industriales
etsii

Escuela Técnica Superior de Ingeniería Industrial

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería Industrial

Desarrollo de librerías de comunicación para SDI-12 bajo uPython

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

Autor: **Guillermo Pagán Martínez**
Director: Roque Torres Sánchez
Codirector: Ana Belén Toledo Moreo

Cartagena, 10/07/2019

Universidad Politécnica de Cartagena

Índice

1. Objetivo del trabajo

El objetivo de este proyecto es desarrollar funciones para microcontroladores ESP32[1] programados en MicroPython[2] con la finalidad de comunicarse con sensores y actuadores de uso agrícola y medioambiental bajo protocolo SDI-12[3].

Ya existen programas en MicroPython que permiten esta comunicación, pero emplean dos líneas para ello, una para enviar y otra para recibir los datos, apoyándose en lo que se conoce como UART[4] (Figura 1). El objetivo de este trabajo es conseguir esta misma comunicación pero empleando para ello un único pin, tanto para enviar como para recibir datos.

Debido a que los sensores que usan el protocolo SDI-12 únicamente disponen de un canal físico para transmisión y recepción de datos, emplear un único pin aporta numerosas ventajas frente al uso de dos, como la reducción del hardware externo necesario para llevar a cabo la transmisión de datos o la simplicidad de dicha transmisión. Esto implica una reducción del tiempo, trabajo y dinero necesarios para fabricar los controladores de los sensores.

En los programas existentes es necesario emplear un adaptador tri-estado (Figura 1). Este adaptador se conecta a los canales Rx y Tx de la UART (Ver Figura 2) mientras que el ACT_SDI12 es lo que realiza la conmutación entre recepción y transmisión en el hilo SDI-12.

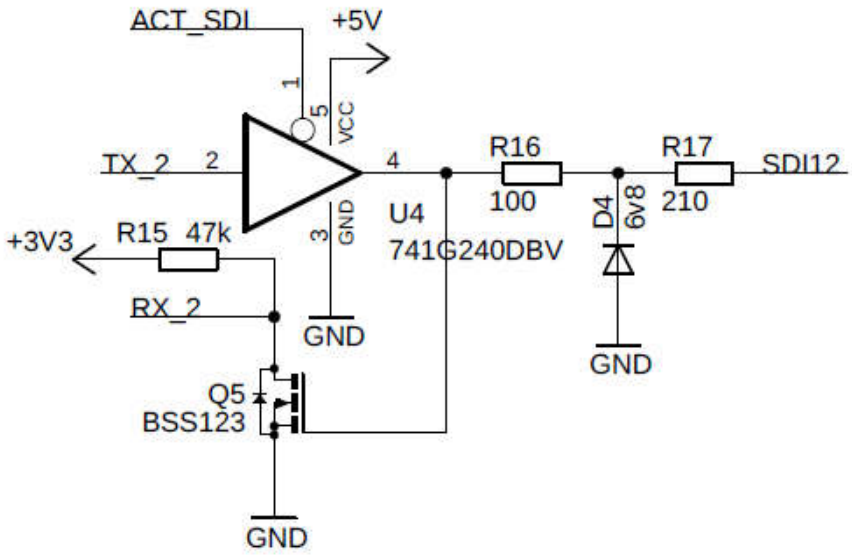

Figura 1. Adaptador tri-estado.

[1] Los dispositivos ESP-32 son una serie de microcontroladores System On a Chip que se caracterizan por poseer conectividad WiFi y Bluetooth integrada y por emplear un microprocesador Tensilica Xtensa LX6, entre otras características. Están creados y desarrollados por Esperessif Systems como un sucesor a los microcontroladores ESP8266. (https://en.wikipedia.org/wiki/ESP32)

[2] MicroPython es una implementación del lenguaje de programación Python 3 optimizada para funcionar en microcontroladores y entornos restringidos. (https://micropython.org/)

[3] Ver apartado 5

[4] Ver apartado 3

2. Propuesta de resolución

Para alcanzar el objetivo deseado, será necesario programar una UART que permita la comunicación en ambos sentidos, transformando caracteres en niveles lógicos y viceversa.
Para ello, en primer lugar, se explicará el funcionamiento de una UART de forma breve y se presentará el microchip a emplear en el proyecto. Posteriormente se realizará un estudio en profundidad del protocolo SDI-12, sus tiempos y su estructura con el objetivo de desarrollar una librería para la comunicación entre los sensores y el microcontrolador.
Una vez explicados estos conceptos, se abordará de manera más específica la resolución.

3. Funcionamiento de una UART

"UART, son las siglas en inglés de *Universal Asynchronous Receiver-Transmitter*, en español: Transmisor-Receptor Asíncrono Universal. Es el dispositivo que controla los puertos y dispositivos serie.
Las funciones principales de chip UART son: manejar las interrupciones de los dispositivos conectados al puerto serie y convertir los datos en formato paralelo, transmitidos al bus de sistema, a datos en formato serie, para que puedan ser transmitidos a través de los puertos y viceversa." (Wikipedia, *Universal Asynchronous Receiver-Transmitter*)

El objetivo de una UART es transmitir, recibir y gestionar datos, permitiendo las comunicaciones entre dispositivos. Para ello, su funcionamiento se basa en la traducción de información y el envío o recepción de la misma. De esta forma, la UART convierte los caracteres a enviar a binario con una estructura específica que puede ser identificada por la otra UART que recibirá el mensaje.

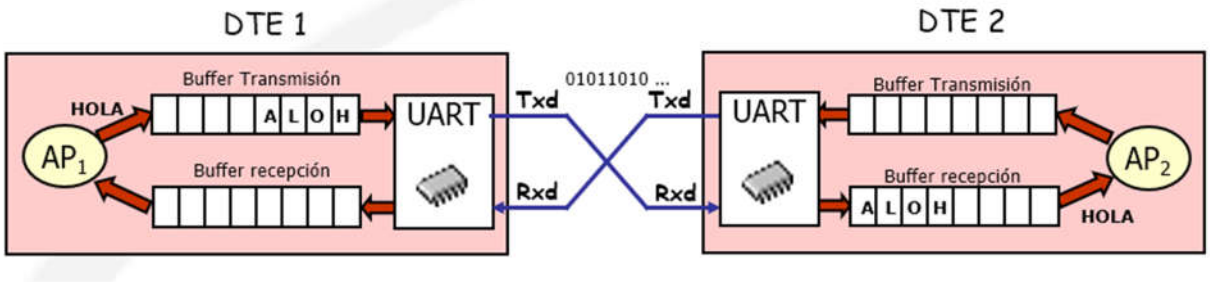

Figura 2. UART

Fuente: Asignatura de Comunicaciones Industriales. Tema 2. (UPCT)

La velocidad con la que se envían o reciben estos mensajes depende del baudrate. Este baudrate se mide en baudios, que representan el número de símbolos por segundo. El tiempo de bit (tiempo que dura la transmisión de un único bit) no tiene por qué coincidir con el baudrate, si bien en este caso sí lo hace, debido a que los eventos son únicamente cambios de voltaje (ver Tabla 1 posteriormente). Por tanto, la duración de un bit será (en este caso) la inversa del baudrate:

$$Tiempo\ de\ bit = \frac{1}{baudrate}\ (segundos\ por\ bit)$$

4. Microcontrolador empleado: Pycom WiPy 3.0

Antes de comenzar con la explicación del protocolo SDI-12, conviene mencionar el hardware empleado en el proyecto: el Sistem On Chip (SoC) WiPy 3.0, de Pycom. Se trata de una plataforma de desarrollo alimentada con código en MicroPython y basada en ESP-32 realizada por la compañía Pycom. Emplea un microprocesador de doble núcleo Tensilica Xtensa LX6[5], y al igual que el resto de microcontroladores ESP-32, también posee conectividad WiFi y Bluetooth. Además, posee un alcance de WiFi de 1 km, tiene un tamaño muy reducido y un bajo consumo de energía.

Se elige dicho System On Chip debido a que el grupo en el que se enmarca este proyecto emplea nodos WiPy para la realización de despliegues en campo que recogen y envían datos de sensores que utilizan protocolo SDI-12 (entre otros).
Posteriormente, durante el desarrollo de la librería, aparecerán algunas limitaciones para el desarrollo de esta que se deben a la elección de este SoC.

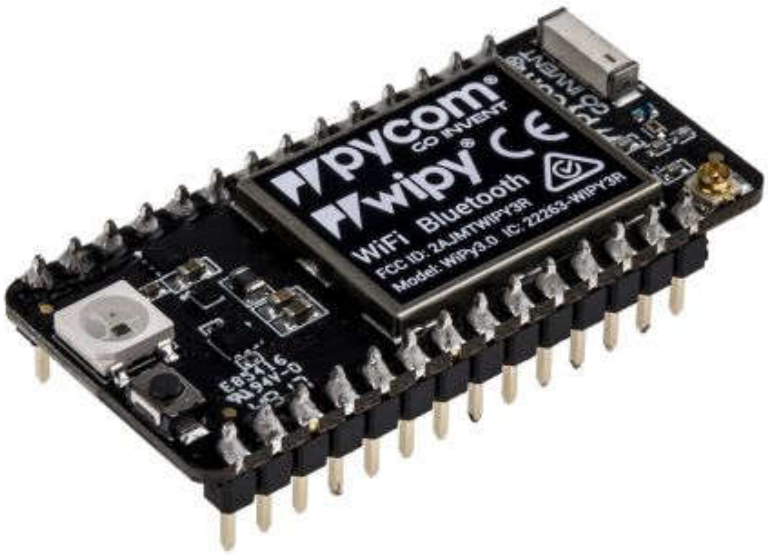

Figura 3. WiPy 3.0

Fuente: https://pycom.io/product/wipy-3-0/

5. El protocolo SDI-12

A continuación se realizará una exposición sobre los aspectos del protocolo SDI-12 más relevantes para el proyecto. Esta información está extraída directamente de la página web del grupo de apoyo del SDI-12 (http://www.sdi-12.org/specification.php). Se trata de información de dominio público cuya copia y distribución gratuita está permitida.

[5] Microprocesador fabricado por la compañía Tensilica, ahora parte de Cadence Design Systems, que se caracteriza por sus posibilidades de customización y configuración. (https://en.wikipedia.org/wiki/Tensilica)

5.1. Introducción

El protocolo SDI-12 es un estándar de comunicación entre los llamados "data recorders" (registradores de datos) y los sensores basados en microprocesadores. Las siglas SDI-12 significan Serial/Digital Interface at 1200 baud, es decir, interfaz serial/digital a 1200 baudios.

Este protocolo se emplea cuando se tienen los siguientes requerimientos:

- Realización de operaciones donde la energía se suministra con batería y se desea un mínimo consumo de corriente.

- Bajo coste del sistema

- Uso de un simple registrador de datos con múltiples sensores empleando un solo cable.

