Sistemes operatius encastats

José María Gómez Cama

PID_00177253

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índex

Introducció

Tal com s'ha indicat anteriorment, l'objectiu fonamental d'un sistema encastat (SE) és aconseguir dur a terme una sèrie de tasques, d'acord amb uns requeriments, de manera eficient i optimitzant l'ús dels recursos disponibles.

Aconseguir aquest objectiu de manera genèrica no és evident, a causa fonamentalment de la innombrable quantitat de plataformes, arquitectures i aplicacions possibles. Això fa que sigui virtualment impossible donar una resposta òptima per a tots els possibles escenaris.

No obstant això, la introducció dels sistemes operatius (SO) permet gestionar els diferents elements d'un sistema basat en processador d'una manera eficient. Alhora, presenta al programador/usuari una màquina virtual que és equivalent, independent de la plataforma. Això en simplifica de manera notable l'ús i la programació. El preu són els recursos de memòria i temps que requereix el mateix sistema operatiu.

Partint del que s'acaba de dir, sembla lògic pensar que l'ús d'un SO podria ser la manera més adequada d'aconseguir que un SE compleixi el seu objectiu fonamental. No obstant això, la realitat actual mostra que això no és així, ja que el nombre de sistemes encastats que inclouen un SO és molt reduït. En general, una programació *ad hoc* sol ser la solució preferida. Els motius poden ser molts i depenen del punt de vista. Per posar-ne exemples, aquests en podrien ser alguns si tenim com a objectiu l'eficiència:

- Els recursos del SE són molt reduïts i la inclusió d'un SO els minvaria de manera notable, per la qual cosa es requeriria una nova plataforma amb més prestacions.

- Les tasques que cal dur a terme són molt senzilles i estan molt ben delimitades, per la qual cosa afegir un SO amb prou feines milloraria els resultats.

- Les tasques que cal dur a terme porten al límit una plataforma d'altes prestacions, per la qual cosa qualsevol sobrecost a causa del SO impediria aconseguir els requeriments.

D'altra banda, des del punt de vista de la màquina virtual en podríem tenir els següents:

- La plataforma és tan senzilla que la màquina virtual que proporciona el SO és més complexa que l'original.

- No hi ha una adaptació del SO a la plataforma que s'utilitzarà, amb la qual cosa el sobrecost a causa de la complexitat de l'arquitectura de la plataforma queda compensat per la possible dificultat de l'adaptació del SO.

En aquests casos, és evident que el SO perd enfront d'una programació *ad hoc*, la qual cosa no significa que no s'utilitzin elements d'un SO per a ajudar en la programació. A tall d'exemple –si l'aplicació és senzilla i, per tant, el nombre de tasques que cal dur a terme no és alt, l'arquitectura del sistema encastat és reduïda, i els perifèrics no tenen una gran complexitat (ADC, GPIO, USART, I2C...)–, és molt comú fer una programació basada en un gestor de tasques en bucle.

En resum, no hi ha una regla d'or a l'hora de treballar amb SO sobre un SE. En general, el seu ús dependrà de:

- l'aplicació,
- la plataforma,
- l'experiència del programador.

No obstant això, és important destacar que les noves plataformes proporcionen cada vegada més recursos, per la qual cosa és previsible que de la mateixa manera que s'ha fet el salt de la programació en assemblador, cosa que ha donat pas a uns altres de més nivell com el C, en un futur proper es vegi com un pas natural la programació de SE basada en SO.

Objectius

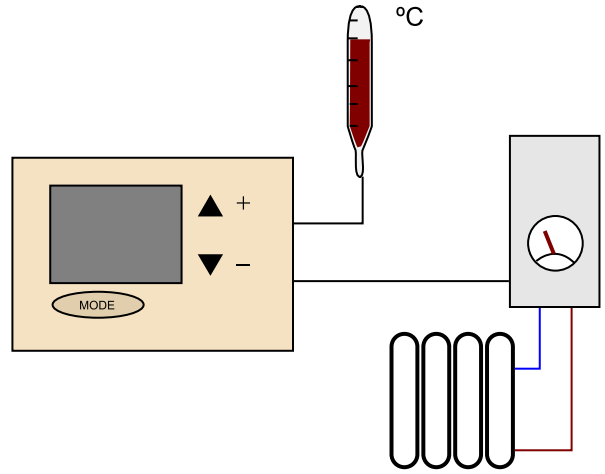
L'estudi d'aquest mòdul didàctic us permetrà assolir els objectius següents:

1. Conèixer els elements bàsics d'un sistema operatiu per a sistemes encastats.
2. Dominar les metodologies més habituals per a la gestió de processos en sistemes encastats.
3. Entendre el concepte de *controlador de perifèrics*.
4. Saber desenvolupar un controlador de perifèrics a partir del seu full de característiques.
5. Conèixer alguns dels sistemes operatius de propòsit específic.

1. Un cas bàsic introductori

Suposem que ens demanen dissenyar un sistema encastat que controli la temperatura d'una habitació mitjançant una calefacció, i que també proporcioni informació a l'usuari d'aquesta temperatura. En la figura següent se'n mostra un exemple:

Esquema d'un sistema de calefacció



Les tasques que cal dur a terme són:

- Prendre la temperatura de l'habitació.
- Gestionar els botons de temperatura de consigna.
- Gestionar el botó de mode de visualització (temp. real o temp. consigna).
- Mostrar la temperatura a la pantalla.
- Actuar el relé per a encendre/apagar la caldera.

D'acord amb aquestes tasques, el sistema constarà dels elements següents:

- Sensor de temperatura amb una resolució d'una desena de grau.
- Pantalla per a mostrar la temperatura actual i la de consigna.
- Botonera:
  - Dos botons per a pujar o baixar la temperatura de consigna.
  - Un botó per a canviar de visualització de temperatura actual a temperatura de consigna.

- Relé per a controlar l'engegada de la calefacció.
- Microcontrolador.

Tenint en compte els temps de resposta d'un sistema de calefacció, que poden anar de minuts a hores, és evident que no es necessita un sistema especialment ràpid. Per aquest motiu, els processos es poden dur a terme de manera seqüencial.

D'acord amb aquests elements, podem determinar que, pel que fa als perifèrics del nostre microcontrolador, haurem d'utilitzar:

- **Sensor de temperatura**: convertidor analògic digital connectat a sensor de temperatura.
- **Pantalla**: interfície sèrie.
- **Botonera**: pins d'entrada de propòsit general.
- **Relé**: pin de sortida de propòsit general.

Amb tot això, podem veure que la complexitat de l'aplicació és reduïda, la qual cosa en permet fer una implementació de manera *ad hoc*. En aquest cas, els passos que cal fer serien:

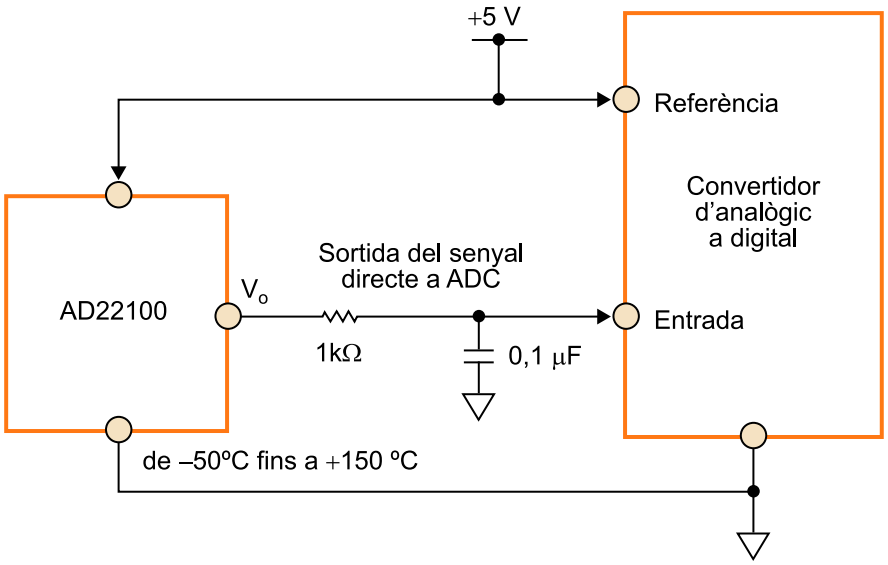
- Definició dels tipus i estructures necessaris.
- Inicialitzar el sistema:
  - Configurar el rellotge.
  - Configurar els ports d'entrada i sortida.
  - Inicialitzar les variables principals.
- Bucle principal:
  - Consultar les entrades.
  - Processar la informació.
  - Generar les sortides.

La implementació dels diferents elements es presenta a continuació, començant pel sensor de temperatura, que requereix un processament específic.

1.1. Sensor de temperatura

Per al cas del sensor de temperatura, podem usar el circuit d'Analog Devices AD22100, que ens proporciona una variació d'una tensió en funció de la temperatura. L'esquema es mostra en aquest esquema:

Esquema del sensor de temperatura



Amb aquest circuit, es té una variació de 22,5m V/°C. Suposant que l'ADC disponible és de 12 bits, podem obtenir els valors per a les diferents temperatures a partir de les equacions següents:

$$\begin{cases} V[V] = \frac{22{,}5\,mV}{^\circ C} T[^\circ C] + 1{,}375\,V \\ V[ADC] = (2^{12\,\text{bits}} - 1) \cdot \frac{V[V]}{5\,V} \end{cases} \Rightarrow \text{Valor}[ADC] = \lfloor 18{,}43 \cdot T[^\circ C] + 1.126{,}125 \rfloor$$

en què $[x]$ és el valor sencer més proper a 0 respecte de x (equivalent a la funció *floor* en C). Així, doncs, els voltatges que obtindrem per a les diferents temperatures seran:

Temperatura (C)	Voltatge (V)	Valor (ADC)
–50	0,25	204
–25	0,8125	665
0	1,375	1.126
25	1,9375	1.586
50	2,5	2.047
100	3,625	2.968
150	4,75	3.890

Per a fer el pas invers de manera senzilla en un microcontrolador, en principi, ens interessaria obtenir una temperatura amb precisió de desena de graus. Això ho podríem fer utilitzant variables en coma flotant, però en un microcontrolador impliquen un consum més gran de recursos, per la qual cosa utilitzarem en el seu lloc l'escalat de la variable de temperatura, de manera que puguem treballar amb variables senceres.

Tenint en compte que l'aplicació és un control de temperatura d'una habitació, la resolució d'una desena de grau es pot considerar més que suficient. Per tant, el que podem fer és treballar amb unitats de desena de grau centígrad. Així, doncs, la temperatura d'ebullició de l'aigua és 100,0 °C o 1.000 d°C (decigraus Celsius).

Per a això, el procés que seguirem serà el següent:

1) Passar el valor de l'ADC a un enter amb signe de 32 bits (valor$[ADC]_{32}$).

2) Escalar el valor rebut per 32, o, el que és equivalent, desplaçar a l'esquerra 5 posicionis valor$(ADC)_{32}$.

3) Seguidament, dividir el valor obtingut per 59. Aquest valor l'obtenim en multiplicar per 32 els 18,43 de la fórmula del valor(ADC), i dividir-la per 10 perquè doni el resultat en d°C.

4) A aquest valor, cal restar-hi 610, perquè els 0 graus quedin en el valor sencer 0.

En la taula següent, podem veure els valors obtinguts:

Conversió del senyal del sensor o temperatura

Temperatura(°C)	Valor (ADC)	Escalat	Divisió	Temperatura (d°C)
–25	665	21.280	360	–250
0	1.126	36.032	610	0
25	1.586	50.752	860	250
50	2.047	65.504	1.110	500

El resultat final el podem guardar en un enter de 16 bits amb signe, que ens permet anar dels –32.768 als 32.767 d°C o, el que és equivalent: dels –3.276,8 als 3.276,7 °C.

1.2. Definicions

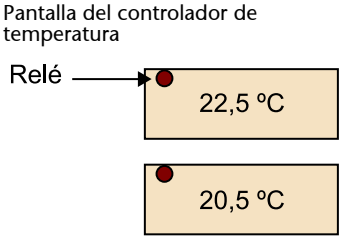
Per a dur a terme la nostra aplicació, un dels punts importants és establir la manera com guardarem les dades que no són nombres sencers.

D'altra banda, per a l'estat dels botons farem ús d'un booleà que ens indicarà si el botó està premut (true) o no (false). Els estats dels tres botons, els agruparem en una estructura, perquè sigui més fàcil d'utilitzar.

En el cas del relé, també farem ús d'un booleà, que ens indicarà si es troba tancat o no. Per al tipus booleà, farem ús de l'arxiu de capçalera stdbool.h, el qual defineix el tipus bool i els valors true i false. L'estructura seria:

```
struct buttons_t {
      bool up;
      bool down;
      bool mode;
};
```

Els estats de la pantalla es mostren en la figura següent:



Per a l'estat de la pantalla, farem ús d'una enumeració:

- INIT: iniciant.
- SENSOR: temperatura del sensor.
- TARGET: temperatura de consigna.

Per a la temperatura, com hem dit, utilitzarem un enter de 16 bits amb signe. En aquest cas, ens trobem amb la dificultat que en llenguatge C no hi ha *a priori* el tipus 16 bits amb signe. Cada compilador pot definir la grandària del tipus sencer. Així, doncs, un sencer (int) en una arquitectura és de 16 bits, i en una altra, de 32. Per a solucionar-ho, podem fer ús de l'arxiu de capçalera stdint.h, el qual defineix els tipus sencers bàsics en funció del seu signe i grandària. Així, per exemple, tenim un enter de 8 bits sense signe (uint8_t), o un enter de 16 amb signe (int16_t).

Per conveniència, integrarem l'estat de la pantalla i els valors de la temperatura en una estructura:

```
struct state_t {
      enum {
            INIT, SENSOR, TARGET
      } screen;
      int16_t targetTemp;
      int16_t sensorTemp;
      bool relay;
};
```

A continuació, podem inicialitzar els components.

1.3. Etapa d'inicialització

Per a dur a terme la inicialització, es pot fer en un únic mètode, o bé es pot fer d'una manera més estructurada, separant-la en funció del tipus de dispositiu. En el nostre cas, escollirem la segona opció, perquè és la que permet una transferència més fàcil entre diferents arquitectures de microcontrolador.

Com s'ha indicat en la secció anterior, el primer element que cal inicialitzar és el rellotge. A continuació, s'indica el codi d'inicialització per al microcontrolador escollit:

```
void clockInit(void)
};
```

El pas següent és configurar els ports. En general, els microcontroladors se solen inicialitzar amb tots els pins en mode entrada i per a propòsit general. Aquesta configuració minimitza els possibles problemes per al microcontrolador. Per contra, si hi ha elements externs al microcontrolador, poden tenir un comportament indeterminat. Això es pot solucionar en alguns casos incloent-hi resistències de *pull-up* o *pull-down*, la qual cosa, al seu torn, incrementa el consum dinàmic. Però és evident que és important inicialitzar des del primer moment els ports de sortida, seguint una seqüència que eviti al màxim els possibles problemes.

En el cas d'aquesta aplicació, la sortida que pot tenir un efecte més indesitjat és la del relé. Per aquest motiu, sembla oportú configurar aquesta primer. El següent element de sortida seria la pantalla, encara que aquesta és menys preocupant, en incloure el seu sistema de control propi. Finalment, tenim les entrades dels botons i l'ADC.

Així, doncs, el primer element que configurarem serà el relé com a sortida digital.

```
void relayInit(struct state_t *state) {
      state->relay = false;
      // Relay output initializing code
      ...
}
```

Seguidament, configurarem la pantalla. Per a això durem a terme els passos següents:

- Aquesta començarà a treballar en estat INIT.
- Posarem el valor de la temperatura de consigna en el valor 25 °C (targetTemp = 250 d°C).

- Configurarem el mòdul SPI per poder-nos comunicar amb l'LCD.
- Iniciarem els *buffers* de comunicació amb l'LCD.
- Enviarem els missatges de configuració de l'LCD.
- Canviem l'estat de la pantalla a TARGET.
- Presentarem a la pantalla l'estat consigna i la temperatura de consigna.

També haurem de configurar l'estat actual de la pantalla, que serà el d'inicialització.

```
void lcdInit(struct state_t *state) {
       state->screen = INIT;

       // LCD hardware initializing code
       ...
       // End LCD hardware initializing code

       state->targetTemp = 250;
       state->screen = TARGET;

       setLCD(state);
}
```

A continuació, inicialitzarem les tres entrades digitals de la botonera.

```
void buttonsInit(struct buttons_t *buttons) {
      buttons->down = false;
      buttons->mode = false;
      buttons->up = false;
      // Buttons input initializing code
      ...
}
```

Finalment, farem el mateix amb l'ADC, la qual cosa ens permetrà mesurar la temperatura. Per a això, configurarem l'entrada com a analògica i la mesura.

```
void tempInit(struct state_t *state) {
      state->sensorTemp = state->targetTemp;
      // Sensor ADC initializing code
      ...
}
```

Tots aquests mètodes els integrarem en un de sol, per a fer el codi més llegible.

```
void initialize(struct buttons_t *buttons, struct state_t *state) {
       clockInit();
       relayInit(state);
       lcdInit(state);
```

```
        buttonsInit(buttons);
        tempInit(state);
}
```

Una vegada finalitzat, implementem el bucle principal.

1.4. Bucle principal

Una de les principals característiques d'un sistema encastat és que no sol parar mai. Una vegada arrencat, no para, tret que hi hagi algun error en el codi, o un esdeveniment imprevist en el codi que comporti aquesta situació.

Per aquest motiu, el programa d'aquest tipus de sistemes se sol basar en la utilització d'un bucle principal que no finalitza mai i que va fent els diferents passos necessaris. El primer és la lectura de les entrades, en aquest cas dels botons i el sensor de temperatura. Per a això, executarem dues funcions. La primera serà molt senzilla, ja que només requereix consultar un registre:

```
void getButtons(struct buttons_t *buttons) {
        int data = IOPORT;

        if ((data & UP_BUTTON) != 0) buttons->up = true;
        else buttons->up = false;
        if ((data & DOWN_BUTTON) != 0) buttons->down = true;
        else buttons->down = false;
        if ((data & MODE_BUTTON) != 0) buttons->mode = true;
        else buttons->mode = false;
}
```

en què IOPORT és el port al qual estan assignats tots tres botons. En cas que no fos així, s'hauria de modificar el codi per a tenir-lo en compte. D'altra banda, tenim les definicions UP_BUTTON, DOWN_BUTTON i MODE_BUTTON que són les màscares per a cadascun dels botons.

La segona serà una mica més complexa, ja que la mesura de temperatura requereix un petit càlcul per a passar del valor mesurat per l'ADC a graus centígrads. En qualsevol cas, suposem que el valor el llegim del registre corresponent.

```
int16_t getTemp(void) {
        uint16_t adcValue16 = ADCVALUE;
        int32_t adcValue32 = adcValue16;
        int16_t result;
        adcValue32 <<= 5;
        adcValue32 /= 59;
        result = (int16_t) (adcValue32 - 610);

        return result;
```

```
}
```

en què ADCVALUE és el registre que guarda el valor llegit per l'ADC.

En funció dels valors rebuts, haurem de prendre les decisions oportunes. El primer pas que farem serà veure si l'usuari ha premut algun botó i actuar en conseqüència.

El primer pas és establir la precedència dels botons.

1) Si es prem un únic botó, es duu a terme l'acció per a aquest botó.

- **Up:** es passa al mode de pantalla temperatura de consigna i es puja la temperatura una desena de grau.
- **Down:** es passa al mode de pantalla temperatura de consigna i es baixa la temperatura una desena de grau.
- **Mode:** es canvia al mode de pantalla següent.

2) Si es premen dos botons, podem tenir tres casos més:

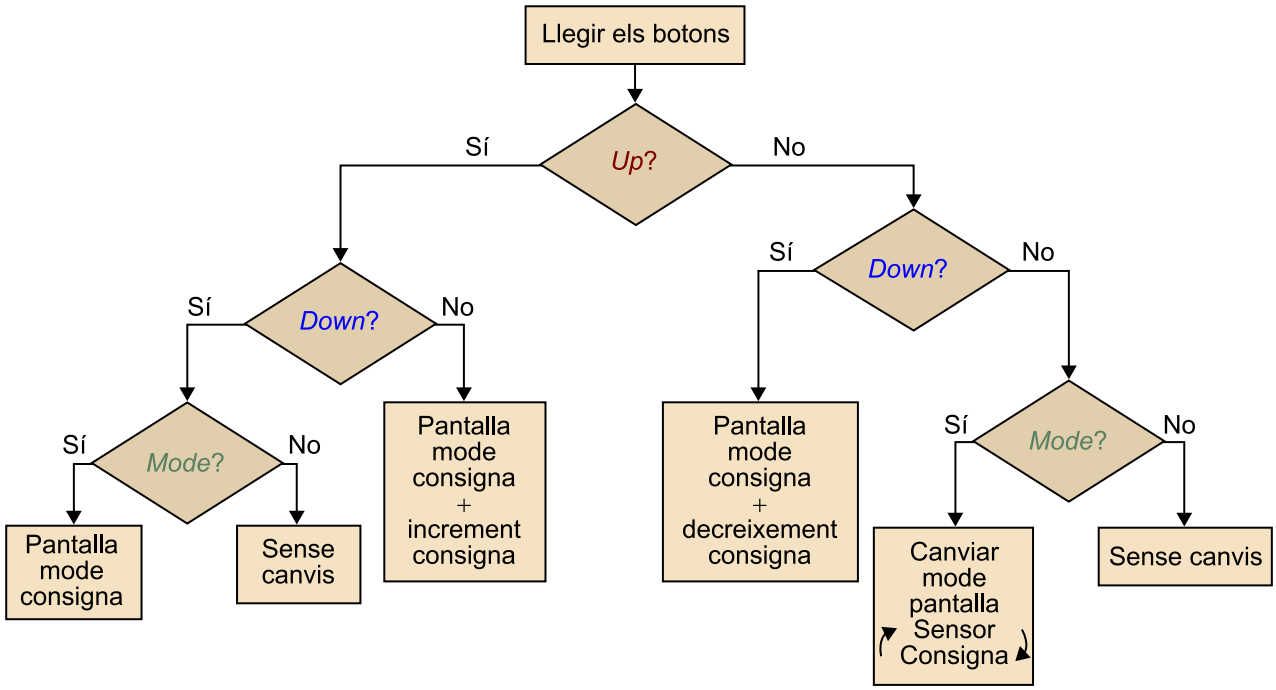
- **Up i Mode:** la mateixa acció que Up sol.
- **Down i Mode:** la mateixa acció que Down sol.
- **Up i Down:** no es fa cap acció.

3) Si es premen els tres alhora, es passa a mode temperatura de consigna i ja està.

Finalment, se saturarà la temperatura màxima a 50 graus (500 d°C) i la mínima, a –25 graus (–250 d°C).

Aquest arbre de decisió es mostra en la imatge següent:

Diagrama de flux del control dels botons



Aquest es pot implementar en un mètode que tingui com a entrada ambdues estructures i modificarà l'estat i el valor de consigna.

```
void buttonsState(struct buttons_t *buttons, struct state_t *state) {
      if (buttons->up == true) {
            if (buttons->down == true) {
                  if (buttons->mode == true) {
                        state->screen = TARGET;
                  }
             } else {
                   state->screen = TARGET;
                   state->targetTemp += 1;
            }
       } else {
                  if (buttons->down == true) {
                        state->screen = TARGET;
                        state->targetTemp -= 1;
                  } else {
                        if (buttons->mode == true) {
                             if (state->screen == TARGET) {
                                   state->screen = SENSOR;
                             } else {
                                  state->screen = TARGET;
                             }
                        }
                  }
         }
```

```
        if (state->targetTemp > 500)
              state->targetTemp = 500;
        else if (state->targetTemp < -250)
              state->targetTemp = -250;
  }
```

D'acord amb el contingut de les temperatures de la variable state, prendrem la decisió referent a l'estat del relé. En principi, el relé estarà en circuit obert.

- Si la temperatura de consigna és més alta que la del sensor en més de cinc desenes de grau, canviarem l'estat del relé a circuit tancat.

- Si la temperatura de consigna és igual que la del sensor, canviarem l'estat del relé a circuit obert.

- En cas contrari, es mantindrà l'estat anterior.

