

# **SISTEMA OPERATIVO PicOs**

**Ver 1.2**

**Autore: Francesco Conversi**

## **Indice Generale**

<b>1) Introduzione.....</b>	<b>2</b>
<b>2) Principio di funzionamento.....</b>	<b>2</b>
<b>3) System Calls.....</b>	<b>5</b>
<b>4) Organizzazione del kernel.....</b>	<b>6</b>
<b>5) Configurazione del kernel.....</b>	<b>6</b>
<b>6) Struttura tipica di un'applicazione.....</b>	<b>7</b>
<b>7) Esempio di utilizzo di alcune system calls.....</b>	<b>8</b>
<b>8) Area di applicazione di PicOs.....</b>	<b>10</b>

## 1) Introduzione

Il sistema operativo (SO) PicOs è stato scritto e testato per i microcontrollori Microchip della famiglia PIC16F87X ed ha come obiettivo quello di gestire la risorsa “tempo di CPU” fra più task. PicOs è stato scritto per essere impiegato in applicativi sviluppati in “C” con il compilatore PicC della Hitech, Ver. 7.87. Tuttavia esso è utilizzabile su tutti i PIC di tipo 16FXXX. Le caratteristiche di PicOs sono:

- è non preemptive;
- ha uno scheduler che realizza una politica basata su priorità;
- gestisce fino a 4 task più un idle task;
- utilizza complessivamente 40 byte di memoria RAM;
- utilizza circa 250 word di memoria FLASH più 20 word, circa, per ogni system call invocata;
- fornisce i servizi associati al tempo.

I vantaggi di questo SO, rispetto ad altri SO dedicati allo stesso tipo di microcontrollori, sono che:

- ha un tempo di scheduling, costante, di 28 cicli (=5.6us@20MHz) contro i 36 o più del SO Salvo della Pumpkin (22 della OS\_Yield() + 14-188 della OS\_Sched());
- ha un tempo di gestione dei timer software associati ai task di, massimo, 20 cicli (=4us@20MHz);
- permette l'uso di funzioni non rientranti;
- non consuma livelli di hardware stack.

Le limitazioni sono le seguenti:

- è non preemptive;
- lo stack è condiviso da tutti i task;
- le priorità dei task sono fisse e strettamente decrescenti;
- non ci sono meccanismi dedicati alla comunicazione tra i task.

PicOs rientra nella categoria dei SO detti cooperativi poiché tutti i tasks devono collaborare nel rilasciare la CPU, dopo averla utilizzata per un certo tempo, al fine di dar modo anche agli altri tasks di poterla utilizzare. Dato che in nessun caso il SO può interrompere la CPU mentre sta elaborando un task ed assegnarla ad un altro, ne segue che tutti i task devono essere di tipo non bloccante. È responsabilità quindi di chi crea l'applicazione di rispettare questa regola di funzionamento al di là del valore di priorità che viene assegnata ai tasks.

PicOs è adatto, come si vedrà più avanti, per applicazioni di tipo soft real time, ossia per applicazioni in cui i vincoli sui tempi di risposta del sistema sono statistici ed il sistema di calcolo fa del suo meglio per soddisfarli.

Il progetto di PicOs è stato indirizzato più all'efficienza, in termini di potenza di calcolo consumata dalle system calls e memoria RAM/FLASH impiegata, che alla generalità, ossia arbitrario numero di task, disponibilità di un ricco set di system calls, etc. Questo perché si è ritenuto che in microcontrollori “piccoli” come i 16F87X i limiti con cui più spesso ci si scontra sono i MIPS e l'ampiezza della memoria.

## 2) Principio di funzionamento

I microcontrollori Microchip della famiglia 16F87X si basano su una CPU RISC ad 8 bit con architettura di tipo Harvard. Tale microprocessore, data la sua semplicità, ha le seguenti limitazioni:

- ha uno stack hardware di soli 8 livelli, gestito in maniera circolare, in cui viene salvato il solo indirizzo di ritorno da una interrupt service routine (ISR) o da una chiamata a subroutine;
- non sono presenti istruzioni di PUSH o POP per manipolare lo stack.

Il fatto che in nessun modo si possa agire sullo stack porta come conseguenze che:

- l'unica tecnica di task switching realizzabile per questo uP è quella di salto da un task ad un altro.

