FCEFyN Facultad de Ciencias Exactas Físicas y Naturales

UNC Universidad Nacional de Córdoba

UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS y NATURALES

Trabajo Final Electrónica digital III

Autores:

Matrícula	Apellidos y Nombres	Mail
43553345	Joaquin Otalora Leani	joaquin.otalora@mi.unc.edu.ar
43421890	Federico Richter	federico.richter@mi.unc.edu.ar
43173858	Santiago Recalde	santi.recalde@mi.unc.edu.ar

Índice

Introducción

En el marco de la materia electrónica digital 3 se abordan temáticas relacionadas con el desarrollo de sistemas embebidos utilizando la placa LPC1769 de la marca NXP, sus periféricos y características y escribiendo Software en C utilizando la IDE Mcuxpresso y los Drivers correspondientes a las librerías de CMSIS (Arm's *Common Microcontroller Software Interface Standard)*. Utilizando estas herramientas como base se realiza un trabajo final integrador el cual, en nuestro caso, consta de una herramienta capaz de realizar las tareas de un osciloscopio, medir la frecuencia de una señal de entrada y generar una onda cuadrada en uno de sus pines de salida.

A lo largo de este informe se describe la forma de trabajo de cada uno de los periféricos, herramientas utilizadas y el modo de uso.

1 Desarrollo

1.1 Concepto de funcionamiento general

El proyecto en sí consta de una serie de herramientas utilizadas en mediciones de electrónica: un osciloscopio, un medidor de frecuencia y la generación de una onda de calibración cuadrada.
En un primer lugar, se hace uso del ADC para poder muestrear los datos a través del canal 0 del mismo y luego guardarlos en uno de los dos buffers que contienen las muestras de la señal, los cuales funcionan en modo tic toc, ya que, mientras se ingresan datos en uno de ellos, desde el otro se leen para enviarlos al puerto UART. Es por esto que, al mismo tiempo que el ADC, el DMA lee los datos de uno del buffer ya escrito por el ADC y los envía a la cola FIFO de transmisión de manera continúa y en paralelo. Luego es el módulo de comunicación UART quien enviará los datos en paquetes de 8 bits a través de una interfaz USB hacia una PC, donde se reconstruirá la señal.

Con respecto a la funcionalidad de cálculo de frecuencia, se usa el ADC también, pero esta vez para calcular el promedio de la señal entrante para así verificar cada vez que se cruza este mismo de manera ascendente, es decir, cuando la señal pasa de estar en un punto por debajo del promedio a estar por encima del mismo. Luego, haciendo uso de uno de los timers, se cuenta el tiempo transcurrido entre cruce y cruce para calcular la frecuencia de la señal de entrada. Utilizando otro de los timers y los pines de salida GPIO, se muestra por una serie de displays de 7 segmentos la frecuencia estimada.

Por último, para la generación de la onda cuadrada, se utiliza otro de los timers y uno de sus pines de salida de match en modalidad toggle. Para esto se setea una determinada frecuencia a la que el valor de salida cambiará de estado alto a bajo y viceversa. Con el uso de un pulsador el cual generará una interrupción externa a través de uno de los pines, se podrá duplicar la frecuencia de salida.

1.2 Presentación de funciones

Configuraciones de la placa:

1. **configADC:** utilizada para configurar el canal 0 del Analog to Digital Converter incluído en la placa de desarrollo.
2. **configDMA:** para la configuración de Direct Memory Access, declarando las Linked List
3. **configTimer0 ,1 , 2 y 3:** seteo de timers con sus matchs
4. **configGPIO:** configuración de puertos de salida para los displays
5. **configEINT:** configuración de la interrupción externa para el botón de cambio de frecuencia de la onda cuadrada de salida
6. **configUART:** configuración del controlador de UART con sus FIFO de entrada y salida

Funciones de manejo de datos:

1. **calcFrec:** es la encargada de encontrar el valor de la frecuencia de la señal que ingresa por el ADC y guardar su valor en un arreglo de frecuencias.
2. **divFrec:** separa el valor de la frecuencia en sus dígitos y los guarda en un arreglo.
3. **prepArray:** toma tres dígitos del valor de la frecuencia, aquellos significativos para la medición, para mostrarlos por los displays.
4. **frecValue:** determina un valor de frecuencia estable y razonable luego de sucesivas muestras de la misma.
5. **calcProm:** calcula el promedio de un arreglo que contiene los valores de frecuencia adaptados.
6. **saveMinValues:** elimina los valores atípicos máximos del arreglo de frecuencias.