Amb aquest funcionament, tenim una certa histèresi, que ens filtra el possible soroll provinent de l'entrada de l'ADC. Per a això definim prèviament una constant global:

```
  const uint16_t HISTERESYS_TEMP = 5;
```

I seguidament, el mètode associat:

```
  void relayState(struct state_t *state) {
        if (state->targetTemp > (state->sensorTemp + HISTERESYS_TEMP)) {
              state->relay = true;
        } else {
              if (state->targetTemp <= state->sensorTemp) {
                    state->relay = false;
              }
        }
  }
```

Amb el resultat obtingut, s'executa l'acció sobre el relé, fent ús del mètode:

```
  void setRelay(bool close) {
        if (close) IORELAY |= RELAY_PIN;
        else IORELAY &= ~RELAY_PIN;
  }
```

en què IORELAY és el registre del port que controla el relé i RELAY_PIN és la màscara d'aquest pin.

Finalment, quedaria pendent la presentació en pantalla en funció de l'estat, la qual durà a terme les accions següents:

- Mirar la temperatura que cal mostrar en funció del valor de la variable screen (TARGET, SENSOR).
- Convertir la temperatura de binari a decimal.
- Presentar la temperatura associada.

Aquests passos els integrarem en el mètode:

```
void setLCD(struct state_t *state) {
      switch (state->screen) {
      case TARGET:
            printf("Target Temp: %.1f\n", state->targetTemp / 10.0);
            break;
      case SENSOR:
            printf("Sensor Temp: %.1f\n", state->sensorTemp / 10.0);
            break;
      default:
            fprintf(stderr, "Unknown lcd state: %d", state->screen);
}
}
```

Amb tot això, tindrem una aplicació bàsica per a poder controlar la temperatura d'una habitació. El codi complet es mostra a continuació:

```
int main(void) {
      struct buttons_t buttons;
      struct state_t state;

      initialize(&buttons, &state);

      for (;;) {
            // Main loop
            getButtons(&buttons);
            state.sensorTemp = getTemp();
            buttonsState(&buttons, &state);
            relayState(&state);
            setRelay(state.relay);
            setLCD(&state);
      }
      return EXIT_SUCCESS;
}
```

Encara que l'aplicació en si és molt senzilla, la seva anàlisi ens proporciona les bases per a entendre què es requereix per a un sistema operatiu encastat. Si no disposem d'un sistema de desenvolupament basat en microcontrolador, podem fer ús de les funcions indicades en l'apèndix.

2. Elements d'un sistema operatiu

D'acord amb l'aplicació anterior, podem establir quins elements necessitem per a poder dur a terme qualsevol tasca en un sistema encastat. Seguirem una organització diferent de la utilitzada en l'apartat anterior, més semblant a la dels llibres d'aquesta temàtica.

Recordem que els elements que formen part del nucli o *kernel* d'un sistema operatiu són:

- planificador de tasques,
- gestor de tasques,
- controladors de perifèrics,
- *buffers*.

Per tal d'identificar-los, podem analitzar els diferents mètodes i funcions de l'aplicació que són dins del bucle principal:

- getButtons: llegeix les dades dels botons (perifèric).
- getTemp: llegeix dades de temperatura (perifèric).
- buttonsState: processament dels botons (tasca).
- relayState: processament de l'estat del relé (tasca).
- setRelay: modificar la sortida del relé (perifèric).

Tenint en compte l'anterior, podem veure que en el codi anterior apareix un gestor de tasques primitiu, que seria el bucle principal. D'altra banda, tenim els controladors de perifèrics, que serien els mètodes associats.

Veiem, també, que apareix la figura dels *buffers*, que permeten intercanviar la informació entre els diferents mètodes (tasques i controladors). En aquest cas, són les estructures state i buttons les que permeten aquest intercanvi.

No obstant això, el planificador de tasques és inexistent i veurem que només serà necessari en el cas de SO d'altes prestacions.

Analitzem cadascun d'aquests components, a partir d'exemples basats en solucions actuals.

3. El gestor de processos

A diferència d'un ordinador de propòsit general, un sistema encastat està molt focalitzat a donar resposta a una necessitat o un problema determinats. Aquesta necessitat o aquest problema se solen dividir en **tasques**, que són els elements atòmics necessaris per a resoldre els problemes.

En aquest sentit, les tasques que ha de fer estan definides *a priori*, i qualsevol modificació en sol implicar una reprogramació.

> En general, podem considerar una tasca com un element atòmic requerit per a resoldre un problema. Aquesta tasca precisarà una informació d'entrada, que és la que processarà, i d'acord amb aquesta informació donarà una resposta.

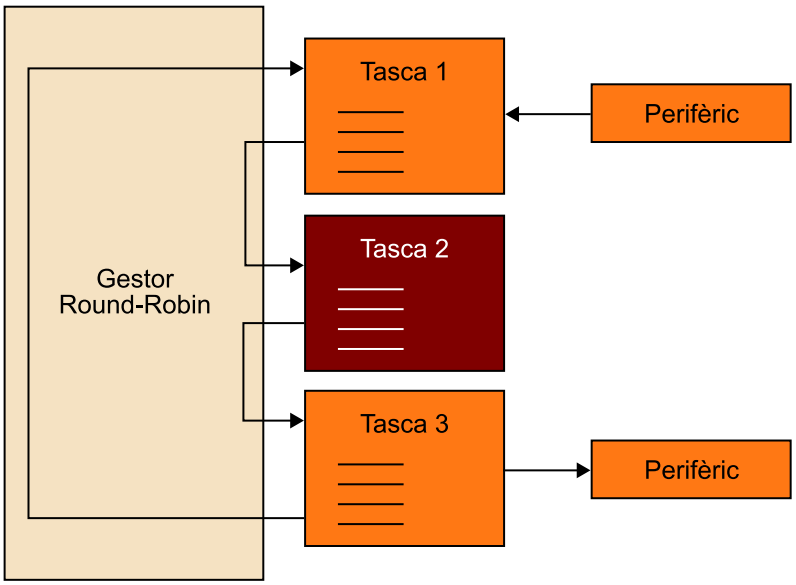
Per a dur a terme aquestes tasques, és necessari que s'executin les instruccions necessàries en el CPU. Per a això, es requereixen, també, recursos de memòria i possiblement accés a informació de determinats perifèrics. Aquests recursos han de ser proporcionats d'alguna manera dins del SO.

En funció dels requeriments temporals d'aquestes tasques, s'incrementarà o reduirà la complexitat del gestor dels processos. Si és molt alta, apareixerà la necessitat del planificador de tasques.

3.1. Sistemes operatius Round-Robin

Aquesta solució és la més bàsica de totes. El seu funcionament és equivalent a la solució del control de caldera: repetició constant d'un conjunt de tasques. Aquestes tasques es troben organitzades en una llista circular. Es comença per la primera i fins que no finalitza no es passa a la següent, tal com es mostra en la figura següent:

Diagrama d'un sistema Round-Robin



Cada tasca se sol implementar en un mètode independent, aquests mètodes agafen entrades de perifèrics o d'altres tasques, les processen, i generen sortides que destinen a perifèrics, o a altres tasques.

Per tal de determinar-ne el punt més crític quant a velocitat, hem d'analitzar les tasques:

- **Mesurar la temperatura de l'habitació**. La tasca no cal fer-la amb gaire freqüència, ja que, com hem dit anteriorment, el temps de variació de la temperatura de l'habitació poden ser minuts. No obstant això, moltes vegades, l'usuari vol comprovar si el sensor funciona apropant-hi una font de calor. Per aquest motiu, és important que el temps de resposta no sigui gaire lent. Considerarem que un segon és un retard raonable.

- **Gestió de botons**. La seva funció és veure l'estat dels botons i, en funció d'aquests, modificar la temperatura de consigna o la visualització de la pantalla. Aquesta tasca està fonamentalment dirigida a permetre la interacció amb l'usuari i, per tant, el temps de resposta ha de ser especialment ràpid (ningú no està disposat a trigar 30 segons a aconseguir que pugi la temperatura uns graus). Per aquest motiu, el temps ha d'estar per sota de la desena de segon. També s'encarrega de comprovar l'estat del botó de canvi de mode de visualització.

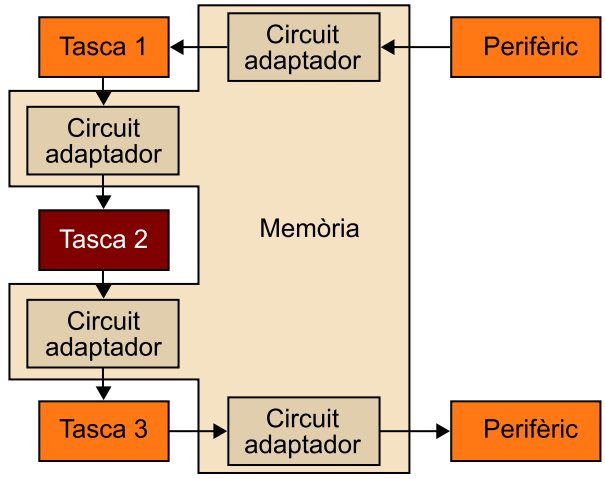
- **Actuar el relé de la caldera**. El seu funcionament es basa a comparar la temperatura de l'habitació i la de consigna. Per tant, haurà d'estar atenta als canvis que es produeixin en les dues tasques anteriors.

- **Mostrar informació en pantalla**. Aquesta tasca s'encarrega de gestionar la pantalla. El seu funcionament depèn de les anteriors, ja que només caldrà fer canvis quan es modifiquin els paràmetres o estats.

Podem observar que la tasca que s'ha de fer amb més freqüència és la de gestió de botons. Com que la més ràpida determina la freqüència del bucle, tot el bucle ha de poder dur a terme les tasques en menys d'una desena de segon.

És evident que les tasques necessiten intercanviar informació (temperatures, estat del relé, estat de la pantalla). Per a això, es creen unes variables en memòria que duen a terme aquesta funció, tal com es mostra en la següent:

Tasques en el bucle intercanviant informació per mitjà de la memòria



Encara que, en general, es desaconsella l'ús de variables, en aquest cas, és l'única opció, ja que en C no és possible intercanviar dades entre funcions que no reben paràmetres.

3.1.1. La implementació

La implementació d'un gestor Round-Robin és relativament senzilla. Tan sols es requereix una cua jeràrquica en què es guarda cadascuna de les referències a les tasques que cal fer. El primer pas és definir el tipus punter a tasca (en el nostre cas, task_t). Per a això, utilitzem la sentència typedef.

Punters a funció

Un punter a funció és equivalent a un punter a variable. Ambdós permeten accedir de manera indirecta al seu contingut. La segona proporciona el valor contingut, mentre que la primera executa el codi associat a aquesta funció.

L'existència dels punters en qüestió, a més, també permet crear llistes com les utilitzades per al gestor.

```
typedef void (*task_t)(void);
```

Tal com podem observar, les tasques no tenen ni entrades ni sortides. Per tant, serà necessari utilitzar un *buffer* en memòria que permeti intercanviar aquesta informació. Aquest serà generalment una variable o estructura global.

D'altra banda, tenim la cua de tasques. Aquesta la definirem com una estructura amb un vector de tasques. A més, hi inclourem una variable que indiqui el nombre de tasques i una segona variable que serà l'índex de la tasca associada.

```
struct {
      int number;
      int pointer;
      task_t queue[NUMBER_TASKS];
} tasks;
```

Seguidament, hem d'iniciar la cua de tasques. Per a això, definim el mètode següent, en què posem totes les entrades al seu valor inicial.

```
void tasksInit(void) {
      int i;

      tasks.number = 0;
      tasks.pointer = 0;

      for (i = 0; i < NUMBER_TASKS; i++){
            tasks.queue[i] = NULL;
}
}
```

Creem, també, el mètode per tal d'afegir tasques noves a la cua, en què s'inclou un control per a no superar la grandària màxima d'aquesta cua.

```
int tasksAdd(task_t task) {
      int pointer = tasks.number;

      if ((pointer+1) >= NUMBER_TASKS) {
            return -1;
      } else {
            tasks.queue[pointer] = task;
            tasks.number++;

            return pointer;
      }
}
```

Finalment, tenim el gestor de tasques, el procés del qual és anar executant les diferents tasques d'una manera seqüencial.

```
void tasksRun(void) {
      int i;
      task_t tp;

      for (;;){
            for (i = 0; i < tasks.number; i++){
                  tp = tasks.queue[i];
                  tp();
            }
      }
}
```

L'intercanvi d'informació es fa mitjançant *buffers* en la memòria. En funcionar únicament una tasca cada vegada, els *buffers* en aquest tipus de gestors poden ser simples variables i estructures. En el nostre cas, crearem les estructures globals state i buttons:

```
struct buttons_t {
      bool up;
      bool down;
      bool mode;
} buttons;

struct state_t {
      enum {
            INIT, SENSOR, TARGET
      } screen;
      int16_t targetTemp;
      int16_t sensorTemp;
      bool relay;
} state;
```

La definició és la mateixa que hem fet en el cas anterior. L'única diferència és que les hem creat fora del mètode main, amb la qual cosa es converteixen en variables globals accessibles des de tots els mètodes.

Pel mateix motiu, modifiquem tots els mètodes per treballar directament sobre aquestes estructures, i no fer ús del pas per paràmetres. A tall d'exemple, hem mostrat el mètode d'inicialització de l'LCD:

```
void lcdInit(void) {
      state.screen = INIT;
      // LCD hardware initializing code
      ...
      // End LCD hardware initializing code
      state.targetTemp = 250;
      state.screen = TARGET;
```

```
        setLCD();
}
```

Com es pot observar, en tot moment se suposa que la variable state està accessible.

De manera equivalent, tenim la lectura de la temperatura, la qual es guarda en la variable state:

```
void getTemp(void) {
        uint16_t adcValue16 = ADCVALUE;
        int32_t adcValue32 = adcValue16;
        int16_t result;
        adcValue32 <<= 5;
        adcValue32 /= 59;
        result = (int16_t) (adcValue32 - 610);

        state.sensorTemp = result;
}
```

D'altra banda, les tasques de processament segueixen una fórmula semblant:

```
void buttonsState(void) {
        if (buttons.up == true) {
                if (buttons.down == true) {
                        if (buttons.mode == true) {
                                state.screen = TARGET;
                        }
                } else {
                        state.screen = TARGET;
                        state.targetTemp += 1;

                }
        } else {
                if (buttons.down == true) {
                        state.screen = TARGET;
                        state.targetTemp -= 1;
                } else {
                        if (buttons.mode == true) {
                                if (state.screen == TARGET) {
                                        state.screen = SENSOR;
                                } else {
                                        state.screen = TARGET;
                                }
                        }
                }
```

```
        }

        if (state.targetTemp > 500)
              state.targetTemp = 500;
        else if (state.targetTemp < -250)
              state.targetTemp = -250;
  }
```

En aquest cas, les dades de l'estructura buttons són utilitzades per a modificar l'estructura global state.

Una vegada modificats tots els mètodes per a treballar d'acord amb variables globals, l'estructura del programa ens quedarà de la manera següent:

```
  int main(void) {
        initialize();

        tasksAdd(getButtons);
        tasksAdd(getTemp);
        tasksAdd(buttonsState);
        tasksAdd(relayState);
        tasksAdd(setRelay);
        tasksAdd(setLCD);

        tasksRun();

        return EXIT_SUCCESS;
  }
```

en què `initialize()` inclourà el mètode `tasksInit()`. Amb aquest disseny, s'observa que tot el procés segueix un comportament molt previsible, ja que es coneix per endavant la seqüència que se seguirà en tot moment.

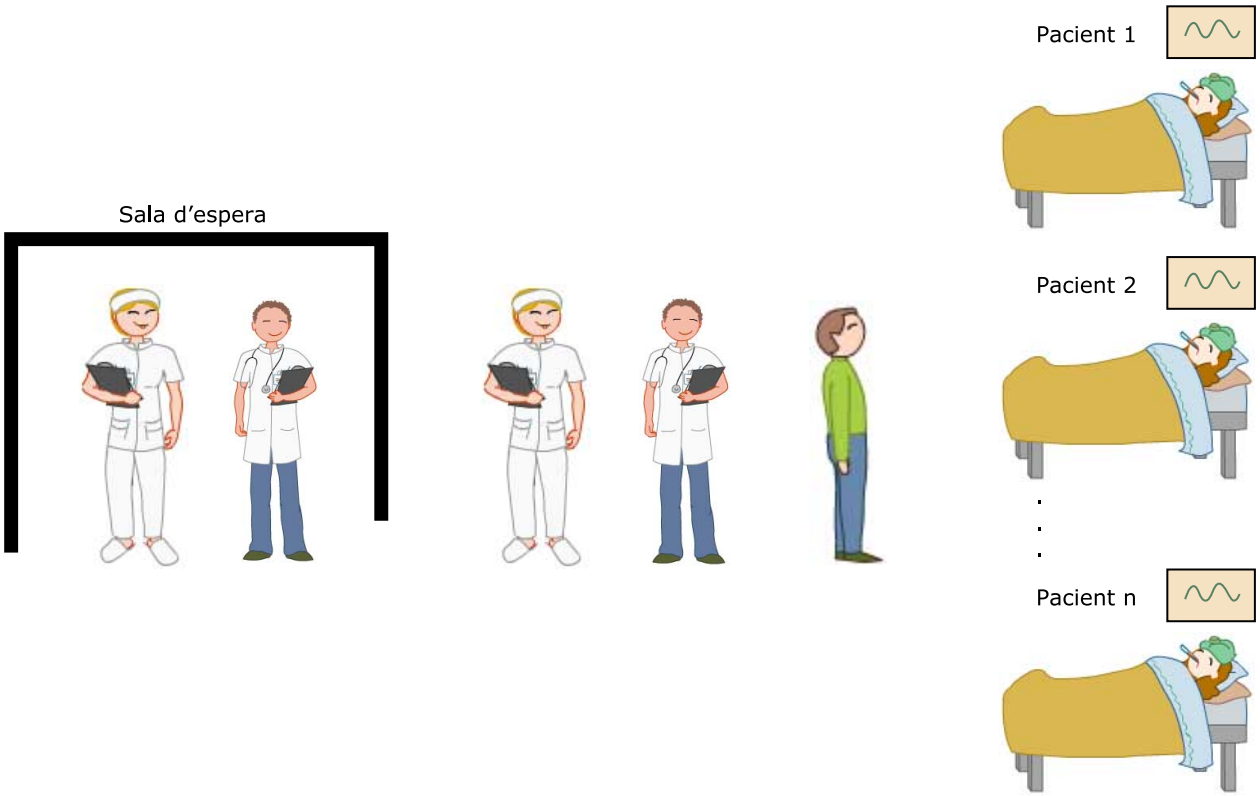
3.1.2. Possibles millores: bucle síncron

El principal inconvenient d'aquesta solució és el consum innecessari de recursos, ja que bona part del temps no hi haurà canvis i, per tant, el CPU estarà fent un treball innecessari. Si a això hi sumem que, en general, aquests sistemes treballen amb bateries, podem veure que gastarem les bateries molt de pressa. En aquest sentit, aquesta solució no és especialment òptima. Per veure-ho, en posarem un exemple.

Suposem que tenim un servei d'urgències. Per fer-lo funcionar, muntem una sala on rebem els pacients. Aquests poden tenir un gran nombre de símptomes, però per a simplificar, suposarem que per la sala passen tres infermeres: dues que els mesuren la temperatura i la freqüència cardíaca, i una tercera que els subministra la medicació. A més, passen dos metges especialistes que,

d'acord amb les mesures preses per les dues primeres infermeres, proporcionen un diagnòstic i indiquen la medicació que ha d'administrar la tercera. Només una infermera o especialista pot ser a la sala, per la qual cosa el torn en què passen cadascun d'ells per la sala és controlat per un bidell. Una vegada han acabat la infermera o el metge, el bidell avisa el següent.

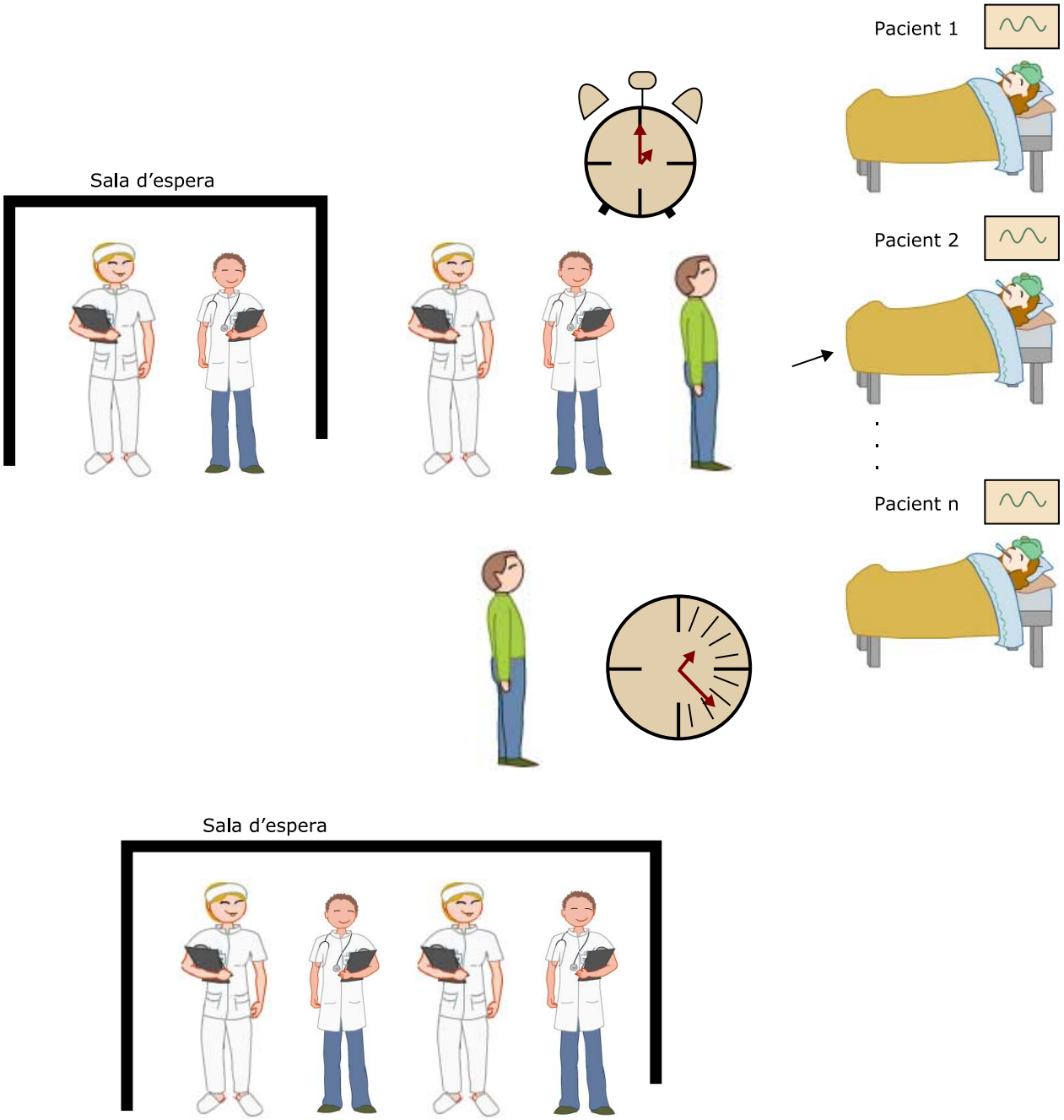
Diagrama de les infermeres i especialistes passant de sala en sala, avisats pel bidell



Com a informació de partida, sabem que cada infermera o especialista necessita com a màxim cinc minuts per a fer la seva tasca. També sabem que el temps mínim entre pacients és d'una hora. Per tant, tenim temps que el pacient sigui reconegut per tots els implicats abans que arribi el següent, amb la qual cosa aconseguim el nostre objectiu, com en el cas del Round-Robin.

No obstant això, suposem que aquest procés es fa de nit i que, per tant, els especialistes aprofiten per a dormir mentre no estan diagnosticant. És evident que seria més convenient despertar les infermeres i especialistes cada hora, ja que no val la pena que estiguin passant per una sala buida, quan sabem que com a màxim n'hi haurà un cada hora. Per a aconseguir-ho, caldria proporcionar un despertador al bidell. El posarà en funcionament el bidell quan hagi passat per la sala l'última infermera o especialista, i sonarà al cap de mitja hora.

