Honeywell Linux Assessment

Sr Advanced Embedded Engr.

Generated by Doxygen 1.11.0

Chapter 1

Honeywell Linux Assessment Documentation

1.1 Introduction

The test consisted of creating an application for a temperature sensor connected to an embedded Linux development board.

DISCLAIMER:
As a candidate for Senior Advanced Embedded Engineer position I decided to carry
this project to the physical implementation, meaning that all the hardware and
software cited in here was tested in the loop by me (Mauricio Gutierrez).

1.1.1 Yocto with Poky for Raspberry Pi

Yocto is an open-source project that allows the creation of custom Linux systems for embedded devices. Poky is the reference build system in Yocto, providing the tools and metadata necessary to create Linux images. Using Yocto with Poky on a Raspberry Pi allows for the creation of a highly customized and optimized operating system for specific applications.

1.1.2 AHT10 Temperature Sensor

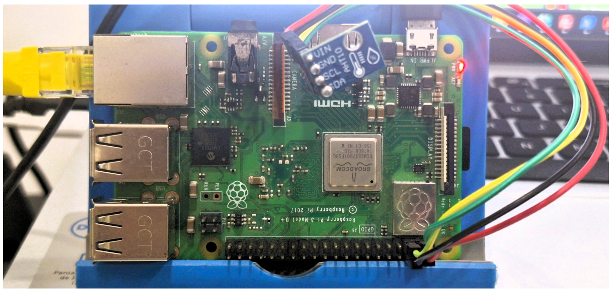
The AHT10 is a high-precision temperature and humidity sensor widely used in embedded applications. It offers a simple I2C interface for communication with microcontrollers and embedded systems. This project uses the AHT10 to measure ambient temperature and send this data over a TCP/IP network.

1.1.3 Client-Server Architectures in Embedded Systems

Client-server architectures are fundamental in data communication in embedded systems. In this project, an embedded server on the Raspberry Pi communicates with multiple clients to provide real-time temperature data. The server manages TCP connections and uses threads to handle client requests and data acquisition from the AHT10 sensor.

1.2 Hardware setup

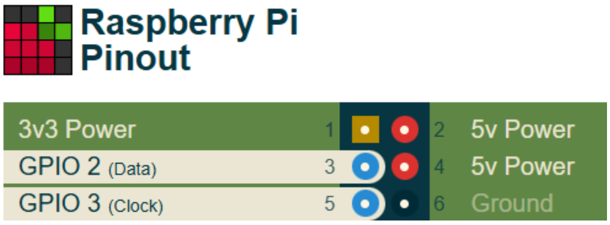
1.2.1 Step 1: I2C Connections



AHT10 temperature sensor has a I2C interface for communication with only 4 pins:

- VIN (5v Power)
- GND (Ground)
- SCL (GPIO 3 Clock)
- SDA (GPIO 2 Data)

Which can be directly connected to IC2 pins on Raspberry Pi 3B+ (model used for this project):

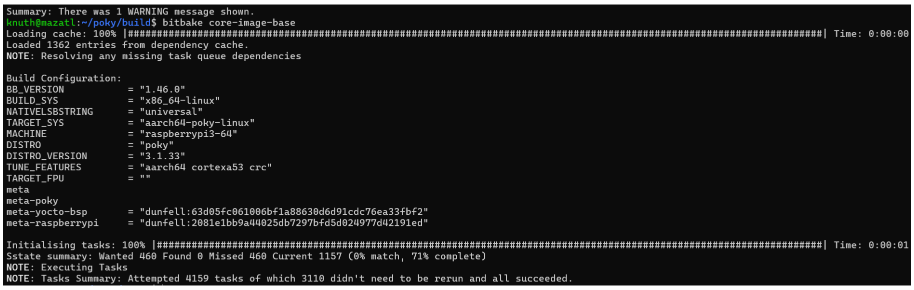


Raspberry Pi
Pinout

3v3 Power	1	■	●	2	5v Power
GPIO 2 (Data)	3	●	●	4	5v Power
GPIO 3 (Clock)	5	●	●	6	Ground

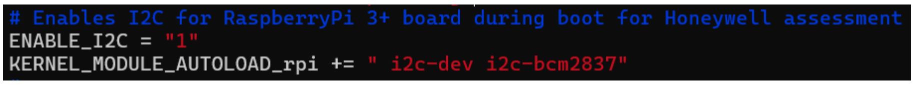
1.3 Software setup

We decided to use Poky *meta-raspberrypi layer* with Yocto (dunfell release) to build our custom image for Raspberry Pi 3 B+ (BCM2837 processor).



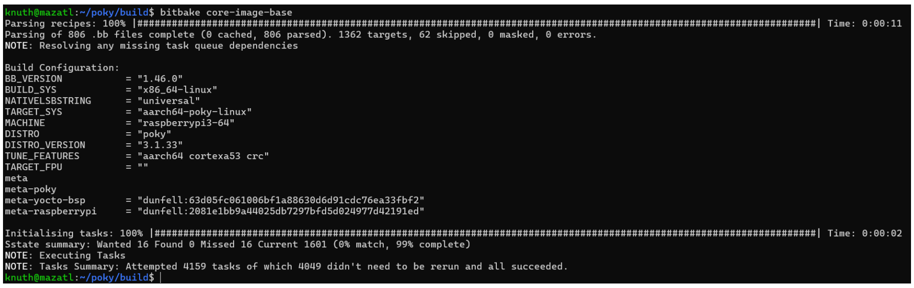
```
Summary: There was 1 WARNING message shown.
knuth@mazatl:~/poky/build$ bitbake core-image-base
Loading cache: 100% |##########################################################################################################| Time: 0:00:00
Loaded 1362 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "1.46.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "aarch64-poky-linux"
MACHINE              = "raspberrypi3-64"
DISTRO               = "poky"
DISTRO_VERSION       = "3.1.33"
TUNE_FEATURES        = "aarch64 cortexa53 crc"
TARGET_FPU           = ""
meta
meta-poky
meta-yocto-bsp       = "dunfell:63d05fc061006bf1a88630d6d91cdc76ea33fbf2"
meta-raspberrypi     = "dunfell:2081e1bb9a44025db7297bfd5d024977d42191ed"

Initialising tasks: 100% |#####################################################################################################| Time: 0:00:01
Sstate summary: Wanted 460 Found 0 Missed 460 Current 1157 (0% match, 71% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 4159 tasks of which 3110 didn't need to be rerun and all succeeded.
```

Adding i2d_dev to /etc/modules yields:
# modprobe -v -n -D i2c_dev
# insmod /lib/modules/5.4.72-v8/drivers/i2c/i2c-dev.ko
Editing the file poky/build/conf/local.conf file to configure enabling I2C during boot.



