Sistemas de Tiempo Real y RTOS
Conceptos fundamentales basado en XINU RTOS

Índice general

1 Sistemas de Tiempo Real y RTOS 1
1.1 Sistemas de tiempo real 1
1.2 Sistema Operativo de Tiempo Real 3
2 El limite del bucle infinito 5
2.1 Ejemplo: robot movil con odometria y temperatura . . . 6
2.2 Ejemplo de un sistema con eventos asincronicos 9
3 Patrones tipicos de uso de tareas con un RTOS 12
3.1 Tarea periodica 12
3.2 Tarea activada por evento 12
3.3 Productor-consumidor via interrupciones 13
3.4 Productor-consumidor 15
4 XINU RTOS en atmega328p 16
4.1 XINU para arquitectura AVR (atmega328p) 16
4.2 Creacion y finalizacion de tareas 17
4.3 Tareas periodicas 18
4.4 Ejemplo completo: cuatro tareas con prioridades distintas 19
4.5 Instrucciones para utilizar el código fuente de XINU RTOS 21
5 Licencia 23

*

Chapter 1

Sistemas de Tiempo Real y RTOS
Conceptos fundamentales basado en XINU RTOS

Rafael Ignacio Zurita
Universidad Nacional del Comahue

Muchos sistemas embebidos deben responder a eventos dentro de plazos temporales definidos. En estos casos, no alcanza con que los resultados computados sean correctos: también es necesario que sean producidos dentro del tiempo especificado. A este tipo de aplicaciones se las denomina sistemas de tiempo real.

En este capítulo se introducen los conceptos fundamentales de los sistemas de tiempo real, las diferencias entre sistemas hard y soft real-time, y el papel que cumplen los sistemas operativos de tiempo real (RTOS) en su implementación. Se analiza por qué los esquemas tradicionales basados en un bucle infinito principal presentan limitaciones cuando existen múltiples requisitos temporales, y cómo un RTOS permite organizar la aplicación en tareas concurrentes con prioridades definidas.

Además, se presentan los mecanismos básicos que ofrecen los RTOS modernos, incluyendo planificación apropiativa, sincronización, comunicación entre tareas y manejo de eventos asincrónicos. Finalmente, se describen patrones habituales de diseño de tareas en sistemas embebidos y se introduce una adaptación del sistema operativo XINU para microcontroladores AVR como plataforma de estudio y experimentación.

Keywords: sistemas de tiempo real, RTOS, planificación de tareas, sistemas embebidos, concurrencia, sincronización, XINU

> –
> One of my most productive days was throwing away 1000 lines of code.
> — Ken Thompson

1 Sistemas de Tiempo Real y RTOS

1.1 Sistemas de tiempo real

Un sistema es de tiempo real cuando en la especificación de sus requerimientos existen requisitos de tiempo de respuesta limites. Por lo que su correctitud, una vez implementado, esta definida no sólo por la correctitud de los resultados computados, sino también porque cumple con los tiempos de respuesta definidos en sus especificaciones. Si el sistema no es capaz de responder a un evento

o tarea en el tiempo maximo permitido (evento y tiempo límite definidos en las especificaciones), entonces se dice que el sistema falló, y no es de tiempo real.

Se los categoriza en dos tipos. **Hard Real-Time (tiempo real estricto)**: Cuando los plazos temporales deben ser cumplidos siempre. En este tipo de sistemas están los sistemas llamados ***críticos***, en donde una resputa fuera de tiempo puede poner en riesgo la vida humana o la estabilidad del sistema. Por ejemplo, la activación de un airbag en un automóvil. **Soft Real-Time (tiempo real flexible)**: Son los sistemas en donde incumplir ocasionalmente un tiempo de respuesta degrada el rendimiento, pero no provoca un fallo catastrófico. Por ejemplo, un sistema de reproducción de video en streaming.

Un Sistema es de Tiempo Real si..

1. Los resultados computados son correctos.
2. Se cumplen los tiempos de respuesta definidos en sus especificaciones.

Un resultado producido fuera del plazo definido equivale, en este contexto, a un sistema que NO es de tiempo real.

Usualmente, al diseñar un sistema de tiempo real, se definen funcionalidades principalmente de dos categorías: periódicas o activadas por eventos. Para cada una de estas se definen también, con precision, los tiempos de respuesta maximos permitido por parte del sistema (sobre todo para las funcionalidades activadas por eventos). Luego, este diseño se implementa[1] generalmente como tareas independientes, que seran ejecutadas de manera concurrente en tiempo de ejecución.

Como software de apoyo, se utiliza un sistema operativo de tiempo real (RTOS, por su sigla en ingles en Real Time Operating System). Estos se ejecutan principalmente en microprocesadores de baja complejidad y, mayormente, en microcontroladores. Este software es el encargado de ejecutar las tareas de manera concurrente, y también da soporte para lograr los tiempos de respuesta requeridos. Sin embargo, esto no es un proceso automatico. Es decir, utilizar un sistema operativo de tiempo real no implica que el sistema logrado sea de tiempo real.

Importante

Utilizar un RTOS NO garantiza automaticamente que el sistema sea de tiempo real. Para lograrlo, los desarrolladores deben diseñarlo apropiadamente. Por ejemplo, asignando prioridades correctas a cada tarea (si el sistema gestiona tareas con prioridades), asignando eventos a cada tarea, y definiendo como deben sincronizarse y comunicarse entre si.

