

**PENGEMBANGAN FIRMWARE AIR QUALITY MONITORING  
MENGGUNAKAN SENSOR SENSIIRION SPS30 DAN MCU STM32  
DENGAN PROTOKOL KOMUNIKASI I2C**

SKRIPSI



**Disusun oleh:**

**I MADE PRADHESTA SURYA WIBAWA**  
**19/443582/TK/48778**

**PROGRAM STUDI TEKNIK ELEKTRO  
DEPARTEMEN TEKNIK ELEKTRO DAN TEKNOLOGI INFORMASI  
FAKULTAS TEKNIK UNIVERSITAS GADJAH MADA  
YOGYAKARTA  
2023**

## HALAMAN PENGESAHAN

# PENGEMBANGAN *FIRMWARE AIR QUALITY MONITORING MENGGUNAKAN SENSOR SENSIRION SPS30 DAN MCU STM32 DENGAN PROTOKOL KOMUNIKASI I2C*

## SKRIPSI

Diajukan sebagai Salah Satu Syarat untuk Memperoleh

Gelar Sarjana Teknik

pada Departemen Teknik Elektro dan Teknologi Informasi Fakultas Teknik  
Universitas Gadjah Mada

Disusun oleh :

I Made Pradhesta Surya Wibawa  
19/443582/TK/48778

Telah disetujui dan disahkan

pada tanggal, 12 Juni 2023

Dosen Pembimbing I



Dr. I Wayan Mustika, S.T., M.Eng.  
NIP. 198109212014041001

Dosen Pembimbing II



Ir. Agus Bejo, S.T., M.Eng., D.Eng., IPM.  
NIP. 198001012015041002



**SKRIPSI**

**PENGEMBANGAN FIRMWARE AIR QUALITY MONITORING MENGGUNAKAN  
SENSOR SENSISSION SPS30 DAN MCU STM32 DENGAN PROTOKOL KOMUNIKASI  
I2C**

Dipersiapkan dan disusun oleh

**I Made Pradhesta Surya Wibawa**  
19/443582/TK/48778

Telah dipertahankan di depan dewan penguji  
pada tanggal : **12 Juni 2023**

**Susunan Dewan Penguji**

Pembimbing Utama

**Dr. I Wayan Mustika, S.T., M.Eng.**

Pembimbing Pendamping

**Ir. Agus Bejo, S.T., M.Eng., D.Eng., IPM.**

Anggota Dewan Penguji Lain

**Ir. Prapto Nugroho, S.T., M.Eng., D.Eng., IPM. Dr.Eng. Ir. Adha Imam Cahyadi, S.T., M.Eng.,  
IPM.**

Skripsi ini telah diterima sebagai salah satu persyaratan  
untuk memperoleh gelar Sarjana Teknik

Tanggal: 11 Juli 2023  
Pengelola Program Studi: Sarjana Teknik Elektro

**Dr.Eng. Ir. Adha Imam Cahyadi, S.T., M.Eng., IPM.**  
NIP 197911022008121001

Mengetahui,  
Ketua Departemen Teknik Elektro dan Teknologi Informasi



**Prof. Ir. Hanung Adi Nugroho, S.T., M.Eng., Ph.D., IPM., SMIEEE.**  
NIP 197802242002121001



## PERNYATAAN BEBAS PLAGIASI

Saya yang bertanda tangan di bawah ini :

Nama : I Made Pradhesta Surya Wibawa  
NIM : 19/443582/TK/48778  
Tahun terdaftar : 2019  
Program Studi : Program Sarjana Program Studi Teknik Elektro  
Fakultas : Teknik Universitas Gadjah Mada

Menyatakan bahwa dalam dokumen ilmiah Skripsi ini tidak terdapat bagian dari karya ilmiah lain yang telah diajukan untuk memperoleh gelar akademik di suatu lembaga Pendidikan Tinggi, dan juga tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang/lembaga lain, kecuali yang secara tertulis disitasi dalam dokumen ini dan disebutkan sumbernya secara lengkap dalam daftar pustaka.

Dengan demikian saya menyatakan bahwa dokumen ilmiah ini bebas dari unsur-unsur plagiasi dan apabila dokumen ilmiah Skripsi ini di kemudian hari terbukti merupakan plagiasi dari hasil karya penulis lain dan/atau dengan sengaja mengajukan karya atau pendapat yang merupakan hasil karya penulis lain, maka penulis bersedia menerima sanksi akademik dan/atau sanksi hukum yang berlaku.

Yogyakarta, 18 Mei 2023



I Made Pradhesta Surya Wibawa  
19/443582/TK/48778

## **HALAMAN PERSEMBAHAN**

Ambil yang baik, abaikan yang buruk.

## KATA PENGANTAR

Puji syukur ke hadirat Tuhan Yang Maha Esa atas berkat dan petunjuk-Nya sehingga skripsi yang berjudul "Pengembangan *Firmware Air Quality Monitoring Menggunakan Sensor Sensirion SPS30 dan MCU STM32 dengan Protokol Komunikasi I2C*" telah selesai dikerjakan. Penyusunan skripsi ini tidak lepas dari dukungan dan bimbingan dari berbagai pihak. Maka dari itu, penulis mengucapkan terima kasih kepada:

1. Ir. Hanung Adi Nugroho, S.T., M.E., Ph.D., IPM. selaku Ketua Departemen Teknik Elektro dan Teknologi Informasi Fakultas Teknik Universitas Gadjah Mada.
2. Ir. Lesnanto Multa Putranto, S.T., M.Eng., Ph.D., IPM. selaku Sekretaris Departemen Teknik Elektro dan Teknologi Informasi Fakultas Teknik Universitas Gadjah Mada.
3. Ir. Adha Imam Cahyadi, S.T., M.Eng., D.Eng., IPM. selaku Ketua Program Studi S1 Teknik Elektro Fakultas Teknik Universitas Gadjah Mada.
4. Dr. I Wayan Mustika, S.T., M.Eng. selaku Dosen Pembimbing 1 dan Ir. Agus Bejo, S.T., M.Eng., D.Eng., IPM. selaku Dosen Pembimbing 2 pada penyusunan skripsi ini.
5. Kedua Orang Tua yang selalu mendukung dari awal perkuliahan hingga proses penyusunan skripsi ini.
6. Bapak Syukron Abu Ishaq Alfarizi, S.T., Ph.D dan rekan-rekan tim Capstone A02 yang banyak membantu dan mengarahkan menjelang penggerjaan skripsi dimulai.
7. Rasyid dan Angga selaku teman satu bimbingan Pak Wayan yang senantiasa berdiskusi bersama selama penelitian.
8. Mas Edwin dan teman-teman PT Lunar Inovasi Teknologi yang selalu membantu selama penelitian.
9. Kholid Ibnu Falah, Rifqi Maulana, Raihan Ramadhan, dan teman-teman lainnya di DTETI UGM yang senantiasa berjuang bersama selama perkuliahan.
10. Nadine Stephanie, Anisa Satya, Annisa Indah, dan teman-teman KKN Girimulyo lainnya yang senantiasa mendukung dan memotivasi selama penyusunan skripsi.

Akhir kata, semoga skripsi ini dapat memberikan manfaat yang baik untuk kita semua baik secara langsung maupun tidak langsung.

## DAFTAR ISI

HALAMAN PENGESAHAN .....	ii
PERNYATAAN BEBAS PLAGIASI .....	iv
HALAMAN PERSEMBAHAN .....	iv
KATA PENGANTAR .....	v
DAFTAR ISI.....	vi
DAFTAR TABEL .....	ix
DAFTAR GAMBAR .....	x
DAFTAR SINGKATAN.....	xiii
INTISARI.....	xiv
ABSTRACT .....	xv
BAB I Pendahuluan .....	1
1.1 Latar Belakang .....	1
1.2 Rumusan Masalah .....	3
1.3 Tujuan Penelitian .....	3
1.4 Batasan Penelitian .....	3
1.5 Manfaat Penelitian .....	4
1.6 Sistematika Penulisan.....	4
BAB II Tinjauan Pustaka dan Dasar Teori .....	6
2.1 Tinjauan Pustaka .....	6
2.2 Dasar Teori .....	8
2.2.1 Indeks Kualitas Udara .....	8
2.2.2 <i>Particulate Matter</i> .....	10
2.2.3 <i>Optical Particle Counter</i> .....	11
2.2.4 Sensirion SPS30.....	12
2.2.5 Akurasi dan Presisi.....	14
2.2.6 OLED SSD1306 .....	15
2.2.7 Raspberry Pi 3 Model B .....	16
2.2.8 STM32.....	16
2.2.9 P-NUCLEO-WB55 .....	17
2.2.10 STM32CubeIDE .....	18
2.2.11 <i>Firmware</i> .....	18
2.2.12 Bahasa C.....	19
2.2.13 Bilangan <i>Floating Point</i> .....	20
2.2.13.1 Konversi Bilangan <i>Floating Point</i> .....	22
2.2.14 Bilangan <i>Integer</i> .....	23
2.2.15 I2C .....	23

2.2.15.1	Kondisi <i>Start</i> dan <i>Stop</i> .....	25
2.2.15.2	Alamat .....	25
2.2.15.3	Bit Data .....	26
2.2.15.4	Bit ACK/NACK.....	27
2.2.15.5	<i>Repeated Start</i> .....	27
2.2.15.6	<i>Clock Stretching</i> .....	27
2.2.16	UART .....	28
2.2.17	RealTerm .....	28
2.2.18	Grabserial.....	29
2.2.19	Python .....	29
2.3	Analisis Perbandingan Metode .....	29
BAB III	Metode Penelitian.....	31
3.1	Alat dan Bahan Tugas akhir .....	31
3.1.1	Alat Tugas akhir.....	31
3.1.2	Bahan Tugas akhir .....	31
3.2	Metode yang Digunakan.....	31
3.3	Alur Tugas Akhir .....	32
3.4	Studi Literatur .....	33
3.5	Persiapan <i>Hardware</i> dan <i>Software</i> .....	33
3.6	Perancangan Sistem.....	35
3.6.1	Perancangan Perangkat Keras .....	36
3.6.2	Pengembangan <i>Firmware</i> .....	37
3.6.2.1	Pembuatan Fungsi Perintah Sensor SPS30 .....	38
3.6.2.2	Algoritma Pengukuran Konsentrasi Partikulat .....	51
3.6.3	Pengolahan Data .....	54
3.6.4	Batasan Sistem .....	56
3.7	Pengujian Sistem.....	57
BAB IV	Hasil dan Pembahasan.....	59
4.1	Pengujian Seluruh Perintah pada sensor SPS30 .....	59
4.1.1	<i>Start Measurement</i> .....	59
4.1.2	<i>Read Measured Values</i> .....	61
4.1.3	<i>Device Reset</i> .....	69
4.1.4	<i>Read Data-Ready Flag</i> .....	69
4.1.5	<i>Stop Measurement</i> .....	71
4.1.6	<i>Sleep</i> .....	71
4.1.7	<i>Wake-up</i> .....	72
4.1.8	<i>Start Fan Cleaning</i> .....	73
4.1.9	<i>Read/Write Auto Cleaning Interval</i> .....	73
4.1.10	<i>Read Product Type</i> .....	75

4.1.11	<i>Read Serial Number</i> .....	76
4.1.12	<i>Read Version</i> .....	77
4.1.13	<i>Read Device Status Register</i> .....	79
4.1.14	<i>Clear Device Status Register</i> .....	80
4.2	Pengujian Komunikasi I2C antara Sensor SPS30 dan P-Nucleo-WB55 ....	82
4.2.1	Kondisi <i>Start</i> dan <i>Stop</i> .....	82
4.2.2	Alamat Sensor .....	83
4.2.3	Bit Data dan ACK/NACK .....	85
4.3	Pengujian Waktu <i>Start-up</i> Sensor SPS30 .....	87
4.3.1	Perbandingan antara Dengan dan Tanpa Waktu <i>Star-up</i> .....	87
4.3.2	Performa Sensor .....	93
4.4	Pengujian Sensor SPS30 untuk Mengukur Konsentrasi Partikulat .....	96
4.4.1	Pengujian di Ruang Produksi .....	97
4.4.2	Pengujian di Area <i>Outdoor</i> .....	99
4.4.3	Pengujian di Ruang <i>Showroom</i> .....	102
4.4.4	Perbandingan Kualitas Udara di Ketiga Lokasi .....	104
BAB V	Kesimpulan dan Saran .....	106
5.1	Kesimpulan .....	106
5.2	Saran .....	107
DAFTAR PUSTAKA .....		108

## DAFTAR TABEL

Tabel 2.1	Konversi Nilai Konsentrasi Parameter ISPU .....	8
Tabel 2.2	Konversi Nilai Konsentrasi Parameter ISPU .....	9
Tabel 2.3	Bilangan <i>floating point</i> khusus .....	21
Tabel 3.1	Daftar Perintah pada Sensor SPS30 .....	39
Tabel 3.2	Spesifikasi dan Batasan Sistem .....	56
Tabel 4.1	Pengelompokan Bit Bilangan <i>Float</i> pada Nilai Konsentrasi Massa PM1 .....	63
Tabel 4.2	Nilai Pengukuran dalam Bit dan Konversinya Menjadi <i>Float</i> .....	64
Tabel 4.3	Nilai Pengukuran dalam Bit dan Konversinya Menjadi <i>Integer</i> .....	66
Tabel 4.4	Nilai Pengukuran dalam 10 Iterasi .....	68
Tabel 4.5	Hasil Analisis Konsentrasi Massa Partikulat dan <i>Typical Particle Size</i>	94
Tabel 4.6	Hasil Analisis Konsentrasi Jumlah Partikulat .....	95
Tabel 4.7	Tingkat Presisi Setiap Parameter Nilai Partikulat.....	95
Tabel 4.8	Perbandingan Nilai ISPU PM2.5 dan PM10 di Tiga Lokasi Selama Pengujian .....	105

## DAFTAR GAMBAR

Gambar 2.1	Pemrosesan data pada OPC .....	12
Gambar 2.2	Sensirion SPS30 .....	13
Gambar 2.3	Prinsip penghamburan laser pada sensor Sensirion SPS30 .....	13
Gambar 2.4	Perbedaan akurasi dan presisi: (a) akurat dan presisi, (b) tidak akurat tetapi presisi, (c) akurat tetapi tidak presisi, dan (d) tidak akurat dan tidak presisi .....	14
Gambar 2.5	OLED SSD1306 .....	16
Gambar 2.6	Tipe-tipe STM32 .....	17
Gambar 2.7	P-NUCLEO-WB55 .....	18
Gambar 2.8	Diagram bilangan biner untuk (a) <i>single precision</i> dan (b) <i>double precision</i> pada <i>floating point number</i> .....	21
Gambar 2.9	<i>Pull-up resistor</i> pada I2C .....	24
Gambar 2.10	Diagram bit ketika (a) <i>master transmit</i> ke <i>slave</i> dan (b) <i>master receive</i> data dari <i>slave</i> .....	24
Gambar 2.11	Kondisi <i>start</i> dan <i>stop</i> .....	25
Gambar 2.12	Urutan bit dalam alamat 10 bit .....	26
Gambar 2.13	Diagram urutan bit dalam UART .....	28
Gambar 3.1	Diagram alir penelitian .....	32
Gambar 3.2	Konfigurasi piranti I/O yang akan digunakan .....	34
Gambar 3.3	Konfigurasi <i>clock</i> .....	35
Gambar 3.4	Diagram blok sistem .....	36
Gambar 3.5	Diagram <i>wiring</i> rangkaian .....	36
Gambar 3.6	Diagram bit komunikasi I2C pada perintah <i>Device Reset</i> .....	42
Gambar 3.7	Diagram bit komunikasi I2C pada perintah <i>Wake-up</i> .....	43
Gambar 3.8	Diagram bit komunikasi I2C pada perintah <i>Start Measurement</i> ..	46
Gambar 3.9	Diagram bit komunikasi I2C pada perintah <i>Read Measured Values</i> ..	50
Gambar 3.10	Diagram alir program .....	52
Gambar 4.1	Rangkaian sistem .....	59
Gambar 4.2	Sinyal I2C perintah <i>Start Measurement</i> pada osiloskop .....	60
Gambar 4.3	Kuat arus yang mengalir pada mode <i>measurement</i> .....	61
Gambar 4.4	Sinyal I2C Perintah <i>Read Measured Values</i> dengan format data <i>float</i> pada osiloskop. (a) dan (b) merupakan bagian pertama, yaitu mengirim <i>pointer</i> alamat perintah. (c), (d), dan (e) merupakan bagian kedua, yaitu alamat sensor dan data pengukuran dari sensor. (f) merupakan akhir dari bagian kedua. ....	62
Gambar 4.5	Nilai pengukuran format data <i>float</i> pada <i>serial monitor</i> . jumlah angka di belakang koma dibulatkan menjadi 2 angka. ....	64
Gambar 4.6	Sinyal I2C Perintah <i>Read Measured Values</i> dengan format data <i>integer</i> pada osiloskop. (a) dan (b) merupakan bagian pertama, yaitu mengirim <i>pointer</i> alamat perintah. (c), (d), dan (e) merupakan bagian kedua, yaitu alamat sensor dan data pengukuran dari sensor. (f) merupakan akhir dari bagian kedua. ....	65
Gambar 4.7	Nilai pengukuran format data <i>integer</i> pada <i>serial monitor</i> .....	67

Gambar 4.8	Perbedaan sinyal I2C antara (a) sinyal <i>pointer</i> alamat perintah dan data yang dibaca dan (b) sinyal data yang dibaca saja .....	68
Gambar 4.9	Sinyal I2C perintah <i>Device Reset</i> pada osiloskop .....	69
Gambar 4.10	Pembacaan <i>data-ready flag</i> setiap 250 ms ketika nilai pengukuran konsentrasi partikulat dibaca atau diambil setiap 1 detik .....	70
Gambar 4.11	Kuat arus yang mengalir pada mode <i>idle</i> .....	71
Gambar 4.12	Kuat arus yang mengalir pada mode <i>sleep</i> .....	72
Gambar 4.13	Kuat arus yang mengalir pada mode <i>wake-up</i> .....	73
Gambar 4.14	Sinyal I2C perintah <i>Wake-up</i> pada osiloskop .....	74
Gambar 4.15	Kuat arus yang mengalir pada saat dilakukan pembersihan kipas .	75
Gambar 4.16	Interval pembersihan kipas otomatis secara <i>default</i> .....	76
Gambar 4.17	Interval pembersihan kipas otomatis yang baru .....	76
Gambar 4.18	Keluaran perintah <i>Read Product Type</i> pada monitor .....	77
Gambar 4.19	Keluaran perintah <i>Read Serial Number</i> pada monitor .....	77
Gambar 4.20	Nomor seri sensor SPS30.....	78
Gambar 4.21	Keluaran perintah <i>Read Version</i> pada monitor .....	78
Gambar 4.22	Sinyal I2C perintah <i>Read Version</i> pada osiloskop .....	79
Gambar 4.23	Keluaran perintah <i>Read Device Status Register</i> pada monitor dan keterangannya .....	81
Gambar 4.24	Sinyal I2C perintah <i>Clear Device Status Register</i> pada osiloskop .....	81
Gambar 4.25	Sinyal I2C untuk kondisi <i>start</i> pada osiloskop .....	82
Gambar 4.26	Sinyal I2C untuk kondisi <i>stop</i> pada osiloskop .....	83
Gambar 4.27	Sinyal I2C yang menunjukkan alamat sensor SPS30 ketika (a) <i>transmit</i> dan (b) <i>receive</i> .....	84
Gambar 4.28	Sinyal I2C ketika tidak ada <i>slave</i> yang terhubung pada master....	85
Gambar 4.29	Urutan <i>frame</i> yang berisi 8 bit data dan 1 bit ACK/NACK .....	85
Gambar 4.30	Bit NACK yang terbentuk pada komunikasi.....	86
Gambar 4.31	Satu bit sinyal pada jalur SDA dan SCL .....	87
Gambar 4.32	Grafik data pengukuran dengan menggunakan waktu <i>start-up</i> selama 30 detik untuk parameter nilai konsentrasi massa partikulat	88
Gambar 4.33	Grafik data pengukuran tanpa waktu <i>start-up</i> untuk parameter nilai konsentrasi massa partikulat .....	89
Gambar 4.34	Grafik data pengukuran dengan menggunakan waktu <i>start-up</i> selama 30 detik untuk parameter nilai konsentrasi jumlah partikulat .....	90
Gambar 4.35	Grafik data pengukuran tanpa waktu <i>start-up</i> untuk parameter nilai konsentrasi jumlah partikulat .....	91
Gambar 4.36	Grafik data pengukuran dengan menggunakan waktu <i>start-up</i> selama 30 detik untuk parameter nilai <i>typical particle size</i> partikulat .....	92
Gambar 4.37	Grafik data pengukuran tanpa waktu <i>start-up</i> untuk parameter nilai <i>typical particle size</i> partikulat .....	93
Gambar 4.38	Rangkaian sistem untuk pengujian mengukur konsentrasi partikulat .....	96
Gambar 4.39	Penempatan sistem sensor di ruang produksi.....	97
Gambar 4.40	Grafik data pengukuran PM2.5 dan PM10 di ruang produksi .....	98
Gambar 4.41	Grafik ISPU PM2.5 dan PM10 di ruang produksi .....	99
Gambar 4.42	Penempatan sistem sensor pada area <i>outdoor</i> .....	99

Gambar 4.43	Grafik data pengukuran PM2.5 dan PM10 di area <i>outdoor</i> .....	100
Gambar 4.44	Grafik ISPU PM2.5 dan PM10 di area <i>outdoor</i> .....	101
Gambar 4.45	Penempatan sistem sensor di ruang <i>showroom</i> .....	102
Gambar 4.46	Grafik data pengukuran PM2.5 dan PM10 di ruang <i>showroom</i> ....	103
Gambar 4.47	Grafik ISPU PM2.5 dan PM10 di ruang <i>showroom</i> .....	104

## **DAFTAR SINGKATAN**

ACK	= Acknowledgement
AQG	= Air Quality Guideline
AQI	= Air Quality Index
CAQI	= Common Air QUality Index
HVAC	= Heating, Ventilation, and Air Conditioner
I2C	= Inter-Intergrated Circuit
I/O	= Input/Output
IDE	= Intergrated Development Environment
ISPU	= Indeks Standar Pencemar Udara
KHLK	= Kementerian Lingkungan Hidup dan Kehutanan
LSB	= Least Significant Bit/Byte
MCU	= Microcontroller Unit
MSB	= Most Significant Bit/Byte
NACK	= Negative Acknowledgement
OLED	= Organic Light-Emitting Diode
PM	= Particulate Matter
R/W	= Read or Write
RSD	= Relative Standrd Deviation
SDA	= Serial Data
SCL	= Serial Clock
TPS	= Typical Particle Size
UART	= Universal Asyncrhonous Receiver-Transmitter
WHO	= World Health Organization

## INTISARI

Udara dapat tercemar oleh berbagai macam zat pencemar, salah satunya adalah partikulat (PM2.5 adn PM10). Keberadaan partikulat di udara memiliki dampak yang berbahaya bagi kesehatan makhluk hidup jika terhirup. Penelitian ini memiliki tujuan untuk mengembangkan *firmware* sistem pemantauan konsentrasi partikulat di udara sehingga dapat diketahui indeks kualitas udaranya berdasarkan standar ISPU dan mempelajari secara detail terkait implementasi protokol komunikasi I2C antara mikrokontroler dan sensor.

Sistem pemantauan bekerja dengan menggunakan sensor Sensirion SPS30 dan mikrokontroler STM32. Hasil pembacaan kemudian ditampilkan pada layar OLED dan disimpan di dalam komputer, yaitu Raspberry Pi 3 Model B. *Firmware* dikembangkan dengan membuat fungsi-fungsi untuk setiap perintah yang tersedia pada sensor. Dalam proses pengembangannya, pemahaman mengenai komunikasi I2C dibutuhkan agar perintah yang dijalankan sesuai. Algoritma pengukuran konsentrasi partikulat juga harus diperhatikan untuk menghasilkan pengukuran yang sesuai.

*Firmware* yang dikembangkan berhasil untuk komunikasi antara mikrokontroler dan sensor. *Firmware* diuji untuk mengukur konsentrasi massa partikulat di tiga lokasi yang berbeda di area UGM Press, yaitu ruang produksi, area *outdoor*, dan ruang *showroom*. Hasil pengujian menunjukkan bahwa nilai ISPU di area *outdoor* adalah yang terendah, yaitu 56,03 sedangkan nilai ISPU di ruang produksi adalah yang tertinggi, yaitu 60,20. Namun, secara umum indeks kualitas udara harian di ketiga lokasi tersebut berada pada level Sedang.

**Kata kunci :** Indeks kualitas udara, partikulat, Sensirion SPS30, STM32, I2C

## ABSTRACT

*Air can be polluted by various kinds of pollutant, one of them is particulate matter (PM2.5 and PM10). The presence of particulate matter in the air has harmful impacts on the health of living organisms when inhaled. This research aims to develop a particulate matter concentration monitoring system firmware, so that the air quality index can be determined based on the ISPU standard, and to study in detail the implementation of I2C communication protocol between microcontroller and sensor.*

*Monitoring system works using Sensirion SPS30 sensor and STM32 microcontroller. The reading results are then displayed on an OLED screen and stored in a computer, which is Raspberry Pi 3 Model B. Firmware is developed by creating functions of all the commands that available on the sensor. In the development process, understanding of I2C communication is necessary, so that the commands run well. The algorithm of particulate matter measurement should also be considered to obtain proper measurements.*

*The developed firmware successfully enables communication between microcontroller and sensor. The firmware was tested to measure the mass concentration of particulate matter at three different locations, i.e. production room, outdoor area, and showroom. The test result showed that the ISPU value in the outdoor area was the lowest, at 56.03, while the ISPU value in the production room was the highest, at 60.20. However, overall, the daily air quality index in all three locations was at a Moderate level.*

**Keywords :** Air quality index, particulate matter, SPS30, STM32, I2C

# BAB I

## PENDAHULUAN

### 1.1 Latar Belakang

Pencemaran udara adalah peristiwa tercampurnya komponen-komponen fisik, kimia, atau biologi dalam udara dengan jumlah yang dapat membahayakan kehidupan makhluk hidup. Udara tercemar yang masuk ke dalam manusia dapat merusak organ pada sistem pernapasan. Gangguan pernapasan tersebut antara lain asma dan kanker paru-paru. Pencemaran udara dapat terjadi karena faktor alami ataupun faktor manusia [1]. Contoh faktor alami adalah letusan gunung berapi yang menghasilkan gas karbon dioksida ( $\text{CO}_2$ ). Penyebab udara tercemar dari faktor manusia dapat datang dari banyak bidang. Contohnya dari kendaraan bermotor, pembangkit listrik, dan limbah dari industri atau pabrik. Hal yang juga sering dikenal dengan sebutan polusi udara ini merupakan epidemi yang terkait pada polusi paling buruk dengan menyebabkan jumlah kematian sekitar tujuh juta setiap tahunnya [2]. Dengan demikian, kualitas udara perlu dipantau. Kementerian Lingkungan Hidup dan Kehutanan (KHLK) Republik Indonesia mengeluarkan Peraturan Menteri Lingkungan Hidup dan Kehutanan nomor 14 tahun 2020 tentang Indeks Standar Pencemar Udara (ISPU). ISPU merupakan angka tanpa satuan yang menggambarkan kualitas udara ambien di lokasi dan waktu tertentu dan berdasar pada dampaknya terhadap kesehatan manusia, nilai estetika, dan makhluk hidup lainnya. Zat pencemar atau polutan terdiri dari banyak jenisnya. Di dalam peraturan tersebut dijelaskan bahwa terdapat tujuh parameter pada perhitungan ISPU, yaitu karbon monoksida (CO), Sulfur dioksida ( $\text{SO}_2$ ), Natrium dioksida ( $\text{NO}_2$ ), Ozon ( $\text{O}_3$ ), hidrokarbon, PM10, dan PM2.5 [3].

Salah satu jenis polutan yang harus diperhatikan adalah *particulate matter* (PM) atau partikulat. Efek keberadaannya dalam jangka pendek maupun jangka panjang dapat menyebabkan bahaya yang berujung kematian kepada manusia. Partikulat itu sendiri merupakan suatu campuran kompleks partikel di dalam udara dengan komponen yang memiliki karakteristik fisik dan kimia yang beragam. Kandungannya yang kompleks menyebabkan potensi bahaya partikulat bervariasi dengan ukuran, karakteristik fisik, komposisi kimia, dan asal partikel tersebut. Karakteristik partikulat yang berbeda dapat menyebabkan bahaya yang berbeda pula. Untuk itu, WHO memberikan rekomendasi untuk kualitas udara yang disebut dengan *air quality guideline* (AQG). Terdapat AQG untuk PM dengan ukuran kurang dari atau sama dengan  $2,5 \mu\text{m}$  (PM2.5) atau  $10 \mu\text{m}$  (PM10). Untuk PM25, rekomendasi tahunannya adalah  $5 \mu\text{g}/\text{m}^3$ , sedangkan untuk PM10, rekomendasi tahunannya adalah  $15 \mu\text{g}/\text{m}^3$ . Sementara itu, rekomendasi dalam jangka pendek atau dalam 24 jam bernilai  $15 \mu\text{g}/\text{m}^3$  untuk PM2.5 dan  $45 \mu\text{g}/\text{m}^3$  untuk PM10 [4]. Meskipun terdapat nilai AQG yang dikeluarkan WHO tersebut, data yang diambil dari IQAir

menunjukkan bahwa kandungan PM2.5 dalam udara Indonesia melebihi nilai tersebut, khususnya di kota-kota besar. Pada tahun 2021, kandungan PM2.5 dalam udara di Jakarta mencapai  $39.2 \mu\text{g}/\text{m}^3$  yang berarti hampir delapan kali lipat dari nilai AQG tahunan. Begitu pula pada tahun 2020, kandungan PM2.5 di Jakarta mencapai  $39.6 \mu\text{g}/\text{m}^3$  [5]. Jika dibiarkan dalam jangka panjang, kesehatan warga Jakarta dapat terganggu meskipun hanya menjalankan kegiatan sehari-harinya.

Tidak hanya di luar, kualitas udara di dalam ruangan atau bangunan juga harus diperhatikan. Berada di dalam ruangan tidak menghindarkan kita dari polusi udara. Empat elemen yang memengaruhi kualitas udara di dalam ruangan adalah sumber polutan, sistem HVAC (*Heating, Ventilation, and Air Conditioner*) di dalam ruangan, jalur aliran udara, dan penghuni ruangan tersebut. Ruangan dengan sistem sirkulasi udara yang buruk dapat menyebabkan partikel-partikel kecil seperti debu halus, spora jamur, serbusk sari tanaman, dan tungau debu mencemari udara [6, 7]. Tergantung dari kegiatan dan pekerjaan masing-masing, manusia menghabiskan sebagian besar dari waktunya di dalam ruangan. Sekitar satu per tiga waktu dalam sehari dihabiskan untuk bekerja dan satu per empat digunakan untuk tidur. Dengan demikian, manusia bisa terpapar udara yang tercemar juga di dalam ruangan. Bahaya yang dapat timbul dari partikulat PM2.5 dan PM10 adalah gangguan sistem pernapasan, pneumonia, iritasi mata, alergi, dan bronkitis kronis. Terlebih lagi PM2.5 dapat masuk ke dalam paru-paru bagian yang lebih dalam sehingga dapat menimbulkan emfisma paru, asma bronkial, kanker paru-paru, dan gangguan kardiovaskular. Upaya yang dapat dilakukan untuk mengurangi paparan partikulat, antara lain membersihkan ruangan dari debu dengan kain pel basah atau penyedot debu, memasang penangkap debu pada ventilasi rumah dan dibersihkan secara rutin, menanam tanaman di sekitar rumah untuk menghalangi debu masuk ke rumah, dan khusus untuk dapur didesain memiliki ventilasi yang memiliki bukaan setidaknya 40% dari luas lantai dengan sistem silang sehingga ada aliran udara atau dapat juga memanfaatkan teknologi penyedot asap. Selain itu, untuk mendukung upaya-upaya tersebut dan agar dapat mengetahui bagaimana konsentrasi partikulat di udara, pemantauan kualitas udara juga dapat dilakukan [8].

Pemantauan kualitas udara sangat penting untuk dilakukan. Teknologi yang kini berkembang pesat memudahkan kita dalam melakukan pemantauan kualitas udara. Pemantauan konsentrasi partikulat di udara dapat dilakukan dengan menggunakan sensor yang didedikasikan khusus untuk pengukuran partikulat. Penggunaan sensor memungkinkan kita untuk melakukan pengukuran dan pemantauan konsentrasi partikulat secara *real-time*. Pembacaan secara *real-time* penting dilakukan supaya tindakan pencegahan untuk mengurangi panjuran terhadap partikel yang berbahaya dapat diambil. Sensor ini juga dapat menunjang sistem HVAC yang umumnya sudah terpasang di sebuah ruangan. Untuk menunjang fleksibilitas peletakan sensor tersebut, sensor-sensor saat ini juga su-

dah banyak yang berukuran kecil dengan tetap memiliki performa yang mumpuni. Akan tetapi, sensor tidak dapat berdiri sendiri. Sensor dapat melakukan pengukuran ketika diberi perintah. Perintah tersebut diberikan oleh mikrokontroler yang memiliki *firmware* di dalamnya [9]. Untuk itu, agar sensor dapat memberikan hasil pengukurannya, *firmware* perlu dikembangkan. Sangat beragamnya jenis mikrokontroler, sensor, dan protokol yang tersedia mengharuskan *firmware* dikembangkan secara spesifik sesuai dengan spesifikasi alat yang digunakan.

Sesuai pemaparan di atas, tujuan dari tugas akhir ini adalah mengembangkan sebuah *firmware* yang akan diimplementasikan untuk pembacaan data nilai pengukuran konsentrasi partikulat dari sebuah sensor dan meneliti bagaimana protokol komunikasi digunakan untuk komunikasi antara mikrokontroler dengan sensor yang digunakan. Tujuan utama pengembangan *firmware* ini adalah menyajikan data pengukuran konsentrasi partikulat di udara sehingga dapat menjadi indikator baik buruknya kualitas udara di sekitar sensor.

## 1.2 Rumusan Masalah

Dari penjelasan latar belakang di atas, rumusan masalah pada penelitian ini disusun sebagai berikut:

1. Dibutuhkannya *firmware* agar mikrokontroler dapat memerintah sensor untuk melakukan pengukuran konsentrasi partikulat.
2. Diperlukan implementasi protokol komunikasi yang tepat sesuai dengan spesifikasi sensor dan mikrokontroler.
3. Konsentrasi partikulat di udara perlu dipantau dan diperhatikan karena pada konsentrasi tinggi dapat berdampak buruk bagi kesehatan.

## 1.3 Tujuan Penelitian

Tujuan dari penelitian ini adalah sebagai berikut:

1. Mengembangkan *firmware* untuk pembacaan konsentrasi partikulat di udara menggunakan sensor Sensirion SPS30 dengan mikrokontroler STM32.
2. Mempelajari secara lebih detail serta mengimplementasikan protokol komunikasi yang digunakan antara mikrokontroler dan sensor.
3. Mengetahui kualitas udara di lokasi pengujian dengan berdasarkan pada standar indeks kualitas udara.

## 1.4 Batasan Penelitian

Batasan pada penelitian ini adalah sebagai berikut:

1. Objek penelitian: Pengembangan *firmware* dan protokol komunikasi I2C yang digunakan pada sebuah sensor dan mikrokontroler untuk pemantauan konsentrasi partikulat di udara.
2. Metode penelitian: Penelitian pengembangan *firmware* pada sebuah mikrokontroler dan sensor.
3. Waktu dan tempat penelitian: Waktu penelitian adalah November-Mei 2023 di UGM Press.
4. Populasi dan sampel: Populasi adalah udara di lokasi pengujian selama waktu pengukuran. Sampel udara diambil setiap 2 menit.
5. Variabel: Variabel bebasnya adalah skenario dan kondisi lokasi pengujian, dan variabel terikatnya adalah indikator kualitas udara yang didapat.
6. Hipotesis: Bawa kondisi masing-masing lokasi akan memiliki nilai konsentrasi partikulat yang berbeda-beda.
7. Keterbatasan Penelitian: Tingkat akurasi dan presisi pembacaan sensor terbatas pada dokumentasi spesifikasi yang ada, tidak dibandingkan dengan hasil pembacaan alat ukur lainnya. Protokol komunikasi yang diteliti berfokus pada protokol komunikasi I2C antara sensor dengan mikrokontroler.

## 1.5 Manfaat Penelitian

Manfaat dari penelitian ini adalah sebagai berikut:

1. Dengan dikembangkannya *firmware* untuk mikrokontroler dan sensor, pengguna dapat mengetahui bagaimana konsentrasi partikulat pada udara di sekitarnya.
2. Pengguna dapat mengetahui bagaimana indeks kualitas udara di sekitar.
3. Pembaca dapat memahami lebih dalam mengenai protokol komunikasi I2C yang digunakan untuk komunikasi antara mikrokontroler dengan sensor.

## 1.6 Sistematika Penulisan

Bab I adalah pendahuluan. Bab ini membahas mengenai latar belakang, rumusan masalah, batasan masalah, tujuan tugas akhir, manfaat tugas akhir, dan sistematika penulisan.

Bab II adalah tinjauan pustaka dan dasar teori. Bab ini membahas tinjauan terkait dengan penelitian terdahulu dan dasar teori mengenai pengembangan *firmware* pemantauan kualitas udara menggunakan sensor Sensirion SPS30 dan STM32.

Bab III adalah metode penelitian. Bab ini membahas mengenai alat dan bahan yang dipakai serta metodologi atau langkah-langkah dalam pengembangan *firmware* ini selama penelitian.

Bab IV adalah hasil dan pembahasan. Bab ini menjelaskan tentang hasil dan penelitian dan pembahasannya.

Bab V adalah kesimpulan dan saran. Bab ini berisi kesimpulan yang diperoleh dari penelitian yang dilakukan dan saran yang dapat dilakukan selanjutnya.

## **BAB II**

### **TINJAUAN PUSTAKA DAN DASAR TEORI**

#### **2.1 Tinjauan Pustaka**

Pada penelitian dengan judul “*Measuring Particulate Matter (PM) using SPS30*”, peneliti menganalisis kualitas udara di Slovakia, tepatnya di kota Košice dan desa kecil bernama Habura. Besaran-besaran yang diukur adalah kandungan partikulat menggunakan sensor SPS30, suhu dan kelembapan menggunakan sensor SHT30, dan suhu dan tekanan barometrik menggunakan sensor MS5611. Untuk pengambilan data, sensor-sensor tersebut dihubungkan dengan *development board* Arduino Mega. Selain itu, digunakan juga modul RTC untuk membaca waktu dan kartu microSD untuk menyimpan data hasil pengukuran. Data diukur dan disimpan ke dalam sebuah *file* yang bernama *log.csv* setiap lima detik. Data yang diambil kemudian dianalisis menggunakan MATLAB. Dengan menggunakan MATLAB pun juga dihitung korelasi antara besaran-besaran yang diukur [10]. Pada penelitian ini, dapat dijelaskan bagaimana kondisi kualitas udara di kedua lokasi pengambilan sampel. Akan tetapi, tidak dijelaskan secara detail bagaimana *firmware* yang digunakan pada Arduino Mega. Banyaknya sensor yang digunakan juga membuat tidak fokus membahas lebih dalam pada satu sensor yang terkait dengan spesifikasi, *wiring*, dan protokol komunikasinya.

Pada penelitian lain dengan judul “*Development of Air Quality Boxes Based on Low-Cost Sensor Technology for Ambient Air Quality Monitoring*”, peneliti melakukan pengembangan sebuah perangkat *atmospheric exposure low-cost monitoring* (AELCM). Di dalam perangkat tersebut terdapat berbagai macam sensor serta modul RTC yang terhubung ke *development board* Arduino Mega 2560 Rev3. Besaran-besaran yang diukur beserta sensornya adalah suhu dan kelembapan menggunakan BME280, O<sub>3</sub> menggunakan MQ131 atau DGS-O<sub>3</sub>, NO<sub>2</sub> menggunakan MiCS-2714 atau DGS-NO<sub>2</sub>, CO menggunakan MiCS-4514 atau DGS-CO, dan partikulat menggunakan SPS30. Hasil pengukuran sensor dikalibrasi menggunakan model *multiple linear regression* untuk rata-rata per jam besaran-besaran gas dan rata-rata per 15 menit besaran partikulat. Kalibrasi tersebut dilakukan menggunakan nilai referensi dari *atmospheric exposure monitoring station* (AEMS) [11]. Pada penelitian ini, dapat ditunjukkan bagaimana performa dari berbagai macam sensor *low-cost* yang digunakan pada AELCM. Namun, penelitian tersebut hanya berfokus pada hasil pengukuran dari setiap sensor tersebut. Indikator mengenai baik buruknya kualitas udara yang terukur tidak dijelaskan secara detail. Padahal hal tersebut penting untuk diketahui di dalam pemantauan kualitas udara. Selain itu, *firmware* yang digunakan untuk pengukuran dan kalibrasi yang dilakukan tidak dipaparkan secara jelas.

Penelitian lain yang berjudul “*Rancang Bangun Sistem Monitor Kualitas Udara*

*dalam Ruangan Berbasis System-on-Chip ESP32*" pada tahun 2019, dilakukan pengembangan sistem pemantauan kualitas udara. Peneliti menggunakan tiga jenis sensor untuk mengukur lima parameter. Sensor dan besaran yang diukurnya, antara lain sensor K30 untuk mengukur CO<sub>2</sub>, sensor Sharp GP2Y1014AU0F untuk mengukur partikulat di udara, dan sensor BME680 untuk mengukur temperatur, kelembapan relatif, dan VOC (*Volatile Organic Compounds*). Sensor-sensor tersebut dikendalikan dengan *System-on-Chip* ESP32 yang diprogram dengan ESP-IDF. Data hasil pengukuran setiap sensor dikirim ke sebuah basis data dengan protokol komunikasi MQTT (*Message Queueing Telemetry Transport*) dan bantuan VPS (*Virtual Private Server*). Dari basis data, data pengukuran sensor ditampilkan pada sebuah *dashboard* Metabase. Dengan demikian data-data tersebut dapat diolah dan dilihat dengan mudah. Pengembangan sistem ini menghasilkan pembacaan parameter yang sesuai dengan nilai tunda yang rendah sehingga memungkinkan memantau kualitas udara secara mendekati *real-time* [12]. Pada penelitian ini, sistem dikembangkan dengan memanfaatkan konsep *Internet of Things* (IoT) yang menggunakan MQTT sebagai protokol komunikasi sehingga membuat komunikasinya menjadi ringan dan memiliki skalabilitas tinggi. Pemanfaatan RTOS dalam pengembangan *firmware*-nya juga membuatnya dapat melakukan fungsi *multitasking*. Namun, pada sistem ini pengaturan koneksi WiFi masih dilakukan secara *hardcode* yang membuat penggunaannya menjadi tidak fleksibel. Selain itu, fitur keamanan juga belum ada pada sistem ini.

Pada tahun 2022, Wibowo melakukan penelitian yang berjudul "*Rancang Bangun Sistem Detektor Konsentrasi Karbon Dioksida (CO<sub>2</sub>) dan Particulate Matter (PM<sub>2.5</sub> & PM<sub>10</sub>) untuk Sistem Pemantauan Lingkungan Ruang Huni*" [13]. Peneliti mengembangkan sistem pemantauan kualitas udara dengan menggunakan sensor MH-Z19B untuk mengukur konsentrasi CO<sub>2</sub> dan sensor SDS011 untuk mendeteksi partikulat PM<sub>2.5</sub> dan PM<sub>10</sub>. Pengujian konsentrasi CO<sub>2</sub> dilakukan dengan stimulus napas manusia sedangkan pengujian partikulat menggunakan stimulus bubuk tapioka. Kedua sensor tersebut dikendalikan dengan menggunakan *development board* Arduino Mega 2560. Data hasil pengukuran disimpan di dalam SD *card* dan ditampilkan pada LCD. Program akan berjalan secara terus menerus dengan *delay* 10 detik. Pada penelitian ini, didapatkan nilai rata-rata kesalahan baca, nilai akurasi, dan nilai *repeatability* dari kedua sensor. Proses membaca dan menyimpan data juga memiliki tingkat keberhasilan 100% dengan waktu akuisisi data selama 10,15 detik per data. Pengujian partikulat dilakukan dengan menggunakan pembanding, yaitu sensor sensor HT-9600. Pada pengujian tersebut, stimulus bubuk tapioka tidak ideal digunakan karena ukurannya yang relatif lebih besar untuk pengukuran PM<sub>10</sub> dan tidak disebar dengan rata. Oleh karenanya, terdapat perbedaan hasil dari tiga pengujian yang dilakukan. Maka dari itu, agar mendapatkan hasil yang lebih baik, lebih baik menggunakan bahan uji yang telah tersertifikasi.

Dari penelitian-penelitian di atas, pengembangan mengenai sistem pemantauan

kualitas udara telah banyak dilakukan dengan menggunakan berbagai macam sensor, mikrokontroler, *tools*, dan metode. Akan tetapi belum ada penelitian yang menjelaskan bagaimana *firmware* dikembangkan untuk mikrokontroler STM32 dan sensor Sensirion SPS30 secara lebih detail. Penelitian mengenai protokol komunikasi yang digunakan antara sensor dan mikrokontroler juga belum ada. Oleh karena itu, pada penelitian ini akan dikembangkan *firmware* untuk melakukan pemantauan kualitas udara, khususnya pemantauan konsentrasi partikulat di udara, dan protokol komunikasi yang digunakan, yaitu I2C, hingga data pengukuran sensor dapat terbaca.

## 2.2 Dasar Teori

### 2.2.1 Indeks Kualitas Udara

Indeks kualitas udara adalah angka yang digunakan untuk melaporkan kualitas udara dalam rentang tertentu di suatu lokasi. Angka ini digunakan untuk menggambarkan seberapa bersih atau tercemar udara, termasuk bahaya yang mungkin harus diperhatikan pada angka tersebut. Indeks kualitas udara dibuat agar masyarakat dapat adaptasi pola hidup pada indeks kualitas tertentu dan dapat mengambil langkah untuk menjaga kualitas udara tetap sesuai standar. Setiap negara memiliki standar baku mutunya sendiri terhadap kualitas udara. Misalnya Amerika Serikat memiliki AQI (*Air Quality Index*) yang diatur oleh US EPA dan negara-negara di Eropa memiliki CAQI (*Common Air Quality Index*). Secara umum, perbedaan keduanya terletak pada rentang indeks yang dipakai dan rentang konversi pada setiap level untuk setiap parameter yang diukur. AQI memiliki nilai rentang dari 0 hingga 500 yang dibagi menjadi enam level kategori (*Good*, *Moderate*, *Unhealthy for Sensitive Groups*, *Unhealthy*, *Very Unhealthy*, dan *Hazardous*), sedangkan CAQI memiliki nilai rentang dari 0 hingga >100 yang dibagi menjadi lima level kategori (*Very Low*, *Low*, *Medium*, *High*, dan *Very High*) [14, 15].

Tabel 2.1. Konversi Nilai Konsentrasi Parameter ISPU

ISPU	PM2.5 ( $\mu\text{g}/\text{m}^3$ )	PM10 ( $\mu\text{g}/\text{m}^3$ )	$\text{SO}_2$ ( $\mu\text{g}/\text{m}^3$ )	CO ( $\mu\text{g}/\text{m}^3$ )	$\text{O}_3$ ( $\mu\text{g}/\text{m}^3$ )	$\text{NO}_2$ ( $\mu\text{g}/\text{m}^3$ )	HIC ( $\mu\text{g}/\text{m}^3$ )
0-50	15,5	50	52	4000	120	80	45
51-100	55,4	150	180	8000	235	200	100
101-200	150,4	350	400	15000	400	1130	215
201-300	250,4	420	800	30000	800	2260	435
>300	500	500	1200	45000	1000	3000	648

Di Indonesia, indeks kualitas udara yang digunakan diatur dalam Peraturan Menteri Lingkungan Hidup dan Kehutanan Republik Indonesia nomor 14 tahun 2020 tentang Indeks Standar Pencemar Udara (ISPU). Kondisi mutu udara dalam ISPU didasarkan pada dampak terhadap kesehatan manusia, nilai estetika, dan makhluk hidup lain. Dalam

ISPU, parameter yang diukur antara lain Karbon monoksida (CO), Sulfur dioksida ( $\text{SO}_2$ ), Natrium dioksida ( $\text{NO}_2$ ), Ozon ( $\text{O}_3$ ), hidrokarbon (HIC), PM10, dan PM2.5 [3].

