Interface de Utilizador com Placa de Microcontrolador stm32

Guia de Prática Laboratorial

Docentes responsáveis:
ADRIANO TAVARES, JOSÉ MENDES

Licenciatura em Engenharia Eletrónica Industrial e Computadores
Escola de Engenharia
Universidade do Minho

1 Objetivo geral

A utilização de microcontroladores em dispositivos e sistemas electrónicos é atualmente ubíqua. Se abrirmos um dispositivo, é quase certo que encontraremos nele um ou mais microcontroladores. Quase sempre também, os microcontroladores têm necessidade de interagir com outros sistemas, por exemplo, fazendo uso de topologias de ligação em rede, ou então de forma mais simples, recorrendo à comunicação série RS232. Um exemplo de utilização de comunicação série pode ser observado na ligação entre um programa terminal de um computador portátil (PC) e um microcontrolador numa plataforma de desenvolvimento.

O *objetivo geral* desta prática laboratorial (PL) é desenvolver uma interface, do tipo linha de comandos, para a placa com microcontrolador a usar nesta e em PLs seguintes. Esta interface, responde a pedidos na forma de comandos, recebidos através de comunicação série com norma USB ou RS-232 e obedece a uma relação do tipo cliente-servidor:

- Do lado do microcontrolador, o servidor, a interface "escuta" a comunicação série e recebe pedidos efetuados pelo cliente, o PC. A cada pedido/comando recebido, a aplicação no microcontrolador reage executando a operação especificada, retornando valores, ou devolvendo mensagens de erro.

- A aplicação de cliente, executada no PC, é uma aplicação externa ao sistema desenvolvido, e por isso passaremos a designá-la por "sistema externo". Nesta PL, será *apenas* usado um programa terminal, com o propósito de validar a funcionalidade da interface.

No "sistema externo", o *software* está organizado em camadas que asseguram a operacionalidade do mesmo. Encontramos nestas camadas, o *sistema operativo*, que entre várias funcionalidades, reúne um conjunto de serviços prestados às aplicações, nos quais se destaca a abstração do *hardware* subjacente, através de um conjunto de *device drivers*. Estes *device drivers* (ou simplesmente *drivers*), fornecem uma forma simplificada de configuração e o acesso para leitura e escrita, de um dispositivo de *hardware*. É exemplo deste *driver*, o nodo criado no gestor de dispositivos: COMx (e.g., COM1). Este permite a correta configuração da UART[1] no PC, sem que o utilizador necessite de conhecer os seus detalhes de implementação, e a posterior leitura e escrita de forma ordenada. A Figura 1 apresenta um diagrama simplificado, onde se podem observar as partes integrantes de ambos os sistemas.

[1] *Universal asynchronous receiver transmitter*

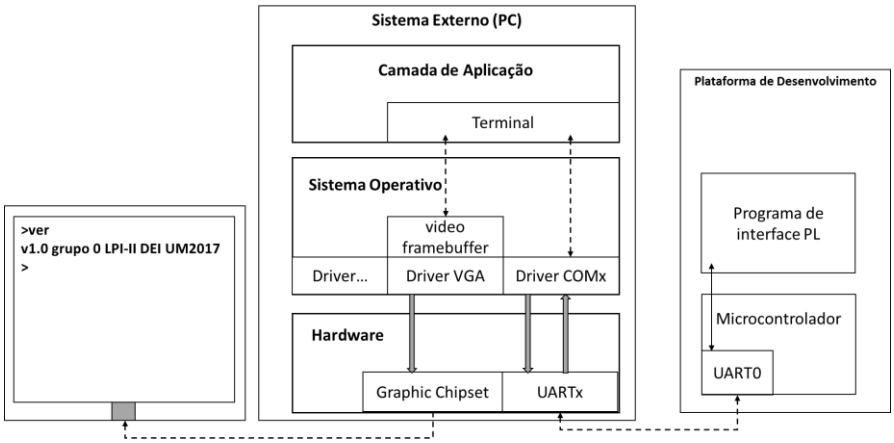

Figura 1 – Diagrama simplificado entre a aplicação no PC e a interface no microcontrolador

No contexto do Projeto Integrador a aplicação no PC será um programa a desenvolver pelos grupos de projeto, com funcionalidades especificas para o utilizador, fazendo uso de interfaces gráficas que se baseiam em ambiente de janelas. Este programa comunicará com o microcontrolador utilizando uma versão similar (melhorada e particularizada) da interface a desenvolver nesta PL. A interface por sua vez, será uma componente do '*firmware'* a desenvolver. Veja-se a Figura 2.

