FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

'Driver' Modbus para Ambiente de Desenvolvimento IEC 61131-3

Vasco Alexandre Martins das Neves Fernandes

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Orientador: Professor Doutor Mário Jorge Rodrigues de Sousa

Julho de 2009

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES 2008/2009

A Dissertação intitulada

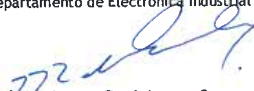
""DRIVER" MODBUS PARA AMBIENTE DE DESENVOLVIMENTO IEC 61131-3"

foi aprovada em provas realizadas em 16/Julho/2009

o júri

Presidente Professor Doutor Paulo José Lopes Machado Portugal
Professor Auxiliar do Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Jaime Francisco Cruz Fonseca
Professor Associado do Departamento de Electrónica Industrial da Universidade do Minho



Professor Doutor Mário Jorge Rodrigues Sousa
Professor Auxiliar do Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projecto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extractos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são correctamente citados.

Autor - VASCO ALEXANDRE MARTINS DAS NEVES

Faculdade de Engenharia da Universidade do Porto

Resumo

Nas últimas décadas tem-se vindo a presenciar grandes esforços de normalização no campo da Automação Industrial, sendo a Comissão Electrotécnica Internacional a entidade que lidera o processo de produção e publicação de normas nesse domínio.

A norma IEC 61131, que estabelece um conjunto de características para os autómatos programáveis (*Programmable Logic Controllers* (PLCs)), tem na sua parte 3 a definição de um modelo de programação para os mesmos, incluindo a definição de quatro linguagens de programação, bem como uma linguagem de definição de máquinas de estados. Os fabricantes dos PLCs têm vindo a adaptar as suas ferramentas de programação a esta norma, no entanto apresentam algumas inconsistências, e formas de impossibilitar a portabilidade do desenvolvimento nessas ferramentas.

A procura de soluções de código fonte aberto, que se mantenham em total conformidade com a norma e que possibilitem a portabilidade entre ferramentas de edição de projectos IEC 61131-3, levou à criação do Beremiz, um ambiente de desenvolvimento integrado (*Integrated Development Environment* (IDE)) de projectos para PLCs, que permite escrever programas em conformidade com a norma, e gerar código ANSI-C correspondente, através de um compilador interno, possibilitando a sua execução nas mais variadas plataformas. A sua arquitectura modular e código fonte aberto permite que terceiros possam aspirar ao desenvolvimento de novos módulos e os integrem na ferramenta.

Com o objectivo de potenciar a comunicação que o softPLC possa fazer com outros dispositivos físicos ou lógicos, o presente projecto apresenta uma solução integrada no IDE que possibilita ao utilizador do Beremiz fazer configurações para que o sotfPLC actue como um Cliente/TCP numa rede Modbus, um protocolo amplamente usado na Automação Industrial, cuja popularidade assenta na simplicidade de implementação.

Abstract

On the last decades we have seen big efforts of standardization on Indutrial Automation field, whit the International Electrotechnical Comitee being the organization leading the process of producing and publishing those standards.

The standard IEC 61131, which establish a set of rules and characteristics that Programmable Logic Controllers (PLCs) should follow, has in its third part the definition of a programming framework for them, including the definition of four programming languages, as well as a state machine defition language. The PLCs vendors started to adapt their programming tools to this standard, however they still show some inconsistences and ways to lock users in the development made in those tools.

The search for Open Source solutions, that stay conform to the established with the standard and allow portability between tools that are used to edit IEC 61131-3 projects, led to the creation of Beremiz, an Integrated Development Environment (IDE) of PLCs projects that allow programming in accordance to the standard and to generate ANSI-C code, through a internal compiler, enabling its execution in several platforms, including Windows and Linux. The modular arquitechture of Beremiz and its Open Source code allow third parties to aspire on the development of new modules and integrate them on the IDE.

With the goal in mind of increasing the communication that the softPLC may do with other physical or logical devices, this project presents an integrated solution within the IDE that allows Beremiz user to configure his project and make the softPLC to behave as a TCP Client on a Modbus network, a communication protocol broadly used on Automation field, which popularity is based on its implementation simplicity.

Résumé

Les dernières décennies ont été marquées par de croissants efforts en vue d'une normalisation de l'Automatisation Industrielle, avec la Commission Electrotechnique Internationale comme entité à la tête du processus de production et de publication des normes du secteur.

La Norme IEC 61131 qui établit un ensemble de caractéristiques pour les automates programmables (*Programmable Logic Controllers* (PLCs)) présente dans sa troisième partie la définition d'un modèle de programmation pour ces dits automates, contenant la définition de quatre langages de programmation ainsi qu'un langage de définition de machines d'état. Les fabricants de PLC adaptent régulièrement leurs outils de programmation à cette norme, mais ils présentent néanmoins quelques incompatibilités et des freins à la portabilité du développement des dits outils.

La recherche de solutions de logiciel libre qui se maintiendraient en totale conformité avec la norme et qui permettraient la portabilité entre outils d'édition de projets IEC 61131-3, a abouti à la création de Beremiz, un environnement de développement intégré (*Integrated Development Environment* (IDE)) de projets pour PLC qui permet d'écrire des programmes en conformité avec la norme et gérer le code ANSI-C correspondant, au travers d'un compilateur interne, permettant son exécution sur les plateformes les plus diverses. Son architecture modulaire et son code source ouvert donnent la possibilité à des tiers de participer au développement de nouveaux modules et de les intégrer à l'outil.

Avec pour objectif de rendre possible la communication du softPLC avec d'autres dispositifs physiques ou logiques, ce projet présente une solution intégrée dans l'IDE qui permet à l'utilisateur du Beremiz de faire des configurations pour que le SoftPLC agisse comme un Client/TCP du réseau Modbus, un protocole amplement utilisé dans l'Automatisation Industrielle, dont la popularité tient à sa simplicité d'implémentation.

Agradecimentos

Agradeço aos meus pais todo o apoio, em todas as circunstâncias, e o facto de verem sempre a minha formação como um meio e não um fim.

Ao meu irmão, pelo génio na observação e visão de futuro que sempre transmitiu com enorme pragmatismo. Mas acima de tudo pela grande amizade que sempre demonstrou.

À minha namorada, por ser a minha melhor amiga. Aturou-me de tal forma, que ao longo da dissertação os toques de telemóvel e rejeições constituiam um protocolo de comunicação!

Ao meu orientador, pelas dimensões humana e profissional que emprestou no processo de orientação.

Conteúdo

1 Introdução **1**
1.1 Estrutura da Dissertação 2

2 Tecnologias Usadas **3**
2.1 Beremiz 3
2.1.1 Visão Geral 3
2.1.2 PLC Builder GUI 4
2.1.3 PLCOpen Editor 5
2.1.4 Compilador IEC 61131-3 7
2.1.5 Plugins 8
2.2 Modbus 10
2.2.1 Visão Geral 10
2.2.2 Serviços 11
2.2.3 Modelo de Dados 12
2.2.4 Implementação TCP/IP 12

3 Arquitectura do 'Driver' **15**
3.1 Biblioteca de funções Modbus 15
3.2 Funcionamento do PLC 17
3.3 Solução Proposta 18

4 Desenvolvimento **23**
4.1 Python 23
4.1.1 Tipos de Dados 24
4.1.2 Funções 25
4.1.3 Classes 26
4.1.4 Módulos 27
4.2 Módulo *plugger* 27
4.3 Interface Gráfica 29
4.4 Geração Automática do *back-end* 30

5 Testes e Validação **35**
5.1 Validação dos dados introduzidos 35
5.2 Funcionamento do back-end 37

6 Conclusões **39**
6.1 Satisfação dos objectivos 39
6.2 Potencial de desenvolvimento 40

Referências 42

Lista de Figuras

2.1 PLC Builder GUI, no caso concreto de um projecto a usar o plugin Modbus . . . 5
2.2 Visualização do PLCOpen Editor, com um programa em SFC em edição 6
2.3 Herança do modelo de dados TC6-XML Schema e posterior conversão textual (adaptação de duas imagens cuja fonte é [1]) 7
2.4 Etapas globais de compilação e organização do código gerado (Fonte: [1]) 8
2.5 Acções numa transação Modbus, livre de erros (Fonte: [2]) 11
2.6 Application Data Unit do Modbus/TCP (Fonte: [3]) 13

3.1 Organização da Biblioteca Modbus (adaptação livre de esquema presente na documentação da biblioteca) 15
3.2 Interface entre o SoftPLC e um plugin do Beremiz 18
3.3 Diagrama de Classes UML para o plugin Modbus, no caso de uso Cliente 18
3.4 Diagrama de Sequência UML para a rotina de inicialização 20
3.5 Diagrama de Sequência UML para a rotina de finalização 20
3.6 Diagrama de Sequência UML para as rotinas de subscrição e publicação 21

4.1 Diagrama de Classes UML focando o módulo *plugger* 28
4.2 Aspecto gráfico da interface criada, com uma instância de cada classe 30
4.3 Mapeamento entre variáveis do PLC e zonas de memória de um servidor Modbus 32
4.4 Visão externa da manipulação de ficheiros pelo método *PlugGenerate_C()* 33

5.1 Aspecto gráfico do Shop Floor Simulator 38

6.1 Visão sobre as frentes de desenvolvimento futuro do presente projecto 40

Lista de Tabelas

2.1 Prefixos para a localização e tamanho das *directly represented variables* (Fonte: [4]) 9
2.2 Exemplo da utilização dos plugins e associação a variáveis no PLC(Fonte: [1]) . 10
2.3 Tipos de dados básicos usados no protocolo Modbus (Fonte: [2]) 12

3.1 Parâmetros comuns das funções modbus da biblioteca 17
3.2 Exemplos de funções modbus encontradas na API da biblioteca Modbus 17

Abreviaturas e Símbolos

ADU	Application Data Unit
ASCII	American Standard Code for Interchange Information
FBD	Function Block Diagram
GPL	General Public License
GUI	Graphical User Interface
HMI	Human Machine Interface
IDE	Integrated Development Environment
IEC	International Electroctechnical Commission
IL	Instruction List
I/O	Inputs/Outputs
IP	Internet Protocol
LD	Ladder Diagram
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PLC	Programmable Logic Controller
POU	Program Organization Unit
RTU	Remote Terminal Unit
SFC	Sequential Function Chart
ST	Structured Text
TCP	Transport Control Protocol
UML	Unified Model Language
XML	eXtensible Markup Language

Capítulo 1

Introdução

As últimas décadas têm vindo a presenciar grandes esforços de normalização nos mais diversos domínios da Engenharia Electrotécnica, que se reflectem também no campo da Automação Industrial. A Comissão Electrotécnica Internacional lidera o processo de produção e publicação de normas nesse domínio. Uma dessas normas, a IEC 61131-3, define um modelo de programação para os *Programmable Logic Controllers* (PLCs). Os fabricantes destes dispositivos, depois de um período em que difundiram no mercado as suas soluções proprietárias, têm vindo a adapta-las à normalização, mas mantem-se em muitas situações a impossibilidade de exportar o desenvolvimento feito num dado equipamento, importando para um outro.

No desenvolvimento de *software*, um novo paradigma de propriedade intelectual e comercial ganhou força, que potencia a liberdade de utilização, cópia, modificação ou redistribuição de módulos, partes de aplicações ou aplicações completas.

É neste contexto que surge o Beremiz, um ambiente de desenvolvimento integrado (*Integrated Development Environment* (IDE)) de projectos para PLCs, que permite a escrita de programas em conformidade com o estabelecido na norma IEC 61131-3, e gerar código ANSI-C correspondente à aplicação de controlo, possibilitando a sua execução nas mais variadas plataformas. A sua arquitectura modular e código fonte aberto permite que terceiros possam aspirar ao desenvolvimento de novos módulos e os integrem na ferramenta.

O objectivo do presente projecto é arquitectar, desenvolver e testar um módulo que permita ao utilizador do Beremiz fazer um conjunto de definições para que o seu softPLC possa comunicar com outros dispositivos numa rede, usando o protocolo Modbus. Usando uma analogia ao que habitualmente se encontra em dispositivos físicos, será como que disponibilizar ao utilizador uma carta Mestre ou Escravo que "fale" Modbus, bem como os meios necessários ao nível da IDE para as configurar devidamente.

1.1 Estrutura da Dissertação

Para além da introdução, esta dissertação contém mais 5 capítulos. No capítulo 2, são descritas as tecnologias usadas no presente projecto: a ferramenta Beremiz e o protocolo Modbus. No capítulo 3, identificam-se os elementos que condicionam a arquitectura do 'driver' Modbus, e apresenta-se uma solução para a mesma. No capítulo 4 explicam-se as opções tomadas no desenvolvimento, enquadrando-as na arquitectura pensada. Como introdução, faz-se uma breve apresentação da linguagem de programação Python, na qual o Beremiz está programado, por forma a enquadrar o leitor com essas mesmas opções. No capítulo 5 descrevem-se os testes efectuados à conformidade de funcionamento da interface gráfica e do código C gerado. No capítulo 6 apresentam-se as conclusões sobre a satisfação dos objectivos propostos, bem como o potencial de expansão do módulo criado.

Capítulo 2

Tecnologias Usadas

Neste capítulo pretende-se explorar os dois grandes componentes que se deseja integrar no presente projecto. Por um lado a ferramenta Beremiz, enquanto ambiente de desenvolvimento de projectos para autómatos programáveis (*Programmable Logic Controllers* (PLC)) (tanto na óptica da sua utilização para esse fim como na óptica da sua estrutura interna), e por outro lado o protocolo Modbus, com especial incidência na implementação regularmente designada por Modbus/TCP.

