INSTITUTO DE INGENIERÍA ELÉCTRICA

FACULTAD DE INGENIERÍA UDELAR

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Criterios básicos de diseño (generales y con RTOS)

Sistemas embebidos para tiempo real

NEON
Network of Competence on Internet of Things

Co-funded by the Erasmus+ Programme of the European Union

Índice

- Introducción (gran parte: repaso)
- Principios básicos de diseño
- Encapsulamiento
- Consideraciones: tiempo real “hard”
- Ahorro de memoria
- Ahorro de energía

Introducción

- Especificación de software de tiempo real es difícil:
  - especificar las acciones
  - tiempo de respuesta de cada acción
  - cuán crítico es cada requerimiento temporal
- Se necesita conocer
  - Hardware: características del microcontrolador (velocidad del procesador, sus periféricos)
  - Aplicación

Principios básicos de diseño

- Operación general:
  - no se hace nada hasta que un evento requiera una respuesta (eventualmente también pasado un tiempo)
- Interrupciones son el motor del software embebido
  - evento externo provoca una interrupción
  - el paso del tiempo provoca una interrupción mediante un temporizador

Principios básicos de diseño

- Diseñador debe determinar:
  - División del trabajo entre ISR y tareas.
  - Definir tareas: división de trabajo entre ellas.
  - Asignar prioridad a las tareas.
  - Comunicación de datos entre tareas.
  - Diseño de software que interactúa con hardware.
  - Cómo se evitan los problemas de datos compartidos.

Criterios para escribir ISR

Criterio básico y esencial: escribir ISRs cortas

- Fundamentación:
  - ISRs largas incrementan el tiempo de respuesta de:
    - las ISRs de menor prioridad (si hay prioridades & anidamiento)
    - del resto de las ISRs (si no hay prioridades)
    - de todas las tareas, incluso tareas de alta prioridad
  - ISRs tienden a ser más:
    - difíciles de depurar
    - propensas a bugs

Criterios para escribir ISR

- Típicamente tendremos eventos requieren múltiples respuestas secuenciales (con deadlines diferentes para cada respuesta)
- Ejemplo:
  - guardar dato recibido, analizar el dato recibido, estructurar una respuesta, responder
- Criterio básico:
  - ISR realiza acciones “inmediatas” luego encomienda el resto a una tarea

Balance entre ISR y tareas

- ISR extensa:

  Se resuelve todo:

  - Lee caracteres recibidos
  - Guarda en un buffer
  - Verifica EOM (fin de mensaje)
  - Interpreta y ejecuta el mensaje

- ISR mínima:

  La ISR solo:

  - Envía cada carácter a una tarea vía una cola

  La tarea:

  - Recibe cada carácter y en caso de EOM continúa procesamiento

Punto medio: entre ISR y tareas

- Ambos extremos son malas ideas:
  - ISR extensa: más compleja
    - difícil de depurar
    - aumenta latencia del resto de las respuestas
  - ISR mínima: más simple pero
    - demasiados mensajes enviados
    - overhead muy significativo
- Mejor solución:
  - ISR: guarda cada carácter en buffer y “le avisa” a la tarea cuando hay mensaje completo
  - Tarea: interpreta y ejecuta mensaje.

Criterios para definir tareas

- Pregunta: ¿Cuántas tareas? Más tareas implica...

Ventajas

- tareas más pequeñas, simples y fácil de depurar
- mejor control de tiempos de respuesta relativos
- más fácil de hacer el código modular
- más fácil de encapsular datos y hardware

Desventajas

- más datos compartidos y problemas asociados
- más necesidad de uso de semáforos (datos compartidos)
- más necesidad de uso de colas (comunicación entre tareas)
- más memoria
- más tiempo en cambio de contexto
- incremento general de overhead

Comparación: pros vs. contras

- Atención: al aumentar tareas
  - Desventajas: son directas
  - Ventajas: sólo si se diseña bien y se reparten cuidadosamente las tareas
- Recomendación final de Simon:
  - “siendo lo demás igual, usar tan pocas tareas como puedas, agregar tareas sólo si existen razones claras”

Tarea: estructura

vtaska.c

```
!! Private static data is declared here

void vTaskA (void)
{
  !! More private data declared here,
  !! either static or on the stack

  !! Initialization code, if needed.

  while (FOREVER)
  {
    !! Wait for system signal
    !! (event,msg, etc.)

    switch (!! type of signal)
    {
      case !! signal type 1:
         ...
         ...
        break;

      case !! signal type 2:
         ...
         ...
        break;
         ...
    }
  }
}
```

- Recomendación
  - Bucle infinito
    - espera de un mensaje
  - Bloqueo: un solo lugar
    - comportamiento fácil de entender
  - No tiene datos públicos

    sin problemas de:
    - datos compartidos
    - mal uso semáforos
  - Tiempo de respuesta
    - predecible
    - fácil de determinar

Otras consideraciones

- Creación y destrucción de tareas
  - funciones que más tiempo consumen
  - difícil de destruir una tarea sin provocar bugs
    - si tiene tomado un semáforo
    - si tiene mensajes encolados
  - crear todas las tareas al inicio
  - si una tarea no tiene que hacer permanece bloqueada
    - sólo consume memoria

Otras consideraciones

- Reparto del tiempo
  - Algunos RTOS permiten tareas de igual prioridad y división del tiempo entre ellas.
  - Ventajas:
    - Equidad: cada tarea hace algún progreso
  - Desventaja
    - Aumenta la frecuencia de cambio de contexto y el overhead

Otras consideraciones

- Para ahorrar recursos en un RTOS

  Para ahorrar memoria se pueden configurar

  - Remover servicios que no serán utilizados.
  - Adaptar el uso, por ejemplo: en lugar de 5 *pipes* y 1 cola utilizar 6 *pipes.*

Encapsulamiento

- Principio de *Information hiding*:
  - Ocultar decisiones de diseño que son probables de cambiar, de esta manera se protegen las otras partes del programa si el diseño es modificado.
- Características esenciales del encapsulamiento:
  - Interfaces: hacen las operaciones visibles, ocultan datos e implementación.
  - Implementación: puede ser cambiada sin cambiar la interface. Los programas están protegidos de cambios de bajo nivel.

