Desenvolvendo software para a placa STM32L476 Discovery

Hans Jorg Schneebeli

24 de abril de 2017
Vitória-ES

Índice

Desenvolvendo software para a placa STM32L476 Discovery

Introdução

A placa SMT32L476 Discovery [1] é uma placa de baixo custo (US$) da linha de placas de desenvolvimento ofertada pela ST para marketing de seus microcontroladores STM32.

A placa STM32L476 possui um microcontrolador STM32L476VG [2], que tem as seguintes caraterísticas:

- Processador com núcleo ARM Cortex M4F com memóriaRAM de 128 Kbytes, memória Flash de 1 Mbyte, encapsulamento de LQFP100 e frequencia de relógio de até 80 MHz.
- Sistema de programação e depuração ST-Link/V2-1 com interface USB (com conector miniUSB), baseado num microcontrolador STM32F103
- Display LCD com 6 caracteres de 14 segmentos
- Dois LEDs controlados por software (Vermelho e Verde).
- Joystick (5 botões)
- Interface USB OTG com conector microUSB
- Saída de audio estereo
- Microfone
- Acelerometro
- Magnetometro
- Memória flash 128 Mbit com interface SPI
- Medidor de consumo de energia do microcontrolador (MPX)
- Giroscópio
- Alimentação por bateria ou interface USB (
- Conectores para expansão (similar aos shields do Arduino).

Deve ser observado que a licença de uso da placa impede seu uso em produtos comerciais[1]

Ambiente de desenvolvimento

Para desenvolver software para um microcontrolador são necessários:

- Compilador
- Ferramenta de Programação da Flash
- Interface de Depuração
- Bibliotecas para acesso ao hardware

O fabricante indica diversos compiladores [3], mas todos são pagos, embora alguns possam ter versões grátis que geram código de tamanho limitado. Estes compiladores tem uma interface integrada de desenvolvimento, que simplifica bastante o processo de desenvolvimento. Mas insere algumas camadas de software entre o código desenvolvido e o acesso às ferramentas.

Existe uma versão do compilador para o Cortex-M baseado no gcc e patrocinada pela própria ARM, disponibilizada em www.launchpad.net/gcc-arm-embedded com versões para Windows, Linux e MacOS. No entanto, esta versão somente suporta acesso por linhas de comando. Mas o uso de

1 Ver licença em http://www.st.com/st-web-ui/static/active/en/resource/legal/legal_agreement/license_agreement/EvaluationProductLicenseAgreement.pdf

interfaces por linhas de comando tem vantagens didáticas, pois se sabe o que está acontecendo por debaixo dos panos. E é também a forma preferida de muitos desenvolvedores experientes.

O ambiente descrito é para uma máquina Linux, mas com poucas alterações, é possível se usar maquinas Windows.

Compilador. Instalar o obtido em http://www.launchpad.net/gcc-arm-embedded.
Ferramenta de Programação da Flash. Existem duas alternativas para Linux: st-flash [2] e OpenOCD[3]. Para Windows, o fabricante oferece uma versão grátis[4].

Observação importante. No momento, existem atualizações disponiveis para o software do microcontrolador da interface de desenvolvimento[5]

Quanto a bibliotecas, para facilitar a migração de software entre microcontroladores Cortex-M, a ARM desenvolveu um padrão chamado CMSIS (Cortex Microcontroller Software Interface Standard) [4]. Este padrão especifica formas de acesso ao hardware e a dispositivos padrão como controlador de interrupção (NVIC), Contador (SysTick) e Configuração. Isto faz com que tarefas complexas sejam protáveis, mas acender ou apagar um LED, não, pois não existe padrão para GPIO (General Purpose Input Output).

Para a programação da família STM32L4 o fabricante disponibiliza um manual de referencia [5], que explica os diversos periféricos que podem ser usado para uma versão do microcontrolador. Explica também como usá-los e os registradores para acesso a eles.

O fabricante também disponibiliza uma biblioteca para acesso ao hardware, chamada STM32L4Cube[6]. Faz parte desta biblioteca utilitários que geram códigos de inicialização. Mas ela é complexa e com muitos erros. Além disso, a licença somente permite o seu uso em microcontroladores STM32. Somente serão usadas os arquivos CMSIS.

Projeto modelo

Um projeto para o microcontrolador, que segue o padrão CMSIS, deve ter:

- arquivos-fonte com sufixo .c (minimamente, o arquivo main.c)
- arquivos-cabeçalho com sufixo .h, que contem informações sobre os arquivos-fonte.
- Arquivo de inicialização start_stm32l476.c
- Arquivo com rotinas padrão system_stm32l476.c
- Arquivo com instruções para ligação stm32l476.ld que instrui o loader (linker) como gerar o código objeto.
- Arquivo Makefile, que permite automatizar as tarefas de compilação, gravação e inicio de uma sessão de depuração.

Os arquivos startup_stm32l476.c, system_stm32l476.c, system_stm32l476.h e system_stm32l476.ld geralmente sofrem pouca ou nenhuma modificação. O arquivo Makefile necessitará de ser modificado para inclusão de bibliotecas adicionais.

Arquivo de inicialização startup_stm32l476.c

O arquivo startup_stm32l476.c define o vetor de interrupções, que deve ficar localizado no memória a partir do endereço 0. Em particular, o endereço 0 o valor do apontador de pilha após o

2 Disponível em https://github.com/texane/stlink
3 Disponível em http://www.openocd.org/
4 Disponível em http://www.st.com/web/en/catalog/tools/PF260219
5 Disponível em http://www.st.com/web/en/catalog/tools/PF260217
6 Disponível em http://www.st.com/web/en/catalog/tools/PF261908.

reset e o endereço 4, o valor do contador de programa carregado após o reset. No caso, o endereço da rotina Reset_Handler.
O código de inicialização fica na rotina Reset_Handler e é executado logo após o reset. As ações de inicialização incluem:
1 – Inicializar variáveis estáticas (incluindo as globais) com os valores definidos no código. Para isto, os valores iniciais devem ficar numa memória não volátil, no caso, na memória flash. O linker é instruído para armazenar estes valores na Flash, imediatamente a seguir do final das instruções. O endereço do final das instruções é definido pelo simbolo _etext.
2 – Inicializar varáveis estáticas (incluindo as globais) que não tiveram o valor inicial definido com 0.
3 – Chamar a rotina SystemInit, geralmente definida pela aplicação. Se não for definida será usada uma default, que não faz nada.
4 – Chamar a rotina _main, que geralmente inicializa o necessário para uma biblioteca. Se não for definida, será usada uma default, que não faz nada.
5 – Chama a rotina main, que é justamente a definida no programa principal. Ela não deve retornar, mas caso isto aconteça, há um ciclo infinito que trava o processador.
Os elementos do vetor de interrupção apontam para uma rotina default. Para se apontar para uma determinada rotina, basta se definir uma rotina com o nome adequado. Por exemplo, para uma rotina processar interrupções da UART4, ela deve ser chamada USART4_IRQHandler.

O arquivo de sistema system_stm32l476.c e o respectivo cabeçalho

O arquivo de sistema define a implementação de duas rotinas exigidas pelo CMSIS: SystemInit e SystemCoreClockUpdate. No SystemInit é definido o funcionamento default do relógio, é ativada a FPU, são disabilitadas todas as interrupções e chamada a rotina SystemCoreClock para colocar na variável global SystemCoreClock o valor da frequencia da CPU.

No projeto modelo este arquivo define uma rotina chamada SystemCoreClockSet, que não é padrão CMSIS. Usando-se os parametros definidos em system_stm32l476.h, pode-se alterar a velocidadee do relógio da CPU.

O arquivo script para o linker stm32l476.ld

O arquivo especifica como o arquivo executável será montado, determinando qual parte ficará armazenada em flash e onde se armazenarão as informações.

A primeira informação a ser especificada é o mapa de memória

- Memória Flash como sendo uma memória capaz de armazenar código e suportando apenas operações de leitura, com 1 Mbyte a partir do endereço 0
- Memória RAM com sendo uma memória capaz de armazenar código, suportando tanto leitura como escrita, com tamanho 98 Kbyte a partir do endereço 0x2000000
- Memória RAM com deteção e correção de erro com 32 Kbytes a partir do endereço 0x1000000.

O código objeto gerado a partir da compilação de uma unidade é dividido em seções;
Text também conhecida como Code é onde se armazenarão as instruções
Data é a seção onde se armazenam as variáveis inicializadas
Rodata é a seção onde ser armazenam constantes
BSS é a seção que contém os dados não inicializados
Stack é onde são armazenadas as variáveis automáticas

Tomando como exemplo os trechos de um programa.

```
...
const char *s = “Equação”;              // s em DATA, string em RODATA
int contador;                           // BSS

int outrocontador = 0;                  // DATA
main(void) {
int a,b,c;                              // STACK
       contador = 0;
       …
       printf(“a = %g b = %g c = %g\n”,a,b,c); // STRING em RODATA
       ...
}
```

Assim o que está na seção TEXT, RODATA vão para a flash. O que está na seção BSS e STACK ficam na RAM. O que está na seção data (valores iniciais de variáveis) deve ficar armazenado na flash mas as variáveis ficam alocadas na RAM. Assim os valores devem ser copiados da memória FLASH para a RAM antes do inicio da execução do programa. Isto é feito na rotina Reset_Handler. Também a especificação da linguagem C exige que todas as variáveis globais não inicializadas, que estão na seção BSS devem ser inicializadas com 0. Isto também é feito na rotina Reset_Handler.

O arquivo Makefile

Permite principalmente que o projeto seja compilado, sem que se façam compilações desnecessárias. Todos os arquivos fonte são analisados. Caso eles sejam mais recentes que o correpodnente arquivo objeto, ele é compilado e gerando um arquivo objeto atualziado. Depois é executada a etapa de ligação gerando o arquivo executável.