2.1 Beremiz

2.1.1 Visão Geral

O Beremiz é um Ambiente de Desenvolvimento Integrado (*Integrated Development Environment* (IDE)) de código fonte aberto para o desenvolvimento de programas para PLC's em conformidade com o disposto na norma IEC 61131-3, disponibilizada ao público livremente sob a licença de *software* GNU GPLv2 ou posterior.

A norma IEC 61131 estabelece um conjunto de regras que todos os PLC's devem seguir (desde características mecânicas e eléctricas a aspectos lógicos e de comunicações). A parte 3 da norma [4] aborda as questões associadas à sua programação, definindo a semântica e sintaxe de quatro linguagens: duas textuais (*Structured Text* (ST) e *Instruction List* (IL)) e duas gráficas (*Function Block Diagram* (FBD) e *Ladder Diagram* (LD)). Define igualmente um modelo de programação, onde se especificam: tipos de dados, identificadores, Unidades de Organização de Programas (*Program Organization Units* (POU)) que incluem *Functions*, *Function Blocks* e *Programs*, elementos dos *Sequential Function Charts* (SFC) (para definição de máquinas de estados, baseado em Grafcet), elementos das configurações (variáveis globais, recursos, tarefas) e as respectivas relações entre estes.

As principais motivações para o desenvolvimento de uma ferramenta com estas características, assumidas pelos autores em [1] e [5], listam-se de seguida:

- Identificação de uma falta de soluções de cógido fonte aberto neste domínio;
- Apesar da normalização sobre a programação de PLC's, há dificuldades na portabilidade dos programas desenvolvido para os mesmos;
- A concepção e manutenção de aplicações desenvolvidas para PLC's dependem directamente das respectivas IDE's associadas ao hardware dos fabricantes;
- O processo de aprendizagem da norma IEC 61131-3 envolve normalmente a aquisição de licenças dispendiosas, limitando ou até impossibilitando o uso pelos estudantes de um *software* nos seus próprios recursos informáticos ao longo do período de ensino não assistido;
- O facto do código fonte dos compiladores e do *runtime* serem fechados impede que se explore e se prove determinados modos de operação que se pretende que as aplicações cumpram.

O Beremiz foi desenvolvido de forma modular, encontrando suporte em diversos sub-projectos:

- **PLC Builder GUI** - Visão Global dos Projectos
- **PLCOpen Editor** - Editor dos programas IEC 61131-3
- **MATIEC** - Compilador IEC 61131-3 -> ANSI-C
- **CanFestival** - CANOpen *Framework* para interface com I/O físicos
- **SVGUI** - Ferramenta para integração de HMIs

O PLC Builder GUI e o PLCOpen Editor estão programados em Python [6], usando o módulo adaptador WxPython [7] para a biblioteca WxWidgets [8]. O uso destas tecnologias torna estas interfaces portáveis entre diferentes plataformas (incluindo Windows e Linux).

Nas secções seguintes focar-se-ão os três primeiros, por serem estruturantes de todo o IDE, sendo que os dois últimos se inserem na categoria do que os autores denominam de plugins, para os quais se incidiu, numa secção única, a atenção na interface que disponibilizam ao código que é gerado após compilação.

2.1.2 PLC Builder GUI

A interface denominada de PLC Builder GUI confere ao utilizador uma perspectiva global do projecto, apresentando três áreas principais, como se pode ver na Figura 2.1:

- Barra de Ferramentas
- Área de Gestão dos plugins
- *Log Console* - Informação textual ao utilizador

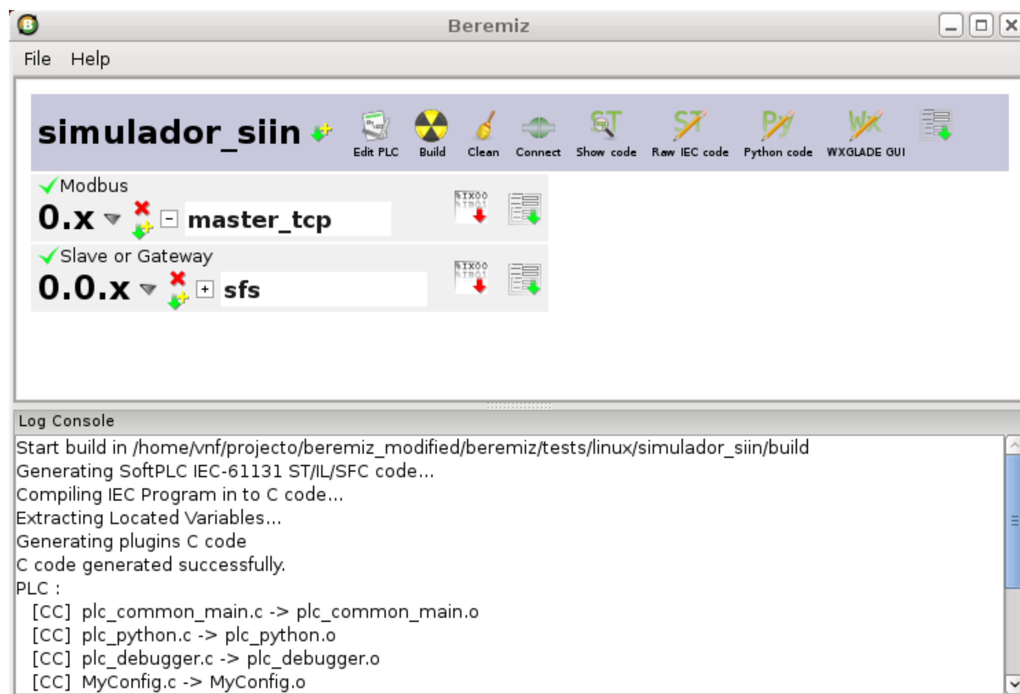


Figura 2.1: PLC Builder GUI, no caso concreto de um projecto a usar o plugin Modbus

A barra de ferramentas é dinâmica, ou seja, as opções disponíveis variam com o estado do projecto. Inclui opções como: definir a plataforma alvo, adicionar plugins, definir opções de compilação, compilar o projecto, ver o código IEC 61131-3 gerado, transferir para o SoftPLC, iniciar e parar a execução do algoritmo de controlo.

Os plugins são adicionados numa estrutura em árvore, podendo a um dado plugin serem associados outros, dependentes do primeiro (se aplicável face ao plugin escolhido). Estes apresentam algumas semelhanças gráficas com a barra de tarefas, concretamente em relação à associação de novos plugins, bem como na forma como são mostrados os parâmetros específicos. São aliás, do ponto de vista de programação da GUI, classes que herdam de uma mesma classe abstracta.

A consola textual apresenta informação relativa ao resultado das acções tomadas pelo utilizador. É também para esse espaço que são endereçadas as mensagens provenientes do código C, quando este se encontra a ser executado.

2.1.3 PLCOpen Editor

O utilizador do Beremiz escreve os seus programas nesta ferramenta. Do ponto de vista gráfico, esta apresenta um conjunto de divisões, como mostra a figura 2.2.

Na divisão à esquerda, encontram-se duas árvores: uma de selecção, que conta com os tipos de dados definidos pelo utilizador, as POU's e ainda as configurações, recursos e tarefas, em conformidade com o estabelecido na norma IEC 61131-3; a outra onde são listadas todas as instâncias usadas no projecto (com elevado nível de detalhe, mostrando inclusivé as acções e transições dos

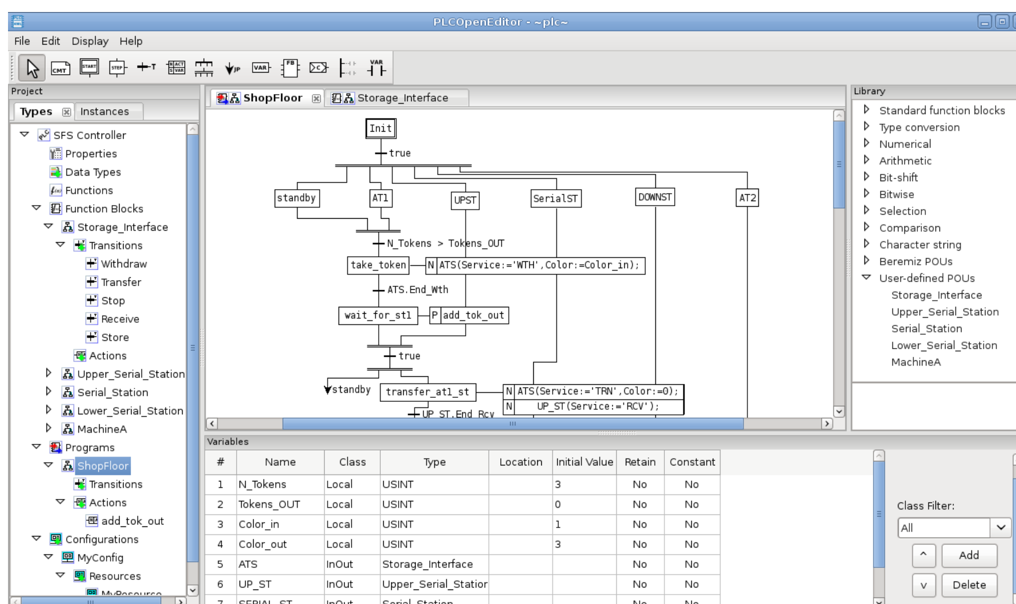


Figura 2.2: Visualização do PLCOpen Editor, com um programa em SFC em edição

SFC's, e apresentando um grafismo próprio para cada tipo de instância, que torna a identificação bastante simples).

A escrita das instruções que cada POU deve executar é feita na divisão central (notar que a referência à escrita não se limita às linguagens textuais, inclui também as linguagens gráficas), sendo que esta se torna um editor especializado para uma das cinco linguagens, em função da POU activa no momento e da linguagem escolhida (inclui-se aqui SFC como linguagem, apesar de que em [5] se explica que apenas com SFC's não é possível escrever completamente um programa, havendo por isso renitência em atribuir-lhe a designação de linguagem de programação). Dependendo da linguagem gráfica em uso (LD, FBD ou SFC), uma barra de objectos aparece no topo, que mostra os elementos que a linguagem em uso comporta.

Na divisão do fundo são listadas as variáveis associadas à POU em edição num determinado momento. É igualmente viável adicionar e eliminar variáveis nessa mesma lista, e editar os seus campos: nome, âmbito, tipo, localização, valor inicial, retenção (se o valor deve ser mantido após um *warm start*), e se o identificador deve ser declarado como constante ou não.

Na divisão à direita, encontra-se uma outra árvore de selecção com um conjunto de *Functions* e *Function Blocks*, agrupados por âmbito de utilização, que incluem para além das POU's da biblioteca padrão, as POU's já definidas pelo utilizador. Com a selecção de uma POU desta lista, pode-se arrastar a mesma para a divisão central, especialmente útil nas linguagens gráficas, pois é de imediato apresentado um bloco com a respectiva interface. No caso de ser um *Function Block Type* (abstracção da norma similar a uma classe nas linguagens orientadas a objectos), é requerido um nome para a instância desse FB, que é automaticamente adicionado à lista de variáveis da POU em edição.

Os projectos são guardados em formato XML, cujo modelo de dados segue a estrutura definida

no *XML Official Schema* do comité técnico TC6 da organização PLCOpen [9]. Essa estrutura oficial, que na prática se traduz num ficheiro *.xsd [10], é usada na criação de um projecto e estabelece todas as relações entre os objectos que compõem o mesmo. Essa valência permite que o PLCOpen Editor seja usado para validar se um determinado projecto nesse formato segue a estrutura base referida. Esta organização está patente na figura 2.3.

O PLCOpen Editor dispõe ainda de um módulo responsável pela conversão das componentes do projecto que incluam o uso de linguagens gráficas em equivalentes textuais. Em concreto, as linguagens FBD e LD são convertidas em ST equivalente, enquanto que os elementos dos SFC's têm na norma uma especificação textual própria, que é usada para este efeito.

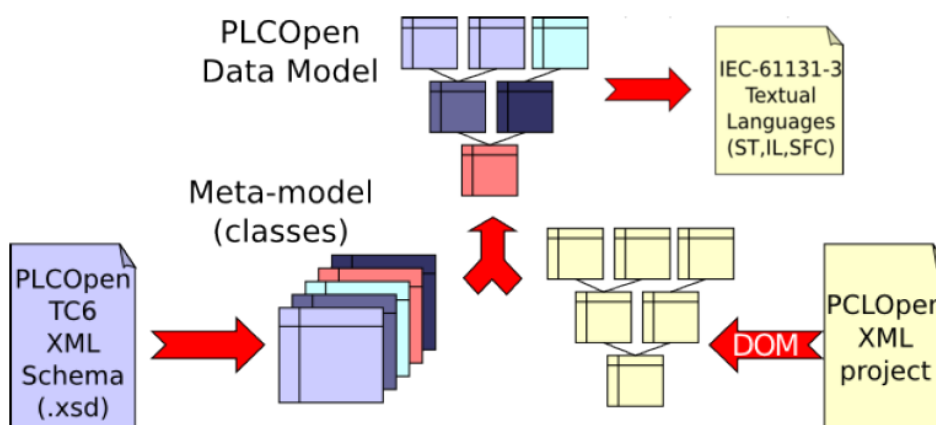


Figura 2.3: Herança do modelo de dados TC6-XML Schema e posterior conversão textual (adaptação de duas imagens cuja fonte é [1])

2.1.4 Compilador IEC 61131-3

O resultado da conversão textual de um projecto IEC 61131-3, referido no final da secção anterior, é o objecto consumido pelo compilador MATIEC para produzir o equivalente em código C, cuja organização está presente na figura 2.4. O compilador funciona em quatro etapas: *lexical analyzer*, *syntax parser*, *semantics analyzer* e *code generator*. Os detalhes de funcionamento são explicados em [5].

