

Informe de Prácticas

Práctica 5

Lectura de datos de ADC simulado



Christopher Carmona
Aitor Salazar

Microprogramación en C,
Marzo, 2024

Índice

1. Contexto general	2
2. Problemas a resolver	2
2.1. Primer ejercicio: Incluir el fichero	2
2.2. Segundo ejercicio: Captura por milisegundo	2
2.3. Tercer ejercicio: Procesado de datos	3
2.4. Cuarto ejercicio: Visualizar los datos en el emulador	4
3. Conclusiones	5
A. Código de función principal	7

1. Contexto general

Los conversores ADC son los encargados de formar un puente entre el mundo analógico y el mundo digital. El objetivo de estos dispositivos es hacer que las mediciones de magnitudes físicas de interés puedan ser digitalizadas para poder procesar esa información. No es raro encontrar conversores ADC integrados en micros, pues aportan la información necesaria para su correcto funcionamiento. Como ejemplo, se puede tomar información obtenida de un sensor de temperatura para verificar que el *core* en sí trabaja en un rango de temperaturas adecuado. Como es lógico, estos procesos son de vital importancia y deben efectuarse de forma frecuente.

Es por ello que en esta práctica se simulará el manejo de un ADC. Con esta práctica se pondrán aplicar conocimientos como el uso de interrupciones, uso de periféricos como relojes y tratamiento de datos teniendo en cuenta el tipo de dato para maximizar las capacidades del micro.

2. Problemas a resolver

Para hacer esta práctica se deben tener en cuenta que los datos simulados provienen de un documento de cabecera que contiene un arreglo bidimensional que simula las muestras. Las dimensiones de este arreglo van acorde con el número de canales (4) y el número de muestras (128) que se deben procesar.

Los datos obtenidos son de 17 bits con signo incluido, por lo tanto, se pueden leer datos dentro del rango de ± 65535 . El rango de entrada del ADC es de ± 10 V.

2.1. Primer ejercicio: Incluir el fichero

El proceso para incluir el fichero se puede hacer de varias maneras. La primera opción es incluir el archivo de cabecera con los datos (**constantes.h**) en la carpeta de cabeceras del proyecto. De esta forma, solo habría que incluir la cabecera al archivo *main* y ya se podría acceder a ella.

Otra opción es incluir desde *CodeWarrior* la ruta donde se encuentra el archivo de **constantes.h**. Esta suele ser la opción más usada ya que de esta manera no se tienen que copiar archivos al proyecto y los cambios que pudieran ocasionarse podrían referenciarse desde cualquiera de los proyectos que incluya este fichero.

Por último, también existe la opción de insertar el archivo en la carpeta de fuentes e incluir el archivo con las comillas dobles. Esto le da la orden al compilador de que busque la cabecera en el directorio actual así que de forma interna sería lo mismo que la opción anterior.

2.2. Segundo ejercicio: Captura por milisegundo

Una vez se pueda acceder a los datos se deben capturar. El ejercicio pide que se capturen los datos de los 4 canales cada milisegundo. Esto indica que la captura de datos no puede ser un proceso interno de la función *main*, la cual no sabemos concretamente cada cuanto se ejecuta. Para ello se ha hecho uso de la interrupción de milisegundo codificada en la práctica anterior. Mediante esta interrupción se valida la condición de que la captura

se efectúa cada milisegundo. Cada uno de los integrantes del grupo ha implementado el código de captura de forma algo diferente. Por un lado, la primera alternativa llenando un buffer y levantando un flag cuando se alcanza la cantidad de muestras de una captura, tal y como se ve en el bloque de Código 1.

```
1 void pit0_isr(void)
2 {
3     PIT_TFLG0 |= PIT_TFLG_TIF_MASK; //Clear LPT Compare flag
4     for (canal_int = 0; canal_int < 4; canal_int++)
5     {
6         buffer[contador_ms][canal_int]= tabla[contador_ms][canal_int];
7     }
8     contador_ms++; //Aumentar el contador de milisegundos
9
10 #ifdef DEBUG
11     if (contador_ms == 125)
12     {
13         Flag_interrup = 0;
14     }
15 #endif
16     if (contador_ms == 128)
17     {
18         Flag_interrup = 1;
19         contador_ms = 0;
20     }
21 }
```

Código 1: Rutina de interrupción de milisegundo con la funcionalidad

El proceso para obtener la información consiste en rellenar un arreglo bidimensional del mismo tamaño que la tabla, cuyo nombre es *buffer*. Cada vez que se da una interrupción se leen las muestras de los 4 canales y se copian al *buffer*. Una vez el *buffer* esté lleno con las 128 muestras se activa una *flag* que indica que el *buffer* está lleno y este *flag* es leído por la función principal como se observa en la línea 12 del bloque de Código 3.

