

Interface com o Utilizador para Placa de Microcontrolador

Guia de Prática Laboratorial

Autores:


SÉRGIO LOPES, ADRIANO TAVARES

Docentes responsáveis:

ADRIANO TAVARES, SÉRGIO LOPES

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

Informação de direitos de autor:

Universidade do Minho  Licença Creative Commons 3.0 Attribution Share-Alike

1 Objectivo geral

A utilização de microcontroladores em dispositivos e sistemas electrónicos é actualmente ubíqua. Se abriremos 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 uma ligação em rede, ou então de forma mais simples, recorrendo à comunicação série RS232. Um exemplo de comunicação série é a que acontece entre um programa terminal de um computador portátil (PC) e um microcontrolador numa placa de desenvolvimento.

O objectivo geral desta prática laboratorial (PL) é desenvolver uma interface com o utilizador do tipo linha de comandos para microcontroladores.. Os comandos a implementar constituem um pequeno “monitor” de microcontrolador. A interface utiliza comunicação série (com ligação USB ou RS-232) e segue o modelo cliente-servidor:

- O microcontrolador é o servidor que “escuta” a comunicação série e recebe pedidos efectuados pelo cliente. A cada pedido/comando recebido, a aplicação no microcontrolador reage executando a operação especificada, retornando valores, ou devolvendo mensagens de erro.
- O cliente é o PC e a aplicação que nele é executada é externa ao sistema a desenvolver: é *apenas* um programa terminal, que serve o propósito de validar a funcionalidade da interface.

Pretende-se desenvolver a aplicação de interface utilizando a placa KIT8051USB com um microcontrolador simples de 8 bits, e depois migrar a aplicação para uma placa STM32. Um objectivo importante é isolar e maximizar o código que é independente das placas a utilizar. Esse código constitui uma camada superior da aplicação e que é utilizável em qualquer placa/microcontrolador programável em C padrão. O restante código constitui a camada inferior e que é dependente da placa e do microcontrolador,

Os sistemas baseados em microcontrolador interagem com o mundo físico real e têm tipicamente múltiplos inputs e outputs. Os acontecimentos do mundo real sucedem-se a qualquer momento e é necessário que o sistema seja capaz de lidar eles de forma assíncrona. No caso, desta PL isso corresponde a poder receber os caracteres da comunicação série de um novo comando enquanto o comando anterior está a ser executado e o resultado produzido. Para tal, pretende-se implementar uma arquitectura de software adequada.

2 Pressupostos

Este trabalho requer os seguintes conhecimentos prévios:

- Saber escrever e depurar código C que envolva funções.
- Utilizar ferramentas de desenvolvimento para a placa KIT805USB, como por exemplo o IDE “Keil uVision” e o gerador de configurações “Configuration Wizard 2”.
- Identificar e utilizar informação de referência (e.g., *datasheet*) sobre o microcontrolador e a placa.

- Fazer uma configuração base da família de microcontroladores 8051, nomeadamente relógio e portos de IO.
- Fazer a configuração de periféricos como UART e ADC.
- Configurar o controlador de interrupções e escrever rotinas de atendimento a interrupção.

3 Programação concorrente e arquitectura de software

Um processador executa diferentes fluxos de instruções. Em processadores multicore esses fluxos podem ser executados em paralelo, enquanto em processadores com um único CPU, como é o caso dos microcontroladores, são executados alternadamente. Os fluxos de execução “competem” pelo CPU e daí o termo “programação concorrente”.

Um fluxo de execução pode corresponder a um processo ou *thread* num sistema operativo genérico, que executa esses fluxos de acordo com uma determinada política de escalonamento. No contexto dos microcontroladores, os sistemas operativos de tempo-real (SOTR) oferecem o mecanismo de tarefa, e na ausência de um SOTR (ou programação *bare-metal*), esses fluxos designam-se por rotinas e estão “penduradas” em interrupções.

A rotina principal inicia execução quando um sinal externo activa a entrada de *reset*, forçando o CPU a extrair a instrução a executar do endereço de programa 0. Essa rotina é a que executa a função *main* de um programa em C/C++.

