# DISEÑO Y MONTAJE DEL PROTOTIPO

Resumiendo todo lo estudiado en los capítulos anteriores del proyecto:

El dispositivo electrónico va a estar formado por una Raspberry Pi Zero que va a ser alimentado por una sola celda de batería LiFePo4 mediante un HAT llamado LiFePo4wered/Pi, que está diseñado para dicha tarea.

Dicha celda será cargada sin contacto directo, mediante un cargador de inducción que nos proporciona la tensión necesaria para cargar la batería.

Por un lado vamos a tener el montaje del sistema de carga, y por otro lado, el montaje del sistema de alimentación.

## SISTEMA DE CARGA

Para el funcionamiento de la carga, es necesario un transformador, para transformar la corriente alterna (AC) que proporciona cualquier enchufe convencional de una vivienda (100-240V) en una salida de 9 a 12V de corriente continua (CC).

Éste irá conectado a la entrada del circuito inductor. (En este proyecto se utiliza un transformador que ofrece 12V a la entrada del circuito inductor).

Una vez ofrecida la tensión necesaria al circuito inductor, se conecta la parte del circuito receptor al módulo de carga de la batería, y cuando las bobinas se unen comienza la carga.



1. Módulo de carga inalámbrico.

El módulo de carga, ya integrado en el HAT de alimentación, se basa en un proyecto llamado LiFePO4wered/USB también diseñado por silicognition LLC y de venta en tindie.

Data sheet disponible en ANEXO II.<https://lifepo4wered.com/files/LiFePO4wered-USB-Product-Brief.pdf>

Es un módulo de carga diseñado para una correcta carga de baterías LiFePO4 , que permite a LiFePo4wered/Pi estar en estado de carga mientras se encuentra en funcionamiento y alimentando la Raspberry Pi, evitando sobrecargas y proporcionando seguridad a la batería.



1. Módulo de carga. Recuperado de Tindie.

Características:

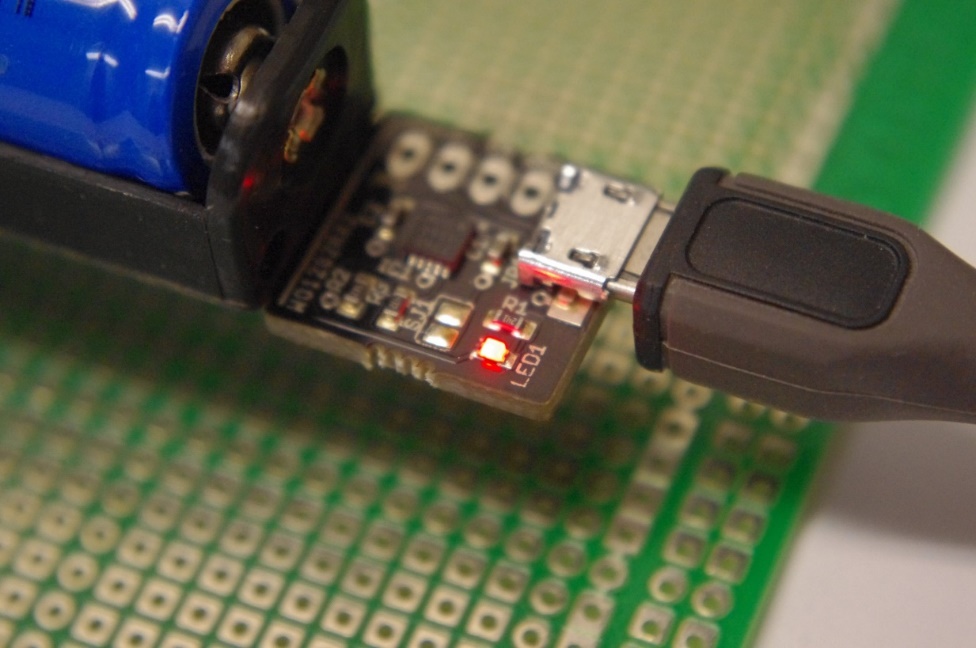
Cuando el cargador está desconectado ofrece una fuga ultra baja de 0,5µA.

Ofrece una carga mediante un conector de carga micro USB, por lo que tiene compatibilidad con cargadores habituales de teléfonos móviles.

La tensión de entrada puede oscilar entre 4.5V y 6.5V, siendo el valor más corriente a usar una tensión de 5V (que es la tensión que ofrece a la salida el cargador de inducción utilizado).

La corriente de carga es 245mA tardando en cargar una batería totalmente descargada 2.5h, pero ofrece la posibilidad de puentear SJ1 ofreciendo una corriente de carga de 480mA, tardando la mitad en cargar (carga rápida).

Tiene una tensión mínima de protección de 1,45V, así como una corriente mínima de corte de 25mA.



1. Conexión carga micro USB. Recuperado de Tindie.

## SISTEMA DE ALIMENTACIÓN

Para la conexión del HAT a la Raspberry Pi, es necesario conocer cada una de las entradas y salidas que nos ofrecen y como van conectadas para un correcto funcionamiento.

Los pines del modelo de Raspberry Pi Zero son los mismos del modelo B+. Son conocidos como GPIO.

Consta de 40 pines de entrada y salida y cada uno de ellos está configurado para una función específica.

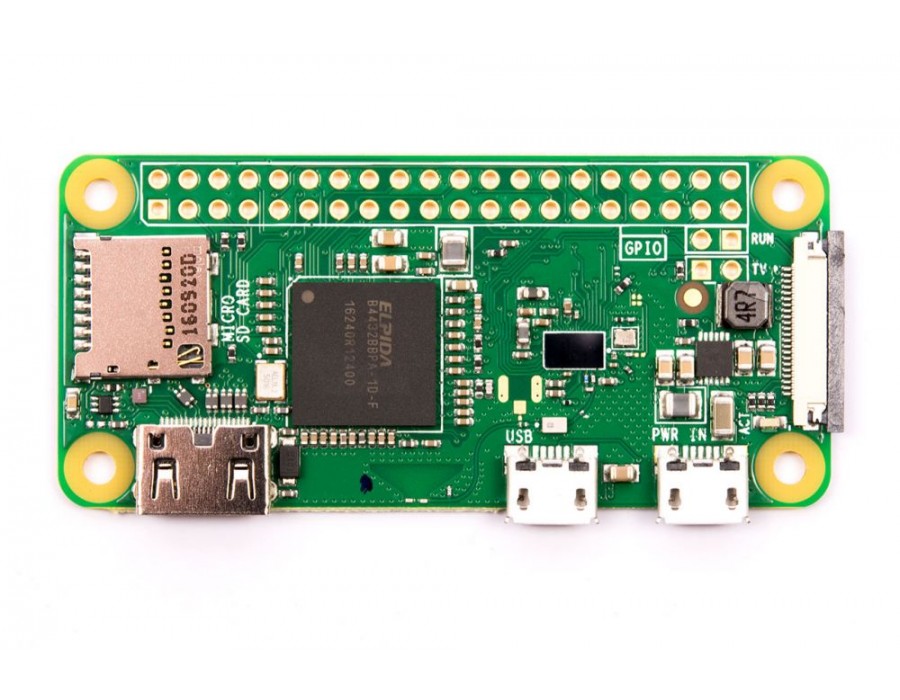
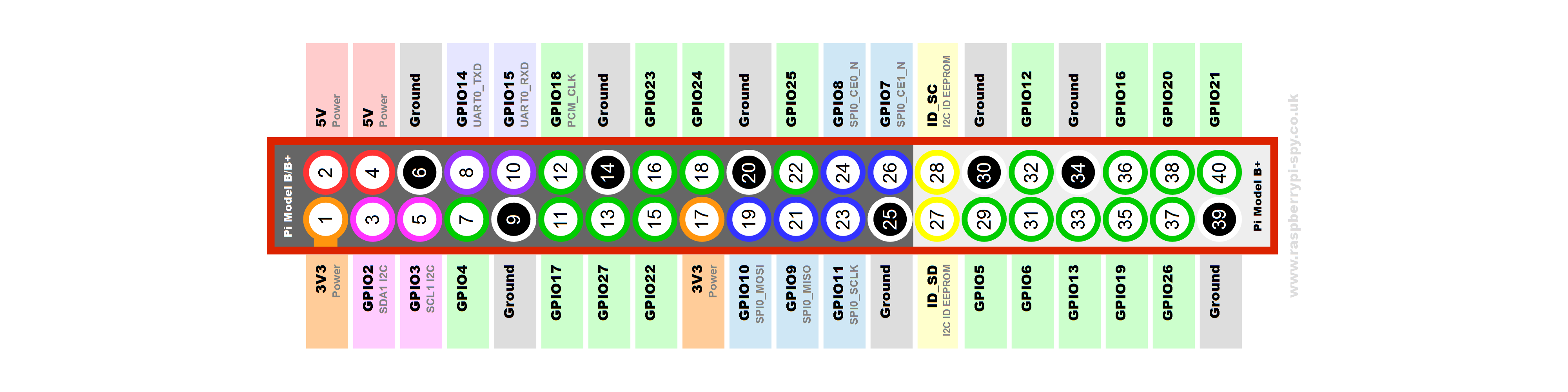
### INSTALACIÓN HARDWARE

