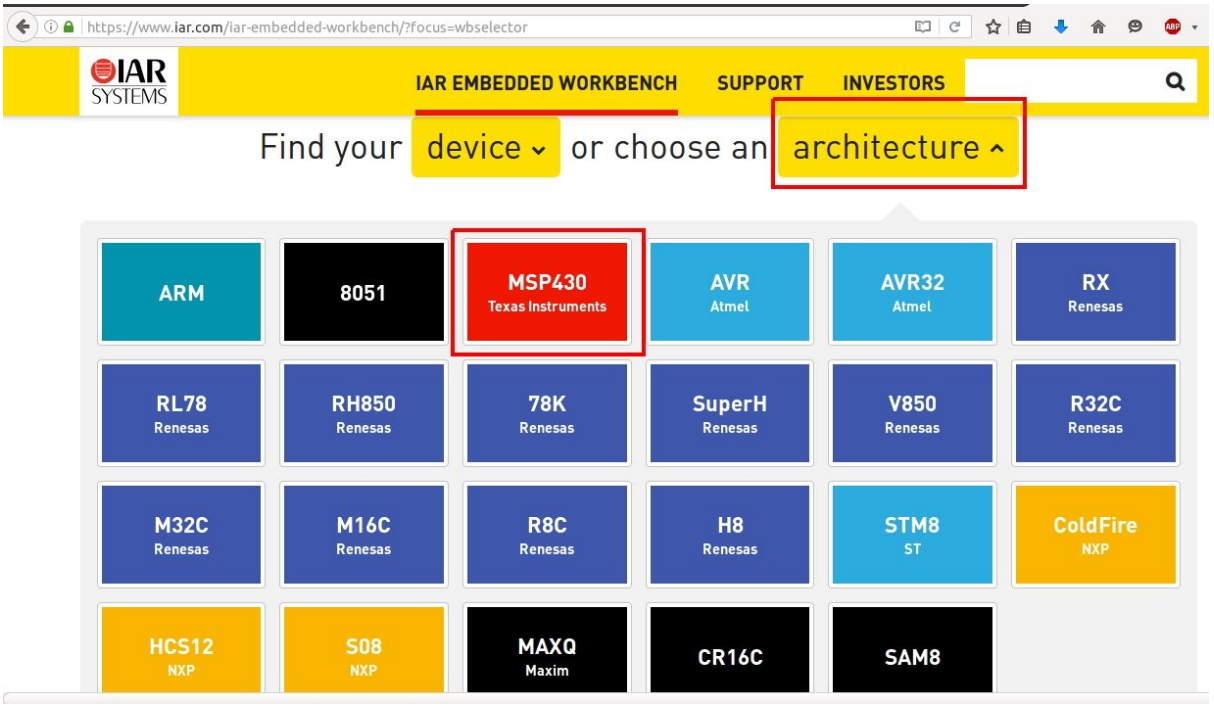
Switch

Software

Se requiere software para:
-Compilar código, descargarlo al procesador y debuggear
Opcion 1 (windows): IAR Embedded Workbench
Ir a https://www.iar.com/ -> Find your tool -> Architecture -> MSP430



O directamente:
http://netstorage.iar.com/SuppDB/Protected/PRODUPD/011269/EW430-6501-Autorun.exe
Instalar versión Kickstart (gratis pero con limitación de tamaño de código de 8 kb)

Opción 2 (windows, linux, mac): Code Composer Studio (CCS)
http://www.ti.com/tool/ccstudio

-Simular procesador + periféricos (señal, display, micro sd)
Proteus (windows): seguir las instrucciones en
http://embeddy.blogspot.com.ar/2010/05/1-how-to-install-proteus-76-sp4.html

-Leer hex (micro sd)
https://mh-nexus.de/en/hxd/ (windows)
http://home.gna.org/bless/ (linux)
http://ridiculousfish.com/hexfiend/ (mac, no lo probé)

IAR y MSP430

MSP430

MSP430 es una familia de microprocesadores low power. De la datasheet (http://www.ti.com/lit/ds/symlink/msp430g2553.pdf)

FEATURES

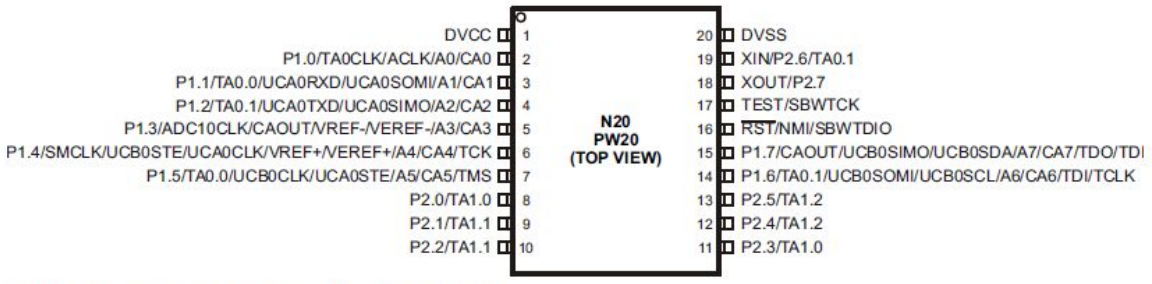
- Low Supply-Voltage Range: 1.8 V to 3.6 V
- Ultra-Low Power Consumption
  - Active Mode: 230 µA at 1 MHz, 2.2 V
  - Standby Mode: 0.5 µA
  - Off Mode (RAM Retention): 0.1 µA
- Five Power-Saving Modes
- Ultra-Fast Wake-Up From Standby Mode in Less Than 1 µs
- 16-Bit RISC Architecture, 62.5-ns Instruction Cycle Time
- Basic Clock Module Configurations
  - Internal Frequencies up to 16 MHz With Four Calibrated Frequency
  - Internal Very-Low-Power Low-Frequency (LF) Oscillator
  - 32-kHz Crystal
  - External Digital Clock Source
- Two 16-Bit Timer_A With Three Capture/Compare Registers
- Up to 24 Capacitive-Touch Enabled I/O Pins
- Universal Serial Communication Interface (USCI)
  - Enhanced UART Supporting Auto Baudrate Detection (LIN)
  - IrDA Encoder and Decoder
  - Synchronous SPI
  - I²C™
- On-Chip Comparator for Analog Signal Compare Function or Slope Analog-to-Digital (A/D) Conversion
- 10-Bit 200-ksps Analog-to-Digital (A/D) Converter With Internal Reference, Sample-and-Hold, and Autoscan (See Table 1)
- Brownout Detector
- Serial Onboard Programming, No External Programming Voltage Needed, Programmable Code Protection by Security Fuse
- On-Chip Emulation Logic With Spy-Bi-Wire Interface
- Family Members are Summarized in Table 1
- Package Options

Resaltado:

- Baja alimentación y consumo
- Reloj interno
- Timer
- SPI (protocolo de comunicación de las tarjetas micro sd, entre otros)
- Conversor analógico-digital de 10 bits

Pinout del procesador (versión 20 pines):

Device Pinout, MSP430G2x13 and MSP430G2x53, 20-Pin Devices, TSSOP and PDIP



NOTE: ADC10 is available on MSP430G2x53 devices only.

Los nombres en cada pin indican las funciones que este puede cumplir (esto debe programarse por software). Esto se detalla en la tabla 2 de la datasheet. Por ejemplo, para el pin 2:

TERMINAL				I/O	DESCRIPTION
NAME	NO.				
	PW20, N20	PW28	RHB32		
P1.0/	2	2	31	I/O	General-purpose digital I/O pin
TA0CLK/					Timer0_A, clock signal TACLK input
ACLK/					ACLK signal output
A0					ADC10 analog input A0(1)
CA0					Comparator_A+, CA0 input

Es decir que se puede utilizar como input/output digital, señal de reloj (TACLK input o ACLK output), input del ADC10, o input del comparador. Por ahora, nos alcanza con saber que cada pin tiene una lista de posibles funciones, que se debe configurar por software que función cumplirá y que esto determinará su comportamiento.

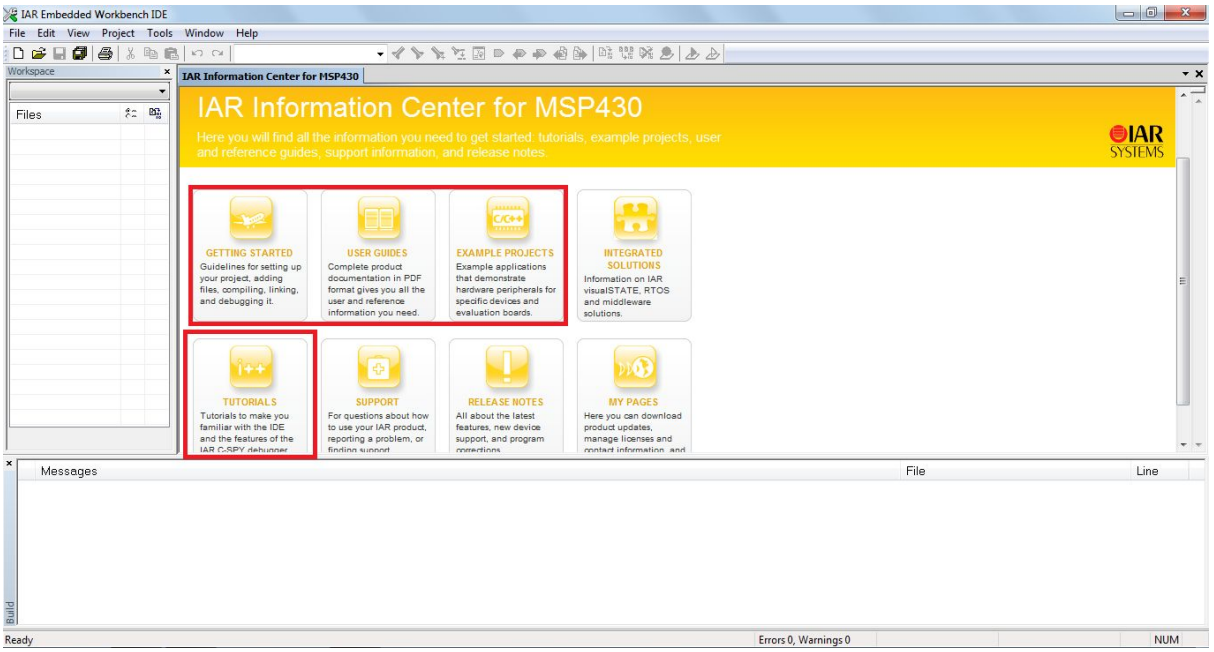
Los dos documentos fundamentales que hay que tener siempre a mano son:

1. Datasheet: http://www.ti.com/lit/ds/symlink/msp430g2553.pdf
2. User Guide: http://www.ti.com/lit/ug/slau144j/slau144j.pdf

IAR

IAR es un entorno de desarrollo integrado que facilita el desarrollo de software para los procesadores. Es como una versión más amigable y user friendly del **gcc** (que usamos para compilar por línea de comandos), que además está desarrollada específicamente para trabajar con procesadores msp430.
El lenguaje de programación es C, es decir que se trabaja con la misma sintaxis, estructura (variables, funciones, main…) de siempre.

Pantalla principal

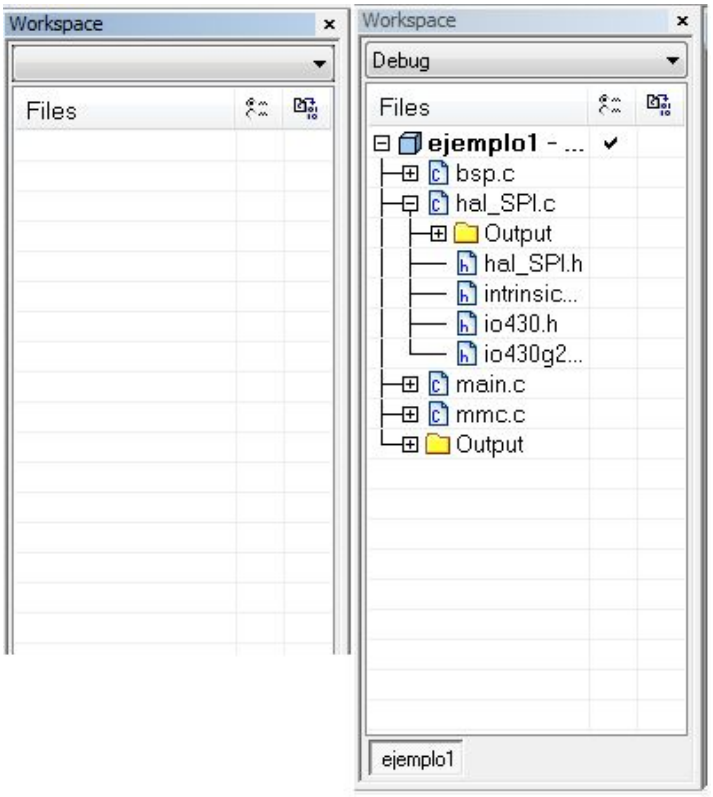


El programa viene con tutoriales y guías incluidos (recomendado el “Getting started”, subido: EW_GettingStarted.ENU.pdf)

El entorno se maneja en términos de “Workspaces” y “Projects”. Un Workspace puede contener muchos proyectos. El proyecto es lo que se descarga al procesador, que contiene uno o varios archivos relacionados (linkeados al compilar).

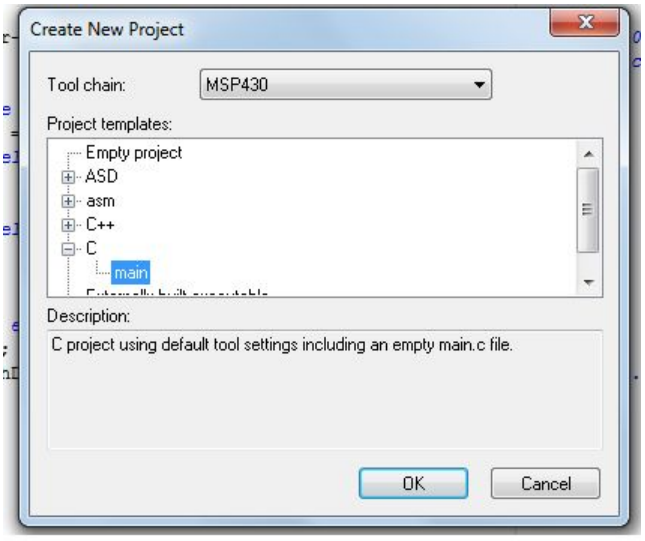
Para empezar a trabajar

Crear un Workspace: File -> New -> Workspace
Guardar el workspace
o abrir un workspace: File -> Open -> Workspace
A la izquierda de la pantalla se ve:

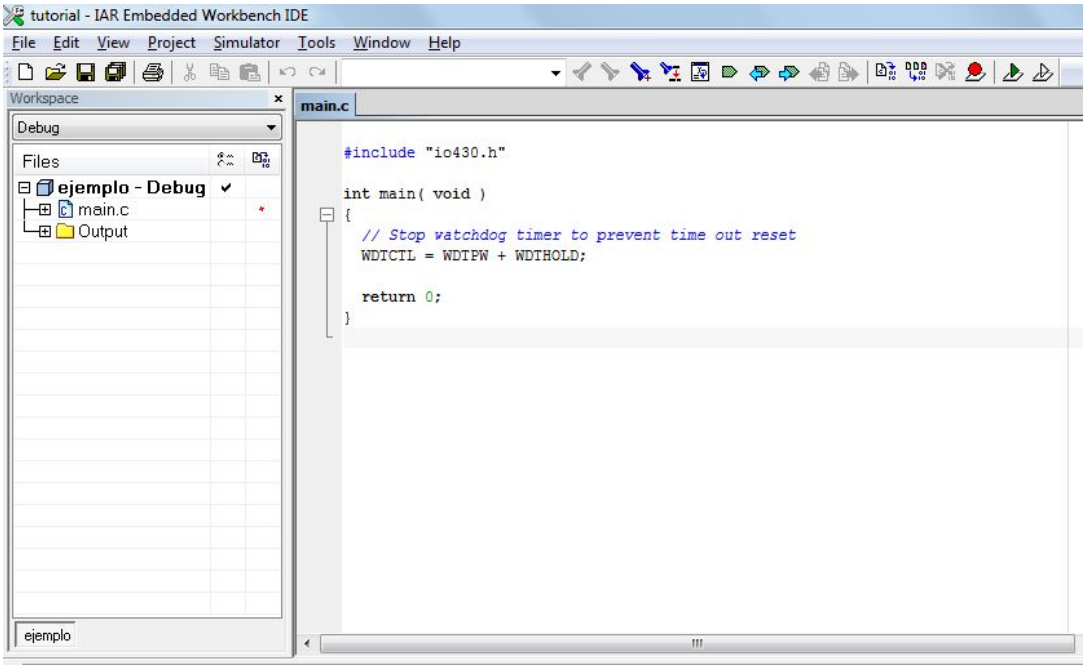


El de la izquierda corresponde a un Workspace nuevo (vacío). El de la derecha a uno que contiene el proyecto “ejemplo1”, que a su vez contiene los archivos bsp.c, hal_SPI.c …