A activação de uma entrada de interrupção, que esteja habilitada, provoca que o estado da computação da rotina principal seja guardado e seja iniciada a execução de uma rotina associada à entrada de interrupção activada. Essa rotina designa-se em Inglês por *interrupt service routine* (ISR), e geralmente termina com uma instrução que retorna a execução à rotina principal.

Assim, um evento de interrupção força a alteração do curso do programa, suspendendo o fluxo de execução corrente e lançando a execução da respectiva ISR. Isto permite ao microcontrolador responder de imediato ao acontecimento que provocou a interrupção, como seja a passagem de um intervalo de tempo, uma interacção 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 uma ISR, digamos de nível 2, seja interrompida por outra interrupção, digamos de nível 1. Neste caso, quando a ISR de nível 1 termina, a execução retorna à ISR de nível 2.

Desta forma, é possível distribuir a programação do microcontrolador por rotinas, que devem ser concebidas e codificadas de forma que maximize a independência e isole a necessária comunicação entre elas. Um exemplo apresenta-se na Figura 3 que tem três rotinas:

- A rotina da função *main*, que inicializa o sistema e implementa as funções de interface com um utilizador humano ou, em geral, com o “mundo exterior”.

- A rotina `ISR_ADC`, que faz a aquisição de todos os sinais de entrada, colocação em memória, e, eventualmente, o seu processamento gerando valores a ser utilizados para activar saídas.
- A rotina `ISR_UART`, que realiza a recepção e envio das mensagens trocadas com o “sistema exterior”.

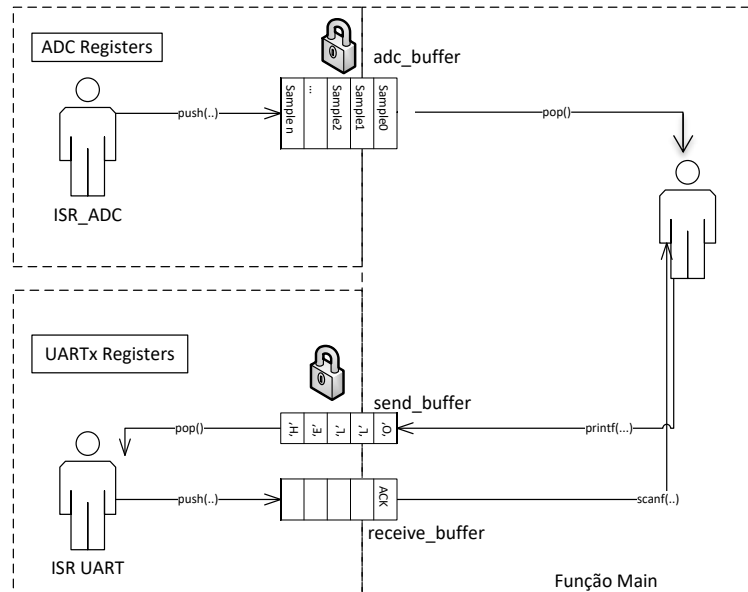


Figura 1 - Estrutura exemplo de uma aplicação de microcontrolador

Neste exemplo, a comunicação entre rotinas também é fácil de imaginar. A rotina principal partilha com a rotina da porta-série os *buffers* com as mensagens, de onde lê as mensagens recebidas e onde escreve as mensagens a enviar. A rotina `ISR_UART` faz o simétrico. A rotina `ISR_ADC` partilha, com a rotina principal, memória onde escreve valores obtidos das entradas analógicas.

4 Estruturação de código

A estruturação do código em módulos, ou pares de ficheiros `.h` e `.c` na linguagem C, é fundamental para a legibilidade, facilidade de manutenção e evitar a repetição de código.

Os conceitos de módulo e rotina são independentes/ortogonais: uma rotina é um elemento de execução de um programa, e um módulo é um elemento de estrutura física do código. Por isso, é normal a função “principal” de uma rotina invocar funções de outros módulos, e que as funções de um módulo sejam utilizadas pelas funções de mais que uma rotina. Isto é normal acontecer também em ambientes multitarefa, como é o caso de aplicações baseadas em sistema operativo de tempo-real.

A partilha de variáveis entre diferentes módulos requer a definição num único módulo (sem *extern*) a declaração com *extern* nos outros módulos.