- Questo tipo di task switching deve avvenire al medesimo livello per tutti i task, ossia non è possibile avere task switching all'interno di funzioni chiamate dai tasks, al fine di non provocare mal funzionamenti dello stack. La stessa limitazione è presente anche nel SO Salvo della Pumpkin ed in Leanslice della B Knudsen Data.
- Tutti i task devono necessariamente condividere l'unico stack esistente.

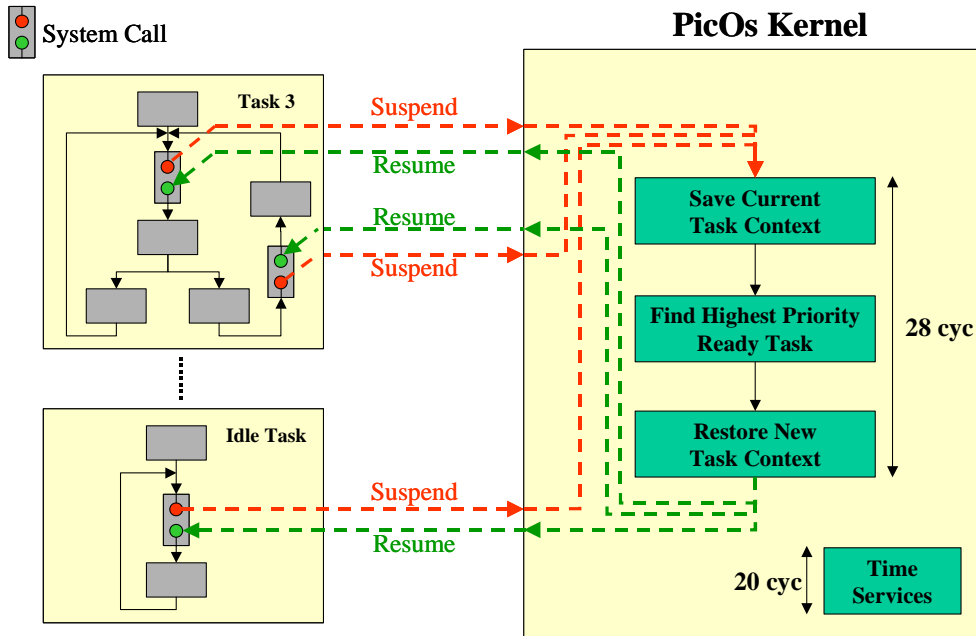


Fig. 1

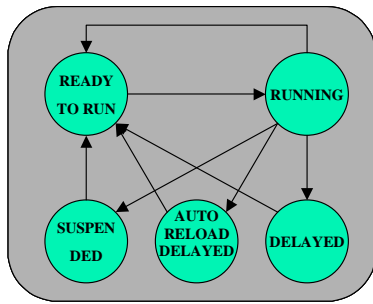
Ad ogni task è associata una struttura dati, chiamata generalmente Task Control Block (TCB), composta dalle seguenti informazioni:

- TaskEnPt (2 Byte) Task entry point, ossia l'indirizzo del punto del programma dal quale riprendere l'esecuzione,
- TaskStReg (1 Byte) Valore di ripristino dello status register,
- TaskCnt (1 Byte) Valore del contatore per le funzioni associate al tempo.

Le system calls di PicOs consistono in una serie di macro scritte per mezzo dell'assembler in linea il quale è supportato dal compilatore "C" della Hitech. In particolare le system call che richiedono un task switch comprendono l'invocazione della macro OS\_SwTaskFrom $n$ () che opera come segue:

- salva il contesto memorizzando lo STATUS register, che comprende le flag della CPU e l'indicazione dei banchi RAM e PROGRAM FLASH in uso, e l'indirizzo di ripristino, che consiste nell'indirizzo del codice immediatamente successivo a dove la macro è collocata;
- esegue un salto assoluto allo scheduler che ricerca il task pronto più prioritario;
- ripristina il contesto, ossia lo status register, ed esegue il ritorno al punto precedentemente lasciato del task scelto tramite un'istruzione di salto.

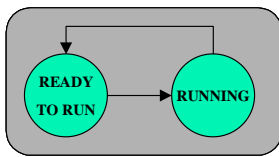
Queste tre fasi sono riportate in Fig. 1. I salti utilizzati da PicOs sono tutti di tipo "far" in modo che le posizioni in cui si trovano i tasks ed il sistema operativo possono essere arbitrarie ed indipendenti. Il fatto che questo SO basi il task switch su dei salti e non su delle chiamate di subroutine fa sì che nessun livello di stack sia consumato dal SO e quindi rimane inalterata il livello di annidamento con cui si possono chiamare le funzioni "C".



