

ปฏิบัติการบน RaspberryPi ครั้งที่ 7:

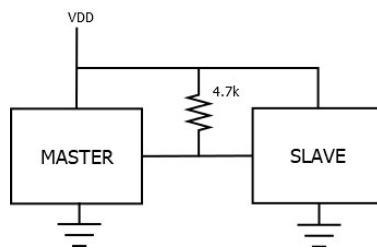
การเชื่อมต่อกับอุปกรณ์ผ่าน 1-wire

หลักการพื้นฐานของบัส 1-wire

บัส 1-wire แรกเริ่มนั้นได้รับการออกแบบพัฒนาโดย Dallas Semiconductor Corp ซึ่งออกแบบมาให้มีลักษณะเป็นการติดต่อสื่อสารแบบอนุกรมที่ความเร็วต่ำ และใช้สายสัญญาณเพียงเส้นเดียว โดยวงจรขับของอุปกรณ์ที่เชื่อมต่อจะมีลักษณะเป็น open collector ที่เมื่อจะส่งสัญญาณก็จะดึงสัญญาณลงกราวด์ ดังนั้นการเชื่อมต่อแบบ 1-wire จึงต้องการตัวต้านทานที่จ่ายกระแสจากไฟเลี้ยงของวงจรด้าน master ค่าตัวต้านทานส่วนมากมีขนาดประมาณ 4.7k (ค่าที่ใช้จริงอาจคลาดเคลื่อนได้บ้าง)

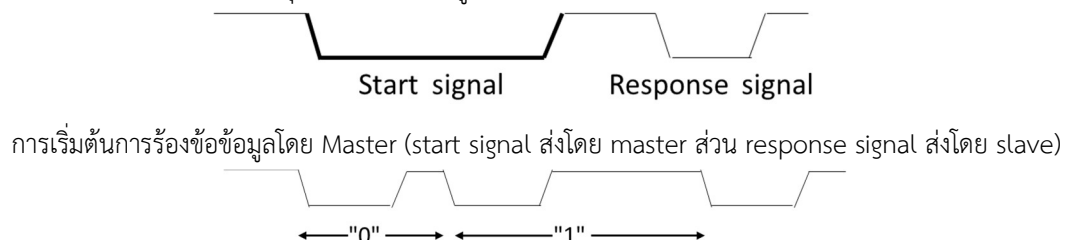
วงจรส่วนทาง slave นั้น โดยปกติอาจจะใช้ไฟเลี้ยงจากแหล่งภายนอก หรือในกรณีที่วงจรกินพลังงานน้อยมากๆ ภายในวงจร slave อาจจะมีตัวเก็บประจุเพื่อใช้ดึงกระแสไฟจากขาสัญญาณมาเก็บไว้เพื่อเลี้ยงการทำงานของวงจรก็ได้ ซึ่งทำให้การเชื่อมต่อสัญญาณระหว่างวงจร master กับ slave เหลือเพียงเส้นเดียวเท่านั้น (ไม่นับกราวด์ที่ต้องเชื่อมเข้าด้วยกัน)

อุปกรณ์ที่ใช้การรับส่งสัญญาณแบบ 1-wire มักจะเป็นวงจรที่ไม่ได้มีความซับซ้อนในการจัดการเท่าใดนัก ตัวอย่างอุปกรณ์ที่ใช้การเชื่อมต่อแบบ 1-wire ได้แก่ เซ็นเซอร์อุณหภูมิ ความชื้น เป็นต้น



ลักษณะการเชื่อมต่อวงจรแบบ 1-wire

การส่งข้อมูลจะเริ่มต้นจากอุปกรณ์ master ส่งสัญญาณร้องขอข้อมูลในลักษณะของพัลส์ 0 ที่มีช่วงเวลาตามที่กำหนด(ตามสเป็คของ slave) จากนั้นอุปกรณ์ slave จะส่งสัญญาณตอบรับกลับมาด้วยพัลส์ 0 ที่มีช่วงเวลาตามที่กำหนด แล้วจึงตามด้วยบิตของข้อมูลตามจำนวน ไปจนกระทั่งหมดชุดบิตข้อมูล ก็จะตามด้วยบิตปิดท้าย เป็นอันสิ้นสุดกลไกการส่งข้อมูล



รูปแบบการส่งบิต 0 และ 1 จะอาศัยการกำหนดช่วงเวลาของสัญญาณ Hi ที่มีความยาวแตกต่างกัน

การจัดการบัส 1-wire บน Raspberry Pi ผ่านทาง Raspberry Pi OS

สำหรับระบบปฏิบัติการ Raspberry Pi OS นั้น มีไดรเวอร์ที่ใช้อ่านข้อมูลผ่านบัส 1-wire มาให้แล้ว และรองรับเซ็นเซอร์จำนวนหนึ่ง โดยตัวระบบปฏิบัติการนั้นจะจัดการอุปกรณ์ที่เชื่อมต่อในลักษณะของไฟล์ในหน่วยความจำสำรอง (ตามมาตรฐาน POSIX) ดังนั้นหากเราใช้อุปกรณ์ที่รองรับโดยตัวระบบปฏิบัติการ การเชื่อมต่อก็จะกระทำได้ง่าย

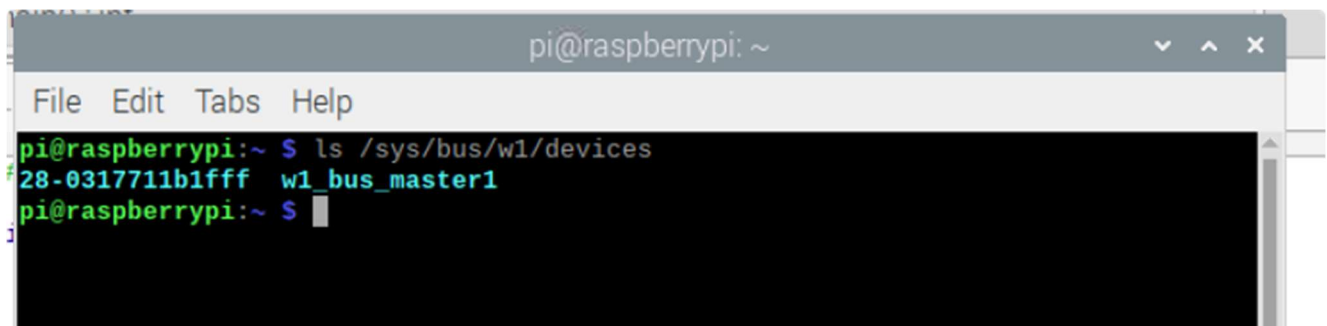
สำหรับพอร์ตที่ Raspberry Pi OS เตรียมไว้ให้ นั่นคือ GPIO หมายเลข 4 (wpi หมายเลข 7) โดยจะต้องเปิดใช้บริการพอร์ต 1-wire เสียก่อนผ่าน raspi-config (ตามที่ได้อธิบายไปแล้วในปฏิบัติการครั้งที่ 1)

หมายเหตุ เนื่องจากการจัดการการเชื่อมต่อแบบ 1-wire เป็นกลไกของระบบปฏิบัติการ เราจึงสามารถเปลี่ยนพอร์ตการเชื่อมต่อไปยังพอร์ตอื่น หรือกำหนดให้มีพอร์ตเชื่อมต่อแบบ 1-wire ได้มากกว่าหนึ่งพอร์ตพร้อมกัน โดยให้เพิ่มบรรทัดเหล่านี้ลงใน /boot/config.txt แล้วบูต Raspberry Pi ใหม่เพื่อให้การกำหนดนี้ไปใช้งาน

```
dtoverlay = w1-gpio // บรรทัดนี้กำหนดให้เปิดการใช้งาน 1-wire ปกติจะถูกสร้างโดย raspi-config อยู่แล้ว
dtoverlay = w1-gpio,gpiopin=4 // กำหนดให้พอร์ต GPIO นี้ใช้สำหรับ 1-wire สามารถเขียนหลายบรรทัดเพื่อเปิดการใช้งาน
// หลายๆ พอร์ตพร้อมกันได้
```

การจัดการเซ็นเซอร์อุณหภูมิ DS18B20 ผ่านบัส 1-wire อาศัย Raspberry Pi OS

หลังจากเปิดการใช้งาน 1-wire และรีบูตเครื่องแล้ว รวมทั้งมีการต่ออุปกรณ์ 1-wire ที่ Raspberry Pi OS รองรับ (ในที่นี้เราต่อเซ็นเซอร์อุณหภูมิ DS18B20 เข้ากับ GPIO-4) เมื่อเราดูรายการอุปกรณ์โดยใช้คำสั่ง ls ตามปกติก็จะได้ดังนี้



```
pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ ls /sys/bus/w1/devices
28-0317711b1fff w1_bus_master1
pi@raspberrypi:~ $
```

จากรูป จะเห็นว่าอุปกรณ์ 1-wire ที่เชื่อมต่อประกอบไปด้วย w1_bus_master1 ซึ่งคือตัว Raspberry Pi และ อุปกรณ์ที่มีเลข ID เป็น 28-0317711b1fff เป็นตัวเซ็นเซอร์อุณหภูมิ DS18B20 ทั้งนี้ หมายเลข ID ของอุปกรณ์จะเป็นค่าเฉพาะตามประเภทอุปกรณ์และรุ่น/บริษัท ดังนั้น ก่อนจะเขียนโปรแกรมจัดการอุปกรณ์ 1-wire ที่รองรับนั้น สิ่งแรกที่เราควรจะทำนั้น จะต้องตรวจสอบหมายเลข ID ของอุปกรณ์เสียก่อนว่าเป็นตัวเลขใด เพื่อจะได้เขียนโปรแกรมติดต่อได้อย่างถูกต้อง และเป็นการยืนยันว่าตัว Raspberry Pi OS นั้นรองรับอุปกรณ์ที่เชื่อมต่อนี้จริง

