El programari

Francisco Hernández Ramírez
Juan Daniel Prades García

PID_00177252

Universitat Oberta de Catalunya
www.uoc.edu

Índex

Introducció

Tal com s'ha descrit en mòduls anteriors, els sistemes encastats són sistemes informàtics orientats a l'execució d'una tasca específica amb unes especificacions de treball molt ben definides. Aquest al nivell d'especificació permet simplificar els elements de maquinari al mínim imprescindible per a dur a terme la tasca minimitzant-ne el cost.

Exemple

Com a especificacions, podríem parlar de la necessitat o no de treballar en temps real o de les limitacions en el consum d'energia.

Des del punt de vista del programari, aquest fet fa que, sovint, els programes que governen el funcionament dels sistemes encastats, el microprogramari, s'hagin d'executar en uns recursos de maquinari limitats: poca memòria, potència de càlcul moderada, interfícies d'entrada i sortida inexistents o amb limitacions importants, etc. Per tant, en el món dels sistemes encastats resulta especialment important elaborar codi compacte i eficient que permeti un aprofitament màxim del maquinari disponible. En l'apartat 2, "Conceptes bàsics del programari per a sistemes encastats", descriurem algunes de les tècniques i eines de programació que permeten incrementar l'eficiència dels nostres programes.

En particular, les aplicacions en les quals s'utilitzen sistemes encastats estan relacionades amb el control de processos de característiques molt diferents, des del control d'un ascensor, passant per la gestió del motor d'un cotxe fins a un dispositiu d'electrònica de consum. Totes aquestes aplicacions tenen dues coses en comú.

D'una banda, és necessari proveir el sistema encastat d'informació sobre l'entorn que ha de governar. Des del punt de vista del maquinari, això s'aconsegueix mitjançant la integració de diferents perifèrics connectats als elements de procés mitjançant diferents busos, tal com s'ha descrit en mòduls anteriors. Des del punt de vista del programari, és necessari que els senyals provinents dels perifèrics modifiquin el desenvolupament del flux d'execució del programa. Els elements que permeten controlar el flux de programari es descriuen en el subapartat 2.1, "Interrupcions".

D'una altra banda, s'espera que aquest tipus de sistema treballi en **temps real**, és a dir, que hagin de complir especificacions estrictes quant al temps d'execució de les diferents tasques.

Exemple

És obvi que retards de diversos mil·lisegons en l'execució d'un programa, que poden ser acceptables, per exemple, en un ordinador convencional, no ho siguin en el sistema encastat responsable de disparar el coixí de seguretat d'un automòbil.

Computació en temps real

Entenem per *computació en temps real* o *computació reactiva* tots els sistemes informàtics que han de complir, d'una manera estricta, alguna restricció en el temps de resposta, és a dir, el temps des de la successió d'un esdeveniment i l'execució de la resposta pel sistema.

En els subapartats 2.2 i 2.3 –"Funcions i punters" i "Particularitats de la programació de sistemes encastats", respectivament–, s'estudiaran les característiques del programari que permet treballar en temps real i estratègies per a organitzar i prioritzar l'execució de les diferents tasques del sistema, respectivament.

Finalment, no cal oblidar que la prioritat del programador ha de ser aconseguir que el sistema encastat es comporti d'acord amb les seves especificacions funcionals. Hi ha diferents models de programació o arquitectures de programari que faciliten la síntesi de programes amb comportaments ben determinats que s'estudiaran en l'apartat 3, "Models de programació".

Tots aquests factors afegeixen un grau de complexitat a la programació per a sistemes encastats que no existeix en la programació d'ordinadors de propòsit general convencionals. Per aquest motiu, és recomanable partir d'uns coneixements sòlids en programació abans d'iniciar l'estudi d'aquest mòdul.

Objectius

Els materials didàctics d'aquest mòdul contenen les eines necessàries per a assolir els objectius següents:

1. Saber triar les eines de programació i el llenguatge més adequat per a desenvolupar programari per a sistemes encastats.

2. Conèixer el concepte d'*interrupció* com a mecanisme bàsic per a comunicar-se amb els perifèrics d'un sistema encastat en temps real.

3. Entendre els punters com a eines per a una utilització eficient de la memòria i saber-los utilitzar.

4. Conèixer les funcions *inline* i externes com a tècniques de programació que permeten compactar i fer més modular el codi.

5. Comprendre la necessitat i el funcionament de sistemes encastats basats en bucles de control simple amb interrupcions o sense.

6. Saber dissenyar sistemes basats en màquines d'estats per a processos de monitoratge i control de complexitat intermèdia.

7. Introduir-se al funcionament dels sistemes multitasca.

1. Qüestions preliminars

Tot seguit abordarem alguns aspectes introductoris.

1.1. Elecció del llenguatge de programació

Com en la programació convencional, abans de començar a programar sistemes encastats, cal triar amb cura el llenguatge de programació que farem servir. En els sistemes encastats aquesta és una decisió particularment rellevant, ja que, en molts casos, programar de manera eficient requereix poder accedir a les característiques de baix nivell del maquinari. Aquestes són algunes de les consideracions que cal tenir en compte a l'hora de prendre aquesta decisió:

- Les unitats de processament (ja siguin microprocessadors de propòsit general o bé microcontroladors o DSP dels sistemes encastats) només accepten instruccions en codi màquina.

- El codi màquina és, per definició, el llenguatge de l'ordinador i no el del programador. Per tant, la interpretació del codi màquina és farragosa per als programadors i fàcilment indueix a errors.

- Qualsevol programa, ja estigui escrit en assemblador, C, C++, Java, etc., ha de ser traduït a codi màquina abans de la seva execució. Per aquest motiu, no té gaire sentit crear un codi font "perfecte" si es fa servir un traductor poc eficient que faci que el codi no funcioni com es pretenia.

- Comparats amb els microprocessadors d'ordinadors convencionals, els sistemes encastats solen tenir un potència de càlcul limitada i poca memòria disponible; per tant, el llenguatge utilitzat ha de ser eficient.

- Per a programar sistemes encastats sovint cal accedir a un baix nivell als recursos de maquinari; és a dir, com a mínim, poder llegir i escriure d'una posició de memòria concreta adreçable de manera directa. Això ho fan possible els punters o altres mecanismes equivalents que estudiarem en el subapartat 2.2, "Funcions i punters".

També hi ha uns altres aspectes que no són purament tècnics a l'hora de triar un llenguatge de programació:

- Des d'un punt de vista d'enginyeria de programari, cal poder reaprofitar el codi desenvolupat en projectes previs. Per tant, el llenguatge triat ha de permetre emprar biblioteques que facilitin la reutilització de components de codi validats anteriorment. Desitjablement, també ha de perme-

tre adaptar **codis antics a noves versions del maquinari** amb un esforç mínim.

- Molts sistemes encastats tenen un llarg temps de vida en el mercat, durant el qual cal introduir millores i revisions. Per tant, el llenguatge triat ha de **facilitar el manteniment del codi**, tant després d'haver transcorregut molt de temps com per l'acció de diferents programadors.

- Per tal d'assegurar la integrabilitat en el sistema de components de programari desenvolupats per tercers, garantir la compatibilitat amb tecnologies més o menys estàndard, tenir disponibles àmplies fonts d'informació i reduir el temps de formació del personal implicat en el projecte, el llenguatge triat ha de ser d'**ús comú**.

Aquesta llista breu de les característiques desitjables en un llenguatge de programació per a sistemes encastats ja fa veure que hi ha una paradoxa, entre les necessitats d'eficiència i de control a baix nivell pròpies del codi màquina i la intel·ligibilitat i facilitat d'ús dels llenguatges d'alt nivell.

> Inevitablement, aquí cal prendre una decisió de compromís i que s'aproximi tant com sigui possible a l'ideal:
>
> "[...] d'un llenguatge eficient, d'alt nivell, que proporcioni accés de baix nivell al maquinari i que estigui ben definit i, òbviament, que sigui compatible per a les plataformes de maquinari que volem utilitzar".

Una tria habitual, que satisfà bona part d'aquests aspectes, ha estat el C. A continuació, se'n resumeixen algunes de les característiques principals.

- És un llenguatge de "nivell mitjà", amb característiques d'alt nivell (com el suport de funcions, mòduls i biblioteques) i característiques de baix nivell (com un bon accés al maquinari mitjançant punters).

- És força eficient.

- Hi ha bons compiladors disponibles optimitzats per a la majoria de processadors d'ús habitual en sistemes encastats (des de 8 bits fins a 32 bits o més).

- És fàcil d'aprendre per a programadors acostumats a llenguatges de propòsit general com Java o C++.

- És d'ús comú i està molt ben documentat.

1.2. Eines per a la programació de sistemes encastats

Els dissenyadors de sistemes encastats, com qualsevol altre programador, necessiten compiladors, enllaçadors i depuradors per a desenvolupar el microprogramari del sistema. No obstant això, com que la programació es fa en una plataforma (PC) diferent de la que executarà finalment el codi (sistema encastat), cal emprar algunes eines específiques que faciliten aquesta transició:

- Depuradors sobre maquinari final o emuladors.

Vegeu també

En trobareu més informació en el mòdul "Simulació i test".

- Utilitats per a afegir sumes de comprovació al programa, de manera que el sistema encastat en pugui validar l'autenticitat i integritat abans d'executar-lo.

- Per a sistemes que integrin un DSP, els desenvolupadors poden necessitar eines de processament matemàtic com MATLAB/Simulink, Scilab/Scicos, MathCad, Mathematica, etc., per simular el comportament dels algorismes que cal implementar.

- En aquests mateixos casos, sol ser necessari l'ús de biblioteques compatibles amb el maquinari triat que implementin aquestes rutines matemàtiques.

- En alguns casos, pot ser necessari l'ús de compiladors i enllaçadors personalitzats que millorin l'optimització per a un maquinari determinat.

- Tanmateix, alguns sistemes encastats basats en maquinaris avançats poden disposar del seu llenguatge de programació propi.

- Alternativament, es pot integrar un sistema operatiu encastat en el sistema.

- Per als sistemes encastats basats en màquines d'estats, hi ha eines que permeten simular-les i validar-ne el comportament, i també generar el codi font que les implementa de manera automàtica.

Els proveïdors d'aquestes eines de programari poden ser ben diversos:

- Empreses de programari especialitzades en el mercat dels sistemes encastats.

- Eines lliures procedents d'algun projecte GNU.

- A vegades, també es possible emprar eines de programació de propòsit general gràcies a les similituds de la unitat de procés triada amb els microprocessadors d'ordinadors convencionals.

En qualsevol cas, l'elecció final d'una eina o d'una altra és condicionada per les necessitats del projecte, la seva complexitat, l'experiència del personal implicat i el pressupost disponible.

2. Conceptes bàsics del programari per a sistemes encastats

La necessitat de simplificar els elements de maquinari que conformen els sistemes encastats per tal de reduir-ne el cost i la complexitat tècnica, al mateix temps que s'hi ha de treballar en temps real amb unes especificacions molt condicionades a l'aplicació final, fan imprescindible desenvolupar arquitectures de programari amb unes característiques ben determinades.

En aquesta secció repassarem conceptes bàsics de programació no exclusius d'aquest tipus de sistemes, però que aplicats convenientment i de manera intel·ligent permeten millorar-ne el funcionament global, com són la utilització d'interrupcions i punters. Malgrat que la utilització òptima dels recursos d'un microprocessador integrat a un sistema encastat únicament seria possible desenvolupant programari en llenguatge màquina, la complexitat en la seva interpretació i també els nombrosos errors humans derivats de treballar-hi fan inviable aquesta aproximació des d'un punt de vista tècnic i econòmic.

Finalment, en la tercera i última part de la secció veurem que la utilització de funcions *inline* i funcions externes poden ser útils per a optimitzar, si més no d'una manera parcial, el funcionament d'aquests sistemes.

2.1. Interrupcions

Les interrupcions recollides dins del programari de control de sistemes que funcionen en temps real són sovint un dels factors clau que permet el seu funcionament correcte en una aplicació determinada.

Podem definir una interrupció com el senyal que rep el microprocessador per a redirigir el flux del programa que s'hi està executant en un moment determinat.

Exemple

Els senyals que els perifèrics d'un ordinador envien a la CPU i que fan que comenci una nova tasca per a, posteriorment, reprendre'n la primera una vegada finalitzada la condicionada pel perifèric seria un exemple típic d'interrupció.

En dispositius reals, la majoria de microprocessadors integrats en sistemes encastats estan connectats a components capaços de generar interrupcions molt variades. En canvi, pel que fa al desenvolupament de programari, les mateixes interrupcions, i també les rutines d'interrupció de servei o ISR[1] necessàries per a gestionar-les de manera adequada són sovint passades per alt pels programadors novells, ja que el més petit error en la seva configuració i en el seu disseny acostuma a originar una fallada general dels sistemes molt difícil de corregir *a posteriori*. De fet, aquests programadors acostumen a fer servir l'esquema més simple possible de programació per evitar utilitzar-los i que es basa en preguntar d'una manera recursiva als dispositius generadors d'interrupcions,

(1)Sigla d'*interrupt service routine*. A vegades, també es fa servir el terme *interrupt service process* (ISP).

com, per exemple, perifèrics, si necessiten algun servei del microprocessador. Òbviament, aquesta solució, coneguda en anglès com a *polled communication*, funciona bé quan el processador és ràpid i potent, però en el cas de sistemes encastats podem identificar els desavantatges següents:

- **Consum de molts recursos de microprocessador**, ja que, independentment de si el perifèric sol·licita servei o no, el processador ha de preguntar de manera recursiva sobre el seu estatus.

- **Codi de programari poc ordenat**. A cada iteració del programa principal, aquest ha de fer preguntar al perifèric sobre el seu estatus i, en cas d'originar-se una necessitat de servei, s'ha d'incloure dins el programa principal el codi per a tractar-la. Tot plegat acaba complicant força el codi final del programa.

- **Estats de latència elevats dels perifèrics**. Si el microprocessador està ocupat fent una altra funció, el dispositiu perifèric és ignorat fins que el primer li pregunti sobre el seu estatus. En cas que el microprocessador sigui lent, aquest punt pot arribar a ser crític per a determinades aplicacions.

Per tant, és evident que la programació basada en *polled communication* no és l'aproximació més adequada per a controlar sistemes encastats amb capacitats de càlcul limitades. Tot plegat fa imprescindible dissenyar el seu programari de control considerant la necessitat d'incloure-hi interrupcions.

La gestió correcta d' interrupcions en sistemes en temps real requereix una relació estreta entre programari i maquinari, fet que determina el llenguatge de programació més adequat per a programar les ISR. Per tant, la pregunta directa que se'n deriva és: què és millor, llenguatge màquina o d'alt nivell? En el primer cas, es fa relativament senzill estimar el temps d'execució de la ISR, ja que per a un microprocessador determinat amb una capacitat de càlcul coneguda *a priori*, cada instrucció necessita aproximadament *x* microsegons per a ser executada (depenent de factors intrínsecs al maquinari, com el rellotge intern o els temps d'espera entre estats). Així, doncs, podem arribar a acotar el temps màxim que necessitarà el sistema per a gestionar la interrupció formada per *y* instruccions.

Contràriament, la programació d'ISR en un llenguatge de nivell mitjà-alt esdevé molt més problemàtica. De fet, i malgrat el que sovint afirmen els proveïdors de compiladors d'aquests tipus de llenguatges, no hi ha cap manera òptima d'escriure una ISR en un llenguatge com C, ja que no podem conèixer ni tan sols quant de temps necessitarà la nostra màquina per a executar una única línia de codi, tal com demostra J. Ganssle en el seu llibre *The Art of Designing Embedded Systems*. Fins i tot per a una instrucció simple com és una iteració FOR, el temps d'execució pot anar des d'uns pocs mil·lisegons fins a consumir molts recursos de màquina des d'un punt de vista de temps i memòria. Aquesta dispersió dependrà fonamentalment de com el compilador

Bibliografia recomanada

J. G. Ganssle (2000). *The Art of Designing Embedded Systems* (1a. ed.). Woburn, Massachusetts: Newnes (Elsevier).

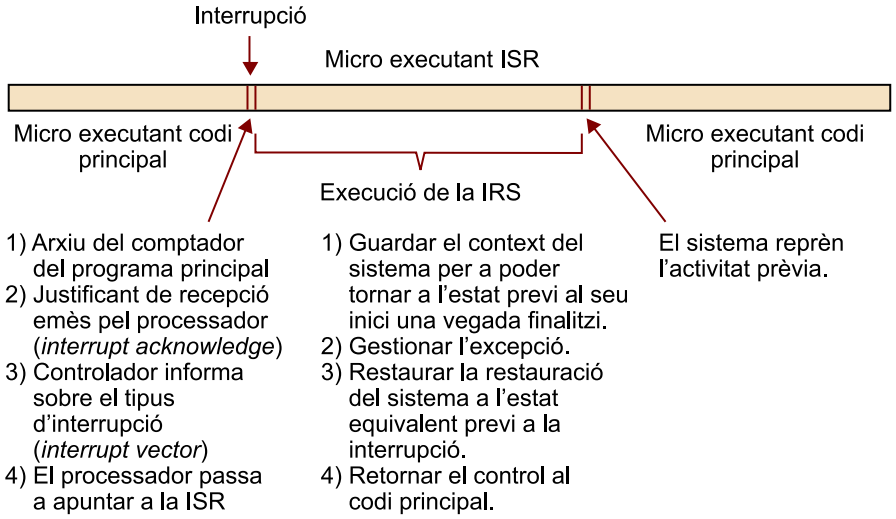
en faci la conversió al llenguatge màquina del codi en llenguatge d'alt nivell. Així, doncs, en primera aproximació podem dir que no hi ha cap impediment per a desenvolupar ISR en llenguatges d'alt nivell com C, però el seu temps d'execució haurà de ser convenientment verificat per a comprovar si satisfà les especificacions requerides al sistema encastat que s'hi està programant. En el pitjor dels casos, que hi hagi un risc de generació ràpida d'interrupcions concatenades, es recomana optar directament pel llenguatge màquina com a solució per a evitar el col·lapse del sistema. No cal dir que aquí el terme *ràpid* és totalment qualitatiu i indefinit, que depèn del significat fonamentalment de l'aplicació, del dispositiu final o, fins i tot, de les preferències personals del programador.

Veiem, doncs, que la gestió optimitzada d'interrupcions esdevé una feina només a l'abast de desenvolupadors experimentats. En aquest text introductori, evidentment l'objectiu no és aconseguir aquest nivell de coneixement i expertesa, sinó repassar la seqüència d'esdeveniments que acaben configurant una interrupció estàndard, i també familiaritzar-nos amb els conceptes que independentment del llenguatge de programació emprat es fan servir per a descriure-les.

2.1.1. Fonaments conceptuals de les interrupcions

La generació d'una interrupció provoca que el sistema encastat segueixi una sèrie d'esdeveniments al llarg del cicle associat a la seva gestió, tal com es pot veure en la figura següent:

Esquema de les accions típiques d'un procés d'interrupció



Primer, l'activació d'una interrupció i l'avís posterior del controlador que la recull al microprocessador del sistema genera els esdeveniments i accions següents en el maquinari:

- **Arxivament del comptador del programa principal en execució a la pila del processador**. L'adreça de memòria que el processador estava exe-

cutant abans de la interrupció queda guardada juntament amb una altra informació, com, per exemple, els continguts dels seus registres interns. Aquest últim punt dependrà del model de processador utilitzat.

- **Generació d'un justificant de recepció (*interrupt acknowledge*) pel processador**. El justificant de recepció és enviat al perifèric o controlador que genera la interrupció sol·licitant un vector d'interrupció (*interrupt vector*), el contingut del qual dependrà del tipus d'interrupció. Un vector d'interrupció no és més que una indicació de l'adreça de memòria on el microprocessador ha d'anar a buscar la ISR o codi de programa associat a la interrupció i el qual s'ha d'executar per a poder-la gestionar.

- **Execució de la ISR**. El microprocessador, una vegada rebut el vector d'interrupció, apunta a l'adreça de memòria on hi ha la ISR i executa les funcions previstes per a gestionar la interrupció.

Una vegada la interrupció ja estat tractada convenientment, el processador executa les tasques següents:

- **Recuperació de l'adreça de retorn i altra informació arxivada a la pila**. El processador, per a poder tornar a l'estat previ a la generació de la interrupció, recupera la informació arxivada a la pila. Generalment, l'adreça de retorn (*return address*, en anglès) és gairebé sempre l'adreça de memòria on hi ha la instrucció posterior que s'hauria executat segons el programa principal si la interrupció no hagués existit.

- **Continuació de l'execució del programa principal**. El processador continua executant el programa principal. Si el programador ha gestionat correctament la interrupció, és evident que aquest codi principal ni tan sols notarà que hi ha hagut una interrupció pel mig, fet que per regla general haurà ocupat uns pocs mil·lisegons de CPU.

Analitzant la seqüència d'esdeveniments anterior que constitueixen una interrupció estàndard, podem afirmar que els punts crítics del procés són la generació del vector d'interrupció i la seva interpretació correcta per part del processador i l'execució de la ISR. En els apartats següents, aprofundirem més en aquests dos aspectes i veurem els requisits generals que han de satisfer els vectors i les ISR per a evitar problemes amb el funcionament final de dispositius encastats.

2.1.2. Vectors d'interrupció

Tots els processadors necessiten rebre un vector d'interrupció quan se'ls notifica l'existència d'una interrupció per tal de poder executar la ISR corresponent. Tal com hem vist abans, els vectors indiquen al processador on anar a buscar el servei necessari per a poder gestionar una interrupció de manera adequada.

Per regla general, els vectors són simplement un número que el processador tradueix a una adreça de memòria on hi ha la primera instrucció de la ISR (vegeu la taula següent).

Adreces de memòria contingudes als vectors associats a cadascuna de les quatre interrupcions que pot processar un microprocessador 80188

Tipus d'interrupció	Adreça de memòria al vector corresponent
INT0	00030h
INT1	00034h
INT2	00038h
INT3	0003Ch

No obstant això, en alguns processadors més antics, també podien contenir codi executable tot i que aquesta aproximació cada vegada és menys utilitzada.

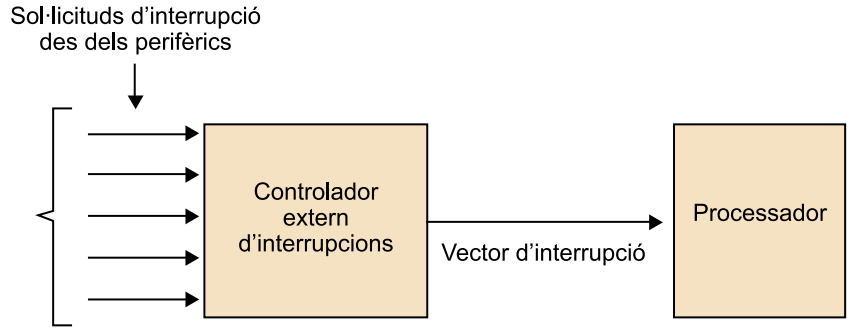
Actualment, molts processadors estan preparats per a gestionar una interrupció especial i no ordinària. Aquest tipus d'interrupció, conegut com a ***non-maskable interrupt*** (NMI), és utilitzada quan succeeix un error fatal en el sistema i no pot ser ignorada pel processador, el qual, una vegada rep un avís d'interrupció, sol·licita el vector corresponent i finalitza el funcionament del dispositiu. A efectes pràctics, això pot provocar pèrdues de dades o situacions no volgudes en determinades aplicacions; per tant, molts programadors s'estimen més dissenyar sistemes encastats que tractin interrupcions crítiques deixant el dispositiu inactiu a l'espera d'una revisió tècnica.

Hi ha tres mètodes per a generar els vectors d'interrupció sol·licitats pel processador: a partir d'un controlador extern, des d'un controlador intern o des dels mateixos perifèrics. La figura següent esquematitza el funcionament de cadascun dels tres models. A efectes pràctics, tots tres produeixen el mateix resultat, tot i que en el maquinari existeixen diferències significatives, sobre les quals no aprofundirem.