[1] Etapa de desarrollo y programación del código fuente

1.2 Sistema Operativo de Tiempo Real

Los sistemas operativos de tiempo real son herramientas de desarrollo, provistas usualmente en formato código fuente. Permiten desarrollar un sistema con tareas concurrentes, y por su tamaño y eficiencia, muy utilizados para desarrollar sistemas embebidos clásicos y sistemas embebidos de tiempo real. Los RTOS no son sistemas operativos completos. Estan compuestos, generalmente, por un gestor de tareas apropiativo y de mecanismos para la sincronizacion y comunicacion entre las mismas. Este reducido conjunto de caracteristicas esta directamente relacionado con el hardware donde se ejecuta: los RTOS son utilizados mayormente en plataformas de computo basado en microcontroladores de bajos recursos para automatizacion y control.

Existen distintos métodos y mecanismos que un software de apoyo RTOS puede incluir para soportar el desarrollo de un sistema de tiempo real. Uno ampliamente utilizado es el uso de un RTOS con planificador de CPU basado en *prioridades y apropiativo*. Cuando el sistema está en ejecucion, cada vez que el RTOS activa una tarea, esto es, la coloca en estado de ”listo para ejecutar”, verifica su prioridad. Si esta prioridad es más alta que la de la tarea que actualmente utiliza la CPU, entonces el RTOS realiza un cambio de contexto y coloca a la nueva tarea de mas alta prioridad a ejecutar.

Si existen varias tareas listas para ejecutar, con la misma prioridad que la tarea que esta actualmente utilizando la CPU, entonces el RTOS se apropia de la CPU en intervalos regulares, empleando la planificación *Round-Robin* para asignar la CPU a otra tarea. Aquí, cada tarea se ejecuta durante un QUANTUM de tiempo fijo, antes de ser interrumpida nuevamente, para que el planificador asigne la CPU a otra tarea de la misma prioridad. Luego de que todas las tareas de la misma prioridad se ejecutaron durante su QUANTUM, el RTOS volverá a asignar la CPU a la primera tarea de esta cola FIFO.

Cuando un RTOS funciona con estas caracteristicas, los desarrolladores pueden diseñar un sistema cuyo funcionamiento es predecible, lo cual es una condicion necesaria para luego evaluar su comportamienteo y conocer si el sistema resultante cumple con los plazos definidos.

En el momento de modelado y diseño del sistema, con esta técnica en mente, el desarrollador debe especificar para cada tarea una prioridad, de acuerdo a si esa tarea debe tener o no un tiempo de respuesta limite. También relacionar las tareas con los eventos, y definir como deben sincronizarse y comunicarse esas tareas. En tiempo de ejecucion, el sistema operativo de tiempo real será el encargado de asegurar que siempre se este ejecutando la tarea de mayor prioridad.

Como el comportamiento del sistema bajo un RTOS es predecible el desarro-

llador del sistema embebido puede evaluar el modelo diseñado ante todas las situaciones posibles, para corroborar que siempre se cumplen los tiempos de respuestas definidos. Si la cantidad de estados del sistema y situaciones posibles son inabarcables, los desarrolladores pueden realizar simulaciones y aproximaciones para evaluar si el sistema cumplirá con los plazos requeridos.

Existe también un mecanismo formal para validar un modelo de sistema de tiempo real el cual se basa en convertir los eventos asincronicos a periodicos, y evaluar luego todo el sistema matematicamente. Puede leer mayor documentacion al respecto en *"Sistemas de Tiempo Real y Lenguajes de Programación" 3ra. edición, Alan Burns y Andy Wellings, Addison-wesley 2003, ISBN:9788478290581.*

1.2.1 Caracteristicas de un RTOS

En sintesis, un RTOS que implementa el esquema de prioridad fija y apropiatividad (que es el caso más común) debe contar con las siguientes caracteristicas:

- Administrador de tareas: debe poder ejecutar tareas diferentes de manera concurrente y ser apropiativo (preemptive).
- Planificador de CPU basado en prioridades: cada tarea debe contar con una prioridad, y el RTOS se encarga de que la tarea de mayor prioridad es la que utiliza la CPU. Usualmente el planificador de CPU trabaja en modo Round-Robin para las tareas con igual prioridad.
- Sincronizacion y comunicacion entre tareas: el RTOS debe contar con mecanismos para que las tareas puedan coordinarse y compartir datos de forma segura (semaforos, mutex, colas de mensajes, etc.).
- Herencia de prioridad: debe existir un mecanismo de herencia de prioridad para evitar la inversion de prioridades, esto es, que una tarea de baja prioridad no bloquee indefinidamente a una de alta prioridad.
- Eventos asincronicos: el RTOS debe permitir eventos asincronicos, como pueden ser las interrupciones de E/S, permitiendo reaccionar ante eventos externos sin consulta activa (polling).

1.2.2 Donde se utilizan los RTOS

Los RTOS son utilizados mayormente en plataformas de computo basados en microprocesadores y microcontroladores de bajos recursos para automatizacion y control.