Handlers de las interrupciones:

1. **ADC_IRQHandler:** toma los valores de la conversión, los adapta para su uso, permite el cálculo de la frecuencia y guarda los valores en la memoria correspondiente.
2. **DMA_IRQHandler:** informa el fin de la transferencia de datos de una memoria y cambia de memoria al ADC.
3. **EINT0_IRQHandler:** modifica el valor de la frecuencia de la señal cuadrada de calibración.
4. **TIMER2_IRQHandler:** se encarga de realizar el multiplexado de displays y de generar el valor de frecuencia estabilizado.
5. **TIMER3_IRQHandler:** carga el nuevo valor del promedio de la señal y reinicia las variables de cálculo del mismo.

1.3 Funcionamiento específico

1.3.1 Captación de alterna, guardado y envío de datos:

Para poder captar la señal de entrada alterna se utilizó el canal 0 del ADC, ubicado en el pin 23 del puerto 0, utilizando la máxima frecuencia de muestreo de 200 kHz. Además, se configuran las interrupciones y se enciende el periférico, sin embargo, no será hasta más adelante que se inicie la conversión y se habiliten las interrupciones para evitar problemas de sincronización. Todo esto lo podemos ver en la función **configADC**. Para poder guardar los datos y como ya se ha mencionado anteriormente, se utilizan dos buffers de tamaño estático declarados como arrays donde el ADC irá guardando los datos convertidos pero cortando los 4 bits menos significativos, ya que, luego para la transmisión por UART esto nos dará mayor velocidad sin que se vea alterada significativamente la reconstrucción de la señal.

```
void configADC(){

    PINSEL_CFG_Type pinsel_cfg;
    pinsel_cfg.Portnum = 0;
    pinsel_cfg.Pinnum = 23;
    pinsel_cfg.Funcnum = 1;
    pinsel_cfg.Pinmode = 0;
    pinsel_cfg.OpenDrain = 0;
    PINSEL_ConfigPin(&pinsel_cfg); //AD0.0

    ADC_Init(LPC_ADC,200000);
    ADC_BurstCmd(LPC_ADC,DISABLE);
    ADC_IntConfig(LPC_ADC,ADC_ADGINTEN,ENABLE);

}
```

Luego de esto, en la función **configDMA,** se realizan la declaración de las linked list que tienen como source o fuente de datos uno de los buffers estáticos cada una y como destino la cola FIFO de transmisión del módulo UART. Para lograr esto se configuran las transmisiones desde memoria hacia periférico, incremento de la fuente pero no del destino (para enviar los datos de manera secuencial a UART), transmisiones de un dato con 8 bits de ancho y que generé interrupciones cada vez que se finalice una transmisión, todo esto con los bits de control de las linked list.

```
void configDMA() {

    NVIC_DisableIRQ(DMA_IRQn);
```

```
	// Configura el primer LLI para transmitir array1
	lli0.SrcAddr = (uint32_t)buf0;
	lli0.DstAddr = (uint32_t)&LPC_UART2->THR;
	lli0.NextLLI = (uint32_t)&lli1; // Apunta al segundo LLI
	lli0.Control = (((bufSize) & ~(0x3F<<18)) & ~(1<<27)) | (1 << 26) |
(1<<31);


	// Configura el segundo LLI para transmitir array2
	lli1.SrcAddr = (uint32_t)buf1;
	lli1.DstAddr = (uint32_t)&(LPC_UART2->THR);
	lli1.NextLLI = (uint32_t)&lli0; // Apunta al primer LLI para un bucle
infinito
	lli1.Control = (((bufSize) & ~(0x3F<<18)) & ~(1<<27)) | (1 << 26) |
(1<<31);

	GPDMA_Init();
	// Configura el canal DMA
	GPDMA_Channel_CFG_Type GPDMACfg;
	GPDMACfg.ChannelNum = 0;
	GPDMACfg.SrcMemAddr = (uint32_t)buf0; // Buffer que contiene ambos arrays
	GPDMACfg.DstMemAddr = 0; // Transmisión UART
	GPDMACfg.TransferSize = (bufSize); // Tamaño del buffer
	GPDMACfg.TransferWidth = 0; // Ancho de transferencia de 8 bits
	GPDMACfg.TransferType = GPDMA_TRANSFERTYPE_M2P; // De memoria a
periférico (UART)
	GPDMACfg.SrcConn = 0; // Conexión de fuente (0 para memoria)
	GPDMACfg.DstConn = GPDMA_CONN_UART2_Tx; // Conexión de destino para UART2
	GPDMACfg.DMALLI = (uint32_t) &lli1; // El primer LLI
	GPDMA_Setup(&GPDMACfg);

	GPDMA_ClearIntPending(GPDMA_STATCLR_INTTC,0);
	GPDMA_ClearIntPending(GPDMA_STATCLR_INTERR,0);
	LPC_GPDMACH0->DMACCControl = (((bufSize) & ~(0x3F<<18)) & ~(1<<27)) | (1
<< 26) | (1<<31);
}
```