Para crear un proyecto: Project -> Create new project
En la ventana que se abre se elige Tool Chain MSP430 (familia del procesador), C (lenguaje), main



Esto crea un proyecto nuevo, con un archivo *main.c*. El archivo *main.c* es donde va a estar el código principal, todo proyecto debe tener un *main.c*

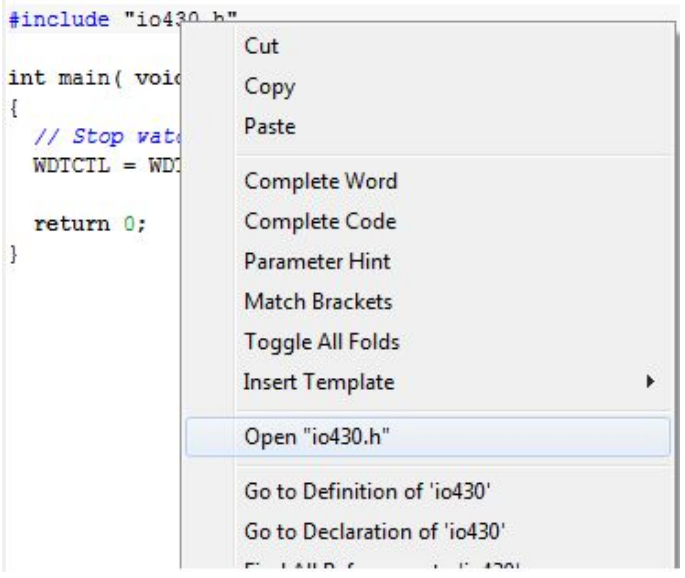


A la izquierda vemos que se creó el proyecto “ejemplo” y contiene el main.c. Haciendo doble click en los archivos a la izquierda se abren en la derecha. En la figura está abierto main.c
La estructura del main.c es la usual:

1) Incluir: #include “...” (librerias, por lo general archivos .h)
2) definiciones, variables, estructuras, funciones
3) función main
4) **Interrupciones**

Algunas diferencias:

- Para ver que contiene un archivo (de los *include*): click derecho sobre el nombre, “Open “...” ”. Se abre el archivo seleccionado. En el caso de variables/registros (más adelante) se puede buscar la definición con “Go to definition of “...” ”



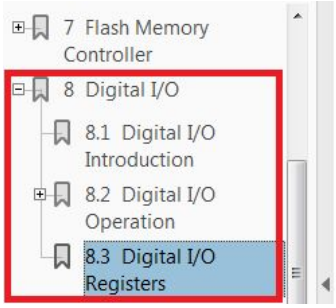
- Siempre se debe incluir el archivo “io430.h”. Este archivo a su vez incluye (según el procesador específico) otro archivo específico de cada procesador, que contiene todos los registro definidos (los nombres de las cosas disponibles)
- **Interrupciones:** la gran diferencia con la programación de siempre. Como adelanto: ciertos eventos pueden hacer que se interrumpa la ejecución de la función main y se ejecute un pedazo de código.
- La línea de código *WDTCTL = WDTPW + WDTHOLD;* es otra que debe estar siempre. Evita que el procesador se resetee automáticamente.

Registros

Una diferencia/dificultad en la programación del procesador es que, además de la programación usual (funciones que operan sobre variables, operaciones, etc), debe programarse cada función específica que se quiera realizar. Esto se hace mediante *Registros de bits*.

Los registros son bloques de bits con nombres específicos. Parte de la programación consiste en dar valores específicos a estos bloques (activar ciertos bits), lo que determina cierto comportamiento del procesador. Hay registros que controlan el reloj interno, timers, el conversor analógico-digital, etc.

Por ejemplo, para que el Pin 2 se comporte como salida lógica y tome el valor ON (ver en la figura anterior que el Pin 2 puede configurarse como I/O digital), se deben incluir las siguientes líneas:
P1DIR_bit.P0 = 1;
P1OUT_bit.P0 = 1;
Para saber qué registros se deben configurar y como para cada función, hay que recurrir a la User Guide. En cada capítulo hay una sección con los registros pertinentes. En este caso, en el capitulo “Digital I/O”:



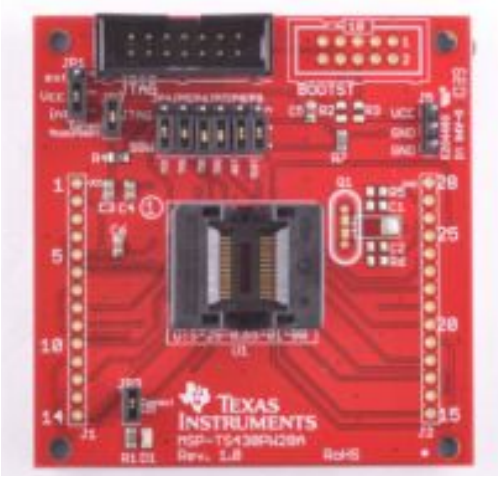
8.3 Digital I/O Registers

The digital I/O registers are listed in Table 8-2.

Table 8-2. Digital I/O Registers

Port	Register	Short Form	Address	Register Type	Initial State
P1	Input	P1IN	020h	Read only	-
	Output	P1OUT	021h	Read/write	Unchanged
	Direction	P1DIR	022h	Read/write	Reset with PUC
	Interrupt Flag	P1IFG	023h	Read/write	Reset with PUC
	Interrupt Edge Select	P1IES	024h	Read/write	Unchanged
	Interrupt Enable	P1IE	025h	Read/write	Reset with PUC
	Port Select	P1SEL	026h	Read/write	Reset with PUC
	Port Select 2	P1SEL2	041h	Read/write	Reset with PUC
	Resistor Enable	P1REN	027h	Read/write	Reset with PUC

Programación y depuración en Launchpad



El launchpad + debugger permiten programar y testear aplicaciones en el procesador (otra alternativa sería usar un software simulador). Se coloca el procesador en el zocalo central, los pines laterales están conectados a los pines del procesador (del lado izquierdo los pines 1-10 conectan con 1-10 del procesador, del derecho 19-28 conectan con 11-20).
En el conector de 14 pines (arriba izquierda) se conecta el FET-Debugger, que a su vez se conecta por USB a la computadora y hace de interfaz para poder programar y testear en el procesador.

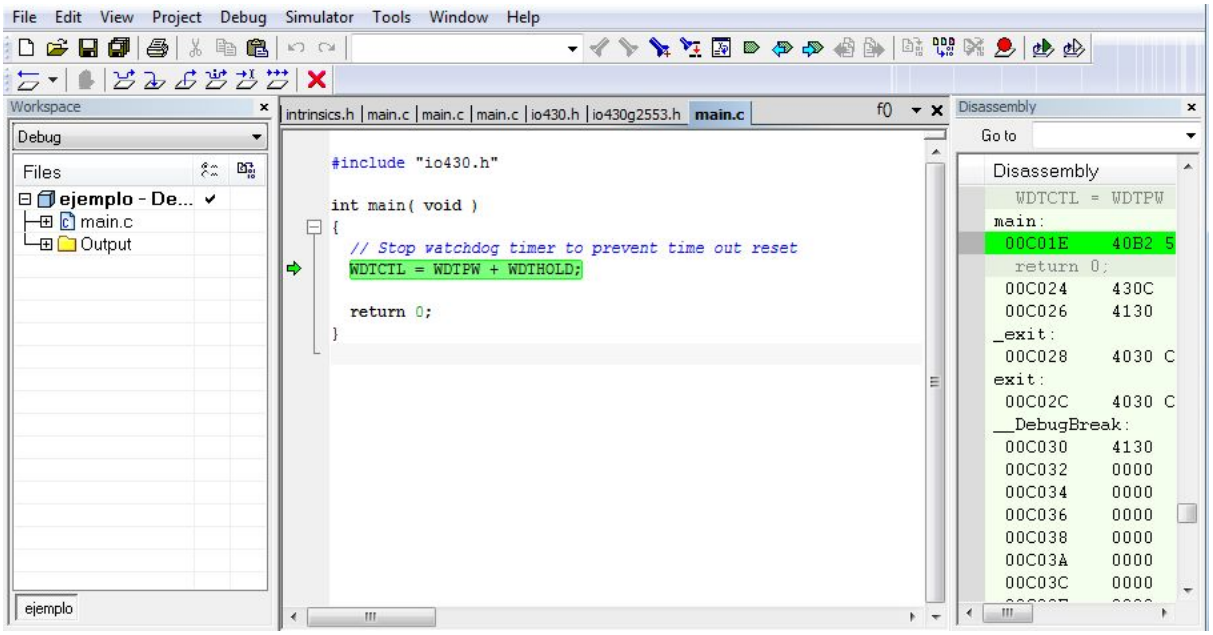
Descargar un programa

Dos configuraciones básicas para trabajar en launchpad: **Project -> Options**

- En la categoría“General Options”, pestaña “Target” hay que elegir el procesador específico en “Device”
- En la categoría “Debugger”, pestaña “Setup” hay que elegir FET-Debugger en “Driver”.

Antes de descargar, hacer Save All. Luego Project -> Rebuild All (compilar y linkear todos los archivos).

Para descargar la aplicación y probarla (debugging) se hace Project -> Download and Debug o Ctrl+D. Se debe abrir una ventana asi:



En esta ventana vemos a la izquierda el proyecto, a la derecha una ventana nueva, y en el centro el código, con la diferencia de que ahora aparece la flecha verde y la línea resaltada que indican en qué línea de código está el procesador en el momento. Se puede hacer que avance de a una línea o hasta cierto punto, y controlar qué valores toman los registros y variables en el proceso, para chequear que ande todo bien. En la próxima sección va esto más en detalle.

Una vez que la aplicación está testeada se sale de la sesión de Debugging con la cruz roja en el menú.
Para descargar los archivos sin debuggear: Project -> Download -> Download active application
Es recomendable borrar primero la memoria del procesador: Project -> Download -> Erase memory
Con la aplicación ya descargada el software resetea automáticamente el procesador (el código no se ejecuta). Cuando se alimenta nuevamente el procesador (desconectar y conectar usb o alimentación externa) comienza a ejecutarse el código.

Hello World y debugging

Tedioso pero necesario, cuando el código falla (o antes), está bueno revisar que el programa está haciendo lo que uno espera. Lo bueno de estos procesadores es que permiten hacer debugging en el mismo procesador.

Usamos como ejemplo el programa más básico: prender un led. El launchpad trae incorporado un led en P1.0 (Pin 2). Este puerto se puede usar como Output digital, por lo que para prender el led hay que hacer que este puerto tome valor ON o HIGH (3.3V).

Acá hay que empezar a leer con cuidado el capítulo de la User Guide de *Digital I/O.*
Queremos configurar P1.0 como output digital con valor high (prendido). En este caso los registros relevantes son:

8.2.5 Function Select Registers PxSEL and PxSEL2

Port pins are often multiplexed with other peripheral module functions. See the device-specific data sheet to determine pin functions. Each PxSEL and PxSEL2 bit is used to select the pin function - I/O port or peripheral module function.

Table 8-1. PxSEL and PxSEL2

PxSEL2	PxSEL	Pin Function
0	0	I/O function is selected.
0	1	Primary peripheral module function is selected.
1	0	Reserved. See device-specific data sheet.
1	1	Secondary peripheral module function is selected.

Es decir que para configurar como I/O hay que setear PxSEL y PxSEL a 0 para elegir la función I/O

8.2.3 Direction Registers PxDIR

Each bit in each PxDIR register selects the direction of the corresponding I/O pin, regardless of the selected function for the pin. PxDIR bits for I/O pins that are selected for other functions must be set as required by the other function.

Bit = 0: The port pin is switched to input direction

Bit = 1: The port pin is switched to output direction

PxDIR a 1 para que sea un output.

8.2.2 Output Registers PxOUT

Each bit in each PxOUT register is the value to be output on the corresponding I/O pin when the pin is configured as I/O function, output direction, and the pullup/down resistor is disabled.

Bit = 0: The output is low

Bit = 1: The output is high

PxOUT a 1 para que tome valor high.

Ahora tenemos que especificar cual es el puerto al que nos referimos.
Primero hay que diferenciar entre los puertos 1.x y los puertos 2.x. Para cada uno tenemos un registro distinto:

P**1**SEL, P**1**SEL2, P**1**DIR, P**1**OUT se refieren a puertos 1.x
P**2**SEL, P**2**SEL2, P**2**DIR, P**2**OUT se refieren a puertos 2.x

En general, cada registro consiste en un número de bits. Los registros que refieren a puertos son 8 bits, uno para cada puerto específico (P1.0 a P1.7; P2.0 a P2.7), que pueden ser configurados independientemente.
Por ejemplo:
P1SEL = 0010 0001
El primer bit (desde la derecha) corresponde al primer pin del puerto 1, es decir, P1.0. El segundo a P1.1, etc.
En este ejemplo P1SEL vale 1 para los puertos 1.0 y 1.5, y vale 0 para los demás.

Para prender el led necesitamos que P1DIR y P1OUT valgan 1 para 1.0 y 0 para todos los demás, y P1SEL y P1SEL2 valgan 0 para 1.0.

	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7
P1SEL	0	0	0	0	0	0	0	0
P1SEL2	0	0	0	0	0	0	0	0
P1DIR	1	0	0	0	0	0	0	0
P1OUT	1	0	0	0	0	0	0	0

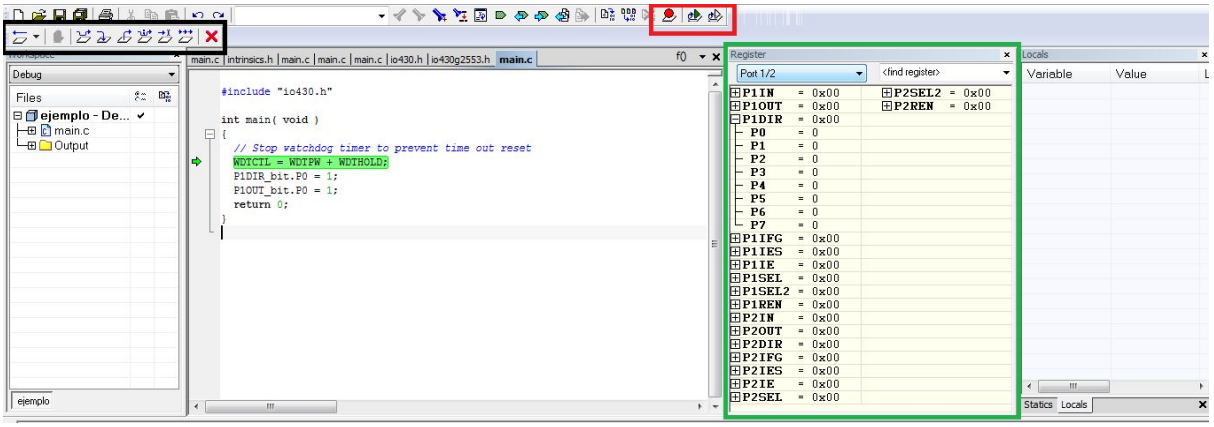
Comentario: *los registros vienen (al encender) con valor 0, por lo que no hace falta asignar explícitamente valor 0.*

Hay varias formas de darle valores a registros. La más cómoda para mí es la siguiente:
P1DIR_bit.P0 = 1;
Es decir: registro_bit.PX = Y;
Donde registro es el que se quiere modificar, X es el número del puerto (1.**0**) e Y es el valor que se le quiere dar. Es fácil de leer: P**1**DIR_bit.P**0** = **1** le asigna al puerto 1.0 valor 1 en el registro DIR

Otras formas:
- P1DIR = 10; que se lee como 10 = 0000 1010 (en binario). Es decir que da valor 1 a 1.1 y 1.3. La desventaja es que configura todos los puertos simultáneamente
- P1DIR = BIT2; que se lee BIT2 = 0000 0100 (los bits del registro se enumeran BIT0, BIT1,...).