En los ultimos años, con el crecimiento exponencial de desarrollo de dispositivos IOT, su uso ha crecido en ambientes de sensores conectados a Internet, también controlados generalmente por microcontroladores. Algunos ejemplos de aplicación embebidas típicas desarrolladas utilizando un RTOS son:

- Robots moviles, : control de motores, lectura de encoders opticos, calculo de odometria.
- Sistemas de adquisicion de datos: muestreo periodico de sensores con restricciones temporales.
- Controladores de procesos industriales: automatizacion con respuesta garantizada ante eventos.
- Dispositivos IOT: sensores conectados que reportan valores dentro de tiempos limite.
- Sistemas de navegacion: fusion de datos de magnetometro, acelerometro y encoders con plazos estrictos.

2 El limite del bucle infinito

Para apreciar por qué es muy comun utilizar un RTOS al desarrollar un sistema embebido comenzaremos con un ejemplo típico. Antes de introducir un RTOS, muchos sistemas embebidos de baja complejidad se implementan con un esquema simple: un bucle infinito principal (superloop) que atiende la logica de la aplicación, combinado con rutinas de servicio de interrupcion (ISR) para manejar eventos asincronicos. Este esquema es facil de entender y funciona bien cuando existe un unico requisito de tiempo de respuesta o las tareas tienen plazos muy similares. Pero en cuanto aparecen dos o mas requisitos temporales diferentes, el diseno se vuelve rapidamente inviable.

La razon de fondo es sencilla: en un bucle infinito, todas las tareas comparten el mismo hilo de ejecucion. No existe ningun mecanismo que garantice que una tarea critica sea atendida a tiempo si otra tarea menos critica esta en ese momento ocupando la CPU. La ISR puede despertar un flag o actualizar una variable, pero si el bucle principal esta en medio de una operacion larga, el procesamiento del evento llegara tarde.

2.1 Ejemplo: robot movil con odometria y temperatura

Suponga que se debe desarrollar un sistema embebido robotico que navega y controla la temperatura en diferentes ambientes. El robot realiza dos funciones principales:

- Odometria: el sistema debe leer encoders opticos que permiten calcular la velocidad de cada rueda del robot, y con esta información, calcular la posicion del robot **una vez cada 10ms** al menos. Al implementar la funcionalidad se evalúa y se demuestra que el computo demora **3ms** en este microcontrolador.
- Temperatura: el sistema debe leer varios sensores de temperatura. Si al procesar esos valores se exede un umbral se activa una alarma y se accionan actuadores de control. Esta función debe realizarse al menos **una vez cada 5ms**. Si el calculo se realiza luego de 5ms se accionarán los actuadores fuera del margen aceptable. Al implementar esta funcionalidad se evalúa y se demuestra que esta funcionalidad se logra en **3ms** utilizando el microcontrolador seleccionado. **Esta tarea se define como la de mayor importancia**.

2.1.1 Intento con bucle infinito e interrupciones

Una primera implementación tradicional es utilizar dos timers por hardware para activar las diferentes tareas a tiempo, mediante interrupciones (cada 10ms y cada 5ms). Luego, procesar todo en el bucle principal de main.
. En el Código Fuente 1.1 se observa la implementación con un bucle infinito.

Donde falla este diseño

- calcular_odometria() demora 3ms, y debe finalizar este cálculo en menos de 10ms desde que se activó la tarea.
- procesar(temperaturas) demora 3ms, y debe finalizar este cálculo en menos de 5ms desde que se activó la tarea.

El problema con la solución tradicional es que si las tareas se activan en algun momento casi simultaneamente, con tarea_odom en primer lugar, entonces tarea_temp (temperatura) finalizará a los 6ms, lo que producirá como resultado que la **tarea crítica superó el tiempo límite**. Tal incumplimento del plazo producirá un error irrecuperable. En este ejemplo, es posible que al no accionar los

```
ISR(TIMER0_OVF_vect) {        // interrupcion cada 10 ms
  velocidad = leer_encoder();
  tarea_odom = 1;
}

ISR(TIMER1_OVF_vect) {        // interrupcion cada 5 ms
  temperaturas = leer_sensores_temp();
  tarea_temp = 1;
}

int main(void) {

  for(;;) {
          if (tarea_odom) {       /* odometria */
              tarea_odom = 0;
              calcular_odometria(velocidad);      // demora 3ms
                                                  // PROBLEMA !!
          }

          if (tarea_temp) {       /* temperatura */
              tarea_temp = 0;
              PELIGRO = procesar(temperaturas);   // demora 3ms
              if (PELIGRO) {
                  accionar_actuadores();
                  alarma_ON();
              }
          }
  }
}
```

Listing 1.1: Resolución de bucle infinito

actuadores a tiempo se produzca un incendio o problema en la planta que se esté controlando.

Una alternativa que inmediatamente muchos estudiantes intentan proponer es modificar las rutinas de atención de interrupciones, y colocar dentro el procesamiento de cada tarea. De esta manera, si se está procesando la odometría, y aparece una interrupción de temperaturas, se ejecutará inmediatamente la rutina de atención de interrupciones procesando las temperaturas a tiempo. Esta solución, aparentemente apropiada en principio, tiene el inconveniente de que las rutinas de atención de interrupciones tomarán mucho tiempo en ejecutarse. Como se vió en módulos teóricos previos, una rutina de atención de interrupciones debe activarse, realizar alguna acción lo más rápido posible y finalizar. No es

aceptable tener a estas rutinas en ejecución por varios ms. La principal razón es que otras interrupciones no serán atendidas (mientras se ejecuta una rutina de este tipo las interrupciones están desactivadas). En caso de activar las interrupciones dentro de rutinas críticas, para no perder otras interrupciones, aún sigue siendo un gran problema mantener este código. Casi imposible ampliarlo con mayores funcionalidades por rutinas de atención de interrupciones anidadas.