Por otro lado, la segunda alternativa contempla leer directamente el valor del contador de milisegundos desde la función *main* en lugar de levantar un *flag*. El guardado de las muestras en un *buffer* se hace de forma similar.

2.3. Tercer ejercicio: Procesado de datos

El procesado de datos se hace en la función principal pero cada uno de los códigos que se han implementado tiene sus particularidades. En el Anexo A se muestra una de las dos posibles soluciones.

```
1     memcpy(regist, buffer, sizeof(int) * 128 * 4); // volcar al
2     registro la muestra entera
3     memset(buffer, 0, sizeof(int) * 128 * 4); // Limpiar el
4     buffer
5
6     for (canal = 0; canal < 4; canal++)
7     {
8         for (muestra = 0; muestra < 128; muestra++)
9         {
10            if (regist[muestra][canal] > max[canal])
11            {
12                max[canal] = regist[muestra][canal];
13            }
14        }
15    }
```

```

11     }
12     if (regist[muestra][canal] < min[canal])
13     {
14         min[canal] = regist[muestra][canal];
15     }
16
17     accumulative[canal] += regist[muestra][canal];
18     quadratic[canal] += toVolts(regist[muestra][canal])* toVolts
(regist[muestra][canal]);
19     if ( muestra == 127)
20     {
21         Flag_interrup= 0;
22     }
23 }
24 }
25
26 for (canal = 0; canal < 4; canal++)
27 {
28     max_f[canal]      = toVolts(max[canal]);
29     min_f[canal]      = toVolts(min[canal]);
30     promedio[canal]   = toVolts((int) (((float) accumulative[canal])
/ 128.0));
31     promedioQ[canal]  = sqrt(quadratic[canal] / 128.0);
32
33     //Limpiar los registros de las variables que se visualizan.
34     max[canal]        = 0;
35     min[canal]        = 0;
36     accumulative[canal] = 0;
37     quadratic[canal]   = 0;

```

Código 2: Instrucciones de procesamiento del programa principal

El objetivo del procesamiento es obtener el valor máximo, mínimo, media aritmética y media cuadrática de todos los canales. Para aprovechar el tiempo de ejecución del hardware se manipulan los datos en formato *int* exceptuando el caso de las medias cuadráticas, pues el valor cuadrático de algunos casos excede la longitud de bits de un *int*.

Como primer paso en la ejecución se hace un volcado de lo recibido en el *buffer* a un registro de las mismas dimensiones llamado *regist*. En caso de ser un ADC real el contenido del *buffer* podría cambiar durante la obtención de estos datos. De esta manera se asegura que se está procesando el último conjunto de datos recibido. Una vez volcado se limpia el *buffer* para una siguiente lectura. A partir de aquí se procesa el arreglo bidimensional *regist*. Tras hacer todas las instrucciones necesarias para obtener los datos, el resultado se puede ver en los arreglos *max_f*, *min_f*, *promedio* y *promedioQ*, los cuales son de tamaño 4, uno por canal. Para finalizar se limpian los arreglos intermedios que se usan en el procesamiento.

2.4. Cuarto ejercicio: Visualizar los datos en el emulador

El resultado de los cálculos debe comprobarse mediante el emulador. Para conseguir esto se ha lanzado una ejecución de depurado y se ha observado el resultado de los cálculos hechos en la ventana de variables del editor. El resultado se puede observar en la Figura 1.

Para comprobar que los resultados obtenidos eran correctos se ha optado por hacer





▼  max_f	0x20007f94	0x20007f94
(x)= [0]	0.175479	0x20007f94
(x)= [1]	0.196841	0x20007f98
(x)= [2]	2.10575	0x20007f9c
(x)= [3]	8.28565	0x20007fa0
▼  min_f	0x20007f84	0x20007f84
(x)= [0]	-0.129702	0x20007f84
(x)= [1]	-0.56611	0x20007f88
(x)= [2]	-2.10575	0x20007f8c
(x)= [3]	-8.28565	0x20007f90
▼  promedio	0x20007f74	0x20007f74
(x)= [0]	0.02288853	0x20007f74
(x)= [1]	-0.184634	0x20007f78
(x)= [2]	0.0	0x20007f7c
(x)= [3]	0.0	0x20007f80
▼  promedioQ	0x20007f64	0x20007f64
(x)= [0]	0.110299	0x20007f64
(x)= [1]	0.326884	0x20007f68
(x)= [2]	1.48898	0x20007f6c
(x)= [3]	5.85884	0x20007f70

Figura 1: Ventana de variables del editor de textos *CodeWarrior*

el mismo calculo en un documento de Excel y compara el resultado. El resultado ha sido satisfactorio pues los valores numéricos del Excel coinciden con los de la ventana de variables con punto de ruptura en la línea 58 del Código 3.

