# I2C timing analyses on Artemis / Apollo3

Different challenges with SMBus / I2C on Apollo3 . This document is investigating that.

## Table of Contents



**Disclaimer**

While the recorded timing is correct and reproducible, performing the pin -setting and clearing also add about 1.3us each time.

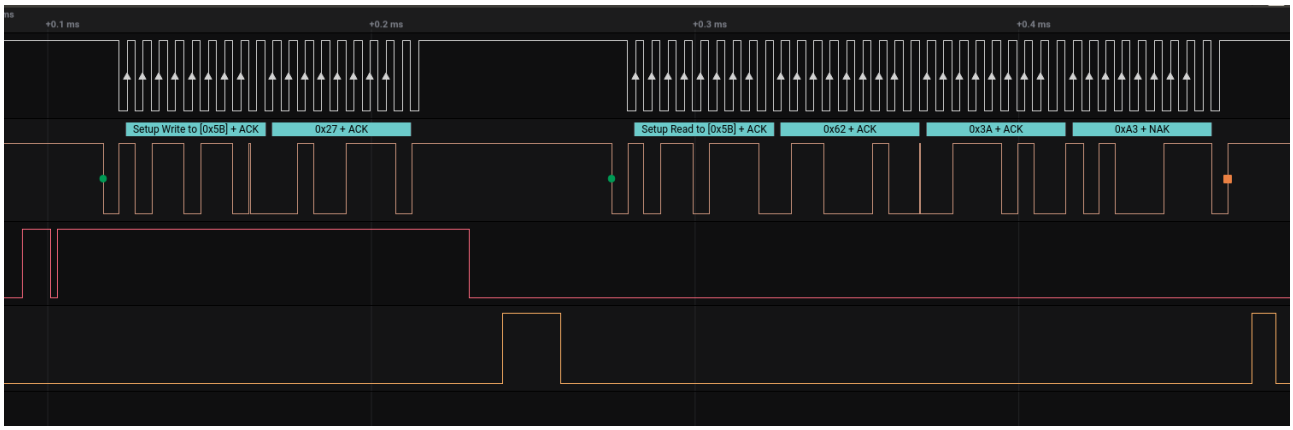
Don’t expect the timing behind the decimal point to be 10000% correct, but it is correct enough to support the conclusions.

I have used a sample sketch for the MLX90615 which is attached in Appendix A for all measurements. Also for the UNO.  In order to get it to work on the Apollo3 I had make changes to the https://github.com/Seeed-Studio/Digital_Infrared_Temperature_Sensor_MLX90615 and remove all the references to the propriety i2cMaster.cpp, which works on AVR only.

# V1 library performance

The following is taking from library version 1.2.1.

Line 1 is SCL (white), line 2 is SDA (orange).




The 3rd line, red line is showing the timing during write. A write address + register that we want to read in the next call. When set high first it is doing different checks, it then goes low writing the data to be sent (address + register) and returns high when the waiting for “send” command that has been written to the IO-Module (IOM) to complete. The IOM will start sending and the program then polls the data from status and when detected the IOM is completed/idle it returns low.

The 4th line, yellow line is showing timing during reading. This is reading the 3 bytes from the registers. When set high first it is doing different checks and it is set low when the “read” command has been written to the IOM. When the 3 bytes have been received it returns high. The program then polls the data from status and when detected the IOM is completed/idle it returns low.