Konversi nilai parameter terukur menjadi indeks kualitas udara berdasarkan ISPU mengikuti tabel 2.1 di atas. Nilai parameter tersebut merupakan nilai rata-rata dalam 24 jam. Perhitungan konversi ditunjukkan pada persamaan 2-1.

$$I = \frac{I_a - I_b}{p_a - p_b} (x - p_b) + I_b \quad (2-1)$$

Keterangan:

- $I$  : ISPU terhitung
- $I_a$  : batas atas ISPU
- $I_b$  : batas bawah ISPU
- $p_a$  : batas atas nilai konsentrasi parameter
- $p_b$  : batas bawah nilai konsentrasi parameter
- $x$  : nilai konsentrasi parameter terukur

Berikut salah satu contoh cara hitung konversi nilai parameter yang terukur menjadi ISPU. Diketahui konsentrasi partikulat PM2.5 di udara rata-rata dalam 24 jam adalah  $31,4 \mu\text{g}/\text{m}^3$ . Maka, batas atas nilai konsentrasi parameternya ( $p_a$ ) sesuai tabel di atas adalah  $55,4 \mu\text{g}/\text{m}^3$  dan batas bawah nilai konsentrasi parameternya ( $p_b$ ) adalah  $15,5 \mu\text{g}/\text{m}^3$ . Sementara itu, batas atas ISPU  $I_a$  sama dengan 100 dan batas bawah ISPU  $I_B$  sama dengan 50. Seluruh nilai tersebut kemudian dimasukkan ke dalam Persamaan 2-1 dan didapatkan nilai ISPU:

$$I = \frac{100 - 50}{55,4 - 15,5} (31,4 - 15,5) + 50 = 69,92 \quad (2-2)$$

Tabel 2.2. Konversi Nilai Konsentrasi Parameter ISPU

Rentang ISPU	Kategori	Kode Warna
1-50	Baik	Hijau
51-100	Sedang	Biru
101-200	Tidak Sehat	Kuning
201-300	Sangat Tidak Sehat	Merah
>300	Berbahaya	Hitam

Nilai ISPU yang diperoleh dari konversi memiliki indikator sehat atau tidaknya seperti yang ditunjukkan pada Tabel 2.2 di atas. Dengan demikian. nilai ISPU 69,92 berdasarkan tabel tersebut masuk ke dalam kategori Sedang. Keterangan dari tiap kate-

gorinya adalah sebagai berikut:

1. Baik

Kualitas udara sangat baik dan tidak memberikan efek negatif terhadap makhluk hidup.

2. Sedang

Kualitas udara masih dapat diterima pada kesehatan makhluk hidup, tetapi kelompok sensitif sebaiknya mengurangi aktivitas fisik yang terlalu berat atau lama.

3. Tidak Sehat

Kualitas udara mulai merugikan makhluk hidup. Kelompok sensitif masih boleh melakukan aktivitas ringan dengan istirahat yang lebih sering serta perhatikan gejala seperti napas sesak atau batuk. Bagi penderita asma dianjurkan untuk membawa obat asma dan mengikuti petunjuk kesehatan asma. Bagi penderita penyakit jantung perlu memerhatikan gejala seperti detak jantung yang terlalu cepat, sesak napas, atau kelelahan yang berlebihan karena dapat menandakan masalah serius.

4. Sangat Tidak Sehat

Kualitas udara dapat memengaruhi risiko kesehatan bagi sebagian populasi yang terpapar. Bagi orang umum dianjurkan untuk menghindari kegiatan yang terlalu lama di luar ruangan, sedangkan bagi kelompok sensitif harus menghindari semua kegiatan di luar dan memperbanyak kegiatan di dalam ruangan.

5. Berbahaya

Kualitas udara dapat berdampak pada kesehatan serius dan perlu penanganan yang cepat. Bagi semua orang dianjurkan untuk menghindari semua kegiatan di luar ruangan, sedangkan untuk kelompok sensitif dianjurkan untuk mengurangi kegiatan dan tetap diam di dalam ruangan.

### **2.2.2 *Particulate Matter***

*Particulate matter* (PM) atau partikulat adalah campuran partikel di dalam udara dengan komponen yang memiliki karakteristik fisik dan kimia yang beragam. Bahaya paparan partikulat terhadap manusia beragam tergantung pada sumber, karakteristik fisik, komposisi kimia, dan ukurannya [4]. Sumber partikulat dapat berasal dari kegiatan manusia ataupun alami. Misalnya pembakaran batu bara, tempat pembangunan, kendaraan bermotor, dan rokok merupakan dari kegiatan manusia, sedangkan abu vulkanik dan debu. Kandungan utama dalam partikulat adalah debu, sulfat, amonia, natrium klorida, dan air. Sementara itu, kandungan partikulat yang diemisikan dari bahan bakar kendaraan bermotor mengandung karbon dan abu metalik. Selain itu, pembakaran tidak sempurna juga menghasilkan hidrokarbon aerosol. Ukuran partikulat berada dalam rentang 0,0002

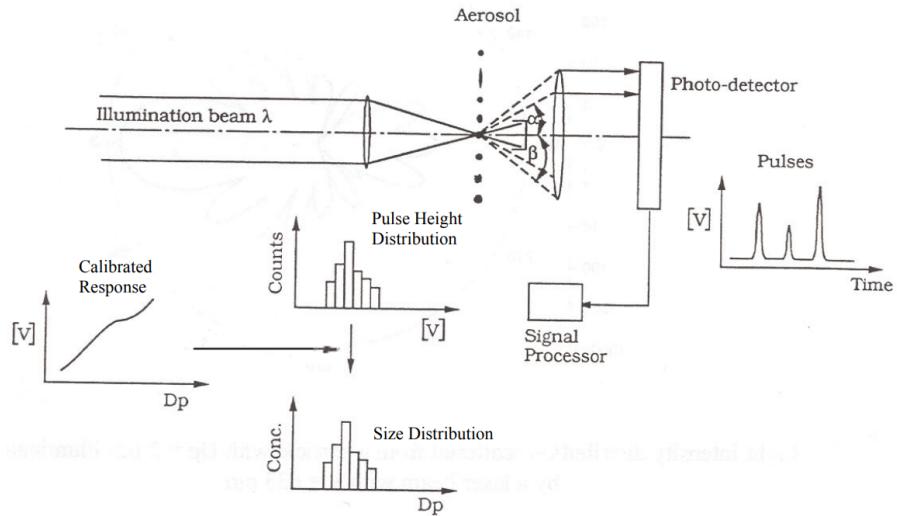
hingga 500 mikron. Dengan ukuran dan kandungannya tersebut, partikulat dapat bertahan di dalam udara dalam bentuk tersuspensi selama antara beberapa detik hingga beberapa bulan. Daya tahannya di udara bergantung pada kecepatan pengendapan di udara [16].

Secara umum, partikulat dapat dikelompokkan berdasarkan ukurannya. Terdapat dua ukuran yang menjadi fokus utama dalam pengelompokannya, yaitu diameter aerodinamik kurang dari atau sama dengan  $2,5 \mu\text{m}$  (PM2.5) dan kurang dari atau sama dengan  $10 \mu\text{m}$  (PM10). Partikulat dengan ukuran yang lebih kecil dapat bertahan di udara lebih lama karena beratnya yang lebih ringan. Dengan begitu, dapat tertutup udara lebih jauh pun. PM2.5 termasuk jenis dengan ukuran yang kecil sehingga dapat bertahan berminggu-minggu di udara dan dapat berada ratusan kilometer dari sumber aslinya. Sementara itu, PM10 termasuk jenis yang berat sehingga hanya dapat bertahan dalam hitungan menit atau jam dan dapat berada antara 100 meter hingga 50 km dari sumber aslinya [17]. Sumber dari kedua ukuran ini berbeda. PM2.5 dihasilkan dari emisi pembakaran bahan bakar, minyak, dan kayu. Sedangkan PM10 dihasilkan dari tempat pembangunan, pembakaran sampah, debu, kebakaran hutan, serbuk sari, dsb. Karena perbedaan ukuran yang dimilikinya, ketika terhirup manusia, kedua ukuran tersebut memberikan dampak yang berbeda. PM2.5 dapat mencapai paru-paru bagian dalam, sedangkan PM10 dapat mengendap di paru-paru dengan saluran yang lebih besar. Dampak dari pengendapan PM2.5 di paru-paru adalah bronkitis, asma, penyakit jantung, hingga dapat menyebabkan kematian dini. Di sisi lain, pengendapan PM10 di paru-paru dapat menyebabkan penyakit paru-paru obstruktif kronik (PPOK) dan asma [18].

### 2.2.3 *Optical Particle Counter*

*Optical Particle Counter (OPC)* merupakan instrumen yang digunakan untuk mengukur konsentrasi partikel di dalam udara. Penerapan OPC banyak dilakukan pada situasi ruang bersih yang mengharuskan udara terbebas dari partikel-partikel debu halus. Ukuran partikel yang dapat diukur dengan OPC bervariasi tergantung tujuan masing-masing. Akan tetapi, yang tersedia saat ini sudah mampu mengukur ukuran dalam rentang 50 nm hingga ratusan mikro meter. OPC menerapkan prinsip penghamburan cahaya terhadap partikel yang melewati penceran cahaya (umumnya laser). Udara masuk disedot dengan kipas ke sel pengukuran yang berisi detektor cahaya, yaitu foto dioda. Berdasarkan Gambar 2.1, hanya satu partikel yang disinari pada satu waktu di dalam sel pengukuran. Laser diarahkan ke aliran udara yang mengandung partikel-partikel. Laser yang terkena partikel tersebut terhambur ke detektor cahaya. Detektor cahaya ini akan mengubah cahaya terhambur menjadi sinyal listrik berupa pulsa tegangan. Sinyal kemudian dikuatkan dengan *amplifier*. Sinyal yang telah dikuatkan tersebut kemudian diolah untuk mendapatkan nilai konsentrasi partikel [19, 20].

Pulsa tegangan listrik yang dihasilkan detektor cahaya proporsi terhadap banyak-



Gambar 2.1. Pemrosesan data pada OPC

nya cahaya yang diterima. Sinyal yang telah dikuatkan kemudian diklasifikasikan menjadi tegangan diskrit untuk membentuk *pulse height distribution*. Pembentukan tersebut umumnya dilakukan oleh *pulse height analyzer (PHA)* atau *multichannel analyzer (MCA)*. Untuk mendapatkan nilai distribusi ukuran partikel, distribusi ketinggian pulsa tersebut dibandingkan dengan sebuah pembanding berupa tegangan *threshold* yang telah ditentukan dengan kalibrasi. Pada umumnya, nilai tegangan *threshold* didapatkan dengan kalibrasi *polystyrene latex*. Nilai ukuran partikel ini dapat diolah menjadi konsentrasi partikel dengan mengetahui laju aliran udaranya. Caranya adalah dengan menghitung jumlah dari peristiwa penghamburan per satuan waktu.

#### 2.2.4 Sensirion SPS30

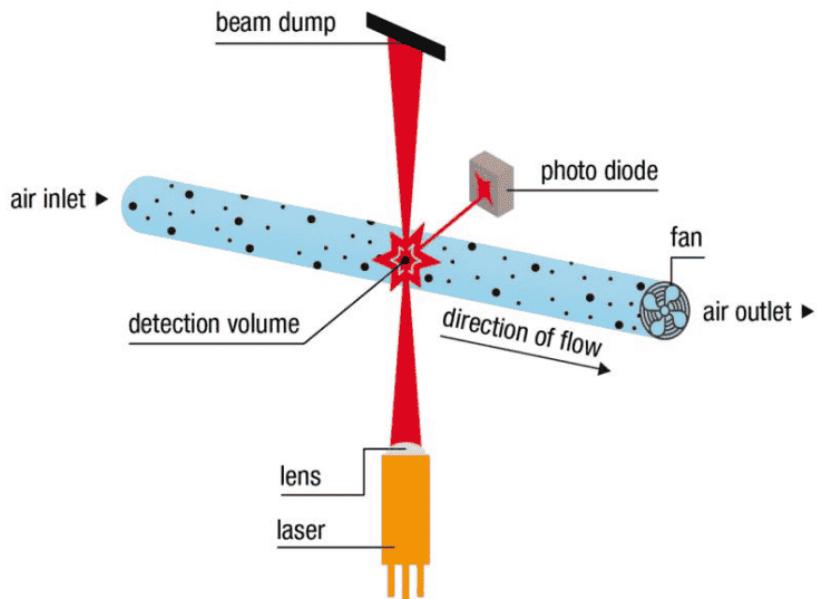
Sensirion SPS30 adalah sebuah sensor untuk mengukur konsentrasi partikulat keluaran dari Sensirion yang digunakan untuk kebutuhan pemantauan dan kendali kualitas udara. Fisik dari sensor ini dapat dilihat pada Gambar 2.2 di atas. Sensor ini dapat mengukur mulai dari PM1, PM2.5, PM4, hingga PM10. Tingkat presisi sensor ini adalah  $\pm 10\%$  untuk PM1 dan PM2.5, dan  $\pm 25\%$  untuk PM4 dan PM10. Proses pengukuran dilakukan dengan menggunakan prinsip penghamburan laser dan memanfaatkan inovasi teknologi *contamination-resistance* dari Sensirion. Dengan begitu, sensor ini dapat mengukur partikulat dengan sangat presisi dan memiliki jangka umur penggunaan yang panjang. Keandalan yang tinggi dan ukurannya yang kecil membuatnya banyak diaplikasikan pada berbagai macam kebutuhan, mulai dari HVAC hingga aplikasi pada industri [21].

Sensor Sensirion SPS30 adalah sebuah *optical particle counters (OPC)* yang berbasis pada penghamburan laser. Pendektsian partikulat di dalamnya ditunjukkan pada Gambar 2.3 Dalam manufakturnya, sensor ini dikalibrasi dengan sebuah instrumen pembanding yang secara reguler dijaga dengan baik. Instrumen tersebut adalah TSI Optical



Gambar 2.2. Sensirion SPS30

Particle Sizer Model 3330 atau TSI DustTrak™ DRX 8533. Dalam [22] disebutkan bahwa pengguna mengkalibrasi dengan menggunakan instrumen referensi yang sama, tetapi dapat terjadi perbedaan yang umumnya lebih dari 20%. Hal ini dapat terjadi karena beberapa faktor, seperti konfigurasi instrumen referensi yang berbeda atau cara kalibrasi ulang yang berbeda.

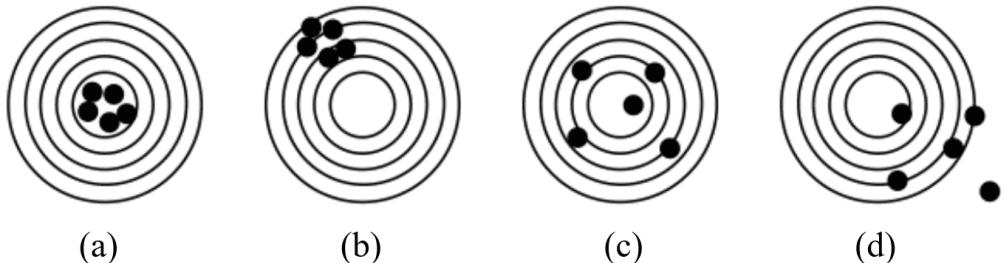


Gambar 2.3. Prinsip penghamburan laser pada sensor Sensirion SPS30

Ada berbagai macam fitur yang dimiliki Sensirion SPS30 untuk menunjang kemampuannya yang tinggi. Sensor ini memiliki kipas yang dapat digunakan untuk membersihkan partikel-partikel pada area sensor agar kualitas pengukuran presisi. Antarmuka sensor ini dapat menggunakan UART dan I2C. Terdapat pin selektor yang dapat dikonfigurasi untuk memilih antarmuka apa yang akan dipakai. Sensor ini juga memiliki tiga mode operasi, yaitu *idle*, *measurement*, dan *sleep*. Ketika pertama kali dinyalakan, sensor akan masuk ke mode *idle* dan siap untuk menerima perintah yang diberikan. Pada mode ini kipas dan laser dimatikan untuk mengurangi konsumsi daya. Mode *measurement* hanya dapat dimasuki melalui mode *idle*. Pada mode ini, pengukuran dijalankan dengan pembacaan baru tersedia setiap satu detik. Mode *sleep* hanya dapat dimasuki dari

mode *idle*. Pada mode ini hampir semua elektronik dimatikan termasuk laser dan kipas. Bahkan UART dan I2C juga dimatikan. Tujuannya adalah untuk mengurangi konsumsi daya [21].

### 2.2.5 Akurasi dan Presisi



Gambar 2.4. Perbedaan akurasi dan presisi: (a) akurat dan presisi, (b) tidak akurat tetapi presisi, (c) akurat tetapi tidak presisi, dan (d) tidak akurat dan tidak presisi

Akurasi adalah karakteristik yang penting dalam sebuah instrumen pengukuran. Tingkat akurasi merujuk seberapa dekat nilai hasil pengukuran terhadap nilai aslinya. Dengan begitu, tingkat akurasi pada instrumen pengukuran memberikan gambaran simpangan terjauh ketidaktepatan nilai yang diperoleh oleh instrumen terhadap nilai aslinya. Tingkat akurasi pada umumnya dituliskan dalam persentase skala penuh atau skala pembacaan. Nilainya bisa didapatkan dari menghitung nilai galat antara nilai acuan dengan nilai yang terukur [23]. Perhitungan galat ditunjukkan pada persamaan 2-3. Dari nilai persentase galat tersebut, nilai akurasi diperoleh dengan mengurangi nilai 100% dengan persentase galat yang ditunjukkan pada persamaan 2-4.

$$\%error = \left| \frac{nilai\ acuan - nilai\ terukur}{nilai\ acuan} \right| * 100\% \quad (2-3)$$

$$akurasi = 100\% - \%error \quad (2-4)$$

Presisi memiliki makna yang berbeda dengan akurasi. Presisi didefinisikan sebagai kemampuan sensor dalam menghasilkan nilai pengukuran yang berdekatan satu sama lain. Nilai presisi memberikan gambaran tentang seberapa dekat nilai pengukuran satu dengan yang lainnya di dalam kondisi yang sama. Apabila hasil pengukuran yang dilakukan berkali-kali saling berdekatan satu sama lain, maka sensor tersebut memiliki tingkat presisi yang tinggi. Sebaliknya, jika data pengukuran bervariasi atau tersebar secara acak, maka sensor tersebut memiliki tingkat presisi yang rendah.

Nilai presisi dapat direpresentasikan galat *repeatability* atau simpangan baku relatif (RSD), yaitu galat yang disebabkan ketidakmampuan sensor untuk memberikan nilai

pengukuran yang sama pada kondisi yang diasumsikan sama. Nilai tersebut dapat diperoleh dengan mencari rata-rata dan simpangan baku terlebih dahulu. Persamaan 2-5 digunakan untuk mencari rata-rata dan persamaan 2-6 digunakan untuk mencari simpangan baku.

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n} \quad (2-5)$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}} \quad (2-6)$$

Keterangan:

- $\bar{x}$  : rata-rata
- $x_i$  : sampel ke-*i*
- $n$  : jumlah sampel
- $\sigma$  : simpangan baku

Untuk mendapatkan nilai RSD, hasil kedua persamaan di atas dihitung dengan persamaan 2-7. Dari persamaan tersebut diketahui bahwa galat *repeatability* merupakan persentase antara simpangan baku dibandingkan dengan rata-rata. Semakin kecil nilai keduanya, maka semakin tinggi tingkat presisi sensor yang digunakan [24].

$$RSD = \frac{\sigma}{\bar{x}} \quad (2-7)$$

## 2.2.6 OLED SSD1306

OLED (*Organic Light Emitting Diode*) SSD1306 adalah sebuah layar *display* dengan panel OLED yang menggunakan IC *driver* SSD1306. Layar ini memiliki dimensi yang kecil yang umumnya berukuran 0,96 atau 1,3 inci. Terdapat beberapa pilihan resolusi dengan yang paling umumnya adalah 128x64 piksel. Tingkat kecerahan layar dapat diatur dengan menggunakan sinyal PWM 8 bit. Dengan begitu, kecerahan layar dapat diatur dari 0 hingga 255 tingkatan. Layar ini memiliki tiga pilihan antarmuka untuk berkomunikasi dengannya, yaitu antarmuka I2C, SPI, dan paralel [25]. Ketersediaannya di pasaran pada umumnya sudah berbentuk modul yang dilengkapi dengan pin-pin untuk menyesuaikan antarmukanya. Misalnya, ketika dipilih antarmuka I2C, maka modul tersebut memiliki pin VDD, GND, SDA, dan SCL seperti yang ditunjukkan Gambar 2.5.



Gambar 2.5. OLED SSD1306

### 2.2.7 Raspberry Pi 3 Model B

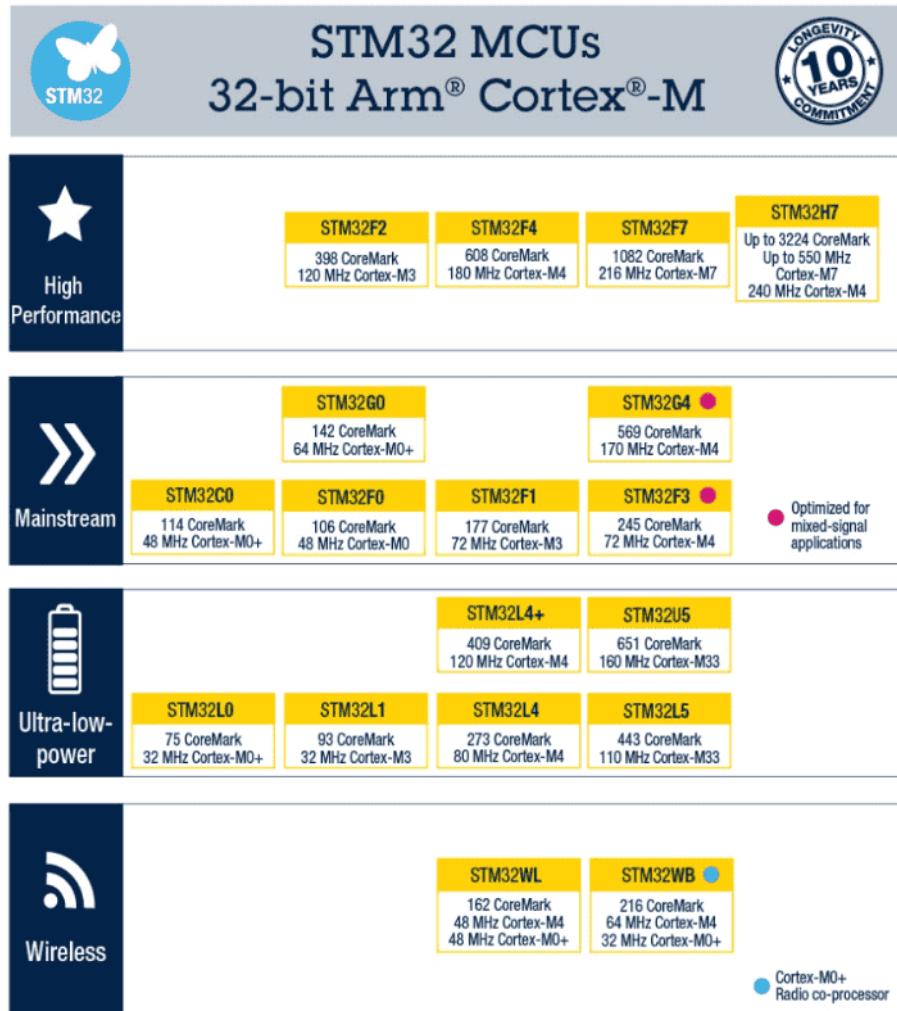
Raspberry Pi adalah sebuah *Single Board Computer (SBC)* yang dirancang oleh Raspberry Pi Foundation di Inggris. Komputer ini memiliki dimensi layaknya kartu debit. Tujuan Raspberry Pi dirancang adalah untuk menghadirkan platform komputer murah dan edukasi dalam pemrograman komputer. Sebagai sebuah SBC, Raspberry Pi memiliki fungsi seperti komputer umumnya. Raspberry Pi menggunakan *System-on-Chip (SoC)* dari Broadcom yang berbasis ARM dan *on-chip GPU* yang terintegrasi pada papan sirkuit [26].

Raspberry Pi 3 Model B adalah model pertama dari Raspberry Pi generasi ketiga. Seri ini memiliki SoC Broadcom BCM2837 dengan CPU 64 bit *quad core* 1.2 GHz, GPU VideoCore IV, dan RAM sebesar 1 GHz. Raspberry Pi generasi ketiga ini adalah Raspberry Pi pertama yang memiliki fitur konektivitas *wireless LAN* dan *Bluetooth*. Raspberry Pi 3 Model B dapat menjalankan berbagai macam sistem operasi, salah satunya adalah Raspberry Pi OS. Sistem operasi tersebut disimpan pada sebuah kartu microSD yang sekaligus juga menjadi tempat penyimpanan data-data lainnya. Dengan sistem operasi tersebut, Raspberry Pi 3 Model B dapat difungsikan selayaknya komputer *desktop*, namun dengan kecepatan komputasi yang terbatas [27].

### 2.2.8 STM32

STM32 adalah keluarga mikrokontroler dari STMicroelectronics yang berbasis pada prosesor 32 bit seri ARM Cortex-M. Keluarga mikrokontroler ini dikembangkan dengan dukungan fitur seperti performa sangat tinggi, kapabilitas *real-time*, memproses sinyal digital, operasi daya rendah, dan konektivitas, sembari tetap menjaga integrasi penuh dan kemudahan dalam pengembangannya di berbagai bidang. Sesuai dengan Gambar 2.6, mikrokontroler STM32 secara garis besar dibagi menjadi beberapa tipe sesuai dengan fitur utamanya, yaitu performa tinggi, *mainstream*, daya sangat rendah, dan nirkabel seperti yang ditunjukkan pada Gambar 2.6 [28]. Pemanfaatan ARM Cortex-M sebagai prosesor menjadikan seluruh keluarga STM32 pada umumnya berdaya rendah. Dengan ukurannya yang juga kecil, kode yang sederhana, kinerja tinggi, STM32 banyak

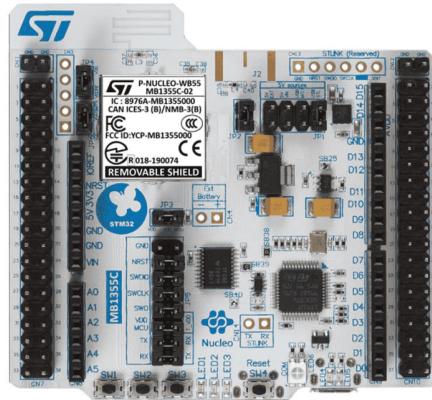
diimplementasikan pada sistem tertanam [29].



Gambar 2.6. Tipe-tipe STM32

### 2.2.9 P-NUCLEO-WB55

P-NUCLEO-WB55 adalah salah satu *development board* STM32 dengan tipe *wireless*. *Board* ini memiliki dukungan untuk konektivitas *ultra-low-power radio* sesuai dengan spesifikasi *Bluetooth Low Energy* (BLE) SIG v5.0 dan IEEE 802.15.4-2011. Sama seperti dengan keluarga STM32 lainnya, papan ini juga berdaya rendah dan tetap memiliki performa yang tinggi. Sesuai dengan namanya, papan ini menggunakan mikrokontroler STM32WB dalam paket VFQFPN68 dengan prosesor 32 bit ARM Cortex M0+ untuk *real-time radio layer*. Konektivitas *Bluetooth*-nya menggunakan *transceiver* RF 2.4 GHz. Pada papan ini juga sudah terdapat *on-board* ST-LINK/V2-1 *debugger/programmer* sehingga untuk memasukkan kode hanya memerlukan kabel USB Micro-B. Selain itu, fitur-fitur seperti I2C, USART, LPUART, SPI, 12-bit ADC, dll juga sudah tersedia. Bentuk fisik dari P-NUCLEO-WB55 dapat dilihat pada Gambar 2.7 [30].



Gambar 2.7. P-NUCLEO-WB55

### 2.2.10 STM32CubeIDE

STM32CubeIDE adalah platform pengembangan untuk mikrokontroler dan mikroprosesor STM32 berbasis bahasa C/C++ yang menghadirkan fitur konfigurasi periferal, pembuatan kode, *compile* kode. STM32CubeIDE dibangun berbasis Eclipse IDE sehingga dapat terintegrasi dengan banyak *plugins* dan dapat dijalankan di Windows, MacOS, dan Linux. STM32CubeIDE menggunakan GCC *toolchain* untuk pengembangannya dan GDB untuk *debugging*. Dengan IDE ini, pengembangan program untuk STM32 dapat dilakukan dengan satu *tool* ini saja. Oleh karena itu dapat menghemat waktu instalasi dan pengembangan [31].

Di dalam STM32CubeIDE terdapat banyak pilihan jenis mikrokontroler dan mikroprosesor STM32. Tak hanya itu, dimungkinkan pula untuk mengatur sendiri periferal-periferal yang akan digunakan dari jenis mikrokontroler yang telah dipilih. Contohnya adalah konfigurasi pin I/O mana yang akan digunakan atau tidak, konfigurasi khusus suatu pin digunakan sebagai apa, dll. Konfigurasi tersebut dilakukan dengan tampilan antarmuka yang menarik. Setelah melakukan konfigurasi tersebut, kode dasar untuk konfigurasi dalam bahasa C sudah tertulis secara otomatis. Walaupun diberikan kemampuan pengaturan yang fleksibel, IDE ini tetap memberikan kemudahan bagi penggunanya. Kode program utama dapat ditulis pada file *main.c* menggunakan teks editor yang telah tersedia dalam STM32CubeIDE. Kode yang telah selesai langsung dapat di-*compile* dan diunggah ke perangkat STM32.

### 2.2.11 Firmware

*Firmware* adalah sebuah program yang tertanam pada perangkat keras. *Firmware* tidak sama dengan perangkat keras (*hardware*) dan perangkat lunak (*software*). Secara sederhana, *firmware* bertindak sebagai otak dari perangkat keras yang berfungsi untuk mengendalikannya. Tanpa perintah *firmware*, perangkat keras tidak dapat berbuat apa-apa. *Firmware* memungkinkan perangkat keras untuk mengakses I/O, membaca sensor,

berkomunikasi dengan perangkat lain, dan sebagainya [9]. Pada perangkat keras yang sederhana, seperti *keyboard*, ketika perangkat dinyalakan, *firmware* mengirim instruksi kepada prosesor untuk menjalankan tugasnya karena tidak ada *software* yang menggantikannya. Pada perangkat yang lebih kompleks, seperti komputer, laptop, dsb, beberapa *firmware* melakukan tugasnya masing-masing dengan tujuan untuk *load operating system* (OS).

*Firmware* merupakan *file* program biner dalam format *.bin* yang berisi rentetan dari informasi dalam karakter biner. Program biner tersebut sering juga dikenal dengan bahasa mesin. Bahasa mesin tentu saja merupakan bahasa yang sulit dimengerti oleh bahasa manusia. Untuk itu, pengembangan *firmware* dapat dilakukan dengan bahasa pemrograman tingkat tinggi dan kemudian diubah menjadi bahasa mesin yang berisi rentetan bilangan biner tersebut. Pada umumnya, *firmware* disimpan di dalam *non-volatile memory*, yaitu sebuah memori yang tetap dapat menyimpan datanya ketika daya mati. Berdasarkan tempat penyimpanan dan kompleksitas dari kegunaannya, *firmware* dibagi menjadi tiga level, sebagai berikut:

1. *Low-Level Firmware* : yaitu level *firmware* yang disimpan di *non-volatile memory chip* seperti ROM, PROM, dan struktur PLA (*Programmable Logic Array*). Pada level ini, *firmware* disimpan pada *read-only chip* yang tidak dapat di-update, sehingga *firmware* ini merupakan bagian dari *hardware* itu sendiri.
2. *High-Level Firmware* : yaitu disimpan pada *flash memory* sehingga dapat diupdate. Instruksinya lebih kompleks dari *low-level firmware*.
3. *Subsystems* : yaitu perangkat atau unit yang merupakan bagian semi independen dari sistem yang lebih besar. Sering kali level ini menyerupai perangkat itu sendiri karena memiliki mikrokode yang tertanam di *flash chips*, CPU, dan *high-level firmware* [32].

### 2.2.12 Bahasa C

Bahasa C adalah bahasa pemrograman tingkat tinggi serba guna terstruktur yang pertama kali dikembangkan oleh Dennis M. Richie di Laboratorium Bell pada tahun 1972 dalam usahanya membuat sistem operasi UNIX. Saat ini, bahasa C banyak digunakan untuk membuat program dalam berbagai bidang. Namun, karena sangat berguna untuk membuat *compiler* dan sistem operasi, bahasa C adalah bahasa pemrograman sistem yang paling populer. Meskipun merupakan salah satu bahasa tingkat tinggi, bahasa C dapat menangani aktivitas tingkat rendah. Kecepatan kode yang dihasilkan dengan bahasa C dapat menyamai program yang ditulis menggunakan bahasa *assembly*. Contoh penggunaan bahasa C, yaitu sistem operasi, *compiler*, *assembler*, editor teks, *database*, dan lain-lain [33].

Dalam bahasa C terdapat berbagai macam tipe data. Contohnya yang fundamental adalah *integer*, *float*, dan *characters*. Tipe-tipe data tersebut dapat diturunkan lagi dengan *arrays*, *structures*, *unions*, dan *pointer*. Tipe data digunakan untuk menentukan jenis dari variabel. Hal ini memengaruhi ukuran memori yang digunakan oleh variabel untuk menyimpan suatu nilai karena setiap tipe data membutuhkan ukuran memori yang berbeda-beda. Fungsi-fungsi dalam bahasa C juga didefinisikan dengan menentukan tipe datanya. Fungsi digunakan untuk menjalankan perintah tertentu yang telah didefinisikan di dalamnya. Fungsi tersebut dapat mengembalikan nilai di dalam suatu tipe data, *structure*, *union*, atau *pointers* [34].

```

1 #include <stdio.h>
2 int a = 5;
3 int main() {
4     // Program bahasa C
5     printf("Halo, dunia!");
6     return 0;
7 }
```

Bahasa C dapat ditulis pada sebuah teks editor. Kode bahasa C ditulis secara tersusunan, seperti kode di atas. Bagian dasar dalam struktur bahasa C yang diperlukan agar dapat berjalan adalah *preprocessor commands*, fungsi, variabel, komentar, *expression*, dan *statements*. Kode dengan struktur tersebut dapat dimengerti oleh manusia tetapi tidak oleh mesin. *Source code* tersebut harus diubah menjadi bahasa mesin atau dikenal dengan sebutan *di-compile*. Oleh karena itu, dibutuhkan sebuah *compiler* agar program dapat dieksekusi oleh komputer. *Compiler* yang banyak digunakan dan bisa didapatkan secara gratis adalah *GNU C/C++ compiler* [33].

### 2.2.13 Bilangan *Floating Point*

*Floating point number* atau bilangan titik mengambang adalah format bilangan yang digunakan untuk merepresentasikan bilangan *real* atau desimal yang sangat besar atau kecil. Bilangan ini dituliskan dalam bentuk eksponensial, sebagai berikut:

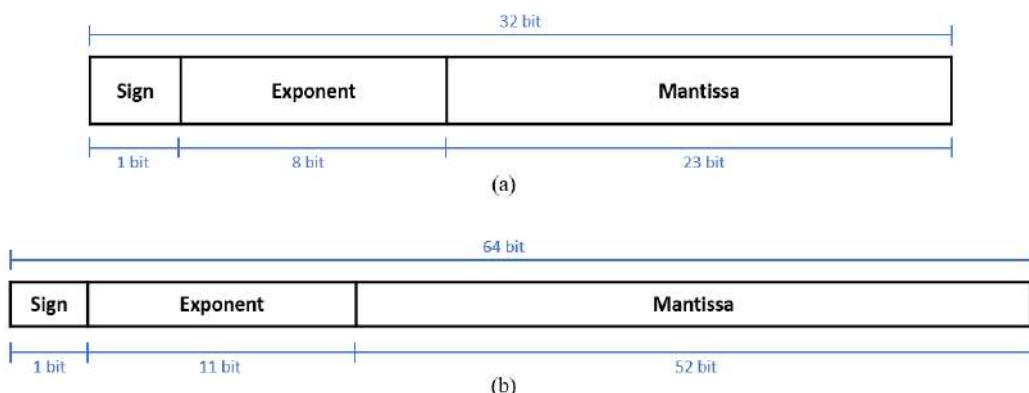
$$\pm M * B^{\pm E} \quad (2-8)$$

- *M* : signifikan yang sering disebut dengan *mantissa*
- *B* : basis bilangan
- *E* : eksponen

Dalam dunia komputasi, bilangan *floating point* di sebuah memori disimpan dalam bentuk bilangan biner. Terdapat standar yang mengaturnya agar penulisannya sera-

gam, yaitu IEEE 754 [35]. Pada standar tersebut dijelaskan bahwa bilangan bilangan ini diubah dalam bentuk biner dengan panjang 32, 64, atau 128 bit. Pada umumnya, bilangan biner sepanjang 32 bit ini dikenal dengan *single precision* dan yang 64 bit dikenal dengan *double precision*. Dalam bahasa pemrograman, masing-masing dikenal dengan tipe data *float* dan *double* [36]. *Floating point* dalam bentuk biner terbagi menjadi tiga bagian, yaitu:

1. *Sign* : merupakan bit penanda positif atau negatif. 0 untuk positif dan 1 untuk negatif.
2. *Eksponen terbias* : merupakan bit eksponen yang terdiri dari 8 bit untuk *single precision* dan 11 bit untuk *double precision*. Bit eksponen harus dapat merepresentasikan positif atau negatif sehingga dalam penggunaannya ditambah dengan bias, yaitu 127 untuk *single precision* dan 1023 untuk *double precision*
3. *Mantissa ternormalisasi* : merupakan bit bilangan signifikan dalam *floating point* dengan hanya 1 bilangan di depan koma.



Gambar 2.8. Diagram bilangan biner untuk (a) *single precision* dan (b) *double precision* pada *floating point number*

Dalam standar IEEE 754 ini ada beberapa bilangan yang khusus, yaitu disajikan dalam tabel berikut.

Tabel 2.3. Bilangan *floating point* khusus

Jenis	Eksponen	Mantissa
0	0	0
Tak hingga	255	0
Bilangan denormalisasi	0	bukan 0
<i>Not a Number</i> (NaN)	255	bukan 0

### 2.2.13.1 Konversi Bilangan *Floating Point*

Konversi bilangan *floating point* menurut standar IEEE 754 dapat dilakukan dengan beberapa langkah-langkah [20]. Perlu diperhatikan bahwa bilangan desimal memiliki dua bagian, yaitu angka di depan dan di belakang koma. Langkah-langkahnya adalah sebagai berikut:

1. Bit pertama diisi dengan 0 jika bilangan positif atau 1 jika negatif
2. Bagi angka di depan koma dengan 2 terus menerus sampai tidak dapat dibagi lagi. Simpan setiap sisa pembagiannya. Sisa pembagian terakhir diletakkan di paling depan.
3. Kali angka di belakang koma dengan dua terus menerus hingga perkaliannya menghasilkan bilangan dengan angka di depan koma adalah 1. Simpan setiap angka di depan koma dari semua perkalian. Hasil perkalian pertama diletakkan di paling depan.
4. Gabungkan dua bilangan biner tersebut dengan dipisahkan dengan tanda koma.
5. Normalisasi bilangan tersebut menjadi seperti  $1, bbb\dots * 2^E$  dengan 'b' adalah bilangan biner dan 'E' adalah eksponen.
6. Bagian eksponen terbias didapat dari menambahkan E dengan bias, yaitu 127 untuk *single precision* atau 1023 untuk *double precision*. Kemudian ubah menjadi bilangan biner.
7. Hilangkan angka 1 di depan koma dari bilangan biner sebelumnya kemudian tambahkan 0 di belakangnya hingga mencapai 23 bit untuk *single precision* atau 1023 untuk *double precision* agar mendapatkan *mantissa* ternormalisasi.

Langkah-langkah konversi bilangan *floating point* dari bentuk basis biner ke basis desimal adalah sebagai berikut:

1. Kelompokkan bilangan biner menjadi tiga, yaitu bit tanda (*sign*), eksponen, dan *mantissa*.
2. Lihat bit tanda (*S*). Jika 0 maka positif, atau jika 1 maka negatif.
3. Eksponen (*E*) didapat dari mengurangi nilai bit eksponen terbias dengan biasnya, yaitu 127 untuk *single precision* atau 1023 untuk *double precision*.
4. Konversi nilai bit *mantissa* (*m*) menjadi *mantissa* basis desimal (*M*) dengan cara sebagai berikut:

$$M = \sum_{i=0}^{n-1} b_i * 2^{-(i+1)} \quad (2-9)$$

Keterangan:

- $M$  : mantissa dalam basis desimal
  - $n$  : jumlah bit mantissa, yaitu 23 untuk *single precision* atau 52 untuk *double precision*
  - $b$  : bit mantissa ke- $i$
5. Bilangan desimal dalam basis desimal ( $D$ ) diperoleh dengan persamaan berikut:

$$D = (-1)^s * (1 + M) * 2^E \quad (2-10)$$

### 2.2.14 Bilangan Integer

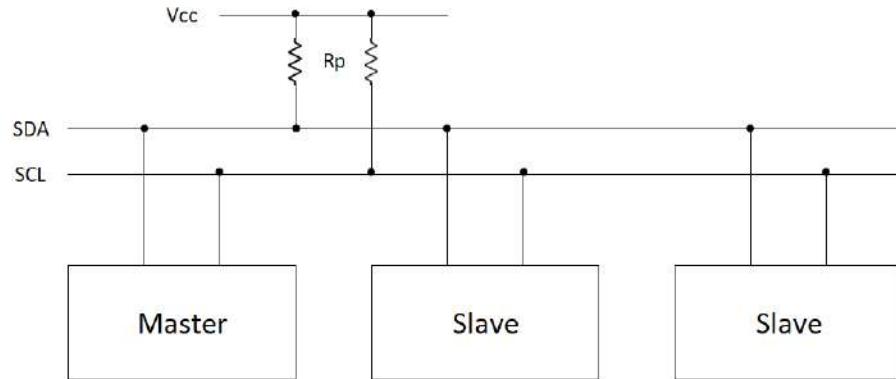
*Integer* secara harfiah adalah bilangan bulat, baik positif maupun negatif. Dalam sebuah memori komputer, nilai *integer* disimpan dalam bilangan biner. Ukurannya bisa bermacam-macam, dari 8 bit, 16 bit, hingga 32 bit. Ukuran tersebut mengindikasikan seberapa besar nilai angka yang dapat disimpan sebuah memori. Jadi, memori 16 bit dapat menyimpan nilai *integer* dari sejumlah  $2^n = 2^{16} = 65536$  bilangan. Format penyimpanan bilangan *integer* pun hanya sesederhana konversi bilangan bulat dalam basis desimal menjadi basis biner. Contohnya adalah angka 17 dikonversi menjadi angka biner 16 bit menjadi 0000 0000 0001 0001. Akan tetapi, konversi bilangan bulat basis desimal menjadi biner untuk bilangan negatif sedikit berbeda [37].

Bilangan negatif dalam biner tidak sesederhana ditandai dengan MSB bernilai 1. Ada beberapa cara untuk merepresentasikan bilangan negatif dalam bilangan biner. Namun, cara yang digunakan oleh sebuah komputer adalah menggunakan metode komplemen 2. Metode ini bekerja dengan menginversi bilangan biner positif dan menambahkannya dengan 1. Contohnya, angka -17 didapatkan dengan inversi biner angka 17, yaitu 1111 1111 1110 1110 yang ditambah dengan 1. Sehingga -17 dalam bilangan biner komplemen 2 adalah 1111 1111 1110 1111. Metode komplemen 2 ini digunakan agar tidak ada bilangan -0.

Dalam bahasa pemrograman, seperti bahasa C, *integer* dapat dipilih apakah bertanda (*signed*) atau tidak bertanda (*unsigned*). Pemilihan ini dilakukan dengan memilih tipe data 'int8\_t', 'int16\_t', atau 'int32\_t' untuk *signed integer* dan 'uint8\_t', 'uint16\_t', atau 'uint32\_t' untuk *unsigned integer* dengan angka di sana adalah ukuran memori. Oleh karena itu, memori *unsigned integer* dapat menyimpan nilai dari 0 hingga  $2^n - 1$ , sedangkan *signed integer* dapat menyimpan nilai dari  $-2^{n-1}$  hingga  $-2^{n-1} - 1$ .

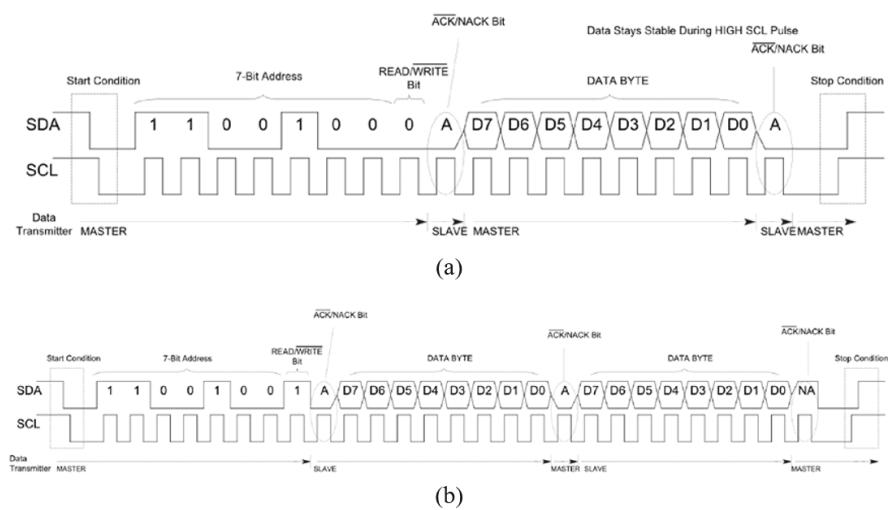
### 2.2.15 I2C

I2C (*Inter-Integrated Circuit*) merupakan salah satu protokol komunikasi serial yang pertama kali dikembangkan oleh Phillips pada tahun 1982. Protokol komunikasi



Gambar 2.9. *Pull-up resistor* pada I2C

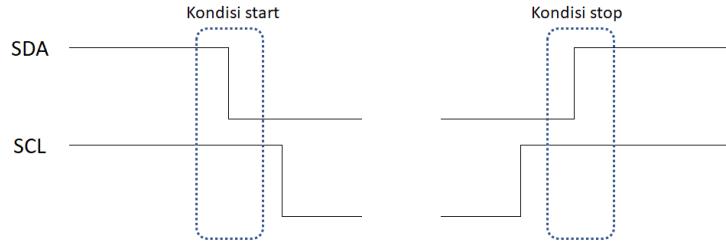
I2C membutuhkan dua jalur, yaitu jalur SDA (*Serial Data*) dan SCL (*Serial Clock*). SDA merupakan jalur yang dilalui oleh data sedangkan SCL merupakan jalur untuk *clock*. Karena I2C merupakan komunikasi sinkron, maka bit *clock* juga dikirimkan sehingga pesan yang dikirim dapat dimengerti oleh penerima. Kecepatan transmisi pada protokol ini umumnya adalah 100 kHz. Namun, juga dapat lebih cepat lagi, yaitu dengan *fast mode* mencapai 400 kHz hingga *ultra-fast mode* mencapai 5 MHz [38]. Jalur SDA digunakan secara bergantian antar perangkat sehingga jenis komunikasi pada I2C ini bersifat *half duplex bidirectional*. Protokol I2C merupakan komunikasi yang menerapkan *master* dan *slave*. Bit *clock* pada SCL dihasilkan oleh *master* yang di sini bertindak sebagai pengendali. Jumlah *master* dan *slave* dalam satu bus dapat lebih dari satu. *Driver* bus I2C ini bersifat *open drain*. Artinya adalah I2C hanya dapat *drive* sinyal menjadi *low* dan tidak dapat *drive* sinyal menjadi *high*. Oleh karena itu, dibutuhkan sebuah *pull-up resistor* pada masing-masing jalur SDA dan SCL dalam satu bus. Dengan demikian, ketika tidak ada data yang hendak dikirim, maka normalnya sinyal pada kedua jalur bernilai *high* [39].