En conclusión, para prender el led en 1.0, alcanzan estas dos líneas:
P1DIR_bit.P0 = 1;
P1OUT_bit.P0 = 1;

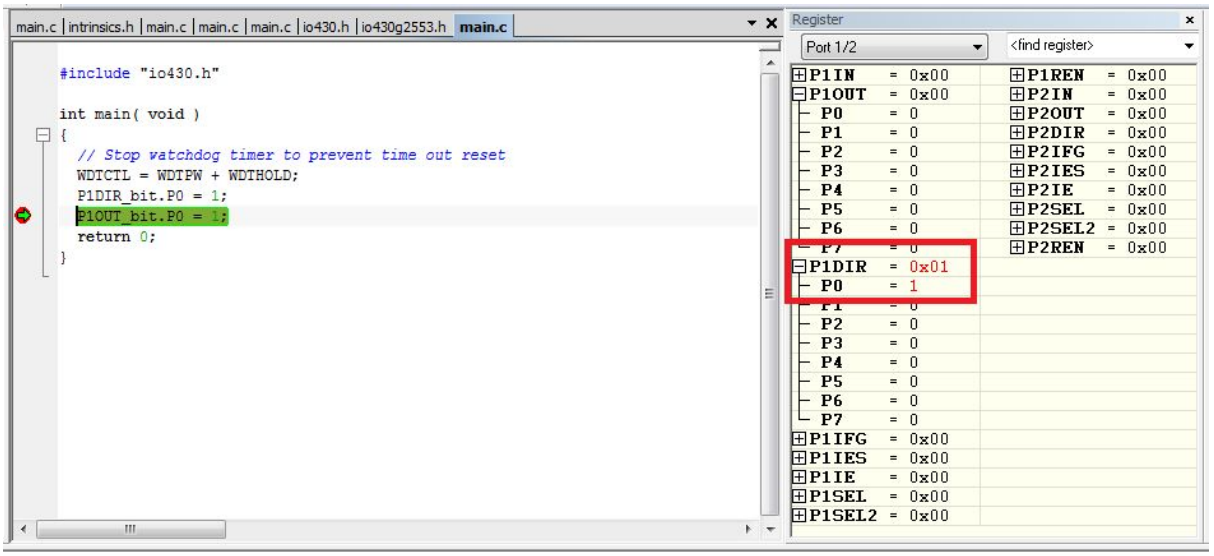
Pasamos a debuggear para ver que todo funcione. Dentro de la sesión se puede controlar cómo van cambiando los valores de los registros y las variable.
Para ver los registros: View -> Registers
Para ver las variables: View -> Statics/Locals



En recuadro negro, controles: **Reset** (vuelve al principio), **Stop** (frena donde esta), **Step Over** (avanza una línea), **Step Into** (si hay un llamado a otro archivo, entra a ese archivo), **Step Out** (si está dentro de un archivo, por ej una función que se llamó, avanza hasta salir), Next statement (también avanza un paso, no se cual es la diferencia con step over), Run to cursor (avanza hasta donde esté el cursor), **Go** (avanza hasta llegar a un *breakpoint*).

Un *breakpoint* es un punto que se establece como freno. Por ejemplo, a uno le puede interesar verificar que antes de hacer un llamado a una función, las variables estén bien definidas, o que los registros están adecuadamente configurados. Para poner un breakpoint en la posición del cursor se usa el botón rojo dentro del recuadro rojo de la imagen. Cuando uno da **Go** el código corre hasta llegar a la línea del breakpoint.

En el recuadro rojo se muestran los valores de los registros.
Ejemplo:



Breakpoint en la línea P1OUT_bit.P0 = 1; -> frena antes de ejecutar esa línea. A la derecha se puede ver que se configuró adecuadamente el registro P1DIR (los registros modificados se muestran en rojo), pero aún no se modificó el P1OUT.

Un ejemplo apenas más complicado es hacer titilar el led. Hay que incluir un loop y se puede poner un delay para ajustar la frecuencia (aun no hablamos del reloj):

```
int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  while(1){
          P1DIR_bit.P0 = 1;
          P1OUT_bit.P0 ^= 1;
          __delay_cycles(5000);
  }
  return 0;
}
```

El loop es C. El comando __delay_cycles(x) cuenta hasta x (cuando hablemos del reloj va a ser más claro) antes de seguir.
El operador **^=** es un OR exclusivo (ver http://www.tutorialspoint.com/cprogramming/c_operators.htm)

Simulación

Comentario para la sección: simular es una herramienta que tiene sus ventajas (quedará claro con los ejemplos) pero no es fundamental. Esta sección puede leerse obviando todo lo referente a Proteus, e implementar los ejemplos en el launchpad.

Para esta sección:
http://microembebidos.com/2013/05/07/tutorial-msp430-configuracion-de-herramientas-de-trabajo/
https://www.youtube.com/watch?time_continue=198&v=JvIGafQuXUU

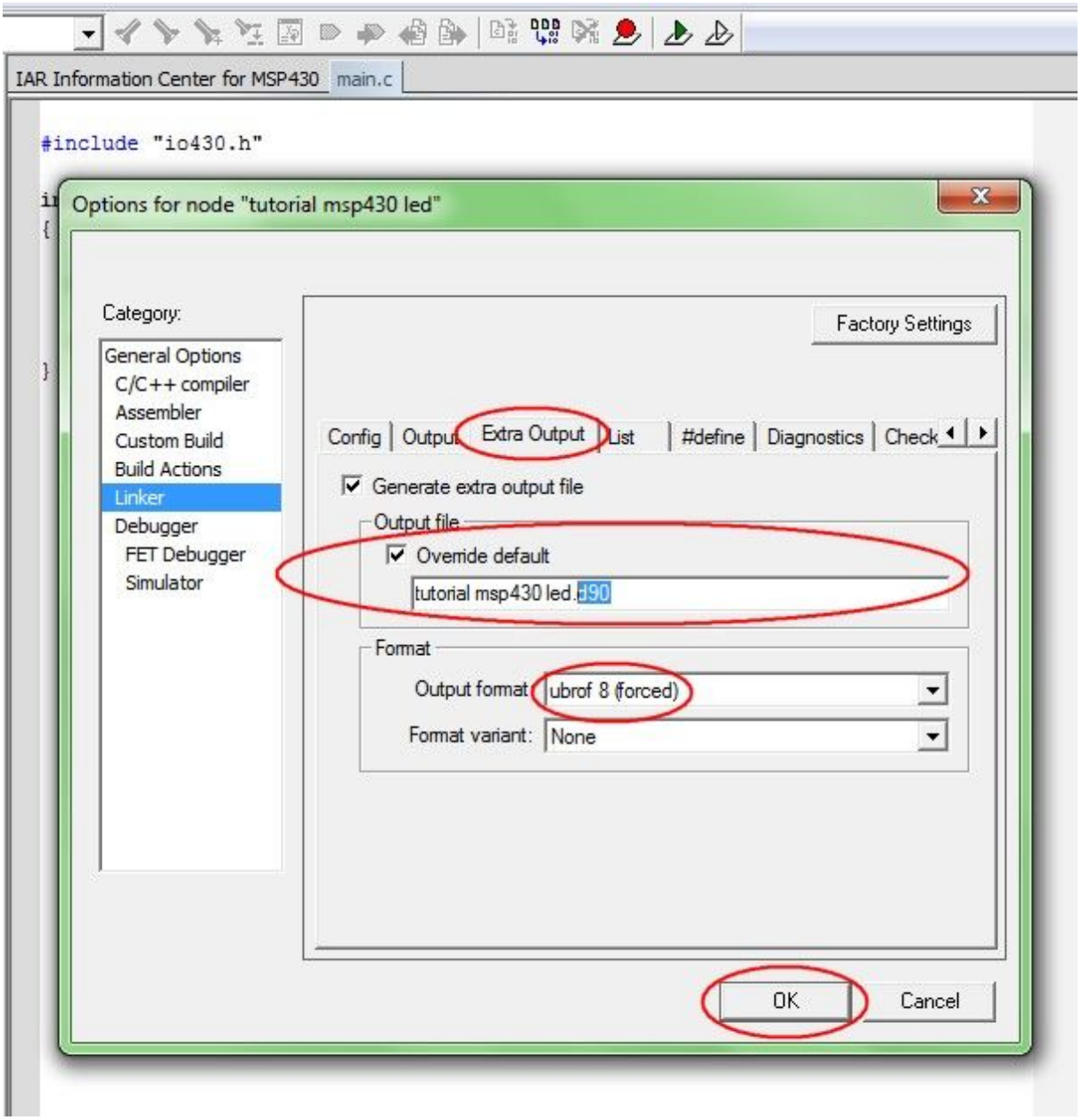
Mientras se programa, suele ser útil usar un simulador para ver si el programa se comporta como uno quiere, más que nada cuando se agregan periféricos. Por ejemplo, si se quiere comprobar que el ADC10 esta andando correctamente, lo mejor es tener una señal conocida como entrada y ver si la registra bien; o al agregar la tarjeta sd, para ver si está interactuando correctamente o escribiendo los datos.

Proteus es un software que cumple: puede simular fuentes, circuitos, procesador y tarjeta. Una de las desventajas es que no tiene en la base de datos tantos modelos del procesador, pero por lo general (como la programación es en todos igual, salvo detalles puntuales) se puede usar con algún procesador similar.
La estrategia es iterar un programa, compilandolo para un procesador que se pueda simular, hasta llegar a una versión más completa, compilarlo para el procesador real, descargarlo y verificar que funcione.
En retrospectiva creo que la mayor virtud de la simulación es para aprender a programar. Con un poco más de experiencia o con un programa con mas forma, la opción de trabajar en el procesador real me parece mejor.

Cargar programa y configuraciones

Lo primero que hay que hacer es buscar un procesador en Proteus lo más parecido posible al real, o que al menos tenga las mismas funcionalidades (o al menos las que se necesiten). Por ejemplo el msp430f2232

Tiene mismo RAM, usci, timer, adc10.
En IAR, configurar el Target para este procesador. También:
Opciones -> Linker, pestaña Output -> Allow C-SPY specific extra output
Pestaña Extra Output:



Luego en Proteus doble click sobre el procesador, en Program File buscar el archivo .d90 generado por IAR.
A partir de ahi se arma el circuito. Los diseños para los ejemplos estan subidos en la carpeta.

Ejemplos con funciones útiles

Los ejemplos están subidos en la carpeta “Ejemplos (código + simulación)
Pueden probarse tanto en simulación como en Launchpad

Led básico

El mismo código de antes, solo para reforzar (probar la simulación, jugar con los tiempos) y para introducir la calibración del oscilador (DCO en adelante).

Vamos a usar **como fuente** del reloj al DCO, que se controla por software. **Ver capítulo 5 de la User Guide**
El DCO es un oscilador interno que se configura por software. Toda operación requiere un reloj, desde avanzar una línea de código, hasta un timer, el ADC10, etc. Entonces a partir de este oscilador se configuran uno o más relojes.
El primero es el **master clock** (MCLK), también se puede usar un **sub-main clock** (SMCLK) y un auxiliar (ACLK). A grandes rasgos, el tic del reloj define la velocidad de operación, hay procesos que requieren más tiempo, otros menos y a veces puede ser útil drivear operaciones distintas a distintas velocidades. También hay funciones que requieren el SMCLK.
Ejemplo: si se quiere medir una señal que varía lento, el ADC10 no necesita “operar” rápido. Entonces se puede usar un reloj más lento (SMCLK) para drivear el ADC10 y un reloj más rápido (MCLK) para ejecutar el código en sí.

Lo primero que hay que hacer es configurar el DCO. Son 2 registros:

5.3.1 DCOCTL, DCO Control Register

7	6	5	4	3	2	1	0
DCOx			MODx				
rw-0	rw-1	rw-1	rw-0	rw-0	rw-0	rw-0	rw-0

DCOx	Bits 7-5	DCO frequency select. These bits select which of the eight discrete DCO frequencies within the range defined by the RSELx setting is selected.
MODx	Bits 4-0	Modulator selection. These bits define how often the f_{DCO+1} frequency is used within a period of 32 DCOCLK cycles. During the remaining clock cycles (32-MOD) the f_{DCO} frequency is used. Not useable when DCOx = 7.

5.3.2 BCSCTL1, Basic Clock System Control Register 1

7	6	5	4	3	2	1	0
XT2OFF	XTS(1)(2)	DIVAx		RSELx			
rw-(1)	rw-(0)	rw-(0)	rw-(0)	rw-0	rw-1	rw-1	rw-1

XT2OFF	Bit 7	XT2 off. This bit turns off the XT2 oscillator 0 XT2 is on 1 XT2 is off if it is not used for MCLK or SMCLK.
XTS	Bit 6	LFXT1 mode select. 0 Low-frequency mode 1 High-frequency mode
DIVAx	Bits 5-4	Divider for ACLK 00 /1 01 /2 10 /4 11 /8
RSELx	Bits 3-0	Range select. Sixteen different frequency ranges are available. The lowest frequency range is selected by setting RSELx = 0. RSEL3 is ignored when DCOR = 1.

Los bits importantes están resaltados. Básicamente, con BCSCTL1 se define el rango (RSELx) y con DCOCTL se define un valor dentro de ese rango.
Lo bueno es que los procesadores vienen con calibraciones incluidas (1-8-12-16 MHz), por lo que no hace falta volverse loco con esto. Alcanza con poner:
BCSCTL1 = CALBC1_XMHZ;// Usar calibracion X MHz
DCOCTL = CALDCO_XMHZ;// Usar calibracion X Mhz
Para tener el DCO calibrado para correr a X MHz, y el MCLK siguiéndolo (misma frecuencia)

Veamos un poco en detalle cómo jugar con el MCLK y SMCLK (para reforzar como asignar valores a registros también). El registro que hay que modificar es BCSCTL2:

5.3.3 BCSCTL2, Basic Clock System Control Register 2

7	6	5	4	3	2	1	0
SELMx		DIVMx		SELS	DIVSx		DCOR(1)(2)
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

SELMx	Bits 7-6	Select MCLK. These bits select the MCLK source.
		00 DCOCLK
		01 DCOCLK
		10 XT2CLK when XT2 oscillator present on-chip. LFXT1CLK or VLOCLK when XT2 oscillator not present on-chip.
		11 LFXT1CLK or VLOCLK
DIVMx	Bits 5-4	Divider for MCLK
		00 /1
		01 /2
		10 /4
		11 /8
SELS	Bit 3	Select SMCLK. This bit selects the SMCLK source.
		0 DCOCLK
		1 XT2CLK when XT2 oscillator present. LFXT1CLK or VLOCLK when XT2 oscillator not present
DIVSx	Bits 2-1	Divider for SMCLK
		00 /1
		01 /2
		10 /4
		11 /8
DCOR	Bit 0	DCO resistor select. Not available in all devices. See the device-specific data sheet.
		0 Internal resistor
		1 External resistor

Mirar la parte superior. Los números representan el bit, que va de 0 a 7, es decir que el registro tiene 7 bits. Los bits 6 y 7 corresponden a SELMx. Mirando la lista, vemos que
SELMx Bits 7-6 Select MCLK. These bits select the MCLK source
Que explica la función de estos bits (elegir quien es la fuente del reloj principal). Si estos bits valen 00 o 01, la fuente es el DCO.

Supongamos que tenemos el DCO corriendo a 8 MHz y queremos configurar el MCLK a 4 MHz y el SMCLK a 1 MHZ. Necesitamos que el registro tenga los siguientes bits:

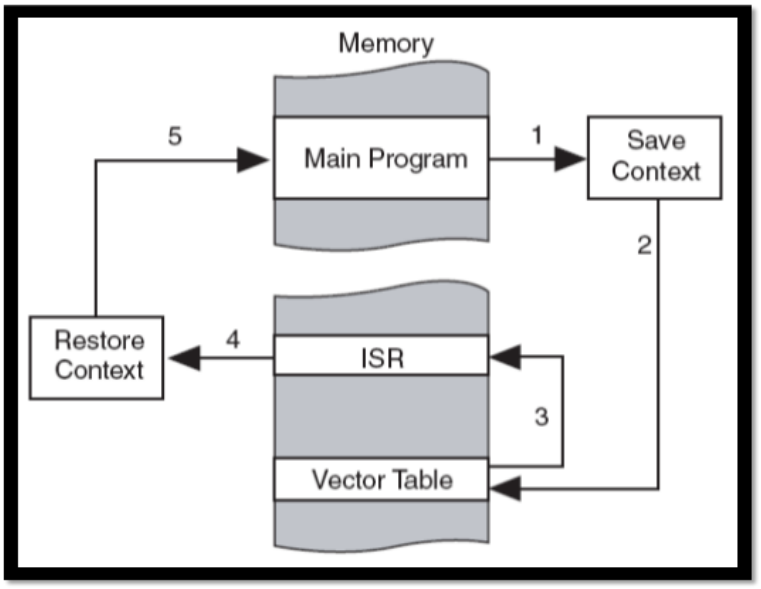
SELMx	DIVMx	SELS	DIVSx	DCOR
00	01	0	10	0

Podemos asignar estos bits como antes, uno por uno:
BCSCTL2_bit.DIVM0 = 1
BCSCTL2_bit.DIVM1 = 0
BCSCTL2_bit.SELM0 = 0
BCSCTL2_bit.SELM1 = 0 ...
o
BCSCTL2 = BIT2 + BIT4;
o
BCSCTL2 = DIVS1 + DIVM0;

Ante la duda, para ver si estamos asignando bien los valores se puede hacer un debug.