หากต่ออุปกรณ์แล้วยังไม่พบ เนื่องจากไดรเวอร์สำหรับตัว 1-wire และตัวอุปกรณ์ DS18B20 นี้ยังไม่ได้ติดตั้ง เราจะต้องทำการติดตั้งไดรเวอร์ด้วยคำสั่ง

```
sudo modprobe w1-gpio
sudo modprobe w1-therm
```

บรรทัดแรกเป็นการโหลดไดรเวอร์ของ 1-wire ส่วนบรรทัดที่สองเป็นการโหลดไดรเวอร์สำหรับอุปกรณ์อุณหภูมิในตระกูล DS13x20 ซึ่ง Raspberry Pi OS นั้นรองรับ

```

pi@raspberrypi: /sys/bus/w1/devices/28-0317711b1fff
File Edit Tabs Help
pi@raspberrypi:~ $ ls /sys/bus/w1/devices
28-0317711b1fff w1_bus_master1
pi@raspberrypi:~ $ cd /sys/bus/w1/devices/28-0317711b1fff
pi@raspberrypi:/sys/bus/w1/devices/28-0317711b1fff $ ls
driver hwmon id name power subsystem uevent w1_slave
pi@raspberrypi:/sys/bus/w1/devices/28-0317711b1fff $ cat w1_slave
ab 01 4b 46 7f ff 0c 10 20 : crc=20 YES
ab 01 4b 46 7f ff 0c 10 20 t=26687
pi@raspberrypi:/sys/bus/w1/devices/28-0317711b1fff $

```

จากนั้น หากเราเข้าไปในไดเรกทอรีย่อยของอุปกรณ์ slave (ในที่นี้คือ 28-0317711b1fff) และดูรายการภายใน เราจะเห็นว่ามียาละเอียดต่างๆ เกี่ยวกับอุปกรณ์ 1-wire slave ที่ตัวระบบปฏิบัติการอ่านมาไว้แสดงให้เห็น และเมื่อเราใช้คำสั่ง cat เพื่ออ่านไฟล์ w1_slave ระบบปฏิบัติการจะส่งคำสั่งอ่านข้อมูลไปยังอุปกรณ์ 1-wire (ในที่นี้คือตัวเซ็นเซอร์วัดอุณหภูมิ) และจะแสดงข้อมูลที่ตอบกลับมา โดยจะเห็นว่าในที่นี้แสดงค่า t=22687 ซึ่งหมายถึงค่าอุณหภูมิเท่ากับ 26.687 องศาเซลเซียส (สำหรับค่าตัวหารจะเป็นเช่นใดนั้น ขึ้นอยู่กับสเปคของอุปกรณ์ slave ที่เชื่อมต่อเป็นหลัก)

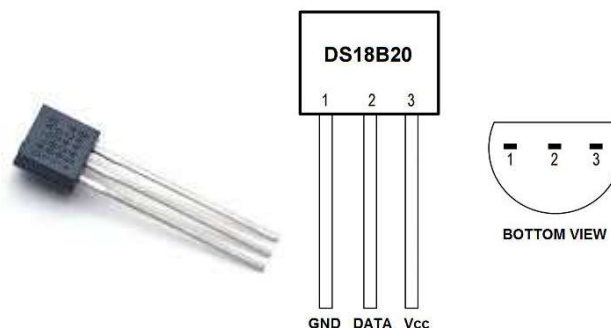
การเขียนโปรแกรมอ่านข้อมูลจากเซ็นเซอร์วัดอุณหภูมิ DS18B20

เซ็นเซอร์วัดอุณหภูมิรุ่น DS18B20 นี้มีการนำมาใช้งานกันอย่างแพร่หลาย เนื่องจากมีความทนทานต่อความชื้นและอุณหภูมิ (สามารถแช่ในน้ำได้โดยไม่เสียหาย) และรองรับโดยระบบปฏิบัติการ Raspberry Pi OS

สำหรับการต่อวงจรในที่นี้ เราจะเชื่อมต่อขาข้อมูล (ขาสีเหลือง) เข้ากับ GPIO-4 และเชื่อมต่อตัวต้านทานขนาด 4.7Kohm ดังรูป



แบบมีโลหะครอบกันน้ำ ขาสัญญาณ DQ (เหลือง) VDD (แดง) GND (ดำ)



แบบตัวถังพลาสติกเปลือย

อุปกรณ์ที่ต้องการ

- บอร์ด Raspberry Pi พร้อมแอดแดเตอร์
- เซ็นเซอร์อุณหภูมิ DS18B20 1 ตัว
- R 4.7kohm 1 ตัว

โปรแกรมตัวอย่าง ใช้ไดเรกทอรีของลินุกซ์ที่รองรับ

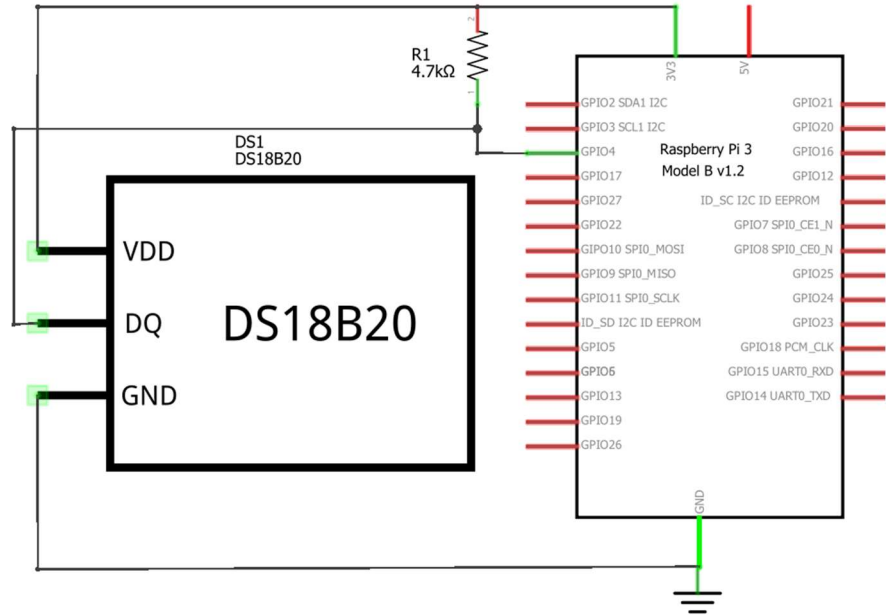
เซ็นเซอร์อุณหภูมิ DS18B20

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <dirent.h>
#include <string.h>
#include <time.h>
```

```
int main(){
    char path[64] =
"/sys/bus/w1/devices/";
    char rom[20];
    char buf[256];
    DIR *dirp;
    struct dirent *direntp;
    int fd = -1;
    char *temp;
    float value;
    // mount 1wire if not exists
    system ("sudo modprobe w1-gpio");
    // mount DS-dbx20 temperature sensor
    system ("sudo modprobe w1-therm");

    //Search for temperature device
    if((dirp = opendir(path))!=NULL){
        printf("Error connecting 1-wire temperature sensor!\n");
        return -1;
    }
    while((direntp = readdir(dirp))!=NULL)
    {
        if(strstr(direntp->d_name,"28-5")){
            strcpy(rom,direntp->d_name);
            printf("Found temperature device at %s\n",rom);
        }
    }
    closedir(dirp);
    //Identified target file to be read
    strcat (path,rom);
    strcat (path,"/w1_slave");