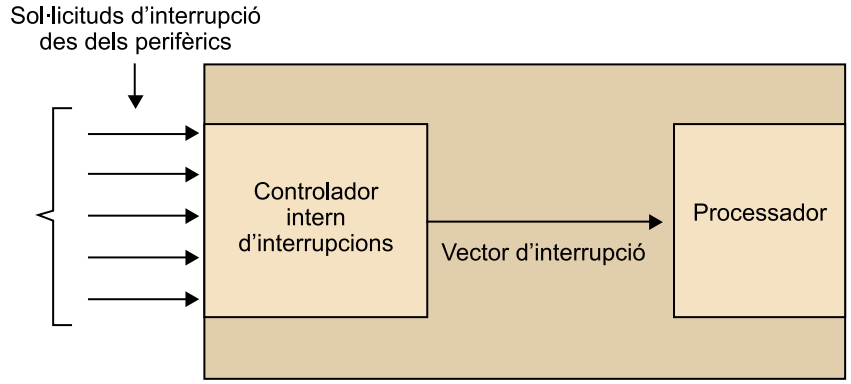
Exemple

En el cas de controladors interns al processador, no s'arriba a generar cap vector d'interrupció real, ja que tota la informació necessària per a gestionar la interrupció és tractada dins del mateix processador, però el programador acaba tenint la impressió que s'ha fet servir un vector d'interrupció virtual que modifica el funcionament del sistema.

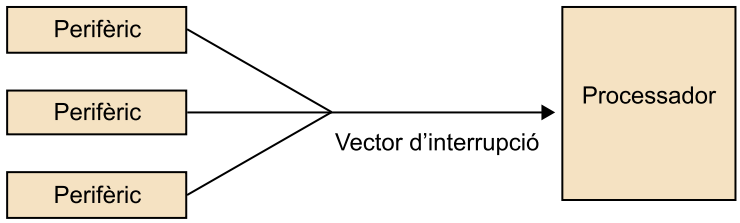
Mecanismes de generació de vector d'interrupció



Processador amb controlador extern d'interrupcions



Processador amb controlador intern d'interrupcions



Perifèrics que sol·liciten interrupcions directament al processador

2.1.3. Rutines d'interrupció de servei (ISR)

En el moment en què una interrupció succeeix, el processador executa una ISR. Per a poder garantir el funcionament correcte del sistema encastat, totes les ISR han de satisfer els tres punt següents:

1) Fer el servei necessari per a gestionar la interrupció generada.

2) Permetre al sistema acceptar noves interrupcions durant la seva execució.

Excepcions NMI

Les excepcions NMI són l'excepció, ja que per definició avorten el funcionament del sistema a causa d'un error fatal. Per tant, només satisfan el primer dels tres punts, ja que tenen prioritat màxima.

3) Una vegada finalitzada, permetre i garantir que el sistema torni a l'estat previ a la interrupció.

Exemple

Per exemple, la família dels 68.000 permet a un perifèric que emet una interrupció avaluar la seva prioritat, i correspon al programador la responsabilitat de gestionar els conflictes potencials entre dos requisits d'igual grau de prioritat que hi pugin arribar al processador. Pel contrari, el processador Intel 8259 permet assignar al programador les prioritats de totes les possibles interrupcions que es poden generar al sistema.

El primer i tercer punts són obvis i constitueixen dues de les accions bàsiques per a poder gestionar una interrupció, tal com hem vist anteriorment. En canvi, per a justificar el segon punt hem d'introduir el concepte de *prioritat*. Per regla general, totes les interrupcions tenen assignada una prioritat determinada. Aquest paràmetre determina quan un processador respon a la sol·licitud d'interrupció d'un perifèric. Així, doncs, una interrupció d'alta prioritat hauria de poder interrompre l'execució d'una de més baixa. Contràriament, una de baixa serà ignorada sempre que una d'alta estigui sent gestionada. Què determina la prioritat dependrà del microprocessador que es faci servir.

Així, doncs, per a justificar la condició que tota ISR ha de permetre acceptar noves interrupcions durant la seva execució, hem de pensar en el cas extrem que una segona interrupció d'alta prioritat es genera mentre una primera de baixa s'hi està executant. En cas que el programador no ho hagi previst, el més probable, exceptuant el cas d'una NMI, és que la segona sigui ignorada fins a la finalització de la primera, ja que la majoria de processadors desconnecten tots els interruptors i entrades des dels perifèrics que el poden avisar de l'existència de noves sol·licituds d'interrupcions. El lector pot imaginar que aquesta situació pot originar situacions no volgudes. Per aquest motiu, cada vegada més processadors permeten el tractament d'interrupcions simultànies o asíncrones.

En l'aproximació més simple i tampoc recomanable, qualsevol ISR és finalitzada si una nova interrupció és generada durant la seva execució, independentment de les seves prioritats respectives. Pel contrari, en el segon esquema de funcionament conegut com a interrupcions imbricades (*nested interrupts*), evidentment més complicat des d'un punt de vista de programació, una ISR només pot ser interrompuda per una de més alta prioritat, per a després ser represa una vegada la segona ha finalitzat. Ens podem imaginar que per a poder reprendre la primera ISR al punt d'interrupció cal guardar registres i adreces de memòria a la pila. Per tant, que aquest mode de treballar sigui factible dependrà fonamentalment de la capacitat i dels recursos del processador, ja que hem de recordar que a la pila tenim guardada prèviament la informació relativa al programa principal del sistema que ha estat interromput. A més, no es pot obviar el grau de complexitat que se'n deriva del disseny d'interrupcions imbricades, fet que en restringeix l'ús correcte a programadors experimentats i molt familiaritzats amb el tema.

2.1.4. Interrupcions: consideracions finals

Les interrupcions són senyals que reben els microprocessadors per a redirigir el flux del programa que s'hi està executant i d'aquesta manera satisfer els requisits de parts del sistema, com, per exemple, els perifèrics. El codi que s'hi executa per a gestionar aquesta demanda es coneix com a *rutina d'interrupció de servei* o *ISR*.

La programació d'interrupcions és complexa i pot induir fàcilment a errors crítics en els sistemes. Per aquest motiu, els desenvolupadors novells sovint fan servir el mètode de programació alternatiu basat en el que es coneix com a *polled communication*. Aquesta solució funciona bé amb processadors potents, però en el cas dels sistemes encastats, esdevé necessari fer servir interrupcions per a aconseguir-ne un funcionament òptim.

No hi ha cap impediment en programar les ISR amb un llenguatge d'alt nivell com C, però la solució òptima és fer servir llenguatge màquina. Tot plegat en dificulta la implementació.

Per regla general, quan un processador proporciona un servei a una interrupció, segueix el procés següent:

1) Un component maquinari genera una sol·licitud d'interrupció que és enviat al microprocessador.

2) El microprocessador prioritza les diferents sol·licituds que rep i en selecciona una.

3) El microprocessador respon al component maquinari amb un justificant de recepció.

4) El component maquinari o un controlador responsable envia un vector d'interrupció al microprocessador.

5) El microprocessador llegeix l'adreça de memòria indicada al vector, guarda la informació necessària per a recordar el seu estat present i salta a la ISR.

6) S'hi executa la ISR i es gestiona la sol·licitud d'interrupció.

7) La ISR finalitza i el microprocessador reprèn la seva activitat inicial.

En aplicacions reals, els programadors han de tenir present les diferents prioritats de les interrupcions que s'hi poden generar i també les característiques del programari que integra el seu dispositiu per tal d'aconseguir optimitzar-ne el funcionament.

2.2. Funcions i punters

A diferència de les interrupcions, les quals conceptualment són fàcils d'entendre però difícils d'implementar en sistemes reals; els punters poden ser utilitzats per programadors novells, però el coneixement profund de què fa la màquina amb ells només s'aconsegueix després d'anys d'experiència.

La combinació de punters i funcions dins un programa ofereix una sèrie d'avantatges des d'un punt de vista d'ús de microprocessador i gestió de la memòria de la màquina, fet que els fa atractius per a ser utilitzats en nombrosos llenguatges de programació d'alt nivell. De fet, els punters han esdevingut una eina fonamental per a desenvolupar programari amb llenguatge C.

Des d'un punt de vista de sistemes encastats, els punters es fan servir exactament igual que en el cas d'ordinadors amb processadors més potents. En canvi, la seva utilització és comparativament més beneficiosa, ja que permeten contrarestar la pobra capacitat de càlcul dels microprocessadors que els integren.

En aquesta secció repassarem la definició de *punter*, en veurem alguns exemples de com operen a nivell de memòria de la màquina i finalment analitzarem en detall com poden ser combinats amb funcions per tal d'optimitzar-ne el programari. En tots casos, els exemples il·lustratius seran en llenguatge C.

2.2.1. Fonaments dels punters. Definicions

Un punter és una variable que representa la posició (no pas el valor) d'una altra dada. És a dir: una variable el valor de la qual és una adreça de la memòria. Dins la memòria dels computadors, cada dada està emmagatzemada ocupant una o més cel·les contigües (paraules o bytes adjacents). El nombre de cel·les requerides per a emmagatzemar una dada depèn bàsicament del seu tipus.

Exemple

Un caràcter necessitarà un byte (8 bits) de memòria.

Punters

Suposem que *v* és una variable que representa una dada determinada. El compilador assignarà automàticament cel·les de memòria per a guardar-lo. A la pràctica, podrem accedir a la dada sempre que coneguem la localització o adreça de la primera cel·la de memòria on està guardat *v*.

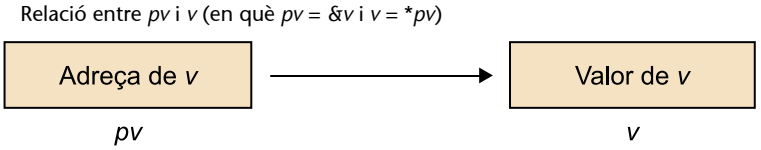
En llenguatge C, l'adreça de memòria de *v* es representa mitjançant l'expressió *&v*, en què & és l'operador adreça que proporciona l'adreça de l'operand. Si assignem aquesta adreça de *v* a una altra variable, *pv*. Així:

$$pv = \&v$$

Aquesta nova variable és un punter a *v* perquè *apunta* a la posició de memòria on hi ha *v*. Cal recordar, però, que *pv* representa l'adreça de *v* i no pas el seu valor. Per tant, *pv*

[(2)] Sempre que *u* i *v* hagin estat declarats com el mateix tipus de dada.

és definida com una variable apuntadora que està relacionada amb *v* segons l'esquema de la figura següent:



A la dada representada per *v*, s'hi pot accedir fàcilment mitjançant l'expressió ***pv,*** en què * és l'operador inadreça, que només opera sobre una variable punter. Així, doncs, ***pv*** i *v* representen la mateixa dada (contingut de les cel·les de memòria). A més, si escrivim *pv* = *&v* i *u* = ***pv***, llavors *u* i *v* representen el mateix valor[2].

Aquesta relació entre variables, adreces de memòria i punters associats es pot visualitzar de manera clara a partir del programa següent en C:

```
#include <studio.h>

void main() {
      int u = 3;
      int v;
      int *pu;       /* punter */
      int *pv;       /* punter */

      pu = &u;       /* assignar adreça de u a pu */
      v = *pu;       /* assignar valor de u a v */
      pv = &v;       /* assignar adreça de v a pv */

      printf("\nu=%d &u=%X pu=%X *pu=%d", u, &u, pu, *pu);
      printf("\n\nv=%d &v=%X pv=%X *pv=%d", v, &v, pv, *pv);
}
```

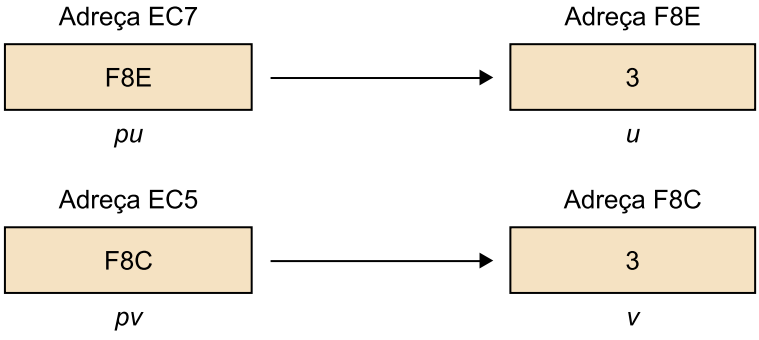
Segons aquest senzill codi *pu* és un punter a *u* i *pv* un punter a *v*. Per tant, *pu* representa l'adreça de *u* i *pv* l'adreça de *v*.

Una possible sortida produïda per l'execució del programa anterior seria:

```
u=3      &u=F8E      pu=F8E      *pu=3
v=3      &v=F8C      pv=F8C      *pv=3
```

La figura següent representa la relació existent entre totes aquestes variables. Cal indicar que les posicions en memòria dels dos punters no són escrites pel programa anterior.

Esquema gràfic amb la relació entre les variables utilitzades en l'exemple proposat.



Els punters, com qualsevol altra variable, han de ser declarats abans de ser usats en un programa en C. No obstant això, la interpretació correcta de la declaració d'un punter és diferent de la declaració d'altres variables. Quan es declara una variable punter, el nom de la variable ha d'anar precedit per un asterisc (*). A més, el tipus de dada que hi apareix es refereix a l'objecte del punter, és a dir, a la dada que s'emmagatzema a l'adreça de memòria representada pel punter i no pas el punter en si mateix. Vegem-ne un exemple senzill:

```
float u, v;
float *pv = &v;
```

Aquí, les variables *u* i *v* són declarades com a variables en coma flotant i *pv* és declarada com una variable punter que apunta a d'altres variables també en coma flotant.

Una vegada fetes les principals definicions necessàries per a recordar què és un punter, i vist com s'hi treballa en llenguatge C, la resta de l'apartat analitzarà com els punters poden interactuar amb les funcions.

Bibliografia recomanada

Si voleu aprofundir en la resta de funcionalitats dels punters es recomana consultar les obres sobre programació en llenguatge C que existeix en la bibliografia.

2.2.2. Pas de punters a una funció. Arguments per referència

A vegades els punters són passats a les funcions com a arguments. Això permet que dades del programa des del qual es crida la funció siguin objecte d'accés i alterades directament per aquesta, per a poder ser finalment tornades al programa ja modificades. Aquesta manera de treballar segons la qual els arguments de les funcions són punters es coneix com a **arguments per referència**, en contraposició a passar arguments per valor, que acostuma a ser la solució habitual.

Si es passen arguments per valor a una funció, la dada és copiada al seu interior. Per tant, qualsevol alteració feta dins de la funció no queda reflectida en el programa principal, tret que el programador així ho especifiqui, introduint línies de codi addicional i alentint d'aquesta manera el funcionament del programa. Al contrari, quan un argument es passa per referència, l'adreça de memòria és indicada a la funció. Al contingut d'aquesta adreça, es pot accedir lliurement des de dins de la mateixa funció i, per tant, qualsevol canvi que es faci serà reconegut per la funció, però també des de fora.

Veiem que l'ús de punters com a arguments de la funció permet alteracions i modificacions globals de les dades des de dins de la funció, cosa que ajuda a estalviar línies de codi i alleugereix la quantitat de línies de codi.

En sistemes encastats en què la velocitat d'execució del programari i la seva mida són crítics, la utilització dels punters és una opció atractiva.

A continuació, tenim un exemple senzill en C que il·lustra la diferència entre passar arguments per valor o referència:

```
#include <studio.h>

void func1(int u, int v);       /* prototip de funció*/
void func2(int *pu, int *pv);       /* prototip de funció*/

void main() {

      int u = 1;
      int v = 3;
```

```
        rintf("\ Abans de la func1; u=%d v=%d", u, v);
        func1(u, v);
        printf("\ Després de la func1; u=%d v=%d", u, v);

        printf("\ Abans de la func2; u=%d v=%d", u, v);
        func2(&u, &v);
        printf("\ Després de la func2; u=%d v=%d", u, v);

}

void func1(int u, int v) {
       u = 0;
       v = 0;
    printf("\ Dins func1; u=%d v=%d", u, v);
    return;

}

void func2(int *pu, int *pv) {
       *pu = 0;
       *pv = 0;
       printf("\ Dins func2; *pu=%d *pv=%d", *pu, *pv);
       return;

}
```

Aquest programa té dues funcions, anomenades `func1` i `func2`. La primera rep dues variables enteres com a arguments que tenen originàriament assignats els valors 1 i 3, respectivament. Els valors són alterats a 0,0 al seu interior. No obstant això, els nous valors no són reconeguts a main, perquè els arguments hi van arribar per valor i qualsevol canvi sobre els arguments és local a la funció dins la quals s'han produït els canvis.

Analitzem ara la segona funció, `func2`. Aquesta rep dos punters a variables senceres com a arguments. Els arguments són identificats com a punters pels operadors d'inadreça (asterisc) que hi ha en la declaració d'arguments. A més, la declaració d'arguments indica que els punters representen adreces de memòria de quantitats senceres.

Dins de `func2`, els continguts d'adreces apuntades són assignats amb valors 0,0. Com que les adreces són reconegudes a la `func2` i a main, les variables senceres *u* i *v* també hauran canviat a 0,0 a tot arreu.

Les sis instruccions `printf` il·lustren els valors de *u* i *v* i els seus valors associats **pu* i **pv* dins de main i dins de les dues funcions. Per tant, quan s'executi el programa es generarà la sortida següent:

```
Abans de la func1:      u=1      v=3
Dins func1:             u=0      v=0
Després de la func1:    u=1      v=3

Abans de la func2:      u=1      v=3
Dins func2:            *pu=0     *pv=3
Després de la func1:    u=1      v=3
```

L'exemple anterior és il·lustratiu del potencial dels punters per a modificar les dades de memòria dins un programa amb poques línies de codi i funcions simples. Evidentment, aquestes variables ofereixen potencialitats més avançades que permeten operar de manera controlada amb formacions unidimensionals i multidimensionals, assignar dinàmicament la memòria del dispositiu o construir formacions amb elles mateixes.

Consulta recomanada

Us aconsellem la consulta d'un llibre introductori a la programació en llenguatge C per a saber-ne més.

2.2.3. Pas funcions a altres funcions

A més de les operacions e1erades anteriorment, els punters permeten una funcionalitat extra d' interès especial en la programació i en el disseny programari dels sistemes encastats; el pas d'una funció com a argument a una altra funció com si la primera fos una variable. Definirem la primera funció com la **funció hoste** i la segona com la **funció amfitriona**. D'aquesta manera, l'hoste és passada a l'amfitriona, des d'on s'hi pot accedir lliurement. Crides successives a la funció amfitriona dins el programa poden passar diferents punters (diferents funcions hostes) a l'amfitriona. És evident que aquest esquema de programació permet compactar els codis de programari força, i alleugerir la càrrega de feina del microprocessador, sempre que tinguem present allò que realment està fent la màquina per a evitar errors conceptuals.

Punters (continuació)

Aquí entenem que el lector ja té coneixements previs de programació en llenguatge C per a poder seguir l'esbós il·lustratiu següent. Aquest programa consta de quatre funcions: main, processar, func1 i func2. Notar que processar és una funció amfitriona per a func1 i func2. Cadascuna de les tres funcions subordinades torna un valor sencer.

```
int processar(int(*)(int,int)); /* prototip funció amfitriona*/
int func1(int, int);            /* prototip funció hoste*/
int func2(int, int);            /* prototip funció hoste*/

void main() {
      int i, j;
      ......
      i = processar(func1);     /* passem func1 a processar; torna
                                   un valor per a i */
      ......
      j = processar(func2);     /* passem func2 a processar; torna
                                   un valor per a j */
      ......
      }

processar(int (*pf)(int,int))   /*definició funció amfitriona*/
{
      int a, b, c;
      ......
      c = (*pf) (a,b);          /* accés a la funció hoste */
                                /* torna un valor de c */
```

```
      ......
      return(c);
}

func1(int a, int b) { /*definició funció hoste*/
      int c;
      c = ....   /* fa servir a i b per avaluar c*/
      return(c);
}

Func2(int x, int y) { /*definició funció hoste*/
      int z;
      z = ....   /* fa servir x i y per avaluar z*/
      return(z);
}
```

Així, doncs, aquest esbós de programa conté tres declaracions de funcions. Les declaracions per a `func1` i `func2` són directes. D'altra banda, la declaració de processar necessita un petit aclariment. Aquesta declaració la defineix com una funció amfitriona que torna un valor sencer i al mateix temps té un argument. L'argument és un punter a una funció hoste que torna un valor sencer i té dos arguments sencers.

Algunes aplicacions de programació es poden formular més fàcilment en termes de pas d'una funció a una altra, amb la simplificació conseqüent en codi i càlcul que això implica. Això és particularment útil si el programa conté diferents equacions matemàtiques, de les quals l'usuari selecciona una cada vegada que s'executa el programa.

Exemple

Una funció pot representar una equació matemàtica i l'altra pot contenir una estratègia computacional de resolució. En aquest cas, la funció que representa l'equació pot ser passada a la funció que processa l'equació.

2.2.4. Funcions i punters: consideracions finals

Els punters són variables que es fan servir en llenguatges de nivell mitjà i alt i que representen la posició en memòria d'una altra dada. La seva utilització correcta dins els programes permeten modificar els continguts de la memòria de la màquina estalviant línies de codi en comparació d'altres opcions de programació menys sofisticades. Entre els avantatges de fer servir punters, és especialment interessant la seva combinació amb les funcions existents dins un programa.

Vegeu també

Hi ha un exemple resolt d'aquest esquema de funcionament en la secció d'activitats.

2.3. Particularitats de la programació de sistemes encastats

La programació dels sistemes encastats, tal com hem vist fins ara, és bàsicament idèntica a la de sistemes estàndard. No obstant això, algunes solucions que poden ser interessants per als segons han de ser utilitzades de manera adequada en els primers si la seva funcionalitat es vol garantir a conseqüència dels seus recursos limitats. Com a exemple, en aquest últim apartat dedicat als conceptes bàsics del programari per a sistemes encastats, veurem les consideracions necessàries per a integrar satisfactòriament les funcions *inline* i les funcions externes en aquest tipus de sistemes.

2.3.1. Funcions *inline*

En llenguatge C i C++, el programador pot sol·licitar al compilador que introdueixi el codi complet d'una funció a qualsevol part del programa on aquesta funció sigui cridada a actuar, en comptes d'anar a buscar la funció a la posició de memòria on ha estat definida inicialment. Malgrat que el compilador no està obligat a respectar aquesta sol·licitud i de fet moltes vegades no ho fa, aquesta estratègia de programació coneguda com ***inline expansion*** o simplement ***inlining*** és interessant perquè en primera aproximació ajuda a millorar el temps d'execució del programari. Com a contrapartida, augmenta la mida total del programa.

Inlining és una estratègia habitual per a optimitzar els funcionaments de programes en què hi ha moltes crides a funcions, especialment si aquestes són petites (poques línies de codi). Utilitzant-la, el programador aconsegueix els avantatges següents:

- S'elimina el cost de fer servir les instruccions "function call" i "return" cada vegada que es crida la funció d'interès. Addicionalment, també desapareix el codi pròleg i epíleg que el compilador afegeix al programa de manera automàtica.

- En mantenir tot el codi que cal executar al costat d'una posició de memòria determinada, la velocitat d'execució del programa augmenta considerablement, com a mínim des d'un punt de vista teòric.

En canvi, treballar amb funcions "inline" pot no ser la solució més adequada per diferents motius; en presentem els inconvenients següents:

- Exceptuant el cas particular de programadors molt experimentats, el compilador sol conèixer millor que l'ésser humà quan generar funcions *inline*. Fins i tot, moltes vegades les sol·licituds d'*inlining* fetes pel desenvolupador de programari són impossibles de fer per raons tècniques. Tot plegat genera problemes de compilació o rendiments del programari no previstos inicialment.

- La proliferació no controlada de funcions *inline* dins un programa en C dispara el temps de compilació, ja que el codi és copiat en cada punt del codi en què la funció és cridada a actuar.

- L'augment en el temps de compilació és directament proporcional a l'augment de la mida del programa compilat.

- L'augment de la mida del programa principal pot fer que aquest no càpiga correctament a la memòria cau de la màquina, la qual cosa genera pèrdua de dades o simplement la fa funcionar de manera més lenta.

Per tant, veiem que la conveniència o no de fer servir funcions *inline* dependrà fonamentalment del tipus de sistema encastat que s'hi estigui programant. El desenvolupador de programari, abans d'optar per l'*inlining* com a mecanisme per a optimitzar el seu sistema haurà de considerar els tres punts següents:

- En aplicacions en què la mida del programa que cal executar és més important que la velocitat d'execució, l'*inlining* s'acostuma a evitar.

- L'*inlining* consumeix registres de màquina addicionals, que s'acaba traduint en accessos no volguts a la memòria RAM.

- Si el codi augmenta molt la seva mida, la capacitat limitada de la memòria RAM en sistemes encastats pot ser superada. El programa simplement no funcionarà o s'hi generarà hiperpaginació o *trashing*.