Existem outras maneiras de usar o make, ativando-se objetivos diferentes.

make edit:	abre todos os arquivo texto do objeto para edição
make flash:	grava o arquivo imagem no memória do microcontroladores
make doxygen:	gera documentação
make clean:	apaga todos os arquivos gerados a partir do makefile
make size:	lista informação sobre as seções do arquivo executavel
make nm:	gera a lista de simbolos
make list:	gera o arquivo assembly

Manuseio dos registradores

Registradores de controle de hardware tem endereço fixo. Em C, para acessar estes registradores, a técnicas mais usada é usar ponteiros, que são basicamente variáveis que contém um endereço (e estão associados a um tipo de dados).

Assim para se acessar o registrador de 32 bits no endereço 0x12345678, usa-se a sequencia abaixo;.

```
uint32_t *p; // apontador
       ...
       p = (uint32_t *) 0x12345678;          // conversão inteiro para ponteiros
       ...
       *p = 0;                               // escreve zero no registrador
       uint32_t w = *p;                      // le e coloca em w o valor do registrador.
```

Isto pode ser condensado em:

```
       *( (uint32_t *) 0x12345678) = 0;      // escreve zero no registrador
```

```
	uint32_t w = *((uint32_t*) 0x12345678); // le e coloca em w o valor do registrador
```

Para se aumentar a legibilidade, pode-se usar o preprocessador, que faz uma substituição textual de um simbolo na cadeia de caracteres desejada.

```
#define HWREG *( (uint32_t *) 0x12345678)
	...
	HWREG = 0;                          // escreve zero no registrador
	uint32_t w = HWREG;                 // le e coloca em w o valor do registrador
```

Geralmente, as definições de acesso a registros ficam em um arquivo cabeçalho, muitas vezes fornecido pelo próprio fabricante.

Como muitos registradores são relacionados a um determinado módulo, no CMSIS, usa-se uma técnica baseada em um ponteiro para o endereço base. Como exemplo, um módulo é controlado por quatro registradores de 32 bits, como mostrado abaixo.

Endereço base + 0 Registrador ENTRADA
Endereço base + 4 Registrador SAIDA
Endereço base + 8 Registrador CONFIGURACAO
Endereço base + 12 Registrador STATUS

Ao invés de se definir quatro registradores para cada módulo, define-se uma estrutura como abaixo.

```
typedef struct {
	uint32_t	ENTRADA;
	uint32_t	SAIDA;
	uint32_t	CONFIGURACAO;
	uint32_t	STATUS;
} Modulo_s;
```

E um apontador (para cada módulo) definido como a seguir.

```
#define Modulo1 ((Modulo_s *) EnderecoBase)
```

Assim o acesso a um registrador pode ser feito como

```
	Modulo1-> SAIDA = 0;
	uint32_t w = Modulo1-> ENTRADA;
```

Manipulação de bits

Para se setar um bit (fazê-lo igual a 1) de um registrador, deve-se fazer o ou lógico bit a bit , ou seja usar o operador & (não confundir com o ou lógico, operador &&) com uma mascara onde o único bit 1 está na posição que se deseja modificar.

Para se setar o bit 4 do registrador X, deve-se fazer

```
	X = X | 0x10;
```

ou ainda

```
	X |= 0x10;
```

Para se aumentar a legibilidade e se evitar constantes cabalísticas pode-se definir símbolos como abaixo

```
// O bit menos significativo e o zero
#define BIT0 0x01
#define BIT1 0x02
#define BIT2 0x04
#define BIT3 0x08
```

```
#define BIT4 0x10
#define BIT5 x020
#define BIT6 0x40
#define BIT7 0x80
```

Ou ainda, se o compilador otimizar expressões constantes

```
#define BIT(N) (1U<<(N))
```

Neste caso pode-se usar BIT(0), BIT(1), e assim por diante, ou então definir os mesmos símbolos acima como mostrado abaixo.

```
#define BIT0 BIT(0)
#define BIT1 BIT(1)
#define BIT2 BIT(2)
#define BIT3 BIT(3)
#define BIT4 BIT(4)
#define BIT5 BIT(5)
#define BIT6 BIT(6)
#define BIT7 BIT(7)
```

Para se zerar um bit, deve-se usar

```
X = X&(~BIT4);
```

ou então

```
X &= ~(BIT4);
```

Pode-se também definir macros como abaixo, que podem ser muito mais legíveis.

```
#define BITCLEARN(X,N)    (X)&=~BIT(N)
#define BITSETN(X,N)      (X)|=BIT(N)
#define BITCLEAR(X,M)     (X)&=~(M)
#define BITSET(X,M)       (X)|=(M)
```

Campos de bits

Quando tiver que se alterar campos de bits, o principal cuidado a ser tomado é não ter dados transitórios no registrador. Para isto, deve haver uma única atribuição como mostrado abaixo e o campo deve ser zerado antes da modificação.

```
X = (X&~M)|V;
```

que pode ser implementado através da macro abaixo

```
#define BITFIELDSET(VAR,MASK,VAL)       VAR = ((VAR)&~(MASK))|(VAL)
```

Outra macro interessante é uma que usa os indices dos bits para definir o campo a ser alterado. Para isto é usada a macro BITMASK que gera um inteiro com todos os bits zerados exceto os entre M e N, com M>N.

```
#define BITMASK(M,N)                    ((BIT((M)-(N)+1)-1)<<(N))
#define BITFIELDSETMN(VAR,M,N,V)        (VAR) = (VAR)&~BITMASK((M),(N))|((V)<<(N))
```

Exemplo 1 – Blinker muito simples

A placa tem dois LEDs que podem ser controlados por software. O LED vermelho está mapeado no bit 2 da Porta B (na documentação do microcontrolador, PB2). O LED verde está mapeado no bit 8 da Porta E (PE8).

LED	Porta	GPIO	Bit	Label	Pino	Symbol	Value
Vermelho	PB	GPIOB	2	PB2	37	LED_RED	2
Verde	PE	GPIOE	8	PE8	39	LED_GREEN	8

Para facilitar o uso e aumentar a legibilidade do código foram definidos os simbolos como abaixo.

```
#define LED_RED           (2)
#define LED_GREEN   (8)
```

Ambos pinos são controlados por periféricos do tipo *General Purpose Input Output* (GPIO). Deve ser observado que cada fabricante tem arquiteturas diferentes para este tipo de periférico e há pouca compatibilidade entre eles.

O STM32L476 inicializa com quase todos os periféricos desativados, inclusive os GPIO. Portanto, o primeiro passo, é ativar o relógio para o periférico em questão. No caso, os GPIO estão conectados ao barramento AHB e o registrador AHB2EN tem bits que habilitam diversos periféricos (Ver Item 6.4.17 do Manual de Referencia [5]).

Para ativá-los devem ser usadas as instruções abaixo, que usam constantes pré-definidas nos arquivos cabeçalho.

```
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOEEN;   // Enable GPIO Port E (bit 4)
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;   // Enable GPIO Port B (bit 1)
```

Recomenda-se esperar o efeito da ativação. Para tal, chama-se a rotina de sincronismo _DSB.

Uma vez, ativado o módulo GPIO, deve-se configurá-lo para a uso desejado (Ver Item 7.3 do Manual de Referencia [5]). A configuração é feita acessando diversos registradores do módulo, atribuindo-se valores a campos de 2 bits (um para cada pino):

MODER Modo de operação: 0: Entrada, 1: Saída, 2: Alternativo, 3: Analógico

OTYPE Tipo de saída: 0: Push-pull, 1: Open drain

OSPEED Frequência de relógio do módulo (Ver Tabela 60 do Datasheet [4])

PUPD Ativa-se Pull-up ou Pull Down ou nenhum dos dois

É possível se configurar cada pino isoladamente. Para a configuração do LED vermelho é necessário armazenar no campo de bits 5-4 (relativos ao pino 2) do registrador MODER o valor 1, que equivale a configurá-lo como saída. Para isto, usou-se a constante LED_RED (igual a 2) que especifica qual o pino. A posição correspondente no registrador é LED_RED*2+1 a LED_RED*2 (bits 5 a 4). O 3 (igual a 11 em binário) é usado para definir uma mascara que permite se apagar todos os bits do campo.

```
// Set to output
GPIOB→MODER = (GPIOB→MODER&~(3<<(LED_RED*2)))|(1<<(LED_RED*2)));
```

Esta instrução poderia ser escrita, de forma menos legível, como

```
GPIOB→MODER = (GPIOB→MODER&0xFFFFFFCF|0x00000010;
```

Não foi modificado o registrador OTYPE pois o valor default é 0, configuração Push-Pull, é adequada.

Para a configuração da velocidade da porta no registrador OTYPER, procede-se de maneira similar ao MODER. Devem ser observadas as restrições de velocidade em função da tensão de alimentação e da frequência de relógio na Tabela 60 do *datasheet* [4]. Isto não é problema pois o dispositivo está funcionando a 4 MHz.

```
// Set to high speed
GPIOB->OSPEEDR = ((GPIOB→OSPEEDR&~(3<<(LED_RED*2)))|(3<<(LED_RED*2)));
```

Mais uma vez, procede-se de forma similar para ativar o resistor de Pull-up.

```
// Set to pull up
GPIOB->PUPDR  = ((GPIOB→PUPDR&~(3<<(LED_RED*2)))|(1<<(LED_RED*2)));
```

Então finalmente, pode-se acender o LED ativando o pino correspondente do registrador de saída (ODR).

```
GPIOB->ODR |= (1<<LED_RED);
```

Para se configurar o LED verde, muda-se o periférico (de GPIOB para GPIOE) e qual o pino (LED_RED para LED_GREEN).

Para que os LEDs pisquem, eles devem ter o valor trocado com uma frequência menor do que 20 Hz (persistência retiniana). Para trocar o valor basta se usar o operador ^ (Ou exclusivo). Para se ajustar a frequência, usou-se um mecanismo muito primitivo de atraso. O valor inicial do contador (variável x) foi setado em 7000 empiricamente. Serão vistos mecanismos mais eficientes e precisos posteriormente.

