Architetture per software embedded

Braione Pietro

Corso Sistemi Embedded

Anno accademico 2020-21

Strutturare il software

- La struttura del software determina quanto siamo in grado di controllare il tempo di risposta
- Questo dipende da:
  - Requisiti applicativi
  - Velocità del sistema di elaborazione (processore)
- Maggiore controllo = maggiore complessità architetturale
- Ma anche: maggiore complessità architetturale = maggiore lentezza

Il framework generale

- Il sistema embedded è collegato ad un insieme di periferiche
- E deve effettuare un insieme di computazioni
- Le computazioni sono guidate dall'input/output delle periferiche stesse: quando si verifica un evento significativo su una periferica occorre effettuare una computazione
- La computazione è strutturata in due parti:
  - Bisogna gestire la periferica stessa (esempio: trasferire dati da/verso i suoi buffer, riattivare i suoi interrupt…); di solito indispensabile perché la periferica possa elaborare un successivo evento
  - Bisogna implementare la logica applicativa associata all'evento (esempio: elaborare il dato ricevuto dalla periferica e presentarlo)

Quattro architetture software

- Round-robin
- Round-robin con interrupts
- Function-queue-scheduling
- Real-time operating system

Round-robin

- Niente interrupt, solo polling periferiche
- Il codice è strutturato come un ciclo che:
  - Legge lo stato dei dispositivi di I/O a turno
  - Effettua le corrispondenti elaborazioni in funzione dello stato

Round-robin: Struttura

```
void main() {
   while (true) {
      if (/* device 1 needs service */) {
         /* take care of device 1*/;
         /* handle data to/from device 1 */;
      }
      …
      if (/* device n needs service */) {
         /* take care of device n*/;
         /* handle data to/from device n */;
      }
   }
}
```

Esempio round-robin: multimetro digitale

```
void vDigitalMultimeterMain() {
   enum { OHM_1, … VOLT_100 } eSwitchPos;

   while (true) {
      eSwitchPos = /* read rotary switch position */

      switch (eSwitchPos) {
      case OHMS_1:
         /* Read from probes and convert to ohm */
         break;
      …
      case VOLTS_100:
         /* Read from probes and convert to volt */
         break;
      }
      /* Send read data to LCD */
   }
}
```

Round-robin: Discussione

- Vantaggi:
  - Molto semplice
  - Facilmente analizzabile: Il WCRT è pari ad un periodo del ciclo
- Svantaggi:
  - Il periodo del ciclo è data dalla somma del WCET di tutti i task
  - La latenza di reazione al cambio di stato di un dispositivo è sempre un periodo del ciclo
  - Conseguenza: fragilità se vengono aggiunti dispositivi o funzionalità

Round-robin con interrupt

- Sempre round-robin, ma con routine di interrupt
- Ogni routine di interrupt serve un dispositivo ed imposta un flag a true
- Il ciclo principale individua i flag impostati a true ed elabora i dati corrispondentemente

Round-robin con interrupt: Struttura

```
volatile bool fDevice_1 = false;
…
volatile bool fDevice_n = false;

void interrupt vHandleDevice_1() {
   /* take care of device 1 */;
   fDevice_1 = true;
}
…
void interrupt vHandleDevice_n() {
   /* take care of device n */;
   fDevice_n = true;
}
```

```
void main() {
   while (true) {
      if (fDevice_1) {
         fDevice_1 = false;
         /* handle data to/from device 1*/;
      }
      …
      if (fDevice_n) {
         fDevice_n = false;
         /* handle data to/from device n*/;
      }
   }
}
```

Esempio round-robin con interrupt: bridge

```
static QUEUE fInBuffer;
static bool fOutReady = false;

/* interrupts when data available */
void interrupt vHandleInChan() {
   char ch = /* read from in chan */;
   enqueue(&fInBuffer, ch);
}

/* interrupts when data sent */
void interrupt vHandleOutChan() {
   fOutReady = true;
}

/* the queue routines manage data
   sharing with the interrupt
   routines and must deal with the
   shared data problem */
```

```
void main() {
   init(&fInBuffer);
   enable_interrupts();

   while (true) {
      bool someData = !empty(&fInBuffer);
      if (someData && fOutReady) {
         char ch = dequeue(&fInBuffer);
         disable_interrupts();
         /* send ch to out chan */
         fOutReady = false;
         enable_interrupts();
      }
   }
}
```

Round-robin con interrupt: Discussione

- Vantaggi:
  - La latenza al cambio di stato di un dispositivo è inferiore che nel caso round-robin, al livello di priorità dell'IRQ
  - Inoltre dipende solo dagli interrupt a priorità maggiore, non dai task nel ciclo principale
- Svantaggi:
  - Routine di interrupt e ciclo principale devono sincronizzarsi sui dati condivisi
  - La durata di un periodo è meno predicibile
  - Tutto il codice del ciclo principale funziona allo stesso livello di priorità
    - Il WCRT di ogni task ciclico è ancora la somma dei WCET di tutti gli altri task
    - Unico trade-off: spostare codice dal task ciclico alla routine di interrupt, ma questo aumenta la latenza degli interrupt a priorità inferiore
    - Architettura poco adatta se uno dei blocchi nel task ciclico deve fare una computazione lunga