A efectes pràctics, els desenvolupadors de programari i programadors, tot i que tinguin present aquest problema quan hi treballen, acaben incorporant heurístiques als seus compiladors, les quals decideixen quines funcions seran *inline* i quines no després d'avaluar el funcionament del programa final.

2.3.2. Funcions externes

En sistemes reals, el codi de control normalment és format per diferents arxius. Això és especialment vàlid en programes que fan servir funcions molt grans o complexes, en què cadascuna d'elles pot arribar a ocupar un únic arxiu. Aquesta solució de programació també s'acostuma a fer servir quan hi ha moltes funcions petites relacionades entre elles, unes poques de les quals s'agrupen en un únic arxiu.

Aquests arxius es compilen de manera individual i a continuació s'enllacen per a generar un únic programa. Tot plegat, fa més fàcil la redacció i depuració del programa, ja que els arxius tenen una mida més raonable. A més, els programes multiarxiu acostumen a ser més flexibles quan són modificats o actualitzats *a posteriori*.

Dins un programa multiarxiu, una funció externa és reconeguda al llarg de tot el programa, mentre que una funció estàtica només ho és dins l'arxiu en què s'hagi definit. En tot cas, el seu tipus s'estableix col·locant extern o *static* al començament de la definició de la funció. Cal recordar que, per defecte i en cas de no indicar-ho, totes les funcions seran sempre externes.

Funcions externes

A continuació, podem veure un programa senzill en llenguatge C que genera el missatge "Hola" des d'una funció externa. El programa és format per dues funcions: main i sortida. Cada funció és dins un arxiu diferent.

Primer arxiu (ARXIU1.C)

```
/* programa simple multiarxiu per a escriure "Hola" */

#include <stdio.h>

extern void salida(void);    /* prototip de funció */

void main() {
      salida();
}
```

Segon arxiu (ARXIU2.C)

```
extern void sortida(void);      /* definició de funció externa */
{
      printf("¡Hola!");
      return;
}
```

En el moment de compilar el programa anterior, simplement generarem un projecte comú que inclogui els dos arxius. Com fer-ho dependrà exclusivament del compilador que es faci servir.

Els principals avantatges de fer servir funcions externes i, per extensió, la programació de codi amb diferents arxius són:

- La feina es pot distribuir entre diferents programadors. Cadascun d'ells se'n pot ocupar d'un de diferent.

- Es pot fer servir un estil orientat a objectes. Cada arxiu defineix un tipus particular d'objecte com un tipus de dada, i les operacions en aquest objecte com a funcions. La implementació de l'objecte es pot mantenir en privat a la resta del programa.

- S'aconsegueixen programes ben estructurats els quals són fàcils de mantenir, actualitzar i en cas que sigui necessari millorar.

- Els arxius poden contenir totes les funcions d'un grup relacionat, per exemple totes les operacions amb matrius.

- Objectes ben implementats o definicions de funcions poden ser reutilitzats per altres programes, cosa que a la llarga redueix el temps de desenvolupament.

- Quan es fan canvis a un únic arxiu, només aquest necessita tornar a ser compilat per a poder reconstruir el programa, amb el consegüent estalvi de temps i recursos que això implica.

Per tant, podem afirmar que l'ús de funcions externes és una eina atractiva per a desenvolupar programari de control modular per a sistemes encastats, fàcil de modificar i millorar en el futur. No obstant això, la seva implementació requereix una planificació i estructuració prèvia del seu disseny que sovint no es fa, especialment per part dels programadors novells.

3. Models de programació

Hi ha moltes maneres d'implementar el programari que governa un sistema encastat. No obstant això, habitualment se segueix una arquitectura o model de programació en particular, que està ben provat i facilita que altres programadors comprenguin i reutilitzin el codi. En aquesta secció, descriurem només alguns d'aquests models.

3.1. Estratègies bàsiques

Els models de programació o estratègies bàsiques són els que indicarem tot seguit.

3.1.1. Bucle de control simple

Els sistemes encastats es caracteritzen per estar orientats a l'execució d'una tasca específica que es fa contínuament. Aquest fet provoca que, en molts casos, el nucli bàsic del programa consisteixi, simplement, en un bucle continu que s'encarrega de cridar de manera successiva les diferents subrutines encarregades de gestionar els diferents elements de maquinari o programari.

En aquest paradigma tan bàsic, si es vol introduir la capacitat d'interactuar amb el món exterior, pot ser necessari comprovar en algun moment l'estat de certs perifèrics. En aquests casos, caldrà implementar alguna rutina que en verifiqui l'estat.

Per tant, un sistema així d'elemental només podrà detectar canvis de l'entorn i respondre-hi només durant l'execució d'aquesta rutina de control. Així, doncs, el temps d'execució de cadascuna de les iteracions d'aquest bucle és el que determina el temps màxim de resposta del sistema.

En conclusió, tot i ser un model de programació molt simple i eficient, presenta limitacions importants quant al funcionament en temps real. Vegem-ne un exemple:

Volem implementar un equip que controli un rètol lluminós mitjançant un sistema encastat. El funcionament ha de ser el següent:

> "Inicialment, cal il·luminar tot el rètol, que consisteix en les lletres de la paraula *OFERTA* durant 3 segons. Tot seguit, cal apagar-lo durant 0,5 segons, i anar encenent cada una a intervals d'1 segon. Quan s'ha completat el procés, cal apagar-lo durant 0,5 segons i repetir la seqüència."

Es tracta d'un funcionament cíclic i, per tant, resulta adequat plantejar-se un funcionament basat en bucle simple de control. Observem que es tracta d'una situació en la qual no s'espera rebre cap senyal dels perifèrics que pugui alterar el funcionament del sistema durant tot el cicle de funcionament. A més a més, no es tracta d'una aplicació crítica, de manera que, tot i haver d'oferir una resposta en temps real, no requereix una precisió extrema.

A continuació, es presenta una possible solució al problema:

```
#define TEMPS_CICLE 50      //constant que recull el temps d'execució
                           //d'un cicle del bucle de control (en ms)

#define TOT_ON             3000      //intervals seqüènc. encesa (ms)
#define TOT_OFF            3500
#define O_ON               4500
#define OF_ON              5500
#define OFE_ON             6500
#define OFER_ON            7500
#define OFERT_ON           8500
#define OFERTA_ON          9500
#define TOT_FINAL_OFF      10000

void main() {

      int temps = 0;
      int j = 0;

      while (1) {            // el bucle de control s'executa SEMPRE

            if (temps > 0 && temps < TOT_ON) {
                  display.ofertaON();
            } else if (temps > TOT_ON && temps < TOT_OFF) {
                  display.OFF();
            } else if (temps > TOT_OFF && temps < O_ON) {
                  display.oON();
            } else if (temps > O_ON && temps < OF_ON) {
                  display.ofON();
            } else if (temps > OF_ON && temps < OFE_ON) {
                  display.ofeON();
            } else if (temps > OFE_ON && temps < OFER_ON) {
                  display.oferON();
            } else if (temps > OFER_ON && temps < OFERT_ON) {
                  display.ofertON();
            } else if (temps > OFERT_ON && temps < OFERTA_ON) {
                  display.ofertaON();
            } else if (temps>OFERTA_ON && temps<TOT_FINAL_OFF) {
```

```
                    display.OFF();
                } else {
                    j = 0;
                }
                j++;
                temps = j * TEMPS_CICLE;
            }
        }
```

En primer lloc, cal parar atenció al fet que el cicle de control s'executa sempre. Per tant, només s'atura quan s'apaga l'equip.

En segon lloc, observem que s'ha establert un sistema de control de temps basat en el temps d'execució mitjà del bucle de control simple. Segons prova i error, s'ha arribat a la conclusió que el temps necessari per a executar el bucle és de prop de 50 mil·lisegons en el maquinari disponible. D'acord amb aquest resultat, un simple comptador d'iteracions (*j*), s'encarrega de mantenir actualitzada una variable (temps) que recull el temps estimat en la seqüència d'il·luminació actual. Si bé aquesta solució pot ser adequada per a aquesta aplicació, és clar que no seria convenient en sistemes en els quals el temps d'execució d'aquest bucle fos incert.

En tercer lloc, en cadascun dels intervals de la seqüència d'encesa, es crida la subrutina corresponent (per exemple, display.OFF[]) que s'encarrega d'executar les accions necessàries. Aquesta és una manera de procedir habitual en els sistemes basats en bucle simple de control: l'estructura del mètode principal es manté simple (el bucle i poca cosa més) i els detalls de les accions que cal dur a terme en cada iteració es programen, de manera modular, en subrutines independents.

3.1.2. Control per esdeveniments

El control per esdeveniments sorgeix de la necessitat de dissenyar sistemes encastats capaços d'interactuar amb el món exterior en temps real. En aquest sentit, permet superar les limitacions del bucle de control simple.

En aquest cas, s'assumeix que tot el funcionament del sistema encastat és determinat per la successió d'esdeveniments que estigui prevista detectar i processar. O, de manera equivalent, per l'activació successiva de diferents interrupcions d'acord amb els mecanismes descrits en el subapartat 2.1, "Interrupcions".

Què provoca les interrupcions?

L'origen d'aquestes interrupcions pot ser molt divers. Per exemple, poden ser generades per un temportizador intern a una freqüència determinada o per un byte rebut al port sèrie. Això vol dir que idealment les tasques del sistema s'executen únicament a instàncies de diferents tipus d'esdeveniments interns o externs al sistema.

Cal tenir en compte, però, la naturalesa impredictible de la seqüència d'interrupcions (tant pel que fa al seu ordre i nombre com al retard entre elles): si no es fa un tasca exhaustiva de planificació de la resposta del sistema en les diferents situacions els sistema pot no ser robust, ni estable, ni predictible.

L'experiència demostra que els sistemes controlats per interrupcions són eficaços si: **a)** les subrutines de gestió d'interrupció són breus i simples (com ara el cas dels temps d'execució breu), o **b)** i si poden ser accedides ràpidament (com en el cas d'un baix temps de latència). Aquestes dues característiques ajuden a prevenir retards i col·lapses en el sistema, per exemple, davant d'una allau d'interrupcions.

Des d'un punt de vista pràctic, se solen implementar sistemes híbrids entre els paradigmes de bucle de control simple i de control per interrupcions. Habitualment, els sistemes executen un tasca simple i no gaire sensible a retards imprevistos en el bucle principal. Paral·lelament, les subrutines de gestió d'interrupcions només s'encarreguen d'afegir tasques a la seqüència del bucle principal que s'executaran en la propera iteració. Així, l'atenció a interrupcions resulta ràpida, tot prevenint col·lapses, alhora que manté un flux d'execució simple, periòdic i predictible. En el subapartat 3.3.1, "Disseny i captura de funcionalitat", en mostrem un exemple.

3.2. Màquines d'estat

En sistemes encastats orientats a aplicacions de control, sovint resulta útil organitzar la programació en funció de l'estat del sistema que s'ha de controlar. Seria equivalent a disposar d'un únic sistema de control amb diversos modes de funcionament en funció de les circumstàncies.

Les màquines d'estats es basen en l'assumpció que el sistema només es pot trobar en un únic estat en cada instant. Cadascun dels estats s'associa a una situació de funcionament determinada, de manera que és fàcil restringir l'execució de tasques i l'atenció d'interrupcions només a aquelles que siguin pròpies de l'estat.

En un sistema controlat per bucle simple, amb interrupcions o sense, els diferents estats es podrien visualitzar com diferents seqüències de subrutines que cal executar de manera alternativa dins del bucle principal.

Les màquines d'estats resulten, per tant, una bona estratègia per a l'estructuració i la compartimentació del codi i les seves funcionalitats: es pot modificar el codi d'un estat determinat sense afectar-ne la resta.

3.2.1. Disseny i captura de funcionalitat

Vegem un exemple de com s'ha de capturar la funcionalitat d'un sistema mitjançant una màquina d'estats, i també els seus avantatges respecte a la programació seqüencial tradicional.

Controlador d'ascensor implementat amb màquina d'estats

Les especificacions requereixen que el sistema de control d'un ascensor es comporti de la manera següent:

"Moveu l'ascensor ja sigui amunt o avall fins a arribar a la planta sol·licitada. Una vegada hi hàgiu arribat, obriu la porta, com a mínim durant 10 segons, i deixeu-la oberta fins que la planta sol·licitada canviï. Assegureu-vos que la porta no s'obre mai mentre s'està en moviment. No canvieu d'adreça, tret que no hi hagi més plantes sol·licitades en l'adreça del moviment actual (per exemple, plantes superiors si esteu pujant, plantes inferiors si esteu baixant)."

Aquesta seria una possible implementació d'aquest exemple emprant programació seqüencial tradicional[3].

```
void UnitControl()
{
      up = down = 0; open = 1;
      while (1) {
      while (req == floor);
      open = 0;
      if (req > floor) { up = 1;}
      else {down = 1;}
      while (req != floor);
      up = down = 0;
      open = 1;
      delay(10);
      }
}
```

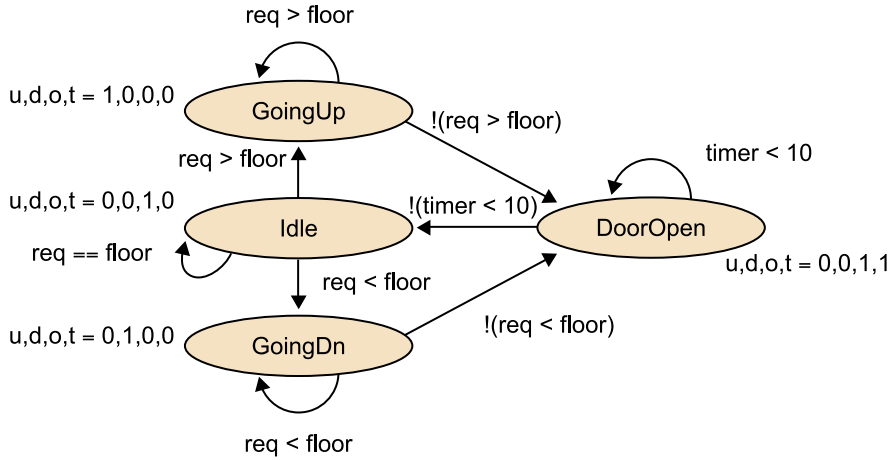
(3) Nota: `up`, `down` i `open` són variables booleanes que determinen l'execució de les tasques d'ascendir, descendir i obrir portes, respectivament. `floor` i `req` són enters que emmagatzemen la informació de la planta actual i la planta sol·licitada, respectivament.

Resulta força clar que si ens valem només de la programació seqüencial convencional resultarà fàcil cometre errors i haver de perdre molt de temps per a aconseguir implementar la funcionalitat volguda. Una manera més simple, directa i sistemàtica d'implementar aquest tipus de sistemes són les **màquines d'estats finits**[4].

(4) En anglès, *finite state machine* (FSM).

La manera habitual de representar el funcionament d'una FSM és el **diagrama d'estats**. La figura següent mostra el diagrama d'estats de la màquina que implementa el control de l'ascensor segons la descripció anterior.

Diagrama d'estats d'una possible implementació de l'ascensor de l'exemple que acabem d'analitzar



Els diagrames d'estat se solen representar amb la convenció de signes següent.

- Cercles: representen els estats. A cadascun dels estats, s'hi associa una seqüència binària diferent de la resta coneguda com a **codi de l'estat**.

- Fletxes: representen les transicions permesos entre estats. Solen anar acompanyades de la **condició** que provoca aquest canvi.

Aquest tipus de diagrama es pot elaborar fàcilment seguint els passos següents:

1) Fer una llista de tots els estats possibles

2) Declarar totes les variables, solen ser modificades per les rutines d'atenció a interrupció en rebre un senyal exterior indicatiu d'un canvi d'estat. També poden ser modificades en certs estats de la màquina d'estats.

3) Per cada estat, e1erar les possibles transicions a altres estats, i identificar-ne les condicions.

4) Per cada estat i/o transició, fer una llista de totes les accions necessàries (tasques, canvis de variables, etc.).

5) Assegurar-se que, per cada estat, les condicions de transició són mútuament excloents i completes (és a dir, que només una condició pot ser certa al mateix temps i que en tot moment hi ha, com a mínim, una condició certa).

Exemple (continuació): controlador d'ascensor implementat amb màquina d'estats

En l'exemple que ens ocupa, aquests cinc punts es traduirien en:

1) Necessitem quatre estats:

- *Idle* = l'ascensor espera amb la porta oberta.
- *GoingUp* = l'ascensor tanca la porta i puja.
- *GoingDown* = l'ascensor tanca la porta i baixa.

- *DoorOpen* = l'ascensor obre la porta.

2) Necessitarem tres variables:

- *req* = planta sol·licitada
- *floor* = planta actual.
- *timer* = hora del temporitzador.

3) En la taula següent identifiquem les transicions entre estats següents, associades a un seguit de condicions entre les variables:

Transicions entre estats i condicions

Transició	Condició
Idle → *Idle*	req == floor
Idle → *GoingUp*	req > floor
Idle → *GoingDown*	req < floor
GoingUp → *GoingUp*	req > floor
GoingUp → *DoorOpen*	!(req > floor)
GoingDown → *GoingDown*	req < floor
GoingDown → *DoorOpen*	!(req < floor)
DoorOpen → *DoorOpen*	timer < 10
DoorOpen → *Idle*	!(timer < 10)

4) Les tasques que ha de dur a terme el sistema en els diferents estats són les següents:

- *u* = desplaçar l'ascensor amunt.
- *d* = desplaçar l'ascensor avall.
- *o* = obrir les portes.
- *t* = iniciar el temporitzador.

En la taula següent s'e1eren les tasques que cal dur a terme en cadascun dels estats. Fixeu-vos que si s'associa una variable binària a cadascuna d'elles, es genera automàticament un codi binari identificatiu diferent per a cadascun dels estats. Els codis no poden ser iguals ja que dos estats que facin les mateixes tasques haurien de ser, per força, els mateixos.

Llista dels estats, accions i codis

Estat	u	d	o	t
Idle	0	0	1	0
GoUp	1	0	0	0
GoDown	0	1	0	0
DoorOpen	0	0	1	1

5) Finalment, es deixa com a exercici per al lector verificar que les condicions de transició imposades són mútuament excloents i completes tenint en compte els canvis de variables associats a cadascuna de les tasques.

3.2.2. Implementació amb codi

Una vegada completat el disseny de la màquina d'estats seguint els passos anteriors ja es pot passar a la seva implementació en codi font.

Exemple (continuació): controlador d'ascensor implementat amb màquina d'estats

Vegem la implementació de l'exemple anterior:

```
#define IDLE 0
#define GOINGUP 1
#define GOINGDN 2
#define DOOROPEN 3

void UnitControl() {
      int state = IDLE;
      while (1) {
      switch (state) {
      IDLE: up=0; down=0; open=1; timer_start=0;
            if  (req==floor) {state = IDLE;}
            if  (req > floor) {state = GOINGUP;}
            if  (req < floor) {state = GOINGDN;}
            break;
      GOINGUP: up=1; down=0; open=0; timer_start=0;
            if  (req > floor) {state = GOINGUP;}
            if  (!(req>floor)) {state = DOOROPEN;}
            break;
      GOINGDN: up=1; down=0; open=0; timer_start=0;
            if  (req < floor) {state = GOINGDN;}
            if  (!(req<floor)) {state = DOOROPEN;}
            break;
      DOOROPEN: up=0; down=0; open=1; timer_start=1;
            if (timer < 10) {state = DOOROPEN;}
            if (!(timer<10)){state = IDLE;}
            break;
      }
      }
}
```

Fixem-nos, en primer lloc, que gràcies a la feina de disseny d'una màquina d'estat, aquesta implementació és molt més clara que la basada en codi seqüencial.

En segon lloc, observem que el programa consisteix en un bucle principal que es repeteix de manera indefinida, però que fa diferents accions en cada iteració en funció de l'estat en el qual es troba el sistema.

En tercer lloc, cal parar atenció al fet que les estructures *switch-case* són particularment adequades per a implementar aquest tipus de programari, ja que afavoreixen l'estructuració i la compartimentació del codi esmentades anteriorment. A continuació, presentem una plantilla de codi que pot ser emprada per a implementar qualsevol màquina d'estat:

```
#define S0      0
#define S1      1
...
#define SN      N
```

```
void StateMachine() {
      int state = S0;       //indicar aquí l'estat inicial.
      while (1) {
      switch (state) {
      S0:
                  //indicar les accions de l'estat S0:
                  /*actions*/

                  //indicar transicions Ti de sortida de S0:
                  if(condition T0 == true ) {
                        state = estat següent segons T0;
                        /*actions*/
                  }
                  if(condition T1 == true ) {
                        state = estat següent segons T1;
                        /*actions*/
                  }
            ...
                  if(condition Tm == true ) {
                       state = estat següent segons Tm;
                       /*actions*/
                  }
            break;
      S1:
                  //indicar les accions de l'estat S1:
                  ...
                  //indicar transicions Ti de sortida de S1:
                  ...
            break;
      ...
      SN:
                  //indicar les accions de l'estat SN:
                  ...
                  //indicar transicions Ti de sortida de SN:
                  ...
            break;
      }
      }
}
```

Finalment, val la pena reflexionar sobre el fet que el model de programació basat en màquines d'estat facilita al dissenyador la visualització de tots els possibles estats i transicions entre estats en funció de les diferent entrades o variables del sistema. D'aquesta manera, resulten una manera força natural de pensar i resoldre els problemes de seqüenciació típics de sistemes encastats.

Com hem vist, l'arquitectura de màquina d'estats és particularment adequada per a dissenys que duen a terme una única funció. No obstant això, sovint cal implementar sistemes complexos amb múltiples funcions més o menys independents governades per diferents màquines d'estat. Les màquines d'estat jeràrquiques solucionen, de manera simple, aquesta necessitat.

3.2.3. Màquines d'estat jeràrquiques

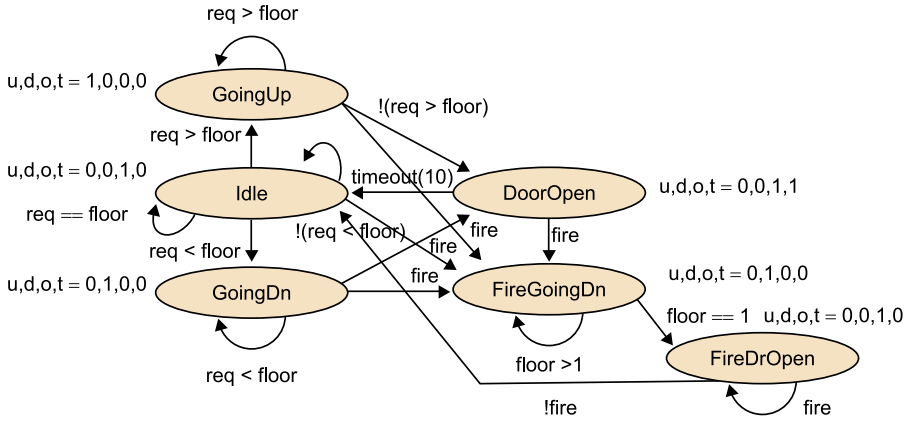
Exemple: controlador d'ascensor amb mode d'incendis implementat amb màquina d'estats convencional

Continuant amb l'exemple anterior, suposem que volem afegir un mode de funcionament en cas d'incendi al sistema de control de l'ascensor anterior. Una possible descripció funcional d'aquest sistema seria:

"En cas d'incendi, desplaceu l'ascensor a la primera planta i deixeu les portes obertes".

Si ens basem en una màquina d'estats convencional, aquesta petita modificació complica notablement el diagrama d'estats, ja que cal afegir-hi múltiples estats, variables, transicions; vegeu la figura següent:

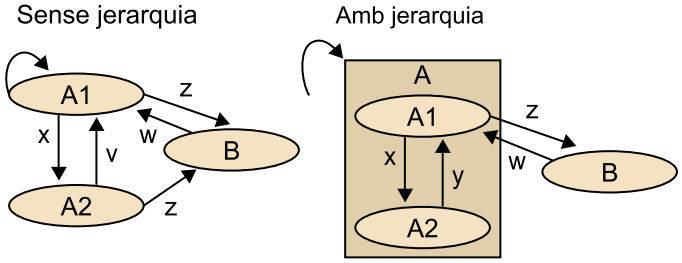
Diagrama d'estats d'un controlador d'ascensor amb mode d'incendis



En les **màquines d'estats jeràrquiques**, alguns estats poden consistir en màquines d'estat completes i independents que s'executen només quan s'accedeix a aquests estats.