    while(1){
        if((fd=open(path,O_RDONLY))<0){
            printf("Error reading temperature device\n");
            return -1;
        }
        if(read(fd,buf,sizeof(buf))<0){
            printf("Read error\n");
            return -1;
        }
        // Search for location of t=xxxxx
        temp = strchr(buf,'t');
        sscanf(temp,"t=%s",temp);
        value =atof(temp)/1000;
        printf("temp : %3.3f c\n",value);
        sleep(1);
    }
    return 0;
}
```



อย่าลืมเปิดการใช้งานพอร์ต 1-wire ใน raspi-config ก่อนใช้งาน
ตัวอย่างนี้ (และปิดก่อนทดลองตัวอย่างถัดๆ ไปในปฏิบัติการนี้)

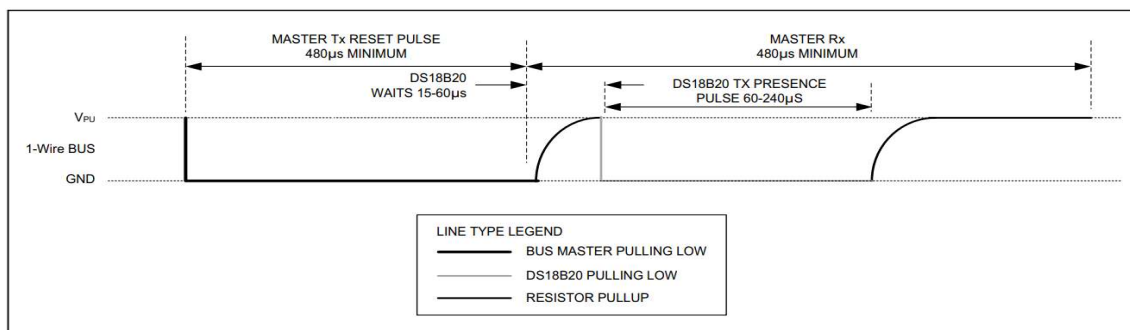
ใช้ ls /sys/bus/w1/devices/ เพื่อดูหมายเลขอุปกรณ์ 1-
wire ที่ขึ้นต้นด้วย 28 แล้วแก้จุดนี้ให้สัมพันธ์กัน

จากโปรแกรมตัวอย่างข้างต้น เริ่มต้นด้วยการใช้ modprobe เพื่อทำการติดตั้งไดรเวอร์สำหรับ 1-wire และ DS18B20 (หากไดรเวอร์ถูกติดตั้งอยู่ก่อนแล้ว จะไม่มีผลใดๆ) จากนั้นเป็นการสืบทอดเข้าไปยังไดเรกทอรีย่อยของอุปกรณ์ ซึ่งในที่นี้เราใช้คำค้น (wildcard) คือ 28-0 (อักขระตั้งต้นของ ID อุปกรณ์ของเรา) จากนั้นประกอบกับชื่อไฟล์ w1_slave เพื่อให้ได้ชื่อไฟล์และ path ของอุปกรณ์ที่จะทำการอ่านข้อมูลโดยใช้กลไกการอ่านไฟล์ตามปกติของภาษาซี แล้วจึงค้นหาค่า t=ค่าอุณหภูมิเพื่ออ่านค่าอุณหภูมิมา (ในรูปของข้อมูลสตริง) และนำมาแปลงเป็นค่าที่เราต้องการเพื่อแสดงบนหน้าจอต่อไป (เราหารค่าที่ได้ด้วย 1000 เพื่อให้ได้ค่าอุณหภูมิแท้จริงตามสเปคของ DS18B20)

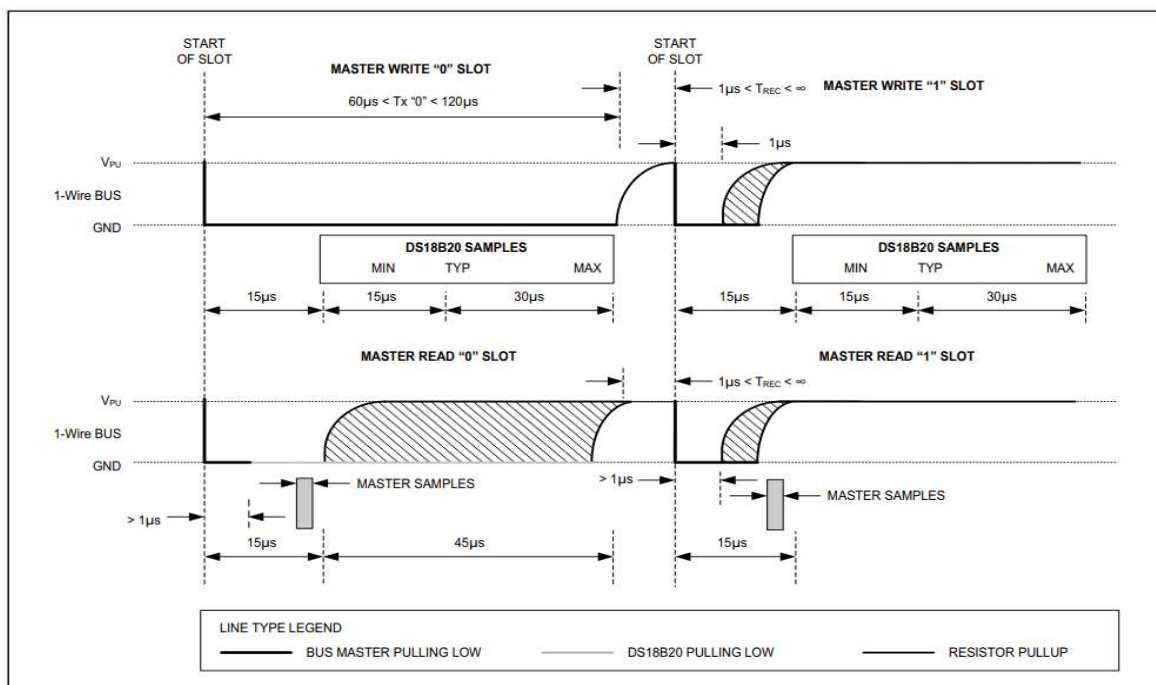
การอ่านข้อมูลจากเซ็นเซอร์วัดอุณหภูมิ DS18B20 โดยอาศัยกลไก bitbang

นอกเหนือจากการใช้บัส 1-wire ที่ Raspberry Pi รองรับแล้ว เราอาจจะเลือกเขียนโปรแกรมที่จะจำลองบัส GPIO ใดๆ ของ Raspberry Pi มาทำงานเช่นเดียวกับบัส 1-wire ได้ แต่ทั้งนี้ จะต้องมีการศึกษาถึงข้อกำหนดของอุปกรณ์หรือเซ็นเซอร์ที่นำมาต่อ ว่ามีลักษณะการรับส่งสัญญาณอย่างไร หลักการโดยทั่วไปนั้น อุปกรณ์ master (ในที่นี้คือ Raspberry Pi) จะต้องมีการดึงสัญญาณบัส 1-wire ลงกราวด์ในระยเวลานั้นๆ จากนั้นจะปล่อยบัสลอย (สำหรับ Raspberry Pi เราทำได้โดยการเปลี่ยน GPIO ให้เป็นอินพุต) แล้วรอรับการดึงบัส 1-wire โดยอุปกรณ์ slave ในช่วงเวลาที่กำหนด

สำหรับเซ็นเซอร์ DS18B20 นั้น มีการกำหนดการสั่งงานและการตอบสนองต่างๆ ดังนี้



เมื่อจะเริ่มติดต่อใดๆ กับ DS18B20 ทุกครั้ง master จะต้องส่งลอจิก 0 ไปยังบัส 1-wire เป็นเวลาไม่น้อยกว่า 480 ไมโครวินาที จากนั้นรอสัญญาณตอบกลับเป็นลอจิก 0 จาก DS18B20 ซึ่งจะมีการดึงสัญญาณลงอยู่ประมาณ 60-240 ไมโครวินาที



ส่วนการส่งการใดๆ ต่อจากนั้น ในกรณีการส่งข้อมูลไปยัง DS18B20 ข้อมูลแต่ละไบต์ จะส่งไปในลักษณะบิตต่ำสุดส่งไปก่อน และบิตศูนย์จะมีช่วงความยาวลอจิกศูนย์ที่ยาวประมาณ 60-120 ไมโครวินาที ส่วนบิตหนึ่งจะมีช่วงลอจิกศูนย์ที่ยาวประมาณ 1 ไมโครวินาที ดังรูปประกอบ ตัว DS18B20 จะรอเวลานับจากสัญญาณถูกถึงลง 0 ราว 15 ไมโครวินาทีแล้วจึงอ่านสถานะบนบัสว่าเป็นศูนย์หรือหนึ่ง

ในกรณีของการอ่านข้อมูลที่ส่งคืนจาก DS18B20 อุปกรณ์ master จะดึงสัญญาณ 1-wire เป็นศูนย์ในช่วงเวลาสั้นๆ ประมาณ 1 ไมโครวินาที จากนั้นปล่อยบัสและรอให้ DS18B20 ดึงสัญญาณ 1-wire ลงศูนย์ ซึ่งจะต้องทำเช่นนี้ในแต่ละบิตข้อมูลที่จะรับ โดยระยะเวลาการดึงลอจิกศูนย์ของ DS18B20 จะมีช่วงเวลาที่ยาวกว่ากรณีลอจิกหนึ่ง ดังรูปประกอบ อุปกรณ์ master เมื่อปล่อยบัสแล้ว จะจับเวลาต่อประมาณ 5-15 ไมโครวินาทีแล้วอ่านสถานะบัส 1-wire เพื่อให้ได้สถานะลอจิกที่ต้องการ

ส่วนคำสั่งต่างๆ ของ DS18B20 ที่น่าสนใจมีดังนี้

COMMAND	DESCRIPTION	PROTOCOL	1-Wire BUS ACTIVITY AFTER COMMAND IS ISSUED	NOTES
TEMPERATURE CONVERSION COMMANDS				
Convert T	Initiates temperature conversion.	44h	DS18B20 transmits conversion status to master (not applicable for parasite-powered DS18B20s).	1
MEMORY COMMANDS				
Read Scratchpad	Reads the entire scratchpad including the CRC byte.	BEh	DS18B20 transmits up to 9 data bytes to master.	2
Write Scratchpad	Writes data into scratchpad bytes 2, 3, and 4 (T _H , T _L , and configuration registers).	4Eh	Master transmits 3 data bytes to DS18B20.	3
Copy Scratchpad	Copies T _H , T _L , and configuration register data from the scratchpad to EEPROM.	48h	None	1
Recall E ²	Recalls T _H , T _L , and configuration register data from EEPROM to the scratchpad.	B8h	DS18B20 transmits recall status to master.	
Read Power Supply	Signals DS18B20 power supply mode to the master.	B4h	DS18B20 transmits supply status to master.	

Note 1: For parasite-powered DS18B20s, the master must enable a strong pullup on the 1-Wire bus during temperature conversions and copies from the scratchpad to EEPROM. No other bus activity may take place during this time.

Note 2: The master can interrupt the transmission of data at any time by issuing a reset.

Note 3: All three bytes must be written before a reset is issued.

ภายในของ DS18B20 จะมีเรจิสเตอร์เก็บข้อมูลจำนวน 9 ไบต์ (ซึ่งถูกเรียกว่า scratchpad) โดยสองไบต์แรก(16บิต) เป็นค่าอุณหภูมิคูณด้วย 16 (ดังนั้นเวลาอ่านค่า เราจะนำมารหาร 16 เพื่อให้ได้ค่าอุณหภูมิต่อไป) คำสั่งที่สำคัญของ DS18B20 คือคำสั่ง 0xCC ใช้กระโดดข้าม ROM code information (ตัว DS18B20 สามารถนำมาต่อได้หลายตัวบนบัสเดียวกัน และใช้ ROM code เพื่อแยกแยะว่าจะติดต่อกับตัวใด) สำหรับปฏิบัติการนี้เราใช้อุปกรณ์เพียงตัวเดียว ดังนั้นเราจึงไม่ต้องใช้ ROM code คำสั่งถัดไปคือการสั่งให้ DS18B20 อ่านค่าอุณหภูมิด้วย 0x44 และ คำสั่ง 0xBE ใช้อ่านค่าจาก scratchpad โดยเราเลือกอ่านแค่สองไบต์แรกที่ใช้เก็บอุณหภูมิ

จากความเข้าใจในกลไกการทำงานข้างต้น เราจึงได้โปรแกรมอ่านค่าอุณหภูมิจาก DS18B20 เป็นดังนี้

```
#include <pigpio.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdint.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
```



