xsl:text>
    <xsl:variable name="tasks" select="osal/tasks/task" />
    <xsl:for-each select="$tasks">
      <xsl:text>#include "</xsl:text>
      <xsl:value-of select="@name" />
      <xsl:text>.hpp"&#xA;</xsl:text>
    </xsl:for-each>
    <xsl:text>&#xA;struct TMP_OSAL_CONFIG {&#xA;</xsl:text>
    <xsl:text>&#09;typedef mpl::vector&lt; </xsl:text>
    <xsl:for-each select="$tasks">
      <xsl:value-of select="@name" />
      <xsl:text>_t</xsl:text>
```

```
        <xsl:if test="position() &lt; count($tasks)">
          <xsl:text>, </xsl:text>
        </xsl:if>
      </xsl:for-each>
      <xsl:text>&gt; V_OSAL_TASKS;&#xA;</xsl:text>
      <xsl:text>&#09;typedef TMP_OSAL_CNF_t&lt;V_OSAL_TASKS,
          TMP_OSAL_POWER_SAVING_</xsl:text>
      <xsl:value-of select="osal/@power_saving" />
      <xsl:text>&gt; OSAL_CNF;&#xA;</xsl:text>
      <xsl:text>&#09;typedef Cosal_t&lt; OSAL_CNF &gt; Cosal;&#xA;</xsl
          :text>
      <xsl:text>};&#xA;&#xA;</xsl:text>
      <xsl:text>#endif /* __COSAL_CONFIG_H__ */</xsl:text>
    </xsl:template>
  </xsl:stylesheet>
```

A.2 Geração do *Array* de Tarefas do OSAL

Código A.2: Geração do *array* de tarefas do OSAL

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
    Transform">
  <xsl:strip-space elements="*"/>
  <xsl:output method="text" encoding="iso-8859-1" indent="no"/>
  <xsl:template match="/">
    <xsl:text>#ifndef __COSAL_TASKS_INIT_H__&#xA;</xsl:text>
    <xsl:text>#define __COSAL_TASKS_INIT_H__&#xA;&#xA;</xsl:text>
    <xsl:text>template &lt; typename T_CONFIG &gt;&#xA;</xsl:text>
    <xsl:text>OSAL_TASK_RUNTIME_ACCESS_t Cosal_t&lt; T_CONFIG &gt;::
        task_arr[ mpl::size&lt; typename T_CONFIG::V_TASKS &gt;::
        value ] = {</xsl:text>
    <xsl:text>&#xA;</xsl:text>
    <xsl:variable name="tasks" select="osal/tasks/task" />
    <xsl:for-each select="$tasks">
      <xsl:text>&#09;{</xsl:text>
      <xsl:text>&amp;T_CONFIG::V_TASKS::item</xsl:text>
      <xsl:value-of select="position()-1" />
      <xsl:text>::messages.events, </xsl:text>
      <xsl:text>&amp;T_CONFIG::V_TASKS::item</xsl:text>
      <xsl:value-of select="position()-1" />
      <xsl:text>::events, </xsl:text>
      <xsl:text>T_CONFIG::V_TASKS::item</xsl:text>
      <xsl:value-of select="position()-1" />
      <xsl:text>::processEvents </xsl:text>
      <xsl:text>}</xsl:text>
      <xsl:if test="position() &lt; count($tasks)">
        <xsl:text>,</xsl:text>
      </xsl:if>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
    <xsl:text>};&#xA;&#xA;</xsl:text>
    <xsl:text>#endif /* __COSAL_TASKS_INIT_H__ */</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

Bibliografia

[1] M. Fajar, K. Hisazumi, T. Nakanishi, and A. Fukuda, "A Domain-Specific Modeling for Dynamically Reconfigurable Environmental Sensing Applications," vol. 41, no. Icsca, 2012.

[2] L. Pomante, S. Olivieri, L. Lavagno, and P. Torino, "An Extended Framework for the Development of WSN Applications," *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 International Congress on*, no. 978, pp. 745–749, 2010.

[3] M. Janota, J. Kiniry, and G. Botterweck, "Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines," *Lero Technical Report*, 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.155.2869

[4] N. Cardoso, "Middleware e ferramentas para desenvolvimento de sistemas de vigilância para segurança, controlo e conforto (SVSC 2-M Toolkit)," Ph.D. dissertation, 2013.

[5] D. Lohmann and O. Spinczyk, "Developing embedded software product lines with AspectC++," *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*, p. 740, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1176617.1176702

[6] S. Pinto, T. Castro, J. Mendes, S. Lopes, M. Ekpanyapong, and A. Tavares, "Exploiting template metaprogramming to customize an object-oriented operating system," in *IEEE International Symposium on Industrial Electronics*. Taipei, Taiwan: IEEE, May 2013, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6563724

[7] T. Instruments, "CC2530," 2015. [Online]. Available: http://www.ti.com/ww/en/analog/cc2530/

[8] I. Systems, "IAR Embedded Workbench for 8051," 2015. [Online]. Available: https://www.iar.com/iar-embedded-workbench/8051/

[9] S. University, "C++ Template Metaprogramming in 15 Minutes," 2013. [Online]. Available: http://stanfordacm.com/\protect\char"2026\relax/Template-Metaprogramming.pdf

[10] D. Beman and D. Abrahams, "Portability Boost MPL," 2015. [Online]. Available: http://www.boost.org/doc/libs/1_46_1/libs/mpl/doc/tutorial/portability.html

[11] C. Y. Chong and S. P. Kumar, "Sensor networks: Evolution, opportunities, and challenges," in *Proceedings of the IEEE*, vol. 91, no. 8, 2003, pp. 1247–1256.

