EA871 – LAB. DE PROGRAMAÇÃO BÁSICA DE SISTEMAS DIGITAIS

EXPERIMENTO 5 – Módulos SIM, PORT, GPIO e SysTick

Profa. Wu Shin-Ting

OBJETIVO: Apresentação de interfaces de periféricos paralelos.

ASSUNTOS: Configuração e programação do LCD, programação do KL25Z para processamento de interface paralela com sinais GPIO, uso de SysTick para temporização dos sinais.

O que você deve ser capaz ao final deste experimento?

Saber usar sinais de habilitação para multiplexar os sinais num mesmo pino físico.

Saber gerar por *software* (GPIO) um pulso com uma largura pré-definida.

Saber compatibilizar os tempos de execução de dois dispositivos de velocidades diferentes.

Saber programar um *display* LCD.

Entender o princípio de funcionamento de um temporizador.

Saber computar um intervalo de tempo com base nos dados de um temporizador.

Saber configurar um temporizador para geração de eventos periódicos.

Saber declarar as *strings* e usar as funções disponíveis na biblioteca de C.

Saber coordenar as instruções simples e rápidas nas rotinas de serviço com instruções complementares e mais lentas do fluxo de controle principal.

Saber aplicar comandos de C para aumentar a legibilidade dos códigos.

Saber modularizar os códigos, com mínima dependência dos dados entre os módulos.

INTRODUÇÃO

Nos experimentos anteriores, limitamo-nos a utilizar sinais digitais, controlando cada periférico *bit* a *bit* para acionar periféricos simples, como LEDs e botoeira. Programamos o KL25Z para gerenciar até três LEDs e três botões no *shield* FEEC871 [2]. Os valores dos pinos aos quais os LEDs/botões estavam conectados eram manipulados individual- e diretamente através dos registradores de dados. Detalhamos no roteiro 4 [16] as configurações básicas dos registradores de configuração/controle do microcontrolador KL25Z, personalizando sua execução para as tarefas designadas.

Neste experimento, exploraremos dois periféricos populares em projetos de sistemas embarcados: o *display* LCD (*Liquid Crystal Display*) 2x(16 *bitmaps* 5x8) e o *latch* 74573, que está conectado a um conjunto de 8 LEDs (vermelhos) no *shield* FEEC871 [2]. Ambos suportam transferências paralelas de 8 *bits* (1 *byte*). Além dos 8 sinais de dados simultâneos, o microcontrolador deve gerar sinais de controle para que o LCD e o *latch* operem corretamente.

O esquemático do *shield* FEEC871 [2] mostra que os dois periféricos compartilham os 8 pinos `PTC0-PTC7` da porta C para sinais de dados e usam pinos distintos para o controle desses sinais: `PTC8` (RS) e `PTC9` (E) para o LCD e `PTC10` (LE) para o *latch*. Ambos os periféricos distinguem os sinais de dados em dois níveis lógicos '0' e '1'. Dado que as formas de onda desses periféricos não seguem um padrão específico, optamos por usar o módulo `GPIOC` *(General Purpose Input /Output*) para interfaceá-los com o microcontrolador.

A Figura 1 apresenta um diagrama de blocos do controle de envio de dados para o *latch*, usando os registradores dos módulos `SIM`, `PORTC` e `GPIOC`. Note a semelhança com o diagrama da figura 5 do roteiro 4 [16], que envolve apenas 1 sinal digital nas transferências.

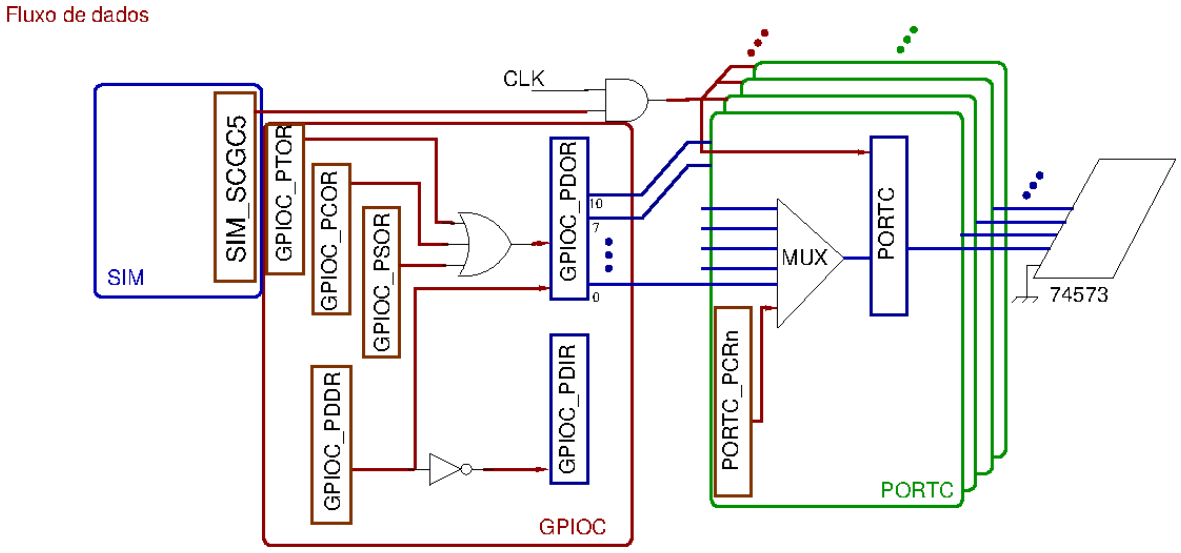


Figura 1: Diagrama de blocos de controle da transferência de *bytes* para *latch* 74573.

Periféricos: *Latch* 74573 e LCD

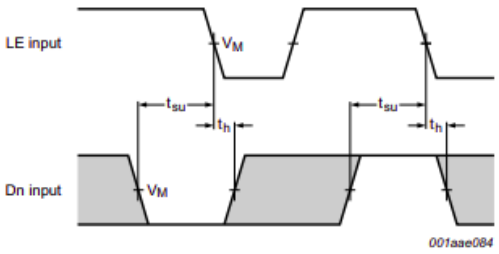
***Latch* 74573** é um circuito integrado projetado para transferir sinais entre os 8 pinos de entrada D0-D7 (Dn) e o latch, bem como entre o latch e os correspondentes pinos de saída Q0-Q7 (Qn). Essa transferência ocorre sempre que os sinais de habilitação LE e OE estão em níveis lógicos '1' e em '0', respectivamente [4].

Os valores mínimos de tensão associados aos níveis lógicos '0' e '1' são 0 e 2,7V, respectivamente, sendo compatíveis com os padrões do microcontrolador. Como $\overline{OE}$ está aterrado, os pinos Qn refletem sempre os valores dos pinos Dn, sujeitos a um atraso de propagação na ordem de dezenas de nanosegundos. Esse processo ocorre contanto que as restrições temporais na relação entre os sinais LE e Dn, conforme especificado pelo fabricante, sejam respeitadas.

A Figura 2 mostra as formas de onda de LE e Dn consideradas válidas para o *latch*, com os tempos *setup* e *hold* de LE superiores a 20ns e 15ns, respectivamente. Ou seja, se conseguirmos programar um microcontrolador para gerar essas formas de onda específicas, conseguiremos controlar o estado do *latch*/8 *leds* de forma precisa por meio de *software*.

Vimos no roteiro 3 [9] que uma instrução que não envolve acessos à memória requer, em média, 1 ciclo/tique de relógio, aproximadamente 48ns, para sua conclusão. Podemos usar essa informaçãopara **programar as formas de onda** da Figura 2.

Se setarmos os *bits* correspondentes a Dn no *byte* menos significativo do registrador `GPIOC_PDOR` e aplicarmos um pulso positivo de 48ns, gerado por um par de instruções de escrever '1' e escrever '0', no *bit* 10 de `GPIOC_PDOR`, teremos um tempo *setup* em torno de 48ns. Sendo a borda de descida de LE o instante em que o *latch* amostra os sinais nos pinos de entrada, é assegurado o tempo *hold* se escrevermos um valor em Dn depois deste pulso. Podemos certificar as relações das formas de onda programadas nos pinos com um osciloscópio ou um analisador lógico [13].



Measurement points are given in Table 8.

The shaded areas indicate when the input is permitted to change for predictable output performance.

Fig 10. Set-up and hold times for data input (Dn) to latch input (LE)

Table 8. Measurement points

Type	Input	Output
	V_M	V_M
74HC573	$0.5V_{CC}$	$0.5V_{CC}$
74HCT573	1.3 V	1.3 V

Figura 2: Formas de onda esperadas pelo *latch* 74573 (Fonte: [4]).

LCD (*Liquid Crystal Display*) é uma tela de tecnologia de cristal líquido usada para renderizar informações por um conjunto de pontos. Ele possui um controlador integrado que opera com um conjunto reduzido de instruções e dados de 8 *bits*. Ao contrário do *latch*, o *byte* enviado ao **LCD** pode ser um comando, dependendo do nível lógico da linha RS. Se RS=0, o *byte* é interpretado como um comando; caso contrário, como um dado (Seção 9/página 12 em [5]).

Cada *byte* recebido pelo LCD precisa de um intervalo de tempo, variando de micro a milisegundos, para ser processado (Seção 9/página 12 em [5]). Os glifos renderizados no visor do LCD são *bitmaps* de 5x8 pontos (*dots*), disponíveis numa tabela *look-up* na memória (CG)ROM do LCD (Seção 12/página 14 em [5]). **O endereço do *bitmap* de cada caractere alfa-numérico nessa tabela é o seu próprio código ASCII de 7 *bits***.

