ISEL | DEETC

Sistemas Embebidos I

'Tacógrafo'

Semestre de Inverno de 2010/2011

Autores:
31401 – Nuno Cancelo
33595 – Nuno Sousa

Indicie

Introdução

Ao longo deste semestre foram transmitidos (e adquiridos) conhecimentos sobre a forma como a arquitectura ARM® (nomeadamente do ARM®7 integrante no modelo LPC2106 da NXP). Os temas leccionados cobriram os temas em cinco actividades práticas que se resumem em implementar uma funcionalidade do LPC e o respectivo programa de teste.

A nossa abordagem à cadeira fugiu um pouco ao que era sugerido nas actividades propostas. Sendo estas actividades guias para orientar a implementação, em breve notámos que a estrutura do trabalho não estaria uniforme tornando complicada a sua manutenção Desta forma optamos por estruturar o nosso código de forma a possibilitar a evolução de cada componente do nosso processador.

A nossa estruturação implementa um sistema de camadas (layers), tendo cada camada uma responsabilidade e uma API[1]. Esta separação de responsabilidades e com o suporte da respectiva API permite desacoplar o hardware do software, permitindo criar uma base para o desenvolvimento de um sistema operativo sobre o processador. Esta separação por camadas permite também migrar o hardware para outra versão da mesma família, sendo somente necessário alterar os respectivos ficheiros header que reflictam as alterações sem que seja necessário alterar as assinaturas das funções públicas.

Ao longo deste documento, pretende-se esclarecer a solução adoptada, analisando cada camada e justificando a implementação.

1 Application programming interface

Camadas de Implementação

Como foi mencionado a nossa solução foi separada em camadas, tendo cada camada um responsabilidade. Tenciona-se que, desta forma, cada camada seja independente havendo relação de simbiose entre camadas através da interface pública.

Pode-se verificar a identificação de cada camada na Figura 1: Descrição das Camadas o esboço dessa estruturação.

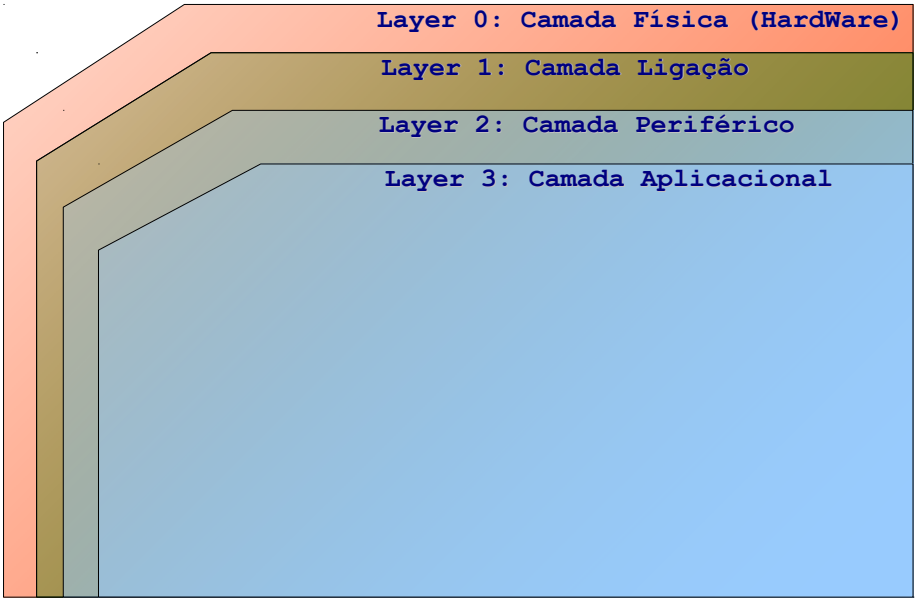

Figura 1: Descrição das Camadas

Na realidade existem dois grupos de camadas. A camada 0 e 1 estão relacionadas com os componentes que compõem o processador, a identificação das suas estruturas de dados, seus endereços lógicos e as suas constantes. A camada 2 e 3 são aplicação de funcionalidades configurando o processador para a sua implementação como seja a configuração de um teclado, de um LCD 16x2 entre outros na camada periférico, como a criação de software que utiliza recursos do processador (como sejam as interrupções, temporizadores, alarmes, relógio, etc).

Nota: Denominamos periféricos como componentes de hardware externos ao processador, que precisam de ser configurados utilizando a API das camadas 0/1.

Tentaremos demonstrar em cada camada a estrutura adoptada e a componente teórica que a envolve, assim como as dificuldades encontradas e a solução adoptada para ultrapassar o obstáculo.

Camada 0: Camada Física (Hardware)

Após a ligação física do módulo USB DLP-2232M-G ao modulo lpc-h2106 (que contem o processador para o qual estamos a programar), ficámos uma com uma estrutura que permite o envio do nosso ficheiro binário para a flash de forma a podermos executar o nosso programa num sistema embebido.

Esta estrutura permite também analisar e testar o código num ambiente de debug num computador, tendo acesso aos componentes do processador do LPC2106 sem que o binário esteja a ser executado a partir do mesmo. Existe alguma degradação de performance neste modo de funcionamento mas é indicado para que se possa analisar todo código, verificar os erros, entendê-los e corrigir-los.

É de salientar que utilizámos o openocd como forma de estabelecer comunicação entre o computador e o sistema embebido.

Antes de podermos utilizar uma linguagem de (mais) alto nível para programar sobre este processador é necessário preparar o sistema para o suportar. Para este efeito é exigido proceder a uma etapa 0 em dois passos, relacionados mas distintos.