Para el envío de datos se utiliza el módulo de comunicación serial UART 2, y se setea con la función **configUART**. En esta, se comienza configurando los pines de transmisión (P0.10) y recepción (P0.11) de UART, es decir con su función 1. Luego, y haciendo uso de los structs que proporciona la librería de CMSIS, se setean las colas FIFO para que reciba datos desde el DMA y cada vez que haya un dato presente en la FIFO, se comience el envío. Por otro lado se determina el tamaño de paquete de 8 bits, no se agregá bit de paridad y un solo bit de parada. Por último se setea el máximo baud rate al cual puede funcionar la placa, el cual es de 460800 b/seg.

```
void configUART(void){

    PINSEL_CFG_Type PinCfg;
    PinCfg.Funcnum = 1;  // Función 1 para UART
    PinCfg.Portnum = 0;  // Puerto 0
    PinCfg.Pinnum = 10;   // P0.10 para TXD2
    PINSEL_ConfigPin(&PinCfg);
    PinCfg.Pinnum = 11;   // P0.11 para RXD2
    PINSEL_ConfigPin(&PinCfg);

      // Configura el periférico UART
    UART_CFG_Type UARTConfigStruct;
    UART_FIFO_CFG_Type FIFOCfg;

    FIFOCfg.FIFO_DMAMode = ENABLE;
    FIFOCfg.FIFO_Level = UART_FIFO_TRGLEV0;
    FIFOCfg.FIFO_ResetRxBuf = ENABLE;
    FIFOCfg.FIFO_ResetTxBuf = ENABLE;
    UARTConfigStruct.Baud_rate = 460800;
      UARTConfigStruct.Databits = UART_DATABIT_8;  // 8 bits de datos
      UARTConfigStruct.Parity = UART_PARITY_NONE;  // Sin bit de paridad
      UARTConfigStruct.Stopbits = UART_STOPBIT_1;  // 1 bit de parada
    UART_FIFOConfigStructInit(&FIFOCfg);
    UART_Init(LPC_UART2, &UARTConfigStruct);
    UART_FIFOConfig(LPC_UART2, &FIFOCfg);
}
```

Luego de habilitar las interrupciones en el main, tanto de ADC como de DMA, dentro de los handlers de ambos es donde se gestiona el manejo de buffers. Para esto se utilizan una serie de variables globales en la función main y que se modifican dentro de los handlers. La primera de estas es **uartEnd**, la cual se pone en alto cuando se termina la transmisión de datos por DMA hacia UART y luego se pone de vuelta en 0 dentro del main. Por otro lado, la variable **buffer** determina en cuál de los dos se guardan los datos provenientes del ADC y se modifica en la interrupción de DMA (**DMA_IRQHandler**) cada vez que en la interrupción de ADC se guardan 2048 valores (se utiliza la variable **count** y llega hasta el valor que representa el tamaño del buffer dentro de **ADC_IRQHandler**).

```
void ADC_IRQHandler(void){
    uint16_t valor = ((ADC_GlobalGetData(LPC_ADC)>>4)&0xFFF);
    uint8_t valAux = valor >> 4;
    ADC_StartCmd(LPC_ADC,ADC_START_NOW);
    calcFrec(valor);
    if(saveADC){
        if(buffer){
              buf1[count] = valAux;
        }else{
              buf0[count] = valAux;
```

```
        }
        count++;
    }
    if(count == bufSize){
        count = 0;
        saveADC = 0;
    }
}

void DMA_IRQHandler(void){
    uartEnd = 1;
    buffer = (buffer+1)%2;
    count = 0;
    saveADC = 0;
    GPDMA_ClearIntPending(GPDMA_STATCLR_INTTC,0);
    GPDMA_ClearIntPending(GPDMA_STATCLR_INTERR,0);
    NVIC_ClearPendingIRQ(DMA_IRQn);
}
```

Por último en el main se habilita el canal del ADC y las interrupciones tanto de DMA como de ADC:

```
ADC_ChannelCmd(LPC_ADC,0,ENABLE);
NVIC_EnableIRQ(ADC_IRQn);
ADC_StartCmd(LPC_ADC,ADC_START_NOW);
NVIC_EnableIRQ(DMA_IRQn);
```