Led + timing (DCO y Timer_A) + interrupciones

El objetivo ahora es aprender a controlar el timer y entender las **interrupciones.**
Las interrupciones son bloques de código que se ejecutan eventualmente al cumplirse cierta condición. No pertenecen al programa principal. Son la versión digital de “quiero que pase esto cuando aprieto este botón”. De entrada uno no sabe cuándo, ni cuantas veces se va a apretar el botón por lo que es imposible embeber esta operación en un programa “usual” que lee líneas secuencialmente. Ver figura



Tenemos por un lado la función main, que se va ejecutando. En algún momento ocurre un evento que provoca una interrupción, por ejemplo (ahora lo vamos a ver en detalle) un timer cuenta hasta cierto valor. En ese momento, esté donde esté, el código principal se detiene y pasa a la rutina de interrupción. Primero identifica la interrupción y después ejecuta el código de la interrupción. Una vez que terminó, retoma el código principal donde lo había dejado.

El Timer_A es un contador de 16 bits (puede contar hasta 2^16 = 65536), que es accesible (puede leerse en cualquier momento por cuanto va la cuenta)
Para la configuración alcanza un registro:

12.3.1 TACTL, Timer_A Control Register

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

Unused	Bits 15-10	Unused
TASSELx	Bits 9-8	Timer_A clock source select
	00	TACLK
	01	ACLK
	10	SMCLK
	11	INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock.
	00	/1
	01	/2
	10	/4
	11	/8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power.
	00	Stop mode: the timer is halted.
	01	Up mode: the timer counts up to TACCR0.
	10	Continuous mode: the timer counts up to 0FFFFh.
	11	Up/down mode: the timer counts up to TACCR0 then down to 0000h.
Unused	Bit 3	Unused
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request.
	0	Interrupt disabled
	1	Interrupt enabled
TAIFG	Bit 0	Timer_A interrupt flag
	0	No interrupt pending
	1	Interrupt pending

TASSEL elige la fuente (el tic). Notar que necesita SMCLK (no puede usar el MCLK). ID sirve para ir más lento que el reloj, por ejemplo si el SMCLK está a 8MHz ID2 (divide por 4) hace que el timer corra a 2MHz.

Se puede definir un parámetro TACCR0 (menor a 65536) que juega en el modo de operación:
Stop mode: timer parado
Up mode: Cuenta desde 0 hasta TACCR0 y vuelve a empezar. Cada vez que llega a TACCR0 genera una interrupción
Continuous mode: cuenta hasta 65536
Up down: cuenta hasta TACCR0 y después hasta 0 (sube y baja)

Ejemplo:

```
CCTL0 = CCIE;
TA0CTL = TASSEL_2 + ID_0 + MC_1; // Source + Divisor + Modo
TACCR0 = 61; // Cuenta hasta TACCR0 + 1
```

La primera línea habilita las interrupciones del timer. En la segunda se configura con fuente SMCLK, divisor 0 (misma frecuencia que el reloj) y modo UP (cuenta hasta TACCR0). En la tercera línea se le dice hasta cuando cuenta.

Esta es la configuración que usamos en el programa final. Con el reloj (SMCLK) corriendo a 1MHz (y el MCLK a 8MHz) las interrupciones ocurren cada 62 tics (cuenta hasta 61 y le lleva 1 tic volver a 0) es decir a una frecuencia de:
f_timer = 1.000.000 / 62 = 16.129 Hz

ADC10

El ADC10 es un conversor analógico digital de 10 bits. Va a ser el que recibe las señales analógicas y las transforma en algo que pueda manejar el procesador. Conviene leer de la guia, al menos las secciones 22.2.1.1, 22.2.3, 22.2.5, 22.2.6 (solo la tabla)

A configurar:
- Referencia: determinan el rango. El conversor es de 10 bits (1024 divisiones).
- Sample and hold time (SHT):
La medición lleva un tiempo (pensarlo como un capacitor que se carga). Se puede (hay que) configurar cuanto tiempo espera este llenado. Tener en cuenta el proceso completo de medición y conversión:

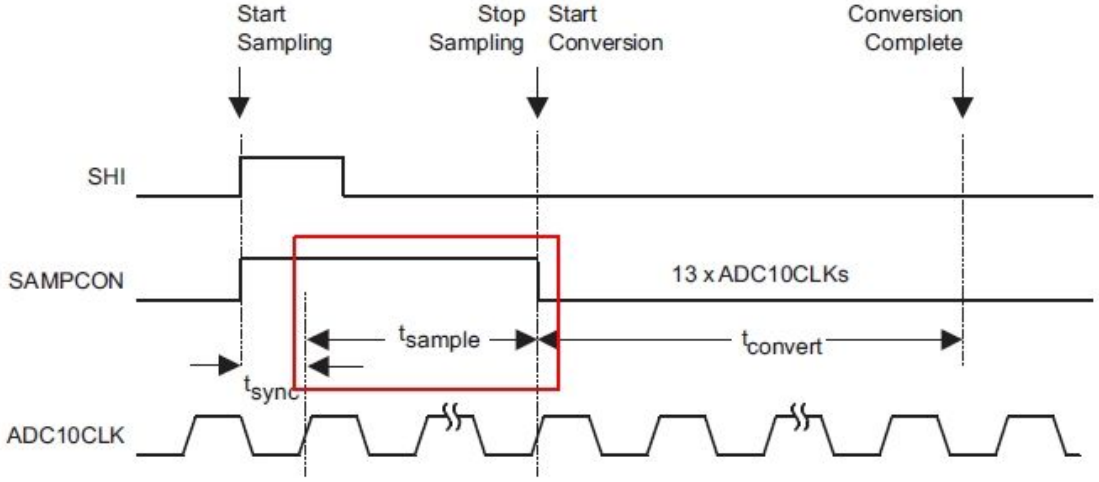


Figure 22-3. Sample Timing

Es decir que lleva mínimo 13 tics del reloj + tiempo de sampleo. Por ejemplo si se configura el SHT a 16 tics, serían 29 tics (notar como esto juega con la frecuencia de adquisición). De la User Guide da una idea de como estimar el tiempo mínimo:

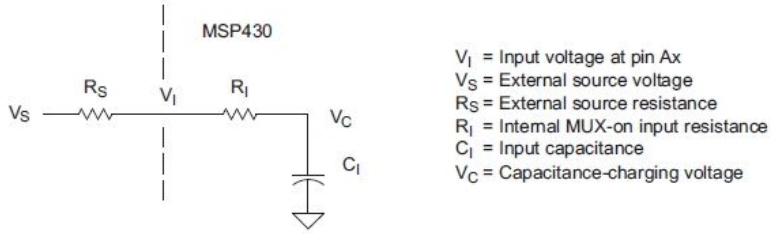


Figure 22-4. Analog Input Equivalent Circuit

The resistance of the source R_S and R_I affect t_{sample}. The following equations can be used to calculate the minimum sampling time for a 10-bit conversion.

$$t_{sample} > (R_S + R_I) \times \ln(2^{11}) \times C_I$$

Substituting the values for R_I and C_I given above, the equation becomes:

$$t_{sample} > (R_S + 2\ k\Omega) \times 7.625 \times 27\ pF$$

(En los programas estoy usando 16 tics y anda bien)
- Puertos habilitados: establecer que puertos van a estar disponibles para medir (no dice en cual se va a medir de hecho). Si se va a medir más de un canal alternativamente, hay que habilitar ambos.
- Modo de medición: uno o varios canales, una muestra o varias
- Puerto medicion: en qué puerto se mide efectivamente. Tiene que estar habilitado.

Los registros relevantes son:

22.3.1 ADC10CTL0, ADC10 Control Register 0

15	14	13	12	11	10	9	8
SREFx			ADC10SHTx		ADC10SR	REFOUT	REFBURST
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
MSC	REF2_5V	REFON	ADC10ON	ADC10IE	ADC10IFG	ENC	ADC10SC
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

Can be modified only when ENC = 0

Los bits básicos que vamos a usar son:
SREF = referencia
SHT = sample and hold time
ADC10ON = enciende el conversor
ADC10IE = habilita las interrupciones generadas por el conversor
ENC = habilita la medicion
AD10SC = comienza la conversión
Estos ultimos dos se configuran en una linea aparte, marcan el inicio de la medición

22.3.2 ADC10CTL1, ADC10 Control Register 1

15	14	13	12	11	10	9	8
INCHx				SHSx		ADC10DF	ISSH
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ADC10DIVx			ADC10SSELx			CONSEQx	ADC10BUSY
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r-0

Can be modified only when ENC = 0

INCH = seleccionar canal donde se mide
ADC10DIV = divisor del reloj

ADC10SSEL = fuente del reloj
CONSEQ = modo de conversión

22.3.3 ADC10AE0, Analog (Input) Enable Control Register 0

7	6	5	4	3	2	1	0
ADC10AE0x							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

ADC10AE0x	Bits 7-0	ADC10 analog enable. These bits enable the corresponding pin for analog input. BIT0 corresponds to A0, BIT1 corresponds to A1, etc. The analog enable bit of not implemented channels should not be programmed to 1. 0 Analog input disabled 1 Analog input enabled

Habilitar puertos para usar como input del ADC10

En el ejemplo usamos la siguiente configuración:

```
ADC10CTL0 = SREF_2 + ADC10SHT_2 + ADC10ON + ADC10IE;
ADC10AE0 |= BIT1;
ADC10CTL1 = CONSEQ_0 + INCH_1;
ADC10CTL0 |= ENC + ADC10SC;
```

La primera línea define la referencia, el SHT enciende el ad10 y habilita interrupciones. Luego se habilitan los puertos que se van a usar. Despues se elige el modo de medición y el canal y finalmente comienza la medición.

Si en algún momento se quiere cambiar una configuración, es importante recordar que los registros ADC10CTLx solo pueden modificarse si ENC = 0. Hay dos formas de hacerlo, depende de lo que quiera hacer uno:

```
ADC10CTLx &= ~ENC;
```
Solo desactiva el bit ENC, todos los demas quedan iguales

```
ADC10CTLx &= ~0xFFFF;
```
Borra todos los bits

Ejemplo, en el programa final medimos en dos canales, secuencialmente:
```
ADC10AE0 |= BIT1 + BIT2;
```
Habilitamos los dos canales
```
ADC10CTL0 &= ~ENC;
```
Solo borramos ENC ya que no queremos cambiar la configuracion de este registro
```
ADC10CTL1 &= ~0xFFFF;
```
Borramos todo el registro porque tenemos que cambiar de canal:
```
ADC10CTL1 = CONSEQ_0 + INCH_2;
```

ADC10 + timer

El objetivo ahora es combinar estas dos cosas, de manera de tener mediciones cada intervalos definidos (por el timer)
La idea ahora es tener una conversión del ADC10 cada vez que el timer dice. Para hacer esto combinamos lo visto en las secciones anteriores. Usamos la misma configuración, DCO, relojes, timer y ADC:

```
  CCTL0 = CCIE;
  TA0CTL = TASSEL_2 + ID_0 + MC_1;
  TACCR0 = 61;
  ADC10CTL0 &= ~ENC;                          // Disable Conversion
  ADC10CTL0 = SREF_0 + ADC10SHT_2 + ADC10ON + ADC10IE;
  ADC10AE0 |= BIT2;                       // PA.2 ADC option select
  ADC10CTL1 = CONSEQ_0 + INCH_2;                          // input A2
```

Donde configuramos para medir en A2 cada 62 tics (62us si el reloj SMCLK corre a 1MHz)

Cómo hacemos para que el trigger active la medición? Encendemos el ADC **dentro** de la interrupción del timer:

```
#pragma vector=TIMER0_A0_VECTOR
__interrupt void interrupt_timer(void)
{
  P1OUT_bit.P0 ^= 1;
  ADC10CTL0 |= ENC + ADC10SC;
  while (ADC10CTL1 & BUSY);
}
```

O sea, cada vez que el timer interrumpe, prende el ADC, que vuelve a interrumpir con su propia rutina. Como antes, para que el código siga dando vueltas ponemos en el main un *while(1)*.

```
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void)
{
  while (ADC10CTL1 & BUSY);// espero a que se desocupe
  audio.In=(ADC10MEM>>2);           //convert audio to 8 bit sample....  >> binary shift ->
mueve los bits 2 lugares
  audio.bufferBytes++;          //new byte in the buffer
  audio.lsb=ADC10MEM & 0x03;
}
```

Dentro de la interrupción grabamos el valor medido, de 10 bits, en dos variables de 8 y 2 bits respectivamente.

Micro SD

Generalidades (bloques memoria, pinout)

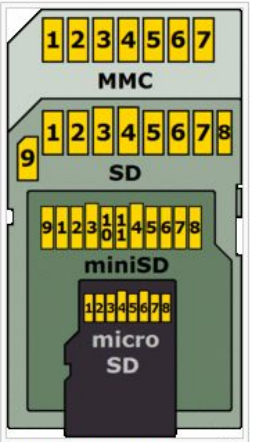
Las memorias uSD se comunican mediante el protocolo SPI:
http://elm-chan.org/docs/spi_e.html
http://elm-chan.org/docs/mmc/mmc_e.html (muy técnico)
No voy a entrar en detalle de como es el protocolo (tampoco sé mucho), lo importante es que es un modo de comunicación sincrónico (regulado por un reloj). Lo que hay que tener presente es que las tarjetas micro sd (o cualquier dispositivo) necesitan un protocolo específico, una serie de comandos necesarios (prender, conectar, apagar,...) y que otra gente se ocupó de generar el código pertinente.

Una cosa importante que hay que tener presente es la estructura de las memorias. Están divididas en sectores de 512 bytes. Esto quiere decir que antes de escribir información uno tiene que “montar” un sector (ie prepararlo) y que al terminar de escribirlo, hay que “desmontarlo”. También quiere decir que no se puede escribir 1 byte (o al menos, que escribir 1 byte va a ser tan costoso en tiempo como escribir 512)
Es igual cuando se lee la tarjeta, vamos a tener que leer de a sectores completos.

Respecto a la parte “física” de la tarjeta, es importante tener el pinout claro

SPI Bus Mode

MMC Pin	SD Pin	miniSD Pin	microSD Pin	Name	I/O	Logic	Description
1	1	1	2	nCS	I	PP	SPI Card Select [CS] (Negative logic)
2	2	2	3	DI	I	PP	SPI Serial Data In [MOSI]
3	3	3		VSS	S	S	Ground
4	4	4	4	VDD	S	S	Power
5	5	5	5	CLK	I	PP	SPI Serial Clock [SCLK]
6	6	6	6	VSS	S	S	Ground
7	7	7	7	DO	O	PP	SPI Serial Data Out [MISO]
	8	8	8	NC nIRQ	. O	. OD	Unused (memory cards) Interrupt (SDIO cards) (Negative logic)
	9	9	1	NC	.	.	Unused
		10		NC	.	.	Reserved
		11		NC	.	.	Reserved



Official pin numbers for each card type (top to bottom): MMC, SD, miniSD, microSD. This shows the evolution from the older MMC, on which SD is based. NOTE: This drawing doesn't show 8 new UHS-II

Las líneas que nos interesan son: CS, MOSI (Master Out Slave In, ie el procesador le habla a la tarjeta), Ground, Power, SCLK (señal del reloj), MISO (Master In Slave Out, ie la tarjeta le responde al procesador).

Código

Vamos a aprovechar el codigo anterior para ir convergiendo a lo que queremos llegar. Hasta ahora tenemos la capacidad de realizar una medición y guardarla regularmente en una variable. Ahora queremos enviar esta información a la tarjeta.

Lo que hay que hacer antes de enviar datos es:
1- Configurar el procesador adecuadamente para el protocolo de comunicación
2- Establecer la comunicación
3- Montar un sector

Esta es la parte más oscura, todo el código es “prestado” (google). Hay que incluir archivos extra: mmc.h, mmc.c, hal_SPI.h, hal_SPI.c
Ahi esta toda la magia. En el código se llama a la funcion
disk_initialize() para inicializar la tarjeta
*mmcWriteMount(flashDisk.currentSector*512)* monta un sector
mmcWriteUnmount(flashDisk.currentByte) cierra el sector
mmcWriteByte(data) escribe *data* en la tarjeta (8 bits)

También hay que agregar una parte de código que se ocupa de verificar si se llegó al final de un sector y pasar al siguiente. Basicamente hay que llevar la cuenta de la cantidad de bytes enviados, cuando se llega a 512, se cambia de sector.