5.2. Utilización

Los sensores SDI-12 no son la opción más económica, pero poseen ventajas que los hacen ideales en determinadas aplicaciones. La posibilidad de conectar un gran número de sensores a un mismo "data recorder" es una de esas ventajas, al igual que su amplia compatibilidad. Este protocolo de comunicaciones fue desarrollado para el sector de los recursos hídricos, estando sus aplicaciones principales en el campo de la agricultura.

Hay multitud de sensores que se encargan de medir distintas variables. Las más significativas son:

- Humedad del suelo
- Potencial de agua
- Nivel de agua
- Conductividad eléctrica
- Temperatura
- Humedad relativa
- Presión de vapor y barométrica
- Velocidad del aire

La medida de estas variables permite un análisis adecuado del hábitat y salud de la planta, así como la realización de estudios de estos factores.
Por ejemplo, la medida de la humedad del suelo y el potencial y nivel del agua permiten determinar con exactitud qué presencia de agua hay en el terreno y hacia dónde se mueve, mostrando qué cantidad de dicho agua está disponible para la planta.

5.3. Conexión eléctrica

Los sensores y data recorders se comunican a través del bus SDI-12, un cable que conecta varios dispositivos SDI-12. Es posible conectar hasta 10 sensores con una longitud de cable de unos 60

metros. Esta distancia se puede aumentar reduciendo el número de sensores. Dentro de dicho cable se encuentran 3 líneas:

- Línea de datos
- Línea de 12 V
- Línea de tierra

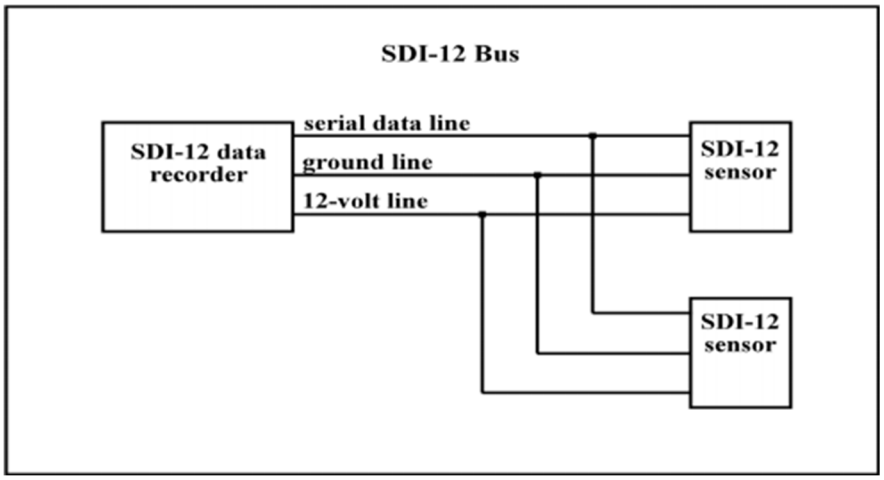

Figura 4. Esquema de líneas del cable SDI-12

Fuente: http://www.sdi-12.org/specification.php

El canal de transmisión es una línea de datos bidireccional con 3 estados posibles, detallados en la Tabla 1:

Tabla 1. Estados de la línea de datos

Fuente: http://www.sdi-12.org/specification.php

Condition	Binary state	Voltage range
marking	1	-0.5 to 1.0 volts
spacing	0	3.5 to 5.5 volts
transition	undefined	1.0 to 3.5 volts

El circuito equivalente de conexión para los sensores se encuentra en la Figura 5:

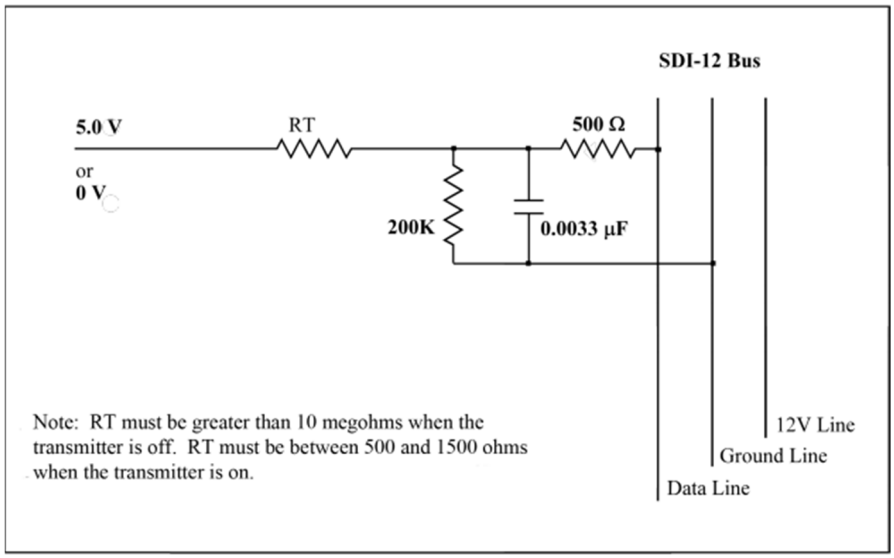

Figura 5. Circuito equivalente

Fuente: http://www.sdi-12.org/specification.php

5.4. Protocolo de comunicaciones

Antes de entrar en lo propio del protocolo SDI-12, es conveniente explicar los fundamentos de los protocolos maestro-esclavo, entre los que se encuentra este protocolo.

Un protocolo maestro-esclavo es un procedimiento de comunicación en el que hay dos tipos de elementos, los maestros y los esclavos. En este tipo de protocolos son los maestros los que se encargan de ordenar a los esclavos que realicen funciones como la toma de datos o el envío de estos mientras que los maestros se encargan del control de los ciclos y la temporización. En el protocolo SDI-12, los sensores son los esclavos y el "data recorder" es el maestro. (Ver Figura 4).

La comunicación en el protocolo SDI-12 se realiza mediante el intercambio de caracteres ASCII en la línea de transferencia de datos de la siguiente forma:
En primer lugar, el data recorder despierta a los sensores conectados en la línea de datos (que se encuentran en modo bajo consumo para ahorrar energía) con un "break", un espaciado continuo en la línea de datos durante más de 12 milisegundos.
Después del break el data recorder envía el comando a uno de los sensores, y se recibe una respuesta de este, que se guarda en el data recorder.
El primer carácter que se envía en un comando se corresponde con la dirección de uno de los sensores (con excepción del comando de consulta de dirección), de forma que este será el único que responda al comando, mientras que los demás lo ignoran y permanecen en espera.
Un típico ejemplo de comunicación entre el sensor y el data recorder para obtener una medida es el siguiente:

Paso 1: Se despiertan todos los sensores del bus de datos con un break.

Paso 2: Se envía un comando a un sensor específico, ordenándole que tome una medida.

Paso 3: El sensor responde en menos de 15 milisegundos, devolviendo el tiempo máximo en el que las medidas estarán listas y el número de valores a devolver.

Paso 4: Si la medida no está inmediatamente disponible, el data recorder debe esperar hasta que se cumpla el tiempo de espera para poder enviar el comando que ordena al sensor transmitir los datos.

Paso 5: Después de enviar dicho comando, el sensor devuelve las medidas realizadas y vuelve a ponerse en modo de bajo consumo.

5.4.1. Estructura de los bytes y caracteres

Cada byte en protocolo SDI-12 equivale a un carácter y está compuesto por los 10 bits siguientes:

0 0000000 0 1

- Bit de inicio (Nivel positivo de tensión)
- 7 bits de datos, codificados según el código USASCII (bit menos significativo se transmite primero) y con polaridad inversa.
- Bit de paridad (par)
- Bit de parada (Nivel negativo de tensión)

Respecto a los caracteres para la comunicación, todos deben ser caracteres ASCII imprimibles. Tanto la codificación de bits como sus respectivos caracteres se encuentran en la Tabla 2:

Tabla 2. Conversión binario-USASCII

Bits ***765*** / ***4321***	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DEL	SP	0	@	P		p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	:	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

5.4.2. Comandos y respuestas

Como se ha dicho anteriormente, el primer carácter de un comando es la dirección del sensor al que va dirigido (con la excepción del comando de consulta de dirección). Del mismo modo, la respuesta de dicho sensor tendrá como primer carácter su dirección, de forma que sirva de comprobación de que ha respondido el sensor correcto.

Las direcciones de los sensores son un simple carácter. Por defecto, los sensores tienen inicialmente asignada la dirección 0, pero este valor puede cambiarse.
Las direcciones estándar son los caracteres "1" a "9", soportadas por todos los sensores. Si fuera necesario disponer de más direcciones, se pueden usar en el rango de ASCII "A" hasta ASCII "Z" y de ASCII "a" hasta ASCII "z".

En la Tabla 3 Se encuentra una lista de los comandos disponibles. Aquellos que son especialmente relevantes para el proyecto se encuentran recuadrados y se explicarán posteriormente.

Tabla 3. Lista de comandos

Fuente: http://www.sdi-12.org/specification.php

Name	Command	Response
Break	Continuous spacing for at least 12 milliseconds	None
Acknowledge Active	a!	a<CR><LF>
Send Identification	aI!	allccccccccmmmmmmmvvvxxx...xx<CR><LF>
Change Address	aAb!	b<CR><LF> (support for this command is required only if the sensor supports software changeable addresses)
Address Query	?!	a<CR><LF>
Start Measurement*	aM!	atttn<CR><LF>
Start Measurement and Request CRC*	aMC!	atttn<CR><LF>
Send Data	aD0!	a<values><CR><LF> or a<values><CRC><CR><LF>
	.	a<values><CR><LF> or a<values><CRC><CR><LF>
	.	a<values><CR><LF> or a<values><CRC><CR><LF>
	.	a<values><CR><LF> or a<values><CRC><CR><LF>
	aD9!	a<values><CR><LF> or a<values><CRC><CR><LF>
Additional Measurements*	aM1!	atttn<CR><LF>
	.	atttn<CR><LF>
	.	atttn<CR><LF>
	.	atttn<CR><LF>
	aM9!	atttn<CR><LF>
Additional Measurements and Request CRC*	aMC1! ... aMC9!	atttn<CR><LF>
Start Verification*	aV!	atttn<CR><LF>
Start Concurrent Measurement	aC!	atttnn<CR><LF>
Start Concurrent Measurement and Request CRC	aCC!	atttnn<CR><LF>
Additional Concurrent Measurements	aC1!	atttnn<CR><LF>
	.	atttnn<CR><LF>
	.	atttnn<CR><LF>
	.	atttnn<CR><LF>
	aC9!	atttnn<CR><LF>
Additional Concurrent Measurements and Request CRC	aCC1! ... aCC9!	atttnn<CR><LF>
Continuous Measurements	aR0! ... aR9!	a<values><CR><LF> (formatted like the D commands)
Continuous Measurements and Request CRC	aRC0! ... aRC9!	a<values><CRC><CR><LF> (formatted like the D commands)

Como se puede observar en la tabla, cuando los comandos son enviados por el maestro terminan en "!". Este símbolo solo se puede usar en un comando como carácter final e indica que dicho comando ha terminado. Todas las respuestas de los sensores acaban con un "carriage return" (CR, en la práctica \r) y un "lane feed" (LF, en la práctica \n).
El máximo número de caracteres que puede devolver un sensor en el apartado de <values> es 35, excepto si el comando D (para enviar las medidas) se emplea como respuesta a un comando de medida continua o a uno de medida de alto volumen, donde el máximo en ambos casos es de 75 caracteres.

