
Dispositivo hardware de bajo coste para bancos de pruebas de desarrollo de drivers USB en Linux



TRABAJO FIN DE GRADO

Guillermo Gascón Celdrán y Javier Rodríguez-Avello Tapias

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid
Curso 2022-2023

Dispositivo hardware de bajo coste para bancos de pruebas de desarrollo de drivers USB en Linux

Memoria de Trabajo Fin de Grado

Guillermo Gascón Celdrán y Javier Rodríguez-Avello Tapias

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid
Curso 2022-2023

Resumen

El protocolo de comunicación USB, ampliamente usado en numerosos dispositivos en todo el mundo, resulta complejo a la hora de realizar modificaciones para determinados periféricos, sobre todo a la hora de realizar cambios en la programación del dispositivo en cuestión. En este proyecto se ha trabajado en la realización de un firmware para un chip determinado (ATTiny85) que sea capaz de realizar distintas funciones sobre periféricos, que trabajen con un driver cargado en Linux, todo ello programado en lenguaje C.

La complejidad de la tecnología/especificación USB unido al tiempo limitado que puede dedicarse a describir aspectos de bajo nivel de entrada-salida en las titulaciones de Informática, dificulta la adquisición de conocimientos sobre desarrollo de drivers USB, muy valorados en sectores estratégicos. Asimismo, para iniciarse en el desarrollo de este tipo de drivers es preciso disponer de hardware suficientemente sencillo y de bajo coste. Un ejemplo de dispositivo para iniciación es el dispositivo Blinkstick Strip, que cuenta con 8 LEDs de colores cuyo estado puede alterarse individualmente.

En este proyecto se diseña y programa un dispositivo USB de bajo coste que pueda usarse masivamente como banco de pruebas para el desarrollo de drivers USB. Dicho dispositivo estará formado por varios componentes (LEDs, pantallas LCD, sensores, etc.), y permitirá al desarrollador familiarizarse con distintos aspectos y modos de interacción con el hardware USB, para poder afrontar la complejidad de USB de forma gradual. Además del diseño del hardware, y la construcción de un prototipo del mismo empleando un microcontrolador, como el ATTiny85, será necesario el desarrollo del firmware del dispositivo así como de un conjunto de drivers de ejemplo (preferentemente en Linux), para exponer al usuario los distintos componentes del dispositivo USB.

palabras clave: Firmware, USB, endpoint, kernel Linux, report-ID, AVR, V-USB, Display LCD, Digispark.

Abstract

The USB communication protocol, widely used in numerous devices around the world, is complex when making changes to certain peripherals, especially when making changes to the programming of the device in question. In this project we work on the realization of a firmware for a certain chip (ATTiny85) that is capable of performing different functions on peripherals, that work with a driver loaded in Linux, all programmed in C language.

The complexity of the USB technology/specification together with the limited time that can be devoted to describing low-level input-output aspects in Computer Science degrees, makes it difficult to acquire knowledge on USB driver development, highly valued in strategic sectors. Likewise, to start developing this type of drivers it is necessary to have sufficiently simple and low-cost hardware. An example of a starter device is the Blinkstick Strip device, which has 8 colored LEDs whose state can be individually altered.

In this project, a low-cost USB device is designed and programmed that can be used massively as a test bench for the development of USB drivers. Said device will be made up of several components (LEDs, LCD screens, sensors, etc.), and will allow the developer to become familiar with different aspects and modes of interaction with USB hardware, in order to gradually face the complexity of USB. In addition to the hardware design, and the construction of a prototype of it using a microcontroller, such as the ATTiny85, it will be necessary to develop the firmware of the device as well as a set of example drivers (preferably in Linux), to expose the user to the various components of the USB device.

Autorización de difusión y utilización

Los abajo firmantes, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado “Dispositivo hardware de bajo coste para bancos de pruebas de desarrollo de drivers USB en Linux”, realizado durante el curso académico 2022-2023 bajo la dirección de Juan Carlos Sáez Alcaide en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Guillermo Gascón Celdrán y Javier Rodríguez-Avello Tapias

Juan Carlos Sáez Alcaide

Índice general

Resumen	iii
Abstract	v
Autorización de difusión y utilización	i
1 Introducción	1
1.1 Motivación	1
1.2 Objetivos	1
1.3 Plan de trabajo	2
1.4 Organización de la memoria	2
2 Hardware y firmware utilizado con USB	3
2.1 Microcontroladores AVR	3
2.2 Microcontrolador ATTiny85	3
2.3 Librería V-USB	5
2.3.1 ReportIDs en V-USB	7
2.4 Placa Digispark con interfaz USB	8
2.5 Placa con ATmega y puerto serie	9
3 Periféricos utilizados para el desarrollo del firmware	11
3.1 Anillo de LEDs de colores	11
3.1.1 Smart Shift Register en NeoPixel LED	12
3.1.2 Protocolo One-Wire	13
3.2 Pantalla LCD OLED	14
3.3 Sensor de temperatura	16
4 ATmega328P: el microcontrolador elegido para el proyecto	17
4.1 Características técnicas del ATmega328P	17
4.2 Depuración con ATmega328P	18
5 Drivers desarrollados en el proyecto	21
5.1 BasicInterrupt - Driver asíncrono con comunicación INTERRUPT IN	21

5.1.1	Descripción del funcionamiento	21
5.1.2	Aspectos relevantes del código desarrollado	21
5.1.3	Uso	26
6	Conclusiones y trabajo futuro	29
6.1	Conclusiones	29
6.2	Valoración del TFG	29
6.3	Trabajo futuro	29
	Introduction	31
6.4	Motivation	31
6.5	Objectives	31
6.6	Work plan	32
6.7	Organization of the report	32
	Guía para flashear un firmware propio	33
6.8	Hardware	33
6.9	Software	33
6.10	Diagrama de pines	33
6.11	Cómo flashear un sketch de Arduino	34
6.12	Flashear un firmware propio a partir de un archivo .hex	35
	Monitorizar el tráfico USB usando Wireshark	37
6.13	Software	37
6.14	Abrimos Wireshark	37
6.15	Seleccionamos la interfaz usbmonX correcta	37
6.16	Empezamos a capturar tráfico	38
6.17	Filtrar el tráfico mostrado	38
	Bibliografía	41

Índice de cuadros

Índice de figuras

2.1	Chip ATTiny85	4
2.2	Integración del entorno Arduino con ATTiny85	4
2.3	Detalle de los pines del ATTiny85	5
2.4	Array de ReportIDs utilizados en nuestro proyecto	8
2.5	Placa de desarrollo Digispark	9
3.1	Tira de LEDs circular de NeoPixel sobre una placa Arduino	11
3.2	Disposición de los pines y del controlador en cada LED	12
3.3	Animación sobre la disposición de los bits dentro del controlador de cada LED	13
3.4	Pantalla OLED utilizada en el proyecto	14
4.1	Chip ATmega	17
4.2	Placa con el ATmega328P y un micro USB	18
5.1	Salida dmesg. Muestra la carga del módulo y el dispositivo reconocido .	26
5.2	Salida dmesg. Muestra la carga del módulo y el dispositivo reconocido .	27
6.1	Menú de herramientas de Arduino IDE	34
6.2	Mensaje de salida una vez flasheado el firmware	34
6.3	Captura mostrando el tráfico USB de Wireshark capturado	38
6.4	Ejemplo de un filtro aplicado	39

Capítulo 1

Introducción

En los siguientes apartados se explican los motivos de la realización del proyecto, los objetivos y la planificación de las tareas para llevarlo a cabo.

1.1 Motivación

El protocolo USB es ampliamente usado en todo el mundo para la comunicación con casi cualquier periférico. De forma sencilla, cualquier usuario con un ordenador puede conectar un dispositivo, da igual la funcionalidad que tenga, al puerto USB de su ordenador y éste reconocerlo para que pueda comunicarse con la CPU y realizar la función para la que ha sido desarrollado.

Para que todo esto se pueda dar, el protocolo USB tiene una complejidad enorme en cuanto a paquetes de comunicación entre la CPU y el dispositivo o la controladora USB, a parte de necesitar un driver instalado en el sistema operativo para que pueda interpretarlo. A pesar de lo sencillo que pueda parecer el hardware (2 cables de datos), hay muchos elementos en la comunicación USB que tienen lugar para que el dispositivo pueda ser reconocido.

Para poder estudiar todo el protocolo USB y la interacción entre el dispositivo y la CPU, en este proyecto hemos desarrollado un firmware en C que se comunica con diferentes sensores y dispositivos, utilizando un microcontrolador AVR. Así como un conjunto de drivers de ejemplo para mostrar el funcionamiento y la comunicación entre el ordenador y el dispositivo.

1.2 Objetivos

El objetivo principal de este proyecto es construir un dispositivo USB de bajo coste, que pueda ser utilizado para familiarizarse con el complejo protocolo que maneja este

estandar de conexión, entre otros aspectos podemos destacar los paquetes de información utilizados, *URBs*, y los pasos en la comunicación entre el dispositivo USB y el host. Así como con el desarrollo de drivers para dispositivos USB, utilizando la API síncrona y asíncrona del kernel Linux.