Simulación

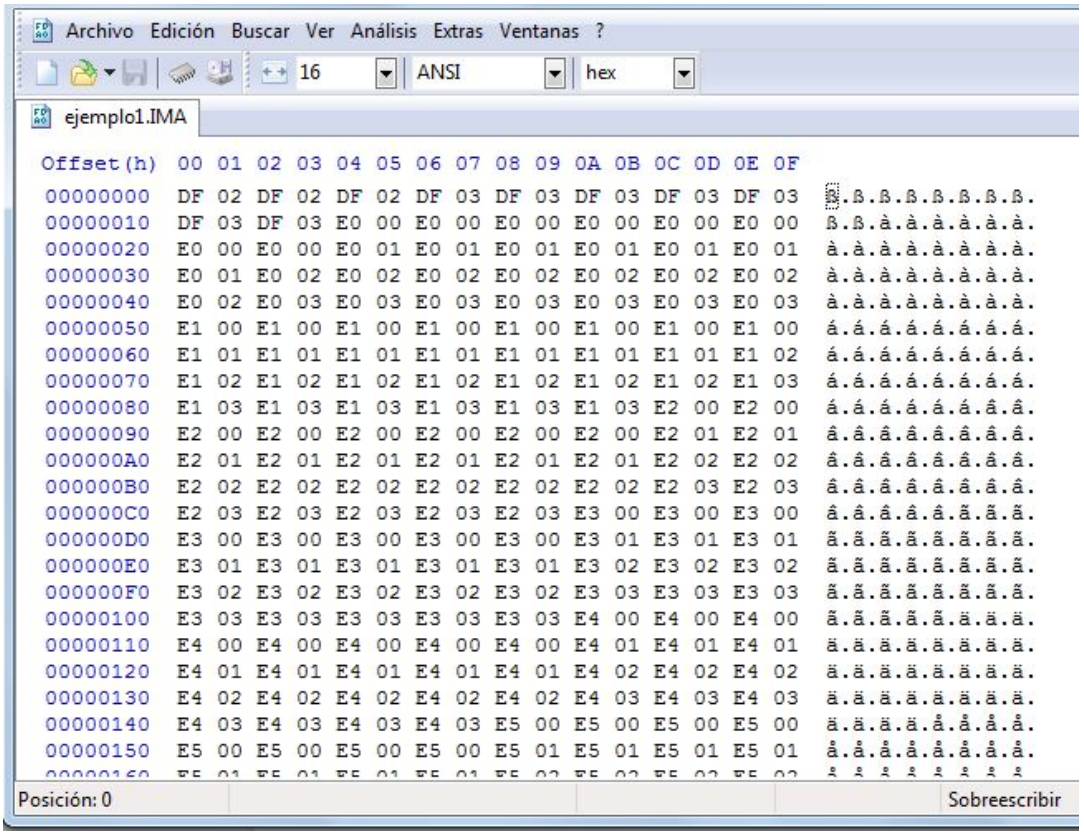
Para simular un tarjeta se puede usar Proteus. Se agrega una MMC y se conectan las líneas CS, DI, DO, CLK con los puertos correspondientes del procesador. Luego hay que cargar una *imagen* (algo como un archivo, que simula ser una tarjeta). Botón derecho -> Edit Properties -> Card Image File

La imagen *imagen1.IMA* está en la carpeta correspondiente. El contenido de la imagen se puede leer con HxD (siguiente sección)

Hex editor y lectura de datos

Para ver qué es lo que se está guardando y laburar en algún código para recuperarlo correctamente hace falta leer la memoria de la tarjeta. El problema es que al escribir directamente sobre los bytes de memoria (sin implementar un sistema de archivos), cuando se abre la tarjeta con cualquier explorador aparece vacía (incluso Windows pide formatearla y falla).
Para armar el código y probarlo, lo mejor es usar una imagen de las simulaciones con valores conocidos escritos. Estas imágenes pueden abrirse con editores Hex:
https://mh-nexus.de/en/hxd/ (windows)
http://home.gna.org/bless/ (linux)
http://ridiculousfish.com/hexfiend/ (mac, no lo probé)

Ejemplo, en HxD, luego de medir una señal senoidal se ve así:



Que es algo razonable si uno recuerda cómo se está organizando la información. Los datos de 10 bits se envían en dos paquetes, el primero de 8 bits, el segundo de 2.
El hecho de que en el editor aparezcan letras se debe a que está representando los datos en **hexadecimal**.

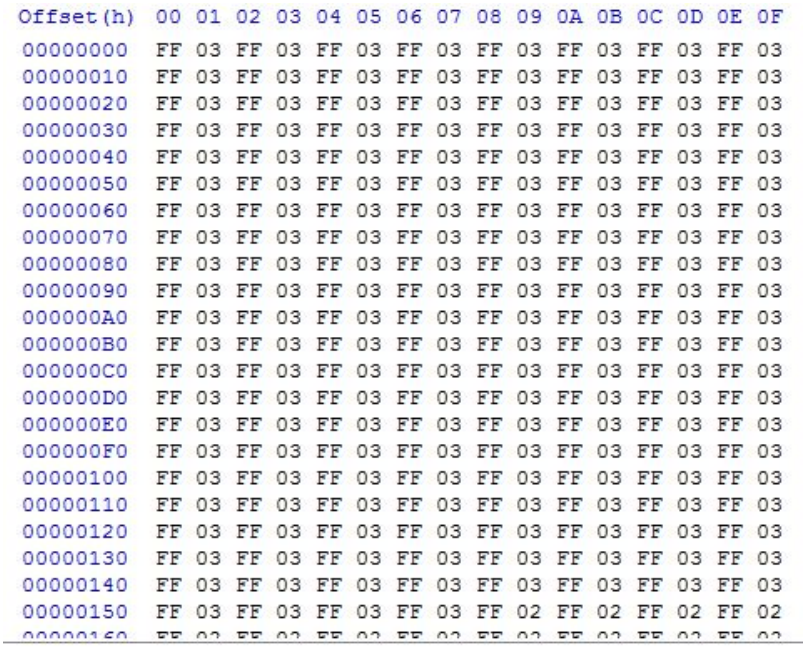
Voy a seguir la línea de razonamiento para reobtener los datos para este caso, hay que tener en cuenta que esto es completamente dependiente de **cómo se decide guardar la información al programar el procesador**.

Uso h para referirme a números en hexadecimal (h1, hA,..) y b para binario (b1, b101,...).
Un dígito hexadecimal va de h0 = b0 = 0, a hF = b1111 = 15. Es decir que un hexa es igual a 4 bits. Entonces está claro, viendo la figura anterior que los dos primeros hexas corresponden a los 8 primeros bits enviados (1 byte) y que los siguientes 2 son los otros 2 bits. Ver que la estructura se repite (**hacer** ejemplo)
Entonces para reconstruir el primer dato tenemos:
hDF = b1101 1111
b02 = b10
Para reconstruir el dato los concatenamos:
b1101 1111 10 = 894

Codigo en C para lectura

La base del código está sacada de:
http://www.nerdkits.com/forum/thread/2599/

Para hacer pruebas es conveniente usar un archivo conocido, preferentemente una imagen que se sepa que contiene.
Como ejemplo usamos la imagen “ejemplo1.IMA” (en archivos), que corresponde a una señal senoidal de baja frecuencia. Abriendola con HxD:



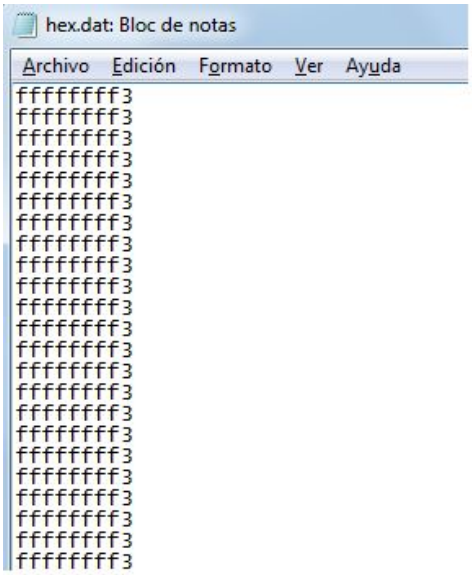
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000010	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000020	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000030	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000040	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000050	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000060	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000070	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000080	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000090	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
000000A0	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
000000B0	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
000000C0	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
000000D0	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
000000E0	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
000000F0	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000100	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000110	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000120	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000130	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000140	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03	FF	03
00000150	FF	03	FF	03	FF	03	FF	03	FF	02	FF	02	FF	02	FF	02

Como sabemos cual fue la estructura de grabado de datos = 8 bits + 2 bits podemos reconstruir qué significa esto. Cada conjunto de dos caracteres es un byte, representado como un hexadecimal de dos dígitos. Un hexadecimal = 4 bits (0-15)
El primer byte es hFF = 1111 1111
El segundo byte es h03 = 11
Por lo que el primer dato es la concatenación = 1111 1111 11 = 1023 (decimal)

El programa va a hacer lo siguiente:

1. Abrir el disco
2. Copiar los datos en un buffer
3. Guardar los datos en hexadecimal
4. Transformar a decimal
5. Guardar en decimal

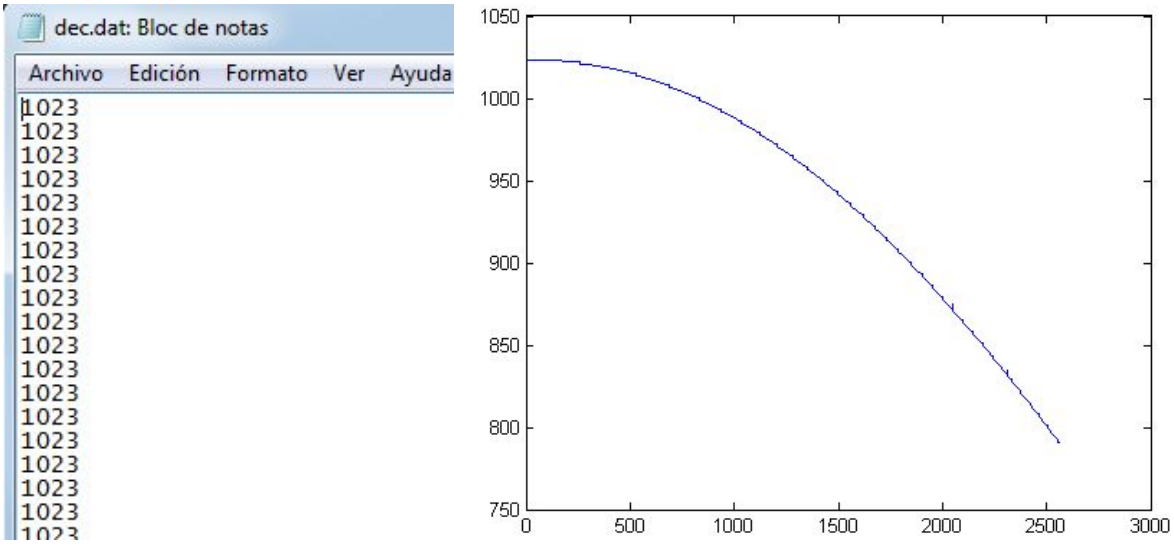
Los primeros pasos son bastante directos, los datos se guardan en un archivo llamado “hex.dat” ya concatenados. Por un tema de formato, si el byte es mayor a 128 aparecen f’s al principio. Entonces el byte FF, lo lee como FFFFFF**FF**, el FE como FFFFFF**FE,** etc.
El hex.dat se ve así:



Para pasarlo a decimal, hay algunas vueltas, pero la idea es:

1. Pasar el lsb
2. Pasar el msb y multiplicarlo por 4
3. sumar

Hay algunos detalles técnicos, para más detalle ver el código comentado. En la figura siguiente: dec.dat y plot:

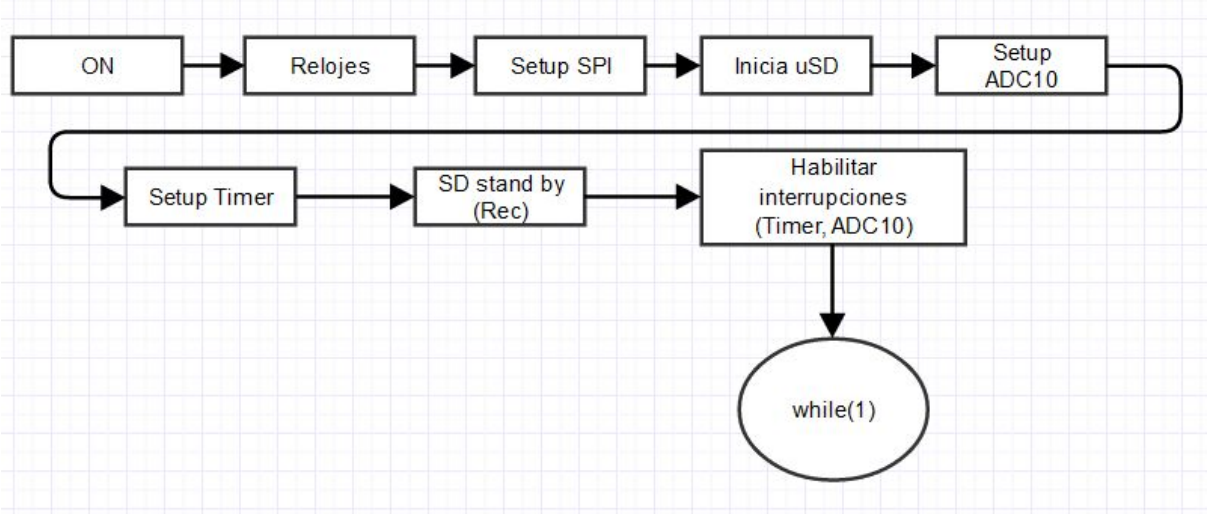


Código final

El código final es una combinación de los tres elementos que se vieron antes: Timer + ADC + SPI. El grueso del código (más que nada por la configuración de micro sd y SPI) está sacado de http://www.whereisian.com/files/DIYLife-MSP430-audio-firmware.zip (tambien subido)
El código final está subido en la carpeta “Firmware audio v0.4”

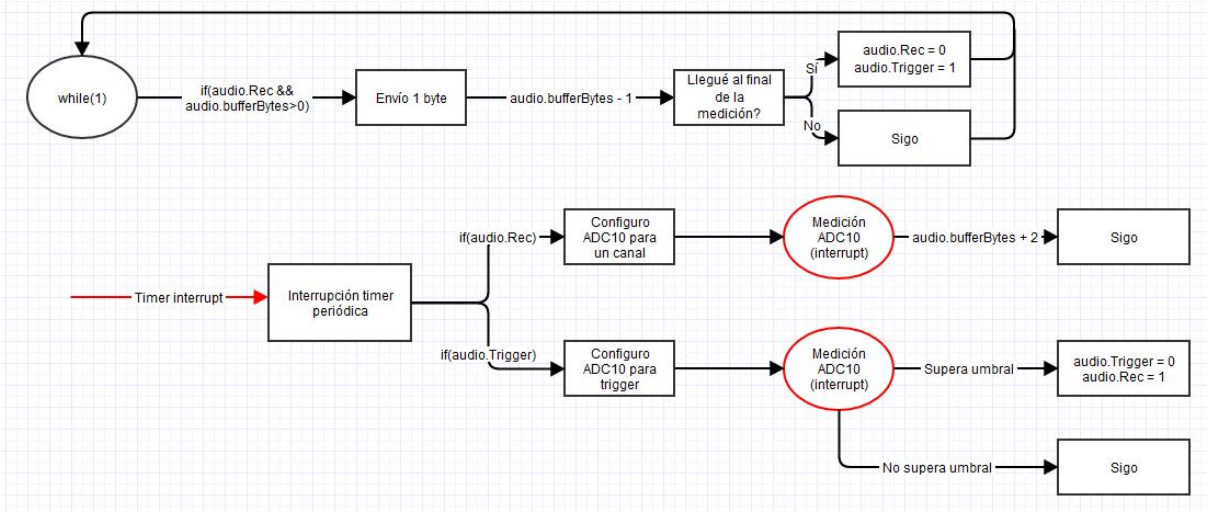
Flujo

Resulta más fácil separar el programa en dos partes. En la primera se hace todo el setup, en la segunda comienzan a jugar las interrupciones. La primera parte puede representarse por el siguiente diagrama



Luego de encender, comienza a correr el código. Primero se configuran los relojes (DCO, MCLK, SMCLK).
Luego se inicializa la tarjeta en dos pasos: 1° se configura el protocolo de comunicación SPI (se configuran los pines correspondientes de la manera adecuada, se establece el canal de comunicación), 2° se inicializa la tarjeta en sí (se envían comandos a la tarjeta de acuerdo al protocolo, diciéndole que se encienda, cual es la longitud de sector, etc)
A continuación se configura el ADC10 parcialmente. Una parte de la configuración (canal) va a ir cambiando (para medir en dos canales consecutivamente)
Después se configura el timer. El timer se ocupará de interrumpir periódicamente el código, va a ser nuestro “reloj” que determine la frecuencia de adquisición.
Finalmente se deja a la tarjeta en stand by, se habilitan las interrupciones y se entra en un loop infinito.