En el ANEXO I se puede ver que los pines utilizados por este HAT van a ser los 8 primeros pines de la Raspberry Pi Zero.

Para conectar el módulo es necesario apagar Raspberry Pi, desconectar cualquier tipo de alimentación que pueda tener, y conectar el LiFePo4wered/Pi a los 8 primeros pines de Raspberry Pi Zero. Una vez instalado el modulo, basta con tener presionado el botón táctil unos segundos (2s) y se enciende.

Este módulo espera unos segundos antes de encender para evitar que se encienda de forma accidental.

¿Qué función tienen los pines?



1. GPIO Raspberry Pi Zero

* PIN1 – 3.3V

Es un pin de alimentación a 3.3V que tiene una corriente máxima de 50mA.

El HAT lleva este pin desconectado.

* PIN2/PIN4 – 5V

Es un pin de alimentación a 5V que va conectado directamente con la alimentación de entrada de la Raspberry Pi.

En el HAT es el pin mediante el cual el LiFePO4wered/pi proporciona energía a la Raspberry Pi.

* PIN3 – SDA

Es uno de los pines de comunicación I2C.

I2C (Inter Integrated Circuit) son unos pines de Raspberry que sirven para comunicar controladoras con diferentes periféricos. Es un bus maestro- esclavo, y lo forman dos líneas de señal (la señal de reloj CLK y la línea de datos SDA).

El Pin 3 es el pin de datos de I2C utilizado por Raspberry para comunicarse con LiFePO4wered/Pi.

* PIN5 – SCL

Es la señal de reloj del bus I2C que utiliza Raspberry para comunicarse con LiFePO4wered/Pi.

* PIN6 – GND

Es el pin de referencia a tierra. Hay 8 pines de masa conectados entre sí, pudiendo usar cualquiera de ellos.

* PIN7 – GPIO4

En el HAT va desconectado.

* PIN8 – TX0

Este pin se puede usar como pin de transmisión UART (receptor/transmisor universal asíncrono) o como serial para mostrar una consola.

Pero LiFePO4wered/Pi no lo usa como tal, sino que solo monitorea el nivel lógico a través de él.

Durante el estado de apagado se controla esta señal, y cuando esta deja de conducir, la Raspberry Pi habrá terminado su apagado.



1. LiFePO4wered/Pi conectado a Raspberry Pi Zero

### INSTALACIÓN SOFTWARE

Para la recogida de datos sobre el estado de la betería que ofrece este HAT, es necesaria la instalación de bibliotecas disponibles en GITHUB.

Pero antes de ver eso, es necesario saber el sistema operativo con el que se va a trabajar.

#### Linux

El sistema operativo libre Linux o GNU/Linux, es el software necesario para que el ordenador te permita realizar diversas acciones (editar textos, internet…).

Es posible utilizar Linux con una interfaz gráfica como usa Windows o MacOS, o mediante una ventana de comandos como usa DOS.

Raspbian o Debian son distribuciones de este sistema operativo.

Para la utilización de Raspberry Pi, Raspbian es el sistema operativo oficial, y en la página oficial se ofrece NOOBS que es un instalador de Raspbian para poner en funcionamiento la Raspberry Pi.

Una vez instalado y puesto en marcha, se va utilizar este sistema desde la ventana de comandos, que te permite una fácil navegación entre directorios, instalar, crear archivos, programas, leer y escribir texto, etc. A través de unas ordenes básicas para realizar las diferentes acciones necesarias.

#### Instalación de Bibliotecas

Para obtener los datos facilitados por LiFePO4wered/Pi sobre el estado de energía de la batería es necesario ejecutar el software proporcionado.

Este software proporciona acceso a unas bibliotecas compartidas que da un acceso fácil al LiFePO4wered/Pi con órdenes de usuario, así como proporciona un programa CLI que permite al usuario configurar desde la línea de comandos.

El paquete del software se puede encontrar el Github:

<https://github.com/xorbit/LiFePO4weredPi>

Se ejecuta primero la siguiente orden para asegurar que las herramientas de compilación y Git están instaladas:

Sudo apt-get –y install build-essential git

Luego, se clona el paquete de software de Github:

git clone <https://github.com/xorbit/LiFePO4weredPi.git>

Y se entra en el directorio recién creado llamado LiFePO4wered-Pi con la orden “cd”.

Dentro de este directorio se construye el software ejecutando:

python build.py

Y se instala ejecutando:

sudo ./INSTALL.sh

Esto no solo instala el software en el sistema sino que realizará la configuración necesaria para su funcionamiento como habilitar el bus I2C y el GPIO UART.

# ENSAYOS

## RECOGIDA DE DATOS DE LA ALIMENTACION DE RASPBERRY PI

En este capítulo se van a describir los programas usados, tanto para la recogida de datos de la alimentación de la Raspberry PI Zero, como para los datos de la batería cuando esta está alimentando a la Raspberry Pi Zero.

El LiFePO4wered/Pi permite acceder a un conjunto de registros desde la Raspberry Pi a través de bus I2C. Por defecto la dirección del dispositivo de 7 bits es 0x43 pero puede cambiarse en caso de conflicto.

Como se ha comentado en el capítulo anterior, se ha instalado con el paquete del software una herramienta llamado lifepo4wered-cli que permite obtener y establecer valores de los registros que se quieren conocer o establecer, sin tener que conocer detalles de implementación.

Con esta herramienta, basta con escribir la orden lifepo4wered\_cli get, y el registro que se quiera obtener, para que devuelva ese valor deseado. O bien, lifepo4wered-cli set, para establecer un valor (mostrando los datos en mV o segundos).

Enumeración de los registros a los que se puede acceder obtenidos del ANEXO I:

* I2C\_ADDRESS

Dirección del bus de 7 bits del dispositivo.

Valor por defecto 0x43.

* LED\_STATE

Se usa para configurar el estado del LED cuando Raspberry está encendida.

Esto se hace para ahorrar energía o para indicar el estado de un programa.