5.4.2.1. Comando de Reconocimiento de Actividad (a!)

Este comando se emplea para comprobar que el sensor responde de forma correcta. Los caracteres que lo componen son los siguientes:

- "a": La dirección del sensor
- "!": Fin del comando

El sensor responde con los siguientes caracteres:

- "a": La dirección del sensor
- \r(CR) y \n (LF): Terminan la respuesta del sensor

5.4.2.2. Comando de Envío de Identificación (aI!)

Este comando ordena al sensor enviar información sobre su versión SDI-12, vendedor, modelo o número de serie. Se compone de los siguientes caracteres:

- "a": La dirección del sensor
- "I": Comando de identificación
- "!": Fin del comando

La respuesta del sensor tiene la siguiente forma:

$$allcccccccmmmmmmvvvxxx \dots xxx\backslash r\backslash n$$

Que se corresponden con la siguiente información:

- "a": La dirección del sensor
- "ll": Dos números que indican la versión de compatibilidad SDI-12.
- "cccccccc": Ocho caracteres que identifican al vendedor, generalmente se trata del nombre de una empresa o su abreviación.
- "mmmmmm": Seis caracteres especificando el número de modelo del sensor.
- "vvv": Tres caracteres correspondientes a la versión del sensor.
- "xxx…xxx": Un campo opcional de hasta 13 caracteres que se emplea para aportar un número de serie u otra información no relevante para el funcionamiento del data recorder.
- \r(CR) y \n (LF): Terminan la respuesta del sensor

5.4.2.3. Comando de Consulta de Dirección (¿!)

Al emplearse en un comando el signo de interrogación "?" en vez de la dirección del sensor, el sensor responde con su dirección de forma semejante a como responde al comando "a!". De esta forma es posible determinar la dirección del sensor cuando es desconocida.
Sin embargo, si este comando se envía cuando hay más de un sensor conectado al bus, todos responderán, causando un colapso del bus e impidiendo su correcta lectura.

5.4.2.4. Comando de Cambio de Dirección (aAb!)

Con este comando es posible cambiar la dirección del sensor. Está conformado por los siguientes caracteres:

- "a": La dirección del sensor
- "A": Comando de cambio de dirección
- "b": La nueva dirección del sensor
- "!": Fin del comando

El sensor responderá a este comando con su nueva dirección "b", de forma que se pueda confirmar que se ha efectuado dicho cambio.
La respuesta del sensor es, por tanto:

- "b": La nueva dirección del sensor
- \r(CR) y \n (LF): Terminan la respuesta del sensor

5.4.2.5. Comando de Inicio de Medida (aM!)

Este comando es el más importante, ya que es el que ordena al sensor efectuar una medida. Sin embargo, funciona de forma diferente al resto de comandos. En primer lugar se envía el comando:

- "a": La dirección del sensor
- "M": Comando de medida
- "!": Fin del comando

Tras recibir el comando, el sensor inicia su proceso de medida, pero no la devuelve automáticamente. En su lugar devuelve una respuesta que tiene la siguiente forma:

$$atttn\backslash r\backslash n$$

- "a": La dirección del sensor
- "ttt": El tiempo, en segundos, que tarda el sensor en tener la medida lista.
- "n": El número de medidas que el sensor realizará.
- \r(CR) y \n (LF): Terminan la respuesta del sensor

Tras recibir este comando, el data recorder deberá esperar el tiempo especificado antes de poder enviar un comando D0 para recoger las medidas.

Si el sensor recibe un break antes de que se haya cumplido el tiempo especificado para tomar la medida, se abortará la toma de esta.

5.4.2.6. Comando de Envío de Datos (aD0!)

Después de cumplirse el tiempo especificado por el sensor como respuesta a un comando de medida, el data recorder debe enviar el comando D0 para ordenar el envío de las medidas realizadas por el sensor. En el caso de que el sensor no responda con todas las medidas esperadas ante el comando D0, se pueden emplear los comandos D1, D2, etc. Hasta que se hayan recibido los valores de todas las medidas.

La respuesta a este comando es la siguiente:

$$a < valores > \backslash r \backslash n$$

- "a": La dirección del sensor
- "<valores>": Se devuelven tantas medidas como haya efectuado el sensor como respuesta a un comando D0. Las medidas son devueltas con la forma "pd.d":
  - "p": Signo de polaridad (+ o -)
  - "d": Dígitos numéricos enteros
  - ".": El punto decimal (opcional)
  - "d": Dígitos numéricos decimales

- \r(CR) y \n (LF): Terminan la respuesta del sensor

5.4.3. Esquema de tiempos

Probablemente, la parte más importante a la hora de comprender el protocolo SDI-12 de cara al proyecto es el tiempo en el que sucede cada uno de los eventos anteriormente mencionados.

En la Figura 6 se encuentra un esquema temporal a tener en cuenta con sensores SDI-12.

Este esquema permite tener una visión general de los tiempos que hay que respetar a la hora de comunicarse con los sensores. Sin embargo, el dato clave a tener en cuenta es la velocidad de transmisión, que es de 1200 baudios, lo que equivale en el protocolo SDI-12 a una duración de bit de unos 833 microsegundos.

Estos datos permitirán el desarrollo de la UART con la que poder intercambiar información con los sensores.

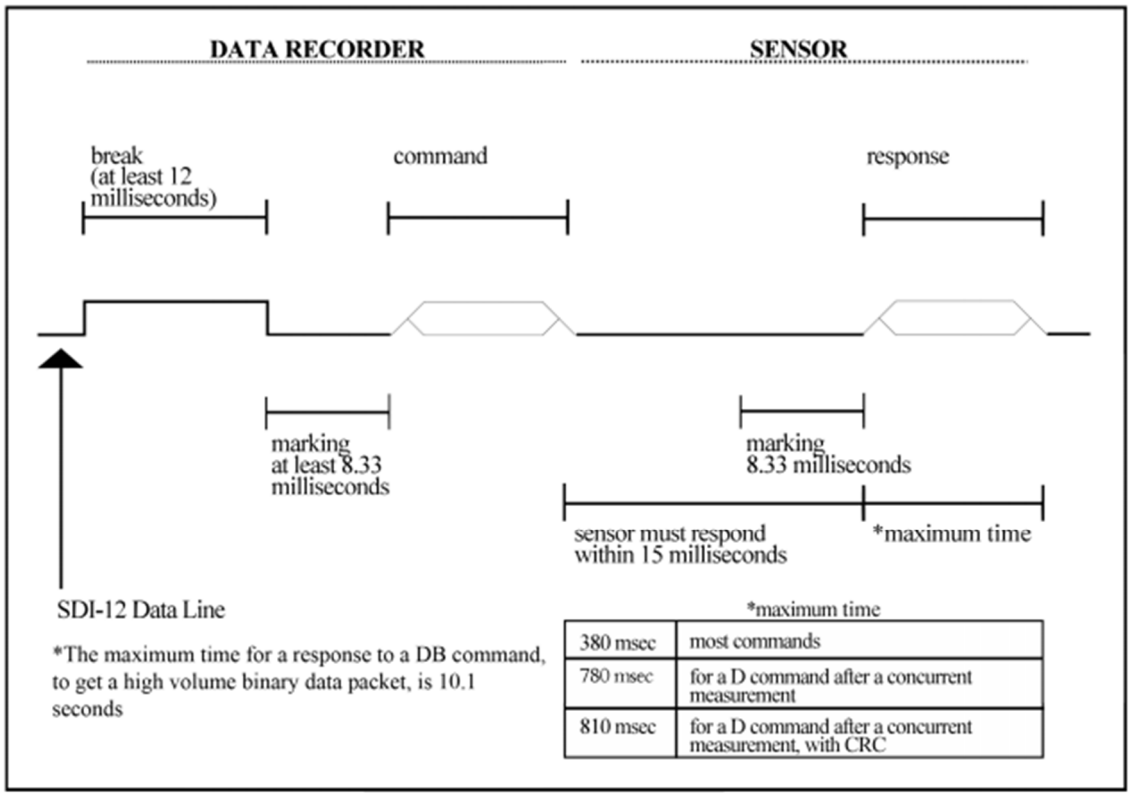

Figura 6. Esquema temporal

Fuente: http://www.sdi-12.org/specification.php

6. Desarrollo de la UART

En este apartado se realizará la explicación del código, dividiéndose en segmentos según su propósito. La totalidad del código se puede encontrar en el apéndice.

Como anotación previa a la explicación del código en sí, es conveniente mencionar una serie de cuestiones.

En primer lugar, para llegar a comprender correctamente los sensores, su funcionamiento y su forma de transmitir datos, se ha empleado un osciloscopio. El uso de este dispositivo ha sido crucial en numerosas situaciones para el avance del proyecto, al permitir leer e identificar los pulsos enviados y recibidos, comprobar que son correctos, y realizar comprobaciones, como se verá en los apartados anteriores.

También hay que indicar que se han estructurado las funciones como parte de una clase común. Esto permite una mejor organización, así como la utilización de varios pines GPIO, ya que la clase puede emplear como atributo inicial el pin que se utilizará para la comunicación.
Del mismo modo, los atributos de clase son una forma más útil y conveniente de guardar variables, permitiendo su uso y modificación por varias funciones pertenecientes a dicha clase. En ocasiones, y como se verá posteriormente, esto permite ahorrar el envío de argumentos a las funciones, ya que guardar los datos en atributos de clase los hace perfectamente accesibles desde ellas.

La UART debe ser capaz de enviar comandos al sensor y de leer la respuesta que se recibe de él. Para ello, se ha estructurado en tres partes:

- **Traducción.** Esta parte de la UART se encargará de transformar los caracteres de los que se compone el comando que queremos enviar a bytes con la estructura del protocolo SDI-12, así como transformar de nuevo dichos bytes en caracteres para poder interpretar la respuesta del sensor.
- **Envío.** Parte encargada de enviar los caracteres por la línea de datos una vez han sido traducidos a bytes por medio de cambios de estado de un pin GPIO.
- **Lectura.** Su función es identificar los pulsos que se reciben en el pin GPIO y convertirlos en bytes adecuados para su posterior traducción y presentación.

