Informe de Prácticas

Práctica 4

Funciones e interrupciones

Christopher Carmona
Aitor Salazar

Microprogramación en C,

Febrero, 2024

Índice

1. Contexto general

Las interrupciones son el mecanismo que tienen los microprocesadores y microcontroladores de responder a eventos imprevistos y de organizar las incidencias según su prioridad. Son fundamentales para controlar la comunicación de periféricos externos y facilitan la lectura y escritura de estos. Además, las interrupciones resuelven los problemas que presentan mecanismos con el *polling*, el cual mantiene a la CPU ocupada cumpliendo una única tarea.

La cuarta práctica de laboratorio se centra en el manejo de las interrupciones y las rutinas de servicio, con una aplicación concreta, programar contadores de tiempo. El objetivo de esta práctica será programar a través del uso de interrupciones, en primer lugar un contador de segundos; en segundo lugar, un contador de microsegundos; y, en tercer lugar, mostrar el valor del contador por la pantalla LCD integrada en el Kinetis K40.

2. Problemas a resolver

Se plantean tres ejercicios en los que hacer uso de las herramientas de control de interrupciones que incluye el hardware. Además, en el tercer ejercicio se accederá a la pantalla LCD del micro, lo cual requiere configurar los registros de control del LCD.

2.1. Primer ejercicio: *Funciones*

En primer lugar, el primer ejercicio consiste en definir funciones con diferentes datos de entrada y salida y analizar el código ensamblador generado por el copilador. Dependiendo de los argumentos de entrada y salida, el código ensamblador generado es diferente, al igual que si los parámetros se introducen como dato o por referencia (empleando un apuntador o *pointer*).

```
1 void fun1();
```

Código 1: *Declaraciones de funciones con varios tipos de argumntos de entrada y salida.*

En la sección de las declaraciones que se muestra en el Código 1, con las primeras tres funciones lo único que ha hecho el compilador es crear las referencias a las funciones en la parte superior del archivo. Las últimas funciones en cambio sí que están definidas y por ello se puede ver que se reserva espacio en la pila para realizar sus operaciones en ella. El primero recibe un apuntador como dato de entrada mientras que el segundo recibe el valor en sí. En el código de ensamblador se puede apreciar que en la función `fun4`, lo que se guarda en la pila es precisamente la dirección del dato y cuando va a acceder a los valores de a y b para hacer al suma, toma el valor de a directamente de la pila mientras. Con la función `fun_suma` en cambio, lo que se guarda en la pila son directamente a y b.

Cuando se usa la función, se puede observar que carga los valores en registros y después salta a la dirección de la función, es decir, hace una llamada a la función. El v_3++ se ha añadido para quitar el *warning* de variable no usada. Queda claro que la función usa los registros y la pila.

2.2. Segundo Ejercicio: *Interrupciones*

El segundo ejercicio está divido en dos partes. En primer lugar, se pide programar un contador de segundos de tres formas diferentes. En segundo lugar, se debe añadir un contador de milisegundos al contador de segundos para comprobar la precisión del contador de segundos y ver cómo afecta al funcionamiento del microprocesador. Las tres formas de programar el contador de segundos son:

1. Utilizando el gestor de interrupciones integrado en el propio microprocesador,
2. Gestionando las interrupciones manualmente, y
3. Utilizando funciones que hagan tiempo, sin usar interrupciones.

Las tres formas son válidas y tienen ventajas e inconvenientes.

2.2.1. Contador de segundos

Usando el gestor de interrupciones. El Kinetis K40, como cualquier otro microprocesador, tiene un gestor de interrupciones que se encarga de ejecutar la rutina de servicio correspondiente cuando salta una interrupción. Para que esto suceda, las interrupciones deben estar habilitadas, por lo tanto, Lo primero es habilitar las interrupciones. Se hace poniendo a 1 el bit TIE del CSR. La función `lptmr_interrupt()` se encarga de eso y de activar el *timer* para que empiece a contar. Se configura para que use el reloj LPO de 1 kHz, por lo tanto, habrá 1000 ciclos de reloj por segundo y como el timer incrementa con cada ciclo del reloj, para contar 1 s el valor de comparación debe ser 1000.

```
1  void using_default_manager() {
2
3    lptmr_interrupt();
4
5    /* Para que cuando salte la interrupcion se ejecute la rutina de
      servicio, hay que escribir
6     * la direccion de la funcion en el vector de interrupciones. En C es
      suficiente con poner el
7     * nombre de la funcion en la direccion correspondiente.
8     */
9
10   for (;;)
11   {
12     /* La rutina de servicio incrementa el contador y pone a 1 la
      variable global LPTMR_INTERRUPT,
13       * de esta forma, aqui se detecta ese cambio y se muestra el segundo
      . Despues de mostrarlo, se
14       * vuelve a poner a cero. El tiempo que transcurre al mostrar el
      valor en pantalla y para
15       * activar y desactivar la variable gloabl, hace que hay
      imprecisiones en el mostrado del segundo.
16       */
17     if (LPTMR_INTERRUPT == 1) {
18       printf("Tiempo transcurrido: %d\n", int_counter);
19       LPTMR_INTERRUPT = 0;
20     }
21   }
```

```
}
```