1.3.2 Cálculo de frecuencia:

El cálculo de frecuencia se realiza cada vez que el ADC termina de convertir, mediante el handler de interrupción del mismo.
Este cálculo se realiza mediante la utilización del timer0, con la configuración dada en **configTimer0**, la cual determina un tamaño mínimo de paso para mejorar la resolución del resultado. En la función **calcFrec** podemos notar su funcionamiento:

```
void calcFrec(uint16_t ADCvalue){ // Se ejecuta cada vez que el ADC presenta un
nuevo valor.
	// Verificamos si la señal está en el semiciclo negativo
	if(ADCvalue<promf){
		stat = 1;
	// Seteamos la variable de estado de semiciclo negativo
	}
	// Verificamos si la señal está en el semiciclo positivo y si vino de uno
negativo
	if((ADCvalue>promf) && (stat==1)){
		crossZero = 1;
	// Variable indica que hubo un cruce por cero
		frecArray[frecArrayIndex]=25000000/(LPC_TIM0->TC);
		frecArrayIndex++;
		frecArrayIndex = frecArrayIndex%frecSize;		// Se reinicia
el índice de frecuencia
		TIM_ResetCounter(LPC_TIM0);
// Reiniciamos el contador
		stat = 0;
	// Limpiamos la variable de estado de semiciclo negativo
	}
	if(ADCvalue>maxVal){
// Encontramos el valor máximo de la señal
		maxVal = ADCvalue;
	}
	if(ADCvalue<minVal){
// Encontramos el valor mínimo de la señal
		minVal = ADCvalue;
	}
	prom = (minVal + maxVal)/2;
// Encontramos el valor promedio de la señal (offset)
}
```

Primero nos encargamos de encontrar un cruce por el valor medio de la señal de forma ascendente, esto con el objetivo de iniciar el cálculo de frecuencia siempre analizando el mismo punto de la onda periódica.
Al realizar el cruce (obviando unos pasos que serán detallados a continuación), el TC del timer0 se reinicia y comienza su cuenta ascendente, esperando al próximo cruce. Al llegar a él, se guarda el dato calculado de frecuencia en un arreglo de 20 datos, con la intención

posterior de obtener el promedio de los mismos y así resultar en un valor de frecuencia más acertado. Posteriormente se reinicia nuevamente el contador y la secuencia se repite.
Por otro lado mediante la variable **crossZero** se le indica al ADC que puede comenzar a guardar los datos en la memoria si es que está en condiciones de hacerlo, esto con la intención de que los datos en las memorias siempre correspondan su inicio al mismo punto de la señal.
Finalmente también se buscan los valores máximos y mínimos de la señal con el objetivo de poder obtener el valor medio de la misma y así determinar el cruce por cero mencionado anteriormente.
Luego, utilizando la interrupción por match0 del timer2 (del cual se puede notar su configuración en la función **configTimer2**), se procede a, contador mediante para actualizar el resultado cada 1 segundo, calcular el valor de la frecuencia a mostrar por display (notar que el mismo timer define el multiplexado de displays; esto se explicará más adelante).
La función que determina el valor de de la frecuencia es **frecValue**:

```
void frecValue(void){
	saveMinValues(); // Prepara un arreglo sin valores atípicos superiores
	calcProm();  // Obtiene el promedio del arreglo con los nuevos valores
}
```

Esta crea un arreglo de 20 posiciones, el cual es manipulado por las otras funciones para obtener el valor de frecuencia.

```
void saveMinValues(void){
	uint8_t index;
	uint32_t maxFrec;
	for(int i = 0; i<frecSize;i++){		// Bufferea el arreglo de frecuencias
		aux[i] = frecArray[i];
	}
	for(int j = 0; j<10;j++){			// Elimina los tres valores más grandes de este arreglo, ya que son datos atípicos
		maxFrec = 0;
		for(int i = 0; i<frecSize;i++){
			if(maxFrec <= aux[i]){
				maxFrec = aux[i];
				index = i;
			}
		}
		aux[index] = 0;
	}
}
```

Esta función le copia a un arreglo auxiliar todos los valores del arreglo de frecuencias para poder trabajar sin modificaciones del mismo. Luego se encarga de buscar los 10 valores más

grandes de este arreglo auxiliar y los elimina del mismo, ya que estos se consideran valores atípicos (luego se explicará a profundidad este detalle).

```
void calcProm(void){ // Calcula el promedio
	uint32_t auxFrec = 0;
	for(int i = 0; i<frecSize;i++){
		auxFrec += aux[i];
	}
	frec = (uint32_t)auxFrec/10;
}
```

Esta función obtiene el valor promedio de los 10 valores que quedaron cargados en el arreglo.