6.1. Traducción

Como primer paso para llevar a cabo la traducción, se crean dos diccionarios en los que se almacenarán los caracteres imprimibles y sus correspondientes 7 bits. De esta forma, asignando el mismo número de entrada de los diccionarios al carácter y al código, podemos realizar una fácil traducción en cualquiera de los dos sentidos:

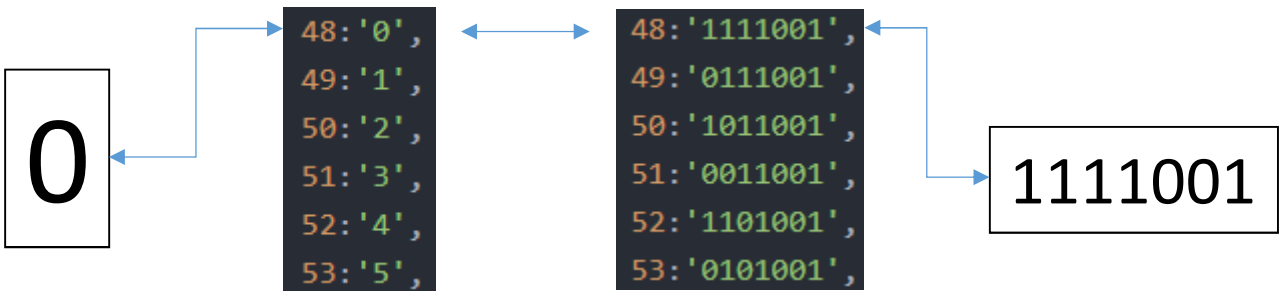

Figura 7. Conversión con diccionarios.

En el caso de que queramos enviar un carácter "0", la librería buscará número por número en el diccionario de caracteres hasta llegar al número que se corresponde con él, y usará ese número en el otro diccionario para obtener los bits. El funcionamiento en sentido contrario es semejante, con la diferencia de que en ese caso se busca en el diccionario de bits aquellos bits que coinciden con los introducidos y se extrae del otro diccionario el carácter que representan.
En el caso de la lectura, la obtención de los caracteres es el final de la traducción, pero si estamos enviando, será necesario calcular el bit de paridad y añadir los bits de inicio y final.

Para calcular el bit de paridad basta con saber que, al ser par, el número de bits de nivel lógico 1 debe ser par contando los 7 bits de datos y a sí mismo. Es decir, que si por ejemplo en los 7 bits de datos hay 3 bits de nivel 1 (o cualquier número impar), el bit de paridad será también de nivel lógico 1 para crear la paridad, y si se trata de un número par, será de nivel lógico 0 para mantener también dicha paridad. Este bit, una vez calculado, se añade a continuación de los 7 bits de datos.

Finalmente, basta con añadir el bit de inicio (1 al principio) y final (0 al final) para completar la estructura del byte. Es importante recordar que, tal y como se establece en la Tabla 1, los niveles lógicos positivos en el protocolo SDI-12 se corresponden con niveles de voltaje bajos, por lo que

en el código se emplean niveles lógicos invertidos de forma que cuando se le proporcione al pin el nivel lógico "1", se corresponda con un nivel de voltaje alto.

A continuación se realiza un análisis de código en MicroPython del caso de traducción para el envío. El caso de traducción en la recepción, al ser más simple (una vez obtenidos los bytes enviados por el sensor simplemente se extraen los 7 bits de datos y se obtiene el carácter que representan), se tratará con dicha parte.

Como entrada de la función de envío, se proporciona una cadena ("comando") que contiene los caracteres que se han de enviar. Dicha cadena debe tener un formato como el siguiente:

comando = ['a' , 'I' , '!']

De forma que se puedan elegir cada uno de los elementos e introducirlos en una variable de forma correcta, y que se puedan comparar con los caracteres del diccionario (deben estar en su mismo formato).

Las operaciones para crear el byte se han de repetir para cada uno de los caracteres que se envían, motivo por el cual se establece un "for" con el número de dichos caracteres.

```
for i in range (0, len(comando)):
    byte_enviar=list('1000000000')
    caracter=comando[i]
    caracter_binario=[]
    y=0
    while y < 128:
        if caracter==diccionario_usascii_caracteres[y]:
            caracter_binario=diccionario_usascii_invertido[y]
            break
        else:
            y+=1
    cadena_binario=list(caracter_binario)

    z=0
    while z < len(cadena_binario):
            byte_enviar[z+1]=cadena_binario[z]
            z+=1

    #Ahora hay que calcular el bit de paridad

    if ((byte_enviar.count('1')-1) % 2) == 0:
        byte_enviar[8]=0
    else:
        byte_enviar[8]=1

    #Byte (byte_enviar) listo para enviar, lo guardo en una lista
```

Figura 8. Traducción para envío de caracteres

A continuación se crea una lista con un byte predefinido, una variable con el carácter a enviar y otra en blanco donde se guardarán los bits de datos.
Una vez comparado el carácter con todos los contenidos del diccionario, se usa el número que le corresponde en el segundo diccionario para extraer los bits en dicha variable vacía.
Después, se añaden los bits de datos a la lista en blanco y se calcula el bit de paridad contando el número de bits positivos (menos el de inicio) y viendo si es par o impar.
Finalmente, se añade el byte a una lista de bytes que se enviarán posteriormente.

Para realizar la comparación de los caracteres a enviar con los guardados en los diccionarios, se emplea un "while" en el que se van comparando todos los caracteres del diccionario con el que enviamos, según el valor de "y". Cuando se encuentra dicho carácter, el valor asignado en el diccionario será el mismo de "y", y se emplea en el otro diccionario para extraer los 7 bits de datos.

La operación se repite tantas veces como bytes haya que enviar, y una vez completados, se avanza hacia la siguiente parte, que es el envío de dichos caracteres.

6.2. Envío

Como se ha mencionado anteriormente en el apartado del protocolo SDI-12, el envío de un comando a los sensores requiere que se les "despierte" previamente con un "break".
La función que se emplea para dicho cometido es la siguiente:

```
def __break(self):

    self.pin_unico=Pin(self.Pin_SDI12, mode=Pin.OUT, pull=None)


    self.pin_unico.value(1)
    time.sleep_ms(20)

    self.pin_unico.value(0)
    time.sleep_ms(20)

    self.pin_unico=Pin(self.Pin_SDI12, mode=Pin.OPEN_DRAIN)
```

Figura 9. Función "break"

En la que, después de definir el pin empleado como pin de salida, se le dan los valores 1 y 0 durante 20 ms cada uno. De esta forma nos aseguramos de que se cumplen las condiciones del break (espaciado en la línea de 12 ms mínimo), ya que si no iniciamos el break con el pin en 1, no podemos asegurar que ocurra esa transición a 0 y se mantenga durante 20 ms, ya que el pin podría estar bloqueado con otro valor.

Después del break se ejecuta la función de envío ("send") en la cual se ha omitido el contenido del "for" referente a la traducción, ya que se ha explicado antes su funcionamiento. También se puede apreciar cómo la función recibe como argumento la variable "comando", que se corresponde con la cadena de caracteres a enviar con el formato antes especificado.

La función, quitando la parte de la traducción, es muy sencilla. Al principio se vacía la lista de bytes que se enviaron la vez anterior, y se inicializa el pin en modo de salida. A continuación se realiza la traducción, y se guardan los bytes en la lista. En este momento, con todos los bytes guardados, estamos listos para enviarlos. Para ello, extraemos los bytes uno por uno de la lista, y proporcionamos al pin el valor de cada uno de sus bits durante el tiempo de duración que establece el protocolo SDI-12.

En este momento surgieron los primeros problemas de tiempos, debidos a que si el tiempo que se espera es de 833 microsegundos, el comando solo se enviaba correctamente en ciertas ocasiones, debido a que no se había tenido en cuenta el tiempo de ejecución del código. Para arreglarlo fue necesario ajustar el tiempo a esperar para que junto con el tiempo de ejecución del código, se cumpliera el tiempo que debía estar el byte en cada estado.
Para la realización de este ajuste fue muy útil el osciloscopio, ya que además de medir la anchura del pulso, permite ver si el sensor está respondiendo o no al comando, aunque no estemos preparados para leer la respuesta.

De esta forma, mediante dos "loops", se pueden enviar tantos caracteres como se quiera, gracias a que los bytes ya han sido guardados previamente. Esto es un aspecto importante que destacar, ya

que, si no se guardaran los bytes previamente, el tiempo de ejecución del código entre cada envío sería diferente para cada carácter, además de ser mucho mayor del actual, resultando en errores seguros en el envío de los bytes originados por desfases temporales.

```
def send (self, comando):
    self.lista_bytes_enviar=[]
    self.pin_unico=Pin(self.Pin_SDI12, mode=Pin.OUT, pull=None)
    for i in range (0, len(comando)):        #Traducción
    #Ahora que están todos los bytes guardados para enviar, los mando uno por uno:
    for x in range (0, len(self.lista_bytes_enviar)):
        byte=self.lista_bytes_enviar[x]
        b=0
        while b < 10:
            self.pin_unico.value(int(byte[b]))    #Envío los bits
            time.sleep_us(800)
            b+=1
    return()
```

Figura 10. Función "send"

Del mismo modo, es necesario tener en cuenta otro detalle. Como se verá posteriormente en el apartado de lectura, cuando hay más de un bit seguido con el mismo estado, no se produce un flanco positivo al comienzo de cada bit, sino que si, por ejemplo, hay tres bits en estado 1, el primero de ellos tendría un flanco (paso de 0 a 1) pero el resto no; la señal se mantiene en positivo durante la duración de los 3 bits. Esto se puede apreciar con claridad en la Figura 11, correspondiente a una captura hecha en un osciloscopio del envío de un comando por parte del microcontrolador (parte izquierda) y de la respuesta del sensor (parte derecha).
En ella se aprecia claramente cómo los diferentes estados tienen distinta duración (cada bit son 833 microsegundos, luego si se tienen 3 bits con el mismo estado seguidos, se verán como un pulso uniforme en el mismo estado, de duración igual al triple de lo que dura un bit normal).

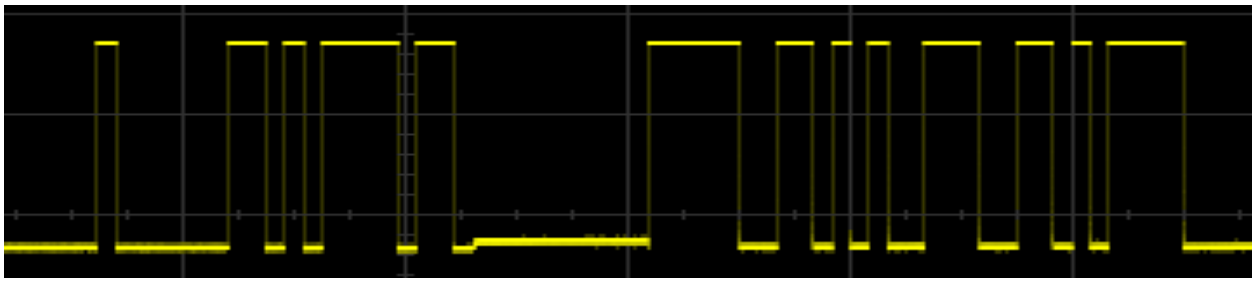

Figura 11. Pulsos enviados y recibidos

Por esto es importante que el valor del pin no se cambie hasta que no nos encontremos un bit en el otro estado. De esta forma nos aseguramos de que no creamos interrupciones donde no debe haberlas.
Después de finalizar el envío de los comandos, se llama inmediatamente a la función de lectura ("read") para la recepción de los pulsos que envía el sensor como respuesta.