Además, también pretende familiarizarse con los distintos dispositivos y sensores, como pantallas o lectores de temperatura, que se usan conectados al dispositivo USB que en este proyecto se desarrolla.

1.3 Plan de trabajo

Para el desarrollo de este proyecto, se han mantenido distintas reuniones con los directores del proyecto y entre los desarrolladores. En la parte inicial del proyecto, se ha estudiado el funcionamiento de la librería V-USB con distintos proyectos que utilizan otros microchips, para ver qué uso se hacen de los report-IDs y las distintas funciones empleadas en el código. Se ha utilizado el software Wireshark [1] para estudiar los endpoints usados y los paquetes URBs en la comunicación, y así estudiar la configuración del software y la posibilidad de utilizar, por ejemplo, interrupciones.

Para el desarrollo de los periféricos que se van a usar conjuntamente en la placa, Guillermo se ha encargado de estudiar y desarrollar el código correspondiente al sensor de temperatura y Javier al correspondiente con la pantalla LCD 2x16, que posteriormente se ha utilizado una pantalla LCD OLED.

Para la integración del código, se ha utilizado GitHub.

1.4 Organización de la memoria

Para la realización de esta memoria, se ha dividido la misma en varios capítulos. En el primero, se explica a modo de introducción el hardware utilizado y algunos aspectos de bajo nivel sobre la librería utilizada. En el siguiente capítulo, se explican los elementos hardware utilizados, como el anillo circular de leds o la pantalla LCD OLED. Después, se explica con detalle el microcontrolador utilizado finalmente para el prototipo final, ya que éste no tiene las limitaciones que contábamos con el ATTiny85, al poder tener más líneas de comunicación con él.

En la parte de apéndices, contamos con las aportaciones de cada integrante del proyecto, así como distintas instrucciones para poder analizar los paquetes URBs de USB con el software Wireshark, o también para poder *flashear* nuestro propio firmware, con indicaciones sobre los pines concretos a utilizar en la placa.

Capítulo 2

Hardware y firmware utilizado con USB

En este capítulo se detalla el uso de la librería utilizada para el desarrollo del firmware, que está basado en V-USB. También se explican los detalles hardware del microchip utilizado para ejecutar el firmware, y la placa embebida que utiliza la interfaz USB para la comunicación con el driver de nuestro sistema operativo.

2.1 Microcontroladores AVR

La familia de microcontroladores AVR provienen del fabricante Atmel, diseñados para el desarrollo de sistemas empujados y la ejecución de aplicaciones relacionados con robótica y una gran variedad de usos dentro del sector industrial.

Esta familia de microcontroladores AVR están caracterizados por su bajo consumo energético, además de tener una alta velocidad de procesamiento y una interfaz de programación relativamente fácil de usar. Cuentan con una arquitectura RISC, permitiendo un procesamiento más rápido y un uso eficiente de los recursos. [2]

Los microcontroladores AVR suelen tener una memoria flash para el almacenamiento de programas, SRAM para el almacenamiento de datos y una gran variedad de periféricos como temporizadores, UART y ADC para interactuar con dispositivos externos. Muchos de estos microcontroladores son usados en Arduino para el desarrollo de proyectos basados en la interfaz USB. [3]

2.2 Microcontrolador ATTiny85

Este es el microcontrolador usado en el desarrollo del firmware USB para el control de un anillo circular de leds junto a una pantalla LCD OLED y un sensor de temperatura.

Como se ha descrito previamente, este microcontrolador forma parte de la familia de AVR. Es un microchip de 8 bits de bajo consumo y alto rendimiento, cuenta con 8

kilobytes de memoria flash, 512 bytes de EEPROM y 512 bytes de SRAM. Tiene 6 pines de entrada/salida de propósito general (GPIOs), que se pueden usar para distintos propósitos, como entrada o salida digital, PWM o entrada analógica. También tiene un oscilador interno con frecuencias ajustables, eliminando la necesidad de un cristal o resonador externo. [4] [5]

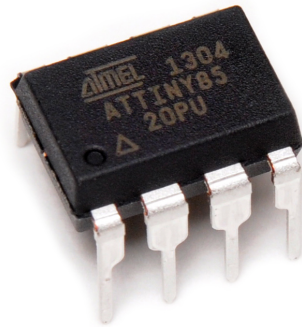


Figura 2.1: Chip ATTiny85

El ATTiny85 es un microcontrolador versátil y se puede usar en una gran variedad de aplicaciones, como sistemas de control, monitoreo de sensores y dispositivos que funcionan con baterías. Uno de los usos más populares de ATTiny85 (y es en el que se ha trabajado en este proyecto) es el desarrollo de proyectos pequeños y de bajo costo.

Para facilitar el desarrollo de estos proyectos, se ha añadido compatibilidad con este microcontrolador al entorno Arduino, lo que facilita en gran manera el programar e interactuar con dispositivos externos.

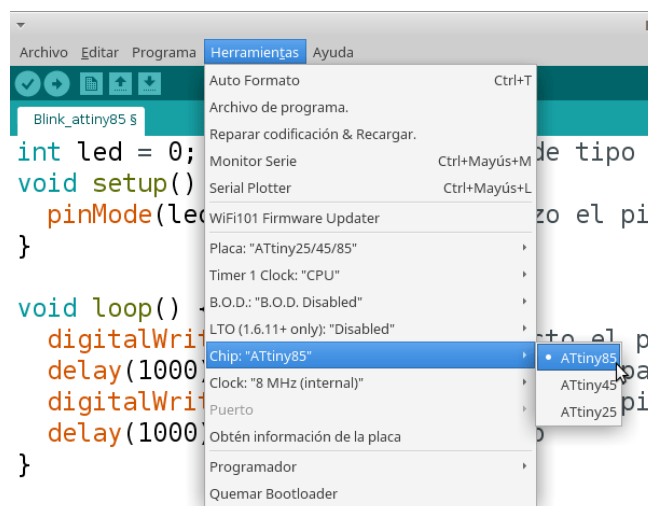


Figura 2.2: Integración del entorno Arduino con ATTiny85

En cuanto a los componentes hardware que integra, contamos con dos temporizadores 8 bits con PWM, un convertidor analógico-digital (ADC) de 8 canales, y una interfaz serie

que se puede configurar para usarse como SPI, I2C o UART. Para la pantalla OLED, se ha utilizado el protocolo I2C con sus correspondientes librerías en C, que posteriormente se detallará.

Un factor a destacar de este microchip es el bajo consumo que requiere para funcionar, por lo que para futuras versiones del proyecto se puede integrar una batería para demostrar que no necesita de una cantidad muy grande de energía y puede aprovechar al máximo la duración de la batería. [6]

A continuación, se detalla el esquema de la configuración de cada uno de los pines para el chip.

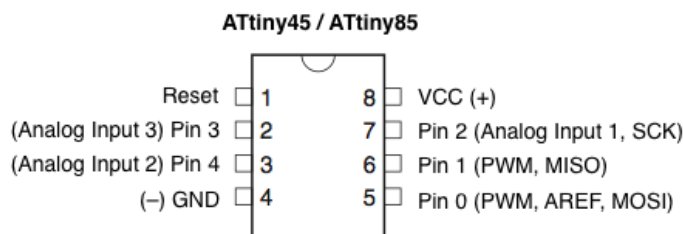


Figura 2.3: Detalle de los pines del ATTiny85

2.3 Librería V-USB

La librería V-USB [7] se usa para la implementación de software para dispositivos USB de baja velocidad que utilizan microcontroladores AVR (como los ATTiny o Atmel). Permite que estos microcontroladores actúen como un dispositivo USB, permitiendo la comunicación con el driver del host u otros dispositivos utilizando este protocolo. Más adelante se detallan algunos aspectos de esta comunicación, como el uso que hace esta librería de los report-id o los endpoints.

La gran ventaja de V-USB con respecto a otras librerías es el pequeño tamaño y bajo consumo que hace de los recursos disponibles. Para su funcionamiento, se requiere unos de pocos kilobytes de código firmware y una pequeña cantidad de memoria RAM para funcionar. Esto es especialmente ventajoso para dispositivos como el ATTiny85. Además, esta librería no necesita de ningún componente hardware adicional, solamente un chip con interfaz USB, reduciendo notablemente su costo y diseño del circuito.

Para la comunicación USB, esta librería hace uso de dos endpoints en el chip ATTiny85: el endpoint 0 y endpoint 1. El endpoint 0 es usado para las transferencias de control, es decir, inicializar y configurar la interfaz USB. El endpoint 1, se usa para la transferencia de datos, es decir, para enviar y recibir los informes USB HID. [8]

Para la configuración de los endpoints, están definidos en el archivo cabecera `usbconfig.h`. En el caso del ATTiny85, dicha configuración se puede encontrar en

el directorio `usbdv`. A continuación, se destacan los puntos más importantes de la configuración software utilizada en la librería.