La figura següent mostra un exemple d'una màquina d'estats amb jerarquia i sense. En ambdós casos, la funcionalitat és la mateixa, però l'ús de jerarquia permet visualitzar fàcilment que l'execució de B és independent del funcionament normal de A i que només cal una única transició *z* per a accedir-hi.

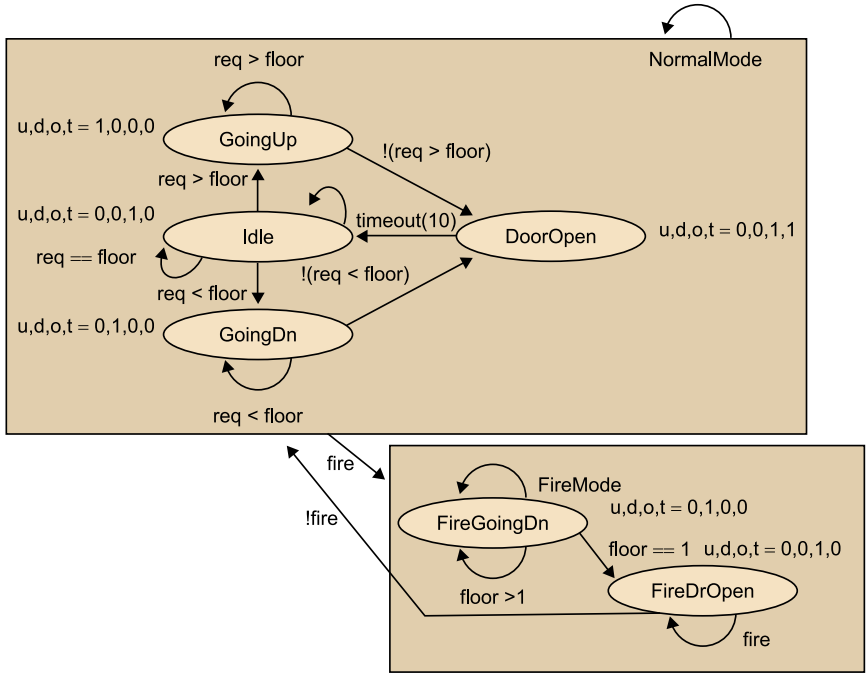
Diagrama d'estat d'una màquina d'estats simple implementada amb jerarquia i sense



Exemple: controlador d'un ascensor implementat amb una màquina d'estats amb jerarquia

Tal com es mostra en la figura, l'exemple del control de l'ascensor amb mode en cas d'incendi se simplifica notablement si fem servir una màquina d'estat jeràrquica que atengui de manera específica la situació d'incendi `(fire == 1)`.

Diagrama d'estats d'una possible implementació de l'ascensor del subapartat 3.2.1 amb màquines d'estat jeràrquiques



En aquest cas, el codi font es veuria modificat de la manera següent:

```
#define NORMAL 0            // definim modes de treball
#define FIRE 1

#define N_IDLE 0            // definim estats del 1r. mode
#define N_GOINGUP 1
#define N_GOINGDN 2
#define N_DOOROPEN 3

#define F_GOINGDN 0      // definim estats del 2n. mode
#define F_DOOROPEN 1

void UnitControl() {
        int mode = NORMAL;
        int state = N_IDLE;

        while (1) {
        switch (MODE) {
              NORMAL:
                    switch (state) {
```

```
                N_IDLE: up=0; down=0; open=1; timer_start=0;
                     ...
                N_GOINGUP: up=1; down=0; open=0; timer_start=0;
                     ...
                N_GOINGDN: up=1; down=0; open=0; timer_start=0;
                     ...
                N_DOOROPEN:up=0; down=0; open=1; timer_start=1;
                     ...
                }

                FIRE:
                     switch (state) {
                F_GOINGDN: up=0; down=1; open=0; timer_start=0;
                     ...
                F_DOOROPEN:up=0; down=0; open=1; timer_start=0;
                     ...
         }
         }
}
```

3.3. Sistemes multitasca

Les estratègies de programació vistes fins al moment no resulten adequades per a atendre un gran nombre de tasques que cal fer simultàniament.

Vegem un exemple del que entenem per un *sistema amb múltiples tasques concurrents*.

Sistemes multitasca elementals

Es demana programar un sistema capaç de dur a terme les accions següents:

1) Llegir dos nombres diferents *X* i *Y*.
2) Imprimir el missatge "Hola, Món" cada *X* segons
3) Imprimir el missatge "Com estàs?" cada *Y* segons.

Es tracta, doncs, d'una aplicació que requereix l'execució de dues tasques de manera concurrent (per exemple, imprimir dos missatges, cadascun amb el seu ritme de repetició independent) que no seria obvi de capturar en funció de màquines d'estat o qualsevol de les estratègies vistes anteriorment. Una possible implementació d'aquesta funcionalitat podria ser la següent:

```
void main_ConcurrentTaskExample() {
       x = ReadX();
       y = ReadY();

       Call concurrently {
              PrintHelloWorld(x) and PrintHowAreYou(y)
       }
}

void PrintHelloWorld(x) {
       while( 1 ) {
              print("Hello world.");
              delay(x);
       }
```

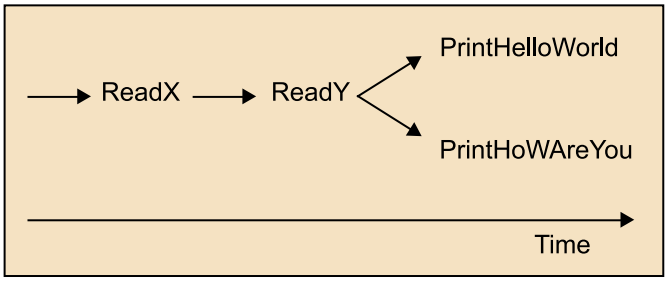
```
}


void PrintHowAreYou(x) {
       while( 1 ) {
              print("How are you?")
              delay(y);
       }

}
```

Segons aquest codi, la seqüència d'execució de les diferents rutines al llarg del temps seria:

Seqüència de funcionament del codi de l'exemple de l'apartat 3.3.3



Parem atenció al fet que les diverses tasques d'un sistema encastat multitasca poden i solen estar basades, cadascuna d'elles, en un bucle simple de control.

L'aspecte de la sortida del programa podria ser:

```
Enter X: 1
Enter Y: 2
Hello world. (Time = 1 s)
Hello world. (Time = 2 s)
How are you? (Time = 2 s)
Hello world. (Time = 3 s)
How are you? (Time = 4 s)
Hello world. (Time = 4 s)
...
```

Si bé la majoria de llenguatges de programació permet elaborar programes amb múltiples tasques, la seva correcta execució requereix tenir en compte un seguit de conceptes teòrics relacionats amb la gestió de múltiples tasques.

S'anomena *gestió multitasca* el procés de planificar tasques de manera que **sembli que s'executen simultàniament**. Aquesta funció inclou l'activació/desactivació de tasques, el control de les prioritats i la planificació de la seva execució.

Atès que bona part de les implementacions reals de sistemes multitasca es basa en la utilització de sistemes operatius, primerament, es dedica una petita secció a la seva introducció. Tot seguit, oferim diversos apartats en què es descriuen les característiques fonamentals de la gestió multitasca.

Vegeu també

El mòdul "Controladors i sistemes operatius" es dedica íntegrament a l'estudi de l'ús dels sistemes operatius.

3.3.1. Introducció als sistemes operatius en temps real

Una de les maneres més eficaces d'implementar característiques de funcionament en temps real en sistemes encastats amb funcionalitat multitasca complexa és mitjançant l'ús de sistemes operatius en temps real[5].

(5)En anglès, *real-time operating systems* (RTOS), també coneguts com a *real-time executives* o *real-time kernels*.

Aquesta metodologia de treball es basa en l'ús d'un programari bàsic predesenvolupat: el sistema operatiu, que, entre d'altres, gestiona l'accés al maquinari per a l'execució de les diferents tasques previstes. Tradicionalment, aquesta necessitat se satisfeia implementant controladors específics per a cada sistema encastat i cada perifèric. A mesura que aquests perifèrics han crescut en complexitat i que les interfícies estàndard s'han fet més habituals, s'ha demostrat que reescriure aquests elements de programari des de zero per a cada aplicació és una estratègia de programació molt poc eficient. Els sistemes operatius incorporen aquests controladors i estalvien molta feina al programador, la qual cosa facilita notablement el desenvolupament de sistemes encastats que interactuïn amb gran varietat de perifèrics.

Al mateix temps, i a mesura que la funcionalitat dels sistemes encastats creix en complexitat, es fa necessari gestionar de manera eficient l'execució simultània de diferents tasques amb els recursos disponibles.

Per tant, de la mateixa manera que els sistemes operatius convencionals, els RTOS controlen els diferents recursos del sistema encastat i gestionen l'accés als mateixos de les diferents tasques. Els RTOS tenen una característica addicional que els diferencia dels sistemes operatius convencionals: són **deterministes**, en el sentit que el temps necessari per a executar una tasca determinada és conegut i està completament especificitat. Aquesta característica permet avaluar el temps que cal per a executar les diferents tasques i, per tant, permet acomplir requisitts de funcionament en temps real.

Habitualment, els RTOS es distribueixen en forma o bé de nuclis (*kernels*), o bé de sistemes operatius complets. Les versions nucli normalment implementen un sistema bàsic de gestió de tasques i de memòria. Les versions de sistema operatiu complet poden incloure controladors per a una gran varietat de dispositius com unitats de disc, interfícies i ports. Una altra característica habitual dels RTOS és que solen requerir que el maquinari del sistema generi un interrupció periòdica (anomenada *comptador de temps* o *temporitzador*) que és utilitzada com a base de temps de tot el sistema de temps real i que permet planificar les tasques.

En resum, les funcions principals d'un RTOS són:

- Gestió multitasca.
- Comunicació i sincronització de tasques.
- Gestió de recursos i memòria.

En les seccions següents, es continuaran descrivint les metodologies i eines que permeten el funcionament multitasca des d'una perspectiva propera a les implementacions reals, és a dir, assumint en la majoria dels casos que es treballa amb un RTOS.

3.3.2. Activació/desactivació de tasques

En aquesta secció es descriu com es manté el control de les diferents tasques que cal dur a terme en entorns multitasca.

La majoria de sistemes multitasca, especialment els basats en RTOS, manté una llista que recull les diferents tasques que cal executar, juntament amb la seva prioritat, anomenada **llista d'execució**.

Quan una tasca passa a estar llesta (*ready*) per a la seva execució, s'afegeix a la llista i s'executa quan correspongui, en funció dels diferents esquemes de prioritització i planificació que es descriuen en els apartats següents.

Tasques actives

Es considera que una tasca està activa quan s'està executant de manera efectiva en el processador. Una tasca llesta simplement es troba en llista pendent de la seva execució.

Si una tasca deixa d'estar llesta (*not ready*) és eliminada immediatament de la llista de tasques.

Quan una tasca està activa (per exemple, si s'està executant al processador) es pot donar alguna circumstància que n'impedeixi temporalment l'execució, com, per exemple, que depengui de la finalització d'altres tasques no completades. En aquest cas, la tasca deixaria d'estar llesta, s'eliminaria de la llista d'execució i se n'interrompria l'execució fins a la finalització de les tasques de les quals depèn.

Exemple

Una tasca de gestió de dades introduïdes per teclat només estaria llesta quan s'hagués premut alguna tecla.

Els mecanismes lògics que menen a la inclusió o eliminació d'una tasca determinada de la llista d'execució s'anomenen *planificació de tasques*.

Aquesta capacitat d'activar o desactivar tasques permet als sistemes multitasca respondre de manera molt complexa als canvis en el seu entorn. En certa manera, emulen el funcionament habitual del món real. Per exemple, podem tenir previst treballar tot el dia a l'oficina, però una topada amb el cotxe de camí a la feina pot fer que tota la jornada de treball es retardi. Més tard, la reunió prevista es cancel·la i cal ocupar el temps en alguna altra activitat.

3.3.3. Prioritat

Resulta obvi que no totes les tasques que ha de dur a terme un sistema encastat requereixen el mateix grau de diligència.

Exemple

En cas d'accident, és més important que el sistema de control del coixí de seguretat garanteixi el seu desplegament a temps, que no pas que atengui a una rutina d'autoverificació que, casualment, estigui programada per a executar-se al mateix temps.

S'anomena *prioritat* la *urgència relativa d'una tasca o interrupció en comparació d'una altra*. Se sol indicar amb un nombre enter associat a cada tasca o a cada tipus d'interrupció.

En el cas de tasques, determina el temps i el torn de procés que li assignarà el planificador per a la seva execució.

Nota

Cal recordar que hi ha interrupcions amb prioritat màxima sempre, com l'NMI destinada a avisar d'un error fatal del sistema.

En el cas d'interrupcions, determina com cal procedir en cas que es produeixin altres interrupcions mentre és atesa per la ISR.

3.3.4. *Time-slicing*

Si es disposa d'una única unitat de procés, el maquinari només podrà atendre l'execució d'una única tasca en cada instant de temps. Aquest és el cas de la majoria de sistemes encastats. Això, però, no impedeix que molts d'aquest sistemes puguin oferir un comportament aparentment de multitasca.

El *time-slicing* és una de les possibles maneres de simular el funcionament multitasca en una única unitat de procés. Es basa a dividir el temps d'execució en intervals regulars (o franges de temps) i, durant aquest temps, dedicar el processador i la resta de recursos del sistema a l'execució d'una única tasca.

Per a determinar aquests intervals, s'utilitzen les interrupcions generades pel comptador de temps del sistema.

En rebre la interrupció de final d'interval, el programari encarregat de la gestió multitasca, habitualment un RTOS, atura l'execució de la tasca actual i passa el control a la següent, que s'executa durant l'interval següent.

Aquest procés es repeteix cíclicament recorrent totes les tasques incloses en la llista d'execució (per exemple, tasques llestes). Així, doncs, el temps necessari per a completar una tasca determinada depèn fortament del nombre de tasques que s'executen simultàniament en el sistema.

Amb aquesta metodologia, resulta molt simple tenir en compte també la prioritat de les diferents tasques: el planificador de tasques pot preveure que les que gaudeixin d'una alta prioritat siguin executades durant més d'un interval de temps consecutiu.

Tanmateix, si una tasca finalitza la seva activitat abans d'acabar l'interval assignat, pot donar el temps sobrer a la tasca següent, cosa que fa possible un aprofitament més gran del temps.

Per tal de garantir, en el moment de reprendre una tasca, que aquesta continua exactament en el punt on havia estat aturada cal desar un seguit d'informació.

S'anomena **context** les dades contingudes en els diferents registres i piles del sistema en el moment de la interrupció.

Per tant, cal restituir aquest context per a poder continuar treballant en les mateixes condicions. El context inclou, entre d'altres, la informació relativa a la següent instrucció pendent d'execució (punter d'instruccions) i totes les dades associades a la tasca (punter de pila, espai de memòria assignat, registres de dades, etc.).

El gestor de multitasca, sovint una part de l'RTOS, és el responsable de desar i restituir el context corresponent en el moment d'activar i desactivar les diferents tasques. Per això, utilitza les eines de control de tasques.

Vegeu també

Podeu veure que es descriuen en el subapartat 3.4.6, "Eines per a la comunicació i sincronització de tasques".

Si es disposa d'una única unitat de processament, el maquinari només podrà atendre l'execució d'una única tasca en cada instant de temps. Aquest és el cas de la majoria de sistemes encastats. Això, però, no impedeix que molts d'aquests sistemes puguin oferir un comportament aparentment de multitasca.

3.3.5. Planificació de tasques

Vegem ara diferents tècniques de planificació de tasques.

Planificació seqüencial

La planificació seqüencial implica que les diferents tasques que cal dur a terme s'executen de manera successiva, una darrere de l'altra, passant el control a la següent només quan la primera ha acabat.

Aquesta estratègia és molt simple d'implementar, però resulta particularment ineficient quan cal fer tasques que deixen el sistema en espera, com, per exemple, esperar que l'usuari premi una tecla o esperar a l'estabilització de la tem-

peratura dins d'una cambra. Resulta obvi que, en aquests casos, la planificació seqüencial porta el sistema a llargs períodes d'inactivitat que podrien ser utilitzats en tasques més profitoses.

Un altre inconvenient dels sistemes de planificació seqüencials és que atenen totes les tasques amb la mateixa prioritat.

Exemple

Aquest fet pot arribar a ser problemàtic; per exemple, si quan es rep una interrupció urgent, el sistema ha d'esperar a finalitzar la tasca en curs per a poder-la atendre.

Un tercer inconvenient de la planificació seqüencial és la seva ineficiència per a gestionar un nombre elevat de tasques. Poden donar-se situacions en les quals els recursos del sistema no donin a l'abast per a atendre una allau de tasques llestes per a la seva execució en un interval de temps curt. Els sistemes seqüencials no solen implementar mecanismes que evitin el desbordament de les capacitats computacionals del sistema per un excés de tasques pendents.

Per tots aquests motius:

> Els sistemes que únicament implementen planificadors seqüencials es considera que no són capaços de treballar en multitasca.

En qualsevol cas, cal reconèixer el principal avantatge que té. En un sistema amb planificació seqüencial, és possible predir quines instruccions seran executades pel processador, amb quin ordre i en quin moment ho farà. De fet, si es coneixen totes les entrades al programa, és possible predir-ne els resultats i el comportament, independentment que el sistema executi les instruccions amb més o menys celeritat. Per aquesta mateixa raó, en aquest tipus de sistema resulta simple analitzar el temps necessari per a la finalització de les tasques; i, per tant, faciliten la implementació de sistemes en temps real.

Planificació prioritària

A la pràctica, el funcionament seqüencial és rar. Fins i tot en sistemes basats en bucles de control simple, és molt freqüent haver d'atendre interrupcions. Quan es rep una interrupció, la ISR es posa en marxa per a atendre-la i altera el flux normal de l'execució.

Per tant, tot i que sempre cal garantir el funcionament correcte del sistema, el paradigma de seqüencialitat i predictibilitat de la sèrie d'instruccions executades al processador rarament es compleix.

Una de les estratègies més comunes per a aconseguir que el trencament de la seqüencialitat prevista no vagi en contra de la funcionalitat que es vol és la utilització de la prioritat de les diferents tasques quan se'n planifica l'execució.

La **planificació prioritària** (*preemptive schedulling*) és el mètode més comú de planificar tasques en RTOS i es basa en el principi següent:

"Una tasca s'executa fins que s'acaba i fins que una tasca de prioritat més alta ho requereix."

- Idealment, en un sistema basat en planificació prioritària, una tasca de baixa prioritat quedaria interrompuda en el moment en què una tasca de prioritat alta entrés a la llista d'execució.

- De manera similar, si calgués executar una tasca de més prioritat (com, per exemple, la ISR), el planificador li cediria el control.

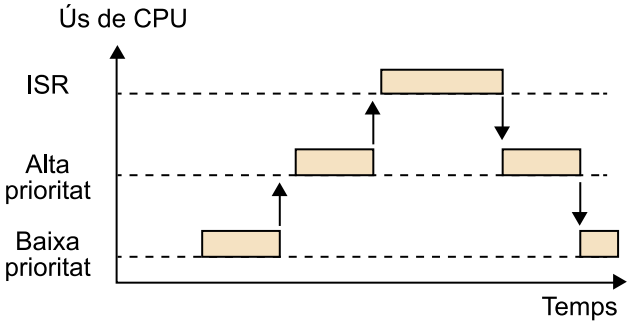
- Una vegada acabada la tasca de més prioritat, s'emprendria la resta de tasques per ordre invers de prioritats (vegeu la figura següent).

Aquest procés de jerarquització de les prioritats s'anomena *interrupcions imbricades*, o *nested interrupts*, tal com s'han definit en la secció destinada a descriure les interrupcions.

Vegeu també

Us recomanem repassar l'apartat 2.1, "Interrupcions".

Seqüència temporal de la transferència del control



Per exemple, tasca activa o en execució en cada instant de temps, en un sistema multitasca amb planificació prioritària que mostra prioritats imbricades. Típicament, la ISR sol tenir assignada una prioritat més gran a la resta de tasques. Quan la ISR finalitza, retorna el control a la tasca de més prioritat pendent d'execució.

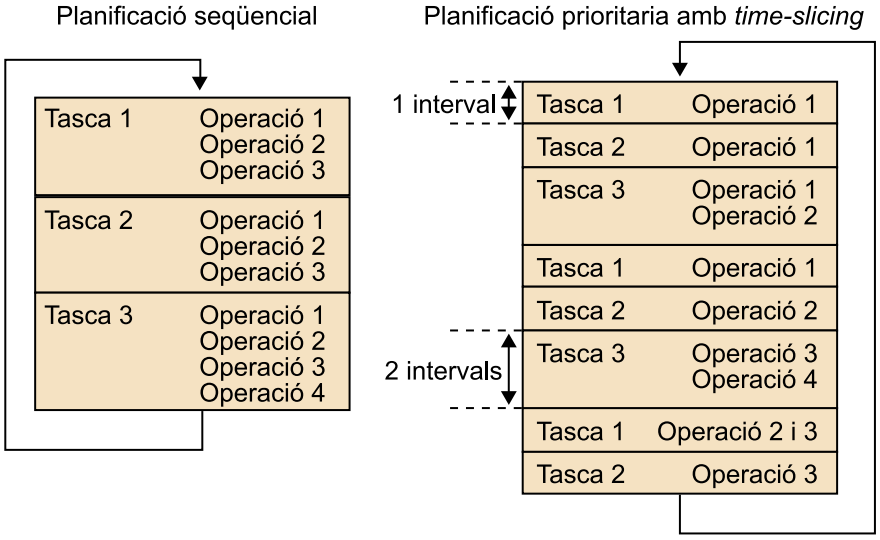
A la pràctica, els planificadors prioritaris solen utilitzar esquemes de *time-slicing* per a repartir el temps de processador entre les diferents tasques. Com a regla general, a una tasca d'alta prioritat simplement, s'hi assignen més intervals de temps d'execució que a una de baixa prioritat. Només en casos extrems, com, per exemple, si cal atendre una interrupció de molt alta prioritat, es pot arribar a recórrer a interrompre la tasca en curs abans que finalitzi el seu interval. Sovint, però, el que es fa és atendre les interrupcions en finalitzar l'interval en curs.

Hi ha nombrosos algorismes que permeten determinar el nombre de cicles òptim que s'ha d'assignar a cada tasca en funció de la seva prioritat, de manera que s'optimitzi algun paràmetre de funcionament del sistema, com, per exemple, el temps d'execució de totes les tasques, la tasca més prioritària, etc. Entre els més utilitzats, destaca el *rate monothonic algortihm* (RMA), que permet maximitzar el nombre de tasques que es completen dins dels terminis previstos.

Tal com es mostra en la figura següent, mentre que en una planificació seqüencial les tasques s'executen fins a la seva finalització, en una planificació prioritària amb *time-slicing* les tasques s'executen fins que se'ls acaba el temps

disponible. Aquesta diferència soluciona la majoria dels problemes habituals de la planificació seqüencial i permet desplegar tot el potencial dels sistemes multitasca.

Esquemes de dues metodologies d'execució de múltiples tasques

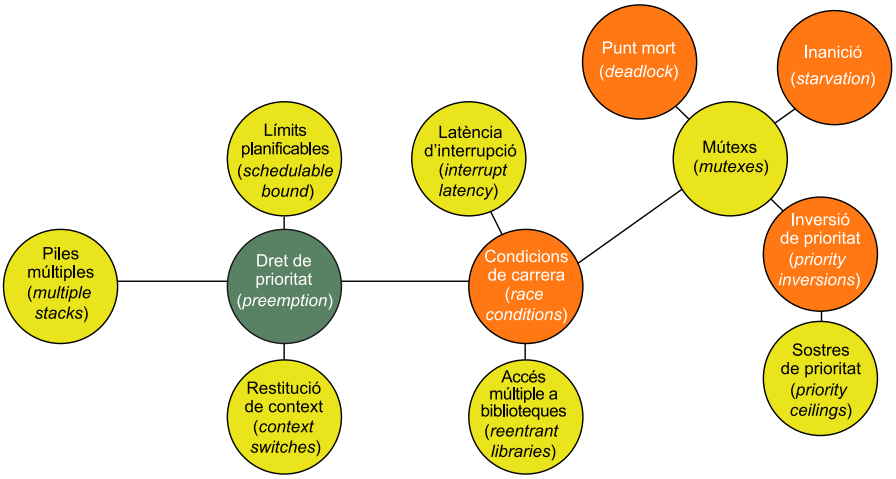


Suposem que el sistema ha d'executar tres tasques de 2, 3 i 4 operacions, respectivament. La darrera tasca té una prioritat més alta que les anteriors. En un sistema basat en un planificador seqüencial, cada tasca està activa (per exemple, s'executa) fins que es completa l'execució de totes les seves operacions. En canvi, en un sistema basat en *time-slicing*, el temps d'execució es divideix en intervals i es va repartint cíclicament entre les diferents tasques. Fixeu-vos que a les tasques de prioritat més alta, se'ls pot assignar més d'un interval. En aquest cas, les operacions de cada tasca s'executen fins que s'acaba el temps de l'interval assignat. Fixeu-vos que, si una operació no té temps de finalitzar, continua en l'interval següent (vegeu "Tasca 1, operació 1"). Si l'operació no utilitza tot l'interval assignat, la resta del temps es pot destinar a completar altres operacions de la mateixa tasca (vegeu "Tasca 1, operacions 2 i 3").