Gambar 2.10. Diagram bit ketika (a) *master transmit* ke *slave* dan (b) *master receive* data dari *slave*

Pada protokol komunikasi I2C, sinyal pada jalur SDA ditransmisikan dalam suatu urutan. Pada umumnya, urutan sinyal tersebut terbagi menjadi dua bagian, yaitu *byte* alamat dari perangkat *slave* dan satu atau lebih *byte* data. Di akhir setiap *byte*, terdapat bit ACK/NACK. Untuk memulai dan mengakhiri transmisi I2C terdapat kondisi *start* dan *stop*. Untuk memastikan bahwa komunikasi dengan I2C berhasil atau tidak, urutan sinyal ini harus diperhatikan.

### 2.2.15.1 Kondisi Start dan Stop



Gambar 2.11. Kondisi *start* dan *stop*

Kedua jalur SDA dan SCL pada kondisi normal berada pada kondisi *high*. Komunikasi dimulai dengan mengirimkan kondisi *start*. Kondisi *start* ini diinisiasi oleh perangkat *master*. Untuk masuk ke kondisi *start*, sinyal pada jalur SDA diubah dari *high* ke *low* ketika sinyal pada jalur SCL bernilai *high*. Selang beberapa saat, bit *clock* dimulai pada jalur SCL dan data juga ditransmisi pada jalur SDA. Telah dijelaskan sebelumnya bahwa dalam satu bus bisa ada lebih dari satu *master*. Jika dua *master* mencoba untuk memulai transmisi pada waktu yang sama, *master* yang pertama mengubah sinyal pada SDA menjadi *low* akan pegang kendali pada bus tersebut. Selama satu *master* memegang kendali pada bus, *master* lain tidak dapat mengendalikan bus tersebut.

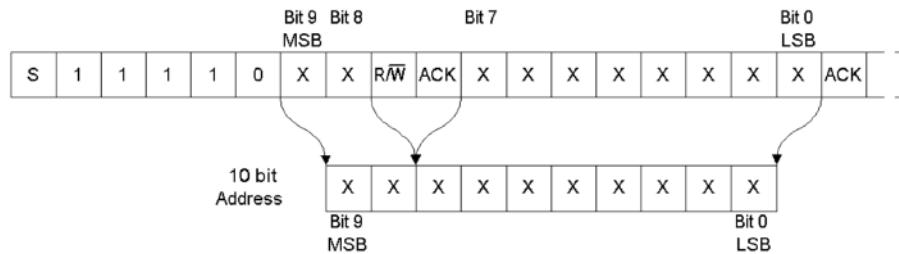
Saat seluruh data telah ditransmisikan, *master* harus mengirimkan kondisi *stop* untuk mengakhiri komunikasi I2C yang sedang berlangsung. Sama seperti kondisi *start*, kondisi ini juga diinisiasi oleh perangkat *master*. Kondisi *stop* terjadi ketika sinyal pada jalur SDA diubah dari *low* ke *high* saat sinyal pada jalur SCL bernilai *high*. Perlu diperhatikan bahwa pada transmisi data, sinyal pada jalur SDA hanya dapat berubah ketika sinyal SCL bernilai *low*. Dengan dikirimkannya kondisi *stop* ini, maka kedua jalur kembali bernilai *high* dan *master* tidak lagi memegang kendali pada bus [40, 41].

### 2.2.15.2 Alamat

Untuk mengidentifikasi perangkat mana yang akan saling berkomunikasi, setiap perangkat *slave* memiliki alamat. Dalam protokol I2C, alamat *slave* terdiri dari bilangan biner 7 bit ataupun 10 bit. Bit alamat ini merupakan pesan atau sinyal pertama yang ditransmisi *master* setelah kondisi *start*. Pada alamat 7 bit, bit ke-8 diisi dengan bit R/W (*Read/Write*). Bit R/W berguna untuk menentukan apakah *master* ingin mengirim data

ke *slave* atau ingin menerima data dari *slave*. Jika *master* ingin mengirim (*write*) pesan ke *slave*, maka bit tersebut bernilai *low*. Sebaliknya, jika *master* ingin membaca (*read*) data dari *slave*, maka bit tersebut bernilai *high*.

Alamat 10 bit diperkenalkan karena keterbatasan jumlah pada alamat 7 bit. Penggunaan alamat 10 bit sedikit berbeda dengan 7 bit. Seperti yang telah dijelaskan sebelumnya, pesan dikirim setiap 1 *byte* yang terdiri dari 8 bit. Untuk itu, alamat 10 bit ini dipecah menjadi dua *frame* data. Namun, penggunaannya untuk *master* sebagai *transceiver* dan *receiver* data berbeda.



Gambar 2.12. Urutan bit dalam alamat 10 bit

1. *Transceiver*: *Frame* pertama bernilai '11110XX(R/W)'. XX tersebut adalah 2 bit MSB dari alamat 10 bit. *Frame* kedua bernilai sisa 8 bit alamat tersebut. Pada mode *transceiver* ini, bit R/W pada *frame* pertama bernilai 0. Setiap *frame* tersebut juga ditambah bit ACK/NACK.
2. *Receiver*: Untuk masuk ke mode *receiver*, bit R/W pada *frame* pertama tidak langsung diganti menjadi 1. Perlu diperhatikan bahwa untuk dapat mengirim *frame* kedua, maka bit R/W pada *frame* pertama harus bernilai 0. Jadi kedua *frame* tersebut ditransmisikan sama seperti pada mode *transceiver*. Kemudian dilakukan kodisi *repeated start*. Setelah itu, diikuti dengan mentransmisikan *byte* yang sama dengan *frame* pertama tetapi bit R/W diganti menjadi 1 [42].

### 2.2.15.3 Bit Data

Dalam protokol komunikasi I2C, setiap *frame* data terdiri dari 1 *byte* data pesan dan 1 bit ACK/NACK. Bit ACK/NACK selalu mengikuti di setiap 1 *byte* data yang di-transmisi. Oleh karena itu, 1 *frame* data pada I2C terdiri dari 9 bit. Tidak ada maksimal jumlah data dalam sekali transmisi. Bit pada jalur SDA disinkronkan dengan *clock* pada jalur SCL. Perubahan sinyal *high* atau *low* pada jalur SDA terjadi ketika sinyal pada jalur SCL bernilai *low*. Sinyal *low* pada jalur SCL menandakan nilai bit baru. Dengan begitu, penerima akan membaca nilai sinyal pada jalur SDA ketika sinyal SCL bernilai *high*.

#### 2.2.15.4 Bit ACK/NACK

Setiap satu *byte* data dikirim, perangkat penerima akan mengirim bit ke 9, yaitu bit *Acknowledgement* (ACK) atau *Negative Acknowledgement* (NACK). Perangkat penerima mengirim bit ACK dengan membuat sinyal pada jalur SDA menjadi *low*, sedangkan bit NACK didapat dengan perangkat penerima membiarkan SDA berada pada kondisi normalnya, yaitu *high*. Pada saat kondisi *start* dikirim oleh *master*, data pertama yang dikirim adalah alamat *slave*. Perangkat *slave* dengan alamat yang sesuai kemudian memberikan nilai *low* pada jalur SDA (ACK), sedangkan *slave* yang tidak sesuai akan mengabaikan jalur SDA (NACK). Bit ACK ini mengindikasikan bahwa data telah dikirim dan diterima dengan sukses. Dengan begitu, transmisi dapat dilanjutkan, baik itu *byte* pesan selanjutnya, kondisi stop, atau bahkan *repeated start*. Sementara itu, pada [43] bit NACK memiliki beberapa maksud, yaitu:

1. Bit NACK setelah *byte* alamat berarti tidak ada perangkat *slave* yang memiliki alamat tersebut.
2. Bit NACK ketika *master* mengirim data berarti *slave* tidak mengenali perintah yang dikirim atau sedang tidak dapat menerima data.
3. Bit NACK ketika *master* sebagai penerima data berarti *master* sudah tidak ingin menerima data lagi.

#### 2.2.15.5 *Repeated Start*

Setelah bit ACK dikirim, kondisi *repeated start* terjadi ketika perubahan nilai sinyal pada jalur SDA dari *low* ke *high* saat jalur SCL bernilai *low*. Dengan begitu, kedua jalur SDA dan SCL bernilai *high* tanpa masuk ke kondisi *stop*. Perlu diingat bahwa kondisi *stop* terjadi saat sinyal pada jalur SDA berubah dari *low* ke *high* ketika jalur SCL bernilai *high*. Pada kondisi ini, *master* lain pada bus yang sama tidak dapat memegang kendali pada bus karena kondisi *stop* tidak dikirim. Oleh karena itu, komunikasi pada bus ini masih dipegang oleh *master* yang sama dan dapat mengirim kondisi *start* lagi. Kondisi *repeated start* ini dapat digunakan pada kasus *master* ingin mengubah pesan tanpa memberikan perangkat *master* lain kesempatan untuk memegang kendali jalur komunikasi di bus [40].

#### 2.2.15.6 *Clock Stretching*

Pada protokol komunikasi I2C, bit *clock* pada jalur SCL dikendalikan oleh perangkat *master*. Namun, pada kasus tertentu, perangkat *slave* dapat mengendalikan jalur SCL secara sementara. Hal ini disebut dengan *clock stretching*. *Clock stretching* dilakukan oleh *slave* pada kasus-kasus seperti *master* sangat cepat sehingga ada operasi-operasi pada *slave* yang belum selesai. Untuk melakukan *clock stretching*, perangkat *slave* akan

menahan sinyal *clock* agar tetap *low*. Dengan demikian, *master* harus menunggu jalur SCL dilepas oleh *slave* agar dapat melanjutkan transmisinya. Akan tetapi, hal ini dapat menyebabkan masalah karena tidak semua perangkat mendukung fitur yang satu ini [43].

### 2.2.16 UART

UART (*Universal Asynchronous Receiver-Transmitter*) adalah salah satu protokol komunikasi serial asinkron yang dapat digunakan untuk komunikasi antara dua perangkat secara *half-duplex*. Contohnya adalah komunikasi antara mikrokontroler dengan komputer, sensor, atau sesama mikrokontroler. Karena asinkron, bit *clock* tidak dikirimkan bersamaan dengan bit data. Kedua perangkat harus diatur untuk memiliki kecepatan transmisi (*baudrate*) yang sama, misal 9600 *bauds* atau 115200 *bauds*. Komunikasi antara dua perangkat hanya dihubungkan dengan dua kabel, yaitu Tx dan Rx. Pin Tx pada satu perangkat dihubungkan dengan pin Rx pada perangkat lainnya. Pin Tx bertugas untuk mengirim data dari satu perangkat, sedangkan pin Rx bertugas untuk menerima data tersebut [44].



Gambar 2.13. Diagram urutan bit dalam UART

Komunikasi UART secara normal ketika tidak mengirim data akan memiliki logika *high*. Transmisi dalam komunikasi UART terjadi dengan urutan bit seperti yang ditunjukkan pada Gambar 2.13. Untuk memulai transmisi, sinyal pada jalur transmisi akan berubah menjadi *low* atau yang lebih dikenal dengan *start bit*. Kemudian diikuti dengan bit-bit data yang dapat berjumlah 5 hingga 9 bit. Setelah rentetan bit data, dapat diisi dengan bit paritas atau pun tidak. Jika menggunakan bir paritas, maka bit data hanya dapat berjumlah hingga 8 bit. Bit paritas berfungsi untuk mendeteksi apakah ada *error* atau tidak pada data yang dikirim. Bit paritas itu sendiri terdiri dari dua jenis, yaitu bit paritas ganjil dan genap. Bit paritas ganjil akan membuat jumlah seluruh logika *high* menjadi ganjil. Bit paritas genap akan membuat jumlah seluruh logika *high* menjadi genap. *Receiver* dapat mendeteksi apabila jumlah logika *high* tersebut sesuai atau tidak dengan yang seharusnya. Setelah bit paritas, rentetan bit ditutup dengan *stop bit* yang ditandai dengan membuat logika *high* selama 1 atau 2 bit. Semua konfigurasi seperti *baudrate*, jumlah data, hingga bit paritas harus ditentukan pada kedua perangkat agar pesan yang diterima dapat terbaca sesuai dengan yang dikirim [45].

### 2.2.17 RealTerm

RealTerm adalah perangkat lunak berupa terminal serial yang didesain untuk menangkap, mengirim, dan men-debug data pada komunikasi serial. Umumnya, RealTerm

ini digunakan untuk komunikasi UART. RealTerm tersedia pada sistem operasi Windows. Fitur di dalamnya sangat beragam, seperti dapat menampilkan data dalam berbagai macam format, *timestamp*, dan menangkap data menjadi *file* teks. Untuk menggunakan-nya, konfigurasi komunikasi serial perlu disesuaikan agar data dapat ditampilkan dengan benar [46].

### 2.2.18 Grabserial

Grabserial adalah sebuah program yang digunakan untuk membaca data pada *port* serial dan menyimpannya di komputer. Grabserial berjalan pada sistem operasi Linux. Karena berfungsi untuk membaca data dari komunikasi UART, pada saat menjalankan program ini, perlu disesuaikan *baudrate* dan parameter-parameter lainnya. Program ini dijalankan dengan mengirim perintah "grabserial" pada terminal yang diikuti dengan pa-rameter lain di belakangnya. Grabserial juga menawarkan fitur *timestamp* yang dapat mencatat kapan setiap data dibaca. Selain itu, masih banyak lagi fitur yang tersedia untuk menyesuaikan bagaimana data akan dibaca dan disimpan [47].

### 2.2.19 Python

Python adalah bahasa pemrograman tingkat tinggi yang saat ini sangat populer. Python dikembangkan oleh Guido van Rossum pada tahun 1991. Python merupakan bahasa pemrograman *interpreted*, yaitu kode dieksekusi per baris untuk mengeluarkan *output*-nya. Sintaksis yang dimiliki Python cukup mudah untuk dibaca, sehingga mem-buatnya dapat mudah dimengerti oleh pemula. Sintaksis tersebut juga memungkinkan ko-de yang dihasilkan memiliki baris yang lebih sedikit dari beberapa bahasa pemrograman lainnya [48].

Python dapat diimplementasikan secara luas, mulai dari penyelesaian persamaan matematika sederhana, pengembangan web, pengembangan aplikasi, hingga pengolah-an data. Kemampuannya tersebut didukung dengan tersedianya berbagai macam *library*, seperti NumPy, Pandas, dan Matplotlib. Komunitasnya yang besar juga menjadi kele-bihan dari Python. Komunitas tersebut dapat menjadi sarana untuk berdiskusi. Dengan demikian, baik pemula maupun ahli, dapat belajar dan menyelesaikan masalah dengan berdiskusi secara langsung atau melihat diskusi orang lain terkait masalah serupa.

## 2.3 Analisis Perbandingan Metode

Berdasarkan hasil analisis pada Tinjauan Pustaka, pada penelitian ini akan dikembangkan *firmware* yang berfungsi untuk mengukur konsentrasi partikulat di udara. Setelah *firmware* selesai dikembangkan, dilakukan analisis hasil pengukuran, dan analisis protokol komunikasi I2C yang digunakan antara mikrokontroler dengan sensor. Hasil pengukuran dianalisis untuk mengetahui konsentrasi partikulat di lokasi pengujian. Ni-

lainnya akan dibandingkan dengan indikator standar yang dipakai di Indonesia. Indikator tersebut tertera dalam Peraturan Menteri KHLK nomor 14 tahun 2020 tentang ISPU.

Mikrokontroler yang digunakan berbasis pada STM32. Mikrokontroler ini digunakan karena terdapat berbagai konfigurasi yang dapat dilakukan secara manual, yang mana tidak dapat dilakukan jika menggunakan mikrokontroler lain seperti Arduino atau ESP32. Dengan begitu, perlu pemahaman yang lebih mendalam untuk menggunakan-nya. Sensor yang digunakan di sini adalah Sensirion SPS30, yaitu sebuah sensor yang berfungsi untuk mengukur konsentrasi jumlah dan massa partikulat di udara. Kelebihan sensor ini dibanding sensor lainnya adalah konsumsi daya yang rendah, ukurannya yang *compact*, dan kemampuannya dalam mengukur 10 parameter nilai, yaitu konsentrasi massa (PM1, PM2.5, PM4, dan PM10), konsentrasi jumlah (PM0.5, PM1, PM2.5, PM4, dan PM10), dan *typical particle size*. Salah satu antarmuka yang dapat digunakan sensor ini adalah protokol komunikasi I2C. Agar *firmware* dapat dikembangkan, diperlukan studi literatur terlebih dahulu, khususnya mengenai protokol komunikasi I2C tersebut. Langkah selanjutnya adalah mengimplementasikan protokol komunikasi I2C ke dalam *source code* yang juga disesuaikan dengan *datasheet* sensor SPS30. Maka dari itu, analisis mengenai I2C juga akan dilakukan pada penelitian ini.

## **BAB III**

### **METODE PENELITIAN**

#### **3.1 Alat dan Bahan Tugas akhir**

##### **3.1.1 Alat Tugas akhir**

Pada penelitian ini, alat-alat yang digunakan adalah sebagai berikut:

1. Laptop dengan spesifikasi minumum sistem operasi Windows 10, RAM 2 GB dengan rekomendasi RAM 4GB, 6 GB ruang penyimpanan kosong untuk yang bukan pengembang STM32 MPU OpenSTLinux Distribution atau 15 GB untuk penggunaan STM32 MPU OpenSTLinux Distribution. Pada penelitian ini digunakan laptop dengan OS Windows 11, prosesor Intel Core i7-10510U CPU, dan RAM 8 GB.
2. P-Nucleo-WB55 sebagai *development board*
3. Sensirion SPS30 sebagai sensor untuk mengukur konsentrasi konsentrasi partikulat
4. OLED SSD1306 0,96 inci sebagai layar
5. Raspberry Pi 3B sebagai *data logger*
6. STM32CubeIDE sebagai *tools* pengembangan *firmware*
7. RealTerm sebagai *serial monitor*
8. Osiloskop Rigol MSO1104
9. Multimeter untuk mengukur kuat arus
10. Kabel USB *micro-B* untuk *upload* program dan jalur data ke *serial monitor*
11. Resistor  $10k\Omega$  sebagai resistor *pull-up*
12. *Breadboard* untuk merangkai rangkaian
13. Kabel *jumper* untuk merangkai rangkaian

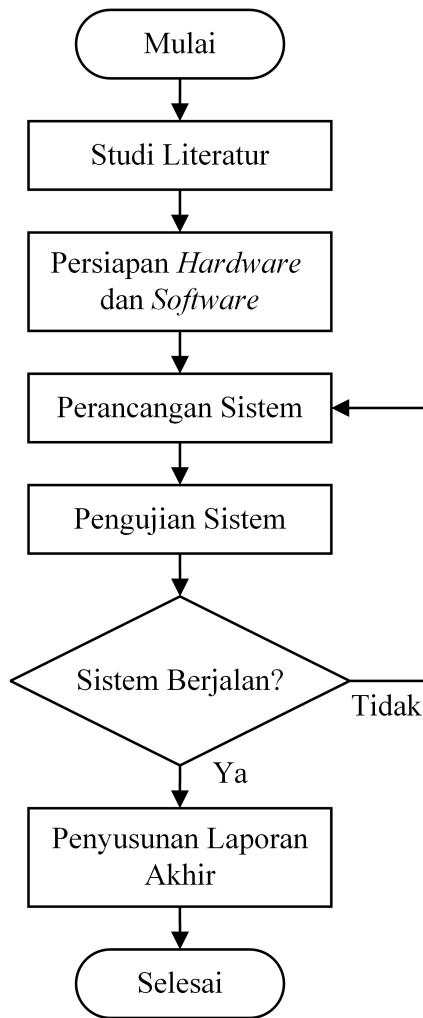
##### **3.1.2 Bahan Tugas akhir**

Bahan-bahan yang digunakan pada penelitian ini adalah materi yang didapat dari hasil studi literatur mengenai sensor yang digunakan, protokol komunikasi, dan manual referensi dari STM32.

#### **3.2 Metode yang Digunakan**

Metode yang digunakan pada penelitian ini adalah pengembangan *firmware* pada mikrokontroler dan sensor menggunakan bahasa pemrograman C. Pengembangan dil-

kukan dengan cara memahami spesifikasi sensor melalui *datasheet* dan memahami *framework* pengembangan *firmware* pada STM32. Algoritma *firmware* yang dikembangkan menyesuaikan dengan panduan pada *datasheet* dan dokumen-dokumen lain pada situs resmi sensor Sensirion SPS30. *Firmware* yang dikembangkan akan diuji di tiga lokasi, yaitu dua area *indoor* dan satu area *outdoor* UGM Press untuk mengukur konsentrasi partikulat di udara ketiga lokasi tersebut. Hasil pengukuran konsentrasi partikulat di udara selanjutnya akan dianalisis dan diolah untuk mengetahui indeks kualitas udara menurut standar ISPU. Tingkat akurasi dan presisi pembacaan sensor terbatas pada dokumentasi spesifikasi yang ada. Selain itu, protokol komunikasi I2C antara sensor dan STM32 juga akan diuji. Protokol komunikasi I2C akan diteliti untuk melihat bagaimana komunikasi tersebut bekerja, apakah sesuai dengan standar protokolnya atau tidak.



Gambar 3.1. Diagram alir penelitian

### 3.3 Alur Tugas Akhir

Penelitian dilakukan berdasarkan pada diagram alir yang ditunjukkan oleh Gambar 3.1. Penelitian dimulai dengan studi literatur mengenai topik yang diangkat. Kemu-

dian, dilanjutkan persiapan *hardware* dan *software*, perancangan sistem, dan pengujian sistem. Ketika pengujian sistem, sistem diuji dan diperbaiki secara kontinu hingga dapat berjalan sesuai dengan keinginan. Sistem itu pun pada akhirnya akan menyesuaikan beberapa pengujian yang dilakukan. Artinya kebutuhan sistem berbeda-beda tergantung pengujian apa yang akan dilakukan. Setelah seluruh pengujian selesai dilakukan, penelitian diakhiri dengan penulisan laporan akhir ini.

### 3.4 Studi Literatur

Sebelum memulai penelitian, studi literatur dilakukan untuk mendapatkan materi dan referensi yang berkaitan dengan topik penelitian. Materi dan referensi tersebut melingkupi jurnal ilmiah, laporan penelitian, artikel, hingga *datasheet* tentang pencemaran udara dan pengembangan sistem pemantauannya. Tahap studi literatur ini berguna untuk mendapat gambaran latar belakang masalah, rancangan sistem, dan landasan teori yang akan digunakan pada penelitian. Dalam hal pengembangan sistem, studi literatur dimulai dengan mempelajari pengembangan *firmware* pada STM32, mempelajari spesifikasi sensor Sensirion SPS30, serta protokol komunikasinya.

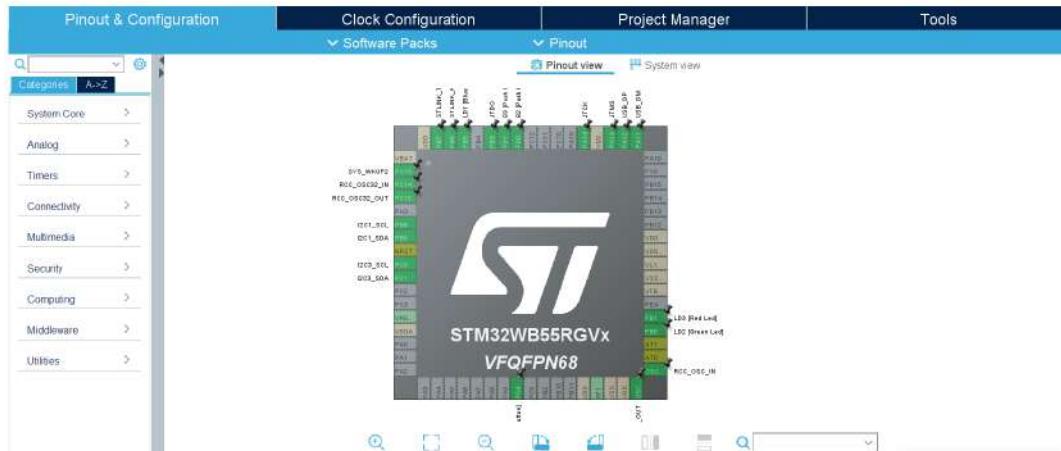
### 3.5 Persiapan *Hardware* dan *Software*

Pada tahapan studi literatur sebelumnya, didapatkan pengetahuan dan wawasan yang lebih dalam mengenai penelitian yang akan dilakukan. Dengan demikian, gambaran secara umum mengenai perancangan sistem telah didapat. Maka dari itu, *hardware* dan *software* yang akan digunakan perlu dipersiapkan. Pada penelitian ini akan dikembangkan *firmware* untuk pemantauan kualitas udara yang dikhususkan pada pengukuran partikulat di udara. *Firmware* yang digunakan oleh perangkat keras, yaitu mikrokontroler dan sensor, memerlukan perangkat lunak untuk mengembangkannya. Selain itu, data yang diperoleh dari pengukuran sensor akan ditampilkan pula. Oleh karena itu, pada tahap ini dilakukan persiapan berupa riset perangkat keras dan lunak yang akan digunakan serta pemasangan *environment* yang dibutuhkan.

Dari sisi *hardware*, yang pertama kali dilakukan adalah memahami spesifikasi dan cara kerja sensor Sensirion SPS30. Sumber utama riset sensor ini adalah dokumen *datasheet*. Di dalam *datasheet* dijelaskan spesifikasi elektronis, spesifikasi antarmuka, hingga seluruh fitur yang dimilikinya. Informasi yang diberikan oleh *datasheet* sangat penting dalam bagaimana *firmware* akan dikembangkan. Berdasarkan *datasheet*, sensor SPS30 memiliki dua pilihan antarmuka untuk berkomunikasi dengan mikrokontroler, yaitu UART dan I2C. Penggunaan antarmuka tersebut dipilih dengan menggunakan pin SEL (*interface select*). Pin SEL dibiarkan *floating* untuk memilih UART atau dihubungkan ke *ground* untuk memilih I2C. Karena fokus penelitian ini adalah komunikasi I2C, maka pin SEL dihubungkan ke *ground*. Dengan demikian, untuk mendukung komunikasi I2C,

diperlukan kabel *jumper* dan dua buah resistor *pull-up* untuk rangkaianya.

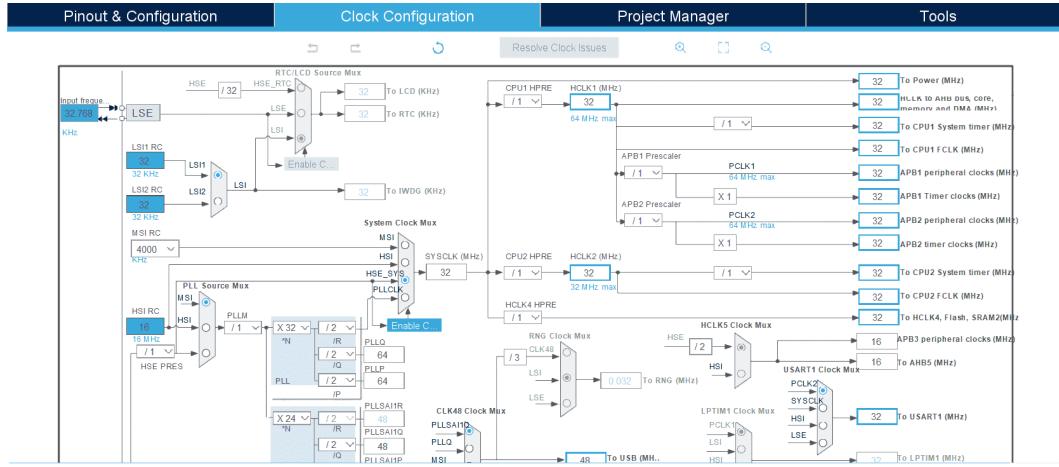
Dengan spesifikasi kebutuhan sensor untuk menunjang komunikasinya, maka dibutuhkan mikrokontroler yang mendukung komunikasi I2C. Pada penelitian ini digunakan mikrokontroler STM32 P-Nucleo-WB55. P-Nucleo-WB55 memiliki dua periferal I2C. Salah satu periferal I2C tersebut digunakan untuk komunikasi dengan sensor SPS30. Kemudian, untuk kebutuhan *monitoring* satu periferal I2C lainnya digunakan untuk komunikasi dengan layar OLED SSD1306. Modul OLED tersebut dibekali dengan kemampuan komunikasi I2C juga. Akan tetapi, berdasarkan *datasheet*-nya, diketahui bahwa modul OLED tersebut harus dihubungkan dengan komunikasi I2C dengan kecepatan *fast mode*, yaitu 400 kHz. Sementara itu, kecepatan maksimum dari sensor SPS30 adalah *standard mode*, yaitu 100 kHz. Dengan demikian, kedua komponen tersebut tidak bisa terhubung pada bus yang sama sehingga kedua periferal I2C pada mikrokontroler harus diaktifkan untuk menunjang kedua komunikasinya.



Gambar 3.2. Konfigurasi piranti I/O yang akan digunakan

Dari sisi *software*, dilakukan persiapan, yaitu pemasangan aplikasi-aplikasi yang akan digunakan dalam penelitian ini pada laptop. Pengembangan *firmware* dilakukan dengan menggunakan STM32CubeIDE. STM32CubeIDE merupakan *framework* pengembangan *firmware* yang didesain untuk STM32. *Software* tersebut diunduh melalui situs resminya. Pada STM32CubeIDE terdapat berbagai macam jenis mikrokontroler atau *board* STM32 yang tersedia di pasaran. Karena STM32 memiliki banyak macam dengan spesifikasi yang berbeda-beda, mikrokontroler atau *development board* harus disesuaikan dengan yang akan digunakan. P-Nucleo-WB55 dipilih dan dilanjutkan dengan melakukan konfigurasi piranti I/O dan *clock*. I/O yang akan digunakan harus diaktifkan terlebih dahulu agar berfungsi, sedangkan yang tidak digunakan tidak perlu diaktifkan. Konfigurasi *clock* digunakan untuk mengatur sumber *clock* dan frekuensinya, baik dari osilator internal maupun eksternal seperti yang ditunjukkan pada Gambar 3.3. Konfigurasi tersebut selanjutnya akan menentukan barisan kode yang akan dibuat secara otomatis pada *file main.c* untuk inisialisasi. Dengan demikian, pengguna dapat fokus dalam program utama

yang akan dikembangkan. Program pertama yang dijalankan dalam tahap studi pemrograman berbasis STM32 adalah *blinking LED*. Setelah berhasil, dilanjutkan dengan studi uji coba menggunakan komunikasi UART dan I2C sebagai persiapan untuk memprogram sensor SPS30.



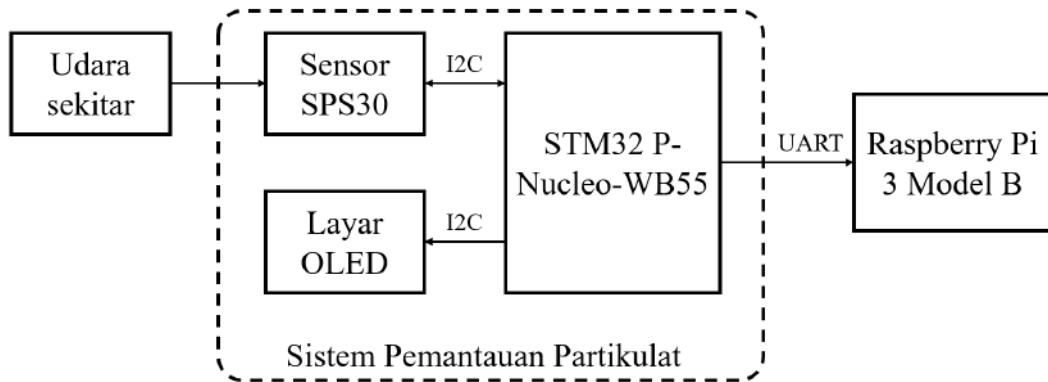
Gambar 3.3. Konfigurasi *clock*

*Software* lain yang digunakan dalam penelitian ini adalah aplikasi *serial monitor* untuk menampilkan hasil pengukuran dari sensor. Aplikasi yang digunakan dan dipasang di laptop adalah RealTerm. Hasil pembacaan sensor oleh mikrokontroler akan dikirim ke laptop menggunakan USB dengan protokol komunikasi UART. Agar dapat terbaca, konfigurasi UART pada RealTerm harus sesuai dengan komunikasi UART pada mikrokontroler, seperti *baudrate*, jumlah bit, bit paritas, dsb. Penggunaan *serial monitor* ini dapat membantu dalam tahap pengembangan *firmware* untuk memastikan bahwa program yang dibuat berhasil atau terdapat *error*. Untuk itu, pesan *error* perlu dibuat pada program untuk mengetahui ketika ada *error* pada baris kode tertentu. Secara umum aplikasi *serial monitor* dapat menangkap pesan yang masuk untuk disimpan dalam *text file* (.txt). Untuk mengakali agar data pengukuran dapat diolah, pesan yang dikirim ke *serial monitor* harus dipisahkan dengan koma tiap nilainya sehingga *file* .txt yang terbentuk dapat diubah menjadi *file* .csv. Data dalam bentuk .csv selanjutnya dapat diolah menggunakan Microsoft Excel atau program Python.

### 3.6 Perancangan Sistem

Pada penelitian ini, sistem yang bekerja ditunjukkan oleh diagram blok pada Gambar 3.4. Konsentrasi partikulat diukur oleh sensor SPS30. Data pengukuran dibaca oleh mikrokontroler dan ditampilkan pada layar OLED. Data pengukuran kemudian akan disimpan pada komputer Raspberry 3 Model B. Tahap perancangan sistem tersebut dibagi menjadi tiga, yaitu perancangan perangkat keras, pengembangan *firmware*, dan pengolahan data. Perancangan perangkat keras mencakup rangkaian komponen yang akan digunakan sehingga dapat berjalan sesuai dengan *firmware* yang akan dikembangkan.

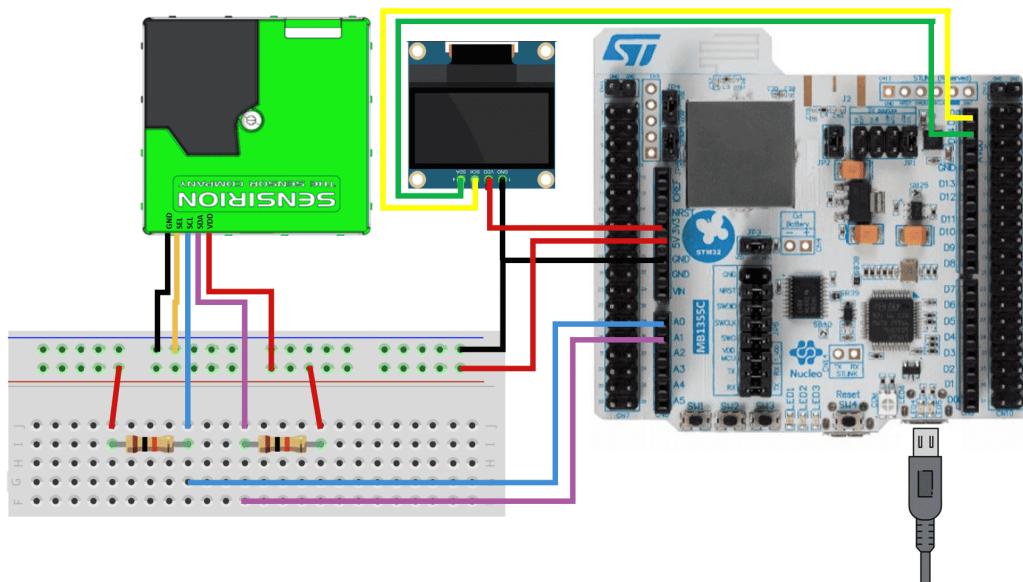
Pengembangan *firmware* mencakup pembuatan program pada STM32CubeIDE sehingga mikrokontroler dapat berkomunikasi dengan sensor SPS30 untuk mengukur partikel debu di udara dan layar OLED SSD1306 untuk menampilkan data hasil pengukurannya. Pengolahan data mencakup menghitung indeks kualitas udara dan menampilkannya dalam bentuk grafik.



Gambar 3.4. Diagram blok sistem

### 3.6.1 Perancangan Perangkat Keras

Berdasarkan persiapan *hardware* pada bagian sebelumnya, komponen-komponen yang akan digunakan adalah sebagai berikut: P-Nucleo-WB55, sensor SPS30, dan OLED SSD1306. Semua komponen tersebut akan dirangkai pada *breadboard* seperti yang ditunjukkan pada Gambar 3.5.



Gambar 3.5. Diagram *wiring* rangkaian

Berdasarkan *datasheet*, sensor SPS30 membutuhkan tegangan masukan sebesar 5 V. Untuk menggunakan protokol komunikasi I2C, pin SEL dihubungkan dengan *ground*.

Karena menggunakan protokol I2C, kedua pin SDA dan SCL sensor memiliki *pull-up resistor* dengan besar resistansi yang digunakan pada perancangan ini adalah  $10\text{ k}\Omega$ . *Pull-up resistor* digunakan untuk memenuhi kondisi normal komunikasi I2C, yaitu berada dalam kondisi *high* ketika normal. Untuk lebih jelas, kelima pin sensor SPS30 dihubungkan dengan mikrokontroler sebagai berikut:

- VDD : 5 V
- SDA : pin A1 (I2C3\_SDA)
- SCL : pin A0 (I2C3\_SCL)
- SEL : GND
- GND : GND

Modul OLED SSD1306 yang digunakan memiliki ukuran 0,96 inci dengan dukungan protokol komunikasi I2C. Berbeda dengan sensor SPS30, OLED membutuhkan tegangan masukan sebesar 3,3 V. *Pull-up resistor* tidak perlu ditambahkan lagi karena sudah dirangkai pada modul tersebut. Untuk lebih jelas, pin-pin modul OLED dihubungkan dengan mikrokontroler sebagai berikut:

- VDD : 3,3 V
- SDA : pin D14 (I2C1\_SDA)
- SCL : pin D15 (I2C1\_SCL)
- GND : GND

P-Nucleo-WB55 sebagai mikrokontroler menggunakan daya dari komputer sebagai catu dayanya melalui kabel USB *micro-B*. Kabel USB tersebut dapat sekaligus digunakan untuk komunikasi UART dengan komputer. Mikrokontroler mengirim pesan melalui protokol UART untuk diterima dan ditampilkan pada *serial monitor* komputer. Penggunaan kabel USB sebagai jalur daya dan komunikasi membuat rangkaian menjadi lebih efisien karena tidak perlu menambahkan rangkaian sendiri untuk komunikasi UART. Komputer yang digunakan sebagai catu daya terdapat dua pilihan, yaitu laptop dan Raspberry Pi. Keduanya digunakan secara bergantian sesuai dengan pengujian yang akan dilakukan. Pengujian akan dijelaskan nanti.

### 3.6.2 Pengembangan *Firmware*

Pengembangan *firmware* sejatinya adalah membuat program untuk mikrokontroler. *Firmware* dikembangkan setelah memahami garis besar cara kerja sensor SPS30. Untuk itu, *firmware* yang dikembangkan mencakup semua fitur atau perintah yang tersedia pada sensor SPS30. Setiap perintah diprogram dengan membuat fungsi masing-masing.

Dari fungsi-fungsi yang telah dibuat, kemudian dikembangkan algoritma atau rutin yang akan dijalankan sesuai dengan pengujian yang ingin dilakukan. Di luar pengembangan program sensor, tidak lupa pula ditambahkan barisan kode untuk menampilkan nilai pengukuran pada layar OLED. Seluruh *firmware* yang dikembangkan tersebut menggunakan STM32CubeIDE sebagai kerangka kerjanya.

Hal pertama yang dilakukan setelah membuka STM32CubeIDE dan membuat *project* baru adalah memilih *development board* yang akan digunakan, yaitu P-Nucleo-WB55. Kemudian pada konfigurasi piranti I/O, komunikasi I2C diaktifkan dengan memilih menu "*Connectivity*" pada menu di sebelah kiri. Seperti pada perancangan *hardware*, sensor SPS30 dihubungkan dengan periferal I2C3, sedangkan layar OLED dihubungkan dengan periferal I2C1. Yang harus diperhatikan ketika mengaktifkannya adalah pengaturan parameternya. Keduanya harus dipastikan menggunakan mode alamat 7 bit. Namun, mode kecepatannya berbeda menyesuaikan spesifikasi layar OLED dan sensor SPS30. I2C1 menggunakan *Fast Mode* (400 kHz), sedangkan I2C3 menggunakan *Standard Mode* (100kHz). Berbeda dengan I2C, komunikasi UART sudah diaktifkan secara *default* karena P-Nucleo-WB55 mendukung penggunaan kabel USB yang terhubung dengan periferal UART. Sama halnya dengan periferal lain seperti LED dan *push button* yang sudah aktif secara *default*. Akan tetapi, konfigurasi komunikasi UART pada perancangan ini diatur agar panjang pesan diubah dari 7 bit menjadi 8 bit. Setelah seluruh konfigurasi piranti I/O selesai dan konfigurasi *clock* dibiarkan sesuai bawaannya, barisan kode untuk menginisialisasi semua periferal tersebut akan terbentuk secara otomatis ketika klik "*Save*". Setelah langkah-langkah tersebut dilakukan, akan terbentuk *file main.c* yang merupakan tempat seluruh program dikembangkan.

### 3.6.2.1 Pembuatan Fungsi Perintah Sensor SPS30

Bagian paling penting dari pengembangan ini adalah dapat diaksesnya sensor SPS30 oleh mikrokontroler. Di dalam komunikasi I2C, setiap perangkat berkomunikasi dengan menggunakan alamat. Alamat digunakan oleh *master* untuk mengetahui tujuan komunikasinya. Pada sistem yang dirancang ini, P-Nucleo-WB55 bertindak sebagai *master*, sedangkan sensor SPS30 bertindak sebagai *slave*. Dari *datasheet* diketahui bahwa alamat sensor SPS30 adalah 0x69. Setiap kali mikrokontroler ingin berkomunikasi dengan sensor SPS30, alamat 0x69 harus dituliskan pada *byte* pertama sinyal I2C di jalur SDA. Dengan demikian, sensor dapat menanggapi sinyal yang dikirim master dan siap untuk menjalankan perintah-perintahnya.

Sensor SPS30 memiliki sejumlah perintah yang dapat dijalankan, mulai dari perintah untuk membaca nilai pengukuran hingga untuk pembersihan kipas. Dalam komunikasi I2C, perintah-perintah tersebut merupakan *pointer* alamat 16 bit di mana data akan ditulis atau dibaca. Setiap perintah memiliki kegunaannya masing-masing sehingga me-

merlukan cara komunikasi yang berbeda-beda untuk melakukan perintahnya. Untuk itu, perlu diperhatikan bahwa secara garis besar setiap perintah tersebut terbagi menjadi tiga tipe transfer, yaitu:

1. *Set Pointer*: mengirim *pointer* alamat 16 bit dari perintah tanpa mengirim parameter data lain lagi. Tipe transfer ini digunakan hanya untuk menjalankan perintah yang diinginkan.
2. *Set Pointer & Write Data*: mengirim *pointer* alamat 16 bit yang diikuti dengan data lain di belakangnya. Data dikirim setiap paket 2 *byte* dengan dilindungi *checksum* untuk setiap paketnya.
3. *Set Pointer & Read Data*: mengirim *pointer* alamat 16 bit dan membaca data dari sensor. Data dikirim sensor ke mikrokontroler setiap paket 2 *byte* yang dilindungi *checksum* untuk setiap paketnya. Terdapat dua proses pada tipe ini, yaitu komunikasi I2C dari mikrokontroler ke sensor dan dari sensor ke mikrokontroler. Ketika hendak membaca data secara terus menerus, *pointer* alamat cukup dikirim sekali saja.

Tabel 3.1. Daftar Perintah pada Sensor SPS30

Alamat Pointer	Nama Perintah	Tipe Transfer	Panjang Parameter (byte)	Panjang Respons (byte)	Waktu Eksekusi
0x0010	<i>Start Measurement</i>	<i>Set Pointer &amp; Write Data</i>	3	-	< 20 ms
0x0300	<i>Read Measured Values</i>	<i>Set Pointer &amp; Read Data</i>	-	<i>float</i> : 60 <i>integer</i> : 30	-
0xD304	<i>Device Reset</i>	<i>Set Pinter</i>	-	-	< 100 ms
0x0104	<i>Stop Measurement</i>	<i>Set Pointer</i>	-	-	< 20 ms
0x0202	<i>Read Data-Ready Flag</i>	<i>Set Pointer &amp; Read Data</i>	-	3	-
0x1001	<i>Sleep</i>	<i>Set Pointer</i>	-	-	< 5 ms
0x1103	<i>Wake-up</i>	<i>Set Pinter</i>	-	-	< 5 ms
0x5607	<i>Start Fan Cleaning</i>	<i>Set Pinter</i>	-	-	< 5 ms
0x8004	<i>Read/Write Auto Cleaning Interval</i>	<i>Set Pointer &amp; Read/Write Data</i>	<i>read</i> : - <i>write</i> : 6	<i>read</i> : 6 <i>write</i> : -	<i>read</i> : - <i>write</i> : < 20 ms
0xD002	<i>Read Product Type</i>	<i>Set Pointer &amp; Read Data</i>	-	12	-

0xD033	<i>Read Serial Number</i>	<i>Set Pointer &amp; Read Data</i>	-	max. 48	-
0xD100	<i>Read Version</i>	<i>Set Pointer &amp; Read Data</i>	-	3	-
0xD206	<i>Read Device Status Register</i>	<i>Set Pointer &amp; Read Data</i>	-	6	-
0xD210	<i>Clear Device Status Register</i>	<i>Set Pinter</i>	-	-	< 5 ms

Detail dari setiap perintah yang dimiliki sensor SPS30 ditunjukkan pada Tabel 3.1. Di dalam program *main.c*, alamat sensor SPS30 dan perintah-perintahnya didefinisikan di awal. Hal ini bertujuan untuk memudahkan penulisan program sehingga seluruh alamat tersebut tidak perlu diingat atau berulang kali dilihat ketika mengembangkan programnya. Dalam bahasa C, pendefinisian ditunjukkan pada baris kode di bawah. Tahap selanjutnya setelah itu adalah membuat fungsi-fungsi untuk menjalankan perintah tersebut.

```

1 #define SPS30_ADDR 0x69
2 #define SPS30_CMD_START_MEASUREMENT 0x0010
3 #define SPS30_CMD_FLOAT_FORMAT 0x0300
4 #define SPS30_CMD_INTEGER_FORMAT 0x0500
5 #define SPS30_CMD_STOP_MEASUREMENT 0x0104
6 #define SPS30_CMD_READ_DATA_READY_FLAG 0x0202
7 #define SPS30_CMD_READ_MEASURED_VALUES 0x0300
8 #define SPS30_CMD_SLEEP 0x1001
9 #define SPS30_CMD_WAKEUP 0x1103
10 #define SPS30_CMD_START_FAN_CLEANING 0x5607
11 #define SPS30_CMD_AUTO_CLEANING_INTERVAL 0x8804
12 #define SPS30_CMD_READ_PRODUCT_TYPE 0xd002
13 #define SPS30_CMD_READ_SERIAL_NUMBER 0xd033
14 #define SPS30_CMD_READ_VERSION 0xd100
15 #define SPS30_CMD_READ_DEVICE_STATUS_REGISTER 0xd206
16 #define SPS30_CMD_CLEAR_DEVICE_STATUS_REGISTER 0xd210
17 #define SPS30_CMD_RESET 0xd304

```

Agar fungsi dapat berjalan sebagaimana yang diharapkan, harus dipahami cara mengimplementasikan protokol komunikasi I2C pada STM32. Pemrograman pada STM32 berbasis pada *Hardware Abstraction Layer (HAL)*. Terdapat dua fungsi yang sangat penting pada HAL Library untuk komunikasi I2C pada perancangan ini, yaitu `HAL_I2C_Master_Transmit()` dan `HAL_I2C_Master_Receive()`. Fungsi tersebut masing-masing digunakan ketika ingin mengirim dan menerima data. Keduanya memiliki lima parameter masukan, yaitu:

1. \*hi2c : *pointer* ke struktur *I2C\_HandleTypeDef* yang berisi konfigurasi untuk periferal I2C yang digunakan
2. DevAddress : alamat *slave*. Perlu diperhatikan bahwa untuk alamat *slave* 7 bit harus digeser ke kiri 1 bit. Bit ke-8 (bit R/W) diisi oleh 0 ketika *transmit* dan 1 ketika *receive*. Bit R/W tersebut sudah otomatis diisi ketika bit alamat digeser ke kiri.
3. \*pData : *pointer* ke *buffer* data
4. Size : ukuran data yang akan dikirim atau diterima
5. Timeout : durasi *timeout* ketika komunikasi berlangsung

Setelah mengetahui setiap perintah dan implementasi komunikasi I2C pada STM32, selanjutnya dibuat fungsi untuk menjalankan perintah-perintah tersebut. Sederhananya, isi dari fungsi yang akan dibuat adalah cara menyusun struktur nilai pada sinyal I2C yang hendak dikirim atau diterima sesuai dengan masing-masing perintah tersebut. Dengan demikian, untuk satu perintah dengan tipe transfer yang sama dengan perintah lainnya relatif memiliki struktur program yang sama.