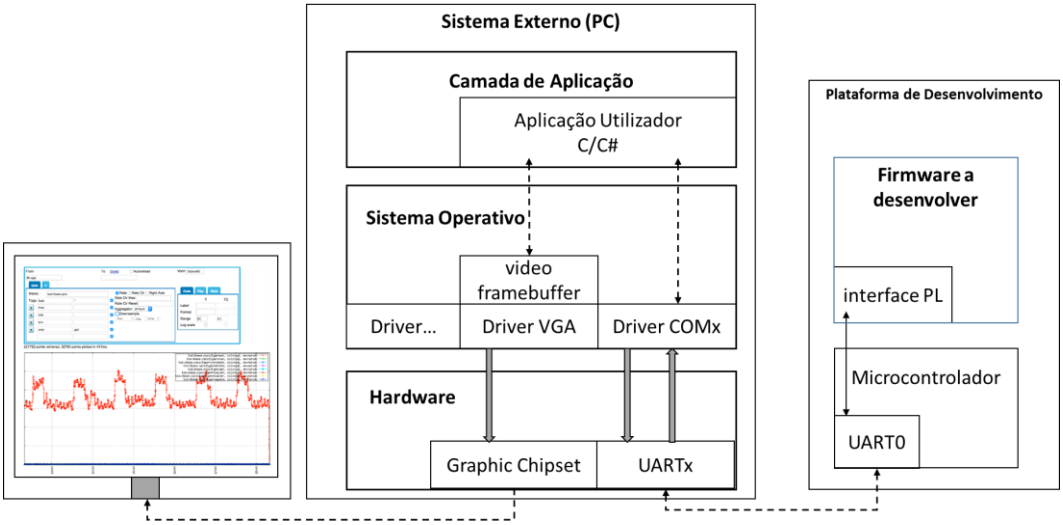

Figura 2 – Diagrama simplificado entre a aplicação no PC e firmware no microcontrolador - projeto integrador

2 Programação concorrente e arquitetura de *software*

Um microcontrolador frequentemente realiza múltiplas tarefas de um programa, alternando a execução entre tarefas, obedecendo a uma política de escalonamento que assegura o bom funcionamento de um dispositivo eletrónico.

Nesta situação, desenvolver um programa para um microcontrolador escrito como um só bloco de código levanta vários problemas, alguns sem solução possível, e torna o programa difícil de depurar e/ou de acrescentar funcionalidades extra, ao logo da vida útil do dispositivo.

Para evitar tais problemas, recorre-se a modelos de *programação estruturada e concorrente*. Neste tipo de modelos o programa a executar é dividido em vários módulos que, conceptualmente, são executados independentemente uns dos outros, podendo ser todos executados num certo intervalo de tempo, donde a designação de "computação concorrente". Dizer que os módulos são todos executados num certo intervalo de tempo não significa que eles são executados ao mesmo tempo[2], mas sim de forma alternada, independente, podendo dizer-se metaforicamente que os módulos "disputam" entre si o tempo de acesso ao recurso CPU. O adjetivo "concorrente" pode, assim, ser interpretado como os módulos concorrerem para a realização das funcionalidades e como os módulos "competirem" pela execução.

Por exemplo, uma forma (não a única[3]) de realizar programação e computação concorrente é utilizar as funcionalidades de interrupção de um microcontrolador. Como se poderá saber, o chamado "programa principal", mais exatamente o *main thread*, inicia execução quando um sinal externo ativa a entrada de *reset*, forçando a unidade processadora central a extrair a instrução a executar do endereço lógico 0.

Se permitida, a ativação de uma entrada de interrupção provoca que o estado da computação no *main thread* seja guardado e seja iniciado um *thread* executando um módulo associado à entrada de interrupção ativada. Um módulo com esta associação designa-se em Inglês por *interrupt service routine* (ISR). Usualmente, uma ISR terminará com uma instrução que retorna a execução ao *main thread*.

Assim, um evento de interrupção permitido força a alteração de curso de um programa, suspendendo a execução corrente e lançando a execução da ISR respetiva, o que permite ao microcontrolador responder de imediato ao acontecimento que provocou a interrupção, como seja a passagem de um intervalo de tempo, a interação com o utilizador ou a alteração de uma grandeza física.