```
void gpio_stop(int sig);
int running = 1;
#define DS18B20_PIN 16 // GPIO_16
#define udelay(us) gpioDelay(us)
int DS18B20_Init();
void DS18B20_Write(uint8_t data);
uint8_t DS18B20_Read(void);

int main(){
    uint8_t tempL,tempH;
    float temp;

    if(gpioInitialise()<0) return -1;

    signal(SIGINT,gpio_stop);
    while(running){
        if(!DS18B20_Init()){
            printf("No DS18B20 connected!\n");
            sleep(1);
            continue;
        }
        usleep(1000);
        DS18B20_Write(0xCC); //Skip rom
        DS18B20_Write(0x44); //Read temperature

        usleep(100000);
        if(!DS18B20_Init ()){
            printf("No DS18B20 connected!\n\r");
            sleep(1);
            continue;
        }
        usleep(1000);
        DS18B20_Write (0xCC); // skip ROM
        DS18B20_Write (0xBE); // read first two bytes from scratch pad

        tempL = DS18B20_Read();
        tempH = DS18B20_Read();
        temp = ((float)((tempH<<8)|tempL))/16;
        printf("TH=0x%.2X TH=0x%.2X Temperature=%f \r\n",tempH,tempL,temp);
        sleep(1);
    }
    gpioTerminate();
    return 0;
}

int DS18B20_Init(){
    uint8_t response=0;
    gpioSetMode(DS18B20_PIN,PI_OUTPUT);
    gpioWrite(DS18B20_PIN,0);
    udelay(480); //Delay according to data sheet

    gpioSetMode(DS18B20_PIN,PI_INPUT);
    gpioSetPullUpDown(DS18B20_PIN,PI_PUD_OFF);
    udelay(80); //Wait for DS18B20 to acknowledge;

    if(!gpioRead(DS18B20_PIN)) response = 1;
    udelay(480); //Wait until DS18B20 ready to receive command
    return response;
}

void DS18B20_Write(uint8_t data){
    for (int i=0; i<8; i++){
        {
            gpioSetMode(DS18B20_PIN,PI_OUTPUT); // set as output
            gpioWrite(DS18B20_PIN, 0); // pull the pin LOW
```

```

        udelay (1); // wait for 1 us
        if(data&1){
            gpioWrite(DS18B20_PIN, 1); // pull the pin HIGH
            gpioSetMode(DS18B20_PIN,PI_INPUT); //Release Low pull
            udelay(60); // wait for another 60 us (1 us low + 59 us high pull floating)
        }else{
            udelay(60); // wait for 60-120 us according to datasheet (60 us low pull)
            gpioWrite(DS18B20_PIN, 1); // pull the pin HIGH
            gpioSetMode(DS18B20_PIN,PI_INPUT); //Release Low pull
        }
        data >>=1;
        udelay(5); // Wait for next bit
    }
    gpioSetMode(DS18B20_PIN,PI_INPUT); //Release 1-wire bus
}

uint8_t DS18B20_Read(void){
    uint8_t value=0;

    for(int i=0;i<8;i++){
        gpioSetMode(DS18B20_PIN,PI_OUTPUT); // set as output
        gpioWrite(DS18B20_PIN, 0); // pull the pin LOW
        udelay(1);
        gpioWrite(DS18B20_PIN, 1); // pull the pin HIGH
        gpioSetMode(DS18B20_PIN,PI_INPUT); // master releases 1-wire bus

        value >>=1;
        udelay (5); // wait for < 15us from the start of pulling LOW
        if(gpioRead(DS18B20_PIN) // CCheck 1-wire status
            value |= 0x80; // read = 1
            udelay (60); // wait for 50 us (The whole bit is atleast 60 us not including 1 us
between bits)
        }
        return value;
    }
}