Para se compilar, basta executar o comando

```
make build
```

Para transferir o executável para a memória flash, deve-se conectar o cabo entre o PC e a porta Mini-USB e executar o comando

```
make flash
```

Reiniciando-se a placa, pressionando o botão preto, o programa entra em ação.

Uma alternativa usando símbolos de preprocessador

Uma maneira melhor, que permite maior legibilidade e não tem custos adicionais em termos de velocidade do aplicativo e da memória demandada por ele é definindo-se símbolos. Podem ser usado símbolos definidos para o preprocessador ou constantes.

Uma macro do tipo BITFIELD(V,N) gera uma constante inteira com o bit menos significativo do valor V situado no bit N (O bit menos significativo é o zero).

Para tornar o código mais legível, foram criados simbolos para os valores de campos usados para configurar o periférico, como abaixo.

```
#define GPIO_MODE_INP           0
#define GPIO_MODE_OUT           1
#define GPIO_MODE_ALT           2
#define GPIO_MODE_ANA           3
#define GPIO_MODE_MASK          3

#define GPIO_OSPEED_LOW         0
#define GPIO_OSPEED_MED         1
#define GPIO_OSPEED_HIGH        2
#define GPIO_OSPEED_VERYHIGH    3
#define GPIO_OSPEED_MASK        3

#define GPIO_PUPD_NONE          0
#define GPIO_PUPD_UP            1
#define GPIO_PUPD_DOWN          2
#define GPIO_PUPD_RES           3
#define GPIO_PUPD_MASK          3
```

Deve ser observado que todos os valores são codificados em 2 bits. Além disso, foram definidos símbolos com o sufixo _MASK com todos os bits iguais a 1, que serão usado para zerar o campo antes de uma atribuição.

Para definir o bit correspondente no registrador de saída, ODR, é usado um campo de 1 bit. Para configurar o pino são usados campos de 2 bits nos registradores MODER, OSPEED e PUPDR. Assim, foram definidos simbolos para especificar os pinos usados como abaixo.

```
#define LED_GREEN_PE_PIN   (8)
#define LED_RED_PB_PIN     (2)
```

A mascara para ODR é definida com o uso da macro BIT.

```
#define LED_GREEN_PE            BIT(LED_GREEN_PE_PIN)
```

Para se definir as mascaras para os registradores é usada a macro BITFIELD.

```
#define LED_GREEN_PE_MODE       BITFIELD(GPIO_MODE_OUT,LED_GREEN_PE_PIN*2)
#define LED_GREEN_PE_MODE_M     BITFIELD(GPIO_MODE_MASK,LED_GREEN_PE_PIN*2)
#define LED_GREEN_PE_OSPEED     BITFIELD(GPIO_OSPEED_VERYHIGH,LED_GREEN_PE_PIN*2)
#define LED_GREEN_PE_OSPEED_M   BITFIELD(GPIO_OSPEED_MASK,LED_GREEN_PE_PIN*2)
#define LED_GREEN_PE_PUPD       BITFIELD(GPIO_PUPD_UP,LED_GREEN_PE_PIN*2)
#define LED_GREEN_PE_PUPD_M     BITFIELD(GPIO_PUPD_MASK,LED_GREEN_PE_PIN*2)
```

Foi usado nos nomes dos simbolos qual a porta para ajudar a evitar que se use este simbolo em outra porta gerando um erro difícil de descobrir.

```
    // Configurate GPIO for Green LED
    GPIOE->MODER    = (GPIOE->MODER&~LED_GREEN_PE_MODE_M)|LED_GREEN_PE_MODE;
    GPIOE->OSPEEDR  = (GPIOE->OSPEEDR&~LED_GREEN_PE_OSPEED_M)|LED_GREEN_PE_OSPEED;
    GPIOE->PUPDR    = (GPIOE->PUPDR&~LED_GREEN_PE_PUPD_M)|LED_GREEN_PE_PUPD_M;
    GPIOE->ODR     &= ~LED_GREEN_PE;   ;
```

O ciclo principal fica então como mostrado abaixo.

```
  for (;;) {
       ms_delay(500);
       GPIOB->ODR ^= LED_RED_PB;       // Use XOR to toggle output
       GPIOE->ODR ^= LED_GREEN_PE;     // Use XOR to toggle output
    }
```

Outra modificação é a modificação da frequência da CPU, que agora funciona a 48 MHz com a instrução abaixo que chama a função SystemCoreClockSet definida em system_stm32l476.c

```
SystemCoreClockSet(MSI48M_CLOCKSRC, 0, 3, 0);
```

Isto significa que o valor inicial do contador (variável x) em ms_delay teve que ser aumentado para manter a frequência aproximadamente a mesma.

Uma alternativa usando macros para manipulação de bits

Esta variante do código anterior usa as macros para manipulação de bits listadas abaixo.

```
#define BIT(N)                      (1UL<<(N))
#define BITFIELD(V,N)               ((V)<<(N))
#define BITFIELDMASK(M,N)           ((BIT((M)-(N)+1)-1)<<(N))

#define BitSet(V,M)                 (V)|=(M)
#define BitClear(V,M)               (V)&=~(M)
#define BitToggle(V,M)              (V)^=(M)

#define BitFieldSet(VAR,MASK,VAL)  (VAR) = ((VAR)&~(MASK))|(VAL)
#define BitFieldClear(VAR,MASK)    (VAR) &= ~(MASK)
```

Usando-se estas macros a configuração pode ser escrita como abaixo.

```
   BitFieldSet(GPIOE->MODER,LED_GREEN_PE_MODE_M,LED_GREEN_PE_MODE);
    BitFieldSet(GPIOE->OSPEEDR,LED_GREEN_PE_OSPEED_M,LED_GREEN_PE_OSPEED);
    BitFieldSet(GPIOE->PUPDR,LED_GREEN_PE_PUPD_M,LED_GREEN_PE_PUPD_M);
    BitClear(GPIOE→ODR,LED_GREEN_PE);
```

O ciclo principal fica então como abaixo.

Usando-se estas macros a configuração pode ser escrita como abaixo.

```
    for (;;) {
       ms_delay(500);
       BitToggle(GPIOB->ODR,LED_RED_PB);
       BitToggle(GPIOE->ODR,LED_GREEN_PE);
    }
```

Exemplo 2 – Blinker com abstração da GPIO

O uso do periférico GPIO pode ser feito através de um conjunto de rotinas. Deste modo, a função main terá a seguinte implementação.

```
int main(void) {

    //    SystemCoreClockSet(MSI48M_CLOCKSRC, 0, 3, 0);

    GPIO_Init(0,LED_GREEN|LED_RED);

    GPIO_Write(LED_GREEN,LED_RED);

    for (;;) {
       ms_delay(500);
       GPIO_Write(LED_RED,LED_GREEN);
       ms_delay(500);
       GPIO_Write(LED_GREEN,LED_RED);
    }
}
```

Para que isto funcione e para que no chamado das funções não seja necessário especificar qual a porta a ser usada, a definição dos pinos é feita através de uma constante de 32 bits (uint32_t), que é o tamanho nativo de inteiro dos processadores ARM. Os bits 31 a 16 especificam pinos da Porta E e de 15 a 0, pinos da Porta B, o que é feito através da macro MKWORD

```
#define LED_GREEN   MKWORD(BIT(8),0)
#define LED_RED     MKWORD(0,BIT(2))
```

Duas macros são usadas para extrair os dados das portas B e E do numero gerado acima definidas conforme mostrado abaixo.

```
#define GETPORTB(X) ((X)&0xFFFF)
#define GETPORTE(X) ((X)>>16)
```

Deve ser observada a diminuição do tamanho e a facilidade de se descobrir o que o programa deve fazer, mesmo sem ter conhecimento total da implementação.

As rotinas principais são:

GPIO_Init(input,output)	Inicializa os periféricos GPIO B e E, definindo os bits especificados por input como saída e os especificados em output como saída
GPIO_Write(zeroes,ones)	Os bits especificados por zeroes são zeros e os por ones setados.

As funções foram projetadas para uso geral, podendo configurar qualquer pino das Portas B e E para ser usado nas chamadas subsequentes (Ficou faltando a rotina GPIO_Read!!).

A rotina chave é a rotina mk2from1 que gera um numero inteiro de 32 bits com 16 campos de 2 bits gerados a partir de um numero inteiro com 16 bits, preenchendo cada campo com o valor especificado. Assim é possível se gerar os parâmetros de configuração para os registradores MODER, OSPEEDR e PUPDR.

A rotina de saída GPIO_Write tem dois parâmetros, um que especifica quais os pinos que devem ser zerados e outros, quais devem ser setados. Ambos tem 32 bits e especificam os pinos das Portas B e E.

A rotina GPIO_Write usa um ciclo de leitura, modificação e escrita usando o registrador ODR como mostrado abaixo.

```
GPIOB->ODR = (GPIOB→ODR&~(pinones|pinzeroes))|pinones;
```

Seria possível o uso do registrador BSRR mas isto complicaria a interface das rotinas.

Exemplo 3 – Criando uma HAL para as GPIOs

A implementação anterior é feita usando um único arquivo. Mas a possibilidade de reuso e a facilidade de teste é aumentada separando-se a implementação do seu uso. Isto é feito construindo-se um nível intermediário, uma camada de abstração de hardware (HAL-Hardware Abstracting Layer) que apresenta para a aplicação uma Interface de Programação para a Aplicação (API-Application Programming Interface) como mostrado abaixo.

Aplicação
Biblioteca GPIO (gpio.c,gpio.h)
Hardware

A implementação é feita no arquivo gpio.c, que mostra a rotina GPIO_Init e as rotinas de uso interno mk2from1 e GPIO_ConfigurePort, que não são acessíveis externamente, pois foram definidas como static.