Los estados posibles son: LED apagado, LED encendido, LED pulsado o flash rápido.

Valor por defecto encendido.

* TOUCH\_STATE

Indica estado del botón.

Si el botón no se toca el valor va a ser cero, y distinto de cero si es presionado.

En caso de presionar durante unos segundos el sistema se apagará.

* TOUCH\_CAP\_CYCLES

Número total de ciclos de carga y descarga generados y medidos.

* TOUCH\_THRESHOLD

Indica el umbral del toque táctil y es uno de los parámetros que se pueden personalizar para ajustar la sensibilidad del toque táctil.

Valor por defecto 12.

* TOUCH\_HYSTERESIS

El sistema de detección táctil tiene una histéresis para garantizar la detección táctil y se puede personalizar en caso de querer equilibrar la sensibilidad.

Valor por defecto 2.

* DCO\_RSEL y DCO\_DCOMOD

Estos valores vienen establecidos de fábrica para que el reloj del microcontrolador funcione a 12MHz.

* VBAT

Este valor representa el voltaje de la batería en mV.

* VOUT

Este valor representa la tensión de salida. Es decir, representa el suministro a Raspberry Pi.

* VBAT\_MIN

Este valor determina el voltaje mínimo de la batería. Si el voltaje de la batería cae por debajo de este valor, la Raspberry será apagada. Este valor es solo un protocolo de seguridad que nunca se debe dar, ya que hay otro valor más alto en el que la Raspberry debe de ser apagada.

Valor por defecto 2,85V.

(Se recuerda que el tipo de batería usada no debe de caer por debajo de 2V).

* VBAT\_SHDN

Este valor determina el valor de tensión de la batería a la que la Raspberry debe de ser apagada.

Valor por defecto 2,95V.

* VBAT\_BOOT

Este valor determina el valor del voltaje de la batería a la que puede ser arrancada la Raspberry Pi.

Valor por defecto 3,15V.

* VOUT\_MAX

Este valor determina el voltaje de salida mínimo para el cual LiFePO4wered/Pi se niega a arrancar la Raspberry cuando está apagada. Este valor es un valor de seguridad para evitar daños cuando las Raspberry se alimentan desde una fuente diferente.

Valor por defecto 3,88V.

* VOFFSET\_ADC

Proporciona un valor de calibración para las mediciones de voltaje ADC.

Valor por defecto 0V.

* AUTO\_BOOT

Este registro por defecto tiene un valor 0 que hace que la Raspberry permanezca apagada mientras no se pulsa el botón táctil de encendido.

Pero es un valor que se puede modificar a 1 para que la Raspberry se encienda sola cuando tenga un valor de tensión suficiente (VBAT\_BOOT).

* WAKE\_TIME

Este valor permite establecer un tiempo en minutos para que la Raspberry permanezca apagada antes de que se encienda. Es una función despertador.

El valor por defecto es cero por lo que la Raspberry permanecerá apagada hasta que se presione el botón.

* SHDN\_DELAY

Establece el número de tics que transcurren desde que el sistema es apagado hasta que se apaga, medido por la línea UART TX).

Valor por defecto 65.

* PI\_BOOT\_TO

Si LiFePO4wered/Pi es arrancado y no detecta respuesta de Raspberry en un tiempo, se apaga. Este valor determina este tiempo de espera para recibir una respuesta.

Valor por defecto 5min.

* PI\_SHDN\_TO

LiFePO4wered/Pi se apagará cuando este sea puesto en estado de apagado y la señal TX del UART pase un nivel bajo. Si esa señal se queda en nivel alto, el sistema permanecerá en estado de apagado si apagarse hasta que se finalice la batería.

Por eso se ha establecido este valor para que aunque esa señal siga en nivel alto, se apague cuando pase un tiempo determinado.

Valor por defecto 2min.

* WATCHDOG\_CFG

Es una función de vigilancia para garantizar al usuario que la aplicación se está ejecutando correctamente, mediante un temporizador que cuenta regresivamente cada 10 segundos y cuando llega a cero se toma una acción. Esa acción es determinada por este registro.

Cuando el valor es cero no se realiza ninguna acción y la función de vigilancia es desactivada.

Valor por defecto 0.

* WATCHDOG\_GRACE

Establece un periodo de gracia desde que el sistema comienza a arrancar hasta que la aplicación arranque el temporizador de vigilancia.

Valor por defecto 20s.

* PI\_RUNNING

Este valor determina el estado de potencia de la Raspberry, cuando esta desactivada o en arranque este valor es 0 y cambia a 1 cuando esta es arrancada. Esto cambiará el estado del LED de pulsante a encendido y permitirá entradas por el usuario.

* CFG\_WRITE

Este registro hace posible guardar de forma permanente el flash los cambios de configuración por el usuario.

Valor por defecto 0.

Una vez estudiados los registros posibles se puede proceder a la creación del programa que devuelve los valores de la batería deseados.

### Shell Scripts

[23] Para realizar la recogida de datos de una forma automática se van a usar Shell scripts.

Esto es una herramienta para la automatización de tareas y ejecutarlas desde la consola.

Está formado por un archivo de texto, el cual contiene unas órdenes que se ejecutan de principio a fin cada vez que se ejecute el archivo.

La forma de crearlo es creando un archivo con la extensión .sh, y luego abrirlo con cualquier editor de texto para editarlo.

#### Shell Script Descarga

Para consturir el programa de recogida de datos de los valores de tensión de la batería cuando esta está alimentando a la Raspberry Pi Zero, lo primero que se hace es crear un Shell script que se ha llamado curva-carga.sh y un archivo .txt llamado log.txt, que servirá como argumento para el script, y donde se guardaran de forma automática y secuencial los datos obtenidos por el programa.

Estor archivos se crean con la orden touch, y antes de ejecutar por primera vez el archivo se debe dar permiso para ello al usuario con la orden chmod.

Una vez creados estos programas se procede a escribir las órdenes del script con un editor de texto como puede ser vi, que es el utilizado en este caso.

**Curva-carga.sh**

#!/bin/sh -- (1)

if [ $# -lt 1 ]; then

echo “indica el archivo de log como argumento”

exit

fi

LOG=”$1”

echo “logging data to $LOG”

:>”$LOG”

while true: do

# (2)

VBAT=$(lifepo4wered-cli get vbat)

echo $VBAT >> “$LOG”

sync

echo “Read $VBAT”

sleep 2m

done

Para comenzar el script hay que indicar al sistema que se va a presentar un conjunto de instrucciones a procesar seguido del programa a usar para interpretar el script (1).

A continuación, se añade un condicional que en caso de que al ejecutar el archivo no se indique el argumento donde se van a guardar los datos, muestre por pantalla un mensaje que pida el argumento y se cierre.

Si $# que es el número de parámetros o argumentos, es menor que uno, muestra el mensaje y cierra el programa. Si el parámetro si ha sido indicado continuara con las siguientes funciones.

El resto del programa lo que realiza es lo siguiente:

Primero declara una variable, cuyo valor es el primer parámetro indicado al ejecutar el archivo. Que en el caso del proyecto va a ser el archivo .txt donde se quieren guardar los datos.

Esa variable luego va a ser llamada con el valor “$1”.

Se define la variable VBAT que en este caso es el valor de la tensión de la batería que nos es proporcionada por el programa lifepo4wered-cli. (2)

Se le dice al programa que ese dato se guarde en el archivo txt (>>) y lo muestre por pantalla.

Con la orden sync se obliga a que esos datos obtenidos se guarden en el disco y no en la memoria, para que en caso de fallo del sistema no se pierdan dichos datos.