Num primeiro passo configura-se um ficheiro em linguagem assembly , que vai realizar os seguintes passos:

- Inicializar os vectores das interrupções, estabelecendo o endereço da rotina que trata desse tipo de interrupção
- Reservar o bloco de memoria para stack
    - Para cada vector, para poder proceder a operações de tratamento de interrupções, permitindo guardar registos e outro tipo de informação
    - O restante espaço disponível fica reservado para a utilização genérica
- Limpa o espaço reservado para a BSS
- Inicia o oscilador do processador
- por fim chama a rotina main, da linguagem C

Este ficheiro deverá ser indicado ao linker para que ele possa resolver os nomes atribuídos e não deve ser incluído em nenhuma biblioteca. Esta última condição deve-se à necessidade de tornar portável a solução e depende de cada sistema embebido.

Num segundo passo, configura-se o ficheiro ldscript que indicará ao linker algumas configurações de inicialização em relação ao programa, entre as quais:

- Indicação do ponto de entrada que inicia o programa, ou seja, qual nome da rotina em assembly que prepara o sistema para suporta a linguagem de alto nível.
- Indicação do endereço físico a atribuir ao espaço reservado

para a Flash e para a RAM assim como tamanho do espaço disponível para cada um deles. É possível indicar o endereço de algum periférico externo que poderá fazer boot.

- Por fim indicação das secções suportadas pela nossa arquitectura. Estas secções, por omissão, deverão ter as mesma que a linguagem C têm, para garantir compatibilidade, assim como colocar o nome das secções personalizadas.

Este ficheiro deve ser fornecido ao linker para que o mesmo configure devidamente os endereços, de acordo com o que foi estipulado.

A partir deste momento temos suporte à linguagem C, significando que de um modo geral não é necessário voltar modificar ambos os ficheiros, ou mesmo programar funções em linguagem Assembly. Isto de modo geral, pois existem situações em que o código gerado pelo compilado é demasiado complexo que o que se propõe executar, ou então quando se deseja proceder a acções sobre processador, nesta alturas é ideal criar algumas funções para o efeito.

Tirando partido de estarmos a trabalhar com a linguagem C, a utilização de macros foi muito utilizada para evitar ter valores fixos embebidos no código fonte, assim como ter algumas funções *inline*. Esta forma de programar tem duas consequências:

1. Como as macros e a funções *inline* são traduzidas em tempo de pré-compilação e é realizada uma substituição directa, o ficheiro binário final ficará maior.
2. Como esta substituição evita a chamada a funções, torna a execução do código mais rápida.

Assim temos a estrutura desta camada na Figura 2: Camada 0

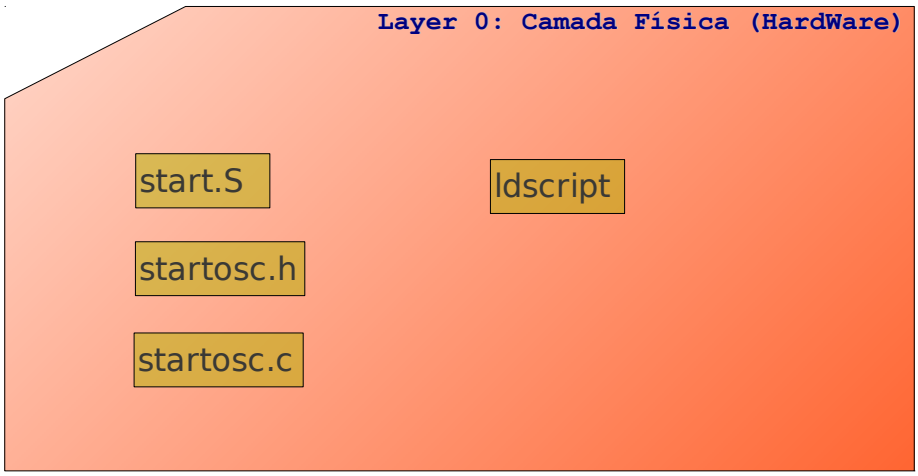

Figura 2: Camada 0

Camada 1: Camada de Ligação

Nesta etapa implementamos os elementos leccionados nas aulas, tendo em atenção às definições encontradas no manual e nos anexos encontrados no site da NXP, sendo todos eles bastantes úteis havendo no entanto espaço para melhoria na documentação de apoio ao programador.

Ao examinar a documentação foi evidente qual a sequência de módulos a serem implementados, começando desde logo a tornar a nossa API independente.

Assim o primeiro passo de um longo caminho foi dado na implementação de um ficheiro fonte (header file) que permitisse a portabilidade do código entre diversas arquitecturas, não comprometendo a implementação do restante em relação ao domínio das variáveis. A este ficheiro fonte foi chamado **TYPES.h**.

A partir desde momento criámos algumas linhas de apoio para o nome das funções, variáveis e macros sendo esta a descrições das regras impostas:

- __NOME_DA_VARIAVEL__ : Macro que define um valor de apoio à API
- pNOME_DA_VARIAVEL: ponteiro para um endereço de memória referente a uma estrutura
- NOME_DA_VARIAVEL: Macro que define um valor de assinatura pública

De seguida, agora seguindo o manual, construímos outro ficheiro fonte (**SCB.h**) tendo já este algumas estruturas que definem os seguintes elementos do processador:

- Estrutura de Interrupções Externas : ***pEXT_INT***
- Estrutura do Memory Accelerator Module: ***pMAM***
- Estrutura do Phase Locked Loop: ***pPLL***
- Estrutura do Power Control: ***pPOWER***
- Estrutura do VPB Divider: ***pAPBDIV***
- Estrutura do System Control Block: ***pSCS***
- Definição de Macros com valores por omissão e máscaras para os respectivos bits com nomes sugestivos

Após este ficheiro estar concluído, procedemos à alteração de um módulo da camada anterior: **startosc.c** e **startosc.h,** uma vez que na versão anterior tínhamos muitos valores colocados no código e com a implementação do SCB.h e do TYPES.h não fazia sentido manter como estava.

Aproveitou-se a oportunidade para criar uma função que devolve a velocidade do relógio do periféricos, uma vez que a mesma poderá não ser a mesma que o do sistema central. A implementação teve alguns

problemas uma vez que existe um bug conhecido pela NXP na leitura do registo vpbdiv e é necessário proceder a duas leitura para obter o valor correcto.