O bloco inferior da figura 2.4 reflecte a forma como o código C gerado pelo compilador fica organizado. Todas as variáveis e os parâmetros das POU's são declaradas como estruturas C em árvore, e as variáveis para as quais se define uma localização são declaradas como externas, e efectivamente declaradas no âmbito do plugin em causa.

O algoritmo de controlo do SoftPLC é executado por iniciativa de um módulo próprio (*Target Specific Code* na figura), responsável por gerir o relógio específico da plataforma alvo, e gerar interrupções cadenciadas para execução das tarefas.

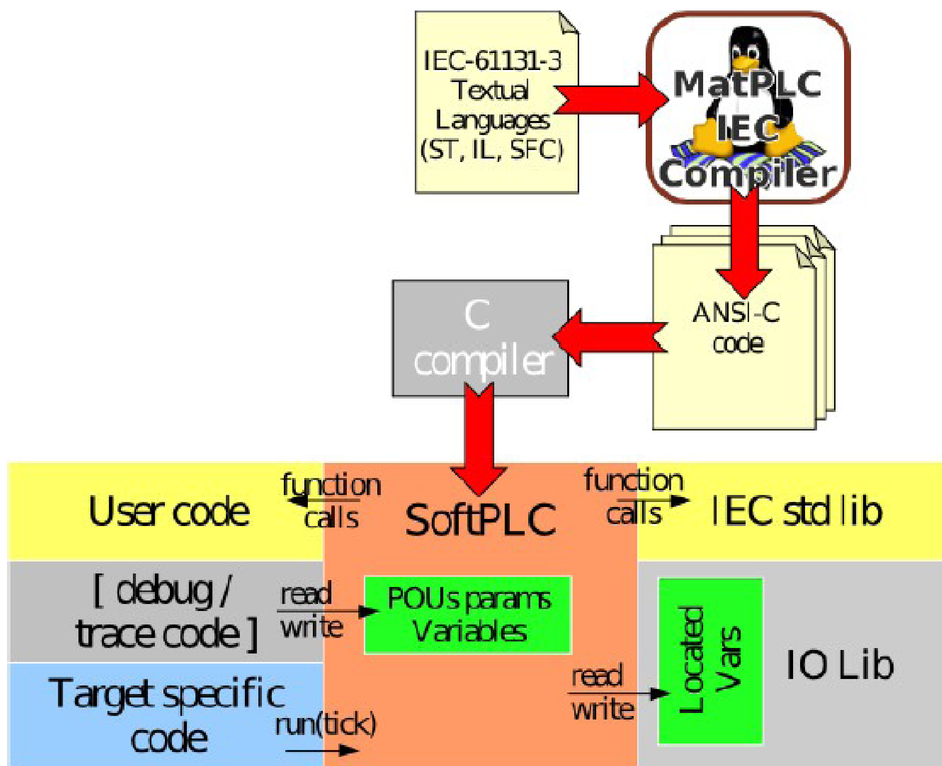


Figura 2.4: Etapas globais de compilação e organização do código gerado (Fonte: [1])

O programa acede a um conjunto de Funções e FB, sejam eles da biblioteca padrão (IEC std lib) ou criados pelo utilizador (User Code), que se encontram definidos num módulo próprio e que recebem como parâmetros as estruturas C acima referidas.

Dispõe ainda de um outro módulo (em fase de produção), cujo objectivo é o de consumir o estado actual de todos os parâmetros do programa, reproduzindo esse estado de funcionamento ao utilizador em *runtime*, através do PLCOpen Editor.

Importa salientar que o Beremiz apresenta uma restrição face ao disposto na norma IEC 61131-3. Esta refere no seu modelo de programação que uma dada configuração contém um ou mais recursos, cada um dos quais contém um ou mais programas executados sob controlo de zero ou mais tarefas. No Beremiz, é possível instanciar um único recurso e um único programa sob controlo de uma tarefa, sendo que esse programa pode (e aqui sim, em conformidade com o disposto na norma) instanciar as funções ou FB que se pretenda.

2.1.5 Plugins

Os plugins no Beremiz cumprem o objectivo de permitir ao SoftPLC comunicar com o mundo exterior. Todo o conjunto de acções de controlo está intimamente associado à necessária comunicação com dispositivos físicos ou lógicos que estão em contacto directo com o processo a

controlar (sensores, actuadores, interfaces homem-máquina, entre muitos outros dispositivos que regularmente se encontram em sistemas de automação).

Todos os plugins do Beremiz são compostos por uma interface com o utilizador e uma componente de código C, que disponibiliza um conjunto de serviços ao SoftPLC (ver adiante figura 3.2 na secção 3.3).

Importa enquadrar o uso dos plugins com o disposto na norma IEC 61131-3 sobre as denominadas *directly represented variables*. Está definida uma simbologia própria para a associação de variáveis a dispositivos físicos ou lógicos na estrutura de entrada, saída ou memória do PLC.

Essa simbologia é composta pela concatenação de:

’%’ + localização + tamanho + um ou mais inteiros separados por ’.’

Os prefixos para a localização e para o tamanho podem ser consultados na tabela 2.1. A norma refere ainda que quando a representação inclui os inteiros separados por pontos finais, estes devem ser interpretados como um endereçamento numa lógica hierárquica, com o inteiro mais à esquerda a representar o mais alto nível e decrescendo então da esquerda para a direita.

Tabela 2.1: Prefixos para a localização e tamanho das *directly represented variables* (Fonte: [4])

Tipo	Prefixo	Significado
Localização	I	Entrada
	Q	Saída
	M	Memória
Tamanho	X ou vazio	1 bit
	B	8 bits
	W	16 bits
	D	32 bits
	L	64 bits

No Beremiz, a cada plugin é associado um número inteiro que obrigatoriamente permanece único (quer seja instanciado a partir da raíz ou a partir de um plugin já existente). Este número (ou sequência de números), denominado na ferramenta de *IEC_Channel*, segue precisamente a lógica referida no parágrafo anterior, e é o formato que deve ser usado quando queremos fazer a associação de uma variável na programação do PLC a um dispositivo (o plugin em causa), através do campo localização no PLCOpen Editor (recordar secção 2.1.3).

No acto de compilação, as *directly represented variables*, ou seja, as variáveis declaradas no âmbito do programa, recurso ou configuração do PLC para as quais se tenha definido no campo *location* uma simbologia como a descrita anteriormente, são endereçadas para a árvore dos plugins de acordo com a sua localização, e consumidas pelos plugins quando estes geram a sua componente de código. A tabela 2.2 confere um exemplo ilustrativo desta funcionalidade.

Tabela 2.2: Exemplo da utilização dos plugins e associação a variáveis no PLC(Fonte: [1])

	Plugin IEC_Channel	Possible Variable Location
CANOpen plugin	0	
1st CANOpen Network	0.0	%IX0.0.3.323.1
2nd CANOpen Network	0.1	%IX0.1.3.323.1
HMI plugin	1	
1st Display	1.0	%IX1.0.3.323.1
2nd Display	1.1	%IX1.1.3.323.1

2.2 Modbus

O Modbus é um protocolo de comunicação aberto, introduzido no mercado em 1979 pela Modicon. A sua simplicidade de implementação tornou-o muito popular no seu principal domínio de aplicação, a Automação Industrial.

Não sendo objecto de normalização, viu na Modbus-IDA [11] (uma organização sem fins lucrativos composta por fabricantes e aberta a integradores e utilizadores) a principal responsável pela divulgação e evolução do protocolo, tendo publicado um conjunto de documentos de referência para a implementação, bem como gerindo aplicações que visam testar a conformidade dos dispositivos que queiram comunicar usando este protocolo.

2.2.1 Visão Geral

O Modbus é um protocolo que se situa na camada de aplicação do modelo OSI para comunicação entre dispositivos em rede, essencialmente para troca de dados no campo da automação.

Nesse nível, o protocolo segue o paradigma cliente-servidor, e diz-se *stateless*, ou seja, uma dada troca de informação é totalmente independente do histórico de trocas de informação precedentes ou eventualmente em curso. Essas trocas de informação designam-se por transacções, que consistem num pedido e numa resposta (Figura 2.5). Em determinados domínios de implementação concreta, nomeadamente barramentos série, como EIA/TIA-232, pelas especificidades da gestão do acesso ao meio, o paradigma mestre-escravo é usado com frequência, onde nesse caso é válida a correspondência:

O mestre é um cliente.

O escravo é um servidor.

Os referidos pedidos e respostas baseam-se em tramas simples, designadas de *Protocol Data Unit* (PDU), independentes das camadas inferiores. A especificação [2] define três tipos de PDU's:

- **Request PDU** que consiste em:
  1. um código que indica uma função (1 *byte*)
  2. dados correspondentes à função (tamanho variável)
- **Response PDU** que consiste em:

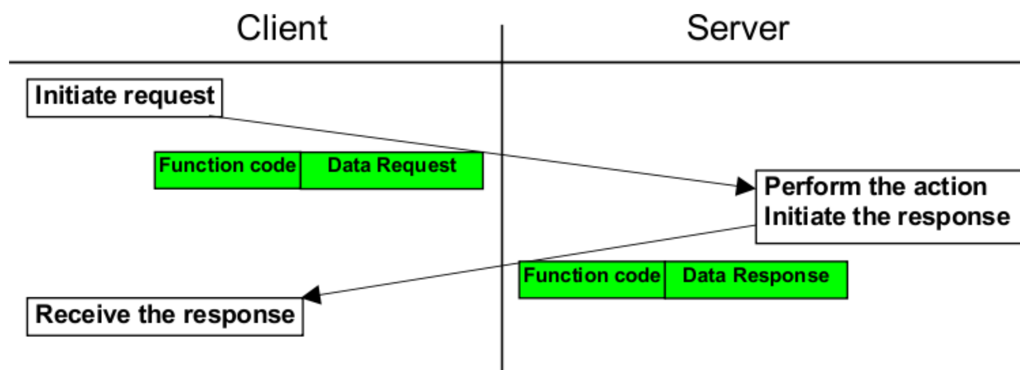


Figura 2.5: Acções numa transação Modbus, livre de erros (Fonte: [2])

1. o código da função correspondente ao pedido (1 *byte*)
2. dados correspondentes à resposta (tamanho variável)

- **Exception Response PDU** que consiste em:
  1. o código da função correspondente ao pedido + 0x80 (128) (1 *byte*)
  2. um código identificador do tipo de excepção (1 *byte*)

2.2.2 Serviços

A especificação do protocolo define um conjunto de funções, cada uma das quais com o seu código único. Estes códigos encontram-se no intervalo [1-127], sendo que o intervalo [129-255] está reservado para os códigos de excepção na resposta, como foi referido anteriormente.

São definidas também três categorias de códigos de funções:

- ***Public*** - São garantidamente únicos, e estão associados a funções bem definidas e documentadas. São validadas pela organização Modbus-IDA e existem testes de conformidade para as mesmas;
- ***User Defined*** - A especificação define os intervalos [65-72] e [100-110] como códigos de funções para implementação livre por parte dos utilizadores. Os respectivos códigos não são garantidamente únicos, pois dependem dessas mesmas implementações;
- ***Reserved*** - Estes códigos são usados por alguns fabricantes para soluções proprietárias e não estão disponíveis para uso público. Não são sequer discutidos na especificação, sendo o leitor remetido para o anexo A do documento [2] para detalhes sobre os códigos reservados.

Uma função encontra-se documentada com:

1. a descrição da função, isto é, o seu propósito, os seus parâmetros, e valores de retorno (incluindo eventuais códigos de erro nas excepções).

2. o código da função que lhe está associado.

3. o formato de cada uma das três PDU's - *Request*, *Response* e *Exception Response*

Na especificação [2] encontram-se documentadas as funções da categoria *Public*.

Em determinadas condições, a resposta de um servidor é uma excepção. Esta é identificada pela presença no campo correspondente à função de um código de erro (código da função original + 128). Os detalhes sobre o tipo de erro que ocorreu são indicados em campo próprio, dependem da função invocada e podem ser igualmente consultado na documentação de cada função.

2.2.3 Modelo de Dados

As funções públicas básicas foram desenvolvidas para troca de dados na área da automação. Na tabela 2.3 apresentam-se os tipos de dados básicos que o documento [2] especifica. O mesmo não impõe nenhuma forma sobre como esses tipos de dados devem estar organizados num dispositivo (apresenta apenas dois exemplos dessa organização).

No entanto, essa organização tem um reflexo no endereçamento usado pelas funções básicas, pelo que ao implementar um servidor deve-se documentar a organização dos dados. Analogamente, ao comunicar com um dado dispositivo servidor deve-se previamente consultar o método a ser usado para o endereçamento.

Para cada uma das linhas da tabela 2.3, o protocolo prevê o acesso a 65536 (2^{16}) items individuais, e as operações de leitura e escrita estão concebidas para se propagarem por múltiplos items consecutivos, cujo limite depende da função usada.

Tabela 2.3: Tipos de dados básicos usados no protocolo Modbus (Fonte: [2])

Nome	Tipo	Acesso
Discrete Input	1 bit	somente leitura
Discrete Output	1 bit	leitura/escrita
Input Registers	palavra 16 bits	somente leitura
Output Registers	palavra 16 bits	leitura/escrita

Salienta-se o facto de que a nomenclatura usada para o mesmo tipo de dados varia frequentemente na literatura, incluindo a própria especificação (por exemplo o tipo *Discrete Output* é referido como *Coil*, ou o tipo *Output Registers* como *Holding Registers*), facto que se propaga também para os nomes usados nas funções de acesso a estes tipos de dados.

2.2.4 Implementação TCP/IP

O protocolo Modbus conhece diversas implementações, sendo que as mais populares trabalham sobre TCP/IP e sobre transmissões série assíncronas (onde os meios físicos mais comuns são EIA/TIA-232 e EIA/TIA-485). Pode-se encontrar no documento [3] a especificação da implementação TCP/IP, para além de descrições funcionais de um cliente, servidor e *gateway* (dispositivo que garante a interface entre a rede IP e barramentos série).