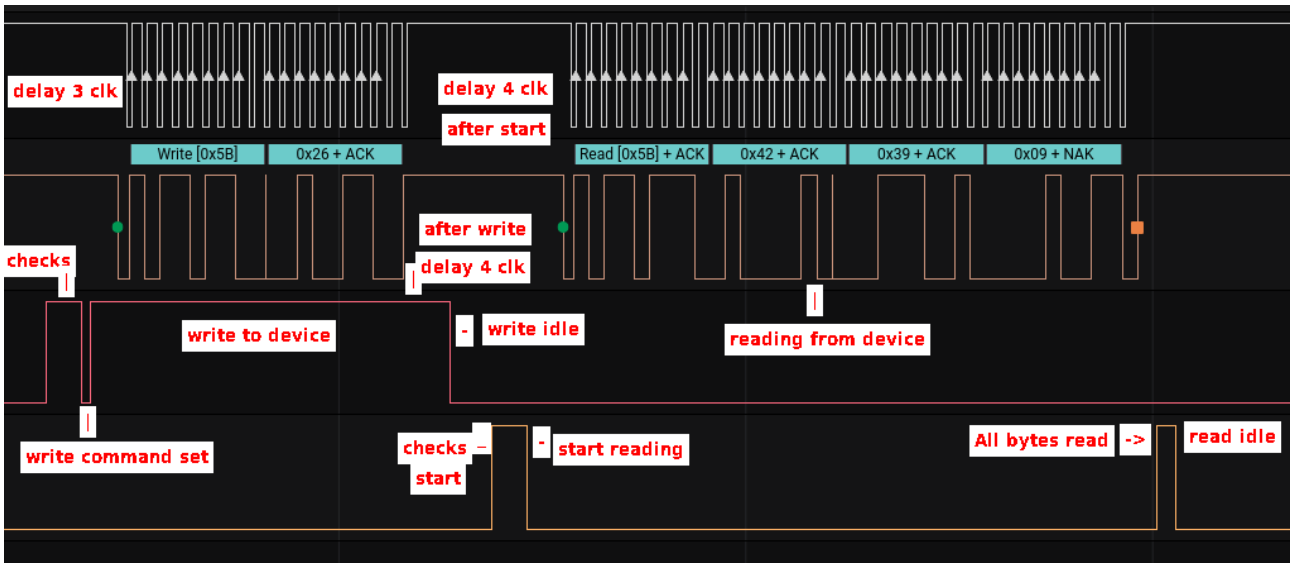
There are some interesting time delays, where the time delay is depending on the clock-speed of the SCL.  In general it takes 3 SCL clock-cycles from the moment a write command was set (the red line) and it will take 4 SCL clock-cycles from the moment a READ command was set (the yellow line).

This in real numbers :

| | 100KhZ | 200KHZ | 400KHZ |
|---|---|---|---|
| After write command set & before start SDA (start) | 30 us | 14 us | 7 us |
| Write command IDLE after last SCL | 30 us | 14 us | 10 us |
| Return from Wire.cpp after Write to Read | 10 us | 10 us | 10 us |
| Checks before Read submitting command | 8 us | 8 us | 8 us |
| After read command set & before start SCL | 40 us | 20 us | 10.5 us |
| Read command IDLE after last SDA (stop) | 12.8 us | 7 us | 4.3 us |
| | | | |
| SCL cycle time | 10 us | 5 us | 3.8 us |
| | | | |
| # SCL cycles after write command and SDA (start) | 3 | 3 | 2 |
| # SCL cycles after write command finished (Idle) | 3 | 3 | 3 |
| # SCL cycles after read command and start SCL | 4 | 4 | 4 |
| # SCL cycles after last read SDA (stop) and Idle | 1 | 1 | 1 |

There is NO configuration register has been found that can be set to reduce or impact this timing.

To show on a picture (taken at 400Khz)