En el siguiente fragmento de código, se encuentra la configuración definida para el puerto de entrada/salida usado en la configuración USB (para el ATtiny85, se utiliza el puerto B), junto con los bits usados para las señales D+ y D-. También se establece en la última línea la frecuencia a la que trabajará el dispositivo USB.

```
#define USB_CFG_IOPORTNAME      B
#define USB_CFG_DMINUS_BIT      3
#define USB_CFG_DPLUS_BIT       4
#define USB_CFG_CLOCK_KHZ       (F_CPU/1000)
```

A continuación se muestra la configuración para las interrupciones en los endpoints, en este caso al estar la variable a 1 se hace uso de interrupciones. En la macro `USB_CFG_INTR_POLL_INTERVAL` se indica el intervalo en milisegundos para las interrupciones por polling.

```
#define USB_CFG_HAVE_INTRIN_ENDPOINT  1
#define USB_CFG_HAVE_INTRIN_ENDPOINT3 1
#define USB_CFG_EP3_NUMBER             3
#define USB_CFG_SUPPRESS_INTR_CODE     0
#define USB_CFG_INTR_POLL_INTERVAL     100
```

```
#define USB_CFG_IS_SELF_POWERED        0
#define USB_CFG_MAX_BUS_POWER          40
#define USB_CFG_IMPLEMENT_FN_WRITE     1
#define USB_CFG_IMPLEMENT_FN_READ      1
#define USB_CFG_IMPLEMENT_FN_WRITEOUT  0
#define USB_CFG_HAVE_FLOWCONTROL        0
#define USB_CFG_LONG_TRANSFERS         0
#define USB_COUNT_SOF                   1
```

En el siguiente párrafo de código, se muestra la configuración definida para establecer diversos parámetros como el Vendor-ID o el Device-ID, a parte de los distintos nombres para el dispositivo, o el número de serie.

```
#define USB_CFG_CHECK_DATA_TOGGING     0
#define USB_CFG_HAVE_MEASURE_FRAME_LENGTH 1
#define USB_CFG_VENDOR_ID              0xA0, 0x20
#define USB_CFG_DEVICE_ID               0xE5, 0x41
#define USB_CFG_DEVICE_VERSION          0x00, 0x02

#define USB_CFG_VENDOR_NAME             'x', 'G', 'G', 'C'
#define USB_CFG_VENDOR_NAME_LEN         4
```

```

#define USB_CFG_DEVICE_NAME
'p', 'w', 'n', 'e', 'd', 'D', 'e', 'v', 'i', 'c', 'e'
#define USB_CFG_DEVICE_NAME_LEN 11

#define USB_CFG_SERIAL_NUMBER
'N', 'I', 'T', 'R', 'A', 'M', '0', '1', '-', '1', '.', '4'
#define USB_CFG_SERIAL_NUMBER_LEN 12
#define USB_CFG_DEVICE_CLASS 0
#define USB_CFG_DEVICE_SUBCLASS 0
#define USB_CFG_INTERFACE_CLASS 3
#define USB_CFG_INTERFACE_SUBCLASS 0
#define USB_CFG_INTERFACE_PROTOCOL 0
#define USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH 58

```

Las siguientes líneas muestran la configuración relacionada con el manejo de interrupciones para el dispositivo USB. La macro `USB_INTR_CFG` nos indica la máscara de bits para el cambio de puerto de E/S para permitir interrupciones en el pin D+. Así, la macro `USB_INTR_CFG_SET` especifica el registro de desplazamiento para dicho cambio.

```

#ifndef SIG_INTERRUPTO
#define SIG_INTERRUPTO _VECTOR(1)
#endif
#define USB_INTR_CFG PCMSK
#define USB_INTR_CFG_SET (1<<USB_CFG_DPLUS_BIT)
#define USB_INTR_ENABLE_BIT PCIE
#define USB_INTR_PENDING_BIT PCIF
#define USB_INTR_VECTOR SIG_PIN_CHANGE

```

La macro `USB_INTR_ENABLE_BIT` especifica el bit en el registro de control de interrupción de cambio de pin para habilitar las interrupciones de cambio de pin. Por otra parte, `USB_INTR_PENDING_BIT` especifica el bit que indica la presencia de una interrupción pendiente.

2.3.1 ReportIDs en V-USB

Para dotar de funcionalidad al firmware, éste se comunica mediante mensajes que son nombrados mediante un número hexadecimal, al que se les asigna un uso y un tiempo de espera (en bytes) que debe esperar en caso de que llegue un mensaje con dicho ID. En el siguiente ejemplo del array de ReportIDs, se declaran 3 ReportIDs de uso indefinido (ya que se usan para funciones personalizadas). El primero es de 8 bytes, cuando llega un mensaje con el ID=1 es un mensaje de cambio de color; el segundo y tercer ID son mensajes que contienen 32 bytes y es texto que puede almacenar la EEPROM, se usan para el número de serie o el nombre del dispositivo. [9]

```

const PROGMEM char usbHidReportDescriptor[USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH] = {
  /* USB report descriptor */

  0x06, 0x00, 0xff,          // USAGE_PAGE (Generic Desktop)
  0x09, 0x01,                // USAGE (Vendor Usage 1)
  0xa1, 0x01,                // COLLECTION (Application)
  0x15, 0x00,                // LOGICAL_MINIMUM (0)
  0x26, 0xff, 0x00,          // LOGICAL_MAXIMUM (255)
  0x75, 0x08,                // REPORT_SIZE (8)
  0x85, 0x01,                // REPORT_ID (1)
  0x95, 0x03,                // REPORT_COUNT (3)
  0x09, 0x00,                // USAGE (Undefined)
  0xb2, 0x02, 0x01,          // FEATURE (Data,Var,Abs,Buf)
  0x85, 0x02,                // REPORT_ID (2)
  0x95, 0x20,                // REPORT_COUNT (32)
  0x09, 0x00,                // USAGE (Undefined)
  0xb2, 0x02, 0x01,          // FEATURE (Data,Var,Abs,Buf)
  0x85, 0x03,                // REPORT_ID (3)
  0x95, 0x20,                // REPORT_COUNT (32)
  0x09, 0x00,                // USAGE (Undefined)
  0xb2, 0x02, 0x01,          // FEATURE (Data,Var,Abs,Buf)
  0xc0                       // END_COLLECTION
};

```

Figura 2.4: Array de ReportIDs utilizados en nuestro proyecto

2.4 Placa Digispark con interfaz USB

La placa Digispark cuenta con el chip ATTiny85, es la placa elegida para desarrollar el firmware de este proyecto.

Los detalles técnicos con los que cuenta esta placa son los siguientes:

- Microcontroller: ATTiny85
- Operating Voltage: 5V
- Input Voltage (recommended): 7-12V
- Input Voltage (limits): 5-16V
- Digital I/O Pins: 6
- PWM Channels: 3
- Analog Input Channels: 4
- Flash Memory: 8 KB (of which 2.5 KB is used by the bootloader)
- SRAM: 512 bytes
- EEPROM: 512 bytes

Es una placa realmente compacta, con un tamaño de 25 mm x 18 mm. Incluye una interfaz USB para programación y comunicación con el host, así como un regulador de voltaje y LED de encendido. [10]

Una característica única de la placa Digispark es que viene preprogramada con un cargador de arranque que permite programarla a través de USB usando el IDE de Arduino (llamado Micronucleous Bootloader). Con ello se puede programar código desde Arduino fácilmente a la placa sin necesidad de un programador independiente. Aunque para el desarrollo de nuestro proyecto, se ha optado por utilizar la librería V-USB sin el entorno de Arduino, por lo que se requiere de un compilador de AVR y un programador externo. [11]



Figura 2.5: Placa de desarrollo Digispark

2.5 Placa con ATmega y puerto serie

Completar!

Capítulo 3

Periféricos utilizados para el desarrollo del firmware

En este capítulo se describe con detalles de bajo nivel los periféricos utilizados en el firmware desarrollado.

3.1 Anillo de LEDs de colores

Para probar el firmware, en la primera versión se ha hecho uso de la tira de leds circular fabricado por la empresa NeoPixel [12]. Dichos leds se encuentran en disposición circular con colores RGB direccionables individualmente.

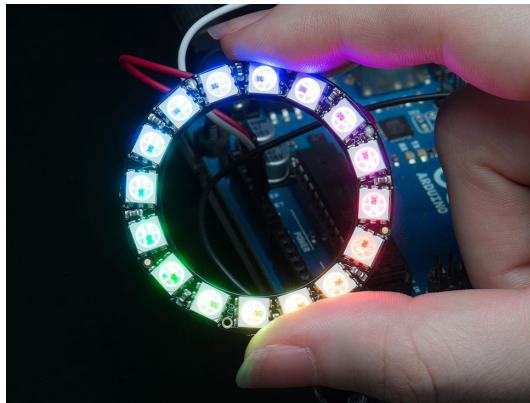


Figura 3.1: Tira de LEDs circular de NeoPixel sobre una placa Arduino

Estos LEDs están conectados en cadena y cada LED está controlado por una sola línea de datos. La línea de datos usa un protocolo de comunicación específico llamado *One-Wire Protocol*, que se implementa a través de software y se usa para controlar el color y el brillo de cada LED individual.