La segunda parte del código, lo que se encuentra dentro del loop y las interrupciones, se representa en el siguiente esquema:



La parte superior corresponde al código “usual” que se ejecuta linealmente. En cada vuelta del loop se verifican si se cumplen 2 condiciones que determinan si hay que enviar información a la tarjeta. La primera es si se está en modo “grabar”, que se codifica en la variable audio.Rec. Esta variable vale 1 si se está haciendo una medición (post trigger) o 0 si se está en stand by o esperando la señal de trigger.
La segunda condición es si hay información para enviar, es decir si hay algún dato medido que aún no se ha enviado. Esto se codifica en la variable audio.bufferBytes, que cuenta cuántos bytes hay para enviar.
Si ambas condiciones se cumplen se envia 1 byte (y por lo tanto se reduce la variable audio.bufferBytes en 1). Luego se verifica si se llegó al final de la medición (típicamente una medición durará un tiempo predefinido, por ejemplo 10 segundos, que se traduce en una cantidad fija de bytes por medición). Si no se llegó al final, sigue igual. Si se llegó al final, se deshabilita la función “grabar” (audio.Rec = 0) y se habilita la función “trigger” (audio.trigger = 1). Esto hace que en la próxima vuelta del loop no se entre en esta sección del código (i.e. si se envió el último byte correspondiente a una medición no se envía nada más hasta el próximo trigger).

La sección inferior representa la lógica de las interrupciones. Hay que recordar que las interrupciones son justamente eso, *interrupciones.* Mientras el código se ejecuta, al cumplirse la condición que desencadena la interrupción, se detiene la ejecución del código y se ejecuta el código contenido en la interrupción.
La primera interrupción es la del timer, que se produce periódicamente y será la que determine la frecuencia de adquisición. En la interrupción se chequea si se está en modo “grabar” o modo “trigger”. En el primer caso, se mide un canal (más detalle a continuación), se guarda el valor medido y se informa que hay bytes nuevos para enviar (audio.bufferBytes + 2). En el segundo caso, se mide el canal definido como trigger y se ve si supera un umbral. Si es asi, se pasa a modo “grabar”, si no se continúa en modo “trigger”.

Lógica de triggereo

Por el momento, el triggereo consiste simplemente en ver si uno de los canales supera un umbral que se debe definir por anticipado (durante la programación). También debe definirse cuál será el canal usado para triggerear.
En el programa hay dos variables que controlan si se graba o se espera el trigger.
audio.Rec = 1 y audio.Trigger = 0 para grabar
audio.Trigger = 1 y audio.Rec = 0 para esperar trigger
Cuando se está en modo trigger se mide repetidamente un canal y se compara el valor con el umbral definido. Solo cuando una medición supera el umbral se pasa a modo grabar.

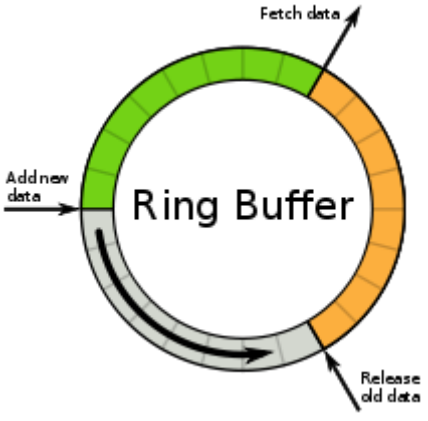
Una vez en modo grabar, los valores adquiridos se van guardando en un buffer (siguiente sección) y luego son transmitidos.

Buffer

En una primera versión no tenía buffer. Esto quiere decir que se adquiría una medición y habia que (si o si) hacer la transmisión antes de la siguiente adquisición. Si no se llegaba a enviar, el dato nuevo tapaba al dato anterior.
Esto es problemático principalmente porque hay operaciones referentes a la tarjeta que llevan más tiempo que el tiempo entre mediciones.

1- cambio de sector: cuando se llena un sector, cerrarlo y abrir el siguiente
2- cambio de bloque: las tarjetas “preparan” bloques grandes (algunos kb’s, ~100 sectores) para optimizar la velocidad media.

Para atacar ambos problemas implementé un buffer circular. La idea es poder guardar la mayor cantidad de datos posibles (limitado por el RAM del procesador), no perder los datos medidos en estos tiempos muertos y enviarlos cuando se pueda.



Observación: para que pueda vaciarse el buffer la frecuencia de adquisición debe ser menor a la de transmisión.

Lógica de transmisión

Se lleva un contador con la cantidad de datos que hay para enviar. Luego de cada medición en modo grabar, este contador aumenta en 1.
En el loop while, se verifica primero que se esté en modo grabar (si se está en modo trigger no hay que guardar los datos), y si hay bytes para enviar. Si se cumplen ambas condiciones se envian los dos bytes correspondientes para calcular en qué posición del buffer está:

buffer_out = buffer_in-audio.bufferBytes+1;

Donde *buffer_in* es el índice del último dato entrante y *audio.bufferBytes* es la cantidad de datos que están disponibles para enviar.

Dentro del código de escritura (sdWrite):
Si no se llegó al final de un sector, no pasa nada.
Si se llegó al final de un sector Y se llegó al último sector correspondiente a la medición (terminó la medición) entonces se vuelve a modo trigger.
Si se llegó al final de un sector pero la medición sigue, se cierra el sector y se abre el siguiente.

Limitaciones y compromisos

Frecuencia de adquisición

La frecuencia de adquisición está definida por el timer. El tiempo que tarda entre interrupciones consecutivas es el tiempo entre mediciones.
El límite de la frecuencia de adquisición viene determinado esencialmente por la velocidad de transmisión. Si soy capaz de transmitir 1.000 datos por segundo, no tiene sentido adquirir 2.000 (es más, debido a la lógica de interrupciones, se va todo al carajo).
En la práctica, lo mejor es que la frecuencia de adquisición sea un poco menor a la frecuencia de transmisión, para que si por alguna razón se acumulen datos para enviar, el procesador sea capaz de ir reduciendo el número.

Hay que tener en cuenta también que el cambio de sector lleva tiempo (más que una medición). Para atacar este problema implementé un buffer.

En resumen: buffer para cuando se acumulan datos (cambio de sector) + adquisición un poco más lenta que transmisión (para poder deshacerme de estos datos acumulados).

Tiempos muertos

Área oscura. Por lo que probé e investigué (google) las tarjetas uSD están optimizadas para mantener cierta velocidad media de escritura (por ejemplo, las que dicen “Clase 4” aseguran 4Mb/s mínimo). La cuestión es que para llegar a esta velocidad mínima tienen que realizar operaciones sobre bloques grandes de memoria (varios kbs),

Entonces, cada tanto (cada cierta cantidad de sectores escritos), la tarjeta se toma un tiempo para preparar un bloque. Acá aparece el problema cuando uno tiene poco RAM (nosotros tenemos 512b). En una computadora está todo bien, porque tiene un buffer infinito que después escupe, pero nosotros podemos guardar 512b nomás (que dependiendo de la frecuencia de adquisición representa cierto tiempo definido). Si la tarjeta tarda tanto tiempo que el buffer se rebalsa, se empieza a perder información. Y eso es lo que pasa. Haciendo pruebas con una tarjeta particular (kingston 4gb) encontré que cada 44 sectores (0.7 segundos) se toma 40ms. No es tan grave pero es para tener en cuenta que nos estamos comiendo el 6% del tiempo.

Tener en cuenta que esto también juega con la frecuencia de adquisición. Cuanto más alta la frecuencia de adquisición, mayor es la fracción de tiempo perdida.

Implementación código final

Como repaso y para darle un cierre a toda la programación, vamos a ver el programa completo en funcionamiento.

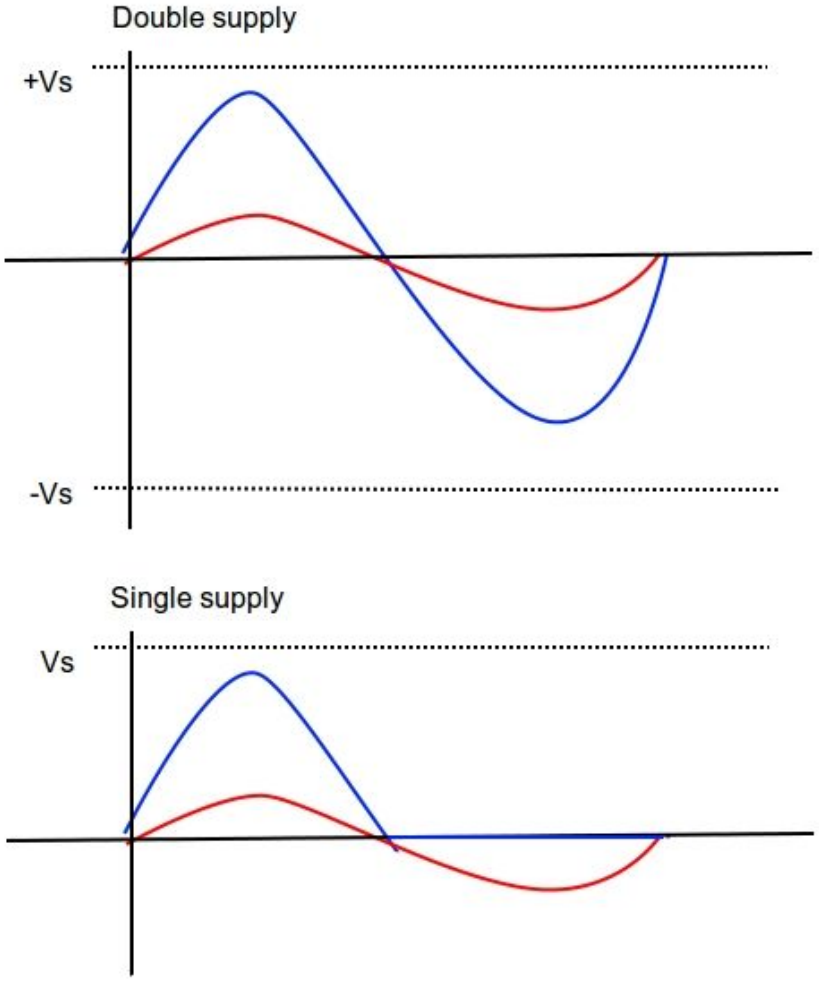
- Usar el “Firmware v0.4” en la carpeta *Archivos*.
- Repasar el código (definiciones, trigger, flujo)
- Definir donde esta el LED (en el launchpad esta en P1.0)
- Descargarlo a un procesador
- Conectar tarjetero a lo pines correspondientes.
- Debuggear en placa (con tarjetero conectado)
- Pruebas: con y sin tarjeta, como detectar tiempos muertos (opc)
- Desconectar y conectar alimentación, ver que funciona solo
- Recuperar los datos guardados en la tarjeta

Diseño circuito

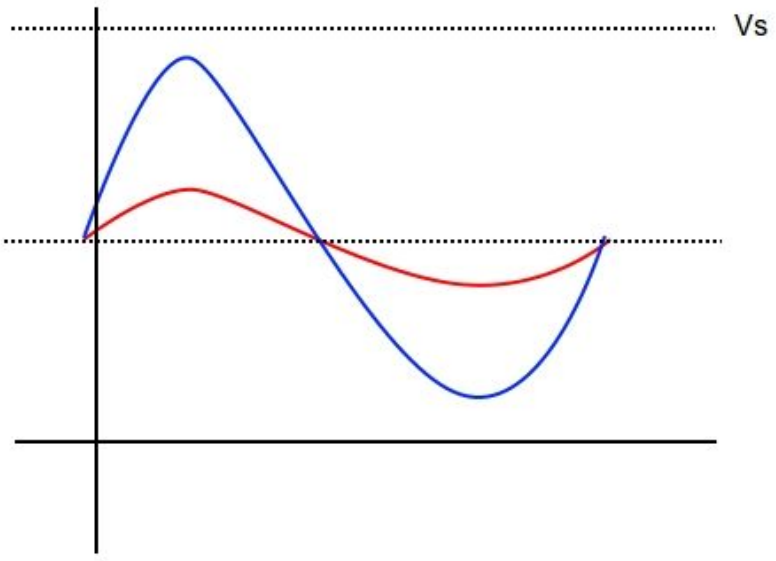
Lo que sigue es para el circuito que está testeado y andando ya. Es una versión que usa un Op-amp de dos canales, single supply (OPA2336 o LM358).

Op amp single supply

Single supply quiere decir que puede ponerse el V- del op amp a tierra, se conecta V+ a Vs (pila) y V- a 0V. Los double supply deben conectarse a +Vs y -Vs.
Como siempre, el output está limitado entre los valores de alimentación, o sea que en single supply **la salida está entre 0V y Vs.** Hay que tener esto en cuenta en el diseño, porque **la parte negativa de la entrada se pierde**. En la figura siguiente la señal roja es la entrada, la azul es la salida:



Para evitar esto se puede referenciar la entrada a Vs/2. De esta manera la parte negativa de la entrada pasa a estar entre 0V y Vs/2, por lo que el op amp la puede amplificar.



Las dos opciones de op amp tienen características distintas:

	Consumo	Rango	Ancho de banda
OPA2336	~20uA/canal	0 - Vcc	100kHz
LM358	~500uA/canal	0 - Vcc-1.5V	1MHz

En cuanto al consumo, no creo que sea crítico el cambio, teniendo en cuenta (ver más adelante) que en total se está consumiendo ~15-20mA
El rango del OPA es óptimo, permite sacar tensión en todo el rango de alimentación, pero tiene bajo ancho de banda. El ancho de banda (GBW) define hasta que frecuencia puede amplificar:

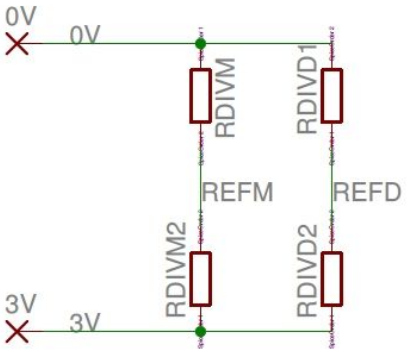
GBW = G * f_max

Donde G es ganancia y f_max, la maxima frecuencia. Frecuencias mayores son amplificadas pero cada vez menos. Para el OPA, poniendo la ganancia en 200 (para músculo) la frecuencia máxima queda en 500Hz. Si queremos tener hasta 3kHz, la ganancia máxima es 33.

La ventaja del LM es que nos permite subir la ganancia sin problema. La desventaja es el menor rango de salida.

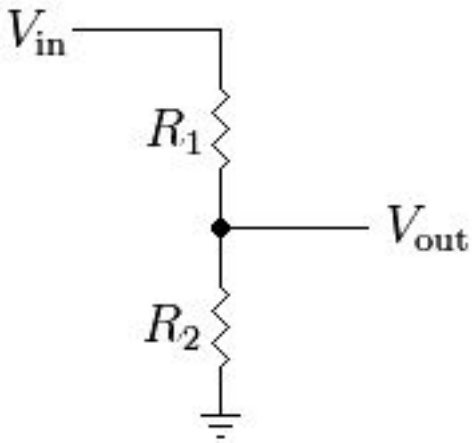
Divisor de tensión

Para poder referenciar las señales a Vs/2, se puede poner un divisor de tensión, simplemente dos resistencia en serie entre Vs y 0V. Esto genera dos tensiones de referencia REFM para el micrófono y REFD para el amplificador diferencial.