Os endereços dos códigos, em 8 *bits*, mostrados no visor do LCD são, por sua vez, armazenados numa outra memória de escrita do LCD, chamada DDRAM. O tamanho da DDRAM varia de acordo com o tamanho do visor do LCD. Para um visor 2x16 (2 linhas de 16 caracteres) mostrado na Figura 3, a DDRAM dispõe de 2x40 *bytes* para armazenar os endereços de até 80 *bitmaps*. Porém, apenas uma janela (móvel) de 2x16 *bitmaps* é visível no visor [6].

Para exibir um caractere alfa-numérico numa posição específica do visor do LCD, basta escrever no endereço correspondente da DDRAM o seu código ASCII (Seção 11/página 13 em [5]) que é decodificado internamente como o endereço do seu *bitmap* em CGROM (Seção 12/página 14 em [5]).

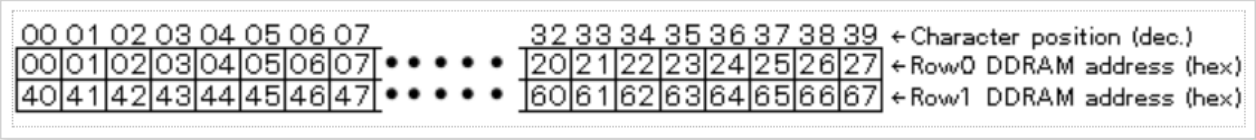


Figura 3: DDRAM do visor de um LCD 2x16 (fonte: [6]).

Além dos *bitmaps* predefinidos, o LCD utilizado nesta disciplina dispõe de uma memória adicional de 64 *bytes*, denominada de CGRAM, na qual se pode gravar até 8 *bitmaps* personaliados. Cada *bitmap* ocupa um espaço de endereços de i*8 a i*8+7 da memória CGRAM, onde i é o endereço do *bitmap* mapeado na tabela *look-up* (memória CGROM). As referências [7] [8] dispõem um editor de *bitmaps* 5x8 para facilitar essa personalização.

O LCD tem dois comandos para setar o contador de endereço referente às memórias DDRAM (visor) e CGRAM (*bitmaps* personalizados) (Seção 9/página 12 em [5]). Ao enviar um comando "*Set DDRAM Address*" 0b1(AC6)(AC5)(AC4)(AC3)(AC2)(AC1)(AC0), todos os *bytes* transferidos na sequência são armazenados em DDRAM sequencialmente a partir do endereço 0b0(AC6)(AC5)(AC4)(AC3)(AC2)(AC1)(AC0). A escrita no endereço 0b00(AC5)(AC4)(AC3)(AC2)(AC1)(AC0) CGRAM ocorre de forma análoga com o comando "*Set CGRAM Address*" (0b01(AC5)(AC4)(AC3)(AC2)(AC1)(AC0)) (Figura 4). Uma vez setado um endereço, ele é incrementado automaticamente em cada acesso de escrita.

Set CG RAM Address	0	0	0	1	AC 5	AC 4	AC 3	AC 2	AC 1	AC 0	Sets CG RAM address in address counter.	39 µS
Set DD RAM Address	0	0	1	AC 6	AC 5	AC 4	AC 3	AC 2	AC 1	AC 0	Sets DD RAM address in address counter.	39 µS

Figura 4: Comandos distintos para endereçamento de DDRAM e CGRAM (fonte: [5]).

Assim como o *latch*, o LCD espera que os sinais de dados (DB0 – DB7) e o sinal de habilitação (E) guardem a relação temporal conforme especificada pelo fabricante na Figura 5. O sinal E deve ter uma largura superior a 450ns, com tempos *setup* e *hold* de 60ns e 5ns, respectivamente [5]. Ao contrário do *latch*, a resposta do LCD é da ordem de dezenas a centenas de microsegundos, significativamente maior do que um ciclo de instrução em KL25Z. A última coluna na figura 4 mostra que o tempo de processamento dos comandos "*Set DDRAM Address*" e "*Set DDRAM Address*" é de 39us. Para evitar sobreposição de dados enviados ao LCD, devemos espaçar os dois envios subsequentes de acordo com os tempos estipulados pelo fabricante (Seção 9/página 12 em [5]).

Outra diferença no controle entre o *latch* e o LCD é a fase de inicialização. Enquanto o *latch* dispensa qualquer processo de inicialização, o LCD requer uma sequência específica de instruções quando é energizado, conforme detalhado na Seção 10/página 13 em [5], de acordo com as especificações do fabricante.

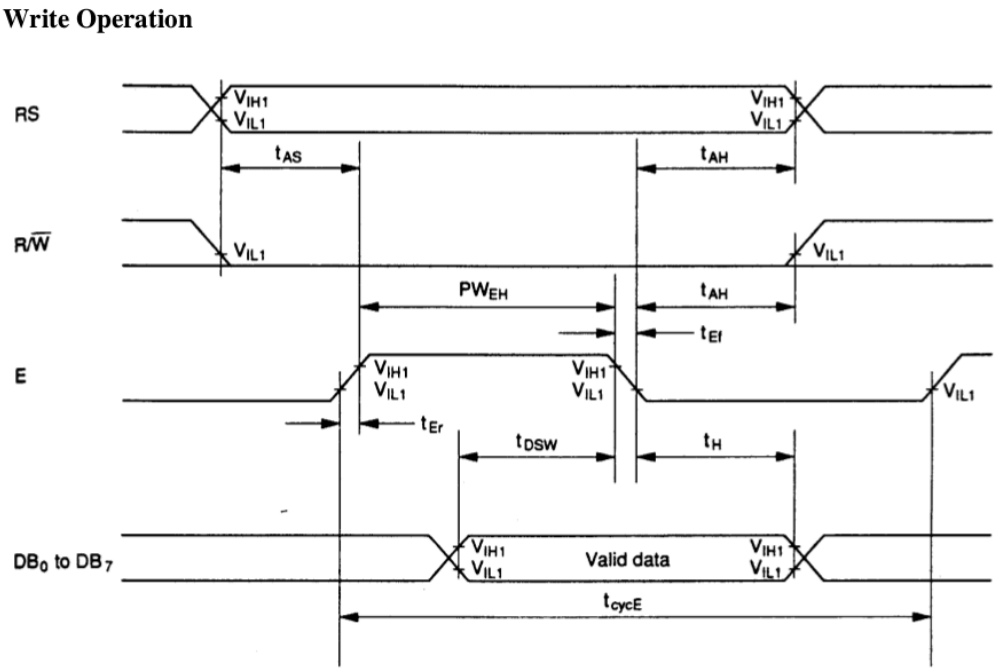


Figura 5: Diagrama de tempo do LCD (fonte: [5]).

Em outras palavras, se conseguirmos **programar um microcontrolador para gerar os sinais processáveis pelo LCD**, poderemos controlá-lo por meio de *software*. Uma opção para que o módulo `GPIOC` gere formas de onda compatíveis com as mostradas na Figura 5 é setar os valores DB0 – DB7 no *byte* menos significativo de `GPIOC_PDOR`. Isso deve ser feito antes de aplicar um pulso de largura igual ou maior que 450ns, usando três instruções específicas: escrever '1', inserir um tempo de espera maior que 450ns e escrever '0' no *bit* 9 de `GPIOC_PDOR`). Podemos certificar as relações das formas de onda geradas nos pinos com um osciloscópio ou um analisador lógico [13].

O espaçamento de tempo requerido para processamento de cada *byte* capturado pode ser por espera ocupada (*busy-waiting*, aguardando até que a *flag* `BUSY` do LCD seja resetada), por *polling* (fazendo amostragem periódico da a *flag* `BUSY` e executando outros comandos no intervalo entre as amostragens), ou por espera pré-fixada com base nos tempos especificados pelo fabricante. Adotamos essa última solução nesta disciplina porque na forma como o pino R/$\overline{W}$ do LCD está aterrado no *shield* FEEC871, não podemos fazer acessos de leitura da *flag* `BUSY`. Usaremos a função `espera_2us` implementada no roteiro 3 [23] para o processador esperar pela conclusão do processamento do LCD antes de uma nova traansferência. Essa programação vale para o envio da sequência de instruções recomendada pelo fabricante na inicialização do LCD, que exige a habilitação dos pinos `PORTC0` – `PORTC9` do microcontrolador.

Temporizadores

Temporizadores constituem módulos básicos de qualquer microcontrolador, desempenhando um papel essencial na coordenação das tarefas e ações por meio dos eventos periódicos que geram [21]. Eles são intimamente relacionados com os **contadores**. O princípio de funcionamento de um temporizador consiste em contar a quantidade N dos ciclos/tiques de relógio de frequência *f* conhecida, a partir dos quais podemos computar o intervalo de tempo *t* correspondente por

$$t = N \times \frac{1}{f} \qquad (1)$$

Se substituirmos a fonte de sinais de relógio por uma fonte de eventos externos, o mesmo circuito pode ser aproveitado para fazer contagem de eventos externos.

O valor máximo REF de contagem de um contador depende da quantidade dos *bits* que ele possui. Quando a contagem atinge este valor máximo, caracterizamos issso como um ***Overflow***. Todos os temporizadores conseguem detectar este evento e, em conjunto com um circuito de interrupção, podem tratá-lo como uma interrupção. Para ajustar a frequência dos sinais de relógio (*clock ticks*) provenientes da fonte, muitos temporizadores possuem divisores de frequência, comumente conhecidos como ***prescalers***. Além disso, alguns temporizadores apresentam um circuito integrado que divide a frequência dos *overflows* do contador, reduzindo a taxa de sua ocorrência; esse componente é chamado de ***postscaler***.