Tendo estas questões sido ultrapassadas procedemos à realização dos módulos leccionados ao longo do semestre, sendo eles os seguintes:

GPIO.c (GPIO.h)

Foi construída uma estrutura para representar o GPIO, estando organizada para conter os registos necessários à configuração de um porto de general input/output.

Como os portos têm multiplexagem de funções, um porto é configurado pela função ***void gpio_init_PINSEL0(U32 mask);*** ou ***void gpio_init_PINSEL1(U32 mask);***, porém como temos o jtag instalado para poder realizar o debug, 15 dos 16 bits do pinsel1 estão reservados razão pela qual optamos por não disponibilizar macros que configuram correctamente cada função para o porto/função desejada, ficando somente definidos para todos os portos do pinsel0.

São oferecidas como interface pública as seguintes funções:

- U32 gpio_read(U32 mask);
- void gpio_clear(U32 mask);
- void gpio_write(U32 mask, U32 value);
- void gpio_set_direction(U32 mask, U8 direction);
- void gpio_init_PINSEL0(U32 mask);

TIMER.c (TIMER.h)

A arquitectura do LPC2106 contém dois temporizadores internos, que permite ter alguma versatilidade no desenho de aplicações, sem ficar muito restringido.

Torna-se claro que para poder programar de forma genérica acções sobre os temporizadores, o mais elegante é não repetir código pois ambos os temporizadores fazem exactamente as mesma coisas. Tendo isso em mente criou-se uma estrutura com os registos que lhe são atribuídos e uma longa lista de macros com as diversas identificações que os bits representam nos diversos registos.

Assim, na assinatura das funções todas ela recebem um ponteiro para uma estrutura que representa o TIMER de forma a poder proceder acções sobre a mesma. Como existem funções que procedem a uma/duas instruções, optámos por criar funções macros de forma a evitar o overhead do ACPS para estas funções. Outra forma de ultrapassar a questão, dada o numero de instruções, era colocar as funções como ***inline***, facilidade concedida pela linguagem C.

Para as acções normalmente efectuadas pelos temporizadores, foram realizadas funções para que um dado temporizador (indicado em parâmetro) funcione em modo external match ou em modo captura, potencializando a solução com temporizadores que realizam algo mais que

contar tempo.

Uma vez que a versão que possuímos é a versão 00, não nos foi possível implementar e testar o temporizador em modo de captura como contador de eventos externos, pelo que ficámos um pouco desiludidos.

São oferecidas como interface pública as seguintes funções:

- void TIMER_init(pLPC_TIMER timer, U32 countNbr);
- void TIMER_delay(pLPC_TIMER timer, U32 elapse);
- void TIMER_capture_init(pLPC_TIMER timer,U8 channel, U32 captureMask, U32 countNbr,tCtcrFunction ctcrFunction);
- void TIMER_ext_match_init(pLPC_TIMER timer,U8 channel, U32 MatchMask, U32 countNbr,tEmrFunction emrFunction);
- void TIMER_ext_match_changeTime(pLPC_TIMER timer,U8 channel, U8 dif);
- void TIMER_ext_match_stop(pLPC_TIMER timer); #Como Macro
- void TIMER_ext_match_start(pLPC_TIMER timer); #Como Macro
- void timer_sleep_microseconds(pLPC_TIMER timer,U32 last_u_sec); #Como Macro
- void timer_sleep_miliseconds(pLPC_TIMER timer,U32 last_m_sec); #Como Macro
- void timer_sleep_seconds(pLPC_TIMER timer,U32 last_sec); #Como Macro
- void timer_start(pLPC_TIMER timer); #Como Macro
- void timer_stop(pLPC_TIMER timer); #Como Macro
- U32 timer_elapsed(pLPC_TIMER timer,U32 last_elapsed); #Como Macro
- U32 timer_now(pLPC_TIMER timer); #Como Macro
- U32 timer_capture1_time(pLPC_TIMER timer); #Como Macro
- void timer_reset(pLPC_TIMER timer); #Como Macro

RTC.c (RTC.h)

Este periférico interno do LPC tem dois modos de funcionamento simultâneos, como calendário e como alarme. Foi somente realizado a solução para o modo de calendário, restringindo de alguma forma a potencialidade da solução.

Respeitando as considerações obtidas no manual só houve uma pequena questão na solução que resultou num pequeno atraso no desenvolvimento, que foi a utilização (por distracção) do operador && em deterioramento do operador & que era o pretendido. Resolvida esta questão este módulo funciona sem problemas (na medida dos nossos testes).

Foram criados alguns tipos de dados para representar a Data e a Hora de forma a simplificar a forma como as funções são chamadas e as estruturas são iniciadas/afectadas.

São oferecidas como interface pública as seguintes funções:

- void rtc_init();
- void rtc_setDate(U16 year,U8 month, U8 day);
- void rtc_setTime(U8 hour,U8 minute, U8 seconds);
- void rtc_setDOW(U8 dow);

- void rtc_setDOY(U16 doy);
- void rtc_getDate(DATE* date);
- void rtc_getTime(TIME* time);
- void rtc_setDateTime(DATE_TIME* datetime);
- void rtc_getDateTime(DATE_TIME* datetime);
- void rtc_setDom(U8); #Como Macro
- void rtc_setMonth(U8); #Como Macro
- void rtc_setYear(U16); #Como Macro
- void rtc_setHour(U8); #Como Macro
- void rtc_setMin(U8); #Como Macro
- void rtc_setSec(U8); #Como Macro
- U8 rtc_getDom(); #Como Macro
- U8 rtc_getMonth(); #Como Macro
- U16 rtc_getYear(); #Como Macro
- U8 rtc_getHour(); #Como Macro
- U8 rtc_getMin() ; #Como Macro
- U8rtc_getSec() ; #Como Macro

I2C.c (I2C.h)

A realização deste módulo é implementado o protocolo I2C utilizando o GPIO, e não tirando partido das funções fornecidas pelo processador.