6.3. Lectura

La lectura y transformación de los pulsos que envía el sensor constituyen la parte más compleja del proyecto, y la que más tiempo de optimización y depurado ha requerido. En ocasiones se pueden consumir semanas investigando un problema hasta encontrar la solución válida, como se verá después.

Para abordar esta sección de la librería se han planteado diversas soluciones para la lectura y almacenamiento de los pulsos hasta que una ha resultado ser válida. En este apartado se tratarán en su totalidad todas estas versiones, las conclusiones derivadas de ellas, y el motivo por el que han acabado funcionando o no.

6.3.1. Empleo de interrupciones y temporizadores

La primera solución que se planteó consistía en utilizar la detección de interrupciones del microcontrolador para detectar variaciones en el estado del pin. Dentro de esta solución se plantearon diferentes variantes. En un primer lugar, se buscaba que mediante interrupciones activadas con flancos positivos y negativos se guardara el estado del pin, pero como se ha mencionado al final del apartado de envío, es necesario tener en cuenta que no se producen flancos al final y comienzo de cada bit necesariamente, sino que si hay bits seguidos en el mismo estado, el pulso es continuo y con duración equivalente al número de bits iguales. Esto hace imposible esta forma de medida por si sola, y requiere la utilización de temporizadores que midan el tiempo que se mantiene el pulso en ese estado, para poder obtener la cantidad de bits a la que corresponde.

Llevando esto a cabo, se creó un código con el que medir los pulsos formado por varias funciones.

En la función principal se encontraba la llamada a los “callbacks”, las funciones que se activan cuando se detecta un flanco. Dado que el primer flanco que se recibe de un sensor siempre es positivo (bit de inicio del primer byte), en esta función únicamente era necesario introducir la interrupción con flanco positivo, y posteriormente, en el interior de ese callback se situaría la interrupción con flanco negativo.
Con cada inicio de un callback, se guardaba el tiempo y se reiniciaba un temporizador que tenía como objetivo medir la duración de los pulsos. Estos valores se guardarían para su posterior conversión en bytes.

Sin embargo, esto no era funcional. Las pruebas realizadas mostraban que los callbacks no se activaban correctamente, y los tiempos que se obtenían de los temporizadores variaban de formas inexplicables y eran diferentes en cada ocasión. Se intentó solucionar este problema empleando diversos tipos de temporizador para intentar aumentar la precisión, o colocando el segundo callback a continuación del primero en lugar de dentro, pero nada daba resultado.

Después de semanas sin realizar avances, buscando información en los foros de Pycom acerca del funcionamiento específico de las interrupciones y los callbacks en sus dispositivos, se encontró a numerosos usuarios que habían probado que las interrupciones tenían un retraso respecto al cambio de valor del pin. Este retraso era aleatorio, y podía variar desde los 50 hasta los 900 microsegundos en ocasiones, haciendo imposible la medida correcta de tiempos, ya que el retraso de la interrupción podía incluso superar la duración del bit y suponer un error de medida.

La información encontrada en estos artículos (disponibles en la bibliografía) explicaba los resultados obtenidos, y hacía evidente que no sería posible emplear este método para el registro de los pulsos, por lo que se abandonó la idea.

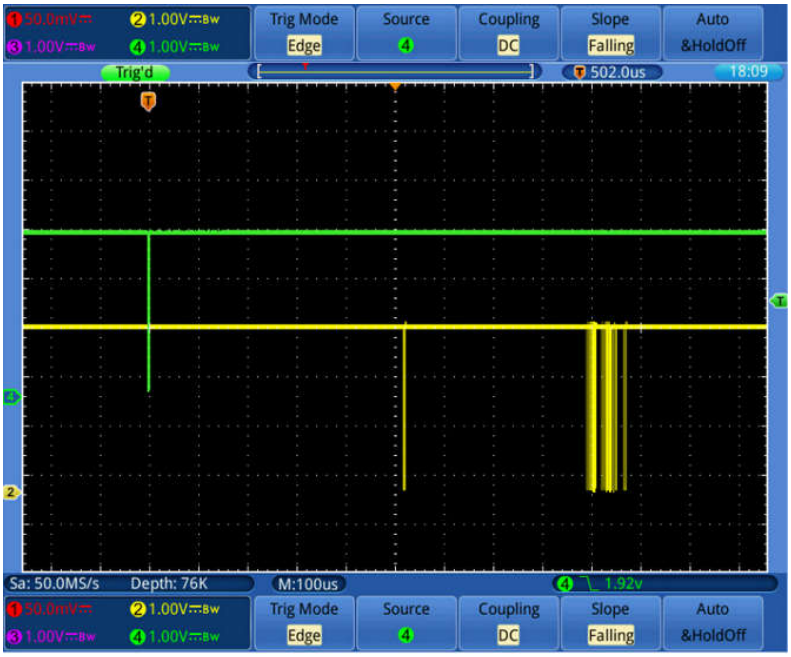

Figura 12. Retraso de la respuesta a una interrupción

(https://forum.pycom.io/topic/936/pin-interrupt-latency)

6.3.2. Función pulses_get()

Después del fracaso de la primera opción, se dedicó un tiempo a la investigación de nuevas posibilidades para afrontar el problema. Esto llevaría al descubrimiento de una función que se había pasado por alto en un primer momento, pero que estaba presente en la documentación que proporciona Pycom: la función pulses_get(). La función parecía perfecta y muy sencilla de usar.
Su funcionamiento consiste en almacenar pulsos guardando el estado el pin y el tiempo que se mantiene en él (justo las dos variables que se necesitan) y proporciona una lista en la que cada entrada tiene dos elementos, el estado del pin, y la duración del estado. La función deja de leer gracias a un "timeout" que es definido por el usuario, y que detiene la lectura si no cambia el estado del pin durante ese tiempo. Para implementar esta función se empleó el código de la Figura 13:

```
def __read(self):

    self.pin_unico=Pin(Pin_SDI12, mode=Pin.IN)

    pulsos=pycom.pulses_get(self.pin_unico, 300)

    lectura=self.convertir_pulsos(pulsos)

    return(lectura)
```

Figura 13. Función de lectura empleando pulses_get()

Un ejemplo de respuesta por parte de la función pulses_get() es el siguiente:

[(1, 4146), (0, 1659), (1, 1656), (0, 835), (1, 827), (0, 830), (1, 2487), (0, 1661), (1, 828), (0, 1663), (1, 829), (0, 1658), (1, 1657), (0, 1659), (1, 1657), (0, 833), (1, 2486), (0, 831), (1, 2487), (0, 828), (1, 828), (0, 834), (1, 828), (0, 830), (1, 828), (0, 830), (1, 2487), (0, 2494), (1, 828), (0, 1661), (1, 3315), (0, 2494), (1, 829), (0, 830), (1, 4146)]

Donde el primer elemento de cada entrada es el estado del pin y el segundo la duración de dicho estado, en microsegundos (aproximadamente múltiplos de 833 microsegundos). Como se puede apreciar, el código es significativamente más sencillo y reducido que empleando interrupciones y callbacks, y la función realiza todo el trabajo por sí sola y de forma consistente.

Después del almacenamiento de los pulsos, el siguiente paso es la conversión de estos a bytes para su posterior traducción. El código de dicha función se encuentra en la Figura 14.

En primer lugar, se crean dos listas, una para los bits y otra para los bytes. Una vez creadas se establecerá un bucle de duración igual al número de entradas de la lista devuelta por la función. Dentro de este bucle, se añade el elemento correspondiente a una variable y se prepara una excepción para el caso de duración 0 del estado con un condicionante "if". Esta excepción es necesaria para el caso donde el pulso final es recogido con duración 0 debido a que es el último y en él no se produce un cambio de estado, sino una finalización de la lectura por parte del timeout. Por tanto, y dado que el último byte de respuesta del sensor es siempre un lane feed o /n y sus últimos bits son 1 y 0, que nos encontremos con un tiempo de duración de byte 0 siempre supondrá la detección de este último bit, y se deberá identificar como una duración de un bit normal.

```
def convertir_pulsos(self, pulses):
    resp=0
    self.lista_bits=[]
    self.lista_bytes=[]
    for i in range (0, len(pulses)):
        pulso=pulses[i]
        #num_bits=0
        if pulso[1]==0:
            num_bits=1
        else:
            num_bits=int(pulso[1]/800)
        #print(num_bits)
        for x in range (0, num_bits):
            self.lista_bits.append(pulso[0])
        if len(self.lista_bits)==10:
            self.lista_bytes.append(self.lista_bits)
            #print(self.lista_bytes)
            self.lista_bits=[]
    resp=self.traducir_binario_a_caracteres()
    return(resp)
```

Figura 14. Traducción de pulsos a bytes.

A continuación, se establece la forma normal para calcular el número de bits que se mantienen en un mismo estado, mediante una división de la que nos quedamos con el número entero.

Tras haber calculado el número de bits que están en el mismo estado para esa entrada, se guardan en la lista de bits.

Finalmente, se comprueba si el byte está completo, y si es así, se añade a la lista de bytes y se continúan calculando los siguientes bits hasta que toda la lista de la función esté convertida.

Una vez realizada esta conversión, se llama a la función encargada de traducir los bytes a caracteres. No es necesario aportar como argumento de la función el listado de bytes, ya que la función puede acceder a él directamente al estar definido como un atributo de clase.

```
def convertir_cadena(self, cadena):

    s = [str(i) for i in cadena]

    res = "".join(s)

    return(res)

def traducir_binario_a_caracteres(self):
    respuesta=[]
    y=0
    while y < len(self.lista_bytes):
        subcadena=self.lista_bytes[y]
        datos=subcadena[1:8]
        #tengo que cambiar los datos de formato int a str
        #para que coincida con el de los diccionarios
        d=0
        for d in range (0,7):
            datos[d]=str(datos[d])
            d+=1
        x=0
        while x < 128:
            if datos==list(diccionario_usascii_invertido[x]):
                respuesta.append(diccionario_usascii_caracteres[x])
                break
            else:
                x+=1
        y+=1
    return(self.convertir_cadena(respuesta))
```

Figura 15. Traducción de binario a caracteres.

La función de traducción siempre es la misma en todas las soluciones planteadas, debido a que siempre se acaban convirtiendo los pulsos al mismo formato de bytes, de una forma u otra.
Cuando se ha conseguido este formato, y repitiendo el procedimiento para cada elemento en la lista de bytes, la librería lleva a cabo los siguientes pasos:
- Guardar un byte en una variable, y posteriormente los 7 bits de datos de ese byte en otra, que son los que nos interesan para la traducción.
- Cambiar el formato de los datos a str (string, cadena, lista) para que se puedan comparar con los diccionarios.
- Comparar los datos con los diccionarios hasta hallar los bits correspondientes, y extraer el carácter buscado del otro diccionario, añadiéndolo a la respuesta.