Para acessar a rotina GPIO_Init é necessário se conhecer as informações sobre sua interface e seus parametros. Isto é feito no arquivo gpio.h, que ainda mostra a implementação das rotinas GPIO_Write e GPIO_Toggle. A razão destas rotinas estarem neste arquivo é a possibilidade de sua inserção no código no ponto de chamada, indicado pela palavra-chave inline e como isto deve ser feito pelo compilador, a implementação deve ser conhecida já na compilação.

Para evitar conflito de nomes as macros MKWORK, PORTB e PORTE foram renomeadas acrescentando-se o prefixo GPIO_.

Além disso, as macros de manipulação de bits foram agrupadas no arquivo bitmanip,h, tornando fácil o seu uso em outros programas.

Para se usar estas rotinas basta se incluir as instruções abaixo no inicio do arquivo main.c, logo após a inclusão dos arquivos necessários para se usar o hardware.

```
#include "stm32l476xx.h"
#include "system_stm32l476.h"

#include "bitmanip.h"
#include "gpio.h"
```

Para se criar os símbolos necessários para se acessar os LEDS, devem ser criados os símbolos abaixo.

```
/* Bit numbers for LEDS
 * LED     GPIO      Pin
 * Green   GPIOE      8
 * Red     GPIOB      2
 */

#define LED_GREEN   GPIO_MKWORD(BIT(8),0)
#define LED_RED     GPIO_MKWORD(0,BIT(2))
```

O ciclo principal tem então o formato abaixo.

```
int main(void) {

    //    SystemCoreClockSet(MSI48M_CLOCKSRC, 0, 3, 0);

    GPIO_Init(0,LED_GREEN|LED_RED);

    GPIO_Write(LED_GREEN,LED_RED);

    for (;;) {
       ms_delay(500);
       GPIO_Write(LED_RED,LED_GREEN);
       ms_delay(500);
       GPIO_Write(LED_GREEN,LED_RED);
    }
}
```

Exemplo 4 – Blink usando o temporizador

Nesta versão os LEDs são controlados em uma rotina chamada periodicamente quando uma interrupção é gerada pelo SysTick Timer, um temporizador presente em todos os processadores Cortex-M.

A configuração do SysTick é feita logo no inicio do programa, imediatamente após a configuração do relógio, usando a rotina SysTick_Config que faz parte do CMSIS. Ela deve ser refeita toda vez que o relógio for alterado.

SysTick_Config(SystemCoreClock/1000);

Neste caso, é gerada uma interrupção a cada 1 ms.

A rotina de interrupção usa um contador (variável cntgreen) para que a cada 1000 interrupções (ou seja, 1 s), seja executado o código que troca o estado do LED verde. Do mesmo modo, a cada 1200 interrupções, o estado do LED vermelho é alterado. As variáveis cntgreen e cntred devem ser estáticas para que mantenham o valor entre os chamados. Se não fosse, elas seriam reiniciadas com 0 toda vez que rotina fosse chamada.

```
void SysTick_Handler(void) {
static uint32_t cntred = 0;
static uint32_t cntgreen = 0;

    if( cntgreen == 0 ) {
        GPIO_Toggle(LED_GREEN);
        cntgreen = 999;
    } else {
        cntgreen--;
    }

    if( cntred == 0 ) {
        GPIO_Toggle(LED_RED);
        cntred = 1199;
    } else {
        cntred--;
    }
}
```

Outra maneira de escrever a rotina seria usando uma variável global (que já é estática). Neste caso a palavra-chave static tem outro significado, que é privado, ou seja, somente o código neste arquivo consegue acessar esta variável. Neste caso, ajustou-se o código para se carregar com 1000, que é o valor que se quer contar.

```
static uint32_t cntred = 0;
static uint32_t cntgreen = 0;

void SysTick_Handler(void) {

    if( cntgreen == 0 ) {
        GPIO_Toggle(LED_GREEN);
        cntgreen = 1000;
    }
    cntgreen--;

    if( cntred == 0 ) {
        GPIO_Toggle(LED_RED);
```

```
        cntred = 1200;
    }
   cntred--;
}
```

A flexibilidade que se ganha é muito grande. Para se piscar em frequencias diferentes, basta se alterar o valor inicial dos contador. Isto seria muito difícil de ser feito com o código do Exemplo 1.

No ciclo principal, pode-se fazer algo útil ou então fazer o processador entrar num modo de baixo consumo. Quanto acontece a interrupção, o processador acorda, processa e o ciclo principal faz com que ele volte a dormir. A condição para isto é a de que o SysTick use uma outra fonte para o relógio. Isto se traduz numa enorme redução do consumo de energia e em sistema com bateria ou pilhas, maior tempo de uso.

Para isto, basta se mudar o ciclo principal para

```
for (;;) { __WFI(); }
```

A instrução *assembly* WFI (Wait for interrupt) inserida pelo comand __WFI(() faz com que o processador entre no estado de baixa consumo especificado nos registradores SCR e PCON. Além disso, existe uma série de registradores que permitem escolher quais periféricos permanecerão ativos nos diversos modos de baixo consumo.

Exemplo 5 – Acessando o Joystick

A placa contém um joystick, que é um conjunto de cinco botões: Cima, Baixo, Esquerda, Direita e Centro. O joystick usa 5 pinos da Porta A (GPIOA), que devem ser configurados como entrada.

Botão	Pin	Porta	Bit	Label
CENTER	23	GPIOA	0	JOY_CENTER
LEFT	24	GPIOA	1	JOY_LEFT
RIGHT	25	GPIOA	2	JOY_RIGHT
UP	26	GPIOA	3	JOY_UP
DOWN	30	GPIOA	5	JOY_DOWN

Neste exemplo, o LED vermelho piscará com uma frequência que pode ser alterada acionando-se o joystick para cima (aumenta a frequencia) e para baixo (diminui a frequência). O LED verde será controlado pelo joystick para esquerda (apagar) e direita (acender).

Não será feito debounce!!!!

Nesta versão o código da aplicação está em um único arquivo, main.c.

Para aumentar a legibilidade na configuração da Porta A (GPIOA), ela é feita usando-se uma variável auxiliar que contem o valor do registrador, sofre todas as alterações necessárias e enfim é escrita no registrador.

```
/* Configure GPIOA */
t = GPIOA->MODER;
BITFIELDSET(t,BITVALUE(3,JOY_DOWN_PIN*2),0);                          // Set to input
BITFIELDSET(t,BITVALUE(3,JOY_UP_PIN*2),0);                            // Set to input
BITFIELDSET(t,BITVALUE(3,JOY_LEFT_PIN*2),0);                          // Set to input
BITFIELDSET(t,BITVALUE(3,JOY_RIGHT_PIN*2),0);                         // Set to input
GPIOA->MODER = t;

t = GPIOA->PUPDR;
BITFIELDSET(t,BITVALUE(3,JOY_DOWN_PIN*2),BITVALUE(2,JOY_DOWN_PIN*2));// Set pull down
BITFIELDSET(t,BITVALUE(3,JOY_UP_PIN*2),BITVALUE(2,JOY_UP_PIN*2));    // Set pull down
BITFIELDSET(t,BITVALUE(3,JOY_LEFT_PIN*2),BITVALUE(2,JOY_LEFT_PIN*2));// Set pull down
BITFIELDSET(t,BITVALUE(3,JOY_RIGHT_PIN*2),BITVALUE(2,JOY_RIGHT_PIN*2));//Set pull down
GPIOA->PUPDR = t;
```

No ciclo principal, é feita a leitura do registrador de entrada (IDR) e cada pino (bit) é testado. No caso de estar pressionado, é executada a ação correspondente. Um esquema simples é usado para detectar se houve um pressionamento. Uma variável (idrant) armazena o valor do pino no ciclo anterior. Se houve uma mudança, significa que o botão correspondente foi pressionado.

```
uint32_t idrant = 0;
for (;;) {
    uint32_t idr;
    idr = GPIOA->IDR;
    if( idr&JOY_DOWN ) {
        if( (idrant & JOY_DOWN) == 0 ) {
            if( perred>200 ) perred -= 200;
        }
        idrant |= JOY_DOWN;
     } else {
        idrant &= ~JOY_DOWN;
```

```
    }
    ...
}
```

Este esquema não funciona como debouncing. Pode acontecer que a frequência aumente ou diminua mais do que o desejado quando se apertam os botões ACIMA e ABAIXO. Os botões ESQUERDA e DIREITA não tem este problema, pois a ocorrencia de mais do que uma transição não alterará o resultado.

Exemplo 6 – Acessando o Joystick e controlando os LEDs.

Neste exemplo, os botões ESQUERDA e DIREITA apaga e acende, respectivamente, o LED VERDE; os botões ACIMA e ABAIXO, acende e apaga, respectivamente, o LED VERMELHO e o botão CENTRAL apaga os LEDs.

Exemplo 7 – Acessando o Joystick usando interrupções

O esquema de interrupção das GPIOs do STM32L476 é diferente do usual. No caso, um sinal no pino 0 de qualquer GPIO gera uma interrupção EXTI0 e assim para os outros pinos. Apenas um pino pode gerar a interrupção. Em outras CPUs, mesmo sendo Cortex, todos os pinos de uma GPIO geram a mesma interrupção e cabe ao software, inspecionar registradores da GPIO, para determinar a origem.

Uma rotina de interrupção é definida apenas pelo nome que deve ser idêntico ao existente no arquivo startup_stm32l476.c. A rotina de interrupção que processa o pino PA0 (conectado ao botão CENTRAL) tem o código abaixo.

```
void EXTI0_IRQHandler(void) { /* CENTER PIN */
    if( (EXTI->IMR1&EXTI_IMR1_IM0) && (EXTI->PR1&EXTI_PR1_PIF0) ) {
        /* Turn off all LEDS */
        GPIOE->ODR &= ~LED_GREEN;
        GPIOB->ODR &= ~LED_RED;
        EXTI->PR1 |= EXTI_PR1_PIF0;
    };
    NVIC_ClearPendingIRQ(EXTI0_IRQn);
}
```

A rotina de interrupção antes de retornar deve sinalizar que atendeu a interrupção chamando a rotina (definida pelo CMSIS) NVIC_ClearPending.