Diagrama de les infermeres i especialistes passant del dormitori a la sala i viceversa, amb el despertador

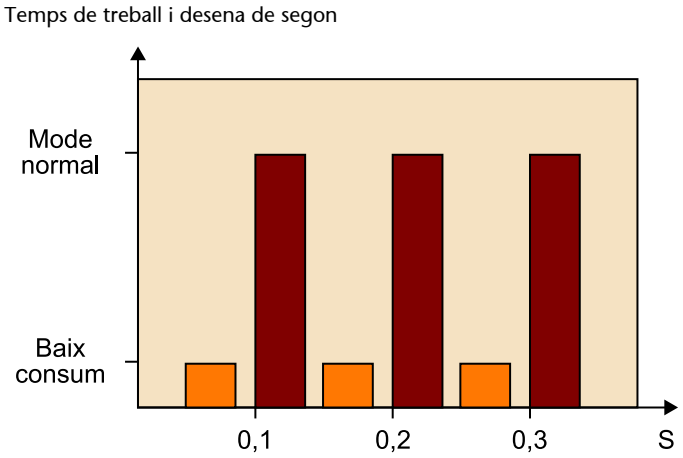


Aquesta mateixa solució la podem utilitzar en un microcontrolador que treballi en dues freqüències de rellotge o dos modes:

- **Mode normal**. El CPU sol treballar a la freqüència de rellotge més ràpida possible, per a dur a terme el màxim de tasques en el mínim temps possible.

- **Mode baix consum**. El CPU i la majoria dels perifèrics estan aturats i només treballen els perifèrics imprescindibles.

Podem considerar que les tasques del control de temperatura, botons i relé són les infermeres, mentre que les tasques de decisió serien els especialistes. Per la seva banda, el bidell seria el gestor de tasques. Si aquestes tasques es fan prou de pressa (menys una desena de segon), el microcontrolador es pot adormir el temps des que s'acaben les tasques fins que arriba la desena següent, de manera equivalent a com ho fan les infermeres i els especialistes. Aquest procés es mostra en l'esquema següent:



Així, doncs, el que podem fer és deixar en marxa un temporitzador que cada desena de segon doni un senyal que desperti el CPU (el despertador). Una vegada despert, el CPU farà un bucle de control i, quan acabi, es posarà en estat de baix consum fins que arribi el següent senyal de rellotge.

Com a hipòtesi, suposem que, en mitjana, es modifica l'estat del sistema vuit vegades cada hora. Aquests canvis són equivalents a 2 pulsacions de botó i 2 modificacions de l'estat del relé. Suposem, també, que el temps que triga a dur-se a terme el bucle quan no hi ha canvis és inferior a una mil·lèsima de segon; quan n'hi ha, és una desena de segon. Tenint en compte que el nombre de bucles que es duen a terme en una hora són:

$$n_{\text{bucles}}\,[1\text{ hora}] = \frac{10\text{ bucles}}{1\text{ segon}} \cdot \frac{3.600\text{ segons}}{1\text{ hora}} = 36.000\text{ bucles}$$

Podem veure que la potència necessària es redueix a:

$$t_{\downarrow}\text{equivalent} = (n_{\downarrow}\text{canvi} \cdot t_{\downarrow}\text{canvi} + n_{\downarrow}\text{estable} \cdot t_{\downarrow}\text{estable}) \;/\; (n_{\downarrow}\text{canvi} + n_{\downarrow}\text{estable}) =$$
$$(n_{\downarrow}\text{canvi} \cdot t_{\downarrow}\text{canvi}) + (n_{\downarrow}\text{total} - n_{\downarrow}\text{canvi}) \cdot t_{\downarrow}\text{estable} \;/\; n_{\downarrow}\text{total} =$$

Això implica que reduïm el consum pràcticament a una mil·lèsima part de l'inicial, amb un increment notable de la durada de la bateria. El cost és la simple generació d'una interrupció de rellotge.

Per a aconseguir-ho, són necessàries únicament dues modificacions:

1) Implementar una interrupció de rellotge, que cada cert període de temps desperti el sistema.

2) Afegir en el mètode tasksRun un pas en mode baix consum que deixi el micro aturat fins que es produeixi la interrupció de rellotge.

La implementació d'interrupcions és sempre una tasca que s'ha de fer amb extrema cura, ja que aquesta atura de manera asíncrona el treball que fa el CPU i interromp la tasca que s'està duent a terme. Aquest accés asíncron també implica la possibilitat de carreres crítiques, en poder-se modificar de manera "concurrent" el valor d'una mateixa variable/posició de memòria.

En el nostre cas, la possibilitat de carreres crítiques ha de ser considerada impossible. El motiu és que la interrupció només s'habilita quan el gestor de tasques ha fet un cicle complet (com passava amb el bidell). D'aquesta manera, sempre que salti una interrupció el CPU ja estarà en estat de baix consum i, per tant, el CPU no estarà fent cap tasca, i no hi haurà carrera crítica.

En aquesta situació, la rutina de servei de la interrupció serà molt simple:

```
void timeInterrupt(void){
      disableTimeInterrupt();
      standardMode();
}
```

en què s'assumeix que el mètode standardMode() és el que torna a activar el CPU a la velocitat requerida. Aquest mètode haurà de ser substituït pel corresponent a l'arquitectura.

D'altra banda, el gestor de tasques també es veu modificat de la manera següent:

```
void tasksRun(void) {
      int i;
      task_t tp;

      for (;;) {
            for (i = 0; i < tasks.number; i++) {
                  tp = tasks.queue[i];
                  tp();
            }

            lowPowerMode();
      }
}
```

en què el mètode lowPowerMode posa el CPU en baix consum. Finalment, hauríem de modificar tant l'estructura tasks com el mètode d'inicialització de tasques. En la primera, hi afegiríem un camp per establir el període de la interrupció (usperiod), i un comptador per a saber el temps transcorregut (ustime), en què l'us indica que treballen en unitats d'us, ja que generalment els microcontroladors treballen amb períodes inferiors. L'estructura quedaria de la manera següent:

```
struct {
      int number;
      int pointer;
      uint32_t ustime;
      uint32_t usperiod;
      task_t queue[NUMBER_TASKS];
} tasks;
```

En el mètode d'inicialització de tasques, hi hauríem d'afegir el mètode per cridar les interrupcions, que inclouria com a paràmetres la tasca d'interrupció i el període del rellotge. Quedaria de la manera següent:

```
void tasksInit(const uint32_t usperiod) {
      int i;

      tasks.number = 0;
      asks.pointer = 0;
      tasks.usperiod = usperiod;

      for (i = 0; i < NUMBER_TASKS; i++) {
            tasks.queue[i] = NULL;
      }

      startInterruptHandler(timeInterrupt, tasks.usperiod);
}
```

en què assumim que startInterruptHandler fa les tasques següents:

1) Donar d'alta el mètode timeInterrupt en la taula d'interrupcions.

2) Indicar a la interrupció del rellotge el temps que ha d'esperar, valor recollit en la variable usperiod.

3) Habilitar la interrupció de rellotge.

Com en el cas inicial, en l'apèndix s'han inclòs els mètodes que permeten emular el comportament del microcontrolador en un sistema Posix.

3.2. Sistemes operatius basats en esdeveniments

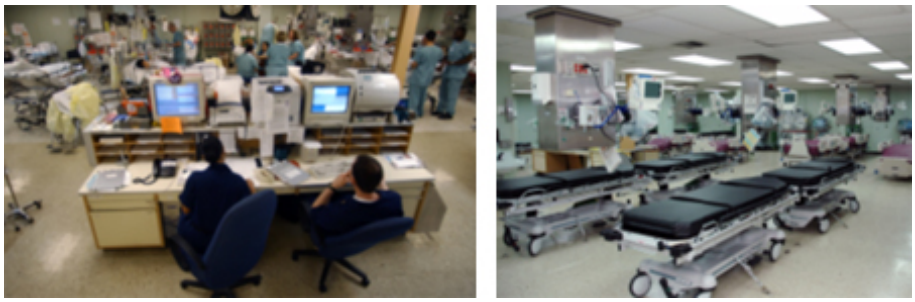
Els microcontroladors actuals proporcionen molts recursos que, utilitzats adequadament, ens poden permetre una gran flexibilitat. Seguint amb el símil d'un hospital, suposem que ara organitzem una unitat de vigilància intensiva amb els últims sistemes de monitoratge.

En aquest cas, tenim un conjunt de pacients amb monitors que registren diferents biosenyals, que ens en proporcionen informació. Aquests monitors són capaços de generar una alarma quan se superen certs llindars i indicar el metge que està de guàrdia per a aquest pacient.

A més, la sala disposa d'una biblioteca on es recullen els historials de cada pacient a la UVI, amb la informació dels diferents monitors, i també els diagnòstics duts a terme pel metge i els tractaments que cal fer.

Hi afegim dos últims condicionants: sempre hi ha una infermera de guàrdia i només hi pot haver un metge a la UVI. Si ens demanen que organitzem aquesta UVI d'una manera que sigui òptima i segura, quines opcions tenim d'UVI?

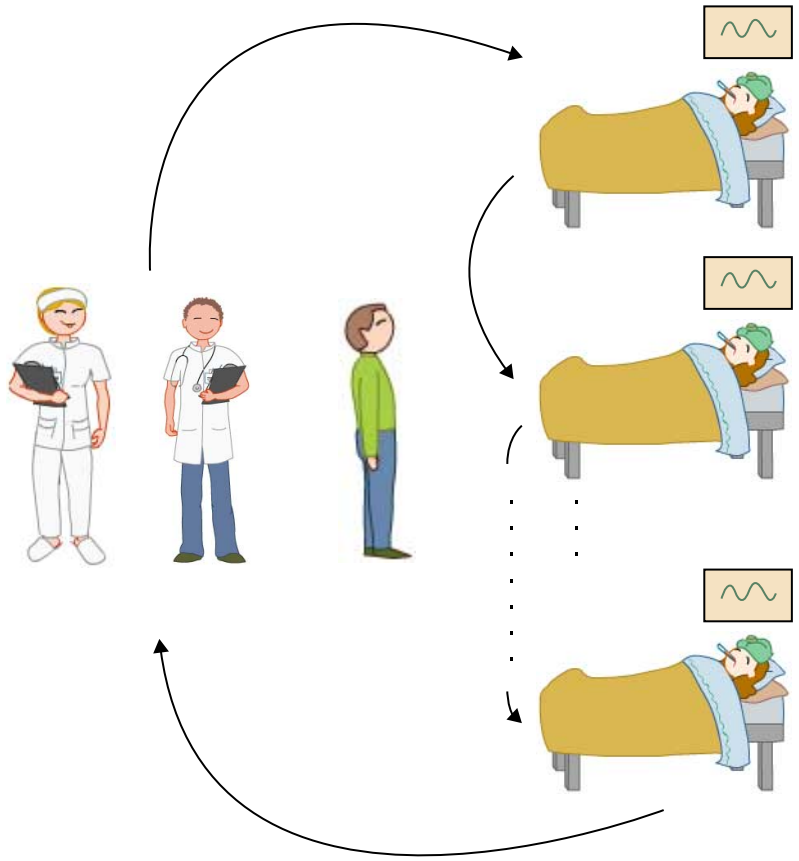
Imatges de dues UVI



La de l'esquerra mostra la sala amb els pacients i els monitors centralitzats en un ordinador. La de la dreta mostra la sala buida, amb els monitors distribuïts per llit. Font: imatges de domini públic, US Navy

Una primera seria que la infermera avisés cada metge perquè passés de manera cíclica pels pacients que li toquen. El metge miraria si s'han superat els llindars i, en aquest cas, indicaria si cal dur a terme alguna acció. Aquesta solució seria equivalent a la que hem proposat inicialment per a urgències, i la podem considerar poc adequada, tenint en compte que els monitors inclouen la possibilitat de generar alarmes.

Metges visitant de manera cíclica els diferents pacients



Una segona seria fer ús de les alarmes dels monitors. Cada vegada que salta una alarma, la infermera pren nota del motiu en l'historial del pacient i avisa el metge indicat per a deixar-lo entrar a la sala. Mentre és a la sala, el metge consulta l'historial del pacient, analitza les dades noves i escriu en l'historial si cal dur a terme alguna acció. Quan acaba, surt de la sala i la infermera descansa fins que arriba l'alarma següent.

En aquesta situació, és possible que algun metge no tingui els recursos necessaris per a diagnosticar el pacient; en aquest cas, pot sol·licitar que s'avisi un metge nou. Aquest, en principi, disposarà de més informació i podrà donar un diagnòstic i tractament per a situacions menys comunes.

Aquesta solució funciona bé quan la possibilitat que es produeixi una alarma mentre hi hagi un metge a la sala és nul·la o, dit d'una altra manera, quan el temps entre alarmes és molt superior al mateix temps que triga el metge a fer una actuació. No obstant això, això ocorre quan el temps entre alarmes és indeterminat, encara que la mitjana és sempre superior al temps d'actuació. En aquest cas, ens trobem amb dues alternatives.

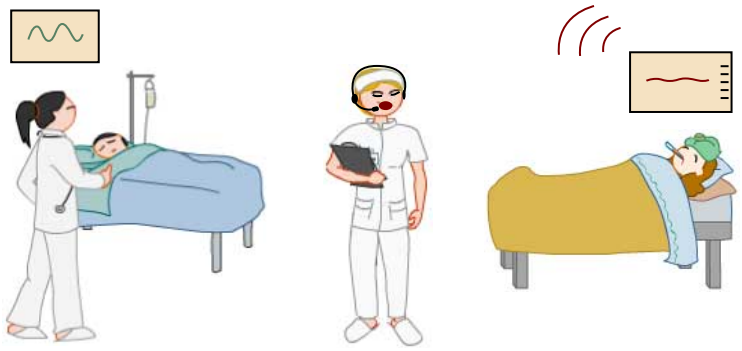
L'escenari seria el següent: ha saltat una primera alarma i un metge és a la sala diagnosticant el pacient. Durant aquest temps, salta una segona alarma. La infermera pot fer diverses coses: la primera és obviar l'alarma i esperar que

el metge acabi la feina i, llavors, mirar l'alarma. Aquesta solució, no obstant això, planteja que l'alarma estarà sonant constantment i molestant la resta de pacients, per la qual cosa la descartem.

Una segona opció és que la infermera miri l'alarma i prengui nota del metge. Amb aquesta informació, també pot fer dues coses: fer fora el metge que és a la sala, avisar el segon metge i fer que passi a la sala. Quan acabi el segon metge, torna a donar pas al primer perquè acabi la feina. Aquesta opció té l'avantatge que l'alarma deixa de molestar, però és probable que el metge que s'ha de quedar esperant es molesti una miqueta. A més, afegeix la incertesa de no saber mai quant temps trigarà un metge a poder ser interromput en multitud d'ocasions.

Una tercera alternativa és que la infermera, en lloc de fer fora el metge que és en aquests moments a la sala, avisi el segon metge perquè esperi a la sala. Quan acabi el primer metge, el segon hi pot entrar i dur a terme la feina. Aquesta opció té l'avantatge que fins que no s'acaba amb un pacient no es comença amb el següent, la qual cosa fa que el temps que triga cada metge sigui més fàcil de predir.

Infermera avisant el metge segons l'indicat pel monitor del pacient, mentre el metge tracta el pacient



Encara que la introducció de la utilització del rellotge redueix el consum del sistema, és evident que, en el cas de tenir esdeveniments asíncrons, com les alarmes d'una UVI, la solució del rellotge no és especialment adequada.

En el context d'un microcontrolador, entendrem per *esdeveniment* qualsevol modificació d'una entrada o d'un perifèric, independentment del seu origen, que sigui notificada de manera asíncrona al CPU per mitjà d'una interrupció.

De manera equivalent a les alarmes de la UVI, l'esdeveniment sol requerir un mínim processament que és portat a terme per la rutina de servei de la interrupció, per exemple, la comparació entre un senyal i un llindar. Si detecta la necessitat, afegeix una nova tasca (metge) al gestor de tasques (infermera). Aquestes tasques es van executant seguint un esquema; la primera que arriba

és la primera que s'executa (*first in, first executed*). I així fins que es duen a terme totes les tasques. En cas necessari, una tasca pot afegir al gestor de tasques una de nova, de manera equivalent a com ho fa la ISR.

Un aspecte important en aquest tipus de SO és l'intercanvi d'informació entre tasques. Per a entendre-ho, seguirem amb l'exemple de la UVI.

Suposem que un metge i la infermera són amb un pacient determinat. El metge està escrivint el tractament en l'historial. Just en aquest moment, salta una alarma en el monitor d'aquest pacient. En aquest cas, què fem amb l'historial? En principi, la infermera l'hi ha d'agafar, afegir-hi les noves dades de l'alarma, avisar-lo que ha de tornar a entrar quan hagin passat la resta de metges i retornar-li l'historial perquè continuï escrivint el tractament.

La pregunta és: com afegeix les dades la infermera? Si escriu just després d'on ho està fent el metge, podem tenir un problema greu. A tall d'exemple, suposem que el metge està escrivint la dosi d'un medicament que són 100 mg i, just després d'escriure l'1, salta l'alarma. La infermera agafa l'historial i hi escriu al darrere el codi de l'alarma, que és un tres. Una vegada fet, retorna l'historial al metge, el qual simplement escriu els dos zeros i les unitats que falten. Resultat: en lloc de posar 100 mg en l'historial, posa 1.300 mg, la qual cosa pot tenir un resultat fatal.

En aquest cas, és evident que s'ha d'aconseguir que quan el metge escrigui alguna cosa, ho escrigui tot sencer. Un procés que garanteix que s'escriu o es llegeix una cosa d'una manera completa es denomina *atòmic*. En aquest cas, es pot assegurar que no apareixeran modificacions que poguessin produir errors en el sistema.

Per a això, s'ha de prendre alguna precaució. Aquesta pot ser simplement limitar les possibles alarmes mentre el metge escriu en l'historial. Una vegada acabat, es tornen a activar i, si ha saltat alguna alarma, la infermera fa les accions pertinents sense afectar el metge. En cas que necessiti una altra vegada que visiti el pacient, el torna a posar a la cua.

Aquesta solució és la que s'acostuma a utilitzar en els microcontroladors. Per a evitar que una tasca i una interrupció modifiquin alhora un *buffer*, i que generin el que es coneix com a *carreres crítiques*, es deshabiliten les interrupcions. Una vegada modificat el *buffer*, es tornen a habilitar. Tenint en compte que les interrupcions poden requerir una resposta immediata, és important que la seva deshabilitació sigui tan curta com sigui possible.

Per a aconseguir-ho, se segueix un procés en dos passos:

1) Es prepara tota la informació que cal afegir en el *buffer* en variables temporals internes de la funció.

2) Una vegada es té tota la informació, es crida una funció que fa els canvis oportuns.

D'aquesta manera, es redueix el temps durant el qual s'han de bloquejar les interrupcions.

3.2.1. Implementació

La implementació, en aquest cas, és molt semblant a la del Round-Robin síncron. La principal diferència és que la cua de tasques és dinàmica. Per tant, passa a ser un *buffer* de tasques, amb la qual cosa haurà de rebre el mateix tractament que qualsevol altre *buffer*. En aquest sentit, serà necessari que els accessos siguin atòmics.

Seguint amb l'exemple del control de calefacció, hem de fer dos canvis en la manera de treballar. La primera és determinar l'ordre en què es duen a terme les tasques i la segona, la manera com passarem la informació entre les mateixes.

Per a això, el primer pas és determinar quines interrupcions tindrem. En el nostre cas, veiem que en tindrem dues de principals:

- **temperatureInterrupt**: ens la generarà l'ADC quan hagi acabat la mesura de temperatura;

- **buttonsInterrupt**: la generarà el bloc d'entrades/sortides quan rebi el canvi d'estat del polsador.

Una vegada rebudes aquestes interrupcions, hauran de cridar alguna de les tasques que teníem definides:

- **getButtons**: llegeix les dades dels botons;
- **getTemp**: llegeix dades de temperatura;
- **buttonsState**: processa els botons;
- **relayState**: processa l'estat del relé;
- **setRelay**: modifica la sortida del relé.

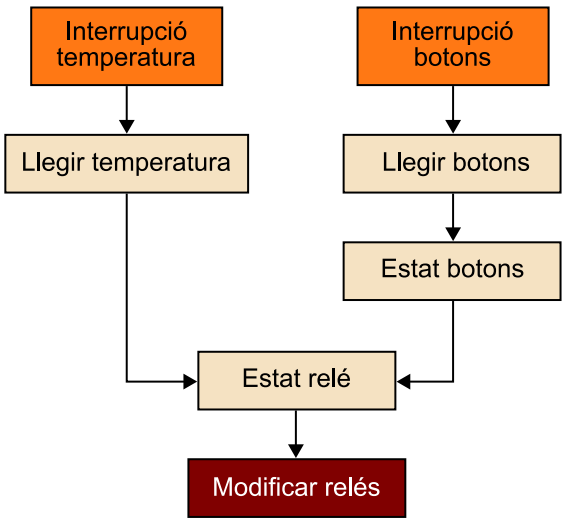
Veiem que tenim dos camins per seguir:

1) De temperatureInterrupt passem a getTemp. D'aquesta passem a relayState i, finalment, a setRelay.

2) De buttonsInterrupt hem de passar a getButtons que seria la tasca companya. D'aquesta a buttonsState, que passaria a relayState i a setRelay.

Tal com podíem esperar, hi ha un punt d'unió que és relayState, ja que és la que defineix la sortida següent. Podem veure el comportament en el diagrama de flux de la figura següent:

Diagrama de tasques del controlador de calefacció



Segons aquest diagrama, es pot determinar quina tasca ha de cridar cadascuna de les tasques. Cada interrupció afegeix una tasca de gestió de les dades al gestor de tasques. D'aquesta manera, es minimitza el temps que el microcontrolador es troba en una interrupció. Cada tasca, al seu torn, afegeix la tasca o tasques següents al gestor, i així fins que s'arriba a la sortida.

La principal modificació es troba en el gestor de tasques, el qual ha de tenir en compte la possibilitat que una interrupció sol·liciti afegir una tasca mentre està modificant el contingut de la llista. Aquesta passa a ser més senzilla, ja que desapareix la interrupció de rellotge i els camps associats, de manera que és molt semblant al cas Round-Robin. No obstant això, s'ha modificat el nom del camp number per last, per a indicar que la llista pot créixer d'una manera arbitrària.

```
struct {
        int last;
        int pointer;
        task_t queue[NUMBER_TASKS];
} tasks;
```

El mètode d'inicialització de la llista també és semblant al Round-Robin, amb l'excepció comentada anteriorment.

```
void tasksInit(void) {
        int i;

        tasks.last = 0;
        tasks.pointer = 0;
```

```
        for (i = 0; i < NUMBER_TASKS; i++) {
                tasks.queue[i] = NULL;
        }
}
```

No obstant això, els mètodes de gestió de la llista sí que es veuen modificats de manera notable. En primer lloc, apareixen els mètodes startCriticalSection i endCriticalSection, els quals s'encarreguen d'evitar que cap altra tasca o interrupció modifiqui l'estat de la llista de tasques mentre es fa aquesta operació. S'ha buscat que el nombre d'operacions que es duu a terme dins d'aquesta secció sigui el mínim necessari per a assegurar que la llista de tasques continuï funcionant una vegada s'estigui fora de la secció crítica. Concretament, el camp last de la llista i l'assignació a aquesta llista s'han de fer de manera atòmica per a assegurar la consistència de les dades en la llista.

```
int tasksAdd(task_t task) {
        int pointer;

        startCriticalSection();

        pointer = tasks.last;

        tasks.queue[pointer] = task;

        pointer++;

        if (pointer >= NUMBER_TASKS) {
                pointer = 0;
        }

        tasks.last = pointer;

        endCriticalSection();

        return pointer;
}
```

De la mateixa manera, es duu a terme la gestió de les tasques que cal executar. En aquest cas, es comprova en primer lloc que la llista no estigui buida (pointer igual a last). Si no ho està, es determina la tasca següent que cal fer. Una vegada feta, se surt de la zona crítica. En funció de si hi ha tasca per fer o no, s'executa la tasca, o es passa a l'estat de baix consum.