Alguns erros foram identificados durante os testes com a EEPROM, mas deveram-se ao esquecimento de corrigir a orientação dos portos GPIO, que quando identificados foram corrigidos.

São oferecidas como interface pública as seguintes funções:

- void I2C_init();
- void I2C_start();
- void I2C_stop();
- void I2C_write_byte(U8 value);
- U32 I2C_slave_ack();
- U8 I2C_read_byte();
- void I2C_master_ack();
- void I2C_master_nack

WATCHDOG.c (WATCHDOG.h)

Este módulo foi dos mais simples de implementar não causando qualquer incomodo.

São oferecidas como interface pública as seguintes funções:

- void WD_reset();
- void WATCHDOG_init(U32 value);

- void WD_ISRUNNING(); #Como Macro
- void WD_ENABLE(); #Como Macro
- void WD_DISABLE(); #Como Macro
- void WD_RESET_ENABLE(); #Como Macro
- void WD_RESET_DISABLE(); #Como Macro
- Bool WD_COME_FROM_RESET(); #Como Macro
- U32 WD_READ_TIME(); #Como Macro

VIC.c (VIC.h)

Se no módulo anterior a implementação foi simples, a realização deste deu origem a muitas horas de volta do kit de desenvolvimento em modo de debug para análise da razão pela qual as interrupções não estariam a funcionar como esperado.

Após de muitas experiências, conseguimos ter as interrupções a funcionar e entendemos os nossos dois erros. O primeiro erro (e mais importante) verificou-se ao simples facto de não estarmos a habilitar as interrupções ao nível do processador, dessa forma tornava-se complicado conseguir tratar interrupções vectorizadas. O segundo devia-se ao facto de não estarmos a limpar a interrupção que foi tratada.

Com esta situação resolvida, a nossa implementação deste módulo funcionou como esperado.

A concretização da função ***Bool VIC_ConfigIRQ(U8 peripherical, U8 priority,void (*fx)(void))*** é a grande mais valia deste módulo que permite vectorizar qualquer periférico (dos que estão disponíveis), com uma prioridade e atribuindo o endereço da função que irá tratar dessa interrupção. Apesar de desencorajar a atribuição do mesmo periférico a várias prioridades e/ou funções de tratamento, é possível proceder dessa forma. No entanto o VIC executará a primeira ocorrência da função de tratamento desse periférico.

Outra funcionalidade implementada foi implementar uma forma de activar/desactivar as interrupções ao nível do processador.

São oferecidas como interface pública as seguintes funções:

- Bool VIC_ConfigIRQ(U8 peripherical, U8 priority,void (*fx)(void));
- void VIC_init();
- void disableIRQ(U8 peripherical);
- void enableIRQ(U8 peripherical);
- void interrupt_enable();
- void interrupt_disable();

Desta forma temos a nossa camada de ligação(Figura 3: Camada 1) implementada. Existem mais periféricos do processador implementados mas não houve oportunidade de os testar afincadamente, como seja o caso da UART.

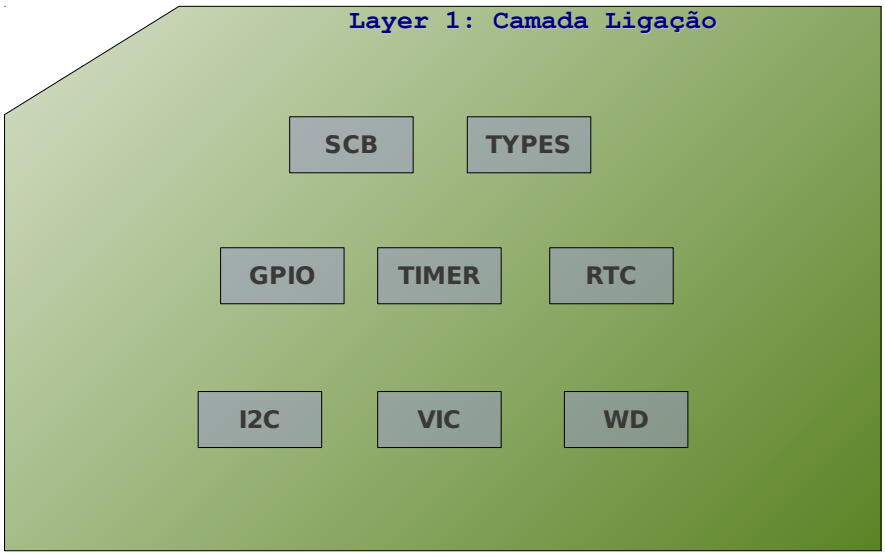

Figura 3: Camada 1

Nota: Nomeámos esta camada de Ligação, no sentido de conectar um objecto de programação a uma entidade do processador, e não no sentido do Modelo OSI/TCP-IP.

Camada 2: Camada de Periférico

Desde já, começa a ser visível a flexibilidade da solução com a implementação da camada anterior. No entanto não é muito interessante se não estiverem periféricos externos ligados ao kit de desenvolvimento.

É aqui que se sente a necessidade de criar uma camada que interaja com as funcionalidades da camada anterior e que proporcione mais funcionalidades.

Atendendo ao propósito do nosso projecto, implementámos nesta camada as seguintes funcionalidades:

Keyboard.c (Keyboard.h)

Foi utilizado como periférico um teclado matricial 4x4 conectado com os portos GPIO P0.09 até P0.15. Como forma de reduzir a corrente que entra nos respectivos portos, colocamos resistências de pull-up na ordem dos 50K ohm. No entanto, como o porto P0.14 é utilizado no processo de boot do processador, e o mesmo está em pull-up, foi colocada uma resistência em paralelo de 400ohm para reduzir a tensão neste porto durante a utilização do teclado de forma a poder fornecer valores correctos.

Atendendo ao facto de os portos GPIO são de entrada e saída, utilizamos uma técnica de captura por bitmap. Ou seja, vamos construindo um bitmap que mostra o estado de cada um dos 8 bits que representam as quatro linhas e as quatro colunas.