Os pinos 5 a 9 são agrupados para serem tratados por uma única rotina de interrupção (EXIT9_5_IRQn).

Para habilitar a GPIO a gerar interrupção deve-se configurar o *Extended Interrupts and Events Controller* (EXTI), habilitando os pinos que podem gerar interrupção, no caso, de 0 a 5 (exceto 4) e qual ou quais as transições que gerarão interrupção, subida (rising) ou descida (falling).

```
EXTI->IMR1  |=  (BIT(0)|BIT(1)|BIT(2)|BIT(3)|BIT(5));      // linhas habilitadas
EXTI->RTSR1 |=  (BIT(0)|BIT(1)|BIT(2)|BIT(3)|BIT(5));      // interrupção na subida
```

O *System Configuration Controller* (SYSCFG) também deve ser configurado para permitir a interrupção. Neste caso, existe um grupo de 4 bits, que especificam qual a a porta que pode gerar a interrupção. O registrador EXTICR1 configura as linhas EXTI0 a EXTI3 e o registrador EXTICR2, as linhas EXTI4 a EXTI7. O arquivo cabeçalho especifica estes registradores como um array, EXTICR, indexado de 0 a 3).

```
   t = SYSCFG->EXTICR[1];
    SETBITFIELD(t,BITVALUE(7,4),0);  /* EXTI5 : Down Pin */
    SYSCFG->EXTICR[1] = t;
```

E finalmente configurar o *Nested Vectored Interrupt Controller* (NVIC), que faz parte de todo Cortex M, para habilitar a interrupção usando a rotina CMSIS abaixo.

```
    NVIC_EnableIRQ(EXTI0_IRQn);
```

Isto deve ser repetido para cada linha (pino).

Na rotina de interrupção pode ser verificado qual o evento que causou a interrupção, acessando-se os registradores IMR1, que sinaliza se a interrupção está ativa, e PR1, que sinaliza se existe uma interrupção pendente na linha.

O código acima tem problemas de modularidade. Se houver mudança no pino onde está conectado um botão, deve ser alterada a linha que define o simbolo, por exemplo, JOY_UP_PIN e a linha que habilita interrupção, onde o BIT(3) corresponde ao pino.

Exemplo 8 – Piscamento controlado pelo joystick

Este exemplo é uma fusão do Exemplo 7 (Acessando o Joystick usando interrupções) com o Exemplo 4 (Piscamento controlado pelo SysTick). As ações são especificadas dentro das rotinas de interrupção.

Exemplo 9 – Criando uma HAL para os LEDs

O uso de Camadas de Abstração de Hardware (HAL – Hardware Abstraction Layer), simplifica a programação, aumenta a portabilidade, permite o reuso de código e diminui, consideravelmente, a quantidade de erros.

Este exemplo, mostra uma camada de abstração de hardware para os LEDs. Isto é feito através de uma implementação no arquivo led.c e a exportação dos símbolos, no arquivo led.h.

No arquivo led.h, são definidos os símbolos que serão usados nas chamadas das rotinas.

```
#define LED_GREEN_PIN   (8)
#define LED_RED_PIN     (2)
#define LED_BIT(N)      (1UL<<(N))
#define LED_GREEN       LED_BIT(LED_GREEN_PIN)
#define LED_RED         LED_BIT(LED_RED_PIN)
#define LED_ALL         (LED_GREEN|LED_RED)
```

Neste mesmo arquivo, são especificadas as rotinas abaixo.

```
uint32_t LED_Init(uint32_t leds);
uint32_t LED_Write(uint32_t on, uint32_t off); // on and then off
uint32_t LED_Toggle(uint32_t leds);
```

O arquivo led.c mostra a implementação das rotinas, que em principio, não necessitam ser conhecidas pelo programador que as está usando.

LED_Init	encarrega-se de toda a inicialização inicializando apenas os LEDs especificados
LED_Write	acende o LED e/ou os apaga conforme os parametros
LED_Toggle	troca o estado da(s) saída(s) especificada(s)

Em geral, o uso de uma HAL implica numa certa ineficiência devido ao custo de chamado das rotinas, mas isto pode ser minorado através de rotinas inline ou macros, definidos no arquivo de exportação de símbolos.

Em geral, o código fica muito mais legível e menor. No caso do exemplo 1, o arquivo main.c ficaria reduzido ao seguinte.

```
#include "stm32l476xx.h"
#include "system_stm32l476.h"
#include “led.h”

void ms_delay(volatile int ms) {
   while (ms-- > 0) {
      volatile int x=700;
      while (x-- > 0)
         __NOP();
   }
}

int main(void) {

       LED_Init(LED_ALL);
       for (;;) {
              ms_delay(500);
```

```
            LED_Toggle(LED_RED|LED_GREEN);
    }
}
```

Neste exemplo, o impacto é menor, pois ainda não há uma HAL para o joystick, mas o efeito pode ser visualizado abaixo.

```
void SysTick_Handler(void) {

    if( debounce_counter ) debounce_counter--;

    if( blinking ) {
        if( cnt_red == 0 ) {
            LED_Toggle(LED_RED);
            cnt_red = semiperiod_red;
        }
        cnt_red--;

        if( cnt_green == 0 ) {
            LED_Toggle(LED_GREEN);
            cnt_green = semiperiod_green;
        }
        cnt_green--;
    }
}
```

Exemplo 10 – Criando uma HAL para o *joystick*

A ideia de uma camada de Abstração de Hardware (HAL) pode ser estendida para o *joystick*. Para leituras são possíveis duas abordagens: varredura (*polling*) e interrupção. Uma implementação usando varredura é mais simples e será usada em exemplos futuros,. Mas esta abordagem tem restrições e eventos podem ser ignorados se as varreduras não forem frequentes.

Uma abordagem baseada em interrupção exige que se registrem rotinas, que serão chamadas sempre que houver uma interrupção, mas todo o tratamento fica interno a implementação. No caso, em questão foi usada uma estrutura composta de apontadores para funções (*function pointers*), uma para cada botão, que são rotinas de *callback*.

```
typedef struct JoyStick_Callback {
    void (*CenterButtonPressed)(void);
    void (*LeftButtonPressed)(void);
    void (*RightButtonPressed)(void);
    void (*UpButtonPressed)(void);
    void (*DownButtonPressed)(void);
} JoyStickCallBack;
```

Na rotina de inicialização Joystick_Init é passado um apontador para uma estrutura deste tipo. Depois é só aguardar que a rotina especificada seja chamada.

O programa principal fica muito reduzido.

```
int main(void) {

    SystemCoreClockSet(MSI48M_CLOCKSRC,0,2,0);

    APBPeripheralClockSet(0,0); /* Enable APBx */

    LED_Init(LED_ALL);

    JoyStick_Init(&joystick_callback);

    SysTick_Config(SystemCoreClock/1000);   /* 1 ms */

    for (;;) {}
}
```

O processamento fica dentro das rotinas de *callback*, como no exemplo abaixo.

```
void CenterButtonProcessing(void) {

    if( debounce_counter == 0 ) {
        if( blinking ) {
            /* Turn off all LEDS */
            LED_Write(0,LED_ALL);
            blinking = 0;
        } else {
            blinking = 1;
        }
        debounce_counter = 40;
        cnt_red = 0;
        cnt_green = 0;
    }
}
```

Neste caso, já foi implementado um esquema de *debounce*. O contador debounce_counter é decrementado na rotina SysTick_Handler e somente quando chega a zero, o processamento é reativado. No caso, durante 40 ms após a detecção da transição, sinais do botão são ignorados.

O aspecto mais evidente é a separação entre a aplicação e os detalhes de acesso ao hardware.

Exemplo 11 – Usando uma arquitetura gatilhada por tempo

Os exemplos mostrados até agora usam uma de duas abordagens:

Super loop	Todo o processamento é feito no programa principal, que é um ciclo
Back/Foreground	O processamento é feito parcialmente ou em todo, nas rotinas de interrupção

Ambos esquemas apresentam limitação quando a complexidade do programa aumenta, principalmente em relação a temporização.

Um esquema apresentado por Michael Pont em *Patterns for Time Triggered Embedded Systems[6]*[7] evita o uso de interrupções de uma maneira geral (somente uma de tempo é usada). A implementação é mostrada em tte.c e a interface em tte.h.

```
uint32_t Task_Init(void);
uint32_t Task_Add(void (*task)(void), uint32_t period, uint32_t delay);
uint32_t Task_Delete(uint32_t i);
uint32_t Task_Dispatch(void);
uint32_t Task_Update(void);
```

A rotina Task_Update deve ser chamada dentro da rotina de interrupção periodica (*tick*). A rotina Task_Dispatch, no ciclo principal. Cada tarefa (Task) é uma rotina C que não recebe nem retorna parametros. Ela é adicionada a lista de tarefas controladas especificando-se o período de ativação. Pode-se especificar também o atraso inicial, assim, varias tarefas com mesmo período seriam ativadas em *ticks* diferentes.