1.3.3 Muestra de la frecuencia por displays:

Utilizando los pines del 0 al 7 del puerto 0, configurados con la función **configGPIO**, y los pines del 0 al 2 del puerto 2, configurados también en la misma función, podremos controlar los displays que muestran los valores de frecuencia. Los primeros 7 corresponden a los leds de los displays. El 0 corresponde al segmento “a”, el 1 al “b” y así sucesivamente. Los otros 3 corresponden al multiplexado de displays.
Utilizamos el timer2 para realizar el multiplexado. Este hará interrupción por match0 cada 5ms, tiempo suficiente como para que no se note el apagado y prendido de los displays.

```
void TIMER2_IRQHandler(void){ // Interrumpimos por MATCH0 cada 5 ms
	if(counterTimer == 200){  // Cada un segundo se ingresa a este cálculo
		frecValue();		// Se obtiene un valor de frecuencia
estable
		divFrec();			// Se descompone el valor de frec
en sus dígitos individuales
		prepArray();		// Se prepara el arreglo de valores de
displays
		counterTimer = 0;
	}
	counterTimer++;
	// Puerto 0 con FIOMASK que solo permite modificar los primeros 8 bits
	// display0 es el menos significativo
	LPC_GPIO2->FIOCLR |= (1<<((display+2)%3));		// Apaga el
display que está prendido
	LPC_GPIO0->FIOPIN = valuesDisp[dispFrec[display]];  // Escribe el número
por el puerto
	LPC_GPIO2->FIOSET |= (1<<(display%3));			// Prende el
próximo display
	display++;
	// Cambia el display para la próximo
	display = display%3;
// Reinicia los valores a máximo dos
	LPC_TIM2->IR |= 1;
}
```

Notamos al principio del handler parte del cálculo de frecuencia mencionado anteriormente.
Lo siguiente es el multiplexado de los displays, donde se apaga el display encendido, se cambian los valores que se sacan por los segmentos y se prende el siguiente display. Los valores a utilizar se obtienen de un arreglo que contiene las configuraciones de encendido de segmentos que forman los números a mostrar. Estos valores están ordenados según justamente el número que representan. Como argumento de este arreglo se selecciona iterativamente uno de los tres valores más significativos del cálculo de frecuencia. Este arreglo de valores significativos se genera en la función **prepArray**:

```
void prepArray(void){
	uint8_t a = 2;		// Variable de límite de dato
	uint8_t trun = 1;		// Variable de estado de escritura
	for(uint8_t i = 5; i>a; i--){	// Carga el array de display
		if((digFrec[i-1] == 0) && (trun) && (i>3)){
			a--;
	// Mueve el límite de dato
		}else{
			dispFrec[i-1-a] = digFrec[i-1];  // El array es cargado
			trun = 0;
	// Modifica la variable de estado
		}
	}
}
```

Esta función se encarga de seleccionar los 3 valores más significativos de un arreglo de frecuencia que se construye a partir del valor calculado de frecuencia con la función **divFrec**:

```
void divFrec(void){ // Descompone el valor de frec en sus dígitos individuales
	uint32_t buf = frec;			// Variable de buffer para no modificar
frec
	for(uint8_t i = 5; i>0; i--){
		if(i == 3){  // Caso del punto, se encuentra entre k y centena
			digFrec[i-1] = 10 + buf/(uint32_t)pow(10,i);// Posición del
punto en valuesDisp
			buf = buf%(uint32_t)pow(10,i); // Muevo la centena a la
decena, posición luego del punto
		}else{
			digFrec[i-1] = buf/(uint32_t)pow(10,i);// Cargo el dígito:
posición = exp de 10
			buf = buf%(uint32_t)pow(10,i);   // Elimino el dato cargado
		}
	}
}
```

Esta función, como se adelantó, se encarga de separar el valor de la frecuencia en sus componentes individuales: centenas de mil, decenas de mil, unidades de mil, centenas y decenas (se ignoran las unidades ya que presentan mucho error). Todos estos valores se guardan en el orden mencionado de mayor a menor en las posiciones del arreglo. Particularmente a la hora de guardar la unidad de mil, se le suma 10 (ya que este valor es el que corresponde mostrar un punto encendido en el display, indicando el cambio de mil a centena), lo que hace que al buscar en el arreglo de configuraciones de displays, la configuración sea la correspondiente al número con punto.