Levando em conta esses divisores, define-se como **período** de um temporizador o intervalo de tempo *T* gasto para fazer uma contagem completa até que ocorra um evento de *Overflow* no temporizador:

$$T = REF \times \frac{prs}{f} \times pos, \quad (2)$$

sendo *prs* e *pos* divisores de frequência *prescaler* e *postscaler*, respectivamente. A expressão nos mostra que é possível controlar o período de um temporizador por 4 parâmetros: REF, *prs*, *pos* e *f*. Mais especificamente, fixados *f*, *prs* e *pos*, podemos controlar a periodicidade de interrupções T de um temporizador através da configuração de REF

$$REF = T \times \frac{f}{prs \times pos}. \quad (3)$$

Os temporizadores propriamente ditos não geram nem capturam sinais. Portanto, eles não precisam de pinos para trocas com o mundo externo. Pela tabela da Seção 10.3.1/página 161 em [1] pode-se constatar que não há pinos alocados para módulos que só geram eventos periódicos, `SysTick` (Seção 3.3/página 275 em [3]), `RTC` (Capítulo 34/ página 597 em [1]) e PIT (Capítulo 32/ página 573 em [1]).

Módulo SysTick

No roteiro 4 [16] vimos que são integrados nos microcontroladores baseados na arquitetura ARM um temporizador conhecido por `SysTick` (Seção B3.3/página 275 em [3]) visando essencialmente à geração de interrupções acuradas para escalonar diferentes tarefas de um sistema operacional em tempo real (*Real Time Operating System*, RTOS) [20]. No entanto, muitos desenvolvedores utilizam este temporizador para aumentar a acurácia das funções de espera que atrasam (*delay*) o tempo de execução do processador. Neste experimento, ele é aplicado na implementação do controle da periodicidade de mudança dos estados do LED RGB.

O `SysTick` é um temporizador de 24 *bits*, de contagem decrescente, que não incorpora circuitos de *pre* e *postscaler*. Ele recarrega automaticamente o registrador `SYST_CVR` com o valor REF quando atinge 0 (*wrap-on-zero*). Neste momento, o *bit* de estado `COUNTFLAG` é automaticamente setado em '1'. Esse *bit* é sempre reiniciado ao ocorrer um acesso de escrita (*clear-on-write*) no contador `SYST_CVR`. A Seção B3.3.1/página 275 em [3] fornece uma descrição detalhada das operações do `SysTick`.

A fonte de relógio do `SysTick` é tipicamente a do próprio processador. Nesta disciplina, adotamos `MCGOUTCLK/MCGFLLCLK` (20.971.520Hz) [16]. O circuito do temporizador `SysTick` é configurável através dos registradores `SYST_CSR`, responsável por habilitar o contador e sua interrupção, e `SYST_RVR`, que contém o valor REF. Além disso, o registrador `SYST_CSR` inclui o *bit* `COUNTFLAG`. Sendo os eventos gerados pelo `SysTick` sincronizados com os sinais de relógio

do processador, eles são associados ao número de exceção 15 (Tabela 3-7/página 52 em [1]). É declarado `SysTick_Handler` como a sua rotina de serviço em `Project_Settings/Startup_Code/kinetis_sysinit.c`.

O período T máximo configurável no `SysTick` é aproximadamente 0.8s, pois a contagem máxima 2^{24}-1 na frequência 20.971.520Hz. Porém, mesmo na falta de um circuito *postscaler*, podemos fazer a contagem de ocorrências de *Overflows* através de *software*, em `SysTick_Handler` para onde é desviado o fluxo de controle em cada período T (Eq. 2). Segue-se uma implementação em que o controle da multiplicidade de T é feita através dos incrementos da **variável local estática** `multiplos`:

```
void SysTick_Handler () {
     static uint32_t multiplos = 0;
     multiplos++;
     if (multiplos == pos) {
          // perfez um período pos x T
          multiplos = 0;
     }
}
```

No projeto `rot5_aula` [22] aplicamos esta técnica para criar distintos intervalos de tempo usando `T` como a unidade base de tempo. Note que um período T correponde a um ciclo completo de contagem, indo de 0 a REF, representando assim um total de REF pulsos/tiques de relógio. Portanto, sempre que possível, faz-se um acesso de escrita no contador `SYST_CVR` para que a contagem inicia em REF (Seção B3.3.1/página 275 em [3]) antes de iniciar uma nova contagem de período. Além disso, é relevante notar que o estado de interrupção do módulo é automaticamente resetado assim que o evento é atendido.

Processamento do LCD

O LCD é um dos periféricos mais utilizados para realimentar os eventos de interrupção gerados pelos usuários. Uma implementação mais simplificada e direta seria incluir a transferência de uma sequência de *bytes* para o LCD nas rotinas de serviço, onde são tratados os eventos de interrupção. No entanto, devido à sua menor velocidade de processamento em comparação com o processador, são inseridos diversos estados de espera, contratando com o enfoque apresentado no roteiro 4 [16]: as rotinas de serviço devem ser mais simples e lineares possíveis, para evitar que o sistema fique preso nesses processamentos sem fazer avanços significativos no fluxo de controle do sistema [24].

Para que tais rotinas não virem fontes de *bugs* e problemas, é recomendável executar as instruções relacionadas com o LCD fora das rotinas de serviço, mas coordenadas com os eventos de interrupção. O projeto `rot5_aula` [22] apresenta uma solução baseada em divisão das tarefas entre a rotina de serviço e o fluxo de controle principal implementado na função `main`. Dentro da rotina de serviço são apenas executadas as instruções essencialmente necessárias ou de baixo impacto em termos de ciclos de relógio, enquanto as instruções mais complexas e demoradas são adiadas para o fluxo de controle principal, que consiste em um laço contínuo e cíclico de verificação dos estados do sistema e execução de ações pertinentes.

Se considerarmos que as ações complexas envolvidas numa transição de estado incluem a atualização do visor do LCD, do estado do LED e a atualização do *timeout*, essas ações serão então repetidamente executadas no fluxo de conttrole principal até ocorrer uma mudança para um novo estado. Neste caso, a contagem de *timeout* será resetado em cada iteração, travando a contagem em zero e impedindo o progresso do fluxo. Para evitar esse problema, diferenciamos esse estado de

transição em dois estados distintos: a transição propriamente dita, na qual as atualizações necessárias para o novo estado são realizadas uma única vez, e o estado de espera pelo próximo estado, que chamamos de estado de regime, no qual nenhuma atualização deve ser feita.

Em `rot5_aula` distinguimos três estados de regime, `VERMELHO`, `VERDE` e `AMARELO`, correspondendo a 3 estados de transição, `VERMELHO_VERDE`, `VERDE_AMARELO` e `AMARELO_VERMELHO`, respectivamente. Dentro da rotina de serviço `SysTick_Handler`, são apenas tratados os estados de regime atulizados na função `main`, enquanto os estados de transição processados na função `main` são atualizados na rotina de serviço. Esse método de zigue-zague de controle assegura o sequenciamento adequado das ações do sistema, evitando conflitos e garantindo um funcionamento suave e eficiente.

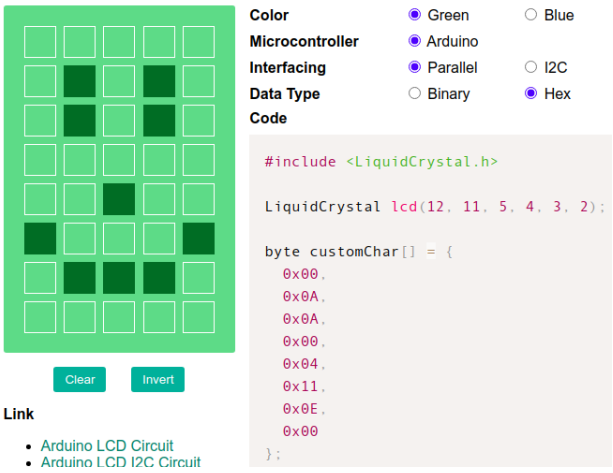
***Strings* em C**

Do ponto de vista do envio de dados, seja para o *latch* 74573 (representado por Dn) ou para o LCD (representado por DBn) via KL25Z, o tipo de dados mais apropriado para representá-los em C é `char` (tipo de dado nativo de C), `uint8_t` (tipo de dado definido em `stdint.h`) ou `byte` em alguns sistemas. No roteiro 2 [14], vimos que a unidade de processamento desses tipos de dados é em *bytes* (8 *bits*). Assim, podemos processar uma sequência de *bytes* usando o arranjo (*array*) de tipo de dados `char`/`uint8_t`/`byte`.

Figura 6(a) ilustra a definição dos pontos (*dots*) de um *bitmap* a serem enviados para CGRAM como uma sequência de *bytes* em hexadecimal (`customChar`), enquanto a Figura 6(b) mostra duas sequências de *bytes,* em ASCII de 7 *bits* (`str1`) e hexadecimal (`str2`), a serem enviadas para DDRAM. Em ambos os casos, usam-se **arranjos/vetores de elementos do tipo `char`** para alocar espaços na memória.

Um arranjo/vetor do tipo `char` contendo uma sequência de caracteres seguida do terminador '\0' (0x00 em hexadecimal) é tratado em C como uma ***string*** [10]. **A quantidade de elementos declarada para uma *string* deve ser sempre a quantidade máxima de caracteres mais 1 *byte* reservado para o terminador**. A biblioteca padrão de C oferece uma série de funções para processamento de *strings*, todas considerando que o valor 0x00 ('\0') marca o término da sequência de caracteres.