void gpio_stop(int sig){
    printf("User pressing CTRL-C");
    running = 0;
}

```

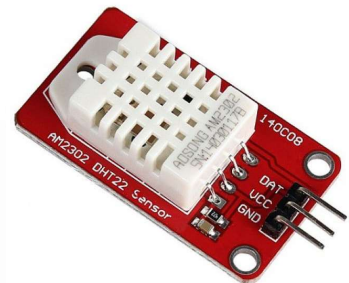
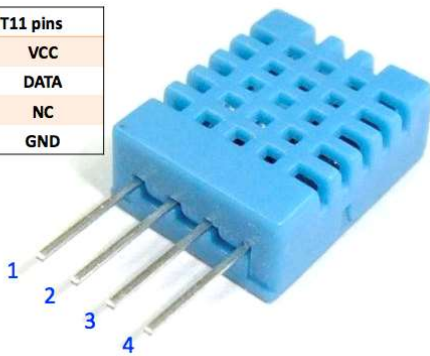
[illegible]

การเขียนโปรแกรมจัดการอุปกรณ์ 1-wire โดยตรงกับอุปกรณ์ DHT11/DHT21/AM2301/DHT22/AM2302

สำหรับอุปกรณ์ที่ระบบปฏิบัติการไม่รองรับนั้น เราจำเป็นต้องเขียนโปรแกรมในการอ่านข้อมูลโดยตรง และเนื่องจากตัว Raspberry Pi นั้นมีความเร็วในการประมวลผลอยู่มากพอ เราจึงสามารถเขียนโปรแกรมในลักษณะที่เรียกว่า bit banging หรือการใช้ซอฟต์แวร์ในการจัดการการสร้างสัญญาณพัลส์ และการอ่านสัญญาณที่ตอบกลับโดยอ่านจากสถานะที่เกิดขึ้นตามช่วงเวลาต่างๆ ได้โดยตรง

ผลเสียของการเขียนโปรแกรมในลักษณะเช่นนี้ก็คือ เนื่องจากระบบปฏิบัติการ Raspberry Pi OS นั้นไม่ใช่ระบบปฏิบัติการแบบเวลาจริง (real-time OS) ดังนั้นค่าเวลาที่เรากำหนดและใช้ในโปรแกรมจะมีความคลาดเคลื่อนไปจากที่ควรจะเป็น และอาจมีค่าความคลาดเคลื่อนที่ไม่แน่นอน ในตัวอย่างการประยุกต์กับอุปกรณ์ DHT11/DHT21/AM2301 เราอาจนำกลไกการทำ bit banging มาใช้ได้เนื่องจากมีบิตตรวจสอบข้อมูลมาเพื่อใช้คำนวณตรวจสอบว่าค่าที่อ่านได้นั้นถูกต้องหรือไม่

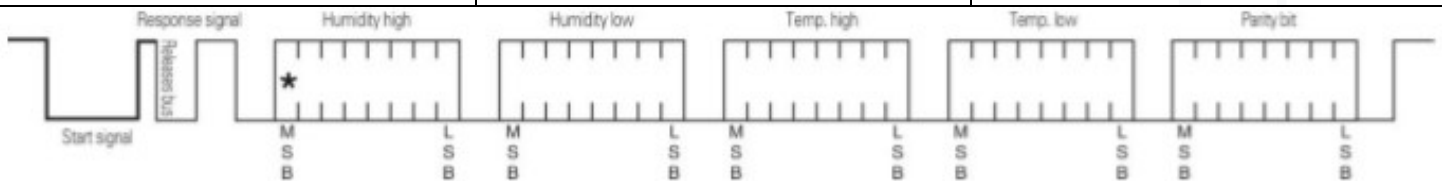
DHT11 pins	
1	VCC
2	DATA
3	NC
4	GND



เซ็นเซอร์วัดอุณหภูมิและความชื้น รูปทางซ้ายคือ DHT11 รูปกลางคือ DHT21/AM2301 รูปทางขวาคือ DHT22/AM2302

เซ็นเซอร์ที่เราจะนำมาใช้ต่อไปนี้เป็นเซ็นเซอร์ที่ใช้วัดอุณหภูมิและความชื้นได้พร้อมกัน โดยเซ็นเซอร์ DHT11 เป็นเซ็นเซอร์ราคาถูก ที่มีย่านการวัดได้แคบกว่า ในขณะที่เซ็นเซอร์ DHT21 หรืออีกชื่อหนึ่งคือ AM2301 นั้นมีราคาแพงและมีย่านการวัดที่กว้างกว่า มีคุณภาพดีกว่า ในขณะที่ DHT22/AM2302 มีประสิทธิภาพอยู่ระหว่างสองรุ่นแรกที่กำลังกล่าวมาข้างต้น

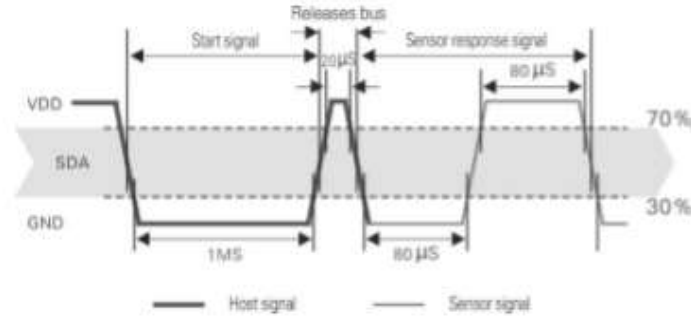
เซ็นเซอร์	ย่านความชื้นที่รองรับ	ย่านอุณหภูมิที่รองรับ
DHT11	30-90% ที่ 0 °c 20-90% ที่ 25 °c 20-80% ที่ 50 °c	0 °c ถึง 50 °c
DHT22/AM2302	0-100%	-40 °c ถึง 80 °c
DHT21/AM2301	0-99.9%	-40 °c ถึง 80 °c



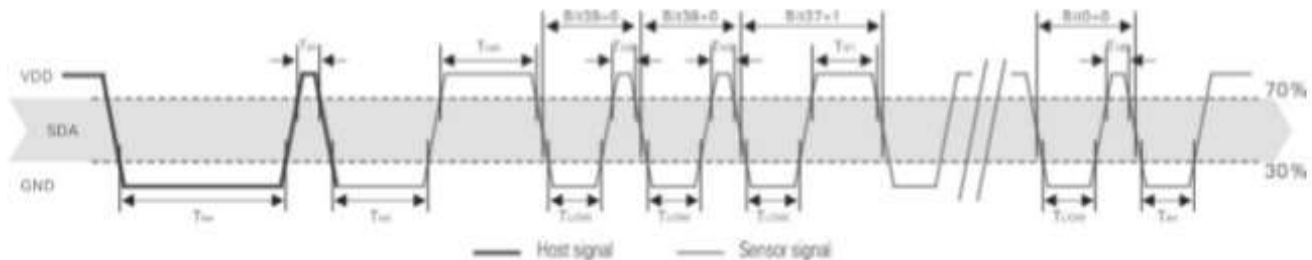
รูปแบบการรับส่งข้อมูลของ DHT11/DHT21/AM2301/DHT22/AM2302

สำหรับการรับส่งข้อมูลระหว่าง master ซึ่งในที่นี้คือ Raspberry Pi และตัวเซ็นเซอร์นั้น เริ่มต้นตัว master จะต้องส่งสัญญาณ start signal ออกไป ซึ่งตัว DHT11 ต้องการสัญญาณพัลส์ที่มีความยาว 18ms ในขณะที่ DHT21/AM2301 จะมีสัญญาณพัลส์ที่มีช่วงความยาวระหว่าง 1ms ถึง 20

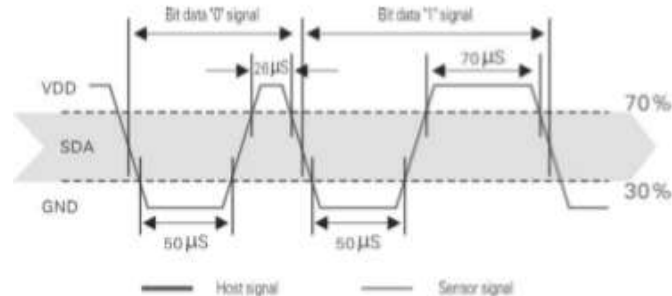
ms (ในตัวอย่างที่ให้ไว้จะใช้ค่า 18ms ซึ่งจะทำให้สามารถใช้งานได้กับเซ็นเซอร์ทั้งสามรุ่น) จากนั้นเซ็นเซอร์จะส่งสัญญาณพัลส์กลับมาภายในเวลา 20-40 μ s โดยจะส่งพัลส์มีช่วงเวลาเท่ากับ 80 μ s จากนั้นจึงส่งบิตข้อมูลจำนวนทั้งหมด 40 บิตออกมาอย่างต่อเนื่อง แล้วจบด้วยพัลส์ปิดท้าย



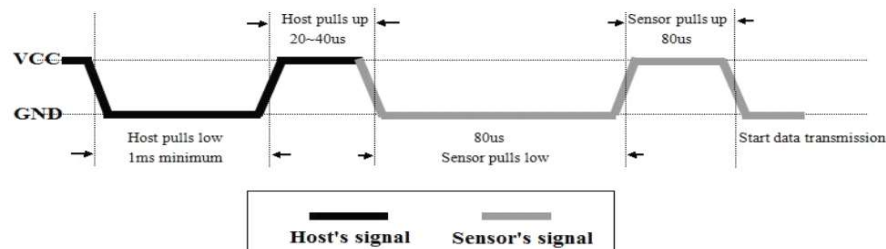
การเริ่มต้นทำงานของ DHT21/AM2301 (สำหรับ DHT11 นั้นคาบเวลา start signal จะเท่ากับ 18ms)



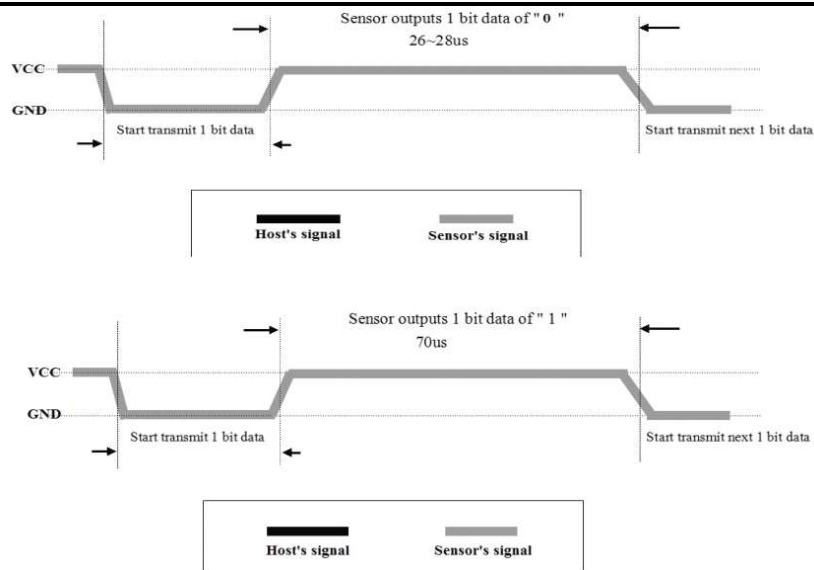
การรับส่งข้อมูลของ DHT11/DHT21/AM2301



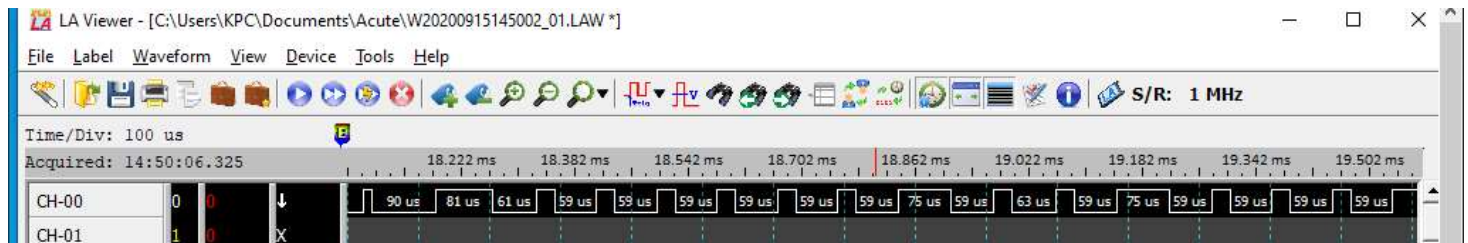
ตัวอย่างลักษณะบิต “0” และ บิต “1” ของ DHT11/DHT21/AM2301



การลั้งเริ่มต้นทำงานของ DHT22/AM2302



ลักษณะการจำแนกบิตข้อมูล 0 และ 1 ของ DHT22/AM2302



ตัวอย่างสัญญาณบางส่วนของข้อมูลที่ส่งจาก AM2301 (พัลส์ 0 ความยาว 90 μs ที่เห็นตัวแรกคือพัลส์ที่ส่งคืนกลับมาโดย AM2301 หลังจากที่ Raspberry Pi ส่งสัญญาณ start signal ไปแล้ว

การใช้งาน DHT11

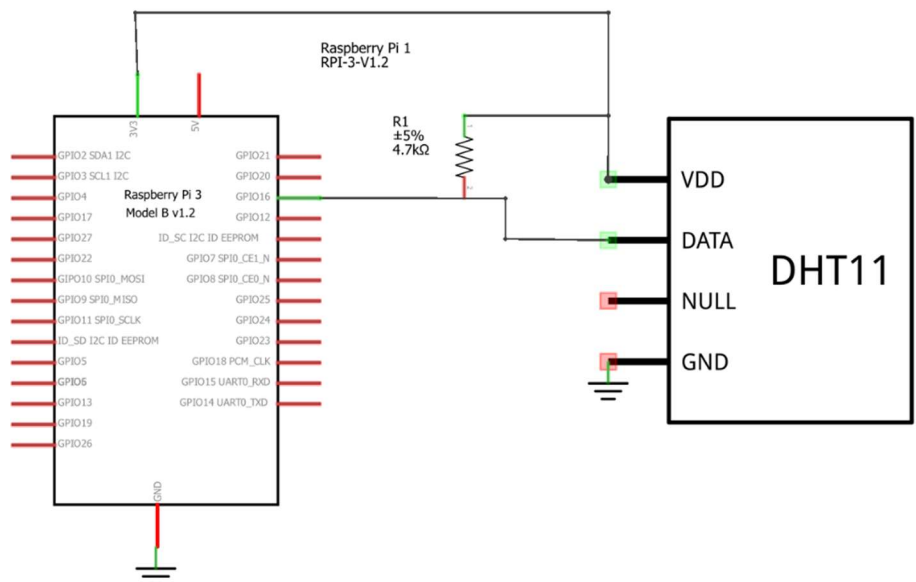
อุปกรณ์ที่ต้องการ

- บอร์ด Raspberry Pi พร้อมแอดapter
- เซ็นเซอร์วัดอุณหภูมิ DHT11 1 ตัว
- R 4.7kohm 1 ตัว

โปรแกรมตัวอย่างสำหรับ DHT11

```
#include <pigpio.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdint.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
```

