Controlador PID - Notas de implementação

Esta implementação mostra uma forma simples de construir um código que realiza *non-preemptive multitasking*, a partir de interrupções do Timer 0. Esta forma de multitasking exige que nenhuma tarefa fique em loop; ela deve realizar suas funções e retornar o mais rapidamente possível. Este tipo de multitasking foi utilizado, por exemplo, no Windows 3.1 e em vários sistemas operacionais para celulares.

Construções adicionais em linguagem C

A implementação do controlador PID utiliza-se de construções em linguagem C que, possivelmente, são desconhecidas pelos alunos, embora presentes em praticamente todas as implementações desta linguagem.

#define

Normalmente, a diretiva `#define` é utilizada para constantes e macros nos programas em linguagem C. Aqui foi utilizada uma definição para configurar o código para o modo de DEBUG no qual alguns valores internos são enviados pelo canal serial para se saber que determinado trecho de programa está funcionando.

```
#define DEBUG
```

Esta linha deve ser comentada quando não estiver em modo DEBUG e os trechos de código de DEBUG não serão compilados. Essa diretiva é utilizada nos arquivos `pid.c` e `protocol.c`.

No código é verificado se a constante está definida com:

```
#ifdef DEBUG
  // trecho de código compilado se DEBUG estiver definido
#endif
```

Tipos de dados abstratos (TDA)

A linguagem C permite a declaração de TDA - “tipos de dados abstratos ou definidos pelo usuário”, ou seja, a definição de novos tipos de dados. De forma geral, a construção typedef permite que isso seja feito. Uma das formas mais utilizadas de TDA é a utilização de tipos estruturados, como:

```
typedef struct {
   Task taskFunction;
   int   scheduleInterval;
   int   lastActivation;
} TaskControlBlock;
```

```
static TaskControlBlock tsk;
static TaskControlBlock tsk_tasks[NUM_TASKS];
```

O exemplo mostra que foi criado o tipo `TaskControlBlock`, composto por diversos campos. Depois, foram declaradas 2 variáveis: uma escalar (`tsk`) e um vetor (`tsk_tasks`). A utilização dá-se de acordo com a notação:

```
tsk.lastActivation = 0;
tsk_tasks[3].scheduleInterval += 3;
```

Notar a semelhança (e as diferenças!) com orientação a objetos.

Function pointer

Ponteiros (*pointers*) normalmente apontam para variáveis. Mas podem também apontar para funções (*function*) e possibilitar a sua ativação. *Function pointers* são frequentemente utilizados em software de infra-estrutura escritos em linguagem C, para que se implemente o conceito de *callback*. Este é o caso da implementação do *multitasking*: cada *task* é implementado em uma função e, quando o controle decide que é hora de ativar o *task*, aciona a função (ou seja, realiza o *callback*).

A utilização de *function pointers*, de forma geral, implica em:

```
// 1. definir um TAD para cada tipo de função
   Por exemplo, Task é o nome do tipo de dado de
   uma funcao void

typedef void (*Task)(void);

// 2. definir uma função
   (no exemplo, void)

void function pid(void) {
   // código da função
}

// 3. armazenar o endereço da função

Task tsk;
tsk = &pid; // o operador & permite obter o endereço

// 4. acionar a função cujo endereço está na variável tsk

     tsk();
```

Notar que, no multitasking, esta estrutura é utilizada para permitir que o mesmo código possa armazenar o endereço da função e acioná-la a intervalos regulares de tempo.

Declaração de variáveis que são alteradas na rotina de interrupção

No compilador XC8 todas as variáveis globais alteradas na rotina de interrupção devem declaradas como volatile, como por exemplo:

```
volatile long taskInterval = 0;
```

`volatile` significa que não existe garantias de que a variável vai manter o seu valor entre acessos. Isso ocorre pois se uma variável tem o seu valor alternado numa função de interrupção, do ponto de vista da função que utiliza o seu valor, ele pode variar mesmo que não seja feita uma atribuição de valor nesta função ou em qualquer função chamada por ela. `volatile` indica para o compilador que esta variável dever ser armazenada de uma maneira especifica para poder funcionar assim.

Visão geral da implementação

A implementação foi realizada através dos seguintes módulos:

- **`position_controller_main.c e .h`**, contendo basicamente:

  `isr()` : função de interrupção, acionada pelo Timer0 a 1 kHz. Esta função atualiza o tempo decorrido, para acionamento das tarefas do multitasking. Além disso, a interrupção por mudança no Port B também é tratada nesta função para a atualização do valor da posição através do tratamento dos sinais do encoder.

  `main()`: responsável por realizar a inicialização do software e ficar num loop infinito, executando os tasks.