La tira de LED circular se alimenta con una fuente de alimentación de 5 V que proporciona la placa Digispark. Cada LED consume una cierta cantidad de energía, y el consumo total de energía de la tira depende de la cantidad total de LEDs y la configuración de brillo.

La tira de LEDs circular generalmente está controlada por un microcontrolador, en nuestro caso el ATTiny85 se encarga de ello, aunque también existen interfaces de comunicación para estas tiras de LEDs especializado, que envía los datos de color y brillo a los LED mediante el protocolo *One-Wire*. El microcontrolador se puede programar para crear una amplia variedad de efectos de iluminación, incluidos ciclos de color, animaciones y patrones.

3.1.1 Smart Shift Register en NeoPixel LED

Cada LED individual en la tira de LEDs circular de NeoPixel contiene un registro de cambio inteligente, con un pequeño microcontrolador y un LED RGB (Red, Green, Blue). El microcontrolador es responsable de controlar el color y el brillo de los LED y comunicarse con los otros LED de la cadena.

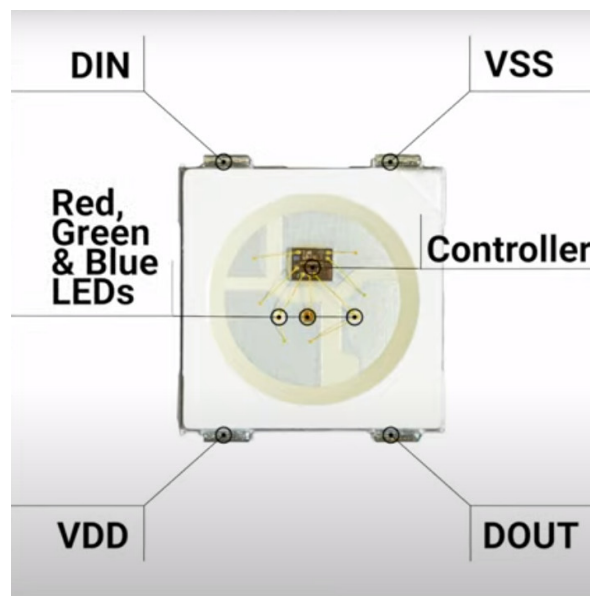


Figura 3.2: Disposición de los pines y del controlador en cada LED

El Smart Shift Register LED utiliza un protocolo de comunicación específico llamado *One-Wire Protocol* para recibir datos del microcontrolador. Dicho protocolo se basa en otro protocolo serie de bajo nivel que utiliza una sola línea de datos para transmitirlos entre el microcontrolador y los LED.

Para enviar datos al LED Smart Shift Register, el microcontrolador envía una serie de pulsos en la línea de datos, que el LED interpreta como una secuencia de comandos.

Los comandos incluyen instrucciones para configurar el color y el brillo de cada LED individual, así como para configurar el tiempo y la sincronización de la cadena de LEDs.

El LED de registro de desplazamiento inteligente también incluye un conjunto de registros de desplazamiento, que se utilizan para almacenar los datos de color y brillo de cada LED de la cadena. Los registros de desplazamiento permiten que el LED de registro de desplazamiento inteligente reciba y almacene datos del microcontrolador, y luego pase esos datos al siguiente LED de la cadena.

Además de los registros de desplazamiento, el Smart Shift Register LED también incluye un conjunto de controladores de corriente constante, que se utilizan para controlar el brillo de cada LED individual. Los controladores de corriente aseguran que cada LED reciba una cantidad constante de energía, independientemente del número de LEDs en la cadena o la configuración de brillo. [13]

3.1.2 Protocolo One-Wire

El protocolo *One-Wire* es un protocolo de comunicación serie de bajo nivel que se utiliza para comunicarse con dispositivos que tienen una sola línea de datos. Fue desarrollado por *Dallas Semiconductor* en la década de los 90 y ahora se usa en una amplia gama de aplicaciones, incluidos sensores de temperatura, tiras de LED y EEPROM.

Dicho protocolo está diseñado para ser lo más simple, confiable y con bajo costo. Utiliza una sola línea de datos, que es bidireccional y normalmente está conectada al microcontrolador. La línea de datos también está conectada a uno o más dispositivos *One-Wire*, que suelen ser sensores u otros dispositivos de bajo consumo.

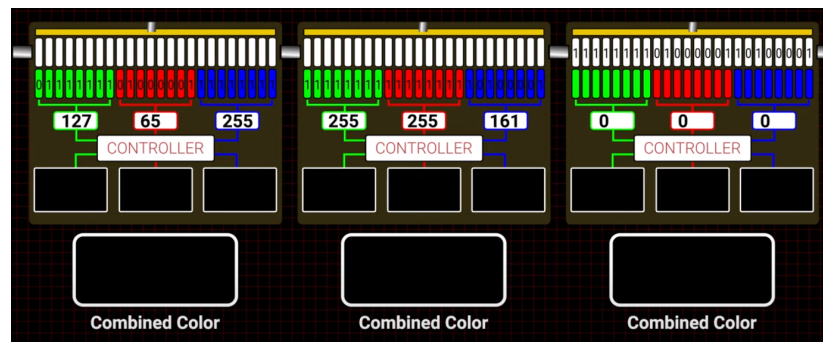


Figura 3.3: Animación sobre la disposición de los bits dentro del controlador de cada LED

Para comunicarse con un dispositivo *One-Wire*, el microcontrolador envía una serie de pulsos en la línea de datos, que el dispositivo interpreta como comandos. Los pulsos se generan utilizando una secuencia de tiempo específica, que incluye una combinación de niveles de voltaje alto y bajo en la línea de datos.

El protocolo *One Wire* utiliza una variedad de comandos diferentes para comunicarse con el dispositivo, incluidos comandos para leer y escribir datos, establecer opciones de configuración y enviar señales de control. Estos comandos se envían en una secuencia específica y se cronometran de acuerdo con las especificaciones del protocolo.

Una de las características más importantes es que incluye un mecanismo CRC (comprobación de redundancia cíclica) incorporado, que se utiliza para garantizar la integridad de los datos que se transmiten. El mecanismo CRC utiliza un algoritmo matemático para generar una suma de verificación para los datos, que luego se transmite junto a ellos. El dispositivo receptor puede usar el algoritmo para generar la suma de verificación para los datos recibidos, y compararla con la suma de verificación transmitida para verificar que los datos se hayan transmitido correctamente.

3.2 Pantalla LCD OLED

Otro de los periféricos usado para el desarrollo del firmware, es una pantalla LCD tipo OLED, en el que se muestran cadenas de texto.

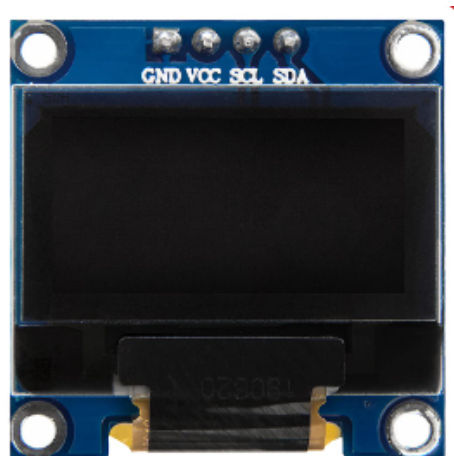


Figura 3.4: Pantalla OLED utilizada en el proyecto

Para su implementación en el firmware del dispositivo, se hace uso de la biblioteca `oled.h`. El cambio fundamental es en la función `uchar usbFunctionWrite(uchar *data, uchar len)` en el que se ha añadido un nuevo report ID (el reportID 4) que hace uso de esta librería para mostrar una cadena de texto guardada en el array `uchar *data` que le llega por parámetro a la función.

```
} else if (reportId == 4) {  
    display7sSet(data[1]);  
  
    return 1;  
}
```

En la función del firmware `uchar usbFunctionRead(uchar *data, uchar len)` no se ha implementado ninguna condición ya que no se lee ningún dato a través de la pantalla.

La función que se encarga de inicializar el dispositivo `usbMsgLen_t usbFunctionSetup(uchar data[8])` hace lo siguiente:

```
} else if (reportId == 4) {  
  
    bytesRemaining = 1;  
    currentAddress = 0 ;  
  
    return USB_NO_MSG;  
  
}
```

En el que retornando la macro `USB_NO_MSG` indicamos al firmware que debe ejecutarse la función de escritura mencionada previamente con el *reportID* correspondiente (La función de Setup es la primera que se ejecutará nada más encender el dispositivo).