Perills de la planificació prioritària

Cal tenir present que el fet de no controlar de manera directa la commutació entre les diferents tasques pot ser una font de problemes. La figura següent recull fins a deu possibles problemes associats a una mala utilització de la planificació amb prioritat.

Esquema dels principals problemes que pot generar la planificació prioritària



En groc (vegeu-ne la versió electrònica del PDF o el web) s'indiquen els problemes que poden ser considerats simples inconvenients. En taronja, s'identifiquen els aspectes que comporten errors de funcionament.

Analitzarem les situacions més rellevants a continuació:

- **Pèrdua de rendiment per excés de planificació vinculant**: es pot demostrar que per a aconseguir un màxim grau d'acompliment dels terminis d'execució d'un conjunt arbitrari de tasques mitjançant algorismes RMA pot ser necessari desaprofitar (per exemple, deixar un processador en estat inactiu) fins al 31% del temps de procés disponible, en el pitjor dels casos. Aquest fet implica arribar a disposar només del 69% de les presentacions del maquinari si es necessita disposar de funcionament multitasca planificat en temps real.

- **Piles múltiples**: és una necessitat intrínseca dels sistemes multitasca planificats amb prioritat, ja que la necessitat de reprendre tasques pausades en el punt en què es van aturar implica l'obligació d'emmagatzemar en memòria tota la informació de la pila de cadascuna de les tasques mentre no s'hagin finalitzat completament. En sistemes amb un gran nombre de tasques concurrents, aquest fet dóna encara més rellevància a les limitacions de memòria habituals en sistemes encastats.

- **Restitució de context**: relacionat amb el problema anterior, la continuació d'una tasca pausada implica la restitució completa del context del sistema; és a dir, dels valors de múltiples registres tal com estaven inicialment. A banda d'ocupar espai de memòria, la restitució d'aquesta informació implica executar un seguit d'instruccions addicionals que prenen temps de procés a l'execució efectiva de les tasques.

En la majoria d'aplicacions, les tasques estan fortament relacionades i intercanvien informació. Sovint, aquest intercanvi d'informació requereix manipular les dades d'una mateixa regió de la memòria en els diferents intervals d'execució assignats a cada tasca. Això implica un risc intrínsec de corrupció de les dades si no es treballa amb cura.

- **Condicions de carrera**: són les situacions en les quals els efectes combinats de dues o més tasques concurrents varien segons l'ordre precís en el qual s'ha executat la seqüència d'instruccions combinades de l'una i de l'altra. Tot seguit, en veurem alguns exemples.

Exemple: condicions de carrera en tasques accedint a variables globals

a) Suposem que *x* i *y* són variables globals:

```
global int x;
global int y:

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA {
      x = 1;
      y = 2;
      if ( y != 0 ) {
           z = x/y;
      }
}

void TaskB {
      y = 0;
}
```

Si la tasca B interromp l'execució de la tasca A just després d'avaluar la condició i abans d'efectuar la divisió, el retorn a la tasca A portarà a una divisió per zero.

b) Suposem ara que hi ha cinc tasques concurrents que executen el mateix codi sobre la variable global *x*:

```
global int x;

void main() {
      Call concurrently {
            5*ConcurrentTask();
      }
}

void ConcurrentTask {
      for ( int i = 0; i< 100000; i++ ) {
            x = x + 1;
      }
}
```

En la majoria dels casos, aquest programa no comportarà la solució esperada (*x* = 500.000). La raó és que cada addició implica accedir al valor de *x*, efectuar l'increment i tornar-lo a desar a la variable global. En un esquema de planificació prioritària, les diferents tasques es poden anar interrompent mútuament, de manera incerta, entre cadascun d'aquests passos de manera que s'efectuïn addicions sobre valors no actualitzats.

- **Latència d'interrupció**: el temps d'espera entre l'enviament d'un senyal d'interrupció i l'execució efectiva de la ISR (temps de latència d'interrupció) sol ser més gran en sistemes basats en planificació prioritària pel fet que aquests poden desencadenar condicions de carrera en les tasques que es trobessin en execució. S'anomenen *seccions crítiques* els blocs de codi que s'han d'executar de manera estrictament seqüencial, ja que, en cas de no fer-ho, poden produir resultats erronis. En aquests casos, la solució passa per protegir aquestes seccions crítiques deshabilitant temporalment l'anotació a interrupcions durant la seva execució. Aquesta estratègia, òbviament, pot tenir conseqüències en la capacitat del sistema per a respondre en temps real.

- **Accés múltiple a biblioteques**: és possible que les condicions de carrera es produeixin entre dos accessos concurrents a una mateixa biblioteca compartida. Es tracta de situacions especialment preocupants, ja que són condicions de carrera entre dues seccions de codi idèntiques que poden intentar fer el mateix sobre els mateixos recursos, simultàniament. En aquest cas, no sol ser possible deshabilitar les interrupcions, ja que es tracta de seccions de codi tancades i de propòsit molt general en les quals cal evitar introduir comportaments inesperats. La solució sol ser fer que aquestes parts del codi s'executin de manera anormalment lenta, i evitar així que puguin arribar a ser concurrents.

- **Mútex**: com a solució per a evitar corrupcions de dades associades amb condicions de carrera, se sol recórrer a implementar indicadors binaris (els mútexs) que permetin bloquejar l'accés a un conjunt de dades en memòria. Formen part de les eines incloses habitualment en els RTOS. La tasca del programador consisteix a identificar les dades que poden ser corrompudes i les seccions crítiques del codi per, tot seguit, crear un mútex associat a aquestes dades. A partir d'aquí cal que el codi doni a cada secció crítica indicacions per a reservar-se l'accés al mútex abans d'iniciar-ne l'execució i d'alliberar-lo en finalitzar.

Exemple: condició de carrera evitada amb un mútex

El primer exemple del subapartat 3.4 es podria protegir mitjançant un mútex de la manera següent.

```
global int x;
mutex mutex1(x);      // associem un mutex a la variable x

void main() {
      Call concurrently {
            5*ConcurrentTask()
      }
}


void ConcurrentTask {
      for ( int i = 0; i< 100000; i++ ) {
            mutex.lock();      // bloquegem el mutex
            x = x + 1;
            mutex.unlock();    // alliberem el mutex
      }
}
```

En el pla pràctic, els mútexs solen ser causa d'altres problemes i, per tant, cal evitar-los.

- **Inanició**: si dues tasques de prioritat molt diferent comparteixen un mateix mútex, es pot donar el cas que la de menys prioritat no pugui mai finalitzar en el termini esperat, ja que la de més prioritat (que pot tenir assignats molts més temps d'execució) no allibera mai el mútex en el moment de cedir el control a la de baixa prioritat i, per tant, es veu forçada a retornar el control a l'anterior.

- **Punt mort (*deadlock*)**: en aquest cas, dues tasques de prioritat similar reserven un mútex en el seu torn d'execució, però si el seu interval d'execució finalitza abans de poder-lo alliberar, l'altra sempre el troba bloquejat. Això comporta una situació indefinida en la qual, a totes dues tasques, se'ls assignen torns d'execució als quals van renunciant alternativament ja que troben un mútex bloquejat. En aquests casos, la solució sol ser reiniciar el sistema. A continuació, veurem un exemple de punt mort.

Exemple: mútexs múltiples que causen una situació de punt mort

```
mutex mutex1, mutex2;

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA() {
      while( 1 ) {
            ...
            mutex1.lock();
            /* secció crítica 1 */
            mutex2.lock();
            /* secció crítica 2 */
            mutex2.unlock();
            /* secció crítica 1 */
            mutex1.unlock();
      }
}

void TaskB() {
      while( 1 ) {
            ...
            mutex2.lock();
            /* secció crítica 2 */
            mutex1.lock();
            /* secció crítica 1 */
            mutex1.unlock();
            /* secció crítica 2 */
            mutex2.unlock();
      }
}
```

En aquest exemple, calen dos mútexs (mutex1, mutex2) cridats en dues tasques concurrents (TaskA i TaskB). Seguint la seqüència d'execució s'observa que:

- TaskA bloqueja mutex1 per a executar la secció crítica 1 de manera segura.
- TaskB bloqueja mutex2 per a executar la secció crítica 2 de manera segura.
- Task i TaskB executen les seccions crítiques 1 i 2, respectivament.
- TaskA intenta bloquejar mutex2 per a treballar ara sobre la secció crítica 2 i queda bloquejada a l'espera que TaskB alliberi mutex2.
- TaskB intenta bloquejar mutex1 per a treballar ara sobre la secció crítica 1 i queda bloquejada a l'espera que TaskA alliberi mutex1.

Veiem, per tant, que aquesta forma de programació comporta una situació de punt mort.

- **Inversió de prioritat**: aquesta situació és un xic més subtil que les dues anteriors, ja que requereix el concurs de tres tasques de prioritats creixents en què les de més i menys prioritat comparteixen un mútex. Si la de més prioritat quedés a l'espera que la de menys prioritat alliberi el mútex,

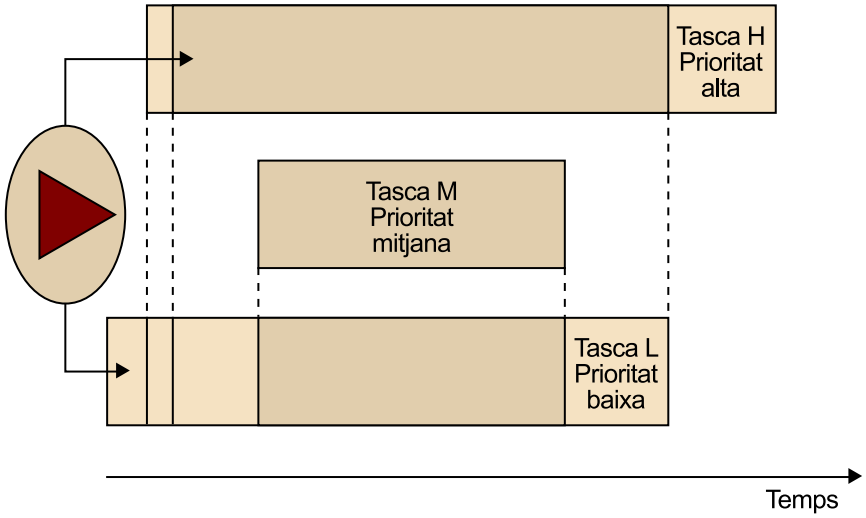
es podria donar la situació que una tasca de prioritat intermèdia impedís l'execució de la de menys prioritat i, per tant, impedís l'alliberament del mútex. El resultat final és que una tasca de menys prioritat (la intermèdia) evita que una altra de més prioritat pugui accedir al processador.

La inversió de prioritat sol ser considerada un error del sistema, ja que viola les prioritats naturals.

Hi ha dues solucions habituals. La primera, anomenada **herència de prioritat**, consisteix a assignar a la tasca de baixa prioritat la mateixa prioritat que a la tasca d'alta prioritat durant el temps que bloqueja el recurs compartit amb aquesta.

La segona, l'establiment de **sostres de prioritat**, consisteix a assignar a cada recurs compartit (que són els que, bloquejats mitjançant mútexs, generen les situacions d'inversió de prioritat) una prioritat tan elevada, com a mínim, com la tasca de més prioritat que l'utilitzi. Malauradament, aquesta solució també viola les prioritats naturals, ja que pot fer que una tasca de baixa prioritat, que utilitza un recurs considerat d'alta prioritat, avanci a una tasca de prioritat intermèdia.

Una tasca de baixa prioritat (Tasca L) i una d'alta prioritat (Tasca H) comparteixen un recurs controlat amb un mútex



Suposem que just després que L bloquegi el mútex, H queda llesta per a executar-se. No obstant això, H ha d'esperar que L deixi de necessitar el recurs (l'alliberi) per a poder continuar. Per tant, queda com a no llesta i pendent d'execució. Si, en aquestes circumstàncies, una tasca de prioritat intermèdia (Tasca M) queda llesta per a executar-se, avançarà a L i li impedirà continuar amb la seva activitat, gràcies a la seva prioritat. El resultat global és que la tasca M de prioritat intermèdia s'ha executat abans que la tasca H d'alta prioritat. Per tant, s'ha produït el fenomen de la inversió de prioritat.

Com es veurà més endavant, si es fa servir un RTOS modern, es té accés a eines alternatives (les bústies de correu), que són el mètode segur i correcte d'intercanviar informació entre tasques, ja que eviten part dels problemes de bloqueig de recursos i també qualsevol corrupció de les dades.

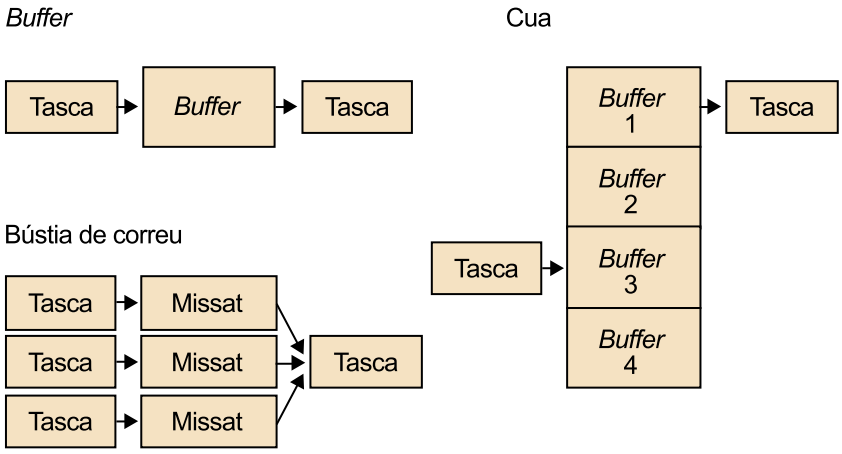
Tot el que s'ha descrit en aquesta secció té el seu origen en el caràcter incert (a efectes pràctics és quasi aleatori) amb la qual cosa s'intercalen les tasques en els sistemes basats en planificació amb prioritat. En les properes seccions es descriuen algunes de les eines que permeten prevenir i evitar els inconve-

nients de la planificació amb prioritat. En qualsevol cas, és responsabilitat del programador decidir si aquest tipus de planificació és la més adequada per a la seva aplicació, o bé si n'hi hauria prou amb altres aproximacions més simples.

3.3.6. Eines per a la comunicació i sincronització de tasques

Hi ha diverses eines de programació que permeten satisfer les necessitats d'intercanviar informació entre tasques (comunicació) i controlar-ne el ritme i l'ordre d'execució (sincronització). En sistemes basats en RTOS, el mateix sistema operatiu les posa a disposició del programador. Poden ser també implementades pel programador en sistemes dissenyats des de zero. Vegem-ne les més habituals. La figura següent mostra uns esquemes del seu funcionament bàsic:

Representació esquemàtica de les eines per a la comunicació entre tasques més habituals



Els semàfors permeten regular la sincronia entre tasques. En l'exemple, la tasca A crida la tasca B i queda a l'espera de la seva finalització. Per això, roman inactiva fins que el semàfor, activat per la finalització de la tasca B, es posa en verd. Els *buffers* permeten intercanviar dades entre diferents tasques de manera simple. Per a intercanvi d'informacions permanents, es poden emprar les cues, consistents ens seqüències de *buffers* que es van omplin successivament. Cada vegada que un *buffer* queda ple, pot ser llegit per una altra tasca. Finalment, les bústies de correu permeten l'enviament de missatges entre diferents tasques.

Buffers

El *buffers* permeten intercanviar dades entre tasques. Típicament, la tasca font de dades sol·licita un *buffer* a l'RTOS (o a qualsevol altre programari que s'encarregui de la gestió dels *buffers*), hi transfereix les dades i demana que siguin transferides a la tasca de destinació. En aquest moment, l'RTOS passa a la tasca de destinació un punter al *buffer* de dades amb informació sobre la longitud de les dades que ha de llegir.

Exemple: intercanvi de dades entre tasques mitjançant un *buffer*

El codi següent mostra un exemple de com cal transferir dades entre dos processos concurrents mitjançant un *buffer*:

```
int N = bufferSize;
dataArray buffer[N];
int count = 0;

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
```

```
}

void TaskA() {
      int i;
      while( 1 ) {
            produce(&data);
            while( count == N );      //espera si buffer és ple
            buffer[i] = data;
            i = (i + 1) % N;
            count = count + 1;
      }
}

void TaskB() {
      int i;
      while( 1 ) {
            while( count == 0 );      //espera si buffer és buit
            data = buffer[i];
            i = (i + 1) % N;
            count = count - 1;
            consume(&data);
      }
}
```

Observem que la TaskA és només font de dades, mentre que la TaskB n'és la usuària.

Atès que cada tasca genera o consumeix dades contínuament, s'ha implementat una verificació (basada en un bucle while) que comprova que, o bé el buffer té posicions disponibles (TaskA), o bé és completament buit.

Aquesta solució no és la millor, ja que és possible que es corrompin les dades de la variable global count (que registra el nombre de posicions ocupades en el *buffer*). En el darrer exemple del subapartat 3.4.5, "Mútexs múltiples que causen una situació de punt mort", es mostra una solució basada en els mútexs.

Cues

Les cues són cadenes de *buffers* que permeten l'intercanvi d'un volum més gran de dades d'una manera seqüencial. Per exemple, metre la tasca destinació processa la informació del primer *buffer* de la cua, la tasca font pot començar ja a omplir de dades el segon.

Els *buffers* i les cues representen les formes més bàsiques d'intercanviar informació entre tasques. En combinació amb les eines de sincronització, que es descriuen a continuació, es pot protegir les dades del *buffer* davant la corrupció (ja que sovint són recursos compartits).

Mútexs

Els mútexs són indicadors binaris utilitzats usualment per a protegir recursos compartits d'accessos simultanis durant l'execució de seccions crítiques de codi.

Per tal d'il·lustrar-ne el funcionament, podem pensar que el mútex funciona de manera anàloga a com es gestiona la clau del lavabo en un bar:

- Quan un client (tasca) vol accedir a aquest recurs, demana la clau al cambrer (RTOS).

- Si aquest no té la clau, vol dir que hi ha un altre client que l'utilitza i, per tant, ha d'esperar.

- En el moment en què la clau es retorna (s'allibera el mútex), el recurs queda disponible per a la resta de clients.

D'aquesta manera, el mútex actua alhora com a bloquejador d'un recurs i com a indicador que està ocupat.

Com caldria procedir en el cas de disposar de diversos recursos similars, però no equivalents (com és el cas de disposar d'una lavabo per a homes i un altre per a dones)? Per a gestionar correctament aquests dos recursos, caldria establir dos mútexs, un de diferent per a cadascun dels recursos, de manera que cada tipus de tasca sabés quan pot accedir al que li correspon.

Exemple: control d'accés a *buffers* mitjançant mútex

El codi següent mostra com es pot protegir el comptador de posicions de *buffer* ocupades (*count*) davant la corrupció de dades mitjançant un mútex.

```
int N = bufferSize;
dataArray buffer[N];
int count = 0;
mutex mutex_count(count);             // associem un mútex a count

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA() {
      int i;
      while( 1 ) {
            produce(&data);
            while( count == N );      //espera si buffer és ple
            buffer[i] = data;
            i = (i + 1) % N;
            mutex_count.lock();      // count bloquejada
            count = count + 1;
            mutex_count.unlock();      // count alliberada
      }
}

void TaskB() {
      int i;
      while( 1 ) {
            while( count == 0 );      //espera si buffer és buit
            data = buffer[i];
            i = (i + 1) % N;
            mutex_count.lock();      // count bloquejada
            count = count - 1;
            mutex_count.unlock();      // count alliberada
            consume(&data);
      }
}
```

Cal recordar que, tal com es descriu en apartats anteriors, l'ús de mútexs pot comportar situacions d'inversions de prioritat.

Semàfors

Els semàfors són estructures de dades que tenen un comportament similar a un comptador i que s'utilitzen per a la sincronització entre tasques. La seva finalitat és intercanviar indicacions de sincronia entre tasques.

Atès que els semàfors poden tenir més de dos estats, són adequats per a intercanviar diversos tipus de missatge entre tasques (seguint l'exemple, indicar quantes vegades s'ha de prémer el botó).

Per tal de garantir que funcionin correctament, s'han d'organitzar de manera que cada tasca interactuï amb un mateix semàfor d'una **única** manera: o bé incrementant-lo o bé llegint-lo.

Sincronització entre tasques mitjançant un semàfor

Imaginem dues tasques, la primera responsable de detectar la pulsació del botó d'engegada d'un sistema mitjançant una interrupció (diguem-li `power_button_pushed`) i la segona encarregada d'activar un perifèric com a conseqüència d'aquesta acció. Utilitzant semàfors aquesta funcionalitat es podria implementar fent que:

- la primera tasca incrementi el valor del semàfor en el moment en què es produeixi la pulsació/interrupció, o que
- la segona estigui a l'espera de detectar aquest increment per a iniciar la seva activitat.

Aquesta funcionalitat es podria implementar de la manera següent:

```
interrupt irq(power_button_pushed);      // creem una interrupció
semaphore power_button;                  // creem un semaphore

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA() {
      while(1) {
            while(irq == false);      // esperem que premi el botó
            power_button.increment();
      }
}

void TaskB() {
      int i;
      while( 1 ) {
            if (i < power_button.value()) {
                  /* actions */
            }
            i = power_button.value();
            delay(10);

      }
}
```

La TaskA resta a l'espera de rebre la interrupció. Cada vegada que la rep, incrementa el valor del semàfor.

La TaskB manté un registre del valor del semàfor. Si s'ha produir un increment, executa /* actions */. Si no s'ha incrementat durant la darrera iteració, no es fa res. En aquest tipus d'implementació, en què els terminis d'execució no s'han especificat prèviament, es pot afegir un petit retard per a donar temps que el semàfor incrementi el seu valor de manera segura.

Observem que totes dues tasques es basen en un bucle bàsic de control que itera contínuament. Per tant, ambdues tasques es mantenen actives mentre el sistema estigui engegat.

Actualment, hi ha eines avançades que permeten comunicar i sincronitzar tasques al mateix temps i que eviten bona part dels problemes de les eines descrites fins al moment. Tot seguit les descriurem.

Bústies de correu

Les bústies de correu permeten enviar missatges entre tasques que continguin tant dades com informació de sincronització.

Es comporten com el correu habitual:

- El programari gestor d'aquest correu (per exemple, l'RTOS) s'encarrega d'emmagatzemar els missatges en bústies fins que són llegits per la tasca corresponent.

- És possible enviar missatges a múltiples tasques i rebre'n també de diverses, però no és possible modificar-ne el contingut una vegada enviats.

- També sol ser possible establir prioritats en els missatges.

Per tot això, les bústies poden emular les funcionalitats dels *buffers*, cues, semàfors i mútexs i prevenir qualsevol risc intrínsec, ja que en cap moment no utilitzen espais de memòria compartits.

Comunicació entre tasques mitjançant bústies de correu

A continuació, es mostra una implementació alternativa del segon exemple del subapartat 3.4.5, "Condició de carrera evitada amb un mútex", basada en bústies de correu.

```
int N = bufferSize;
dataArray buffer[N];
int count = 0;
mutex mutex_count(count);             // associem un mútex a count

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA() {
      while( 1 ) {
            produce(&data);
            send(TaskB, &data);      // s'envien dades
            /* actions 1 */
            receive(TaskB, &data); // s'espera a rebre dades
            consume(&data);
      }
}

void TaskB() {
      while( 1 ) {
            receive(TaskA, &data);      // s'espera a rebre dades
            transform(&data)
            send(Task A, &data);      // s'envien dades
            /* actions 2 */
      }
}
```

El subapartat 3.4.5, "Condició de carrera evitada amb un mútex", s'ha ampliat fent que TaskB retorni les dades, una vegada transformades a TaskA, per a la seva reutilització. La instrucció send() envia un missatge amb el punter a data a la tasca destinatària indicada. La instrucció receive() espera rebre un missatge provinent de la tasca indicada.