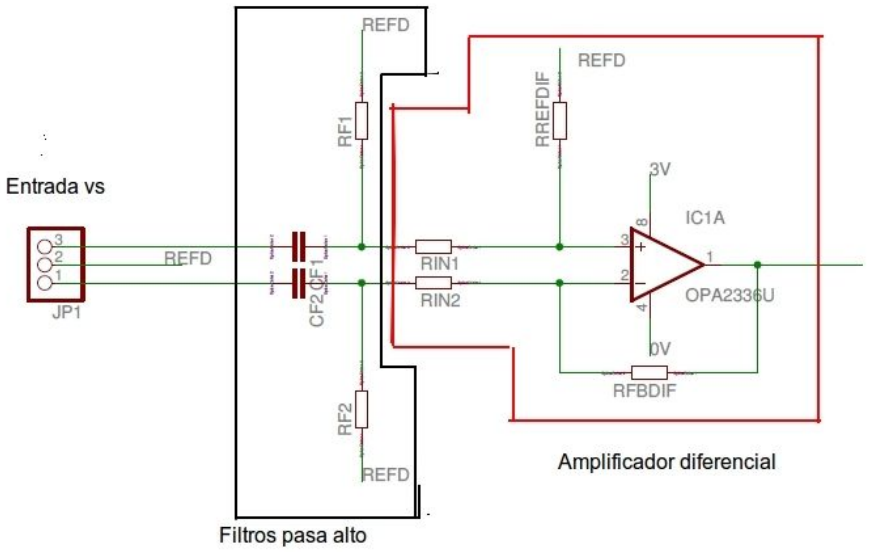
Estoy usando divisores distintos más que nada para evitar cross talk entre los canales.

Edit (25/8):
Al usar el LM358 y para aprovechar completo el rango de salida, ya no es conveniente referencias las señales a Vcc/2 (en general conviene referenciar a Vmax/2). Para corregir esto hay que cambiar levemente el divisor, recordando que



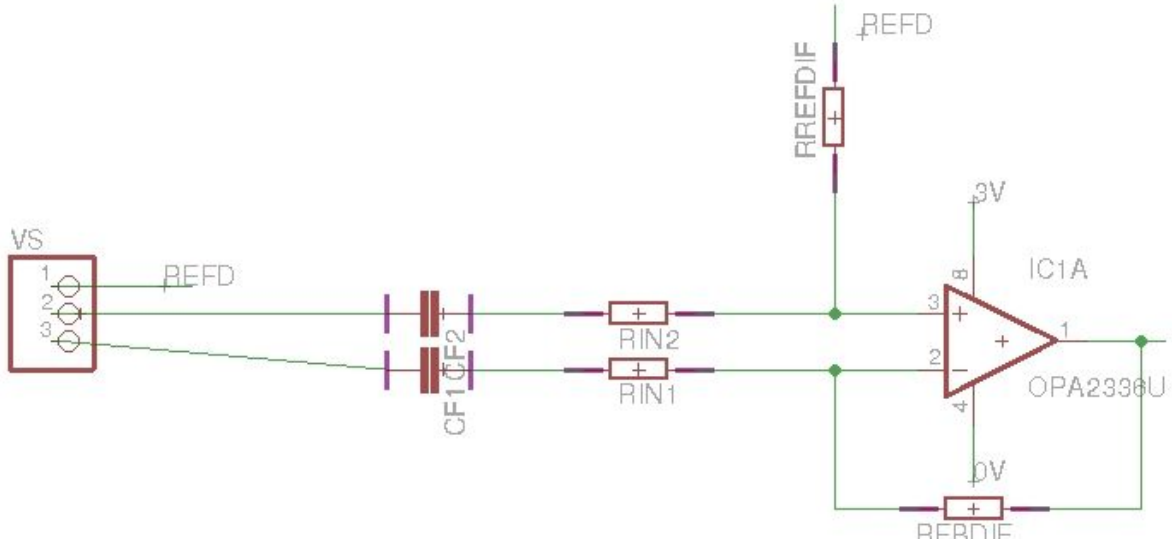
$$V\,out \;=\; V\,in * \frac{R2}{R1+R2}$$

Diferencial



Tres pines de entrada, dos para señal de músculo, uno para el tercer electrodo. Este tercer electrodo es una duda no resuelta aún, si está bien ponerlo a REFD.
El recuadro negro es de los filtros pasa alto, esto está **bien y testeado** que **descargan a REFD, no a tierra**. Si RIN1 = RIN2 y RREFDIF = RFBDIF, entonces la ganancia es RFBDIF/RIN1.
Valores usados:
C = 220nF, R = 4,7k que da una frecuencia de corte de $f = \frac{1}{2\pi * RF * CF} = 154Hz$
RIN = 4,7k, RREFDIF = RFDIF = 1M, lo cual da ganancia G = 212 (similar al 220 que usamos en los preamp de los finches).

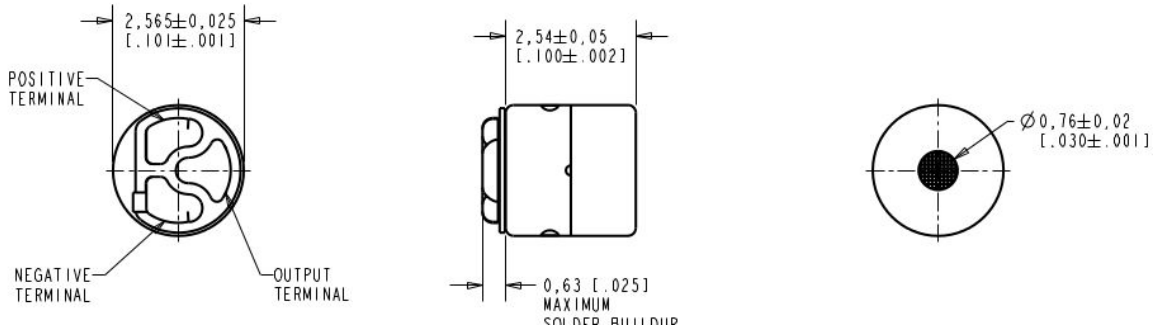
Edit (25/8), respecto a los filtros:
Las resistencias RF1,2 no hacen falta



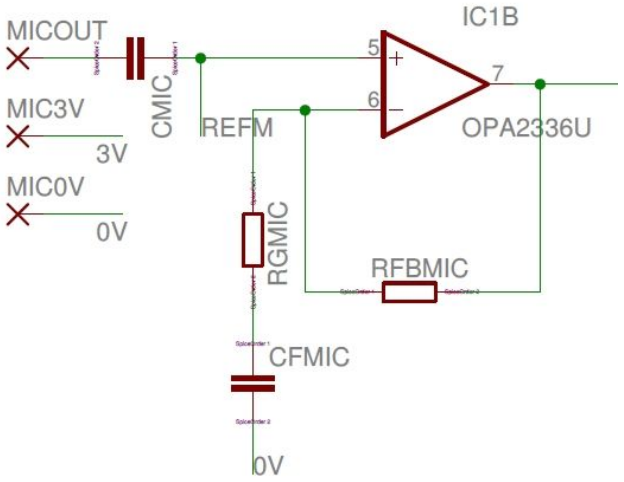
La frecuencia de corte es $\frac{1}{2\pi * RIN * CF}$.

Micrófono

Los micrófonos mini son de la línea Knowles (ver datasheet FG-23329-C05.pdf). Tienen 3 terminales, alimentación, tierra y output.



El micrófono se alimenta con los 3.3V del regulador.



La salida del micrófono no está referenciada a Vs/2, por lo que hay que filtrar una componente de continua antes de amplificar. Para filtrar se agregan capacitores CMIC y CFMIC.

Luego de CMIC se conecta a REFM (tensión de referencia para el micrófono). Esto hace que la salida del op amp esté referenciada a este valor. Por ejemplo, si la señal de entrada tiene una amplitud de 10mV y la ganancia es 100, la salida tiene amplitud 1V, o sea que va de 1.5V - 0.5V a 1.5V + 0.5V.

La frecuencia de corte está dada por RGMIC y CFMIC, como en un filtro: $f_c = \frac{1}{2\pi RC}$

Valores usados:

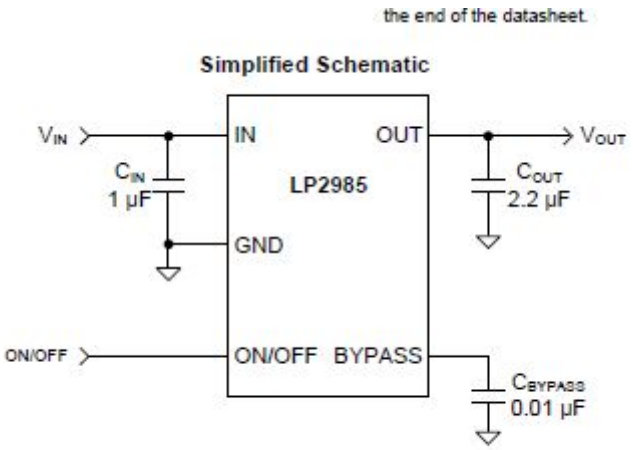
CMIC = CFMIC = 1uF

RGMIC = 10k RFBMIC = 1M

Da una ganancia de 100 y frecuencia de corte de 16Hz (se podría subir si hiciera falta)

Regulador de tensión

Una de las complicaciones con que me encontré fue que no arrancaba la tarjeta cuando alimentaba el circuito con una pila de 3V. El procesador arrancaba bien, pero no lograba inicializar la tarjeta.
Haciendo pruebas con alimentación USB (~3.4V) vi que así funcionaba, aún cuando en todas las especificaciones dicen que las tarjetas andan de 2.7V a 3.6V.
Mi solución fue usar usar un regulador de tensión. El que estoy usando ahora es el integrado LP2985-N. La función es mantener Vout (lo que va a alimentar el procesador, op amp, micrófono y uSD) constante en ~3.3V. Se alimenta con Vin, que tiene que ser mayor a ~3.3-3.4V
http://www.ti.com/lit/ds/symlink/lp2985-n.pdf



Tiene input de hasta 16V, y un pin ON/OFF (estoy pensando si lo puedo usar más adelante)
Dos cosas buenas de este regulador:
Bajo dropout (LDO): el dropout es la mínima diferencia entre la entrada y la salida. Por ejemplo, si Vout = 3.3V y el dropout es 1V, el mínimo voltaje para que funcione es 4.3V. El dropout de este regulador es muy muy bajo, depende de la corriente que esté saliendo, del orden de decenas de mV a lo sumo.
Bajo consumo: la corriente a tierra es menor al 1% de la corriente de carga, o sea que prácticamente toda la corriente que se extrae de la batería va al circuito.

Procesador (pin reset)

Al encender el procesador se produce un POR (se resetea)

A POR is a device reset. A POR is only generated by the following three events:

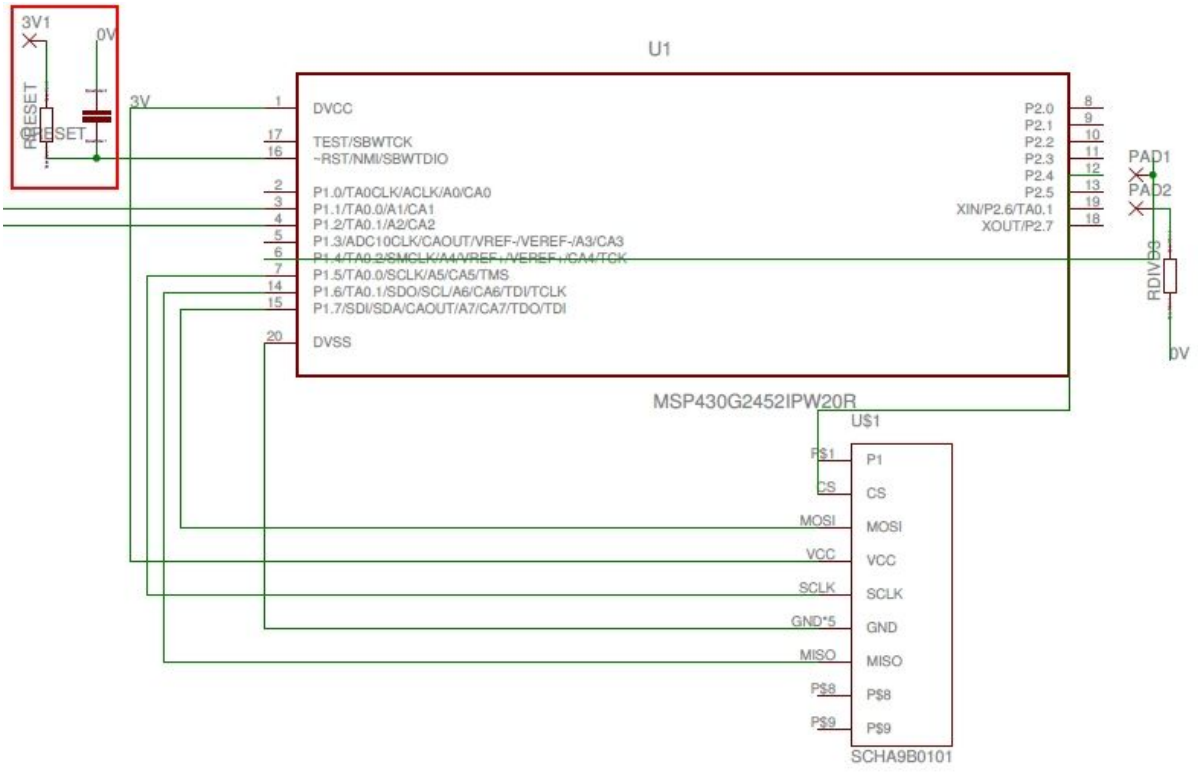
- Powering up the device
- A low signal on the $\overline{\text{RST}}$/NMI pin when configured in the reset mode
- An SVS low condition when PORON = 1.

Notar que c**uando la señal del pin RST baja, se resetea.**

2.2.1.1 Reset/NMI Pin

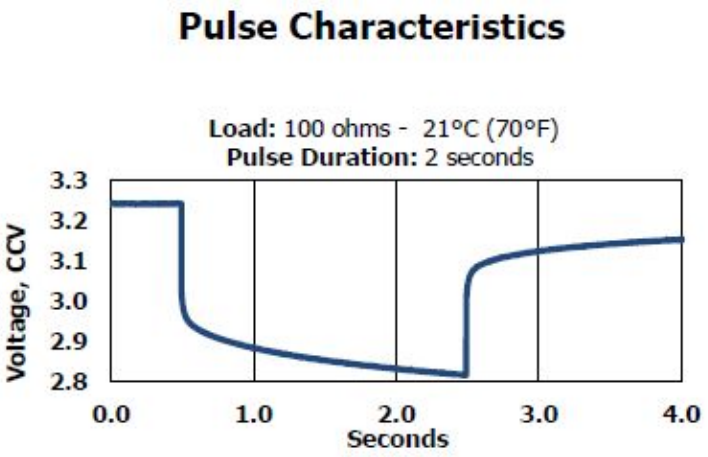
At power-up, the $\overline{\text{RST}}$/NMI pin is configured in the reset mode. The function of the $\overline{\text{RST}}$/NMI pins is selected in the watchdog control register WDTCTL. If the $\overline{\text{RST}}$/NMI pin is set to the reset function, the CPU is held in the reset state as long as the $\overline{\text{RST}}$/NMI pin is held low. After the input changes to a high state, the CPU starts program execution at the word address stored in the reset vector, 0FFFEh, and the RSTIFG flag is set.

Esto dice que mientras el pin RST no esté high, el CPU no arranca. Lo que hay que hacer siempre es poner una resistencia pull-up en el pin (pull-up quiere decir que va del pin a Vcc, es decir que lo “levanta”). Esto cumple la función de mantener el pin RST alto, para que encienda y no se resetee.
Se suele poner también un capacitor del pin a tierra.



Baterías

Las pilas botón no funcionaron. Sospecho que la razón fue el tipo de descarga. Aún cuando en la datasheet da una capacidad (tomo como ejemplo una CR2032) de 240mAh, esto está medido con una descarga continua por una resistencia de 15k (corriente 0,2mA).
En nuestro caso, el consumo no es continuo, sino que por la naturaleza del proceso de escritura en la tarjeta, hay muchos picos de corriente.
Acá fue un punto donde que encontré medio oscuro, porque los fabricantes de memorias comerciales no dan datasheets tan completas (a veces ni siquiera hay) como las que estamos acostumbrados. Investigando en foros y demás, parece que la programación interna de las tarjetas (esto depende de cada fabricante) hace que requieran picos de corriente mientras escriben.
El problema está en el comportamiento de las pilas con los pulsos de corriente



Lo que creo que pasó (y la diferencia con las baterías Lipo parece confirmarlo) es que las pilas no están pensadas para descargarse por pulsos y/o corrientes altas y que esto reduce su capacidad.

Otra evidencia en esta dirección es que hice una prueba poniendo baterías en paralelo para reducir la amplitud del pulso de corriente requerido para cada batería, y la vida útil aumentó más del doble.