1.3.4 Generación de la señal de calibración:

La señal de calibración se genera mediante el toggle de un pin de match. Específicamente del match1 del timer1. Este toggle se realiza originalmente cada 0.5ms, dando lugar a una onda cuadrada de 1ms de periodo, es decir, de 1kHz de frecuencia.
Cada vez que ocurre una interrupción externa EINT0, este valor de match se actualiza, reduciéndose a la mitad, y por lo tanto, duplicando la frecuencia de la onda generada. Esto aumenta de esta forma hasta 32kHz, donde deja de aumentar y vuelve nuevamente al valor de 1kHz.

```
void EINT0_IRQHandler(void){ // Se modifica el valor de la frecuencia de la
señal de onda cuadrada generada
	if((testingCuadrada*2) > 50000){
		testingCuadrada = 1000;
	}
	else{
		testingCuadrada *= 2;
	}
	TIM_Cmd(LPC_TIM1,DISABLE);
	TIM_ResetCounter(LPC_TIM1);

TIM_UpdateMatchValue(LPC_TIM1,1,(uint32_t)(25000000/(testingCuadrada*2)));
	EXTI_ClearEXTIFlag(EXTI_EINT0);
	TIM_Cmd(LPC_TIM1,ENABLE);
}
```

1.3.5 Representación de la señal:

Una vez que los datos llegan a la PC, con la ayuda de una aplicación escrita en Python se guardan los datos en un buffer y se van ploteando, haciendo uso de la librería matplotlib. Para esto, se leen los datos del puerto, USB al cual se conecta la salida de la placa y se leen de manera consecutiva y constante los datos que van llegando:

```
# Configura el puerto secountrial
ser = serial.Serial('/dev/ttyUSB0', baudrate=460800, timeout=1)  # Reemplaza
'/dev/ttyUSB0' con el puerto USB a UART

def leer_datos_uart():
   global datos_visualizacion, datos_escritura, contador_datos

   while True:
       # Lee datos desde el puerto UART
       data = ser.read(1)  # Lee un byte de datos
         valor = ord(data) if data else 0  # Convierte el byte en un valor
numérico
       valor = valor * (3.3/256)
       # Actualiza el array de escritura
       datos_escritura[contador_datos] = valor
       contador_datos += 1
         # Si el array de escritura está lleno, copia los datos al array de
visualización
       if contador_datos == len(datos_escritura):
           with lock:
                    datos_visualizacion, datos_escritura = datos_escritura,
datos_visualizacion
           contador_datos = 0
```

Luego en otra función y cada un 1 segundo, se actualiza el gráfico con los datos que se encuentran guardados en el arreglo datos_visualizacion.

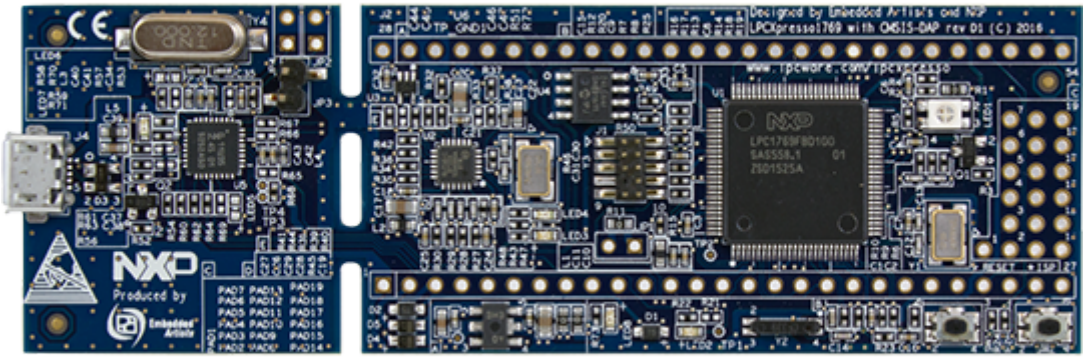
```
def actualizar_grafico():
    global escala,datos_visualizacion,inicio_ventana
    ax.clear()
    with lock:
        y = datos_visualizacion[inicio_ventana:inicio_ventana + escala]
    #ax.plot(x[inicio_ventana:inicio_ventana + escala],y)
    ax.plot(y)
    ax.set_ylim(0, 4)
    ax.set_xlabel('Tiempo')
    ax.set_ylabel('Valor')
    canvas.draw()
```

Además de esto se incluyen las posibilidades de agrandar o aumentar la escala de visualización con un par de botones que funcionan de manera recíproca.

2 Hardware utilizado

2.1 Componentes:

- **LPC 1769** : Comenzando por la placa, es importante remarcar su core de la familia de los ARM Cortex-M3, el cual puede funcionar a una frecuencia máxima de 100 Mhz, además de los periféricos que se encuentran integrados, desde ADC, DMA, UART, DAC hasta los propios GPIO, entre otros.