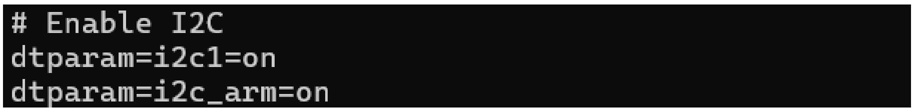
```
# Enables I2C for RaspberryPi 3+ board during boot for Honeywell assessment
ENABLE_I2C = "1"
KERNEL_MODULE_AUTOLOAD_rpi += " i2c-dev i2c-bcm2837"
```

Rebuiding Yocto:



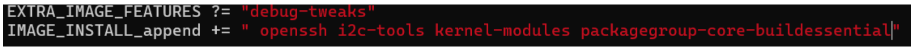
```
knuth@mazatl:~/poky/build$ bitbake core-image-base
Parsing recipes: 100% |######################################################################################################################| Time: 0:00:11
Parsing of 806 .bb files complete (0 cached, 806 parsed). 1362 targets, 62 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "1.46.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "aarch64-poky-linux"
MACHINE              = "raspberrypi3-64"
DISTRO               = "poky"
DISTRO_VERSION       = "3.1.33"
TUNE_FEATURES        = "aarch64 cortexa53 crc"
TARGET_FPU           = ""
meta
meta-poky
meta-yocto-bsp       = "dunfell:63d05fc061006bf1a88630d6d91cdc76ea33fbf2"
meta-raspberrypi     = "dunfell:2081e1bb9a44025db7297bfd5d024977d42191ed"

Initialising tasks: 100% |###################################################################################################################| Time: 0:00:02
Sstate summary: Wanted 16 Found 0 Missed 16 Current 1601 (0% match, 99% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 4159 tasks of which 4049 didn't need to be rerun and all succeeded.
knuth@mazatl:~/poky/build$ |
```

Output found at
/build/tmp/deploy/images/raspberrypi3-64/bootfiles/config.txt

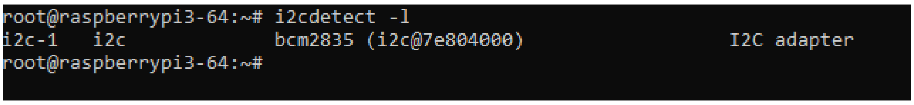


```
# Enable I2C
dtparam=i2c1=on
dtparam=i2c_arm=on
```

While I2C device is now present and correctly loaded at /dev/, we want the development tools to start programming by adding the following line in the Yocto configuration file.



```
EXTRA_IMAGE_FEATURES ?= "debug-tweaks"
IMAGE_INSTALL_append += " openssh i2c-tools kernel-modules packagegroup-core-buildessential"
```



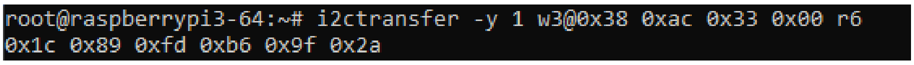
```
root@raspberrypi3-64:~# i2cdetect -l
i2c-1   i2c             bcm2835 (i2c@7e804000)                  I2C adapter
root@raspberrypi3-64:~#
```

We can see that now i2ctransfer is working by executing:
Handshaking:

- i2ctransfer -y 1 w3@0x38 0xE1 0x33 0x00 r6

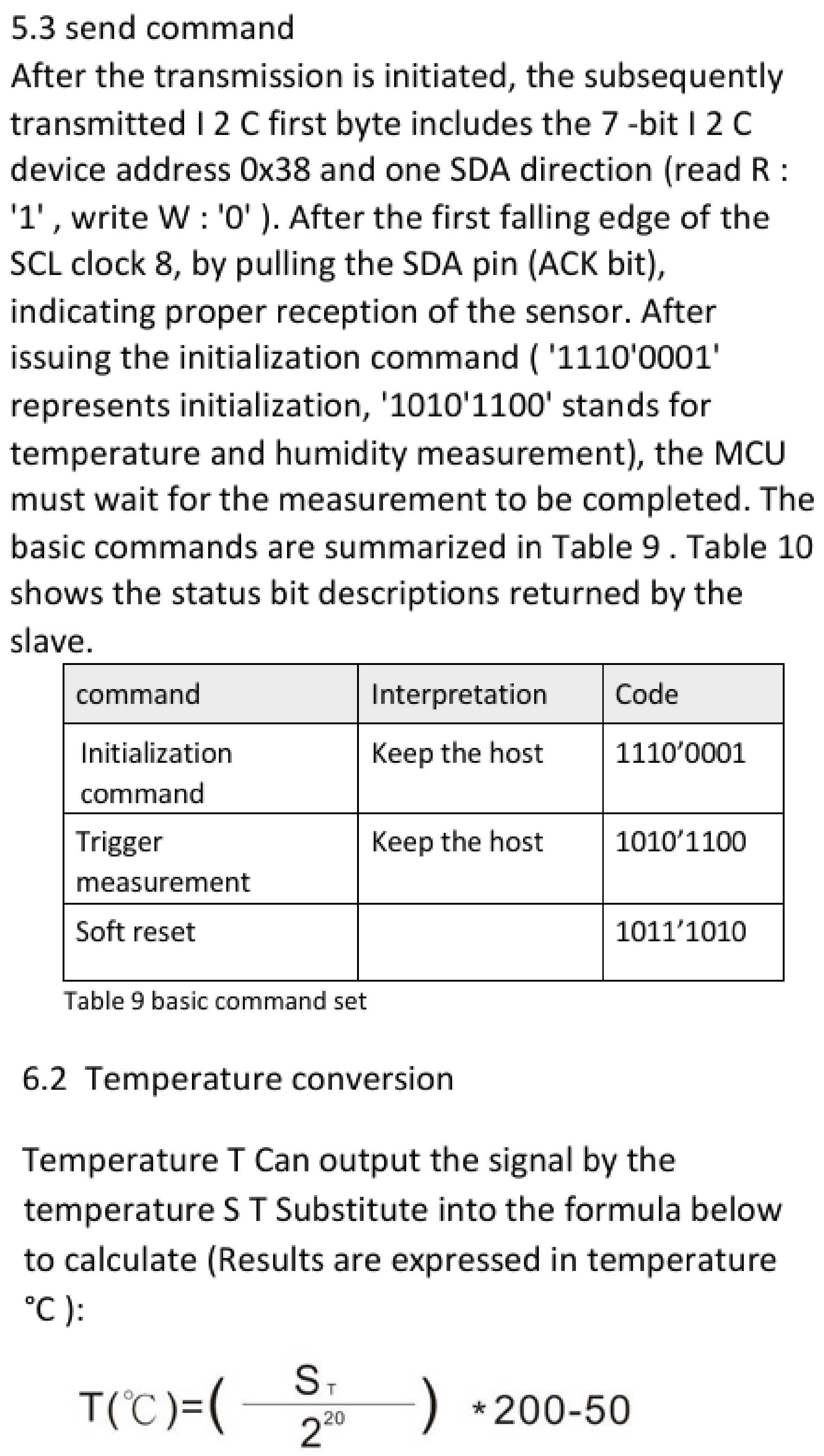
Data request:

- i2ctransfer -y 1 w3@0x38 0xac 0x33 0x00 r6



```
root@raspberrypi3-64:~# i2ctransfer -y 1 w3@0x38 0xac 0x33 0x00 r6
0x1c 0x89 0xfd 0xb6 0x9f 0x2a
```

Conditioning of the signal according to AHT10 datasheet:



5.3 send command

After the transmission is initiated, the subsequently transmitted I 2 C first byte includes the 7 -bit I 2 C device address 0x38 and one SDA direction (read R : '1' , write W : '0'). After the first falling edge of the SCL clock 8, by pulling the SDA pin (ACK bit), indicating proper reception of the sensor. After issuing the initialization command ('1110'0001' represents initialization, '1010'1100' stands for temperature and humidity measurement), the MCU must wait for the measurement to be completed. The basic commands are summarized in Table 9 . Table 10 shows the status bit descriptions returned by the slave.

command	Interpretation	Code
Initialization command	Keep the host	1110’0001
Trigger measurement	Keep the host	1010’1100
Soft reset		1011’1010