Arquitectura de *software* a utilizar

Com o desenvolvimento do programa de interface, pretende-se praticar a programação concorrente, a qual será necessária em PLs seguintes. Para tal, o programa deverá ser dividido em 3 rotinas, uma principal e duas de atendimento a interrupção.

A arquitectura representada graficamente na figura 4, segue um modelo denominado produtor/consumidor, caracterizado por:

- um espaço de memória partilhada;
- um Produtor que produz informação e a adiciona à memória partilhada;
- um Consumidor que consome/retira informação da memória partilhada;
- o Produtor e Consumidor podem ser executados a cadências diferentes e a memória partilhada deve estar dimensionada para os fluxos de dados que se pretende suportar.

A comunicação série é realizada pela rotina Rx_ISR, que trata da recepção, e a rotina Tx_ISR, que trata do envio. A figura 4 contém 2 ocorrências (ou instâncias) do modelo produtor/consumidor:

- O da esquerda, que partilha o buffer Rx entre a rotina Main (no papel de consumidora) e a rotina Rx_ISR (produtora).
- O da direita, que partilha o buffer Tx entre a rotina Main (no papel de produtora) e a rotina Tx_ISR (consumidora).

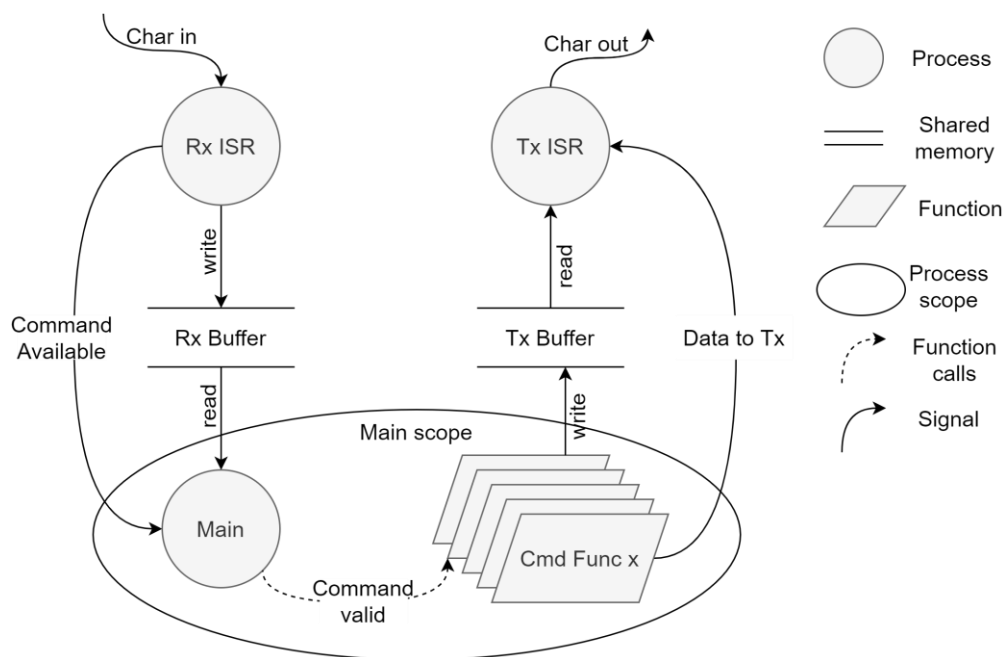


Figura 2 - Arquitectura de software a utilizar.

A chegada de um carácter à porta-série desencadeia uma interrupção que executa a rotina de recepção. Esta rotina coloca o carácter recebido num buffer de recepção e, eventualmente, poderá sinalizar a chegada de um comando, para ser analisado pela rotina Main.

O fim da transmissão de um caractere pelo hardware desencadeia uma interrupção que executa a rotina de envio. Esta rotina verifica se há mais caracteres no buffer de transmissão para enviar. Eventualmente, a rotina principal poderá sinalizar a existência de uma nova resposta para ser enviada.

A rotina Main é responsável pela inicialização do sistema e pela análise sintáctica dos comandos recebidos, no sentido de determinar se eles são ou não válidos. Caso um comando seja válido, esta rotina desencadeia a execução do comando invocando a função respectiva indicada na figura como “Cmd Func”. Cada função que implementa um comando coloca o conjunto de caracteres da resposta no buffer de transmissão. Consequentemente, a rotina sinaliza que existe informação pronta a ser transmitida.