2.1.2 Solucion con RTOS y prioridades

Utilizando como herramenta de apoyo un RTOS, cada requisito se convierte en una tarea independiente con su propia prioridad. El planificador siempre asigna la CPU a la tarea de mayor prioridad lista para ejecutarse. Esto es, independientemente de lo que este haciendo alguna otra tarea de menor prioridad. Cuando una tarea de mayor prioridad se activa, mientras otra está utilizando la CPU, el planificador de CPU del RTOS realiza un cambio de contexto, y pone a ejecutar la tarea de mayor prioridad.

Aqui, si tarea_odometria se está ejecutando, y en ese instante vence el periodo de espera de 5ms de la tarea_temperatura, el planificador interrumpe a tarea_odometria y le da la CPU a tarea_temperatura. Cuando esta termina, el RTOS bloquea a tarea_temperatura por otros 5ms y le devuelve la CPU a tarea_odometria para que continúe su ejecución.

Es decir, sleepms() no hace una espera activa. sleepms() es una llamada al sistema del RTOS que pone a la tarea en estado SLEEPING. En ese estado, la tarea nunca consume CPU. Cuando sucedan exactamente 5ms, el RTOS pone a la tarea_temperatura en estado de LISTO. En este punto, inmediatamente la coloca en estado EJECUTANDO y realiza un cambio de contexto para que utilice la CPU, ya que al ingresar al estado de LISTO, y ser de la mas alta prioridad, el RTOS asegura su ejecución inmediata. El plazo de 5ms se cumple siempre, sin necesidad de calcular manualmente cuanto tarda cada función.

Un lector atento observará que poner sleepms(5); no sería adecuado. Si la tarea demora 3ms en ejecutarse, entonces la tarea se estaría ejecutando cada 8ms (y no cada 5ms). sleepms(2) sería mas apropiado. Otra solución mas eficiente podría ser utilizando un timer y un semáforo provisto por el RTOS para activar la tarea, o tener una tarea que haga de temporizador con igual mecanismo.

El lector puede además, utilizando este ejemplo, observar lo sencillo que sería agregar mayores funcionalidades. Si el sistema requiere otras tareas, simplemente se las desarrolla de manera independiente, como en este caso. Utilizando las prioridades adecuadas se puede modelar el comportamiento de manera predecible, y cumplir con los tiempos de respuesta necesarios para cada funcionalidad.

```
process tarea_odometria(void)        // Prioridad 15: tiempo de respuesta < 10ms
{
   for (;;) {
        velocidad = leer_encoder();
        calcular_odometria(velocidad);      // demora 3ms

        sleepms(10);
    }
}

process tarea_temperatura(void)      // Prioridad 40: tiempo de respuesta < 5ms
{
   for (;;) {
        temperaturas = leer_sensores_temp();

        PELIGRO = procesar(temperaturas);     // demora 3ms
        if (PELIGRO) {
            accionar_actuadores();
            alarma_ON();
        }

        sleepms(5);
    }
}

void main(void)
{
    resume(create(tarea_odometria, 256, 15, "odom", 0));
    resume(create(tarea_temperatura, 256, 40, "temp", 0));
}
```

Listing 1.2: Resolución utilizando un RTOS

2.2 Ejemplo de un sistema con eventos asincronicos

Ahora se agrega, al mismo ejemplo de la sección anterior, un requisito extra: el sistema debe poder atender comandos que llegan por una interfaz de hardware serie desde un dispositivo remoto. Los comandos no llegan en forma periodica. Pueden llegar en cualquier momento, y el sistema debe responder dentro de los 20ms desde la recepcion. Si no responde a tiempo, el sistema se vuelve inestable.

Al implementar esta funcionalidad se evalúa que su ejecución demora 10ms.

Como la tarea de temperatura es muy justa en tiempo de ejecución con respecto a su tiempo de activación, aún sigue siendo la tarea más crítica. Agregando este requisito al escenario anterior:

- Tarea Odometria: cada 10 ms. Prioridad baja.
- Tarea Temperatura: cada 5 ms. Prioridad maxima.
- Tarea de Comandos: responder en menos de 20 ms desde la llegada del comando a procesar. Prioridad alta.

Se debe agregar la nueva funcionalidad como tarea independiente, y su prioridad debe estar definida con un valor intermedio, de manera que todas las tareas se cumplan a tiempo.