Fixem-nos que la implementació, tot i tenir una funcionalitat més gran, és més simple que les anteriors i, sobretot, més segura.

3.3.7. Gestió de recursos

Memòria

En entorns multitasca, bona part dels recursos del sistema, i no sols el temps de processador, són compartits per les diferents activitats que cal dur a terme. La memòria és un altre recurs intrínsecament compartit. A banda de gestionar els problemes derivats de la corrupció de dades vistos anteriorment, sigui quina sigui l'arquitectura del nostre sistema, cal establir mecanismes que permetin repartir la memòria disponible entre les diferents tasques.

La solució més freqüent consisteix a fer que les tasques sol·licitin memòria al programari encarregat de la seva gestió (per exemple, l'RTOS) cada vegada que la necessiten. Tanmateix, aquest programari s'encarrega d'alliberar-la quan les tasques finalitzen.

Gràcies a aquest procés continu de reserva i alliberament de petites porcions de memòria (**assignació dinàmica**), el sistema pot funcionar, executant múltiples tasques, amb unes necessitats de memòria moderades.

Per simplicitat, se sol optar per adreçar la memòria disponible per blocs. Un bloc de memòria és una porció de memòria a la qual s'accedeix amb una adreça lògica única. D'aquesta manera, forma el mínim conjunt de posicions de memòria contigües accessibles i assignables a una única tasca.

Si els blocs són massa grans per a les necessitats habituals de les tasques del sistema, es produirà un malbaratament important d'espai de memòria.

Exemple

Cada tasca bloqueja un bloc, independentment de l'ús que en faci.

Al mateix temps, hi haurà pocs blocs disponibles per assignar i, per tant, pot limitar innecessàriament el nombre de tasques actives simultàniament.

Per contra, si els blocs són massa petits, tasques que requereixin més d'un bloc, típicament necessitaran que siguin contigus i, pel mateix procés d'assignació/alliberament, pot ser que això ja no sigui possible al cap d'un cert temps de funcionament. Aquest fenomen es coneix com a **fragmentació de la memòria** i pot comportar un alentiment innecessari del sistema.

Vegeu també

Vegeu el subapartat 3.3.2, en la tercera figura: "Diagrama d'estats d'una possible implementació de l'ascensor de l'exemple que acabem d'analitzar".

La conclusió de tot plegat és que per a aconseguir un màxim aprofitament de la memòria i un màxim rendiment cal dimensionar correctament els blocs en funció de les necessitats del nostre sistema.

Interrupcions

És obvi que un sistema en temps real necessita gestionar interrupcions. Com s'ha descrit anteriorment, l'execució de les tasques es veu aturada en el moment en què es rep una interrupció.

En aquest instant, se cedeix el control a la ISR que n'identifica el tipus i actua en conseqüència i afegeix la tasca d'atenció corresponent a la llista d'execució.

A partir d'aquest moment, ja és responsabilitat del planificador de tasques decidir si aquesta nova tasca té prou prioritat per a alterar els seus plans o bé, simplement, serà atesa en un moment posterior i continuar amb la tasca que s'havia interromput inicialment.

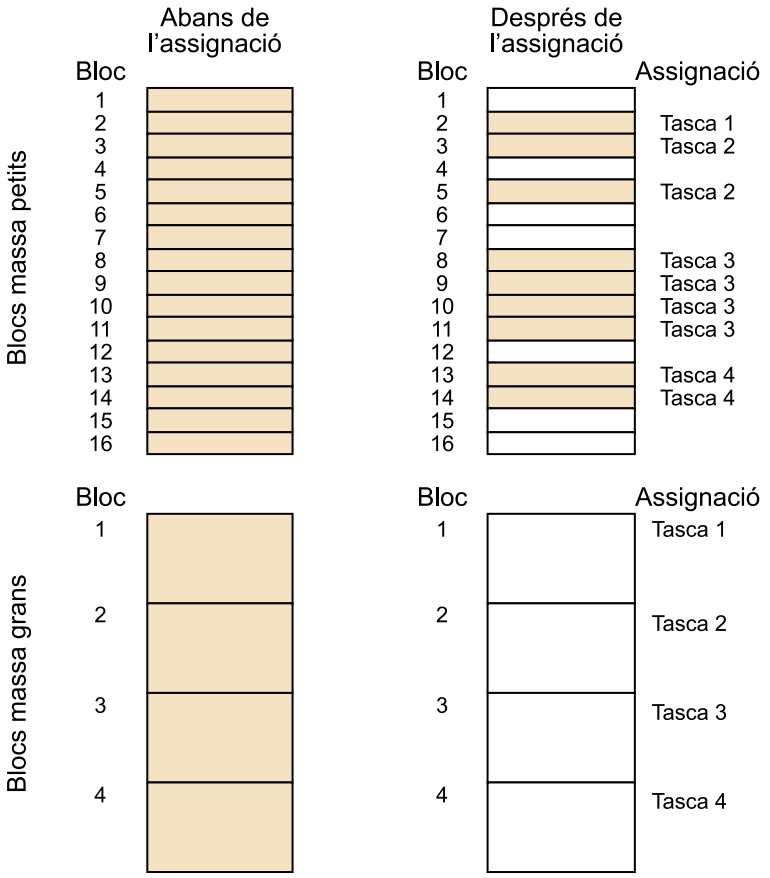
Per a fer el seguiment d'aquest procés se solen implementar, com a mínim, els serveis d'atenció a interrupcions següents:

- **Entrada d'interrupció**: notifica al sistema l'arribada d'una interrupció i emmagatzema el context de la tasca que queda interrompuda de manera que sigui possible que continuï.

- **Atenció d'interrupció**: consulta el vector d'interrupcions i identifica la tasca d'atenció a interrupció que cal afegir a la llista de tasques.

- **Sortida d'interrupció**: notifica al sistema que s'ha completat l'atenció a la interrupció i li indica que pot continuar l'execució, i fa que el planificador reavaluï la nova llista de tasques.

Esquema dels problemes habituals en l'assignació de blocs de memòria



En el cas d'emprar blocs massa petits, es produeix una fragmentació. Si els blocs són massa petits, es malbarata molta memòria.

Resum

En aquest mòdul didàctic hem repassat conceptes bàsics relatius al programari i la seva arquitectura, i hem vist que la seva aplicació en els programes que governen els sistemes encastats poden ajudar a treballar d'una manera més optimitzada en condicions de memòria limitada, potència de càlcul moderada o requisits per a treballar en temps real. Ho hem fet analitzant els aspectes següents:

a) En primer lloc, hem repassat les característiques pròpies dels sistemes encastats parant atenció a:

- L'**elecció del llenguatge** de programació per a desenvolupar el seu programari de control.
- **Identificar les eines** existents per a desenvolupar aquest programari.

b) En segon lloc, hem analitzat conceptes bàsics de programació que ajuden a agilitzar el codi de control dels sistemes encastats. Els principals conceptes que hem presentat són els següents:

- Les **interrupcions**, com un mecanisme bàsic de comunicar-se amb els perifèrics i altres elements del sistema en temps real.
- Els **punters**, com a variables que ens permeten manipular la memòria d'una manera més eficient.
- **Funcions *inline*** i **funcions externes**, com a exemples de tècniques de programació que permeten compactar i fer més modular el codi de control.

c) Finalment, hem vist els models de programació existents que hi ha, hem identificat els avantatges i inconvenients de la seva utilització, sempre intentant-los relacionar amb exemples pràctics. Els models de programació analitzats han estat:

- Els **simples**, basats en la utilització d'un bucle de control simple o el control d'esdeveniments.
- Les **màquines d'estat** com a alternativa que permet simplificar el codi quan el potencial dels models simples es torna insuficient.
- Els **sistemes multitasca** com a solució per a intentar apropar-nos el màxim possible a sistemes operatius ideals treballant en temps real. S'han identificat els conceptes necessaris per a entendre'n el funcionament, i també

la metodologia per a poder planificar i executar les tasques d'una manera adequada. S'ha parat atenció a la necessitat de sincronitzar-les i també a gestionar els recursos de la màquina de la millor manera possible.

Al llarg de tot el mòdul s'han intercalat problemes i exercicis pràctics amb la voluntat d'aclarir el màxim possible els conceptes aquí presentats. Finalment, hem cregut necessari incloure exercicis d'autoavaluació corregits al final del mòdul, ja que la programació és una disciplina que només s'arriba a controlar mitjançant la pràctica continuada.

Activitats

1. Feu un programa en llenguatge C que calculi els valors futurs de dipòsits d'estalvis mensuals suposant que apliquem el càlcul d'interès compost, i que aquest pot variar del 10 al 20% anual. Utilitza el formalisme de programació consistent a passar una funció a una altra mitjançant punters.

El programa hauria de considerar els tres casos que hi ha depenent de la freqüència amb què aquest interès se suma al dipòsit:

a) Per períodes superiors al mes.

b) Diàriament.

c) De manera contínua.

En cada cas, la quantitat futura F es relaciona amb les imposicions mensuals A, la taxa d'interès anual i, el nombre d'anys n i el nombre de composicions m (m = 1 per a períodes anuals, m = 2 per a semestrals, m = 4 per a trimestrals, m = 12 per a mensuals i m = 360 per a diaris) mitjançant les fórmules següents:

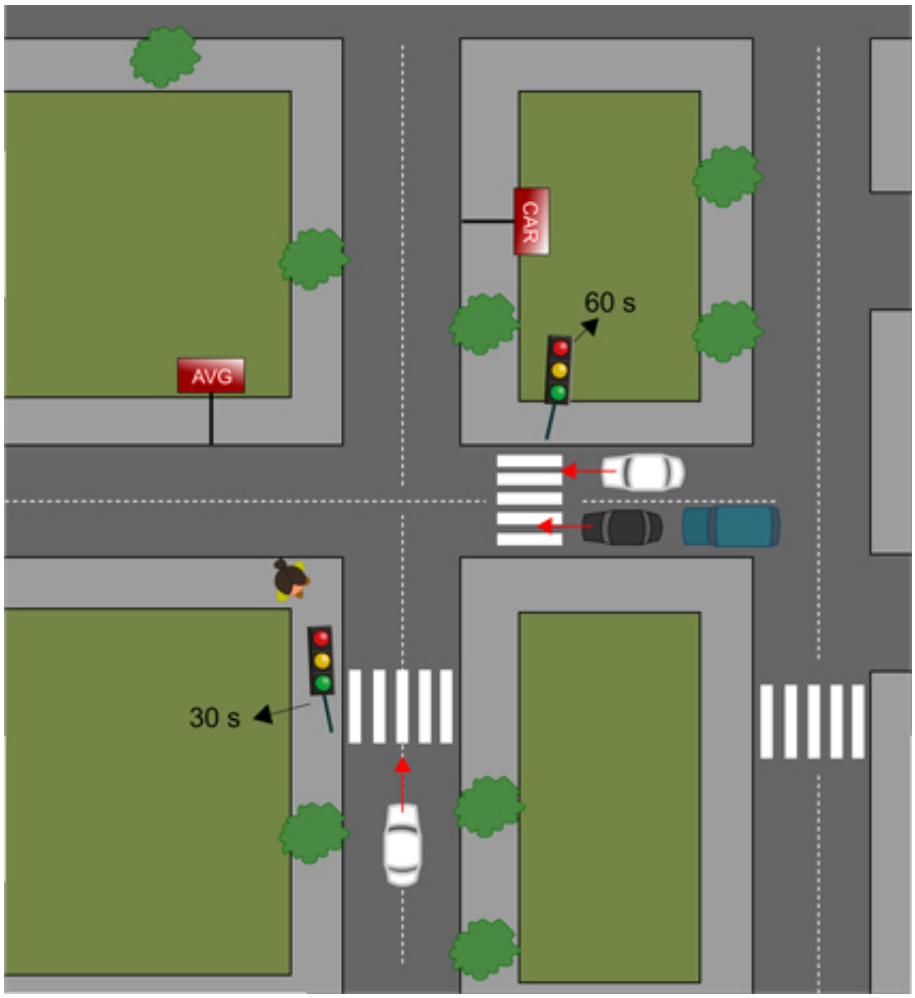
a) $F = 12A \cdot \left[\frac{(1+i/m)^{mn}-1}{i}\right]$

b) $F = A \cdot \left[\frac{(1+i/m)^{mn}-1}{(1+i/m)^{m/12}-1}\right]$

c) $F = A \cdot \left[\frac{e^{n}-1}{e^{i/12}-1}\right]$

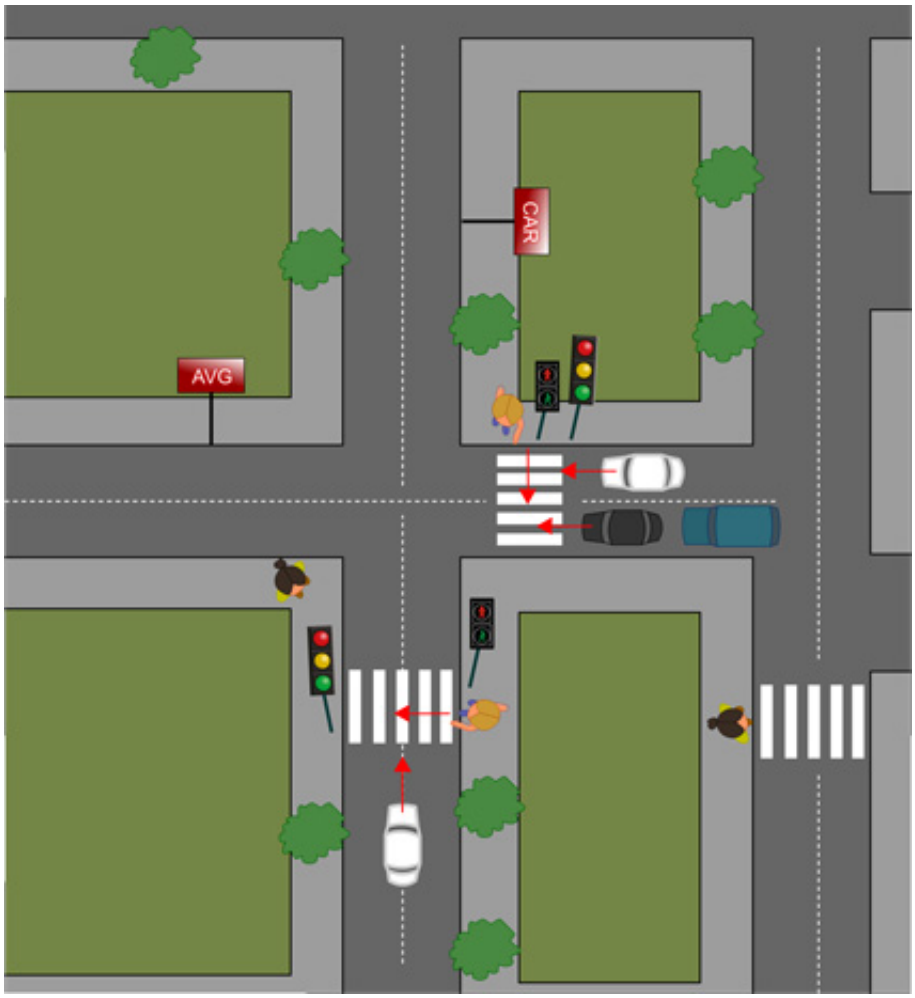
2. Dissenyeu una màquina d'estats que permeti implementar la descripció funcional següent:

"S'ha de regular la circulació de vehicles en l'encreuament entre una avinguda principal (AVG) i un carrer secundari (CAR) mitjançant semàfors de tres posicions (verd, taronja i vermell). Per a garantir la fluïdesa del trànsit, s'ha de permetre el pas de vehicles per AVG durant 60 segons i per CAR durant 30 segons, en cada torn. Per seguretat, el senyal taronja (que indica canvi a vermell imminent) s'ha de mantenir durant 2 segons i s'han d'esperar 3 segons entre que es talla la circulació per un dels carrers (llum vermell) i se'n permet el pas per l'altre (llum verd)."



Identifiqueu els estats i variables necessaris, i també les seves transicions i tasques que cal fer. Elaboreu el diagrama d'estats corresponent i proposeu una implementació en codi font.

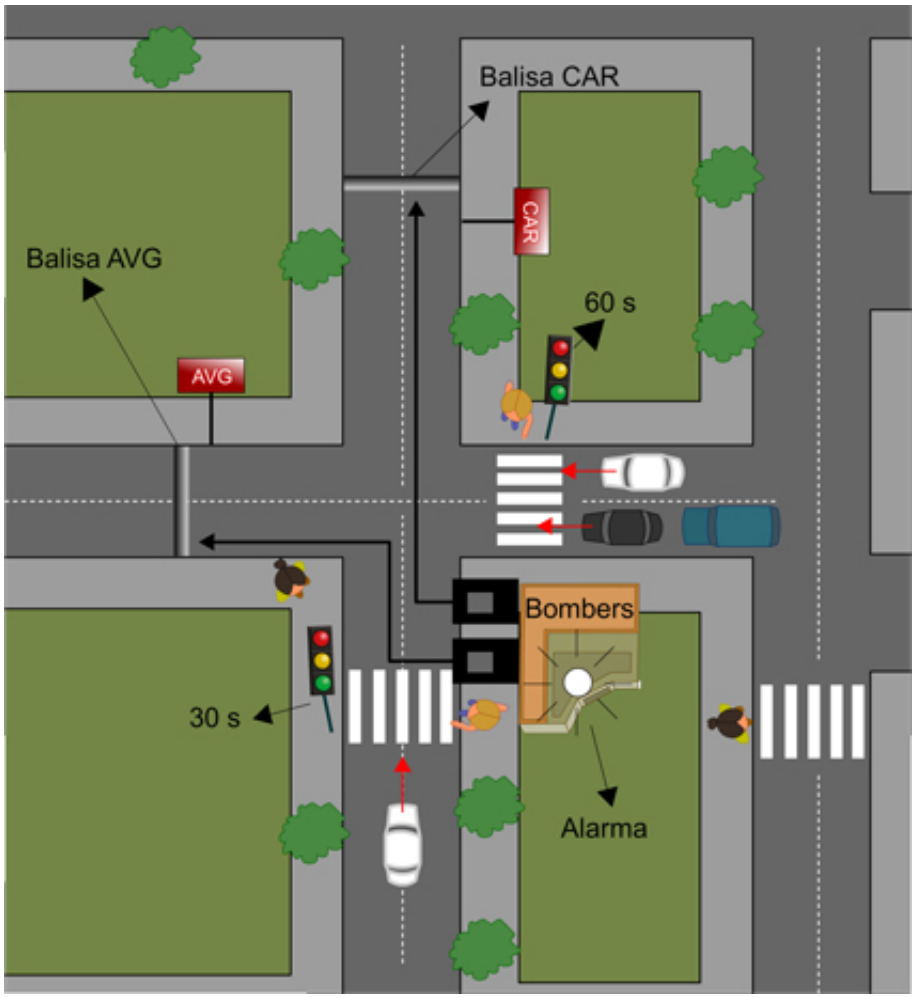
3. Modifiqueu el disseny de l'activitat 2 perquè implementi també un semàfor que reguli el pas de vianants per l'avinguda AVG. Suposeu que el semàfor de vianants disposa de 2 llums i que la transició de verd a vermell cal indicar-la fent pampalluguejar el llum verd durant 2 segons. E1ereu únicament els canvis que caldria fer en el disseny anterior.



4. S'ha instal·lat una estació de bombers al carrer CAR tal com indica la figura. Modifiqueu el disseny de l'activitat 2 perquè implementi un sistema de sortida d'emergència dels vehicles de bombers segons la descripció següent:

"En sonar l'alarma a l'estació de bombers, cal tallar la circulació a AVG i a CAR per a permetre el pas de vehicles de bombers. Els semàfors retornen al funcionament normal només després que el vehicle de bombers hagi superat les balises de seguretat situades a tots dos carrers. Cal aturar el trànsit de tots dos carrers amb seguretat."

Implementeu el sistema amb màquines d'estat jeràrquiques.



5. Dissenyeu una màquina d'estats que permeti implementar la descripció funcional següent.

"S'ha de controlar el funcionament d'un contestador automàtic. Cal reproduir un missatge de benvinguda de 7 segons, seguit d'un xiulet d'1 segon després de rebre tres tons consecutius. A continuació, cal activar la gravació del missatge durant no més de 60 segons. En completar-se la gravació, cal incrementar la pantalla que compta els missatges enregistrats una unitat. La gravació s'atura en el moment en què l'interlocutor penja. També cal preveure un mode que reprodueixi els missatge, quan n'hi hagi de gravats i que només s'activi en polsar un botó de l'equip. En aquest mode, el contestador no pot atendre trucades."

Empreu màquines d'estats jeràrquiques. Identifiqueu els estats i variables necessaris, i també les seves transicions i tasques que cal dur a terme. Elaboreu el diagrama d'estats corresponent i proposeu-ne una implementació en codi font.

6. Responeu a les qüestions següents sobre conceptes fonamentals del mòdul:

a) Com es defineix una interrupció?

b) Què és una ISR?

c) Quines són les accions i els esdeveniments que defineixen un procés d'interrupció?

d) Quins són els principals avantatges de fer servir interrupcions en el funcionament dels sistemes encastats?

e) Què és un vector d'interrupció?

f) E1ereu els cinc passos que cal seguir en el disseny d'una màquina d'estats.

g) E1ereu fins a deu possibles problemes causats per la planificació prioritària.

Exercicis d'autoavaluació

Indiqueu si les interrupcions següents són vertaderes o falses, i justifiqueu la vostra resposta:

1. Els vectors d'interrupció no contenen codi executable.

2. Les interrupcions s'han de programar en un llenguatge de baix nivell.

3. Hi ha tres mètodes per a generar vectors d'interrupció.

4. La prioritat de les interrupcions sempre pot ser definida pel programador.

5. En un sistema encastat en què la velocitat d'execució no és prioritària, la utilització de la tècnica d'*inlining* pot ajudar a optimitzar-ne el funcionament.

6. Un programador experimentat sempre inclou funcions *inline* en el seu codi.

7. La utilització de funcions externes permet reduir el cost econòmic de desenvolupament del programari de sistemes encastats.