```
#define udelay(us) gpioDelay(us)
#define DHT11_PIN 16
```



```
#define DHT11_DELAY 78

struct DHT11_data{
    float temp;
    float humidity;
}data;

int gpioLED[4] ={4,5,6,12};
int running=true;

void initGPIO();
void gpio_stop(int sig);
int piHiPri(const int pri);
//void udelay(const long us);

int DHT11_read(struct DHT11_data *data);
#define DHT11_readOneByte(x) { \
    register int _i,_j; \
    for(_i=0;_i<8;_i++){ \
        for(_j=0;_j<100;_j++){ \
            if(gpioRead(DHT11_PIN)==0) break; \
            udelay(1); \
        } \
        udelay(DHT11_DELAY); \
        x <<=1; \
        if(gpioRead(DHT11_PIN)) \
            x|=1; \
    } \
}

void *checkDistance(void *param);
void *showLED(void *param);

int main(){
    pthread_t tid[2];
    pthread_attr_t attr[2];
    void *(*thread[2])(void *)={checkDistance,showLED};
    int i;

    initGPIO();
    signal(SIGINT,gpio_stop);

    for(i=0;i<2;i++){
        pthread_attr_init(&attr[i]);
        pthread_create(&tid[i],&attr[i],thread[i],NULL);
    }

    printf("Waiting all threads to stop...\n");
    fflush(stdout);
    for(i=0;i<2;i++){
        pthread_join(tid[i],NULL);
    }
    for(i=0;i<2;i++){
        pthread_attr_destroy(&attr[i]);
    }
    gpioTerminate();
    return 0;
}

void initGPIO(){
    int i;

    if(gpioInitialise() < 0) exit(-1);

    gpioSetMode(DHT11_PIN,PI_INPUT);
```

```
    gpioSetPullUpDown(DHT11_PIN, PI_PUD_OFF);
    for (i=0; i<4; i++)
        gpioSetMode(gpioLED[i], PI_OUTPUT);
}

void *checkDistance(void *param){
    while(running){
        if(DHT11_read(&data))
            printf("Temp = %5.1fc, Humidity = % 5.1f%%\r", data.temp, data.humidity);
        else printf("Error reading data \r");
        fflush(stdout);
        usleep(200000);
    }
    pthread_exit(NULL);
}

void *showLED(void *param){
    while(running){
        if(data.temp<21) gpioWrite(gpioLED[0],0);
        else gpioWrite(gpioLED[0],1);
        if(data.temp<23) gpioWrite(gpioLED[1],0);
        else gpioWrite(gpioLED[1],1);
        if(data.temp<25) gpioWrite(gpioLED[2],0);
        else gpioWrite(gpioLED[2],1);
        if(data.temp<27) gpioWrite(gpioLED[3],0);
        else gpioWrite(gpioLED[3],1);
        usleep(100000);
    }
    pthread_exit(NULL);
}

void gpio_stop(int sig){
    printf("Exiting..., please wait\n");
    running = false;
}

int DHT11_read(struct DHT11_data *data){
    int i;
    uint8_t temp_l,temp_h,hum_l,hum_h,crc;
    char tmp[16];

    //Sending Start signal
    hum_h=hum_l=temp_h=temp_l=crc=0;
    gpioSetMode(DHT11_PIN, PI_OUTPUT);
    gpioWrite(DHT11_PIN,0);
    usleep(18000);
    gpioWrite(DHT11_PIN,1);
    //waiting for response
    gpioSetMode(DHT11_PIN, PI_INPUT);
    for (i=0; i<100; i++){
        if(gpioRead(DHT11_PIN)==0) break;
        udelay(1);
    }
    for (i=0; i<100; i++){
        if(gpioRead(DHT11_PIN)==1) break;
        udelay(1);
    }
    // Read data
    DHT11_readOneByte(hum_h);
    DHT11_readOneByte(hum_l);
    DHT11_readOneByte(temp_h);
    DHT11_readOneByte(temp_l);
    DHT11_readOneByte(crc);

    printf("hum_h = %.2X  hum_l = %.2X temp_h = %.2X temp_l = %.2X crc= %.2X\n",
```

```

hum_h,hum_l,temp_h,temp_l,crc);

fflush(stdout);
// Check if data is valid
if ((hum_h+hum_l+temp_h+temp_l)%0xff)!=crc)
    return 0;

sprintf(tmp,"%u.%u",hum_h,hum_l);
data->humidity = atof(tmp);
sprintf(tmp,"%u.%u",temp_h,temp_l);
data->temp = atof(tmp);

return 1;
}

```

จากโปรแกรมตัวอย่าง มีประเด็นที่น่าสนใจดังต่อไปนี้

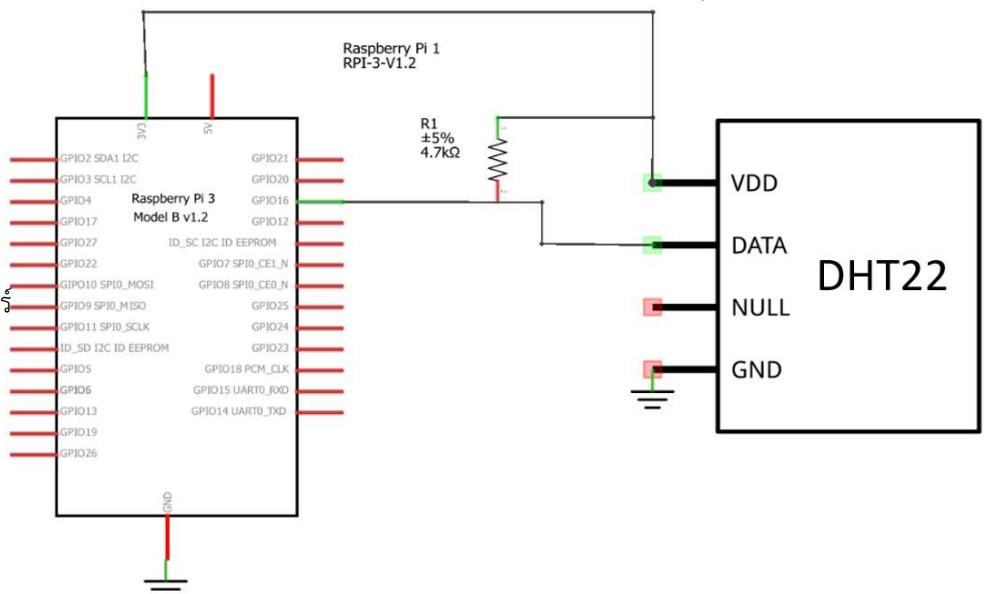
- ฟังก์ชัน gpioDelay() เป็นฟังก์ชันของ pigpio ที่ใช้เพื่อหน่วงเวลาในหน่วยไมโครวินาที ทั้งนี้เนื่องจาก usleep() ของ Raspberry Pi เองนั้นไม่มีความแม่นยำเพียงพอ เนื่องจากระบบปฏิบัติการที่เราใช้ไม่ใช่ RTOS ดังนั้น หากเราไม่ใช่ gpioDelay() ที่ pigpio สร้างไว้ให้ เราจะต้องสร้างฟังก์ชันขึ้นเองเพื่อให้ได้ความแม่นยำเพียงพอที่จะนำมาใช้งานได้
- มีการนิยามมาโคร DHT11_readOneByte () เพื่อใช้ในการดักสัญญาณพัลส์ที่สร้างขึ้นโดย DHT11 แทนการเขียนเป็นฟังก์ชัน ทั้งนี้เนื่องจากการเรียกใช้ฟังก์ชันจะเสียเวลาในการเขียนและอ่านค่าและจัดการกับสแต็ก และการกระโดดของโปรแกรมไปมา การใช้มาโครเป็นทางออกแทนการเขียนโค้ดดังกล่าวเป็นจำนวน 5 ครั้งตรงจุดที่มีการเรียกใช้ในฟังก์ชัน DHT11_read() ทำให้การอ่านค่ามีความแม่นยำมากขึ้น
- เซ็นเซอร์มีการคำนวณความถูกต้องโดยการนำเอาไบต์ข้อมูลสี่ตัวแรกมาบวกกัน (บิตที่เกินจะถูกตัดทิ้งไป) แล้วเทียบค่ากับไบต์ที่ห้า หากตรงกันแสดงว่าข้อมูลนั้นรับมาถูกต้อง
- สำหรับ DHT11 นั้น ค่าความชื้นจะประกอบไปด้วยค่าตัวเลขหน้าทศนิยมและหลังทศนิยม โดยจะเห็นการใช้ฟังก์ชัน sprintf() เพื่อพิมพ์ข้อความเป็นเลขหน้าทศนิยมแล้วตามด้วยเลขหลังทศนิยม จากนั้นจึงแปลงเป็นข้อมูลด้วย atof() อีกทีหนึ่ง
- การคำนวณค่าอุณหภูมิ ของ DHT11 นั้น มีลักษณะคล้ายคลึงกับการหาค่าความชื้น ซึ่งจะมีเลขหน้าทศนิยมและเลขหลังทศนิยม กระบวนการแปลงเป็นข้อมูลกระทำเช่นเดียวกับการแปลงค่าความชื้น
- อนึ่ง สำหรับเซ็นเซอร์ DHT21 และ AM2301 นั้นจะแตกต่างออกไป การคำนวณค่าความชื้น ทำได้โดยการนำไบต์สูงและไบต์ต่ำของไบต์ความชื้น (ไบต์ที่หนึ่ง และไบต์ที่สอง) มาประกอบกัน แล้วหารด้วย 10 ส่วนการคำนวณค่าอุณหภูมินั้นจะมีบิตสูงสุด (MSB) ของไบต์สูงของค่าอุณหภูมินำไปกำหนดว่าค่าที่อ่านได้เป็นค่าบวกหรือลบ ส่วน 15 บิตที่เหลือ (7บิตของไบต์สูง และ 8 บิตของไบต์ต่ำ) จะถูกนำมาใช้คำนวณเป็นค่าอุณหภูมิ โดยหารด้วย 10 (ในกรณีที่เป็ค่าติดลบ ให้ค่าที่เหลือมาคำนวณโดยทันที ไม่ต้องทำ two-complement อีก)

การใช้งาน DHT22

อุปกรณ์ที่ต้องการ

- บอร์ด Raspberry Pi พร้อมแอดapter
- เซ็นเซอร์วัดอุณหภูมิ DHT22 1 ตัว
- R 4.7kohm 1 ตัว

โปรแกรมตัวอย่างสำหรับ DHT22