Table 9 basic command set

6.2 Temperature conversion

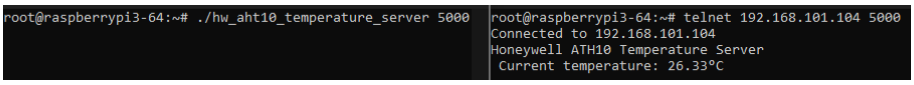
Temperature T Can output the signal by the temperature S T Substitute into the formula below to calculate (Results are expressed in temperature ℃):

$$T(℃)=\left(\frac{S_T}{2^{20}}\right)*200-50$$

After making threads for

- 1) data gathering,
- 2) data processing and
- 3) service requesting,

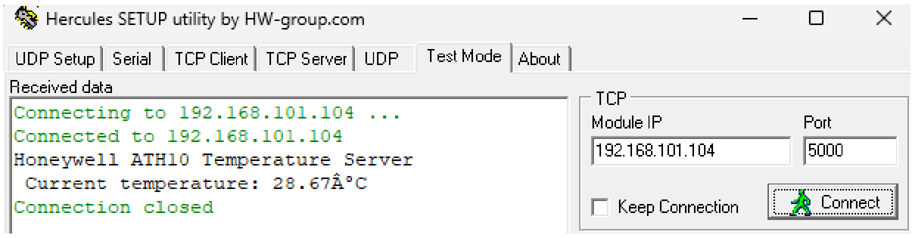
this is the console output:



```
root@raspberrypi3-64:~# ./hw_aht10_temperature_server 5000
```

```
root@raspberrypi3-64:~# telnet 192.168.101.104 5000
Connected to 192.168.101.104
Honeywell ATH10 Temperature Server
 Current temperature: 26.33°C
```

On the left side of the console, we can see the silent server working at port 5000. On the right we can see the request to the server by using telnet as client. We double checked the correct behavior by using another external client application called Hercules:



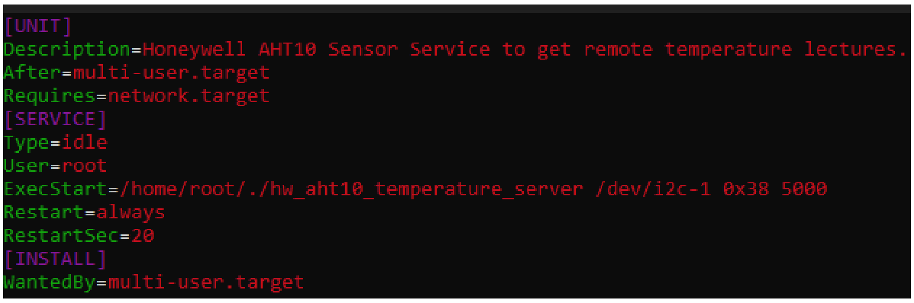
```
Connecting to 192.168.101.104 ...
Connected to 192.168.101.104
Honeywell ATH10 Temperature Server
 Current temperature: 28.67Â°C
Connection closed
```

Module IP: 192.168.101.104
Port: 5000
□ Keep Connection

For watchdog purposes, we need to add the package in Yocto and rebuild the image again with Systemd enbaled:

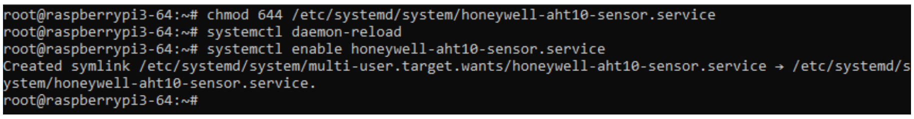
- DISTRO_FEATURES:append = " systemd"
- DISTRO_FEATURES_BACKFILL_CONSIDERED += "sysvinit"
- VIRTUAL-RUNTIME_init_manager = "systemd"
- VIRTUAL-RUNTIME_initscripts = "systemd-compat-units"
- INIT_MANAGER = "systemd"

A Systemd service was designed to automatically start the application after the user level (3) is reached. As per project requirements (5, 6), the application will be restarted after 20 seconds if it crashed, killed or closed, as shown below:



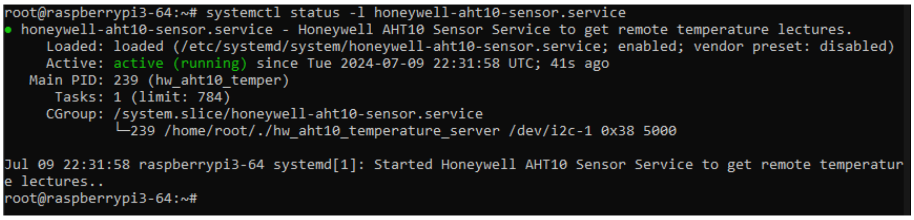
```
[UNIT]
Description=Honeywell AHT10 Sensor Service to get remote temperature lectures.
After=multi-user.target
Requires=network.target
[SERVICE]
Type=idle
User=root
ExecStart=/home/root/./hw_aht10_temperature_server /dev/i2c-1 0x38 5000
Restart=always
RestartSec=20
[INSTALL]
WantedBy=multi-user.target
```

–

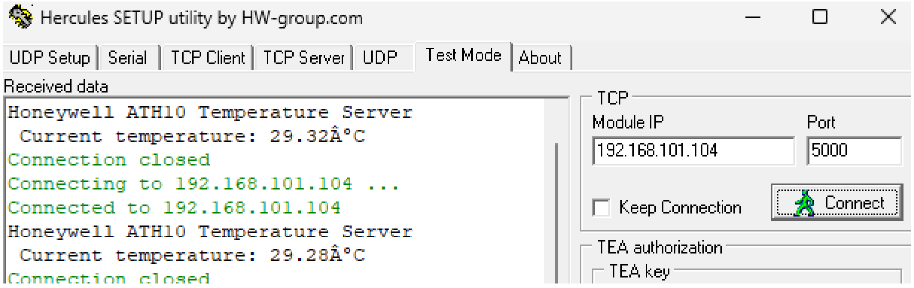


```
root@raspberrypi3-64:~# chmod 644 /etc/systemd/system/honeywell-aht10-sensor.service
root@raspberrypi3-64:~# systemctl daemon-reload
root@raspberrypi3-64:~# systemctl enable honeywell-aht10-sensor.service
Created symlink /etc/systemd/system/multi-user.target.wants/honeywell-aht10-sensor.service → /etc/systemd/s
ystem/honeywell-aht10-sensor.service.
root@raspberrypi3-64:~#
```

After reboot we can see the service is active and correctly working:



```
root@raspberrypi3-64:~# systemctl status -l honeywell-aht10-sensor.service
● honeywell-aht10-sensor.service - Honeywell AHT10 Sensor Service to get remote temperature lectures.
     Loaded: loaded (/etc/systemd/system/honeywell-aht10-sensor.service; enabled; vendor preset: disabled)
     Active: active (running) since Tue 2024-07-09 22:31:58 UTC; 41s ago
   Main PID: 239 (hw_aht10_temper)
      Tasks: 1 (limit: 784)
     CGroup: /system.slice/honeywell-aht10-sensor.service
             └─239 /home/root/./hw_aht10_temperature_server /dev/i2c-1 0x38 5000

Jul 09 22:31:58 raspberrypi3-64 systemd[1]: Started Honeywell AHT10 Sensor Service to get remote temperatur
e lectures..
root@raspberrypi3-64:~#
```