Cuando la rutina de atención de interrupciones ejecuta signal(sem_comando), el planificador del RTOS despierta a tarea_comandos. Como su prioridad (20) es mayor que la de tarea_odometria (15), si esta ultima se estaba ejecutando, es interrumpida inmediatamente. El tiempo entre la llegada del byte via el hardware serie y el inicio de procesar_comando() es del orden de microsegundos: el plazo de 20ms se cumple con margen. Si en cambio, la tarea en ejecución fuese tarea_temperatura, al activarse la tarea_comando, entonces el RTOS pondrá a ejecutar esta última recién luego de que tarea_temperatura haya finalizado su procesamiento y delegue la CPU voluntariamente con sleepms(5).

```
// ---- Solucion utilizando un RTOS ----

sid32 sem_comando;                    // semaforo global, inicializado en 0
char cmd_byte = 0;

ISR(USART_RX_vect)                    // rutina de atención de interrupción
{
    cmd_byte = UDR0;                  // leer dato desde el hardware serial
    signal(sem_comando);              // activar a tarea_comandos
}

process tarea_odometria(void)         // Prioridad 15: tiempo de respuesta < 10ms
{
   for (;;) {
        velocidad = leer_encoder();
        calcular_odometria(velocidad);      // demora 3ms

        sleepms(10);
    }
}

process tarea_temperatura(void)       // Prioridad 40: tiempo de respuesta < 5ms
{
   for (;;) {
        temperaturas = leer_sensores_temp();

        PELIGRO = procesar(temperaturas);     // demora 3ms
        if (PELIGRO) {
            accionar_actuadores();
            alarma_ON();
        }

        sleepms(5);
    }
}

process tarea_comandos(void)          // Prioridad 20: tarea intermedia
{
   for (;;) {
        wait(sem_comando);            // bloquea a la espera de un comando
        accionar_comando(cmd_byte);   // demora 10ms
    }
}

void main(void)
{
    sem_comando = semcreate(0);
    resume(create(tarea_odometria, 256, 15, "odom", 0));
    resume(create(tarea_temperatura, 256, 40, "temp", 0));
    resume(create(tarea_comandos, 256, 20, "comm", 0));
}
```


Listing 1.3: Ejemplo de un sistema con eventos asincronicos

3 Patrones tipicos de uso de tareas con un RTOS

Mas alla de los ejemplos anteriores, en la practica, casi todas las tareas de un sistema embebido con RTOS caen en uno de los siguientes tres patrones. Conocerlos permite disenar un sistema de forma adecuada.

3.1 Tarea periodica

La tarea realiza su trabajo y luego ”duerme”(es bloqueada por el RTOS) durante un periodo fijo. Al activarse nuevamente vuelve a ejecutarse. Este patron es muy utilizado para el muestreo de diferentes eventos u observaciones. Por ejemplo, para leer un sensor con una frecuencia definida. O para la actualizaciónde un control PID, o la actualización de una pantalla, o calculos o acciones en momentos periódicos. En el Código Fuente 1.4 se presenta una tarea posible de control PID a 100hz. Cada vez que la tarea finaliza su actividad delega voluntariamente la CPU al RTOS via la llamada sleepms(); El RTOS garantiza que, si la tarea tiene la prioridad adecuada, sea activada a tiempo, independientemente de lo que hagan las demas.

Listing 1.4: Ejemplo de una tarea periódica

```
process control_pid(void)   // tarea a 100hz
{
    for (;;) {
        int error = setpoint - leer_sensor_velocidad();
        int salida = calcular_pid(error);
        aplicar_pwm(salida);

        sleepms(10);
    }
}
```

3.2 Tarea activada por evento

En esta categoría la tarea permanece bloqueada hasta que ocurre un evento. Mientras espera, no consume CPU. El planificador la ignora y ejecuta en modo round-robin (y/o en base a sus prioridades) otras tareas. En cuanto el evento que espera la tarea bloqueada ocurre la tarea se desbloquea y, si su prioridad es la mas alta entre las tareas listas, el planificador le da la CPU inmediatamente. Este patron es ideal cuando la tarea no debe estar activa continuamente, sino unicamente cuando hay trabajo que realizar. Por ejemplo, cuando llegan datos desde un hardware externo, u otra tarea productora genera el evento, etc.

Usualmente, los RTOS proveen, como mecanismo de sincronización para este tipo de tareas, semáforos. El concepto en sí y su mecanismo no se explica en este apunte. Puede leerse sobre los mismos en algun libro sobre conceptos de Sistemas Operativos. En el Código Fuente 1.5 puede leerse una tarea que se activa a través de una rutina de atención de interrupciones. En este ejemplo, el ADC está configurado para generar interrupciones. Cada vez que el hw ADC finaliza una conversión de analógico a digital, genera una interrupción de E/S. El sistema salta automáticamente a ejecutar la rutina de atención de interrupciones, la cual lee el dato desde el HW y activa a la tarea consumidora mediante un semáforo. La tarea consumidora procesa el nuevo dato generado por el ADC y vuelve a bloquearse esperando el próximo evento.

Listing 1.5: Tarea activada por evento

```
sid32 sem_conversion_adc;
int resultado_adc;                  // buffer compartido

/* La siguiente rutina de atención de interrupciones se activa
 * cuando el conversor analógico a digital finaliza de realizar
 * una conversión.
 */
ISR(ADC_vect)
{
    resultado_adc = ADCW;           // colocar el nuevo dato en buffer
    signal(sem_conversion_adc);     // generar evento para desbloquear tarea
}

process adc(void)
{
    for (;;) {
        wait(sem_conversion_adc);   // bloqueada hasta que suceda evento
        procesar_muestra(resultado_adc);
        iniciar_nueva_conversion();
    }
}
```

3.3 Productor-consumidor via interrupciones

Los datos son producidos por interrupciones de hardware en ráfagas, y frecuentemente. La ISR no los puede procesar, así que sólo los deposita en un buffer y activa a la tarea que los procesa (consumidora). La tarea consumidora procesa datos desde el buffer compartido, sin riesgo de perder interrupciones. Este patron es util para perifericos que genera datos a alta velocidad en ráfagas, y que su pro-

cesamiento está a cargo de alguna tarea que eventualmente obtendrá la CPU. La rutina de atención de interrupciones es básica y rapida; el procesamiento pesado queda en la tarea.