És important destacar que només les tasques relacionades amb la gestió de la llista queden dins de la zona crítica. Els motius són dos. El primer és que la crida al mètode no es veu afectada per l'entrada d'una nova interrupció, més

enllà de retardar-ne l'engegada, per la qual cosa es pot treure fora. El segon és que, si es fes córrer la tasca dins de la zona crítica, les interrupcions continuarien deshabilitades, per la qual cosa es deixaria de rebre dades noves.

```
void tasksRun(void) {
	task_t tp;

	for (;;) {
		startCriticalSection();

		if (tasks.pointer != tasks.last) {
			tp = tasks.queue[tasks.pointer];
			tasks.queue[tasks.pointer] = NULL;

			tasks.pointer++;
			if (tasks.pointer >= NUMBER_TASKS) {
				tasks.pointer = 0;
			}
		} else {
			tp = NULL;
		}
		endCriticalSection();

		if (tp != NULL) {
			tp();
		} else {
			lowPowerMode();
		}
	}
}
```

A aquests canvis, cal afegir-hi l'aparició de les interrupcions de la lectura de temperatura i de l'estat dels botons. Per a aquests canvis, hem modificat lleugerament les estructures buttons i state, inclosos dos camps en què les interrupcions copiessin els valors de les entrades.

```
volatile struct buttons_t {
	uint16_t portValue;
	bool up;
	bool down;
	bool mode;
} buttons;

volatile struct state_t {
	enum {
		INIT, SENSOR, TARGET
	} screen;
```

```
        uint16_t adcValue;
        int16_t targetTemp;
        int16_t sensorTemp;
        bool relay;
} state;
```

En el cas de buttons, es diu portValue, i en el de state, adcValue. La seva funció es dedueix fàcilment a partir del comportament de les dues interrupcions.

Variables de tipus volatile

Una variable de tipus volatile és aquella el valor de la qual es pot modificar de manera asíncrona (per exemple, una interrupció), per la qual cosa no volem que el compilador optimitzi el codi per al seu contingut.

```
void buttonsInterrupt(void) {
        standardMode();

        buttons.portValue = getButtonsValue();
        tasksAdd(getButtons);
}
```

Veiem que la interrupció comença tornant el microcontrolador a l'estat normal i que, seguidament, copia l'estat dels botons a portValue. Com en casos anteriors, getButtonsValue és un mètode inline que retorna el contingut del registre de port associat als botons. Es manté com a mètode per a evitar la modificació d'aquest registre per error. Una vegada copiat el valor, es llança la tasca getButtons, que és l'encarregada de traduir del format del port d'interrupció a l'estructura que utilitzem.

Aquest mateix procés segueix la interrupció de l'ADC, tal com es mostra a continuació.

```
void adcInterrupt(void) {
        standardMode();

        state.adcValue = getADCValue();
        tasksAdd(getTemp);
}
```

Les tasques associades són bàsicament iguals que en el cas del Round-Robin, amb dues excepcions:

1) la utilització de buttons o state com a entrada, i

2) que al final inclouen el mètode taskAdd, amb la tasca que segueix a continuació, en aquest cas buttonsState.

```
void getButtons(void) {
```

```
        if ((buttons.portValue & (1<<UP_BUTTON)) != 0)
               buttons.up = true;
        else
               buttons.up = false;
        if ((buttons.portValue & (1<<DOWN_BUTTON)) != 0)
               buttons.down = true;
        else
               buttons.down = false;
        if ((buttons.portValue & (1<<MODE_BUTTON)) != 0)
               buttons.mode = true;
        else
               buttons.mode = false;

               tasksAdd(buttonsState);
}
```

Aquesta mateixa situació se segueix per a getTemp.

```
void getTemp(void) {
        uint16_t adcValue16 = state.adcValue;
        int32_t adcValue32 = adcValue16;
        int16_t result;

        adcValue32 <<= 5;
        adcValue32 /= 59;
        result = (int16_t) (adcValue32 - 610);

        state.sensorTemp = result;

        tasksAdd(relayState);
}
```

La resta de tasques inclou la modificació d'afegir taskAdd.

Aquesta mateixa filosofia és la que segueixen SO com el TinyOS per a gestionar les diferents tasques i interrupcions. Es pot analitzar el codi resultant del programa nescc, i comparar-lo amb el presentat anteriorment.

3.2.2. Planificació per prioritats

En l'exemple de la UVI, hem tractat de la mateixa manera totes les alarmes. No obstant això, és evident que un senyal de parada cardíaca ha de ser tractat més ràpid que una pujada de temperatura. Per aquest motiu, en molts casos s'estableix una prioritat en funció de la gravetat de l'alarma.

Aquest mateix procés es pot dur a terme en un SO. En aquest cas, podem establir la prioritat de les interrupcions i tasques. Quan acaba una tasca, es mira quina és la primera que té més prioritat i és la que es comença a fer. I així successivament fins que acaben totes les tasques de la llista.

En aquest cas, la tasca ja no pot ser definida com un punter de funció, sinó com una estructura, que inclourà el camp a punter a la tasca i un segon punter per a especificar la prioritat. A més, s'afegeix un punter a la tasca següent de la mateixa prioritat.

```
struct task_t {
       taskp_t pointer;
       uint8_t prioriy;
       struct task_t* next;
};
```

L'estructura tasks es modifica tenint dos vectors de punters a tasques. El primer indica la primera tasca de la cua. El segon indica l'última tasca de cada cua. D'aquesta manera, no és necessari recórrer totes les tasques per a localitzar-ne l'última.

```
struct {
       struct task_t* queue[MAX_PRIORITY];
       struct task_t* last[MAX_PRIORITY];
} tasks;
```

D'acord amb això, s'han modificat el mètode d'inicialització de l'estructura de tasques.

```
void tasksInit(void) {
       int i;

       for (i = 0; i < MAX_PRIORITY; i++) {
              tasks.queue[i] = NULL;
              tasks.last[i] = NULL;
}
}
```

També s'ha modificat la manera d'afegir tasques. Ara ha d'incloure el paràmetre de prioritat, el qual no haurà de superar el màxim permès. És important destacar que es requereix l'ús de memòria dinàmica. Per això, és important controlar la manera com es crea l'estructura de tasques i, sobretot, la manera d'alliberar la memòria quan en finalitza l'ús.

Com s'observa, la creació de l'estructura de la tasca es fa fora de la secció crítica, ja que no es veu afectada per l'entrada d'una nova interrupció. Per a evitar possibles problemes, es limita la prioritat de la tasca a la prioritat màxima permesa, ja que un sistema encastat no es pot parar per aquest motiu.

Un vegada creada i inicialitzada l'estructura d'una tasca, s'accedeix a l'estructura tasks. En aquest moment, es considera que ja s'entra en zona crítica per la qual cosa s'inhabiliten les interrupcions.

```
int tasksAdd(taskp_t task, uint8_t priority) {
        struct task_t* stask;
        struct task_t* pointer;

        if (priority >= MAX_PRIORITY) {
                priority = MAX_PRIORITY-1;
        }

        stask = malloc(sizeof(struct task_t));
        stask->pointer = task;
        stask->prioriy = priority;
        stask->next = NULL;

        startCriticalSection();

        pointer = tasks.last[priority];

        if (pointer != NULL) {
                pointer->next = stask;

                tasks.last[priority] = stask;
        } else {
                tasks.queue[priority] = stask;
                tasks.last[priority] = stask;
        }
        endCriticalSection();

        return 1;
}
```

El gestor de tasques també s'ha de modificar de manera adequada. En aquest cas, el primer pas és mirar les diferents cues, començant per la de més prioritat, i es mira si hi ha alguna tasca pendent. Si és així, es porta a terme el mètode associat. Com que s'està accedint a tasks, es considera de zona crítica.

```
void tasksRun(void) {
        int i;
        struct task_t* stask;
```

```
        struct task_t* pointer = NULL;
        taskp_t tp;

        for (;;) {
               startCriticalSection();

               for (i = 0; i < MAX_PRIORITY; i++) {
                      pointer = tasks.queue[i];
                      if (pointer != NULL)
                             break;
                      }

               if (pointer != NULL) {
                      tp = pointer->pointer;

                      stask = pointer->next;

                      if (stask == NULL) {
                             tasks.queue[i] = NULL;
                             tasks.last[i] = NULL;
                      } else {
                             tasks.queue[i] = stask;
                      }
               } else {
                      tp = NULL;
               }

               endCriticalSection();

               if (pointer != NULL)
                      free(pointer);

               if (tp != NULL) {
                      tp();
               } else {
                      lowPowerMode();
               }
        }
  }
```

Una vegada s'ha determinat el mètode associat a la tasca, i s'han reorganitzat dins de la cua, se surt de la zona crítica. Una vegada fet això, s'allibera la memòria que té ocupada l'estructura de la tasca.

Una vegada fetes aquestes modificacions, és necessari actualitzar les crides al mètode addTasks, i donar les prioritats adequades a les diferents tasques. Concretament, es donarien les prioritats més altes a les tasques requerides per a

controlar els perifèrics temporitzats (getTemp, setRelay). D'altra banda, les relacionades amb l'usuari es poden reduir de prioritat, ja que el temps de resposta requerit sol ser inferior (getButtons, buttonsState, setLCD...). Finalment, les tasques prioritàries no relacionades amb els perifèrics reben una prioritat intermèdia (relayState).

Activitat

Modifiqueu les crides al mètode addTask en les diferents tasques per tal de tenir en compte les prioritats indicades anteriorment.

No obstant això, la solució basada en prioritats tampoc no és la panacea. Seguint amb l'exemple de la UVI, suposem que de sobte salten un gran nombre d'alarmes, unes de gran prioritat com aturades cardíaques, aturades respiratòries... I unes altres de més lleus, com un inici de febre. Seguint el nostre esquema inicial, es van tractant els casos més aguts (més prioritat) i s'espera a finalitzar-los per a començar a tractar els menys prioritaris. No obstant això, passa el temps i un pacient amb febre no és tractat pel seu metge, i supera els 42 °C, amb una entrada en coma del pacient.

Evidentment, aquesta és una situació patològica, però pot ocórrer en un SO. Si moltes tasques prioritàries ocupen els recursos del microcontrolador, les menys prioritàries no es poden executar, i moren d'inanició. Si això es porta al límit, es pot donar el cas que es mostra en el requadre adjacent.

Exemple

Es diu que quan es va voler parar un gran ordinador IBM 7094 del MIT el 1973, es va trobar un procés (l'equivalent d'una tasca en grans equips) que portava des de 1967 sense ser executat. En la imatge, un IBM 7094 de la NASA per al projecte Mercuri.

Diagrama de temps d'un sistema multitasca

Per a solucionar aquest problema, s'han buscat alternatives com l'envelliment de les tasques.

3.2.3. Planificació per envelliment

Per a solucionar el problema que un pacient amb una lleugera pujada de temperatura acabi en coma, el que podem fer és incrementar la prioritat en funció del temps que fa que no es tracta. Així, cada cert temps es revisa la prioritat de les alarmes, i les que fa un temps determinat que no són tractades, són incrementades de prioritat. Passat un temps preestablert, unes tasques de baixa prioritat (pujada de febre) passen a ser d'alta prioritat. Aquest temps de pujada ha de ser prou curt perquè cap pacient acabi amb una patologia greu a causa d'un retard, i prou llarg perquè la priorització no perdi la seva raó de ser.

En el cas d'un microcontrolador, aquesta modificació de la prioritat la pot dur a terme el gestor de tasques cada vegada que en finalitza una. Per a això, haurà de modificar l'estructura de la tasca, de manera que inclogui el moment en què va ser creada la tasca per a poder-ne determinar l'edat. A més, serà necessari disposar d'un comptador del sistema que determini el temps actual des del punt de vista del microcontrolador.

Activitat

Modifiqueu l'estructura `task_t` i els diferents mètodes per tal de tenir en compte l'envelliment.

Aquestes solucions donen resposta quan els problemes són relativament senzills. No obstant això, en molts casos interessa donar resposta a diversos problemes de manera concurrent. Per a això, és necessari saltar a un sistema multitasca.

3.3. Sistemes operatius multitasca

Fins ara, hem treballat amb tasques que executen un conjunt de processos de principi a fi. Una vegada acaben, es passa a la següent, i així de manera consecutiva. Cada tasca ha de donar pas a la següent. I si no ho fa, la resta de tasques simplement no s'executa. Aquest tipus de funcionament es coneix també com a *planificació cooperativa*. És a dir, és la "bona voluntat" de cada tasca la que permet que el sistema funcioni. Si una és "egoista", la resta es queda sense recursos.

El gran avantatge d'aquesta planificació és que les tasques no són interrompudes, amb l'única excepció de les interrupcions que podran prendre el control de manera temporal. Aquesta forma de treball simplifica molt la gestió de recursos, ja que és únicament una tasca la que hi accedeix en cada moment.

No obstant això, és evident que, en aplicacions complexes, dependre que una tasca permeti o no passar a la següent en pot dificultar el disseny. Per aquest motiu, apareixen els sistemes multitasca.

Reprenent l'exemple de la UVI, si un metge es dediqués en exclusiva a cada pacient, i fos amb ell des que entra fins que se li dóna l'alta, la resta de pacients no podria ser visitada. Per aquest motiu, s'estableix un torn de visites, de manera que va passant per cadascun dels pacients. Aquest mateix principi seria el seguit per aquest tipus de sistemes operatius.

3.3.1. Planificació col·laborativa

El funcionament es basa en un principi semblant al Round-Robin síncron. Recordem que, en aquesta solució, el gestor de tasques es posava en funcionament cada cert temps i executava totes les tasques. Una vegada finalitzades, passava a mode baix consum. En aquest cas, en lloc d'esperar que es finalitzi una tasca, és la mateixa tasca la que quan arriba a un punt que no pot continuar i sol·licita un canvi de context.

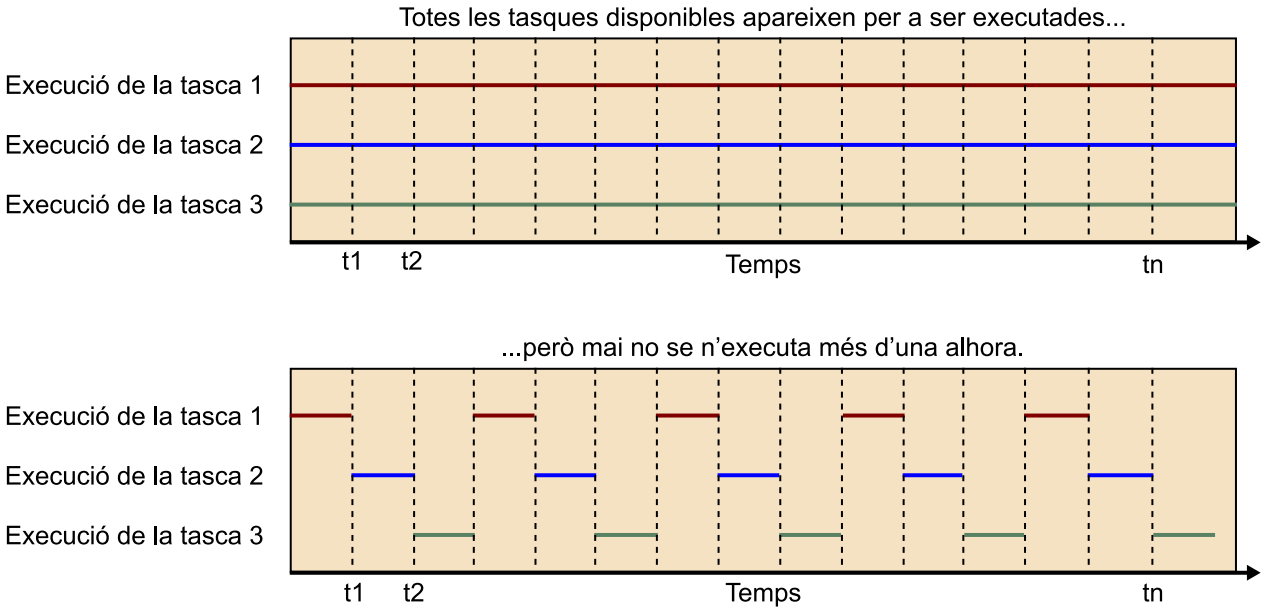
Suposem, per exemple, que la tasca 1 ha emplenat el *buffer* d'enviament de l'RS-232, i que una segona vol enviar. Si no ha donat temps a buidar el *buffer*, la segona es quedaria esperant. El que pot fer és sol·licitar un canvi de context perquè segueixin la resta de tasques i, quan finalitzin, tornar-la a cridar. D'aquesta manera, se'n minimitza l'impacte.

Aquest model té l'inconvenient que les tasques han d'estar ben programades, la qual cosa no sempre és així. Per aquest motiu, és millor que sigui el mateix gestor el que estableixi el temps de treball per a cada tasca, i el moment de dur a terme un canvi de context.

3.3.2. Planificació anticipativa (*preemptive*)

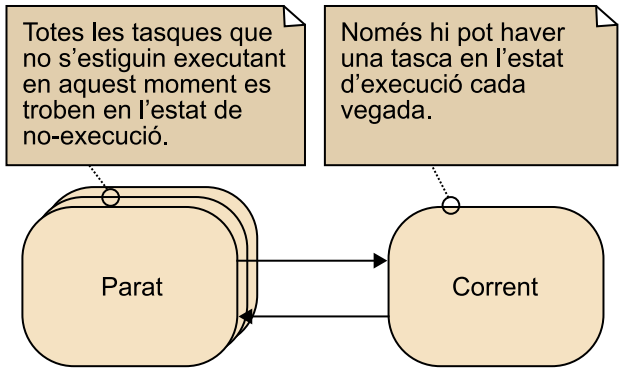
En aquest cas, el gestor de tasques és interromput de manera periòdica; en cada interrupció, aquest deté la tasca que està en marxa i n'executa la següent. Així fins que una de les tasques finalitza, en aquest cas es treu de la llista. Amb aquest mètode, dóna la sensació que duen a terme múltiples tasques alhora, encara que el que fan en realitat és anar executant parts d'una tasca cada vegada.

Diagrama de temps d'un sistema multitasca



Les diferents tasques van passant de l'estat parades a l'estat corrent de manera continuada.

Pas de l'estat parat a corrent d'una tasca

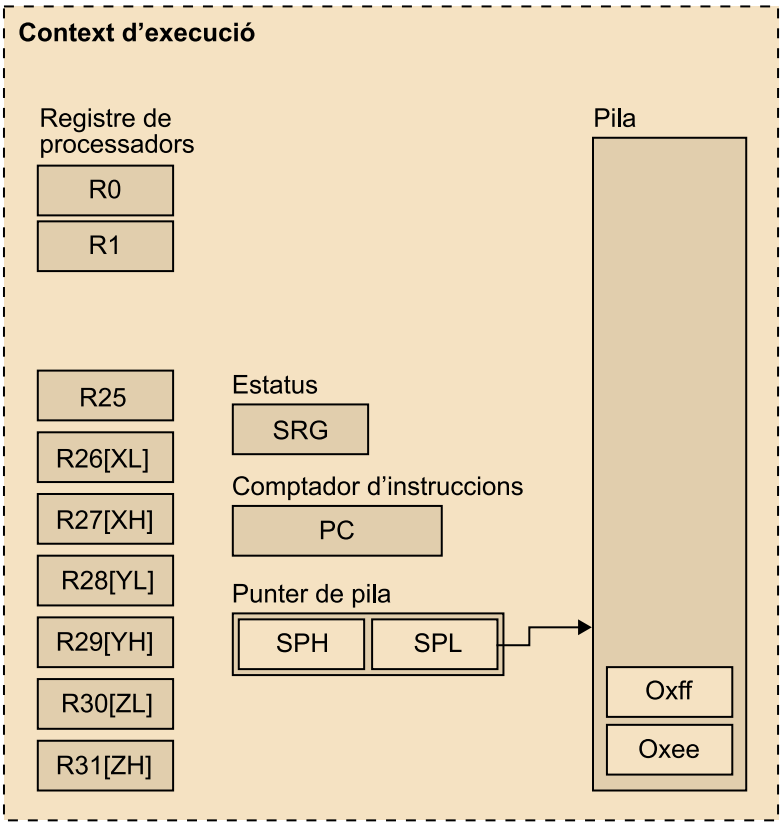


No obstant això, en estar fent diferents tasques alhora, apareixen nous reptes que cal solucionar. Si tornem a l'exemple de la UVI, veiem que si el metge va canviant de pacient, també ha d'anar canviant d'història clínica, cosa que requereix que la torni a llegir per a recordar la informació que recull. En cas contrari, podria tractar d'una manera incorrecta els diferents pacients, en basar-se en informació errònia.

Aquesta mateixa situació es produeix en un CPU. Si una tasca és interrompuda per a dur-ne a terme una altra, és necessari que guardi la informació recollida en els registres del CPU. En cas contrari, en tornar-se a executar, els registres podrien haver estat modificats i donar resultats erronis. Aquest procés es denomina *canvi de context*. En general, s'han de guardar els registres i la pila, que són els que recullen l'estat del CPU en aquest moment. Això comporta un increment dels recursos necessaris per a cada tasca, i també un increment del temps per a passar d'una tasca a una altra, ja que és necessària la còpia dels di-

ferents registres. A tall d'exemple, per a un ATMega s'han de copiar 32 registres. Per a simplificar tot aquest procés, totes aquestes còpies se solen fer sobre la pila, ja que la majoria de microcontroladors inclou un conjunt d'instruccions que faciliten aquest procés.

Context d'execució dels registres de l'AVR



Aquesta és la manera de treballar del FreeRTOS. En altres casos, se'n fa una còpia directa, com seria el cas dels TOSThreads de TinyOS.

Una vegada feta la còpia, es recupera la informació dels registres de la següent tasca que cal executar i se li dóna pas. D'aquesta manera, cada tasca que s'executa veu el microcontrolador en el mateix estat en què es trobava quan s'ha aturat i pot continuar treballant sense possibilitat que es produeixin errors.

Per posar-ne un exemple, ens basarem en els *threads* de Posix, encara que es poden fer solucions semblants amb FreeRTOS. Veurem què passaria amb dues tasques que funcionen de manera continuada i que no estan gaire optimitzades, com les que es mostren a continuació.

```
void* vPrintTask( void* arg ) {
      const char *pcTaskName = (char *) arg;
      volatile unsigned long ul;

      /* As per most tasks, this task is implemented in an infinite loop. */
```

```
        for (;;) {
                /* Print out the name of this task. */

                vPrintString(pcTaskName);

                /* Delay for a period. */
                for (ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) {
                        /* This loop is just a very crude delay implementation. There is
                        nothing to do in here. Later exercises will replace this crude
                        loop with a proper delay/sleep function. */
                }
        }
        return NULL;
}
```

Aquesta tasca s'executa de manera contínua, i mostra un missatge recollit en pcTaskName i espera un cert temps. El temps es defineix per un bucle, que ha de comptar fins a aconseguir el valor mainDELAY_LOOP_COUNT (es pot començar amb 1.000.000). Evidentment, aquesta no és la millor manera d'incloure un bucle d'espera.