Con la orden sleep se le indica que todo esto se repita cada cierto tiempo hasta que se interrumpa la ejecución ya sea de forma intencionada o por agotamiento de la batería.

#### Shell Scripts Alimentación

En el caso de querer obtener los valores de la tensión con la que se alimenta la Raspberry Pi Zero, el programa a realizar es el mismo (curva-descarga.sh) cambiando la variable que pide el dato a obtener por lifepo4wered-cli, que en este caso no será VBAT sino VOUT.

El argumento que se le indique en este caso va a ser log2.txt para que los datos obtenidos sean guardados en un archivo distinto.

**Curva-descarga.sh**

#!/bin/sh

if [ $# -lt 1 ]; then

echo “indica el archivo de log2 como argumento”

exit

fi

LOG=”$1”

echo “logging data to $LOG”

:>”$LOG”

while true: do

VOUT=$(lifepo4wered-cli get vout)

echo $VOUT >> “$LOG”

sync

echo “Read $VOUT”

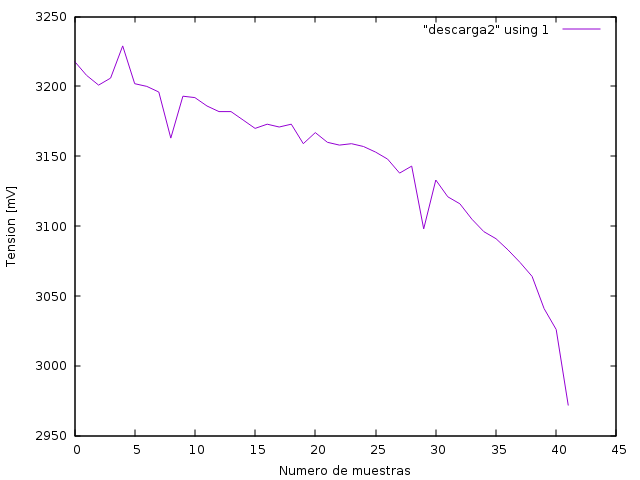
sleep 2m

done

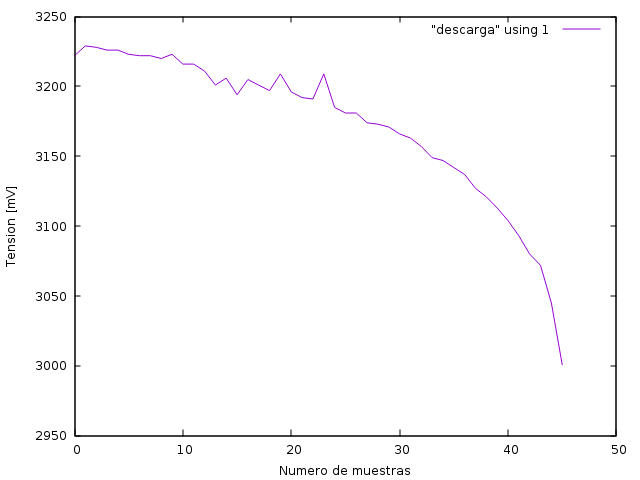
### Resultados

Los resultados obtenidos en los archivos .txt son representados en graficas mediante GNUPLOT, que es un programa para generar graficas a partir de funciones o datos, compatible con Linux.

#### Descarga



1. Datos descarga 1.



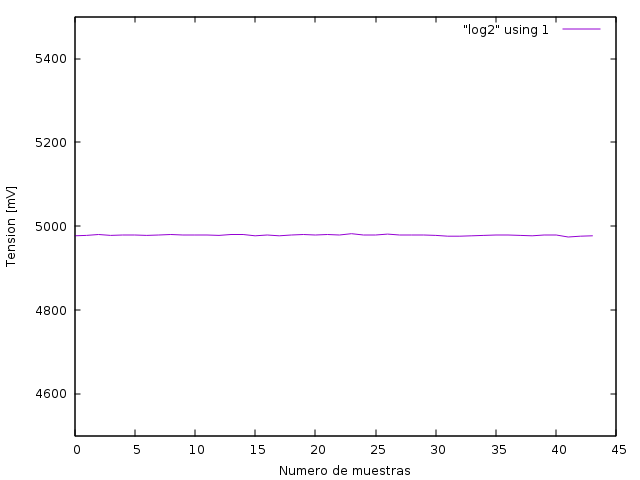
1. Datos descarga 2.

Los datos obtenidos son muestras recogidas cada dos minutos mientras la Raspberry Pi es alimentada por el LiFePO4wered/Pi después de haberse cargado por completo.

Como se puede observar la batería se va descargando durante aproximadamente una hora y media, hasta que la tensión alcanza el valor definido por VBAT\_SHDN para que la Raspberry proceda a apagarse, que como antes se ha indicado, es un valor de 2,95V para que la batería nunca caiga por debajo de valor de tensión de corte de descarga (2V).

Esta curva puede variar dependiendo de los periféricos conectados, aumentando o disminuyendo el tiempo que dure la batería.

#### Alimentación



1. Datos alimentación.

Como se puede observar en la gráfica esta celda de LiFePO4 suministra a la Raspberry Pi una tensión estable de alimentación de casi 5V desde que esta es arrancada, hasta que se apaga.

## RECOGIDA DE DATOS DE CARGA DE LA BATERÍA

En el caso de los datos de la batería en estado de carga, el sistema a usar para la recogida de datos no va a ser el mismo que el capítulo anterior.

Los datos que se quieren obtener, son muestras de la batería desde el instante en que comienza a cargar, hasta que se da por cargada la celda.

Cuando la celda se quiere cargar, es porque anteriormente se ha descargado hasta el valor el cual la Raspberry se ha mandado apagar automáticamente.

Esto quiere decir que con la celda descargada no se podría encender la Raspberry para obtener los datos desde el inicio de la carga, sino que primero habría que poner en estado de carga, (ya que LIFEPO4wered/Pi permite su utilización estando alimentado por su micro USB), para poder encender la Raspberry Pi, y después ejecutar el script de recogida de datos.

En ese proceso se perderían datos, por lo que no se pueden obtener los datos de carga mientras la Raspberry Pi es alimentada por el propio LiFePO4wered/Pi y se ha optado por obtener las muestras de una forma externa con otra Raspberry Pi.

Esta forma de obtener los datos se basa en obtenerlos con un conversor analógico digital que recoja los datos de tensión de la batería conectándolo a sus bornes, ofreciendo los datos desde el inicio del proceso de carga.

El conversor analógico digital (ADC) elegido es el ads1118, cuya data sheet se facilita en el ANEXO IV.

Por lo que se va a conectar la batería en estado de carga a un conversor ads1118, y éste a una Raspberry Pi que leerá los datos obtenidos por el conversor.

Para realizar el software necesario para este proceso, este proyecto se ha basado en el programa desarrollado por Ivan Valbuena Sánchez en el capítulo 4 de su proyecto “Sensores de bajo coste aplicados al campo de la contaminación atmosférica: Desarrollo de un prototipo”, el cual se va a tratar en los siguientes capítulos.

### Lenguaje y construcción del Software

El lenguaje empleado para el software es C++, que es un lenguaje de programación diseñado como una extensión del lenguaje C.

El programa fue desarrollado buscando que fuese un software estable y portable para facilitar su uso en aplicaciones similares, como es el caso de este proyecto.

Para ello se empleó “google Test” que es una biblioteca de pruebas unitarias para programación en C++ que permite testar los programas que componen el código antes de hacer el programa definitivo.