Em princípio, uma ISR deve fornecer uma execução rápida e eficiente, que devolva a execução ao programa suspenso no mais curto espaço de tempo, reestabelecendo os serviços de interrupção na sua plenitude. Esta condição evita que o sistema esteja sujeito a longos períodos de indisponibilidade, devida à execução no contexto de interrupção. Algumas interrupções podem ser priorizadas, permitindo-se que a execução de um módulo de interrupção, digamos de nível 2, seja interrompida por outra interrupção, digamos de nível 1. Neste caso, quando o módulo de nível 1 termina, a execução retorna ao módulo de nível 2.

Desta forma é possível distribuir a programação das tarefas a executar por módulos que podem ser concebidos e codificados independentemente uns dos outros, exceto pela necessária comunicação entre eles, realizada pela escrita e leitura de posições de memória (variáveis do programa) comuns aos módulos que comunicam.

Em aplicações de microcontroladores, um padrão recorrente de arquitetura de *software*, consiste em distribuir o programa em módulos. Um exemplo apresenta-se na Figura 3

[2] Este é um conceito diferente: execução paralela por vários processadores.

[3] Encontra facilmente na literatura termos como multithreading, multiprogramming e multiprocessing que implementam conceitos similares, embora com abordagens tecnológicas distintas.

que tem três módulos. O primeiro módulo corre como função *main* e os restantes dois correm como ISRs. A função *main* inicializa o sistema e implementa as funções de interface com um utilizador humano ou, em geral, com o "mundo exterior". O módulo que corre na ISR_ADC, faz a aquisição de todos os sinais de entrada, colocação em memória e, eventualmente, o seu processamento, gerando valores que podem ser utilizados para ativar saídas. O módulo que corre no ISR_UART, realiza a receção e envio das mensagens trocadas com o "sistema exterior".

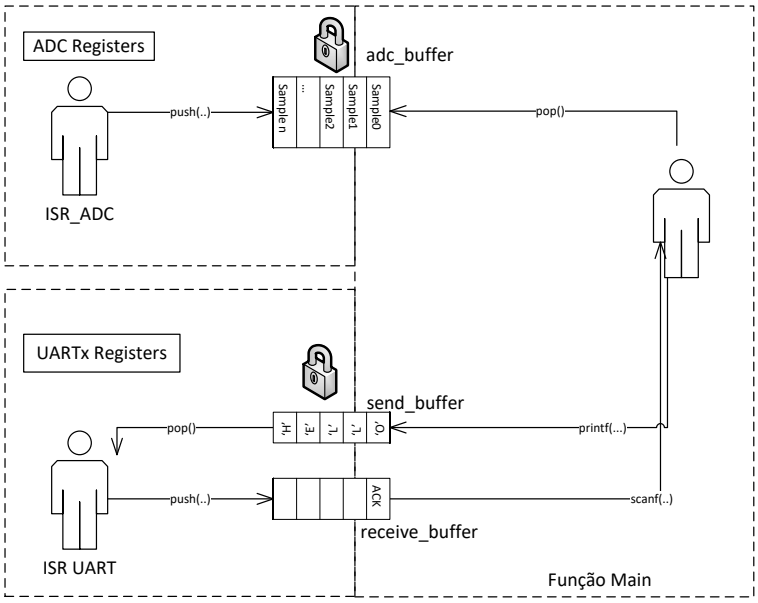

Figura 3 - Estrutura exemplo de uma aplicação de microcontrolador

Neste exemplo, a comunicação entre módulos também é fácil de imaginar. O módulo principal envia valores de parâmetros e variáveis que recebe através da interface para as ISRs, escrevendo-os em posições de memória partilhada com os mesmos (*array* ou FIFO). Inversamente, se comandos recebidos na interface o requererem, a função *main* lê valores de variáveis correspondentes a sinais de entrada e responde com valores e algumas mensagens de contexto.

Modelos mais elaborados de programação concorrente, baseiam-se em ambientes multitarefa e permitem implementar formalmente o conceito de *thread* como um mecanismo disponível no sistema operativo. Tal dá ao utilizador uma forma simples de estruturar uma aplicação num ambiente de execução concorrente, com uma política de escalonamento local. A utilização de um tal ambiente disponibiliza inúmeras funcionalidades, implica, por outro lado, um acréscimo na imprevisibilidade(4), maior *footprint* de memória e *overhead* no processamento.