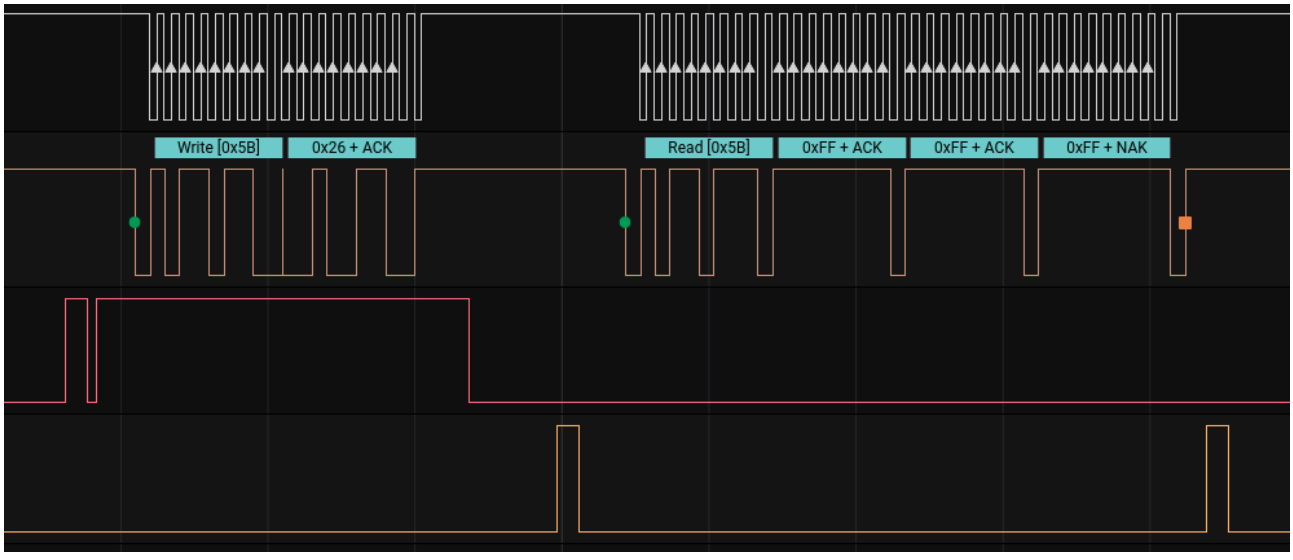


# V2 library performance

This measurement has been performed on library version 2.1.1.

The major difference is that V2 is build around a version of MBED (https://os.mbed.com/mbed-os/). It is based on a frozen version of MBED-5. Not only does this mean that more calls and code is between the application layer (Wire.cpp) and the hardware layer (HAL) functions, but also MBED brings overhead in performing a number of house keeping functions to enable “multi-tasking”. E.g. every microsecond Mbed is triggered to see whether there is something pending to do.

| | **100KhZ** | **200KHZ** | **400KHZ** |
|---|---|---|---|
| After write command set & before start SDA (start) | 25.5 us | 12 us | 6 us |
| Write command IDLE after last SCL | 32.2 us | 18 us | 14 us |
| Return from Wire.cpp after Write to Read | 60 us | 60 us | 60 us |
| Checks before Read submitting command | 15.2 us | 15.2 us | 15.2 us |
| After read command set & before start SCL | 40 us | 20 us | 11 us |
| Read command IDLE after last SDA (stop) | 24.4 us | 20.4 us | 22 us |
| | | | |
| SCL cycle time | 10 us | 5 us | 3.8 us |
| | | | |
| # SCL cycles after write command and SDA (start) | 3 | 3 | 2 |
| # SCL cycles after write command finished (Idle) | 3 | 3 | 3 |
| # SCL cycles after read command and start SCL | 4 | 4 | 4 |
| # SCL cycles after last read SDA (stop) and Idle | 3 | 4 | 5 |




Please note that the “read” is showing oxff for all 3 bytes. It is not that the device can not handle this speed (100Khz) as it provides ‘ack’ but because of the SMBus timing. See later chapter.

# Comparing V1 and V2

When comparing V1 and V2 keep in mind we are using exactly the same sketch, exactly the same hardware and timing taken around the same Apollo3 Hal-function.

## 100Khz frequency

| **100Khz** | **V1** | **V2** |
|---|---|---|
| After write command set & before start SDA (start) | 30 us | 25.5 us |
| Write command IDLE after last SCL | 30 us | 32.2 us |
| Return from Wire.cpp after Write to Read | 10 us | 60 us |
| Checks before Read submitting command | 8 us | 15.2 us |
| After read command set & before start SCL | 40 us | 40 us |
| Read command IDLE after last SDA (stop) | 12.8 us | 24.4 us |
| | | |
| SCL cycle time | 10 us | 10 us |
| | | |
| # SCL cycles after write command and SDA (start) | 3 | 3 |
| # SCL cycles after write command finished (Idle) | 3 | 3 |
| # SCL cycles after read command and start SCL | 4 | 4 |
| # SCL cycles after last read SDA (stop) and Idle | 1 | 3 |

You can see the overhead of MBED on the time:

Between Write of address + register happened and return to start reading. It is 6 x longer !
Performing the checks before reading is 2 x longer !