- **`tasks.c`**: implementação dos tasks. Foram implementados 3 tasks: para LED, para comunicação e para o controle PID
- **`protocol.c e .h`**: funções associadas ao protocolo de comunicação com o Raspberry PI
- **`pid.c e .h`**: funções associadas ao controlador PID
- **`position_sensor.h`**: funções associadas ao encoder
- **`motor.h`**: funções associadas ao motor

  O acionamento do motor é realizado através de 2 PWMs, que determinam a direção e a velocidade de rotação do motor.

As implementações do `position_sensor.h` e do `motor.h` são muito semelhantes às que os alunos devem desenvolver no laboratório de PMR3406 por isso são fornecidas bibliotecas já compiladas no lugar dos arquivos fonte.

Testes iniciais

No arquivo `position_controller_main.c` são fornecidas 3 implementações para o `main()`:

- A utilizada no projeto final conforme descrito na seção "Multitasking" a seguir

- Para testar o encoder, inicia acionando o motor e faz a leitura do enconder continuamente enviando o valor atual pelo canal serial
- Para testar o protocolo de comunicação, recebe os comandos, conforme descrito na seção "Comunicando com o Raspberry Pi" a seguir, pelo serial e envia pelo serial os valores recebidos para o tipo de comando e para o parâmetro em decinal. Valores internos do protocolo podem ser enviados pelo serial se estiver definida a constante DEBUG na primeira linha do arquivo `protocol.c`

Cada uma das implementações pode ser comentada e o código re-compilado para os testes. Somente uma das implementações deve estar descomentada no momento da compilação. Ao baixar o projeto a implementação multitasking estará descomentada.

Interrupção do Timer 0

Para os testes a interrupção do Timer 0 possui um trecho de código que alterna um LED entre aceso e apagado a cada 500 ms e serve para verificar que a interrupção está funcionando. Este trecho de código deve ser comentado para a versão Multitasking do `main()` para não haver conflito com o task `led_task()` que está configurado para alternar o mesnmo LED a cada 2000 ms.

Interrupção na mudança (I-O-C)

A função `void pos_UpdatePosition(char port, char init, char dir)` do `position_sensor.c` é chamada na função de tratamento de interrupções para Interrupt-On-Change (I-O-C) do Port B para atualizar a posição do motor com base na mudança de estados do encoder. São passados como parâmetros: (i) valor lido do Port B; (ii) bit inicial do conjunto de 2 bits do Port B que correspondem ao estado do encoder; (iii) direção da contagem (0 ou 1), se incrementa ou decrementa para rotação no sentido horário (deve se ajudado de acordo com a montagem do motor).

Multitasking

O Timer 0 causa uma interrupção à taxa de 1kHz (1 ms), que corresponde ao timeslice que foi escolhido arbitrariamente para este projeto. Pode ser ajustado alterando-se a configuração do Timer 0 no código.

Os tasks são criados na função `initTasks()`, que realiza chamadas da função `createTask()`, à qual são passados os parâmetros: (i) id do task; (ii) ponteiro para a função que implementa o task; (iii) intervalo de scheduling.

O programa principal, `main()`, aciona a função `executeTasks()`, que determina quando cada task deve ser executado.

Notar que esta infra-estrutura é totalmente re-utilizável em outros projetos.

Para o controle de posição foram implementados 3 tasks:

`void pid_task(void)`: realiza o controle PID

`void protocol_task(void)`: realiza a comunicação serial com o Raspberry PI

`void led_task(void)`: pisca o led, servindo como heartbeat para saber-se que o software está sendo executado normalmente.

O intervalo de scheduling de cada task, em milisegundos, é determinado por:

```
#define PID_INTERVAL 100
#define PROTOCOL_INTERVAL 1
#define LED_INTERVAL 2000
```

Sugere-se manter o PROCOTOL_INTERVAL em 1 ms, já que a taxa de comunicação entre o Raspberry PI e o PID está fixada em 115200 bps. O PID_INTERVAL não deve ser menor do que 5 já que este foi o tempo determinado experimentalmente para a execução do controle PID (embora sem windup).

Comunicação com Raspberry PI

A comunicação com o Raspberry PI, como implementada, baseia-se num protocolo master-slave simplificado: há apenas o comando vindo do master Raspberry PI, não havendo resposta do slave PIC.

Os comandos têm a seguinte estrutura:

`<SOT><ADDRESS><COMMAND><VALUE><EOT>`

Onde:

`<SOT>` :== início de comando, sendo utilizado o caracter `':'` como default, pode ser re-definido no código

`<ADDRESS>` :== 1 caracter definido pela função `pro_setMyId(<ADDRESS>)`; pode ser qualquer caracter menos os definidos para `<SOT>` e `<EOT>`, configurado inicialmente como `'a'` no código

`<COMMAND>` :== caracter, podendo ser `'p'` para posição, `'g'` para ganho proporcional, `'i'` para ganho integral, `'d'` para ganho derivativo e `'h'` para home.