Module IP: 192.168.101.104
Port: 5000
□ Keep Connection
TEA key

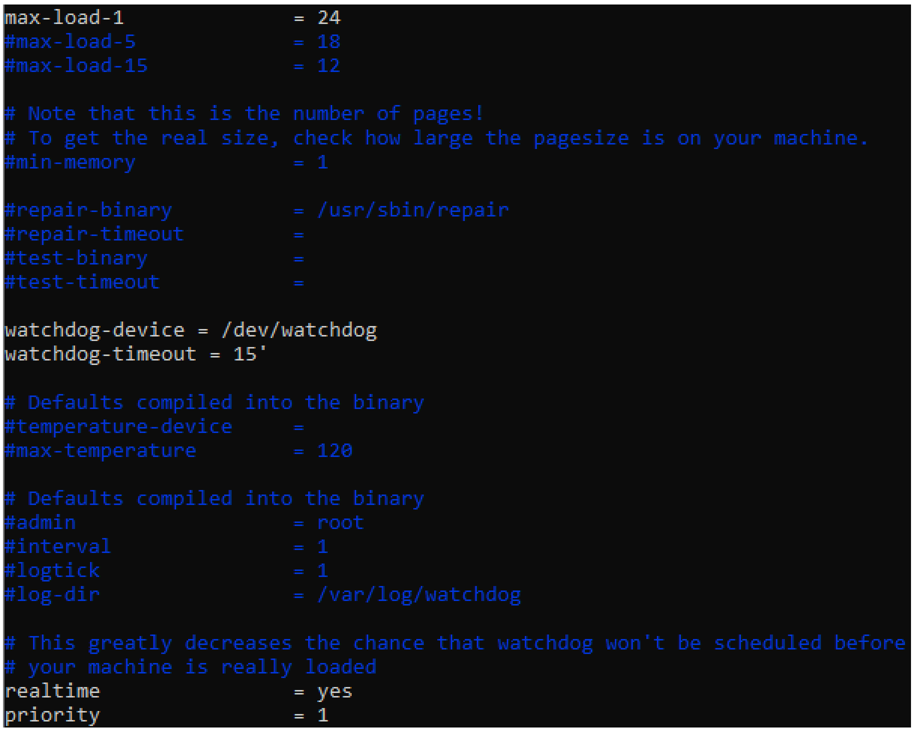
```
Honeywell ATH10 Temperature Server
 Current temperature: 29.32Â°C
Connection closed
Connecting to 192.168.101.104 ...
Connected to 192.168.101.104
Honeywell ATH10 Temperature Server
 Current temperature: 29.28Â°C
Connection closed
```

Finally, we configure the watchdog to monitor the "real-time" system. If the board load goes above the '24' parameter the system will automatically reboot.

- echo 'dtparam=watchdog=on' >> /boot/config.txt

If the system doesn't respond during over 15 seconds, the hardware watchdog signal will reboot the OS.

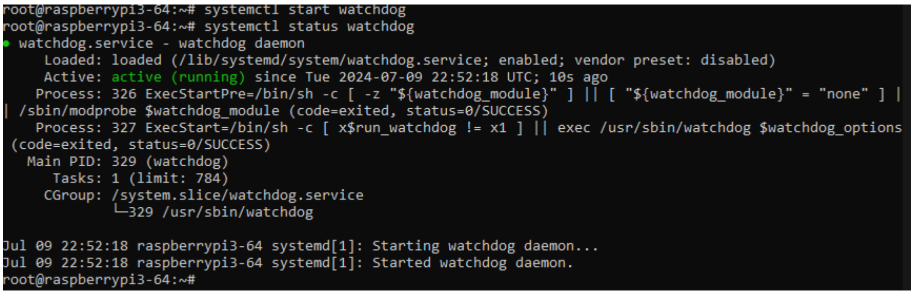
- vim /etc/watchdog.conf



```
max-load-1              = 24
#max-load-5             = 18
#max-load-15            = 12

# Note that this is the number of pages!
# To get the real size, check how large the pagesize is on your machine.
#min-memory             = 1

#repair-binary          = /usr/sbin/repair
#repair-timeout         =
#test-binary            =
#test-timeout           =

watchdog-device = /dev/watchdog
watchdog-timeout = 15'

# Defaults compiled into the binary
#temperature-device     =
#max-temperature        = 120

# Defaults compiled into the binary
#admin                  = root
#interval               = 1
#logtick                = 1
#log-dir                = /var/log/watchdog

# This greatly decreases the chance that watchdog won't be scheduled before
# your machine is really loaded
realtime                = yes
priority                = 1
```

- systemctl enable watchdog

After correctly configuring the board's watchdog we can enable the WD service via Systemd, as showed below:

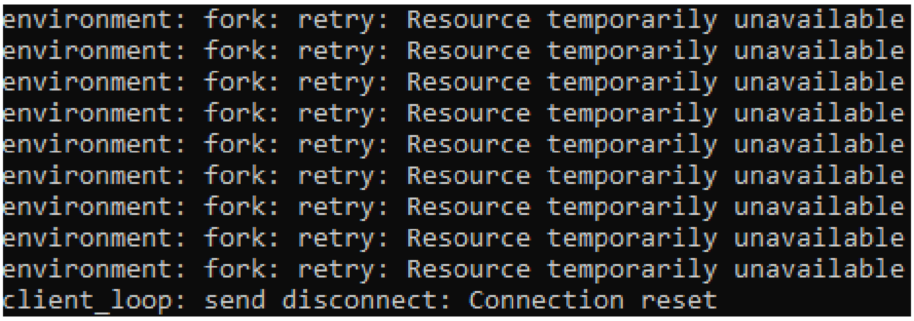


```
root@raspberrypi3-64:~# systemctl start watchdog
root@raspberrypi3-64:~# systemctl status watchdog
● watchdog.service - watchdog daemon
     Loaded: loaded (/lib/systemd/system/watchdog.service; enabled; vendor preset: disabled)
     Active: active (running) since Tue 2024-07-09 22:52:18 UTC; 10s ago
    Process: 326 ExecStartPre=/bin/sh -c [ -z "${watchdog_module}" ] || [ "${watchdog_module}" = "none" ] |
| /sbin/modprobe $watchdog_module (code=exited, status=0/SUCCESS)
    Process: 327 ExecStart=/bin/sh -c [ x$run_watchdog != x1 ] || exec /usr/sbin/watchdog $watchdog_options
 (code=exited, status=0/SUCCESS)
   Main PID: 329 (watchdog)
      Tasks: 1 (limit: 784)
     CGroup: /system.slice/watchdog.service
             └─329 /usr/sbin/watchdog

Jul 09 22:52:18 raspberrypi3-64 systemd[1]: Starting watchdog daemon...
Jul 09 22:52:18 raspberrypi3-64 systemd[1]: Started watchdog daemon.
root@raspberrypi3-64:~#
```

We ran the famous fork bomb to test it:

- bash -c ':(){ :|:& };:'

After the fork bomb exploited the system frozen during 15 seconds proximately and the automatically restarted.



```
environment: fork: retry: Resource temporarily unavailable
environment: fork: retry: Resource temporarily unavailable
environment: fork: retry: Resource temporarily unavailable
environment: fork: retry: Resource temporarily unavailable
environment: fork: retry: Resource temporarily unavailable
environment: fork: retry: Resource temporarily unavailable
environment: fork: retry: Resource temporarily unavailable
environment: fork: retry: Resource temporarily unavailable
environment: fork: retry: Resource temporarily unavailable
client_loop: send disconnect: Connection reset
```

This covers the 6th and last requirement from the Honeywell Linux Assessment.