En el ejemplo del Código Fuente 1.6 se observa una rutina de atención de interrupciones productora, que obtiene datos de una interfaz serie, y los deposita en un buffer. Además, activa a la tarea consumidora por cada dato que llega, utilizando sincronización mediante un semáforo. Si la tarea que consume, demora en procesar un dato, y llegan nuevos datos, la rutina de atención de interrupciones depositará automaticamente los nuevos datos en el buffer, y activará el semáforo del consumidor tantas veces como datos existan en el buffer. Este ejemplo es muy similar al Código Fuente 1.5. Pero observe que ahora los datos son colocados en un buffer. Si la tarea consumidora no obtiene CPU porque, por ejemplo, alguna otra tarea crítica está en ejecución, los datos no se pierden.

IMPORTANTE: Para que este mecanismo funcione correctamente el buffer debe ser lo suficientemente grande como para albergar a todos los datos recibidos a la máxima frecuencia, hasta que el RTOS ponga a ejecutar la tarea consumidora. Además, en una ráfaga de CPU la tarea consumidora debe ser capaz de procesar y vaciar el buffer. Si el buffer es muy chico se perderán datos (la ISR los descartará). Si el buffer es muy grande consumirá mucha RAM (recurso muy excaso en microcontroladores). Si la máxima frecuencia de producción de datos es mayor a la capacidad de procesamiento en una rafaga (QUANTUM) de una tarea, el problema no es el tamaño del buffer, es de procesamiento.

Listing 1.6: Productor-Consumidor via interrupciones

```
sid32 sem_consumidor;         // inicializado en 0

/* rutina de atención de interrupciones */
ISR(USART_RX_vect) {
    buffer_put(UDR0);
    signal(sem_consumidor);   // activar consumidor
}

process consumidor_uart(void) {
    while (TRUE) {
        wait(sem_consumidor);        // bloqueado hasta que
                                     // existan datos para procesar
        char d = buffer_get();
        procesar_caracter(d);
    }
}
```

3.4 Productor-consumidor

Un variante del caso anterior es un clásico. Son dos tareas, una que produce datos y la otra las que los procesa. En el Código Fuente 1.7 se observa un ejemplo, utilizando las herramientas del RTOS que sincroniza este problema de manera sencilla. Como se observa, el productor no produce mas datos hasta que el consumidor no lo active nuevamente. Es el típico patrón productor-consumidor sincronizados mediante dos semáforos provistos por el RTOS.

Listing 1.7: Sincronizacion productor-consumidor entre tareas

```
#include <xinu.h>

sid32 produced;
sid32 consumed;
int n = 0;        // buffer

void prod(void)
{
    int i;

    for (i=0; i<9999; i++) {
        wait(consumed);
        n++;                // producir n
        signal(produced);
    }
}

void cons(void)
{
    int i,n;

    for (i=0; i<9999; i++) {
        wait(produced);
        procesar(n);          // consumir n
        signal(consumed);
    }
}

void main(void) {
    consumed = semcreate(1);
    produced = semcreate(0);

    resume(create(prod, 256, 20, "pro", 0));
    resume(create(cons, 256, 20, "con", 0));
}
```

4 XINU RTOS en atmega328p

XINU es un sistema operativo academico desarrollado originalmente por Douglas Comer en la Universidad de Purdue, a fines de los anos 70. Desde entonces, ha sido re-escrito y portado a una gran variedad de plataformas de hardware y arquitecturas, y es utilizado, principalmente, como herramienta de investigacion y educacion.

A pesar de que XINU comparte algunos conceptos y alguna terminologia de componentes con UNIX, el diseño interno de XINU difiere completamente de UNIX. XINU es un sistema operativo pequeño (microkernel) con un diseño elegante, cuya estructura interna esta conformada por una jerarquia multinivel de componentes esenciales, pero con las cuales puede luego desarrollarse otras mas complejas. La versión actual de XINU para computadoras complejas tiene soporte para la creacion dinamica de procesos, reserva dinamica de memoria, sistemas de archivos, soporte de red, y funciones de E/S independiente de los dispositivos. También ofrece servicios de comunicacion y sincronizacion entre procesos.

XINU no implementa memoria virtual, ni tiene soporte para hardware de paginacion. En tiempo de ejecucion, la imagen de XINU con un conjunto de aplicaciones se carga completamente en memoria RAM, utilizando un unico espacio de direcciones para todos los procesos. Esto significa que los procesos comparten la memoria, aunque XINU gestiona para cada proceso su propia pila. El código fuente del kernel de XINU contiene aproximadamente 2500 lineas de código en lenguaje C, lo que lo hace completamente explorable en el contexto de una materia de grado. Esta es una de sus principales virtudes academicas: no solo se puede usar como RTOS, sino también entender, modificar y extender su funcionamiento interno.

4.1 XINU para arquitectura AVR (atmega328p)

El port de XINU para arquitectura AVR se ejecuta sobre el microcontrolador atmega328p, un microontrolador de arquitectura Harvard, 32KB de memoria flash y 2KB de RAM. Es el microcontrolador mas popular de la placa Arduino UNO y sus decenas de derivados.