Figura 4: Teclado Matricial 4x4

Por termos resistências de pull-up ligados aos 8 portos, enviamos zeros para os quatros portos de saída, capturamos os quatro portos de entrada. Se houver alguma tecla pressionada, trocamos a orientação dos portos e executamos a mesma operação. Se continuarmos com tecla pressionada guardamos o bitmap.

Como parte da API, estão disponíveis quatro tipos de função:

- Devolve o bitmap
- Devolve uma tecla (no domínio 0-F)
- Verifica se há tecla pressionada ou em buffer
- Ler a tecla pressionada

Desta forma damos versatilidade ao programador em decidir a forma de como executa o tratamento da tecla pressionada.

Na nossa implementação das funções de retorno de tecla, optamos por disponibilizar também uma função que devolve qual foi a tecla anterior à pressionada. Esta função só é útil quando utilizada antes de ler

o valor da tecla (como valor ou bitmap), uma vez que nesta altura em que os valores são actualizados e valor da tecla anterior é esmagado pela tecla devolvida.

A função ***keyboard_hasKey()*** verifica se tem tecla em buffer, se não tem verifica se há tecla pressionada e no fim devolve 0 se não houver, outro valor se houver. Esta função deve-se utilizar sempre antes de fazer chamadas às funções que retornam as teclas.

Para validar o resultado da obtenção com o facto de haver ou não tecla pode-se comparar o resultado com a macro NO_KEY.

São oferecidas como interface pública as seguintes funções:

- void keyboard_clearKey();
- U8 keyboard_getPreviousKey();
- U8 keyboard_getKey();
- U8 keyboard_hasKey();
- U8 keyboard_getPreviousBitMap();
- U8 keyboard_getBitMap();
- U8 keyboard_decodeKey(U8 keyBitmap);
- void keyboard_readKey();
- void keyboard_init(pLPC_TIMER timer);

Como alguns dos portos do teclado irão ser partilhados com o periférico LCD (descrito de seguida), garantimos que nenhum dos portos está a emitir sinais que possam ser interpretados como tecla pressionada, garantindo desta forma que não existem interferências entres estes dois periféricos.

LCD.c (LCD.h)

Utilizámos um ecrã de 2x16 da Nanox (Figura 5: Nanox NDM162) que contem um micro-controlador HD44780,como periférico a ser instalado e testado.

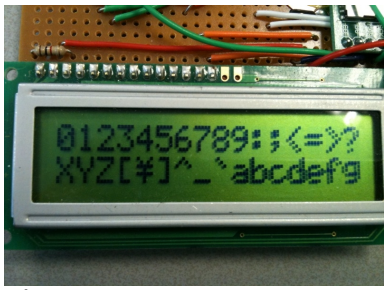

Figura 5: Nanox NDM162

Como estamos a utilizar o pinsel0 do processador, não existe muitos portos de GPIO disponíveis, por essa razão decidimos multiplexar a utilização de alguns portos já atribuídos para o teclado matricial. Assim sendo procedemos ao protocolo de envio de informação a 4 bits, sem estes portos partilhados com o teclado, e configurando dois outros portos distintos para atribuição do sinal de RS e de EN do dispositivo.

Após a configuração dos portos procedeu-se à implementação dos dos métodos relevantes para escrever no ecrã e a sua inicialização, respeitando os valores e os tempos referidos no datasheet (do LCD e do micro-controlador).

São oferecidas como interface pública as seguintes funções:

- void LCD_init(pLPC_TIMER timer);
- void LCD_write(U32 byte);
- void LCD_clearLine(U8 line) ;
- void LCD_posCursor(U8 line, U8 col) ;
- void LCD_setCursor(U8 visible, U8 blinking);
- void LCD_writeChar(U8 c) ;
- void LCD_writeString(Pbyte txt);
- void LCD_writeLine(U8 line, Pbyte txt);
- inline void LCD_setCenter(U8 value) ;

EEPROM.c (EEPROM.h)

Este foi o último periférico configurado, para tal utilizámos o modelo CSi T4H-24C08L (da Catalysy Semiconductor).

Para a sua utilização foi utilizado o módulo I2C (via gpio) implementado na camada anterior e após da resolução de alguns bugs de programação tornou-se possível ler e escrever da mesma.

De forma a optimizar as escritas e as leituras, as funções implementas permitem ler/escrever **até 16 bytes**, sendo estes tamanhos validados antes de proceder à escrita/leitura. Em caso de valores inválidos nada é efectuado.

Figura 6: CSi T4H-24C08L

São oferecidas como interface pública as seguintes funções:

- U8 eeprom_write_block(U32 address, U8 * block, U8 size);
- void eeprom_read_block(U32 address, U8 * block, U8 size);
- void EEPROM_init();

Assim fica concluída a nossa camada, ficando disposta na Figura 7: Camada 2

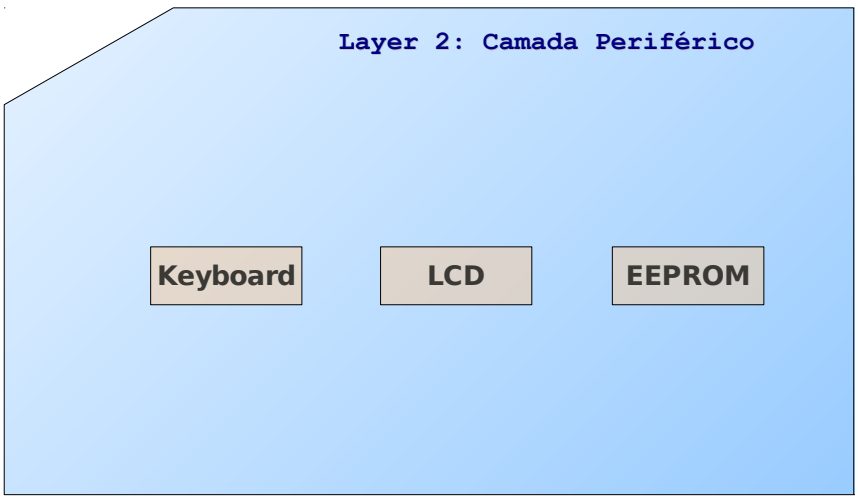

Figura 7: Camada 2

Camada 3: Camada Aplicacional

Será nesta camada onde o frutos da implementação das camadas anteriores serão visíveis no sistema embebido. Pretende-se que seja nesta camada que se proceda à realização de um pedaço de software que utilize a API dos periféricos do processador e dos periféricos externos criados para o efeito.