3. Conclusiones

Las dos implementaciones que se han realizado son muy similares y distan muy poco una de otra. Algunas de las diferencias a destacar son las siguientes. En primer lugar, en un caso se lee directamente el valor del contador de milisegundos para iniciar el procesamiento de los datos, mientras que en el otro se levanta un flag. En segundo lugar, en la opción presentada en el informe se hace un volcado del buffer de lectura en otro en el momento en que se tiene la captura completa, mientras que en la otra alternativa los valores se leen directamente del buffer de lectura una vez se lea la última muestra que forma una captura. A pesar de que el segundo corra el riesgo de leer los datos mientras estos puedan cambiar, se ha visto que el procesamiento ocurre lo suficientemente rápido para que esto no ocurra por lo que no supone un problema. Por lo demás, las dos implementaciones funcionan correctamente y no han mostrado ventajas una frente a la otra.

Un aspecto importante a destacar es la relevancia de las interrupciones a la hora de comunicarse con un periférico. La recepción de datos a un ritmo constante implica utilizar interrupciones para poder disponer del microprocesador en el programa principal, pues en caso de no usar las interrupciones lo siguiente sería recibir datos mediante *polling*, que dejaría la CPU inaccesible. Esta operación no se podría coordinar mediante tiempos de espera por la dependencia de muchos factores que afectan al tiempo de ejecución.

La importancia de conocer el micro y qué instrucciones tiene integradas es otra de las conclusiones evidentes obtenidas. Se ha probado a manipular los datos como flotante y

aun que para esta funcionalidad concreta sea suficiente para ejecutar correctamente el algoritmo, debido a los grandes tiempos entre interrupciones, sabemos que el micro hace uso de funciones de librerías lo cual empeora considerablemente el tiempo de ejecución.

A. Código de función principal

```
1 int main(void)
2 {
3
4     //Configuracion de los timers
5     pit_interrupt_config();
6
7     int max[4] = { 0, 0, 0, 0 };
8     int min[4] = { 0, 0, 0, 0 };
9     int accumulative[4] = { 0, 0, 0, 0 };
10    float quadratic[4] = { 0, 0, 0, 0 };
11    float max_f[4];
12    float min_f[4];
13    float promedio[4];
14    float promedioQ[4];
15    canal = 0;
16    int muestra = 0;
17
18    while (1)
19    {
20        if (Flag_interrup == 1)
21        {
22            memcpy(regist, buffer, sizeof(int) * 128 * 4); // volcar al
23            registro la muestra entera
24            memset(buffer, 0, sizeof(int) * 128 * 4); // Limpiar el
25            buffer
26
27            for (canal = 0; canal < 4; canal++)
28            {
29                for (muestra = 0; muestra < 128; muestra++)
30                {
31                    if (regist[muestra][canal] > max[canal])
32                    {
33                        max[canal] = regist[muestra][canal];
34                    }
35                    if (regist[muestra][canal] < min[canal])
36                    {
37                        min[canal] = regist[muestra][canal];
38                    }
39
40                    accumulative[canal] += regist[muestra][canal];
41                    quadratic[canal] += toVolts(regist[muestra][canal])* toVolts
42                    (regist[muestra][canal]);
43                    if ( muestra == 127)
44                    {
45                        Flag_interrup= 0;
46                    }
47                }
48            }
49
50            for (canal = 0; canal < 4; canal++)
51            {
52                max_f[canal] = toVolts(max[canal]);
53                min_f[canal] = toVolts(min[canal]);
54                promedio[canal] = toVolts(((int) (((float) accumulative[canal])
55                / 128.0)));
56                promedioQ[canal] = sqrt(quadratic[canal] / 128.0);
```

```

53
54     //Limpiar los registros de las variables que se visualizan.
55     max[canal]      = 0;
56     min[canal]      = 0;
57     accumulative[canal] = 0;
58     quadratic[canal]  = 0;
59 }
60
61 }
62 }
63 return 0;
64 }

```

Código 3: *Función principal del programa.*