O programa principal fica muito simples.

```
int main(void) {

    SystemCoreClockSet(MSI48M_CLOCKSRC,0,2,0);

    LED_Init(LED_ALL);

    JoyStick_Init();

    Task_Init();

    taskno_green = Task_Add(Blink_Green,semiperiod_green,0);
    taskno_red   = Task_Add(Blink_Red,semiperiod_red,0);

    taskno_button  = Task_Add(ButtonProcessing,10,5);

    SysTick_Config(SystemCoreClock/1000);   /* 1 ms */

    for (;;) {
        Task_Dispatch();
    }
}
```

7 Disponível em http://www.safetty.net/publications/pttes

O processamento fica dentro das tarefas, que não são executadas dentro das interrupções. A tarefa Blink_Green, encarregada do piscamento do LED verde fica muito simples.

```
void Blink_Green(void) {
    if( blinking )
        LED_Toggle(LED_GREEN);
    else
        LED_Write(0,LED_GREEN);
}
```

Uma modificação teve que ser feita na HAL do joystick. Como não se deve usar interrupções, foi modificada a interface com o acréscimo de uma rotina de varredura (leitura) e a modificação do rotina de inicialização, agora sem parâmetros.

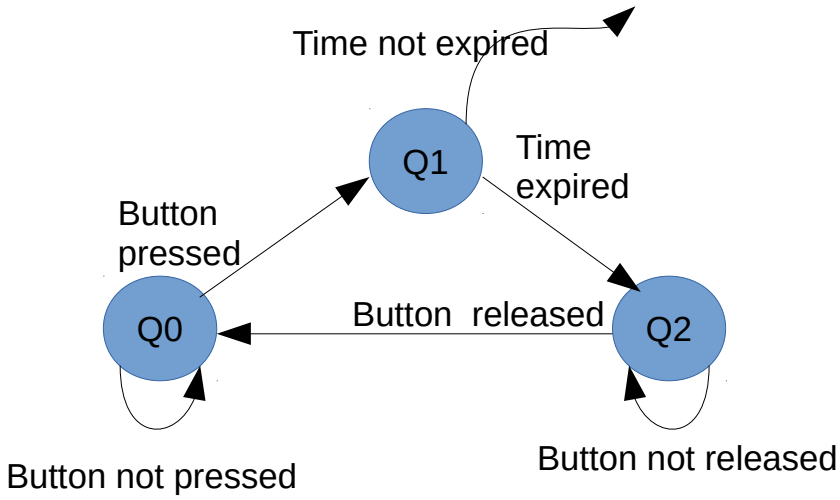
```
uint32_t JoyStick_Init(void);
uint32_t JoyStick_Read(void);
```

A processamento do botão CENTRAL é feito através de uma maquina de estados implementada

```
void ButtonProcessing(void) {
typedef enum { Q0, Q1, Q2 } state_t;
static state_t state = Q0;
static uint32_t debouncing_counter;
uint32_t b;

    switch(state) {
    case Q0: // Waiting Press of Button
        b = JoyStick_Read();
        if( b&JOY_CENTER ) {
            blinking = ! blinking;
            state = Q1;
            debouncing_counter = 10;
        }
        break;
    case Q1: // Ignore transitions
        if( --debouncing_counter ==  0 ) {
            state = Q2;
        }
        break;
    case Q2: // Wait release
        b = JoyStick_Read();
        if( (b&JOY_CENTER)==0 ) {
            state = Q0;
        }
        break;
    }
}
```

Esta rotina implementa uma máquina de estados cujo funcionamento está descrito no diagrama de estados abaixo. No estado Q0, o botão é monitorado. Caso seja detectado que foi pressionado, oindicador *blinking* é invertido, o contador de *debounce* é carregado com o valor inicial e o estado alterado para Q1. No estado Q1, qualquer modificação no valor do sinal do botão é ignorado. Ao final do tempo, contador chega a zero, o estado passa a ser Q2. No estado Q2, o botão é monitorado para verificar se foi solto (o processador é mais rápido que o dedo!). Quando for solto ou se já estiver solto, o estado passa a ser Q0 novamente.



Esta abordagem é simples e segura, requer hardware simples (apenas um temporizador) e sua temporização pode ser garantida conhecendo-se os tempos de execução de cada tarefa. As tarefas são do tipo Run To Completion (RTC), ou seja, devem sempre retornar. Isto configura um sistema cooperativo de multitarefas (*multitasking*). Exatamente isto é o ponto fraco da abordagem. Se uma tarefa não retornar, as outras não serão ativadas e o funcionamento não funcionará corretamente. É possível se acrescentar um código impondo um limite de tempo, que será verificado em Task_Update, e caso este limite seja ultrapassado, medidas podem ser tomadas para minorar o dano.

O funcionamento do sistema gatilhado no tempo é bem simples. Existe uma lista de tarefas e seus atributos. Um deles é o apontador para função que implementa o código.

```
typedef struct {
    void    (*task)(void);  // pointer to function
    // time unit is ticks
    uint32_t    period;     // period, i.e. time between activations
    uint32_t    delay;      // time to next activation
    uint32_t    runcnt;     // if greater than 1, overrun
} TaskInfo;

/// Task info table
static TaskInfo taskinfo[TASK_MAXCNT];
```

Na rotina chamada a cada tick, verifica-se quais os tempos expiraram e quais as tarefas estão prontas para serem executadas, ou seja com runcnt maior do que 0.

```
void Task_Update(void) {
int i;
TaskInfo *p;

    for(i=0;i<TASK_MAXCNT;i++) {
        p = &taskinfo[i];
        if( p->task ) {
            if( p->delay == 0 ) {
                p->runcnt++;
                p->delay = p->period;
            } else {
                p->delay--;
            }
        }
    }
```

```
}
```

A rotina Task_Dispatch chamada continuamente no ciclo principal, somente verifica quais as tarefas tem runcnt maior do que 0 e as executa.

```
uint32_t
Task_Dispatch(void) {
int i;
TaskInfo *p;

    for(i=0;i<TASK_MAXCNT;i++) {
        p = &taskinfo[i];
        if( p->task ) {
            if( p->runcnt ) {
                p->task();
                p->runcnt--;
            }
        }

    }

    return 0;
}
```

Exemplo 12 – Uma arquitetura nova gatilhada por tempo

Em seu novo livro *The Engineering of Reliable Embedded Systems*,[7,8][8] Michael Pont apresenta uma arquitetura ligeiramente modificada. A principal modificação foi simplificar a rotina Task_Update, que é chamada em toda interrupção (tick) . Ela passa apenas a incrementar uma variável. Todo o processamento é feito na rotina Task_Dispatch com os cuidados necessários para que não haja corridas (*hazards*).

```
Void Task_Update(void) {

    task_tickcounter++;

}

uint32_t
Task_Dispatch(void) {
int i;
TaskInfo *p;
uint32_t dispatch;

    __disable_irq();
    if( task_tickcounter > 0 ) {
        task_tickcounter--;
        dispatch = 1;
    }
    __enable_irq();


    while( dispatch ) {
        for(i=0;i<TASK_MAXCNT;i++) {
            p = &taskinfo[i];
            if( p->task ) {
                if( p->delay == 0 ) {
                    p->task();      // call task function
                    if( p->period == 0 ) { // one time tasks are dangerous
                        Task_Delete(i);
                    } else {
                        p->delay = p->period;
                    }
                } else {
                    p->delay--;
                }
            }

        }
        __disable_irq();
        if( task_tickcounter > 0 ) {
            task_tickcounter--;
            dispatch = 1;
        } else {
            dispatch = 0;
        }
        __enable_irq();

    }
    return 0;
}
```

8 Alguns capítulos e o código fonte estão disponíveis em http://www.safetty.net/publications/the-engineering-of-reliable-embedded-systems-second-edition e http://www.safetty.net/publications/the-engineering-of-reliable-embedded-systems

Exemplo 13 – Protothreads[9]

Adam Dunkels apresentou uma maneira de se emular um sistema multitarefas cooperativos, sem que as tarefas aparentassem ser RTC (Run To Completion), mas na verdade, são.

Geralmente o código para tarefas em sistemas multitarefas como FreeRTOS e uc/os tem o modelo abaixo.

```
Void Task(void) {
	// Inicializacao
	while(1) {
		// processo
		// espera
	}
}
```

Em protothreads, é usada uma técnica baseada no mecanismo de Duff, que foi usado originalmente para acelerar a copia de conteudo de uma posição da memória para outra. A codificação convencional para isto em C é mostrada abaixo.

```
send(char *to, char *from, int count) {
int n = count;
	while( n-- > 0 ) {
		*to++ = *from++;
	}
}
```

Na implementação acima, a cada cópia é feito o controle, comparando e decrementando o contador n. Mas usando o fato de que o padrão C não impõe restrições às instruções que podem ser colocadas dentro de um switch e que, se não houver, um break, as instruções seguintes são executadas, pode-se ter um controle a cada 8 instruções, acelerando o processo de cópia.

```
send(char *to, char *from, int count) {
int n=(count+7)/8;
	switch(count%8){
	 case 0: do{	*to = *from++;
	 case 7:	*to = *from++;
	 case 6:	*to = *from++;
	 case 5:	*to = *from++;
	 case 4:	*to = *from++;
	 case 3:	*to = *from++;
	 case 2:	*to = *from++;
	 case 1:	*to = *from++;
			}while(--n>0);
	 }
}
```

Primeiro é calculado quantas vezes o ciclo deve ser repetido, considerando que a cada ciclo, 8 bytes são copiados. Na primeira vez que o switch é executado, o resto da divisão por 8 é o número de bytes copiados. Quando se chega ao while, passa-se a copiar 8 bytes a cada ciclo.

Em protothreads, é feito uso intensivo de MACROS para que uma tarefa tenha a aparencia da tarefa Task mostrada acima.

9 Acessível em http://dunkels.com/adam/pt/

```
PT_THREAD(Blink_Green(struct pt *pt)) {
static uint32_t tstart;

    PT_BEGIN(pt);
    while(1) {
        // processo
        if( blinking )
            LED_Toggle(LED_GREEN);
        else
            LED_Write(0,LED_GREEN);
        // espera
        tstart = msTick;
        PT_WAIT_UNTIL(pt,((msTick-tstart)>=semiperiod_green));
    }
    PT_END(pt);
}
```

Este código após passar pelo preprocessador tem a forma abaixo.

```
char link_Green(struct pt *pt)) {
static uint32_t tstart;

    switch(pt->lc) {
    case 0:
      while(1) {
        // processo
        if( blinking )
           LED_Toggle(LED_GREEN);
        else
           LED_Write(0,LED_GREEN);
        // espera
         tstart = msTick;
        pt->lc = 1;
     case 1:
         if( ! (msTick-tstart)>=semiperiod_green) ) return PT_WAITING;
          pt->lc = 0;
      }
    }
    return PT_ENDED;
}
```

No código acima, pode-se ver que a rotina retorna com o código PT_WAITING ou PT_ENDED.