Fica ao critério do programador a forma sobre a qual irá interagir com todos os componentes da API assim como o propósito da sua realização.

Tacógrafo

Como exemplo de implementação procedemos à execução do trabalho final proposto: a criação de um programa que simule as funcionalidades de um tacógrafo.

A nossa implementação é compostas por componentes, de forma a manter a independência do código e a versatilidade da solução.

Menu.c (Menu.h) MenuFunctions.c (MenuFunctions.h)

O primeiro passo foi implementa o componente de Menu, que contem uma função genérica para interagir com os elementos que o contém. Um menu é composto por ***n*** elementos/funções, mas as acções sobre os mesmos são os mesmos pelo que uma função genérica pareceu-nos uma escolha acertada. Foi implementada uma estrutura ***Option*** que representa um Texto que representa a opção e um ponteiro para a função que terá que executar.

Dada a estrutura do enunciado foram criados dois conjuntos de Options, com as respectivas funções que vão ser executadas. Atendendo ao facto de estas funções procederem a acções sobre os dados do percurso, o endereço do percurso é passado por argumento para uma descodificação mais rápida (uma vez que os nossos argumentos são guardados nos registos).

São oferecidas como interface pública as seguintes funções:

- void Menu_Generic(PVOID course, pOption options[], U8 sizeOf);

Como parte a API é indicado que as tabelas de Options a serem utilizadas estão definidas também neste módulos cujo o seu nome são:

- Option menu2Options;
- Option menu1Options;

O tamanho das tabelas são dadas pelas macros:

- __MAX_FUNCTION_MENU_2__
- __MAX_FUNCTION_MENU_1__

Pelo que devem declarar que as tabelas que pretendem executar num outro programa se encontram definidas noutro módulo.

As funções são definidas no módulo MenuFunctions, como forma de agregar as funções referentes aos menus.

Faz parte da assinatura das funções receberem um ponteiro para void de um objecto. Esta decisão sustenta-se no facto de não pretender-mos ficar comprometidos com uma estrutura de dados especifica, mas sim permitir que seja a implementação da função trate dos dados da forma ideal.

No nosso caso fazemos uma conversão para ***Percurso***, para procedermos a acções sobre os dados do mesmo.

São oferecidas como interface pública as seguintes funções:

Menu 1 Functions

- void printToLCD(char* line0,char* line1);
- void printDateTime(PVOID course);
- void printDistance(PVOID course);
- void printTime(PVOID course);
- void printMaxSpeed(PVOID course);
- void printAvgSpeed(PVOID course);
- void printTotalDistance(PVOID course);
- void printTotalTime(PVOID course);

Menu 2 Functions

- void setClock(PVOID percurso);
- void resetTotal(PVOID percurso);

Clock.c (Clock.h)

Como é visível na API anterior existe uma função setClock, no entanto a mesma é definida noutro módulo, da a especificidade da implementação.

Neste módulo é procedida a uma separação de competências:

- Navegação pelos elementos que compõe a data/hora
- Formatação da data/hora para valores dentro dos seus domínios

A Navegação pelos elementos segue a mesma lógica do Menu, sendo diferenciada na forma como os botões procedem a operações diferentes.

A formatação da data pretende incrementar/decrementar valores garantindo que estão dentro do domínio do secção que dizem respeito, garantindo sempre uma data válida.

Para proceder a esta garantia foi necessário implementar uma função que garantisse que um valor ficasse sempre dentro do domínio, seguindo uma lógica de buffer circular.

No entanto a versão final não foi a primeira que implementamos. A primeira utilizada foi a utilização do operador módulos(%), contudo devido ao facto de por omissão as operações assembly serem sem sinal,

resultados estranhos eram apresentados. Este comportamento foi verificado, ao analisar o assembly da instrução criada e ficamos um pouco decepcionados pela quantidade de instruções geradas para um processamento simples. Como forma de optimizar este troço de código, implementamos a função ***U32 modulos(S32 value, U8 adj, S8 offset, U32 mod***, a qual descrevemos o significado dos seus argumentos e implementação de seguida:

```
U32 modulos(S32 value, U8 adj, S8 offset, U32 mod)
{
  register S8 val = value -adj + offset;
  if (val<0)
            val = mod -1;
  else if(val > mod-1)
            val -= mod;
  return val + adj;
}
```

Os argumentos:

- value: valor actual a ser analisado
- adj: valor do ajuste, permite colocar o valor dentro de um domínio que não comece em Zero
- offset: valor a ser incrementado/decrementado
- mod: numero de elementos do domínio

Com esta implementação, permite-nos utilizar a função para múltiplos propósitos, sejam na rotatividade das opções nos menus que seja no rotação dos valores da data/hora.

Percurso.c (Percurso.h)

Foi criado o objecto Percurso que contem as variáveis necessárias para a manutenção dos valores do programa assim como uma tabela de métodos virtuais com as funções necessárias para a sua manipulação.

Esta tabela foi implementada como forma de manter relação entre o objecto Percurso e os seus valores, estivesse o objecto onde fosse no programa de forma a poder aceder às suas acções sem que houvesse dificuldade no acesso ao objecto e manter o objecto “fechado para mundo”.