Código 2: *Función que configura el gestor de interrupciones para que salte a la rutina de servicio correspondiente cuando ocurra la interrupción y muestre en pantalla el segundo actual.*

El snippet de que se muestra en el Código 2 corresponde a la función que se ejecuta en la función principal y en ella se configura el gestor y se espera a que se active el *flag* (LPTMR_INTERRUPT) que permite mostrar el contador de segundo en pantalla. Este método ha demostrado funcionar muy bien y es relativamente sencillo de implementar, tan solo requiere escribir una rutina de servicio (ver Código 3), añadirla al vector de interrupciones y habilitar las interrupciones. Sin embargo, el tiempo que transcurre para mostrar el valor en pantalla y para activar y desactivar la variable global hace que haya imprecisiones en el mostrado del segundo.

```
void lptmr_isr(void)
{
  LPTMR0_CSR|=LPTMR_CSR_TCF_MASK;  //Clear LPT Compare flag
  LPTMR_INTERRUPT=1;  //Set global variable
  int_counter++;
  // pasa_segundo = 1;
 // printf("\n\nIn LPT ISR!\n\n");
 // printf("Veces en interrupcion %d \n", int_counter);
}
```

Código 3: *Rutina de servicio que se ejecuta cuando se da la interrupción de segundo.*

Gestionando las interrupciones a mano. La segunda opción es gestionar las interrupciones a mano. Si no se habilitan las interrupciones, el gestor no saltará a ninguna rutina de servicio. Por lo tanto, la configuración se hace igual que con el anterior con la diferencia de no habilitar las interrupciones. En este caso, se debe gestionar la situación del bit TCF del CSR que indica si se ha llegado al valor de comparación manualmente. Este método se puede considerar algo más efectivo que el anterior dado que no se pierde el tiempo en hacer saltar al micro de una dirección a otra para ejecutar la actualización y el mostrado del contador. Sin embargo, en este método hay que modificar manualmente el bit TCF una vez se llega al valor de comparación, y por norma general no es recomendable modificar los valores de los registros manualmente.

```
void managing_manually() {
  lptmr_time_counter();

  /*
   * A diferencia del anterior, este no pone TIE a 1.
   */

  for (;;)
  {
    /* En este caso, se mira directamente el valor del bit TCF del CSR y en cuanto se detecta que se
     * pone a 1 (lo que indica que ha superado el valor de comparacion), se incrementa el contador y
     * se muestra en pantalla. Es importante remarcar que la condicion del if cpmprueba que la operacion
     * AND bitwise entre ambos sea distinta de 0 y no 1 en concreto. El True booleano en esta operacion
```

```
14        * no sera numericamente 1.
15        */
16       if ((LPTMR0_CSR & LPTMR_CSR_TCF_MASK) !=0) {
17         int_counter++;
18         printf("Tiempo transcurrido: %d\n", int_counter);
19         LPTMR0_CSR|=LPTMR_CSR_TCF_MASK;
20       }
21     }
22 }
```

Código 4: *Función que configura el gestor de interrupciones sin habilitarlas y comprueba manualmente si se ha llegado al valor de comparación*

Usando una función de espera. La última alternativa es no usar las interrupciones y hacer esperar al programa la cantidad de tiempo justa para que cuente segundos. Para hacerlo de esta forma se ha usado la función `time_delay` la cual consta de tres bucles vacíos anidados cuyo único objetivo es gastar tiempo de ejecución. Este método ha resultado ser el más ineficaz dado que es difícil clavar el tiempo correcto. Además, este método está sujeto a ser interrumpido por cualquier otra interrupción que no se pueda desactivar, lo cual hace que la duración de cada segundo pueda variar en función de si ha habido alguna interrupción mientras se estaba ejecutando `time_delay` o no.

```
1  void managing_waiting() {
2
3    unsigned int t_d = 1;
4    for (;;) {
5      /* Con este metodo se llama a la funcion time_delay para hacer
       tiempo y cuando termina se muestra el
6        * segundo. Ajustando un poco los valores se ha logrado un
       transcurso de tiempo similar al de 1s.
7        */
8      time_delay(t_d);
9      int_counter++;
10     printf("Tiempo transcurrido: %d\n", int_counter);
11   }
12 }
```

