Relatório de Projeto

Sistemas Computacionais
Embebidos
1.º Sem. 2019/20

Weather Station

Pedro Moreira
n.º 85228

Sofia Estrela
n.º 84186

Vasco Candeias
n.º 84196

Grupo 8

Docente: Carlos Almeida

10 de Dezembro de 2019

I. Introdução

Este projeto tem como objectivo desenvolver uma estação meteorológica com o auxilio de um microprocessador PIC16F18875, para o qual será desenvolvido um programa capaz de comunicar com a interface do utilizador, desenvolvido num *embedded Configurable operating system* (eCos).

Para utilizar o código desenvolvido, deverão ser descarregados os ficheiros submetidos:

`pic.hex`;
`ecos.bin`.

Os ficheiros fonte, tanto da PIC como do eCos, estão incluidos no ficheiro `.zip` também submetido.

De seguida, analisa-se a arquitetura do projeto e discute-se o seu funcionamento na secção II. Por fim, analisam-se as estruturas utilizadas na secção III.

II. Funcionamento do Programa

A. *Threads e Prioridades*

Quando o programa começa, são lançadas quatro *threads* na função `main()`: `communicationRead` de prioridade declarada 2, `processing` de prioridade declarada 3, `communicationWrite` de prioridade declarada 4 e `user` de prioridade 5.

As prioridades foram escolhidas de forma a proporcionar o melhor funcionamento do programa. Os valores de prioridades que foram declarados na função `cyg_thread_create()` são inversos à prioridade da *thread*, ou seja, a *thread* de maior importância é a de leitura de valores da PIC, isto porque é extremamente importante ir lidando com os valores que esta envia e não os deixar acumular. A frequência de mensagens que vêm da PIC pode ser muito superior às que vêm do *user* e a sua importância maior para o funcionamento correto do programa. Entre as tarefas de envio de informação à PIC e processamento da informação, considerou-se mais importante que o processamento fosse feito assim que possível. Já a *thread* do *user* é considerada a de menor importância visto que não tem papel no funcionamento correto dos mecanismos do programa e o seu atraso não põe em perigo a integridade do programa. De facto, se estes atrasos forem da ordem dos segundos, não serão sequer preocupantes para a experiência do utilizador.

B. `communicationRead`

Thread responsável por ler e interpretar todas as mensagens enviadas pela PIC ao eCos. Esta *thread* fica bloqueada numa função de leitura de mensagens da PIC pelo *serialIO*, até que esta envie um indicativo de início de mensagem (`SOM` – *start of message*). A leitura de dados é sempre feita *byte* a *byte* e até que seja lido um `SOM`, todos os *bytes* serão descartados. Caso uma mensagem recebida não seja terminada por um `EOM` (*end of message*) ou contenha um formato de erro, esta será interpretada como inválida e o erro será reportado à thread `user`. Sendo esta a *thread* que recebe as mensagens da PIC, será também a *thread* que lida com as notificações de memória cheia. Aquando da receção de uma notificação deste tipo, é ativado um alarme que, periodicamente, se encarregará de acionar uma *event flag*, para que a *thread* `processing` inicie um pedido de transferência de registos.

C. `processing`

Esta *thread* tem três funções: pedir à PIC os registos periódicos, lidar com os *requests* do utilizador e processamento dos novos registos que chegam da PIC. A arquitetura desta *task* pode ser analisada abaixo:

Algoritmo 1: `processing()`

```
initialization;
while not exit do
    flag_wait(0x07,'OR');
    if 1º evento then //novo registo
        flag_maskbit(1);
        process();
    if 2º evento then //alarme periodico
        flag_maskbit(2);
        mbox_put(toWritePic, TRC_10);
    if 3º evento then //user request
        flag_maskbit(3);
        anwser_request();
```

É utilizada uma *event flag* para simular um `select()`. Os *bits* desta *flag* são levantados durante os acontecimentos descritos nos comentários. A função `process()` analisa os registos desde o valor local de `iread` até `iwrite` e verifica se os valores ultrapassam os limites estabelecidos. Os valores iniciais destes limites são 25 para a temperatura e 2 para a luminosidade. A *mailbox* utilizada no segundo evento é na verdade o canal de comunicação entre o *user* e o a comunicação com a PIC, mas devido à sequência em que as mensagens são tratadas, bem como o facto de a mensagem pedida ser equivalente a um pedido de `trc 10` feito pelo *user*, não existe problema de cruzamento de mensagens (a resposta enviada à *thread* do utilizador será ignorada).

D. `communicationWrite`

Thread responsável por todos os envios de mensagens do eCos para a PIC. É uma das *threads* mais simples, em que apenas é necessário encaminhar os pedidos dos utilizadores – já filtrados na *thread* do *user* – para a PIC. Na sua função principal `write_pic()`, no ficheiro `communication.c`, segue a estrutura do Algoritmo 2.