Para la gestión de configuración se empleó “CMake”, que es una herramienta multiplataforma de generación de código. El proceso de construcción se controla creando ficheros CMakeLists.txt en cada directorio que permiten organizar el proyecto.

Para la construcción del ejecutable se emplea “Make” que realiza la compilación del programa y construye el ejecutable. “Make” determina que partes del programa deben de ser compiladas y crea el programa a partir de unas instrucciones escritas en el fichero “Makefile”. Esta función reduce los tiempos de compilación ya que no hay que compilar cada uno de los archivos que componen el programa, sino que el mismo determina que programas han sido modificados y necesitan ser compilados.

El programa principal está compuesto por ficheros fuentes y un fichero principal que es llamado “logger.cpp”, cuando se modifica algo en algún fichero, se compila con “Make”, y este construye un ejecutable que será el programa final.

En este proyecto se modificarán dichos archivos para configurarlos según las necesidades, y se generará un programa final que será el programa a ejecutar para la obtención de los datos de la batería.

### Estructura

Todo el software del proyecto, se ubica en la ruta de enlace siguiente dentro de la controladora: /home/pi/git/AMASIJO/scripts/read-adc.

Dentro de esta carpeta se encuentran diferentes carpetas:

* Bcm2835

Incluye la librería bcm2835, que se utiliza para la comunicación de Raspberry Pi con los periféricos. Es incluido en los archivos “CMakeLists.txt”.

* Build

Es el directorio donde se construye el ejecutable (donde se realiza el “make”) y donde se incluye el archivo .txt que recoge los resultados de ese ejecutable.

Aquí también se encuentra el fichero “Makefile” con las instrucciones de compilación.

* Fmtlib

Incluye los archivos de la biblioteca fmtlilb. Es incluido en los archivos “CMakeLists.txt”.

* Src

Esta carpeta contiene el código fuente y todos los ficheros de cabecera que desarrollan el software (ads1118.hh GPIO.hh RPI.cc RPI.hh SPI.hh). Dentro de esta se encuentra la carpeta logger donde se encuentra en fichero principal logger.cpp.

### Protocolo de comunicación

El protocolo de comunicación utilizado entre la Raspberry Pi y el conversor analógico digital es SPI.

Es un estándar de comunicación para la transferencia de datos como lo es también I2C. A diferencia de I2C el protocolo SPI está compuesto por cuatro líneas, la línea de reloj SCLK, la línea de salida de datos MOSI, la línea de entrada de datos MISO y la línea chip select para seleccionar uno de los dos dispositivos que puede haber por línea. Permite conectar hasta 4 dispositivos y aunque I2C permite conectar un número mayor de dispositivos, la comunicación SPI permite una transferencia de datos más elevada.

### Descripción de ficheros

En el proyecto para el que está dedicado este software, se utilizan, tanto comunicaciones SPI como I2C, así como se realiza la conexión de más de un periférico para la comunicación SPI. Los ficheros a usar van a ser los mismos, pero conciertas modificaciones. También se va a prescindir de los archivos de dicados a los periféricos por comunicación I2C que aquí no se utilizan.

La principal modificación es que solo se va a usar la parte del programa dedicada al conversor analógico digital ads1118 para comunicar mediante SPI un solo periférico, por lo que no se necesita configurar los pines GPIO de forma manual, sino que se usa el modo “Chip select”.

RPI.hh:

#include <bcm2835.h>

#include <system\_error>

namespace RPi {

struct system {

system() {

if (!bcm2835\_init())

throw std::runtime\_error("Failed to init BCM2835");

}

~system() { bcm2835\_close(); }

};

const unsigned SYSTEM\_CLOCK\_FREQ = 400000000;

class base {

static system \_rpi;

};

}

Este archivo se va a usar tal cual se configuró:

La cabecera RPi.hh contiene las características globales del sistema. Define un atributo estático de clase que se utiliza para inicializar y finalizar automáticamente la librería BCM2385, que se utiliza como interfaz de bajo nivel con los periféricos del sistema. Este módulo define también la frecuencia del reloj del sistema.

Finalmente, se declara la clase base instanciar el singleton del sistema de forma automática para cada subsistema. De esta forma se evitan los problemas asociados al singleton y se garantiza que el orden de inicialización es el correcto. La biblioteca se inicializa antes que cualquier subsistema y se finaliza después de cualquier subsistema.

Singleton es un patrón de diseño que permite restringir la creación de objetos perteneciente a una clase, con la intención de garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella.

SPI.hh

Esta cabecera se utiliza para definir el subsistema de comunicación del protocolo SPI.

Como es el protocolo a utilizar para el ads1118 que se usa en este proyecto, este archivo también se va a usar tal cual está configurado.

SPI es un protocolo serie que permite transmitir y recibir a la vez, mediante pines de entrada y salida de datos independientes. El direccionamiento se realiza mediante unas líneas dedicadas de “Chip select” del dispositivo requerido. En la cabecera de pines de la Raspberry Pi se dispone de 2 salidas de “Chip select”.

|  |  |
| --- | --- |
|  | #include "RPi.hh" |
|  | #include <array> |
|  |  |
|  | namespace SPI { |
| 1º | template <class pin\_ctrl> |
|  | class addressing: public pin\_ctrl { |
|  | public: |
|  | addressing() : \_addr(~0) {} |
|  |  |
|  | template <class T> |
|  | addressing(const T& cfg): pin\_ctrl(cfg) {} |
|  |  |
| 2º | void set(uint8\_t addr) { |
| 3º | if(update(addr)) pin\_ctrl::slave\_select(addr); |
|  | } |
|  |  |
|  | protected: |
| 4º | bool update(uint8\_t addr) { |
|  | if (\_addr == addr) return false; |
|  | \_addr = addr; |
|  | return true; |
|  | } |
|  |  |
|  | uint8\_t \_addr; |
|  | }; |
|  |  |
| 5º | class cs\_ctrl { |
|  | public: |
|  | cs\_ctrl() { |
|  | for (uint8\_t cs : \_cspins) |
| 6º | bcm2835\_spi\_setChipSelectPolarity(cs, 0); |
|  | } |
|  |  |
|  | static constexpr size\_t size() {  return sizeof(\_cspins);  } |
|  |  |
|  | protected: |
| 7º | void slave\_select(uint8\_t addr) { |
|  | if (addr >= sizeof \_cspins) return; |
|  | bcm2835\_spi\_chipSelect(\_cspins[addr]); |
|  | } |
|  |  |
|  | private: |
|  | static const uint8\_t \_cspins[3]; |
|  | }; |
|  | template <unsigned \_size> |
| 8º | class gpio\_ctrl { |
|  | public: |
|  | gpio\_ctrl() = delete; |
|  | template <class T> |
|  | gpio\_ctrl(const std::initializer\_list<T>& pins) { |
|  | bcm2835\_spi\_chipSelect(BCM2835\_SPI\_CS\_NONE); |
|  | std::copy(pins.begin(), pins.end(), \_pins.begin()); |
|  | for (auto pin : \_pins) |
| 9º | bcm2835\_gpio\_fsel(pin, BCM2835\_GPIO\_FSEL\_OUTP); |
|  | } |
|  |  |
|  | static constexpr size\_t size() { return \_size; } |
|  |  |
|  | protected: |
| 10º | void slave\_select(uint8\_t addr) { |
|  | if (addr >= \_size) return; |
|  | int target = \_pins[addr]; |
|  | for (int pin : \_pins) |
|  | bcm2835\_gpio\_write(pin, pin == target ? LOW : HIGH); |
|  | } |
|  |  |
|  | std::array<uint8\_t, \_size> \_pins; |
|  | }; |
|  |  |
| 11º | class bus : public RPi::base { |
|  | public: |
|  | bus() { init(); } |
|  |  |
|  | template <class T> |
|  | bus(const std::initializer\_list<T>& cfg) : \_addressing(cfg) { init(); } |
|  |  |
|  | ~bus() { bcm2835\_spi\_end(); } |
|  |  |
| 12º | void set\_speed(unsigned s) { |
|  | if (\_speed == s) return; |
|  | \_speed = s; |
|  | uint16\_t div = static\_cast<uint16\_t>(RPi::SYSTEM\_CLOCK\_FREQ/s); |
|  | // Divider must be even |
|  | bcm2835\_spi\_setClockDivider(div & 1); |
|  | } |
|  |  |
|  | void set\_address(uint8\_t addr) { |
|  | \_addressing.set(addr); |
|  | } |
|  |  |
|  | private: |
| 13º | void init() { |
| 14º | if (!bcm2835\_spi\_begin()) |
|  | throw std::runtime\_error("Failed to init SPI. Are you root?"); |
| 14º | bcm2835\_spi\_setDataMode(BCM2835\_SPI\_MODE1); |
|  | set\_speed(4000000); |
|  | } |
|  |  |
|  | private: |
|  | unsigned \_speed; |
|  | addressing<pin\_ctrl> \_addressing; |
|  | }; |
|  |  |
|  | template <class pin\_ctrl> |
| 15º | class device { |
|  | public: |
|  | explicit device(uint8\_t address) : \_address(address) { |
|  | } |
|  |  |
|  | template <class T> |
| 16º | void read(T& data) const { xfer(data); } |
|  |  |
|  | template <class T> |
| 17º | void write(const T& data) const { T out; xfer(data, out); } |
|  |  |
|  | template <class T> |
| 18º | void xfer(T& data) const { xfer(data, data); } |
|  |  |
|  | template <class T, class U> |
| 18º | void xfer(const T& data\_out, U& data\_in) const { |
|  | \_bus.set\_address(\_address); |
|  | char& in = reinterpret\_cast<char&>(data\_in); |
|  | char& out = const\_cast<char&>(reinterpret\_cast<const char&>(data\_out)); |
|  | bcm2835\_spi\_transfernb(&out, &in, |
|  | std::max(sizeof(T), sizeof(U))); |
|  | } |
|  |  |
|  | }; |
|  |  |