Para hacer uso de la pantalla, se hace uso de la librería `libusb`, por lo que en el main declaramos algo como lo siguiente (se obvia la comprobación de errores para que el código no sea extenso):

```
#define MSG "Hello, World! :) Sent from the computer"  
  
int main() {  
    libusb_device_handle *handle;  
    libusb_init(NULL);  
  
    // Try to find and open device with PID:VID  
    handle = libusb_open_device_with_vid_pid(NULL, 0x20a0, 0x41e5);  
    // Check errors  
  
    result = libusb_detach_kernel_driver(handle, 0);  
    // Check errors  
  
    result = libusb_claim_interface(handle, 0);  
    // Check errors  
  
    setText(handle, 0x4);  
  
    libusb_release_interface(handle, 0);  
    libusb_close(handle);  
    libusb_exit(NULL);  
}
```

```
    return 0;
}
```

La función `setText()` ejecuta el siguiente código:

```
void setText(libusb_device_handle *handle, unsigned int reportID) {
    int res = -1;
    unsigned char data[41];

    data[0] = reportID; //0x4

    for (int i = 1; i < 40; i++)
        data[i] = MSG[i - 1];

    res = libusb_control_transfer(handle, // device
                                  LIBUSB_REQUEST_TYPE_CLASS |
                                  LIBUSB_RECIPIENT_INTERFACE |
                                  LIBUSB_ENDPOINT_OUT, // bmRequestType
                                  0x9, // bRequest 0x1 -> GET 0x9 -> SET
                                  0x0003, // wValue
                                  0, // wIndex
                                  data, // data
                                  41, // wLength
                                  5000); // timeout

    if (res < 0)
        printf(">>> [ERROR]: Cannot send URB! Code: %d\n", res);
}
```

En esta función, llama a la función `libusb_control_transfer` utilizando una petición SET al *report ID* que le llega por parámetro y que en el main habíamos establecido en 0x4. Cabe destacar que el array `data[]` contiene el mensaje y en la posición 0 el *report ID*, por lo que en el primer bucle for se adapta el array para que en la primera posición esté el 0x4.

3.3 Sensor de temperatura

Completar

Capítulo 4

ATmega328P: el microcontrolador elegido para el proyecto

En este capítulo se describen los aspectos técnicos de este microcontrolador, que es el que se ha optado finalmente por utilizar en el proyecto. Aunque en el capítulo 2 se ha hablado del ATTiny85, éste microcontrolador presentaba dificultades a la hora de utilizar determinados pines con los periféricos, debido a la escasez de éstos.

4.1 Características técnicas del ATmega328P

El Atmega328P de 8 bits es un circuito integrado de alto rendimiento que está basado un microcontrolador RISC, combinando 32 KB de memoria flash con capacidad *read-write* al mismo tiempo.



Figura 4.1: Chip ATmega

Tiene 1 KB de memoria EEPROM, 2 KB de SRAM, 23 líneas de E/S de propósito general, 32 registros de proceso general, tres temporizadores flexibles/contadores con modo de

comparación, interrupciones internas y externas, programador de modo USART, una interfaz serial orientada a byte de 2 cables, SPI puerto serial, 6-canales 10-bit Conversor A/D (canales en TQFP y QFN/MLF packages), temporizador “watchdog” programable con oscilador interno, y cinco modos de ahorro de energía seleccionables por software. El dispositivo opera entre 1.8 y 5.5 voltios. Puede alcanzar una respuesta de 1 MIPS, balanceando consumo de energía y velocidad de proceso.



Figura 4.2: Placa con el ATmega328P y un micro USB

4.2 Depuración con ATmega328P

Para poder realizar una depuración del firmware que se flashee en la placa, se ha implementado una función a modo de *printf()* para poder imprimir por una consola UART mensajes de depuración.

Una de las principales limitaciones del ATtiny85 era que únicamente tenía un puerto, el PORTB, por lo que dificultaba mucho programar otros pines para otros usos como en este caso, para una salida por puerto serie.

En el firmware se ha añadido una nueva biblioteca, *oddebug.h*, que es la que tiene implementadas las siguientes funciones para imprimir por la consola de la UART los mensajes que deseemos:

```
int odPrintf(char *fmt, ...) {
    va_list ap;
    char str[256];
    int n;

    va_start(ap, fmt);
    vsnprintf(str, 256, fmt, ap);
    uartPutStr(str); // print char one by one
    va_end(ap);
}
```

```
    return 0;
}
```

Esta función recibe un array de char, lo almacena y elabora el string correspondiente (con el final de línea \0) para que se vaya imprimiendo caracter a caracter con la siguiente función:

```
static void uartPutStr(char *c)
{
    while (*c != '\0')
        uartPutc(*c++);
}
```

Se recorre en un bucle todo el string hasta que se llega al final. Para llamar a la función desde nuestro `main()`, se hace de forma similar a la función `printf()`:

```
odPrintf("v-usb device main\n");
```

Capítulo 5

Drivers desarrollados en el proyecto

En este capítulo se describe los drivers propuestos para la interacción con los periféricos conectados a la placa, además de los detalles de bajo nivel como los endpoints utilizados en cada uno.

5.1 BasicInterrupt - Driver asíncrono con comunicación INTERRUPT IN

5.1.1 Descripción del funcionamiento

Este módulo del kernel implementa un driver USB utilizando el endpoint de tipo IN del dispositivo USB, así como la API asíncrona del kernel. [14]

El funcionamiento del módulo es muy sencillo, una vez cargado en el kernel, expone un dispositivo de caracteres que tiene implementada la operación de lectura. Por ello, si lanzamos una llamada `read()`, por ejemplo con `cat`, iniciaremos la rutina que envía el URB de tipo *INTERRUPT IN* al dispositivo pidiéndole que le rellene el buffer enviado con datos.

Como se usa la API asíncrona de USB la llamada a `read()` devuelve siempre 0 bytes, ya que no se sabe con exactitud cuándo se va a devolver el URB relleno con datos y al ser una función no bloqueante no esperamos a que llegue. El resultado de esta solicitud de datos al dispositivo se mostrará a través de la función de callback que se invoca una vez el URB vuelve desde el dispositivo.

5.1.2 Aspectos relevantes del código desarrollado

Función callback del Driver, se ejecuta cuando se devuelven datos en un URB de tipo INT para procesarlo (después de haber reservado memoria para el URB, y que éste haya sido enviado con `usb_submit_urb`:

```

/*
 * Callback function. Executed when INT IN URB returns back with data.
 */
static void pwnedDevice_int_in_callback(struct urb *urb) {
    struct usb_pwnedDevice *dev = urb->context;

    if (urb->status) {
        if (urb->status == -ENOENT ||
            urb->status == -ECONNRESET ||
            urb->status == -ESHUTDOWN) {
            printk(KERN_INFO "Error on callback!");
            return;
        }
    }

    if (urb->actual_length > 0)
        printk(KERN_INFO "Transferred data: %s\n", dev->int_in_buffer);
}

```

Función open (ejecutada nada más cargar el módulo en el kernel), se inicializa un URB de tipo INT en usb_fill_int_urb inicial y se envía con la función usb_submit_urb:

```

/* Called when a user program invokes the open() system call on the device */
static int pwnedDevice_open(struct inode *inode, struct file *file)
{
    struct usb_pwnedDevice *dev;
    struct usb_interface *interface;
    int subminor;
    int retval = 0;

    subminor = iminor(inode);

    /* Obtain reference to USB interface from minor number */
    interface = usb_find_interface(&pwnedDevice_driver, subminor);
    if (!interface) {
        pr_err("%s - error, can't find device for minor %d\n",
            __func__, subminor);
        return -ENODEV;
    }

    /* Obtain driver data associated with the USB interface */
    dev = usb_get_intfdata(interface);
    if (!dev)

```

```

        return -ENODEV;

    /* Initialize URB */
    usb_fill_int_urb(dev->int_in_urb, dev->udev,
                    usb_rcvintpipe(dev->udev,
                                    dev->int_in_endpoint->bEndpointAddress),
                    dev->int_in_buffer,
                    le16_to_cpu(0x0008),
                    pwnedDevice_int_in_callback,
                    dev,
                    dev->int_in_endpoint->bInterval);
    retval = usb_submit_urb(dev->int_in_urb, GFP_KERNEL);

    /* increment our usage count for the device */
    kref_get(&dev->kref);

    /* save our object in the file's private structure */
    file->private_data = dev;

    return retval;
}

```

Función read() ejecutada cuando se hace un cat en el fichero de /dev (se puede observar que se envía un URB tipo INT con datos al dispositivo):

```

#define MAX_LEN_MSG 35
static ssize_t pwnedDevice_read(struct file *file, char *user_buffer,
                               size_t count, loff_t *ppos)
{
    struct usb_pwnedDevice *dev;
    int retval = 0;

    dev = file->private_data;

    if (*ppos > 0)
        return 0;

    /* Send URB. */
    usb_fill_int_urb(dev->int_in_urb, dev->udev,
                    usb_rcvintpipe(dev->udev,
                                    dev->int_in_endpoint->bEndpointAddress),
                    dev->int_in_buffer,
                    le16_to_cpu(0x0008),