Per a imprimir fem ús de la funció vPrintString, que hem escrit de manera semblant a com ho faríem si haguéssim d'enviar les dades a una pantalla LCD. Per aquest motiu, fem ús d'un putc, en lloc de printf.

```
void vPrintString(const char *pcString) {
        volatile clock_t now;
        char message[80];
        int i;
        int length;

        // Get the current time
        now = clock();

        length = snprintf(message, sizeof(message), "%lu:\t", (unsigned long) now);

        for( i = 0; i < length; i++ ){
                putc(message[i], stdout);
        }

        length = strlen(pcString);

        for( i = 0; i < length; i++ ){
                putc(pcString[i], stdout);
        }
}
```

Podem crear una segona tasca igual i modificar-ne únicament el nom per vTask2, i en la cadena del missatge substituir l'1 pel 2. Així podríem escriure la funció main com a:

```
int main(void) {
      pthread_t       threadID1;
      pthread_t       threadID2;
      void               *exit_status;
      const char* message1 = "Sensor 1: \n";
      const char* message2 = "Sensor 2: \n";

      pthread_create(&threadID1, NULL, vPrintTask, (void *) message1);
      pthread_create(&threadID2, NULL, vPrintTask, (void *) message2);

      pthread_join(threadID1, &exit_status);

      return EXIT_SUCCESS;
}
```

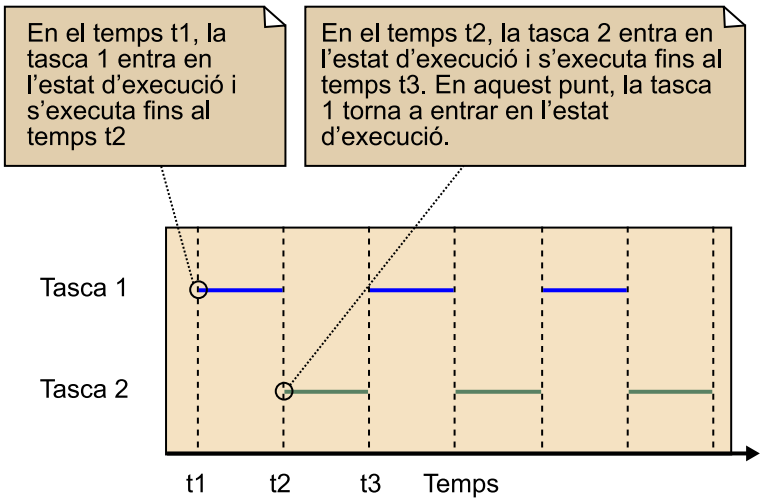
Si executem el programa veiem que, encara que tenim un bucle infinit en ambdues tasques, aquestes s'executen de manera intercalada.

Consola programa amb dues tasques FreeRTOS

```
Terminal — bash — 80×24
Last login: Mon May 30 23:01:37 on ttys000
zagal:~ chema$ /Users/chema/Documents/Workspace.eclipse/PthreadHeaterController/
Debug/PthreadHeaterController
11995704::              Sensor 1:
Sen2s3o7r9 :2   :S e
n2s5o4r3 :1     :S e
nsor 2:
2687:   Sensor 1:
2707:   Sensor 2:
2722:   Sensor 1:
2736:   Sensor 2:
2749:   Sensor 1:
2767:   Sensor 2:
22788062::              SeSnesnosro r2 :1 :

2990:   Sensor 2: 3
010:    3042:   SensSoern s1o:r
2:
3132124:1       :       SeSnesnosro r2 :1 :

3270:   Sensor 2:
3291:   3316:   SensorS e2n:s o
r 1:
3386:   Sensor 2:
```

Així, les tasques van saltant d'un estat a un altre de manera continuada, tal com es mostra en la figura següent:

Pas d'una tasca a una altra en un gestor anticipatiu



Per tant, malgrat que el programa està escrit de manera incorrecta, ambdues tasques s'executen de manera correcta. Podem millorar-ne l'estil, modificant la tasca.

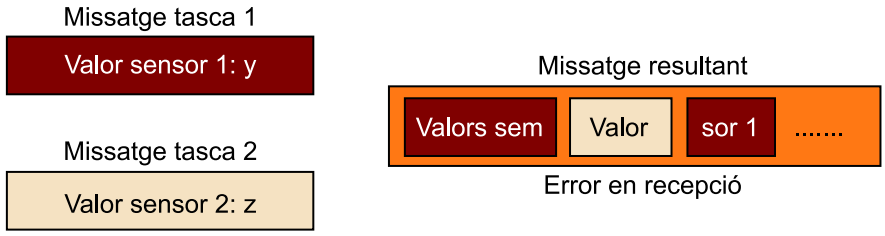
```
void* vPrintTask( void* arg ) {
      const char *pcTaskName = (char *) arg;
      struct timespec tim, tim2;

      /* As per most tasks, this task is implemented in an infinite loop. */
      for (;;) {
            /* Print out the name of this task. */

            vPrintString(pcTaskName);

            tim.tv_sec = 1;
            tim.tv_nsec = 0;

            nanosleep(&tim , &tim2);
      }

      return NULL;
}
```

En aquest cas, l'espera és controlada pel gestor, amb la qual cosa el microcontrolador queda alliberat per a cridar una altra tasca si en queda alguna a la cua.

Una altra dificultat és l'ús dels perifèrics. Si una única tasca ha d'accedir a un perifèric, no hi ha cap problema. No obstant això, si en són diverses, com és el cas que estem mostrant, poden aparèixer problemes. A tall d'exemple, suposem que llegim dades de dos sensors mitjançant dues tasques independents i que volem presentar les dades en una pantalla LCD.

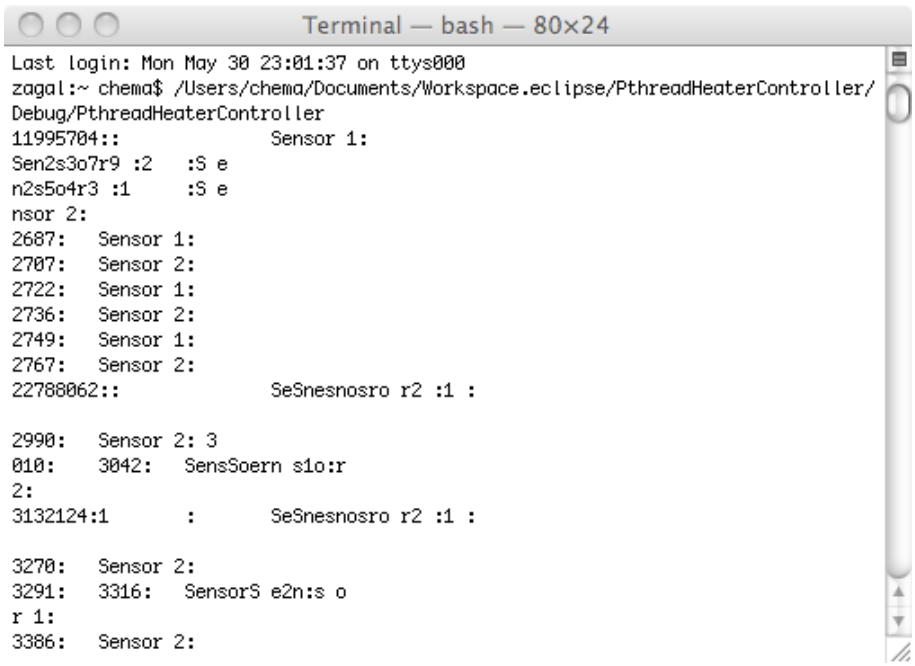
Si la tasca del sensor 1 comença a enviar dades a la pantalla i és interrompuda pel gestor, la segona tasca pot començar a enviar dades, sense que hagi finalitzat la primera. En aquest cas, el que rebrem al PC serà una barreja dels dos missatges.

Exemple d'enviament de missatge sense tenir en compte concurrència



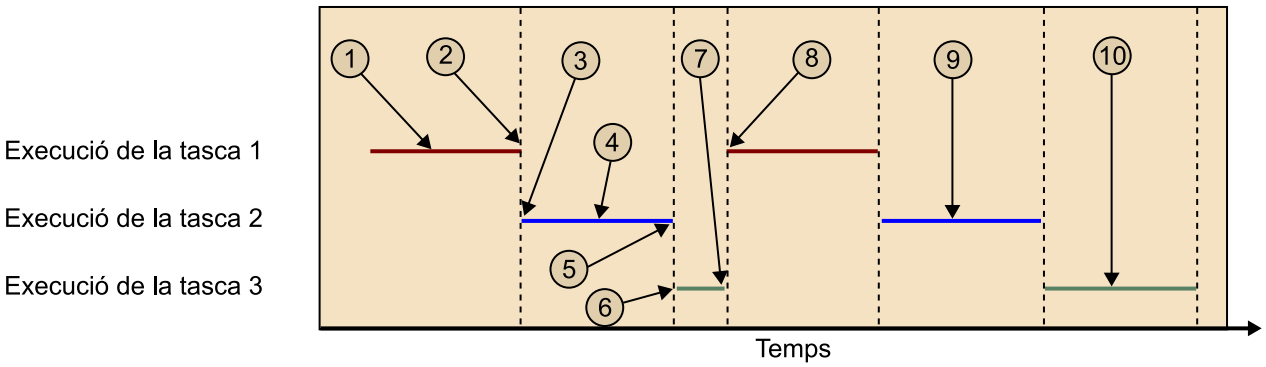
Seria equivalent al fet que dos metges anessin escrivint en un únic historial: l'embolic seria monumental. En el nostre cas, ho podem reproduir posant el valor de tim.tv_sec a 0 i tim.tv_nsec a 500. Al cap d'una estona, ens podem trobar una línia com la següent:

Resultat a la consola de la barreja dels missatges



Per tant, cal trobar-hi alguna solució. La primera alternativa és que la tasca 1 bloquegi la pantalla fins que hagi finalitzat de presentar les dades. Si durant aquesta escriptura, el gestor interromp la tasca 1, la tasca 2 es trobarà el perifèric bloquejat, per la qual cosa no hi podrà accedir. El diagrama es mostra en la figura següent:

Diagrama de temps d'un gestor de tasques



Els diferents punts dins del diagrama de temps es descriuen a continuació.

1) S'està executant la tasca 1.

2) El gestor suspèn la tasca 1...

3) ...i continua la tasca 2.

4) Mentre la tasca 2 s'està executant, bloqueja un perifèric per a poder-hi accedir d'una manera exclusiva.

5) El gestor suspèn la tasca 2...

6) ...i continua la tasca 3.

7) La tasca 3 intenta accedir al mateix perifèric i s'adona que està bloquejat, per la qual cosa se suspèn a si mateixa.

8) El gestor continua amb la tasca 1, etc.

9) En la següent ocasió en què la tasca 2 s'estigui executant, finalitza amb el perifèric i l'allibera.

10) En la següent ocasió en què la tasca 3 s'estigui executant, es troba amb el perifèric alliberat i, en aquesta ocasió, en fa ús fins que és suspesa pel gestor.

Per a dur a terme aquest bloqueig, la primera opció és bloquejar directament el gestor de tasques, o el que és equivalent: parar les interrupcions. D'aquesta manera, s'assegura que no hi haurà problemes. El principal problema d'aquesta solució és que es paren totes les tasques, incloses les que no estan relacionades amb el perifèric. Per a solucionar-ho, el millor és fer ús d'un mútex.

3.3.3. Mútex

Mútex és l'abreviatura d'*exclusió mútua*. En aquest cas, mentre una tasca bloqueja un mútex, la resta de tasques no hi pot accedir.

a) La solució més simple és crear una variable en la memòria que indiqui el bloqueig. Per a modificar-la, es desactiven temporalment les interrupcions, es comprova l'estat de la variable i, en el cas adequat, es modifica. Finalment, es tornen a activar les interrupcions.

b) Una segona opció, que depèn de l'arquitectura i instruccions disponibles pel CPU, és utilitzar un flag que pot ser llegit, comparat i modificat en una sola instrucció. Aquest tipus de flags es coneixen com a **mútuament exclusius**. Quan una tasca vol entrar en un codi crític ordre atòmica que permeti llegir, comparar i modificar un valor en una mateixa instrucció. D'aquesta manera, no és necessari parar les interrupcions.

c) Una tercera és un intercanvi amb un valor de la memòria: si el valor és diferent, el mútex està lliure i, si és igual, està ocupat. La instrucció XCH d'AVR que s'acaba d'introduir permet fer-ho, però no està disponible en els microcontroladors d'Atmel actuals. Es pot modificar el programa anterior per a fer ús del mútex.

En primer lloc, s'ha de modificar el funcionament de vPrintString, per a incloure-hi el mútex, en aquest cas lock. L'hem creat com a variable global, ja que ha de poder ser accedit per tots els threads.

```
pthread_mutex_t lock;

void vPrintString(const char *pcString) {
      volatile clock_t now;
      char message[80];
      int i;
      int length;

      pthread_mutex_lock(&lock);
      // Get the current time
      now = clock();

      length = snprintf(message, sizeof(message), "%lu:\t", (unsigned long) now);
```

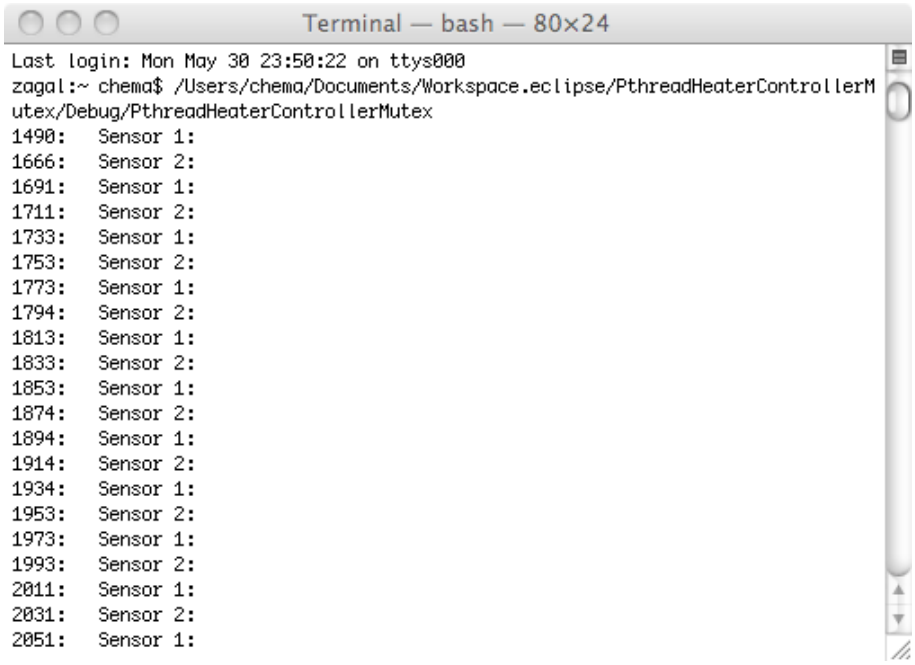
```
        for( i = 0; i < length; i++ ){
              putc(message[i], stdout);
        }

        length = strlen(pcString);

        for( i = 0; i < length; i++ ){
              putc(pcString[i], stdout);
        }
        pthread_mutex_unlock(&lock);
  }
```

També s'ha de modificar la funció main per a iniciar el mútex.

```
  int main(void) {
        pthread_t       threadID1;
        pthread_t       threadID2;
        void             *exit_status;
        const char* message1 = "Sensor 1: \n";
        const char* message2 = "Sensor 2: \n";

        pthread_mutex_init(&lock, NULL);

        pthread_create(&threadID1, NULL, vPrintTask, (void *) message1);
        pthread_create(&threadID2, NULL, vPrintTask, (void *) message2);

        pthread_join(threadID1, &exit_status);

        return EXIT_SUCCESS;
  }
```

D'aquesta manera, s'aconsegueix que la presentació de les dades es faci correctament.

Consola amb el mútex en funcionament



Per les seves característiques, la implementació dels mútex se sol deixar a càrrec del SO. D'aquesta manera, si en intentar accedir a un mútex, es veu que està bloquejat en aquest moment, el gestor de tasques pot donar pas a una altra tasca mentre s'espera que finalitzi el bloqueig. D'aquesta manera, es minimitza l'impacte temporal.

No obstant això, és important indicar que els *buffers* que intercanvien dades amb interrupcions (nucli del SO) no poden utilitzar mútex, ja que si una interrupció es troba un mútex bloquejat, no podria finalitzar la seva tasca, i es perdria la dada.

3.3.4. Accés al nucli

Entendrem per *accés al nucli*, o *crides al sistema*, una sol·licitud perquè faci una acció:

- Afegir/treure tasques.
- Accés a perifèrics.
- Accés a *buffers*.
- Bloquejar/desbloquejar mútexs.

En aquest cas, hi ha dues opcions:

1) Bloquejar el nucli.
2) Treballar amb dues cues.

El bloqueig del nucli seria semblant a un bloqueig d'interrupcions, però per programari. En aquest cas, es crea un mútex, que en el moment en què una tasca crida el sistema, es bloqueja. Quan una altra tasca intenta fer una crida, es comprova l'estat del mútex i, si està bloquejat, la tasca es queda en espera. Quan la primera crida allibera el mútex, la segona ja hi pot accedir.

D'aquesta manera, les interrupcions poden continuar funcionant, i amb això els perifèrics associats. Perquè l'efecte sigui petit, les rutines pròpies del nucli han d'estar molt optimitzades.

Si malgrat tot hi ha aspectes que són lents, per exemple l'accés a algun perifèric, es poden crear diversos mútex, en funció del que es vulgui bloquejar. L'increment del nombre de mútex (granuralitat) també implica un increment de la complexitat del nucli. Aquesta és la solució utilitzada per la majoria de SO Unix, i també pel FreeRTOS.

La segona opció és fer ús de dues cues. La primera, associada al nucli, treballaria per esdeveniments. Per tant, les tasques associades al nucli haurien d'estar molt optimitzades, per a evitar increments en la latència. Una de les tasques del nucli seria, al seu torn, el gestor de tasques anticipatiu, sobre el qual correrien les tasques de l'usuari. Per tant, les tasques de l'usuari viurien en un entorn multitasca, a diferència de les del nucli, que es basarien en esdeveniments.

Quan una tasca d'usuari (presentar dades en pantalla) vol accedir al nucli (enviar les dades a la pantalla), crea un missatge, que no deixa de ser sol·licitar que s'afegeixi una tasca (addCharLCD) del nucli al gestor per esdeveniments, de manera que la tasca de l'usuari queda bloquejada. Quan arriba el torn de la tasca del nucli, aquesta fa la seva operació i, una vegada finalitzada, indica al gestor de tasques anticipatiu que la tasca de l'usuari pot continuar. Aquest model és l'utilitzat per les TOSThreads de TinyOS.

Des del punt de vista de senzillesa, probablement la solució basada en bloqueig de nucli és la més senzilla, en requerir únicament un gestor de tasques. No obstant això, des del punt de vista de modularitat, l'opció de treballar segons dues cues i missatges pot ser més òptima.

3.3.5. Una abraçada mortal

Una abraçada mortal és una situació semblant a un intercanvi d'ostatges. Si cada bàndol diu que només alliberarà els seus ostatges si l'altre ho fa primer, no es durà mai a terme l'intercanvi. Cal buscar una solució alternativa per a assegurar que no es produeixi el bloqueig. En la realitat, se sol utilitzar un terreny neutral, com un pont, que permet que ambdós processos d'alliberament es facin d'una manera simultània.

El mateix succeeix quan dos processos tenen bloquejats recursos –per exemple perifèrics–, i necessiten que l'altre alliberi el que té bloquejat per a poder continuar. Per tant, no es pot sortir d'aquesta situació, i ambdues tasques queden bloquejades. En aquest cas, la situació és més complexa, en no haver-hi l'equivalent a un pont. No obstant això, les tasques no són tan "malpensades", per la qual cosa es poden buscar alternatives en les quals una tasca alliberi primer i, després, segueixi la segona.

Aquest seria el cas de treballar amb dues cues, amb la de nucli basada en esdeveniments. En aquest cas, en haver-se de finalitzar una tasca abans de fer-se la següent, s'evita aquest bloqueig. No és així pel que fa a les tasques d'usuari, en què encara podria succeir. En aquest cas, és la perícia del programador la que permetrà trobar la solució més adequada.

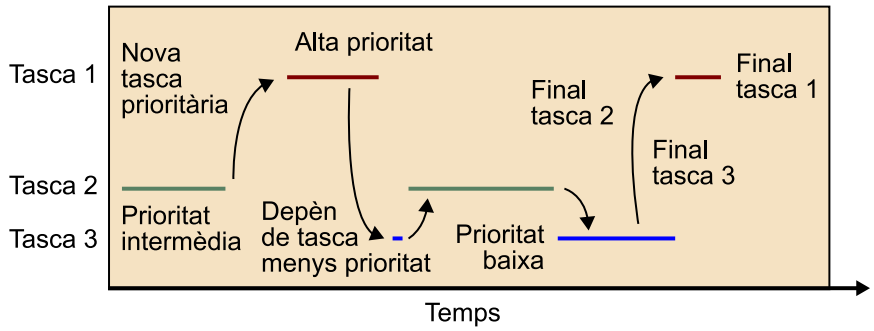
3.3.6. Prioritats

De la mateixa manera que en el cas del gestor basat en esdeveniments, els gestors multitasca també poden treballar per prioritats. En aquest cas, el context és una mica més complex, a causa del gestor anticipatiu.

A tall d'exemple, quan es fa una planificació per prioritats estàtica, pot succeir que una vegada ha començat una tasca, n'aparegui una de nova que tingui més prioritat (de manera equivalent a una interrupció). En aquest cas, és necessari que la tasca que s'està executant actualment deixi pas a la més prioritària, per a això s'ha de dur a terme un canvi de context.

Aquest pas es pot fer de la mateixa manera que en el cas de planificació per franges de temps, però de manera asíncrona. Dit d'una altra manera, en el moment en què la tasca de més prioritat entra en la cua, aquesta ha de passar a l'estat "RUN", i la que estava corrent haurà de passar a l'estat "ESPERA". En el moment en què la tasca prioritària finalitzi, la que estava esperant, si no n'hi ha una de nova de prioritat superior, passarà una altra vegada a l'estat "READY" per a poder córrer.

Diagrama de temps quan apareix una tercera tasca de prioritat intermèdia



Quan es treballa amb prioritats, la principal dificultat és que aparegui una inversió de prioritats. Això es produeix quan una tasca d'alta prioritat en requereix una segona de baixa prioritat, i hi ha una tercera tasca de prioritat inter-

mèdia a la cua. En aquest cas, la tasca de baixa prioritat no es pot executar fins que finalitzi la tasca intermèdia. El resultat és que la tasca d'alta prioritat passa a tenir la mateixa prioritat efectiva que la tasca de menys prioritat de la qual depèn. O el que és equivalent: les prioritats s'han invertit.

La inversió de prioritats pot comportar més problemes que els deguts a la planificació de tasques. En sistemes com els basats en temps real, que tenen requeriments temporals molt estrictes, la prioritat d'inversions pot comportar un retard important en l'execució d'una tasca. Si això succeeix, es pot produir un error en cascada, que deixi totalment inutilitzat el sistema.

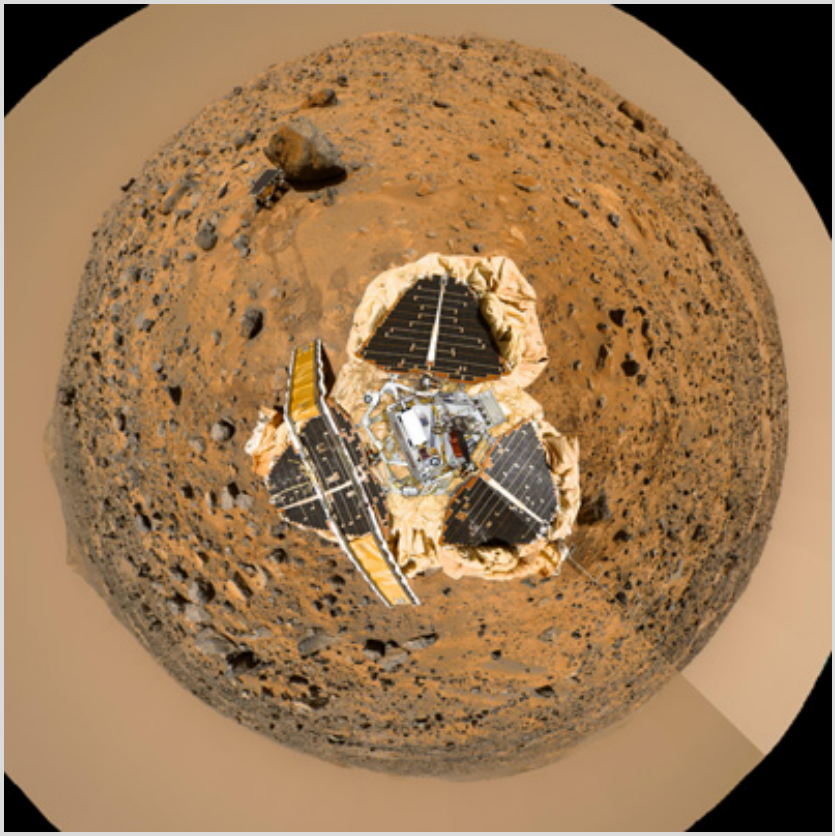
Aquesta situació es va produir en la missió Mars Pathfinder de la NASA. En aquesta missió, s'utilitzava el robot Sojourner, que podia recórrer la superfície marciana, però que pecava de continus resets. Els resets en qüestió obligaven a reinicialitzar tot el robot, amb els inconvenients que això comporta.