O protocolo define uma trama (PDU) cujo formato é independente da implementação, como foi visto na secção 2.2.1. No Modbus/TCP acresce um cabeçalho específico, formando assim uma trama própria da implementação, denominada de *Application Data Unit* (ADU), como mostra a figura 2.6. Esse cabeçalho, denominado de MBAP na especificação, tem 7 *bytes* de comprimento, e é composto por:

1. ***transaction identifier*** (2 *bytes*) - usado para identificação unívoca das mensagens (pedidos e respostas), pois é copiado pelo servidor na resposta a um determinado pedido;

2. ***protocol identifier*** (2 *bytes*) - tem o valor zero por defeito para o Modbus, existindo essencialmente na expectativa de expansões futuras;

3. ***lenght*** (2 *bytes*) - contém o número de *bytes* que se seguem na trama, por forma a ajudar na detecção dos limites da mesma;

4. ***unit identifier*** (1 *byte*) - usado para identificação de um dispositivo remoto localizado numa rede distinta da rede TCP/IP, sendo o elemento que é usado pelas *gateways* para direccionar um pedido que lhe seja endereçado.

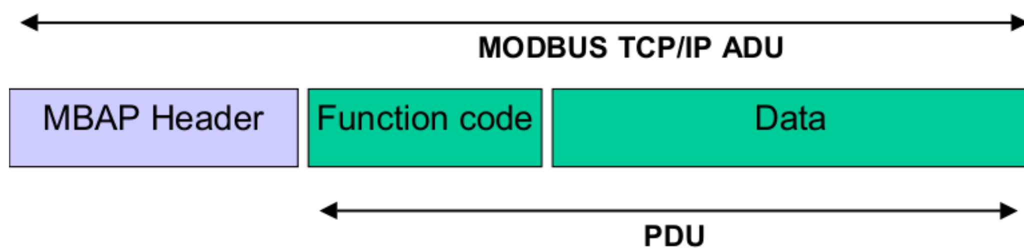


Figura 2.6: Application Data Unit do Modbus/TCP (Fonte: [3])

As configurações e topologias de uma rede Modbus TCP/IP não se encontram definidas na especificação, sendo apresentados apenas exemplos ilustrativos. É perfeitamente viável construir redes com mais do que um cliente, ou mesmo ter dispositivos que funcionem simultâneamente como cliente e servidor. Esta particularidade é vista como uma grande vantagem da implementação TCP/IP do Modbus face às restantes, e justifica o facto de se ter optado por explorar apenas esta implementação, bem como focar o desenvolvimento na mesma. No entanto, chama-se à atenção que um desvio do paradigma Mestre-Escravo tem implicações que o utilizador deve ter em mente, que serão detalhadas mais à frente, ao abordar as condicionantes da Arquitectura do ’driver’ (secção 3.3).

Capítulo 3

Arquitectura do ’Driver’

Neste capítulo faz-se uma discussão sobre os elementos que influenciam o comportamento do ’driver’: por um lado os serviços que o protocolo Modbus oferece, reflectidos na biblioteca de funções que será usada, por outro a filosofia de funcionamento de um PLC e sua interacção com as estruturas de entrada e saída. Por fim, recorrendo à linguagem de modelação *Unified Model Language* (UML), apresenta-se uma proposta de solução para a arquitectura de um cliente TCP que servirá de base ao desenvolvimento futuro.

3.1 Biblioteca de funções Modbus

Um dos pontos de partida para a concepção da arquitectura é a biblioteca de funções Modbus (programada em C), disponibilizada pela orientação. Esta reflecte a organização presente nos documentos que especificam as implementações do protocolo. Encontra-se organizada em duas camadas, como mostra a figura 3.1: a camada inferior inclui as implementações do protocolo sobre TCP/IP e sobre barramentos série (nas duas variantes de codificação de dados RTU e ASCII); a camada superior, implementações dos dispositivos mestre e escravo, que por seu turno usam os serviços das implementações da camada inferior.

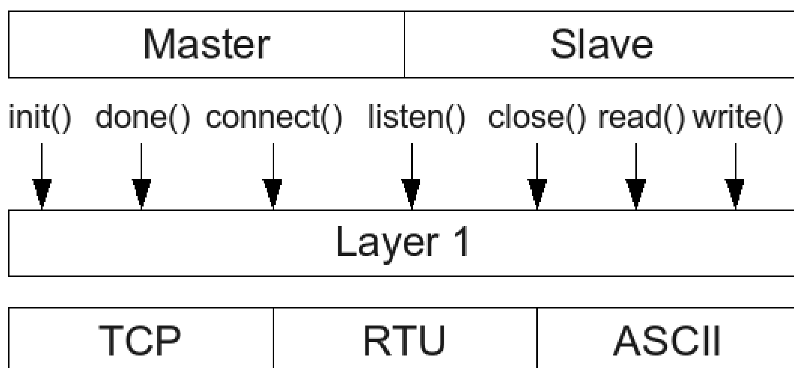


Figura 3.1: Organização da Biblioteca Modbus (adaptação livre de esquema presente na documentação da biblioteca)

Desta organização sobressai o facto de que se pode desacoplar a arquitectura em seis casos de uso, que se designarão por Cliente(Mestre) ou Servidor(Escravo) seguido de um sufixo correspondente à implementação específica do protocolo.

O estudo que se segue focar-se-á na implementação Cliente/TCP. Incidir o foco numa única implementação está ligado a restrições temporais no desenvolvimento, e o facto de se optar pela implementação TCP justifica-se pela sua versatilidade face às restantes:

- um dispositivo pode actuar em simultâneo como cliente e servidor;
- uma rede pode comportar diversos clientes e servidores;
- a camada sobre a qual assenta esta implementação está amplamente difundida, inclusivé em ambiente industrial, e nos dias de hoje faz uso de diversos meios físicos, tendo ganho enorme universalidade.

Os serviços que a implementação Mestre disponibiliza são:

- **mb_master_init** (int nd_count) - Inicializa a biblioteca, reservando memória para os *buffers* e inicializando as estruturas de dados. Recebe como parâmetro um inteiro que corresponde ao número máximo de conexões que se podem estabelecer;
- **mb_master_done** (void) - Finaliza o uso da biblioteca, libertando os recursos previamente alocados;
- **mb_master_connect** (node_addr_t node_addr) - Estabelece uma conexão com um escravo modbus. Recebe como parâmetro uma estrutura de dados que contem o tipo de escravo e seus dados específicos (no caso TCP, o IP e porto do escravo). Retorna um descritor que identifica univocamente a conexão estabelecida;
- **mb_master_close** (int nd) - Termina uma conexão previamente estabelecida, identificada pelo parâmetro que recebe, o descritor dessa mesma conexão;
- **modbus_functions** (common params, specific params) - Implementam os serviços básicos de leitura e escrita sobre os dados da tabela 2.3. Apresentam um conjunto de parâmetros comuns, e um conjunto de parâmetros específicos da função em causa;

A tabela 3.1 resume os parâmetros comuns dessas funções modbus, enquanto que a tabela 3.2 lista, a título ilustrativo, algumas das funções modbus presentes na biblioteca, incluindo a referência ao código a que correspondem na especificação. Todas as funções apresentadas são de acesso a múltiplos items das respectivas zonas de memória no escravo, e lêem ou escrevem *count* items a partir de *start_addr*, passando-os para ou lendo-os de * *data_buffer*, conforme o tipo de operação.

Tabela 3.1: Parâmetros comuns das funções modbus da biblioteca

Nome	**Tipo** (em C)	**Significado**
slave	unsigned char	Identificador do escravo para onde se envia o pedido modbus
fd	int	Descritor da conexão sobre a qual se envia o pedido modbus
send_retries	int	Número de tentativas a efectuar caso não haja resposta
error_code	* unsigned char	Destino onde é escrito o código do erro no caso da resposta ser uma excepção
response_timeout	* const struct timespec	Tempo máximo pelo qual se deve aguardar por uma resposta

3.2 Funcionamento do PLC

Outra particularidade que condiciona a arquitectura do 'driver' é a filosofia de funcionamento de um PLC. É um dispositivo originalmente desenvolvido para efectuar acções de controlo (periodicamente ou activadas por eventos) em que cada ciclo deve desempenhar:

ler entradas -> executar algoritmo de controlo -> escrever nas saídas

Estas operações de leitura das entradas e escrita nas saídas podem ser feitas de forma implícita ou explícita. É perfeitamente viável atribuir a responsabilidade de comunicação com os I/O's a um processo distinto, que funcione de forma concorrente com o algoritmo de controlo. Nesse caso, é necessário criar os mecanismos de comunicação adequados entre o processo responsável pelo algoritmo de controlo e o processo responsável pela actualização dos I/O's, bem como mecanismos que resolvam a problemática que surge com o facto de ambos acederem a uma zona de memória comum.

De forma resumida, o SoftPLC no Beremiz disponibiliza dois serviços ao utilizador: *Run* e *Stop*. O primeiro faz as inicializações da configuração (e em cadeia do recurso, do programa e de todas as POU's a ele associadas), dos plugins que foram adicionados no projecto (rotina *init_*()) e da rotina que gere a execução cíclica do algoritmo de controlo. Cada ciclo de controlo, para além de chamar a rotina de *run_(config)*, que desencadeia o algoritmo de controlo propriamente dito, faz uso das rotinas *retreive_*() e *publish_*() dos plugins, imediatamente antes e após a rotina *run_(config)*. Por fim, o serviço *Stop* oferecido ao utilizador, recorre às rotinas *cleanup_*() dos

Tabela 3.2: Exemplos de funções modbus encontradas na API da biblioteca Modbus

Código Modbus	**Nome na API**
02	read_input_bits
03	read_input_words
15	write_output_words
16	write_output_bits

plugins, antes de terminar os processos em curso, que interrompem as acções descritas. Este comportamento descrito salienta a interface entre o SotfPLC e os plugins do Beremiz, que se esquematiza na figura 3.2.

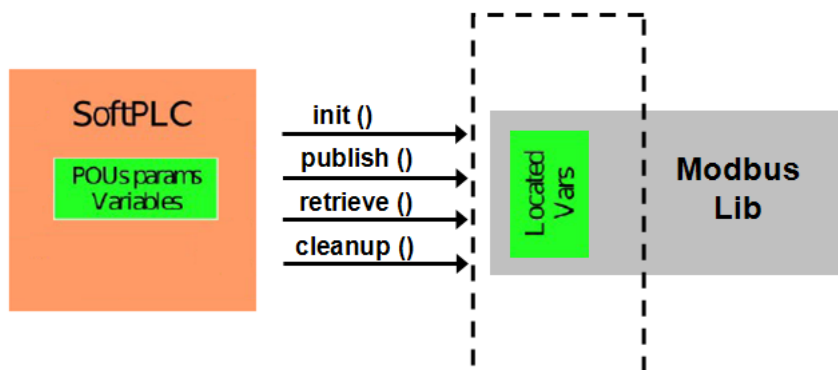


Figura 3.2: Interface entre o SoftPLC e um plugin do Beremiz

3.3 Solução Proposta

A solução que se propõe para a arquitectura contempla o caso de uso Cliente/TCP.

Um cliente Modbus numa rede vai comunicar com um conjunto de servidores ou *gateways*, efectuando a cada um desses dispositivos um ou mais pedidos. Esses pedidos basear-se-ão nas funções de acesso aos tipos de dados básicos do protocolo, podendo ser divididos em dois grandes grupos: operações de leitura e de escrita. O diagrama de classes da figura 3.3 é reflexo desta descrição.

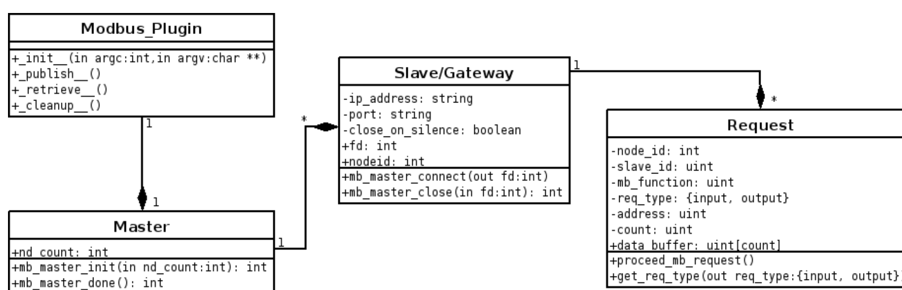


Figura 3.3: Diagrama de Classes UML para o plugin Modbus, no caso de uso Cliente

Como foi referido na secção 2.2.4, há um conjunto de implicações quando nos desviamos do paradigma Mestre-Escravo na implementação TCP. Na implementação sobre barramentos série, um cliente tem de assegurar que existe uma única transacção em curso, i.e., o cliente tem de esperar pela resposta ao primeiro pedido antes de enviar um segundo. Na implementação TCP, o cliente pode enviar vários pedidos sem esperar pelas respectivas respostas. Essa é a razão para a

existência do campo *transaction identifier* (que um servidor se limita a copiar do pedido para a resposta), para que o cliente saiba que resposta corresponde a que pedido.

O número de transacções que um cliente pode iniciar em simultâneo é um parâmetro dependente da implementação, que na especificação é referido como *NumberMaxOfClientTransaction*. De forma análoga, a implementação de um servidor tem um número máximo de pedidos que poderá receber para processamento em simultâneo. Esse parâmetro é referido na especificação como *NumberMaxOfServerTransaction*, e é visto como uma das principais características de um servidor Modbus/TCP, pela forma como afecta o comportamento e o desempenho do mesmo.