```

```

        pwnedDevice_int_in_callback,
        dev,
        dev->int_in_endpoint->bInterval);
retval = usb_submit_urb(dev->int_in_urb, GFP_KERNEL);

if (retval != 0)
    return retval;

return 0; // No bytes returned. Async USB API used.
}

```

Función probe() que se ejecuta cuando se conecta al puerto USB un dispositivo de tipo pwnedDevicestick (se reserva memoria inicial para el URB, que posteriormente será rellenado y enviado):

El punto donde se reserva memoria para el URB sin datos es el siguiente:

```
dev->int_in_urb = usb_alloc_urb(0, GFP_KERNEL);
```

Ésta es la función del Driver completa:

```

/*
 * Invoked when the USB core detects a new
 * pwnedDevicestick device connected to the system.
 */
static int pwnedDevice_probe(struct usb_interface *interface,
                             const struct usb_device_id *id)
{
    struct usb_pwnedDevice *dev;
    struct usb_host_interface *iface_desc;
    struct usb_endpoint_descriptor *endpoint;
    int retval = -ENOMEM;
    int i;

    /*
     * Allocate memory for a usb_pwnedDevice structure.
     * This structure represents the device state.
     * The driver assigns a separate structure to each pwnedDevicestick device
     */
    dev = kmalloc(sizeof(struct usb_pwnedDevice), GFP_KERNEL);

    if (!dev) {
        dev_err(&interface->dev, "Out of memory\n");
    }
}

```

```

        goto error;
    }

    /* Initialize the various fields in the usb_pwnedDevice structure */
    kref_init(&dev->kref);
    dev->udev = usb_get_dev(interface_to_usbdev(interface));
    dev->interface = interface;

    iface_desc = interface->cur_altsetting;

    for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
        endpoint = &iface_desc->endpoint[i].desc;

        if (((endpoint->bEndpointAddress & USB_ENDPOINT_DIR_MASK)
            == USB_DIR_IN)
            && ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
            == USB_ENDPOINT_XFER_INT))
            dev->int_in_endpoint = endpoint;
    }

    if (! dev->int_in_endpoint) {
        pr_err("could not find interrupt in endpoint");
        goto error;
    }

    /* Request IN URB */
    dev->int_in_urb = usb_alloc_urb(0, GFP_KERNEL);
    if (!dev->int_in_urb) {
        printk(KERN_INFO "Error allocating URB");
        retval = -ENOMEM;
        goto error;
    }

    /* save our data pointer in this interface device */
    usb_set_intfdata(interface, dev);

    /* we can register the device now, as it is ready */
    retval = usb_register_dev(interface, &pwnedDevice_class);
    if (retval) {
        /* something prevented us from registering this driver */
        dev_err(&interface->dev,

```

```

        "Not able to get a minor for this device.\n");
    usb_set_intfdata(interface, NULL);
    goto error;
}

/* let the user know what node this device is now attached to */
dev_info(&interface->dev,
        "PwnedDevice now available via pwnedDevice-%d",
        interface->minor);
return 0;

error:
    if (dev->int_in_urb)
        usb_free_urb(dev->int_in_urb);

    if (dev)
        /* this frees up allocated memory */
        kref_put(&dev->kref, pwnedDevice_delete);

    return retval;
}

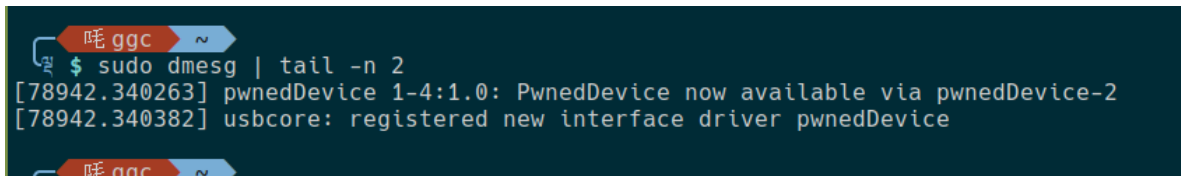
```

5.1.3 Uso

Para usar este driver tan solo hay que compilarlo y cargarlo en el kernel.

```
make
sudo insmod BasicInterrupt.ko
```

Una vez cargado en el kernel, podemos ver en kern.log que ya está disponible y si tenemos conectado el dispositivo USB también será reconocido.



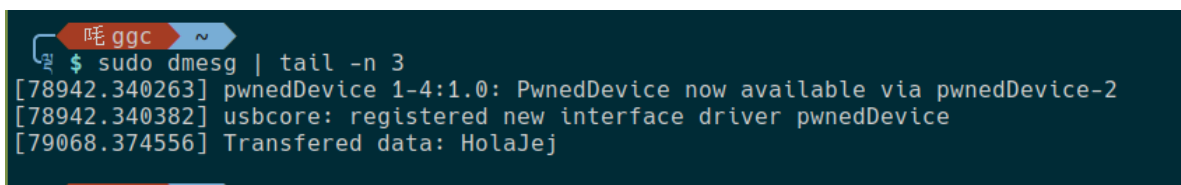
```

$ sudo dmesg | tail -n 2
[78942.340263] pwnedDevice 1-4:1.0: PwnedDevice now available via pwnedDevice-2
[78942.340382] usbcore: registered new interface driver pwnedDevice

```

Figura 5.1: Salida dmesg. Muestra la carga del módulo y el dispositivo reconocido

Y si ejecutamos una llamada a `read()`, por ejemplo usando el comando `cat`, veremos la información que devuelve el dispositivo.

A terminal window with a dark blue background. The prompt is 'ggc' with a red arrow icon. The command 'sudo dmesg | tail -n 3' is entered. The output shows three lines of kernel messages: '[78942.340263] pwnedDevice 1-4:1.0: PwnedDevice now available via pwnedDevice-2', '[78942.340382] usbcore: registered new interface driver pwnedDevice', and '[79068.374556] Transferred data: HolaJej'.

```
ggc $ sudo dmesg | tail -n 3
[78942.340263] pwnedDevice 1-4:1.0: PwnedDevice now available via pwnedDevice-2
[78942.340382] usbcore: registered new interface driver pwnedDevice
[79068.374556] Transferred data: HolaJej
```

Figura 5.2: Salida dmesg. Muestra la carga del módulo y el dispositivo reconocido

Capítulo 6

Conclusiones y trabajo futuro

6.1 Conclusiones

Después de haber realizado este proyecto, estamos más familiarizados sobre el funcionamiento a bajo nivel del protocolo USB, y la gran multitud de firmwares y funcionalidades que se pueden especificar siguiendo este protocolo. Además, hemos estudiado un ejemplo concreto de firmware como es el V-USB [7], con la correspondiente complejidad que lleva el desarrollo de un firmware de estas características, haciendo uso de los distintos endpoints o los report-IDs para las distintas funcionalidades implementadas.

Con el código desarrollado para el firmware de este proyecto, se espera que el desarrollo de dispositivos para realizar tareas concretas con periféricos sea mucho más fácil, y que la gente pueda contribuir con sus propios desarrollos a través de la plataforma GitHub, y que se puedan desarrollar drivers para hacer funcionar estos dispositivos.

6.2 Valoración del TFG

Para poder llevar a cabo el desarrollo de este proyecto, hemos tenido que documentarnos sobre distintos dispositivos hardware e informarnos sobre ellos mediante sus correspondientes *datasheets*, lo cual nos ha hecho coger bastante experiencia para poder pensar en un prototipo que incluyera todos los dispositivos que queremos añadir, además de poder modificar un firmware concreto que va a contener el microchip del dispositivo en cuestión.

6.3 Trabajo futuro

A continuación se listan una posible serie de ampliaciones del proyecto para su desarrollo en un futuro:

-
- Uso de más periféricos como otros sensores o actuadores.
 - Ampliación del firmware V-USB con más endpoints.

Introduction

The following sections explain the reasons for carrying out the project, the objectives and the planning of the tasks to carry it out.

6.4 Motivation

The USB protocol is widely used throughout the world for communication with almost any peripheral. In a simple way, any user with a computer can connect a device, regardless of its functionality, to the USB port of his computer and it will recognize it so that it can communicate with the CPU and perform the function for which it has been developed.

For all this to happen, the USB protocol has an enormous complexity in terms of communication packets between the CPU and the device or the USB controller, in addition to needing a driver installed in the operating system so that it can interpret it. As simple as the hardware may seem (2 data cables), there are many elements in the USB communication that take place so that the device can be recognized.

In order to study the whole USB protocol and the interaction between the device and the CPU, in this project we have developed a firmware in C that communicates with different sensors and devices, using an AVR microcontroller. As well as a set of example drivers to show the operation and communication between the computer and the device.

6.5 Objectives

The main objective of this project is to build a low-cost USB device, which can be used to become familiar with the complex protocol that handles this connection standard, among other aspects we can highlight the information packets used, *URBs*, and the steps in the communication between the USB device and the host. As well as the development of drivers for USB devices, using the synchronous and asynchronous API of the Linux kernel.