[12] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: A survey," pp. 393–422, Mar. 2002. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1389128601003024

[13] L. Mottola and G. P. Picco, "Programming wireless sensor networks," pp. 1–51, 2011.

[14] Techterms, "Framework," 2013. [Online]. Available: http://www.techterms.com/definition/framework

[15] T. Alliance, "TinyOS," 2000. [Online]. Available: http://www.tinyos.net

[16] D. Gay, M. Welsh, E. Brewer, and S. Ave, "The nesC Language : A Holistic Approach to Networked Embedded Systems." [Online]. Available: http://www.tinyos.net/papers/nesc.pdf

[17] MathWorks, "Simulink," 2015. [Online]. Available: http://www.mathworks.com/products/simulink/

[18] ——, "MathWorks," 1984. [Online]. Available: http://www.mathworks.com

[19] ——, "SimEvents," 2015. [Online]. Available: http://www.mathworks.com/products/simevents/

[20] G. Project, "GCC, the GNU Compiler Collection," 1987. [Online]. Available: https://gcc.gnu.org

[21] I. Systems, "IAR," 1983. [Online]. Available: https://www.iar.com

[22] K. Company, "KEIL," 1986. [Online]. Available: http://www.keil.com

[23] Software Product Lines, "What is a software product line?" 2014. [Online]. Available: http://www.softwareproductlines.com

[24] P. L. Engineering, "What is Product Line Engineering?" 2014. [Online]. Available: http://www.productlineengineering.com/overview/what-is-ple.html

[25] R. Barry and R. T. E. Ltd, "freeRTOS," 2015.

[26] T. Instruments, "TIMAC," 2015. [Online]. Available: http://www.ti.com/tool/timac

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-m. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[28] D. Lohmann and O. Spinczyk, "Developing embedded software product lines with AspectC++," *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*, pp. 740–741, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1176617.1176702

[29] M. U. Olaf Spinczyk, Georg Blaschke, Christoph Borchert, Daniel Lohmann, Rainer Sand, Horst Schirmeier, Ute Spinczyk, "AspectC++," 2014. [Online]. Available: http://www.aspectc.org

[30] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison Wesley Professional, 2004.

[31] Nicolai M. Josuttis; David Vandevoorde, "C++ Templates: Metaprograms," p. 8, 2003. [Online]. Available: http://www.informit.com/articles/article.aspx?p=30667

[32] R. Steiner, T. R. Mück, and A. A. Fröhlich, "C-MAC: a Configurable Medium Access Control Protocol for Sensor Networks," in *Proc. of the 9th IEEE Sensors*, Kona, HI, 2010, pp. 845–848. [Online]. Available: http://www.lisha.ufsc.br/pub/Steiner_Sensors_2010.pdf

[33] M. Barr, *Programming Embedded Systems in C and C++*. O'Reilly Media, 1999.

[34] N. Cardoso, J. Vale, J. Cabral, J. Mendes, P. Cardoso, A. Tavares, and J. Monteiro, "Use of template metaprogramming to address the heterogeneity of Video Surveillance Systems," in *2012 IEEE International Conference on Industrial Technology, ICIT 2012, Proceedings*. IEEE, 2012, pp. 384–389.

[35] Beman Dawes; David Abrahams, "Boost C++ Libraries," 1998. [Online]. Available: http://www.boost.org

[36] Microsoft, "Standard Template Library," 2014. [Online]. Available: http://msdn.microsoft.com/en-us/library/c191tb28.aspx

[37] Intel, "Intel 8051," 1980. [Online]. Available: http://www.intel.com/design/embcontrol/

[38] T. Instruments, "A True System-on-Chip Solution for 2.4-GHz IEEE802.15.4 and ZigBee Applications," Tech. Rep., 2011. [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2530.pdf

[39] I. Systems, “IAR C/C++ Compiler for 8051,” Tech. Rep., 2015. [Online]. Available: http://netstorage.iar.com/SuppDB/Public/UPDINFO/006898/ew/doc/EW8051_CompilerReference.pdf

[40] W3C, “Extensible Markup Language (XML),” 2015. [Online]. Available: http://www.w3.org/XML/

[41] ——, “World Wide Web Consortium,” 1994. [Online]. Available: http://www.w3.org

[42] ——, “XSL Transformations (XSLT),” 2015. [Online]. Available: http://www.w3.org/TR/xslt

[43] Microsoft, “Command Line Transformation Utility,” 2015. [Online]. Available: http://www.microsoft.com/en-us/download/details.aspx?id=21714

[44] Microsoft Corporation, “Microsoft,” 1975. [Online]. Available: http://www.microsoft.com

[45] Microsoft, “Visual Studio,” 1997. [Online]. Available: https://www.visualstudio.com

[46] S. Studio, “DSO Quad,” 2015. [Online]. Available: http://www.seeedstudio.com/wiki/DSO_Quad

[47] I. Systems, “IAR Embedded Workbench for 8051,” 2015. [Online]. Available: https://www.iar.com/iar-embedded-workbench/8051/

[48] T. Instruments, “CC2530 Development Kit,” 2015. [Online]. Available: http://www.ti.com/tool/cc2530dk

[49] STMicroelectronics, “STMicroelectronics,” 2015. [Online]. Available: http://www.st.com/web/en/home.html

[50] ——, “STM32F4DISCOVERY,” 2015. [Online]. Available: http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF252419