Pelo exposto, se um cliente não puder ter consciência à partida das limitações dos dispositivos servidores com os quais vai comunicar (que será obviamente o caso mais comum, bem como a generalização desejável), deve garantir que com cada dispositivo inicia uma e uma só transacção em simultâneo (pior caso), pois caso um servidor não possa aceitar mais pedidos, responderá com uma excepção (código 06: servidor ocupado), perdendo-se a oportunidade de efectuar a transacção com o objectivo inicial (ler ou escrever dados).

Estes factores foram preponderantes para a escolha que se fez quanto à forma como o 'driver' deveria funcionar. Optou-se por um funcionamento puramente síncrono, onde o cliente executa cada transação de forma sequencial, aguardando a respectiva resposta (ou expiração de um tempo pré-definido) antes de iniciar nova transação. Alia-se igualmente a simplicidade de implementação de um funcionamento síncrono, bem como o facto de se adequar para a reutilização praticamente sem alterações para as implementações sobre barramentos série do protocolo Modbus.

Os diagramas de sequência para a inicialização (figura 3.4) e finalização (figura 3.5) do 'driver' usam uma abstracção que pretende dar a imagem que podem estar associados vários nós, e que são estabelecidas as respectivas conexões, ou terminadas, de forma sequencial (daí o uso de duas instâncias da classe Slave/Gateway: Node 1 e Node n). Essa abstracção pode ser conseguida através do padrão de concepção *iterator* [12].

O diagrama de sequência para as acções retreive() e publish() (figura 3.6) assentam numa lógica de funcionamento síncrono. Mais uma vez usou-se a abstracção da multiplicidade das instâncias da classe Request. Esta abordagem proporciona uma implementação simples, e segue a natureza do paradigma Mestre-Escravo, mantendo o seu funcionamento independente do modo escolhido (TCP, RTU ou ASCII). O diagrama inclui uma breve referência à execução do algoritmo do PLC, para salientar que o grupo de pedidos de leitura é feito imediatamente antes deste e o grupo de pedidos de escrita imediatamente após.

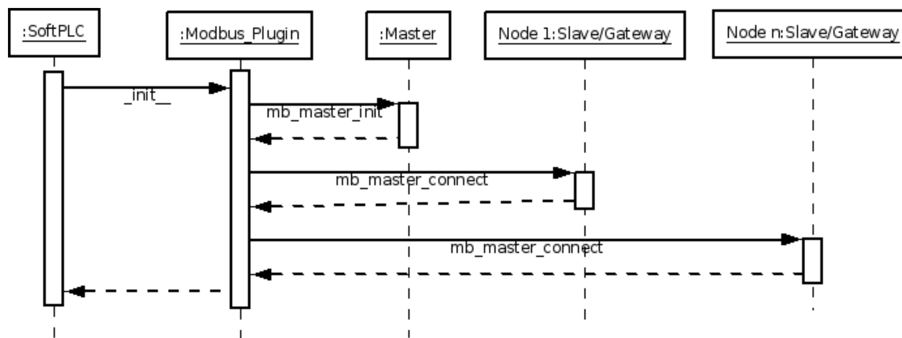


Figura 3.4: Diagrama de Sequência UML para a rotina de inicialização

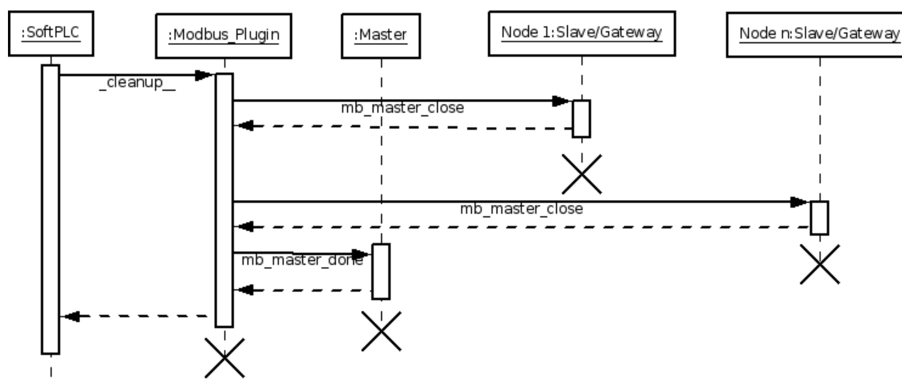


Figura 3.5: Diagrama de Sequência UML para a rotina de finalização

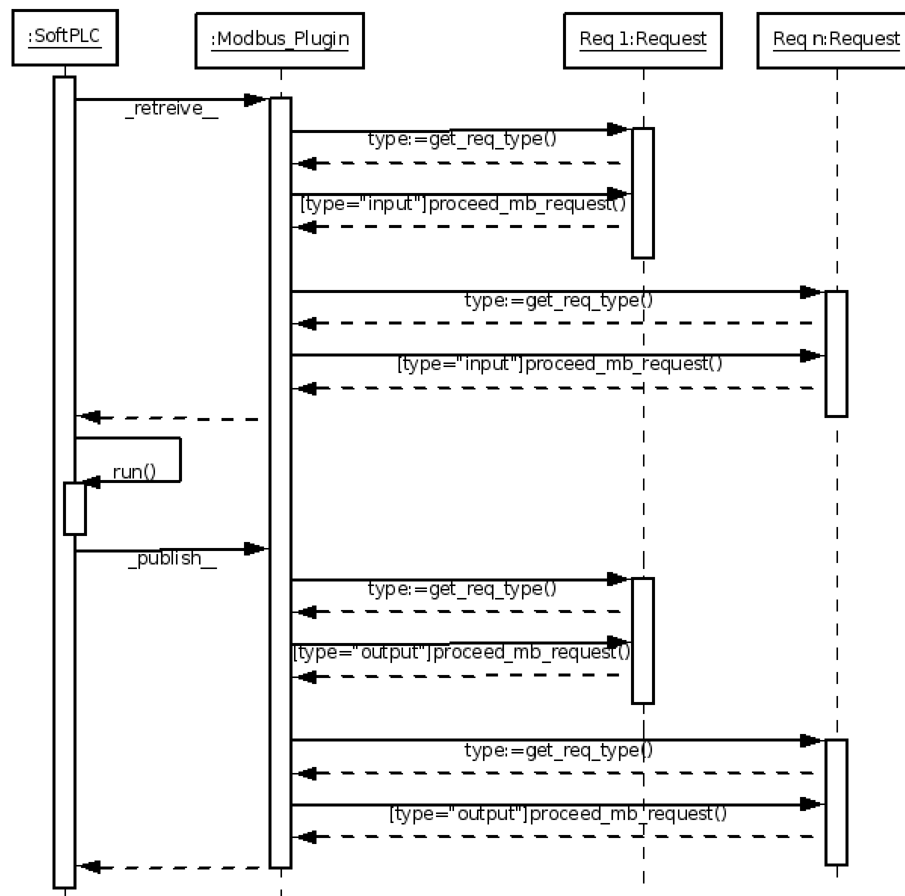


Figura 3.6: Diagrama de Sequência UML para as rotinas de subscrição e publicação

Capítulo 4

Desenvolvimento

Neste capítulo descrevem-se os passos dados no desenvolvimento do plugin Modbus, no que toca ao caso de uso escolhido, Cliente/TCP. Numa primeira secção, abordam-se as principais características da linguagem de programação Python, focando também a atenção em funcionalidades particulares que foram usadas, permitindo ao leitor um melhor enquadramento das opções subsequentes. Na segunda secção, identificam-se os atributos e os métodos de um plugin no Beremiz, e seu relacionamento com a ferramenta. Por fim, nas duas últimas secções, explicam-se as opções tomadas no desenvolvimento das duas grandes componentes em que se subdivide um plugin, a interface com o utilizador (o *front-end*) e o código C gerado para funcionar em conjugação com o SoftPLC (o *back-end*).

4.1 Python

Python é uma linguagem de programação de alto nível, desenvolvida originalmente por Guido van Rossum. Existem inúmeras referências bibliográficas sobre a mesma, das quais se destacam [13] e [14] como as consultadas para o presente trabalho. Não se pretende fazer nenhum tipo de abordagem exaustiva à linguagem. O objectivo é apontar algumas das características do Python e funcionalidades que mais foram usadas.

As principais características que se identificam na linguagem são:

- É uma linguagem interpretada, ou seja, não existe explicitamente um acto de compilação prévio à execução, algo a que regularmente se designa de linguagem de *scripting*. Ilegalidades na semântica ou sintaxe da linguagem levantam excepções, que caso não tenham um tratamento específico na codificação, levam à ocorrência de um comportamento genérico, que é o de transmitir ao utilizador um conjunto de informações de *debugging*, cuja apresentação varia com o ambiente de execução, e interromper a mesma;

- É totalmente orientada a objectos. Tudo na linguagem são objectos, numa definição lata, como referido em [14], onde os objectos não têm necessariamente de ter uma lista de atributos e métodos, mas no sentido em que podem ser associados a variáveis e passados como

argumentos em funções. Os tipos de dados, funções e módulos incluem-se nesta lógica de objectos;

- Quanto ao tratamento dos tipos de dados, a linguagem é *dinamically typed*, ou seja, o tipo de dado de cada identificador é descoberto no momento em que pela primeira vez lhe é atribuido um valor, não necessitando de uma declaração prévia, e é *strongly typed*, que significa que os tipos de dados associados a cada identificador são decisivos nas operações sobre os mesmos, não os podendo tratar como quaisquer outros sem uma prévia e explícita conversão;

- A sintaxe apresenta particularidades interessantes, que muitas vezes são referenciadas como boas práticas na escrita das mais diversas linguagens, para uma boa legibilidade do código produzido, e que nesta linguagem se incorporam na própria sintaxe:

  1. cada instrução é escrita numa única linha, ou seja, não há nenhum sinal especial para finalizar as mesmas (excepto obviamente o próprio *carriage return* do editor);
  2. o início de um bloco de código é identificado pelo uso do caracter ':' à frente da instrução que dá início a esse mesmo bloco, e as instruções que o compõem são identificadas pelo nível de indentação, que tem de ser mantido consistente. Quando se refere blocos de código inclui-se classes, funções, ciclos de repetição (como *for*, *while*) e instruções condicionais (*if...else...*).

4.1.1 Tipos de Dados

Para além dos tipos de dados mais comuns que se encontram na generalidade das linguagens de programação, esta linguagem implementa três tipos de dados que, pelo uso frequente que lhes é dado na programação da ferramenta bem como pelo que lhes será dado no desenvolvimento, se abordam com maior detalhe: dicionários, listas e tuplos.

Um dicionário em Python permite guardar pares chave-valor, e definem-se separando os respectivos pares por vírgulas, associam-se os pares separando com o caracter ':' a chave do valor, e delimitam-se por chavetas, como se exemplifica a seguir:

SomeDict = {1 : 'first' , 2: 'second', 'server' : 'gnomo.fe.up.pt'}

As chaves têm obrigatoriamente de ser únicas, como uma chave primária de uma tabela numa base de dados, mas não têm de partilhar a sua natureza. São sensíveis ao uso de maiúsculas (como todos os identificadores em Python). Pode-se obter os valores a partir das chaves, mas não o contrário. Uma vez que as chaves têm de permanecer únicas, uma atribuição de um valor a uma chave que já exista, modifica o valor que lhe estava associado. Caso não exista, cria um novo par chave-valor. Os elementos num dicionário não dispõem de nenhuma lógica de ordenação. Os valores podem assumir qualquer tipo de dados, como inteiros, strings, objectos ou mesmo outros

dicionários. Quanto às chaves, surgem algumas restrições, pois pela necessidade de serem únicos, só podem assentar sobre tipos de dados imutáveis. A secção 3.1 de [14] detalha os métodos para operar sobre dicionários, desde adicionar, eliminar e substituir elementos individuais, ou limpar totalmente um dicionário.

As listas em Python são semelhantes a *arrays* de elementos noutras linguagens, com a particularidade de poderem manter objectos arbitrariamente (não necessariamente elementos do mesmo tipo) e se poderem expandir dinamicamente à medida que novos elementos são adicionados.

Definem-se envolvendo a lista entre parêntisis rectos e separando os elementos por vírgulas, como ilustra o exemplo seguinte:

SomeList = ['a' , 1 , ['another','list'], 'b']

Encontram-se na secção 3.2 da obra [14] os métodos essenciais para operar com listas, que vão desde o acesso a elementos individuais por via de índices baseados em zero, acesso a subconjuntos de elementos, adição e eliminação de elementos da lista, concatenação de outras listas, procura pela existência de elementos, e os efeitos da aplicação de alguns operadores matemáticos sobre listas.

Os tuplos são vistos como listas imutáveis, ou seja, como que se fossem listas declaradas como constantes. É aconselhável o seu uso para definir um conjunto de elementos que não irá variar ao longo do programa. Têm a vantagem de ficar assim protegidos contra escrita, e o acesso aos mesmos ser mais rápido do que às listas. Declaram-se de forma similar às listas, excepto que se usa parêntisis curvos como delimitadores:

SomeTuple = ('a' , 1 , ['some','list'], ('tuple','inside'), 'b')

Possuem as mesmas características que as listas no que toca aos acessos de leitura, mas não possuem métodos que envolvam escrita, uma vez que são imutáveis. Podem ser usados como chaves de dicionários, pois possuem a característica de imutabilidade necessária, desde que não sejam compostas por nenhum elemento mutável, como no exemplo anterior, que continha uma lista. Os tuplos podem ser convertidos em listas e vice-versa, através das funções **tuple** e **list**. A primeira "congela" uma lista, e a segunda "liberta" um tuplo. A secção 3.3 de [14] aprofunda o uso de tuplos.