In addition, it also aims to become familiar with the different devices and sensors, such as displays or temperature readers, which are used connected to the USB device developed in this project.

6.6 Work plan

For the development of this project, several meetings have been held with the project managers and among the developers. In the initial part of the project, the operation of the V-USB library has been studied with different projects using other microchips, to see what use is made of the report-IDs and the different functions used in the code. The Wireshark software [1] has been used to study the endpoints used and the URBs packets in the communication, to study the software configuration and the possibility of using, for example, interrupts.

For the development of the peripherals to be used together on the Digispark board, Guillermo has been in charge of studying and developing the code corresponding to the temperature sensor and Javier to the one corresponding to the 2x16 LCD display, which later an OLED LCD display has been used.

For the integration of the code, GitHub has been used.

6.7 Organization of the report

For the realization of this report, it has been divided into several chapters. In the first chapter, the hardware used and some low-level aspects of the library used are explained as an introduction. In the next chapter, the hardware elements used, such as the circular LED ring or the OLED LCD screen, are explained. Then, the microcontroller finally used for the final prototype is explained in detail, since it does not have the limitations that we had with the ATTiny85, as it can have more lines of communication with it.

In the appendices, we have the contributions of each member of the project, as well as different instructions to analyze the USB URBs packets with the Wireshark software, or also to *flash* our own firmware, with indications on the specific pins to use on the board.

Guía para flashear un firmware propio

La placa *Digispark ATTiny85* tiene de forma predeterminada un bootloader llamado *Micronucleus* que puede comunicarse con la máquina host a través de una comunicación serie, lo que permite al usuario la opción de flashear código a través del puerto USB y evita el uso de un programador *USBASP*.

Esta característica puede ser útil, pero hay ciertos casos en los que es posible que necesitemos actualizar un gestor de arranque personalizado, firmware *C/C++* o cualquier tipo de programa que necesite borrar completamente la ROM, y esto no se puede hacer usando *Micronucleus*.

6.8 Hardware

- **ICSP Programmer:** lo usaremos para hablar con la ROM integrado en el chip. *USBASP* es el que se va a usar, pero cualquier programador con pines *MISO/MOSI/SCK/RST* funcionará, incluso con un *Arduino* flasheado con el sketch *ArduinoISP*.
- **Digispark ATTiny85**

6.9 Software

- **Arduino IDE:** usaremos las herramientas de *AVR* integradas en el entorno.

6.10 Diagrama de pines

ISCP Programmer	Digispark ATTiny85
MOSI	PB0
MISO	PB1
SCK	PB2
RESET	PB5
5V	5V

ISCP Programmer	Digispark ATTiny85
GND	GND

6.11 Cómo flashear un sketch de Arduino

Lo primero, es hacer click en el menú *Tools* y seleccionar la placa correcta y la CPU. Después, seleccionamos el programador que estamos usando.

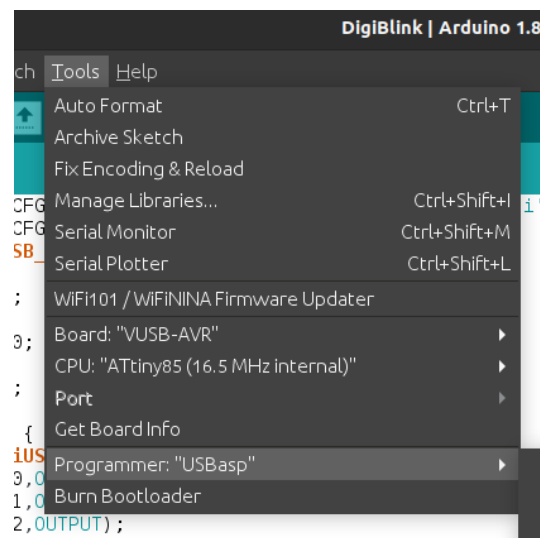


Figura 6.1: Menú de herramientas de Arduino IDE

Una vez que hayamos establecido los parámetros correctos, sólo queda iniciar el proceso de flasheo del firmware y confiar en que todo funcione correctamente. Si ha ido bien, el entorno nos mostrará un mensaje como el siguiente:

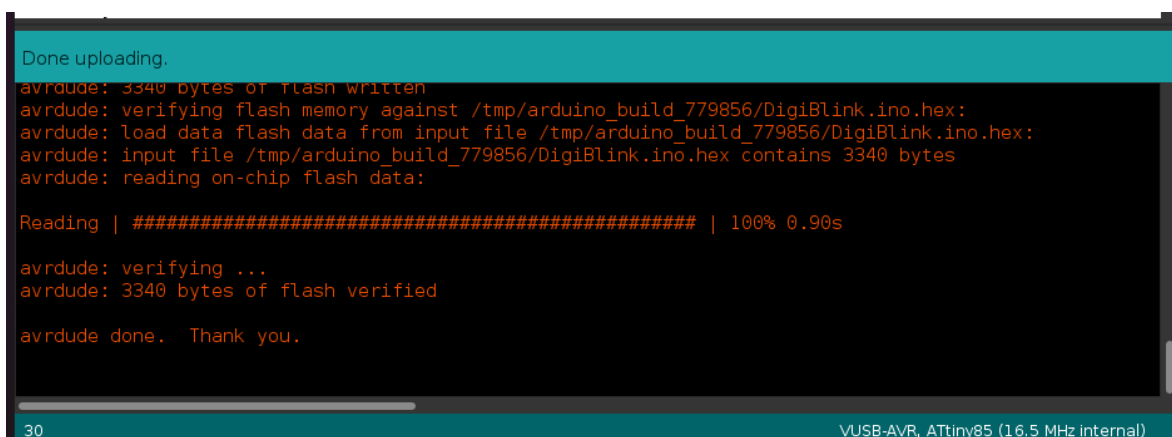


Figura 6.2: Mensaje de salida una vez flasheado el firmware

6.12 Flashear un firmware propio a partir de un archivo .hex

Actualizar un archivo .hex existente puede ser útil si queremos instalar un *firmware/cargador de arranque* precompilado o cargar un programa *C/C++*.

Arduino viene con algunas utilidades *AVR* que podemos usar desde nuestra terminal para compilar y flashear código a *microcontroladores AVR*.

Lo primero es ubicar nuestro directorio *Arduino IDE*, a continuación se muestra una forma de hacerlo:

```
$ ls -l `which arduino`  
lrwxrwxrwx root root 73 B Tue Jul 12 16:44:01 2022  
/usr/local/bin/arduino /home/ggc/arduino-1.8.19/arduino
```

Una vez que sepamos dónde se encuentra, busquemos las siguientes utilidades y archivos:

- `arduino-1.8.19/hardware/tools/avr/bin/avrdude`
- `arduino-1.8.19/hardware/tools/avr/etc/avrdude.conf`

El siguiente paso es flashear el archivo .hex usando la línea de comandos con los siguientes argumentos:

- **-C**, archivo de configuración del entorno *Arduino IDE* por defecto
- **-v**, verbose output
- **-p**, microcontrolador
- **-c**, programador
- **-P**, puerto
- **-U**, instrucción. Sintaxis: *memtype:op:filename[:format]*

```
arduino-1.8.19/hardware/tools/avr/bin/avrdude  
-Carduino-1.8.19/hardware/tools/avr/etc/avrdude.conf  
-v -pattiny85 -cusbasp -Pusb -Uflash:w:main.hex:i
```

Para encontrar más argumentos del programa `avrdude` y una explicación más profunda, se recomienda visitar el siguiente enlace: https://www.nongnu.org/avrdude/user-manual/avrdude_3.html

Una vez terminado el proceso, se mostrará un mensaje como el siguiente:

```
Reading | ##### | 100% 0.79s
```

```
avrdude: verifying ...
```

```
avrdude: 2890 bytes of flash verified
```

```
avrdude: safemode: lfuse reads as E1
```

```
avrdude: safemode: hfuse reads as DD
```

```
avrdude: safemode: efuse reads as FF
```

```
avrdude: safemode: Fuses OK (E:FF, H:DD, L:E1)
```

```
avrdude done. Thank you.
```

```
gpgc ~/Documents/Estudios/Software/TFG/resources/testFirmware ws2812
```

Monitorizar el tráfico USB usando Wireshark

En este anexo se explica cómo capturar y mostrar el tráfico USB con la interfaz GUI de Wireshark que hace que sea muy fácil interactuar con ella.

Todo el software y las pruebas se han realizado bajo Ubuntu 20.04.5 LTS con el kernel 5.15.0-48-generic.