Also interesting that the time to becoming idle after the last SDA change after read (stop) takes about 2 times are much time. This done by polling the status register. While MBED could have an impact here, the same polling is performed waiting on the IOM to become idle after write address + register.

## 200Khz frequency

| **200Khz** | **V1** | **V2** |
|---|---|---|
| After write command set & before start SDA (start) | 14 us | 12 us |
| Write command IDLE after last SCL | 14 us | 18 us |
| Return from Wire.cpp after Write to Read | 10 us | 60 us |
| Checks before Read submitting command | 8 us | 15.2 us |
| After read command set & before start SCL | 20 us | 20 us |
| Read command IDLE after last SDA (stop) | 7 us | 20.4 us |
| | | |
| SCL cycle time | 5 us | 5 us |
| | | |
| # SCL cycles after write command and SDA (start) | 3 | 3 |
| # SCL cycles after write command finished (Idle) | 3 | 3 |
| # SCL cycles after read command and start SCL | 4 | 4 |
| # SCL cycles after last read SDA (stop) and Idle | 1 | 4 |

As with 100Khz you can see the overhead of MBED on the time:

Between Write of address + register happened and return to start reading. It is 6 x longer !
Performing the checks before reading is 2 x longer !

Also interesting that the time to becoming idle after the last SDA change after read (stop) takes about 3 times are much time. This done by polling the status register. While MBED could have an impact here, the same polling is performed waiting on the IOM to become idle after write address + register. On V1 there is a clear correlation between the SCL speed and this delay, but on V2 it is highly impacted by something else.

## 400Khz frequency

| **400Khz** | **V1** | **V2** |
|---|---|---|
| After write command set & before start SDA (start) | 7 us | 6 us |
| Write command IDLE after last SCL | 10 us | 14 us |
| Return from Wire.cpp after Write to Read | 10 us | 60 us |
| Checks before Read submitting command | 8 us | 15.2 us |
| After read command set & before start SCL | 10.5 us | 11 us |
| Read command IDLE after last SDA (stop) | 4.3 us | 22 us |
| | | |
| SCL cycle time | 3.8 us | 3.8 us |
| | | |
| # SCL cycles after write command and SDA (start) | 2 | 2 |
| # SCL cycles after write command finished (Idle) | 3 | 3 |
| # SCL cycles after read command and start SCL | 4 | 4 |
| # SCL cycles after last read SDA (stop) and Idle | 1 | 5 |

As with 100Khz and 200Khz you can see the overhead of MBED on the time:

Between Write of address + register happened and return to start reading. It is 6 x longer !
Performing the checks before reading is 2 x longer !

Also here the time to becoming idle after the last SDA change after read (stop) takes about 5 times are much time on V2 than V1. This done by polling the status register. While MBED could have an impact here, the same polling is performed waiting on the IOM to become idle after write address + register. On V1 there is a clear correlation between the SCL speed and this delay, but on V2 it is highly impacted by something else.

# Compare to I2C on Arduino Uno

Just to compare I have taken an Arduino Uno and programmed the sketch using library version 1.8.3

Included is the timing for 100KHZ. On the Uno there as no noticeable different in the **delay** timing mentioned whether running on 100Khz or 400Khz. The only place where a noticeable differences were found "After read command set & before start SCL" which is 1.7uS on 400Khz and of course the transfer was faster .