Task 0,1,2,3

### Task 0,1,2,3 States Coding

	TaskState Bit	AutoRel Bit	Timer Task
READY TO RUN	1	X	= 0
SUSPENDED	0	0	= 0
DELAYED	0	0	> 0
AR DELAYED	0	1	> 0



Task Idle

### TaskState

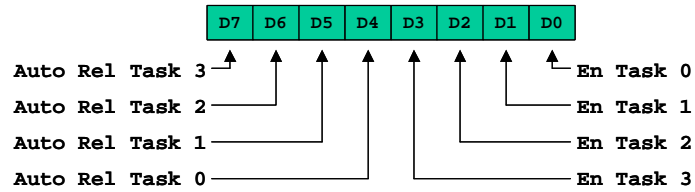


Fig. 2

Le variabili TaskEnPt vengono inizializzate con l'indirizzo delle "funzioni task" cosicché alla prima schedulazione della storia siano invocati i corrispondenti task. PicOs ha 4 task più l'idle task: questi vanno scritti come funzioni "C" che devono necessariamente chiamarsi task\_0, task\_1, task\_2, task\_3, task\_idle. Quindi questi nomi sono riservati a PicOs.

Un task si può trovare in uno degli stati riportati in Fig. 2: lo stato è rappresentato da una variabile TaskState comune a tutti i task e dalle variabili TaskCnt0÷3 per ciascuno dei 4 task.

L'idle task non ha una variabile di stato poiché è sempre READY\_TO\_RUN: esso è schedulato quando non ci sono task pronti più prioritari.

Come si vede dalla Fig. 2, i quattro bit meno significativi di TaskState corrispondono ai quattro task possibili ed indicano se il task è schedulabile (READY\_TO\_RUN) o meno. Tale nibble viene usato per indirizzare una jump table che intercetta le routine di ripristino del contesto di ciascun task. Dato che tale jump table è organizzata come in Fig. 3 si ottiene l'implementazione di una politica a priorità in un tempo di calcolo costante.