Una vez realizado este proceso con todos los bytes, los caracteres obtenidos se juntan en una lista sin espacios ni comas mediante la función convertir_cadena() que es la que generará la respuesta final que se presentará.

Cuando la librería estuvo terminada, se realizaron pruebas en diversos sensores para comprobar su correcto funcionamiento, y se descubrió que en algunos casos no se mostraban todos los datos que debían. Después de comprobar que los sensores estaban enviando todos los datos con ayuda del osciloscopio, se comenzó a investigar otras posibles fuentes de error en el código que estuvieran dando lugar a este fallo.

Finalmente, y después de varios días de búsqueda de información, gracias a la ayuda de varios usuarios de los foros de Pycom se confirmó que el problema era de la función pulses_get(), que al estar diseñada para la lectura de pulsos de infrarrojos, poseía un límite de 128 entradas, lo que provocaba que no fuera posible leer completamente los sensores cuando estos enviaban mucha información.
En un principio se pensó que la función podía leer un máximo de 200 pulsos, equivalentes a 20 bytes o caracteres, pero este número variaba según el sensor o el comando, y finalmente se observó que el máximo se establecía en 128 entradas.

Este error se hace evidente al enviar el comando de identificación (aI!) a un sensor MPS-6. Los pulsos leídos por la función pulses_get() son los siguientes:

[(1, 4146), (0, 1659), (1, 1656), (0, 835), (1, 827), (0, 830), (1, 2487), (0, 1661), (1, 828), (0, 1663), (1, 829), (0, 1658), (1, 1657), (0, 1659), (1, 1657), (0, 833), (1, 2486), (0, 831), (1, 2487), (0, 828), (1, 828), (0, 834), (1, 828), (0, 830), (1, 828), (0, 830), (1, 2487), (0, 2494), (1, 828), (0, 1661), (1,

3315), (0, 2494), (1, 829), (0, 830), (1, 4146), (0, 830), (1, 828), (0, 834), (1, 828), (0, 2490), (1, 2488), (0, 830), (1, 829), (0, 833), (1, 830), (0, 3318), (1, 1658), (0, 2492), (1, 1658), (0, 2489), (1, 1657), (0, 830), (1, 828), (0, 834), (1, 4974), (0, 831), (1, 829), (0, 1662), (1, 829), (0, 829), (1, 828), (0, 1660), (1, 1657), (0, 830), (1, 828), (0, 835), (1, 4146), (0, 829), (1, 829), (0, 831), (1, 828), (0, 834), (1, 828), (0, 1659), (1, 1657), (0, 830), (1, 829), (0, 830), (1, 828), (0, 834), (1, 828), (0, 830), (1, 828), (0, 1659), (1, 829), (0, 830), (1, 1657), (0, 834), (1, 1657), (0, 1659), (1, 828), (0, 1659), (1, 1657), (0, 834), (1, 4975), (0, 829), (1, 829), (0, 1663), (1, 828), (0, 1659), (1, 1658), (0, 1658), (1, 1658), (0,834), (1, 3316), (0, 2489), (1, 829), (0, 1663), (1, 1657), (0, 1659), (1, 829), (0, 1658), (1, 1658), (0, 834), (1, 3317), (0, 2489), (1, 828), (0, 1663), (1, 3317), (0, 2489), (1, 829), (0, 1663), (1, 830), (0, 829), (1, 1658), (0, 2489), (1, 1656), (0, 834)]

Únicamente con estos pulsos podríamos comprobar que hay un error de lectura, ya que el último estado no tiene una duración de 0 microsegundos, pero traduciendo estos pulsos a caracteres se obtiene la siguiente respuesta:

013DECAGON MPS-6 386889

Dado que esta respuesta no termina en /r/n, podemos confirmar que no está completa, y por tanto, la función posee limitación en su medida.

La función pulses_get() no se podía modificar para aumentar su límite de entradas, por lo que al final se decidió abandonar esta idea y plantear una nueva solución que no requiriera de su utilización.

6.3.3. Función con temporizadores

Tras encontrar las limitaciones de la función pulses_get(), se intentó volver a plantear una solución nueva, basada en leer el estado del pin cada 833 microsegundos y guardarlo en una lista.
Para ello se emplearían temporizadores o "timers". La primera iteración de la solución fue la que se encuentra en la Figura 16.

Al igual que anteriormente, tras definir el pin en modo lectura, se crean dos listas en blanco para los bits y los bytes a leer. Para poder saber en qué momento se debía empezar a leer, se establece un bucle infinito al inicio que termina en el momento en el que el pin se encuentre en estado positivo, es decir, cuando se reciba el primer bit de inicio y se empiece a recibir información del sensor.

Tras detectar este primer bit de inicio, se añade a la lista de bits y se esperan 833 microsegundos para pasar al segundo. En este momento, se entra en otro bucle infinito en el que se añadirá el nuevo estado a la lista de bits, se comprobará si la lista de bits está llena, y en caso afirmativo, si se trata o no de un byte vacío que conllevaría la detención de la lectura o si simplemente hay que añadir la lista de bits a los bytes. Después, si no se ha terminado de leer, se espera un tiempo para llegar al siguiente bit y volver a medir otra vez.
Este tiempo está ajustado en función de la duración del código anterior, de forma que la siguiente medida que se realice sea en el siguiente bit.

```
def __read(self):
    self.pin_unico=Pin(Pin_SDI12, mode=Pin.IN, pull=Pin.PULL_DOWN)
    self.lista_bits=[]
    self.lista_bytes=[]
    while True:
        if self.pin_unico()==1:
            break
    self.lista_bits.append(1)
    time.sleep_us(833)

    while True:
        self.lista_bits.append(self.pin_unico())

        if len(self.lista_bits)==10:
            if self.lista_bits==[0,0,0,0,0,0,0,0,0,0]:
                #self.lista_bytes.append(self.lista_bits)
                self.lista_bits=[]
                break
            else:
                self.lista_bytes.append(self.lista_bits)
                self.lista_bits=[]
        time.sleep_us(777)

    return(self.traducir_binario_a_caracteres())
```

Figura 16. Lectura de pulsos con timers

Sin embargo, el control de los tiempos era poco consistente con este método, quedando demostrado cuando al ejecutar el programa 6-7 veces seguidas se empezaban a corromper las medidas.
Se puede emplear como ejemplo el mismo que se usó anteriormente. Al enviar un comando de identificación a un sensor MPS-6, la respuesta correcta del sensor, y que se lee de forma adecuada las primeras 6 veces, es la siguiente:

013DECAGON MPS-6 386889208069\r\n

Al ejecutar el programa la séptima vez seguida, se obtiene:

013DECAGON MPS-6 38688920807FSFENQ

La octava:

013DECAGON MPS-6 3868[FSYCAN[CANESCFSFENQ

Y la novena:

013DECAGONP&()SYNESCPEM[ESC[[NlLnLMNcB

Como se puede ver, el problema va aumentando de forma sucesiva cuando se va ejecutando el código. Cuantas más veces seguidas se haga, se van leyendo de forma incorrecta más bytes progresivamente hasta que al final toda la lectura carece de sentido.

Haciendo pruebas se pudo comprobar que el problema no sucedía si se reiniciaba el microcontrolador manualmente después de cada lectura.

El error estaba probablemente debido a que el añadir elementos a una lista empleando el método "append" de forma continuada podría ir creando un retraso cada vez mayor que al final acabaría con desincronizar todo el código, provocando errores de lectura que aumentan progresivamente.

Por tanto, el primer arreglo consistió en asignar los bytes antes de la lectura, de forma que el tiempo no fuera aumentando progresivamente y no se creara esta desincronización.

Sin embargo, esto no resultaba ser suficiente, ya que los tiempos empleados seguían provocando al final desincronizaciones en la lectura y otros errores.
Para arreglar el problema de forma definitiva, sería necesario comenzar a leer siempre en el centro de la duración de un bit y reiniciar este centrado cada byte para asegurar que no fuera posible sufrir desincronizaciones.

En la Figura 17 se encuentra el código final empleado en la lectura de los pulsos. A continuación se explicarán los cambios aplicados respecto al código anterior, así como los detalles que hizo falta tener en cuenta para el correcto funcionamiento de este.

En primer lugar, como se ha comentado anteriormente, es necesario crear un byte en blanco, así como una lista de bytes con una longitud determinada, de forma que la acción de añadir un valor a estas listas siempre consuma el mismo tiempo.

Otro cambio, y posiblemente el más importante, es la incorporación del bucle para detectar el primer bit al interior del bucle que se repite cada byte. De esta forma, podemos asegurarnos de que los tiempos que tomamos se están reiniciando cada byte.
Después de la detección de este primer bit, se espera la duración equivalente a un bit y medio (más o menos) para comenzar a leer los siguientes bits. En el código se añaden tres temporizadores diferentes debido a que actualmente existe un error en los temporizadores mayores de 1000 microsegundos, así que, para solucionarlo, se emplean tres más pequeños.

Tras la espera, se añade este primer bit detectado y se entra en un bucle for en el que se van rellenando los diferentes espacios de la lista de bits, estableciendo un temporizador de duración igual a la de un bit menos el tiempo consumido por guardar los valores.
Posteriormente se lee el bit de paridad, se añade el bit final, y se guarda el byte en la lista de bytes. En caso de que se detecte que el byte guardado corresponde a "/n", se dejaría de leer, ya que el sensor ha terminado de enviar pulsos.
Al final, se llama a la función de traducción (Figura 15) y se obtienen los caracteres buscados.

Con diferencia, el factor más importante a tener en cuenta es el de los tiempos. Si no se hace que encajen correctamente, pueden ocurrir errores o provocar que no se lean bytes enteros, lo que daría lugar a un bucle infinito ya que no se detectaría correctamente el byte final, y por tanto no se acabaría nunca de leer.

```
def read(self):
    self.pin_unico=Pin(self.Pin_SDI12, mode=Pin.IN, pull=Pin.PULL_DOWN)
    self.lista_bits=[0,0,0,0,0,0,0,0,0,0]
    self.lista_bytes=[None]*50
    y=0
    while True:
        #debug_pin=Pin('P5', mode=Pin.OUT)
        #debug_pin(1)
        #debug_pin(0)
        while True:
            #print('esperando')
            if self.pin_unico()==1:
                break
        time.sleep_us(390)
        time.sleep_us(390)
        time.sleep_us(390)
        self.lista_bits[0]=1

        for x in range (0, 7):
        #    debug_pin(1)
            self.lista_bits[x+1] = self.pin_unico()

        #    debug_pin(0)
            time.sleep_us(814)

        self.lista_bits[8] = self.pin_unico()
        self.lista_bits[9]=0
        #print(self.lista_bits)
        time.sleep_us(450)
        self.lista_bytes[y]=self.lista_bits[:]
        y+=1
        #print(self.lista_bytes)
        if self.lista_bits==[1,1,0,1,0,1,1,1,1,0]:
            break

    return(self.traducir_binario_a_caracteres())
```

Figura 17. Función de lectura definitiva.