Código 5: *Función que ejecuta `time_delay` para hacer tiempo y después actualiza el contador y lo muestra en pantalla.*

2.2.2. Contador de milisegundos

Una vez se tiene el contador de segundos lo siguiente es programar el contador de milisegundos. Para ello, se ha optado por programar una segunda rutina de servicio que cuenta milisegundos utilizando otro de los contadores integrados en el micro, el PIT0.

Para programar la interrupción de milisegundo se usa una segunda rutina de servicio que se llama desde el mismo vector de interrupción. Se programa de tal forma que la interrupción de milisegundo tenga una prioridad mayor que la de segundo, concretamente, nivel de interrupción 1 para la de milisegundo y 3 para la de segundo. Esto se puede hacer escribiendo el valor de prioridad en el registro NVICIPR correspondiente.

```
1 void lptmr_isr(void)
2 {
3   LPTMR0_CSR|=LPTMR_CSR_TCF_MASK;  //Clear LPT Compare flag
```

```
4   ms_contados[int_counter] = contador_ms;
5   contador_ms = 0;
6   //LPTMR_INTERRUPT=1;  //Set global variable
7   int_counter++;
8 }
9
10 void pit0_isr(void)
11 {
12   PIT_TFLG0 |= PIT_TFLG_TIF_MASK; //Clear LPT Compare flag
13   // Tareas a realizer
14   contador_ms++;
15 }
```

Código 6: *Rutinas de servicio que se usan para el contador de segundos y para el de milisegundos. El contador de segundos registra el valor de ms guardado en un array y resetea el contador de milisegundos.*

Otro aspecto importante es que para que los milisegundos se cuenten correctamente, es crucial que la CPU no esté ocupada haciendo tareas no prioritarias, como por ejemplo, imprimir en pantalla la cantidad de segundos transcurridos por momento o la cantidad de milisegundos contados en el segundo anterior. Después de hacer varios tests, queda claro que si se imprime el tiempo transcurrido por momentos se van perdiendo milisegundos y va empeorando a medida que pasan los segundos. Por ello, en este ejercicio se ha quitado el código que muestra los valores contados por pantalla.

Para guardar la cantidad de milisegundos contados en cada segundo, se usa un array. Para probar el código, se ha programado para que cuente 10 segundos y después de contarlos muestre en pantalla los 10 valores almacenados. Cabe destacar que se ha tenido que ajustar el valor de comparación del contador lejos de su valor teórico. Teniendo en cuenta que el periodo del PIT0 es de 48,45 ns, el valor de comparación debería ser 20639. Sin embargo, con ese valor salían 995 ms por segundo, así que se ha ajustado el valor a 20550 con el cual se consiguen contajes consistentes de 999 o 1000 ms por segundo.

2.3. Tercer Ejercicio: *Periféricos*

En este último ejercicio se pide mostrar el valor del contador de segundos por la pantalla LCD integrada en el micro. Para esto, se utilizan la librería proporcionadas en la documentación de la práctica. La librería contiene 5 archivos; 3 archivos de cabecera donde se declaran los tipos y las funciones y 2 archivos que contienen las definiciones de las funciones que controlan la pantalla y las definiciones de las fuentes que se pueden imprimir en pantalla.

```c
void print_timer_val() {
  STRING my_str = "0000";
  int total_length = 32;
  int str_pos;

  lptmr_interrupt();
  _SLCDModule_Init();
  _SLCDModule_TurnOnClockSign();

  while(int_counter < 10) {
    if (LPTMR_INTERRUPT == 1) {
      _SLCDModule_ClearLCD(NO_ARROWS);
      sprintf(my_str, "%d", int_counter);
      str_pos = total_length - 6 * (int)log10(int_counter);
      _SLCDModule_PrintString(my_str, str_pos);
      LPTMR_INTERRUPT = 0;
    }
  }
  _SLCDModule_ClearLCD(NO_ARROWS);
  _SLCDModule_TurnOffClockSign();
}
```

Código 7: *Con esta función se imprimen valores en el lado derecho de la pantalla LCD.*