A especificação da arquitectura feita acima deixa por definir muitas questões sobre as quais é necessário tomar decisões. Por exemplo, a rotina de recepção deve apenas fazer a transferência de caracteres ou deverá também fazer processamento de *parsing* ou verificação da correcção da *string* de caracteres 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.

5 Os comandos da interface a implementar

O conjunto de comandos a implementar, 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 Output – 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
?	Help – Fornece uma lista dos comandos válidos.	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 = $\langle \text{char} \rangle^+ \langle \text{CR} \rangle$

$\langle \text{char} \rangle = \{[a..z], [A..Z]\} \cup [0..9] \cup \spadesuit$

$\langle \text{CR} \rangle = 0Dh$

$\spadesuit = 20h$ (espaço)

E todos os valores ou argumentos são representados em hexadecimal.

Sintaxe dos comandos

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

Memory Read: $\langle \text{char} \rangle^+ = MR\spadesuit\langle \text{addr} \rangle\spadesuit\langle \text{length} \rangle$

Ler $\langle \text{length} \rangle$ posições de memória, a partir do endereço $\langle \text{addr} \rangle$.

Exemplo: ler 10 bytes a partir de endereço de memória 100h:

MR 0100 0A

Memory Write: $\langle \text{char} \rangle^+ = MW\spadesuit\langle \text{addr} \rangle\spadesuit\langle \text{length} \rangle\spadesuit\langle \text{byte} \rangle$

Escrever a palavra de 8 bits $\langle \text{byte} \rangle$, a partir da posição de memória $\langle \text{addr} \rangle$ durante $\langle \text{length} \rangle$ posições.

Exemplo: Escrever 10 bytes a partir de endereço de memória 100h com o valor AAh:

MW 0100 0A AA

Make Pins Input: $\langle \text{char} \rangle^+ = PI\spadesuit\langle \text{portAddr} \rangle\spadesuit\langle \text{pinsSetup} \rangle$

Na porta de endereço $\langle \text{portAddr} \rangle$, configurar como entrada os pinos cujos bits estão a '1' em $\langle \text{pinsSetup} \rangle$.

Exemplo: Programar os pinos 1, 3 e 6 da porta 1 como input:

PI 01 4A

Make Pins Output: $\langle \text{char} \rangle^+ = PO\spadesuit\langle \text{portAddr} \rangle\spadesuit\langle \text{pinsSetup} \rangle$

Na porta de endereço $\langle \text{portAddr} \rangle$, configurar como saída os pinos cujos bits estão a '1' em $\langle \text{pinsSetup} \rangle$.

Read Digital Input: $\langle \text{char} \rangle^+ = RD\spadesuit\langle \text{portAddr} \rangle\spadesuit\langle \text{pinsSetup} \rangle$

Ler da porta $\langle \text{portAddr} \rangle$ o valor digital dos pinos cujos bits estão a '1' em $\langle \text{pinsSetup} \rangle$.

Os valores correspondentes aos pinos cujos bits estão a '0' em $\langle \text{pinsSetup} \rangle$ deve ser '0'.

Write Digital Output:

$\langle \text{char} \rangle^+ = WD\spadesuit\langle \text{portAddr} \rangle\spadesuit\langle \text{pinsSetup} \rangle\spadesuit\langle \text{pinValues} \rangle$

Escrever nos pinos a '1' em $\langle \text{pinsSetup} \rangle$ da porta de endereço $\langle \text{portAddr} \rangle$ os valores correspondentes de $\langle \text{pinValues} \rangle$.

Os pinos da porta correspondentes aos bits que estão a '0' em $\langle \text{pinsSetup} \rangle$, não são alterados.

Exemplo: Escrever nos bits 3 e 7 da porta 1, os valores 0 e 1 respectivamente:

WD 01 88 80

Analog Read: $\langle \text{char} \rangle^+ = RA\spadesuit\langle \text{addr3} \rangle$

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:

<addr> ∈ [0000..FFFF]

<length>, <byte>, <portAddr>, <pinsSetup>, <pinValues> ∈ [00..FF]

<addr3> ∈ [00..10]

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 deve confirmar ao utilizador que a operação foi concluída com sucesso. Para isso pode, por exemplo, fazer 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), deve ser uma mensagem que indique isso ao utilizador.