```

#include <pigpio.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdint.h>
#include <unistd.h>
#include <time.h>
#include <sched.h>
#include <string.h>

void gpio_stop(int sig);
int running = 1;
#define DHT22_PIN 16 // GPIO_16
#define DHT22_DELAY 30 // Delay time for detecting 0 or 1
struct DHT22_data{
    float temp;
    float humidity;
};
#define udelay(us) gpioDelay(us)

int DHT22_Init();
int DHT22_read(struct DHT22_data *data);
#define DHT22_readOneByte(x) { \
    register int _i, _j; \
    for(_i=0; _i<8; _i++){ \
        for(_j=0; _j<100; _j++){ \
            if(gpioRead(DHT22_PIN)==1) break; \
            udelay(1); \
        } \
        udelay(DHT22_DELAY); \
        x <<=1; \
        if(gpioRead(DHT22_PIN)) \
            x|=1; \
        for(_j=0; _j<100; _j++){ \
            if(gpioRead(DHT22_PIN)==0) break; \
            udelay(1); \
        } \
    } \
}

int main(){
    struct DHT22_data data;
    DHT22_Init();
    signal(SIGINT, gpio_stop);
    while(running){
        if(DHT22_read(&data))
            printf("Temp = %5.1fc, Humidity = % 5.1f%%\r", data.temp, data.humidity);
        else printf("Error reading data \r");
        fflush(stdout);
        sleep(1);
    }
    gpioTerminate();
    return 0;
}

int DHT22_Init(){
    if(gpioInitialise()<0) return -1;

    gpioSetMode(DHT22_PIN, PI_INPUT);
    gpioSetPullUpDown(DHT22_PIN, PI_PUD_OFF);
    return 0;
}

int DHT22_read(struct DHT22_data *data){
    int i;
    uint8_t temp_l, temp_h, hum_l, hum_h, crc;
    uint16_t temp, hum;

```

```

//Sending Start signal
hum_h=hum_l=temp_h=temp_l=crc=0;

gpioSetMode(DHT22_PIN, PI_OUTPUT);
gpioWrite(DHT22_PIN, 0);
udelay(1000);
gpioWrite(DHT22_PIN, 1);
udelay(40);
//waiting for response
gpioSetMode(DHT22_PIN, PI_INPUT);
for(i=0; i<100; i++){           // pull low cycle
    if(gpioRead(DHT22_PIN)==1) break;
    udelay(1);
}
for(i=0; i<100; i++){           //pull high cycle
    if(gpioRead(DHT22_PIN)==0) break;
    udelay(1);
}
// Read data
DHT22_readOneByte(hum_h);
DHT22_readOneByte(hum_l);
DHT22_readOneByte(temp_h);
DHT22_readOneByte(temp_l);
DHT22_readOneByte(crc);

printf("hum_h = %.2X hum_l = %.2X temp_h = %.2X temp_l = %.2X crc=
%.2X\n", hum_h, hum_l, temp_h, temp_l, crc);
fflush(stdout);
// Check if data is valid
if(((hum_h+hum_l+temp_h+temp_l)&0xff)!=crc)
    return 0;

hum = (((uint16_t)hum_h)<<8)|(uint16_t)hum_l;
temp = (((uint16_t)temp_h)<<8)|(uint16_t)temp_l;
data->humidity = (float)hum/10.0;
if(temp&0x8000) data->temp = -((float)(hum&0x7fff))/10.0;
else data->temp = (float)temp/10.0;
return 1;
}

void gpio_stop(int sig){
    printf("User pressing CTRL-C");
    running = 0;
}

```

จากโปรแกรมตัวอย่าง DHT22 มีประเด็นที่น่าสนใจ ดังต่อไปนี้

- ฟังก์ชัน gpioDelay() เป็นฟังก์ชันของ pigpio ที่ใช้เพื่อหน่วงเวลาในหน่วยไมโครวินาที ทั้งนี้เนื่องจาก usleep() ของ Raspberry Pi เองนั้นไม่มีความแม่นยำเพียงพอ เนื่องจากระบบปฏิบัติการที่เราใช้ไม่ใช่ RTOS ดังนั้น หากเราไม่ใช่ gpioDelay() ที่ pigpio สร้างไว้ให้ เราจะต้องสร้างฟังก์ชันขึ้นเองเพื่อให้ได้ความแม่นยำเพียงพอที่จะนำมาใช้งานได้
- ตามสเปคของ DHT22 โสสต์จะต้องดึงสัญญาณลง 0 ด้วยคาบเวลาไม่น้อยกว่า 1 มิลลิวินาที ก่อนจะดึงสัญญาณขึ้น 1 อีกประมาณ 20-40 ไมโครวินาที เพื่อเริ่มการวัดอุณหภูมิ (แล้วจะต้องปล่อยสัญญาณออก เปลี่ยนเป็นอินพุต) จากนั้น DHT22 จะดึงสัญญาณลง 0 ประมาณ 80 ไมโครวินาที ตามด้วยสัญญาณ 1 อีกประมาณ 80 ไมโครวินาที ก่อนจะเริ่มส่งบิตแรกออกมา
- การส่งแต่ละบิตข้อมูลนั้น จะเริ่มจากการส่งลอจิก 0 ออกมาประมาณ 50 ไมโครวินาที ตามด้วยสัญญาณบิต หากเป็นบิต 1 จะใช้ลอจิก 1 ด้วยคาบเวลาประมาณ 70 ไมโครวินาที หรือบิต 0 จะใช้ลอจิก 1 ด้วยคาบเวลาประมาณ 26-28 ไมโครวินาที ดังนั้นเวลาโฮสต์อ่านค่าหลังจากดักจับการเปลี่ยนสถานะจาก 0 เป็น 1 (เมื่อดักจับลอจิก 0 นำ) ก็จะจับเวลาต่อไปประมาณ 30-40 ไมโครวินาที แล้วจึงอ่านค่าสัญญาณบนบัส หากอ่านได้ 0 แสดงว่าบิตนั้นเป็นบิต 0 แต่ถ้าอ่านได้เป็น 1 ก็แสดงว่าว่าเป็นบิต 1 แล้วจึงเริ่มดักจับการเปลี่ยนสถานะจาก 1 เป็น 0 เพื่อเริ่มต้นสัญญาณนำของบิตถัดไป

- มีการนิยามมาโคร DHT22_readOneByte () เพื่อใช้ในการดักสัญญาณพัลส์ที่สร้างขึ้นโดย DHT22 แทนการเขียนเป็นฟังก์ชัน ทั้งนี้เนื่องจากการเรียกใช้ฟังก์ชันจะเสียเวลาในการเขียนและอ่านค่าและจัดการกับสแต็ก และการกระโดดของโปรแกรมไปมา การใช้มาโครเป็นทางออก แทนการเขียนโค้ดดังกล่าวเป็นจำนวน 5 ครั้งตรงจุดที่มีการเรียกใช้ในฟังก์ชัน DHT22_read() ทำให้การอ่านค่ามีความแม่นยำมากขึ้น
- เซ็นเซอร์มีการคำนวณความถูกต้องโดยการนำเอาไบต์ข้อมูลสี่ตัวแรกมาบวกกัน (บิตที่เกินจะถูกตัดทิ้งไป) แล้วเทียบค่ากับไบต์ที่ห้า หากตรงกันแสดงว่าข้อมูลนั้นรับมาถูกต้อง
- สำหรับ DHT22 นั้น ค่าความชื้นเป็นข้อมูล 16 บิต เมื่อแปลงเป็นเลขจำนวนเต็มฐานสิบแล้วให้หารด้วย 10 จะได้ค่าความชื้น ส่วนอุณหภูมิ นั้น จะใช้ข้อมูล 15 บิตล่าง(เลขฐานสอง) แปลงมาเป็นเลขฐานสิบแล้วหารด้วย 10 จะได้ค่าอุณหภูมิ หากบิตสูงสุด (b15) มีค่าเป็น 1 แสดงว่าค่าอุณหภูมิเป็นค่าติดลบ

ปฏิบัติการ: การนำอุปกรณ์วัดความชื้นและอุณหภูมิ มาใช้ในการสั่งเปิด-ปิด LED

อุปกรณ์ที่ต้องการ

- บอร์ด Raspberry Pi พร้อมอแดปเตอร์
- สายจัมป์จากขา GPIO ของบอร์ด
- โปรโตบอร์ด และสายจัมป์อีกตามต้องการ
- อุปกรณ์ DS18B20 หรือ DHT11/DHT21/AM2301/DHT22/AM2302 แล้วแต่ที่อาจารย์กำหนด
- LED จำนวน 4 ดวง
- R ขนาด 4.7kohm จำนวน 1 ตัว
- R ขนาด 1kohm จำนวน 4 ตัว

ให้นักศึกษาต่อวงจร LED จาก GPIO พอร์ตใดก็ได้จำนวน 4 ตัว (อาจใช้การต่อวงจรแบบเดียวกันกับในปฏิบัติการครั้งแรกๆ) และต่อวงจรตัววัดอุณหภูมิและความชื้น หรือตัววัดอุณหภูมิ เข้ากับบอร์ด Raspberry Pi ตามตัวอย่างที่ปรากฏในปฏิบัติการครั้งนี้

จากนั้นให้เขียนโปรแกรมแตกเรดออกมาจำนวน 2 เรดด้วยกัน เรดหนึ่งทำหน้าที่คอยอ่านค่าจากเซ็นเซอร์มาเป็นระยะ โดยใช้การอ่านค่าทุกๆ หนึ่งวินาที (อ่านหนึ่งครั้งแล้วหยุดรอหนึ่งวินาที) ทั้งนี้ในกรณีที่นักศึกษาใช้งาน DHT11/DHT21/AM2301 หากการอ่านค่า นั้นผิดพลาด จะต้องอ่านใหม่จนกว่าจะได้ค่าที่ถูกต้องด้วย

ส่วนอีกเรดหนึ่งนั้น ให้นำค่าที่ได้มาพิจารณา โดยให้นักศึกษากำหนดย่านอุณหภูมิที่แสดงตามความเหมาะสม และนำไปเปิด-ปิด LED เพื่อให้เห็นเป็นระดับอุณหภูมิอย่างคร่าวๆ ว่าในขณะนั้นมีค่าเท่าใด

ตัวอย่างเช่น หากอุณหภูมิปกติขณะนั้นเป็น 25°C นักศึกษาอาจแสดง LED ติดจำนวน 2 ดวง และที่เหลืออาจจะกำหนดดังนี้

ต่ำกว่า 25°C	LED ติด 1 ดวง
25-27°C	LED ติด 2 ดวง
27.1-29°C	LED ติด 3 ดวง
มากกว่า 29°C	LED ติด 4 ดวง

โน้ตเพิ่มเติม: ฟังก์ชันหน่วงเวลาความละเอียดสูงในกรณีที่ไม่ได้ใช้ไลบรารีของ pigpio

เนื่องจากตัว Raspberry Pi นั้น ใช้ระบบปฏิบัติการ Raspberry Pi OS (หรือในกรณีตัวอื่นๆ ที่อาจจะพบเห็นตัวอื่นเช่น Ubuntu) ซึ่งเป็นระบบปฏิบัติการลินุกซ์ และตัวระบบปฏิบัติการไม่ได้เป็นระบบปฏิบัติการทันเวลา (Real-time OS) ดังนั้น เมื่อเราพยายามเขียนโปรแกรมที่ต้องมีความแม่นยำในการห้วงเวลาสูงๆ แม้ว่าในลินุกซ์จะมี usleep() ที่มีหน่วยย่อยเป็นไมโครวินาที เราจะพบปัญหาว่าโปรแกรมอาจจะตอบสนองได้อย่างไม่แม่นยำ ดังเช่นตัวอย่างการสร้าง PWM ให้เซอร์โวในปฏิบัติการครั้งที่ 5 เป็นต้น

จากปฏิบัติการครั้งนี้ เรามีความจำเป็นต้องใช้กลไก bit-banging เพื่อสร้างสัญญาณและตอบรับการทำงานกับอุปกรณ์ 1-wire ซึ่งมักจะต้องอาศัยการสร้างสัญญาณลอจิกตอบสนองและสิ่งงานที่ต้องอาศัยความแม่นยำและความเร็ว ด้วยเหตุที่ไลบรารี pigpio นั้นมีฟังก์ชัน gpioDelay() มาให้แล้ว แต่ถ้าเราต้องเขียนโปรแกรมที่ไม่ได้ใช้ไลบรารี pigpio ก็จำเป็นที่จะต้องสร้างฟังก์ชันหน่วงเวลาแทนการใช้ usleep() ที่จะไม่ค่อยมีความแม่นยำ หากช่วงเวลารอคอยสั้นกว่า 1 มิลลิวินาที

ฟังก์ชันที่เราสร้างขึ้น จะมีอยู่สองฟังก์ชันคือ piHiPri() ที่ใช้สำหรับกำหนดให้เธรดปัจจุบันได้รับส่วนแบ่งเวลาการทำงานมากขึ้นกว่าปกติ และฟังก์ชัน udelay() ที่เป็นฟังก์ชันหน่วงเวลาแบบวนรอบตรวจสอบ ดังนี้