São oferecidas como interface pública as seguintes funções:

- void percurso_init(Percurso* percurso, U32 tDistance, U32 tTime);
- Através da tabela de métodos virtuais:

```
typedef struct percurso_methods{
  const void (*resetValues)        (Percurso* this);
        const void (*addStopTime)        (Percurso* this ,U32 time);
        const void (*addSpentTime)       (Percurso* this ,U32 time);
        const void (*updateDistance)     (Percurso* this ,U16 distance);
        const void (*updateAverageSpeed) (Percurso* this);
        const void (*setCurrentSpeed)    (Percurso* this, U8 speed);
        const void (*updateSpeed)        (Percurso* this, U32* tickTime, U32* tickCount);
        const void (*start)              (Percurso* this, U32 tickTime,  U32 tickCount);
}Percurso_Methods;
```

Tacografo.c (Tacografo.h)

Com estes objectos implementados procedemos à implementação do programa principal. Este programa tem algumas considerações que desejamos evidenciar.

Tratamento de interrupções

Uma vez que o modelo do LPC-2106 que possuímos não permite utilizar o Timer como contador de eventos externos, optámos por configurar as interrupções do modo Capture do Timer0 no VIC. A função de tratamento de interrupção incrementa uma variável **tickCount** que contabiliza o número de "voltas" detectadas pelo dispositivo externo e guarda na variável tickTime (incrementando ao valor previamente guardado) o tempo que demorou entre o tick anterior para o presente. Assim, quando consultamos o conteúdo da das variáveis temos o numero de ticks que ocorreram desde a última vez e o tempo que demoraram a decorrer.

Inicialização do Programa

Antes de procedermos à utilização do programa, inicializamos os periféricos que iremos utilizar nomeadamente os Timers, LCD, RTC, VIC, Keyboard e Watchdog. Configuramos também o VIC_ConfigIRQ para o tratamento da interrupção do timer em modo capture e configuramos o timer0 para o modo de capture (nas transições descendentes) e o modo match para fazer reset e fazer toggle do sinal.

Ponto de entrada do Programa

Para desenvolvimento do ponto de entrada do programa principal, a nossa grande preocupação foi tentar garantir que o programa não ficasse "pendurado" em nenhuma das funções que fossem chamadas, de forma a evitar que não houvesse teclas pressionadas que não fossem identificadas. Adicionalmente, pretendemos garantir que a actualização da velocidade seja demonstrada em tempo real. Assim optámos por consulta a variável tickTime e verificar quanto tempo decorreu desde a última actualização (a função para actualizar a velocidade, depois de usar os valores guardados coloca-os a zero, garantindo que o tempo que está guardado em tickTime é o tempo desde a última actualização). Caso o valor seja superior ao valor predefinido (neste caso usámos 1 segundo) então actualiza a velocidade actual, assim como a distância percorrida e o tempo despendido na viagem até então.

Quando o veículo pára deixamos de ter ticks e o tickTime irá ficar a zero depois de actualizar a velocidade. Para garantir que a velocidade é actualizada para zero, verificamos se o calor da contagem do timer (registo TC) ultrapassa o valor necessário para que o sensor conte 1km/h.

Como está a andar a menos de 1km/h e neste caso consideramos que o veículo está parado.

No seu funcionamento normal o programa verifica se existe alguma tecla pressionada válida e processa a função atribuída a essa tecla. No final de cada ciclo verifica-se se já foi ultrapassado os valores limites desde a última escrita na memória não volátil, neste caso procedemos à escrita dos valores de referência na memória.

A utilização da função updateSpeed é utilizada sempre que é detectada a necessidade de actualizar a velocidade (a cada 1 segundo se o veículo estiver acima de 1km/h ou a cada 4 segundos se estiver abaixo desse valor). Neste caso, se o valor de tickTime é diferende de 0, actualiza o tempo gasto na viagem, a distância percorrida, a velocidade instantânea (utilizando o numero de ticks presente do tickCount e o tempo entre ticks presente em tickTime). Coloca ambas as variáveis a zero e actualiza a velocidade média do percurso. Neste caso a velocidade média é calculada descontando o tempo em que o veículo esteve parado.

Se o ticktime estiver a zero que dizer que o veículo se encontra parado no percurso e coloca a velocidade instantânea a zero.

Como forma de testar a nossa aplicação configurarmos também o timer0 em modo de match, de forma proceder à emissão de sinais que o capture irá adquirir. Implementamos também funções que nos permitissem arrancar/parar “o veículo” (o match) assim como teclas para aumentar/reduzir a velocidade (na orderm de 1 e 5 km). Estas funcionalidades permitem-nos simular um sensor de velocidade externo.

No caso do arranque/paragem a função habilita/desabilita a capacidade de match faça reset.

Em todos os ciclos da nossa implementação é efectuada um reset ao contador do watchdog, como forma de evitar que o programa faça reset.

Desta forma apresentamos o esboço da Figura 8: Camada 3.

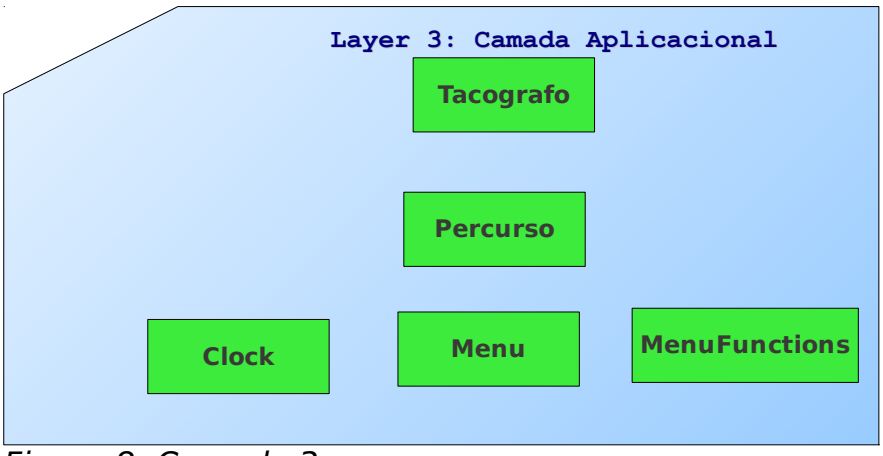

Figura 8: Camada 3

Suporte à Implementação

Como se tornou evidente, é inconcebível gerar os módulos de forma manual. Para auxiliar foi utilizada a ferramenta GNU Make, que permite automatizar as acções.