6.13 Software

Todos los paquetes necesarios se pueden instalar desde los repositorios de Ubuntu usando apt.

```
sudo apt install wireshark libpcap0.8
```

Además, el módulo `usbmon` debe estar habilitado para acceder al tráfico usb.

```
sudo modprobe usbmon
```

6.14 Abrimos Wireshark

Como vamos a acceder al hardware físico, debemos abrir Wireshark como superusuario. Para ello, procedemos a lanzarlo desde la terminal de la siguiente manera:

```
sudo wireshark
```

6.15 Seleccionamos la interfaz `usbmonX` correcta

Una vez que hayamos abierto Wireshark, veremos algunas interfaces `usbmon` desde las que podemos monitorear. Para seleccionar la interfaz correcta, debemos verificar a qué bus está conectado nuestro dispositivo USB. Podemos usar `lsusb` para esa tarea.

```
$ lsusb
```

```
Bus 001 Device 067: ID 16c0:05df Van Ooijen Technische Informatica HID device
```

En este caso, elegiremos *usbmon1* ya que nuestro dispositivo está conectado al bus 001. Además, podemos ver que el *DeviceID* es el número 67, esto será útil más adelante para el filtro de visualización.

6.16 Empezamos a capturar tráfico

Para comenzar a capturar tráfico, solo necesitamos hacer doble clic en la interfaz *usbmon* correcta y automáticamente comenzará a mostrar líneas con los paquetes en tránsito. La mayor parte del tráfico que se muestra no nos será útil, por lo que lo mejor ahora es aplicar algunos filtros para mostrar solo el tráfico relacionado con nuestro dispositivo USB.

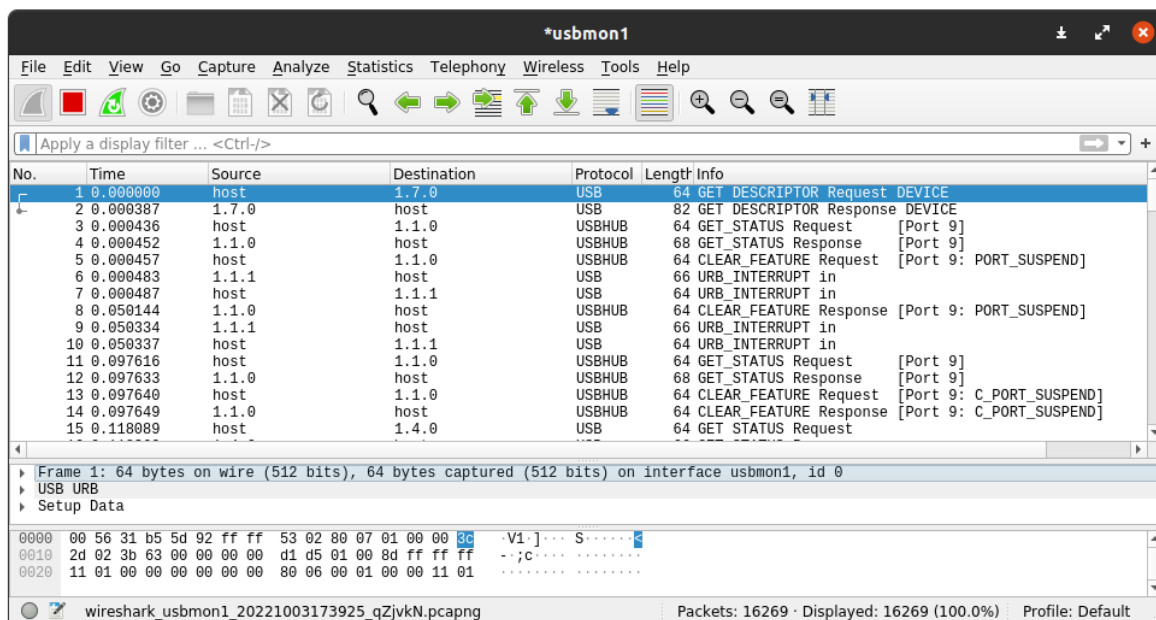


Figura 6.3: Captura mostrando el tráfico USB de Wireshark capturado

6.17 Filtrar el tráfico mostrado

Como se ha mencionado previamente, la mejor manera de mostrar el tráfico que nos interesa es usar filtros de visualización.

La sintaxis es muy simple, por lo que podemos escribirla nosotros mismos o hacer clic derecho en un paquete desde nuestro dispositivo y prepararlo como un filtro. En la línea de filtros, aplicamos lo siguiente:

```
usb.src == "1.67.0" || usb.dst == "1.67.0"
```

Donde el primer número corresponde al bus, el segundo al *device ID* y el tercero al *endpoint ID*.

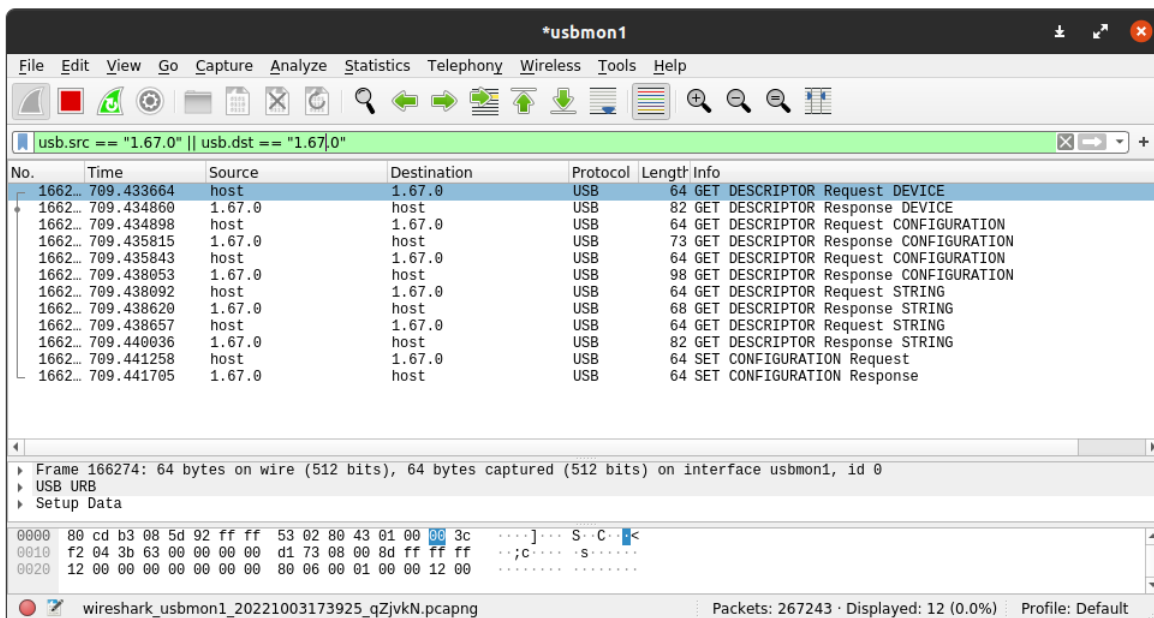


Figura 6.4: Ejemplo de un filtro aplicado

Bibliografía

- [1] G. Combs, «Wireshark software for monitoring networks». <https://www.wireshark.org/>, 1998.
- [2] Microchip, «Family of AVR microcontrollers». <https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/8-bit-mcus/avr-mcus#/>, 1998.
- [3] Microchip y wikipedia, «Specifications about AVR microcontrollers». https://en.wikipedia.org/wiki/AVR_microcontrollers/, 1998.
- [4] Microchip, «ATTiny85 microchip, specifications». <https://www.microchip.com/en-us/product/ATtiny85/>, 1998.
- [5] Microchip, «ATTiny85 datasheet». https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2586-AVR-8-bit-Microcontroller-ATtiny25-ATtiny45-ATtiny85_Datasheet.pdf/, 2000.
- [6] adnanaqeel, «Description and specifications about ATTiny85». <https://www.theengineeringprojects.com/2018/09/introduction-to-attiny85.html#:~:text=ATTiny85%20is%20an%208%20Dbit,can%20operate%20on%20minimum%20power./>, 2018.
- [7] OBDEV, «V-USB open-source Project example projects for LCD Display». <https://www.ob-dev.at/products/vusb/projects.html>, 2011.
- [8] S. Wagner, «VUSB-AVR firmware». <https://github.com/wagiminator/VUSB-AVR/>, 2021.
- [9] F. Zhao, «USB HID Report Descriptors». <https://eleccelerator.com/tutorial-about-usb-hid-report-descriptors/>, 2013.
- [10] AZ-Delivery, «Digispark board, specifications». <https://www.az-delivery.de/en/products/digispark-board/>, 2021.
- [11] Digistump, «Set up environment with Digispark». <http://digistump.com/wiki/digispark/>, 2000.
- [12] NeoPixel, «Specifications about LED strips». <https://learn.adafruit.com/adafruit-neopixel-uberguide/neopixel-strips/>, 2020.

[13] DigiKey, «How Smart Shift Register LEDs Work». https://www.youtube.com/watch?v=PPVi3bI7_Z4/, 2020.

[14] Sysplay, «Drivers in Linux using USB». <https://sysplay.github.io/books/LinuxDrivers/book/Content/Part11.html>, 2005.