1.4 Software compilation method

For this project (assessment) we designed, developed and built 2 applications:

- A HAL application that is able to call from user space the I2C kernel functions, and
- A TCP multithreading server that gets sensor data, transform data to a lecture and attend incoming requests for temperature service subscribers.

Since Raspberry Pi 3B+ board has enough computational resources, we decided to do not crosscompile but compile the sources directly in the board.
In this regard, the command to compile is:
for HAL application:

- *gcc hw_aht10_get_temp.c - o hw_aht10_get_temp*

for TCP server application:

- *gcc -pthread hw_aht10_temperature_server.c hw_aht10_func.c -o hw_aht10_temperature_server*

1.4.1 Running the app

Both applications can run with parameter, however for the HAL application the parameters are madatory:

- Usage: ./hw_aht10_get_temp device_name device_address
- e.g.: ./hw_aht10_get_temp /dev/i2c-1 0x38

This allows the possibility the change/update the sensor without changing the source code.
On the other hand, the TCP server application can run with or without parameters.

- Usage: ./hw_aht10_temperature_server device_name device_address port
- e.g. (default) ./hw_aht10_temperature_server /dev/i2c-1 0x38 5000

This means that the default values are the I2C descriptor 1, the default addres for AHT10 sensor and TCP Port 5000.
NOTE: As we previously showed, is not necessary to manually run the server since it is automatically started and monitored via software (systemd service) and hardware (watchdog).

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 SensorInfo Struct Reference

```
#include <hw_aht10_func.h>
```

Data Fields

- char dev_name [MAX_SIZE]
- int dev_address
- char data [6]

4.1.1 Detailed Description

Definition at line 6 of file hw_aht10_func.h.

4.1.2 Field Documentation

4.1.2.1 data

```
char data[6]
```

Definition at line 9 of file hw_aht10_func.h.

4.1.2.2 dev_address

```
int dev_address
```

Definition at line 8 of file hw_aht10_func.h.

4.1.2.3 dev_name

```
char dev_name[MAX_SIZE]
```

Definition at line 7 of file hw_aht10_func.h.

The documentation for this struct was generated from the following file:

- hw_aht10_func.h

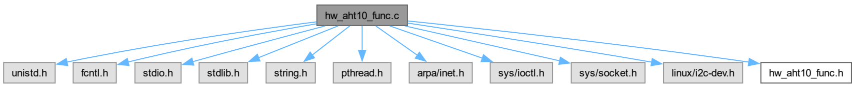
Chapter 5

File Documentation

5.1 hw_aht10_func.c File Reference

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <linux/i2c-dev.h>
#include "hw_aht10_func.h"
```

Include dependency graph for hw_aht10_func.c:



Functions

- int validate_inputs (int argc, char ∗∗argv)
- char ∗ get_data_from_sensor (char ∗device_name, int dev_addr)
- float compute_temperature_celsius (char ∗data)
- int start_listener (unsigned short port)
- void ∗ read_sensor (void ∗arg)
- void ∗ compute_temperature (void ∗arg)
- void send_message (int client_socket)
- void ∗ connection_handler (void ∗socket_desc)

Variables

- pthread_mutex_t mutex
- pthread_cond_t cond
- int data_ready = e_gathering
- float temperature

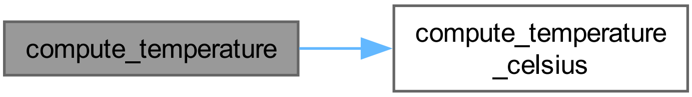
5.1.1 Function Documentation

5.1.1.1 compute_temperature()

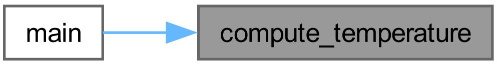
```
void * compute_temperature (
              void * arg)
```

Definition at line 152 of file hw_aht10_func.c.

Here is the call graph for this function:



Here is the caller graph for this function:

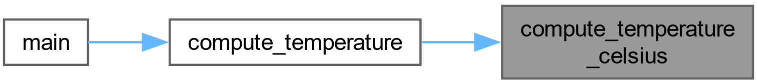


5.1.1.2 compute_temperature_celsius()

```
float compute_temperature_celsius (
              char * data)
```

Definition at line 103 of file hw_aht10_func.c.

Here is the caller graph for this function:

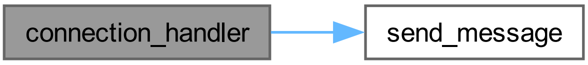


5.1.1.3 connection_handler()

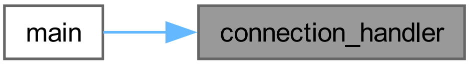
```
void * connection_handler (
              void * socket_desc)
```

Definition at line 175 of file hw_aht10_func.c.

Here is the call graph for this function:



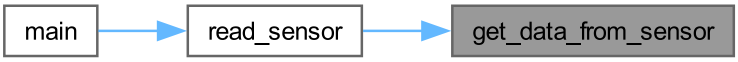
Here is the caller graph for this function:



5.1.1.4 get_data_from_sensor()

```
char * get_data_from_sensor (
              char * device_name,
              int dev_addr)
```

Definition at line 41 of file hw_aht10_func.c.
Here is the caller graph for this function:

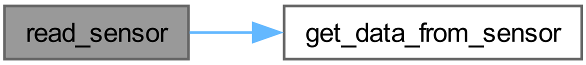


5.1.1.5 read_sensor()

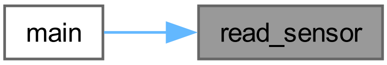
```
void * read_sensor (
              void * arg)
```

Definition at line 136 of file hw_aht10_func.c.

Here is the call graph for this function:



Here is the caller graph for this function:

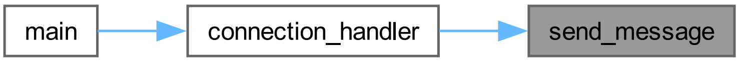


5.1.1.6 send_message()

```
void send_message (
             int client_socket)
```

Definition at line 168 of file hw_aht10_func.c.

Here is the caller graph for this function:

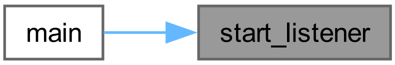


5.1.1.7 start_listener()

```
int start_listener (
             unsigned short port)
```

Definition at line 111 of file hw_aht10_func.c.

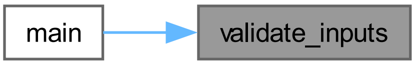
Here is the caller graph for this function:



5.1.1.8 validate_inputs()

```
int validate_inputs (
             int argc,
             char ** argv)
```

Definition at line 19 of file hw_aht10_func.c.
Here is the caller graph for this function:



5.1.2 Variable Documentation

5.1.2.1 cond

```
pthread_cond_t cond
```

Definition at line 15 of file hw_aht10_func.c.

5.1.2.2 data_ready

```
int data_ready = e_gathering
```

Definition at line 16 of file hw_aht10_func.c.

5.1.2.3 mutex

```
pthread_mutex_t mutex
```

Definition at line 14 of file hw_aht10_func.c.

5.1.2.4 temperature