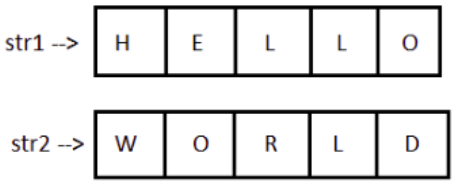
Por exemplo, se processarmos o arranjo de caracteres declarado na Figura 6(a) com essas funções, a execução será interrompida no primeiro elemento do arranjo, pois ele é 0x00, o marcador do fim de uma sequência de caracteres. Este é um dos problemas discutidos em [11] ao usar *strings* para processar arranjos de elementos do tipo `char`. Como 0x00 é um endereço válido para CGROM do LCD, há alternativas para contornar essa incompatibilidade, tais como: (1) renunciar as funções "de prateleira" e usar diretamente a quantidade de caracteres armazenados no arranjo, como feito em `GPIO_escreveBitmapLCD` do arquivo `GPIO_latch_lcd.c` de `rot5_aula` [22]; e (2) evitar o uso do endereço 0x00 do CGROM.



(a) declaração de um *bitmap* como um arranjo de elementos do tipo `char`, que recebe valores em hexadecimal.

char str1[5]={'H','E','L','L','O'};

char str2[5]={0x57,0x4F, 0x52,0x4C,0x44};



(b) declaração de mensagens como arranjos de elementos do tipo `char`, que recebem valores em ASCII (`str1`) e hexadecimal (`str2`).

Figura 6: Uso de arranjos do tipo `char`/`uint8_t` para processamento de dados do *latch* e do LCD.

Em C, é possível declarar arranjos de *strings* para organizar uma sequência de *strings* [12]. Ao declará-los como um arrranjo de *strings* com inicialização, permitimos que o compilador insira automaticamente o terminador e determine a quantidade de *bytes* a serem alocados na memória. A seguinte declaração

```
char cor_nome[3][]= {
        "vermelho",
        "verde",
        "azul"
}
```

é equivalente a declarar um arranjo bidimensional (matriz) de elementos do tipo `char`, onde as linhas correspondem às *strings* e as colunas, os caracteres de cada *string*. Neste caso, o terminador 0x00 ('\0') deve ser inserido explicitamente:

```
char cor_nome[3][9]= {
        {'v', 'e', 'r', 'm', 'e', 'l', 'h', 'o', '\0'},
        {'v', 'e', 'r', 'd', 'e', '\0'},
        {'a', 'z', 'u', 'l', '\0'}
}
```

Ou podemos simplificar ainda mais, permitindo que o compilador acrescente o terminador em cada *string* e determine tanto a quantidade de *strings* quanto a quantidade de *bytes* por *string*. Isso é feito ao declarar um arranjo de ponteiros para o tipo `char` com inicializações:

```
char *cor_nome[]= {
        "vermelho",
        "verde",
        "azul"
}
```

Note a diferença na representação de um caractere e de uma *string*. **Um caractere é delimitado por aspas simples, enquanto uma *string* é por aspas duplas**. Um caractere, por exemplo, 'a', ocupa um *byte* e uma *string* "a", ocupa dois *bytes* – o caractere 'a' e o terminador '\0'.

Enum em C

Essencialmente, o tipo de dado **enum** consiste em associar nomes mnemônicos a um conjunto finito de constantes numéricas, como evidenciado na definição de `enum lcd_RS_tag` (redefinido para o tipo de dado `tipo_lcd_RS` em `GPIO_latch_lcd.h`), `enum boolean_tag` (redefinido para o tipo de dado `tipo_booleano` em `util.h`) e `enum estado_tag` em `rot5_aula` [22]. A primeiro enumeração associa, respectivamente, as constantes numéricas 0 e 1 aos nomes `OFF` e `ON`, a segunda associa `COMANDO` e `DADO` às constantes numéricas 0 e 1, respectivamente, e a terceira se refere aos 7 estados do sistema, `AZUL`, `VERMELHO`, `VERDE` e `AMARELO`, `VERMELHO_VERDE`, `VERDE_AMARELO` e `AMARELO_VERMELHO`. O **pré-processador** de C substitui esses nomes pelos valores numéricos antes da compilação do código.

Switch/case em C

O comando condicional **if-else** é tipicamente aplicado para selecionar/decidir um bloco de instruções com base nos valores de uma expressão aritmética e/ou lógica em linguagens de médio nível para cima, como C. Além de if-else, C suporta o comando condicional **switch-case** que toma decisões comparando o valor de uma variável de decisão com *strings*, valores numéricos inteiros ou constantes enumeradas (do tipo de dado `enum`).

No **switch-case**, as alternativas para um valor de decisão são precedidas pela palavra reservada **`case`** e o bloco de instruções correspondente é encerrado com a palavra reservada **`break`**, que transfere o fluxo de controle para o próximo comando. Caso o bloco não seja encerrado com `break` num `case`, o fluxo de controle segue sequencialmente para o bloco de instruções do `case` seguinte até encontrar um `break` ou uma chave que fecha o escopo do comando `switch`. Opcionalmente, pode-se incluir a alternativa **`default`**, associado a um bloco de instruções que trata dos casos não contemplados pelos `cases`.

Embora menos versátil do que o comando if-else, o **switch-case** tem demonstrando um desempenho melhor do que **if-else** em muitos compiladores, quando a quantidade de alternativas é muito grande e os valores de decisão são limitados a um domínio finito [26]. Além disso, o código fica mais bem estruturado, como demonstra a implementação das funções `main` e `SysTick_Handler` no projeto `rot5_aula` [22]. A organização de todos os casos precedidos pelos nomes mnemônicos dos estados em blocos facilita a leitura dos códigos.

Fatores Internos de Qualidade de *Software*

Cabe introduzir aqui, sob o ponto de vista estrutural de um aplicativo, três fatores internos relevantes para a qualidade de um *software*: legibilidade, modularidade e reusabilidade. A l**egibilidade** está relacionada com o grau de facilidade em entender as instruções codificadas e compreender o fluxo de controle subjacente. A m**odularidade**, po sua vez, consiste na divisão de um programa em trechos de códigos com funções bem definidas, buscando a maior independência possível em relação ao restante do código. Essa abordagem não apenas eleva a confiabilidade, legibilidade e manutenção do *software*, mas também proporciona maior flexibilidade no uso dos códigos. Qaunto à **reusabilidade**, essa característica diz respeito à reutilização de códigos desenvolvidos em diferentes contextos, de forma a reduzir custos de manutenção e produção de

software. Os conceitos de modularidade e reusabilidade são intimamente relacionados. Quanto mais modulares são os aplicativos, mais fácil é a identificação de pontos comuns entre eles, mesmo que eles sejam destinados a finalidades bastante distintas.

MACROS

Uma técnica que aumenta a **legibilidade** dos códigos em C é o uso do tipo de dados `enum` definido pelo desenvolvedor [18], como comentado anteriormente. Uma outra técnica muito difundida em programação de microcontroladores é o **uso de macro(instruçõe)s**. No roteiro 4 [16] vimos que podemos abstrair a interface entre os circuitos dedicados de um microcontrolador e os periféricos em campos de *bits* de uma série de registradores mapeados no espaço de endereços de memória do processador. Vimos no roteiro 2 [14] que, com uso de mascaramentos de *bits*, podemos modificar seletivamente os campos de *bits*. Na maioria dos casos é necessário mais de uma operação binária, tornando o código repetitivo e às vezes difícil de entender. No IDE CodeWarrior encontra-se definida no arquivo-cabeçalho `Project_Headers/MKL25Z4.h` uma série de macros que facilitam manipulações dos campos de *bits* de todos os registradores disponíveis em KL25Z. Para usá-las num programa em C, basta incluir a diretiva

```
#include "derivative.h"
```

Todas as macros definidas no arquivo-cabeçalho `MKL25Z4.h`, incluído por sua vez no arquivo-cabeçalho `derivative.h`, referenciam os registradores seguindo a mesma convenção de nomes utilizados pelo fabricante nos manuais de referência, como em [1]. Os nomes das macros seguem uma convenção que inclui o nome do módulo ao qual o registrador pertence, seguido pelo nome do registrador, separados por um traço de ligação. Por exemplo, a macro `PORTB_PCR19` identifica o registrador `PCR19` do módulo `PORTB`. Os nomes das macros que definem uma máscara com um valor específico num campo de um registrador contêm 3 nomes: do módulo, do registrador e do campo. Por exemplo, a macro `PORT_PCR_MUX (x)` define uma máscara do tamanho de um registrador `POTRx_PCRn` (32 *bits*) com o valor `x` no campo `MUX` e o restante dos *bits* em 0

```
#define PORT_PCR_MUX(x)
(((uint32_t)(((uint32_t)(x))<<PORT_PCR_MUX_SHIFT))&PORT_PCR_MUX_MASK)
```

As macros que definem uma máscara com todos os *bits* de um campo de um registrador de um módulo em '1' tem o sufixo `MASK`, como `PORT_PCR_MUX_MASK` que define uma máscara com todos os 3 *bits* do campo `PORT_PCR_MUX` em '1' e os restantes *bits* em '0'

```
#define PORT_PCR_MUX_MASK 0x700u
```

Há ainda macros que definem deslocamentos necessários para um valor alinhado com o *bit* 0 de um registrador para um campo específico do registrador tem o sufixo `SHIFT`, como `PORT_PCR_MUX_SHIFT` que define a quantidade de *bits* a serem deslocados para que um valor setado num registrador `PORTx_PCRn` seja deslocado para o campo `MUX`

```
#define PORT_PCR_MUX_SHIFT 8.
```

As macros são expandidas pelo **pré-processador** C de forma recursiva até as constantes numéricas (binárias) antes da compilação [17]. Por exemplo, um mascaramento com uma máscara de OU definida pela macro `PORT_PCR_MUX(x)`

```
PORTB_PCR19 |= PORT_PCR_MUX (x);
```