| **100KHZ** | **V1** | **V2** | **UNO** |
|---|---|---|---|
| After write command set & before start SDA (start) | 30 us | 25.5 us | 2 us |
| Write command IDLE after last SCL | 30 us | 32.2 us | 10.5 us |
| Return from Wire.cpp after Write to Read | 10 us | 60 us | 5.4 us (*1) |
| Checks before Read submitting command | 8 us | 15.2 us | 13.5 us (*2) |
| After read command set & before start SCL | 40 us | 40 us | 5.5 us (*3) |
| Read command IDLE after last SDA (stop) | 12.8 us | 30 us | 10.2 us |
| | | | |
| SCL cycle time | 10uS | 10uS | 10uS |
| | | | |
| # SCL cycles after write command and SDA (start) | 3 | 3 | |
| # SCL cycles after write command finished (Idle) | 3 | 3 | |
| # SCL cycles after read command and start SCL | 4 | 4 | |
| # SCL cycles after last read SDA (stop) and Idle | 1 | 3 | |

*1 : While the Uno processor is much less power-full and slower than an Apollo, the software in the library is working mostly directly on the hardware without a lot of calls and hand-offs.

*2 : The check before read submitting command is longer. That is because on an UNO the I2C interface is interrupt driven and in case this is a repeated start (which it is) it needs to take extra care on no previous pending interrupts.

*3 : on 400Khz this timing was 1.7uS for an Uno

When comparing V1 and V2 keep in mind we are using exactly the same code, exactly the same hardware. The Arduino library works directly on the hardware registers BUT is amazing to see the differences in the delay time !!

# SMBus specification

Some devices, like MLX90614 and  MLX90615, are based on the SMBus specifications. (http://smbus.org/specs/smbus20.pdf).

*The System Management Bus (SMBus) is a two-wire interface through which various system component  chips can communicate with each other and with the rest of the system. It is based on the principles of  operation of I2C*

SMBus is defining strict timing around the interaction of Master and Slave.  or Controller and peripheral in new terms to be more political correct.

| Symbol | Parameter | Limits | | Units | Comments |
|---|---|---|---|---|---|
| | | Min | Max | | |
| FSMB | SMBus Operating Frequency | 10 | 100 | KHz | See note 1 |
| TBUF | Bus free time between Stop and Start Condition | 4.7 | - | µs | |
| THD:STA | Hold time after (Repeated) Start Condition. After this period, the first clock is generated. | 4.0 | - | µs | |
| TSU:STA | Repeated Start Condition setup time | 4.7 | - | µs | |
| TSU:STO | Stop Condition setup time | 4.0 | - | µs | |
| THD:DAT | Data hold time | 300 | - | ns | |
| TSU:DAT | Data setup time | 250 | - | ns | |
| TTIMEOUT | Detect clock low timeout | 25 | 35 | ms | See note 2 |
| TLOW | Clock low period | 4.7 | - | µs | |
| THIGH | Clock high period | 4.0 | 50 | µs | See note 3 |
| TLOW: SEXT | Cumulative clock low extend time (slave device) | - | 25 | ms | See note 4 |
| TLOW: MEXT | Cumulative clock low extend time (master device) | - | 10 | ms | See note 5 |
| TF | Clock/Data Fall Time | - | 300 | ns | See note 6 |
| TR | Clock/Data Rise Time | - | 1000 | ns | See note 6 |
| TPOR | Time in which a device must be operational after power-on reset | | 500 | ms | See section 3.1.4.2 |

**Note 1:** The minimum frequency for synchronizing device clocks is defined in section 4.3.3. A master shall not drive the clock at a frequency below the minimum FSMB. Further, the operating clock frequency shall not be reduced below the minimum value of FSMB due to periodic clock extending by slave devices as defined in section 4.3.3. This limit does not apply to the bus idle condition, and this limit is independent from the TLOW: SEXT and TLOW: MEXT limits.

For example, if the SMBCLK is high for THIGH,MAX, the clock must not be periodically stretched longer than 1/FSMB,MIN – THIGH,MAX. This requirement does not pertain to a device that extends the SMBCLK low for data processing of a received byte, data buffering and so forth for longer than 100us in a non-periodic way.