Analitzant aquest inconvenient, es va descobrir que el problema es devia al fet que una tasca (bc_dist) requeria més temps de l'esperat. Aquest retard es devia al fet que aquesta tasca precisava fer ús d'un recurs que era controlat per la tasca ASI/MET, que tenia una prioritat molt inferior. Com que bc_dist no podia accedir a aquest recurs, finalment el planificador detectava el bloqueig de la tasca i reiniciava el sistema.

El SO era VxWorks, que activa l'opció d'herència de prioritat i modifica una variable global. Una vegada confirmat que el canvi resolia el problema, aquest es va dur a terme al robot (a Mart) i es va resoldre el problema. Se'n pot trobar més informació en el web:

http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

Imatge del *Mars Pathfinder* (NASA)



En aquest cas, una possible solució és que la segona tasca hereti la prioritat de la primera, perquè s'executi abans que la tercera tasca. Amb això, aconseguim que la primera tasca s'executi a temps.

4. Sistemes en temps real

Fins ara, hem tractat de sistemes operatius encastats. En aquest tipus de sistemes, és molt comú trobar el concepte de *temps real*. I en multitud de casos s'indica que un sistema operatiu determinat és en temps real; no obstant això, les seves maneres de treballar són completament diferents. Per tant, cal plantejar-se la pregunta següent: què és el que defineix un sistema com en temps real?

Per respondre a aquesta pregunta, definirem tres tipus de sistemes d'acord amb el concepte de *temps real*:

1) Sistemes que no treballen en temps real. Són els que asseguren que es durà a terme una acció a partir d'un estímul determinat de manera correcta en algun moment del futur. Un exemple d'aquest sistema és un termòmetre digital, ja que el temps que triga a donar la temperatura correcta no està prefixat.

2) Sistemes en temps real relaxat. Són els que duran a terme una acció determinada a partir d'un estímul, de manera correcta i que, en general, s'acabarà en un temps predeterminat. Un possible exemple n'és un reproductor de vídeo: en general, sempre reproduirà correctament la pel·lícula, però, si en un moment determinat la pel·lícula es para, no resulta especialment greu.

3) Sistema en temps real estricte. En aquest cas, s'assegura que es durà a terme l'acció enfront d'un estímul de manera correcta en un temps inferior al predeterminat. Aquest seria el cas d'un coixí de seguretat d'automòbil, en el qual un retard pot resultar mortal.

> La diferència entre un sistema en temps real estricte i un que no ho és no depèn de la velocitat amb què fa les tasques, sinó que sigui determinista i predictible.

Per tant, qualsevol opció de gestor de tasques i planificador de les presentades anteriorment podrien ser vàlides per a dur a terme un sistema en temps real.

No obstant això, si mirem la facilitat des del punt de vista del programador per a aconseguir que la seva aplicació sigui en temps real, sembla evident que amb un sistema operatiu anticipatiu basat en prioritats serà més fàcil que amb un Round-Robin. I el motiu és fonamentalment perquè ens proporciona més eines per a aconseguir-ho. Per contra, també necessita més recursos i això s'ha de tenir en compte.

En general, considerarem que un SO està pensat per a treballar en temps real si proporciona prioritats i és anticipatiu.

5. Controladors de perifèrics

Com s'ha indicat anteriorment, els controladors de dispositius són un altre dels elements bàsics d'un SO. Proporcionen una abstracció del perifèric associat. D'aquesta manera, el perifèric es converteix en una caixa negra que proporciona uns serveis determinats. El programador només ha d'interactuar amb una interfície, la qual cosa permet que s'oblidi dels aspectes interns del maquinari concret.

Si la definició de la interfície és prou genèrica, simplificarà la seva utilització pel programador. En general, tothom sap interactuar amb aquests botons, la qual cosa en simplifica l'ús.

Exemple

Tots els dispositius de música tenen un teclat semblant (reproduir, pausa, avanç ràpid, rebobinar).

Hi podem afegir una funcionalitat nova, com ara cançó aleatòria, reproduir en bucle... No obstant això, si volem que el nostre dispositiu tingui èxit, hem d'assegurar que disposa d'aquests quatre botons, que tothom sap identificar i, per tant, utilitzar.

Això mateix succeeix en els sistemes encastats. Hi ha un conjunt de dispositius que és genèric. Una interfície tipus sèrie sol fer ús dels mateixos paràmetres en les diferents plataformes (velocitat de transmissió, nombre de bits, bits de parada, paritat...). Tots aquests paràmetres es poden definir de manera genèrica en una interfície, de manera que quan un programador escriu un programa, no s'ha de preocupar de saber el maquinari concret que hi ha sota. Aquesta interfície es denomina *abstracció de maquinari*. El conjunt de totes les interfícies per a perifèrics es denomina *capa d'abstracció de maquinari*.

Evidentment, hi pot haver peculiaritats específiques per a un perifèric, que el fan diferir de la interfície genèrica. En aquest cas, s'hi poden afegir interfícies que ampliïn la interfície original. No obstant això, és important que afegeixin funcionalitat nova i intentar que siguin també genèriques al seu torn.

Per a fer aquests controladors, és recomanable seguir una sèrie de passos:

1) Analitzar les característiques del sensor.
2) Si hi ha una interfície genèrica, analitzar les funcions associades.
3) Si no n'hi cap, definir-ne una.
4) Programar el codi associat a la interfície.

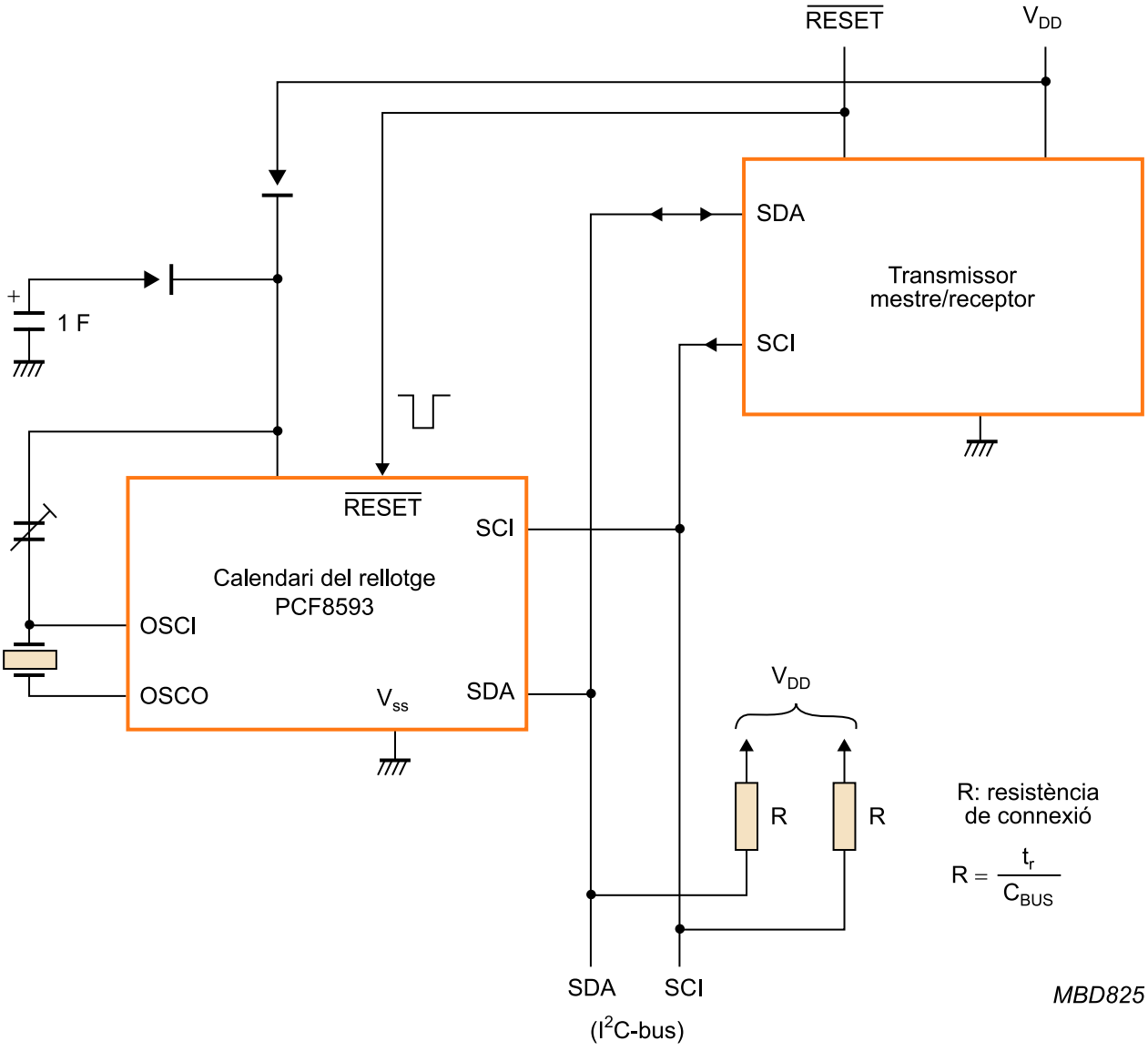
A continuació, els veurem detalladament.

5.1. Informació del maquinari

Com a exemple, analitzarem el cas d'un rellotge en temps real. Aquest tipus de dispositius són rellotges que funcionen de manera contínua mitjançant una bateria. D'aquesta manera, el microcontrolador pot conèixer l'hora actual, consumint molt pocs recursos. La precisió d'aquests rellotges és d'1 segon, aproximadament.

En el nostre cas, farem ús del circuit integrat PCF8593 d'NXP. Si mirem la informació recollida en el full de característiques (*datasheet*), veurem un exemple d'aplicació (vegeu-ho en la figura següent).

Exemple d'aplicació del PCF8593



Hi podem observar dos aspectes fonamentals:

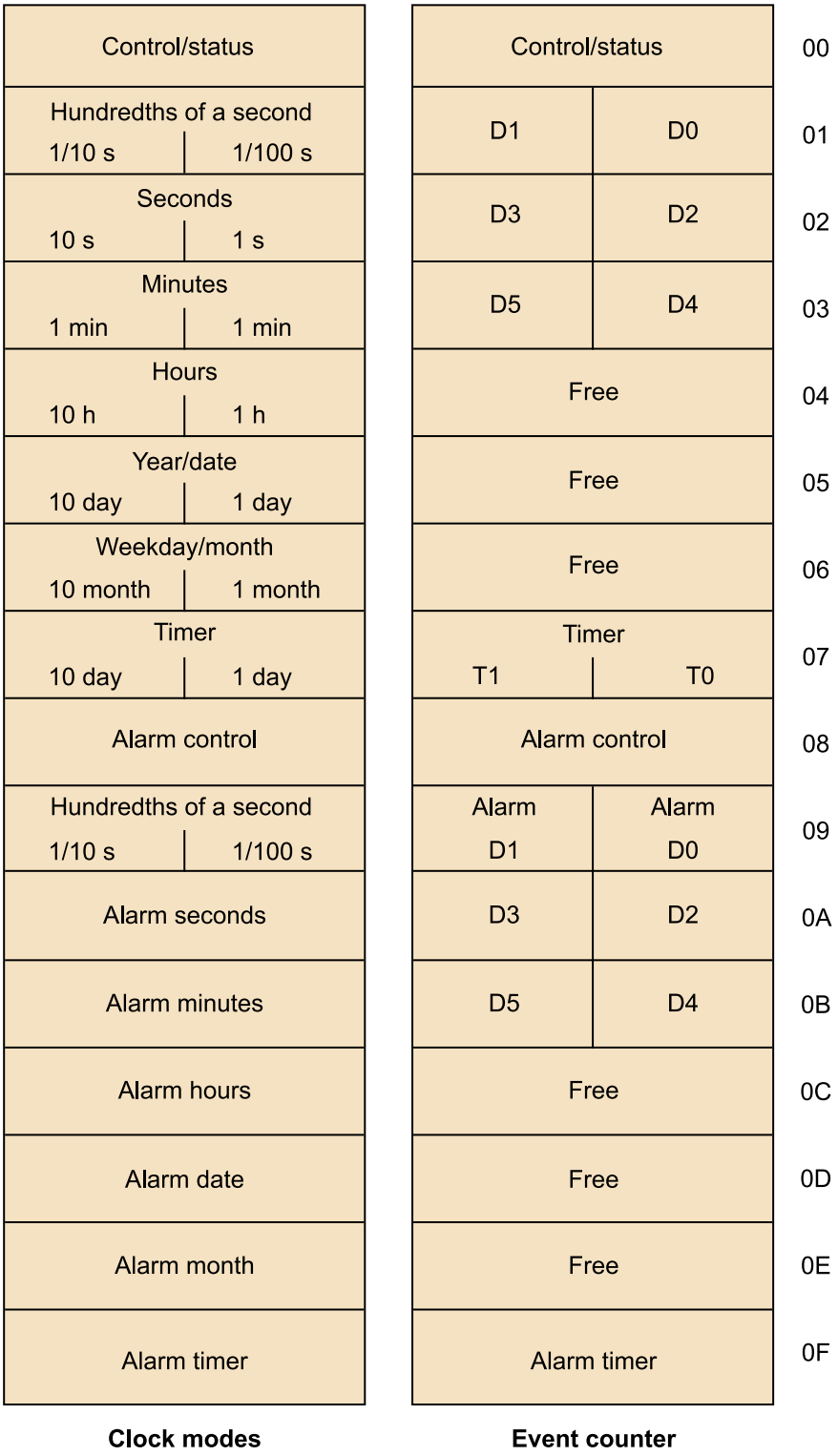
1) La comunicació amb el microcontrolador es basa en un protocol I^2C[1]

(1)*I2C-bus specification and user manual*, NXP. Vegeu: http://www.nxp.com/documents/user_manual/UM10204.pdf

2) Mitjançant una capacitat gran, pot continuar funcionant, en el cas que es perdi l'alimentació en un moment determinat.

Evidentment, cal estudiar el format de trama que enviarà l'I^2C per a poder establir com s'enviaran i es rebran les dades. Per a això, mirem el full de característiques i veiem que té els registres següents que podem modificar (vegeu la figura següent).

Registres del PCF8593



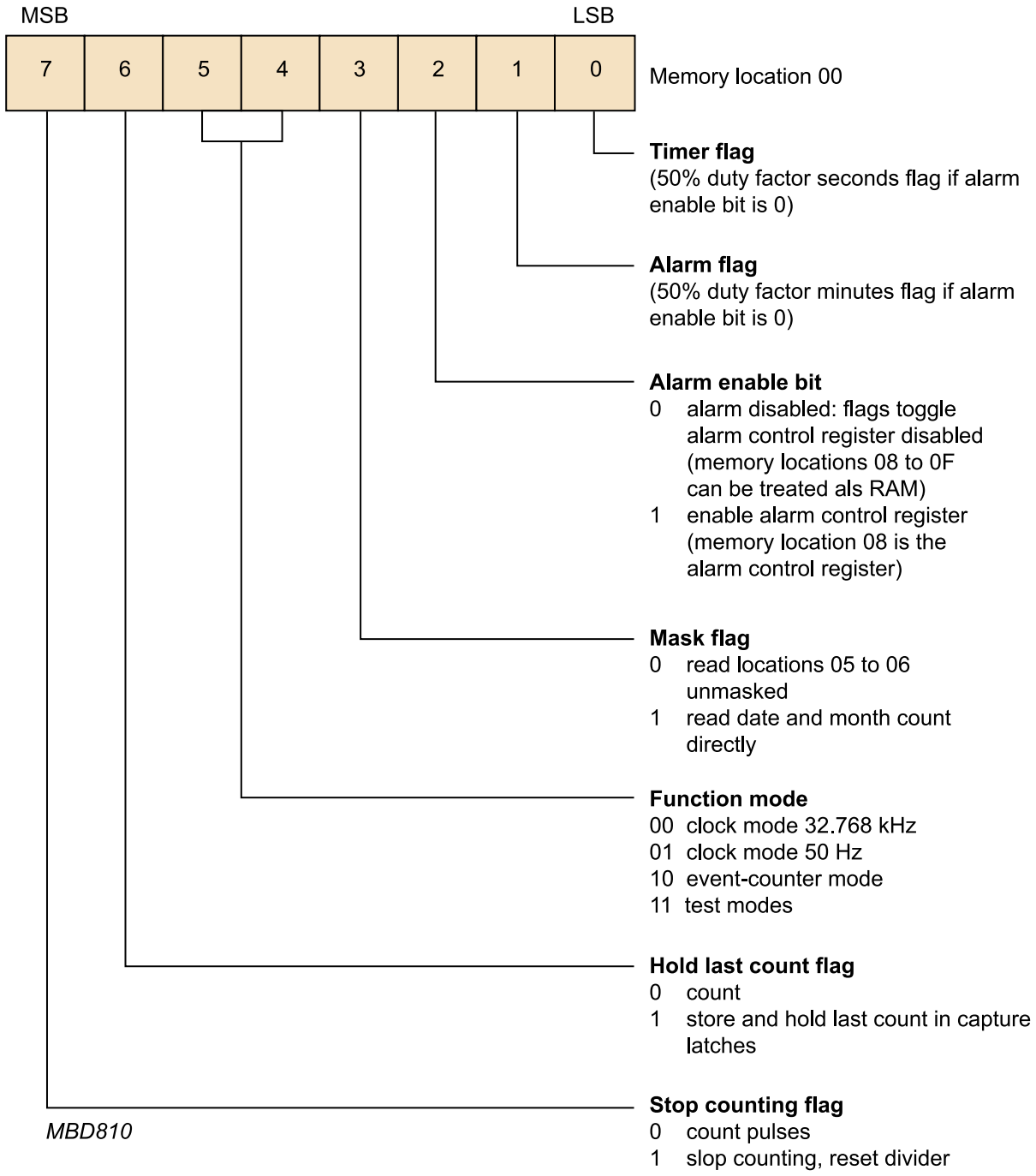
El dispositiu té 16 registres. El zero és de configuració i estat. En funció de la configuració, pot treballar en diferents modes:

- Rellotge basat en un oscil·lador a 32.768 Hz.
- Rellotge basat en un oscil·lador a 50 Hz.
- Comptador d'esdeveniments.

En el nostre cas, treballarem en el mode del rellotge, per la qual cosa ens concentrarem en aquest. Veiem que en els modes de rellotge, del registre 1 al 7 indiquen el moment actual, en decimals codificats en binari (*binary coded decimal*, BCD). Del 8 al 15 permeten establir l'alarma, que nosaltres no utilitzarem.

Per a configurar-lo, hem de tenir en compte el registre 0. Els diferents bits d'aquest registre permeten establir els modes de treball. La seva descripció es mostra a continuació.

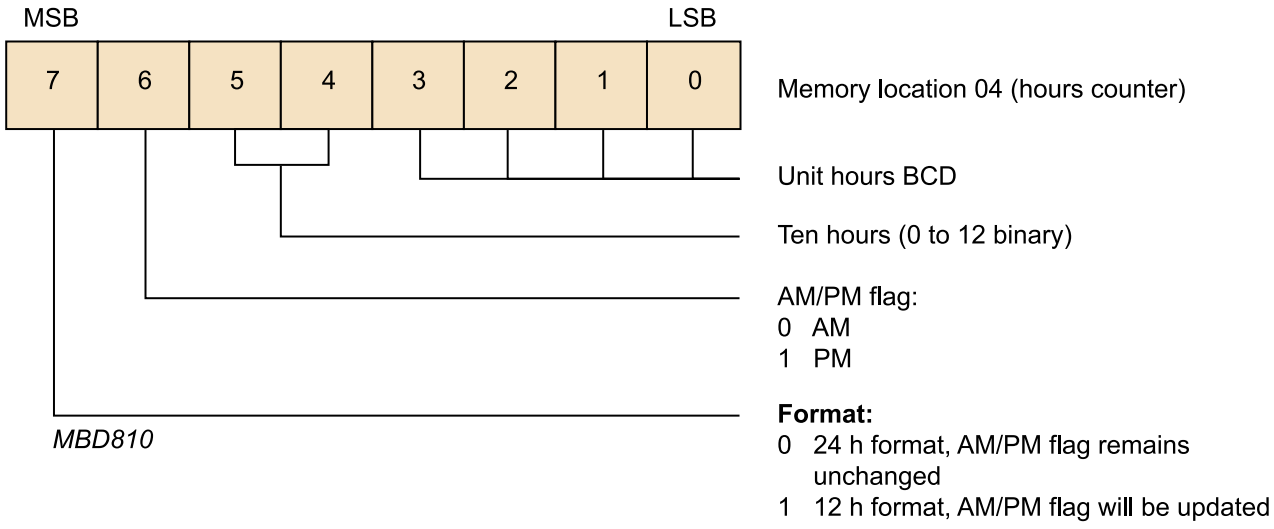
Registre de configuració del PCF8593



Els bits 0 i 1 indiquen l'estat del temporitzador (*timer*) o de l'alarma (*alarm*). Veiem que el mode de funcionament el defineixen els bits 4 i 5. Suposant que treballem amb un rellotge de 32.768 Hz, veiem que hem de posar sengles bits a 0. També veiem que si no volem fer ús de l'alarma, hem de posar el bit 2 a 0. El bit de màscara ens permet decidir si veiem tota la informació dels registres 5 i 6, o bé veiem únicament la informació del dia i el mes. Tenint en compte que volem tota la informació, el posarem a 0. El mateix farem amb el bit de mantenir, amb el bit de stop.

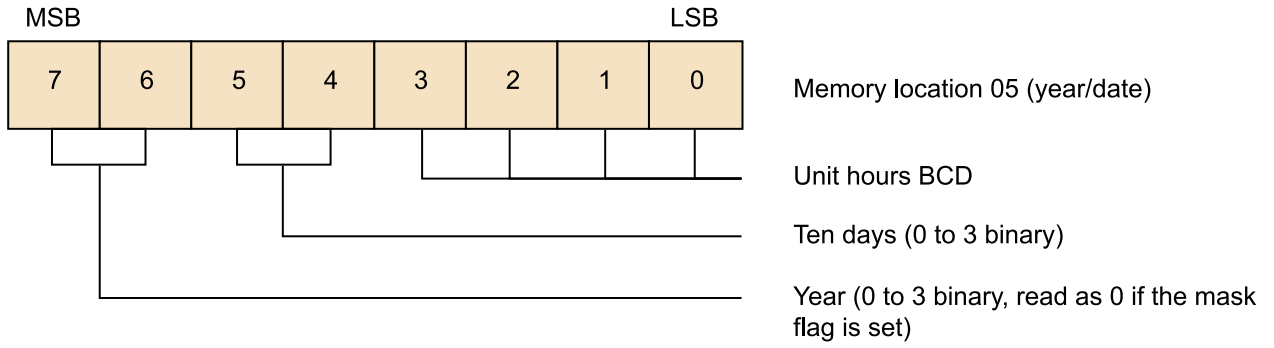
Per a posar en hora l'integrat, hem de tenir en compte les característiques especials dels registres 4, 5 i 6 (vegeu les tres figures següents, respectivament).

Funcionalitat del registre 4 del PCF8593



Es pot modificar el format del registre entre 12 i 24 hores canviant el bit 7. En aquest cas, treballarem en format 24, per la qual cosa el bit haurà de ser posat a 0. En aquest format, el bit 6 no modificarà el seu estat i, per tant, el registre el podem llegir i escriure com un registre BCD.