Em qualquer caso, e à semelhança de outras interfaces baseadas em linha de comandos, o sistema deve fornecer um *prompt*, por exemplo “>”, que confirma ao utilizador que a interface está pronta para aceitar um próximo comando. Naturalmente, o *prompt* deve ser apresentado após inicialização.

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

```
>WD 02 03 03
written digital output
>
```

Neste exemplo, a resposta do sistema replica os dados originais do comando e confirma a execução com sucesso do comando.

6 Objectivos a apresentar em aula

Abaixo encontram-se descritos os objectivos e respectiva **aula de conclusão**. Há objectivos que ocupam apenas 1 aula e outros que ocupam 2 aulas. No início da primeira aula de cada objectivo deve ser apresentado na aula:

- a preparação que é pedida abaixo
- o plano de actividades para atingir o objectivo.

O plano é constituído por uma lista de frases curtas (uma linha), uma para cada actividade a desenvolver.

Na parte final da última aula de cada objectivo apresenta-se:

- o resultado final do trabalho desenvolvido.

Objectivo 1ª aula: análise léxica das mensagens 100% funcional

Decomposição das mensagens em partes (ou *tokens*) de acordo com os caracteres delimitadores definidos na gramática. Pode e deve utilizar funções da biblioteca C para strings, para não “reinventar a roda”.

- **Preparação:** fluxograma com o algoritmo do ciclo léxico e definição clara da entrada e saída. Note-se que um fluxograma de um algoritmo é diferente de um fluxograma que descreve código. Um algoritmo contém os passos para a resolução do problema em abstracto, e são necessárias algumas iterações de refinamento do algoritmo. Posteriormente, é normal a sua tradução para código dar origem a alterações devido aos detalhes de implementação.

Devido à utilização de arrays é natural que a muito limitada capacidade de memória interna do 8051 (128b) seja excedida. Para evitar isso, recomenda-se deslocar as variáveis necessárias para memória externa (de maior capacidade). Exemplo:

```
char xdata command[CMD_LIMIT];
```

Objectivo 2ª aula: análise sintáctica dos comandos 100% funcional

Identificar cada um dos comandos e verificar se as regras da sintaxe são respeitadas.

- **Preparação:** fluxograma com o algoritmo do ciclo sintáctico e definição clara da entrada e saída.

Objectivo 4ª aula: comunicação série 100% funcional

Receber e enviar mensagens via porta-série recorrendo a interrupções, e seguindo a arquitectura definida antes. Procurar isolar as partes do código que são dependentes da placa KIT8051USB das partes que são gerais/portáveis, nomeadamente através da definição de funções.

- **Preparação:** definição da configuração da porta-série e da interrupção; fluxograma das rotinas de recepção e envio; e definição das variáveis a partilhar com a rotina principal.

Neste objectivo reforça-se a importância do planeamento e da necessidade de abordar o desenvolvimento por níveis, como por exemplo, começar por comunicar por polling, testar a comunicação utilizando um único caractere, etc.

Objectivo 6ª aula: processamento dos comandos 100% funcional

Validar os valores dos argumentos de cada comando e implementar o respectivo processamento. A leitura da entrada analógica pode ser feita sem recurso à interrupção. Identificar partes do código que são específicos da placa KIT8051USB.

- **Preparação:** fluxograma da leitura ou escrita digital; fluxograma da leitura do ADC, com o bloco de configuração especificando claramente o modo de operação, bloco de arranque do ADC e bloco de recolha e escrita da mensagem com o valor a enviar pela porta série.

Objectivo 8ª aula: *porting* da aplicação para a placa STM32

Apenas as partes do código dependentes da placa precisam de ser alteradas. Eventuais outras alterações devem ser registadas.

- **Preparação:** diagrama com os nomes de todas as funções e dependências entre elas; pode ser do estilo diagrama de classes.

Nota. Para assegurar que o programa principal está a executar, pode-se programar uma rotina que simplesmente faz piscar um diodo 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