`<VALUE>` :== valor decimal, sempre contendo pelo menos uma casa inteira e uma casa decimal com no máximo 7 dígitos entre a parte decimal e a inteira

`<EOT>` :== terminador, sendo utilizado o caracter `';'` como default, pode ser re-definido no código

Assim, por exemplo, considerando que o endereço do PIC é `'a'` e para posicionar-se o motor em 30 graus, utiliza-se o comando

```
:ap30.0;
```

Para determinar-se o ganho proporcional, utiliza-se o comando

```
:ag1.2;
```

O endereço do PIC na rede é determinado pela função `pro_setMyId()`. No código enviado, é usado o endereço `'a'`.

Notar que pode-se ligar diretamente o PIC a um PC, como feito no laboratório de microprocessadores e, através de uma interface USB - serial (5V), enviar os comandos acima para acionar os motores. Este procedimento pode ser útil para testes e ajuste dos ganhos do controlador.

IMPORTANTE

Ao enviar comandos a partir do Raspberry PI para o PIC, imponha um espaçamento de pelo menos 1 ms entre cada caracter. Isso é necessário para compatibilidade com o intervalo de scheduling do protocolo no PIC, determinado por:

```
#define PROTOCOL_INTERVAL 1
```

PID

O controlador PID, embora apenas com o controle proporcional, foi codificado como segue:

```
void pid_pid(void) {

#ifdef DEBUG
  RA4 = ~RA4; // usado para saber quanto tempo o controle demora, ver pino
6 do PIC no osciloscópio
#endif
  currentPosition = (_PID_MATH)pos_getCurrentPosition();
  error = setPoint - currentPosition/5;
  activation = kProportional * error;

  excitation = pid_scaleExcitation(activation);
  di();
  mot_setExcitation(excitation);
  ei();
#ifdef DEBUG
  RA4 = ~RA4;
#endif
} // pid_pid
```

Notar que:

- O pino RA4 inverte a sua saída no início da função e inverte novamente no final. Esta é a forma de, olhando no osciloscópio, determinar o tempo de execução da função.
- Para que o pino RA4 funcione conforme descrito no item anterior, a constante DEBUG deve ser definida na primeira linha do arquivo `pid.c`
- As interrupções são desabilitadas antes da chamada de `mot_setExcitation()`, porque é uma região crítica e não pode ser interrompida enquanto não estiver completada. **Não alterar esta codificação**.
- O encoder gera 1852 pulsos por rotação do eixo de saída do redutor. A conversão para graus por 1852/360 é aproximada para 5, como está em currentPosition/5.

- O motor utilizado para testar o código não era acionado a menos que fosse excitado com um *duty cycle* de 15%. Por este motivo, a função `pid_scaleExcitation()` utiliza 150 como valor mínimo de excitação. Espera-se que os demais motores apresentem comportamento semelhante mas pode ser necessário alterar este valor.

Para testes iniciais com o robô do laboratório de microprocessadores (PMR3406) é fornecida uma função que gera sinais de PWM e direção (PWM1 e DIR1) compatíveis com o robô. A chama da da função é `mot_setExcitationRobot()` que pode simplesmente substituir a chamada `mot_setExcitation()` no arquivo `pid.c`. Deve-se notar, contudo, que os parâmetros do motor e encoder do robô diferem dos do PI-7 de maneira que o ângulo deslocado pelo eixo do motor não corresponderá ao ângulo do parâmetro do comando.

A função `mot_setExcitation()` também funciona com o robô de PMR3406, mas devido à diferença dos sinais de acionamento dos motores para o PI-7, o que acontece é o seguinte:

- O comando com ângulo positivo causa o movimento do motor esquerdo do robô até o "ângulo" definido como parâmetro do comando.
- O comando com ângulo negativo causa o movimento contínuo do motor direito do robô. Para parar o motor direito pode-se mover a roda esquerda com a mão para a direção contraria ao movimento do motor direito até atingir o "ângulo" definido como parâmetro do comando.

Isso ocorre pois no PI-7 o driver recebe os sinais PWM1 e PWM2 para acionamento de um motor, enquanto que no robô de PMR3406 os sinais de acionamento do motor são PWM1 e DIR1 para a roda esquerda e PWM2 e DIR2 para a roda direita. Além disso, no PI-7 somente são usados os sinais de encoder equivalentes aos da roda esquerda do robô (ENC_A1 e ENC_B1), por isso somente a roda esquerda responde corretamente ao comando de posição.