En un principio, durante la modificación del código se intentó emplear impresiones de texto ("prints") para poder comprobar qué partes de código se estaban ejecutando y cómo. Se descubrió así que el código se quedaba atascado en el primer "while" en el que se lee el bit de inicio durante la lectura del segundo byte.

Sin embargo, la información que se podía obtener de ellos era bastante limitada y no permitía saber por qué estaba ocurriendo esto, por lo que se decidió emplear un segundo pin de salida como debugger, de forma que se activara y desactivara cada vez que se medían los pulsos y justo antes de comenzar el bucle de comprobación del bit inicial. Gracias a esto se pudo ver que en realidad lo que estaba ocurriendo es que las impresiones (que se encuentran comentadas en el código) estaban provocando un retraso enorme, que duraba más que los dos bytes siguientes, de forma que el while se iniciaba cuando el sensor había terminado de enviar comandos y nunca llegaría un estado positivo que activara el break.

En la Figura 18 se puede apreciar cómo el debugger permite comprobar de forma exacta el momento en el que se está midiendo. Los pulsos amarillos corresponden al sensor, mientras que los verdes son los realizados por el debugger.

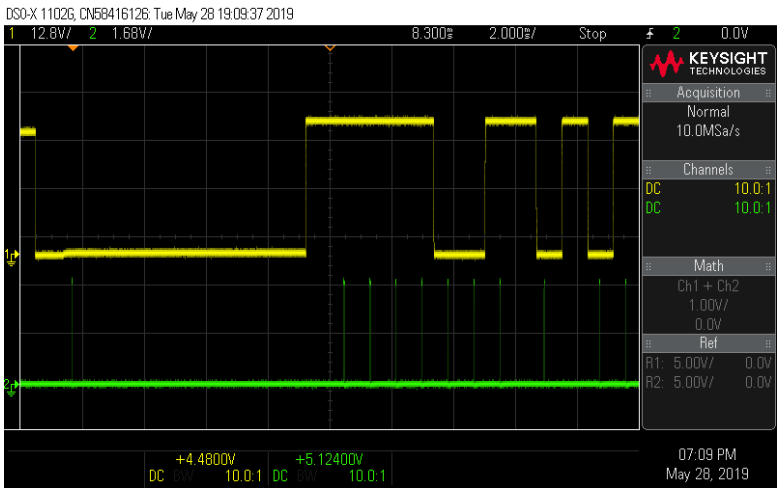

Figura 18. Captura de osciloscopio con debugger.

Se puede observar cómo el primer pulso verde se encuentra justo después de terminar el envío del comando por parte del microchip, y se corresponde con el inicio del bucle infinito del que se sale al detectar el primer pulso positivo.

Como también se puede ver, el siguiente pulso verde se encuentra en mitad del segundo bit, igual que los 6 siguientes, que se encuentran perfectamente centrados. Después de esos 7 bits seguidos, que se corresponden a los bits de datos medidos dentro del bucle for, se mide el bit de paridad y se añade el bit final. Sin embargo, el siguiente while que detecta el comienzo del siguiente byte no se activa de forma inmediata, ya que si lo hiciera, se detectaría el estado positivo del bit de paridad, y se comenzaría a leer de forma errónea.
Es fundamental para el correcto funcionamiento del código que el bucle que comprueba el primer bit se inicie durante el bit final de un byte, que siempre es negativo, o de lo contrario, se pueden provocar errores en la medida.
Debido a esto se añade al código de la Figura 17 un temporizador al final de 450 microsegundos, suficientes para retrasar el inicio del bucle hasta este bit final.

Todo este proceso se puede apreciar con mayor claridad en las Figuras 19 y 20, donde se han resaltado en rojo los pulsos que marcan el comienzo del bucle while de inicio de cada byte. Como se ha mencionado anteriormente, este bucle se inicia siempre en el bit final de un byte, que es siempre negativo.

Del mismo modo, reiniciar la temporización cada byte permite asegurar que no se van a crear retrasos ni errores de medida, sin importar la cantidad de bytes seguidos que se lean o la de veces que se ejecute el programa sin reiniciar manualmente el microchip.

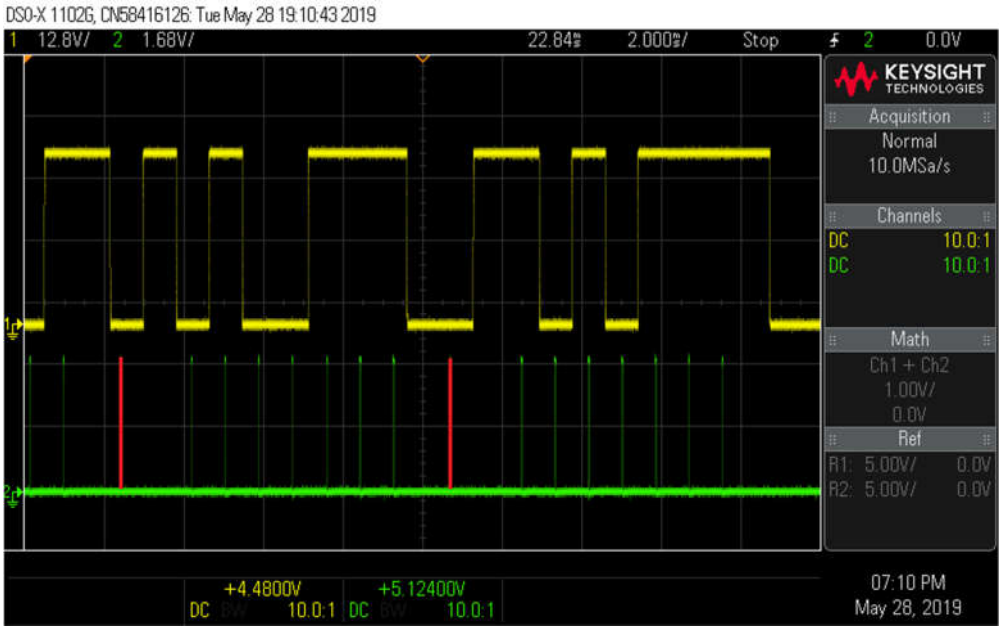

Figura 19. Pulsos anteriores al while inicial (I)

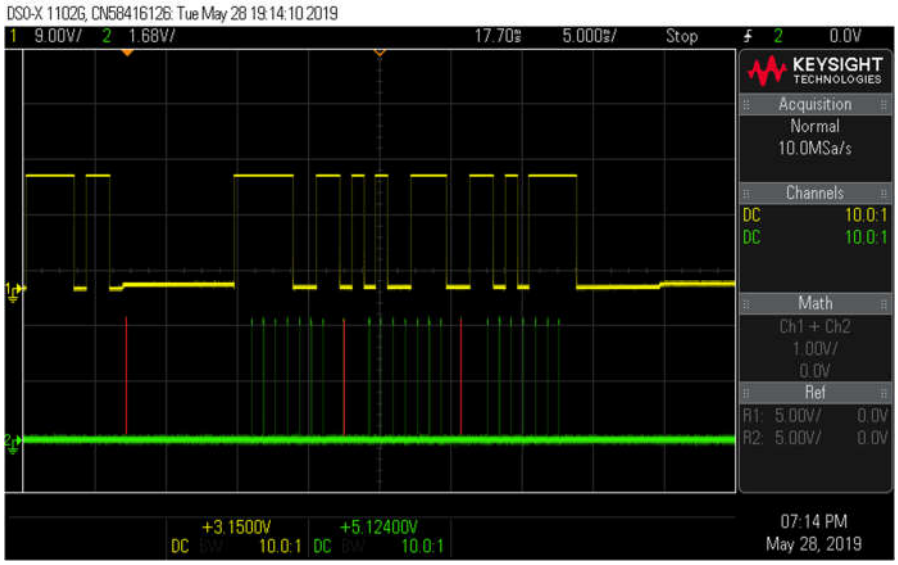

Figura 20. Pulsos anteriores al while inicial (II)

7. Utilización de la librería

La utilización de las funciones que forman la librería es sencilla. En primer lugar, después de importar la clase desde la librería, será necesario definir un objeto que pertenezca a la clase creada. En este caso, la clase se llama "UART_Guille".

```
from UART_Guille import UART_Guille

sensor=UART_Guille()

sensor.__break()

sensor.send(['1','I','!'])

lectura=sensor.read()

print(lectura)
```

Figura 21. Ejemplo de funcionamiento.

Después de definir el objeto, hay que despertar a los sensores con un break para poder enviarles comandos. Después del break se llama a la función send, que como argumento tiene el comando que queremos enviar.
Este comando se ha de proporcionar como una lista con formato string y con la separación con comas, tal y como aparece en la Figura 21.

Finalmente, se llama a la función de lectura y se guardan los datos que devuelve en una variable para su posterior impresión o almacenamiento.

8. Conclusiones

De este proyecto se pueden extraer una serie de conclusiones en base a los resultados obtenidos.

En primer lugar, los resultados confirman que es posible crear una UART completamente funcional para la comunicación entre dispositivos con protocolo SDI-12 en un SoC WiPy 3.0 empleando MicroPython. Hacer que al principio de cada byte se vuelva a reiniciar el sitio de medida de los pulsos garantiza la capacidad de lectura de cualquier cantidad de bytes.

Tal como se explicó al principio de este trabajo, el uso de una UART que funciona con un único pin GPIO permite un ahorro sustancial en hardware externo, así como en tiempo de fabricación y dinero, debido a que no necesita ninguna clase de elemento externo adicional para realizar su función.

La UART creada es, a su vez, una solución más simple que la UART tradicional con dos pines (Rx y Tx) y hace posible la creación de varios buses de datos SDI-12 con un único SoC de manera sencilla, asignando un pin a cada uno de ellos. Dentro de cada uno de estos buses, los sensores tendrán cada uno su dirección, lo que hace posible que sensores con las mismas direcciones funcionen en el mismo SoC al comunicarse a través de pines diferentes.

Finalmente, el descubrimiento de fallos en métodos que en un principio se pensaba que podrían funcionar, así como los ensayos de prueba y error hasta dar con la solución correcta, han permitido lograr un mayor entendimiento del funcionamiento de una UART y de los distintos elementos y funciones propias del SoC empleado.