- **3 x Displays de 7 segmentos:** utilizados para la muestra de la frecuencia
- **Conversor USB-UART:** para leer los datos transmitidos por UART desde la PC
- **3 x Transistores bipolares: BC547-C**
- **6 x Resistencias resistentes**

2.2 Esquemático:

Link al PDF:

Esquemático osciloscopio LPC1769.pdf

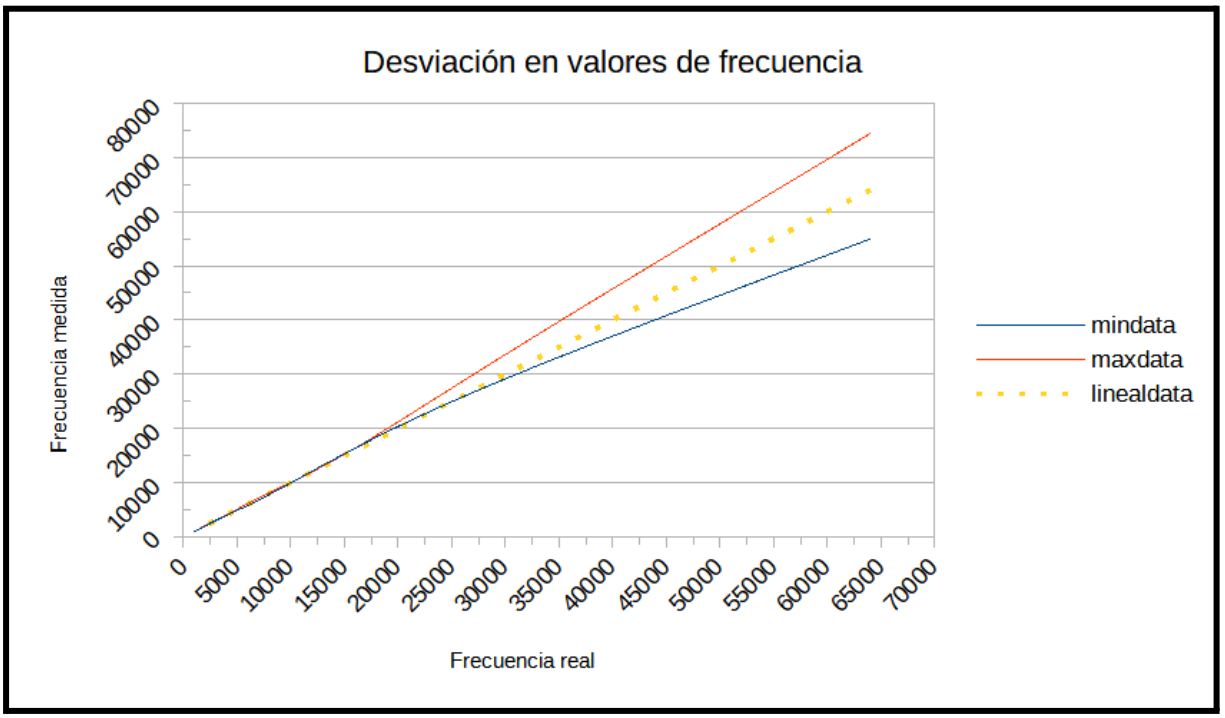
También se adjunta una imagen del mismo al final de este documento.

3 Conclusiones:

3.1 Aspectos a mejorar:

-Se podría mejorar el generador de señales de prueba utilizando el DAC, lo que permitiría probar el comportamiento del osciloscopio en funciones senoidales, triangulares o de cualquier otro tipo de forma. A esta mejora se le podría agregar la funcionalidad de controlar la forma y frecuencia de la señal generada a través de recepción UART.

-Al incrementar la frecuencia, la dispersión de los datos de frecuencia es mayor, por lo que la cantidad de datos "Altos" que son eliminados en la función **saveMinValues()**, es cada vez mayor. Se debe mejorar el tratamiento de los datos de frecuencia para reducir la dispersión y en consecuencia obtener un resultado final más fiel a la realidad.