Se configuró de la siguiente manera:

El direccionamiento se abstrae mediante una clase addressing (1º) que funciona globalmente de forma similar tanto si el direccionamiento es a través de pines de GPIO o a través de líneas de chip select dedicadas.

Para ello se utiliza el patrón CRTP (curiously recursive template pattern). El direccionamiento se realiza mediante el método set (2º) que utiliza para la selección del dispositivo destino una clase auxiliar (3º) con la mínima cantidad de operaciones necesarias para seleccionar un esclavo determinado.

Se han implementado dos clases diferentes para la selección de esclavos: cs\_ctrl (5º) para la selección con pines dedicados de chip select, y gpio\_ctrl (8º) para la selección manual de esclavos mediante la manipulación directa de pines de propósito general.

Como se ha comentado antes en este proyecto los pines a usar son los dedicados a chip select por lo que el modo de selección manual no se va a usar.

La selección de esclavos se hace asumiendo que son señales activas a nivel bajo, como es el caso de la mayoría de los dispositivos SPI. En principio no necesitan tener una clase base común, porque el uso como CRTP ya impone las restricciones de interfaz suficientes.

La clase addressing se encarga de prevenir glitches en los pines de direccionamiento cambiando la selección de esclavo solo cuando realmente ha cambiado la dirección del bus SPI (3º y 4º).

Tras esto, se define la clase bus. Es una clase pública derivada de la clase base de RPi.hh (11º). De la misma forma que la clase system, bus se comporta como un singleton que utiliza la técnica RAII (Resource Acquisition Is Initialization) para inicializar y finalizar apropiadamente las comunicaciones SPI. El constructor inicializa las comunicaciones SPI, pero al derivar de base se garantiza que la librería se inicializa antes. En esta clase también se incluye la función ‘set\_speed’ para fijar la velocidad de las comunicaciones SPI (12º).

Esta clase se define fundamentalmente para implementar el bus de comunicaciones como un objeto.

A continuación, se define la clase ‘device’, que se empleará con cada uno de los dispositivos SPI que se van a utilizar (15º). Es decir, cada dispositivo que se utilice se va a declarar como un objeto de clase ‘device’. El bus SPI se declara como un atributo estático de manera que sea compartido por todos los dispositivos SPI. Es el mismo mecanismo que se utiliza para implementar el singleton de system.

Dentro de la clase device, están las funciones propias de cualquier comunicación SPI, que podrá emplear un objeto de la clase ‘device’: leer (‘read’16º), escribir (‘write’17º), y la combinación de lectura y escritura simultánea, propia de las transferencias del bus SPI (‘xfer’18º). Todas estas funciones están agrupadas como funciones miembro parametrizadas para aprovechar la comprobación estática de tipos.

RPi.cc

Es el archivo que define la plataforma del sistema electrónico.

|  |  |
| --- | --- |
|  | #include "SPI.hh" |
|  | #include "I2C.hh" |
| 1º | RPi::system RPi::base::\_rpi; |
| 2º | I2C::bus I2C::device::\_bus; |
|  |  |
|  | template<> |
| 3º | SPI::bus<RPi::spi\_ctrl> SPI::device<RPi::spi\_ctrl>::\_bus = {}; |
|  | const uint8\_t SPI::cs\_ctrl::\_cspins[3] = { |
|  | BCM2835\_SPI\_CS0, BCM2835\_SPI\_CS1, BCM2835\_SPI\_CS2 |
|  | }; |
|  | void I2C::check\_reason\_code(uint8\_t code) |
|  | { |
| 4º | if (code == BCM2835\_I2C\_REASON\_OK) return; |
|  | if (code == BCM2835\_I2C\_REASON\_ERROR\_NACK) |
|  | throw std::runtime\_error("Received a NACK"); |
|  | if (code == BCM2835\_I2C\_REASON\_ERROR\_CLKT) |
|  | throw std::runtime\_error("Received Clock Stretch Timeout"); |
|  | if (code == BCM2835\_I2C\_REASON\_ERROR\_DATA) |
|  | throw std::runtime\_error("Not all data is sent / received"); |
|  |  |
|  | } |

Se configuró de la siguiente manera:

En primer lugar, se instancian los atributos estáticos (singletons) de las clases relacionadas: el sistema (1º), el bus de datos I2C (2º) y el bus de datos SPI (3º). Al construir el bus era preciso indicar el método de selección de esclavos, pero como en este caso no es manual no se indica ninguna dirección.

Finalmente se definen los errores que saltarán en pantalla si falla el protocolo de comunicación I2C (4º). Que en este caso tampoco afecta al programa ya que no se usa la comunicación I2C.