```
float temperature
```

Definition at line 17 of file hw_aht10_func.c.

5.2 hw_aht10_func.c

Go to the documentation of this file.

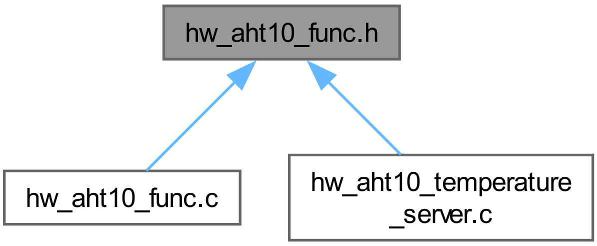
```
00001 #include <unistd.h>
00002 #include <fcntl.h>
00003 #include <stdio.h>
00004 #include <stdlib.h>
00005 #include <string.h>
00006 #include <pthread.h>
00007 #include <arpa/inet.h>
00008 #include <sys/ioctl.h>
00009 #include <sys/socket.h>
00010 #include <linux/i2c-dev.h>
00011 #include "hw_aht10_func.h"
00012
00013 /* Shared data and sync for threads*/
00014 pthread_mutex_t mutex;
00015 pthread_cond_t cond;
00016 int data_ready = e_gathering;
00017 float temperature;
00018
00019 int validate_inputs(int argc, char **argv) {
00020     char *cp, *program_name = argv[0];
00021
00022     // skip over program name
00023     --argc;
```

```
    ++argv;

    if (argc < 3) {
        fprintf(stderr,"\n WARNING %s: not (all) arguments specified", program_name);
        fprintf(stderr,"\n -------> starting server with default value!\n\n", program_name);
        return 0;
    }

    cp = *argv;
    if (*cp == 0) {
        fprintf(stderr,"\n WARNING %s: starting server with default values!\n", program_name);
        return 0;
    }

    return 1;
}

char * get_data_from_sensor(char *device_name, int dev_addr) {
    static unsigned char data[6] = {0};
    int i2c_handler, length, comp_temp;
    float temperature_celsius;

    /* Get I2C handler */
    if ((i2c_handler = open(device_name, O_RDWR)) < 0)
    {
        /* Error getting the file descriptor from device */
        printf("Unable to open I2C device");
        return data;
    }
    /* Calling kernel layer from user space to connect with I2C device */
    if (ioctl(i2c_handler, I2C_SLAVE, dev_addr) < 0)
    {
        /* Bus access failed */
        printf("Unable to connect to low-level device I2C.\n");
        return data;
    }

    /* Handshake (initialization command) according AHT10 sensor datasheet */
    data[0] = 0xE1;
    data[1] = 0x08;
    data[2] = 0x00;
    length = 3;
    /* Send initialization command to I2C device */
    if (write(i2c_handler, data, length) != length)
    {
        /* Unable to handshake with i2c device */
        printf("Unable to handshake the i2c bus.\n");
    }

    /* Wait 20ms before proceeding with the next write command */
    sleep(0.02);

    /* Trigger measurement (request command) acc. AHT10 sensor datasheet */
    data[0] = 0xAC;
    data[1] = 0x33;
    data[2] = 0x00;
    length = 3;
    /* Send trigger measurement command to I2C device */
    if (write(i2c_handler, data, length) != length)
    {
        /* Unable to request data from I2C device */
        printf("Unable to request data from the I2C bus.\n");
    }

    /* Wait 20ms before proceeding with the next read command */
    sleep(0.02);

    /* Read the 6 bytes answer from I2C */
    length = 6;
    if (read(i2c_handler, data, length) != length)
    {
        /* Unable to get data from I2C device */
        printf("Unable to read output temp. from the i2c AHT10 sensor.\n");
    }
    else {
        return data;
    }
}

float compute_temperature_celsius(char *data) {
    int comp_temp;
    float temperature_celsius;
    comp_temp = ((data[3] & 0x0F) « 16) | (data[4] « 8) | data[5];
    temperature_celsius = ((comp_temp*200.0)/1048576) -50.0;
    return temperature_celsius;
}

```

```
int start_listener(unsigned short port) {
    int socket_desc;
    struct sockaddr_in server;

    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc == -1)
    {
        printf("Unable to create socket.");
    }

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( port );

    if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0)
    {
        perror("Unable to bind socket.");
        return 1;
    }

    listen(socket_desc , MAX_SIZE);

    return socket_desc;
}

void *read_sensor(void *arg) {
    SensorInfo *sensor_info = (SensorInfo *)arg;

    char *data = get_data_from_sensor(sensor_info->dev_name, sensor_info->dev_address);

    pthread_mutex_lock(&mutex);
    for (int i = 0; i < 6; ++i) {
        sensor_info->data[i] = data[i];
    }
    data_ready = e_computing;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);

    return NULL;
}

void *compute_temperature(void *arg) {
    SensorInfo *sensor_info = (SensorInfo *)arg;

    pthread_mutex_lock(&mutex);
    while( data_ready != e_computing ) {
        pthread_cond_wait(&cond, &mutex);
    }

    temperature = compute_temperature_celsius(sensor_info->data);

    data_ready = e_ready;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    return NULL;
}

void send_message(int client_socket) {
    char message[MAX_SIZE];

    sprintf(message, "Honeywell ATH10 Temperature Server\n Current temperature: %.2f°C\n", temperature);
    write(client_socket , message , strlen(message));
}

void *connection_handler(void *socket_desc) {
    int client_socket = *(int*)socket_desc;

    pthread_mutex_lock(&mutex);
    while (data_ready != e_ready) {
        pthread_cond_wait(&cond, &mutex);
    }
    send_message(client_socket);
    data_ready = e_gathering;

    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);

    return NULL;
}
```

5.3 hw_aht10_func.h File Reference

This graph shows which files directly or indirectly include this file:



Data Structures

- struct SensorInfo

Macros

- #define MAX_SIZE 256

Enumerations

- enum threads_states { e_gathering , e_computing , e_ready }

Functions

- int validate_inputs (int, char ∗∗)
- int start_listener (unsigned short)
- char ∗ get_data_from_sensor (char ∗, int)
- float compute_temperature_celsius (char ∗)
- void send_message (int)
- void ∗ read_sensor (void ∗)
- void ∗ compute_temperature (void ∗)
- void ∗ connection_handler (void ∗)

Variables

- SensorInfo sensor_info
- pthread_mutex_t mutex
- pthread_cond_t cond

5.3.1 Macro Definition Documentation

5.3.1.1 MAX_SIZE

```
#define MAX_SIZE 256
```

Definition at line 4 of file hw_aht10_func.h.

5.3.2 Enumeration Type Documentation

5.3.2.1 threads_states

```
enum threads_states
```

Enumerator

e_gathering	
e_computing	
e_ready	

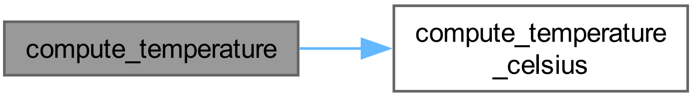
Definition at line 16 of file hw_aht10_func.h.

5.3.3 Function Documentation

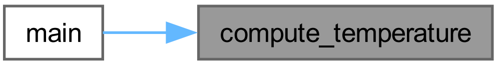
5.3.3.1 compute_temperature()

```
void * compute_temperature (
            void * arg)
```

Definition at line 152 of file hw_aht10_func.c.
Here is the call graph for this function:



Here is the caller graph for this function:

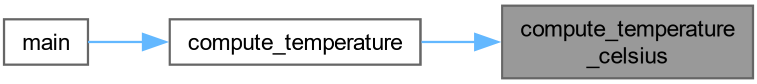


5.3.3.2 compute_temperature_celsius()