4.1.2 Funções

A declaração de uma função em Python faz-se usando um identificador precedido da palavra reservada **def**, seguindo-se entre parêntisis os seus parâmetros, separados por vírgulas, no caso de ser mais do que um:

def SomeFunction(params):

De notar que não é definido nenhum tipo de dado para o retorno. Aliás nem é sequer obrigatório defini-lo. Caso seja usada a palavra reservada **return** ao longo da função, ela retorna o(s) valor(es) designado(s) (o plural justifica-se pois uma função pode retornar de facto um conjunto de valores, devidamente separados por vírgulas após o **return**). Caso não seja usado explicitamente o **return**, a função retorna a abstracção **None**. Isto faz com que não haja, como noutras linguagens, distinção entre funções e rotinas.

Uma outra funcionalidade importante nas funções é o uso de parâmetros opcionais. Quando ao declarar uma função se iguala um parâmetro a um valor, está-se a indicar que esse parâmetro deve assumir esse valor por defeito, e como consequência que a função ao ser invocada prescinde da passagem desse argumento. Mais detalhes sobre a sintaxe correcta para usar esta funcionalidade podem ser encontrados na secção 4.2 de [14].

4.1.3 Classes

A declaração de uma classe em Python é igualmente simples, e não necessita de nenhum tipo de referência externa ou prototipagem prévia. Usa-se a palavra reservada **class** seguido do nome da classe, e opcionalmente a classe da qual deve herdar, caso aplicável (notar que o uso de [] indica a opcionalidade, não fazendo parte da sintaxe):

class SomeClass[(AncestorClass)]:

No interior das classes definem-se os seus métodos, com as mesmas regras de sintaxe das funções, com uma particularidade: o primeiro parâmetro tem de ser sempre um objecto que corresponde à instância invocada, e que por convenção se usa a palavra **self** (que não sendo uma palavra reservada, apenas uma convenção, aconselha-se vivamente a não fazer uso dessa palavra sem este contexto).

É definido também um método especial, **__init__**, que é opcional, mas que se existir é executado imediatamente após ser criada uma instância da classe em questão. Chama-se à atenção para não o interpretar como o construtor de uma classe, conceito muito comum nas linguagens orientadas a objectos, pois na verdade quando o método **__init__** é executado, o objecto já se encontra "construido" e já existe uma referência válida para a instância criada.

Em relação aos atributos, importa saber como a linguagem distingue atributos de classe de atributos de instância. Quando se declara uma classe, todos os atributos aí definidos são interpretados como atributos de classe. Quando se pretende usar atributos ao nível da instância têm de ser declarados no método **__init__**. Os atributos de classe são acessíveis através da referência directa da classe ou através de qualquer uma das suas instâncias.

Para criar uma instância de uma classe basta fazer uma atribuição como se de uma função se tratasse, usando os parâmetros do seu método **__init__** (excepto o anteriormente referido **self**, que está sempre implícito). A excepção a esta regra surge quando se invoca um método da classe pai (quando está presente uma relação de herança) em que se pretende que a classe actual tenha um determinado comportamento, próprio da classe pai, onde aí é necessário explicitamente passar **self** como argumento.

4.1.4 Módulos

O Python dispõe de um conjunto relativamente pequeno das denominadas *built-in functions*. No entanto, a sua distribuição base possui diversos módulos para os mais variados propósitos, desde operações matemáticas (**math**), interação com o sistema operativo (**os**), operações com datas (**datetime**), com variáveis de sistema do próprio interpretador (**sys**), entre muitos outros. Uma lista completa das *built-in functions* bem como dos módulos disponíveis, incluindo alguns específicos de determinadas plataformas, pode ser consultada em `http://docs.python.org/library`.

Importar módulos em Python pode ser feito de duas formas, que cumprem propósitos distintos, e têm consequências na forma como se pode aceder às funções lá definidas. A sintaxe **import *module*** disponibiliza a invocação de todos os objectos definidos em *module* através da sintaxe **module.object**. Já a sintaxe **from *module* import *object*** permite o uso directo do(s) objecto(s), sem usar o prefixo do módulo em causa. Pode também ser usada a sintaxe **from *module* import ***, onde são importados todos os objectos do módulo em causa.

A segunda opção é útil quando se vai fazer uso dos atributos e métodos com frequência, não havendo necessidade de recorrer de cada vez ao prefixo com o nome do módulo, ou então quando apenas se pretende importar um sub-conjunto desses atributos ou métodos. No entanto, quando no módulo actual se usam objectos com o mesmo nome que os usados em módulos importados, tem de ser usada a primeira sintaxe, para evitar conflitos. Em [14] aconselha-se o uso da segunda opção de forma limitada, pois dificulta a legibilidade o código, a percepção sobre em que âmbito estão definidos determinados atributos ou métodos, a correcção ou a reutilização do código. Para além disto, é essencialmente uma questão de estilo, e é frequente encontrar código que usa os dois formatos.

4.2 Módulo *plugger*

Com o estudo efectuado da linguagem fez-se uma introspecção sobre o código da ferramenta, identificando um módulo essencial, o *plugger*, que define os comportamentos dos plugins, desde os seus elementos gráficos aos métodos para geração do código C. Neste contexto, reafirma-se o facto de que a Barra de Ferramentas é do ponto de vista programático um plugin, como já foi referido na secção 2.1.2. O diagrama de classes da figura 4.1 ilustra esse facto, mostrando que a classe pai **PlugTemplate** é uma generalização das classes **PluginsRoot** e **FinalPlugClass**.

A classe **PluginsRoot** é a classe raíz de toda a árvore dos plugins na ferramenta, que graficamente corresponde à barra de tarefas. Herda ainda de uma outra classe, **PLCController** que especifica comportamentos próprios para o SotfPLC, e que se encontra declarada externamente. A classe **FinalPlugClass** surge definida no método **PlugAddChild()** da classe **PlugTemplate** e acaba por ser como que uma derivação da classe **PlugClass** antes desta ser instanciada, permitindo assim que seja executado o método de inicialização da classe **PlugTemplate** e só depois o da classe **PlugClass**, fazendo posteriormente a redefinição de métodos e atributos que se pretendam específicos para determinado plugin.

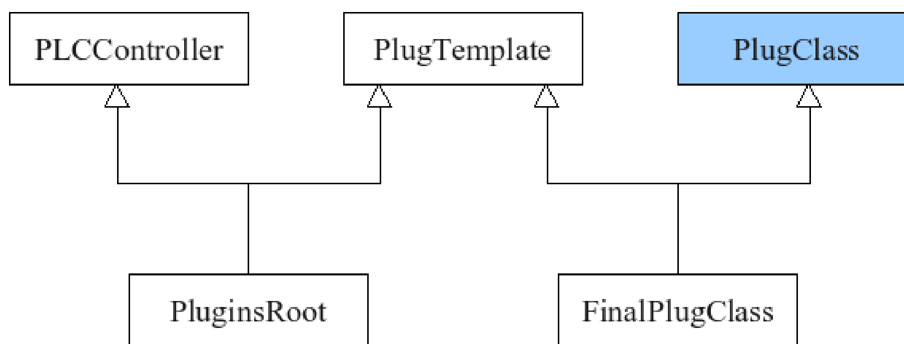


Figura 4.1: Diagrama de Classes UML focando o módulo *plugger*

A classe destacada a azul na figura 4.1 é a classe que efectivamente tem de ser programada para os efeitos do presente projecto, apesar de que a instância final gerada é da classe **FinalPlugClass**, sobre a qual o utilizador opera graficamente. Esta organização permite uma clara independência no desenvolvimento de novos plugins, mantendo o aspecto e funcionalidade base de um plugin.

Coloca-se então a questão de como os plugins são disponibilizados na raíz de um projecto, e que sub-plugins se pode escolher dentro destes. A classe **PlugTemplate** define um atributo - *PlugChildTypes* - que é uma lista de tuplos. Cada tuplo tem três elementos, com o seguinte formato:

(nome da classe na GUI, nome da classe no código, nome a figurar no menú *drop-down*)

Cada elemento da lista corresponde a uma **PlugClass**. Quanto à raíz de um projecto, o que a classe **PluginsRoot** faz na prática é: percorrer todas as pastas contidas no caminho ".../plugins/" em relação à localização do módulo em estudo (plugger), procurar dentro destas um ficheiro de texto "README" que deve ter simplesmente uma *string* com o que se pretende que figure como terceiro parâmetro do tuplo. O nome da classe no código que implementa os plugins instanciáveis a partir da raíz é fixo - **RootClass**. O nome na GUI, que é simplesmente uma *string*, é um elemento que figura no topo da barra correspondente ao plugin em causa, e que só é definido depois de efectivamente instanciado o plugin, após o utilzador fazer uma escolha no menú *drop-down* e digitar um nome para o plugin.

Já cada **PlugClass**, na sua programação, define os seus próprios sub-plugins e classes que os implementam, onde o nome da classe do código é de escolha livre.

Outra questão fundamental prende-se com a forma como são geridos os parâmetros específicos de um plugin. Para cada plugin são mantidos dois ficheiros em formato XML: o baseplugin.xml que mantém os campos que todos os plugins têm de ter (o nome e o *IEC_Channel*); e o plugin.xml, que mantém os parâmetros específicos da instância do plugin em causa. Mais uma vez, a classe **PlugTemplate** implementa um atributo - XSD - que na prática é uma *string* cujo conteúdo corresponde a uma árvore xml, e que cada **PlugClass** redefine ao ser instanciada. Esse atributo é passado como argumento em funções definidas num módulo externo - **xmlclass** - que dispõe de métodos

para transformar a informação constante nessa *string* em elementos gráficos manuseáveis, quando um projecto é aberto, e vice-versa, quando o projecto é gravado.

Cada parâmetro tem um conjunto de campos que se pode definir nessa árvore xml: nome, tipo, obrigatoriedade, valor por defeito, entre outros. Estão implementados diversos tipos de dados, que resultam em interfaces gráficas distintas, usuais em GUI's:

- *string* - caixa de texto livre;
- *combo-box* - tipo originalmente definido como string, podendo-lhe ser associada uma lista em Python, que resulta numa caixa de combinação, limitando o utilizador à escolha das opções;
- *integer* - resulta numa caixa de texto, que limita o utilizador à introdução de dígitos apenas, com setas auxiliares para incrementação e decrementação; Adicionalmente, podem ser definidos limites do intervalo de valores aceitáveis;
- *boolean* - resulta numa caixa de verificação.

A lista é relativamente extensa, e limitou-se a apresentação aos usados na interface do plugin Modbus.

A classe **PlugTemplate** implementa ainda um método muito útil, e muito usado na codificação - **GetParamsAttributes()**. Recebe como parâmetro a instância de um plugin, e retorna uma lista de dicionários enredados com todo o conteúdo da árvore xml. Todos os parâmetros, incluindo os valores definidos pelo utilizador, são assim acessíveis na codificação, com a particularidade que todos, sem excepção são tratados como *strings*.

4.3 Interface Gráfica

A interface gráfica do plugin Modbus inspira-se no diagrama de classes da figura 3.3. Desse diagrama salienta-se a ideia da composição entre as classes apresentadas, que se enquadra claramente com a lógica hierárquica que os plugins oferecem, tanto do ponto de vista gráfico como programático.

Daí resulta a opção de criar três plugins, dependentes entre si em cadeia:

1. **ModbusInstance** - Tem como parâmetros o tipo de dispositivo - *Work_As* - e a implementação do protocolo que deve usar - *Mode*. Corresponde à classe Master pensada na arquitectura, mas surge aqui a ideia de futuramente generalizar este plugin para os vários casos de uso possíveis.
2. **ModbusIPNode** - Tem como parâmetros o Endereço IP e o porto do dispositivo com o qual o mestre deve estabelecer uma conexão. Corresponde à classe Slave/Gateway.
3. **Request** - Tem como parâmetros: a função modbus a usar, o ID do escravo, o endereço do primeiro item a ler ou escrever e o número de items a ler ou escrever.

Estes parâmetros que se oferecem na interface gráfica dos plugins não correspondem à totalidade dos parâmetros anteriormente definidos para cada classe. Os que foram implementados são vistos como os obrigatórios para a parametrização básica de uma rede modbus, e o objectivo de disponibilizar apenas estes é o de minimizar a carga visual da interface criada. Por exemplo, a classe Request tem parâmetros para definir o tempo máximo de espera por uma resposta (*timeout*) e para o número de tentativas a fazer, caso um pedido não obtenha resposta nesse tempo máximo (*number_of_retries*). No entanto, optou-se por considerar estes parâmetros como de utilização avançada, e mante-los no âmbito do *back-end* do plugin Modbus. A figura 4.2 ilustra a solução do ponto de vista gráfico, apresentando uma árvore com uma instância de cada plugin criado.

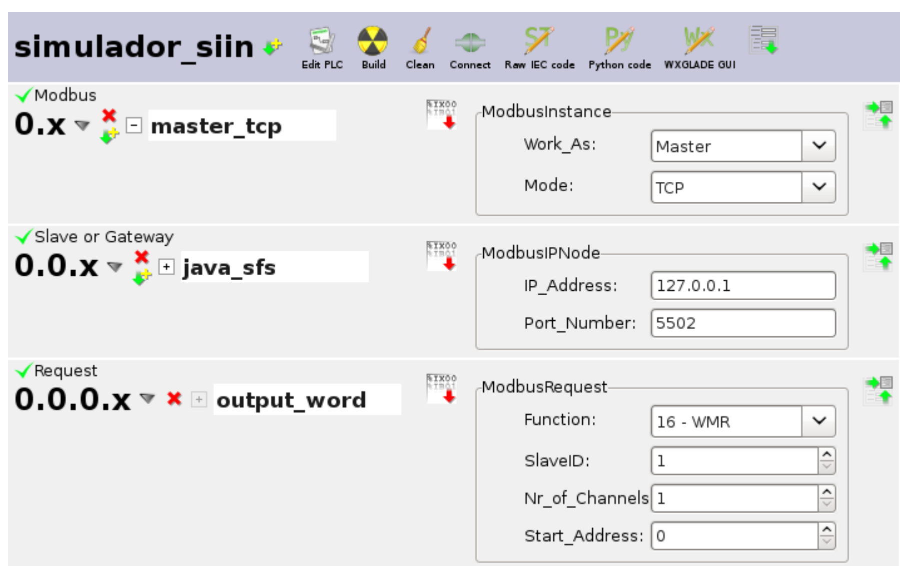


Figura 4.2: Aspecto gráfico da interface criada, com uma instância de cada classe

4.4 Geração Automática do *back-end*

No Beremiz todos os plugins e sub-plugins são vistos como entidades geradoras de código. Isto decorre do facto de que o método *_builder()* da classe que implementa a raíz do projecto para além de efectuar uma série de acções base (gerar o código em ST a partir do ficheiro *.xml resultado da edição no PLCOpen Editor, invocar o MATIEC para gerar o correspondente código C e extrair as *directly represented variables*), percorre todos os plugins e sub-plugins adicionados ao projecto, invocando para cada um o seu método *PlugGenerate_C*.