Também pode-se ver algumas restrições que devem ser obedecidas:

1. Todas as variáveis locais devem ser estáticas, para manterem os valores em cada chamado.
2. O uso de *switch* é complicado pelo fato de que pode se emaranhar com o implantado pelo protothread.
3. O otimizador pode reordenar as instruções e atrapalhar o funcionamento do mecanismo.

O ciclo principal é simples.

```
void main(void) {


    SystemCoreClockSet(MSI48M_CLOCKSRC,0,2,0);

    LED_Init(LED_ALL);

    JoyStick_Init();

    PT_INIT(&pt_BlinkGreen);
    PT_INIT(&pt_BlinkRed);
    PT_INIT(&pt_ButtonProc);
```

```
    SysTick_Config(SystemCoreClock/1000);   /* 1 ms */

    for (;;) {
        PT_SCHEDULE(Blink_Green(&pt_BlinkGreen));
        PT_SCHEDULE(Blink_Red(&pt_BlinkRed));
        PT_SCHEDULE(ButtonProc(&pt_ButtonProc));
    }
}
```

Exemplo 14 – Usando o RIOS[10]

RIOS (Riverside-Irvine Operating Systems) é um núcleo multitarefa cooperativo bastante similar ao mostrado por Pont (Exemplos 11 e 12). Isto pode ser verificado no código abaixo.

```
uint32_t msTick = 0;
void SysTick_Handler(void) {
    msTick++;
    Task_Update();
}

void main(void) {


    SystemCoreClockSet(MSI48M_CLOCKSRC,0,2,0);

    LED_Init(LED_ALL);

    JoyStick_Init();

    Task_Init();

    taskno_green   = Task_Add(Blink_Green,semiperiod_green,0);
    taskno_red     = Task_Add(Blink_Red,semiperiod_red,0);

    taskno_button  = Task_Add(ButtonProcessing,10,5);

    SysTick_Config(SystemCoreClock/1000);   /* 1 ms */

    for (;;) {
        Task_Dispatch();
    }
}
```

Da mesma forma, as tarefas são codificados como rotinas Run To Completion, ou seja, devem sempre retornar para que outra tarefa possa ser executada.

10 http://www.cs.ucr.edu/~vahid/rios

Exemplo 15 – Usando o Super Simple Tasker (SST)[11]

Robert Ward em 2003 [11] e Miro Samek e Robert Ward em 2008 [12][12] apresentaram um esquema que permite uma forma restrita de preempção entre tarefas. O conceito de níveis de interrupção foi expandido para tarefas. Assim, uma tarefa de prioridade alta pode interromper uma tarefa de prioridade baixa. Para isto, há uma interrupção periódica (tick), quando se verifica se uma tarefa de prioridade mais alta deve se executada.

Todas as tarefas devem ser do tipo Run To Completion, isto é, retornar, para dar vez a tarefas de prioridade mais baixa. Só é necessária uma pilha, pois tarefas só podem ser executadas quando todas as tarefas de prioridade mais alta foram executadas.

O Super Simple Tasker é a base da implementação de um sistema de implementação de máquinas hierarquicas de estado, QP apresentado no livro de Miro Samek, Practical UML Statecharts in C/C++ [13].

O programa principal é simples como mostrado abaixo.

```
void main(void) {

    SystemCoreClockSet(MSI48M_CLOCKSRC,0,2,0);


    LED_Init(LED_ALL);

    Button_Init(&joystick_callback);

    SST_init();

    //SST_task(myTask,myTask_ID, myTask_EQ, myTask_EVQL, SST_SIGNAL_TASKINIT, 0);

    SST_task(Task_Blink_Green, BLINK_GREEN_PRIO, BlinkGreenQueue, QUEUE_SIZE,
             SST_SIGNAL_TASKINIT, 0);
    SST_task(Task_Blink_Red, BLINK_RED_PRIO,     BlinkRedQueue,   QUEUE_SIZE,
             SST_SIGNAL_TASKINIT, 0);
    SST_task(Task_Button, BUTTON_PRIO,           0, 0, SST_SIGNAL_TASKINIT, 0);

    SysTick_Config(SystemCoreClock/1000);   /* 1 ms */


    SST_run();
}
```

A rotina de piscar um LED tem uma estrutura simples. A mensagem SST_SIGNAL_TASKINIT é passada na primeira vez que a rotina é chamada para se fazer inicialização. A rotina deve sempre retornar rapidamente.

```
void Task_Blink_Green(SSTEvent event){
static uint32_t lasttick = 0;

    if(event.sig!=SST_SIGNAL_TASKINIT) {
        if( blinking ) {
            if( msTick > (lasttick+semiperiod_green) ) {
                LED_Toggle(LED_GREEN);
                lasttick = msTick;
            };
        } else {
            LED_Write(0,LED_GREEN);
        }
```

11 A versão para ARM Cortex M pode ser achada em https://github.com/upiitacode/SST_ARM.

12 O artigo e o código (para DOS) podem ser achados em http://www.state-machine.com/doc/articles.html

```
    }
}
```

Mas muita coisa acontece nas rotinas de interrupção. Na rotina de interrupção de tempo SysTick_Handler deve ser verificado se não há tarefas de prioridade mais alta a serem executadas. Para evitar problemas de corrida, um protocolo deve se seguido. No ínicio da rotina, deve ser chamada a rotina (na verdade, uma macro) SST_ISR_ENTRY. E no termino do processamento da interrupção, deve ser chamada a rotina (macro) SST_ISR_EXIT.

```
void SysTick_Handler(void) {
int pin;

    SST_ISR_ENTRY(pin,ISR_TICK_PRIO);
//  SST_post(myTask_ID,1,0);
    msTick++;
    SST_post(BLINK_GREEN_PRIO, ISR_TICK_SIG, 0);
    SST_post(BLINK_RED_PRIO, ISR_TICK_SIG, 0);
//    SST_post(BUTTON_PRIO, ISR_TICK_SIG, 0);
    SST_ISR_EXIT(pin,(SCB->ICSR = SCB_ICSR_PENDSVSET_Msk));

}
```

As rotinas de interrupção do joystick devem seguir o mesmo protocolo. Assim elas tem a forma abaixo.

```
void EXTI0_IRQHandler(void) { /* CENTER BUTTON */
int pin;
    SST_ISR_ENTRY(pin,ISR_EXTI0_ID);

    if( (EXTI->IMR1&EXTI_IMR1_IM0) && (EXTI->PR1&EXTI_PR1_PIF0) ) {
        if( callback.CenterButtonPressed )
            callback.CenterButtonPressed();
        EXTI->PR1 |= EXTI_PR1_PIF0;

    };

    SST_ISR_EXIT(pin,(SCB->ICSR = SCB_ICSR_PENDSVSET_Msk));
}
```

A abordagem baseada em SST apresenta algumas vantagens:

1. Necessita uma única pilha.
2. Apresenta possibilidade de preempção.
3. É relativamente simples e pequena.

Exemplo 16 – Usando o FreeRTOS[13]

O FreeRTOS [14] é um núcleo de tempo real preemptivo altamente portável. Como é um sistema preemptivo sem restrições, cada tarefa deve ter uma pilha separada. Basicamente o chaveamento de tarefas corresponde a uma troca de pilha, pois cada pilha tem o endereço de retorno.

O FreeRTOS tem uma API (Application Programming Interface) grande e bem completa. Tem suporte para vários mecanismos de comunicação entre tarefas.

O programa principal tem a forma abaixo. São criadas várias tarefas e dado inicio ao escalonamento com o chamada da rotina vTaskStartScheduler, que em condições normais, nunca retorna.

```
int main(void) {


    SystemCoreClockSet(MSI48M_CLOCKSRC,0,2,0);

    LED_Init(LED_ALL);

    JoyStick_Init(&joystick_callback);

    xTaskCreate(Task_Blink_Red,"Red", 1000,0,1,0);
    xTaskCreate(Task_Blink_Green,"Green", 1000,0,2,0);

    vTaskStartScheduler();

    while(1) {}     // Just in case
}
```

As tarefas são independentes e apresentam a estrutura clássica para sistemas preemptivos, nunca retornando.

```
void Task_Blink_Green(void *pvParameters){
const portTickType xFrequency = 500;
portTickType xLastWakeTime=xTaskGetTickCount();

    while(1) {
        if( blinking ) {
            LED_Toggle(LED_GREEN);
            vTaskDelayUntil(&xLastWakeTime,xFrequency);
        } else {
            LED_Write(0,LED_GREEN);
            vTaskDelay(semiperiod_green);
        }
    }
}
```

Esta técnica representa uma das formas mais poderosas de construir um sistema de tempo real. Um dos problemas é o custo em termos de uso de memória RAM, pois cada tarefa deve ter sua proproa pilha e ela deve ser dimensionada para o máximo.

Para se verificar a temporização, pode-se usar o Rate Monotonic Analysis (RMA) baseado em Liu e Wayland [15]. Considerando-se que [16] as *n* tarefas são independentes e que o final o período é o limite para a resposta, e sabendo-se de cada tarefa, o período T_i e a carga (tempo de execução do processo) C_i, pode-se calcular a taxa de utilização de CPU de cada processo U_i.