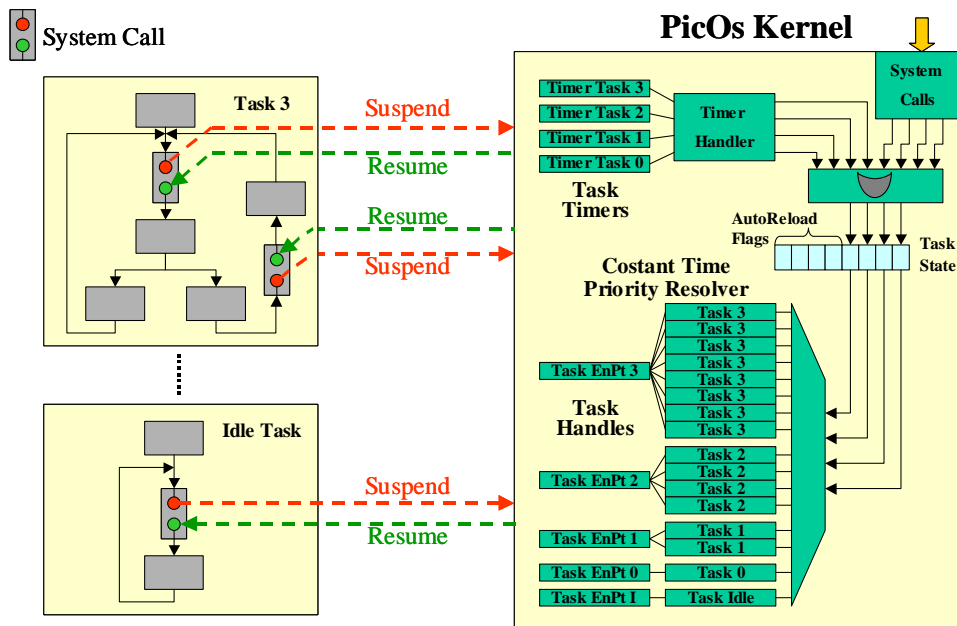


Fig. 3



## Gestione del Tempo:

- OS\_DelayTaskn ( dv ) Disabilita il task specificato per il tempo indicato in tick e rischedula.\*
- OS\_ArDelayTaskn ( dv ) Innesca il meccanismo di abilitazione periodica del task specificato per il tempo indicato in tick e rischedula.\*
- OS\_ArDelayTaskn ( dv ) Sospende il task, soggetto ad abilitazione periodica fino allo scadere del periodo, e rischedula.\*
- OS\_ArWaitTaskn ( ) Attende la fine del periodo di un task soggetto ad abilitazione periodica senza rischedulare.\*
- OS\_DisableArDelayTaskn ( ) Disabilita il meccanismo della abilitazione periodica di un task e rischedula.\*
- OS\_BlDelayTaskn ( dv ) Esegue una pausa senza rilasciare il controllo al SO
- OS\_EnableAutoReload ( tn ) Abilita il meccanismo della abilitazione periodica di un task e ma non rischedula.
- OS\_DisableAutoReload ( tn ) Disabilita il meccanismo della abilitazione periodica di un task e ma non rischedula.

Per tutte le system calls  $n=3,2,1,0$ ;  $tn = 3,2,1,0$ ;  $dv = 0 \div 65535$ .

\* Per ogni task dalla quale si chiama va usata la funzione/macro con il pedice corrispondente.

## 4) Organizzazione del kernel

PicOs è scritto in assembler in linea mentre la memoria per le sue strutture dati è dichiarata in un file "C" così da integrarsi al meglio con l'allocatore del compilatore HiTech; esso è composto dai seguenti file:

- PicOs\_c.c
- PicOs\_c.h
- pic\_cf\_c.h
- PicOs\_a.as
- PicOs\_a.inc
- pic\_cf\_a.inc

Per la configurazione di PicOs si deve operare unicamente sui files pic\_cf\_c.h e pic\_cf\_a.inc.

## 5) Configurazione del kernel

I parametri che ne configurano il funzionamento sono:

- Selezione del banco di memoria Ram di lavoro:  
si deve impostare sia in pic\_cf\_c.h e pic\_cf\_a.inc il banco RAM nel quale allocare le variabili. I possibili banchi sono 0,1,2,3 ma si consiglia di non utilizzare il banco 0 perché utilizzato dal compilatore HiTech per le variabili automatiche.
- Prescaler sul timer e valore di prescaling:  
la system call OS\_TimeService() esegue l'aggiornamento dei timer, che sono uno per ogni task. Due diverse modalità sono possibili per ottenere differenti prestazioni.
  - 1) Nel caso in cui si debba realizzare una applicazione in cui il ritmo del timer interrupt coincide con il tick di sistema allora il prescaler non viene utilizzato: in questo caso ogni volta che la OS\_TimeService() viene invocata vengono processati tutti e 4 i timer. Per ognuno di questi viene decrementato se  $>0$  e se dopo il decremento è uguale a 0 viene abilitato il corrispondente task.
  - 2) Nel caso in cui si debba realizzare un'applicazione in cui è necessario un ritmo del timer interrupt più veloce, oppure non multiplo, del tick di sistema desiderato, allora si può

abilitare il prescaler e settare un appropriato valore di prescaling. Il prescaler fa sì che l'effettiva gestione dei timer associata ai task avvenga una volta ogni periodo di prescaling. Inoltre la gestione dei 4 timer viene sparpagliata in 4 differenti timer interrupt consecutivi così da ridurre il peso relativo della ISR rispetto alle altre attività.

Esempio: si usi un timer interrupt a 125µs perché si deve gestire un'applicazione di audio processing con frequenza di campionamento ad 8KHz e si desidera un tick di sistema di 1mS. Allora si sceglierà un PRESC\_FACTOR pari ad 8 dato che  $8 \cdot 125\mu s = 1mS$ . Il contatore di prescaling ciclerà continuamente nella sequenza 7,6,5,4,3,2,1,0. I timer del sistema operativo saranno gestiti nei cicli 3,2,1,0.