Este método apresenta a seguinte interface:

```
PlugGenerate_C(self, buildpath, locations):
    ...
    return [(C_file_name, CFLAGS),...] , LDFLAGS_TO_APPEND
```

O argumento *builpath* é uma *string* que indica o caminho onde devem ser escritos os ficheiros gerados, enquanto que *locations* implementa uma lista com todas as variáveis que foram definidas para pertencer ao plugin em causa, e onde cada elemento é um dicionário que comporta toda a informação de cada variável, como mostra o excerto seguinte, retirado da documentação da função:

{"IEC_TYPE": the IEC type (i.e. "INT", "STRING", ...)
"NAME": name of the variable (generally "__IW0_1_2"style)
"DIR": direction "Q","I"or "M"
"SIZE": size "X", "B", "W", "D", "L"
"LOC": tuple of integers for IEC location (0,1,2,...)}

A nomenclatura usada nas variáveis de retorno torna-as auto-explicativas (referir apenas que do ponto de vista de programação são *strings*). Elas conferem a ginástica necessária para indicar ao método *_builder()* do projecto como invocar o compilador C: que opções usar na compilação prévia de cada ficheiro C (que cria um ficheiro objecto a ser ligado mais tarde numa biblioteca dinâmica), e que opções usar na "linkagem" final.

Pelo que foi explicado sobre o método *PlugGenerate_C*, verifica-se que é perfeitamente viável implementar métodos "mudos", retornando uma lista vazia para os pares (C_file_name, CFLAGS) e uma *string* de comprimento nulo para as LDFLAGS_TO_APPEND. Essa foi a estratégia usada para as classes ModbusIPNode e Request, fazendo reflectir toda as configurações necessárias ao plugin Modbus na classe ModbusInstance.

O *back-end* do plugin é programado em C, para a compatibilização com todo o código gerado pelo MATIEC. Sendo o C uma linguagem de programação que não assenta na filosofia da orientação a objectos, tornou-se necessário fazer algumas adaptações à arquitectura proposta no capítulo anterior, mas mantendo globalmente a conformidade com a mesma.

Uma técnica regular consiste em implementar os atributos de uma classe como uma estrutura em C. Os objectos de cada classe são vistos como um elemento de um *array* dessas estruturas, *array* esse cujo tamanho é declarado através de uma macro neste caso concreto. As iterações ao longo dos objectos de cada classe são assim feitas através de simples ciclos *for*, com um índice que percorre todos os elementos. O método da classe Request proposto na arquitectura, denominado de *proceed_mb_request()* recorre a uma estrutura de decisão que perante o campo **mb_function** escolhe que função modbus deve invocar, dentro do leque disponível na API da biblioteca de funções descrita na secção 3.1. A única componente dinâmica nesse ficheiro é o nome dado às funções do plugin (recordar figura 3.2), que para o softPLC distinguir univocamente a interface de cada plugin do projecto, surgem concatenadas com o IEC_Channel à frente do nome, como por exemplo *__init_0()*.

A globalidade da componente dinâmica do código, que fica dependente dos parâmetros introduzidos pelo utilizador, surge:

- na definição das macros;
- na declaração dos *arrays* de estruturas;

- no mapeamento entre as variáveis do projecto IEC 61131-3 e os buffers de dados passados como argumentos nas funções modbus.

Esta componente é escrita num cabeçalho, essencialmente por uma questão de organização do código final. As duas primeiras partes são intuitivas, e limitam-se à trivial sintaxe C.

Quanto ao mapeamento, importa recordar o que é referido na secção 2.1.4, em que se afirma que as variáveis para as quais se define uma localização são declaradas como externas e efectivamente declaradas no âmbito dos plugins. Essa organização obriga a que o plugin declare efectivamente cada variável cuja simbologia usada no campo *location* a coloque no âmbito do plugin em causa. No PLCOpen Editor, para que uma variável seja associada a um sub-plugin Request o utilizador deve usar a simbologia definida na norma, já explicada na secção 2.1.5, e referir-se ao IEC_Channel desse sub-plugin acrescido de um inteiro correspondente à posição de memória no escravo com o qual se vai trocar a informação. Na prática, as variáveis são declaradas como apontadores e igualadas à posição do buffer de dados, com a devida adaptação entre uma lista iniciada em zero, como é um *array* em C, e a real posição de memória no escravo, que é enquadrada pelos parâmetros *start_address* e *count*, como ilustra o exemplo usado na figura 4.3.

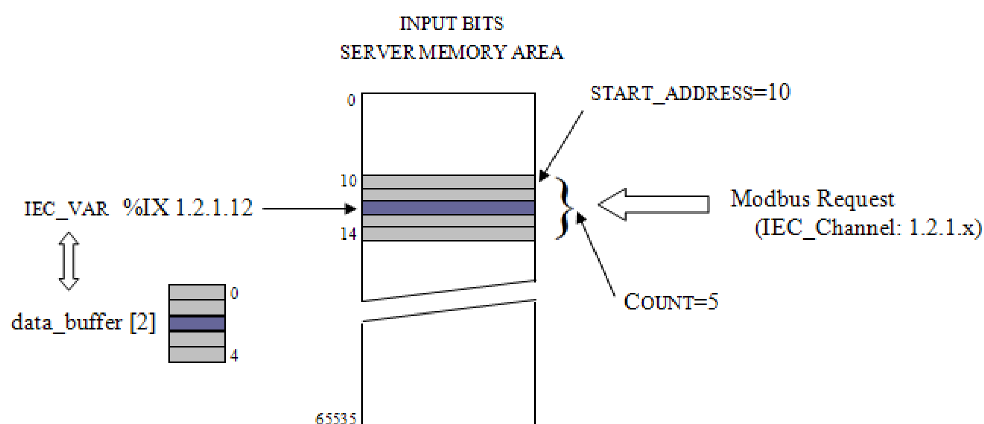


Figura 4.3: Mapeamento entre variáveis do PLC e zonas de memória de um servidor Modbus

O Python dispõe de uma sintaxe para formatação de *strings* muito similar à da família de funções printf em C. A flexibilidade surge no tipo de dados que pode ser passado como argumento. Um caso particular acontece com o uso de dicionários. Ao longo da *string* surgem os locais onde se pretendem ver escritos os argumentos identificados por **%(key-name)type** e passa-se como argumento um dicionário, onde as chaves devem ser iguais aos **key-name** colocados na *string* a ser formatada, e os valores associados a essas chaves o conteúdo que efectivamente se quer passar como argumento, e que deve respeitar o tipo indicado por **type** no exemplo acima. Esses tipos incluem muitas opções, em tudo iguais às que se encontram na família de funções printf em C.

O método PlugGenerate_C baseia-se então em formatar strings em Python com a técnica explicada no parágrafo anterior. Abrem-se os ficheiros *mb_runtime.c* e *mb_runtime.h* pré-formatados

com os componentes estáticos e no lugar dos componentes dinâmicos se encontram as chaves do dicionário que vai sendo criado ao longo do método. São obtidos em momento próprio os elementos da configuração feita pelo utilizador na GUI, para ir construindo o dicionário, e no final escreve-se para os ficheiros *MB_IEC_Channel.c* e *MB_IEC_Channel.h*, que são colocados na directoria passada no argumento *buildpath*. A figura 4.4 confere uma imagem global do descrito.

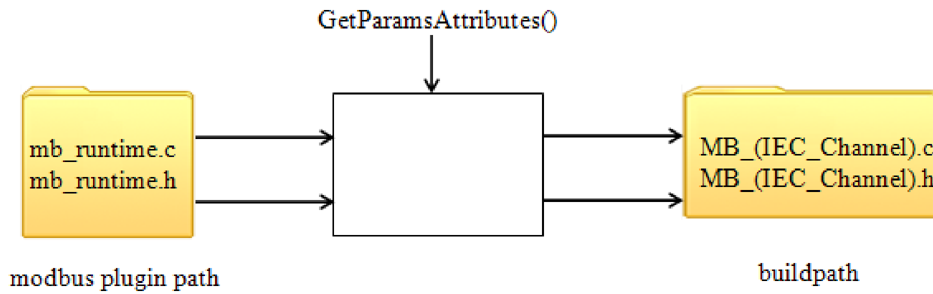


Figura 4.4: Visão externa da manipulação de ficheiros pelo método *PlugGenerate_C()*

No retorno, para além do ficheiro *MB_IEC_Channel.c*, define-se para CFLAGS uma directiva para incluir a localização da pasta que contém a biblioteca de funções (opção -I do compilador), e como LDFLAGS retorna-se os ficheiros *.o necessários no processo de linkagem (mb_master.o , mb_tcp.o e sin_util.o).

Capítulo 5

Testes e Validação

Neste capítulo explicam-se os testes efectuados às duas componentes do ’driver’. Na primeira secção abordam-se os mecanismos que garantem a detecção de não conformidades nos dados introduzidos pelo utilizador face à especificação do protocolo, bem como a informação disponibilizada nesses casos. Na segunda secção, através do desenvolvimento de uma aplicação de controlo concreta, aponta-se para uma validação formal ao funcionamento do *back-end* em ambiente de simulação.

5.1 Validação dos dados introduzidos

A estratégia global na detecção de erros foi a de enviar mensagens de erro para a consola de texto da ferramenta, que surgem destacadas visualmente em cores primárias (letras vermelhas sobre fundo amarelo), e produzir um retorno “mudo” que não interfira na restante geração de código. Deste modo, o utilizador é avisado dos erros cometidos no que diz respeito ao plugin Modbus, mas a compilação e linkagem de todo o projecto não é interrompida, permitindo verificar se os restantes componentes apresentam alguma falha ou não.

Importa lembrar que foram criadas três classes de plugins, que podem ser invocados uns a partir dos outros, numa lógica em árvore - ModbusInstance, ModbusIPNode e ModbusRequest.

No plugin ModbusInstance o utilizador pode definir o tipo de dispositivo - {Master,Slave} - e a implementação a usar - {TCP,RTU,ASCII}. Apesar de só ter sido desenvolvido um caso de uso, imagina-se esta como uma possibilidade de definir o comportamento específico do plugin. Mas de facto, esta opção obriga a que o utilizador seja informado que modos ainda não estão funcionais. Qualquer combinação diferente do par (Master,TCP) imprime uma mensagem de erro. De referir que a funcionalidade gráfica da caixa de combinação impede que o utilizador introduza por digitação uma opção diferente das disponíveis.

No plugin ModbusIPNode encontram-se dois campos para preenchimento. No campo designado por IP_Address foi dada liberdade de preenchimento, definindo-o como uma *string*. Note-se que este tem um nome que pode não reflectir todas as opções do seu conteúdo . O código C interpreta-o igualmente como uma *string* e pode assumir qualquer valor, pois quem irá resolve-lo

é a *stack* TCP. A título de exemplo, pode-se usar o identificador "localhost" para se fazer referência ao IP 127.0.0.1 sem nenhuma diferença. O único teste efectuado é então se a *string* tem comprimento nulo, que produz mensagem de erro, sendo que erros de semântica neste parâmetro só são detectáveis na execução. Quanto ao campo Port_Number ele está definido como inteiro no intervalo [1;65535], e valor 502 por defeito (o porto reservado para o protocolo Modbus). Para lá destas restrições, é uma questão dependente da compatibilização com as configuração do(s) escravos(s).

Em relação ao ModbusRequest, existem quatro campos, mas as validações não se limitam à análise de cada campo em si, mas também de algumas relações entre eles. De seguida apresenta-se a lista das validações que são feitas:

1. O campo da função Modbus tem por base uma caixa de combinação e pelo que foi explicado para os campos do ModbusInstance, é feita apenas a verificação se o campo foi deixado sem opção.

2. O campo SlaveID deve ser usado com o objectivo de endereçar um pedido a um servidor ligado a um barramento série através de uma *gateway*. Nesses casos, são aceites endereços na gama [1;247] (o endereço 0 é usado para mensagens de difusão múltipla que não envolvem resposta). Quando um servidor está directamente ligado à rede TCP/IP o campo é inútil. Neste caso, a especificação [3] numa frase obriga o uso do valor 255 (0xFF), mas no parágrafo seguinte esclarece que é uma recomendação para portabilidade. Pelo exposto, o valor por defeito deste campo é 255 (caso que se assume como mais comum, servidor TCP puro), mas são aceites na GUI valores no intervalo [1;255], sendo que no acto de compilação se produzem mensagens de erro quando são usados valores no intervalo [248;254].

3. A especificação [2] define que os endereços de uma zona de memória é uma questão de implementação de um dispositivo, mas que os endereços lógicos usados pelas funções Modbus se cifram no intervalo [0;65535]. Esse é o intervalo de valores admissível para o campo Start_Address. Como a interface gráfica bloqueia por si só outros valores, não é feita nenhuma verificação adicional a este campo.