Function-queue-scheduling

- L’architettura è basata sull’uso di una coda con priorità di puntatori a funzione
- Le routine di interrupt aggiungono alla coda un puntatore alla funzione che effettua il calcolo associato all’interrupt
- Il main ciclico estrae i puntatori dalla coda ed invoca le rispettive funzioni

Function-queue-scheduling: Struttura

```
QUEUE qFunc;

void interrupt vHandleDevice_1() {
   /* take care of device 1 */;
   enqueue(&func_1, PRI_1);
}
…
void interrupt vHandleDevice_n() {
   /* take care of device n */;
   enqueue(&func_n, PRI_n);
}
```

```
void main() {
   while (true) {
      while (empty(&qFunc))
         ;
      void (*func)() = dequeue(&qFunc);
      func();
   }
}

void func_1() {
   /* handle data to/from device 1 */
}
…
void func_n() {
   /* handle data to/from device 1 */
}
```

Function-queue-scheduling: Discussione

- Vantaggi:
  - L'ordine di esecuzione delle funzioni riflette le priorità degli interrupt
  - In tal caso il WCRT del task a priorità massima è il WCET del task ciclico più lento, molto meglio di round-robin con interrupt
- Svantaggi:
  - D'altra parte l'esecuzione delle funzioni a priorità più basse può essere rimandata indefinitamente se gli interrupt a priorità alta sono troppo frequenti
  - Se il task ciclico più lento è molto lento, il WCRT del task a priorità massima potrebbe comunque essere eccessivo

Real-time operating system

- Sfrutta l'utilizzo di un sistema operativo real-time
- Anche in questo caso, le routine di interrupt effettuano i compiti più urgenti di gestione dei dispositivi
- Ad ogni dispositivo è inoltre associato un task del sistema operativo che effettua la computazione associata all'interrupt
- Le routine di interrupt segnalano al task associato che c'è del lavoro da fare utilizzando le primitive di segnalazione del sistema operativo

Real-time operating system: Struttura

```
OS_SIGNAL *sDevice_1;
…
OS_SIGNAL *sDevice_n;

void interrupt vHandleDevice_1() {
   /* take care of device 1 */;
   OS_notify(sDevice_1);
}
…
void interrupt vHandleDevice_n() {
   /* take care of device n */;
   OS_notify(sDevice_n);
}

void main() {
   sDevice_1 = OS_new_signal();
   …
   sDevice_n = OS_new_signal();
   OS_add_task(&task_1, …);
   …
   OS_add_task(&task_n, …);
   OS_start();
}

void task_1() {
   while (true) {
      OS_wait(sDevice_1);
      /* handle data to/from device 1 */
   }
}
…
void task_n() {
   while (true) {
      OS_wait(sDevice_n);
      /* handle data to/from device n */
   }
}
```

Real-time operating system: Discussione

- Vantaggi:
  - Non bisogna gestire esplicitamente lo scheduling delle computazioni: Questo viene gestito dal sistema operativo
  - Se il sistema operativo è preemptive, può sospendere l'esecuzione di una computazione a bassa priorità "nel mezzo" per eseguirne una a priorità più alta
  - In tal caso il task a priorità massima non attende prima di essere eseguito, e il suo WCRT è uguale al WCET
  - In genere, cambiamenti ai task a priorità più bassa non influiscono sul response time dei task a priorità più alta: quindi il design è più stabile ai cambiamenti
- Svantaggi:
  - Un sistema operativo ha un suo overhead: il miglioramento nel response time è ottenuto a fronte di un peggioramento del throughput
  - Nel caso in cui condividano dati, occorre sincronizzare anche le computazioni (non sono più eseguite in maniera run-to-completion mutuamente)
  - La licenza di un sistema operativo real-time commerciale costa (anche molto!); mitigato dal fatto che oggigiorno esistono sistemi operativi real-time con licenza open-source

Approcci a confronto

	Priorities available	Worst response time for task code	Stability of response when the code changes	Simplicity
Round-robin	All code at same priority	Sum of WCET for all tasks	Poor	Very simple
Round-robin with interrupts	Interrupt routines in priority order, then all task code at the same priority	Sum of WCET for all tasks (plus execution time for interrupt routines)	Good for interrupt routines; poor for task code	Must deal with data shared between interrupt routines and task code
Function-queue-scheduling	Interrupt routines in priority order, then task code in priority order	WCET of the task with maximum WCET (plus execution time for interrupt routines) for the task at highest priority	Relatively good	Must deal with shared data between interrupt routines and task code and must write function queue code
Real-time operating system	Interrupt routines in priority order, then task code in priority order	Zero (plus execution time for interrupt routines) for the task at highest priority	Very good	Most complex (although most of the complexity is managed through operating system primitives); might need to deal with shared data also across task code