## 6) Struttura tipica di un'applicazione

Vediamo com'è organizzato un applicativo basato su PicOs. Nei files sorgenti in cui si usano system calls di PicOs è necessario includere l'header file seguente:

```
#include "pic_os_c.h"
```

Per avere i servizi associati al tempo è necessario disporre di un timer che dia un interrupt periodico al ritmo del tick desiderato. Nella ISR del timer è necessario inserire la chiamata della OS\_TimeService(). Nel seguente esempio si suppone che sia stato impiegato il timer1 per tale scopo:

```

/*****          I S R          *****/
void interrupt PicIsr ( void )
{
    if ( CCP1IF )
    {
        OS_TimeService();
        CCP1IF = 0;
    }
}

```

Vanno quindi creati i tasks che avranno un prologo iniziale, tipicamente dedicato alle inizializzazioni e alle attività che vanno svolte solo una volta all'avvio del sistema, ed un corpo costituito da un ciclo senza fine. In particolare l'idle task deve avere la chiamata OS\_SwTaskFromI() che permette allo scheduler di girare. Tipicamente in questo task si gestisce il rinfresco del watch-dog nel caso che tale meccanismo sia utilizzato

```

/*****          T A S K S          *****/
void task_3 (void)
{
< init >
    for(;;)
    {
        <elaborazione>
    }
}
:
:
:
void task_0 (void)
{
    for(;;)
    {
        <elaborazione>
    }
}

void task_idle (void)
{
    for(;;)
    {
        WatchDogService();
        OS_SwTaskFromI();
    }
}
}

```

Il main program è semplicemente costituito da una parte, eseguita ad interrupt disabilitati, che provvede all'inizializzazione di tutte le attività e di tutti i periferici, compreso il sistema operativo. Dopo di ciò sono abilitati gli interrupt e viene eseguita la prima schedulazione della storia.

```

/*****      M A I N - P R O G R A M      *****/
void main ( void )
{
    di();                // Disabilita tutti gli interrupt
    :
    OS_InitScheduler(TASK_3); // Inizializza Scheduler e abilita il task 3
    :
    ei();                // Abilita tutti gli interrupt

    OS_SchedForEver();   // Inizia la schedulazione dei processi

    task_3();            // Chiamate dummy alle funzioni che definiscono
    task_2();            // i task del sistema operativo per evitare
    task_1();            // che il compilatore le elimini in quanto non
    task_0();            // le vede espressamente invocate in nessun punto
    task_idle();         // del codice.
}

```

Dopo la OS\_SchedForEver() ci sono le chiamate “inutili” alle funzioni task. La loro presenza è necessaria per evitare che l’ottimizzatore elimini il codice dei tasks non rilevandolo espressamente chiamato in nessun punto del codice. Dato che l’esecuzione non torna mai dalla chiamata della OS\_SchedForEver() la presenza di queste chiamate ai tasks è ininfluente ai fini della applicazione in quanto non vengono mai raggiunte.

Si ricorda ancora una volta che tutte le system call di PicOs che provocano una rischedulazione possono essere chiamate solo all’interno dei task e non all’interno di funzioni chiamate dai task.

## 7) Esempio di utilizzo di alcune system calls

Vediamo come organizzare alcuni task per ottenere le funzionalità tipiche di applicativi multitasking.

- Esempio 1: si voglia realizzare un task che esegue una operazione e quindi si auto sospenda per un tempo pari a 10 tick

```

void task_1 (void)
{
    <inizializzazione>
    for(;;)
    {
        <elaborazione>
        OS_DelayTask1(10);
    }
}

```

- Esempio 2: si voglia realizzare un task che esegue una operazione periodicamente ogni 15 tick

```

void task_0 (void)
{
    <inizializzazione>
    OS_ArDelayTask0(15);
    for(;;)
    {
        <elaborazione>
        OS_ArWaitTask0();
    }
}

```

Perché questo meccanismo funzioni è necessario che la durata dell’elaborazione sia minore del periodo impostato.

- Esempio 3: si voglia realizzare un task che viene attivato in conseguenza dell’arrivo di un interrupt, ossia si voglia realizzare una asynchronous service routine (ASR).

```

void interrupt PicIsr ( void )
{

```

```

if ( CCP1IF )          // l'interrupt e` del CCP1 ?
{
    OS_TimeService(); // Gestione degli OS Software Timers
    CCP1IF = 0;
}

if ( RCIF )           // l'interrupt e` del RX UART ?
    if ( RxUartIsr() )
        OS_IsrEnableTask(2);

if ( TXIF )           // l'interrupt e` del TX UART ?
    if ( TxUartIsr() )
        OS_IsrEnableTask(2);
}

void task_2 (void)
{
    OS_SuspendTask2(2);

    for(;;)
    {
        if ( RxFromMaster()==TRUE )
        {
            if ( Elab_Uart_Cmd()==TRUE )
            {
                TxToMaster();
                while (!EndTxToMaster())
                    OS_SuspendTask2(2);
            }
        }
        InitRx();
        OS_SuspendTask2(2);
    }
}