é expandida em

```
 ((PORT_MemMapPtr)0x4004A000u)→PCR[19] |=
(((uint32_t)(((uint32_t)(x))<<PORT_PCR_MUX_SHIFT))&PORT_PCR_MUX_MASK)
```

que equivale a

```
 ((struct PORT_MemMap volatile *)0x4004A000u)→PCR[19] |=
                (((uint32_t)(((uint32_t)(x))<<8))&0x700u)
```

que pode se desdobrar na forma implementada no projeto rot4_aula 30]

```
(*(uint32_t volatile *)0x4004A04Cu) |=
                (((uint32_t)(((uint32_t)(x))<<8))&0x700u)
```

Note que para o campo MUX de PORTB_PCR19 fique em x depois do mascaramento, é necessário que os *bits* no campo sejam '0' antes do mascaramento. Quando os campos dos registradores não estejam em '0' é necessário resetar os seus *bits* em '0' antes de aplicar uma máscara OU. Por exemplo, o registrador PORTA_PCR4_MUX é inicializado em 0b111 (coluna Default na tabela da Seção 10.3.1/página 161 em [1]). Para setar 0b001 (modo de multiplexação GPIO) no seu campo MUX com uma máscara OU, deve-se aplicar o seguinte mascaramento (máscara AND) para zerar os *bits* antes da operação OU:

```
PORTA_PCR4 &= ~PORT_PCR_MUX (0b111);
```

O projeto rot5_aula [22] foi integralmente programado com uso de macros, exceto a função GPIO_initSwithNMI em GPIO_switches.c que não é usada. Foi incluído o arquivo-cabeçalho derivative.h nos arquivos GPIO_latch_lcd.c, GPIO_ledRGB.c, ISR.c e SysTick.c, que fazem acessos diretos aos registradores do microcontrolador. Na perspectiva C/C++ do IDE CodeWarrior pode-se visualizar a expansão de cada macro passando o cursor sobre ela.

Modularização

Para promover a **modularidade** nos códigos e maximizar o encapsulamento dos detalhes de implementação de cada módulo, é recomendável evitar o uso de **variáveis globais no compartilhamento de dados entre diferentes módulos**. Essa prática previne inúmeros problemas que variáveis globais podem causar durante o desenvolvimento de um projeto por uma equipe de programadores e sua subsequente manutenção, como detalhado em [25].

Uma estratégia de **encapsulamento** em C consiste em declarar tais variáveis com o qualificador de acesso static, tornando-as visíveis apenas no arquivo em que são definidas, mas preservando seu conteúdo enquanto o programa está em execução. A troca de dados entre essas variáveis e outros módulos é realizada por meio de um conjunto de funções. Por exemplo, ao implementar rotinas de serviço sem estados de espera em rot5_aula [22], os comandos do LCD foram deslocados de SysTick_Handler para main. Se as funções SysTick_Handler e main estiverem no mesmo arquivo, declarar a variável de estado como uma variável global do arquivo (fora do escopo das duas funções) com o qualificador static é suficiente para compartilhá-la entre elas.

No caso específico da implementação de rot5_aula [22], as duas funções foram estruturadas em arquivos distintos, ISR.c e main.c. No arquivo ISR.c, são declaradas três variáveis globais qualificadas como static (counter, maxCounter e estado). Os valores dessas variáveis são acessados em main.c para controlar o fluxo de execução principal. Quatro funções foram implementadas para possibilitar esses acessos sem o compartilhamento direto do espaço de memória das variáveis: ISR_ResetaCounter, ISR_SetaMaxPostScaler, ISR_escreveEstado e ISR_leEstado.

No projeto rot5_aula dividimos funcionalmente o código em 6 arquivos, além de main.c e sa_mtb.c: GPIO_latch_lcd.c (contém funções de processamento do LCD), GPIO_ledRGB.c (contém funções de processamento do LED RGB), GPIO_switches.c (contém funções de processamento dos botões), ISR.c (contém funções de processamento de

eventos de interrupção), `SysTick.c` (contem funções de processamento do temporizador) e `util.c` (contém funções utilitárias gerais). Isso facilita a **manutenção** e o **reuso**.

Bibliotecas de funções

A ideia de evitar duplicação de códigos-fonte num arquivo, vale também para repetições de uma mesma função em diferentes projetos. Um desenvolvedor se beneficia com o reuso dos códigos amplamente testados, dedicando-se o seu tempo à implementação e aos testes de novas funções. Para **reuso**, é comum **modularizar os códigos** em funções simples e organizá-los em bibliotecas de funções pré-compiladas. As **bibliotecas** podem ser **estáticas**, as ligadas estaticamente para gerar um arquivo executável que inclui todos os "códigos de reuso", ou **dinâmicas**, as ligadas dinamicamente para gerar um arquivo executável que contém somente referências aos "códigos de reuso".

No caso de bibliotecas dinâmicas, embora os códigos de reuso sejam compartilhados entre os códigos executáveis, eles precisam ser carregados e ligados no tempo de execução – uma tarefa executada usualmente pelo sistema operacional [19]. Como não temos um sistema operacional em tempo real (*Real Time Operating System*, RTOS) [20] instalado nos nossos microcontroladores, implementar funções de carga e ligação dinâmicas é trabalhoso. Portanto, é mais comum o uso de bibliotecas estáticas para reuso das funções pré-implementadas em microcontroladores de programação *baremetal*. Vale ressaltar que, para construir um código executável reusando as funções de uma biblioteca, é necessário que os códigos de máquina contidos nas bibliotecas sejam compatíveis com a arquitetura do processador em que o arquivo-alvo vai ser executado.

É importante frisar que disponibilizar os módulos de códigos em formato de bibliotecas sem documentação não assegura a sua reusabilidade, porque para usar as funções de forma correta precisamos saber não só o quê elas fazem (definição) como também a sua interface. Em linguagem C, esta interface, denominada o **protótipo**, ou **declaração**, de **função**, compreende o nome de uma função, os tipos e a ordem dos seus argumentos, e o tipo de seu retorno.

EXPERIMENTO

Neste experimento vamos implementar um temporizador `pomadoro`, uma técnica desenvolvida por Francesco Cirillo nos anos 1980, com o objetivo de aprimorar a eficiência e concentração no trabalho ou estudo [27]. Esse método, conhecido por dividir o tempo em intervalos chamados de **pomodoros** de 25 minutos, seguidos por **pausas curtas** de 5 minutos, visa reduzir a fadiga mental e aumentar a produtividade. Após quatro pomodoros, recomenda-se uma **pausa longa** de 15 a 30 minutos. O ciclo se repete até a conclusão da atividade. O diagrama de máquina de estados do dispositivo é ilustrado na Figura 7. O termo pomodoro (tomate em italiano) provém do temporizador de cozinha em forma de tomate utilizado pelo criador.

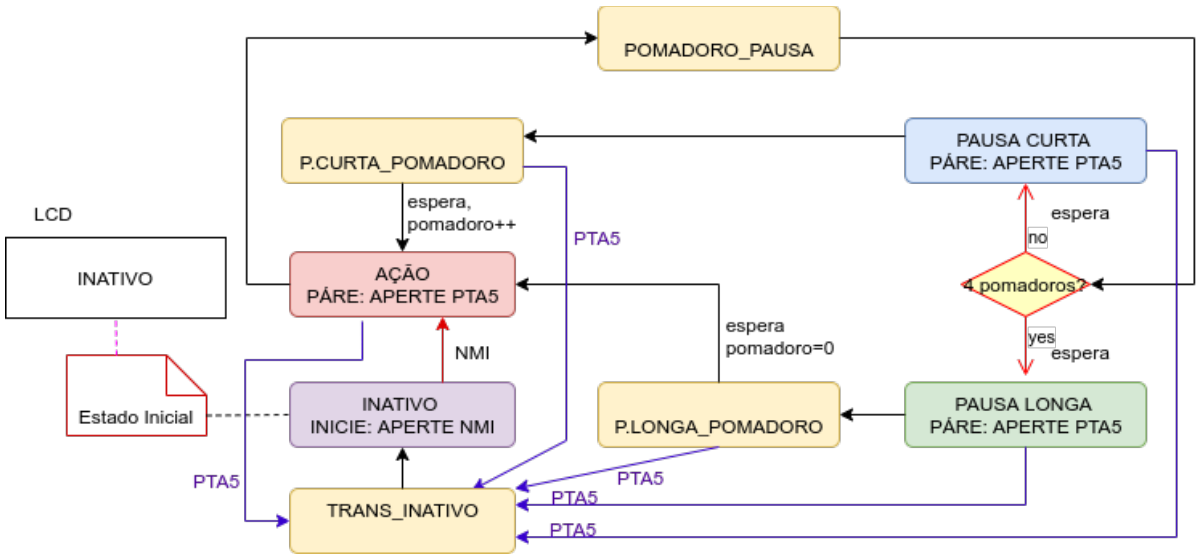


Figura 7: Diagrama de máquina de estados para o projeto `pomadoro` em UML (editado em [28]).

São usados os 3 LEDs para representar os estados de pomadoro (R), pausa curta (B) e pausa longa (G), e 2 botões para ligar o temporizador (NMI) e desligá-lo (PTA5). Para garantir que os usuários daltônicos possam identificar **os estados de regime, estes são mostrados em texto na primeira linha do visor do LCD**, `INATIVO`, `AÇÃO`, `PAUSA CURTA`, `PAUSA LONGA`. Os **estados de transição**, representados pelos blocos amarelos na Fgirua 7, são responsáveis pelas necessárias atualizações dos recursos, conforme as características especificadas para o estado de regime correspondente. **Na segunda linha do visor do LCD, é mostrada a instrução de controle para iniciar ou interromper o temporizador.** Além dos LEDs, botões e LCD, o projeto envolve o uso do temporizador `SysTick` e a programação da lógica para processar os estados e transitar entre eles.