Funcionalitat del registre 5 del PCF8593

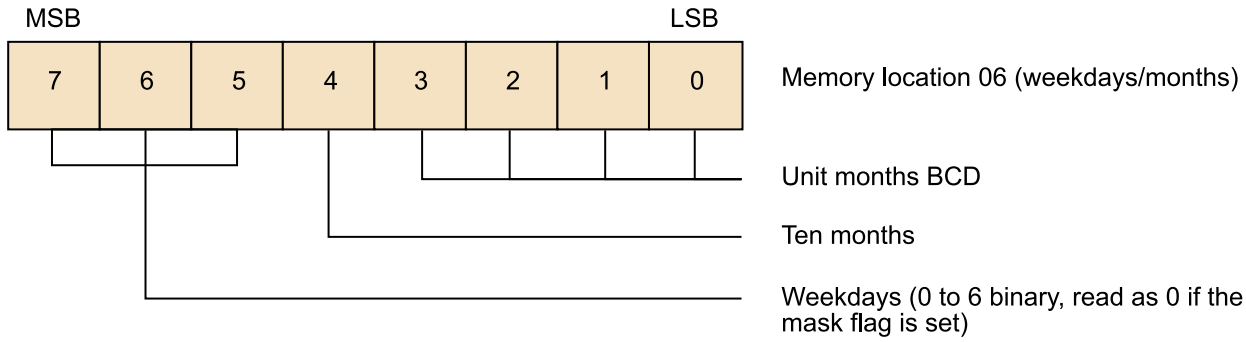


Aquest registre proporciona el dia del mes en els bits 0-5 en BCD. En els dos bits més alts, trobem el valor de l'any dins d'un cicle d'any de traspàs. Això permet determinar el nombre de dies del mes de febrer. A l'hora d'inicialitzar el valor d'aquest registre, ho haurem de tenir en compte.

Exemple

Si és l'any 2010, podem calcular el mòdul 4 de l'any, que ens dóna el valor 2. En aquest cas, posarem un 1 en el bit 7 i un 0 en el 6.

Funcionalitat del registre 6 del PCF8593



Finalment, el registre 6 indica alhora el mes, bits 0-4, en BCD. També inclou el dia de la setmana, bits 5-7. Si volem gravar la data "Diumenge, 2011-05-29 19:52:23.87" haurem d'escriure els valors indicats en la taula següent:

Valors d'inicialització dels registres del PCF8593

Valor	Registre	Bits	Valor hexadecimal	Valor binari	Comentaris
Diumenge	6	7-5	7	111	
2011	5	7-6	3	11	2011%4
05	6	4-0	05	0 0101	
29	5	5-0	29	10 1001	
19	4	5-0	19	01 1001	
52	3	7-0	52	0101 0010	
23	2	7-0	23	0010 0011	
87	1	7-0	87	1000 0111	

Per tal de calcular el valor final del registre 5, ho podem fer de manera senzilla amb la fórmula següent:

$$\text{Registre 5} = (\text{valor}_{\text{any}} << 6) \mid (\text{valor}_{\text{dia}})$$

En aquest cas, el resultat del registre 5 és 1110 1001 en binari, o I9 en hexadecimal. A partir d'aquests valors, podem inicialitzar l'RTC enviant la següent trama I^2C.

Trama I^2C inicialització del PCF8593

Binari (BCD)		HEX	Registre	Comentaris
				Generar condició d'inici I^2C
1010	0010	A2		Adreça de l'esclau I^2C, escriptura
0000	0000	00		Adreça 0, resta de bytes són dades
0000	0000	00	00	Control/estatus 1, es deixen tots els bits a 0
1000	0111	87	01	87 centenes de segon
0010	0011	23	02	23 segons
0101	0010	52	03	52 minuts
0001	1001	19	04	19 hores
1110	1001	I9	05	Tercer any del cicle de traspàs, dia 29
1100	0101	C5	06	Diumenge, mes 5
0000	0000	00	07	Deshabilitem l'alarma i el temporitzador
				Generar condició d'aturada I^2C

Una vegada enviada aquesta trama, l'integrat s'ha inicialitzat de manera adequada. Si volem llegir els valors de minuts i hores, simplement haurem d'enviar la capçalera corresponent d'I^2C.

Trama I^2C de lectura de minuts i hores per al PCF8593

Binari (BCD)		HEX	Registre	Comentaris
				Generar condició d'inici I^2C
1010	0010	A3		Adreça de l'esclau I^2C, lectura
0000	0000	03		Adreça 03, resta de bytes són dades
XXXX	XXXX	XX	03	Minuts
XXXX	XXXX	XX	04	Hores
				Generar condició d'aturada I^2C

Com que la trama és de lectura, s'indica amb X els valors que es rebran, però que no es coneixen. Suposant que hagin passat 15 minuts des que programem l'integrat, rebrem el valor hexadecimal 20 per a l'hora i 07 per als minuts.

Una vegada coneguda la manera de treballar amb el dispositiu, hem de veure com ens hi comuniquem. Per a això, hem de mirar els perifèrics del microcontrolador disponibles. Si té un controlador d'I^2C, el procés se simplifica. En cas contrari, es poden programar les entrades i sortides de propòsit general per a aconseguir-ho, encara que se n'incrementa lleugerament la complexitat.

Un dels paràmetres importants que cal tenir en compte és la velocitat màxima permesa per l'RTC per a comunicar-nos per l'I^2C. En el full de característiques, podem veure que es denomina f_{SCL} i té com a valor màxim 100 kHz. Per tant, hem de limitar la transmissió a aquesta velocitat, si volem que el circuit rebi de manera correcta la informació.

5.2. Programació del controlador

D'acord amb el que s'acaba de dir, podem escriure el programa per tal de comunicar-nos amb l'RTC. En aquest cas, hem de tenir en compte que treballem amb dues capes: la de menys nivell és el controlador per a l'I^2C, mentre que la segona és la de l'RTC. Assumim que l'RTC només l'utilitzarem en casos concrets, atès que en el microcontrolador ja tenim un rellotge de sistema, que pot anar incrementant la data. L'RTC l'utilitzarem quan es torni a iniciar el sistema, perquè hagi estat aturat o per qualsevol altre motiu que pugui deixar el rellotge del sistema desconfigurat. En aquest sentit, l'accés a l'RTC no farà ús d'interrupcions i, per tant, el podem programar com un conjunt de mètodes per a ser utilitzats per l'aplicació.

Comencem creant l'estructura internalClk, en què guardarem els valors del rellotge, en centenes de segon, encara que podria ser superior si ens interessés, i el rellotge del sistema ho permetés.

```
struct date_t{
      uint8_t cs;
      uint8_t sec;
      uint8_t min;
      uint8_t hour;
      uint8_t day;
      uint8_t month;
      uint16_t year;
      uint8_t weekday;
} internalClk;
```

Seguidament, creem un mètode que permet convertir el contingut de l'estructura del rellotge intern al format de trama I^2C per a poder enviar les dades a l'RTC. Suposem que al mètode li arriba un vector de la mateixa grandària que els registres que té l'RTC.

```
#define REGISTER_NUMBER ((uint8_t) 0x10)
```

```
void generateFrameData(struct date_t* date, uint8_t values[REGISTER_NUMBER]) {
      values[0] = 0;
      values[1] = ((date->cs / 10) << 4) | (date->cs
                  % 10);
      values[2] = ((date->sec / 10) << 4) | (date->sec
                  % 10);
      values[3] = ((date->min / 10) << 4) | (date->min
                  % 10);
      values[4] = ((date->hour / 10) << 4)
                        | (date->hour % 10);
      values[5] = ((date->day / 10) << 4) | (date->day
                  % 10) | ((date->year % 4) << 6);
      values[6] = ((date->month / 10) << 4)
                | (date->month % 10) | (date->weekday << 5);
}
```

De la mateixa manera, creem un mètode per a fer el pas invers.

```
void restoreFrameData(uint8_t values[REGISTER_NUMBER], struct date_t* date) {
      uint8_t hour12;
      uint8_t temp;

      temp = ((values[1] >> 4) * 10);
      temp += (values[1] & 0x0F);
      date->cs = temp;

      temp = ((values[2] >> 4) * 10);
      temp += (values[2] & 0x0F);
      date->sec = temp;

      temp = ((values[3] >> 4) * 10);
      temp += (values[3] & 0x0F);
      date->min = temp;

      if ((values[4] & 0x80) != 0) {
            hour12 = values[4] & 0x1F;

            temp = ((hour12 >> 4) * 10);
            temp += (hour12 & 0x0F);

            if ((values[4] & 0x40) != 0) {
                  temp += 11;
            } else {
                  temp -= 1;
            }
            date->hour = temp;
```

```
        } else {
              temp = ((values[4] >> 4) * 10);
              temp += (values[4] & 0x0F);
              date->hour = temp;
        }

        temp = (((values[5] & 0x30) >> 4) * 10);
        temp += (values[5] & 0x0F);
        date->day = temp;

        temp = (values[5] >> 6);
        date->year = temp;

        temp = (((values[6] & 0x10) >> 4) * 10);
        temp += (values[6] & 0x0F);
        date->month = temp;

        temp = (values[6] >> 5);
        date->weekday = temp;
  }
```

Per al mètode d'enviament de la trama, se segueix un esquema semblant al de la taula "Trama I^2C inicialització del PCF8593". En primer lloc, es fa la conversió del format rellotge intern al de dades de l'RTC amb generateFrameData. Seguidament, comencem la transmissió de la trama. Per a això, en primer lloc enviem una marca d'inici d'I^2C. A continuació, enviem l'adreça d'escriptura de l'RTC i l'adreça del primer registre al qual volem accedir (en aquest cas, el 0). Posteriorment, enviem les dades per a programar el rellotge. Finalment, enviem una marca de finalització.

```
  void writeClk(struct date_t* date) {
        uint8_t data[REGISTER_NUMBER];
        int i;

        generateFrameData(date, data);

        startCondition();
        putByte(0xA2);
        putByte(0x00);

        for (i = 0; i < REGISTER_NUMBER; i++) {
              putByte(data[i]);
        }

        stopCondition();
```

```
}
```

Per a la recepció, l'esquema és el de la taula "Trama I^2C de lectura de minuts i hores per al PCF8593". Comencem la transmissió de la trama enviant, en primer lloc, una marca d'inici d'I^2C. A continuació, enviem l'adreça de lectura de l'RTC i l'adreça del primer registre al qual volem accedir, en aquest cas el 0. Posteriorment, llegim les dades per a programar el rellotge. Finalment enviem una marca de finalització. Una vegada rebuda tota la trama, processem les dades desant-les en una estructura de tipus date_t com la del rellotge intern.

```
void readClk(struct date_t* date) {
      uint8_t data[REGISTER_NUMBER];
      int i;

      startCondition();
      putByte(0xA3);
      putByte(0x00);

      for (i = 0; i < REGISTER_NUMBER; i++) {
            data[i] = getByte();
      }

      stopCondition();

      restoreFrameData(data, date);
}
```

En el main, estem suposant que estem emulant en un sistema Posix. En primer lloc, inicialitzem l'I^2C. Seguidament, mirem l'hora actual del sistema (si fos un microcontrolador, aquesta línia no s'utilitzaria per motius evidents). El pas següent és inicialitzar el rellotge intern. Imprimim el contingut del rellotge en pantalla i enviem les dades a l'RTC per I^2C. Una vegada fet això, fem un bucle d'espera d'uns deu segons. Tenint en compte que és possible que ens despertin abans que passin els deu segons, comprovem el temps que ha passat i, si no ha arribat, es torna a esperar.

Una vegada efectuada aquesta acció, llegim el contingut de l'RTC, que hem mostrat en pantalla, i comparem el resultat amb el del rellotge intern. Una vegada finalitzat, s'acaba l'aplicació.

```
int main(void) {
      time_t startTime, endTime;

      i2cInit();
      startTime = time(NULL);
```

```
        internalClk.cs = 0;
        internalClk.sec = 23;
        internalClk.min = 52;
        internalClk.hour = 19;
        internalClk.day = 29;
        internalClk.month = 5;
        internalClk.year = 2011;
        internalClk.weekday = 6;

        printf("%d, %d/%d/%d %d:%d:%d.%d\n", internalClk.weekday,
                    internalClk.year, internalClk.month, internalClk.day,
                    internalClk.hour, internalClk.min, internalClk.sec, internalClk.cs);

        writeClk(&internalClk);

        do {
              sleep(2);
              endTime = time(NULL);
        } while((endTime-startTime) &lt; 10);

        readClk(&internalClk);

        printf("%d, %d/%d/%d %d:%d:%d.%d\n", internalClk.weekday,
                    internalClk.year, internalClk.month, internalClk.day,
                    internalClk.hour, internalClk.min, internalClk.sec, internalClk.cs);

        printf("Start: %ld\n", startTime);
        printf("End: %ld\n", endTime);
        printf("Delta: %ld\n", endTime-startTime);

        return EXIT_SUCCESS;
  }
```

D'aquesta manera, podem intercanviar informació amb l'RTC. En l'annex, es troba el codi per a poder emular en un entorn Posix.

Activitat

Modificar el mètode restoreFrameData perquè corregeixi l'any i tingui en compte l'any actual, i s'aconsegueixi així que no es perdi el valor en rebre les dades de l'RTC.

6. Sistemes operatius i biblioteques de suport

A continuació, es descriuen alguns dels SO i entorns de treball més comuns dins d'aquest àmbit.

6.1. TinyOS

TinyOS és un sistema operatiu lliure i de codi obert basat en components i orientat a plataformes per a xarxes de sensors sense fil (WSN). TinyOS està pensat per a sistemes encastats i es programa en llenguatge nesC, com un conjunt de tasques i processos cooperatius.

El nesC és un dialecte del C, optimitzat per a treballar amb microcontroladors amb pocs recursos de memòria. Es basa en components que podrien ser considerats com a objectes. Aquests proporcionen abstraccions dels dispositius disponibles, de manera equivalent als controladors de dispositius d'un SO de sobretaula.

TinyOS facilita un entorn de desenvolupament basat en línia d'ordres. Aquest entorn integra els diferents components escrits en nesC i els modifica per a generar un únic fitxer en C. Aquest fitxer incorpora el codi dels diferents components, entre els quals s'inclou el gestor de tasques. El programa pot ser compilat de manera creuada fent ús de GCC.

El gestor de tasques està basat en esdeveniments i no incorpora mecanismes predictius. D'aquesta manera, es minimitzen els requeriments de memòria. Tenint en compte que els esdeveniments associats a E/S solen estar relacionats amb interrupcions, és important dur a terme la divisió entre l'esdeveniment i una tasca associada pels motius indicats en el subapartat "Sistemes operatius basats en esdeveniments".

També hi ha un gestor de tasques anticipatiu, denominat TOSThreads. Aquest fa ús de dues cues, tal com s'explica en el subapartat "Accés al nucli" (que pertany al bloc "Sistemes operatius multiàrea"). Aquest permet carregar programes de manera dinàmica, en poder tenir disponible un nucli de TinyOS en la plataforma, si es considera necessari.

El gran avantatge de TinyOS és el gran nombre de plataformes disponibles, i també codi i documentació, la qual cosa redueix la corba d'aprenentatge.

6.2. FreeRTOS

FreeRTOS és un sistema operatiu en temps real per a dispositius encastats. Hi ha adaptacions per a diferents microcontroladors. Es distribueix sota llicència GPL amb una excepció que permet "codi propietari als usuaris que continuen essent de codi tancat, de manera que manté el nucli en ell mateix com de codi obert", circumstància que facilita l'ús de FreeRTOS en aplicacions propietàries.

FreeRTOS està dissenyat per a ser petit i simple. El nucli en si consta de tres o quatre arxius en C. Per a fer llegible el codi i simplificar l'adaptació a noves plataforma, i el seu manteniment, està escrit fonamentalment en C. Hi ha algunes rutines en assemblador, que depenen de l'arquitectura del microcontrolador (canvis de context, mútex...). Les fonts que es poden descarregar contenen configuracions preparades i demostracions per a totes les adaptacions del compilador, la qual cosa permet el disseny ràpid d'aplicacions.

Enllaç recomanat

El web http://www.freertos.org també disposa de programes d'aprenentatge per al seu ús, detalls del seu disseny, i també comparatives de rendiment en diferents plataformes.

En funció de la programació, permet treballar de manera anticipativa, o col·laborativa, per la qual cosa és una bona manera d'endinsar-se en aquest tipus de sistemes operatius.

6.3. Contiki

Contiki es defineix com un sistema operatiu de codi obert, multitasca i fàcilment portable, per a sistemes encastats en xarxa i xarxes de sensors sense fil. Està concebut per a microcontroladors amb una capacitat de memòria reduïda. Una configuració normal de Contiki requereix 2 kB de RAM i 40 kB de ROM.

Contiki és desenvolupat per persones tant de l'àmbit industrial com de l'acadèmic. El dirigeix Adam Dunkels, de l'Institut suec de Ciències de la Computació. Aquest coordina un equip format per membres de SICS, SAP, Cisco, Atmel, NewAE i TU Munich.

Com a sistema operatiu, està basat en esdeveniments, encara que suporta múltiples fils (*threads* o tasques segons la nostra nomenclatura); per a això utilitza un planificador. També permet utilitzar el que denomina *protofils* (*protothreads*), que són una implementació simplificada, que no requereix una pila per a cada fil, ja que cada tasca s'ha de finalitzar abans de continuar amb la següent.

Contiki permet seleccionar per a cada procés si s'utilitzen múltiples. També permet intercanviar informació entre processos mitjançant missatges. Finalment, inclou la possibilitat que l'usuari accedeixi al sistema mitjançant línia d'ordres per Telnet, o de manera gràfica, fent ús del protocol Virtual Network Computing (VNC).

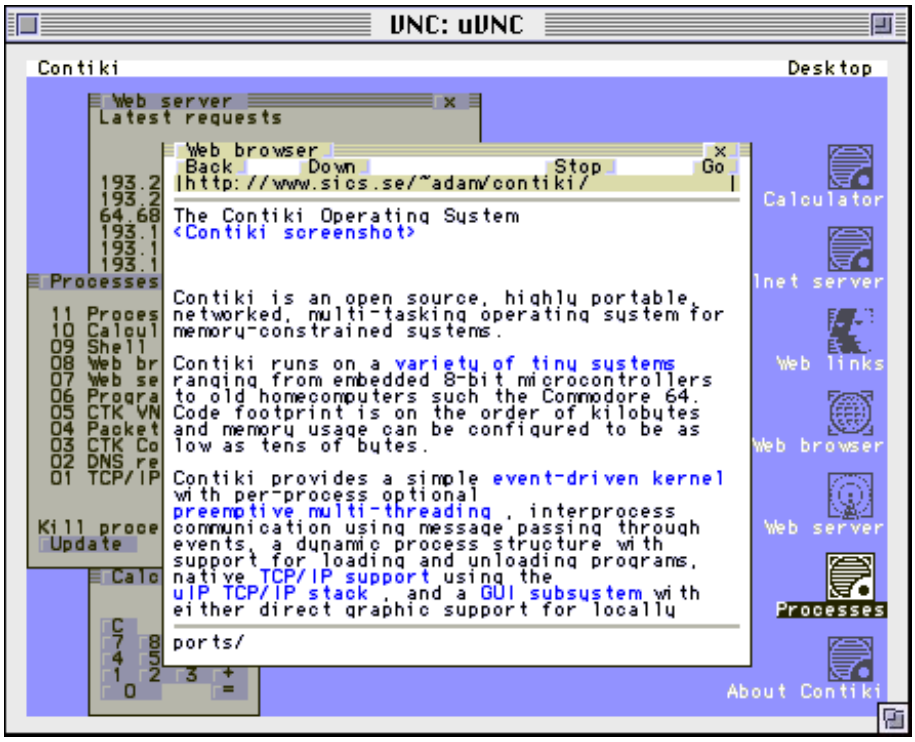
Una instal·lació completa de Contiki inclou les característiques següents:

- Nucli multitasca.

- Subscripció opcional per tasca a multifil.
- *Protothreads*.
- Protocol de xarxa TCP/IP, inclòs IPv6.
- Interfície gràfica per a l'usuari i sistema de finestres.
- Permet accedir-hi mitjançant Virtual Network Computing.
- Un navegador web (que deia que era el més petit del món).
- Servidor de web personal.
- Client senzill de Telnet.
- Protector de pantalla.

S'ha transferit a múltiples arquitectures, inclosa l'AVR d'Atmel. En la figura següent es mostra l'aspecte d'aquest SO en un terminal VNC.

Aspecte de Contiki en un terminal VNC corrent sobre un AVR



Com es pot observar, el resultat és notable, i treballa sobre un petit microcontrolador.

6.4. QNX

QNX és un SO Unix comercial en temps real, dirigit fonamentalment a sistemes encastats comercials. El producte va ser dissenyat originalment per QNX Software Systems, companyia que va ser comprada per Research In Motion (els fabricants dels telèfons BlackBerry).

QNX s'autodefineix com l'únic SO veritablement basat en micronuclis. El motiu adduït és que el nucli de QNX únicament conté el gestor de processos, comunicació entre processos, redirecció d'interrupcions i temporitzadors. La resta de processos del SO corre en mode usuari, que inclou el procés que crea processos nous i gestiona la memòria (proc), i treballa juntament amb el nucli.

Els processos del SO es coneixen com a **serveis**. La seva utilització permet una gran modularitat, en poder activar o desactivar aquesta funcionalitat executant, o no, el servei corresponent. És més: si apareix un problema en un servei, el seu procés es reinicia, sense afectar el micronucli.

QNX Neutrino ha estat adaptat a una sèrie de plataformes i ara funciona en pràcticament qualsevol CPU que s'utilitza al mercat encastat. Això inclou el PowerPC, la família x86, MIPS, SH-4 i els relacionats amb la família ARM.

La interfície de treball amb les aplicacions és Posix, la qual cosa simplifica l'adaptació des d'altres sistemes que compleixin aquest estàndard. Concretament, el perfil 52[2].

(2)IEEE Std 1003.13-2003, IEEE Standard for Information Technology – Standardized Application Environment Profile (AEP) – POSIX® Realtime and Embedded Application Support.

6.5. RTEMS

El sistema executiu multiprocessador en temps real[3] és un sistema operatiu en temps real de codi lliure i obert, dissenyat per a sistemes encastats.

(3)En anglès, *real-time executive multiprocesor system* (RTEMS).

Les sigles *RTEMS* inicialment representaven un sistema executiu per a míssils en temps real, que va passar amb el temps a un ser sistema executiu per a l'àmbit militar en temps real, abans de canviar al significat actual. El desenvolupament de RTEMS va començar al final de 1980, i en van aparèixer les primeres versions disponibles per a FTP el 1993. OAR Corporation és actualment la gestora del projecte RTEMS, en cooperació amb un comitè directiu que inclou representants dels usuaris.

RTEMS està dissenyat per a suportar diversos estàndards oberts, inclosos l'API Posix i l'uITRON. L'API, coneguda actualment com l'API clàssica de RTEMS, es va basar originalment en l'especificació de la definició d'interfície per a temps real executiva (RTEID). RTEMS inclou un port de la pila TCP/IP FreeBSD. També disposa de suport per a diversos sistemes de fitxers, que inclou NFS i el sistema d'arxius FAT.

RTEMS no proporciona cap tipus de gestió de memòria o processos. En la terminologia Posix, implementa un únic procés, que pot contenir diversos fils o tasques. En aquest sentit, s'assumeix un únic processador que no disposa d'unitat de gestió de memòria. Així mateix, també inclou un sistema de fitxers i accés asíncron a perifèrics, per la qual cosa entroncaria directament amb el perfil 52 de Posix, com el QNX.

RTEMS s'utilitza per a moltes aplicacions. La comunitat Experimental Physics and Industrial Control System hi col·labora de manera continuada; també és molt utilitzat en projectes per a l'espai, ja que proporciona suport per a la major part de les arquitectures utilitzades en aquest entorn.

RTEMS es distribueix sota una modificació de la llicència GPL, la qual cosa permet la vinculació d'objectes RTEMS amb altres arxius sense necessitat que els últims siguin GPL, de manera semblant a FreeRTOS. Aquesta llicència es basa en l'actualització GNAT de GPL amb una modificació perquè no siguin específiques per al llenguatge de programació Ada.

Resum

El mòdul ha presentat, des d'un punt de vista pràctic, les necessitats de la gestió de processos i abstraccions de sistema operatiu per a entorns encastats. Aquest bloc ens ha guiat per mitjà d'exemples cap a les tècniques més rellevants per a la gestió de processos en entorns de propòsit específic. Des de les tècniques més senzilles, basades en cues *round-robin*, fins a sistemes preemptius o de temps real, hem vist mostres de com programar aquestes abstraccions, pensant en les restriccions pròpies d'un sistema de propòsit específic. D'altra banda, s'han vist els beneficis d'una programació orientada a esdeveniments, fent ús de les interrupcions maquinari i les particularitats que han de tenir les rutines de servei a la interrupció. El mòdul també ens ha presentat els sistemes operatius més utilitzats actualment en entorns encastats. Podem destacar, entre d'altres, FreeRTOS i TinyOS com dos exponents amb un ús bastant estès. Finalment, a manera de guia d'aprenentatge, el mòdul ens ha endinsat en la creació de controladors de maquinari o *drivers* a partir de l'esquema de característiques (*datasheet*) d'un controlador de rellotge en temps real (RTC).