8. Les màquines d'estat són adequades per a implementar sistemes molt complexos amb necessitats multitasca.

9. La planificació multitasca és sempre millor que la seqüencial.

Solucionari

Activitats

1. La solució aquí proposada en cap cas no és única i, per tant, podeu trobar aproximacions al problema presentat per l'enunciat igualment vàlides. Farem servir una funció que anomenarem *taula*, a la qual, i en funció de les dades que l'usuari introdueixi per teclat, passarem un punter a una altra funció que es basarà en la fórmula adequada d'interès compost per a fer el càlcul sol·licitat. Es recomana que executeu aquest codi per comprovar-ne el resultat final.

```
/* Programa de càlcul de dipòsits mitjançant interès compost */
#include <studio.h>
#include <studio.h>
#include <studio.h>
#include <studio.h>

/* Prototips de funcions que farem servir en el programa */
void taula (double (*pf) (double i, int m, double n), double a,
int m,  double n);
double md1 (double i, int m, double n);
double md2 (double i, int m, double n);
double md3 (double i, int m, double n);

/* Funció principal del programa */
main() {
      int m;       /* # períodes de composició per any */
      double n;       /* # anys */
      double a;       /* Quantitat imposició mensual */
      char freq;       /* Indicador freqüència de la composició */
      /* Entrada de dades */
      printf("\n VALOR FUTUR D'UNA SÈRIE DE DIPÒSITS MENSUALS\n\n");
      printf("\n Quantitat dels pagaments mensuals: ");
      scanf("%lf", &a);
      printf("Nombre d'anys: ");
      scanf("%lf", &n);
      /* Introducció de la freqüència de la composició */
      do {
            printf("Freq. Composició (A,S,Q,M,D,C: ");
            scanf("%lf", &freq);
            freq = toupper(freq); /* convertim a majúscules */
            if (freq == 'A') {
                  m = 1;
                  printf("\n Composició anual\n");
            }
            else if (freq == 'S') {
                  m = 2;
                  printf("\n Composició semestral\n");
            }
            else if (freq == 'Q') {
                  m = 4;
                  printf("\n Composició trimestral\n");
            }
            else if (freq == 'M') {
                  m = 12;
                  printf("\n Composició mensual\n");
            }
            else if (freq == 'D') {
                  m = 360;
                  printf("\n Composició diària\n");
            }
            else if (freq == 'C') {
                  m = 0;
                  printf("\n Composició contínua\n");
            }
            else {
                  printf("\n ERROR\n\n");
            }
} while(freq != 'A' && freq != 'S' && freq != 'Q' && freq != 'M'
&& freq != 'D' && freq != 'C');
/* Fem els càlculs */
```

```
if (freq == 'C')
                  taula(md3, a, m, n); /* composició contínua */
            else if (freq == 'D')
                  taula(md2, a, m, n); /* composició diària */
            else
                  taula(md1, a, m, n); /* altres casos */
}

/* generador de "taula". Aquesta funció accepta un punter a una
altra funció tal com demana l'enunciat del problema */
void taula(double(*pf)(double i, int m, double n), double a,
int m, double n) {
      int cont; /* Comptador del bucle */
      double i; /* taxa d'interès */
      double f; /* valor futur */
      printf("\n Interès      Quantitat futura\n\n");
      for(cont = 1; cont <=20; ++cont){
            i = 0.01 * cont
           f = a * (*pf)(i,m,n); /* LA FUNCIÓ PASSA COM UN PUNTER */
            printf("      %2d           %.2f\n", cont, f);
      }
      return;
}

/* Dipòsits mensuals, composició periòdica */
double md1(double i, int m, double n){
      double factor, rao;
      factor = 1 + i/m;
      rao = 12 * (pow(factor, m*n)-1)/i;
      return(rao);
}

/* Dipòsits mensuals, composició diària */
double md2(double i, int m, double n){
      double factor, rao;
      factor = 1 + i/m;
      rao = (pow(factor, m*n)-1)/(pow(factor, m/12)- 1);
      return(rao);
}

/* Dipòsits mensuals, composició contínua */
double md3(double i, int nulo, double n){
      double rao;
      rao = (exp(i*n)-1)/(exp(i/12)-1);
      return(rao);
}
```

2. Presentem una possible realització. Seguirem l'esquema de solució proposat en l'apartat 3.3, "Màquines d'estat".

1) Necessitarem cinc estats, corresponents a les diferents situacions d'encesa i apagada dels llums del semàfor de cada carrer:

- *AVGVerd:* AVG obert, CAR tallat.
- *AVGTaronja:* AVG a punt de ser tallat, CAR tallat.
- *Tall:* AVG i CAR tallats.
- *CARVerd:* CAR obert i AVG tallat.
- *CARTaronja:* CAR a punt de ser tallat, AVG tallat.

2) Necessitarem una única variable:

- *Timer* = valor de temporitzador que reiniciarem al final de cada estat.
- *Prev* = variable booleana que emmagatzema a quin carrer s'havia permès el pas anteriorment (1 = AVG; 0 = CAR).

3) Taula de transicions:

Transició	Condició
AVGVerd → *AVGTaronja*	*timer* > 60 s

Transició	Condició
AVGTaronja → Tall	*timer* > 2 s
Tall → CARVerd	*timer* > 3 s .AND. prev == 0
Tall → AVGVerd	*timer* > 3 s .AND. prev == 1
CARVerd → CARTaronja	*timer* > 30 s
CARTaronja → Tall	*timer* > 2 s

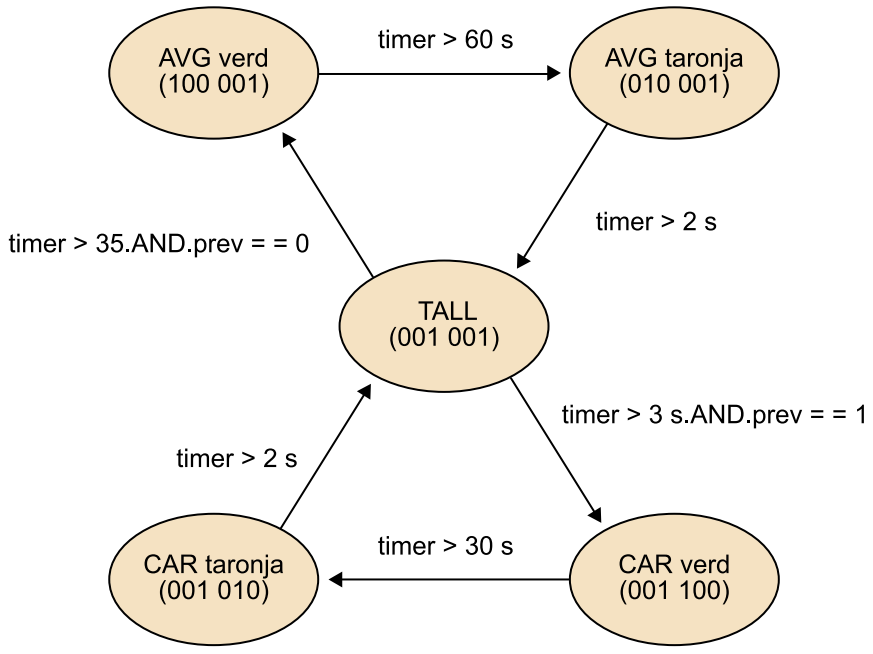
4) La tasca que cal dur a terme en cadascun dels estats es redueix a encendre o apagar els diferents llums del semàfor; per tant:

- *AVGg* = llum verd del semàfor d'AVG.
- *AVGo* = llum taronja del semàfor d'AVG.
- *AVGr* = llum vermell del semàfor d'AVG.
- *CARg* = llum verd del semàfor de CAR.
- *CARo* = llum taronja del semàfor de CAR.
- *CARr* = llum vermell del semàfor de CAR.

Per tant, en cadascun dels estats cal establir les combinacions següents:

Estat	AVGg	AVGo	AVGr	CARg	CARo	CARr
AVGVerd	1	0	0	0	0	1
AVGTaronja	0	1	0	0	0	1
Tall	0	0	1	0	0	1
CARVerd	0	0	1	1	0	0
CARTaronja	0	0	1	0	1	0

Amb tot això, el diagrama d'estats resultant és:



Per als codis de cada estat, fem servir AVGg, AVG o AVGr, o bé CARg, CAR o CARr

I el codi font:

```
#define TALL 0
#define AVGVerd 1
#define AVGTaronja 2
#define CARVerd 3
#define CARTaronja 4


void UnitControl() {
      int state = TALL;
      int timer = 0;
      boolean prev = 0;
      while (1) {
      switch (state) {
      TALL: AVGg=0;AVGo=0;AVGr=1; CARg=0;CARo=0;CARr=1;
            if  (timer>3) {
                        if (prev==0) {
                              state = AVGVerd;
                              timereset();
                        } else if (prev==1) {
                              state = CARVerd;
                              timereset();
                        }
                  }
            break;

      AVGVerd: AVGg=1;AVGo=0;AVGr=0; CARg=0;CARo=0;CARr=1;
            prev = 1;
                  if  (timer>60) {
                        state = AVGTaronja;
                        timereset();
                  }
            break;

      AVGTaronja:AVGg=0;AVGo=1;AVGr=0;CARg=0;CARo=0;CARr=1;
            if  (timer>2) {
                        state = TALL;
                        timereset();
                  }
            break;

      CARVerd: AVGg=0;AVGo=0;AVGr=1; CARg=1;CARo=0;CARr=0;
            prev = 0;
                  if  (timer>30) {
                  state = CARTaronja;
                  timereset();
                  }
            break;

      CARTaronja:AVGg=0;AVGo=0;AVGr=1;CARg=0;CARo=1;CARr=0;
            if  (timer>2) {
                        state = TALL;
                        timereset();//reiniciem comptador temps
                  }
            break;
            }
            timer = currenttime(); //carreguem temps actual
        }
}
```

3. Observem que se'ns demana que afegim un seguit d'accions de control del semàfor de vianants que es duen a terme de manera totalment síncrona (per exemple, simultània) amb els canvis d'estat dels semàfors reguladors de vehicles. Per aquest motiu, no cal afegir cap més estat a la màquina; simplement, cal preveure algunes accions més:

- *AVGvianG* = llum verd del semàfor de vianants.
- *AVGvianInt* = llum verd intermitent del semàfor de vianants.
- *AVGvianR* = llum vermell del semàfor de vianants.

Amb tot això, la taula de tasques s'ha de modificar de la manera següent:

Estat	AVGg	AVGo	AVGr	CARg	CARo	CARr	AVGvi-anG	AVGvi-anInt	AVGvianR
AVGVerd	1	0	0	0	0	1	**0**	**0**	**1**
AVGTa-ronja	0	1	0	0	0	1	**0**	**0**	**1**
Tall	0	0	1	0	0	1	**0**	**0**	**1**
CARVerd	0	0	1	1	0	0	**1**	**0**	**0**
CARTaron-ja	0	0	1	0	1	0	**0**	**1**	**0**

4. Presentem una possible realització. Partirem del disseny de la màquina d'estats de l'activitat 2 (mode NORMAL), a la qual caldrà afegir un nou mode (ALARMA) de funcionament que s'inicia en activar-se l'alarma de l'estació de bombers. Novament, seguirem l'esquema de solució proposat en el subapartat 3.3, màquines d'estat.

1) Necessitarem dos estats nous:

- *BOMTaronja*: aturada dels vehicles que circulen, tant per AVG com per CAR de manera segura; és a dir, cal posar tots els semàfors en taronja durant 2 segons.
- *BOMTall*: AVG i CAR tallats, de manera que permeten el pas dels bombers.

2) Necessitarem tres variables més, que es modificaran mitjançant les rutines ISR corresponents a la interrupció provinent de cadascun dels perifèrics.

- *balisaAVG* = variable booleana que recull el pas del vehicle de bombers per la balisa d'AVG.
- *balisaCAR* = variable booleana que recull el pas del vehicle de bombers per la balisa de CAR.

3) Taula de transicions:

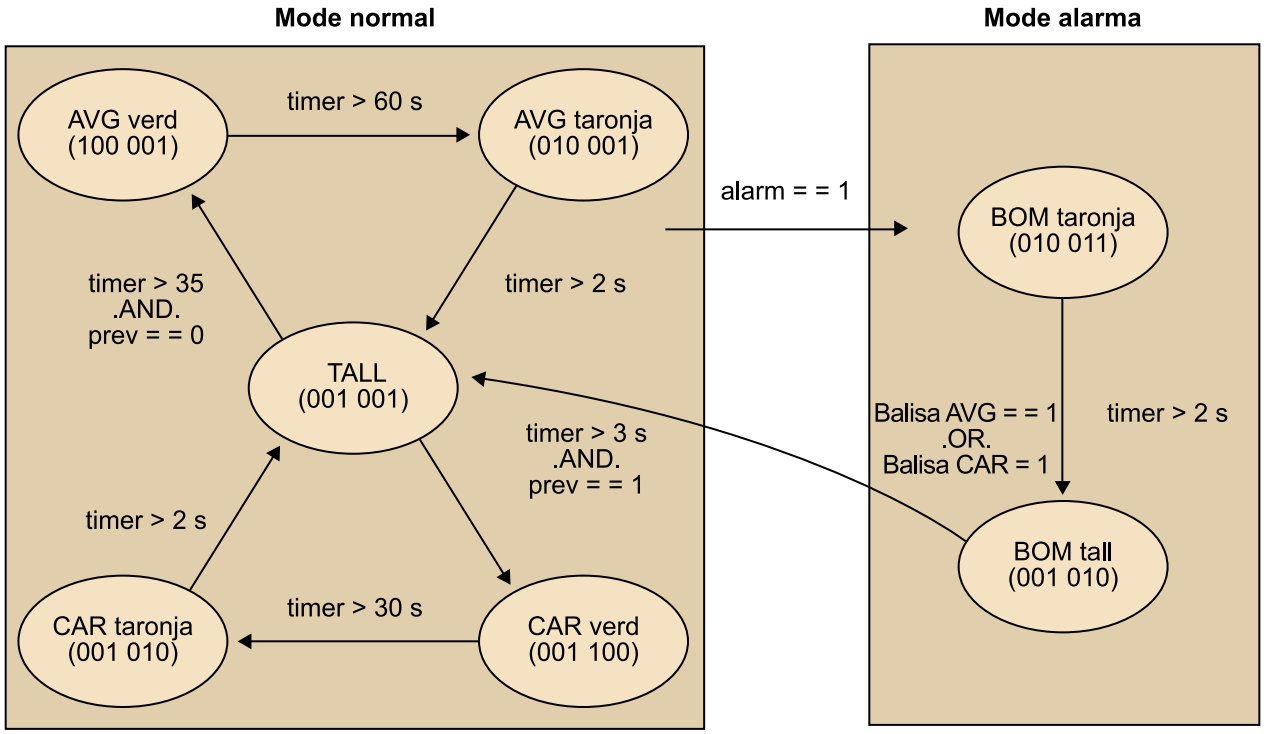
Transició	Condició
NORMAL → ALARMA	alarm == 1
BOMTaronja → BOMTall	*timer* > 2 s
BOMTall → NORMAL(AVGVerd)	balisaAVG == 1 .OR. balisaCAR ==1

4) En la taula de tasques, només cal afegir les accions corresponents als estats nous.

Mode	Estat	AVGg	AVGo	AVGr	CARg	CARo	CARr
NORMAL	*AVGVerd*	1	0	0	0	0	1
	AVGTaronja	0	1	0	0	0	1
	Tall	0	0	1	0	0	1
	CARVerd	0	0	1	1	0	0
	CARTaronja	0	0	1	0	1	0
ALARMA	*BOMTaronja*	0	1	0	0	1	0

Mode	Estat	AVGg	AVGo	AVGr	CARg	CARo	CARr
	BOMTall	0	0	1	0	0	1

Amb tot això, el diagrama d'estats resultant és:



Per als codis de cada estat, fem servir AVGg, AVG o AVGr, o bé CARg, CAR o CARr

I el codi font:

```
#define NORMAL 0
#define ALARM 1

#define TALL 0
#define AVGVerd 1
#define AVGTaronja 2
#define CARVerd 3
#define CARTaronja 4

#define BOMTaronja 0
#define BOMTall 1

global boolean alarm = 0;
global boolean balisaAVG = 0;
global boolean balisaCAR = 0;


void UnitControl() {
      int mode = NORMAL;
      int stateNORMAL = TALL;
      int stateALARM = BOMTaronja;

      int timer = 0;
      boolean prev = 0;

      alarm = 0;
      balisaAVG = 0;
      balisaCAR = 0;
```

```
      while (1) {
   switch (mode) {
      NORMAL:
            switch (stateNORMAL) {
      TALL: AVGg=0;AVGo=0;AVGr=1; CARg=0;CARo=0;CARr=1;
            if  (timer>3) {
                        if (prev==0) {
                              stateNORMAL = AVGVerd;
                              timereset();
                        } else if (prev==1) {
                              stateNORMAL = CARVerd;
                              timereset();
                        }
                  }
            break;

      AVGVerd: AVGg=1;AVGo=0;AVGr=0; CARg=0;CARo=0;CARr=1;
            prev = 1;
                  if  (timer>60) {
                        stateNORMAL = AVGTaronja;
                        timereset();
                  }
            break;

      AVGTaronja:AVGg=0;AVGo=1;AVGr=0;CARg=0;CARo=0;CARr=1;
            if  (timer>2) {
                        stateNORMAL = TALL;
                        timereset();
            }
            break;

      CARVerd: AVGg=0;AVGo=0;AVGr=1; CARg=1;CARo=0;CARr=0;
            prev = 0;
                  if  (timer>30) {
                        stateNORMAL = CARTaronja;
                        timereset();
}
            break;

      CARTaronja:AVGg=0;AVGo=0;AVGr=1;CARg=0;CARo=1;CARr=0;
            if  (timer>2) {
                        stateNORMAL = TALL;
                        timereset();//reiniciem comptador temps
                  }
            break;
            }
            break;

      ALARM:
            switch (stateALARM) {
      BOMTaronja:AVGg=0;AVGo=1;AVGr=0;CARg=0;CARo=1;CARr=0;
            if  (timer>2) {
                        stateALARM = BOMTall;
                        timereset();
                  }
            break;

      BOMTall: AVGg=0;AVGo=0;AVGr=1; CARg=0;CARo=0;CARr=1;
            if  (balisaAVG == 1) {
                        mode = NORMAL;
                        stateNORMAL = TALL;
                        timer = 0;
                        prev = 0;
                        alarm = 0;
                        balisaAVG = 0;
                        balisaCAR = 0;
                  } else if {balisaCAR == 1) {
                        mode = NORMAL;
                        stateNORMAL = TALL;
                        timer = 0;
                        prev = 0;
```

```
                        alarm = 0;
                        balisaAVG = 0;
                        balisaCAR = 0;
                  }
            break;
            }
            break;

      timer = currenttime(); //carreguem temps actual
      if (alarm == 1) {
            mode = ALARM
            stateALARM = BOMTaronja
      }
}

void ISRAlarm() {      // ISR d'atenció a l'activació de l'alarma
      alarm = 1;
}

void ISRbalisaAVG() {  // ISR d'atenció a balisa en AVG
      balisaAVG = 1;
}

void ISRbalisaCAR() {  // ISR d'atenció a balisa en CAR
      balisaCAR = 1;
}
```

5. Presentem una possible realització del contestador automàtic. Seguirem l'esquema de solució proposat en l'apartat 4.c, "Màquines d'estat".

1) Necessitarem dos modes: CONTESTADOR, REPRODUCTOR amb els estats següents:

MODE CONTESTADOR:

- *Wait*: espera l'arribada d'un to de trucada.
- *Ring1*: s'ha rebut 1 to de trucada.
- *Ring2*: s'han rebut 2 tons de trucada consecutius.
- *Messg*: s'han rebut 3 tons de trucada consecutius i es reprodueix el missatge de benvinguda.
- *Beep*: es fa sonar un xiulet durant 1 segon.
- *Record*: s'activa la gravació del missatge.
- *IncCount*: s'atura la gravació i s'incrementa el comptador de missatges.

MODE REPRODUCTOR:

- *Check*: es verifica el nombre de missatges gravats.
- *Play*: es reprodueixen tots els missatges gravats.

2) Necessitarem fins a sis variables:

- *timer* = valor de temporitzador que reiniciarem quan és necessari.
- *msgcnt* = comptador de missatges enregistrats.

Les variables següents es modifiquen mitjançant la ISR d'atenció a perifèric corresponent.

- *to* = comptador del nombre de tons que han sonat.
- *penja* = variable booleana que indica si l'interlocutor ha penjat.
- *polsador* = variable booleana que indica si s'ha premut el botó de reproducció de missatges.

3) Taula de transicions:

Transició	Condició
Wait → *Ring1*	to = = 1
Ring1 → *Ring2*	to = = 2
Ring2 → *Messg*	to = = 3
Ring1 → *Wait*	penja = = 1

Transició	Condició
Ring 2 → Wait	penja = = 1
Messg → Wait	penja = = 1
Beep → Wait	penja = = 1
Messg → Beep	*timer* > 7 s
Beep → Record	*timer* > 1 s
Record → IncCount	*timer* > 60 s .OR. penja = = 1
IncCount → Wait	-
CONTESTADOR → REPRODUCTOR (Check)	Polsador = = 1
Check → Play	msgcnt > 0
Play → Play	msgcnt > 0
REPRODUCTOR → CONTESTADOR (Wait)	msgcnt = = 0

4) La tasca que cal dur a terme en cadascun dels estats es redueix a activar o desactivar els components següents:

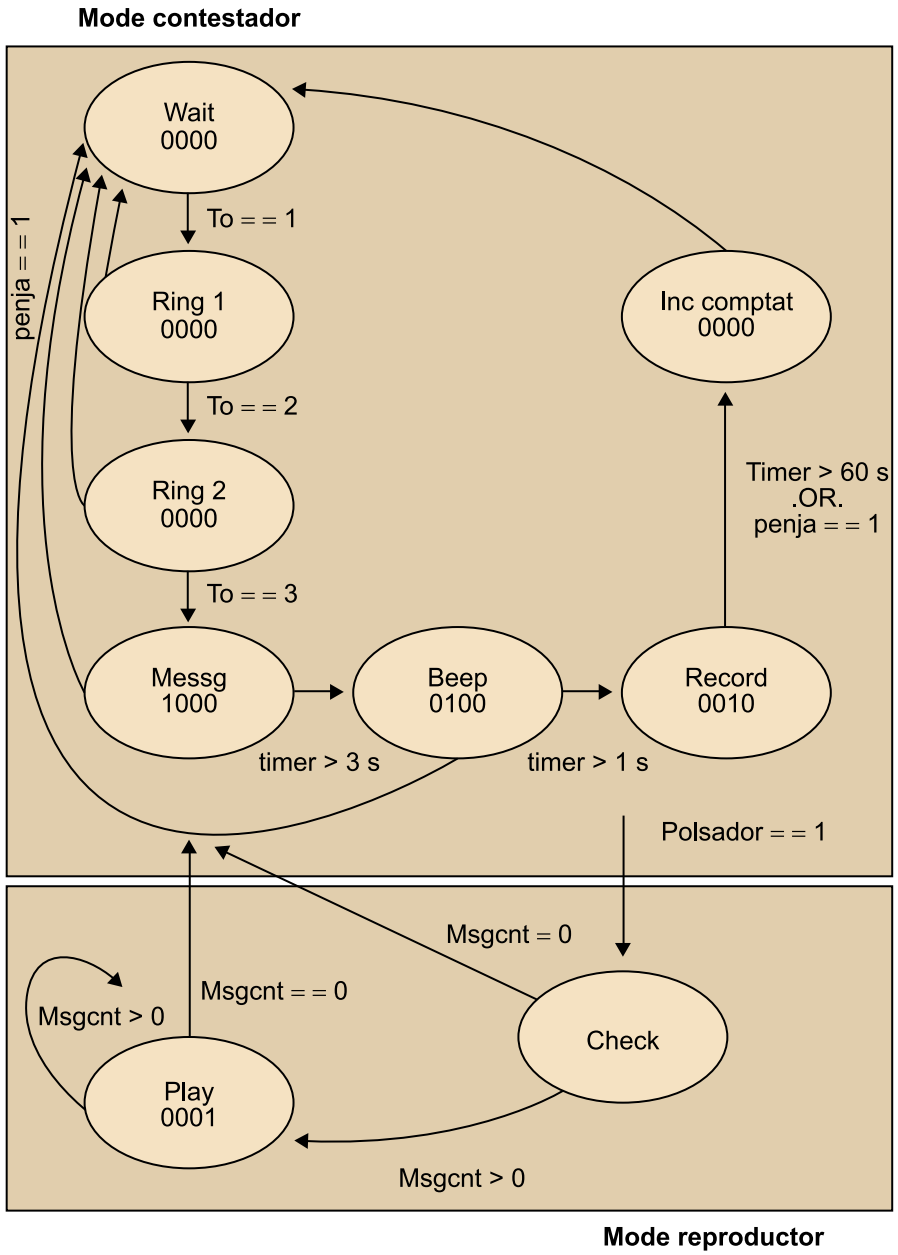
- *m* = reproductor de missatge de benvinguda.
- *b* = generador del xiulet.
- *r* = gravadora de missatges.
- *p* = reproductor de missatge.

Per tant, en cadascun dels estats cal establir les combinacions següents:

Estat	m	b	r	p
Wait	0	0	0	0
Ring1	0	0	0	0
Ring2	0	0	0	0
Messg	1	0	0	0
Beep	0	1	0	0
Record	0	0	1	0
IncCount	0	0	0	0
Check	0	0	0	0
Play	0	0	0	1

Observem que molts dels estats són diferents només lògicament, és a dir, s'hi fan accions diferents dins de la memòria del sistema encastat però no corresponen a cap canvi en l'estat dels perifèrics m, b, r i p.