3.2 Inconvenientes encontrados:

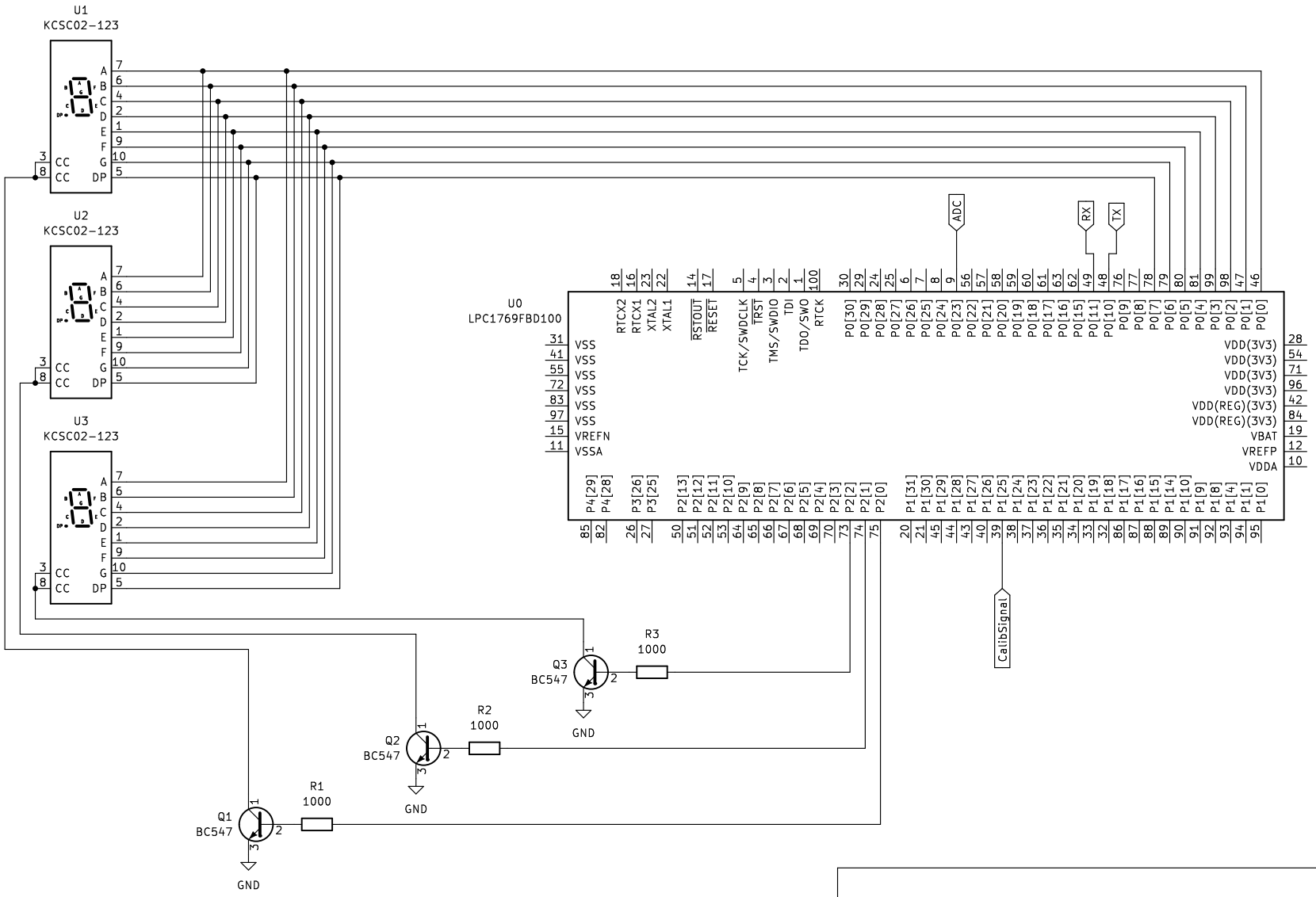
-El principal inconveniente con el que se tuvo que lidiar fue la baja velocidad de transmisión del UART para la aplicación que se está implementado, ya que esta es de un máximo de 46080 datos por segundo, siendo más de 4 veces más lento que el ADC. Para sortear este problema, se utilizaron 2 buffer en configuración tic toc, que permite leer datos mientras se envían otros.

-Otro de los problemas que hubo fue el cálculo de la frecuencia en altas y bajas frecuencias, ya que bajando de 500Hz o superando 50Khz, el resultado ya se alejaba del 10% de error que consideramos tolerable.
La causa del error en bajas frecuencias es de ruido en las lecturas del ADC, que generaba picos en el valor medido. La solución fue la realización de un promedio de 20 muestras de los valores de frecuencia calculados en **calcFrec().**

En el caso del error en altas frecuencias, la causa es de la dispersión de los datos almacenados en el array de 20 muestras utilizado en el promedio. Para solucionar esto se eliminan los valores atípicos superiores y se promedia sobre estos datos.

3.3 Conclusión:

El trabajo fue de utilidad para lograr la comprensión del funcionamiento de la placa en su conjunto, además de tener que utilizar correctamente sus módulos para poder obtener el mejor resultado.
Esto se puede observar claramente en la utilización del módulo DMA en el envío de los datos leídos desde el buffer lleno hacia la PC, mientras que al mismo tiempo el ADC lee y llena el buffer libre.
Se observa la ventaja que da el DMA al permitir multiprocesamiento de los datos en un dispositivo mononúcleo, permitiendo que en la captura de los datos se pierda la menor cantidad posible de estos.



FCEFyN – UNC		
Sheet: /		
File: Esquema ED3.kicad_sch		
Title: Esquemático osciloscopio LPC1769		
Size: A4	Date: 2023-11-23	**Rev:**
KiCad E.D.A. kicad 7.0.9		Id: 1/1