4. Para o campo Nr_of_Channels são aceites valores no intervalo [1;2000]. Este intervalo justifica-se no limite inferior pela semântica (uma função modbus vai ler ou escrever pelo menos um item) e no limite superior pelo maior valor entre os limites no número de items a ler ou escrever na funções modbus. Para além disso, é feita uma verificação individual deste valor para cada função escolhida, produzindo uma mensagem de erro nos casos aplicáveis.

5. É feita ainda uma verificação da combinação dos dois últimos campos referidos, por forma a validar que não se vai tentar aceder a endereços fora da gama [0;65535]. Na prática, testa-se se a sua soma se situa no intervalo [1;65536].

6. Por fim, para cada variável IEC 61131-3 cuja simbologia do campo *location* a coloque no âmbito de um sub-plugin ModbusRequest, é verificado se o último inteiro usado se enquadra

no alcance do pedido Modbus que é construido. Exemplificando, se a uma variável se associou a localização %IX0.0.0.10, a verificação passa por avaliar se o valor 10 está no intervalo [Start_Address ; Start_Address + Nr_of_Channels - 1] do Modbus Request com o IEC_Channel (0.0.0.x).

A validação do funcionamento destas verificações baseou-se num projecto que incluía uma instância de cada plugin. Para os campos que incluíam caixas de combinação, como apresentavam um número limitado de escolhas, foram testadas todas as opções. No campo de digitação livre, testado com comprimento nulo e não nulo. Para os campos que envolvem inteiros, foram feitos testes que envolveram as situações limite, inspirados numa técnica descrita em [15], que sugere como um bom sub-conjunto de testes os valores fronteira em situações onde os parâmetros envolvem intervalos de valores inteiros.

5.2 Funcionamento do back-end

Para proceder a uma validação formal ao funcionamento do código gerado optou-se por aplicar um teste integrador na ferramenta, ou seja, desenvolver no Beremiz uma aplicação de controlo para um simulador de uma célula de fabrico [16], que actua como um servidor modbus numa rede TCP/IP. Dessa forma, para além de testar o 'driver' propriamente dito, foi possível também apurar competências no uso do editor de programas.

O simulador foi desenvolvido em java, recorre à biblioteca ***jamod*** [17] para actuar como servidor modbus, e inspira-se nas funcionalidades disponíveis numa estação de trabalho recentemente adquirida pelo Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto. A sua implementação disponibiliza um ficheiro (plant.properties) onde se podem definir aspectos que vão desde as configurações da conexão TCP aos elementos gráficos que devem fazer parte do simulador.

Para o presente teste, usou-se uma configuração com um armazém, os tapetes de interface com o mesmo, uma estação de processamento série. A figura 5.1 ilustra o seu aspecto gráfico. Na interface TCP configurou-se o IP em *loopback* (127.0.0.1) e o porto 5502. Os componentes usados determinam as zonas de memória que ficam disponíveis no escravo.

O armazém dispõe de um registo para indicar o tipo de peça que se quer retirar e de um actuador para proceder ao armazenamento. Os tapetes simples dispõem de um sensor para indicar a presença das peças no meio dos mesmos, e dois actuadores para activar os movimentos de translação. Os tapetes rotativos, para além das propriedades anteriores, dispõem ainda dos sensores para indicar a sua posição a 0 ou 90°, e actuadores para activar a rotação no sentido directo ou inverso. As máquinas dispõem de sensores para detecção da presença de ferramenta na posição de maquinação e três actuadores, dois deles para rotação da torreta que tem as ferramentas, e um para simular a acção integrada de descida da ferramenta à peça e respectiva maquinação.

O registo para o tipo de peça é mapeado num *Output Register*. Todos os sensores descritos são mapeados como *Input Bits* e os actuadores como *Output Bits*. Esses items estão mapeados

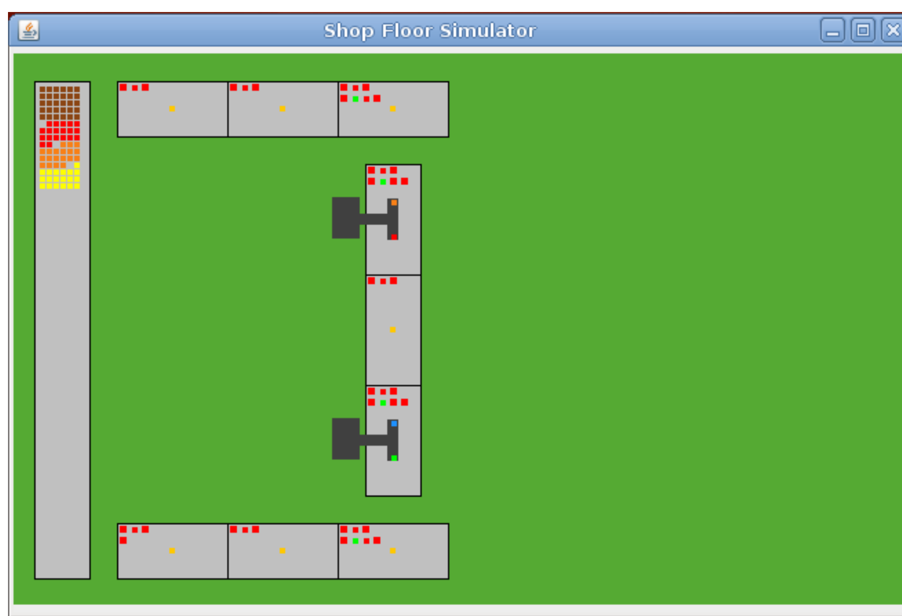


Figura 5.1: Aspecto gráfico do Shop Floor Simulator

em zonas de memória independentes, cada uma delas endereçadas a partir de zero. Esta descrição enquadra totalmente a configuração do plugin modbus.

A aplicação de controlo para este simulador baseia-se num trabalho desenvolvido em ambiente académico, no âmbito da unidade curricular Sistemas de Informação Industriais, cujos detalhes se encontram em [18]. A sua arquitectura estabelece uma divisão topológica dos componentes, com a implementação de *Function Blocks* que gerem o comportamento de cada uma dessas divisões, numa lógica de disponibilização de serviços a uma camada superior que controla todo o processo (o programa principal). Ao utilizador é oferecido um serviço onde pode indicar o tipo de peças a retirar do armazém, o tipo de peças a produzir no final e o número de peças a ser operado. As estações de trabalho têm a inteligência para saber que sequência de operações deve ser efectuada para produzir o tipo de peças finais, em função das peças iniciais.

A execução do algoritmo que controla o simulador comprovou o bom funcionamento do 'driver', o que permite validar o código produzido. Não se exclui a existência de não conformidades, em situações particulares, que só o desenvolvimento de outras aplicações poderá eventualmente confirmar. No entanto, pelo menos em ambiente de simulação, este teste produzido leva a crer que a solução desenvolvida pode ser usada desde já com sucesso.

Capítulo 6

Conclusões

6.1 Satisfação dos objectivos

O objectivo inicial passava por desenvolver um módulo que permitisse ao utilizador do Beremiz configurar o softPLC para funcionar como um dispositivo numa rede Modbus, comunicando com outros dispositivos que usem o mesmo protocolo.

O estudo do protocolo levou à identificação de diferentes casos de uso - Cliente ou Servidor, apoiados nas implementações TCP, RTU ou ASCII. Foi arquitectado, desenvolvido e testado em ambiente de simulação um desses casos de uso - Cliente/TCP.

Optou-se por uma arquitectura simples, mas que mantivesse a possibilidade de com pequenas adaptações se adequar às implementações série do protocolo (RTU e ASCII). Não foi explorado assim todo o potencial da implementação TCP, pois manteve-se o paradigma Mestre-Escravo típico de meios físicos onde o acesso ao mesmo só pode ser gerido por um único dispositivo.

O desenvolvimento envolveu os dois grandes componentes do módulo: uma interface gráfica embebida na ferramenta, apoiada na já existente; e uma outra que produza código C correspondente a uma camada de *software* que use a API da biblioteca de funções Modbus que foi integrada e disponibilize ao softPLC os serviços necessários para a actualização da zona de memória mapeada no Cliente/TCP. Houve o cuidado de produzir código que respeite as boas práticas de modularidade, potenciando a simples modificação ou reutilização do mesmo.

Com os testes levados a cabo, acredita-se que a solução criada possa ter já utilização prática, pelo menos em ambiente de simulação, permitindo dessa forma que sejam testados, corrigidos e validados formalmente projectos IEC 61131-3, interagindo com simuladores que se comportem como servidores modbus. O facto do protocolo ter uma especificação simples fez proliferar as suas implementações nas mais diversas linguagens de programação, o que leva a crer que fica facilitado o desenvolvimento de simuladores com essas características. Aliás, na engenharia, é uma abordagem recorrente o uso de simuladores que representem modelos de processos reais para os quais se pretende desenvolver aplicações de controlo como etapa de validação, prévia à implementação final.

Deixa-se também uma nota importante sobre o editor de programas IEC 61131-3, o PLCOpen Editor, que acabou por ser alvo igualmente de testes, pela aplicação de controlo aí desenvolvida. Este revelou-se de ambiente amigável, uso simples e em conformidade com a norma. Não foram, obviamente, percorridas todas as especificações da norma de forma exaustiva, mas no desenvolvimento da aplicação de controlo foi consultada com frequência a norma em relação a questões de sintaxe para lembrar o seu uso correcto, e que se encontravam algo distorcidas pela prática de desenvolvimento em IDE's que implementam alguns detalhes importantes em não conformidade com a norma.

6.2 Potencial de desenvolvimento

O presente projecto apresenta um potencial de desenvolvimento em diversas frentes, quase independentes, e que por esse facto se ilustram como ortogonais na figura 6.1.

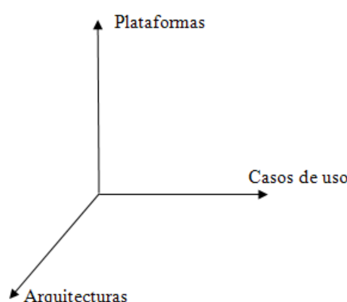


Figura 6.1: Visão sobre as frentes de desenvolvimento futuro do presente projecto

Quanto à questão associada às plataformas, a adaptação passaria por identificar que componentes na biblioteca de funções Modbus são dependentes da plataforma. O desenvolvimento do presente projecto é totalmente independente da plataforma, tanto ao nível da interface gráfica, pelo facto de se apoiar estritamente na linguagem de programação Python, como ao nível do código C gerado, que pela simplicidade da arquitectura do 'driver' se manteve a independência do uso de bibliotecas específicas de um determinado sistema operativo. Sendo um dos objectivos o uso do Beremiz em ambiente académico, seria interessante fazer essa avaliação para o sistema operativo Windows, pela sua ampla difusão.

No que toca ao desenvolvimento de diferentes casos de uso, espera-se que a "semente" lançada neste projecto tenha a robustez necessária para ser vista como um bom alicerce nesse sentido. Os cuidados tidos na concepção da arquitectura do 'driver' para que se mantenha independente da implementação Modbus escolhida, e a modularidade do código produzido tiveram sempre esse objectivo em mente.

Por fim, a possibilidade de implementar diferentes arquitecturas, em especial para o modo TCP do Modbus, é a vertente para a qual o presente projecto menos contribui. No entanto, esta vertente

é vista como muito importante se se pretender uma optimização de recursos, em concreto ao nível temporal, reduzindo o peso que a presente solução acarreta a esse nível quando comparada com uma que use o paradigma Cliente-Servidor no limite das suas possibilidades.

Referências

[1] LOLITECH. Beremiz user manual, 2008. Disponível em http://www.beremiz.org, sob a GNU *Free Documentation License* v1.2.

[2] Modbus-IDA. MODBUS Application Protocol Specification v1.1b, Dezembro 2006.

[3] Modbus-IDA. MODBUS Messaging on TCP/IP Implementation Guide v1.0b, Outubro 2006.

[4] *IEC 61131-3, 2nd Ed. Programmable Controllers - Programming Languages*.

[5] Edouard Tisserant, Laurent Bessard, and Mário de Sousa. An Open Source IEC 61131-3 Integrated Development Environment. *INDIN 2007*, 2007.

[6] Python Programming Language. http://www.python.org/.

[7] Python bindings to the wxWidgets cross-platform toolkit. http://www.wxpython.org/.

[8] WxWidgets. A C++ cross-platform GUI library. http://www.wxwidgets.org/.

[9] PLCopen Technical Committee 6. XML Formats for IEC 61131-3, v2.01. Technical report, PLCOpen, 2009.

[10] PLCOpen TC6 XML Official Schema. http://www.plcopen.org/tc6_xml/tc6_xml_v201.xsd.

[11] Modbus-ida. http://www.modbus.org/.

[12] Erich Gamma. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[13] Guido van Rossum, Fred L. Drake Jr. (editor). *An Introduction to Python (version 2.5)*. Python Software Foundation, 2006.

[14] Mark Pilgrim. *Dive into Python*. 2004. Disponível em http://diveintopython.org, sob a GNU *Free Documentation License*.

[15] Steven R. Rakitin. *Software Validation and Verification: A practicioner's guide*. Artech House, 1997.

[16] André Restivo. Shop floor simulator. Disponível em http://github.com/arestivo/sfs/.

[17] Dieter Wimberger. Java Implementation of Modbus protocol. Disponível em http://jamod.sourceforge.net.

[18] Filipe Ferreira and Vasco Fernandes. Aplicação de controlo de uma célula flexível de fabrico. Trabalho realizado no âmbito da unidade curricular Sistemas de Informação Industriais. Disponível em `http://paginas.fe.up.pt/~ee95230/projecto/siin.pdf`, 2008.