La diferencia con las baterías Lipo (acá si que es casi imposible encontrar datasheets porque se usan más que nada para juguetes) es que estas están pensadas específicamente para cosas que puedan llegar a requerir mucha corriente, y no necesariamente constante.

Una característica que suelen dar los fabricantes es C, el rate discharge. Es fácil entenderlo con un ejemplo.
Si una batería de 150mAh es de 20C, quiere decir que puede descargarse con una corriente de hasta 150mA*20 = 3A.
La batería que probé era justamente 150mAh 20C.

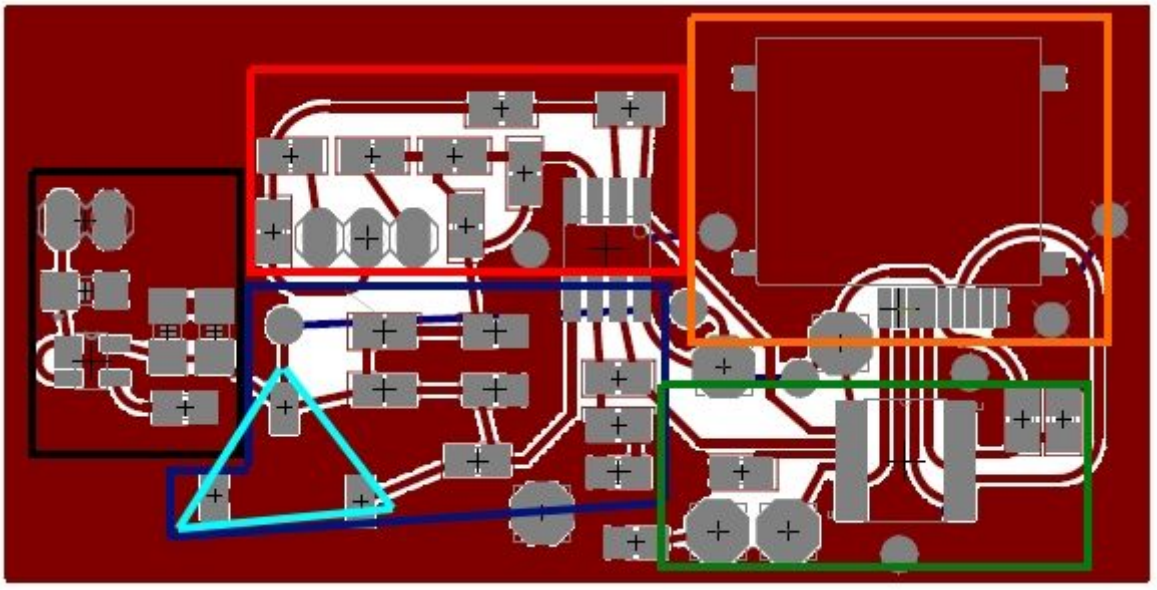
La otra ventaja es que son recargables. Compré un integrado:
http://articulo.mercadolibre.com.ar/MLA-617052410-cargador-de-baterias-de-lithio-lipo-tp4056-con-protector-_JM
Datasheets:
http://www.haoyuelectronics.com/Attachment/TP4056-modules/TP4056.pdf (cargador)
http://www.haoyuelectronics.com/Attachment/TP4056-modules/DW01-P_DataSheet_V10.pdf (protección)
Se conecta por USB a la PC, y la batería a dos terminales. Falta FOTO

La carga se suele recomendar a 1C (para la batería de 150mAh: 150mAh) máximo. El integrado tiene una resistencia que permite regular esta corriente y se puede cambiar (cambié la que venía por una de 10k, que da corriente de 130mAh).

Diseño PCB

Todos los diseños están hechos en Eagle (http://www.cadsoftusa.com/download-eagle/).
Las últimas versiones están en la carpeta “PCB”, tanto en 1 cara (v4) como en dos (v5)

Una cara



Recuadro negro: regulador + capacitores
Rojo: entrada musculo, filtros op amp 1
Azul: microfono, divisor, opa amp 2
Celeste: microfono (3.3V, 0V, output)
Naranja: micro sd
Verde: procesador, led, pin reset (derecha)

Respecto al pinout de la micro sd, ver
http://www.alps.com/prod/info/E/HTML/Connector/microSD_Card/SCHA/SCHA9B0101.html
Tambien esta subido a la carpeta

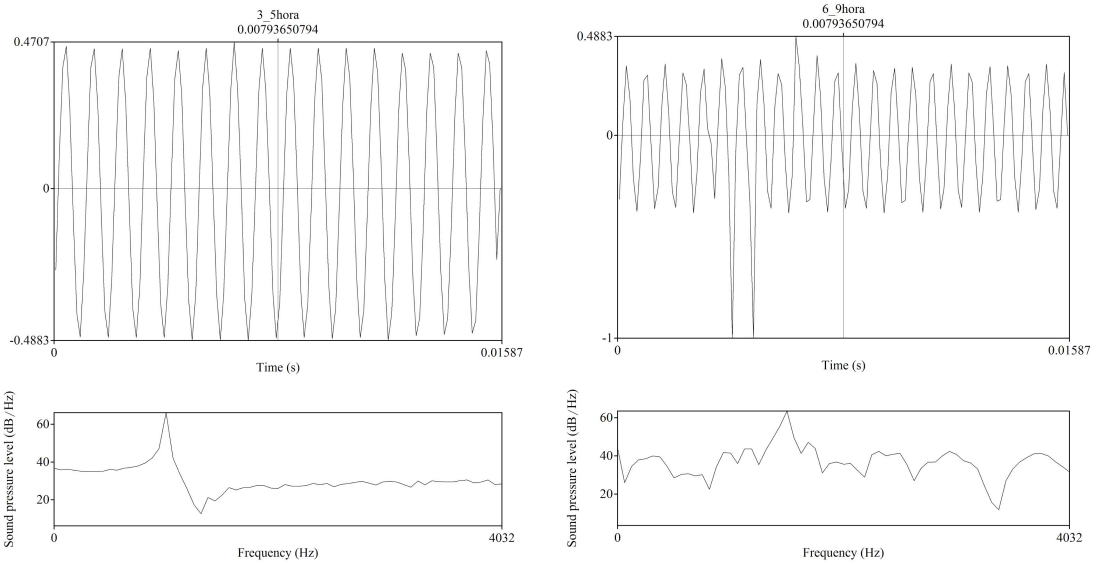
El peso de la placa (sin componentes) es de 5,1 gramos.

Prueba adquisición

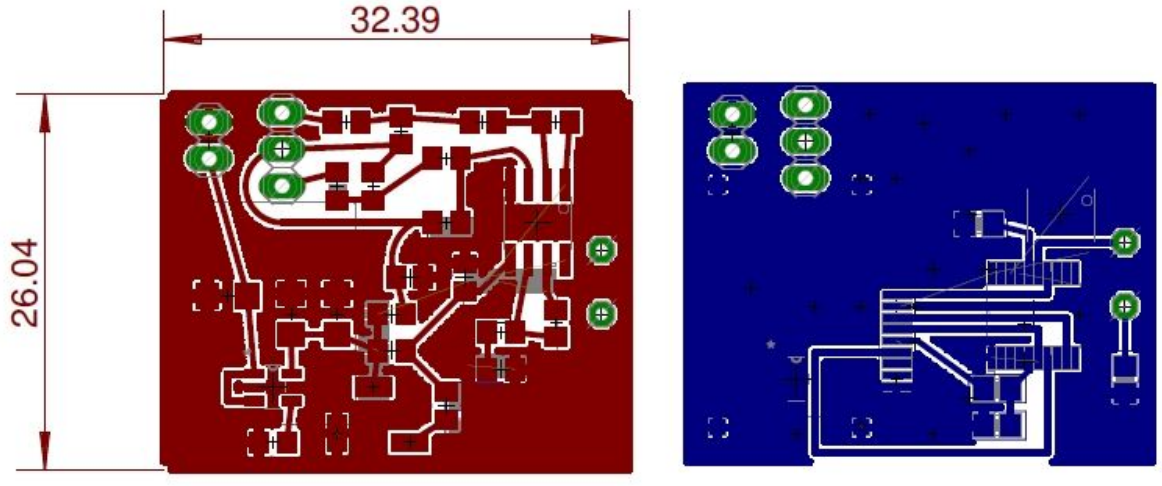
Hice la prueba de adquisición en dos canales, con el micrófono conectado y el generador alimentando con una señal senoidal de +-10mV en lugar de músculo. Se hizo para el peor caso posible en que se mide continuamente (umbral trigger = 0) para determinar el intervalo mínimo de medición.

Cometí un error con el micrófono (conecté al revés la alimentación) por lo que aunque se midió algo y se grabó en la tarjeta, no era sonido. La señal del generador se grabó correctamente hasta las 7 horas (6:55 más exacto). Para verificar esto fui cambiando a intervalos la frecuencia de la señal del generador. En todos los casos se grabó correctamente y es posible reconstruirla y medir correctamente la frecuencia.

En la carpeta “Prueba duración LiPo” están los resultados.
En “leeme.txt” se detallan las frecuencias del generador para cada momento.
Los archivos tienen por nombre la hora a la que corresponden (contando hora 0 = comienzo de medición). Los “*.dat” son los obtenidos con el programa de lectura de sd, los “*.wav” son estos mismos pasados a wav con matlab.

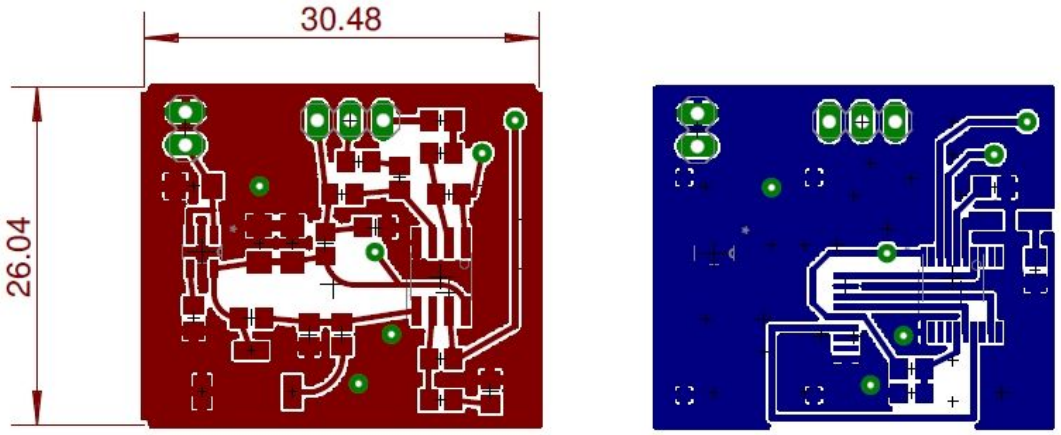


Dos caras

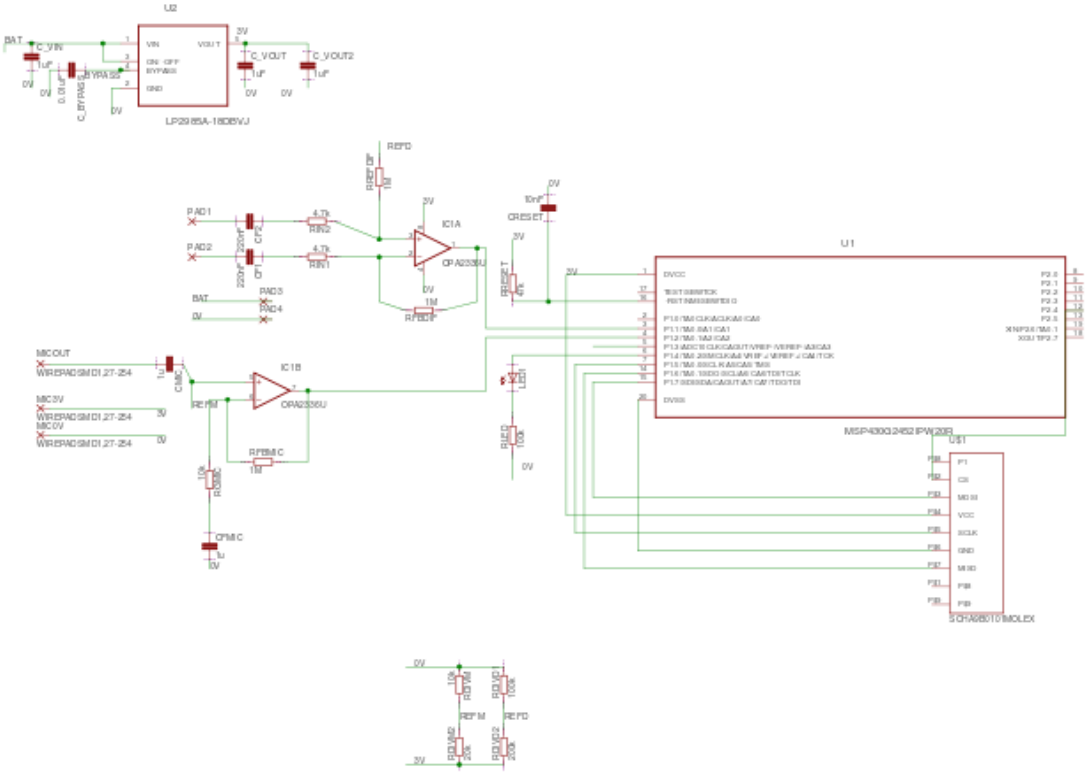


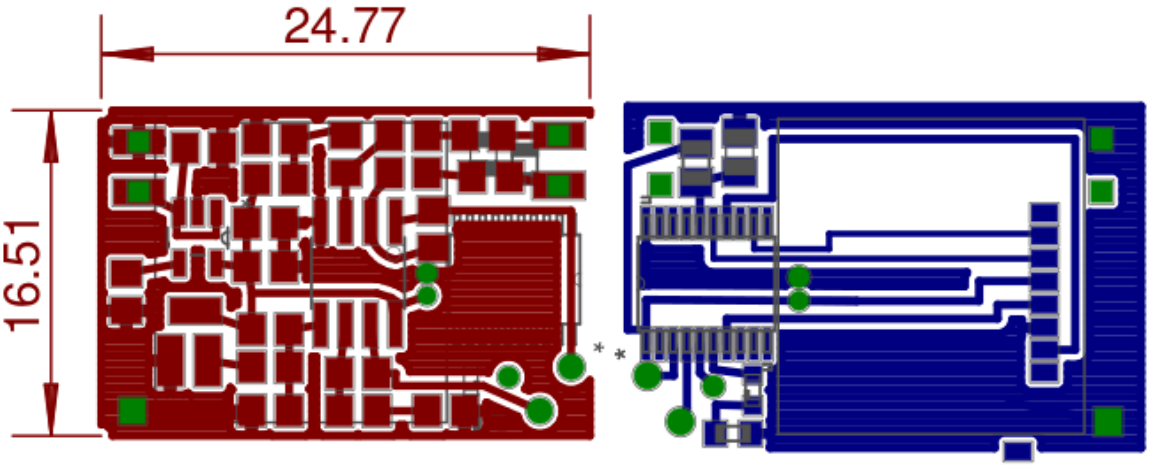
Pasan abajo el procesador, la tarjeta y la batería (en proceso). Se reduce aproximadamente a la mitad el tamaño.

Edit (26/8):
Versiones terminadas (que mandé a imprimir) (2016)



Placa versión 2017





A futuro

Programación en placa

Una gran desventaja/molestia, es que una vez que se programó el procesador hay que soldarlo a una placa (queda descartado usar un zócalo por el tema del peso). O sea que si se cometió un error en la programación o se quiere modificar algo, hay que des-soldar el procesador, lo que tiene el riesgo de arruinar la placa (que salten las pistas), o de ir arruinando el procesador (o directamente hacerlo pelota).

Una solución sería hacer placas que den la posibilidad de programar el procesador ahi, sin usar el Launchpad. Ya tengo el esquemático del launchpad y creo que no debería ser tan complicado implementarlo

Switch

Algo que me anda preocupando respecto a las baterías. Estas baterías no pueden (pueden pero no se debe) descargarse por debajo de los 3V. Me gustaría implementar un switch que detecte cuando la batería llega a una tensión crítica y que desconecte todo el circuito para no seguir gastandola.
Por lo que investigué hay tres caminos:
1- Software: el procesador incluye un supervisor de voltaje (SVS) configurable, que produce un reset.
2- ON/OFF del regulador: el regulador que estoy usando tiene un switch, si el voltaje en el pin supera un valor se enciende, si no no.
3- MOSFET: esta me llama la atención pero no se usar bien transistores. La idea sería encontrar un transistor adecuado (gate voltage) que se cierre cuando la tensión de la batería baje demasiado.