13 www.freertos.org

$$U_i = \frac{C_i}{T_i}$$

e a carga total U

$$U = \sum_{i=1}^{n} U_i$$

Se U for maior que 1, é impossível a alocação. Se U for menor que $n\left(2^{\frac{1}{n}} - 1\right)$ a alocação é possível. Entre este valor e 1, deve ser feita uma analise minuciosa (através de simulação) para verificar a viabilidade.

n	$n\left(2^{\frac{1}{n}} - 1\right)$
1	1
2	0,83
3	0,78
4	0.76
5	0,74
10	0,71
100	0,69

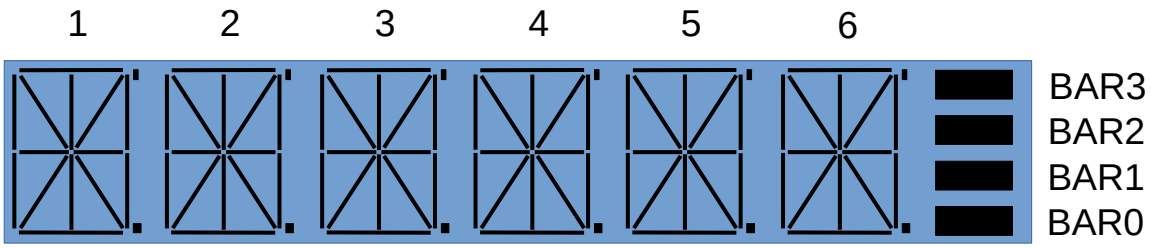
Em geral, esta abordagem leva a soluções garantidas, mas muito custosas, pois C_i é considerado para o pior caso e o sistema considera a possibilidade (baixa) de todos as tarefas demandarem o máximo de tempo.

Além disso, quando as tarefas são interdependentes, deve ser considerado o problema da inversão de prioridade, quando uma tarefa de baixa prioridade controla um recurso (por exemplo, um semáforo), que passa a ser demandado por uma tarefa de prioridade alta. Como a tarefa tem baixa prioridade, ela não consegue ser escalonada e não consegue liberar o recurso, impedindo a tarefa de prioridade alta de ser escalonada.

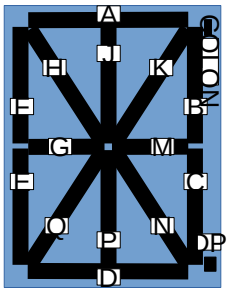
Também, existe o problema de ocorrer um *deadlock*, quando se alocam recursos. Duas tarefas necessitam dos recursos A e B. A tarefa 1 solicita na sequencia A e B e a tarefa 2, na sequencia B e A. Se a tarefa 1 for interrompida logo após conseguir o recurso A, pode acontecer da tarefa 2 conseguir o recurso B e solicitar o A, quando então ela entra em espera. Quando a tarefa 1 é escalonada novamente, ela solicita o recurso B, e entra em espera. Nesta configuração, nenhuma das tarefas consegue ser executada.

Exemplo 17 – Usando o Display LCD

A placa STM32L476 Discovery tem um display LCD GH08172T, que tem 6 caracteres de 14 segmentos, 4 barras, 4 pontos decimais e 4 apóstrofos. O layout está mostrado abaixo.



Cada caractere é construído com 14 segmentos como mostrado.



Se cada segmento fosse controlado por um pino, seriam necessários 96 pinos (= 6 x 14 + 4 + 4 + 4). Isto aumentaria o custo de fabricação dos componentes, das placas e esgotaria os pinos do microcontrolador STM32L476 da placa, que tem 100 pinos.

A solução é a multiplexação. O *display* é controlador por 24 pinos que acionam os segmentos em cada fase e 4 pinos, que sinalizam qual a fase, num total de 28 pinos. Estas fases devem ocorrer em sequencia e numa frequência acima da frequência de persistência retiniana (aprox. 20 Hz).

O microcontrolador STM32L476 tem um periférico LCD capaz de controlar até 40 segmentos com 8 fases (320 segmentos) ou 44 segmentos com 4 fases (176 segmentos). Isto é feito armazenando-se em posições de memória especificas para cada fase quais os segmentos que devem ser acionados.

Antes disso, os pinos devem ser configurados como sendo controlados pelo periférico LCD, o controlador do periférico deve ser configurado adequadamente. Quando se desejar exibir um caractere deve haver uma transformação, que indique quais segmentos devem ser ativados de acordo com a posição no display.

Tudo isto é abstraido pelo código em lcd.c, cuja interface é mostrada em lcd.h. O HAL consiste das rotinas abaixo.

LCD_Init	Inicializa todos os pinos e o controlador
LCD_DeInit	Libera pinos e desliga controlador
LCD_WriteString	Escreve no *display* os 6 primeiros caracteres do parâmetro
LCD_WriteSegments	Escreve quais segmentos serão acesos
LCD_Clear	Limpa o display
LCD_WriteBars	Acende as *barras* de acordo com o parâmetro
LCD_WriteToRAM	Escreve diretamente na RAM

Também define 14 símbolos: SEGA, SEGB, … SEGQ, usando 1 de 14 bits, o que cabe num uint32_t. Internamente, estes segmentos tem outra codificação, usando 1 de 40 bits, sendo

necessários dois uint32_t para tal. Por este motivo, a escrita direta em RAM é dificil, pois é necessário se conhecer a conexão entre o microcontrolador e o *display*.

A implementação é baseada em tabelas. Uma delas especifica quais segmentos devem ser acesos em determinada fase para fazer aparecer o caractere deseja na posição desejada. A matriz tabmcusegfromchar é uma matriz tridimensional, indexada pelo valor inteiro do caractere, pela posição (0 a 5) e pela fase (0 a 3). A matriz tabmcusegfromseg é similar, mas é indexada pelo segmento que se quer acender, a posição e a fase. Nos dois casos, o retorno é um vetor de bits de 40 caracteres.

Exemplo 18 – Usando a UART

O microcontrolador tem diversos periféricos para comunicação serial, com alguns deles aparecendo em mais do que um pino. Alguns tem capacidade de comunicação sincrona (USARTx) e outros apenas de comunicação assíncrona (UARTx).

	Síncrono	Single Wire Half-duplex	RX	TX	CTS	RTS	CK
USART1			PA10 PG10 PB7	PA9 PG9 PB6	PA11 PG11 PB4	PA12 PG12 PB3	PA8 PG13 PB5
USART2			PA3 PD6	PA2 PD5	PA0 PD3	PA1 PD4	PA4 PD7
USART3			PC5 PB11 PD9 PC11	PC4 PB10 PD8 PC10	PA6 PB13 PD11	PB1 PB14 PD12 PD2	PB0 PB12 PD10 PC12
UART4			PA1 PC11	PA0 PC10	PB7	PA15	
UART5			PD2	PC12	PB5	PB4	
LPUART1			PB10 PC0 PG8	PB11 PC1 PG7	PB13 PG15	PB12 PG6	

A Tabela 12 do Datasheet mostra as capacidades de cada periférico.

Nos conectores externos, está disponível apenas o UART4.

Pino conector P1	Nome	Periférico
10	PA0	UART4_TX
12	PA1	UART4_RX

Além disso, a USART1 está conectada através dos jumpers Jpx e Jpy ao processador de depuração. Este por sua vez, cria uma interface virtual usando a porta USB.

Referencias

[1] 32L476GDISCOVERY. Discovery kit with STM32L476VG MCU. DM00163449.

[2] STM32L476xx Datasheet. Ultra-low-power ARM ® Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 1MB Flash, 128 KB SRAM, USB OTG FS, LCD, analog, audio. DM00108832.

[3] UM1928 User manual. Getting started with STM32L476G discovery kit software development tools. DM00217936.

[4] http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php

[5] RM0351 Reference manual. STM32L4x6 advanced ARM ® -based 32-bit MCUs. DM00083560.

[6] Michael Pont. *Patterns for Time Triggered Embedded Systems.* Addison-Wesley. 2001.

[7] Michael Pont. *The Engineering of Reliable Embedded Systems.* SafeTTy Systems. *2015*.

[8] Michael Pont. *The Engineering of Reliable Embedded Systems. 2nd Edition.* SafeTTy Systems. 2106.

[9] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, Muneeb Ali. *Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems*. In Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006), Boulder, Colorado, USA, Novembro 2006.

[10] Bailey Miller, Frank Vahid e Tony Givargis. RIOS: A Lightweight Task Scheduler for Embedded Systems. WESE'12. Tampere. Finland. 2012.

[11] Robert Ward. Ward, Robert. Practical Real-Time Techniques. Proceedings of the Embedded Systems Conference, San Francisco, 2003.

[12] Miro Samek and Robert Ward. Build a Super Simple Tasker. Embedded Systems Magazine. Julho 2006.

[13] Miro Samek. Miro Samek, Practical UML Statecharts in C/C++, 2nd Edition: Event-Driven Programming for Embedded Systems. Newnes. 2008.

[14] Richard Barry. Using the FreeRTOS Real Time Kernel - Standard Edition. FreeRTOS Tutorial Books. 2010.

[15] C. L. Liu e James. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. Journal of ACM. V. 20, N. 1. 1973.

[16] Lui Sha, Mark H. Klein, John, B. Goodenough. Rate Monotonic Analysis for Real-Time Systems. Technical Report. CMU/SEI-91-TR-006. 1991.

Pinagem

P1						P2
1	3V3				5V_USB	1
2	GND				GND	2
3	2V5				5V_INPUT	3
4	GND				VUSB	4
5	3V				5V	5
6	BOOT0				GND	6
7	PB3				PC14	7
8	PB2				PC15	8
9	PE8				PH0	9
10	PA0	UART4_TX			PH1	10
11	PA5				NRST	11
12	PA1	UART4_RX			GND	12
13	PA2				PE11	13
14	PA3				PE10	14
15	PB6				PE12	15
16	PB7				PE13	16
17	PD0				PE14	17
18	NC				PE15	18
19	GND				GND	19
20	GND				GND	20