Existen dos versiones para esta arquitectura:

- Versión RTOS (código fuente): *https://github.com/zrafa/xinu-avr-rtos*
- Página web del port para utilizar un atmega328p como una retro computadora UNIX: *http://se.fi.uncoma.edu.ar/xinu-avr/*

4.2 Creacion y finalizacion de tareas

En XINU RTOS, una tarea[2] es una función en C que se ejecuta de forma concurrente con las demas. La función create() crea la tarea en estado suspendido; resume() la coloca en estado "listo para ejecutar". La combinacion de ambas es la forma estandar de ejecutar una tarea. create() retorna un PID (Process ID) que permite referirse a la tarea proceso en llamadas posteriores. El prototipo de la función create() es:

```
// create(función, tam_pila_bytes, prioridad, nombre, nro_args, args...)

pid32 create(void *function, uint16 stksize, pri8 priority,
             char *name, uint8 nargs, ...);
```

Un programa ejemplo completo de creacion, obtencion de PID, y finalizacion de una tarea se puede ver en el Código Fuente 1.8. Además se documenta brevemente otras llamadas al sistema de XINU RTOS aquí debajo:

```
getpid();        // obtención del PID de la tarea
kill(pid);       // termina una tarea por su PID
kill(getpid());  // termina nuestra tarea
suspend(pid);    // suspende una tarea (queda en estado SUSPENDED)
resume(pid);     // reanuda una tarea suspendida
getprio();       // obtiene la prioridad de una tarea
chprio(pid, nueva_prio);  // cambia la prioridad de una tarea en ejecucion
create();        // crea una nueva tarea
resume(pid);     // pone en estado de LISTO a una tarea
suspend(pid);    // se le solicita al kernel XINU que suspenda a la tarea pid
sleep(n);        // la tarea delega la CPU al kernel XINU por n segundos
sleepms(n);      // la tarea delega la CPU al kernel XINU por n milisegundos
send(pid, msg);  // enviar el mensaje msg (un entero) a pid (no bloqueante)
receive();       // espera por un mensaje (system call bloqueante)
recvclr();       // limpia el buffer de recepción (no bloqueante)
wait();          // espera por un semáforo
signal();        // señaliza un semáforo
```

[2]IMPORTANTE: el término usual para entidades ejecutadas en forma independiente y de manera concurrente es ***proceso*** en la literatura de Sistemas Operativos. Lo es tambien dentro de la terminología del sistema operativo XINU tradicional. Sin embargo, en sistemas embebidos utilizando RTOS el término usual es ***tarea***, y es el que utilizaremos en todo este apunte

Listing 1.8: Creación y finalización de una tarea (programa completo)

```
#include <xinu.h>

process prender_led(int n);   // prototipo

void main(void)
{
    pid32 pid;

    pid = create(prender_led, 256, 20, "led", 1, 13);
    if (pid == SYSERR) {
        return;          // hubo un error al crear la tarea
    }

    resume(pid);   // lo coloca en estado LISTO para ejecutar
}

process prender_led(int n)
{
    gpio_on(n);
    // Al retornar de la función, la tarea finaliza automaticamente.
    // También la puede finalizar otra tarea con kill(pid).
}
```

4.3 Tareas periodicas

Las funciones sleep(s) y sleepms(ms) bloquean la tarea durante la cantidad de segundos o milisegundos indicado como argumento. Esto es, la tarea que ejecuta sleelp(s) o sleepms(ms) delega voluntariamente la CPU por el tiempo requerido. Durante ese tiempo, el planificador cede la CPU a otra tareas listas. Cuando el tiempo vence, la tarea pasa al estado LISTO y será ejecutada cuando el planificador lo decida segun su prioridad. Combinada con un bucle infinito dentro de la tarea, sleep() o sleepms() implementa el patron de tarea periodica de forma directa. A diferencia del bucle + ISR, aqui cada tarea tiene su propio plazo, y el planificador garantiza que la de mayor prioridad siempre se activa primero.

En el Código Fuente 1.9 se observan dos tareas periódicas. Una de gran importancia, que es la ventilación de la planta química, y otra tarea menos crítica que reporta mediante una lámpara el estado de la temperatura. Como la tarea crítica se ejecuta cada cierto tiempo, 50ms en este caso, en cuanto sea activada se le otorgará el uso de CPU inmediatamente. Esto se maneja asignando una mayor prioridad a esta tarea. La tarea temperatura se ejecuta mas frecuentemente, y sólo es interrumpida por un momento cuando se active la de ventilación.

Listing 1.9: Dos tareas periódicas con distintas restricciones temporales

```
#include <xinu.h>

/*
 * Ventilación de la planta: tarea crítica a 20hz
 * Prioridad 40
 */
process ventilacion_planta_quimica(void)
{
    for (;;) {
        calcular_ventilación(leer_encoder());

        sleepms(50);    // 50 ms: restriccion critica
    }
}

/*
 * Control de temperatura: tarea a 100hz
 * Prioridad 15
 */
process temperatura(void)
{
    for (;;) {
        int t = leer_sensor_temp();
        if (t > UMBRAL)
            encender_LED();
        else
            apagar_LED();

        sleepms(10);  // 10 ms
    }
}

void main(void)
{
    resume(create(ventilacion_planta_quimica, 256, 40, "vent", 0));
    resume(create(temperatura, 256, 15, "temp", 0));
}
```

4.4 Ejemplo completo: cuatro tareas con prioridades distintas

El ejemplo en el Código Fuente 1.10 integra los tres patrones en un sistema realista: tarea periodica critica, tarea activada por un evento asincronico, y dos tareas periodicas de baja prioridad. Este ejemplo podría ser un punto de partida de un tipico sistema embebido desarrollado utilizando un RTOS.