No estado `INATIVO`, os LEDs são apagados. Ao pressionar o botão `NMI`, na **borda de descida**, as indicações luminosas acendem ciclicamente, seguindo o fluxo mostrado na Figura 7. Essas indicações se repetem ciclicamente até que o botão `PTA5` seja pressionado e a sua **borda de descida** detectada. O sistema volta para o estado `INATIVO` com os LEDs apagados e a mensagem `INATIVO` exibida no visor do LCD. Para a prototipagem, usamos intervalos de 5.25s, 1.25s e 4.25s para os períodos de atividade, pausa curta e pausa longa, respectivamente. **Interrupções periódidas** do `SysTick` são aplicadas para determinar as transições entre os estados de regime.

Figura 8 mostra o diagrama de componentes a serem utilizados na implementação das funções projetadas.

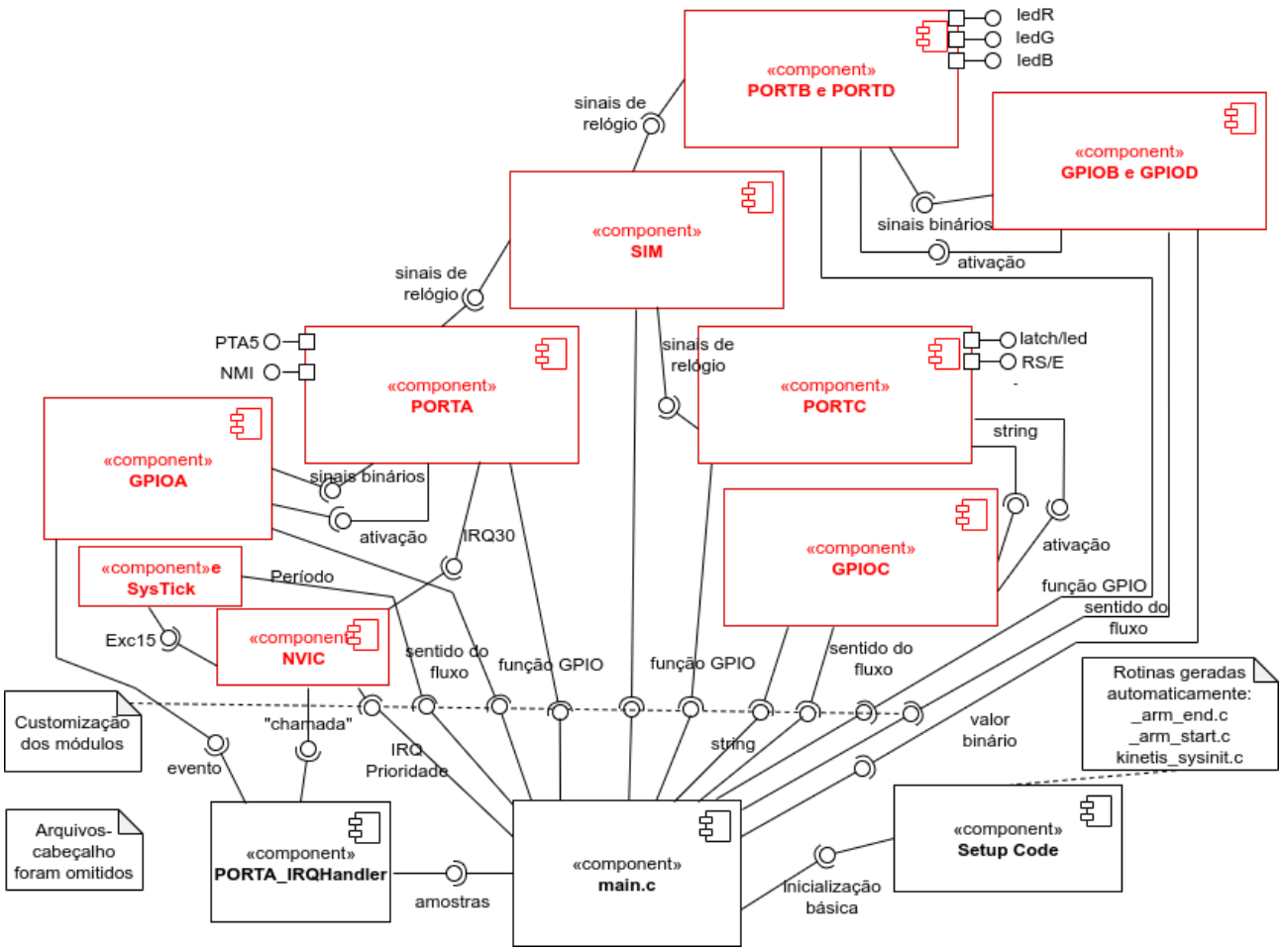


Figura 8: Diagrama de componentes do projeto `pomadoro` (editado em [28]).

Segue-se um roteiro para o desenvolvimento do projeto. Usa-se na implementação do projeto as macros do arquivo-cabeçalho `derivative.h`.

1. **De conceitos para práticas:** Carregue o projeto `rot5_aula` [22] no IDE CodeWarrior, substitua a função `espera_2us` definida em `util.c` pela função que você implementou no roteiro 3 [9] e gere o executável. Conectando o *kit* LED RGB nos pinos de H5 pode ver efeitos dos sinais gerados pelas cores dos LEDs. Conecte quatro canais do analisador lógico Saleae [13] nos pinos `PTE20` (forma de onda de `RS`), `PTE21` (forma de onda de `E`), `PTE22` (intervalo de tempo de processamento em nível alto) e `PTE23` (forma de onda do sinal alternado pelas interrupções de `SysTick`) do microcontrolador, e o seu terra no pino 5 de H5. A pinagem do *shield* FEEC871 é mostrado em [2] e os pinos no *kit* LED RGB, da direita (LED) para esquerda (pinos), correspondem aos pinos 0 a 5 de H5.
   1.a. **Funcionalidade do projeto**: Execute o programa no modo `Run`, e faça uma breve descrição do que você observou ao longo da execução do programa. Anote a sequência das cores observadas no LED RGB, as mensagens mostradas nas duas linhas do LCD e a frequência em que as mensagens mostradas no LCD são alteradas.
   1.b. **Periférico LCD**: Os sinais de controle `RS`, `E` e o processamento de um *byte* só ficam ativos quando ocorrem transferências dos *bytes* do microcontrolador para LCD na ordem de centenas de microsegundos. Em relação ao intervalo de tempo em que a mensagem é mostrada no LCD, uma atualização do LCD ocupa uma fração mínima de tempo. Se visualizarmos esses sinais em uma escala de tempo, veremos padrões que se assemelham a palitos espaçados na escala de segundos. Para examinar mais de perto os detalhes nesses sinais, podemos reduzir a escala de tempo para microsegundos, concentrando-nos em um desses palitos até que

possamos observar formas de onda semelhantes às mostradas na Figura 9. O marcador `0` na figura está posicionado na borda de descida de `E`, que serve de referência para o tempo de *setup* e *hold* da entrada de um *byte* no LCD. A largura do pulso vermelho no Canal 2 indicado o tempo de processamento de um *byte* transferido para o LCD.

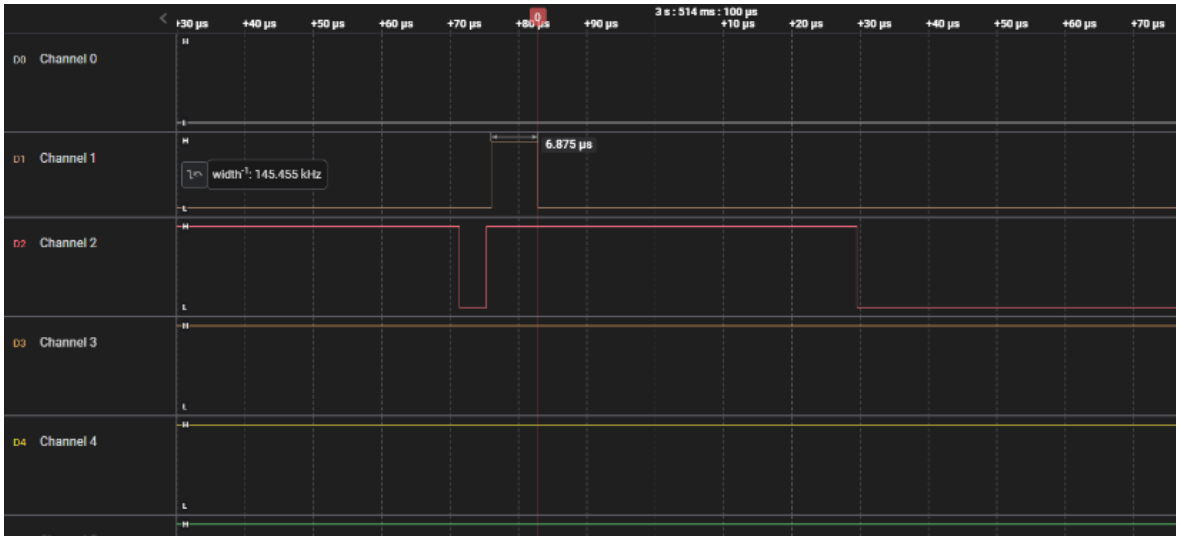


Figura 9: Formas de onda RS (Canal 0), `E` (Canal 1), tempo de processamento de um *byte* no LCD (Canal 2) e periodicidade de `SysTick` (Canal 3).