Untuk tipe transfer *Set Pointer*, yang harus disusun hanyalah *pointer* alamatnya. Misalnya pada perintah *Device Reset*, urutan sinyal I2C yang dikirim adalah alamat sensor 0x69 dan diikuti *pointer* alamat 0xD304. Dalam bahasa C, fungsi *device\_reset()* dituliskan sebagai berikut:

```

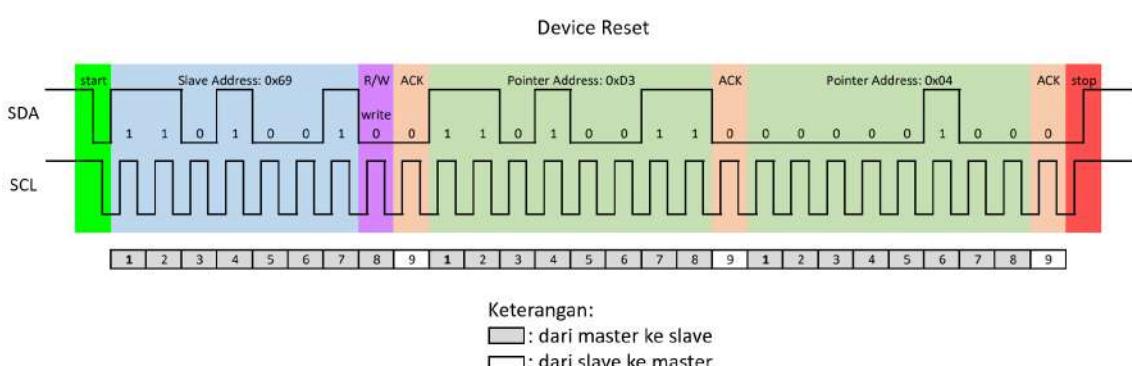
1 int device_reset(void){
2     uint8_t address = SPS30_ADDR; //0x69
3     uint16_t cmd = SPS30_CMD_RESET; //0xD304
4     uint8_t buf[32];
5     uint16_t buf_size;
6     uint16_t idx = 0;
7
8     buf[idx++] = (uint8_t)((cmd & 0xFF00) >> 8);
9     buf[idx++] = (uint8_t)((cmd & 0x00FF) >> 0);
10    buf_size = idx;
11
12    return (int8_t)HAL_I2C_Master_Transmit(&hi2c3, (uint16_t)(
13        address << 1), (uint8_t *)buf, buf_size, 100);
}

```

Pada fungsi tersebut, yang pertama kali dilakukan adalah memasukkan alamat sensor dan *pointer* alamat perintah *Device Reset* ke dalam suatu variabel. Alamat sensor merupakan bilangan 1 *byte* sehingga variabel *address* memiliki tipe data *uint8\_t* yang berukuran 8 bit atau 1 *byte*. Sementara itu, *pointer* alamat merupakan bilangan 2 *byte*

sehingga variabel `cmd` memiliki tipe data `uint16_t` yang berukuran 16 bit atau 2 *byte*. Selanjutnya dibuat sebuah *buffer* untuk menyimpan data yang akan dikirim. Pada *buffer* tersebut nilai sinyal I2C diurutkan. Sesuai dengan fungsi I2C dan teorinya, sinyal I2C dikirimkan setiap 1 *byte*. Oleh karenanya, variabel `buf` [] memiliki tipe data `uint8_t`. Baris 9 dan 10 berisi kode untuk mengisi *buffer* dengan alamat 0xD304. Baris 9 berfungsi untuk memasukkan bilangan 0xD3 ke dalam `buf[0]` dengan cara mengubahnya menjadi `uint8_t` dan menggesernya ke kanan sebanyak 8. Baris 10 berfungsi untuk memasukkan bilangan 0x04 ke dalam `buf[1]` dengan cara mengubahnya menjadi `uint8_t` juga. Dengan demikian, *buffer* saat ini berukuran 2 *byte* yang dihitung oleh variabel `idx`. Selanjutnya nilai variabel `idx` disimpan pada variabel `buf_size`.

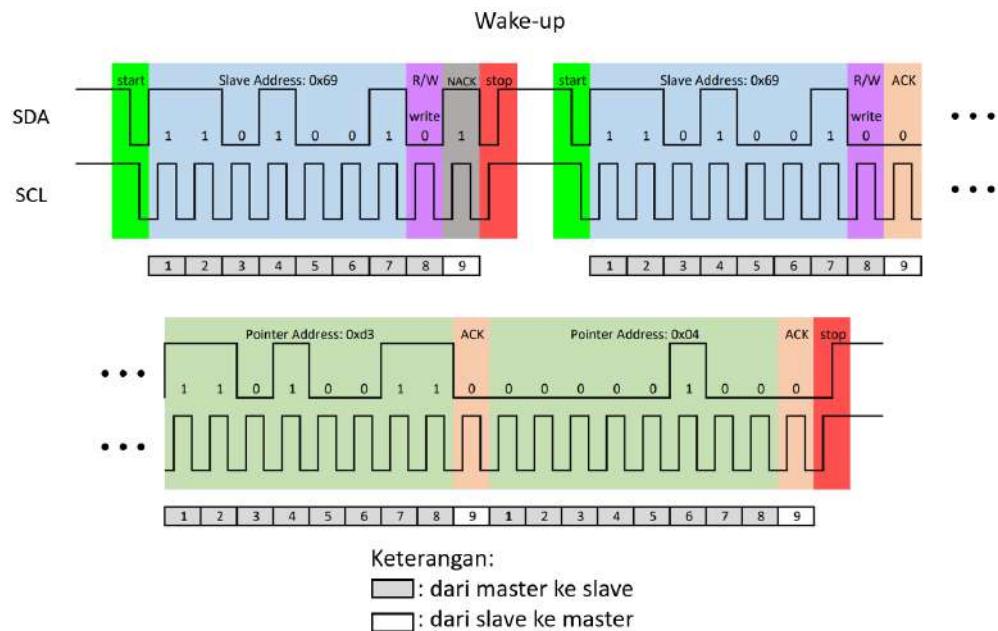
Variabel alamat, *buffer*, dan ukuran selanjutnya digunakan sebagai masukan dari fungsi `HAL_I2C_Master_Transmit()` di *return* fungsi `device_reset()`. Perlu diperhatikan *address* harus digeser ke kiri sebanyak 1 bit untuk memasukkan bit R/W pada LSB-nya. Karena ini adalah fungsi I2C *transmit*, maka bit R/W adalah 0 yang sudah terisi otomatis. *Timeout* yang digunakan adalah 100 ms hanya agar nilainya lebih besar dari waktu eksekusi pengiriman sinyal I2C ini. Sinyal I2C bit ke-9 atau bit ACK/NACK sudah dijalankan langsung oleh fungsi I2C *transmit* tersebut. *Master* akan menyiapkan *clock* bit ke-9 agar sensor dapat mengirim sinyal ACK/NACK ke *master*. Jadi, di dalam program hanya perlu menyiapkan urutan nilai data 8 bit saja di dalam sebuah *buffer* berukuran 8 bit. Dengan fungsi dan masukannya ini, sinyal I2C yang terbentuk pada jalur SDA ditunjukkan pada Gambar 3.6. Kode dan sinyal yang terbentuk juga berlaku untuk perintah tipe transfer *Set Pointer* yang lain, seperti *Stop Measurement*, *Sleep*, *Start Fan Cleaning*, dan *Clear Device Status Register*. Yang membedakan hanyalah *pointer* alamatnya saja.



Gambar 3.6. Diagram bit komunikasi I2C pada perintah *Device Reset*

Dari Gambar 3.6, dapat dilihat bahwa komunikasi I2C dimulai dengan kondisi *start* yang ditandai dengan jalur SDA ditarik menjadi *low* ketika jalur SCL masih *high*. Dengan ini, komunikasi I2C pun dimulai. 7 bit pertama setelah kondisi *start* adalah alamat sensor 0x69 yang telah digeser ke kiri sebanyak 1 bit. Bit ke-8 secara otomatis terisi

0 karena komunikasi yang terjadi adalah *transmit* sinyal dari master ke *slave*. Bit ke-9 diisi ACK oleh sensor yang berarti sensor menanggapi komunikasi I2C karena alamatnya sesuai. 8 bit selanjutnya diisi dengan 0xD3, yaitu *byte* pertama dari *pointer* alamat yang kemudian diikuti dengan bit ACK oleh sensor untuk menandakan bahwa data telah diterima oleh sensor. 1 *byte* terakhir diisi dengan nilai 0x04 yang merupakan *byte* kedua dari *pointer* alamat yang kemudian juga diikuti dengan bit ACK. Karena seluruh data telah dikirim oleh master, komunikasi ditutup dengan master mengirim kondisi *stop* yang ditandai dengan nilai jalur SDA berubah menjadi *high* ketika jalur SCL dalam kondisi *high* juga.



Gambar 3.7. Diagram bit komunikasi I2C pada perintah *Wake-up*

Khusus perintah *Wake-up*, walaupun memiliki tipe transfer *Set Pointer*, ada sedikit perbedaan dengan perintah-perintah lainnya. Perintah *Wake-up* itu sendiri digunakan untuk mengubah mode dari *sleep* ke *idle*. Ketika sensor sedang dalam kondisi *sleep*, untuk mengurangi konsumsi daya, hampir semua elektronik di dalamnya mati, termasuk antarmuka komunikasi I2C dan UART. Dengan demikian dibutuhkan sebuah urutan khusus untuk menghidupkan periferal antarmuka komunikasi tersebut. Dalam komunikasi I2C, terdapat dua cara untuk mengaktifkan periferal komunikasi dan menjalankan perintah *Wake-up*, yaitu mengirim sinyal *low* pada jalur SDA yang diikuti dengan mengirim perintah *Wake-up* dalam kurun waktu 100 ms atau mengirim perintah *Wake-up* dua kali yang mana perintah pertama akan diabaikan dan mengaktifkan periferal komunikasi. Pada perancangan ini, cara kedua digunakan. Implementasinya pada program, merujuk pada kode fungsi *Device Reset* di atas, sebelum baris *return* ditambahkan fungsi I2C *Transmit* yang sama dengan fungsi di baris *return*. Dengan demikian, sinyal I2C pada jalur SDA yang terbentuk ditunjukkan pada Gambar 3.7 dan sensor akan kembali masuk

ke mode *idle*.

Untuk tipe transfer *Set Pointer & Write Data*, struktur fungsinya mirip seperti tipe sebelumnya. Yang membedakannya adalah terdapat parameter data lain yang dikirim setelah *pointer* alamat. Jadi data tersebut harus disiapkan pula urutannya sesuai dengan petunjuk pada *datasheet*. Misalnya pada perintah *Start Measurement*, urutan sinyal I2C yang dikirim adalah alamat sensor 0x69, *pointer* alamat 0x0010, dan diikuti dengan 3 *byte* data lainnya, yaitu format keluaran nilai pengukuran (0x03 untuk format nilai *big-endian IEEE float* atau 0x05 untuk format nilai *unsigned 16-bit integer*), *dummy byte* 0x00, dan *checksum* antara kedua *byte* tersebut. Dengan begitu, diperlukan fungsi tersendiri untuk menghitung *checksum* antara 2 *byte* paket data. Di dalam *datasheet* dijelaskan bahwa *checksum* dihitung dengan algoritma yang ditunjukkan pada Algoritma 1.

---

#### Algorithm 1 Perhitungan CRC

---

```
1: function CALC_CRC(data[2])
2:   crc = 0xFF
3:   for i = 0; i < 2; i + + do
4:     crc ^ = data[i]
5:     for bit = 8; bit > 0; -- bit do
6:       if crc & 0x90 then
7:         crc = (crc << 1)^0x31u
8:       else
9:         crc = (crc << 1)
10:      end if
11:    end for
12:  end for
13:  return crc
14: end function
```

---

Program fungsi untuk perintah tipe transfer *Set Pointer & Write Data*, khususnya untuk perintah *Start Measurement*, dapat dibuat dengan memodifikasi fungsi *Device Reset* di atas. Hasil modifikasi program fungsinya adalah sebagai berikut:

```
1  int start_measurement(void) {
2    uint8_t address = SPS30_ADDR; //0x69
3    uint16_t cmd = SPS30_CMD_START_MEASUREMENT; //0x0010
4    uint16_t arg = 0x0300; //untuk tipe data float
5    // uint16_t arg = 0x0500; //untuk tipe data integer
6    uint8_t buf[32];
7    uint8_t crc;
8    uint16_t idx = 0;
```

```

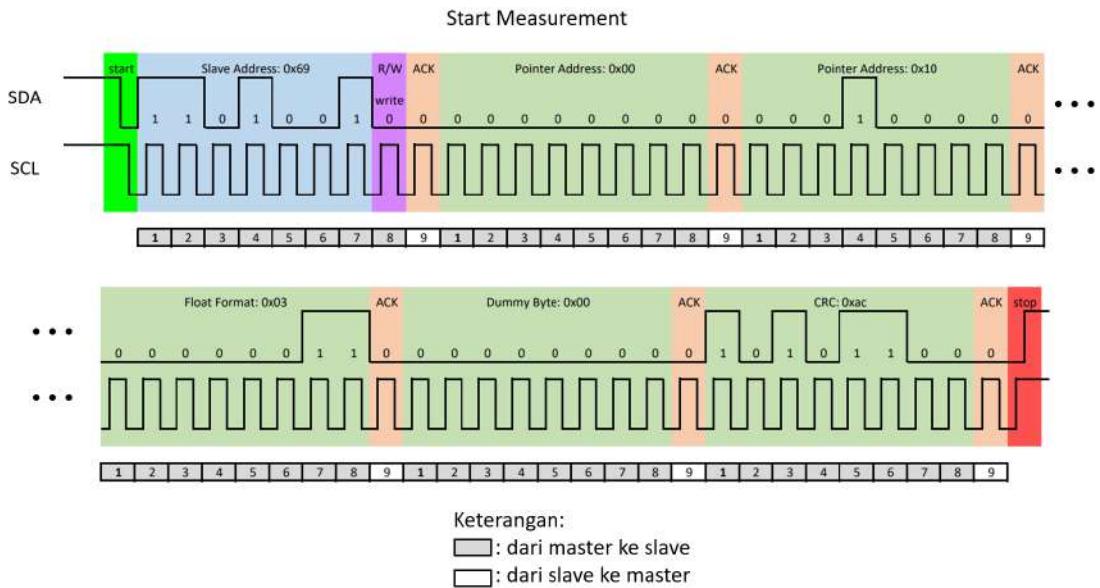
9
10    buf[ idx++ ] = ( uint8_t )(( cmd & 0xFF00 ) >> 8 );
11    buf[ idx++ ] = ( uint8_t )(( cmd & 0x00FF ) >> 0 );
12    buf[ idx++ ] = ( uint8_t )(( arg & 0xFF00 ) >> 8 );
13    buf[ idx++ ] = ( uint8_t )(( arg & 0x00FF ) >> 0 );
14    crc = calc_crc( ( uint8_t * )&buf[ idx - 2 ] );
15    buf[ idx++ ] = crc;
16    buf_size = idx;

17
18    return ( int8_t )HAL_I2C_Master_Transmit( &hi2c3 , ( uint16_t )(
19        address << 1 ), ( uint8_t * )buf , buf_size , 100 );
}

```

Pada fungsi tersebut dapat dilihat bahwa perubahannya tidak banyak. Selain *pointer* alamat perintah, modifikasinya terletak pada penambahan nilai pada *buffer*. Parameter data lain setelah *pointer* alamat perintah dimasukkan dan diurutkan pada variabel *buf* [ ]. Parameter pemilihan format dan *dummy byte* terletak pada *buf* [ 2 ] dan *buf* [ 3 ]. Sementara itu, *buf* [ 4 ] diisi oleh *checksum* yang nilainya didapat dengan menggunakan fungsi *calc\_crc()* yang masukannya adalah nilai pada *buf* [ 2 ] dan *buf* [ 3 ]. Berdasarkan algoritma perhitungan *checksum* di atas, hasil nilai *checksum* yang didapat antara 0x03 dan 0x00 adalah 0xAC. Dengan begitu ukuran dari *buffer* atau variabel *buf\_size* juga bertambah besar. Setelah seluruh parameter data telah masuk ke dalam *buffer*, fungsi *Start Measurement* akan menjalankan komunikasi I2C menggunakan fungsi I2C *Transmit* di dalam *return*-nya. Dengan fungsi I2C *Transmit* yang dijalankan tersebut, sinyal I2C pada jalur SDA akan terbentuk seperti yang ditunjukkan pada Gambar 3.8. Dengan tambahan 3 *byte*, tidak membuat strukturnya berbeda dengan sinyal pada perintah tipe transfer *Set Pointer*. Seluruh kondisi yang terjadi pada komunikasi sama persis. Perintah lain dengan tipe transfer *Set Pointer & Write Data*, yaitu *Write Auto Cleaning Interval*, secara garis besar akan memiliki struktur program fungsi yang mirip dengan perintah *Start Measurement*. Yang perlu diperhatikan adalah *pointer* alamat perintah dan parameter data apa saja yang akan dikirim sesuai dengan petunjuk pada *datasheet*.

Tipe transfer *Set Pointer & Read Data* memiliki struktur fungsi yang berbeda dengan dua tipe lainnya. Seperti yang telah dijelaskan sebelumnya, ada dua tahap komunikasi dalam tipe transfer ini, yaitu *transmit* dan *receive*. Yang perlu diperhatikan dalam pembuatan fungsi tipe transfer ini adalah ukuran data yang akan diterima dari sensor. Dengan begitu, setiap perintah membutuhkan ukuran memori yang berbeda-beda. Contohnya pada perintah *Read Measured Values*. Pada Tabel 3.1 ditunjukkan bahwa perintah ini akan menerima data sebesar 60 *bytes* jika format datanya *float* atau 30 *bytes* jika format datanya *integer*. Perbedaan tersebut terjadi karena ukuran memori yang dibutuhkan



Gambar 3.8. Diagram bit komunikasi I2C pada perintah *Start Measurement*

antara tipe data *float* dan *integer* berbeda. Menurut standar IEEE754, tipe data *float* memiliki ukuran sebesar 4 *bytes* tiap bilangannya. Sementara itu, tipe data *unsigned 16-bit integer* memiliki ukuran 2 *bytes* tiap bilangannya. Setiap kali sensor mengirim data, ada 10 parameter nilai yang dikirim, yaitu:

1. Konsentrasi massa PM1 ( $\mu\text{g}/\text{m}^3$ )
2. Konsentrasi massa PM2.5 ( $\mu\text{g}/\text{m}^3$ )
3. Konsentrasi massa PM4 ( $\mu\text{g}/\text{m}^3$ )
4. Konsentrasi massa PM10 ( $\mu\text{g}/\text{m}^3$ )
5. Konsentrasi jumlah PM0.5 (#/ $\text{cm}^3$ )
6. Konsentrasi jumlah PM1 (#/ $\text{cm}^3$ )
7. Konsentrasi jumlah PM2.5 (#/ $\text{cm}^3$ )
8. Konsentrasi jumlah PM4 (#/ $\text{cm}^3$ )
9. Konsentrasi jumlah PM10 (#/ $\text{cm}^3$ )
10. Ukuran partikel yang khas ( $\mu\text{m}$ )

Dari 10 parameter di atas, tidak lupa ditambah nilai *checksum* setiap 2 *bytes* data. Jadi, satu parameter nilai dalam tipe data *float* membutuhkan ukuran 6 *bytes* termasuk *checksum*, sedangkan satu parameter nilai dalam tipe data *integer* membutuhkan ukuran 3 *bytes* termasuk *checksum*. Dalam implementasinya pada program, kesepuluh parameter nilai tersebut dibuat dalam satu *struct* yang bernama `sps30_measurement` untuk menyimpan masing-masing nilainya. Perlu diperhatikan tipe data pada *struct* ter-

sebut. Ketika format data yang dipilih sebelumnya pada perintah *Start Measurement* adalah *float*, maka tipe data dari kesepuluh parameter nilainya adalah *float*. Akan tetapi, ketika format data yang dipilih adalah *integer*, maka tipe data yang digunakan adalah *uint16\_t*. *Struct* tersebut digunakan untuk menjadi masukan dari fungsi perintah *Read Measured Values*. Fungsi *Read Measured Values* berguna untuk mengirim perintah (*transmit*), menerima data (*receive*), dan memasukkan data yang didapat ke dalam *struct* *sps30\_measurement*. Dengan *pointer* alamat perintahnya adalah 0x0300, fungsi perintah *Read Measured Values* ditunjukkan pada baris kode berikut:

```

1 int read_measured_values( struct sps30_measurement* m){
2     uint8_t data[10][4];
3     int16_t error;
4     uint8_t buf[32];
5     uint16_t cmd = SPS30_CMD_READ_MEASURED_VALUES;
6
7     buf[0] = (uint8_t)((cmd & 0xFF00) >> 8);
8     buf[1] = (uint8_t)((cmd & 0x00FF) >> 0);
9
10    if((int8_t)HAL_I2C_Master_Transmit(&hi2c3 , SPS30_ADDR<<1,
11                                buf , COMMAND_SIZE, 100) != HAL_OK){
12        printf("error read measured values\r\n");
13    } else{
14        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, 1);
15        error=read_bytes(SPS30_ADDR,&data[0][0], NUM_WORDS(data));
16        if (error != NO_ERROR) {
17            return error;
18        }
19        m->mc_1p0 = bytes_to_float(data[0]);
20        m->mc_2p5 = bytes_to_float(data[1]);
21        m->mc_4p0 = bytes_to_float(data[2]);
22        m->mc_10p0 = bytes_to_float(data[3]);
23        m->nc_0p5 = bytes_to_float(data[4]);
24        m->nc_1p0 = bytes_to_float(data[5]);
25        m->nc_2p5 = bytes_to_float(data[6]);
26        m->nc_4p0 = bytes_to_float(data[7]);
27        m->nc_10p0 = bytes_to_float(data[8]);
28        m->typical_particle_size = bytes_to_float(data[9]);
29    }
30    return 0;
}

```

Fungsi *read\_measured\_values()* di atas digunakan untuk format data *flo-*

at. Yang pertama dilakukan pada fungsi ini adalah melakukan komunikasi I2C *transmit* perintah dari mikrokontroler ke sensor. Untuk itu *pointer* alamat 0x0300 harus dimasukkan ke dalam *buffer* sama seperti pada perintah tipe transfer sebelumnya. Apabila komunikasi I2C *transmit* ini berhasil, maka dilanjutkan dengan melakukan komunikasi I2C *receive* data. Komunikasi I2C untuk menerima data dilakukan pada fungsi lain, yaitu fungsi `read_bytes()`. Untuk itu, perlu disiapkan sebuah *buffer* yang akan dipakai untuk menyimpan nilai data yang diterima. Dalam program ini, *buffer* tersebut bernama variabel `data[10][4]`. Dimensi 10x4 menggambarkan bahwa terdapat 10 parameter nilai yang akan diterima dengan masing-masing nilainya terdiri dari 4 *bytes* data nilai *float*. Apabila format data yang akan diterima adalah *integer*, dimensi dari *buffer* tersebut harus diubah menjadi 10x2. Jika data telah berhasil diterima dan masuk ke *buffer* data, langkah selanjutnya adalah memasukkan nilai-nilai dalam *buffer* tersebut ke dalam *struct sps30\_measurement*. Akan tetapi perlu diperhatikan bahwa data tersebut masih dalam bentuk *bytes*, sedangkan *struct* tersebut memiliki tipe data *float*. Oleh karena itu, sebelum dimasukkan ke *struct*, data pada *buffer* harus diubah dahulu menjadi *float* melalui fungsi `bytes_to_float()` dengan masukannya adalah `data[0]` hingga `data[9]`. Fungsi `bytes_to_float()` ditunjukkan pada baris kode berikut:

```

1  uint32_t bytes_to_uint32_t(const uint8_t* bytes) {
2      return (uint32_t)bytes[0] << 24 | (uint32_t)bytes[1] << 16 |
3          (uint32_t)bytes[2] << 8 | (uint32_t)bytes[3];
4  }
5
6  float bytes_to_float(const uint8_t* bytes) {
7      union {
8          uint32_t u32_value;
9          float float32;
10     } uni;
11     uni.u32_value = bytes_to_uint32_t(bytes);
12     return uni.float32;
13 }
```

Fungsi `bytes_to_float()` (baris 6-14) di atas bekerja dengan memanfaatkan union. Union adalah sebuah tipe data yang memungkinkan untuk menyimpan beberapa tipe data yang berbeda pada satu lokasi memori yang sama. Pada fungsi ini, union memiliki dua tipe data yang berbeda, yaitu `uint32_t` dan `float`. Pemanfaatan union untuk mengubah data *bytes* menjadi *float* bekerja dengan cara memasukkan data *bytes* ke dalam tipe data `uint32_t`. Kemudian, nilai yang *di-return* adalah variabel dengan tipe data `float`. Karena keduanya mengakses satu lokasi memori yang sama, maka nilai yang sudah disimpan pada memori tersebut akan menjadi *float* apabila dipanggil sebagai tipe data `float`. Akan tetapi perlu diperhatikan bahwa data *bytes* yang dimasukkan ke da-

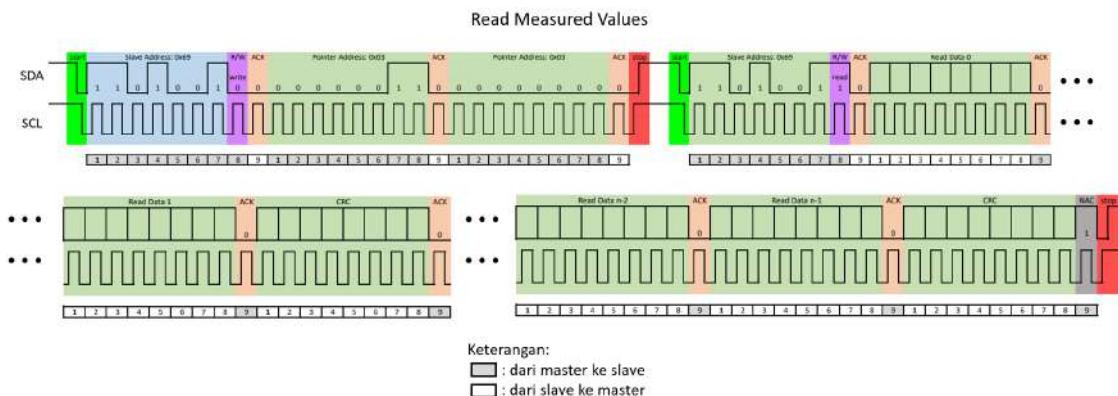
lam union tersebut harus diubah dahulu menjadi `uint32_t` karena tipe `float` berukuran 4 *bytes*. Untuk mengubah data-data *bytes* dari `uint8_t` menjadi `uint32_t` diperlukan fungsi `bytes_to_uint32_t()` seperti yang ditunjukkan pada baris 1 hingga 4 kode di atas. Data yang sudah dalam bentuk `float` dapat dimasukkan ke dalam *struct* dan siap diolah atau pun ditampilkan.

```

1 int8_t check_crc( const uint8_t* data , uint8_t checksum) {
2     if ( calc_crc(data) != checksum)
3         return STATUS_FAIL;
4     return NO_ERROR;
5 }
6
7 int16_t read_bytes(uint8_t address , uint8_t* data ,
8                     uint16_t num_words) {
9     int8_t ret;
10    uint16_t i , j ;
11    uint16_t size = num_words * (WORD_SIZE + CRC8_LEN);
12    uint16_t word_buf[32];
13    uint8_t* const buf8 = (uint8_t*)word_buf;
14
15    ret = (int8_t) HAL_I2C_Master_Receive(&hi2c3 , SPS30_ADDR<<1,
16                                         buf8 , size , 100);
17    if (ret != HAL_OK){
18        HAL_GPIO_WritePin(GPIOB , GPIO_PIN_1 , 1);
19        return STATUS_FAIL;
20    } else {
21        HAL_GPIO_WritePin(GPIOB , GPIO_PIN_0 , 0);
22    }
23    // cek CRC
24    for (i = 0 , j = 0; i < size ; i += WORD_SIZE +
25                      CRC8_LEN) {
26        ret = check_crc(&buf8[ i ] , buf8[ i + WORD_SIZE]);
27        if (ret != NO_ERROR)
28            return ret;
29
30        data[ j++ ] = buf8[ i ];
31        data[ j++ ] = buf8[ i + 1];
32    }
33    return NO_ERROR;
34 }
```

Sebelumnya dijelaskan bahwa komunikasi I2C untuk menerima data dilakukan

pada fungsi `read_bytes()`. Isi dari fungsi tersebut ditunjukkan pada barisan kode di atas (baris 7-32). Masukan fungsi tersebut adalah alamat, *pointer* ke `data[10][4]` pada fungsi `read_measured_values()`, dan jumlah *word*. *Buffer* `data[10][4]` memiliki ukuran 40 *bytes* sehingga dengan dibagi 2 didapat jumlah *word*-nya adalah 20. Kemudian pada variabel `size` dimasukkan nilai ukuran dari data yang hendak diterima, yaitu 60 untuk tipe data *float* dan 30 untuk tipe data *integer*. Pada baris 11 dibuat variabel `word_buf[32]` dengan tipe data `uint16_t` sebagai lokasi memori untuk menyimpan data yang diterima. Baris selanjutnya dibuat sebuah *pointer* konstan `buf8` dengan tipe data `uint8_t`. *Pointer* `buf8` digunakan untuk menunjuk ke satu *byte* dari 2 *byte* data pada `word_buf`. Sederhananya, `buf8` adalah *pointer* yang dapat mengakses memori yang sama seperti `word_buf` tetapi hanya mengakses nilai tiap 1 *byte*-nya. Hal itu dilakukan karena data pada komunikasi I2C dikirim setiap 1 *byte*. Setelah pendefinisian variabel-variabel yang diperlukan, fungsi ini dilanjutkan dengan melakukan komunikasi I2C *receive* dengan fungsi `HAL_I2C_Master_Receive()`. Ketika komunikasi berhasil, program dilanjutkan dengan memeriksa setiap nilai *checksum* antara yang dikirim oleh sensor dan yang dihitung berdasarkan setiap *word*-nya. Pemeriksaan *checksum* dilakukan oleh fungsi `check_crc()` yang didefinisikan pada baris 1-5 kode di atas. Setiap *word* dengan nilai *checksum* yang benar selanjutnya dimasukkan ke dalam *buffer* data untuk selanjutnya digunakan di dalam fungsi `read_measured_values()`.



Gambar 3.9. Diagram bit komunikasi I2C pada perintah *Read Measured Values*

Dengan keseluruhan fungsi untuk perintah *Read Measure Values*, sinyal I2C yang terbentuk pada jalur SDA ditunjukkan pada Gambar 3.9. Dari gambar tersebut terdapat dua komunikasi yang terjadi. Komunikasi pertama, yaitu mengirim *pointer* alamat perintah ke sensor, memiliki struktur sinyal yang sama seperti komunikasi pada tipe transfer *Set Pointer* sebelumnya. Setelah kondisi *stop* pertama, dilanjutkan dengan komunikasi kedua, yaitu menerima data dari sensor. Pada komunikasi yang kedua, bit ke-8 setelah alamat sensor bernilai 1 yang berarti master ingin membaca data dari sensor. Sama seperti sebelumnya, bit ke-8 tersebut sudah otomatis terisi ketika menggunakan fungsi I2C *receive*. Bagian *read Data 0* hingga  $n - 1$  merupakan tempat di mana data yang dikir-

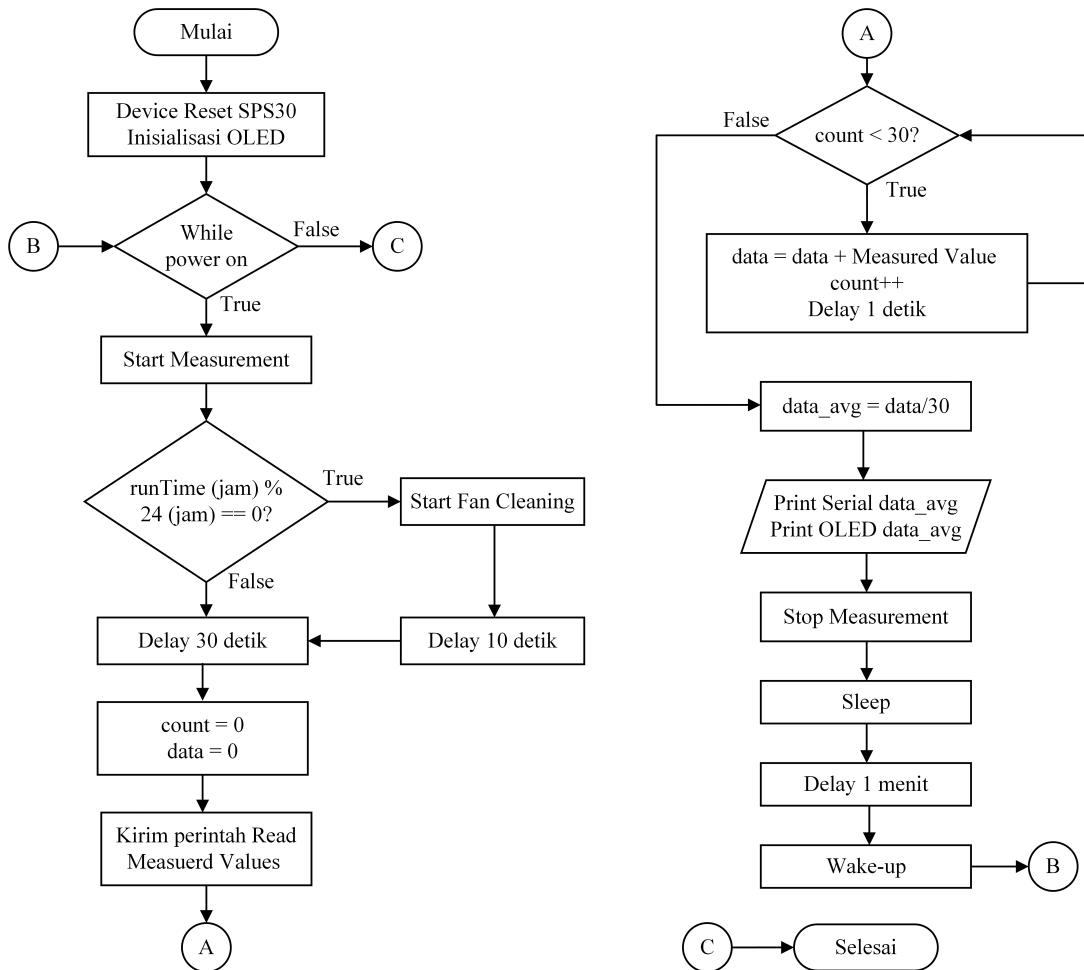
rim sensor berada. Karena pada bagian ini sensor yang mengirim data, maka bit ACK kali ini diisi oleh master yang menandakan komunikasi berhasil dan master menerima data yang dikirim sensor. Setelah semua data dikirim, bit terakhir diisi oleh NACK, bukan ACK. Hal tersebut menandakan bahwa sensor sudah tidak ingin menerima data lagi. Komunikasi selanjutnya ditutup dengan master mengirim kondisi *stop*.

Seperti yang telah dijelaskan sebelumnya, tipe transfer *Set Pointer & Read Data* tidak harus mengirim *pointer* alamat berulang kali ketika ingin membaca data secara terus menerus. *Pointer* alamat cukup dikirim sekali saja. Untuk melakukannya, fungsi untuk mengirim *pointer* alamat dapat dipisah dengan fungsi untuk membaca data. Misalnya ketika ingin membaca data pengukuran, fungsi `read_measured_values()` dapat dipisah menjadi dua fungsi yang berbeda. Jadi, dengan menggunakan metode ini, sinyal I2C yang terbentuk pada pembacaan data yang kedua dan seterusnya hanya bagian kedua dari sinyal I2C pada Gambar 3.9. Hal ini juga berlaku dengan perintah-perintah lainnya dengan tipe transfer yang sama. Perbedaan yang harus diperhatikan pada perintah lain adalah format data dan ukuran data yang akan dibaca. Dengan demikian, format variabel untuk menyimpan data dan ukuran *buffer* dapat disesuaikan pada pengembangan fungsi perintah-perintah tersebut.

### 3.6.2.2 Algoritma Pengukuran Konsentrasi Partikulat

Tahap selanjutnya setelah fungsi untuk seluruh perintah pada sensor SPS30 dibuat adalah membuat algoritma yang akan dijalankan secara terus menerus pada fungsi `main()`. Agar sensor berjalan dengan baik, karakteristik setiap perintah dan mode harus dipahami. Misalnya perintah *Sleep* yang hanya dapat dilakukan ketika sensor dalam mode *idle* atau perintah *Read Measured Values* yang hanya dapat dilakukan ketika sensor dalam mode *measurement*. Dalam hal ini, Sensirion memberikan panduan operasi di dalam dokumen *Low-Power Operation*. Dokumen tersebut memberikan rekomendasi mengenai bagaimana cara mengoperasikan sensor agar menghasilkan pengukuran yang baik dan dapat menghemat daya.

Hal pertama yang harus diperhatikan adalah waktu *start-up*. Waktu *start-up* adalah waktu yang dibutuhkan sensor untuk mendapatkan hasil pembacaan sensor stabil. Sensor menggunakan kipas di dalamnya untuk melakukan pengukuran. Ketika masuk ke mode *measurement* kipas mulai berputar. Akan tetapi, karena adanya inersia, kipas membutuhkan waktu untuk mencapai target kecepatan putarnya. Waktu yang dibutuhkan untuk mencapai pengukuran yang stabil tergantung pada konsentrasi partikulat di udara. Konsentrasi rendah membutuhkan waktu yang lebih lama daripada konsentrasi yang lebih tinggi. Direkomendasikan untuk angka konsentrasi  $> 300 \text{#/cm}^3$  memiliki waktu *start-up* selama 16 detik dan untuk angka konsentrasi  $\leq 300 \text{#/cm}^3$  memiliki waktu *start-up* selama 30 detik.



Gambar 3.10. Diagram alir program

Setelah menunggu waktu yang telah ditentukan ketika masuk mode *measurement*, selanjutnya direkomendasikan untuk mengambil beberapa sampel dan menghitung rata-ratanya. Pengukuran baru diperbarui setiap satu detik. Maka, antara satu pengukuran dengan pengukuran berikutnya diberi jeda waktu satu detik. Yang perlu diperhatikan adalah semakin lama periode pengambilan sampel, semakin besar daya yang dibutuhkan. Hal itu karena mode *measurement* membutuhkan kuat arus sebesar 45-65 mA, sedangkan mode *idle* hanya membutuhkan kuat arus sekitar 330  $\mu$ A. Untuk itu, dalam upaya penghematan daya, hal kedua yang harus diperhatikan adalah waktu dalam mode *sleep*. Kuat arus yang dibutuhkan dalam mode *sleep* adalah <50  $\mu$ A. Akan tetapi, di dalam perancangan ini, catu daya tidak menggunakan baterai sehingga hal itu bukan masalah. Akibatnya, sensor tidak perlu menghabiskan banyak waktu di dalam mode *sleep* sehingga data yang masuk dapat lebih banyak.

Implementasi operasi yang dilakukan pada perancangan ini diwujudkan dalam fungsi main() dengan diagram alirnya ditunjukkan pada Gambar 3.10. Dari diagram alir tersebut, dapat dilihat bahwa ketika rangkaian pada Gambar 3.5 diberi sumber daya, program dimulai dengan melakukan inisialisasi sensor SPS30 dan layar OLED. Sensor

diinisialisasi dengan menjalankan perintah *Device Reset*. Layar OLED diinisialisasi dengan fungsi yang tersedia pada *library*-nya. Maka dari itu, sebelum masuk ke dalam *main()* tidak lupa untuk memasukkan *library* layar OLED tersebut. Setelah inisialisasi, program dilanjutkan dengan *loop while()* yang akan dijalankan kontinu selama rangkaian mendapat catu daya. Perintah pertama yang dijalankan adalah *Start Measurement*. Dengan begitu, sensor saat ini masuk ke dalam mode *measurement* dan siap untuk mengambil data.

Sebelum mengambil data, dilakukan pengecekan kondisi untuk perintah *Start Fan Cleaning*. Pembersihan kipas sebenarnya dapat dijalankan secara otomatis dengan interval standarnya adalah setiap satu minggu. Akan tetapi, hal tersebut hanya dapat dilakukan ketika sensor berada dalam mode *measurement* terus menerus, padahal itu tidak dilakukan pada program ini. Untuk itu, pada perancangan ini, pembersihan kipas dilakukan setiap hari dengan memeriksa waktu jalannya program. Lebih tepatnya memeriksa jumlah data yang telah diambil dan dibandingkan dengan interval pengambilan data. Dalam hal ini data diambil setiap 2 menit. Mengingat 1 hari terdiri dari 1440 menit, maka ketika sudah ada 720 data yang diambil, perintah *Start Fan Cleaning* dilakukan. Pembersihan kipas dilakukan dengan memutar kipas hingga kecepatan maksimumnya selama 10 detik, sehingga dibutuhkan jeda waktu 10 detik sebelum melanjutkan pada perintah selanjutnya. Tujuannya adalah untuk meniup keluar debu yang terakumulasi pada kipas.

Saat bukan waktunya untuk pembersihan kipas atau proses pembersihan kipas sudah selesai, program dilanjutkan dengan memberikan jeda waktu selama 30 detik (waktu *start-up*) agar kipas dapat berputar dengan stabil pada kecepatan normalnya di mode *measurement*. Selanjutnya akan dilakukan pengambilan sampel selama 30 detik. Untuk itu, *pointer* alamat perintah *Read Measured Value* dikirim dan masuk ke *loop while* selama 30 detik. Di dalam *loop*, dilakukan pengambilan 30 sampel dengan jeda antar sampelnya adalah 1 detik. Sampel-sampel tersebut kemudian dirata-rata. Hasil rata-rata inilah yang dikirim ke komputer secara serial dengan komunikasi UART dan ke OLED dengan komunikasi I2C untuk ditampilkan. Data yang dikirim ke komputer adalah 10 parameter nilai yang diukur dengan dipisahkan koma setiap parameter nilainya. Implementasinya pada program ditunjukkan pada kode di bawah ini. Data tersebut kemudian disimpan dalam format *.txt* yang selanjutnya akan diubah menjadi format *.csv* untuk diolah.

```
1 printf ("%0.2f,%0.2f,%0.2f,%0.2f,%0.2f,%0.2f,%0.2f,%0.2f,%0.2f\n",
2 ,%0.2f\r\n", m.mc_1p0, m.mc_2p5, m.mc_4p0, m.mc_10p0,
3 m.nc_0p5, m.nc_1p0, m.nc_2p5, m.nc_4p0, m.nc_10p0,
4 m.typical_particle_size);
```

Untuk keluar dari mode *Measurement*, digunakan perintah *Stop Measurement*. Dengan demikian, sensor sudah kembali ke mode *idle*. Perintah *Sleep* pun dapat dijalankan. *Sleep* dilakukan selama 1 menit. Setelah itu, perintah *Wake-up* dijalankan dan

program kembali ke awal *loop while* untuk mengulang kembali yang dimulai dari perintah *Start Measurement*. Dengan waktu *start-up* selama 30 detik, pengambilan 30 sampel selama 30 detik, dan *sleep* selama 1 menit, maka data pengukuran baru diperoleh setiap 2 menit.

### 3.6.3 Pengolahan Data

Untuk kebutuhan pengambilan data secara kontinu, komputer yang digunakan sebagai catu daya mikrokontroler dan lokasi penyimpanan data adalah Raspberry Pi 3 Model B. Dari periferal USB Raspberry Pi, dihubungkan kabel USB untuk menyalakan mikrokontroler. Melalui kabel tersebut juga komunikasi UART untuk mengirim data dari mikrokontroler ke Raspberry Pi terjadi. Di dalam Raspberry Pi, data dari mikrokontroler ditangkap dengan menggunakan *software* Grabserial. Grabserial dapat menangkap data, menambahkan *timestamp*, dan menyimpannya dalam format *.txt*. Untuk menjalankannya, Grabserial diaktifkan melalui terminal dengan menggunakan perintah *grabserial* yang dilanjutkan dengan memasukkan parameter *port*, *baudrate*, format *timestamp*, dan nama *file .txt*. Setiap kali data masuk, data akan tersimpan pada *file* tersebut. Data diolah setelah mengubah format *file* tersebut menjadi *.csv*.

Data 10 parameter nilai dengan *timestamp*-nya setiap 2 menit selanjutnya diolah. Tujuannya adalah untuk mendapatkan indeks kualitas udara berdasarkan ISPU. Menurut ISPU, nilai indeks digunakan untuk menggambarkan bagaimana kondisi kualitas udara di suatu lokasi dalam 24 jam terakhir. Maka dari itu, data yang diperoleh dicari rataratanya dalam 24 jam terakhir. Dari sini perlu diperhatikan bahwa pengambilan data harus dilakukan setidaknya selama 24 jam untuk dapat memperoleh nilai ISPU. Nilai rata-rata selama 24 jam dihitung menjadi ISPU dengan menggunakan persamaan 2-1 dengan nilai-nilai variabelnya merujuk pada Tabel 2.1. Berdasarkan persamaan dan tabel tersebut, algoritma perhitungannya ditunjukkan pada Algoritma 2 untuk PM2.5 dan Algoritma 3 untuk PM10. Dari nilai ISPU, dapat diketahui bagaimana kondisi kualitas udara di lokasi pengujian, apakah baik, sedang, tidak sehat, sangat tidak sehat, atau berbahaya.