(4)A imprevisibilidade é uma característica intrínseca de ambientes multitarefa e de programação concorrente, e obrigatoriamente tem de ser tida em conta em ambientes *single-core* e sem o suporte de um sistema operativo de tempo real (como é o nosso caso). Assim, e no ambiente em que se pretende trabalhar, o código é explicitamente escrito para garantir a intercalação de tarefas e a sinalização intertarefas, (funções transparentes em ambientes *multi-threading* nativos) de acordo com as necessidades específicas do problema. Esta forma de escrita tem como consequência o facto de que a ordem de escrita das linhas de código pode não corresponder à ordem temporal de execução das tarefas que lhes correspondem.

Arquitetura de *software* a utilizar

Para além da realização do programa de interface, pretende-se que nesta PL os alunos pratiquem a programação concorrente, a qual será necessária em PLs seguintes. Para tal o programa deverá ser particionado em 3 módulos, um *main* e dois de interrupção, para além dos procedimentos associados à execução de comandos.

A arquitetura representada graficamente na figura 4, obedece a um modelo denominado produtor/consumidor, caracterizado por:

- Dois processos independentes que comunicam entre si utilizando uma zona de memória partilhada

- Os dois processos podem correr a velocidades diferentes, desde que o dimensionamento da zona de memória partilhada seja compatível com os fluxos de dados em jogo

- O processo denominado de Produtor, produz informação que é depositada numa zona de memória partilhada, podendo apenas escrever nesse espaço

- O processo denominado de Consumidor, consome informação residente no espaço de memória partilhado, podendo apenas ler esse espaço.

Na figura 4 identificam-se de forma muito simplificada dois modelos produtor/consumidor que se enquadram no desenho que se pretende implementar. Esta representação é uma possibilidade de implementação, pelo que haverá outras abordagens alternativas que podem ser implementadas.

As comunicações estão conceptualmente representadas como dois processos (um produtor e um consumidor), implementados por interrupção.

No processo de receção, a chegada de um caracter desencadeia uma interrupção que coloca o caracter recebido num buffer de receção. Eventualmente o processo de receção poderá sinalizar a chegada de um comando via canal série, pronto a ser analisado por outro processo, neste caso o *main*.

No processo de transmissão, sempre que o sistema físico está em condições de transmitir um caracter, e desde que existam caracteres no buffer de transmissão, é desencadeada uma interrupção que inicia então o processo de transmissão do caracter em memória.

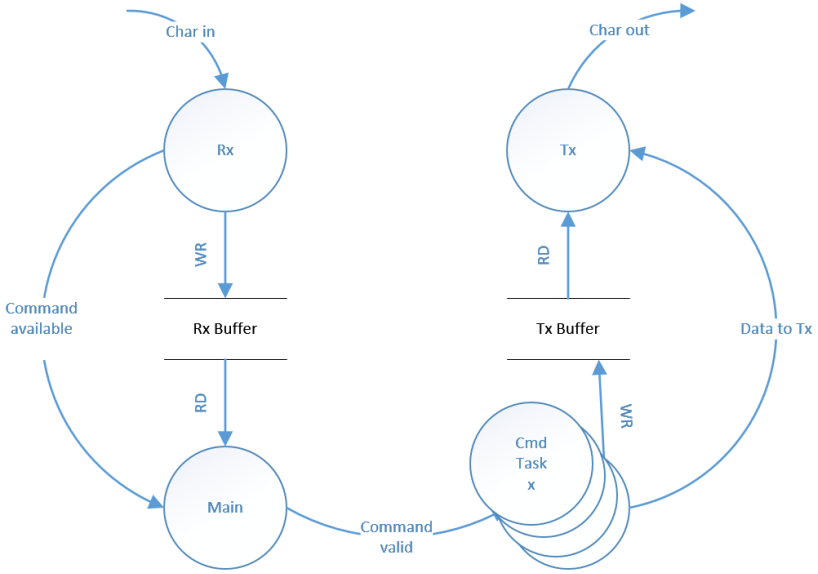

Figura 4- Modelo conceptual da arquitetura de software

O módulo *main* será responsável pela inicialização do sistema e pela análise sintática dos comandos recebidos, no sentido de determinar se eles são ou não válidos, e neste último caso desencadear a execução dos comandos recebidos, ativando um outro processo independente.

Cada processo associado a um comando no modelo apresentado, comporta-se como um Produtor, que ao executar a função que lhe está associada deve gerar uma resposta que será devolvida através do canal série. O conjunto de caracteres dessa resposta é então depositado no buffer de transmissão e é sinalizado o facto de existir informação no buffer pronta a ser transmitida.