```

In quest'esempio si suppone che si debbano gestire dei pacchetti dati ricevuti dalla porta seriale. La funzione RxUartIsr() viene invocata, nella ISR della porta seriale, ogni qual volta viene ricevuto un nuovo carattere: essa, ad esempio, accumula i dati ricevuti in un buffer e quando rileva che un pacchetto completo è stato ricevuto ritorna TRUE. Ciò comporta che dall'interno della ISR venga abilitato il task 2. Tale task è normalmente nello stato SUSPENDED quindi di norma non viene schedato. Quando abilitato esso analizza il pacchetto ricevuto tramite la funzione RxFromMaster() e, se l'analisi ha esito positivo, vengono eseguite le elaborazioni corrispondenti con la Elab\_Uart\_Cmd(), viene preparata la risposta e avviata la trasmissione tramite la TxToMaster(). Dato che anche la trasmissione è gestita ad interrupt, e l'invio di un pacchetto richiede del tempo, il task 2 si sospende nuovamente. Al termine della trasmissione, gestita dalla TxUartIsr(), viene nuovamente abilitato il task 2 che provvede a reinizializzare la ricezione e a sospendersi per l'attesa di un nuovo pacchetto. Il vantaggio di questa architettura firmware è che durante tutto il tempo della ricezione o della trasmissione di un pacchetto possono essere svolte altre attività in background in quanto il task 2 non assorbe tempo macchina.

- Esempio 4: si voglia realizzare un task (2) che viene attivato in conseguenza del risultato dell'elaborazione di un altro task (1).

```

void task_2 (void)
{
    <inizializzazione>
    for(;;)
    {
        OS_SuspendTask2(2)
        <elaborazione>
    }
}

void task_1 (void)
{
    <inizializzazione>
    for(;;)
    {
        <elaborazione>
        if (risultato)
        {
            OS_ResumeTask1(2);
        }
    }
}

```

```

    }
}

```

- Esempio 5: si debba realizzare un task che esegue una elaborazione “lunga” e si desidera che le attività più prioritarie, nel caso vengano richieste, non siano ritardate oltre un certo tempo. Si deve allora suddividere l’elaborazione in più parti la cui durata massima determina la latenza di risposta agli eventi. Tra ogni porzione di elaborazione si deve invocare la OS\_SwTaskFromn() che fa girare lo scheduler.

```

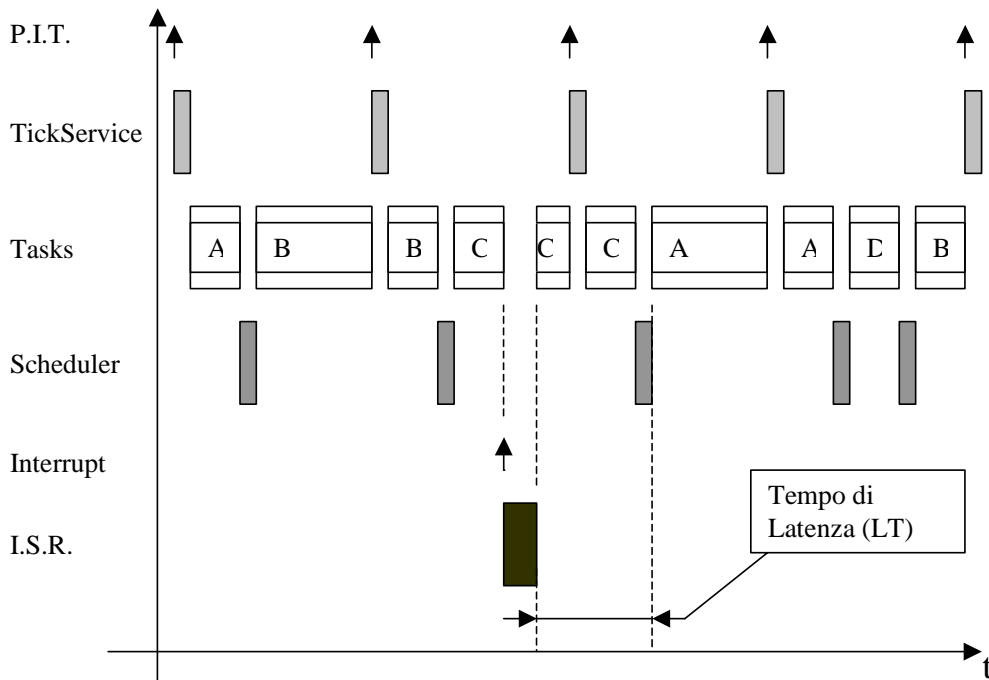
void task_2 (void)
{
    <inizializzazione>
    for(;;)
    {
        <elaborazione_1>
        OS_SwTaskFrom2();
        <elaborazione_2>
        OS_SwTaskFrom2();
        :
        <elaborazione_n>
        OS_SuspendTask2(2)
    }
}