---

#### Algorithm 2 Perhitungan ISPU PM2.5

---

```
1: if pm25 ≥ 0 and pm25 ≤ 15,5 then
2:     Ia = 50
3:     Ib = 0
4:     pa = 15,5
5:     pb = 0
6:     ispu25 = ((Ia - Ib)/(pa - pb)) * (pm25 - pb) + Ib
7: else if pm25 > 15,5 and pm25 ≤ 55,4 then
8:     Ia = 100
9:     Ib = 51
```

---

---

```

10:    $p_a = 55, 4$ 
11:    $p_b = 15, 5$ 
12:    $ispu25 = ((I_a - I_b)/(p_a - p_b)) * (pm25 - p_b) + I_b$ 
13: else if  $pm25 > 55, 4$  and  $pm25 \leq 150, 4$  then
14:      $I_a = 200$ 
15:      $I_b = 101$ 
16:      $p_a = 150, 4$ 
17:      $p_b = 55, 4$ 
18:      $ispu25 = ((I_a - I_b)/(p_a - p_b)) * (pm25 - p_b) + I_b$ 
19: else if  $pm25 > 150, 4$  and  $pm25 \leq 250, 4$  then
20:      $I_a = 300$ 
21:      $I_b = 201$ 
22:      $p_a = 250, 4$ 
23:      $p_b = 150, 4$ 
24:      $ispu25 = ((I_a - I_b)/(p_a - p_b)) * (pm25 - p_b) + I_b$ 
25: else if  $pm25 > 250, 4$  then
26:    $ispu25 = " > 300"$ 
27: end if

```

---

### Algorithm 3 Perhitungan ISPU PM10

---

```

1: if  $pm10 \geq 0$  and  $pm10 \leq 50$  then
2:    $I_a = 50$ 
3:    $I_b = 0$ 
4:    $p_a = 50$ 
5:    $p_b = 0$ 
6:    $ispu10 = ((I_a - I_b)/(p_a - p_b)) * (pm10 - p_b) + I_b$ 
7: else if  $pm10 > 50$  and  $pm10 \leq 150$  then
8:    $I_a = 100$ 
9:    $I_b = 51$ 
10:   $p_a = 150$ 
11:   $p_b = 50$ 
12:   $ispu10 = ((I_a - I_b)/(p_a - p_b)) * (pm10 - p_b) + I_b$ 
13: else if  $pm10 > 150$  and  $pm10 \leq 350$  then
14:    $I_a = 200$ 
15:    $I_b = 101$ 
16:    $p_a = 350$ 
17:    $p_b = 150$ 
18:    $ispu10 = ((I_a - I_b)/(p_a - p_b)) * (pm10 - p_b) + I_b$ 
19: else if  $pm10 > 350$  and  $pm10 \leq 420$  then
20:    $I_a = 300$ 
21:    $I_b = 201$ 
22:    $p_a = 420$ 

```

---

---

```

23:      $p_b = 350$ 
24:      $ispu10 = ((I_a - I_b)/(p_a - p_b)) * (pm10 - p_b) + I_b$ 
25: else if  $pm10 > 420$  then
26:      $ispu10 = " > 300"$ 
27: end if

```

---

Hasil perhitungan ISPU dengan dua algoritma di atas dilakukan pada *file .csv*. Kolom-kolom baru ditambahkan di sebelah kanan kolom *timestamp* dan 10 parameter nilai. Untuk melihat bagaimana kualitas udara dari di lokasi dari waktu ke waktu, data-data tersebut diplot menjadi grafik dengan menggunakan program Python. Untuk itu setiap kolomnya perlu diberi nama pada baris pertama untuk dapat membaca nilai-nilainya. Membuka dan membaca nilai pada program Python *file .csv* dapat dilakukan dengan fungsi `read_csv()` pada *library* pandas. Sementara itu, pembuatan grafik dapat dilakukan dengan fungsi-fungsi yang tersedia pada `matplotlib`.

### 3.6.4 Batasan Sistem

Sistem yang dirancang ini memiliki batasan dan kebutuhan yang harus dipenuhi oleh pengguna. Untuk lebih jelasnya dapat dilihat pada Tabel 3.2 berikut ini.

Tabel 3.2. Spesifikasi dan Batasan Sistem

No.	Parameter	Spesifikasi yang digunakan	Batasan Sistem	Keterangan
1	Tegangan masukan	USB VBUS (5 V)	USB VBUS atau catu daya eksternal (5 V atau 7-12 V)	Penjelasan A
2	Arus pada pin I/O sensor	0,5 mA	Rentang -16 hingga 16 mA	Penjelasan B
3	Antarmuka sensor	I2C	Panjang kabel tidak lebih dari 20 cm	Penjelasan C
4	<i>Data logger</i>	Raspberry Pi	Tidak harus menggunakan <i>data logger</i>	Penjelasan D
5	Temperatur	-	-40 hingga 70°C	Penjelasan E
6	<i>Humidity</i>	-	0 hingga 95 %RH	

- Penjelasan A

Catu daya ke mikrokontroler dapat menggunakan alat apa saja (lihat Penjelasan D), asalkan memenuhi spesifikasi tegangan. Misalnya dapat langsung menggunakan

adaptor 5V atau menggunakan baterai. Untuk penggunaan baterai perlu diperhatikan daya tahannya dengan menghitung daya yang dikonsumsi sistem ini. Pada perancangan ini menggunakan kabel USB dari komputer karena kebutuhan mengirim data ke komputer.

- Penjelasan B

Arus yang perlu diperhatikan di sini adalah pada pin SDA dan SCL. Kedua pin tersebut memerlukan *pull-up resistor*. Besar nilai masing-masing *pull-up resistor* perlu diperhatikan dengan menghitungnya menggunakan persamaan  $I = \frac{V}{R}$  agar arus yang mengalir tidak melewati rentang yang diperbolehkan. Pada perancangan ini, digunakan *resistor* sebesar  $10\text{ k}\Omega$  karena arus yang mengalir sudah sesuai.

- Penjelasan C

Komunikasi I2C dirancang untuk komunikasi jarak pendek. Direkomendasikan untuk tidak menggunakan kabel lebih dari 20 cm untuk menghindari *electronic interference (EMI) noise*. Pada perancangan ini kabel untuk jalur SDA dan SCL memiliki panjang 10 cm.

- Penjelasan D

*Data logger* tidak perlu digunakan jika tidak ingin menyimpan data pengukuran. Sehingga, catu daya dapat langsung menggunakan adaptor atau baterai. Pada perancangan ini, *data logger* digunakan karena kebutuhan menyimpan data dan mengolahnya.

- Penjelasan E

Operasi pada lingkungan ekstrem akan memengaruhi tingkat presisi dan umur sistem. Sistem ini akan diuji pada lingkungan pada kondisi ruangan normal.

### 3.7 Pengujian Sistem

Perancangan menggunakan Raspberry Pi atau program yang dijalankan yang dijelaskan pada bagian sebelumnya dapat diubah-ubah sesuai dengan pengujian yang akan dilakukan. Perancangan program dan pengolahan data tersebut dikhususkan pada pengujian pengambilan data untuk mengetahui indeks kualitas udara di lokasi pengujian. Dalam penelitian ini, sistem yang dirancang dilakukan beberapa pengujian, sebagai berikut:

1. Pengujian seluruh perintah pada sensor SPS30. Pada penjelasan sebelumnya telah dipaparkan bahwa seluruh perintah yang dimiliki sensor dibuat fungsi program untuk melakukannya. Fungsi-fungsi tersebut diuji coba untuk melihat apakah fungsi berjalan sesuai dengan perintahnya. Pengujian ini dilakukan dengan menjalankan masing-masing fungsi secara bergantian dan melihat respons atau keluaran dari

sensor, baik itu nilai yang dikirim ke komputer atau dilihat dengan menggunakan multimeter dan osiloskop.

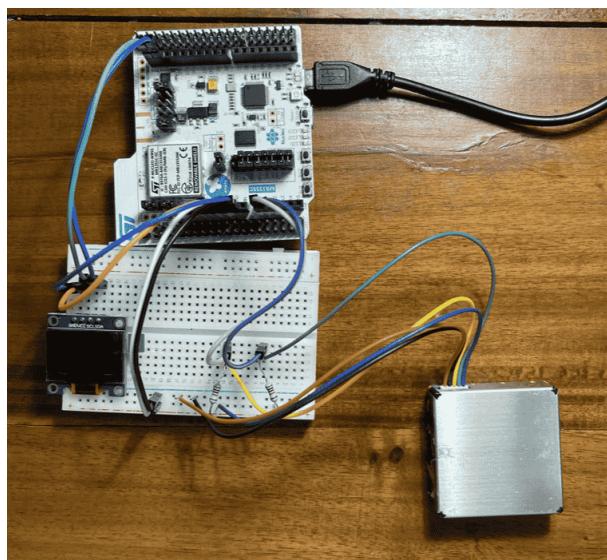
2. Pengujian komunikasi I2C antara sensor dan mikrokontroler. Pengujian dilakukan dengan menjalankan beberapa fungsi yang telah dibuat. Dari fungsi tersebut akan dikirim sinyal I2C antara sensor SPS30 dan *board* P-Nucleo-WB55. Kemudian akan dilihat dan dianalisis kondisi-kondisi yang terjadi pada sinyal I2C saat menjalankan fungsi tersebut.
3. Pengujian waktu *start-up* sensor. Pengujian ini dilakukan untuk melihat bagaimana pengaruh waktu *start-up* saat masuk mode *measurement* terhadap parameter-parameter nilai yang diukur. Pengujian dilakukan untuk membuktikan pernyataan *datasheet* dan dokumen lainnya bahwa dibutuhkan waktu *start-up* untuk menghasilkan nilai pengukuran yang stabil.
4. Pengujian sensor SPS30 untuk mengukur konsentrasi partikulat di beberapa lokasi pengujian. Lokasi pengujian terbagi menjadi tiga bagian, yaitu ruang produksi, area *outdoor*, dan ruang *showroom*. Pengujian dilakukan dengan menjalankan program seperti yang ditunjukkan pada diagram alir pada Gambar 3.10. Masing-masing lokasi diuji selama kurang lebih 1 minggu, lebih tepatnya mencakup hari kerja dan hari libur di UGM Press. Pengujian ini dilakukan untuk melihat dan menganalisis indeks kualitas udara di ketiga lokasi tersebut.

## BAB IV

# HASIL DAN PEMBAHASAN

### 4.1 Pengujian Seluruh Perintah pada sensor SPS30

Pada Bab 3 telah dijelaskan bahwa seluruh perintah yang dimiliki sensor SPS30 dibuat dalam fungsinya masing-masing. Pengujian ini bertujuan untuk memeriksa bahwa fungsi yang telah dibuat dalam program *main.c* dapat berjalan sesuai dengan apa yang tercantum pada *datasheet*. Fungsi harus dipastikan dapat berjalan dengan benar agar selanjutnya sensor dapat digunakan untuk pemantauan konsentrasi partikulat. Untuk itu, masing-masing fungsi diuji secara bergantian dengan mengubah program pada fungsi *main()*. Namun, sebelum itu, komponen dirangkai terlebih dahulu sesuai dengan rangkaian pada Gambar 3.5. Catu daya yang digunakan adalah daya dari laptop melalui kabel USB. Laptop digunakan sekaligus untuk *serial monitor* untuk melihat tanggapan atau keluaran dari sensor ketika diuji. Dengan demikian hasil rangkaianya ditunjukkan pada Gambar 4.1 di bawah ini.



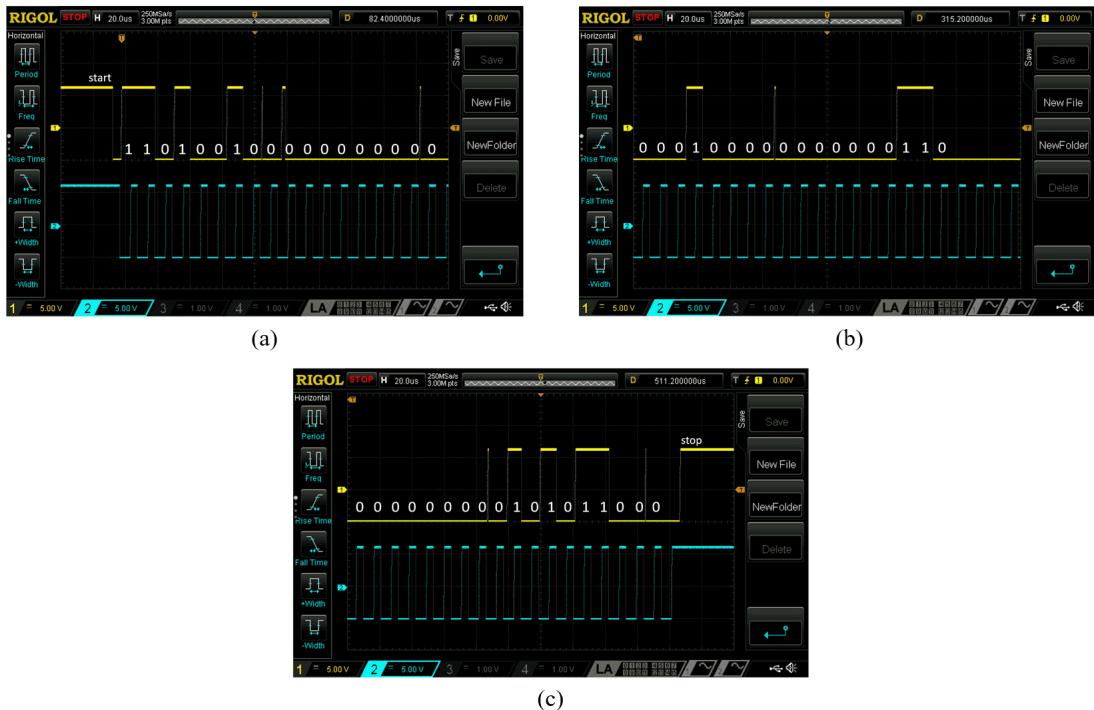
Gambar 4.1. Rangkaian sistem

#### 4.1.1 Start Measurement

Perintah *Start Measurement* dilakukan dengan menjalankan program yang berisi fungsi *start\_measurement()* yang telah dijelaskan pada Bab 3. Perintah ini berguna untuk masuk ke mode *measurement*. Pada mode ini semua elektronik, termasuk kipas dan laser dinyalakan sehingga konsumsi dayanya maksimal. Dengan begitu pengukuran konsentrasi partikulat dan pemrosesan data pengukuran berjalan secara kontinu. Nilai pengukuran baru tersedia setiap satu detik. Untuk menjalankannya, harus dipastikan bahwa

sensor berada dalam mode *idle* ketika akan melakukan perintah ini. Karena sensor sudah berada dalam mode *idle* ketika dinyalakan, maka tidak diperlukan semacam inisiasi sebelum menjalankan fungsi ini.

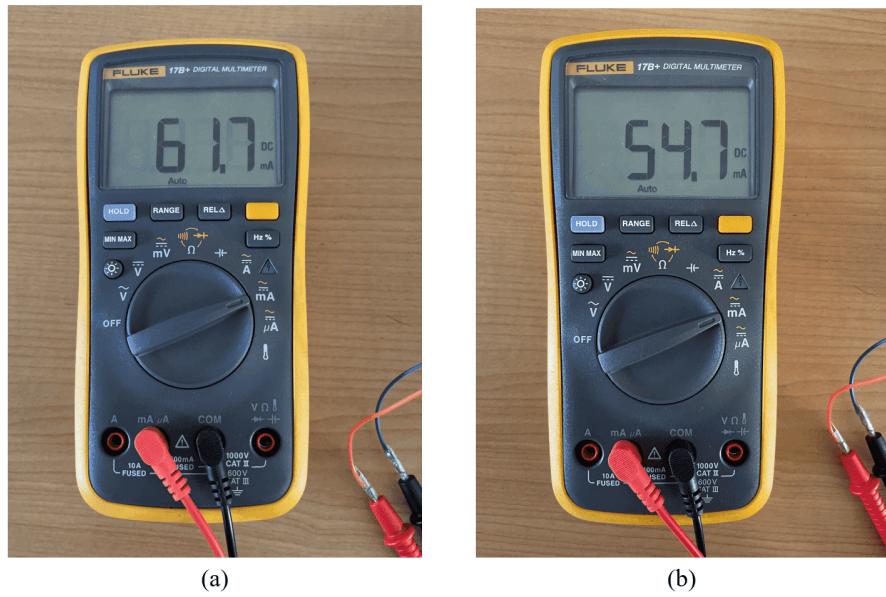
Perintah *Start Measurement* terdapat dua pilihan, yaitu format pengukuran data dalam *float* atau *integer*. Perbedaan keduanya dapat terlihat ketika data pengukuran dibaca menggunakan perintah *Read Measured Values* yang akan dijelaskan pada bagian selanjutnya. Kedua pilihan tersebut diuji secara bergantian. Ketika program dijalankan sinyal I2C yang terbentuk ditunjukkan pada Gambar 4.2.



Gambar 4.2. Sinyal I2C perintah *Start Measurement* pada osiloskop

Dari gambar tersebut dapat dilihat bahwa sinyal yang terbentuk pada kedua jalur SDA dan SCL sesuai dengan Gambar 3.8. *Byte* pertama berisi alamat sensor, yaitu 0x69 (bilangan heksadesimal) atau 1101001 (bilangan biner) dan diikuti dengan bit *write*, yaitu 0. *Byte* kedua dan ketiga berisi *pointer* alamat perintah, yaitu 0x0010 atau 00000000 00010000 dalam biner. *Byte* keempat berisi format yang di dalam gambar tersebut adalah format *float*, yaitu 0x03 atau 00000011 dalam biner. Jika format yang dipilih adalah *integer*, maka *byte* ini bernilai 0x05 atau 00000101. Bit kelima berisi *dummy byte*, yaitu 0x00. Bit keenam berisi *checksum*, yaitu 0xAC atau 01010110 dalam biner. Di setiap *byte* diikuti dengan dengan bit ACK yang bernilai 0. Hal tersebut menunjukkan pada pengujian ini sensor SPS30 dapat menerima seluruh sinyal pada perintah ini. Detail dari sinyal I2C dibahas lebih lanjut pada sub bab 4.2.

Setelah perintah *Start Measurement* dijalankan, kini sensor masuk ke mode *measurement*. Hal ini ditandai dengan nilai kuat arus yang ditunjukkan pada Gambar 4.3.



Gambar 4.3. Kuat arus yang mengalir pada mode *measurement*

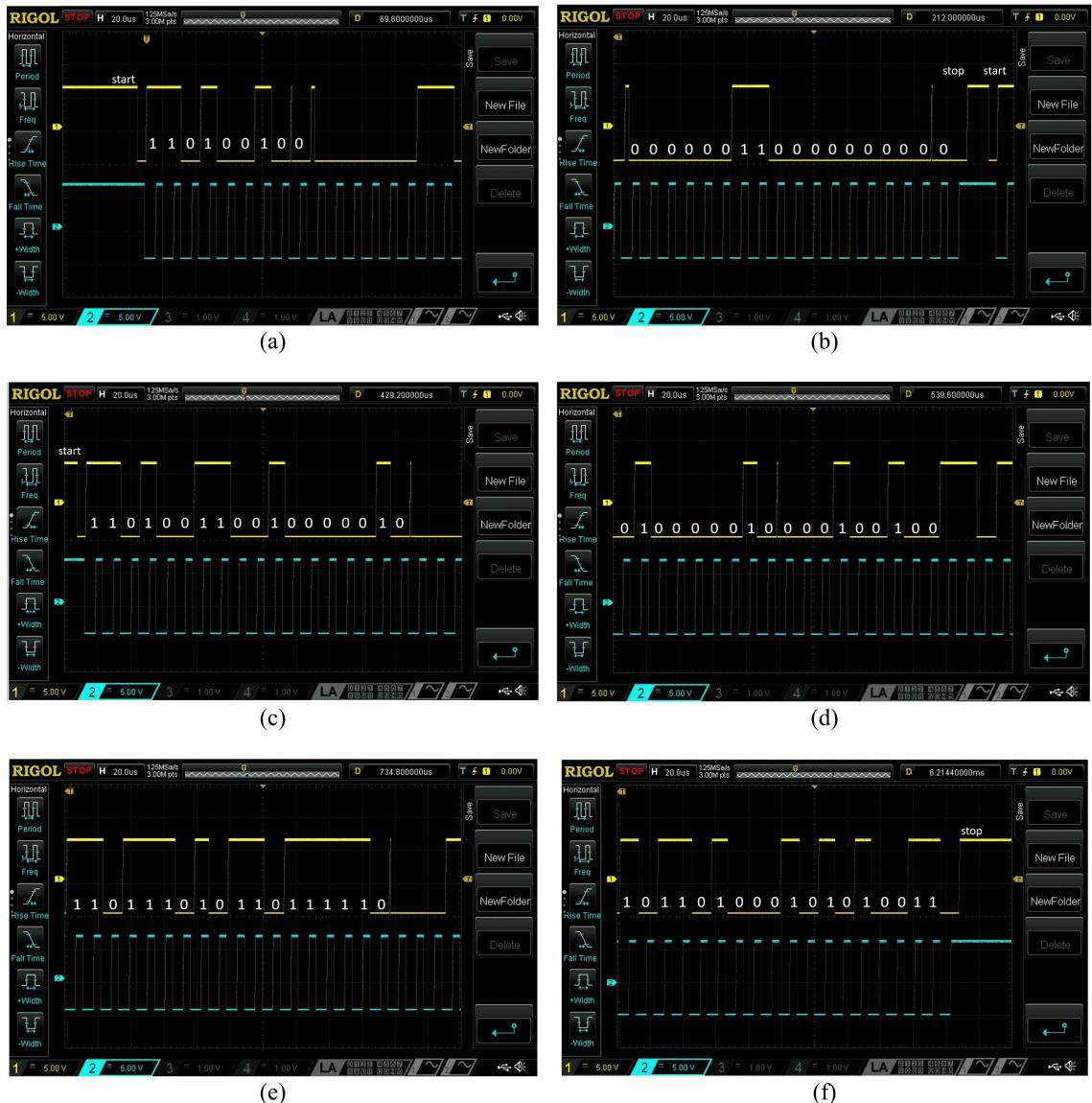
Ketika dalam mode *measurement*, arus yang mengalir awalnya bernilai sekitar 61,7 mA (Gambar 4.3 (a)). Nilai arus menurun perlahan-lahan hingga stabil pada nilai 54,7 mA (Gambar 4.3 (b)). Nilai tersebut sudah sesuai dengan nilai kuat arus yang tercantum dalam *datasheet*, yaitu 45 - 65 mA. Oleh karena itu, fungsi `start_measurement()` telah berhasil menjalankan perintah *Start Measurement*.

#### 4.1.2 *Read Measured Values*

Perintah *Read Measured Values* dilakukan dengan menjalankan program yang berisi fungsi `read_measures_values()`. Perintah ini baru bisa dijalankan ketika sensor SPS30 berada dalam mode *measurement*. Jadi, fungsi `start_measurement()` perlu dijalankan sebelum membaca data pengukuran. *Struct* `sps30_measurement` yang merupakan masukan dari fungsi dan tempat menyimpan data hasil pengukuran disesuaikan dengan format data yang dipilih perintah *Start Measurement*. Pengujian ini dilakukan terhadap kedua format data tersebut untuk memastikan bahwa sensor dapat menjalankan pengukuran dalam kedua format tersebut.

Pengujian pertama dilakukan untuk format data *float*. Sinyal I2C yang terbentuk saat menjalankan perintah `read_measured_values()` dalam format *float* ditunjukkan pada Gambar 4.4. Sinyal I2C yang terbentuk terbagi menjadi dua bagian sama seperti pada Gambar 3.9. Bagian pertama adalah *transmit pointer* alamat perintah 0x0300 atau 00000011 00000000 dalam biner. Pada bagian pertama, sensor menanggapi setiap *byte*-nya dengan mengirim bit ACK atau 0 yang berarti perintah diterima dengan baik. Maka dari itu, setelah kondisi stop, sinyal I2C dilanjutkan dengan sensor mengirim data pengukuran ke mikrokontroler. Data pengukuran terdiri dari 60 *byte* setelah 1 *byte*

alamat sensor. Setiap *byte* sinyal yang dikirim oleh sensor ditanggapi dengan bit ACK oleh mikrokontroler. Artinya data dapat diterima. Khusus *byte* terakhir, mikrokontroler mengirim bit NACK karena seluruh data sudah diterima dan tidak mau menerima data lagi dari sensor. Sinyal kemudian dilanjutkan dengan kondisi stop. Dari kedua bagian sinyal yang terbentuk, dapat dilihat perbedaannya pada 1 bit setelah 7 bit alamat sensor. Karena bagian pertama mikrokontroler hendak mengirim perintah, maka bit tersebut bernilai 0 atau *write*. Sedangkan pada bagian kedua mikrokontroler hendak menerima data maka bit tersebut bernilai 1 atau *read*.



Gambar 4.4. Sinyal I2C Perintah *Read Measured Values* dengan format data *float* pada osiloskop. (a) dan (b) merupakan bagian pertama, yaitu mengirim *pointer* alamat perintah. (c), (d), dan (e) merupakan bagian kedua, yaitu alamat sensor dan data pengukuran dari sensor. (f) merupakan akhir dari bagian kedua.

Hasil pengukuran tersebut kemudian ditampilkan oleh *serial monitor*. Untuk membuktikan bahwa data yang diterima benar sesuai dengan yang ditampilkan *serial*

*monitor*, setiap bit pada sinyal I2C diubah secara manual menjadi *float*. Perhitungannya dijelaskan sebagai berikut. Parameter nilai pertama, yaitu konsentrasi massa PM1, berada pada *byte* pertama hingga keenam. Enam *byte* tersebut adalah 010000010 000100100 110111010 110111110 000101100 001110010. Bit ke-9 pada tiap *byte* tidak dipakai karena merupakan bit ACK. Bit ke-3 dan ke-6 merupakan *checksum*. Jadi nilai sesungguhnya terletak pada bit pertama, kedua, keempat, dan kelima. Nilai 4 *byte* tersebut dikelompokkan menjadi tiga bagian sebagai berikut:

Tabel 4.1. Pengelompokan Bit Bilangan *Float* pada Nilai Konsentrasi Massa PM1

Tanda	Eksponen	Mantissa
0	10000010	0010010110111100010110

1. Bit tanda bernilai 0 berarti bilangannya positif.  $S = 0$
2. Bit eksponen dalam basis desimal bernilai 130. Nilai tersebut harus dikurangi dengan biasanya, yaitu 127.

$$\begin{aligned} E &= 130 - 127 \\ &= 3 \end{aligned} \tag{4-1}$$

3. Nilai bit *mantissa* (m) dikonversi menjadi *mantissa* basis desimal (M) dengan persamaan 2-9.

$$\begin{aligned} M &= 0 + 0 + 1 * 2^{-3} + 0 + 0 + 1 * 2^{-6} + 0 + 1 * 2^{-8} + 1 * 2^{-9} + 0 \\ &\quad + 1 * 2^{-11} + 1 * 2^{-12} + 1 * 2^{-13} + 1 * 2^{-14} + 1 * 2^{-15} + 0 + 0 \\ &\quad + 0 + 1 * 2^{-19} + 0 + 1 * 2^{-21} + 1 * 2^{-22} + 0 \end{aligned} \tag{4-2}$$

$$M = 0,1474330425$$

4. Nilai variabel-variabel yang telah didapat dimasukkan ke dalam persamaan 2-10 menjadi:

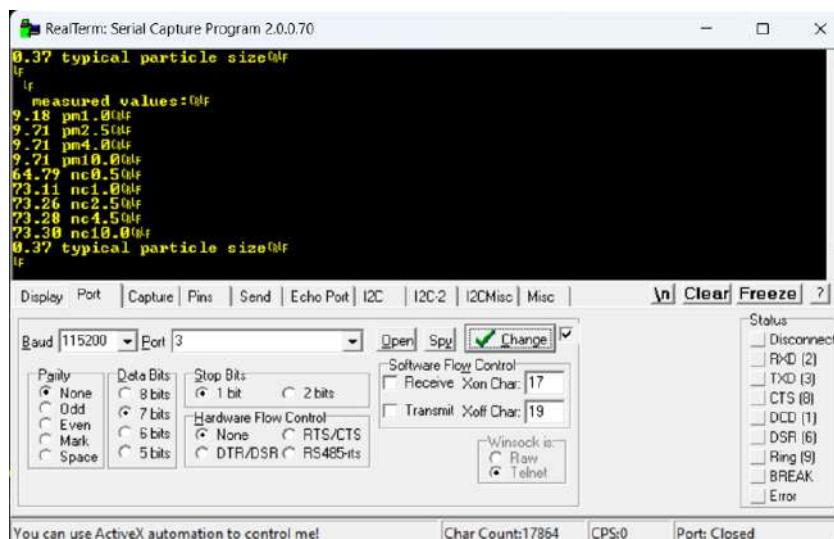
$$\begin{aligned} D &= (-1)^0 * (1 + 0,1474330425) * 2^3 \\ &= 1,1474330425 * 8 \\ &= 9,179464 \end{aligned} \tag{4-3}$$

Dari perhitungan di atas didapat bahwa nilai konsentrasi massa PM1 yang diukur dan dikirim sensor adalah  $9,179464 \mu\text{g}/\text{m}^3$ . Hasil pengamatan sinyal I2C yang terbentuk untuk parameter nilai lainnya dan hasil konversinya menjadi bilangan *float* tertera pada Tabel 4.2. Nilai dalam tabel tersebut sudah bersih tanpa bit ACK dan *byte checksum*. Kesepuluh parameter nilai yang tertangkap pada *serial monitor* ditunjukkan pada Gam-

bar 4.5. Gambar dan tabel tersebut menunjukkan bahwa sinyal yang terbentuk pada jalur SDA dan apa yang ditampilkan *serial monitor* melalui komunikasi UART memiliki nilai yang sama. Nilai *float* yang didapat sudah sesuai dengan standar IEEE 754. Dengan begini perintah *Read Measured Values* untuk format data *float* telah berjalan dengan benar.

Tabel 4.2. Nilai Pengukuran dalam Bit dan Konversinya Menjadi *Float*

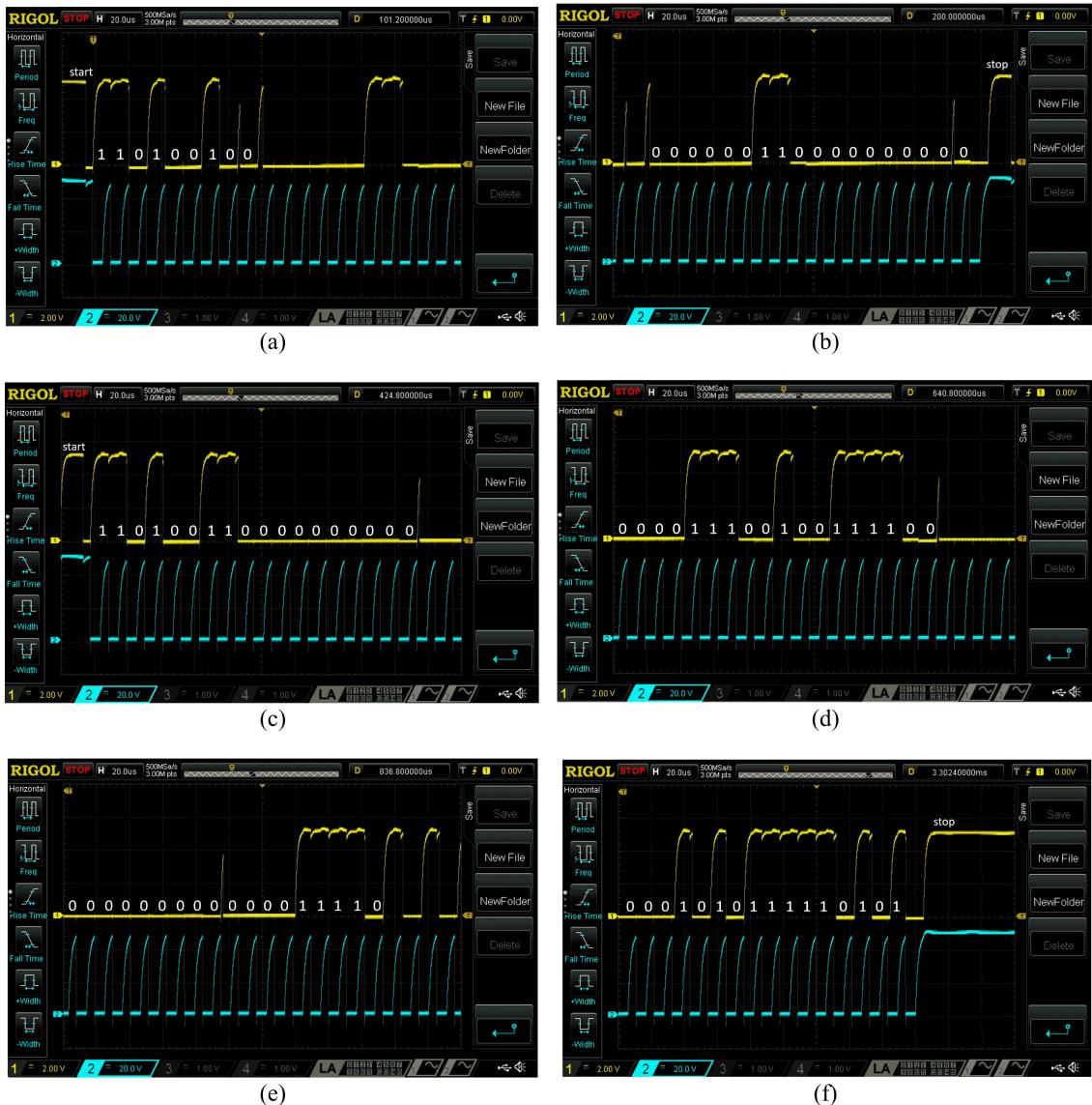
Jenis	Data[1]	Data[2]	Data[3]	Data[4]	<i>Float</i>
MC PM1	01000001	00010010	11011111	00010110	9,179464
MC PM2.5	01000001	00011011	01001111	10011011	9,706935
MC PM4	01000001	00011011	01001111	10011010	9,706934
MC PM10	01000001	00011011	01001111	10010111	9,706931
NC PM0.5	01000010	10000001	10010100	01111111	64,79003
NC PM1	01000010	10010010	00110111	10110010	73,10878
NC PM2.5	01000010	10010010	10000101	00010001	73,25989
NC PM4	01000010	10010010	10001111	10101111	73,28063
NC PM10	01000010	10010010	10011000	00111010	73,29732
<i>Typical particle size</i>	00111110	10111101	01011100	10110100	0,369847



Gambar 4.5. Nilai pengukuran format data *float* pada *serial monitor*. jumlah angka di belakang koma dibulatkan menjadi 2 angka.

Setelah perintah dengan data *float* berhasil, pengujian dilanjutkan untuk format data *integer*. Kali ini, program diunggah ulang setelah mengubah format data pada fungsi *start\_measurement ()* menjadi *integer*. Begitu juga dengan *struct* yang diubah tipe datanya menjadi *uint16\_t* dan *buffer* data pada fungsi *read\_measured\_values ()* diubah ukurannya menjadi *10x2*. Dengan program yang baru ini, sinyal I2C yang terbentuk saat perintah membaca nilai pengukuran ditunjukkan pada Gambar 4.6. Struktur

sinyal yang terbentuk sama persis seperti dengan tipe data *float*. Bagian pertama adalah *transmit pointer* alamat perintah dan bagian kedua adalah nilai pengukuran. Yang membedakannya adalah panjang sinyalnya lebih pendek karena tiap parameter nilai hanya membutuhkan 3 byte termasuk *checksum*-nya. Dengan begitu, data pengukuran terdiri dari 30 byte setelah 1 byte alamat sensor. Pada komunikasi kali ini, setiap byte sinyal diikuti dengan bit ACK yang berarti mikrokontroler menerima nilai yang dikirim. Sama seperti sebelumnya, khusus byte terakhir, mikrokontroler mengirim bit NACK karena sudah semua data diterima.



Gambar 4.6. Sinyal I2C Perintah *Read Measured Values* dengan format data *integer* pada osiloskop. (a) dan (b) merupakan bagian pertama, yaitu mengirim *pointer* alamat perintah. (c), (d), dan (e) merupakan bagian kedua, yaitu alamat sensor dan data pengukuran dari sensor. (f) merupakan akhir dari bagian kedua.

Hasil pengukuran kesepuluh parameter nilai juga ditampilkan pada *serial monitor*. Pengamatan sinyal I2C dan konversi ke *integer* dilakukan secara manual untuk

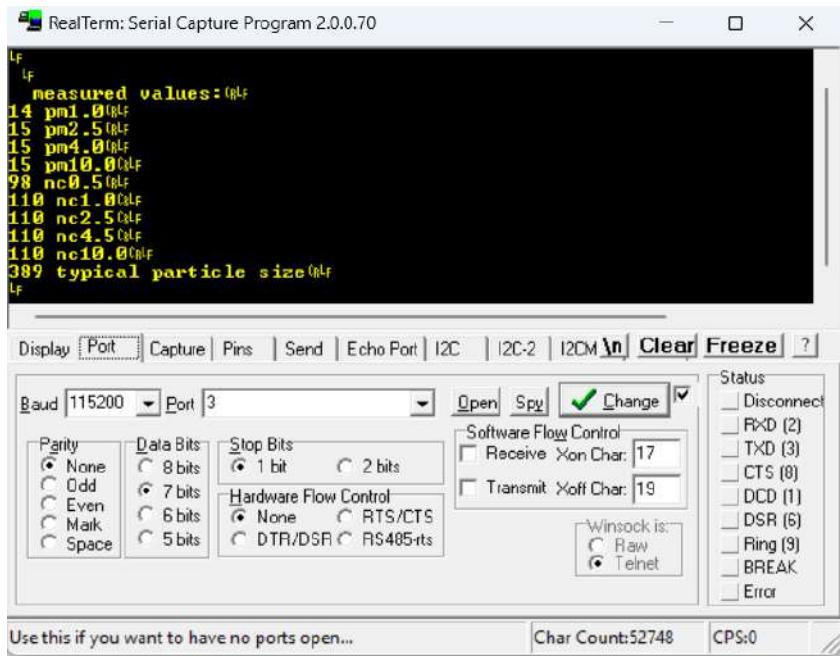
membuktikan bahwa nilai yang terbentuk pada sinyal I2C sama dengan nilai yang ditampilkan *serial monitor*. Tipe data yang dipakai adalah *unsigned 16-bit integer*. *Unsigned* berarti tidak terdapat nilai negatif pada format data ini. Dengan begitu, konversi rentetan nilai bit menjadi *integer* hanya seperti mengubah bilangan basis biner menjadi basis desimal. Contoh untuk konversi parameter nilai PM1 sebagai berikut. Berdasarkan hasil pengamatan, nilai PM1 dan *checksum*-nya terletak pada *byte* pertama hingga ketika, yaitu 00000000 000011100 100111100. Nilai PM1 sebenarnya terletak pada *byte* pertama dan kedua tanpa bit ke-9 yang merupakan bit ACK. Jadi nilai PM1 dalam biner adalah 00000000 00001110. Bilangan basis biner tersebut diubah ke basis desimal menjadi 14. Dengan demikian konsentrasi massa PM1 pada pengukuran ini adalah  $14 \mu\text{g}/\text{m}^3$ . Hasil pengamatan dan konversinya menjadi nilai *integer* untuk parameter nilai lain ditunjukkan pada Tabel 4.3.

Tabel 4.3. Nilai Pengukuran dalam Bit dan Konversinya Menjadi *Integer*

Jenis	Data[1]	Data[2]	<i>Integer</i>
MC PM1	00000000	00001110	14
MC PM2.5	00000000	00001111	15
MC PM4	00000000	00001111	15
MC PM10	00000000	00001111	15
NC PM0.5	00000000	01100010	98
NC PM1	00000000	01101110	110
NC PM2.5	00000000	01101110	110
NC PM4	00000000	01101110	110
NC PM10	00000000	01101110	110
<i>Typical particle size</i>	00000001	10000101	389

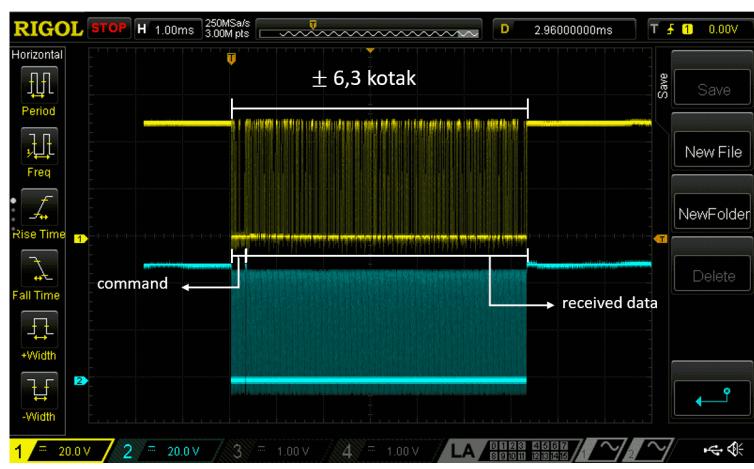
Nilai di dalam tabel tersebut bersih tanpa bit ACK dan *checksum*. Dengan menggunakan format *integer* dapat dilihat bahwa memori yang dibutuhkan lebih kecil untuk menyimpan parameter nilai pengukuran. Namun, perlu diperhatikan pada jenis *typical particle size*. Nilai yang dikirim adalah dalam orde ratusan karena berdasarkan *datasheet* nilai tersebut berada dalam satuan nm. Sementara itu, pada format data *float*, nilai *typical particle size* berada dalam satuan  $\mu\text{m}$ . Untuk itu, agar tidak terjadi kekeliruan, nilai dalam format *integer* ini dapat dibagi 1000 atau ditambahkan "0," saat akan diolah atau ditampilkan.

Seluruh parameter nilai dalam format data *integer* ditunjukkan pada Gambar 4.7. Tabel 4.3 dan gambar tersebut menunjukkan bahwa sinyal yang terbentuk pada jalur SDA komunikasi I2C sama dengan apa yang ditampilkan *serial monitor* melalui komunikasi UART. Dengan demikian, perintah *Read Measured Values* untuk format data *integer* sudah berjalan dengan benar.

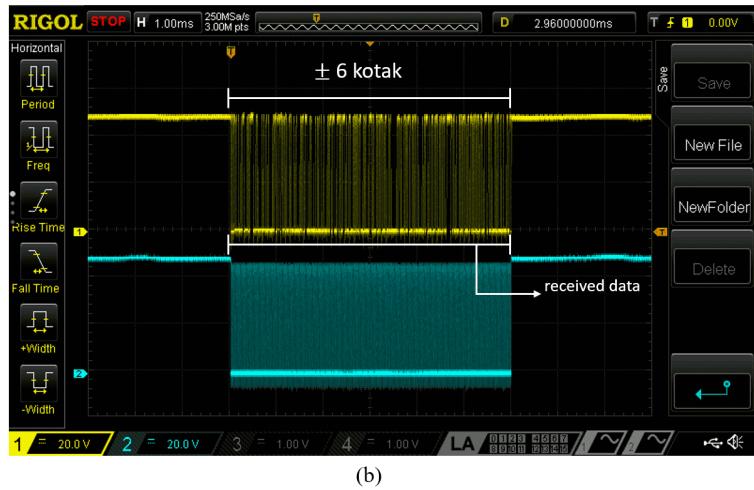


Gambar 4.7. Nilai pengukuran format data *integer* pada serial monitor

Perintah *Read Measured Values* adalah perintah dengan tipe transfer *Set Pointer & Read Data*. Pada bab 3 telah dijelaskan bahwa perintah dengan tipe transfer ini tidak perlu mengirim *pointer* alamatnya jika ingin membaca data secara berulang. *Pointer* alamat cukup dikirim sekali dan proses membaca dapat dilakukan berkali-kali setelahnya. Untuk membuktikannya, pada pengujian ini fungsi `read_measured_values()` dipecah menjadi dua, yaitu `read_measured_command()` dan `get_measured_data`. Fungsi `read_measured_command()` memiliki struktur yang sama persis dengan fungsi perintah *Set Pointer*. Sementara itu, fungsi `get_measured_data()` isinya adalah fungsi `read_bytes` dan memasukkan nilai pengukuran yang dibaca ke dalam *struct sps30\_measurement*. Pengujian ini dilakukan dengan menjalankan fungsi `read_measured_command()` sekali yang dilanjutkan dengan menjalankan fungsi `get_measured_data()` berkali-kali dengan jeda waktu 1 detik di dalam *loop*.



(a)



(b)

Gambar 4.8. Perbedaan sinyal I2C antara (a) sinyal *pointer* alamat perintah dan data yang dibaca dan (b) sinyal data yang dibaca saja

Hasil dari pengujian ini dapat dilihat pada kedua sinyal I2C pada Gambar 4.8. Gambar 4.8 (a) merupakan sinyal I2C ketika perintah *Read Measured Values* dijalankan dengan fungsi `read_measured_values` yang mana terjadi pengiriman *pointer* alamat perintah dan membaca data. Dengan fungsi `read_measured_command()`, sinyal yang terbentuk sama seperti Gambar 4.8 (a) bagian *command*, yaitu hanya mengirim *pointer* alamat perintahnya saja. Setelah itu, data yang dibaca menggunakan fungsi `get_measured_data()` ditunjukkan pada Gambar 4.8 (b). Pada gambar tersebut ditunjukkan bahwa sinyal I2C yang terbentuk hanya sinyal data pengukuran. Data tetap dapat terbaca meskipun *pointer* alamat tidak dikirim lagi. Hal ini juga ditunjukkan panjang sinyal yang berbeda antara kedua gambar tersebut. Gambar (a) lebih panjang daripada (b) karena ada proses pengiriman *pointer* alamat perintah *Read Measured Values*.

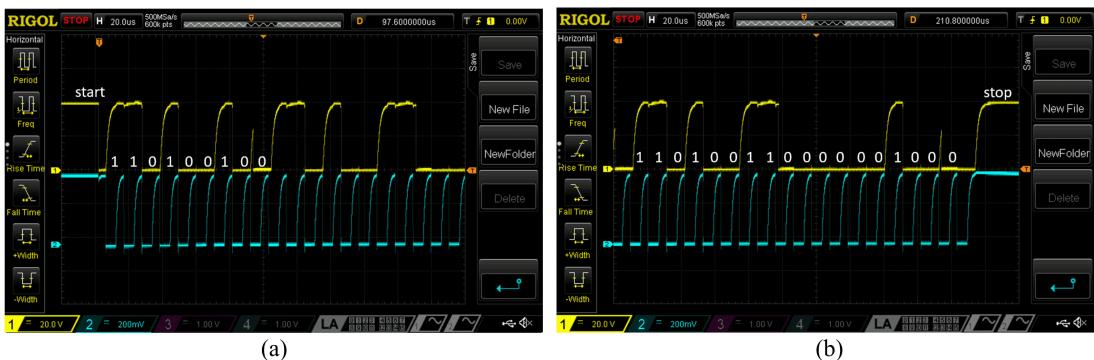
Tabel 4.4. Nilai Pengukuran dalam 10 Iterasi

$i$	MC PM1	MC PM2.5	MC PM4	MC PM10	NC PM0.5	NC PM1	NC PM2.5	NC PM4	NC PM10	TPS
1	11.2	11.93	11.99	12.02	78.85	89.14	89.4	89.44	89.47	0.48
2	11.32	12.05	12.11	12.15	79.65	90.05	90.31	90.35	90.38	0.48
3	11.36	12.1	12.16	12.2	79.97	90.41	90.67	90.72	90.74	0.48
4	11.39	12.12	12.19	12.22	80.12	90.58	90.85	90.89	90.91	0.48
5	11.37	12.11	12.17	12.21	80.02	90.47	90.73	90.78	90.8	0.48
6	11.38	12.11	12.18	12.21	80.06	90.52	90.79	90.83	90.85	0.48
7	11.42	12.16	12.22	12.26	80.34	90.83	91.1	91.14	91.17	0.48
8	11.46	12.2	12.27	12.3	80.63	91.16	91.43	91.47	91.5	0.48
9	11.47	12.21	12.28	12.31	80.69	91.23	91.5	91.55	91.57	0.48
10	11.52	12.27	12.34	12.37	81.09	91.68	91.96	92.00	92.03	0.48

Hasil data pengukuran yang didapat dengan kedua fungsi yang terpisah untuk membaca data ditunjukkan pada Tabel 4.4. Setiap iterasi memiliki jeda waktu 1 detik karena nilai pengukuran baru tersedia setiap 1 detik. Dari pengujian ini ditunjukkan bahwa dengan mengirim *pointer* alamat sekali, P-Nucleo-WB55 tetap dapat membaca nilai pengukuran berulang kali. Kemampuannya ini penting karena nanti dalam mengukur konsentrasi partikulat, dalam sekali pengukuran akan diambil beberapa data secara kontinu untuk dihitung rata-ratanya. Jadi, *pointer* alamat cukup dikirim sekali saja tiap pengukurannya.