**Note 2:** Devices participating in a transfer can abort the transfer in progress and release the bus when any single clock low interval exceeds the value of TTIMEOUT,MIN. After the master in a transaction detects this condition, it must generate a stop condition within or after the current data byte in the transfer process. Devices that have detected this condition must reset their communication and be able to receive a new START condition no later than TTIMEOUT,MAX. Typical device examples include the host controller, and embedded controller and most devices that can master the SMBus. Some simple devices do not contain a clock low drive circuit; this simple kind of device typically may reset its communications port after a start or a stop condition.

A timeout condition can only be ensured if the device that is forcing the timeout holds the SMBCLK low for TTIMEOUT,MAX or longer.

**Note 3:** THIGH,MAX provides a simple guaranteed method for masters to detect bus idle conditions. A master can assume that the bus is free if it detects that the clock and data signals have been high for greater than THIGH,MAX.

**Note 4:** TLOW:SEXT is the cumulative time a given slave device is allowed to extend the clock cycles in one message from the initial START to the STOP. It is possible that, another slave device or the master will also extend the clock causing the combined clock low extend time to be greater than TLOW:SEXT. Therefore, this parameter is measured with the slave device as the sole target of a full-speed master.

**Note 5:** TLOW:MEXT is the cumulative time a master device is allowed to extend its clock cycles within *each byte* of a message as defined from START-to-ACK, ACK-to-ACK, or ACK-to-STOP. It is possible that a slave device or another master will also extend the clock causing the combined clock low time to be greater than TLOW:MEXT on a given byte. Therefore, this parameter is measured with a full speed slave device as the sole target of the master.

# What is the impact on SMBus compliance?

The SMBus states that after 50us it assumes the bus is idle / free and resets. (see Note 3)

This timing is very important for the time AFTER writing the address + register and STARTING to read the data from that register.

## Impact with Apollo3 V1.X

After the address + register has been written it takes 3 clock cycles before it is detected that writing has been completed.

After setting the read command, it takes 4 clock cycles before SCL is starting to enable reading.

In total 7 clock cycles delay.  On 100kHz that is 7 * 10us = 70us, on 200kHZ that is 35uS and on 400Khz that is 20.5uS.

**This means that SMBus timing is missed already on 100Khz and a SMBus compatible device is failing.**

Not only the delay should be taken into account. In an application you first write the address + register. On return you do a requestFrom() call to get the data. The time between returning from write and requesting the “read” takes 10us programming time. Before performing the “read” command a number of checks are done to make sure the right parameters and conditions are in place. This takes 8us programming time. In total 18us.

Already ruling out 100Khz.

**With 200kHz** we are at 35us (delay) + 18us (programming time) = 53us.
A device MIGHT just accept this…..and please also note that setting / clearing a level on a debug-pin takes 1.3us. **In either case we are JUST at the boundary.**

On 400Khz we are at 20.5us (delay) + 18us (programming time) = 38.5us. This means we are good, BUT only if the device is supporting 400Khz,

## Impact with Apollo3 V2.X

We can be very short here… **will not work**. The time after writing the address +register and returning to read (requestFrom() ) is already 60us.  The 50us SMBus time-out will happen..

On 100Khz the time between idle after writing and starting read is 147us. (32 + 60 + 15 + 40). Even if we could optimize the 60us, it is still taking too long,

## Impact with Uno

NO problem at all as the timing takes about 30us

# Appendix A: Sketch

```
#include "MLX90615.h"
#define MLX MLX90615_DefaultAddr
MLX90615 mlx90615(MLX, &Wire);

void setup() {
   Serial.begin(115200);
   while (!Serial); // Only for native USB serial
   delay(2000); // Additional delay to allow open the terminal to see setup() messages
   Serial.println("Setup...");

   Wire.begin();
   Wire.setClock(100000);

   if (mlx90615.isConnected())  Serial.println("we are connected\r");
   else  Serial.println("we are NOT connected\r");
}
void loop() {
   Serial.print("Object temperature: ");
   Serial.println(mlx90615.getTemperature(MLX90615_OBJECT_TEMPERATURE));
   Serial.print("Ambient temperature: ");
   Serial.println(mlx90615.getTemperature(MLX90615_AMBIENT_TEMPERATURE));
   delay(1000);
}
```