```

## 8) Area di applicazione di PicOs

L'area in cui PicOs può essere di ausilio è quella delle applicazioni di tipo soft real time in cui la durata massima dei task è nota e si deve schedulare nel tempo la loro attività. Non è adatto ad applicazioni di tipo hard real time in quanto il tempo in cui un task in attesa di un evento riesce ad ottenere la CPU dipende dal tempo entro il quale il task corrente la rilascia. Tale tempo in generale non sarà costante e nel caso peggiore sarà pari alla durata del task temporalmente più lungo.

Per chiarire meglio questo concetto vediamo l'andamento temporale delle attività di un generico applicativo composto da 4 task, A,B,C,D con priorità decrescente e di durata diversa l'uno dall'altro.

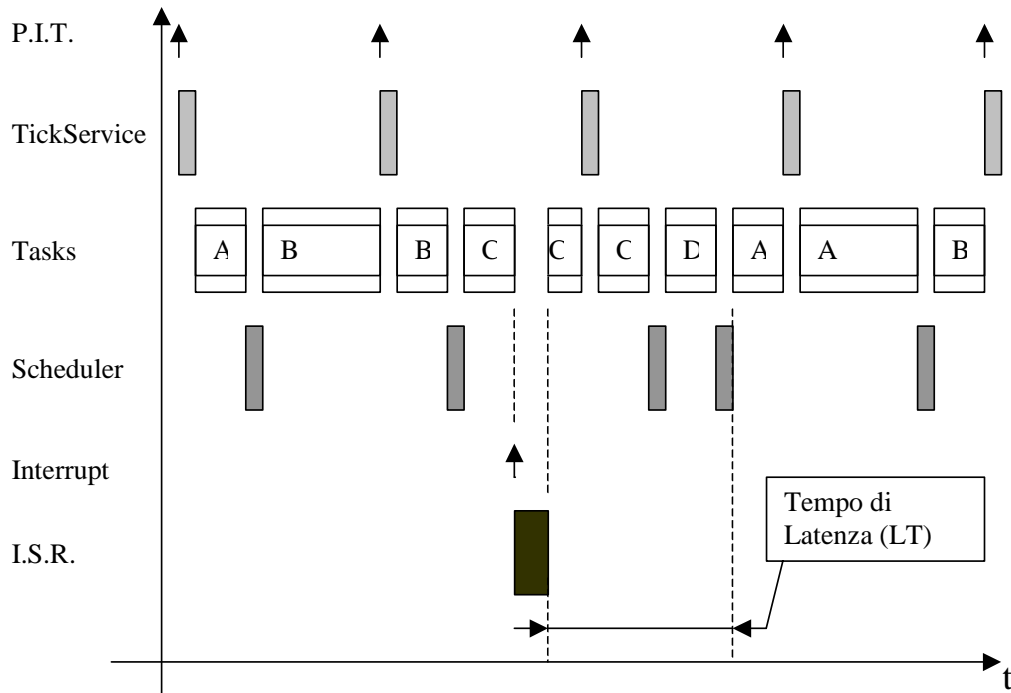


Nella figura sopra sono mostrate le diverse attività che svolge la CPU: con le frecce verticali sono indicati gli interrupt che la sollecitano mentre i blocchi rappresentano porzioni di programma che vengono eseguite. L'asse orizzontale rappresenta il tempo. Il carico elaborativo della CPU è ripartito tra i task ed il sistema operativo. In particolare per quest'ultimo contribuiscono:

1. la routine di servizio del tick, che dà un carico variabile a seconda del numero dei task che hanno richiesto un servizio associato al tempo ma, nella figura è supposto costante;

2. lo scheduler, che è a tempo costante indipendentemente dal numero di task presenti nella applicazione;
3. le system call, che sono state trascurate nella figura.