Encapsulamiento: semáforos y colas

- Idea básica:
  - ocultar los detalles de implementación dentro de funciones
- Ventajas:
  - hace el resto del código más simple, se toma el cuidado en un solo lugar.
  - disminuye probabilidad de bugs resultantes de mal uso.
- Comentarios:
  - Ya hemos usado este concepto (semáforo para la función *cerrors*)
  - A continuación veremos un par de ejemplos

Semáforos y funciones reentrantes

```
void Task1(void)
{
   vCountErrors (1);
}
void Task2(void)
{
   vCountErrors (2);
}
```

```
static int cErrors;
static NU_SEMAPHORE semErrors;

void vCountErrors(int cNewErrors)
{
  NU_Obtain_Semaphore(&semErrors,NU_SUSPEND);
  cErrors += cNewErrors;
  NU_Release_Semaphore(&semErrors);
}
```

- Terminología:
  - Se ha "protegido" `cErrors` con semáforos
- Notación:
  - En este caso RTOS: Nucleus (prefijo NU)

Semáforo: ejemplo

- Temporizador: una variable representa el tiempo se comparte entre tarea e ISR (como en laboratorio).
- Opción 1: *not recommended*
  - Permitimos acceso directo, deben tomar un semáforo.
  - Confiamos todas las rutinas tomen y liberen el semáforo.
- Opción 2:
  - Limitamos el acceso a las rutinas esenciales:
    - inicialización/actualización del tiempo
    - función pública para leer y retornar su valor
  - Uso oculto del semáforo dentro de función reentrante. No es necesario utilizar el semáforo en otra parte.
  - ¿Cómo resolvimos esto en el laboratorio? Deshabilitando interrupciones.

Colas: ejemplo

- Ejemplo: rutina que toma mensajes de una cola
- Potenciales problemas:
  - Mensajes pueden ser ilegítimos: valores ilegales, etc.
  - Mensajes pueden ser enviados a colas equivocadas.
  - Otros (debido a la naturaleza global de la cola).
- Solución:
  - Poner todas las funciones que acceden a la cola en un archivo `.c`
  - Declarar la cola `static` (acceso dentro del archivo).
  - Escribir funciones reentrantes para leer y escribir la cola
  - Otras funciones simplemente llaman `read(...)`, `write(...)` y no saben siquiera como se utiliza la cola (o si se utiliza una)

Encapsulamiento

- ¿Cuáles son los candidatos cantados?
  - Acciones que hacen un código no reentrante:
    - Acceso a recursos compartidos: datos y hardware compartido.
  - Construcciones difíciles de usar sin introducir bugs:
    - Estructuras de datos compartidos, semáforos, colas.
- Conclusión:
  - encapsular (para evitar el acceso directo) a:
    - datos compartidos
    - semáforos
    - colas
    - hardware

Sistemas de tiempo real “hard”

- Diseño debe garantizar que *hard-deadlines* se cumplan.
  - Factores que contribuyen
    - Código de ISR y tareas: eficiente
    - Eficiencia del compilador
    - Asignación de prioridades
    - Frecuencia de las interrupciones y cambio de contexto
    - Performance del microprocesador
  - Para garantizar todos los *deadlines*, se debe saber:
    - Peor caso de tiempo de ejecución de ISR y tareas
    - Máxima frecuencia
  - En un sistema real obtener éstos datos no es fácil.

Ahorro de memoria

- Sistemas embebidos tienen limitada memoria:
  - Código y datos constantes van en ROM
  - Datos variables van en RAM
- Ahorro en datos
  - Empaquetar datos (costo: aumento de código)
  - Elegir adecuadamente
    - tipo de datos (cantidad bits)
    - tipo almacenamiento (estático o automático)
  - Espacio para *stack* de tareas, solo el necesario
    - análisis estático
    - análisis experimental

Ahorro de memoria

- Ahorro en código
  - Examinar y verificar que el código compilado es razonable.
  - Asegurarse que las bibliotecas de terceros no nos están saboteando (ejemplo: usar una función que incluye otras).
  - Evitar funciones redundantes (ejemplo: usar funciones diferentes para cosas similares)
  - Asegurarse que el RTOS incluye sólo los servicios necesarios.

Ahorro de energía

- Enfoque general:
  - Apagar partes/periféricos que no se usen.
  - Procesar a máxima velocidad, y después “irse a dormir”.
- Microcontroladores:
  - Ofrecen modos de ahorro de energía
  - Detalles y grado de actividad en cada modo dependen del fabricante

Actividad en grupo / Deberes

- Bosquejo del diseño de la solución del proyecto
  - Algunos elementos a determinar:
    - Arquitectura de software embebido.
    - Definir tareas.
    - Asignar prioridad a las tareas.
    - División del trabajo entre ISR y tareas, y entre tareas.
    - Comunicación de datos entre tareas.
    - Diseño de software que interactúa con hardware.
  - Grupos:
    - Participantes de cada proyecto.
  - Tiempo:
    - 15 minutos

Bibliografía

- “An Embedded Software Primer”David E. Simon
  - Chapter 8: Basic Design Using a Real-Time Operating System