#### 4.1.3 Device Reset

Perintah *Device Reset* dilakukan dengan menjalankan program yang berisi fungsi `device_reset()` yang telah dijelaskan pada Bab 3. Perintah ini berguna untuk *reset* ulang sensor SPS30 agar kondisinya sama seperti pertama kali dinyalakan. Sederhananya, perintah ini sama seperti *power reset* tanpa harus benar-benar mematikan catu daya sensornya. Tujuan sensor harus di-*reset* adalah agar tidak ada kesalahan dalam menjalankan perintah-perintahnya, seperti kesalahan pembacaan dan pengiriman data pengukuran. Oleh karena itu, perintah ini dapat digunakan sebagai fungsi inisialisasi sensor, yaitu fungsi yang paling pertama dijalankan oleh sensor. Ketika fungsi ini dijalankan, sinyal I2C ditunjukkan pada Gambar 4.9.



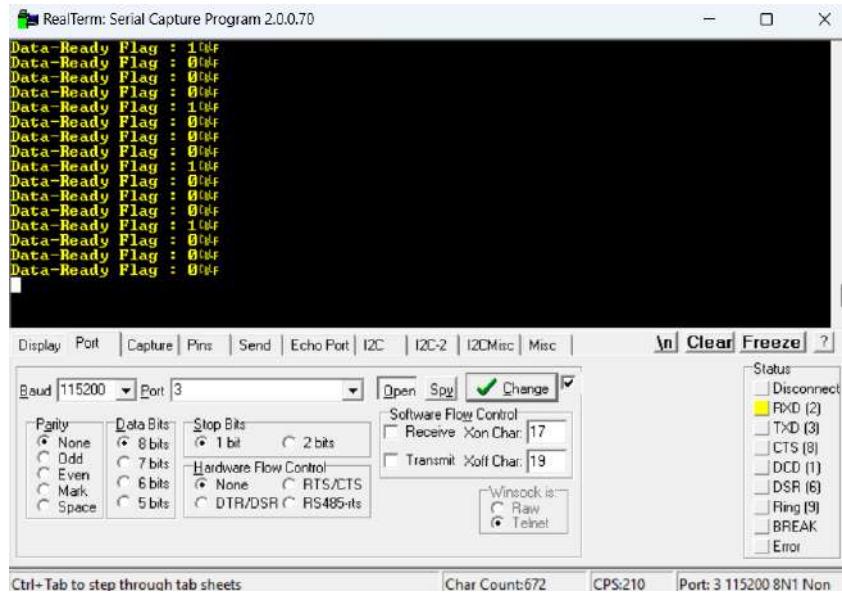
Gambar 4.9. Sinyal I2C perintah *Device Reset* pada osiloskop

Gambar 4.9 menunjukkan sinyal I2C yang sama seperti pada Gambar 3.6. Perintah ini memiliki tipe transfer *Set Pointer* sehingga sinyal yang dikirim hanya *byte* alamat sensor ditambah bit *write*, yaitu 11010010 dan 2*byte* *pointer* alamat perintah, yaitu 11010011 00000100 (0xd304). Setiap *byte* diikuti dengan bit ACK yang berarti sinyal diterima oleh sensor. Dengan demikian, perintah *Device Reset* telah berhasil dilakukan. Jadi, fungsi `device_reset()` telah siap untuk digunakan.

#### 4.1.4 Read Data-Ready Flag

Perintah *Read Data-Ready Flag* dilakukan dengan menjalankan program yang berisi fungsi `read_data_ready_flag()`. Perintah ini digunakan untuk mencari

tahu apakah nilai pengukuran konsentrasi partikulat baru sudah tersedia atau belum. Nilai yang akan dibaca adalah 2 *byte* data. *Byte* pertama hanyalah *dummy byte* yang bernilai 0x00, sedangkan *byte* kedua merupakan nilai pembacaannya. Nilai pembacaan dapat berupa 0 atau 1. 0 berarti belum ada nilai pengukuran yang baru, sedangkan 1 berarti nilai pengukuran baru sudah tersedia. Fungsi `read_data_ready_flag()` memiliki masukan berupa *pointer* dari variabel dengan tipe data `uint8_t`. Untuk menerima nilai dari komunikasi I2C, diperlukan *buffer* dengan ukuran 3 *byte*. Variabel yang digunakan untuk menyimpannya adalah `flag`.



Gambar 4.10. Pembacaan *data-ready flag* setiap 250 ms ketika nilai pengukuran konsentrasi partikulat dibaca atau diambil setiap 1 detik

Gambar 4.10 menunjukkan hasil pengujian yang dilakukan terhadap perintah `Read Data-Ready Flag`. Pengujian tersebut dilakukan pada mode *measurement*. Nilai dalam monitor serial tersebut ditampilkan setiap 250 ms. Dalam setiap detiknya, dijalankan perintah `Read Measured Values` untuk melihat perubahan hasil pembacaan *data-ready flag* ini. Dapat dilihat bahwa nilai 1 ditampilkan setiap 4 baris yang berarti setiap detik. Ketika dilakukan perintah `Read Measured Values`, nilai *data-ready flag* menjadi 0 karena nilai pengukuran sudah diambil atau dibaca. Selain itu, ketika sensor dalam mode *measurement* tetapi tidak data pengukuran konsentrasi partikulat tidak dibaca menggunakan perintah `Read Measured Values`, maka nilai *data-ready flag* akan bernilai 1 terus. Di sisi lain, ketika *data-ready flag* dibaca dalam mode *idle*, nilai yang terbaca akan selamanya 0. Dengan demikian pembacaan *data-ready flag* telah berhasil dilakukan. Jadi, fungsi `read_data_ready_flag()` sudah siap untuk digunakan.

#### 4.1.5 Stop Measurement

Perintah *Stop Measurement* dilakukan dengan menjalankan program yang berisi fungsi *stop\_measurement()*. Perintah ini digunakan untuk membalikkan kondisi sensor dari mode *measurement* menjadi mode *idle* kembali. Untuk itu, perintah ini dilakukan setelah sensor masuk ke mode *measurement* dengan perintah *Start Measurement*. Pengujian untuk perintah ini dilakukan dengan melihat perbedaan arus yang mengalir antara mode *measurement* dan mode *idle*. Gambar 4.3 (b) menunjukkan kuat arus yang mengalir ketika sensor masuk ke mode *measurement*, yaitu sebesar 54,7 mA. Gambar 4.11 menunjukkan kuat arus yang mengalir setelah menjalankan perintah *Stop Measurement*, yaitu 0,35 mA. Dengan begitu, sensor telah berhasil kembali ke mode *idle* dengan perintah ini. Jadi, fungsi *stop\_measurement ()* telah siap untuk digunakan.



Gambar 4.11. Kuat arus yang mengalir pada mode *idle*

#### 4.1.6 Sleep

Perintah *Sleep* dilakukan dengan menjalankan program dengan fungsi *sleep ()* di dalamnya. Perintah ini digunakan untuk sensor masuk ke mode *sleep*. Tujuannya adalah untuk menghemat daya yang digunakan. Pengujian perintah ini dilakukan dengan melihat perbedaan kuat arus yang mengalir antara mode *idle* dan mode *sleep*. Gambar 4.11 menunjukkan kuat arus yang mengalir ketika sensor berada pada mode *idle*, yaitu sebesar 0,35 mA. Gambar 4.12 menunjukkan kuat arus yang mengalir setelah menjalankan perintah *Sleep*, yaitu 57,1  $\mu$ A. Dengan demikian, sensor telah berhasil masuk ke mode *sleep* dengan fungsi *sleep ()*. Jadi, fungsi tersebut telah siap untuk digunakan.



Gambar 4.12. Kuat arus yang mengalir pada mode *sleep*

#### 4.1.7 Wake-up

Perintah *Wake-up* dilakukan dengan menjalankan fungsi `wake_up()` seperti yang telah dijelaskan pada Bab 3. Perintah ini digunakan untuk membalikkan kondisi sensor dari mode *sleep* menjadi mode *idle* kembali. Untuk itu, perintah ini dilakukan setelah sensor masuk ke mode *sleep* dengan perintah *Sleep*. Pengujian ini dilakukan dengan melihat arus yang mengalir antara kedua mode tersebut. Pada pengujian sebelumnya telah ditunjukkan bahwa kuat arus yang mengalir pada mode *sleep* adalah  $57,1 \mu\text{A}$ . Gambar 4.13 menunjukkan kuat arus yang mengalir pada mode *idle* setelah menjalankan perintah *Wake-up* pada mode *sleep*, yaitu  $0,35 \text{ mA}$ . Kuat arus tersebut sama dengan arus yang mengalir setelah menjalankan perintah *Stop Measurement*.

Seperti yang telah dijelaskan pada Bab 3, khusus perintah *Wake-up*, meskipun perintah ini masuk dalam tipe transfer *Set Pointer*, terdapat sedikit perbedaan untuk menjalankannya. Periferal I2C tidak aktif ketika sensor dalam mode *sleep*. Untuk itu diperlukan cara untuk mengaktifkannya kembali, yaitu dapat dengan menjalankan perintah I2C *transmit* dua kali. Sinyal I2C yang terbentuk saat menjalankan fungsi `wake_up()` ditunjukkan pada Gambar 4.14. Gambar tersebut menunjukkan bahwa mikrokontroler berusaha mengirim sinyal dua kali. Pengiriman pertama diabaikan yang ditunjukkan dengan bit ke-9 setelah alamat sensor dan bit *write* adalah bit NACK atau 1. Hal tersebut terjadi karena periferal I2C sensor tidak aktif sehingga sensor tidak merespons. Akan tetapi, sinyal tersebut menyebabkan periferal sensor kembali aktif sehingga pengiriman kedua dapat direspon oleh sensor. Sensor dapat menerima perintah dari mikrokontroler. Dengan demikian, sensor telah berhasil *wake-up* dengan fungsi `wake_up()`. Jadi, fungsi tersebut telah siap untuk digunakan.



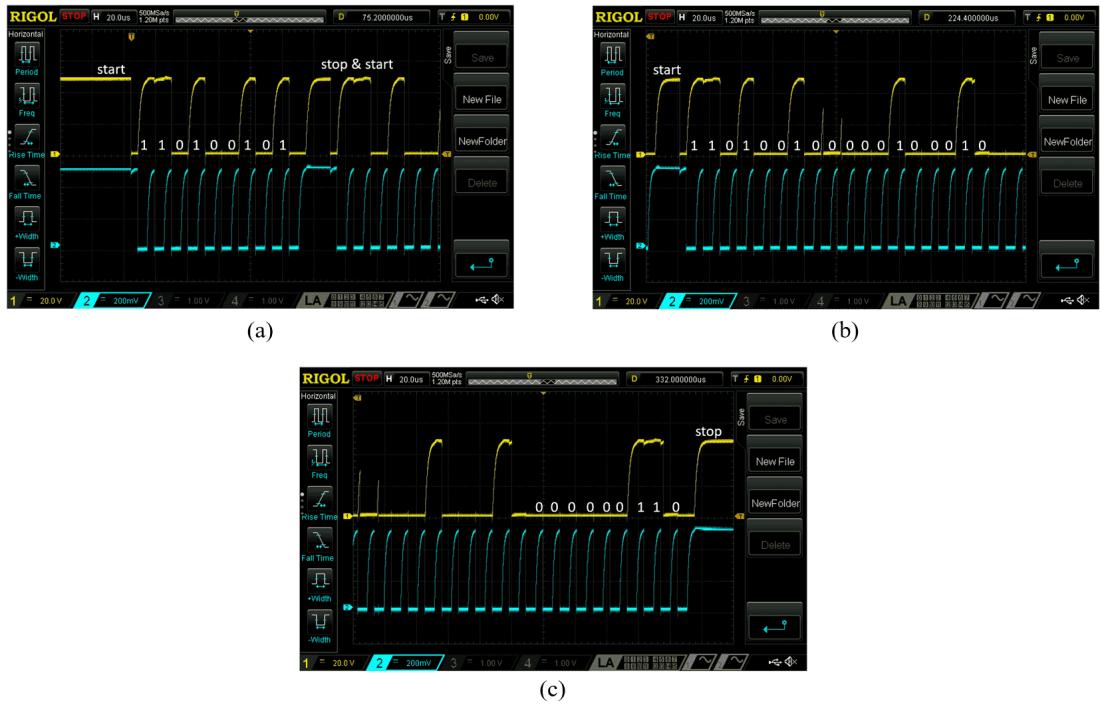
Gambar 4.13. Kuat arus yang mengalir pada mode *wake-up*

#### 4.1.8 *Start Fan Cleaning*

Perintah *Start Fan Cleaning* dilakukan dengan menjalankan perintah program yang berisi fungsi `start_fan_clean()`. Perintah ini digunakan untuk melakukan pembersihan kipas secara manual. Pembersihan kipas secara otomatis hanya dapat dilakukan ketika sensor berada pada mode *measurement* selama interval pembersihan kipas otomatis tersebut. Jika program yang dijalankan untuk mengukur konsentrasi partikulat tidak membuat sensor berada dalam mode *measurement* selama waktu tersebut, pembersihan kipas dapat dijalankan dengan perintah ini. Untuk menjalankan perintah ini, sensor harus dalam mode *measurement* terlebih dahulu. Maka dari itu fungsi `start_fan_clean()` dijalankan setelah fungsi `start_measurement()`. Dengan perintah ini, pembersihan akan dilakukan selama 10 detik. Pengujian dilakukan dengan melihat arus yang mengalir di dalam rentang waktu tersebut. Gambar 4.15 menunjukkan kuat arus yang mengalir selama pembersihan dilakukan. Gambar tersebut menunjukkan arus yang mengalir lebih kecil dari arus yang mengalir pada mode *measurement* yang ditunjukkan pada Gambar 4.3. Selama pembersihan dilakukan, sensor mengeluarkan bunyi mendesing karena kipas diputar dalam kecepatan maksimal. Dengan ini, sensor telah berhasil menjalankan pembersihan kipas secara manual. Jadi, fungsi `start_fan_clean()` telah siap untuk digunakan.

#### 4.1.9 *Read/Write Auto Cleaning Interval*

*Auto Cleaning Interval* adalah interval pembersihan kipas secara otomatis. Pembersihan kipas dapat dilakukan secara otomatis jika sensor berada dalam mode *measurement*. Ketika aktif, kipas akan diputar dengan kecepatan penuh selama 10 detik. Me-



Gambar 4.14. Sinyal I2C perintah Wake-up pada osiloskop

nurut *datasheet*, interval pembersihan kipas dari bawaannya adalah setiap 1 minggu atau 604.800 detik dengan toleransi  $\pm 3\%$ . Perintah *Read Auto Cleaning Interval* berguna untuk membaca interval pembersihan kipas yang disimpan di dalam *non-volatile memory*. Perintah ini dilakukan dengan menjalankan program yang di dalamnya berisi fungsi `get_cleaning_interval()`. Nilai yang dibaca merupakan nilai dengan format *unsigned 32-bit integer*. Untuk itu, fungsi ini memiliki masukan berupa *pointer* dari variabel `interval` dengan tipe data `uint32_t`. *Buffer* yang digunakan untuk menerima data dari komunikasi I2C berukuran 6 *byte*. Nilai yang dibaca ditampilkan pada monitor serial yang ditunjukkan pada Gambar 4.16. Nilai yang terbaca adalah 604.800 detik sehingga sudah sesuai dengan interval bawaannya. Dengan demikian, fungsi `get_cleaning_interval()` sudah siap untuk digunakan.

Interval pembersihan kipas otomatis dapat diubah-ubah secara manual, mulai dari 0 ms hingga nilai maksimum yang dapat disimpan pada tipe data `uint32_t`. Untuk mengubahnya, dapat dilakukan perintah *Write Auto Cleaning Interval*. *Pointer* alamat perintah untuk keduanya adalah sama. Yang membedakannya adalah kali ini akan dilakukan komunikasi untuk *write*. Pengujian ini dilakukan dengan menjalankan program yang berisi fungsi `set_cleaning_interval()`. Fungsi tersebut memiliki masukan berupa variabel dengan tipe data `uint32_t`. Interval baru dimasukkan melalui variabel tersebut yang kemudian akan dikirim melalui komunikasi I2C beserta *checksum*-nya. Pada pengujian ini, interval pembersihan kipas akan diubah menjadi setiap 1 jam atau 3.600 detik. Menurut *datasheet*, interval akan berubah setelah dilakukan *device reset* atau *power reset*. Untuk itu, setelah mengirim interval baru dijalankan fungsi `device_reset()`.



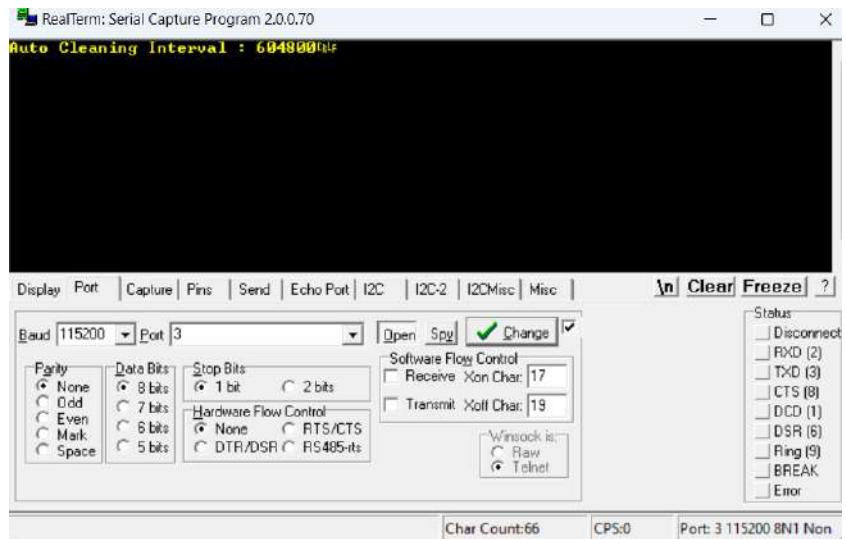
Gambar 4.15. Kuat arus yang mengalir pada saat dilakukan pembersihan kipas

Kemudian, untuk memeriksa apakah interval sudah berubah, dijalankan kembali fungsi `get_cleaning_interval()` untuk membacanya. Hasil pembacaan ditampilkan pada monitor serial yang ditunjukkan pada Gambar 4.17. Dapat dilihat bahwa interval sudah berubah menjadi 3600. Dengan demikian, mengubah interval pembersihan kipas sudah berhasil dilakukan dengan perintah *Write Cleaning Interval*. Jadi, fungsi `set_cleaning_interval()` sudah siap untuk digunakan.

#### 4.1.10 *Read Product Type*

Perintah *Read Product Type* dilakukan dengan menjalankan program yang berisi fungsi `product_type()`. Perintah ini berguna untuk membaca tipe produk dari sensor SPS30. Nilai yang dibaca merupakan nilai dalam format *string* atau *array char* yang berjumlah 8 karakter (8 *byte*). Untuk itu, fungsi dari perintah ini memiliki masukan berupa *pointer* dari variabel dengan tipe data *char* yang berfungsi untuk menyimpan nilai yang dibaca dari sensor. Dengan diketahui jumlah karakter yang akan diterima, ukuran dari *buffer* yang harus disiapkan dapat diketahui. Karena setiap 2 *byte* selalu diikuti dengan *checksum*, maka ukurannya adalah 12.

Variabel yang digunakan untuk menyimpan nilai tipe produk adalah `product [9]`. Elemen terakhir dari variabel tersebut atau `product [8]` perlu diisi dengan *terminating null-character* atau "`\0`". Berdasarkan *datasheet*, perintah ini akan selalu menerima nilai "00080000". Jadi, pengujian ini dilakukan dengan melihat bagaimana nilai yang dibaca dan dikirim ke *serial monitor*. Gambar 4.18 menunjukkan bahwa nilai yang diterima dan disimpan pada variabel `serial` adalah 0008000. Nilai tersebut sudah sesuai dengan apa yang tercantum pada *datasheet*. Dengan demikian, fungsi `product_type()` telah



Gambar 4.16. Interval pembersihan kipas otomatis secara *default*



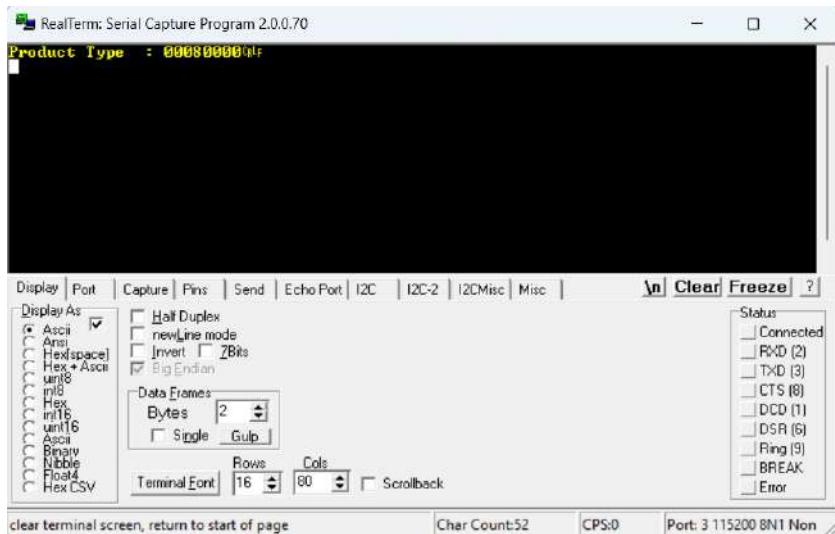
Gambar 4.17. Interval pembersihan kipas otomatis yang baru

siap untuk digunakan.

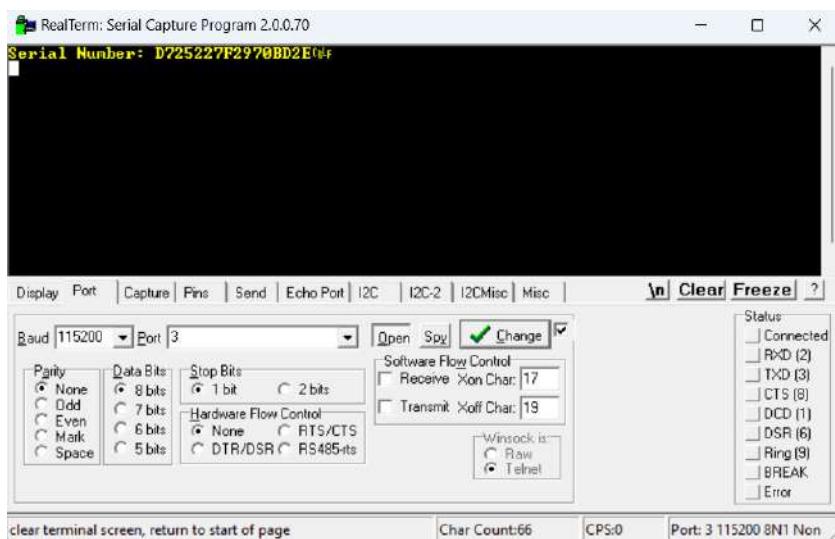
#### 4.1.11 *Read Serial Number*

Perintah *Read Setial Number* dijalankan dengan program yang di dalamnya berisi fungsi `serial_number()`. Fungsi ini mirip seperti fungsi `product_type()`. Yang membedakannya adalah jumlah data yang diterimanya. Nilai yang dibaca berbentuk *string* atau *array char* yang berjumlah 16 karakter (16 *byte*). Jadi, *buffer* yang disiapkan untuk menerima pembacaan melalui komunikasi I2C berukuran 24 *byte* karena terdapat *checksum* juga setiap 2 *byte* data yang dikirim sensor.

Variabel yang digunakan untuk menyimpan nilai nomor seri adalah `serial[17]`. Sama seperti sebelumnya, elemen terakhir dari variabel tersebut atau `serial[16]` perlu diisi dengan *terminating null-character* atau "`\0`" sebagai tanda berakhirnya *string*.



Gambar 4.18. Keluaran perintah *Read Product Type* pada monitor



Gambar 4.19. Keluaran perintah *Read Serial Number* pada monitor

Nilai yang terbaca ditunjukkan pada Gambar 4.19, yaitu "D725227F2970BD2E". Nilai tersebut sudah sesuai dengan nomor seri yang tertera pada sensor SPS30 yang digunakan (Gambar 4.20). Dengan demikian, fungsi `serial_number()` telah siap untuk digunakan.

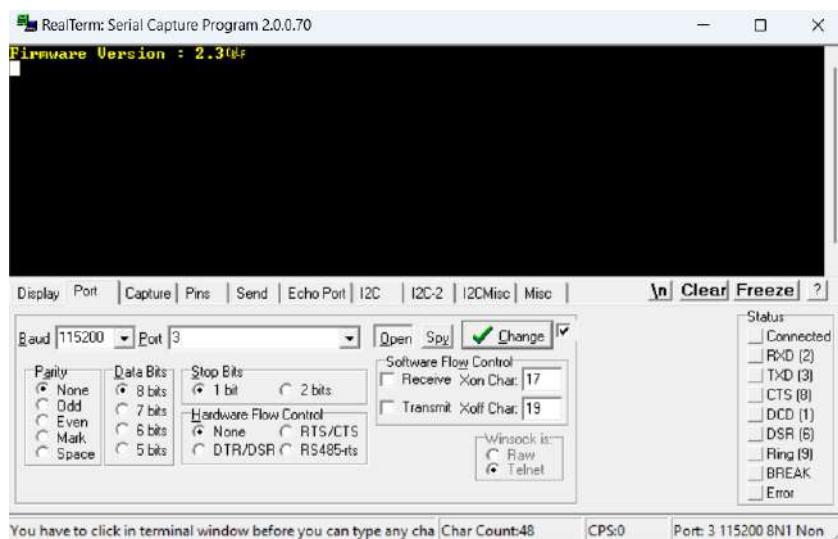
#### 4.1.12 *Read Version*

Perintah *Read Version* dilakukan dengan menjalankan program *firmware* yang berisi fungsi `read_firmware_ver()`. Perintah ini berguna untuk membaca versi *firmware* yang tertanam pada sensor SPS30. Nilai yang dibaca merupakan nilai dengan format *unsigned 16-bit integer* berjumlah dua *byte*. *Byte* pertama adalah nilai versi *major*, sedangkan *byte* kedua adalah nilai versi *minor*. Untuk itu, fungsi dari perintah ini memiliki masukan berupa *pointer* dari dua variabel dengan tipe data `uint8_t`, yaitu `major` dan



Gambar 4.20. Nomor seri sensor SPS30

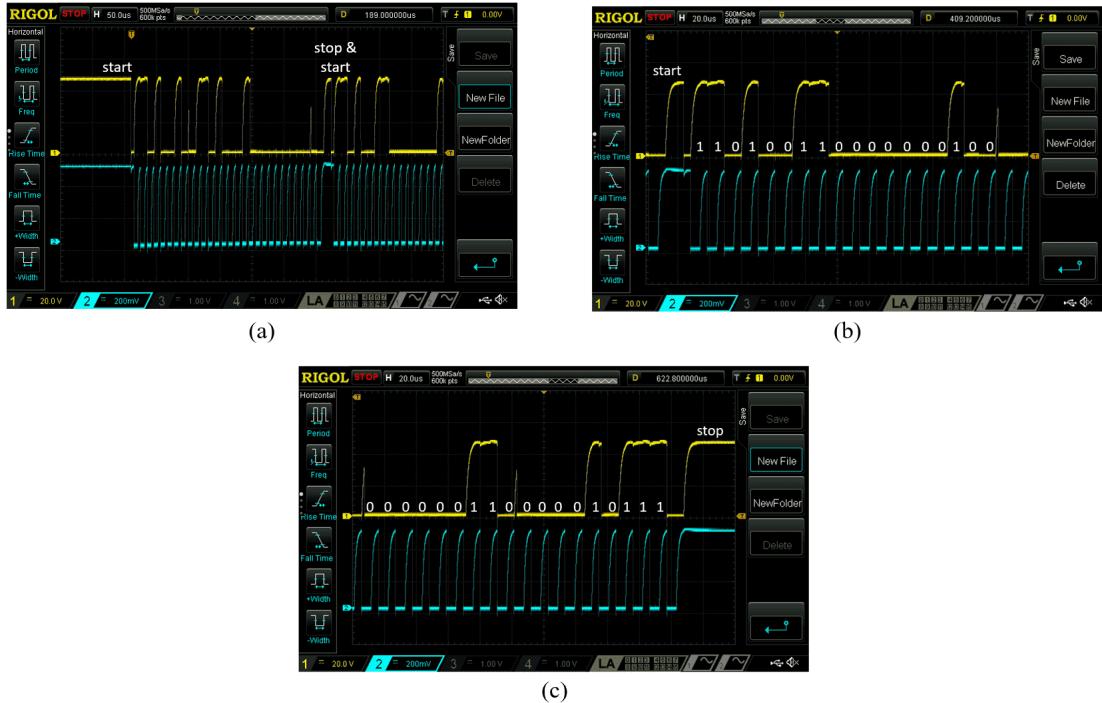
minor. Masing-masing variabel berfungsi untuk menyimpan nilai versi *firmware* yang dibaca dari sensor. Dengan demikian dapat diketahui bahwa *buffer* yang harus disiapkan untuk menerima nilai dalam komunikasi I2C adalah 3 byte karena dua byte data selalu diikuti dengan nilai *checksum*-nya.



Gambar 4.21. Keluaran perintah *Read Version* pada monitor

Gambar 4.21 menunjukkan versi *firmware* yang digunakan sensor adalah versi 2.3. Tidak ada rujukan versi berapa yang tertanam pada sensor yang dipakai ini. Untuk memastikan bahwa memang benar versi *firmware*-nya adalah 2.3, dilihat sinyal I2C yang terbentuk saat komunikasi perintah ini berlangsung. Gambar 4.22 menunjukkan sinyal I2C yang terbentuk. Gambar tersebut menunjukkan bagian ketika sensor mengirim data setelah mikrokontroler mengirim *pointer* alamat perintah. Nilai bit yang dikirim sensor adalah 00000010 00000011 di luar bit ACK dan *checksum*. Pada *datasheet* dijelaskan bahwa byte pertama adalah versi *major*, sedangkan byte kedua adalah byte *minor*. Jika nilai bit tersebut diubah menjadi basis desimal, maka nilainya adalah 2 dan 3. Dengan

demikian nilai yang ditampilkan oleh serial monitor sudah sesuai dengan nilai yang dikirim sensor ke mikrokontroler. Jadi, fungsi `read_firmware_ver()` sudah siap untuk digunakan.



Gambar 4.22. Sinyal I2C perintah *Read Version* pada osiloskop

#### 4.1.13 *Read Device Status Register*

*Device Status Register* adalah *register* yang menyimpan informasi mengenai kondisi internal dari sensor. Saat ini, terdapat tiga buah parameter yang diperiksa oleh sensor dan hasilnya disimpan pada *register* tersebut, yaitu:

1. Bit 21 (Kecepatan putar kipas)
  - Bit bernilai 0 berarti kecepatan kipas tidak ada masalah.
  - Bit bernilai 1 berarti kecepatan kipas terlalu tinggi atau rendah.
  - Kecepatan kipas diperiksa sekali tiap detik dalam mode *measurement*.
  - Kecepatan kipas tidak diperiksa selama 3 detik pertama sensor masuk ke mode *measurement* dan saat sedang menjalankan pembersihan kipas.
  - Kondisi temperatur sekitar yang terlalu tinggi atau rendah dapat menyebabkan kipas membutuhkan waktu lebih lama untuk mencapai kecepatan targetnya yang mana dapat menyebabkan bit bernilai 1. Ketika sudah mencapai kecepatan targetnya, nilai bit akan otomatis menjadi 0.
  - Jika bit bernilai 1 secara konstan, berarti terdapat masalah pada catu daya atau kipas sudah tidak beroperasi dengan normal.

## 2. Bit 5 (Laser)

- Bit bernilai 0 berarti arus laser tidak ada masalah.
- Bit bernilai 1 berarti laser dinyalakan dan arusnya di luar rentang.
- Kondisi laser diperiksa sekali tiap detik dalam mode *measurement*.
- Jika arus laser di luar rentang dua kali berurutan maka bit akan bernilai 1.
- Jika arus laser kembali normal, bit akan otomatis bernilai 0.
- Perlu diperhatikan kegagalan laser dapat terjadi pada temperatur sekitar yang sangat tinggi atau modul laser telah rusak.

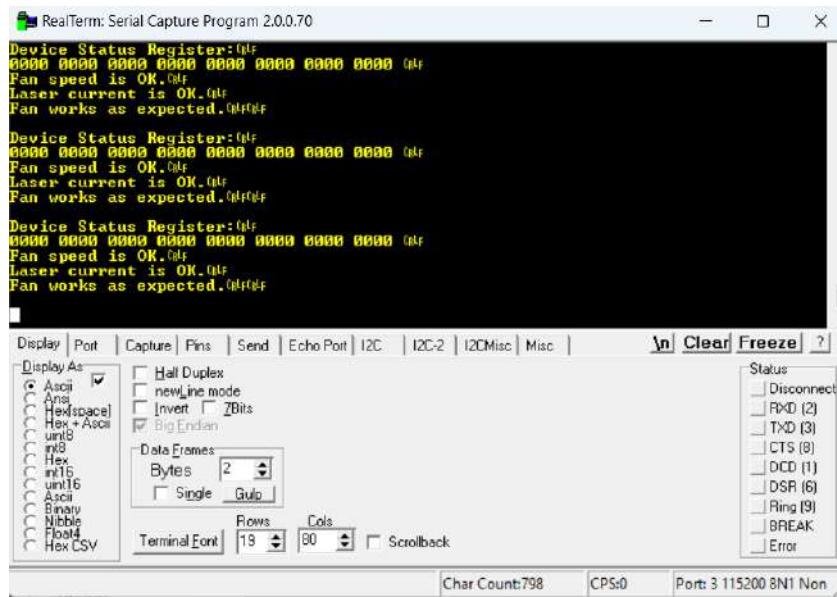
## 3. Bit 4 (Kondisi kipas)

- Bit bernilai 0 berarti kipas tidak ada masalah.
- Bit bernilai 1 berarti kipas dinyalakan, tetapi kecepatan kipas yang terukur adalah 0 rpm.
- Kondisi kipas diperiksa sekali tiap detik dalam mode *measurement*.
- Jika kecepatan 0 rpm terukur dua kali secara berurutan maka bit akan bernilai 1.
- Bit ini tidak akan otomatis kembali bernilai 0.
- Kegagalan kipas dapat disebabkan oleh kerusakan mekanik.

Ketika parameter di atas diperiksa ketika sensor dalam mode *measurement*. Maka dari itu, untuk mengujinya, fungsi `device_status` harus dijalankan beberapa detik setelah fungsi `start_measurement()` dijalankan. Nilai yang dibaca merupakan *register* berukuran 32 bit. Perlu disiapkan *buffer* berukuran 6 *byte* untuk menerima pembacaan tersebut dari komunikasi I2C karena setiap 2 *byte* akan diikuti dengan *checksum*. Nilai pembacaan *register* akan disimpan dalam variabel `status` dengan tipe data `uint32_t` yang juga *pointer* terhadap variabel tersebut menjadi masukan fungsi `device_status()`. Pengujian fungsi tersebut ditunjukkan pada Gambar 4.23. Gambar tersebut menunjukkan bahwa ketiga bit memiliki nilai 0. Hal tersebut menunjukkan ketiga parameter berada dalam kondisi normal. Dengan ini, fungsi `device_status()` sudah siap untuk digunakan.

### 4.1.14 *Clear Device Status Register*

Perintah *Cleaar Device Status Register* dilakukan dengan menjalankan program dengan perintah yang di dalamnya berisi fungsi `clear_device_status()`. Perintah ini digunakan untuk mengembalikan nilai pada *device status register* menjadi semula, yaitu 0. Perintah ini termasuk dalam tipe transfer *Set Pointer*. Karena pembacaan *devi-*



Gambar 4.23. Keluaran perintah *Read Device Status Register* pada monitor dan keterangannya

*ce status register* menggunakan perintah *Read Device Status Register* selalu bernilai 0, pengujian ini dilakukan dengan melihat sinyal I2C yang dikirim pada osiloskop. Gambar 4.24 menunjukkan bahwa sinyal yang dikirim sudah sesuai dengan alamat sensor dan alamat *pointer* perintah ini, yaitu 0xD210. *Byte* pertama diisi dengan alamat sensor yang diikuti dengan bit *write* (0) dan bit ACK (0). *Byte* kedua dan ketiga dapat dilihat adalah "11010010 00010000" yang jika diubah menjadi basis heksadesimal menjadi 0xD210. Kedua *byte* tersebut juga diikuti dengan bit ACK. Demagn demikian, fungsi *clear\_device\_status()* sudah siap digunakan untuk menjalankan peritnah *Clear Device Status Register*.



Gambar 4.24. Sinyal I2C perintah *Clear Device Status Register* pada osiloskop

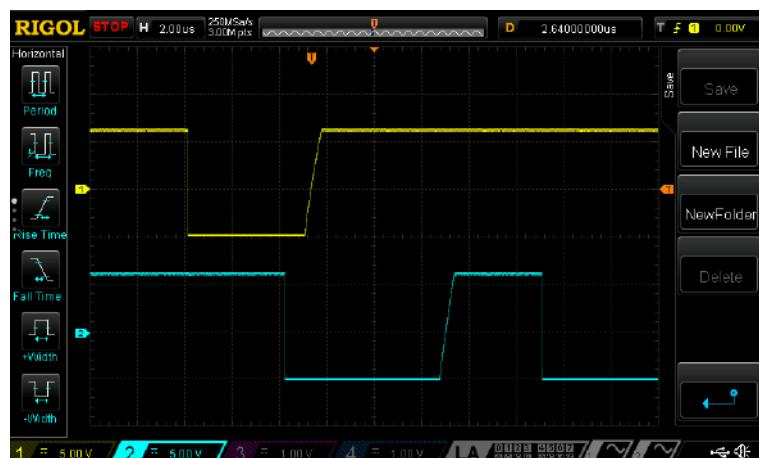
## 4.2 Pengujian Komunikasi I2C antara Sensor SPS30 dan P-Nucleo-WB55

Telah dijelaskan sebelumnya bahwa sensor SPS30 dan *development board* P-Nucleo-WB55 saling berkomunikasi menggunakan protokol komunikasi I2C. Pengujian ini bertujuan untuk mengamati dan memahami lebih detail bagaimana komunikasi I2C yang terjadi antara keduanya. Sinyal I2C dilihat lebih dalam menggunakan osiloskop untuk melihat bagaimana kondisi-kondisi yang terjadi saat komunikasi berlangsung, apakah sudah sesuai dengan teorinya atau belum. Pengujian ini dilakukan menggunakan fungsi-fungsi pada program yang sudah dibuat dan diuji pada bagian sebelumnya. Pada bagian ini, fokusnya adalah ke komunikasi I2C itu sendiri, bukan bagaimana fungsi tersebut menjalankan perintah dari mikrokontroler ke sensor.

Di setiap fungsi untuk menjalankan perintah, program untuk komunikasi I2C menggunakan dua fungsi, yaitu `HAL_I2C_Master_Transmit()` untuk mengirim data ke *slave* dan `HAL_I2C_Master_Receive()`. Kedua fungsi tersebut sudah memfasilitasi bagaimana sinyal yang akan terbentuk pada jalur SDA dan SCL. Yang perlu dimasukkan hanya alamat sensor, *buffer data*, ukuran data, dan waktu *timeout*. Bagaimana komunikasi dimulai, dihentikan, bit ACK/NACK, hingga pembuatan sinyal *clock* sudah dilakukan oleh kedua fungsi tersebut.

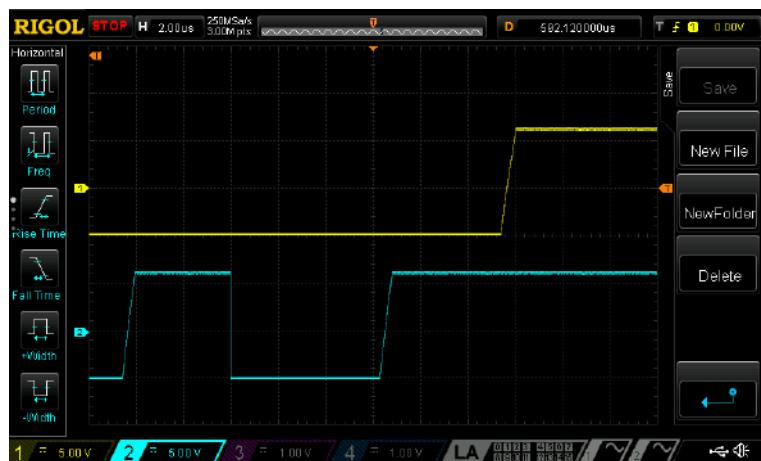
### 4.2.1 Kondisi Start dan Stop

Gambar 4.8 dan 4.24 menunjukkan bagaimana sinyal pada kedua jalur SDA dan SCL ketika dalam kondisi normal (tidak berkomunikasi) dan ketika komunikasi I2C sedang berlangsung. Saat tidak ada komunikasi, kedua jalur tersebut bernilai *high* karena dalam rangkaianya masing-masing diberi sebuah *pull-up resistor*. Namun, ketika komunikasi sedang berlangsung, kedua jalur mulai membentuk sinyal-sinyal yang menggambarkan nilai bit *high* atau *low*. Komunikasi tersebut dimulai dan diakhiri dengan kondisi *start* dan *stop*.



Gambar 4.25. Sinyal I2C untuk kondisi *start* pada osiloskop

Jika gambar sinyal pada I2C pada osiloskop diperbesar pada saat komunikasi dimulai, kondisi *start* akan terlihat. Sinyal I2C saat kondisi *start* ditunjukkan pada Gambar 4.25 yang sedang menjalankan perintah *Start Measurement*. Gambar tersebut menunjukkan bahwa kondisi *start* dimulai ketika sinyal pada jalur SDA berubah menjadi *low* ketika sinyal pada jalur SCL masih berada dalam kondisi normalnya, yaitu *high*. Berubahnya sinyal jalur SDA menjadi *low* tidak lama diikuti dengan berubahnya sinyal pada jalur SCL menjadi *low* juga. Nilai jalur SCL yang *low* ini menandakan dimulainya *clock* pertama pada komunikasi I2C. Dengan begitu, sinyal SDA juga segera membentuk sinyal bitnya dengan pada gambar ditunjukkan berubah menjadi *high* lagi karena akan membentuk nilai alamat dari sensor, yaitu "1101001" atau 0x69. Sinyal yang terbentuk ini sudah sesuai dengan teorinya yang mengatakan bahwa kondisi *start* dimulai ketika jalur SDA berubah dari *high* menjadi *low* ketika sinyal pada jalur SCL bernilai *low*.



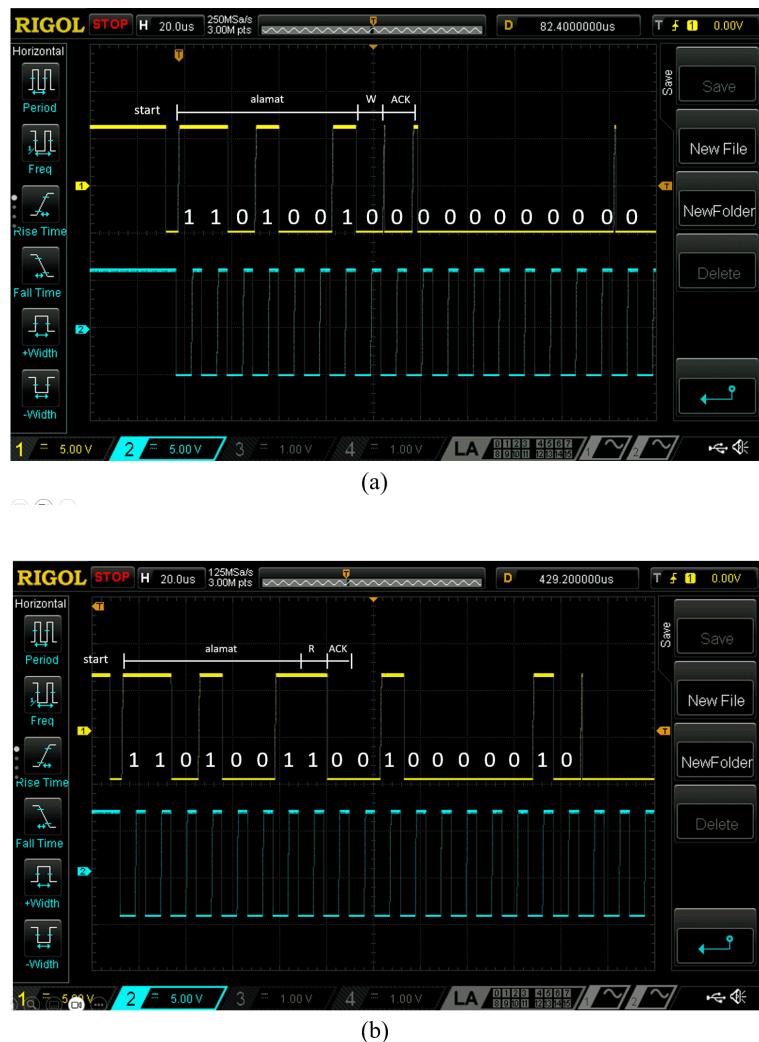
Gambar 4.26. Sinyal I2C untuk kondisi *stop* pada osiloskop

Komunikasi I2C yang dimulai dengan kondisi *start* akan diakhiri dengan kondisi *stop*. Jika sinyal komunikasi I2C pada osiloskop diperbesar ketika komunikasi akan berakhir, akan terlihat bagaimana komunikasi I2C diakhiri. Gambar 4.26 di atas menunjukkan kondisi *stop* pada perintah yang sama, yaitu *Start Measurement*. Pada saat seluruh nilai sudah dikirim dan bit ACK sudah terbentuk, sinyal pada jalur SCL kembali *low* dan kemudian menjadi *high*. Ketika jalur SCL bernilai *high*, sinyal pada jalur SDA yang tadinya *low* berubah menjadi *high*. Setelahnya, kedua jalur tetap bernilai *high* dan komunikasi selesai. Dengan demikian, kondisi *stop* yang terjadi tersebut sudah sesuai dengan teorinya, yaitu kondisi *stop* terjadi ketika sinyal pada jalur SDA berubah dari *low* menjadi *high* ketika sinyal pada jalur SCL bernilai *high*.

#### 4.2.2 Alamat Sensor

Dalam protokol komunikasi I2C, hal pertama yang dilakukan setelah komunikasi dimulai dengan kondisi *start* adalah master mengirim alamat sensor. Sensor SPS30

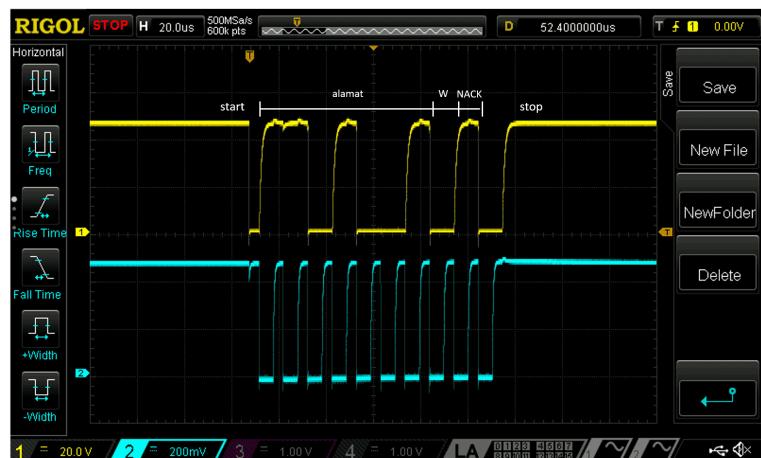
menggunakan alamat 7 bit, yaitu 0x69 atau dalam biner adalah "01101001". Kalau dilihat, alamat tersebut sebenarnya berjumlah 8 bit. Oleh karena itu, di dalam program dengan fungsi HAL\_I2C\_Master\_Transmit () atau HAL\_I2C\_Master\_Receive (), alamat tersebut harus digeser ke kiri 1 bit. Kemudian bit ke-8 akan diisi oleh bit R/W. Bit *read* atau *write* tersebut akan otomatis terisi oleh kedua fungsi tersebut. Ketika menggunakan fungsi I2C *transmit*, bit ke-8 yang terisi adalah *write* atau 0. Di sisi lain, ketika menggunakan fungsi I2C *receive*, bit ke-8 yang terisi adalah *read* atau 1. Sinyal yang terbentuk pada osiloskop untuk kedua fungsi tersebut ditunjukkan pada Gambar 4.27. Gambar 4.27 (a) menunjukkan bit *write* pada fungsi I2C *transmit*, sedangkan (b) menunjukkan bit *read* pada fungsi I2C *receive*.