Supponiamo che il task A, appena è eseguito, si auto sospenda in attesa di essere abilitato da una ISR. Quando l'interrupt associato a tale ISR arriva, che nella figura è supposto avvenga mentre è in esecuzione C, il task corrente è interrotto, viene servito l'interrupt e poi riprende. Il task A, essendo il più prioritario, ottiene la CPU non appena C la rilascia. Il tempo che intercorre tra la fine della ISR e l'avvio di A è detto tempo di latenza (LT) di un task nella risposta ad un evento asincrono. Per ogni evento questo tempo è variabile in quanto dipende dal task che è stato interrotto e dal momento in cui arriva l'interrupt rispetto al



corso della sua elaborazione. Se il task interrotto è l'i-esimo e la sua durata è  $T_i$  allora il tempo di latenza varierà tra 0 e  $T_i$ .

In uno scenario di questo genere si può esprimere il valore di LT nel caso peggiore WCLT (Worst Case Latency Time) e nel caso medio MLT (Mean Latency Time). Detti  $T_i$ ,  $i=1..n$ , i tempi di esecuzione dei task allora:

$$WCLT = \max_{i=1}^n (T_i) \quad MLT = \frac{1}{2n} \sum_{i=1}^n T_i$$

dove, per l'espressione del MLT si è supposto che i task abbiano tutti la stessa frequenza di esecuzione. Per minimizzare questi tempi, in particolare WCTL, si deve minimizzare la durata massima di un task. D'altra parte, detti  $T_s$  e  $T_i$  i tempi di esecuzione dello scheduler e della ISR del timer periodico, il carico elaborativo offerto dal SO alla CPU è tanto più grande quanto più lo sono tali tempi rispetto a quelli dei task.

Se si rendono i task troppo brevi si peggiora l'efficienza mentre se si rendono troppo lunghi si peggiora il LT. Va quindi stabilito un compromesso. In un sistema ben dimensionato il SO assorbe non più del 5% del tempo macchina: il criterio di scelta della durata dei task può essere quello di soddisfare tale vincolo da cui ne seguono le prestazioni per la LT.

E' in ogni modo opportuno porre l'accento sul fatto che con un sistema operativo non preemptive si hanno prestazioni statistiche (ossia non deterministiche): solo nel caso migliore queste possono raggiungere le prestazioni di un sistema operativo preemptive.

Comunque, se non si adotta alcun SO, l'unica soluzione possibile è organizzare il main() come segue,

```
void main ( void )
{
    for (;;)
    {
        task_A();
    }
}
```

```
    task_B();  
    task_C();  
    task_D();  
}
```

ossia un loop senza fine in cui, in sequenza, vengono chiamati tutti i tasks, sempre di tipo non bloccante. Con tale soluzione, come si vede anche dal diagramma temporale, si ha che in nessun modo la sequenza di avvio in esecuzione dei tasks A, B, C, D può essere alterata dagli eventi esterni e quindi il tempo di latenza è superiore rispetto al caso precedente.

In definitiva quindi senza l'uso di PicOs:

- 1) non c'è alcun modo di alterare l'ordine di esecuzione dei tasks, quindi la WCLT e la MLT sono maggiori del caso in cui non si adotta il SO.
- 2) anche quando un task non ha bisogno della CPU esso va comunque in esecuzione: al suo interno poi un opportuno controllo farà sì che venga immediatamente rilasciata la CPU ma, c'è comunque il carico elaborativo dovuto alla chiamata;
- 3) la gestione del tempo va realizzata.

PicOs risolve completamente 2), 3) ed in parte 1), un SO preemptive invece risolve completamente anche 1).

I due principali motivi di difficoltà nella scrittura di un applicativo basato su PicOs sono:

- 1) tutti i tasks devono essere di tipo non bloccante;
- 2) la durata di ogni task deve essere nota affinché siano note le prestazioni del sistema complessivo (ciò invece non è necessario in SO preemptive in quanto il SO ha la facoltà di interrompere un task, comunque di durata infinita, per dare il controllo ad uno più prioritario).