```
float compute_temperature_celsius (
            char * data)
```

Definition at line 103 of file hw_aht10_func.c.

Here is the caller graph for this function:

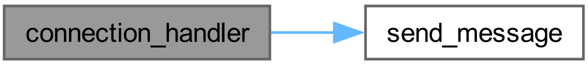


5.3.3.3 connection_handler()

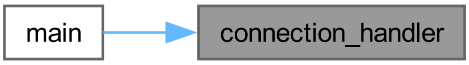
```
void * connection_handler (
            void * socket_desc)
```

Definition at line 175 of file hw_aht10_func.c.

Here is the call graph for this function:



Here is the caller graph for this function:

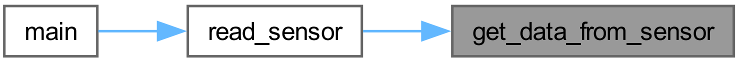


5.3.3.4 get_data_from_sensor()

```
char * get_data_from_sensor (
            char * device_name,
            int dev_addr)
```

Definition at line 41 of file hw_aht10_func.c.

Here is the caller graph for this function:

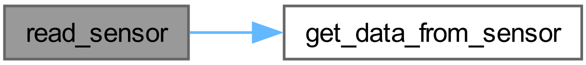


5.3.3.5 read_sensor()

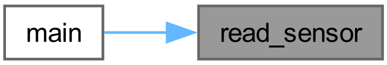
```
void * read_sensor (
             void * arg)
```

Definition at line 136 of file hw_aht10_func.c.

Here is the call graph for this function:



Here is the caller graph for this function:

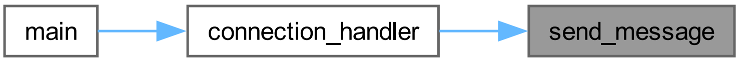


5.3.3.6 send_message()

```
void send_message (
             int client_socket)
```

Definition at line 168 of file hw_aht10_func.c.

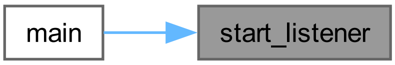
Here is the caller graph for this function:



5.3.3.7 start_listener()

```
int start_listener (
             unsigned short port)
```

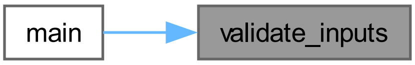
Definition at line 111 of file hw_aht10_func.c.
Here is the caller graph for this function:



5.3.3.8 validate_inputs()

```
int validate_inputs (
             int argc,
             char ** argv)
```

Definition at line 19 of file hw_aht10_func.c.
Here is the caller graph for this function:



5.3.4 Variable Documentation

5.3.4.1 cond

```
pthread_cond_t cond [extern]
```

Condition variable for thread synchronization (prevents race conditions)
Definition at line 15 of file hw_aht10_func.c.

5.3.4.2 mutex

```
pthread_mutex_t mutex [extern]
```

Mutex for thread synchronization (prevents deadlocks)
Definition at line 14 of file hw_aht10_func.c.

5.3.4.3 sensor_info

```
SensorInfo sensor_info [extern]
```

Sensor information structure
Definition at line 12 of file hw_aht10_temperature_server.c.

5.4 hw_aht10_func.h

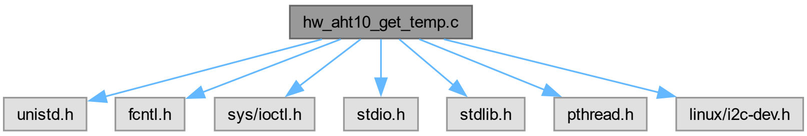
Go to the documentation of this file.

```
#ifndef HONEYWELL_ASSESSMENT_HW_AHT10_FUNC_H
#define HONEYWELL_ASSESSMENT_HW_AHT10_FUNC_H

#define MAX_SIZE 256

typedef struct {
    char dev_name[MAX_SIZE];
    int dev_address;
    char data[6];
} SensorInfo;

extern SensorInfo sensor_info;
extern pthread_mutex_t mutex;
extern pthread_cond_t cond;

typedef enum  {e_gathering, e_computing, e_ready} threads_states;

int validate_inputs(int, char **);
int start_listener(unsigned short);
char * get_data_from_sensor(char *, int);
float compute_temperature_celsius(char *);
void send_message(int);
void *read_sensor(void *);
void *compute_temperature(void *);
void *connection_handler(void *);

#endif //HONEYWELL_ASSESSMENT_HW_AHT10_FUNC_H
```

5.5 hw_aht10_get_temp.c File Reference

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <linux/i2c-dev.h>
```

Include dependency graph for hw_aht10_get_temp.c:



Functions

- int validate_inputs (int, char ∗∗)
- float get_temperature_celsius (char ∗, int)
- int main (int argc, char ∗∗argv)

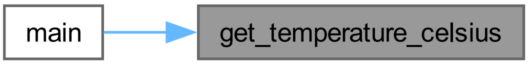
5.5.1 Function Documentation

5.5.1.1 get_temperature_celsius()

```
float get_temperature_celsius (
             char * device_name,
             int dev_addr)
```

Definition at line 47 of file hw_aht10_get_temp.c.

Here is the caller graph for this function:

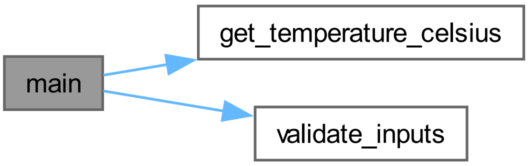


5.5.1.2 main()

```
int main (
            int argc,
            char ** argv)
```

Definition at line 12 of file hw_aht10_get_temp.c.

Here is the call graph for this function:

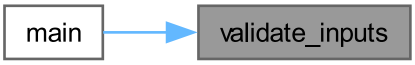


5.5.1.3 validate_inputs()

```
int validate_inputs (
            int argc,
            char ** argv)
```

Definition at line 26 of file hw_aht10_get_temp.c.

Here is the caller graph for this function:



5.6 hw_aht10_get_temp.c

Go to the documentation of this file.