Gambar 4.27. Sinyal I2C yang menunjukkan alamat sensor SPS30 ketika (a) *transmit* dan (b) *receive*

Jika alamat sensor yang dikirimkan oleh master sesuai dengan sensor yang terpasang di bus, maka sensor akan menanggapi dengan mengirim bit ACK. Pada kedua gambar tersebut terlihat bahwa sensor SPS30 mengirim bit ACK karena alamat yang dikirim oleh master sesuai dengan alamat sensor. Setelah bit ACK dikirim oleh sensor,

komunikasi dilanjutkan dengan mengirim data lain sesuai dengan apa yang diinginkan. Namun, akan berbeda jika tidak ada sensor yang terhubung ke jalur SDA dan SCL. Dengan kata lain, kedua jalur tersebut hanya terhubung dengan *pull-up resistor*. Sinyal I2C saat kondisi tersebut ditunjukkan pada Gambar 4.28. Itu adalah sinyal yang terbentuk ketika menjalankan fungsi I2C *transmit* tanpa ada sensor yang terhubung. Dapat dilihat setelah bit *write*, bit ke-9 bukan diisi oleh ACK (0), melainkan diisi oleh NACK (1). Hal tersebut terjadi karena bit ke-1 hingga ke-8, jalur SDA dipegang oleh master, yaitu mikrokontroler. Setelah bit ke-8, jalur SDA dilepas sehingga kembali ke kondisi normalnya, yaitu *high*. Karena tidak ada sensor yang terhubung, maka tidak ada yang membuat jalur SDA menjadi *low*. Maka dari itu, bit NACK terbentuk dan komunikasi terhenti dengan kondisi *stop* karena tidak ada sensor yang menanggapi. Hal ini sama dengan kondisi ketika ada sensor yang terhubung di bus tetapi alamat sensor yang dikirim master tidak sesuai dengan alamat sensor atau *slave*.



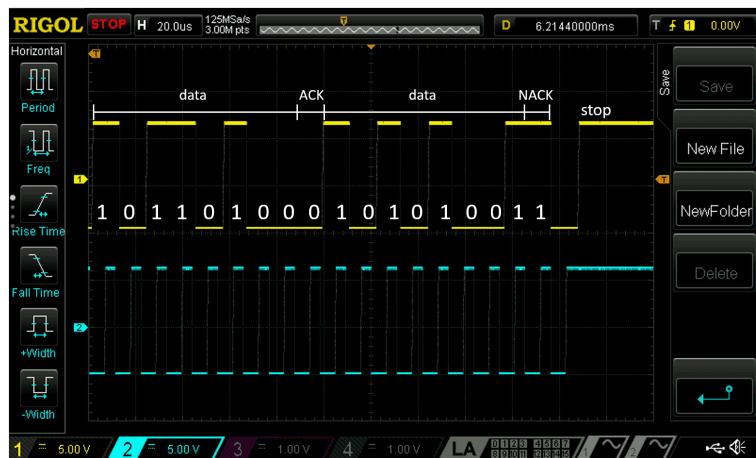
Gambar 4.28. Sinyal I2C ketika tidak ada *slave* yang terhubung pada master

#### 4.2.3 Bit Data dan ACK/NACK



Gambar 4.29. Urutan *frame* yang berisi 8 bit data dan 1 bit ACK/NACK

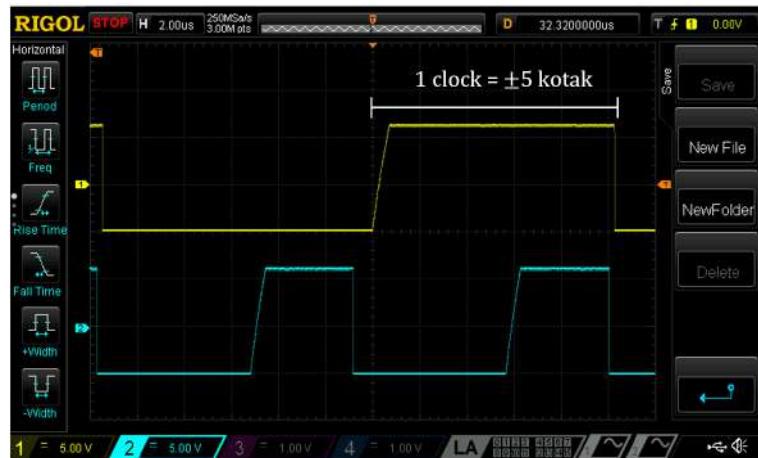
Dalam komunikasi I2C, data dikirim dalam *frame* yang terdiri dari 9 bit, yaitu 1 *byte* (8 bit) data dan 1 bit ACK/NACK. Seperti yang ditunjukkan pada Gambar 4.29, setiap 1 *byte* data selalu diikuti oleh 1 bit ACK/NACK. *Byte-byte* data dibentuk oleh pengirim, sedangkan bit ACK/NACK dikirim oleh penerima. Misalnya, gambar tersebut adalah potongan sinyal I2C yang terbentuk ketika menjalankan perintah *Start Measurement*, yang berarti mikrokontroler mengirim data ke sensor. Jadi setiap *byte* pada sinyal I2C tersebut dikirim oleh mikrokontroler yang di sini bertindak sebagai master. Sementara itu, bit ACK/NACK dikirim oleh sensor untuk menandakan bahwa data dapat diterima. Dalam implementasinya dalam program menggunakan fungsi HAL\_I2C\_Master\_Transmit () dan HAL\_I2C\_Master\_Receive (), parameter *buffer* data memiliki tipe data uint8\_t disebabkan data dikirim setiap 1 *byte*. Sementara itu, bit ACK/NACK tidak perlu dihiraukan ketika pengembangan programnya karena sudah dijalankan sendiri oleh kedua fungsi tersebut.



Gambar 4.30. Bit NACK yang terbentuk pada komunikasi receive data

Pada gambar 4.29, bit ke-9 diisi oleh bit ACK karena memang umumnya bit ACK akan lebih banyak dijumpai pada komunikasi I2C. Hampir setiap *byte* akan selalu diikuti oleh bit ACK. Sementara itu, bit NACK hanya akan dijumpai pada kondisi-kondisi tertentu. Kondisi pertama adalah saat tidak ada *slave* yang terhubung atau alamat sensor yang dikirim tidak sesuai. Kondisi tersebut ditunjukkan pada Gambar 4.28. Kondisi kedua adalah saat tidak perangkat *slave* tidak mengenali perintah yang dikirim oleh master. Terakhir, kondisi ketiga adalah di akhir komunikasi ketika master bertindak sebagai penerima data. Kondisi ketiga ini ditunjukkan pada Gambar 4.30. Sinyal pada gambar tersebut terjadi ketika menjalankan fungsi I2C *receive*. Ketika seluruh data sudah diterima, master mengirim bit NACK sebagai tanda master sudah tidak ingin menerima data lagi. Setelah bit NACK, komunikasi I2C akan segera diakhiri dengan dikirimkannya kondisi *stop*. Namun, bukan berarti untuk mengakhiri komunikasi harus selalu dengan bit NACK. Bit ACK juga dapat menjadi bit terakhir sebelum komunikasi berakhir, yaitu

pada saat komunikasi I2C *transmit* dari ke master *slave*.



Gambar 4.31. Satu bit sinyal pada jalur SDA dan SCL

Gambar 4.31 di atas memperlihatkan bagaimana 1 bit data pada komunikasi I2C. Dalam gambar tersebut dapat dilihat bahwa pergantian antara satu bit dengan bit lainnya terjadi ketika sinyal pada jalur SCL bernilai *low*. Jadi, sinyal *low* menandakan bit saat ini pada jalur SDA adalah nilai bit yang baru. Sebelumnya, nilai pada jalur SDA adalah *low*, kemudian ketika jalur SCL bernilai *low*, nilai pada jalur SDA mengalami perubahan menjadi *high*. Selanjutnya nilai pada jalur SDA berubah kembali menjadi *low* saat nilai pada jalur SCL bernilai *low*. Dari gambar tersebut juga dapat dihitung waktu yang dibutuhkan untuk membentuk 1 bit data. Jika dilihat pada bagian atas, terdapat keterangan nilai  $2,00 \mu\text{s}$  yang berarti 1 kotak pada osiloskop tersebut menggambarkan waktu tersebut. Lalu jika dihitung, 1 bit data atau 1 *clock* membutuhkan  $\pm 5$  kotak. Sehingga, 1 *clock* data membutuhkan waktu kurang lebih sebesar  $5 * 2\mu\text{s} = 10\mu\text{s}$ . Perlu diingat bahwa frekuensi yang digunakan untuk komunikasi antara mikrokontroler dengan sensor ini adalah *Standard Fast* (100 kHz). Jika dihitung dari kecepatan tersebut maka akan diketahui bahwa waktu yang dibutuhkan untuk 1 *clock* adalah  $10 \mu\text{s}$  sesuai pada 4-4.

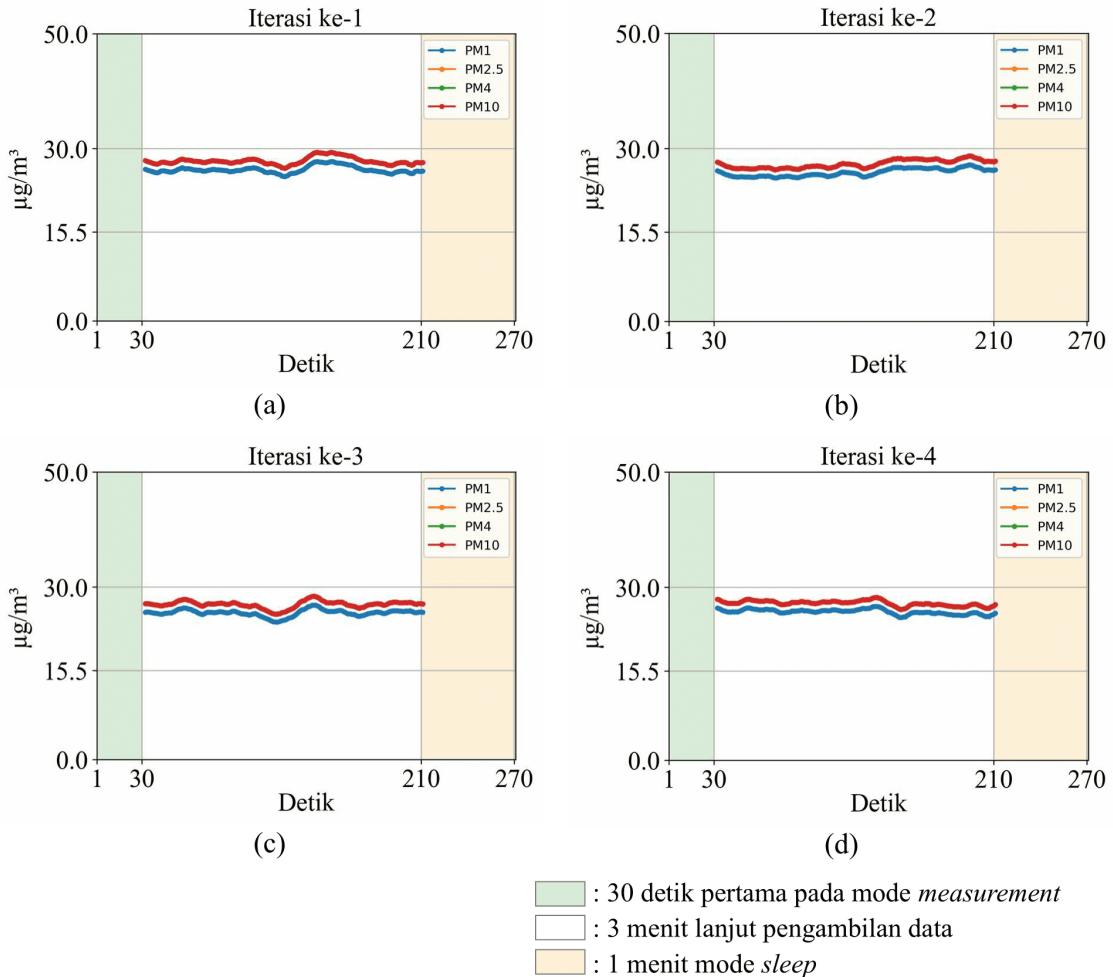
$$\frac{1 \text{ s}}{100 \text{ kHz}} = 10^{-5} \text{ s} = 10 \mu\text{s} \quad (4-4)$$

### 4.3 Pengujian Waktu *Start-up* Sensor SPS30

#### 4.3.1 Perbandingan antara Dengan dan Tanpa Waktu *Star-up*

Untuk mendapatkan data pengukuran yang stabil, sensor SPS30 membutuhkan waktu *start-up* yang cukup. Waktu *start-up* adalah waktu yang terhitung ketika sensor masuk ke mode *measurement*. Telah dijelaskan sebelumnya bahwa sensor SPS30 menggunakan kipas untuk melakukan pengukuran. Kipas mulai berputar ketika sensor masuk ke mode *measurement*. Kipas membutuhkan waktu untuk mencapai kecepatan targetnya. Jika nilai pengukuran dibaca ketika kipas belum berputar dengan kecepatan targetnya,

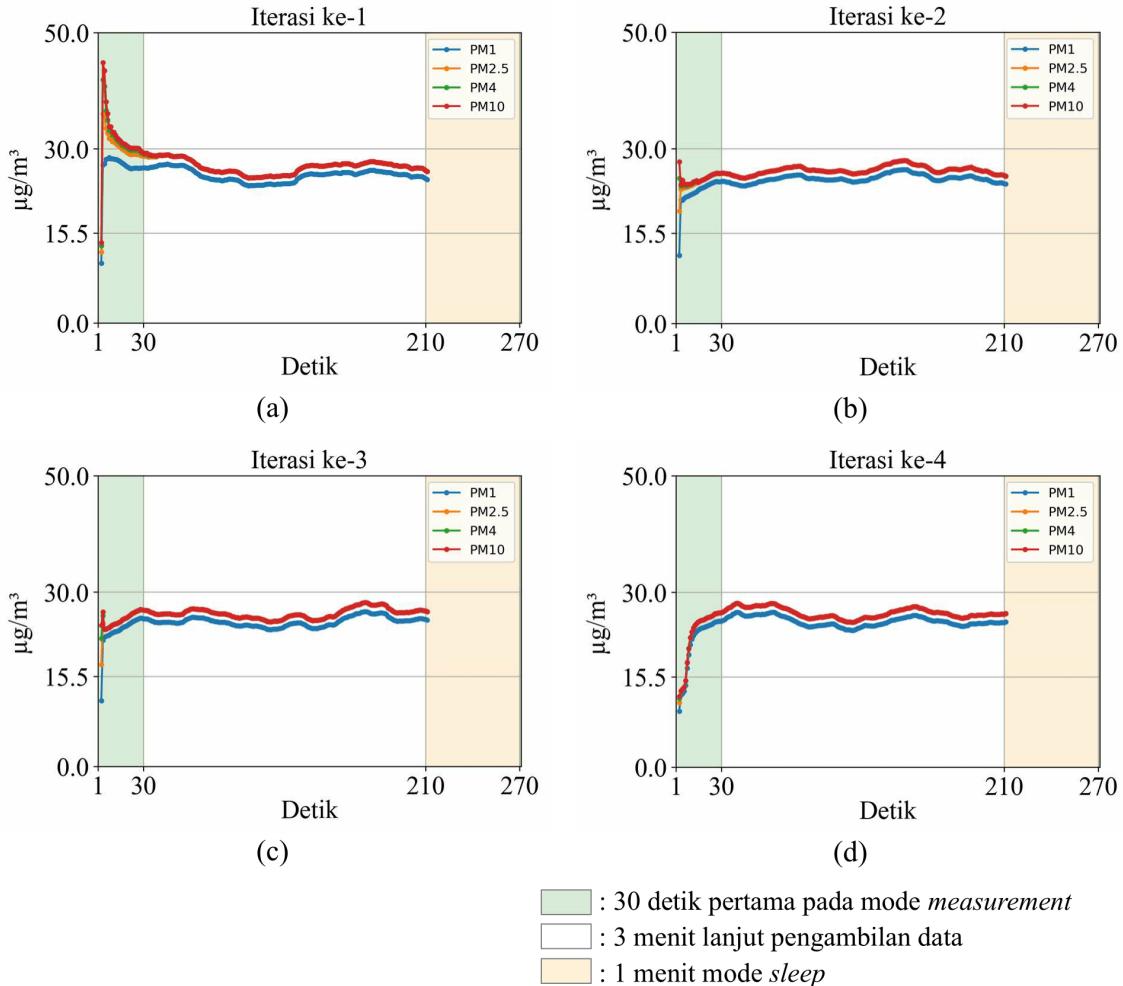
nilai yang dibaca akan tidak sesuai. Untuk itu, dibutuhkan waktu setelah masuk ke mode *measurement* sebelum nilai pengukuran dibaca. Pada pengujian ini akan dilihat perbedaan antara hasil pembacaan nilai pengukuran dengan dan tanpa penggunaan waktu *start-up* yang dilakukan di ruang kerja.



Gambar 4.32. Grafik data pengukuran dengan menggunakan waktu *start-up* selama 30 detik untuk parameter nilai konsentrasi massa partikulat

Pada Bab 3 sebelumnya telah dirancang bahwa waktu *start-up* yang digunakan adalah 30 detik. Pada pengujian ini, sensor pertama akan masuk ke mode *measurement* dan menunggu selama 30 detik. Setelah itu, dilakukan pembacaan nilai pengukuran selama 3 menit dengan jeda tiap pembacaan adalah 1 detik. Program dilanjutkan dengan membuat sensor masuk ke mode *sleep* selama 1 menit. Waktu *sleep* selama 1 menit dimaksudkan untuk memastikan bahwa kipas sudah berhenti berputar. Jadi, ketika berikutnya akan masuk ke mode *measurement* lagi, kipas akan mulai berputar dari keadaan diam. Kemudian, sensor akan *wake-up* dan algoritma tersebut akan berulang-ulang. Dalam pengujian ini dilakukan iterasi sebanyak 4 kali. Gambar 4.32 menunjukkan hasil yang diperoleh dari program tersebut untuk parameter nilai konsentrasi massa PM1, PM2.5, PM4, dan PM10. Warna hijau pada grafik tersebut menunjukkan waktu 30 detik

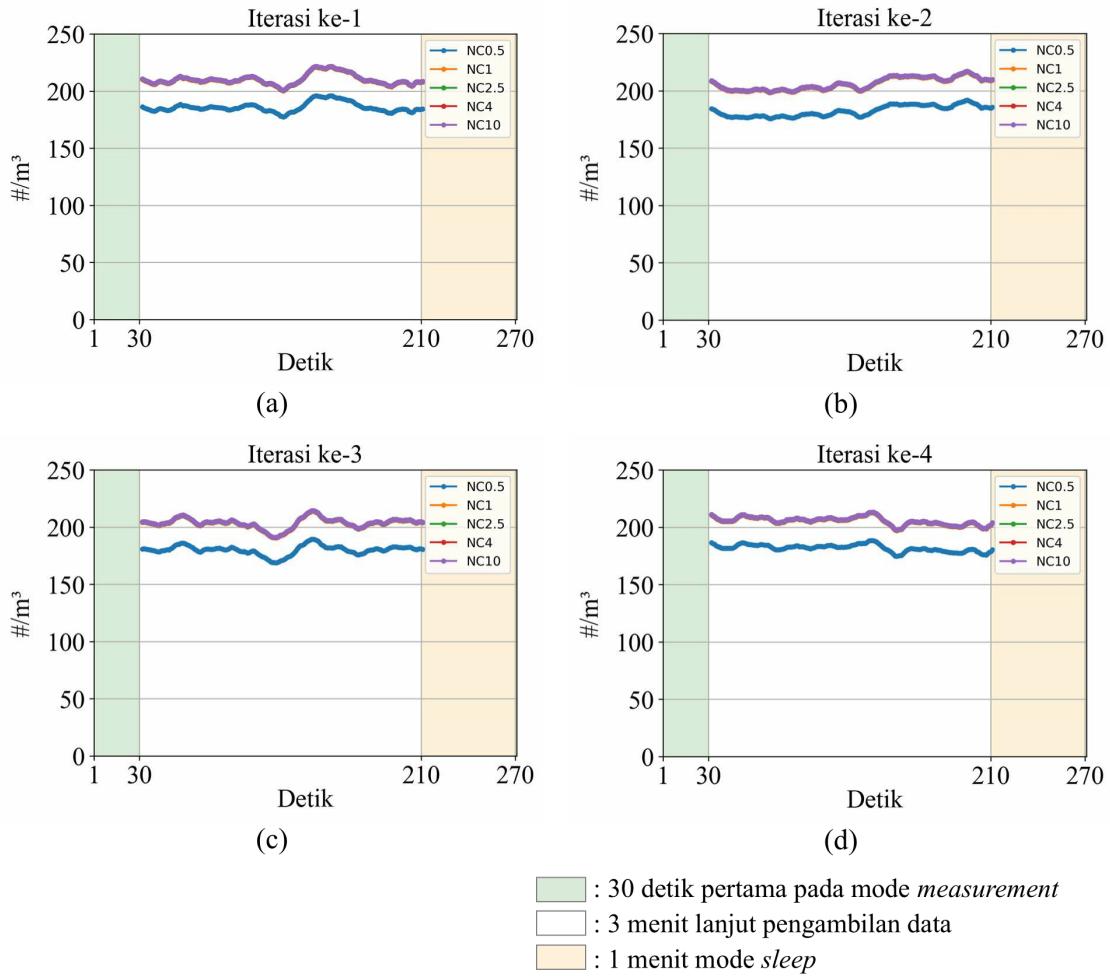
pertama saat sensor masuk ke mode *measurement*, warna putih menunjukkan waktu ketika pembacaan data, dan warna oranye menunjukkan waktu mode *sleep* selama 1 menit. Dari keenam iterasi tersebut, pembacaan keempat parameter nilai dapat dikatakan relatif stabil yang berada pada nilai sekitar 20 hingga  $30 \mu\text{g}/\text{m}^3$ .



Gambar 4.33. Grafik data pengukuran tanpa waktu *start-up* untuk parameter nilai konsentrasi massa partikulat

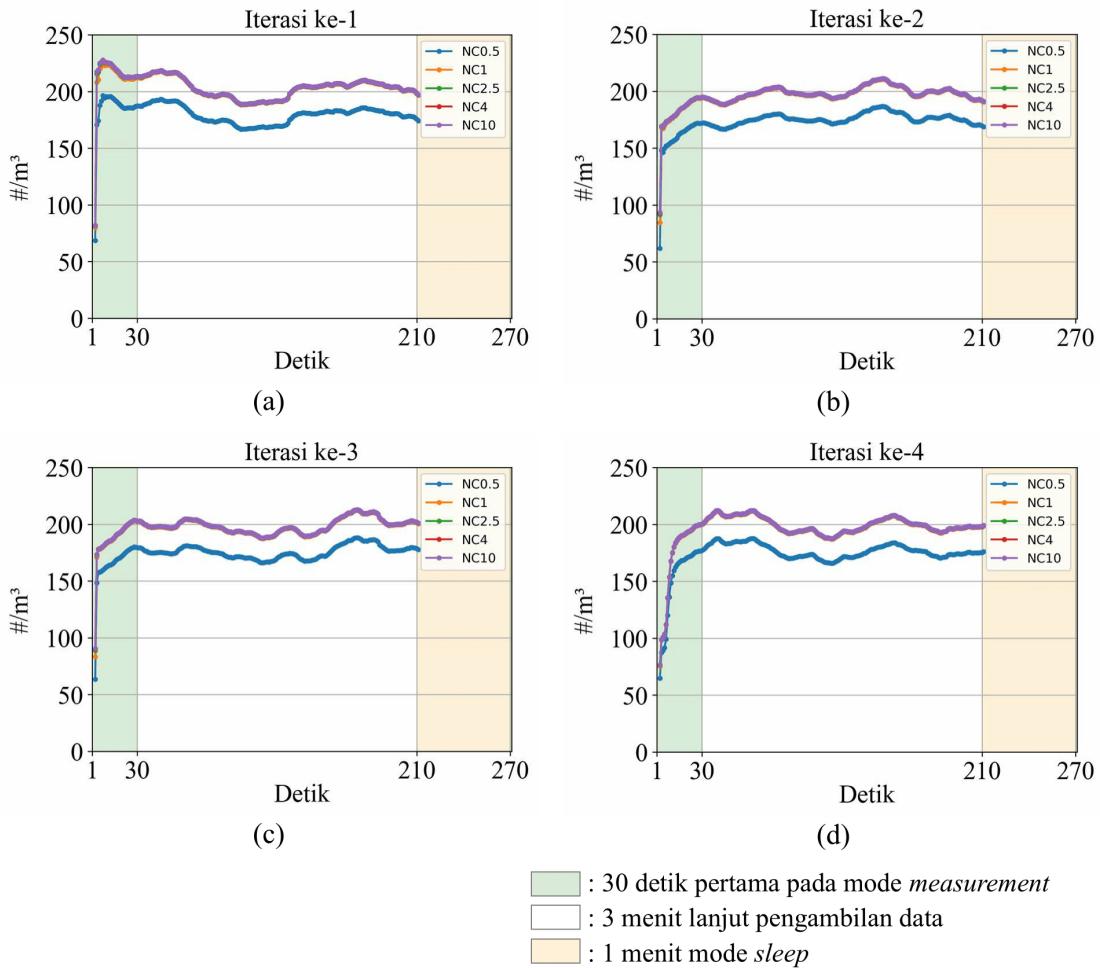
Pengujian tanpa menggunakan waktu *start-up* dilakukan dengan langsung membaca data ketika sensor baru masuk ke mode *measurement*. Untuk membandingkannya dengan yang menggunakan waktu *start-up*, pembacaan data dilakukan selama 3,5 menit terhitung dari sensor masuk ke mode *measurement*. Kemudian dilanjutkan dengan sensor masuk ke mode *sleep* selama 1 menit. Gambar 4.33 menunjukkan grafik hasil pembacaan data tanpa waktu *start-up* dalam empat kali iterasi untuk parameter nilai konsentrasi massa partikulat. Indikator warna menunjukkan hal yang sama dengan sebelumnya. Hanya saja, kali ini pada warna hijau terdapat data pengukuran karena langsung dilakukan pembacaan ketika masuk ke mode *measurement*. Dari grafik tersebut dapat dilihat bahwa pada beberapa detik awal, nilai pengukuran sangat kecil atau besar. Seiring berjalananya waktu, nilai pengukuran mulai menuju angka dikisaran 20 hingga  $30 \mu\text{g}/\text{m}^3$ . Nilai-nilai

awal tersebut didapat ketika kipas belum berputar pada kecepatan targetnya. Namun, ketika sudah 30 detik terlewati, nilai pengukuran cenderung stabil di rentang tersebut selama 3 menit ke depan. Dengan kata lain, nilai pembacaan yang didapat setelah waktu 30 detik mirip seperti pada pengujian dengan waktu *start-up*. Dengan begitu benar bahwa nilai pengukuran pada detik-detik awal sensor masuk ke mode *measurement* belum menunjukkan nilai pengukuran yang sebenarnya. Jadi, untuk mendapat nilai pengukuran yang sesungguhnya, waktu *start-up* ini penting untuk digunakan.



Gambar 4.34. Grafik data pengukuran dengan menggunakan waktu *start-up* selama 30 detik untuk parameter nilai konsentrasi jumlah partikulat

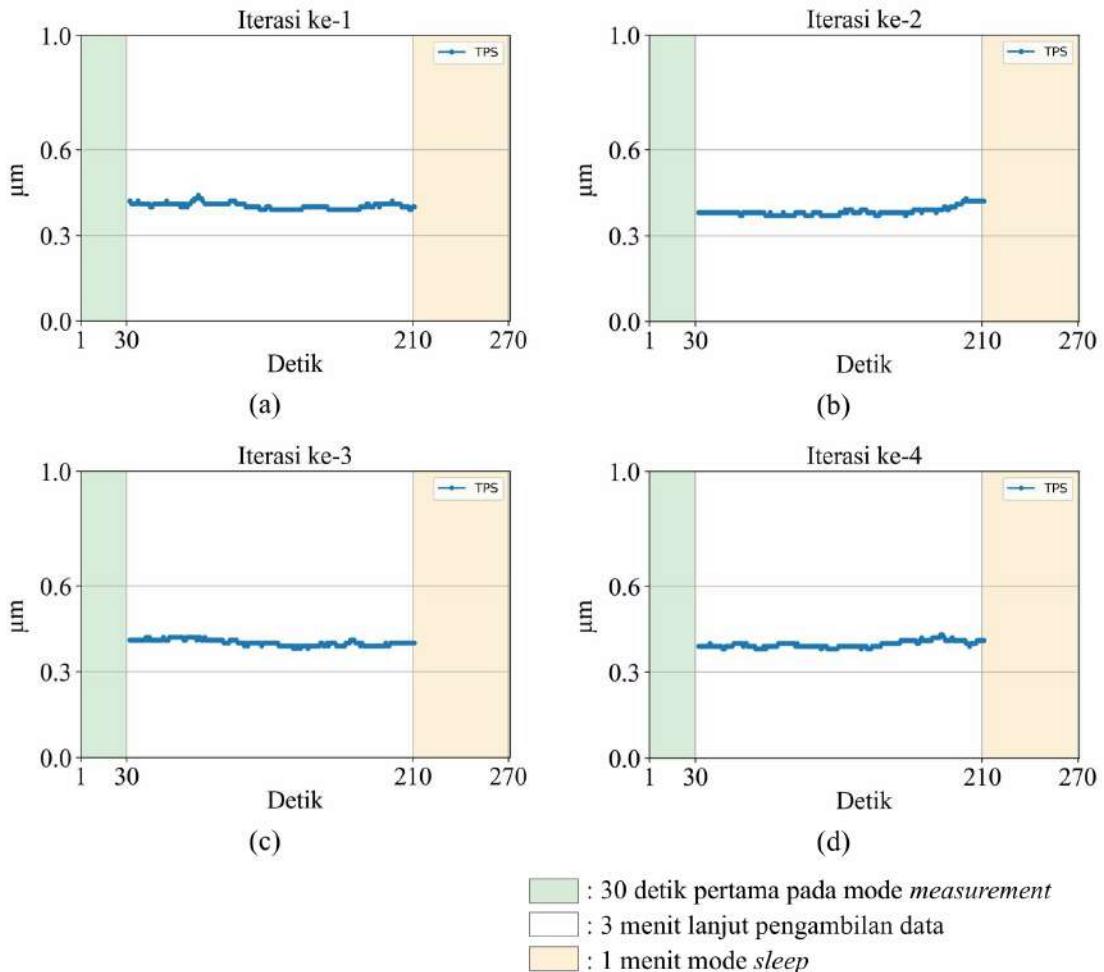
Gambar 4.34 dan Gambar 4.35 menunjukkan perbedaan penggunaan waktu *start-up* untuk parameter nilai konsentrasi jumlah PM0.5, PM1, PM2.5, PM4, dan PM10. Hasil yang menggunakan waktu *start-up* selama 30 detik ditunjukkan pada Gambar 4.34. Dari keempat grafik tersebut, dapat dilihat bahwa nilai konsentrasi jumlah berada di dalam rentang nilai 150 hingga 250 #/m<sup>3</sup>. Sementara itu, Gambar 4.35 menunjukkan grafik hasil pengukuran ketika tidak menggunakan waktu *start-up*. Nilai yang berada pada area berwarna hijau atau pada 30 detik pertama awalnya sangat kecil. Mirip seperti pada parameter nilai konsentrasi massa, nilai yang yang kecil tersebut belum menunjukkan



Gambar 4.35. Grafik data pengukuran tanpa waktu *start-up* untuk parameter nilai kon-sentrasi jumlah partikulat

pembacaan yang benar karena kipas belum berputar dengan kecepatan targetnya. Setelah 30 detik, nilai mulai naik menuju rentang yang sama pada hasil pengujian dengan waktu *start-up*. Jadi, setelah 30 detik, nilai yang ditunjukkan merupakan nilai yang sudah sesuai.

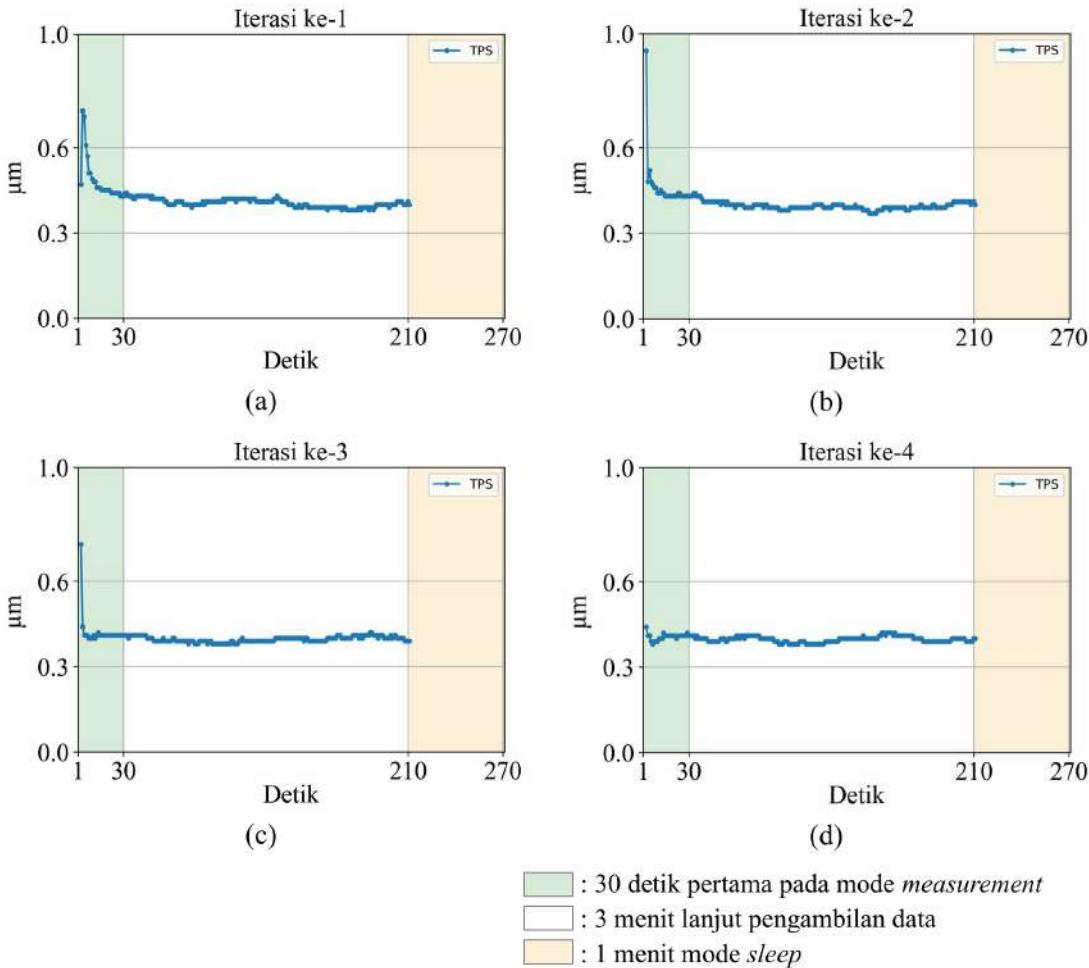
Perbedaan ini juga tampak pada parameter nilai *typical particle size*, yaitu ukuran diameter rata-rata dari partikulat di sampel udara. Gambar 4.36 menunjukkan grafik dari pengukuran menggunakan waktu *start-up* selama empat kali iterasi. Grafik tersebut menunjukkan bahwa selama 3 menit pengambilan data, masing-masing iterasi menunjukkan nilai yang stabil, yaitu pada rentang 0,3 hingga 0,6  $\mu\text{m}$ . Sementara itu, Gambar 4.37 menunjukkan grafik hasil pengukuran pada kondisi tanpa menggunakan waktu *start-up*. Dari keempat grafik tersebut, terdapat beberapa iterasi yang menunjukkan nilai awal yang terukur sangat rendah atau tinggi. Memang mirip seperti kedua nilai parameter parameter lainnya, tetapi beberapa iterasi lainnya menunjukkan nilai awal yang tidak terlalu signifikan berbeda dengan nilai-nilai pembacaan pada detik-detik setelahnya. Hal tersebut dapat terjadi karena jenis partikulat yang dominan adalah PM1. Dan nilai terkecil



Gambar 4.36. Grafik data pengukuran dengan menggunakan waktu *start-up* selama 30 detik untuk parameter nilai *typical particle size* partikulat

untuk tiap jenis ukuran partikulat adalah terhitung dari  $0,3 \mu\text{m}$ . Jadi tidak ada nilai di bawah  $0,3 \mu\text{m}$ . Meskipun demikian, karena masih terdapat pembacaan yang tidak sesuai pada awal masuk mode *measurement*, penggunaan waktu *start-up* tetap menunjukkan hasil yang lebih sesuai.

Dari pengujian yang dilakukan di atas juga dapat dianalisis bahwa terdapat satu jenis partikulat yang dominan, yaitu PM1. Hal itu dapat diketahui dari nilai *typical particle size* yang cenderung berada pada rentang  $0,3$  hingga  $0,6 \mu\text{m}$ . Selain dari parameter *typical particle size*, pernyataan tersebut juga dapat diketahui dari nilai konsentrasi massa. Nilai konsentrasi massa antara PM1 dengan ketiga jenis lainnya memiliki perbedaan yang sangat kecil. Karena nilai PM1 sebenarnya masuk di dalam jenis PM2.5, PM4, dan PM10, maka diketahui bahwa nilai PM1 adalah yang paling dominan.



Gambar 4.37. Grafik data pengukuran tanpa waktu *start-up* untuk parameter nilai *typical particle size* partikulat

### 4.3.2 Performa Sensor

Untuk memberikan gambaran yang lebih detail mengenai performa sensor dalam hasil pengukuran sensor, pada bagian ini akan dihitung tingkat presisinya. Tingkat akurasi tidak dianalisis karena tidak ada nilai pembacaan partikulat lain yang dapat menjadi acuan. Data hasil pengukuran menggunakan waktu *start-up* pada bagian sebelumnya diolah untuk menunjukkan bahwa nilai yang didapatkan stabil. Pengolahan akan dilakukan untuk keempat iterasi pengukuran. Setiap iterasi, pembacaan data dilakukan selama 3 menit yang berarti terdapat 180 sampel tiap iterasinya. Yang perlu diperhatikan di sini adalah kondisi udara tidak benar-benar dalam kondisi yang sama selama 3 menit pengukuran tersebut. Naik turunnya konsentrasi partikulat yang terukur dapat disebabkan kondisi udara yang berubah atau kesalahan dari sensor itu sendiri.

Tabel 4.5 menunjukkan hasil perhitungan tingkat presisi dari parameter nilai konsentrasi massa PM1, PM2.5, PM4, PM10, dan parameter nilai *typical particle size* untuk keempat iterasi.  $\bar{x}$  adalah nilai rata-rata,  $\sigma$  adalah nilai simpangan baku, dan RSD adalah

Tabel 4.5. Hasil Analisis Konsentrasi Massa Partikulat dan *Typical Particle Size*

i	Paramenter Analisis	Parameter Nilai				
		MC PM1 $\mu\text{g}/\text{m}^3$	MC PM2.5 $\mu\text{g}/\text{m}^3$	MC PM4 $\mu\text{g}/\text{m}^3$	MC PM10 $\mu\text{g}/\text{m}^3$	TPS $\mu\text{m}$
1	max.	27.76	29.35	29.35	29.35	0.44
	min.	25.13	26.57	26.57	26.57	0.39
	$\bar{x}$	26.34	27.85	27.85	27.85	0.40
	$\sigma$	0.60	0.63	0.63	0.63	0.01
	RSD	2.26	2.26	2.26	2.26	2.56
2	max.	27.17	28.73	28.73	28.73	0.43
	min.	24.90	26.33	26.33	26.33	0.37
	$\bar{x}$	25.84	27.33	27.33	27.33	0.38
	$\sigma$	0.66	0.70	0.70	0.70	0.01
	RSD	2.57	2.57	2.57	2.57	3.56
3	max.	28.43	30.07	30.07	30.07	0.42
	min.	24.31	25.71	25.71	25.71	0.37
	$\bar{x}$	26.04	27.54	27.54	27.54	0.39
	$\sigma$	1.14	1.20	1.20	1.20	0.01
	RSD	4.37	4.37	4.37	4.37	2.53
4	max.	27.13	28.77	28.88	28.93	0.44
	min.	25.08	26.53	26.53	26.53	0.37
	$\bar{x}$	25.92	27.42	27.43	27.43	0.40
	$\sigma$	0.51	0.55	0.57	0.57	0.02
	RSD	1.96	2.02	2.07	2.09	4.16

nilai simpangan baku relatif. Simpangan baku memberikan nilai tingkat presisi dalam satuan asli dari masing-masing parameter nilai. Sementara itu, RSD memberikan nilai tingkat presisi dalam persentase. Untuk kelima parameter nilai lainnya, yaitu konsentrasi jumlah PM0.5, PM1, PM2.5, PM4, dan PM10, hasil perhitungan ditunjukkan pada Tabel 4.6. Hasilnya serupa dengan parameter nilai sebelumnya. Keduanya menunjukkan bahwa nilai pengukuran untuk seluruh parameter nilai ini berada di dalam rentang yang dibatasi pada *datasheet*-nya.

Rata-rata tingkat presisi dari keempat iterasi ditunjukkan pada Tabel 4.7. Parameter nilai konsentrasi massa dan konsentrasi jumlah menunjukkan nilai RSD-nya adalah sekitar 2,8%. Sementara itu, parameter nilai *typical particle size* memiliki RSD yang sedikit lebih besar, yaitu 3,2%. Ini dapat terjadi karena rentang nilai untuk parameter tersebut lebih kecil. Hasil analisis ini dapat menunjukkan bahwa data pengukuran benar stabil jika pembacaannya dilakukan setelah 30 detik sensor masuk ke mode *measurement*.

Tabel 4.6. Hasil Analisis Konsentrasi Jumlah Partikulat

<i>i</i>	Paramenter Analisis	Parameter Nilai				
		NC PM0.5 #/m <sup>3</sup>	NC PM1 #/m <sup>3</sup>	NC PM2.5 #/m <sup>3</sup>	NC PM4 #/m <sup>3</sup>	NC PM10 #/m <sup>3</sup>
1	max.	195.91	221.06	221.52	221.58	221.63
	min.	177.37	200.15	200.56	200.62	200.66
	$\bar{x}$	185.90	209.76	210.20	210.26	210.30
	$\sigma$	4.21	4.75	4.76	4.76	4.76
	RSD	2.26	2.26	2.26	2.26	2.26
2	max.	191.77	216.40	216.84	216.91	216.96
	min.	175.73	198.29	198.70	198.76	198.81
	$\bar{x}$	182.41	205.83	206.26	206.31	206.36
	$\sigma$	4.69	5.29	5.30	5.30	5.30
	RSD	2.57	2.57	2.57	2.57	2.57
3	max.	200.68	226.45	226.92	226.98	227.04
	min.	171.61	193.65	194.05	194.10	194.15
	$\bar{x}$	183.82	207.42	207.85	207.91	207.95
	$\sigma$	8.03	9.06	9.08	9.08	9.08
	RSD	4.37	4.37	4.37	4.37	4.37
4	max.	191.21	215.95	216.48	216.56	216.61
	min.	177.04	199.78	200.19	200.25	200.29
	$\bar{x}$	182.93	206.44	206.87	206.93	206.98
	$\sigma$	3.55	4.04	4.06	4.06	4.07
	RSD	1.94	1.95	1.96	1.96	1.96

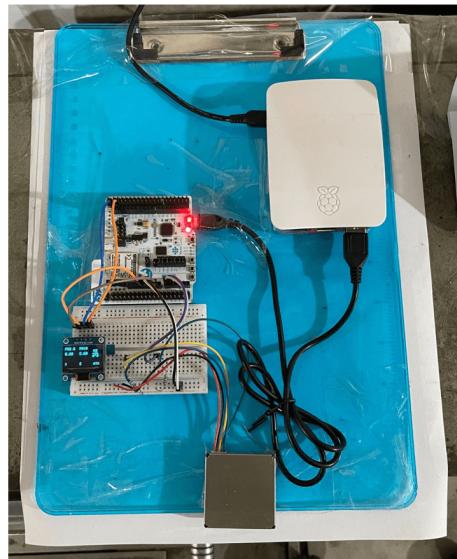
Tabel 4.7. Tingkat Presisi Setiap Parameter Nilai Partikulat

Parameter Nilai	$\sigma$	RSD
MC PM1	0.73 $\mu\text{g}/\text{m}^3$	2.79 %
MC PM2.5	0.77 $\mu\text{g}/\text{m}^3$	2.81 %
MC PM4	0.78 $\mu\text{g}/\text{m}^3$	2.82 %
MC PM10	0.78 $\mu\text{g}/\text{m}^3$	2.82 %
NC PM0.5	5.12 #/m <sup>3</sup>	2.79 %
NC PM1	5.78 #/m <sup>3</sup>	2.79 %
NC PM2.5	5.80 #/m <sup>3</sup>	2.79 %
NC PM4	5.80 #/m <sup>3</sup>	2.79 %
NC PM10	5.80 #/m <sup>3</sup>	2.79 %
TPS	0.01 $\mu\text{m}$	3.20 %

#### 4.4 Pengujian Sensor SPS30 untuk Mengukur Konsentrasi Partikulat

Pengujian ini dilakukan untuk menjalankan sistem secara keseluruhan untuk mengukur konsentrasi partikulat di udara. Pengukuran dilakukan pada tiga lokasi yang berbeda di UGM Press, yaitu ruang produksi, area *outdoor*, dan ruang *showroom*. Pengukuran ini bertujuan untuk menguji sistem dan mendapatkan indeks kualitas udara di ketiga lokasi tersebut. Pada pengujian ini, rangkaian pada Gambar 3.5 diberi sumber daya dari Raspberry Pi 3 Model B yang sekaligus sebagai tempat penyimpanan data selama pengujian berlangsung. Dengan begitu, rangkaian sistem yang digunakan ditunjukkan pada Gambar 4.38. Data pengukuran yang tersimpan pada Raspberry Pi kemudian diolah untuk mencari tahu indeks kualitas udaranya.

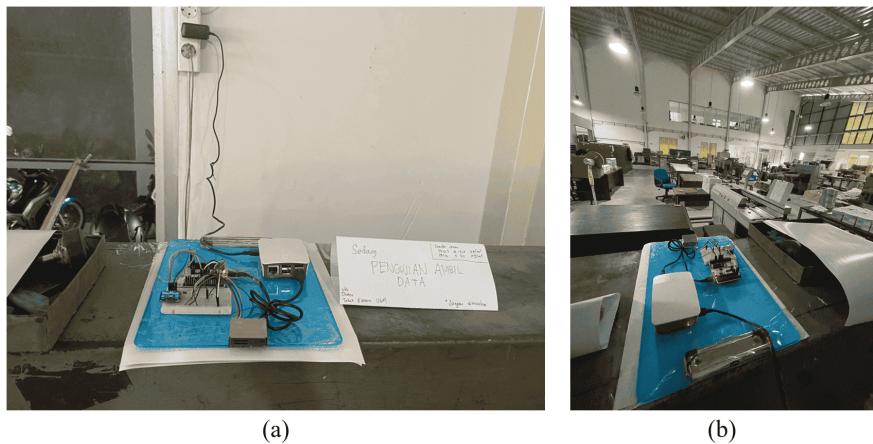
*Firmware* yang digunakan pada pengujian ini ditunjukkan pada diagram alir pada Gambar 3.10. Sistem dijalankan selama kurang lebih satu minggu pada masing-masing lokasi pengujian. Proses pengambilan data pengukuran ini dilakukan secara bergantian mengingat sistem hanya tersedia satu unit. Jadi, proses pengambilan data untuk ketiga lokasi memakan waktu bersih selama tiga minggu pada bulan April hingga Mei. Data pengukuran yang selanjutnya diolah adalah PM2.5 dan PM10 karena dua jenis tersebut yang menjadi tolak ukur untuk kualitas udara menurut ISPU.