Amb tot això, el diagrama d'estats resultant és:



I el codi font:

```
#define CONTESTADOR 0
#define REPRODUCTOR 1

#define Wait 0
#define Ring1 1
#define Ring2 2
#define Messg 3
#define Beep 4
#define Record 5
#define IncCount 6

#define Check 0
#define Play 1

global int to = 0;
global boolean penja = 0;
global boolean polsador = 0;


void UnitControl() {
```

```
        int mode = CONTESTADOR;
        int stateCONTESTADOR = Wait;
        int stateREPRODUCTOR = Check;

        int timer = 0;
        int msgcnt = 0;

        to = 0;
        penja = 0;
        polsador = 0;

        while (1) {
     switch (mode) {
        CONTESTADOR:
              switch (stateCONTESTADOR) {
        Wait: m=0; b=0; r=0; p=0;
                    penja == 0
                    if  (to == 1) {
                          stateCONTESTADOR = Ring1;
                    }
                    break;

        Ring1: m=0; b=0; r=0; p=0;
                    if (penja == 1) {
                          stateCONTESTADOR = Wait;
                    else if  (to == 2) {
                          stateCONTESTADOR = Ring1;
                    }
                    break;

              Ring2: m=0; b=0; r=0; p=0;
              if (penja == 1) {
                          stateCONTESTADOR = Wait;
                    else if  (to == 3) {
                          stateCONTESTADOR = Messg;
                          timereset();
                    }
                    break;

              Messg: m=1; b=0; r=0; p=0;
              to = 0;
                    if (penja == 1) {
                          stateCONTESTADOR = Wait;
                    else if  (timer > 7) {
                          stateCONTESTADOR = Beep;
                          timereset();
                    }
                    break;

              Beep: m=0; b=1; r=0; p=0;
              if (penja == 1) {
                          stateCONTESTADOR = Wait;
                    else if  (timer > 1) {
                          stateCONTESTADOR = Record;
                          timereset();
                    }
                    break;

              Record: m=0; b=0; r=1; p=0;
              if (penja == 1) {
                          stateCONTESTADOR = IncCount;
                    else if  (timer > 60) {
                          stateCONTESTADOR = IncCount;
                    }
                    break;

              IncCount: m=0; b=0; r=0; p=0;
                    msgcnt = msgcnt + 1;
                    stateCONTESTADOR = Wait;
                    break;
              }
              break;
```

```
      REPRODUCTOR:
            switch (stateREPRODUCTOR) {
            Check: m=0; b=0; r=0; p=0;
                  polsador = 0;
                  if (msgcnt == 0) {
                        mode == CONTESTADOR;
                        stateCONTESTADOR = Wait;
                        to = 0;
            else if  (msgcnt > 0) {
                  stateREPRODUCTOR = Play;
            }
            break;

            Check: m=0; b=0; r=0; p=1;
            if (msgcnt == 0) {
                        mode == CONTESTADOR;
                        stateCONTESTADOR = Wait;
                        polsador = 0;
                        to = 0;
                  else if  (msgcnt > 0) {
                        playmeassage();   // accions que permeten
                                          //reproduir un missatge
                        msgcnt = msgcnt - 1;
                  }
                  break;
            }
            break;
      }

      timer = currenttime(); //carreguem temps actual
      if (polsador == 1) {
            mode = REPRODUCTOR
            stateREPRODUCTOR = Check
      }

}

void ISRTo() {            // ISR d'atenció a la recepció d'un to
      to = to + 1;
}

void ISRPenja() {         // ISR d'atenció al tancament de línia
      penja = 1;
}

void ISRPolsador() {      // ISR d'atenció a la pulsació
      polsador = 1;
}
```

6. a) Qualsevol senyal que rep un processador per a redirigir el programa en execució en un moment determinat i passar a executar un codi específic.

b) Una ISR o rutina d'interrupció de servei és qualsevol rutina que s'executa com a conseqüència de la generació d'una interrupció i que satisfà tres requisits:

- Gestiona la interrupció que l'ha generada.
- Permet al processador acceptar noves interrupcions durant la seva execució.
- Retorna el sistema a l'estat previ a la generació de la interrupció associada.

c)

- Arxivament del comptador del programa principal en execució a la pila del processador.
- Generació d'un justificant de recepció (*interrupt acknowledge*) pel processador.
- Execució de la ISR associada.
- Recuperació de l'adreça de retorn i altra informació arxivada a la pila a la finalització de la ISR.
- Continuació de l'execució del programa principal.

d)

- Reducció de la utilització total de recursos del microprocessador, ja que desapareix la necessitat de comunicar-se amb els perifèrics de manera recursiva.
- Eliminació dels estats de latència dels perifèrics.
- Treball amb codis de programari més ordenat i compartimentat.

e) Un vector d'interrupció és una adreça de memòria que indica al processador on buscar la ISR associada a una interrupció determinada.

f)

- Fer una llista de tots els estats possibles.
- Declarar totes les variables.
- Per cada estat, fer una llista de les possibles transicions a altres estats, identificant-ne les condicions.
- Per cada estat i/o transició, fer una llista de totes les accions necessàries (tasques, canvis de variables, etc.).
- Assegurar-se que, per cada estat, les condicions de transició són mútuament excloents i completes.

g)

- Pèrdua de rendiment per excés de planificació vinculant.
- Piles múltiples.
- Restitució de context.
- Condicions de carrera.
- Latència d'interrupció.
- Accés múltiple a biblioteques.
- Mútex.
- Inanició.
- Punt mort (*deadlock*).
- Inversió de prioritat.

Exercicis d'autoavaluació

1. Fals. Per regla general, els vectors d'interrupció efectivament no contenen codi executable, però en alguns models antics de processadors n'hi havia alguns que sí.

2. Fals. No hi ha cap requisit que obligui a programar les ISR d'una interrupció en llenguatge de baix nivell. Efectivament, però, aquests tipus de llenguatge permeten controlar millor el seu temps d'execució malgrat els inconvenients a l'hora d'escriure el codi.

3. Vertader. Els vectors d'interrupció es poden generar des d'un controlador extern, un d'intern o des dels mateixos perifèrics.

4. Fals. Això dependrà del processador que es faci servir. A més, la interrupció NMI sempre té prioritat màxima.

5. Vertader. Si la velocitat d'execució no és prioritària i es busca reduir la mida del codi que controla un sistema encastat, l'ús de funcions *inline* pot ser una bona solució.

6. Fals. La utilització d'*inlining* esdevé complexa i fins i tot els programadors experimentals han de recórrer a heurístiques per decidir si fan servir o no aquesta tècnica. A efectes pràctics els compiladors decideixen millor que els programadors quan l'han d'utilitzar.

7. Vertader. La possibilitat de desenvolupar codi en diferents arxius que obre aquesta tècnica permet distribuir la feina a diferents programadors. A més, qualsevol canvi correctiu o d'actualització només afectarà una part del codi, cosa que simplifica aquests passos.

8. Fals. Les màquines d'estat són adequades per a implementar sistemes encarregats de controlar diferents elements extens amb estats de funcionament ben definits. Aquest fet les fa adequades per a dur a terme tasques força específiques en cadascun d'aquests estats. Per tant, no resulten adequades en entorns multitasca en els quals calgui atendre diferents necessitats d'una manera concurrent. Per practicitat (cal definir, entre d'altres, tots els estats de la màquina), no són convenients en sistemes d'extrema complexitat. En aquests casos cal recórrer a RTOS.

9. Fals. Si bé la planificació multitasca permet atendre múltiples necessitats d'una manera concurrent, les solucions implementades per a emular aquest funcionament (per exemple, la planificació prioritària) introdueixen una certa impredictibilitat en el funcionament del sistema. En aquest sentit, els planificadors seqüencials són totalment predictibles.

Glossari

address bus *m* Vegeu **bus d'adreces**.

adreça física *f* Dades que s'envien mitjançant el bus d'adreces per a indicar la posició de memòria a què es vol accedir.
en physical address

arguments per referència *m pl* Manera de treballar en programació que es basa a passar com a dades punters a una funció.

arguments per valor *m pl* Manera de treballar en programació que es basa a passar dades a una funció, que són copiades dins d'aquestes. La seva modificació no afecta el valor de les dades fora de la funció.

bloc de memòria *m* Porció de memòria que es reserva de manera íntegra i s'assigna per a una tasca concreta. Permet l'assignació dinàmica de la quantitat de memòria que necessita cada tasca en cada moment en entorns multitasca.
en memory block

bus d'adreces *m* Conjunt de línies de comunicacions que connecten un processador amb qualsevol element de memòria amb la finalitat de permetre seleccionar alguna de les seves posicions de memòria. Típicament, si el bus és format per n línies, permet localitzar fins a 2^n adreces individuals. Estratègies avançades de gestió de memòria com la paginació permeten estendre la quantitat de posicions que cal controlar amb n línies de bus d'adreces.
en address bus

central processing unit *f* Vegeu **unitat central de processament**.

check *f* Vegeu **suma de comprovació**.

comunicació entre tasques *f* Pas de dades entre les tasques d'un sistema multitasca. La majoria d'RTOS inclouen multitud de recursos (bústies de correu, cues, etc.) per a transferir dades entre tasques amb seguretat.
en intertask communication

condició de carrera *f* Situació en la qual els efectes combinats de dues o més tasques concurrents varien segons l'odre precís en què s'ha executat la seqüència d'instruccions combinades d'una i de l'altra. Poden ser evitades mitjançant els mútex i la identificació de seccions crítiques.
en race condition

context *m* Dades contingudes en els diferents registres i piles del sistema en el moment de la interrupció i que, per tant, cal restituir, per a poder continuar treballant en les mateixes condicions. El context inclou, entre d'altres, la informació relativa a la següent instrucció pendent d'execució (punter d'instruccions) i totes les dades associades a la tasca (punter de pila, espai de memòria assignat, registres de dades, etc.).

CPU *f* Vegeu **unitat central de processament**.

critical section *f* Vegeu **secció crítica**.

deadline *m* Vegeu **termini**.

digital signal processor *m* Vegeu **processador de senyals digitals**.

DSP *m* Vegeu **processador de senyals digitals**.

exclusió mútua *f* Garantia d'accés exclusiu a un recurs compartit. En els sistemes integrats, el recurs compartit és típicament un bloc de memòria, una variable global, un perifèric o un conjunt de registres. L'exclusió mútua s'aconsegueix en general amb l'ús d'un mútex.
en mutual exclusion

firmware *m* Vegeu **microprogramari**.

funció externa *f* Funció definida en un arxiu diferent d'on està definit el codi principal del programa.

funció *inline* *f* sin. ***inlining***

inlining *m* Estratègia de programació destinada a optimitzar el funcionament de programes en què hi ha moltes crides a funcions. Cada vegada que una funció és cridada (funció *inline*), el compilador introdueix el seu codi complet on ha estat cridada en comptes d'anar-la a buscar a la posició de memòria on ha estat definida inicialment.
sin. **funció *inline***

interrupció *f* Senyal elèctric asíncron enviat per un perifèric al processador. Cada vegada que s'envia aquest tipus de senyal (es diu que s'ha produït una interrupció), s'interromp l'execució normal de la tasca que estigui activa en el processador en aquell moment. Quan això succeeix, es desa el context actual (punter d'instrucció, estat dels registres, etc.) i s'executa una rutina de servei d'interrupció. Quan finalitza la rutina de servei d'interrupció, el processador continua amb l'execució normal de la tasca que havia quedat interrompuda.
en interrupt

interrupció de programari *f* Interrupció originada per un element de programari. Habitualment s'utilitzen per a aplicar els punts d'interrupció i punts d'entrada del sistema operatiu. A diferència de les interrupcions convencionals (originades al maquinari), aquestes es produeixen de manera síncrona amb l'execució del programari; és a dir, es produeixen al començament del cicle d'execució d'una instrucció.
en software interrupt

interrupt *f* Vegeu **interrupció**.

interrupt latency *f* Vegeu **latència d'interrupció**.

interrupt service process *m* Vegeu **procés de servei d'interrupció**.

interrupt service routine *f* Vegeu **rutina de servei d'interrupció**.

interrupt type *m* Vegeu **tipus d'interrupció**.

interrupt vector *m* Vegeu **vector d'interrupció**.

interrupt vector table *f* Vegeu **taula de vectors d'interrupció**.

intertask communication *f* Vegeu **comunicació entre tasques**.

intertask synchronization *f* Vegeu **sincronització entre tasques**.

inversió de prioritats *f* Situació de violació de prioritat induïda per l'existència d'un mútex compartit entre tasques de prioritat molt diferent. En aquestes circumstàncies, la tasca d'altra prioritat pot ser avançada per una tercera de prioritat intermèdia si aquesta entra a executar-se en el moment en què el mútex està bloquejat per la de baixa prioritat.
en priority inversion

ISP *m* Vegeu **procés de servei d'interrupció**.

ISR *f* Vegeu **rutina de servei d'interrupció**.

kernel *m* Vegeu **nucli**.

latència d'interrupció *f* Quantitat de temps entre l'enviament d'un senyal d'interrupció i l'inici de la rutina de servei d'interrupció associada. La velocitat del processador i l'algorisme de planificació emprat en el sistema són alguns dels factors que afecten aquest retard.
en interrupt latency

llista d'execució *f* En entorns multitasca, estructura de dades mantinguda pel sistema (per exemple, el nucli d'un RTOS) i que recull totes les tasques llestes per a l'execució juntament amb la seva prioritat. D'acord amb aquesta informació i amb els recursos disponibles en el sistema en cada instant, el planificador gestiona la manera com cal executar-les.
en task list

memory block *m* Vegeu **bloc de memòria**.

microcontrolador *m* Microprocessador altament integrat dissenyat específicament per al seu ús en sistemes encastats. Els microcontroladors solen incloure un processador integrat, memòria (una petita quantitat de memòria RAM, ROM o ambdós), i altres perifèrics en el mateix xip. Els exemples més comuns són PIC de Microchip, el 8051, 80196 d'Intel i una sèrie de Motorola 68HCxx.
en microcontroller

microcontroller *m* Vegeu **microcontrolador**.

microprogramari *m* Conjunt d'instruccions que governen el funcionament dels sistemes encastats. Es tracta de programari elaborat per a la seva execució en un maquinari específic i que sovint es troba emmagatzemant en memòries tipus flaix o ROM.
en firmware

multitasca *f* Execució de rutines de múltiples rutines de programari de manera pseudosimultània o pseudoparal·lela. El planificador del sistema operatiu és el responsable de distribuir el temps de processador entre les diferents rutines/tasques (que en realitat s'executen de manera individual i consecutiva/seqüencial) de tal manera que se'n simula el processament simultani/paral·lel.
en multitasking

multitasking *f* Vegeu **multitasca**.

mútex *m* Escurçament de *mutual exclusion*, es tracta d'un indicador binari que es pot utilitzar per a sincronitzar les activitats de diverses tasques en entorns multitasca. Com a tal, pot protegir seccions crítiques d'interrupcions indesitjades i recursos compartits d'accessos simultanis.

mutual exclusion *f* Vegeu **exclusió mútua**.

nucli *m* Part essencial de qualsevol RTOS, formada pel planificador de tasques i les rutines de canvi de context.
en kernel

operative system *m* Vegeu **sistema operatiu**.
sigla **OS**

OS *m* Vegeu ***operative system***.

perifèric *m* Element de maquinari (excloent-hi el processador) que fa funcions d'entrada i sortida d'informació. Un o diversos perifèrics es poden trobar integrats en un mateix xip (per exemple, en el processador); en aquests casos, s'anomenen *perifèrics integrats en xip*.
en peripheral

peripheral *m* Vegeu **perifèric**.

physical address *f* Vegeu **adreça física**.

pila *f* Conceptualment, es tracta d'una llista oberta que permet l'addició i eliminació d'elements de manera que el darrer a entrar (ser escrit) sempre és el primer accessible per a sortir (ser llegit). En els sistemes informàtics, aquestes estructures de dades permeten emmagatzemar informació que s'ha de recórrer de manera seqüencial, com, per exemple, un seguit d'instruccions en codi màquina. El seu funcionament particular la fa especialment adequada per a fer el seguiment de tasques molt jerarquitzades. Per exemple, permet afegir instruccions d'atenció a interrupció de manera que, una vegada executades, el programa continua exactament en el punt en què havia quedat interromput. Nota: en aquest context, s'entén que la lectura d'una dada del capdamunt de la pila, n'implica l'eliminació.
en stack

planificació prioritària *f* Mètode de planificació de tasques que es basa a suspendre l'execució d'una tasca quan una altra tasca de més prioritat passa a estar llesta (o bé, a una tasca de la mateixa prioritat, se li concedeix un interval d'execució). Són més complexes d'implementar que els planificadors seqüencials, permeten multitasca, però resulten més adequades per a respondre a esdeveniments externs al sistema.
en preemptive scheduling

planificació seqüencial *f* Mètode de planificació de tasques en què totes les tasques s'executen de manera consecutiva, seguint estrictament l'orde amb què han estat llestes per a la seva execució. En principi, no tenen en compte la prioritat de les tasques. Són simples d'implementar, no permeten multitasca real, ni resulten adequades per a aplicacions complexes en temps real.
en sequential scheduling

planificador *m* Element de programari integrat en el nucli del sistema operatiu responsable de decidir quina tasca ha d'utilitzar el processador en un moment determinat. Diferents metodologies de planificació tenen en compte diferents factors com el nombre i prioritat de les tasques, la presència de recursos per compartir, etc.
en scheduller

pointer *m* Vegeu **punter**, **punter d'instrucció**.

preemptive scheduling *f* Vegeu **planificació prioritària**.

prioritat *f* Urgència relativa d'una tasca o interrupció en comparació d'una altra. En el cas de les tasques, la prioritat és un nombre enter i, en funció de la prioritat de la resta de tasques concurrents, determina el temps i el torn de procés que li assignarà planificador per a la seva execució.
en priority

priority *f* Vegeu **prioritat**.

priority inversion *f* Vegeu **inversió de prioritats**.

procés de servei d'interrupció *m* sin. **rutina de servei d'interrupció**
en interrupt service process
sigla **ISP**

processador de senyals digitals *m* Dispositiu similar a un microprocessador, que ha estat dotat d'una CPU interna optimitzada per al seu ús en aplicacions que involucren el processament de senyals en temps discret. A més de les instruccions del microprocessador estàndard, els DSP solen donar suport a una sèrie d'instruccions especials, com ara multiplicacions i acumulacions, per a fer càlculs comuns de processament de senyal més de pressa. Normalment, es basen en una arquitectura Harvard, que manté separats els espais de memòria que contenen els programes i les dades, ja que permet una velocitat de transferència de dades superior. Algunes famílies habituals de DSP són la 320Cxx de Texas Instruments i la 5600x de Motorola.
en digital signal processor
sigla **DSP**

punter *m*
Variable que representa la posició (no pas el valor) d'una altra dada. Variable el valor de la qual és una adreça de memòria.
en pointer

punter d'instrucció *m*
Registre en un processador que conté l'adreça de la instrucció per executar següent.
en pointer

race condition *f* Vegeu **condició de carrera**.

real-time operative system *m* Vegeu **sistema operatiu en temps real**.
sigla **RTOS**

register *m* Vegeu **registre**.

registre *m* Posició de memòria integrada en un processador o un dispositiu d'entrada/sortida. En comparació de la memòria convencional, permeten velocitats de lectura/escriptura molt superiors. Normalment, la referència a un registre o altre està codificada com a part de la instrucció de baix nivell que executa el processador, i no com una adreça de memòria discreta.
en register

RTOS *m* Vegeu **sistema operatiu en temps real**.

rutina de servei d'interrupció *f* Petit programa que s'executa en resposta a una interrupció determinada. És equivalent al terme *gestor d'interrupcions*.
en interrupt service rutine
sigla **ISR**

scheduller *m* Vegeu **planificador**.

secció crítica *f* Seqüència d'instruccions que s'han d'executar en seqüència i sense interrupció per a garantir el funcionament correcte del programa. Si les instruccions són interrompudes, el funcionament del programa deixa de ser predictible, i per tant, incorrecte.
en critical section

semàfor *m* Estructura de dades que s'utilitza per a la sincronització entre tasques. És una de les eines que proporciona el sistema operatiu per a la sincronització entre tasques i pot ser de dos tipus: binari i amb comptadors (per exemple, tenen més de dos estats).
en semaphore

semaphore *m* Vegeu **semàfor**.

sequential scheduling *f* Vegeu **planificació seqüencial**.

sincronització entre tasques *f* Coordinació de la temporalització i seqüencialització entre les diferents tasques d'un sistema multitasca. La majoria d'RTOS inclouen multitud de recursos (semàfors, mútex, etc.) per a sincronitzar tasques de manera segura.
en intertask synchronization

sistema operatiu *m* En el context dels sistemes encastats, element de programari que fa possible el funcionament en multitasca distribuint els recursos disponibles entre les diferents tasques que cal dur a terme. Un sistema operatiu consisteix típicament en un conjunt de crides al sistema, un planificador de tasques, una rutina de servei d'interrupcions, un comptador/temporitzador de sistema i un conjunt de controladors dels dispositius perifèrics.
en operative system (OS)

sistema operatiu en temps real *m* Sistema operatiu dissenyat específicament per al seu ús en sistemes de temps real. S'utilitzen en els sistemes en què és imprescindible complir unes especificacions temporals determinades. Per exemple, garantir uns temps màxims en la resolució d'un càlcul o en la resposta a un esdeveniment extern.
en real-time operative system
sigla **RTOS**

software interrupt *f* Vegeu **interrupció de programari**.

stack *f* Vegeu **pila**.

suma de comprovació *f* Mètode de control de la integritat de les dades que permet detectar si han estat corrompudes. Una manera simple de fer-ho, i que, típicament, afegeix poques dades redundants al missatge, consistiria a sumar cadascun dels components bàsics d'un sistema (generalment cada byte) i emmagatzemar el valor del resultat. Posteriorment, es faria el mateix procediment i es compararia el resultat amb el valor emmagatzemat. Si ambdues sumes concorden, s'assumirà que les dades probablement no han estat danyades.
en check

tasca *f* Qualsevol càlcul individual, conjunt de càlculs, la lògica de la presa de decisions, intercanvi d'informació o combinació d'elles que ha de dur a terme en temps d'execució el programari. És l'abstracció central dels RTOS. Quan s'utilitza un RTOS, cada tasca ha de mantenir la seva pròpia còpia del punter d'instruccions de la CPU i els registres de propòsit general. La diferència principal amb els processos és que les tasques comparteixen un espai de memòria comuna. Per tant, cal anar amb compte per a evitar sobreescriure codi, dades i piles d'altres tasques.
en task

task *f* Vegeu **tasca**.

task list *f* Vegeu **llista d'execució**.

taula de vectors d'interrupció *f* Llista que conté el vector d'interrupció associat a cada tipus d'interrupció. Ha de ser inicialitzat abans que les interrupcions estiguin habilitades.
en interrupt vector table

temporitzador/comptador *m* Perifèric que compta esdeveniments externs (mode comptador) o cicles de processador (mode automàtic). Pràcticament, tots els microcontroladors tenen un o més comptador integrats en el seu maquinari.
en time/counter

termini *m* En un sistema en temps real, el moment en què un conjunt particular dels càlculs o les transferències de dades han de ser completats. S'anomenen *terminis durs* els que s'han de complir forçosament, ja que són imprescindibles per al funcionament correcte del sistema. S'anomenen *terminis tous* els que no tenen impacte en el funcionament correcte del sistema però sí que en poden afectar les prestacions.
en deadline

time/counter *m* Vegeu **temporitzador/comptador**.

time-slicing *m* Estratègia que permet simular el funcionament multitasca amb una única unitat de procés. Es basa a dividir el temps d'execució en intervals regulars (o llesques de temps) i durant aquest temps, dedicar el processador i la resta de recursos del sistema a l'execució d'una única tasca.

tipus d'interrupció *m* Un nombre únic associat a cada interrupció. Quan es produeix una interrupció, el processador utilitza aquest identificador per a consultar la taula de vectors d'interrupció i decidir com cal procedir.
en interrup type

unitat central de processament *f* sin. **processador**
en central processing unit
sigla **CPU**

vector d'interrupció *m* Adreça de la posició de memòria en què s'inicia una rutina de servei d'interrupció determinada.
en interrup vector

Bibliografia

Beltrán de Heredia, J. (2001). *Lenguaje ensamblador de los 80x86* (13a. ed.). Madrid: Anaya Multimedia.

Ganssle, J. G. (2000). *The Art of Designing Embedded Systems* (1a. ed.). Woburn, Massachusetts: Newnes (Elsevier).

Gottfried, B. (1996). *Programación en C* (2a. ed.). Mèxic, DF: McGraw-Hill / Interamericana de España.

Stuart R. B. (2002). *Embedded Microprocessor Systems: Real World Design*. (3a. ed.). Woburn, Massachusetts: Newnes (Elsevier).

Marvedel, P. (2003). *Embedded System Design* (1a. ed.). Dordrecht: Kluwer Academic Publishers.