a) **Diagrama de tempo:** Registre a quantidade de pulsos contidos dentro de cada palito. Está condizente com o esperado? Registre os tempos de *setup*, *hold* e de processamento medidos no analisador lógico. Descreva sucintamente como você leu esses tempos (pode ser através de um *printscreen*). Compare-os com os valores especificados pelo fabricante (Seção 8/página 10 e Seção 9/página 12 em [5]).

b) **Programação da saída dos sinais de controle e dados do LCD:** Para analisar as relações temporais dos sinais, foram adicionadas as instruções de espelhamento dos sinais dos pinos `PTC` nos pinos `PTE` nas funções `GPIO_escreveStringLCD`, `GPIO_setRSLCD` e `GPIO_escreveByteLCD` de `GPIO_latch_lcd.c`, e implementadas novas funções `GPIO_escreveStringLCDH5Pins`, `GPIO_setRSLCDH5Pin4` e `GPIO_escreveByteLCDH5Pins`. Essas 3 funções foram usadas na implementação de `rot5_aula`. Identifique nelas os blocos de instruções responsáveis pelas formas ondas observadas.

c) **Impacto das relações temporais incorretas dos sinais:** O que acontecerá com as mensagens mostradas no LCD, se aumentarmos (dobrarmos) ou diminuirmos (em 0) os valores passados para a função `espera_2us` na rotina `GPIO_escreveByteLCDH5Pins`? Quando o visor do LCD passa a falhar? Justifique com base nos tempos medidos nas novas formas de onda.

d) **Inicialização do LCD:** Para enviar os comandos de inicialização para LCD, é necessário que a comunicação entre os pinos do processador e os pinos do LCD esteja habilitada. Qual função contém o envio da sequência de instruções de inicialização recomendada pelo fabricante? Quando ela é chamada em relação à função `GPIO_ativaConLCD()`, que habilita os pinos `PTC0-PTC9`?

1.c. **Módulo SysTick**: Aumente a escala de tempo do analisador para segundos.

a) **Período do SysTick:** Registre a largura dos pulsos do sinal no Canal 3 medida com o analisador. Identifique na função `SysTick_Handler` em `ISR.c` a maneira como foi gerado um pulso no sinal do Canal 3. Descreva-a sucintamente. Qual é a largura esperada com base nas configurações realizadas na função `SysTick_init` em `SysTick.c`? Compare-a com a largura medida em (a).

b) **Configuração do período do SysTick:** O que acontecerá com a largura dos pulsos no Canal 3 e a variação da troca das cores do LED RGB, se diminuirmos ou aumentarmos o valor do argumento `periodo` em `SysTick_RVR_RELOAD`? Qual é a relação entre o tempo medido e o valor do `periodo` configurado? Justifique com base no seu conhecimento sobre o mecanismo de temporizadores.

c) **Fontes de interrupção:** Vimos no roteiro 4 [16] que bordas e níveis de um sinal digital de propósito geral podem ser fontes de solicitação de interrupção dos módulos `PORTx/GPIOx`. Quais eventos podem causar solicitações de interrupção em `SysTick`? São eventos síncronos ou assíncronos?

d) ***Postscaler*:** A função `ISR_setaMaxPostScaler` configura, por *software*, o *postscaler* de um contador. O que acontecerá com o tempo de espera em que os LEDs R, G e B ficam acesos, se você diminuir e aumentar o valor do seu argumento `valor`? Afetará o período do `SysTick`? Justifique com base no seu conhecimento sobre o mecanismo de temporizadores. Configure o período T de `SysTick` com o "máximo divisor comum" de 5.25s (vermelho), 1.25s (azul) e 4.25s (verde). Certifique a largura dos pulsos do sinal no Canal 3 medida com o analisador.

e) **Reinicialização do contador do SysTick:** Pela Eq. (2), o período de um temporizador é o tempo necessário para contar de 0 até REF. Quando a contagem do temporizador começa em um valor maior do que zero, isso significa que o temporizador já está em andamento quando a contagem é iniciada. Nesse caso, o tempo decorrido até que ocorra uma interrupção será menor do que o período completo do temporizador. Isso ocorre porque o temporizador já avançou parte do caminho até o REF quando a contagem é iniciada. Qual é a função utilizada em `main` para zerar `SYS_CVR`? Como isso impacta no controle dos espaçamentos das mudanças das cores no tempo?

f) **Habilitação de eventos de interrupção do SysTick:** Quando e onde a interrupção do `SysTick` é ativada e desativada no programa?

1.d. **Processamento do LCD condicionado a uma interrupção**: O tratamento das respostas aos eventos de interrupção do `SysTick` é desmembrada em duas partes, uma em `SysTick_Handler` (`ISR.c`) onde são tratados os dados críticos no tempo e a outra em `main` (`main.c`) onde são processados os dados que não são críticos no tempo. Qual variável é usada para assegurar o correto sequenciamento de execução dessas duas partes? Como essa variável é declarada?

1.e. ***Strings* em C**: Em `main.c` são declarados 8 arranjos do tipo `char` processados pelas funções `GPIO_escreveStringLCDH5Pins`, `GPIO_escreveStringLCD` e `GPIO_escreveBitmapLCD` em `GPIO_latch_lcd.c`.

a) **Terminadores em *strings*:** Quais arranjos são *strings*? Como é detectado o último *byte* em arranjos que não são *strings* em `GPIO_escreveBitmapLCD`? Se passarmos esses arranjos que não são *strings* como o segundo argumento de `GPIO_escreveStringLCD`, o LCD poderá mostrar "lixos" no visor? Justifique.

b) **Processamento de *strings*:** Conforme o bloco de instruções do caso `VERDE_AMARELO` na função `main` em `main.c`, deve-se mostrar 3 glifos de `bell` na segunda linha do visor. Por quê não foram mostrados esses glifos junto com a cor amarela? Dê uma solução para que eles sejam mostrados.

c) **Efeito de espaços brancos de uma *string* no visor:** As *strings* "`VERDE`", "`VERMELHO`", "`AMARELO`" e "`AZUL`" em `main` foram declaradas com a mesma quantidade de elementos, utilizando o tamanho da maior delas como referência. O que acontecerá se removermos os "espaços brancos" na declaração dessas variáveis, alocando apenas o tamanho necessário para cada uma, nas mensagens mostradas no LCD? Com base no que você observou, qual é o efeito dos "espaços brancos" nas mensagens renderizadas no visor?

d) **Conversão explícita de char para outro tipo de dados compatível:** Os tipos de dados esperados pelas funções de `GPIO_latch_lcd.c` são `uint8_t` (inteiro sem sinal de 8

bits) ou `uint8_t*` (ponteiro de inteiro sem sinal de 8 *bits*), enquanto os tipos de dados das *strings* declaradas em `main` são do tipo `char`. Qual operador foi aplicado no segundo argumento de `GPIO_escreveStingLCDH5Pins` para compatibilizar os tipos de dados?

1.f. **Enum em C**: O caso **default** de um comando **switch** é reservado para tratar todos os casos não contemplados do domínio. Está correta a seguinte afirmação?

*No comando **switch** em `SysTick_Handler (ISR.c)` e `main (main.c)` são redundantes os blocos de instruções, podendo usar um ou outro:*

```
case VERMELHO_VERDE:
case VERDE_AMARELO:
case AMARELO_VERMELHO:
        break;
```

e

```
default:
        break;
```

Justifique.

1.g. **MACROS:** Compare as funções implementadas nos arquivos `GPIO_latch_lcd.c`, `GPIO_ledRGB.c` e `ISR.c` com a função `GPIO_initSwitchNMI (uint8_t IRQC)` implementada no arquivo `GPIO_switches.c`. Em quais funções, a identificação dos registradores usados é mais fácil? Explique a convenção aplicada para reconhecer o campo, o nome do registrador e o módulo ao qual o registrador pertence pelos seus nomes.

2. **Praticar as práticas:** Desenvolva o projeto `pomadoro` com os estados dos LEDs R, G e B e do LCD controlados pelos botões `NMI` e `PTA5` e pelo temporizador `SysTick`. Recomenda-se os seguintes passos que procuram reusar os códigos do projeto `rot5_aula` por ter um fluxo de controle similar ao do projeto `pomadoro`.

2.a. Faça uma análise comparativa entre o projeto `rot5_aula` e `pomadoro`, destacando as semelhanças (reuso) e as diferenças (alteração ou nova implementação). Certifique se estão dentre as suas diferenças: (1) dois glifos adicionais, 'Ç' e 'Ã', não são definidos na mamória CGROM do LCD; (2) sequência de indicações luminosas (1 laço contendo 3 estados (R+G+Y) x 1 laço contendo 4 sublaços com 2 estados e 1 estado (4(R+B) + 1(R+G))); (3) modo de operação do `SysTick` (contínuo x sob demanda do usuário); (4) tempos de espera de cada estado de regime; e (5) nomes dos estados,

2.b. Crie um novo projeto (Seção 2.1/página 4 em [29]).

2.c. Sobreescreva o arquivo `main.c` do projeto `rot5_aula` sobre `main.c` do novo projeto e faça os **testes funcionais** para certificar o porte (Seção 2.2.3/página 14 em [29]).

2.d. Substitua as funções com instruções de espelhamento nos pinos `PTE` em `main.c` pelas funções comentadas com "//" e faça os **testes funcionais** para certificar a substituição. Em seguida, remova a função `GPIO_initH5Pins`, que configura os pinos de H5 usados nos testes, de `main.c` e também de `GPIO_latch_lcd.c`.