Para poder imprimir en pantalla, lo primero es inicializar el LCD, lo cual se hace con la función `_SLCDModule_Init`. Después, se ha programado un bucle que pasa por 30 valores del contador que se imprime en pantalla haciendo uso de la función `_SLCDModule_PrintString`, tal y como se puede ver en el Código 7. Tal y como se hacía para imprimir el valor del contador en la consola, se usa la variable global LPTMR_ INTERRUPT para ejecutar `_SLCDModule_PrintString`. Además, para imprimir el valor del contador en la parte derecha pantalla se calcula la posición del string a imprimir en función de la cantidad de cifras del valor. Como *extra* se activa el símbolo del reloj que tiene la pantalla para indicar que hay un contador activo. Cuando el bucle termina se limpia la pantalla y se desactiva el símbolo del reloj. En la Figura 1 se muestra el resultado obtenido en la pantalla LCD.

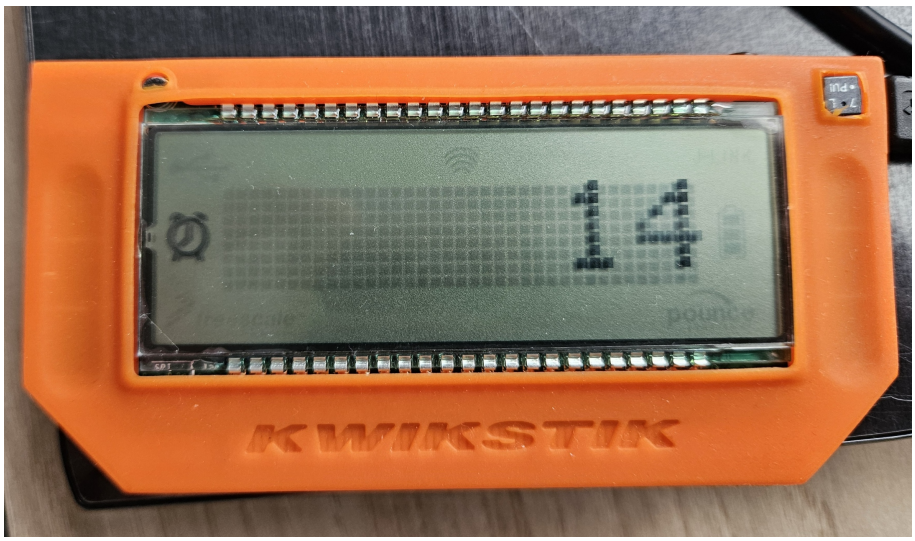

Figura 1: *Pantalla LCD del microprocesador en funcionamiento mostrando el valor instantáneo del contador.*

3. Conclusiones

Por lo general, resulta mas fiable usar métodos de temporización hardware ya que estos no dependen del proceso de ejecución del programa. En caso de necesitar hacer operaciones con dependencia temporal es preferible optar por una reloj hardware. No obstante, para tareas que no precisan de una resolución temporal muy pequeña, como la que estamos haciendo en esta práctica, es suficiente con el uso de temporizadores software. Para tareas RT la variabilidad de los tiempos impediría el correcto funcionamiento de la funcionalidad.

Respecto a la dificultad ha supuesto un salto importante respecto a las anteriores. Mientras que en las anteriores tan solos se pedía analizar el código ensamblador generado por el compilador, en esta se requería obtener un código funcional que fuese capaz de manejar interrupciones y controlar periféricos. Debido a esto, las dificultades que se han ido encontrando han sido más que en las anteriores.

Una de las complicaciones ha sido entender cómo se gestionan las interrupciones en C y en este micro en concreto. El conocimiento previo sobre interrupciones del que disponíamos provenía directamente de la forma en la que se interpretan en ensamblador. En C no es radicalmente distinto pero, ha sido interesante ver cómo se intercala código en el que se ejecutan estructuras condicionales y bucles como se hace en lenguajes de mayor nivel con código en el que se manejan registros directamente bit a bit.

Otro de los puntos más complicados ha sido aprender a utilizar las funciones que gestionan la pantalla LCD. Aunque es cierto que la librería proporcionada contiene funciones que facilitan el uso de la pantalla y sin las cuales habría sido necesario cambiar los valores de los registros del periférico a mano, cabe destacar que la falta de documentación general de cómo se usan ha dificultado el progreso de la práctica. Cierto es que las funciones contienen un breve *docstring* donde se explica brevemente su propósito, sin embargo se concluye que es demasiado breve para su uso académico.

Se concluye que con el uso de este periférico se ha aprendido el proceso de configuración y uso de los diferentes periféricos ya que este ejemplo es extrapolable a otros periféricos.