Listing 1.10: Programa ejemplo completo con cuatro tareas

```
#include <xinu.h>

sid32 sem_uart;
volatile char cmd_byte;

ISR(USART_RX_vect) {             // rutina de atención de interrupcion
    cmd_byte = UDR0;
    signal(sem_uart);
}

process encoder(void) {          // Tarea crítica: tiempo de respuesta < 5ms
    for (;;) {
        calcular_odometria(leer_encoder());
        sleepms(5);
    }
}

process comandos(void) {         // Tarea activada por evento
    for (;;) {
        wait(sem_uart);
        procesar_comando(cmd_byte);
    }
}

process display(void) {          // Actualizar display a 100hz
    for (;;) {
        actualizar_lcd();
        sleepms(10);
    }
}

process led(void) {              // Parpadear un LED a 1hz
    for (;;) {
        led_on();    sleepms(500);
        led_off();   sleepms(500);
    }
}

void main(void) {
    sem_uart = screate(0);
    resume(create(encoder, 256, 40, "encoder",  0));
    resume(create(comandos,256, 30, "comandos", 0));
    resume(create(display, 256, 10, "display",  0));
    resume(create(led, 256, 10, "run",  0));
}
```

El planificador garantiza el siguiente comportamiento en tiempo de ejecucion: la tarea **`encoder`** siempre se activa cuando su periodo de 5ms vence, sin importar que esten haciendo las demás. La tarea **`comandos`** se activa luego de que llega un dato por el serial, con tiempo para cumplir el plazo de 20ms. La tarea **`display`** solo ocupa la CPU cuando las otras dos estan inactivas o bloqueadas. Todo esto sin que el programador deba calcular explicitamente cuanto demora cada función ni en que orden colocarlas en un bucle infinito. La tarea **`led`** tiene la misma prioridad que display, por lo que ambas se ejecutan en modo round-robin.

4.5 Instrucciones para utilizar el código fuente de XINU RTOS

Para descargar la versión actual del código fuente puede utilizar git:

```
git clone https://github.com/zrafa/xinu-avr-rtos
```

XINU RTOS presenta los siguientes directorios de código fuente:

```
config/Configuration  : archivo de configuración general
include/              : cabeceras del kernel de XINU RTOS y sus bibliotecas
lib/                  : código fuente de las bibliotecas de XINU RTOS
system/               : código fuente del kernel de XINU RTOS
main/                 : código fuente de la aplicación embebida
```

En general, el único directorio donde se trabaja es debajo de `main/`.

Esta versión de XINU RTOS trae un gestor de memoria dinámica, gestor de tareas (procesos), un driver del timer 2 que el planificador de CPU del kernel de XINU utiliza para obtener la CPU en intervalos regulares. También para gestionar las tareas que delegaron voluntariamente la CPU via los system calls sleep() o sleepms() (y activarlas cuando sea necesario).

El sistema trabaja en base a prioridades. El planificador pone a ejecutar siempre a la tarea de más alta prioridad. Si existen varias tareas con la misma prioridad el planificador utiliza el algoritmo round-robin para asignar la CPU a cada tarea por turnos.

Cada tarea utiliza la CPU durante un milisegundo (esto es configurable modificando el simbolo QUANTUM). Luego, el timer interrumpe la CPU, y el kernel de XINU obtiene el control de la CPU. Si no existe otra tarea listao para ejecutar con la prioridad necesaria, entonces el kernel de XINU asigna la CPU nuevamente a la tarea que la estaba utilizando.

Cuando una tarea desea voluntariamente delegar la CPU por un lapso de tiempo puede utilizar los system calls `sleep() o sleepms()`. El kernel XINU pone a esa tarea en estado DURMIENDO, y le asigna la CPU a la tarea de mayor prioridad en estado de LISTO.

Si no existe ninguna tarea en estado de LISTO, entonces el kernel de XINU le asigna la CPU a la tarea `null`.

En total, existen aproximadamente 1200 bytes de memoria disponible para las tareas y aplicación embebida. `system/initialize.c` crea la tarea main, y le asigna 256 bytes de esa memoria libre.

Configurar XINU RTOS

```
cd xinu-avr-rtos/compile
```

Editar el Makefile en ese directorio, y seleccionar una de las dos opciones que especifiquen su plataforma:

```
opcion 1:
# PLATFORM      =       -Dlgt8f328p
# HZ            =       4000000

opcion 2:
# PLATFORM        =       -DATMEGA
# HZ              =       16000000
```

Si su arduino tiene un chip atmega original, descomentar la opción dos (quitar el #, y dejar comentada la opcion 1). Para los que tengan el arduino nano chino de 4Mhz descomentar la opción uno.

El archivo `config/Configuration` define los demás parámetros del sistema.

Compilar y verificar

Para poder compilar XINU RTOS debe instalar los siguientes paquetes:

```
bison
flex
gawk
build-essential
```

(en su Linux pueden llamarse distinto).

La carpeta main/ tiene un programa de ejemplo de varias tareas listo para realizar una prueba.

Estando en el directorio de trabajo xinu-avr-rtos/compile/

```
make clean
make
make flash
```

5 Licencia