Notar que existem diversos processos produtores associados a comandos, mas apenas um processo estará (eventualmente) ativo em qualquer momento.

A ativação deste sinal vai iniciar o processo Consumidor que faz a transmissão dos caracteres para o canal série.

De notar que todos estes processos são independentes entre si, e operam de forma totalmente assíncrona.

A especificação da arquitetura feita acima deixa por definir muitas questões sobre as quais os alunos deverão tomar decisões na escrita do *software*. Por exemplo, o módulo 1 deve apenas fazer a transferência de carateres ou deverá também fazer processamento de ‘parsing’ ou verificação da correção da *string* de carateres recebida? E se sim, o que deve fazer se a *string* não for reconhecida como um comando? Este é apenas um pequeno exemplo das muitas questões que se podem pôr e que se podem responder de muitas formas, não sendo óbvio que alguma opção é melhor do que outra.

3 Os comandos da interface a implementar

O programa de interface deve implementar no microcontrolador um interpretador de comandos recebidos via porta série. O conjunto de comandos a implementar, constituindo um pequeno "monitor" de um microcontrolador, apresenta-se na tabela seguinte.

Comando	Descrição	Prioridade / sequência de implementação
MR	Memory Read – Lê e envia para o computador um segmento de memória (que pode ser só um byte).	1
MW	Memory Write – Escreve um valor num segmento de memória (que pode ser só um byte).	1
MI	Make port pin Input – programa pinos de uma porta como input	2
MO	Make port pin Output – programa pinos de uma porta como output	2
RD	Read Digital Input – Lê e envia para o computador o valor dos bits especificados de uma porta.	2
WD	Write Digital Ouput – Escreve um valor de (até) 8 bits numa porta de output.	2
RA	Analog Read – Inicia a conversão, lê o valor resultante e envia para o computador como valor inteiro.	3
<BCKSP>	Backspace – Limpa o último carácter recebido.	4
<ESC>	Abort – Limpa todos os caracteres recebidos.	4
$	Limpa todos os caracteres recebidos e repete o último comando válido.	4
?	Help – Fornece uma lista dos comandos válidos.	4
VER	"version" – Devolve uma string que identifica claramente a versão de firmware e se necessário do hardware da arquitetura, bem como o turno e o grupo de trabalho	4

Dada o tempo relativamente curto para a realização desta PL, apresenta-se na terceira coluna a prioridade / sequência a seguir na sua implementação.

Gramática dos comandos

Todos os comandos implementados devem respeitar a seguinte gramática:

COMANDO = <char>$^{+}$ <CR> | <char>$^{+}$ <control char>
<char> = {'a'..'z','A'..'Z'} ∪ [0..9] ∪ {β, <bckspc>}
<control char> = {<esc>,'$'}
´β´ = 20h
<esc> = 1Bh
<bckspc> = 08h
´$´ = 24h
<CR> = 0Dh

Sintaxe dos comandos

Usando a gramática anterior, a sintaxe dos comandos a implementar é:

Memory Read: <char>$^{+}$="MRβ <addr>β<length>"
Ler <length> posições de memória, a partir do endereço <addr>

Memory Write: <char>$^{+}$= "MWβ <addr>β<length>β<byte>"
Escrever a palavra de 8 bits <byte>, a partir da posição de memória <addr> durante <lenght> posições.
Exemplo: Escrever a partir de endereço de memória 100h 10 bytes com o valor 0AAh:
MWβ0100β0AβAA

MakePinInput: <char>$^{+}$= "MIβ<port addr>β<pin setting>"
Na porta de endereço <port addr>, configurar os pinos cujos bits que estão a '1' em <pin setting>, como pinos de entrada.
Exemplo: Programar os pinos 1, 3 e 6 da porta 1 como input
MIβ01β4A

MakePinOutput: <char>$^{+}$= "MOβ<port addr>β<pin setting>"
Na porta de endereço <port addr>, configurar os pinos cujos bits que estão a '1' em <pin setting>, como pinos de saída

Read Dig Input: <char>$^{+}$="RDβ<port addr> β <pin setting>
Ler da porta <port addr> o valor digital dos pinos a que corresponde o padrão de bits a '1' em <pin setting>.
Os pinos correspondentes aos bits que estão com o valor '0' em <pin setting> deverão sempre devolver o valor '0'.