No entanto, houve algumas reestruturações na hierarquia de directorias para os respectivos módulos até que chegámos à actual versão final.

Atendendo que a solução está separada por camadas, pareceu-nos claro que poderíamos criar bibliotecas de cada camada, de forma a simplificar a componente de geração do objecto final.

Demonstra-se de seguida a estrutura de directórios optada:

```
.
|-- src
|   |-- deploy
|   |   |-- ldscript
|   |   `-- openocd
|   |-- device
|   |   |-- GPIO
|   |   |-- I2C
|   |   |-- RTC
|   |   |-- SCB
|   |   |-- STARTUP
|   |   |-- TIMER
|   |   |-- TYPES
|   |   |-- VIC
|   |   `-- WD
|   |-- include
|   |-- lib
|   |-- peripherical
|   |   |-- EEPROM
|   |   |-- KEYBOARD
|   |   `-- LCD
|   |-- program
|   `-- tests
|       `-- Tacografo
|           |-- Clock
|           |-- Menu
|           `-- Percurso
`-- target
```

Realizando uma síntese da estrutura, referimos que:

- DEPLOY: contem os ficheiros de configuração do LDSCRIPT e OPENOCD
- DEVICE: contem os ficheiros fonte, dos módulos das camadas 0/1
- INCLUDE: contem os ficheiros header de todos os módulos
- LIB: contem as bibliotecas geradas do DEVICE, PERIPHERICAL, LIBGCC e LIBC
- PERIPHERICAL: contem os ficheiros fonte da Camada 2
- PROGRAM: contem o objecto da aplicação que será ligada para originar o ficheiro binário final
- TESTS: Contem os ficheiros fonte da Camada 3
- TARGET: contem o ficheiro binário a ser transferido para o kit de desenvolvimento

Com esta infra-estrutura implementada, a configuração dos ficheiros Makefile foram estruturados de forma a serem iterativos e também não

serem dependentes da máquina onde estão.

Foram criadas regras para proceder à chamada dos respectivos Makefiles referentes às directorias de segundo nível, e nestes proceder à chamada de terceiro nível, de forma a que todos os módulos fossem gerados (caso houvesse alterações), os seus ficheiros header copiados para INCLUDE e no caso do DEVICE e PERIPHERICAL fossem geradas as respectivas bibliotecas.

Desta forma tornou-se bastante simples a geração do objecto final.

O Makefile que está na raiz (SRC) tem algumas regras para enviar logo o ficheiro fonte para o kit de desenvolvimento e para executar o ficheiro em modo debug.

Conclusão

Com o decorrer deste projecto (assim como nas aulas) verificou-se a potencialidade desta arquitectura para diversos tipos de soluções.

Numa análise global a experiência obtida na implementação de cada módulo, assim como a sua utilização no trabalho final foi bastante agradável. A aquisição dos conhecimentos foram eficazes, até porque estivemos motivados e éramos motivados pelo docente, o que tornou a experiência bastante agradável e nos incentiva ainda mais a prosseguir o caminhos dos sistemas embebidos.

No entanto temos alguns reparos a indicar nos recursos fornecidos. No nosso entender, é objectivo da cadeira entender a arquitectura de um sistema embebido (nomeadamente o ARM7, que foi o utilizado) e não propriamente a aprender a codificar as funções/módulos. Não pretendemos dizer que não deve ser um facto de avaliação mas sim salientar que não se deve distribuir código quer seja na documentação disponibilizada que como ficheiros de auxílio, pois não se incentiva ao estudo, ao abraçar da arquitectura e entende-la.

Compreendemos que as actividades são guias para o trabalho a realizar e que os apontamentos fornecidos são para apoio ao estudo, contudo algo falha na transmissão do conhecimento.

É bastante mais lúdico se “não derem o peixe mas ensinem a pescar”, ou seja, em vez de colocarem código coloquem guias de execução. Nós optámos desde inicio por implementarmos nós as nossas funções e não utilizar o código fornecido, aprendemos muito, tivemos problemas, fomos forçados a entendê-los e a encontrar soluções para os resolver. O único módulo que tivemos muitas dificuldades foi na implementação do VIC e o documento disponível não foi útil devido principalmente à utilização da palavra inibir (causando entropia na formulação das funções).

Eventualmente os únicos pedaços de código que deveriam ser disponibilizados seriam o cstart.s e o ldscript.

Esta é a nossa opinião, valendo ela o que for.

Bibliografia

- http://ics.nxp.com/support/documents/microcontrollers/pdf/an10254.pdf
- http://ics.nxp.com/support/techdocs/microcontrollers/pdf/an10302.pdf
- http://www.nxp.com/documents/application_note/AN10369.pdf
- http://www.vas.co.kr/nxp/support/AN10381_1.pdf
- http://www.nxp.com/documents/application_note/AN10404.pdf
- *Documentação fornecida na meta-pagina da Unidade Curricular no Moodle*
- http://infocenter.arm.com/help/index.jsp

Agradecimentos

Não queremos terminar este relatório sem agradecer o auxílio, esforço e dedicação de um elemento sempre presente e que raramente reconhecido.

Este elemento durante todo o semestre nos auxiliou e foi nos exigindo qualidade e dedicação num pedaço do mundo com muitas potencialidades.

Já no passado a cooperação dele foi essencial no nosso desenvolvimento quer como alunos quer como profissionais, pois a sua atitude pró-activa e altruísta leva uma motivação extra na nossa parte.

Por tudo que já passou e pelo que há de vir, obrigado Prof. Pedro Sampaio.