Ads1118.hh

Se configuró de la siguiente manera:

|  |  |
| --- | --- |
|  | #define ADS1118\_HH |
|  | #include "SPI.hh" |
|  | #include <thread> |
| 1º | template <class pin\_ctrl = RPi::spi\_ctrl> |
|  | class ads1118 { |
|  | // CFG register is stored MSB first |
| 2º | struct config\_register { |
|  | unsigned mode: 1; |
|  | unsigned pga: 3; |
|  | unsigned mux: 3; |
|  | unsigned ss: 1; |
|  | unsigned reserved: 1; |
|  | unsigned nop: 2; |
|  | unsigned pull\_up: 1; |
|  | unsigned ts\_mode: 1; |
|  | unsigned rate: 3; |
|  | } \_\_attribute\_\_((packed)); |
|  | public: |
| 3º | ads1118(uint8\_t addr) : \_spi(addr) { |
|  | config\_register cfg = { |
|  | .mode = 1, |
|  | .pga = 1, |
|  | .mux = 0, |
|  | .ss = 0, |
|  | .reserved = 1, |
|  | .nop = 1, |
|  | .pull\_up = 1, |
|  | .ts\_mode = 0, |
|  | .rate = 3, |
|  | }; |
| 4º | \_spi.write(cfg); |
|  | } |
|  | int16\_t in(uint8\_t channel) const { |
| 5º | using namespace std::chrono\_literals; |
|  | config\_register cfg = { |
|  | .mode = 1, |
|  | .pga = 1, |
|  | .mux = channel, |
|  | .ss = 1, |
|  | .reserved = 1, |
|  | .nop = 1, |
|  | .pull\_up = 1, |
|  | .ts\_mode = 0, |
|  | .rate = 3, |
|  | }; |
| 6º | \_spi.write(cfg); |
|  | std::this\_thread::sleep\_for(125ms); |
|  | return \_xfer(cfg); |
|  | } |
| 7º | double in\_V(uint8\_t channel) const { |
|  | return in(channel) \* 4.096 / 32768.; |
|  | } |
|  | }; |
|  | return \_xfer(cfg); |
|  | } |
|  | private: |
| 8º | int16\_t \_xfer(const config\_register& cfg) const { |
|  | unsigned char data[2]; |
|  | \_spi.xfer(cfg, data); |
|  | return (int16\_t) (data[0] << 8 | data[1]); |
|  | } |
|  | SPI::device<pin\_ctrl> \_spi; |
|  | int16\_t \_last[4]; |
|  | }; |
|  |  |
|  | #endif |

En primer lugar, se declara la clase ads1118 como plantilla de la clase ‘template’ de SPI ‘pin\_ctrl’ (1º), dentro de la misma clase se define la estructura del registro de configuración ‘config\_register’ (bits más significativos primero y menos después) para poder enviar la configuración a los A/D (2º).

A continuación, se define el constructor de la clase ADS1118 (3º), lo que hace es escribir una configuración por defecto en el conversor A/D que se inicialice como objeto. Emplea la función ‘SPI\_write’ (4º) al final para escribir la configuración de inicio de las conversiones del A/D correspondiente.

Posteriormente, se define la función “In” (5º) para leer las conversiones de los distintos A/D, esta función recibe un entero que sirve para variar el registro ‘MUX’ y de este modo leer las diferentes entradas (diferenciales ‘Alphasense’ y ‘single-ended’ ‘SPEC’, véase tabla…), se añade al final la función ‘SPI\_write’ (6º) para asegurarse la escritura de la configuración y de este modo obtener siempre la última conversión.

A continuación, se define la función in\_V (7º), que sirve para convertir los valores medidos por la función “In” en su apropiado valor analógico, empleando el fondo de escala y el número de muestras posibles ya que el conversor devuelve muestras de hasta 16 bits, hay 32768 resultados posibles (Valor\*4,096/32768).

Finalmente, se define la función \_xfer (8º), basada en la función xfer definida en SPI.hh, y que sirve para leer y devolver enteros de 16 bits (resultado de las conversiones).

Para poder entender los valores que se han dado es necesario ver cómo funciona el conversor.

El ads1118 integra un amplificador de ganancia programable (PGA), voltaje de referencia, oscilador y un sensor de temperatura de alta precisión.

El PGA ofrece rangos de entrada de 256mV a 6,144V, lo que permite medir tanto señales grandes como pequeñas con alta resolución. Un multiplexor de entrada permite medir dos entradas diferenciales o cuatro entradas de single-ended (entradas únicas referenciadas a masa del conversor).

Para leer los registros del conversor hay que pasarle la configuración de dos palabras de 8 bits desde los más significativos a los bits menos significativos, y devuelve los valores de la misma manera.

La configuración de los bits se detalla en la Tabla 5.

|  |  |  |
| --- | --- | --- |
| Bits | Campo | Descripción |
| 15 | SS | Se usa para comenzar una sola conversión. Cuando la conversión está en curso no tiene efecto.  Valor 0 no tiene efecto y valor 1 comienza una conversión. |
| 14:12 | MUX | Configuración del multiplexor de entrada. Hay dos modos de entradas, diferenciales y single-ended.  Valor del 0 al 3 comprenden las diferentes entradas diferenciales.  Valor del 4 al 7 comprenden las diferentes entradas single-ended referenciadas a masa. |
| 11:9 | PGA | Configura el amplificador de ganancia programable.  000= ±6,144V  001= ±4,096V  010= ±2,048V  011= ±1,024V  100= ±0,512V  101/110/111= ±0,256V |
| 8 | MODE | Valor 0, conversión en modo continuo.  Valor 1, single-shot. |
| 7:5 | DR | Controla la configuración de la velocidad de datos.  000 = 8 SPS  001 = 16 SPS  010 = 32 SPS  011 = 64 SPS  100 = 128 SPS  101 = 250 SPS  110 = 475 SPS  111 = 860 SPS |
| 4 | TS\_MODE | Configura el ADC para convertir la temperatura o las señales de entrada.  Valor 0, ADC.  Valor 1, temperatura. |
| 3 | PULL\_UP\_EN | Habilita una resistencia interna pull up en el pin de salida cuando CS es alto.  Valor 0 desactivada.  Valor 1 activada. |
| 2:1 | NOP | Controlan si los datos se escriben en el registro Config o no.  Para que estos se escriban debe de estar a 1. |
| 0 | Reserved | Siempre se escribe 1. |

1. CONFIGURACION DE BIT DEL ADS1118.

Para la lectura de la tensión de la batería conectada al ADC, se debe modificar la configuración realizada en el asd1118.hh del otro proyecto, ya que las características no son las mismas.

La configuración de los bits en este proyecto para la lectura de la tensión va a ser la siguiente:

ss = 1,

mux = channel,

pga = 1,

mode = 1,

rate= 2,

ts\_mode = 0,

pull\_up = 0,

nop = 1,

Reserved = 1,

Logger.cc

Constituye el fichero principal y en este caso se ha construido un código diferente al del proyecto antes mencionado, para obtener los valores de la señal analógica de nuestra celda cada cinco minutos, y que guarde esos datos en el archivo log.txt.

Lo que hace este programa es indicar al asd1118 que entrada debe activar para leer los datos. (1º)

En este caso se le indica un valor de MUX = 4 que lo que indica es que lea el valor entre la entrada analógica 0 y la masa del conversor, en las cuales se les conectaran el polo positivo y negativo de la batería respectivamente.

A esos datos obtenidos gracias a la configuración realizada en el ads1118.hh, se le indica que sean guardados en el archivo log.txt junto la fecha y la hora de la toma de muestra.