A função de `send_msg()` é também muito simples. Limita-se a juntar o `SOM` e o `EOM` ao comando do primeiro parâmetro e aos argumentos enviados dentro do `request` – cujo tamanho é passado como segundo parâmetro – enviado a mensagem final com recurso a um `cyg_io_write()`.

Todas as mensagens, depois de processadas, são libertadas da memória, tendo sido alocadas na *thread* de origem.

Algoritmo 2: write_pic()

```
initialization;
while 1 do
    request = mbox_get(fromUser);
    switch request->command do
        case CODE_RC do
            send_msg(RCLK, 1);
        case CODE_SC do
            send_msg(SCLK, 4);
        ...
    free(request);
```

E. `user`

Esta *thread* é responsável por ler o *input* do utilizador no ecrã do eCos, filtrá-lo, verificar a sua integridade e valores dos parâmetros. Deve de seguida avisar a *thread* responsável pelo *request* para dar uma resposta. No caso de acessos à memória local, estes são feitos diretamente – com as devidas proteções. Esta *thread* utiliza a estrutura do ficheiro de código exemplo `cmd.c` – a função `monitor()` – pelo que a sua estrutura não será explorada. O esqueleto desta *thread* encontra-se no `user.c` e os comandos no `user_commands.c`.

III. Estruturas e Variáveis Relevantes

A. Mutexes

Foram utilizados três `mutexs` no desenvolvimento do eCos, descritos de seguida. Estes são inicializados antes das *threads* serem lançadas, na função `main()`.

`stdin_mutex`:
Este `mutex` protege as chamadas ao `stdin`, ou seja, todos os `prints` no ecrã para o utilizador estão dentro de um *lock* deste `mutex`. As *tasks* que têm acesso a este `mutex` são a do utilizador, a de comunicação (no caso da *thread* que recebe informação da PIC, para o caso de receber um aviso de memória) e ainda a de processamento (para imprimir no ecrã avisos caso os registos estejam acima dos limites de processamento).

`local_mutex`:
Este `mutex` protege os acessos à memória local, ou seja, o *ring buffer* do eCos. Todas as leituras e escritas de valores nesta estrutura estão dentro de um *lock* a este `mutex`, *lock* este que é sempre suficientemente cumprido para que as alterações feitas lá dentro apenas sejam *unlocked* em novo estado estável, para não por em causa a integridade e validade dos dados. As *tasks* que utilizam este `mutex` são a de processamento para a análise dos registos locais, a de comunicação (apenas a *thread* de leitura da PIC) quando recebe registos da PIC e ainda o utilizador quando pede `irl`, `lr` ou `dr`.

Note-se que por vezes é necessário utilizar ambos os *mutexs* acima explicados. Para tal, o bloqueio é sempre feito primeiro ao `local_mutex` e só depois ao `stdin_mutex`. Esta convenção evita *deadlocks*.

`alarm_mutex`:
Uma vez que o período do alarme pode ser lido/alterado por duas *tasks* em simultâneo, utilizou-se um *mutex* para proteger os acessos a esta região de memória.

B. Mailboxes

`user_com_channel` e `com_user_channel`: canal de comunicação entre o utilizador e a comunicação. O primeiro começa na *thread* do `user` e acaba na `communicationWrite`. O segundo começa da *thread* de `communicationRead` e acaba na *thread* do `user`.
`user_pro_channel` e `pro_user_channel`: Estes dois canais asseguram a comunicação bilateral entre a *thread* de processamento e do utilizador.

C. Memória local – `struct local memory`

Esta estrutura contém três inteiros de oito *bits*: `nr`, `iread` e `iwrite`. Tem ainda um *array* de `buffer` com `NRBUF` de comprimento. Cada entrada do tipo `buffer` tem ainda cinco inteiros de oito *bits*: `hour`, `minute`, `second`, `temperature` e `luminosity`. Esta estrutura é declarada uma vez no programa, e a alteração de valores é feita ao abrigo do `local_mutex`, em funções próprias – presentes no ficheiro `structure.c` – para manter e regular um *circular buffer* no *array* do tipo `buffer`.

D. Request

Esta estrutura contém um inteiro (de 8 *bits*) chamado `cmd` e um *array* de 10 posições destes inteiros chamado `arg`. Esta estrutura é por onde são enviados os *requests* e *replies* do/para o utilizador. O inteiro `cmd` contém um indentficador do comando pedido pelo utilizador e o `arg` contém os argumentos fornecidos ou as respostas dadas pelas outras *threads*.

E. Acknowledge

Esta estrutura contém apenas um *boolean* `error`. Se este estiver a `true` significa que algo se passou no processo e este não foi concluído como previsto. Caso contrário, assume-se que tudo correu bem. Esta mensagem é enviada a resposta de alguns dos requerimentos do utilizador como alterações de valores do sistema.