2.e. Glifos adicionais:

a) Substitua a declaração dos glifos `bell` e `heart` pelos 2 novos glifos, 'Ã' e 'Ç', e faça **testes de unidade** da definição dos glifos, reexecutando o programa. Substitua os nomes dos arranjos por nomes mais apropriados.

b) Para evitar o conflito do endereço 0x00 ('Ã') com o terminador, substitua o endereço por 0x02.

c) Faça **testes de unidade** dos glifos, substituindo, respectivamente, a definição dos arranjos `bells` e `hearts` por

```
char bells[] = {'A',0x01,0x02,'0',' ',' ',0x00};
```

e

```
char hearts[] = {'A','T','E','N',0x01,0x02,'O',0x00};
```

2.f. Sequência de indicações luminosas:

a) Substitua na definição do tipo de dados `tipo_estado` em `ISR.h` os nomes dos estados para `INATIVO`, `ACAO`, `POMADORO_PAUSA`, `PAUSA_CURTA`, `PAUSA_LONGA`, `PCURTA_POMADORO`, `PLONGA_POMADORO` e `TRANS_INATIVO`.

b) Substitua o bloco de inicialização em `main.c`

```
GPIO_escreveStringLCDH5Pins (0x00, (uint8_t *)azul);
ISR_escreveEstado (AZUL);
GPIO_ledRGB(OFF, OFF, ON);
```

por

```
GPIO_escreveStringLCD (0x00, (uint8_t *)"INATIVO ");
GPIO_escreveStringLCD (0x40, (uint8_t *)"INICIE: NMI");
ISR_escreveEstado (INATIVO);
GPIO_ledRGB(OFF, OFF, OFF);
```

Crie uma variável para contar a quantidade de sublaços contendo 2 estados, chamada `pomadoro`. Inicialize `pomadoro` em 0.

c) Atualize os estados (em regime) tratados no comando `switch` em `SysTick_Handler` (`ISR.c`), bem como o direcionamento dos estados de transição em cada caso e os nomes dos estados. Para o estado de regime `ACAO`, direcione para o estado de transição `POMADORO_PAUSA`, enquanto nos estados `PAUSA_CURTA` e `PAUSA_LONGA`, vá para os estados de transição `PCURTA_POMADORO` e `PLONGA_POMADORO`, respectivamente.

d) Atualize os estados (de transição) tratados, incluindo os nomes dos estados, no comando `switch` em `main` (`main.c`). No bloco de tratamento do estado de transição `POMADORO_PAUSA`, adicione o teste do valor de `pomadoro`. Se `pomadoro < 4`, incrementa `pomadoro` e vá para o estado em regime `PAUSA_CURTA`, senão resete `pomadoro` com `pomadoro = 0` e vá para `PAUSA_LONGA`.

e) Gere o executável e faça **testes de unidade** da sequência de cores produzida.

2.g. Modo de operação sob controle do usuário via botoeira:

a) Insira a chamada de `GPIO_initSwitchNMI` em `main` (não se esqueça de incluir `GPIO_switches.h` em `main.c`) para que o botão NMI seja sensível à borda de descida e que tenha máxima prioridade de atendimento.

b) Implemente uma nova função `GPIO_initSwitchNMIPTA5` em `GPIO_switches.c`. Não se esqueça de inserior o protótipo correspondente em `GPIO_switches.h`. Substitua a chamada de `GPIO_initSwitchNMI` por `GPIO_initSwitchNMIPTA5` na função `main`.

c) Inclua em `ISR.c` a definição de `PORTA_IRQHandler` com a verificação de *flag* de interrupção dos pinos 4 e 5 de `PORTA`. Diferentemente do `SysTick`, é necessário aplicar `w1c` para baixar a respectiva *flag* de interrupção levantada:

```
void PORTA_IRQHandler () {
 if (PORTA_PCR4 & PORT_PCR_ISF_MASK) {
    PORTA_PCR4 |= PORT_PCR_ISF_MASK; //limpa flag
 } else if (PORTA_PCR5 & PORT_PCR_ISF_MASK) {
    PORTA_PCR5 |= PORT_PCR_ISF_MASK; //limpa flag
 }
}
```

Faça **testes de unidade** de configuração de interrupção dos botões `NMI` e `PTA5`.

d) Como só precisa ativar a contagem dos tempos quando o botão `NMI` é acionado, desloque `SysTick_ativaInterrupt` da função `main` para `PORTA_IRQHandler` (ativa a interrupção quando *flag* de interrupção do pino 4 é setado) e insira

`SysTick_desativaInterrupt` em `PORTA_IRQHandler` quando o botão `PTA5` é pressionado (desativa a interrupção quando o usuário conclui a tarefa).
Faça **testes de unidade** de controle de temporizador pomadoro pelo usuário, acionando aleatoriamente `NMI` e `PTA5`.

2.h. Tempos de espera de cada cor:
a) Configure o período T de `SysTick` com o valor definido em 1.c.d.
b) Determine o valor de *postscaler* para cada cor e configure valores apropriados para `ISR_setaMaxPostScaler`.
c) Faça **testes de unidade** dos tempos de espera de cada cor.

2.i. Ajuste as mensagens no LCD, conforme a especificação, usando os *bitmaps* definidos em 2.e.

2.j. Habilite *Print Size* para uma simples análise do tamanho de memória ocupado. Gere um executável e faça **teste funcionais** do projeto acionando o par de botões `NMI` e `PTA5` aleatoriamente para ver se a resposta está condizente com a especificação.

2.k. Revise a documentção das funções nos arquivos-cabeçalho. Gere uma documentação do projeto com Doxygen [15].

RELATÓRIO

O relatório deve ser devidamente identificado, contendo a identificação do instituto e da disciplina, o experimento realizado, o nome e RA do aluno. Para este experimento, registre os testes de unidade realizados no item 2, destacando os testes malsucedidos e as suas ações corretivas num arquivo em pdf. Exporte o projeto `pomadoro` **devidamente documentado** num arquivo comprimido no IDE CodeWarrior. Suba os dois arquivos no sistema *Moodle*. **Não se esqueça de limpar o projeto (*Clean* …) e apagar as pastas html e latex geradas pelo Doxygen antes da exportação**.

REFERÊNCIAS

[1] KL25 Sub-Family Reference Manual
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KL25P80M48SF0RM.pdf
[2] Esquemático do *shield* FEEC871
https://www.dca.fee.unicamp.br/cursos/EA871/references/complementos_ea871/Esquematico_EA871-Rev3.pdf
[3] ARMv6-M Architecture Reference Manual.
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/ARMv6-M.pdf
[4] *Datasheet* 74573
https://www.dca.fee.unicamp.br/cursos/EA871/references/datasheet/74HC_HCT573.pdf
[5] *Datasheet* do *display* LCD
https://www.dca.fee.unicamp.br/cursos/EA871/references/datasheet/AC162A.pdf
[6] Rickey's World. LCD Interfacing Tutorial: Basics of Alphanumeric LCD.
https://www.8051projects.net/lcd-interfacing/basics.php
[7] Rickey's World. LCD Interfacing Tutorial: CGRAM Creating custom character.
https://www.8051projects.net/lcd-interfacing/lcd-custom-character.php
[8] LCD Custom Character Generator
https://maxpromer.github.io/LCD-Character-Creator/
[9] Roteiro 3
http://www.dca.fee.unicamp.br/cursos/EA871/1s2024/roteiros/roteiro3.pdf
[10] Embedded Wednesdays: Characters
https://embedded.fm/blog/2016/4/25/ew-characters
[11] Paul-Henning Kamp. The Most Expensive One-Byte Mistake.
https://queue.acm.org/detail.cfm?id=2010365
[12] Array of srtings in C
https://overiq.com/c-programming-101/array-of-strings-in-c/
[13] Analisador Lógico Saleae

https://www.saleae.com/pt/
[14] Roteiro 2
http://www.dca.fee.unicamp.br/cursos/EA871/1s2024/roteiros/roteiro2.pdf
[15] Documentação com Doxygen
https://www.doxygen.nl/manual/docblocks.html
[16] Roteiro 4
http://www.dca.fee.unicamp.br/cursos/EA871/1s2024/roteiros/roteiro4.pdf
[17] Sergio Prado. Use Macros (ou não)
https://sergioprado.org/simplifique-use-macros-ou-nao/
[18] Dan Saks. Enumerations Q & A
https://www.embedded.com/enumerations-q-a/
[19] Magnum. Introdução às Bibliotecas em C
http://mindbending.org/pt/introducao-bibliotecas-em-c
[20] Thiago Pereira do Prado. Usando FreeRTOS para aplicações com a FRDMKL25Z
https://www.embarcados.com.br/freertos-nxp-frdm-kl25z/
[21] Chandrasekaran, Siddharth. Timer/Counter Module – A Controller Independent Guide.
https://embedjournal.com/timer-modules-guide/
[22] rot5_aula.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2024/codes/rot5_aula.zip
[23] Roteiro 3
http://www.dca.fee.unicamp.br/cursos/EA871/1s2024/roteiros/roteiro3.pdf
[24] Khaled Magdy. How to Write ISR handlers?
https://deepbluembedded.com/how-to-write-isr-handlers/
[25] Jack Ganssle. A pox on globals
https://www.embedded.com/a-pox-on-globals/
[26] Geeksforgeeks. Switch vs. if eles
https://www.geeksforgeeks.org/switch-vs-else/
[27] Bruno Gall De Blasi. O que é a técnica Pomodoro Timer?
https://tecnoblog.net/responde/o-que-e-a-tecnica-pomodoro-timer-25-minutos/
[28] Diagrams.net (editor de diagramas *online*)
https://www.diagrams.net/
[29] Wu, S.T. Ambiente de Desenvolvimento – *Software*
https://www.dca.fee.unicamp.br/cursos/EA871/references/apostila_C/AmbienteDesenvolvimentoSoftware_V1.pdf

Revisado em Fevereiro de 2024
Revisado em Janeiro de 2023
Revisado em Fevereiro e Agosto de 2022
Revisado em Maio e Setembro de 2021
Outubro de 2020