9. Apéndice

En este apéndice se recoge la totalidad del código de la librería, incluyendo los diccionarios de caracteres y todas las funciones de las que se compone:

```
import time
import pycom
import machine
from machine import Pin, Timer
from Gestion_fuentes import encender_5Vcc, encender_12Vcc, apagar_5Vcc, apaga
r_12Vcc
diccionario_usascii_invertido={
    0:'1111111',    #NUL
    1:'0111111',    #SOH
    2:'1011111',    #STX
    3:'0011111',    #ETX
    4:'1101111',    #EOT
    5:'0101111',    #ENQ
    6:'1001111',    #ACK
    7:'0001111',    #BEL
    8:'1110111',    #BS
    9:'0110111',    #HT
    10:'1010111',   #LF
    11:'0010111',   #VT
    12:'1100111',   #FF
    13:'0100111',   #CR
    14:'1000111',   #SO
    15:'0000111',   #SI
    16:'1111011',   #DLE
    17:'0111011',   #DC1
    18:'1011011',   #DC2
    19:'0011011',   #DC3
    20:'1101011',   #DC4
    21:'0101011',   #NAK
    22:'1001011',   #SYN
    23:'0001011',   #ETB
    24:'1110011',   #CAN
    25:'0110011',   #EM
    26:'1010011',   #SUB
    27:'0010011',   #ESC
    28:'1100011',   #FS
    29:'0100011',   #GS
    30:'1000011',   #RS
    31:'0000011',   #US
    32:'1111101',   #SPACE
    33:'0111101',   #!
    34:'1011101',   #"
    35:'0011101',   ##
    36:'1101101',   #$
    37:'0101101',   #%
    38:'1001101',   #&
    39:'0001101',   #´
    40:'1110101',   #(
    41:'0110101',   #)
    42:'1010101',   #*
    43:'0010101',   #+
    44:'1100101',   #,
```

```
45:'0100101',  #-
46:'1000101',  #.
47:'0000101',  #/
48:'1111001',  #0
49:'0111001',  #1
50:'1011001',  #2
51:'0011001',  #3
52:'1101001',  #4
53:'0101001',  #5
54:'1001001',  #6
55:'0001001',  #7
56:'1110001',  #8
57:'0110001',  #9
58:'1010001',  #:
59:'0010001',  #;
60:'1100001',  #<
61:'0100001',  #=
62:'1000001',  #>
63:'0000001',  #?
64:'1111110',  #@
65:'0111110',  #A
66:'1011110',  #B
67:'0011110',  #C
68:'1101110',  #D
69:'0101110',  #E
70:'1001110',  #F
71:'0001110',  #G
72:'1110110',  #H
73:'0110110',  #I
74:'1010110',  #J
75:'0010110',  #K
76:'1100110',  #L
77:'0100110',  #M
78:'1000110',  #N
79:'0000110',  #O
80:'1111010',  #P
81:'0111010',  #Q
82:'1011010',  #R
83:'0011010',  #S
84:'1101010',  #T
85:'0101010',  #U
86:'1001010',  #V
87:'0001010',  #W
88:'1110010',  #X
89:'0110010',  #Y
90:'1010010',  #Z
91:'0010010',  #[
92:'1100010',  #\
93:'0100010',  #]
94:'1000010',  #^
95:'0000010',  #_
96:'1111100',  #`
97:'0111100',  #a
98:'1011100',  #b
99:'0011100',  #c
100:'1101100', #d
101:'0101100', #e
102:'1001100', #f
103:'0001100', #g
104:'1110100', #h
105:'0110100', #i
```

```
    106:'1010100', #j
    107:'0010100', #k
    108:'1100100', #l
    109:'0100100', #m
    110:'1000100', #n
    111:'0000100', #o
    112:'1111000', #p
    113:'0111000', #q
    114:'1011000', #r
    115:'0011000', #s
    116:'1101000', #t
    117:'0101000', #u
    118:'1001000', #v
    119:'0001000', #w
    120:'1110000', #x
    121:'0110000', #y
    122:'1010000', #z
    123:'0010000', #{
    124:'1100000', #|
    125:'0100000', #}
    126:'1000000', #~
    127:'0000000', #DEL

    }

diccionario_usascii_caracteres={
0:'NUL',    #NUL.
1:'SOH',    #SOH
2:'STX',    #STX
3:'ETX',    #ETX
4:'EOT',    #EOT
5:'ENQ',    #ENQ
6:'ACK',    #ACK
7:'BEL',    #BEL
8:'BS',    #BS
9:'HT',    #HT
10:'\\n',   #LF
11:'VT',   #VT
12:'FF',   #FF
13:'\\r',   #CR
14:'SO',   #SO
15:'SI',   #SI
16:'DLE',   #DLE
17:'DC1',   #DC1
18:'DC2',   #DC2
19:'DC3',   #DC3
20:'DC4',   #DC4
21:'NAK',   #NAK
22:'SYN',   #SYN
23:'ETB',   #ETB
24:'CAN',   #CAN
25:'EM',   #EM
26:'SUB',   #SUB
27:'ESC',   #ESC
28:'FS',   #FS
29:'GS',   #GS
30:'RS',   #RS
31:'US',   #US
32:' ',   #SPACE
33:'!',   #!
34:'"',   #"
```

```
35:'#',  ##
36:'$',  #$
37:'%',  #%
38:'&',  #&
39:'´',  #´
40:'(',  #(
41:')',  #)
42:'*',  #*
43:'+',  #+
44:',',  #,
45:'-',  #-
46:'.',  #.
47:'/',  #/
48:'0',  #0
49:'1',  #1
50:'2',  #2
51:'3',  #3
52:'4',  #4
53:'5',  #5
54:'6',  #6
55:'7',  #7
56:'8',  #8
57:'9',  #9
58:':',  #:
59:';',  #;
60:'<',  #<
61:'=',  #=
62:'>',  #>
63:'?',  #?
64:'@',  #@
65:'A',  #A
66:'B',  #B
67:'C',  #C
68:'D',  #D
69:'E',  #E
70:'F',  #F
71:'G',  #G
72:'H',  #H
73:'I',  #I
74:'J',  #J
75:'K',  #K
76:'L',  #L
77:'M',  #M
78:'N',  #N
79:'O',  #O
80:'P',  #P
81:'Q',  #Q
82:'R',  #R
83:'S',  #S
84:'T',  #T
85:'U',  #U
86:'V',  #V
87:'W',  #W
88:'X',  #X
89:'Y',  #Y
90:'Z',  #Z
91:'[',  #[
92:'[', #
93:']',  #]
94:'^',  #^
95:'_',  #_
```

```
96:'`',  #`
97:'a',  #a
98:'b',  #b
99:'c',  #c
100:'d', #d
101:'e', #e
102:'f', #f
103:'g', #g
104:'h', #h
105:'i', #i
106:'j', #j
107:'k', #k
108:'l', #l
109:'m', #m
110:'n', #n
111:'o', #o
112:'p', #p
113:'q', #q
114:'r', #r
115:'s', #s
116:'t', #t
117:'u', #u
118:'v', #v
119:'w', #w
120:'x', #x
121:'y', #y
122:'z', #z
123:'{', #{
124:'|', #|
125:'}', #}
126:'~', #~
127:'DEL', #DEL
}
```

```
class UART_Guille:


    def __init__(self, pin):
        self.Pin_SDI12=pin
        self.pin_unico=Pin(self.Pin_SDI12, mode=Pin.OPEN_DRAIN)
        self.lista_bytes=[]
        self.lista_bits=[]
        self.cuenta=0
        self.lista_bytes_enviar=[]

    def __break(self):

        self.pin_unico=Pin(self.Pin_SDI12, mode=Pin.OUT, pull=None)


        self.pin_unico.value(1)
        time.sleep_ms(20)

        self.pin_unico.value(0)
        time.sleep_ms(20)

        self.pin_unico=Pin(self.Pin_SDI12, mode=Pin.OPEN_DRAIN)
```

```
def convertir_cadena(self, cadena):

    s = [str(i) for i in cadena]

    res = "".join(s)

    return(res)

def traducir_binario_a_caracteres(self):
    respuesta=[]
    y=0
    while y < len(self.lista_bytes):
        if self.lista_bytes[y]==None:
            break
        subcadena=self.lista_bytes[y]
        datos=subcadena[1:8]
        d=0
        for d in range (0,7):
            datos[d]=str(datos[d])
            d+=1
        x=0
        while x < 128:

            if datos==list(diccionario_usascii_invertido[x]):

                respuesta.append(diccionario_usascii_caracteres[x])
                break
            else:
                x+=1
        y+=1

    return(self.convertir_cadena(respuesta))

def read(self):
    self.pin_unico=Pin(self.Pin_SDI12, mode=Pin.IN, pull=Pin.PULL_DOWN)
    self.lista_bits=[0,0,0,0,0,0,0,0,0,0]
    self.lista_bytes=[None]*50
    y=0
    while True:

        while True:

            if self.pin_unico()==1:
                break
        time.sleep_us(390)
        time.sleep_us(390)
        time.sleep_us(390)
        self.lista_bits[0]=1

        for x in range (0, 7):

            self.lista_bits[x+1] = self.pin_unico()


            time.sleep_us(814)

        self.lista_bits[8] = self.pin_unico()
        self.lista_bits[9]=0

        time.sleep_us(450)
```

```
            self.lista_bytes[y]=self.lista_bits[:]
            y+=1

            if self.lista_bits==[1,1,0,1,0,1,1,1,1,0]:
                break

        return(self.traducir_binario_a_caracteres())


    def send (self, comando):

        self.lista_bytes_enviar=[]

        self.pin_unico=Pin(self.Pin_SDI12, mode=Pin.OUT, pull=None)
        for i in range (0, len(comando)):
            byte_enviar=list('1000000000')
            caracter=comando[i]
            caracter_binario=[]
            y=0
            while y < 128:
                if caracter==diccionario_usascii_caracteres[y]:
                    caracter_binario=diccionario_usascii_invertido[y]
                    break
                else:
                    y+=1
            cadena_binario=list(caracter_binario)

            z=0
            while z < len(cadena_binario):
                    byte_enviar[z+1]=cadena_binario[z]
                    z+=1


            if ((byte_enviar.count('1')-1) % 2) == 0:
                byte_enviar[8]=0
            else:
                byte_enviar[8]=1
            byte_enviar[9]=0

            self.lista_bytes_enviar.append(byte_enviar)
            y+=1

        for x in range (0, len(self.lista_bytes_enviar)):
            byte=self.lista_bytes_enviar[x]
            b=0
            while b < 10:
                self.pin_unico.value(int(byte[b]))
                time.sleep_us(800)
                b+=1
        self.pin_unico=Pin(self.Pin_SDI12, mode=Pin.IN, pull=Pin.PULL_DOWN)
        return()
```

10. Bibliografía

- Wikipedia, *Universal Asynchronous Receiver-Transmitter.* (Extraído el 27 de junio de 2019 de https://es.wikipedia.org/wiki/Universal_Asynchronous_Receiver-Transmitter)

- Metergroup, *SDI-12: Everything you need to know to be successful.* (Extraído el 27 de junio de 2019 de https://www.metergroup.com/environment/articles/sdi-12-everything-need-know-successful/)

- SDI-12 Support group, *SDI-12 Specification*. (Disponible en http://www.sdi-12.org/specification.php)

- Pycom Forum, *Pin interrupt latency, Time interrupts.* (Disponible en: https://forum.pycom.io/topic/936/pin-interrupt-latency y https://forum.pycom.io/topic/499/time-interrupts)

- Pycom, *WiPy 3.0.* (Disponible en: https://pycom.io/product/wipy-3-0/)