```
int piHiPri(const int pri){
// Source code of this function by Gordon Henderson
    struct sched_param sched;
    memset(&sched,0,sizeof(sched));
    if(pri>sched_get_priority_max (SCHED_RR))
        sched.sched_priority = sched_get_priority_max (SCHED_RR);
    else
        sched.sched_priority = pri;
    return sched_setscheduler(0,SCHED_RR,&sched);
}

void udelay(const long us){
// Delay without yielding process
    long st;
    long tdif;
    struct timespec tnow;

    clock_gettime(CLOCK_REALTIME,&tnow);
    st = tnow.tv_nsec;
    while(1){
        clock_gettime(CLOCK_REALTIME,&tnow);
        tdif = tnow.tv_nsec - st;
        if(tdif < 0) tdif += 1000000000;
        if(tdif > (us*1000)) break;
    }
}
```

และเมื่อนำมาใช้กับเซ็นเซอร์ DHT11 จะได้โปรแกรมดังนี้

```
#include <pigpio.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdint.h>
#include <unistd.h>
#include <time.h>
#include <sched.h>
#include <string.h>

void gpio_stop(int sig);
int running = 1;
#define DHT11_PIN 16 // GPIO_16
#define DHT11_DELAY 79 // Delay time for detecting 0 or 1
struct DHT11_data{
    float temp;
    float humidity;
};
```

```

int piHiPri(const int pri);
void udelay(const long us);

int DHT11_Init();
int DHT11_read(struct DHT11_data *data);
#define DHT11_readOneByte(x) { \
    register int _i, _j; \
    for(_i=0; _i<8; _i++){ \
        for(_j=0; _j<100; _j++){ \
            if(gpioRead(DHT11_PIN)==0) break; \
            udelay(1); \
        } \
        udelay(DHT11_DELAY); \
        x <<=1; \
        if(gpioRead(DHT11_PIN)) \
            x|=1; \
    } \
}

int main(){
    struct DHT11_data data;
    DHT11_Init();
    signal(SIGINT, gpio_stop);
    while(running){
        if(DHT11_read(&data))
            printf("Temp = %5.1fc, Humidity = % 5.1f%%\r", data.temp, data.humidity);
        else printf("Error reading data \r");
        fflush(stdout);
        sleep(1);
    }
    gpioTerminate();
    return 0;
}

int DHT11_Init(){
    if(gpioInitialise()<0) return -1;

    gpioSetMode(DHT11_PIN, PI_INPUT);
    gpioSetPullUpDown(DHT11_PIN, PI_PUD_OFF);
    return 0;
}

int DHT11_read(struct DHT11_data *data){
    int i;
    uint8_t temp_l, temp_h, hum_l, hum_h, crc;
    char tmp[16];
    //Sending Start signal
    hum_h=hum_l=temp_h=temp_l=crc=0;

    piHiPri(50);

    gpioSetMode(DHT11_PIN, PI_OUTPUT);
    gpioWrite(DHT11_PIN, 0);
    usleep(18000);
    gpioWrite(DHT11_PIN, 1);
    //waiting for response
    gpioSetMode(DHT11_PIN, PI_INPUT);
    for(i=0; i<100; i++){
        if(gpioRead(DHT11_PIN)==0) break;
        udelay(1);
    }
    for(i=0; i<100; i++){
        if(gpioRead(DHT11_PIN)==1) break;
        udelay(1);
    }
    // Read data
    DHT11_readOneByte(hum_h);
    DHT11_readOneByte(hum_l);
    DHT11_readOneByte(temp_h);

```

```

DHT11_readOneByte(temp_l);
DHT11_readOneByte(crc);
piHiPri(0);

// Check if data is valid
if((hum_h+hum_l+temp_h+temp_l)!=crc)
    return 0;

sprintf(tmp,"%u.%u",hum_h,hum_l);
data->humidity = atof(tmp);
sprintf(tmp,"%u.%u",temp_h,temp_l);
data->temp = atof(tmp);

return 1;
}

void gpio_stop(int sig){
    printf("User pressing CTRL-C");
    running = 0;
}

int piHiPri(const int pri){
    // Source code of this function by Gordon Henderson
    struct sched_param sched;
    memset(&sched,0,sizeof(sched));
    if(pri>sched_get_priority_max (SCHED_RR))
        sched.sched_priority = sched_get_priority_max (SCHED_RR);
    else
        sched.sched_priority = pri;
    return sched_setscheduler(0,SCHED_RR,&sched);
}

void udelay(const long us){
    // Delay without yielding process
    long st;
    long tdif;
    struct timespec tnow;

    clock_gettime(CLOCK_REALTIME,&tnow);
    st = tnow.tv_nsec;
    while(1){
        clock_gettime(CLOCK_REALTIME,&tnow);
        tdif = tnow.tv_nsec - st;
        if(tdif < 0) tdif += 1000000000;
        if(tdif > (us*1000)) break;
    }
}

```

จากโปรแกรมตัวอย่าง มีประเด็นที่น่าสนใจดังต่อไปนี้

- ฟังก์ชัน piHiPri() ที่สร้างขึ้น เพื่อยกระดับความสำคัญของโปรเซสหรือเธรด ซึ่งอาร์กิวเมนต์เป็นค่า priority มีค่าระหว่าง 0 ถึง 100 โดยค่า priority 0 เป็นระดับปกติที่โปรเซสหรือเธรดจะได้รับ ส่วน priority 100 คือค่าสูงสุด ทั้งนี้ Raspberry Pi OS จะแบ่งสัดส่วนเวลาจัดการกับโปรเซส/เธรดนี้มากขึ้นตามสัดส่วนค่าที่กำหนด

จากโปรแกรมตัวอย่าง เราจะเห็นว่าตอนเริ่มต้นอ่านค่าในฟังก์ชัน DHT11_read() เราจะยกระดับความสำคัญโดยใช้ piHiPri(50); เพื่อต้องการให้ระบบหันมาให้เวลากับโปรเซสของเราทำงานมากขึ้น ทั้งนี้เนื่องจากกลไกการทำ bit-banging ที่เราใช้ในการอ่านค่าจากพอร์ต GPIO จะเกิดความความคลาดเคลื่อน ที่เกิดจากการที่ระบบปฏิบัติการหันไปทำ context switch แบ่งเวลาให้โปรเซสอื่นมาทำงานในขณะช่วงเวลาสำคัญ ทั้งนี้ในโปรเซสของเรามีการใช้ usleep() เป็นระยะ ดังนั้นโปรเซสอื่นจะยังคงมีโอกาสดำเนินการทำงานในช่วงที่เราอคอยได้อยู่

- ฟังก์ชัน `udelay()` ที่เขียนขึ้นใหม่แทนฟังก์ชัน `usleep()` ของ GCC และนำไปใช้ในจุดที่ทำ bitbang เนื่องจาก `usleep()` จะเป็นการพักโปรเซสเซอร์หรือเธรดไว้ชั่วคราว และระบบปฏิบัติการจะกลับมาสั่งทำงานต่อเมื่อครบกำหนดเวลา แต่กลไกการกลับมาทำงานต่อนี้จะไม่แม่นยำมากนัก ซึ่งในการใช้งานปกติอาจจะไม่ค่อยมีปัญหาใดๆ แต่การทำ bitbang นั้นหากสัญญาณมีความถี่สูง จะมีความคลาดเคลื่อนสูงตามไปด้วย (สังเกตว่า ถ้าหันกลับไปใช้ `usleep()` ในจุดต่างๆ แทน `udelay()` จะพบว่าข้อมูลจะอ่านผิดพลาดบ่อยครั้งขึ้น)