Bibliografia

Barr, M.; Oram, A. (ed.) (1998). *Programming Embedded Systems in C and C++* (1a. ed.). Sebastopol, Califòrnia: O'Reilly and Associates.

Kamal, R. (2008). *Embedded Systems: Architecture, Programming and Design*. Nova York: McGraw-Hill.

Labrosse, J. J. (2000). *Embedded Systems Building Blocks* (2a. ed.). San Francisco, Califòrnia: CMP Books.

Annex

Per tal de poder provar de manera senzilla els diferents exemples, incloem unes modificacions que permeten comprovar aquest programa en un ordinador de sobretaula amb una plataforma Posix. Les modificacions que es requereixen estan fonamentalment orientades a les entrades i les interrupcions.

1) Gestor Round-Robin

El primer exemple és el del lector Round-Robin. El primer element necessari és la lectura del teclat. S'utilitza un mètode sense bloqueig, de manera que l'aplicació continuï treballant, de manera equivalent a com funcionaria en un microcontrolador. Per a això modifica el comportament del terminal Posix:

```
ssize_t kbread(void *buf, size_t count) {
      struct termios oldt, newt;
      int ch;
      ssize_t length;

      // Reading the actual terminal attributes
      tcgetattr(STDIN_FILENO, &oldt);
      newt = oldt;
      // Modifying the attributes to avoid blocking the stdin
      newt.c_lflag &= ~(ICANON | ECHO);
      // Modifying the terminal parameters
      tcsetattr(STDIN_FILENO, TCSANOW, &newt);
      // Reading the stdin stream
      length = read(STDIN_FILENO, buf, count);
      // Returning to the initial terminal attributes
      tcsetattr(STDIN_FILENO, TCSANOW, &oldt);

      // Returning the number of read keys
      return length;
```

Ara, només cal adaptar el mètode de lectura dels botons per a treballar amb aquesta funció. Per a això, modifiquem el codi de getButtons.

```
void getButtons(struct buttons_t *buttons) {
      /*
      * int data = IOPORT;
      *
      * if ((data & UP_BUTTON) != 0) buttons->up = true;
      * else buttons->up = false;
      * if ((keys & DOWN_BUTTON) != 0) buttons->down = true;
```

```
        * else buttons->down = false;
        * if ((keys & MODE_BUTTON) != 0) buttons->mode = true;
        * else buttons->mode = false;
        */

        // Keys buffer size
        const int keysLength = 80;
        // Keys buffer definition
        unsigned char keys[keysLength];
        // Button value
        int button;
        // Pointer
        int i;
        // Keys buffer length result
        ssize_t length;

        // Get the available keys
        length = kbread(keys, keysLength);

        // Reset all the buttons
        buttons->up = false;
        buttons->down = false;
        buttons->mode = false;

        // Loop the buffer keys
        for (i = 0; i < length; i++) {
              // Copy to button the key
              button = keys[i];

              // Depending on the button set the
              // corresponding variable
              switch (button {
              case 'u':
                    buttons->up = true;
                    break;
              case 'd':
                    buttons->down = true;
                    break;
              case 'm':
                    buttons->mode = true;
                    break;
              }
        }
  }
```

El pas següent és la lectura de la temperatura. Aquesta l'emulem amb una sinusoide que varia en el temps. Per a això, definim primer les constants que defineixen la sinusoide:

```
const double TEMP_MEAN = 25.0;
const double TEMP_AMP = 5.0;
const int TEMP_FREQUENCY = 10;
```

D'acord amb això definim la funció que ens simula les variacions de temperatura:

```
uint16_t getADCValue(void) {
      // Seconds from the Epoch
      time_t seconds;
      // Simulated temperature
      double tBase;
      // Simulated ADC value
      double vADC;
      // Resulting function value
      uint16_t result;

      // Get the clock value and make the module to avoid sin function saturation
      seconds = clock() % frequency;
      // Temperature evolution function
      tBase = TEMP_MEAN + TEMP_AMP *
                  sin( (2.0 * M_PI * seconds) / TEMP_FREQUENCY );

      // ADC value based on the sensor function
      vADC = 18.43 * tBase + 1126.125;

      // Rounding to get the integer value
      result = lround(vADC);
      return result;
}
```

Seguidament, modifiquem getTemp per utilitzar, en lloc d'un registre, el resultat de la funció anterior.

```
int16_t getTemp(void) {
      uint16_t adcValue16 = getADCValue();
      int32_t adcValue32 = adcValue16;
      int16_t result;
      adcValue32 <<= 5;
      adcValue32 /= 59;
      result = (int16_t) (adcValue32 - 610);

      return result;
```

```
}
```

Amb això, ja és possible simular el comportament de l'aplicació bàsica.

2) Gestor síncron Sound-Robin

Per al gestor síncron, cal incloure-hi l'emulació d'interrupcions, a més de mètodes de gestió del consum. Comencem per aquests últims, i en concret, el de lowPowerMade. Per a això, fem ús del mètode pause, el qual ens permet deixar-la adormida. Aquest mètode s'ha d'utilitzar amb cura, ja que deixa completament parat el mètode on es troba, i només un senyal del sistema operatiu la pot continuar.

```
inline void lowPowerMode(void) {
      pause();
}
```

A més, necessitem crear el mètode standardMode que, en aquest cas, no fa cap acció, però que en un microcontrolador probablement inclouria alguna instrucció.

```
inline void standardMode(void) {
      ;
}
```

Per a generar la interrupció de rellotge, necessitem fer uns passos previs. El motiu es deu al fet que, per a treballar amb senyals, que seria l'equivalent de les interrupcions en un sistema Posix, necessitem un format específic de mètode. No obstant això, en el nostre cas, n'estem utilitzant un més propi dels microcontroladors. Per aquest motiu, hem de crear un mètode "contenidor" que cridi el nostre mètode interrupció. Per fer-ho, definim el nostre propi tipus de mètode (interrupt_t), una variable global amb aquest tipus, per poder-hi accedir des de diferents mètodes (clock_handler), i el mètode contenidor (clockHandler), el qual únicament crida el mètode indicat per clock_handler. Els diferents elements es mostren a continuació.

```
typedef void (*interrupt_t)(void);

interrupt_t clock_handler;

void clockHandler(int signum) {
      clock_handler();
}
```

Seguidament, hem de donar d'alta i de baixa la interrupció de rellotge (en el cas de Posix, el senyal del rellotge). Per tant, creem, en primer lloc, una estructura en què guardarem el mètode que té actualment assignat el senyal (old_action).

```
struct sigaction old_action;
```

Seguidament, creem el mètode per donar d'alta la interrupció (startInterruptHandler) per tal d'assignar el mètode d'interrupció, i també indicar el període en què s'ha de dur a terme. Per a això, hem de fer diferents passos i, com que estem accedint a punts crítics del SO, fer-ho amb totes les precaucions necessàries. El primer pas és mirar si hi ha alguna acció assignada al senyal que volem utilitzar –en el nostre cas, SIGALRM, ja que es comporta de manera equivalent a una interrupció d'un temporitzador en un microcontrolador. Si aquesta és diferent de SIG_IGN, la substituirem per la nostra interrupció. Com hem dit anteriorment, en aquest cas serà el mètode contenidor de la nostra interrupció, clockHandle.

Una vegada donat d'alta el mètode, engeguem la interrupció; assignem a la variable timerActualStatus el temps que ha de transcórrer fins al proper salt en l'estructura it_value. En el camp it_interval, indiquem el període en què es repetirà aquest senyal. Una vegada donats els valors, aquests són assignats al temporitzador ITIMER_REAL mitjançant el mètode setitimer.

```
void startInterruptHandler(interrupt_t handler, const uint32_t usperiod) {
      struct itimerval timerActualStatus;
      struct sigaction new_action;

      /* Specify the interrupt handler */
      clock_handler = handler;
      /* Set up the structure to specify the new action. */
      new_action.sa_handler = clockHandler;
      sigemptyset (&new_action.sa_mask);
      new_action.sa_flags = 0;`

      /* Read de old action */
      sigaction(SIGALRM, NULL, &old_action);

      /* If no action is expected, set the interrupt as the new action. */
      if (old_action.sa_handler != SIG_IGN) {
            sigaction(SIGALRM, &new_action, NULL);

            timerActualStatus.it_value.tv_sec = usperiod/1000000;
            //Conversion to microsecond
            timerActualStatus.it_value.tv_usec = usperiod%1000000;
            //Periodic Wake time
            timerActualStatus.it_interval.tv_sec = timerActualStatus.it_value.tv_sec;
            timerActualStatus.it_interval.tv_usec = timerActualStatus.it_value.tv_usec;
```

```
            /* Set the time period */
            setitimer(ITIMER_REAL, &timerActualStatus, NULL);
      }
}
```

Per a aturar el senyal, seguim el procés invers. Primer, aturem el temporitzador assignant valors nuls a it_value i it_interval i, seguidament, tornem a assignar l'antiga interrupció al senyal.

```
void stopInterruptHandler(void) {
      struct itimerval timerActualStatus;

      timerActualStatus.it_value.tv_sec = 0; //Stop the timer
      timerActualStatus.it_value.tv_usec = 0; //And no microsecond
      timerActualStatus.it_interval.tv_sec = 0; //Next wake up time would be 1 second
      timerActualStatus.it_interval.tv_usec = 0; //And no microsecond

      /* Stop the interrupt setting the time to 0 */
      setitimer(ITIMER_REAL, &timerActualStatus, NULL);
      /* Set the old action as the action that has to be done */
      sigaction(SIGALRM, &old_action, NULL);
}
```

Finalment, convé indicar que, en general, aquest últim mètode no serà utilitzat, ja que en un microcontrolador no hauríem d'aturar mai el rellotge principal. En qualsevol cas, l'incloem com a exemple, per si es vol utilitzar en un altre escenari.

3) Gestor basat en esdeveniments

Per al gestor basat en esdeveniments, cal incloure l'emulació d'interrupcions com en el gestor Round-Robin síncron. Per a això, creem una nova interrupció que mira si s'ha premut algun botó del teclat i, en aquest cas, modifica el valor de l'estructura associada amb setButtonsValue. A més, crida buttonsInterrupt per a emular una interrupció de botons. Al seu torn, en cada cicle de rellotge modifica el valor de l'ADC amb setADCValue i crida la interrupció de l'ADC.

```
void timeInterrupt(void) {
      if (kbhit()) {
            setButtonsValue();
            buttonsInterrupt();
      }

      setADCValue();
      adcInterrupt();
```

```
}
```

El mètode d'inici d'interrupció dóna d'alta la interrupció de temporitzador, que cridarà la rutina timeInterrupt.

```
void interruptInit(void) {
      startInterruptHandler(timeInterrupt, US_PERIOD); // 0.1 sec
}
```

A més, creem la funció getADCValue que llegeix el valor de la variable global adcRegister, i el mètode setADCValue que modifica de manera continuada el valor de la temperatura, de manera equivalent a getADCValue en el punt 1 d'aquest annex.

```
inline uint16_t getADCValue(void) {
      return adcRegister;
}

void setADCValue(void) {
      // Seconds from the Epoch
      time_t seconds;
      // Simulated temperature
      double tBase;
      // Simulated ADC value
      double vADC;
      // Resulting function value
      uint16_t result;
      // Temperature evolution frequency
      const int frequency = 10;

      // Get the clock value and make the module to avoid sin function saturation
      seconds = clock() % frequency;
      // Temperature evolution function
      tBase = TEMP_MEAN + TEMP_AMP * sin( (2.0 * M_PI * seconds) / TEMP_FREQUENCY);

      // ADC value based on the sensor function
      vADC = 18.43 * tBase + 1126.125;

      // Rounding to get the integer value
      result = lround(vADC);

      adcRegister = result;
}
```

Per als botons, seguim un esquema equivalent, encara que la funció setButtonsValue processa les possibles tecles (u, d, m) per a determinar quin bit de la variable buttonsRegister ha de modificar.

```
inline uint16_t getButtonsValue(void) {
      return buttonsRegister;
}

void setButtonsValue(void) {
      // Keys buffer size
      const int keysLength = 80;
      // Keys buffer definition
      unsigned char keys[keysLength];
      // Button value
      int button;
      // Pointer
      int i;
      // Keys buffer length result
      ssize_t length;

      // Get the available keys
      length = kbread(keys, keysLength);

      // Reset all the buttons
      buttonsRegister = 0;

      // Loop the buffer keys
      for (i = 0; i < length; i++) {
            // Copy to button the key
            button = keys[i];

            // Depending on the button set the
            // corresponding variable
            switch (button) {
            case 'u':
                  buttonsRegister |= (1 << UP_BUTTON);
                  break;
            case 'd':
                  buttonsRegister |= (1 << DOWN_BUTTON);
                  break;
            case 'm':
                  buttonsRegister |= (1 << MODE_BUTTON);
                  break;
            }
      }
}
```

Amb aquestes rutines és possible emular el sistema operatiu basat en esdeveniments en un ordinador basat en un SO Posix.

4) Controlador de perifèrics

En aquest cas, fem un emulador del PCF8593, per tal de poder-ne simular el comportament. Comencem definint el nombre de registres (REGISTER_NUMBER) i el temps de període (US_PERIOD).

```
#define REGISTER_NUMBER        ((uint8_t) 0x10)
#defineUS_PERIOD               ((uint32_t) 10000)
```

Seguidament, es defineix el conjunt d'elements que permeten emular el PCF8593. En aquest cas, els registres associats al temps amb les variables centèsima de segon (cs) a any (year). De la mateixa manera, el control de l'estat de la trama I^2C (framePosition), el mode de treball (read_write), els índexs i els *buffers* per a enviar i rebre dades.

```
volatile struct {
      struct {
            uint8_t cs;
            uint8_t sec;
            uint8_t min;
            uint8_t hour;
            uint8_t day;
            uint8_t month;
            uint8_t year;
            uint8_t weekday;
            bool h12_24;
      } RTCtime;
      enum {
            START, ADDRESS, REGISTER, DATA, ERROR
      } framePosition;
      bool read_write;
      uint8_t offset;
      uint8_t index;
      uint8_t regs[REGISTER_NUMBER];
      uint8_t latches[REGISTER_NUMBER];
} pcf8593;
```

Les funcions següents són còpies de les del gestor síncron, per la qual cosa només se n'indica el nom. Aquestes permeten llançar cada cert període de temps un mètode, semblant al procés que seguiria el comptador del PCF8593.

```
struct sigaction old_action;


typedef void (*interrupt_t)(int);
```

```
void startInterruptHandler(interrupt_t handler, const uint32_t usperiod);

void stopInterruptHandler(void);
```

El mètode següent permet copiar les dades dels camps de data en els registres que es transfereixen mitjançant I^2C. Té en compte els diferents bits que utilitza l'integrat per a transferir tota la informació.

```
void generateRegisters(bool latch) {
      uint8_t hour12;
      uint8_t temp;
      uint8_t *buffer;

      if (latch) buffer = pcf8593.latches;
      else buffer = pcf8593.regs;

      temp = ((pcf8593.RTCtime.cs / 10) << 4);
      temp |= (pcf8593.RTCtime.cs % 10);
      buffer[1] = temp;

      temp = ((pcf8593.RTCtime.sec / 10) << 4);
      temp |= (pcf8593.RTCtime.sec % 10);
      buffer[2] = temp;

      temp = ((pcf8593.RTCtime.min / 10) << 4);
      temp |= (pcf8593.RTCtime.min % 10);
      buffer[3] = temp;

      if (pcf8593.RTCtime.h12_24 == false) {
            temp = ((pcf8593.RTCtime.hour / 10) << 4);
            temp |= (pcf8593.RTCtime.hour % 10);
            buffer[4] = temp;
      } else {
            temp = pcf8593.RTCtime.hour;
            hour12 = (temp / 2) + 1;
            temp /= 12;

            temp <<= 6;
            temp |= ((hour12 / 10) << 4);
            temp |= (hour12 % 10);
            temp |= 0x80;
            buffer[4] = temp;
      }

      temp = ((pcf8593.RTCtime.day / 10) << 4);
      temp |= (pcf8593.RTCtime.day % 10);
```

```
        temp |= (pcf8593.RTCtime.year << 6);
        buffer[5] = temp;

        temp = ((pcf8593.RTCtime.month / 10) << 4);
        temp |= (pcf8593.RTCtime.month % 10);
        temp |= (pcf8593.RTCtime.weekday << 5);
        buffer[6] = temp;
  }
```

De la mateixa manera, el mètode següent fa la tasca contrària, convertint les dades del format dels registres a variables.

```
  void generateVariables(void) {
        uint8_t hour12;
        uint8_t temp;

        for( ; pcf8593.offset < pcf8593.index; pcf8593.offset++) {
              switch (pcf8593.offset) {
              case 1:
                    pcf8593.RTCtime.cs = ((pcf8593.regs[1] >> 4) * 10) +
                                (pcf8593.regs[1] & 0x0F);
                    break;
              case 2:
                    pcf8593.RTCtime.sec = ((pcf8593.regs[2] >> 4) * 10) +
                                (pcf8593.regs[2] & 0x0F);
                    break;
              case 3:
                    pcf8593.RTCtime.min = ((pcf8593.regs[3] >> 4) * 10) +
                                (pcf8593.regs[3] & 0x0F);
                    break;
              case 4:
                    temp = pcf8593.regs[4];
                    if ((temp & 0x80) != 0) {
                          hour12 = pcf8593.regs[4] & 0x1F;

                          pcf8593.RTCtime.hour = ((hour12 >> 4) * 10) + (hour12 & 0x0F);

                          if ((temp & 0x40) != 0) {
                                pcf8593.RTCtime.hour += 11;
                          } else {
                                pcf8593.RTCtime.hour -= 1;
                          }
                          pcf8593.RTCtime.h12_24 = true;
                    } else {
                          pcf8593.RTCtime.h12_24 = false;
                          pcf8593.RTCtime.hour = ((pcf8593.regs[4] >> 4) * 10) +
                          (pcf8593.regs[4] & 0x0F);
```

```
                }
                break;
            case 5:
                pcf8593.RTCtime.day = (((pcf8593.regs[5] & 0x30) >> 4) * 10) +
                        (pcf8593.regs[5] & 0x0F);
                pcf8593.RTCtime.year = (pcf8593.regs[5] >> 6);
                break;
            case 6:
                pcf8593.RTCtime.month = (((pcf8593.regs[6] & 0x10) >> 4) * 10) +
                        (pcf8593.regs[6] & 0x0F);
                pcf8593.RTCtime.weekday = (pcf8593.regs[6] >> 5);
                break;
            default:
                break;
            }
        }

        pcf8593.index = 0;
        pcf8593.offset = 0;
    }
```

El mètode següent, s'encarrega d'incrementar els diferents comptadors, actualitzant la data.

```
    void timeInterrupt(int n) {
        uint8_t monthDays;

        pcf8593.RTCtime.cs++;

        if (pcf8593.RTCtime.cs >= 100) {
            pcf8593.RTCtime.cs = 0;
            pcf8593.RTCtime.sec++;

            if (pcf8593.RTCtime.sec >= 60) {
                pcf8593.RTCtime.sec = 0;
                pcf8593.RTCtime.min++;

                if (pcf8593.RTCtime.min >= 60) {
                    pcf8593.RTCtime.min = 0;
                    pcf8593.RTCtime.hour++;

                    if (pcf8593.RTCtime.hour >= 24) {
                        pcf8593.RTCtime.hour = 0;
                        pcf8593.RTCtime.day++;
                        pcf8593.RTCtime.weekday++;

                        if (pcf8593.RTCtime.weekday >= 7) {
```

```
                                    pcf8593.RTCtime.weekday = 0;
                              }

                              switch (pcf8593.RTCtime.month) {
                              case 1:
                              case 3:
                              case 5:
                              case 7:
                              case 8:
                              case 10:
                              case 12:
                                    monthDays = 31;
                                    break;
                              case 4:
                              case 6:
                              case 9:
                              case 11:
                                    monthDays = 30;
                                    break;
                              case 2:
                                    if (pcf8593.RTCtime.year == 0) {
                                          monthDays = 29;
                                    } else {
                                          monthDays = 28;
                                    }
                                    break;
                              default:
                                    pcf8593.RTCtime.month = 1;
                              }

                              if (pcf8593.RTCtime.day > monthDays) {
                                    pcf8593.RTCtime.day = 1;
                                    pcf8593.RTCtime.month++;

                                    if (pcf8593.RTCtime.month > 12) {
                                          pcf8593.RTCtime.month = 1;
                                          pcf8593.RTCtime.year++;

                                          if (pcf8593.RTCtime.year >= 4) {
                                                pcf8593.RTCtime.year = 0;
                                          }
                                    }
                              }
                        }
                  }
            }
      }
```

```
        generateRegisters(false);
}
```

Finalment, el mètode i2cInit inicialitza els diferents elements que emularien l'integrat, i també la interrupció.

```
void i2cInit() {
        int i;

        for (i = 0; i < REGISTER_NUMBER; i++)
                pcf8593.regs[i] = 0;

        pcf8593.RTCtime.cs = 0;
        pcf8593.RTCtime.sec = 0;
        pcf8593.RTCtime.min = 0;
        pcf8593.RTCtime.hour = 0;
        pcf8593.RTCtime.day = 1;
        pcf8593.RTCtime.month = 1;
        pcf8593.RTCtime.year = 0;
        pcf8593.RTCtime.weekday = 0;
        pcf8593.RTCtime.h12_24 = false;

        startInterruptHandler(timeInterrupt, US_PERIOD);
}
```

Amb tot això, s'aconsegueix emular el PCF8593, de manera que el programa pot llegir dades. A més, necessitem el controlador que emula el perifèric d'I^2C del microcontrolador. Comencem amb el mètode startCondition per a poder enviar el missatge.

```
void startCondition(void) {
        if (pcf8593.framePosition == START) {
                pcf8593.framePosition = ADDRESS;
        } else {
                pcf8593.framePosition = ERROR;
        }
}
```

Si la posició actual és START, s'inicia el processament de la trama; en cas contrari, s'indica un error. En la posició següent cal afegir-hi la trama en si, les dades de la qual depenen de si es vol escriure en l'RTC o llegir de l'RTC.

```
void putByte(uint8_t value) {

        switch (pcf8593.framePosition) {
        case ADDRESS:
```

```
            if (value == 0xA2) {
                  pcf8593.framePosition = REGISTER;
                  pcf8593.read_write = false;
            } else if (value == 0xA3) {
                  pcf8593.framePosition = REGISTER;
                  pcf8593.read_write = true;
                  generateRegisters(true);
            } else {
                  pcf8593.framePosition = ERROR;
            }
            break;
      case REGISTER:
            if (value < REGISTER_NUMBER) {
                  pcf8593.offset = value;
                  pcf8593.index = value;
                  pcf8593.framePosition = DATA;
            } else {
                  pcf8593.framePosition = ERROR;
            }
            break;
      case DATA:
            if (pcf8593.read_write == false) {
                  if (pcf8593.index < REGISTER_NUMBER) {
                        pcf8593.regs[pcf8593.index] = value;
                        pcf8593.index++;
                  } else {
                        pcf8593.framePosition = ERROR;
                  }
            } else {
                  pcf8593.framePosition = ERROR;
            }
            break;
      default:
            pcf8593.framePosition = ERROR;
      }
}

uint8_t getByte(void) {
      uint8_t value;

      switch (pcf8593.framePosition) {
      case DATA:
            if (pcf8593.read_write == true) {
                  if (pcf8593.index < REGISTER_NUMBER) {
                        value = pcf8593.latches[pcf8593.index];
                        pcf8593.index++;
                  } else {
```

```
                        pcf8593.framePosition = ERROR;
                  }
            } else {
                  pcf8593.framePosition = ERROR;
            }
            break;
      default:
            pcf8593.framePosition = ERROR;
      }

      return value;
}

void stopCondition(void) {
      pcf8593.framePosition = START;
      if (pcf8593.read_write == false) {
            generateVariables();

            pcf8593.read_write = true;
      }
}
```