#Include “ads1118.hh”

#Include <fstream>

#Include <iostream>

#Include<ctime>

#Include<locale>

#Include<thread>

#Include<cerrno>

Int main () {

ads1118<> adc (0);

Std::ofstream f(“log.txt”);

for (;;) {

using namespace std::chrono\_literals;

float v = adc. in\_V(4); (1º)

std::time\_t now = std::time(nullptr);

chart ts(83);

std::strftime(ts, sizeof(ts), “%Y-%m-%d %H:%M:%S”, std::localtime(&now));

f << ts << “ ” << v << std::endl;

std::cout << ts << “ “ << v << std::endl;

f.flush();

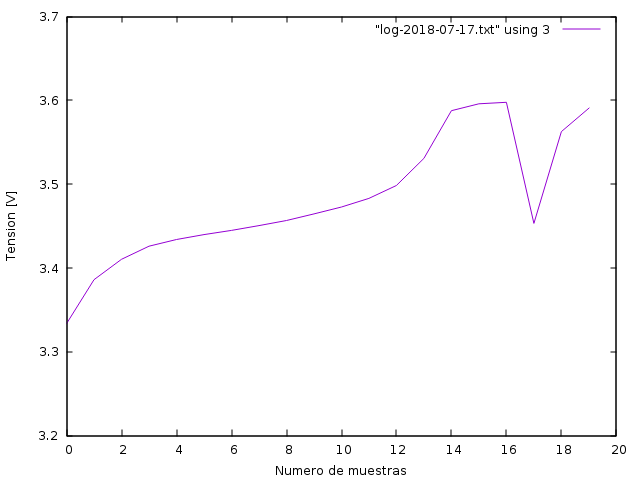
std::this\_thread::sleep\_for(300s);

}

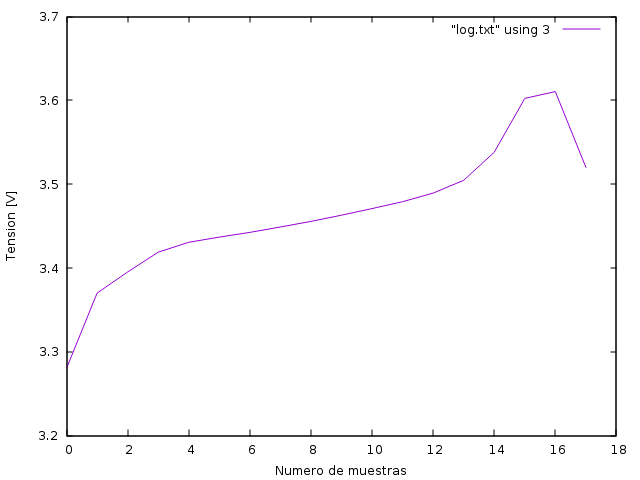
}

### Resultados

De la ejecución de ese programa (logger) se han obtenido los siguientes resultados



1. Datos de carga 1.



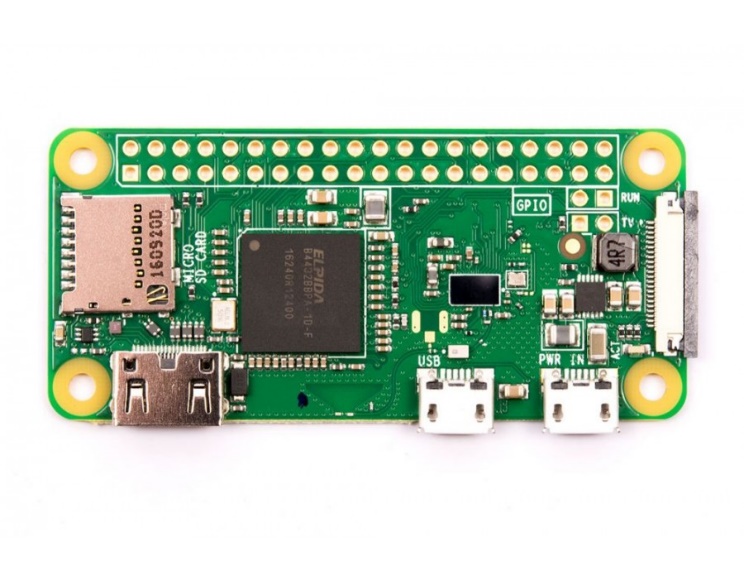
1. Datos de carga 2.

Como se puede comprobar es una carga a tres etapas como se había descrito en el capítulo 7.3 en la que se le aplica una corriente constante desde el comienzo de su carga hasta que alcanza su tensión de pico, una vez alcanzado esa tensión (muestra 14, Fig 21; muestra 16, Fig 22) se mantiene una tensión constante hasta que la corriente disminuye y termina la carga.

Como se puede observar a partir de la muestra que alcanza la tensión de pico, la batería dejó el estado de carga porque ya se había cargado completamente, y durante los siguientes 5 min detectó una bajada de tensión y volvió al estado de carga.

Si se hubiese dejado enchufado y recogiendo datos, oscilaría todo el rato entre esos valores, comenzando y finalizando el estado de carga constantemente.

### CONEXIONES



1. Conexiones conversor ads1118.

Como se ha mencionado en otros apartados, la comunicación a utilizar por el conversor analógico digital es SPI.

En Raspberry hay varios pines destinados a este protocolo de comunicación. Por lo que se va a detallar como se ha realizado la conexión entre ellos.



1. Pines SPI RPi Zero. Recuperado de [https://pinout.xyz/#](https://pinout.xyz/)

|  |  |
| --- | --- |
| Pines ADC | Pines RPi Zero |
| GND | Pin 25 (GND) |
| VIN | Pin 17 (alimentación 3,3V) |
| SCK | Pin 23 (SCLK) |
| CS | Pin 24 (chip select) |
| OUT | Pin 21 (MISO: entrada de datos al maestro y salida de datos del esclavo ) |
| DIN | Pin 19 (MOSI: salida de datos del maestro y entrada de datos al esclavo) |
| GND | Negativo celda LiFePO4 |
| A0 | positivo celda LiFePO4 |

1. CONEXIÓN CONVERSOR ads1118.

# CONCLUSIONES

De los datos obtenidos a lo largo del proyecto se comprueba que el funcionamiento del sistema de carga diseñado para la Raspberry Pi Zero, es el correcto.

Se puede ver que el objetivo del proyecto ha sido conseguido con las siguientes características:

El dispositivo electrónico, o en este caso juguete dedicado a niños, va a estar constituido por una controladora Raspberry Pi modelo Zero.

Como se había mencionado en los objetivos, la principal característica a resolver por este proyecto era un funcionamiento sin cables en el que no hubiese contacto eléctrico ni para la realización de la carga de sus baterías, ni para el funcionamiento del dispositivo.

La forma de alimentar la Raspberry Pi ha sido un módulo diseñado para esa función (LiFePO4wered/Pi), que permite dar una autonomía de hora y media según las muestras recogidas, pero que puede variar en función de los periféricos conectados para la ejecución del juego, pudiendo ser más o menos el tiempo de autonomía.

La alimentación se ha realizado por medio de una celda LiFePO4 que ha sido la elegida por sus características ya definidas en el capítulo 4.

La carga se realiza por medio de un circuito de inducción, en el cual una de las bobinas irá dentro del juguete, y basta con poner dicho modulo del juguete sobre la otra bobina para que la celda sea cargada.

De esta forma queda construido el prototipo de juguete en el que dentro de su módulo llevará incluida la placa Raspberry Pi Zero con un módulo de alimentación acoplado en sus pines, y a su vez la bobina receptora del circuito de inducción, que irá conectada mediante micro USB al módulo de alimentación. Cuando este módulo se acerque a la bobina de inducción se pondrá en marcha la carga, y mientras este permanezca retirado de dicha bobina funcionará con la autonomía de la batería.

Este módulo dispondrá de un botón táctil que activa y desactiva el juguete, siendo este el botón táctil del LiFePO4wered/Pi.

Según las gráficas obtenidas en los resultados, tanto el sistema de carga funciona de una manera correcta proporcionando una buena vida a la celda LiFePO4, como el sistema de alimentación funciona de una forma segura para el funcionamiento de la Raspberry Pi Zero.