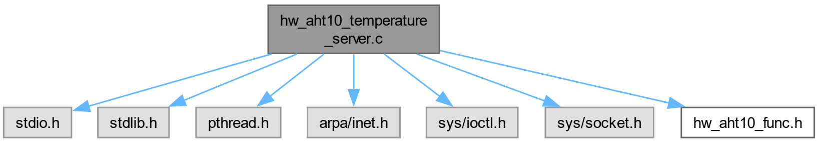
```
00001 #include <unistd.h>
```

```c
#include <fcntl.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <linux/i2c-dev.h>

int validate_inputs(int, char **);
float get_temperature_celsius(char *, int);

int main(int argc, char **argv) {
    /*char *device_name = (char*)"/dev/i2c-1";
    int dev_addr = 0x38;*/
    if( validate_inputs(argc, argv) == 1) {
        char *device_name = argv[1];
        int dev_addr = (int)strtol(argv[2], NULL, 16);
    printf("%0.2f\n", get_temperature_celsius(device_name, dev_addr));
    }
    else {
        printf("\n Usage: %s device_name device address\n\n", argv[0]);
    }
    return 0;
}

int validate_inputs(int argc, char **argv) {
    char *cp, *program_name = argv[0];

    // skip over program name
    --argc;
    ++argv;

    if (argc < 2) {
        fprintf(stderr,"\n %s: not (all) arguments specified\n", program_name);
        return -1;
    }

    cp = *argv;
    if (*cp == 0) {
        fprintf(stderr,"\n %s: argument an empty string\n", program_name);
        return -1;
    }

    return 1;
}

float get_temperature_celsius(char *device_name, int dev_addr) {
    unsigned char data[6] = {0};
    int i2c_handler, length, comp_temp;
    float temperature_celsius;

    /* Get I2C handler */
    if ((i2c_handler = open(device_name, O_RDWR)) < 0)
    {
        /* Error getting the file descriptor from device */
        printf("Unable to open I2C device");
        return -1;
    }
    /* Calling kernel layer from user space to connect with I2C device */
    if (ioctl(i2c_handler, I2C_SLAVE, dev_addr) < 0)
    {
        /* Bus access failed */
        printf("Unable to connect to low-level device I2C.\n");
        return -1;
    }

    /* Handshake (initialization command) according AHT10 sensor datasheet */
    data[0] = 0xE1;
    data[1] = 0x08;
    data[2] = 0x00;
    length = 3;
    /* Send initialization command to I2C device */
    if (write(i2c_handler, data, length) != length)
    {
        /* Unable to handshake with i2c device */
        printf("Unable to handshake the i2c bus.\n");
    }

    /* Wait 20ms before proceeding with the next write command */
    sleep(0.02);

    /* Trigger measurement (request command) acc. AHT10 sensor datasheet */
    data[0] = 0xAC;
    data[1] = 0x33;
    data[2] = 0x00;
    length = 3;
    /* Send trigger measurement command to I2C device */
    if (write(i2c_handler, data, length) != length)
    {
        /* Unable to request data from I2C device */
        printf("Unable to request data from the I2C bus.\n");
    }

    /* Wait 20ms before proceeding with the next read command */
    sleep(0.02);

    /* Read the 6 bytes answer from I2C */
    length = 6;
    if (read(i2c_handler, data, length) != length)
    {
        /* Unable to get data from I2C device */
        printf("Unable to read output temp. from the i2c AHT10 sensor.\n");
    }
    else
    {
        comp_temp = ((data[3] & 0x0F) « 16) | (data[4] « 8) | data[5];
        temperature_celsius = ((comp_temp*200.0)/1048576) -50.0;
    }
    return temperature_celsius;
}
```

5.7 hw_aht10_temperature_server.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include "hw_aht10_func.h"
```

Include dependency graph for hw_aht10_temperature_server.c:



Functions

- int main (int argc, char **argv)

  In this section the TCP server binds connections with clients using threads for:

  .

Variables

- pthread_mutex_t mutex
- pthread_cond_t cond
- SensorInfo sensor_info

5.7.1 Function Documentation

5.7.1.1 main()

```
int main (
        int argc,
        char ** argv)
```

In this section the TCP server binds connections with clients using threads for:

.

- 1) Getting data from AHT10 sensor
- 2) Computing the current temperature, and
- 3) Attending new temperature requests via TCP.

Parameters

argc	Argument count
argv	Argument vector

Returns

Exit status

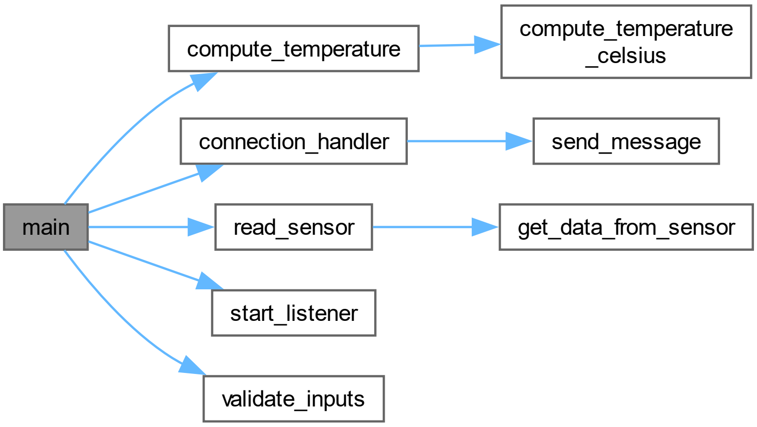
TCP port is 5000 by default
Default values for I2C hardware:

- The device name by default is the file descriptor we found at /dev/i2c-1 and,
- The device address for the AHT10 sensor is 0x38.

NOTE: These values can be modified according to the connected sensor and the board when executing the program via the parameter *argv*.
Main loop:

- Once a client connection is accepted, three threads are created as mentioned above including the mutex and cond sections for **avoiding race conditions and/or deadlocks**.

Definition at line 26 of file hw_aht10_temperature_server.c.
Here is the call graph for this function:



5.7.2 Variable Documentation

5.7.2.1 cond

```
pthread_cond_t cond
```

Condition variable for thread synchronization (prevents race conditions)
Definition at line 11 of file hw_aht10_temperature_server.c.

5.7.2.2 mutex

`pthread_mutex_t mutex`

Mutex for thread synchronization (prevents deadlocks)

Definition at line 10 of file hw_aht10_temperature_server.c.

5.7.2.3 sensor_info

`SensorInfo sensor_info`

Sensor information structure

Definition at line 12 of file hw_aht10_temperature_server.c.

5.8 hw_aht10_temperature_server.c

Go to the documentation of this file.

```
00001
00002 #include <stdio.h>
00003 #include <stdlib.h>
00004 #include <pthread.h>
00005 #include <arpa/inet.h>
00006 #include <sys/ioctl.h>
00007 #include <sys/socket.h>
00008 #include "hw_aht10_func.h"
00009
00010 pthread_mutex_t mutex;
00011 pthread_cond_t cond;
00012 SensorInfo sensor_info;
00026 int main(int argc, char **argv) {
00027     int socket_desc, client_sock;
00029     unsigned short server_port = 5000;
00030     struct sockaddr_in client_addr;
00031     socklen_t client_len;
00032
00041     sprintf(sensor_info.dev_name,"/dev/i2c-1");
00042     sensor_info.dev_address = 0x38;
00043
00044     if (validate_inputs(argc, argv) == 1) {
00045         sprintf(sensor_info.dev_name,"%s", argv[1]);
00046         sensor_info.dev_address = (int)strtol(argv[2], NULL, 16);
00047         server_port = (int)strtol(argv[3], NULL, 10);
00048     }
00049
00050     if ((socket_desc = start_listener(server_port)) < 0) {
00051         printf("Unable to open socket at %d", server_port);
00052         exit(-1);
00053     }
00060     while ((client_sock = accept(socket_desc,
00061                                  (struct sockaddr *)&client_addr,
00062                                  (socklen_t*)&client_len))) {
00063         pthread_t thread1, thread2, thread3;
00064
00065         pthread_mutex_init(&mutex, NULL);
00066         pthread_cond_init(&cond, NULL);
00067
00068         pthread_create(&thread1, NULL, read_sensor, &sensor_info);
00069         pthread_create(&thread2, NULL, compute_temperature, &sensor_info);
00070         pthread_create(&thread3, NULL, connection_handler, &client_sock);
00071
00072         pthread_join(thread1, NULL);
00073         pthread_join(thread2, NULL);
00074         pthread_join(thread3, NULL);
00075
00076         pthread_mutex_destroy(&mutex);
00077         pthread_cond_destroy(&cond);
00078     }
00079     return 0;
00080 }
```

5.9 mainpage.dox File Reference

Index