Write Dig Output: <char>$^{+}$="WDβ<port addr> β <pin setting> β <pin values>
Na porta de endereço <port addr> escrever os bits de <pin values> nos pinos correspondentes da porta, que se encontram a '1' em <pin setting>.
Os pinos da porta correspondentes aos bits que estão a '0' em <pin setting>, não devem sofrer qualquer alteração no seu valor.
Exemplo: Escrever na porta 1, bits 3 e 7, os valores 0 e 1 respetivamente. Os restantes bits permanecem inalterados.
WDβ01β88β80

Analog Read: <char>$^{+}$="RAβ<addr3>
Obter a representação digital do valor analógico presente no canal <addr3> do ADC, utilizando um modo de funcionamento de conversão simples (*single conversion mode*), uma interrupção no fim da conversão para conclusão do processo de leitura e alinhamento dos dados lidos à direita (informação de conversão disponível nos bits D0 ..D11 do registo de dados do ADC).

Em que:

<start>,<end>,<addr>,<org>,<dest>,<value> = <16 bit value>
<16 bit value> = {“0000”..”FFFF” }
<length>,<byte>,<port addr>,<pin setting>,<pin values> = { “00”..”FF” }
<addr3> = {“00”..”10”}

NOTA: Por questões de simplificação assume-se que todos os valores envolvidos estão expressos em hexadecimal.

Resposta *e prompt* da interface

A resposta do programa da interface a um comando que pede uma leitura será naturalmente o envio dos dados requeridos com uma formatação conveniente.

A resposta do programa da interface a um comando que pede uma escrita poderá não ser nenhuma, mas, mais convenientemente, deverá confirmar ao utilizador que a operação foi concluída com sucesso, fazendo por exemplo a leitura das posições que foram escritas e devolver esse resultado ao utilizador, ou em alternativa, comparar o resultado com o que se pretendia escrever.

A resposta a uma sequência de caracteres que não constitui um comando interpretável (comando ilegal), deverá indicar tal facto ao utilizador.

Em qualquer caso, e à semelhança de outros interfaces baseados em linha de comando, o sistema deverá retornar um *prompt*, que dá informação ao utilizador que a interface está pronta para aceitar um próximo comando. Naturalmente, o *prompt* deverá ser apresentado após inicialização.

Suponha-se que o caractere de *prompt* é “>”. Após inicialização aparecerá no Hyper-Terminal do utilizador o *prompt*:

>

Se o utilizador enviar o comando que pede a versão do programa de interface, o Hyper-Terminal poderá apresentar:

>ver
v1.0 grupo 0 LPI-II DEIC UM2021
>

Se o utilizador enviar o comando que pede a escrita dos bits 0 e 1, na porta 2, a 1, o Hyper-Terminal poderá apresentar:

>WD 02 03 03
written digital output
>

Repare que neste último exemplo, a resposta do sistema replica os dados originais do comando. Embora isto não signifique obrigatoriamente que o comando foi bem executado, pelo menos dá uma indicação ao operador que os parâmetros de entrada foram bem interpretados pelo sistema.

4 Objetivos a apresentar em aula

1. O primeiro objetivo a ser realizado e apresentado é a *descrição de alto nível do programa a ser desenvolvido*, incluindo o programa principal e as rotinas de interrupção. Esta descrição pode ser feita em fluxograma ou em pseudocódigo de alto nível. Uma vez esta descrição validada pelo docente, passar-se-á à *codificação* do programa.

2. O segundo objetivo a ser realizado e apresentado é a *implementação* dos comandos de interface descritos acima, de acordo com a sintaxe definida e com a prioridade de realização indicada. Deve ser demonstrado à evidência que as ISR's de receção e de transmissão, bem como o *main* estão construídos de acordo com os pressupostos, e deve também ser demonstrado o seu funcionamento.

Nota. Para nos assegurarmos que o programa principal está a executar, pode-se programar uma rotina que simplemente faça piscar um díodo LED da placa. Se o LED apagar ou se se mantiver continuamente aceso, tal significará, em princípio, que se perdeu controlo da execução.

Bibliografia

Wikipedia (2017). *Concurrent Computing*. Wikimedia Foundation. https://en.wikipedia.org/wiki/Concurrent_computing

Wikipedia (2010). *Non-Blocking Algorithm*. Wikimedia Foundation. http://en.wikipedia.org/wiki/Non-blocking_algorithm