Gambar 4.38. Rangkaian sistem untuk pengujian mengukur konsentrasi partikulat

#### 4.4.1 Pengujian di Ruang Produksi

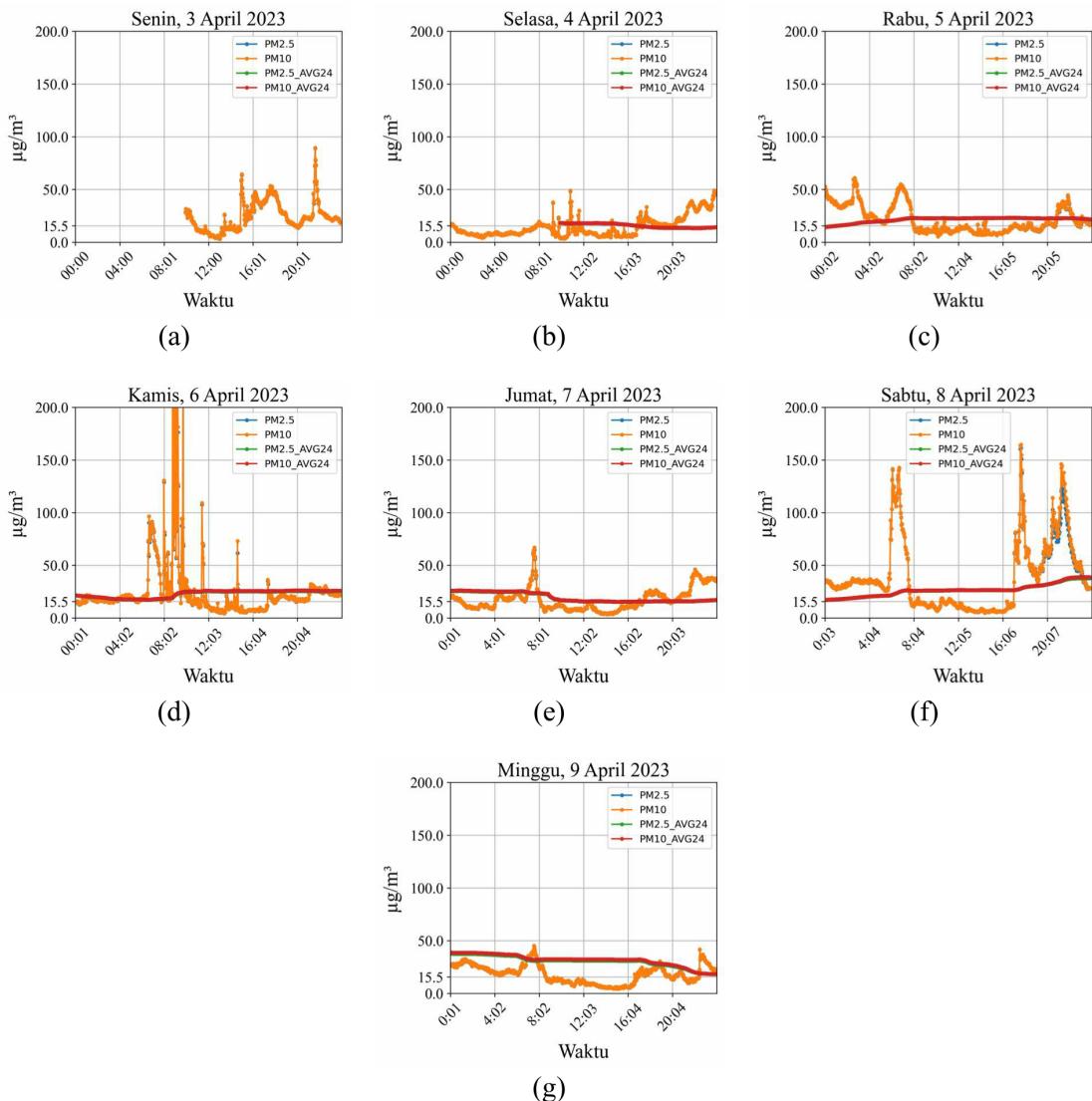
Pengujian pertama dilakukan di ruang produksi UGM Press. Ruang produksi bersih alat-alat untuk keperluan percetakan buku. Rangkaian sistem diletakkan di bagian pinggir ruangan, tepatnya di dekat mesin *binding*. Pada saat pengujian dilakukan, ruang produksi beroperasi dari jam 08:00 hingga 15:00. Pada saat jam operasi, jendela, ventilasi, dan pintu dibiarkan terbuka untuk sirkulasi udara. Sementara itu, saat di luar jam operasi dan hari libur, kondisi ruangan tertutup rapat.



Gambar 4.39. Penempatan sistem sensor di ruang produksi

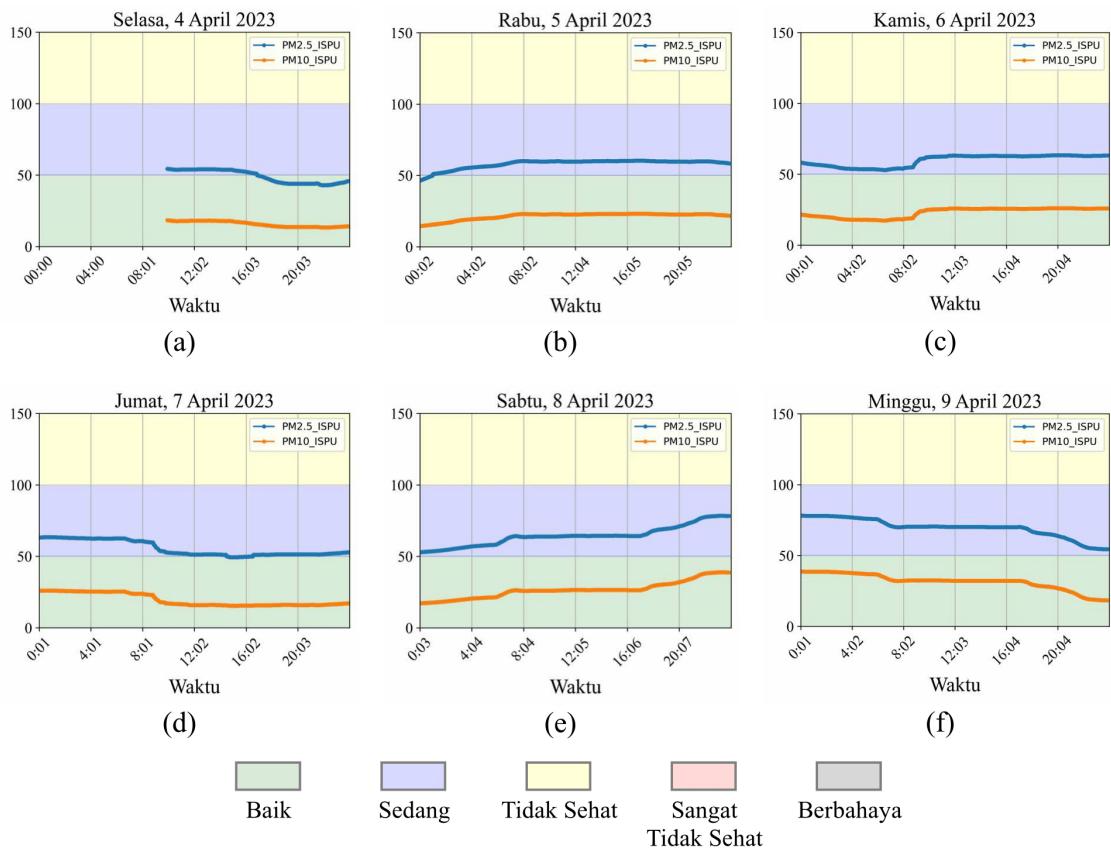
Sistem berjalan di ruang produksi mulai dari hari Senin, 3 April 2023 sekitar pukul 10:00 hingga Minggu, 9 April 2023. Hari Jumat, 7 April 2023 adalah hari libur nasional sehingga ruang produksi tidak beroperasi sama seperti pada hari Sabtu dan Minggu. Grafik data pengukuran ditunjukkan pada Gambar 4.40. Grafik tersebut menunjukkan nilai PM2.5 dan PM10 setiap 2 menit (warna oranye dan biru) dan rata-ratanya dalam 24 jam terakhir (warna merah dan hijau). Grafik menunjukkan bahwa nilai PM2.5 dan PM10 tidak jauh berbeda. Kenaikan nilai salah satu jenis diikuti oleh jenis satunya. Nilai yang terbaca naik turun tergantung kondisi udara di lokasi saat itu. Sesekali konsentrasi massa partikulat bernilai sangat tinggi seperti yang ditunjukkan pada hari Kamis dan Sabtu. Secara umum, grafik tersebut menunjukkan bahwa konsentrasi massa partikulat pada malam hari cenderung lebih tinggi dibanding pada siang hari. Sering kali ketika malam hari, nilai yang terukur di atas nilai rata-rata 24 jamnya. Dengan begini, secara keseluruhan sistem dapat bekerja untuk mengukur kadar konsentrasi partikulat di udara.

Nilai konsentrasi massa partikulat yang telah didapat kemudian diolah untuk mendapatkan indeks kualitas udara yang berdasarkan pada ISPU. Perhitungannya menggunakan Algoritma 1 untuk PM2.5 dan Algoritma 2 untuk PM10. Nilai ISPU baru didapat pada hari Selasa karena membutuhkan nilai rata-rata konsentrasi partikulat dalam 24 jam. Grafik hasil perhitungan ditunjukkan pada Gambar 4.41. Warna pada grafik tersebut menunjukkan keterangan kategori dari nilai ISPU. Pada grafik tersebut, nilai ISPU PM2.5



Gambar 4.40. Grafik data pengukuran PM2.5 dan PM10 di ruang produksi

selama waktu pengujian lebih dominan berada pada kategori warna biru, yaitu kategori Sedang. Sementara itu, nilai ISPU PM10 lebih dominan pada kategori warna hijau, yaitu kategori Baik. Walaupun grafik konsentrasi massa menunjukkan nilai PM2.5 dan PM10 tidak jauh berbeda, nilai ISPU mereka ternyata berbeda jauh. Hal ini disebabkan rentang nilai untuk tiap kategorinya pada PM2.5 dan PM10 berbeda seperti yang ditunjukkan pada Tabel 2.1. Karena kedua jenis partikulat memiliki kategori ISPU yang berbeda, indeks kualitas udara pada lokasi ini mengikuti kategori yang paling tinggi, yaitu dominan Sedang. Jadi, dari sisi partikulat saja, kualitas udara di ruang produksi UGM Press masih dapat diterima, tetapi dianjurkan bagi kelompok sensitif untuk mengurangi aktivitas fisik yang terlalu berat dan lama.



Gambar 4.41. Grafik ISPU PM2.5 dan PM10 di ruang produksi



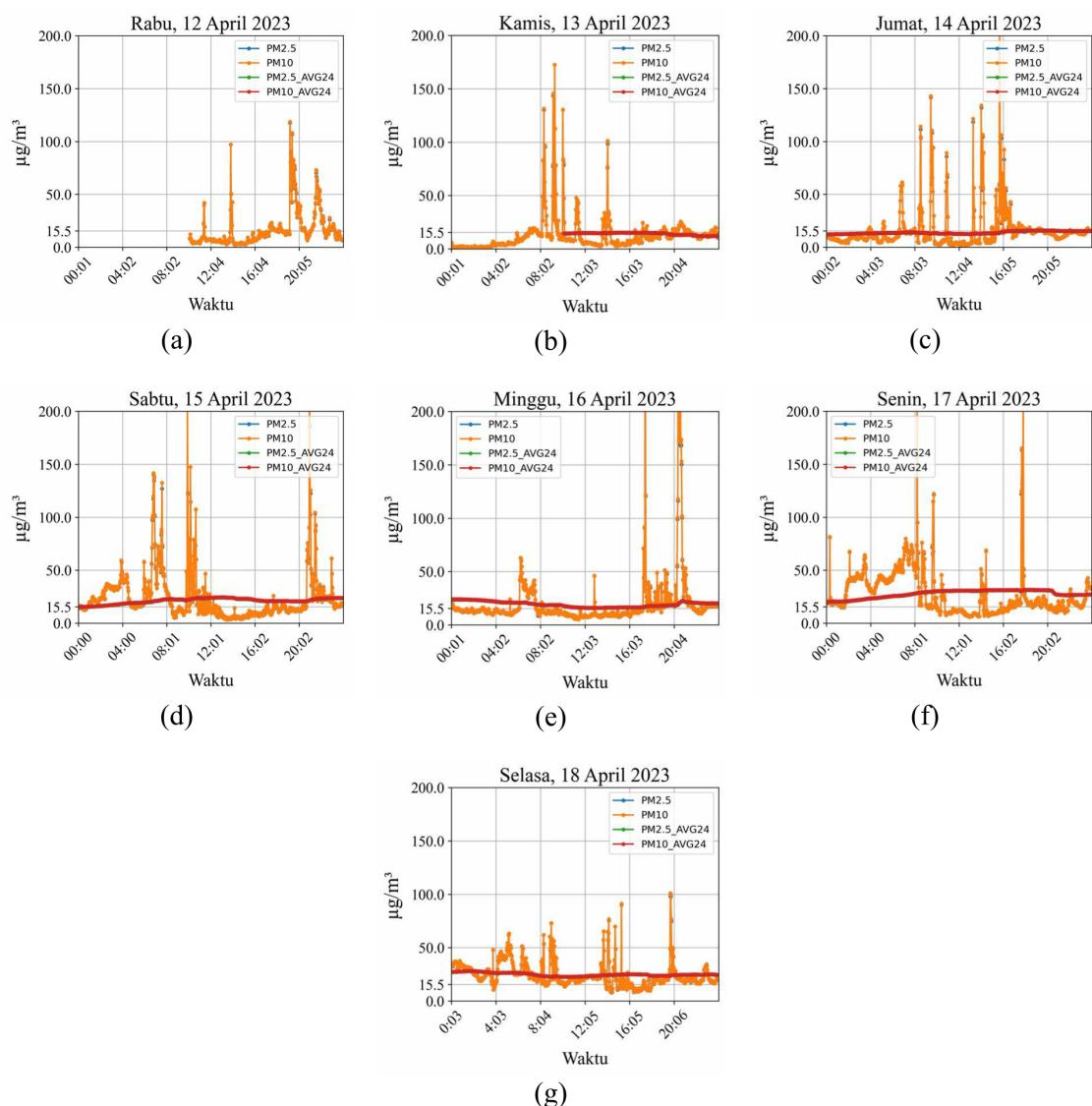
Gambar 4.42. Penempatan sistem sensor pada area *outdoor*

#### 4.4.2 Pengujian di Area *Outdoor*

Pengujian kedua dilakukan pada area *outdoor* di UGM Press. Area *outdoor* yang dimaksud terletak di sisi barat UGM Press dekat ruang generator listrik dan area parkir. Area di depan sensor sering dilalui kendaraan ketika hendak parkir. Karena berada di luar ruangan, tentu saja aliran udaranya lebih lancar dibandingkan di ruang produksi. Namun,

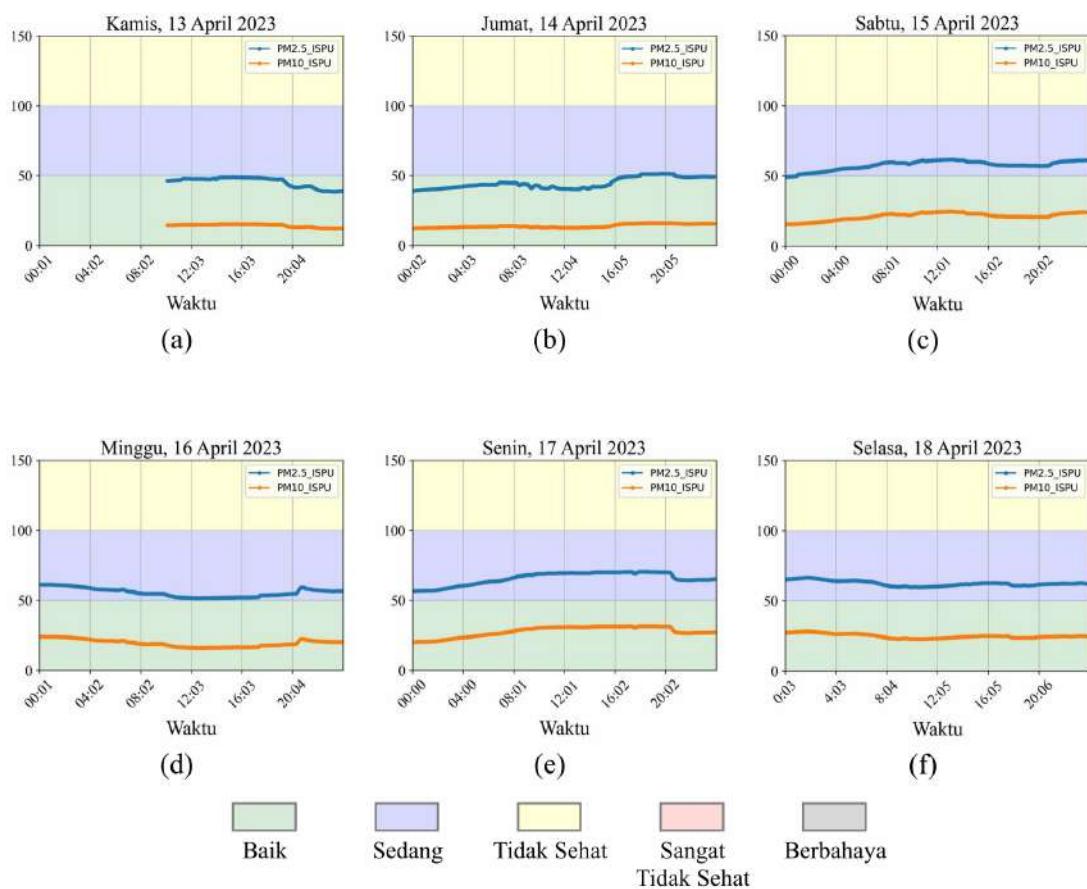
karena di luar ruangan juga, kondisi cuaca saat itu pasti sangat memengaruhi konsentrasi partikulat di udara.

Sistem berjalan di area *outdoor* mulai dari hari Rabu, 12 April 2023 sekitar pukul 10:00 hingga hari Selasa, 18 April 2023. Grafik data pengukuran konsentrasi massa PM2.5 dan PM10 ditampilkan pada Gambar 4.43. Grafik tersebut menunjukkan bahwa nilai konsentrasi partikulat yang terukur memiliki tren yang berbeda dengan di ruang produksi. Konsentrasi massa PM2.5 dan PM10 sering kali loncat sangat tinggi dalam waktu yang singkat. Kemudian turun menjadi seperti semula dalam waktu yang singkat pula. Selain itu, dapat dilihat pula bahwa pada tanggal 15-18 April 2023 dalam rentang pukul 00:00 hingga 09:00, nilai PM2.5 dan PM10 cenderung menjadi lebih tinggi dengan pola yang hampir mirip. Meskipun demikian, sama seperti sebelumnya, nilai PM2.5 dan PM10 tidak jauh berbeda.



Gambar 4.43. Grafik data pengukuran PM2.5 dan PM10 di area *outdoor*

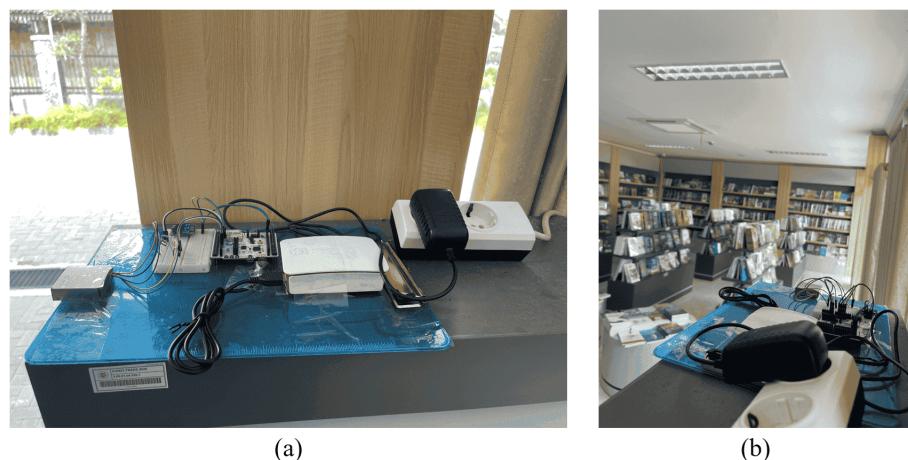
Nilai indeks kualitas udara berdasarkan ISPU untuk area *outdoor* ditunjukkan pada Gambar 4.44. Nilai ISPU area *outdoor* baru didapat pada hari Kamis, 13 April 2023. Sama seperti sebelumnya, nilai ISPU untuk PM2.5 lebih tinggi dari jenis PM10. Namun, secara umum jika dibandingkan dengan nilai pada ruang produksi, nilai ISPU pada area *outdoor* ini cenderung lebih rendah. Pada tanggal 13 dan 14 April 2023, nilai ISPU PM2.5 lebih dominan di area berwarna hijau, yang artinya masuk dalam kategori Baik. Artinya, kualitas udara pada kedua hari tersebut sangat baik dan tidak memberikan efek negatif. Akan tetapi, pada hari-hari berikutnya nilai ISPU untuk hari-hari berikutnya lebih dominan di area berwarna biru atau Sedang. Meskipun masih tergolong baik, kelompok sensitif masih direkomendasikan untuk mengurangi aktivitas fisik yang berat.



Gambar 4.44. Grafik ISPU PM2.5 dan PM10 di area *outdoor*

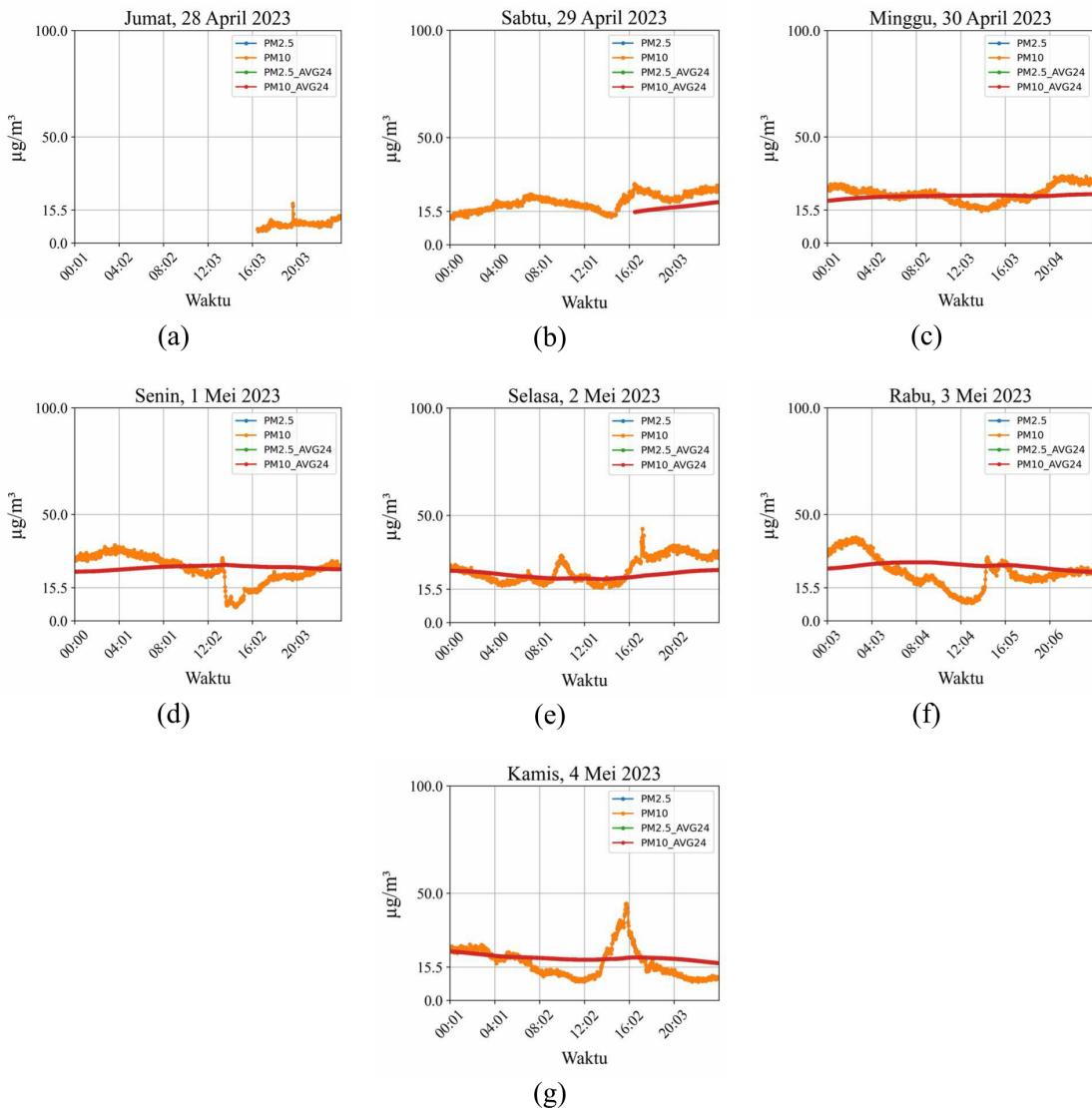
#### 4.4.3 Pengujian di Ruang *Showroom*

Pengujian ketiga dilakukan pada ruang *showroom* UGM Press. Ruangan ini adalah lobi utama dan tempat buku-buku dipajang untuk dijual. Di ruang *showroom* ini terdapat AC yang membantu dalam perputaran udara di ruangan sehingga udara dan suhu di dalamnya membuat orang yang bekerja atau pengunjung menjadi nyaman. Sistem pemantauan pada ruangan ini diletakkan di sisi dekat pintu masuk dan rak-rak buku. Pada saat pengukuran dilakukan, jam operasi UGM Press dimulai dari pukul 08:00 hingga 16:00.



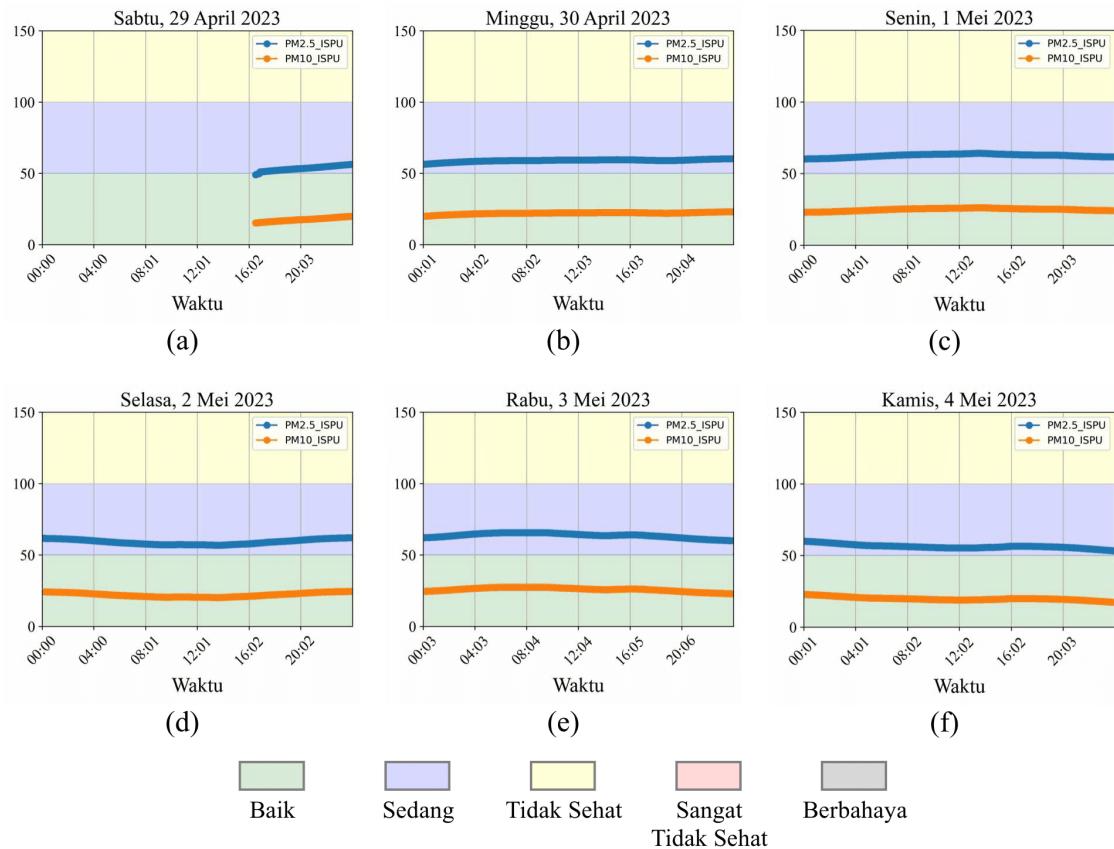
Gambar 4.45. Penempatan sistem sensor di ruang *showroom*

Sistem berjalan di ruang *showroom* mulai dari hari Jumat, 28 April hingga hari Kamis, 4 Mei 2023. Gambar 4.46 menunjukkan grafik pengukuran di ruang *showroom*. Grafik tersebut menunjukkan bahwa hasil konsentrasi massa PM2.5 dan PM10 di ruang *showroom* cenderung lebih stabil dengan nilai yang rendah dibanding dua lokasi lainnya. Tidak ada lonjakan nilai yang terlalu tinggi seperti pada lokasi lain. Selama pengukuran, nilai minimum PM10 adalah  $5,58 \mu\text{g}/\text{m}^3$  dan nilai maksimumnya adalah  $45,18 \mu\text{g}/\text{m}^3$ . Berbeda dengan dua lokasi sebelumnya yang nilai maksimumnya mencapai ratusan. Hasil tersebut menunjukkan bahwa di dalam ruangan tidak ada faktor yang dapat menjadi sumber konsentrasi partikulat yang tinggi. Meskipun demikian, nilai konsentrasi partikulat PM2.5 cenderung masih di atas batas atas untuk kategori Baik, yaitu  $15,5 \mu\text{g}/\text{m}^3$  dalam 24 jam. Hal ini akan lebih jelas terlihat pada grafik nilai ISPU-nya.



Gambar 4.46. Grafik data pengukuran PM2.5 dan PM10 di ruang *showroom*

Gambar 4.47 menunjukkan grafik ISPU yang dikonversi dari nilai rata-rata konentrasi massa PM2.5 dan PM10 dalam 24 jam terakhir di ruang *showroom*. Maka dari itu, nilai ISPU baru diperoleh pada hari Sabtu, 29 April 2023. Pada grafik tersebut dapat dilihat lebih jelas bahwa nilai ISPU untuk PM2.5 berada di kategori Sedang selama waktu pengujian. Nilainya memang lebih stabil tanpa pergerakan naik turun yang dalam. Akan tetapi, bukan berarti indeks kualitas udaranya lebih bersih dari dua lokasi sebelumnya. Jadi, rekomendasi untuk kelompok sensitif masih berlaku pada ruangan ini.



Gambar 4.47. Grafik ISPU PM2.5 dan PM10 di ruang *showroom*

#### 4.4.4 Perbandingan Kualitas Udara di Ketiga Lokasi

Ketiga lokasi pengujian memiliki nilai ISPU PM2.5 dan PM10 yang berbeda-beda. Tabel 4.8 menunjukkan perbandingan nilai ISPU ketiga lokasi dengan melihat nilai maksimum, minimum, dan rata-ratanya selama pengujian. Jika dilihat dari rata-ratanya selama kurang lebih satu minggu, rata-rata nilai ISPU di ruang produksi adalah yang tertinggi, yaitu mencapai 60,20 untuk PM2.5 dan 23,57 untuk PM10. Sementara itu, rata-rata nilai ISPU terendah terjadi di area *outdoor*, yaitu 56,03 untuk PM2.5 dan 20,75 untuk PM10. Ruang *showroom* menjadi di urutan tengah dengan nilai ISPU maksimumnya adalah yang terendah dan nilai ISPU terendahnya adalah yang terbesar dibandingkan dengan dua lokasi lainnya. Hal itu menunjukkan rentang nilai ISPU pada ruang *showroom* adalah yang terkecil.

Perbedaan rata-rata nilai ISPU di ketiga lokasi tidak jauh, sehingga ketiganya memiliki kesamaan pada kategori nilai ISPU-nya. Karena standar ISPU melihat level kategori pada parameter nilai tertinggi, maka indikator kualitas udara pada ketiga lokasi dilihat pada parameter nilai PM2.5. Ketiga lokasi menunjukkan nilai ISPU untuk parameter nilai PM2.5 cenderung berada pada kategori Sedang. Maka dari itu, meskipun kualitas udara pada ketiga lokasi secara rata-rata masih dapat diterima dengan baik untuk

kesehatan makhluk hidup, perlu diperhatikan untuk kelompok sensitif direkomendasikan untuk mengurangi aktivitas fisik yang terlalu berat atau lama.

Tabel 4.8. Perbandingan Nilai ISPU PM2.5 dan PM10 di Tiga Lokasi Selama Pengujian

Lokasi	ISPU PM2.5			ISPU PM10		
	max.	min.	rata-rata	max.	min.	rata-rata
Ruang Produksi	78,45	42,95	60,20	38,94	13,40	23,57
Area <i>Outdoor</i>	70,38	38,53	56,03	31,55	12,07	20,75
Ruang <i>Showroom</i>	65,71	49,12	59,76	27,50	15,29	22,67

## **BAB V**

### **KESIMPULAN DAN SARAN**

#### **5.1 Kesimpulan**

Pada penelitian telah dikembangkan *firmware* untuk pemantauan kualitas udara yang berfokus pada parameter konsentrasi partikulat menggunakan sensor Sensirion SPS30 dengan mikrokontroler STM32. Dari pengembangan dan pengujian yang telah dilakukan, dapat diambil kesimpulan, sebagai berikut:

1. *Firmware* yang dikembangkan berhasil digunakan untuk komunikasi antara mikrokontroler dan sensor. Dengan fungsi-fungsi yang dikembangkan, sensor dapat bekerja untuk mengukur konsentrasi partikulat di udara dan perintah-perintah lainnya yang tersedia. Algoritma untuk operasi pengukuran konsentrasi partikulat perlu diperhatikan. Sensor SPS30 membutuhkan waktu *start-up* saat memasuki mode *measurement* sebelum mulai membaca data pengukuran untuk mendapat hasil pengukuran yang sesuai. Kecepatan putar kipas memengaruhi hasil pengukuran sensor. Tanpa waktu *start-up*, nilai pada detik-detik awal terbaca sangat kecil atau besar. Namun, setelah melewati waktu *start-up* selama 30 detik, pembacaan akan stabil. Ini ditunjukkan pula dengan tingkat presisinya berdasarkan nilai RSD, yaitu sekitar 2,8% untuk parameter nilai konsentrasi massa dan konsentrasi jumlah, dan 3,2% untuk parameter nilai *typical particle size*.
2. Dengan berhasilnya fungsi yang dikembangkan, maka protokol komunikasi I2C yang digunakan antara mikrokontroler dengan sensor berjalan dengan benar. Komunikasi tersebut diatur dan dijalankan dengan menggunakan fungsi I2C *transmit* atau *receive* pada STM32CubeIDE oleh *board* P-Nucleo-WB55. Dengan menyesuaikan alamat *slave*, sinyal pesan yang akan dikirim, spesifikasi sensor, dan rangkaian elektronisnya, komunikasi akan berjalan dengan lancar.
3. Konsentrasi partikulat yang terukur di tiga lokasi berbeda, yaitu ruang produksi, area *outdoor*, dan ruang *showroom* UGM Press memiliki karakteristik yang berbeda. Hal tersebut dipengaruhi oleh sirkulasi udara dan aktivitas yang terjadi di lokasi tersebut. Dari ketiga lokasi tersebut, didapatkan bahwa area *outdoor* memiliki nilai rata-rata ISPU terendah (56,03 untuk PM2.5), sedangkan rata-rata tertinggi dimiliki oleh ruang produksi (60,20 untuk PM2.5). Namun, perbedaan antara ketiganya tidak besar. Ketiganya menunjukkan kesamaan, yaitu indeks kualitas udara menuju ISPU untuk parameter nilai PM2.5 cenderung berada pada kategori Sedang.

## **5.2 Saran**

Berdasarkan penelitian yang telah dilakukan, saran yang dapat dipertimbangkan untuk penelitian selanjutnya adalah sebagai berikut:

1. Dapat dilakukan pengembangan dan penelitian untuk protokol komunikasi satu lagi yang didukung oleh sensor SPS30, yaitu UART.
2. Penelitian konsentrasi partikulat dapat dilakukan berbarengan dengan parameter besaran lain, seperti temperatur atau kelembapan udara untuk mengetahui bagaimana korelasinya satu sama lain.
3. Penelitian dapat dilanjutkan dengan menambahkan fitur IoT yang memungkinkan untuk pengiriman dan visualisasi data secara *real-time*.

## DAFTAR PUSTAKA

- [1] P. . Jurnal, K. Masyarakat, S. A. Kosentrasi, R. Debu, D. Gangguan, K. Pada, M. Di, P. S. Pltu, S. Arba, J. Kesehatan, L. Poltekkes, and K. Ternate, “Kosentrasi respirable debu particulate matter(pm2,5) dan gangguan kesehatan pada masyarakat di pemukiman sekitar pltu,” *Jurnal Kesehatan Masyarakat*, vol. 9, pp. 178–184, 2019.
- [2] M. Simarmata, A. R. O. Pasanda, I. Marzuki, D. Soputra, F. Sudasman, E. Mohamad, M. Syahrir, S. Hardiyanti, M. Mahyati *et al.*, “Pengantar pencemaran udara,” *Yayasan Kita Menulis*, 2022. [Online]. Available: <https://books.google.co.id/books?id=9WR9EAAAQBAJ>
- [3] D. Chaniago, A. Zahara, and I. Ramadhani, “Indeks standar pencemar udara (ispu) sebagai informasi mutu udara ambien di indonesia,” 9 2020. [Online]. Available: <https://ditppu.menlhk.go.id/portal/read/indeks-standar-pencemar-udara-ispu-sebagai-informasi-mutu-udara-ambien-di-indonesia>
- [4] WHO, “Who global air quality guidelines. particulate matter (pm2.5 and pm10), ozone, nitrogen dioxide, sulfur dioxide and carbon monoxide,” *World Health Organization*, 2021.
- [5] IQAir, “World’s most polluted cities (historical data 2017-2021),” 2021. [Online]. Available: <https://www.iqair.com/world-most-polluted-cities?continent=59af92b13e70001c1bd78e53&country=Rqrg4reHqi8taY4re&state=&sort=-rank&page=1&perPage=50&cities=>
- [6] Sensirion, “Indoor air quality the beginning of a new era,” 2021.
- [7] U. EPA, *Building Air Quality. A Guide for Building Owners and Facility Managers*. U.S. Environmental Protection Agency, 12 1991.
- [8] K. K. R. Indonesia, “Peraturan menteri kesehatan republik indonesia nomor 1077/menkes/per/v/2011,” 2011.
- [9] K. Madani, R. Hidayati, and U. Ristian, “Sistem update firmware perangkat iot menggunakan teknik ota berbasis http,” *JURIKOM (Jurnal Riset Komputer)*, vol. 9, 8 2022.
- [10] S. Kiresova, M. Guzan, and P. Galajda, “Measuring particulate matter (pm) using sps30.” Institute of Electrical and Electronics Engineers Inc., 2022.
- [11] P. Gäbel, C. Koller, and E. Hertig, “Development of air quality boxes based on low-cost sensor technology for ambient air quality monitoring,” *Sensors*, vol. 22, 5 2022.
- [12] R. M. F. Z. E. H. Partaningrat, “Rancang bangun sistem monitor kualitas udara dalam ruangan berbasis system-on-chip esp32,” 2019.
- [13] N. R. Wibowo, “Rancang bangun sistem detektor konsentrasi karbon dioksida (co2) dan particulate matter (pm2.5 & pm10 untuk sistem pemantauan lingkungan ruang huni,” 2022.

- [14] C.-M. Liu, "Effect of pm2.5 on aqi in taiwan," pp. 29–37, 2002. [Online]. Available: [www.elsevier.com/locate/envsoft](http://www.elsevier.com/locate/envsoft)
- [15] S. V. D. Elshout, K. Léger, and H. Heich, "Caqi common air quality index - update with pm2.5 and sensitivity analysis," *Science of the Total Environment*, vol. 488-489, pp. 461–468, 8 2014.
- [16] B. Oktara, "Hubungan antara kualitas fisik udara dalam ruang (suhu dan kelembaban relatif) dengan kejadian sick building syndrome (sbs) pada pegawai kantor pusat perusahaan jasa konstruksi x di jakarta timur tahun 2008," *Skripsi Universitas Indonesia*, 2008.
- [17] N. Azizah, "Paparan particulate matter (pm2.5 dan pm10) dan kejadian berat badan lahir rendah di kota makassar," *Tesis Universitas Hasanuddin*, 2015. [Online]. Available: [http://digilib.unhas.ac.id/uploaded\\_files/temporary/DigitalCollection/MzZiZTM5MTJjYTNkYzJINzcxZTJkYmU1NGIzMzkzNjU5NWQ5ZA==.pdf](http://digilib.unhas.ac.id/uploaded_files/temporary/DigitalCollection/MzZiZTM5MTJjYTNkYzJINzcxZTJkYmU1NGIzMzkzNjU5NWQ5ZA==.pdf)
- [18] D. L. Fajri, "Mengenal pm 2.5 dan pm 10, partikel berbahaya bagi tubuh artikel ini telah tayang di katadata.co.id dengan judul "mengenal pm 2.5 dan pm 10, partikel berbahaya bagi tubuh", <https://katadata.co.id/intan/berita/615177e7d841c/mengenal-pm-25-dan-pm-10-partikel-berbahaya-bagi-tubuh> penulis: Dwi latifatul fajri editor: Intan," 9 2021. [Online]. Available: <https://katadata.co.id/intan/berita/615177e7d841c/mengenal-pm-25-dan-pm-10-partikel-berbahaya-bagi-tubuh>
- [19] X. Wang, "Optical particle counter (opc) measurements and pulse height analysis (pha) data inversion," 2002.
- [20] S. Orley and J. Mathes, "A quick summary of ieee 754 notation;," 10 2000. [Online]. Available: <http://class.ece.iastate.edu/arun/cpre305/ieee754/homepage.html>
- [21] Sensirion, "Datasheet sps30 particulate matter sensor for air quality monitoring and control," 2020.
- [22] ——, "Sensor specification statement how to understand specifications of sensirion particulate matter sensors," 2020.
- [23] J. Fraden, "Handbook of modern sensors physics, designs, and applications fifth edition," 2016.
- [24] P. K. Mondal, "Accuracy & precision and conceptualisation to estimation of measurement uncertainty in quantitative analysis of quality control testing of petroleum products," 2012. [Online]. Available: <https://www.researchgate.net/publication/306515421>
- [25] S. Systech, "Solomon systech semiconductor technical data ssd1306 128 x 64 dot matrix oled/plcd segment/common driver with controller," 2008. [Online]. Available: <http://www.solomon-systech.com>
- [26] D. PAZRIYAH, "Penggunaan raspberry pi dalam mendeteksi warna melalui webcam," 2017.

- [27] R. Pi, “Raspberry pi 3 model b,” 2023. [Online]. Available: <https://www.raspberrypi.com/products/raspberry-pi-3-model-b/>
- [28] STMicroelectronics, “Stm32 32-bit arm cortex mcus,” 2023. [Online]. Available: <https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html#overview>
- [29] H. Ashari, “Stm32 arm cortex-m sebagai media pembelajaran mikrokontroler,” 2018. [Online]. Available: <http://eprints.uny.ac.id/62566/1/2.%20Hasim%20Ashari%2014502244005%20A.pdf>
- [30] STMicroelectronics, “Bluetooth 5 and 802.15.4 nucleo pack including usb dongle and nucleo-64 with stm32wb55 mcus, supports arduino uno v3 and st morpho connectivity,” 2023. [Online]. Available: <https://www.st.com/en/evaluation-tools/p-nucleo-wb55.html>
- [31] ——, “Stm32cubeide. integrated development environment for stm32.” 2023. [Online]. Available: <https://www.st.com/en/development-tools/stm32cubeide.html>
- [32] K. Chakraborty, “Firmware,” 9 2022. [Online]. Available: <https://www.techopedia.com/definition/2137/firmware>
- [33] tutorialspoint.com, “C programming tutorial.” [Online]. Available: [https://www.unf.edu/~wkloster/2220/ppts/cprogramming\\_tutorial.pdf](https://www.unf.edu/~wkloster/2220/ppts/cprogramming_tutorial.pdf)
- [34] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall, 1988.
- [35] IEC, *INTERNATIONAL STANDARD Floating-point arithmetic IEEE Std 754™*, 2019. [Online]. Available: [www.iso.org](http://www.iso.org)
- [36] Ayusharma, “Ieee standard 754 floating point numbers,” 3 2020. [Online]. Available: <https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/>
- [37] L. Chaparro, “Integer numbers storage in computer memory,” 8 2019. [Online]. Available: <https://medium.com/@luischaparro/integer-numbers-storage-in-computer-memory-47af4b59009#:~:text=According%20to%20the%20storage%20size,range%20of%20the%20stored%20numbers.>
- [38] SFUPTOWNMAKER, “I2c,” 7 2013. [Online]. Available: <https://www.sparkfun.com/search/results?term=i2c#tutorials>
- [39] F. Surya, “I2c protokol,” 2007. [Online]. Available: <https://comp-eng.binus.ac.id/files/2014/05/Artikel-I2C-Protokol.pdf>
- [40] H. Adams, “Inter-intergrated circuit (i2c),” 2022. [Online]. Available: [https://vanhunteradams.com/Protocols/I2C/I2C.html#V.-Hunter-Adams-\(vha3@cornell.edu\)](https://vanhunteradams.com/Protocols/I2C/I2C.html#V.-Hunter-Adams-(vha3@cornell.edu))
- [41] S. Afzal, “I2c primer: What is i2c? (part 1),” 2018. [Online]. Available: <https://www.analog.com/en/technical-articles/i2c-primer-what-is-i2c-part-1.html>
- [42] i2c bus.org, “10 bit adressing,” 2022. [Online]. Available: <https://www.i2c-bus.org/addressing/10-bit-addressing/>

- [43] F. Baldassari, “I2c in a nutshell,” 1 2020. [Online]. Available: <https://interrupt.memfault.com/blog/i2c-in-a-nutshell#acknack>
- [44] I. J. Sidabutar, “Rancang bangun muatan roket berbasis smartphone dan penambahan algoritma permintaan data ulang jika terjadi packet loss,” 2016. [Online]. Available: [https://elib.unikom.ac.id/files/disk1/700/jbptunikompp-gdl-imranjautt-34963-2-unikom\\_i-2.pdf](https://elib.unikom.ac.id/files/disk1/700/jbptunikompp-gdl-imranjautt-34963-2-unikom_i-2.pdf)
- [45] E. Peña and M. G. Legaspi, “Uart: A hardware communication protocol understanding universal asynchronous receiver/transmitter,” 2020. [Online]. Available: <https://www.analog.com/media/en/analog-dialogue/volume-54/number-4/uart-a-hardware-communication-protocol.pdf>
- [46] O. Pro, “Realterm – serial terminal for embedded debugging: Guide,” 2019. [Online]. Available: <https://openlabpro.com/guide/realterm-guide/>
- [47] elinux, “Grabserial,” 2021. [Online]. Available: <https://elinux.org/Grabserial>
- [48] A. Ma’arif, *BAHASA PEMROGRAMAN PYTHON*, 2020.