Application Notes

I^{2}C Throughput

1. TABLE OF CONTENTS

2. SUMMARY

2.1 Overview

This application note covers I^{2}C performance testing of the USEQ*S devices, from the QFS, QGS and QMS series. The report explains the important data throughput related considerations involved in I^{2}C communications design for these sensors. The focus is on the collection of data within the 1ms frame time (i.e. maximum sampling rate). The question is finding the theoretical limit of the I^{2}C bus on fast mode plus 1 mbit/s and comparing it to the real-life implementation. The USEQ*S sensors are compatible with this I^{2}C setting.

2.2 Performance Results

Configuration	Test	Result
I^{2}C throughput - theoretical calculation	Reading a 5-channel frame of data per millisecond	**4.5 frames**
I^{2}C throughput – API measurement	Reading a 5-channel frame of data per millisecond	**3 frames**

DOCUMENT HISTORY

Version	Date	Change Ref.	Change Details
01	11 JUN 2020	N/A	First Release

3. I²C PERFORMANCE ANALYSIS

3.1 General Considerations

To establish how long it takes for a command to be sent to a USEQ*S sensor in order to request a full frame of data we need to consider the cost of an operation on the I²C bus. In general, a simple transaction will use a certain number of clock cycles in the following manner [1]

2 clocks for the start condition

1 clock for the stop condition

9 clocks for the address

9 clocks per byte sent or received.

The beginning of a transaction between the MCU and the sensor will require 2+1+9+9 = 21 clocks.

A typical FIFO_READ_ACTIVE operation will recover from the sensor a packet of sampled data of a maximum of 17 bytes: 3 bytes per enabled channel plus 2 bytes for the frame counter.

The fast mode plus I²C runs at 1 mbit/s or 1000 kHz at top speed. I²C speed varies according to the length of the transaction. A short transaction will not achieve the maximum I²C average throughput; a long transaction will get closer to the nominal.

3.2 I²C Calculation

3.2.1 I²C Time Cost

For this calculation the number of clock periods required for each operation is considered and the number of bytes required for completing the transaction.

Operation	Bytes	Content
setup write	1	sensor address
write command	1	FIFO active code
setup read	1	sensor address
read response	17	data
setup write	1	sensor address
write command	1	FIFO clear code
setup read	1	sensor address
read response	1	result code
Bytes per transaction	24	

From the table we see that a complete transaction to read a full packet of data from the sensor requires 24 bytes travelling to and from, between MCU and a USEQ*S sensor with all active channels collecting data at the maximum rate of 1ms.

[1] Gosh, A. (2012, november 28). *I²C communications speed across sensors*. Retrieved from Electrical Engineering: https://electronics.stackexchange.com/questions/50011/i2c-communication-speed-across-sensors.

3.2.2 I^{2}C Throughput

Each byte transmitted or received requires 9 clocks to be delivered, and then we have:

9 * 24 = 216 clocks.

In I^{2}C each operation is marked by a beginning and an end using the bus clock. Two clocks are required to start and one to stop.

FIFO active read: start + stop = 3 clocks

FIFO clear : start + stop = 3 clocks

The total numbers of clock cycles required therefore is: 216 + 6 = 222 clock cycles.

Our theoretical I^{2}C clock runs at 1,000,000 clocks per second, during 1ms we have only 1,000 clocks available, so:

1,000 available / 222 required per packet read = **4.50 frames per millisecond.**

These 4.50 frames contain information of 20 channels spread across 4 USEQ*S sensors. **Based on this calculation we could expect to read a maximum of 20 channels per millisecond.**

3.3 I^{2}C on the Messaging Application

Using the messaging application (API) to configure a control board for modules (USEQCSK0000000) with four sensors using five channels each, the logic analyser was used to capture the transaction.

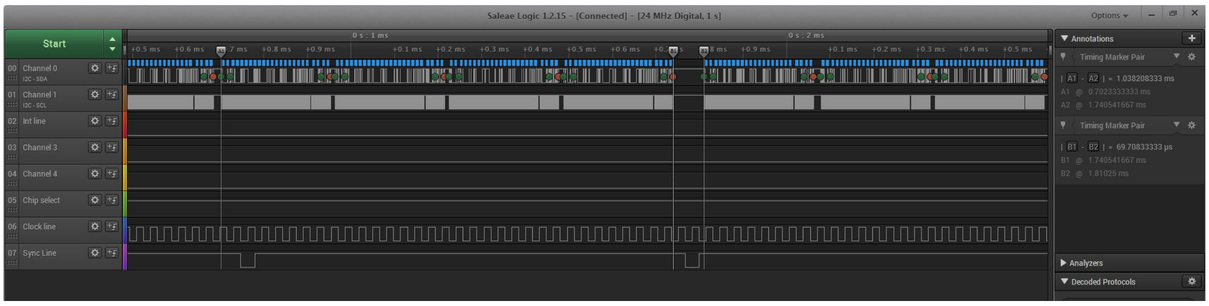


Figure 1 – Logic Analyser Output at Maximum Throughput Captured Data

Figure 1 shows a segment of the I^{2}C communication between the USEQ*S sensor and the MCU. A pair of markers has been set between the beginning and the end of reading a block of four sensors. The logic analyser shows an elapsed time of 1.03 milliseconds to complete the operation. The gap between operations is 0.07 milliseconds. The average clock speed for the I^{2}C, reported by the analyser is 888.9 KHz.

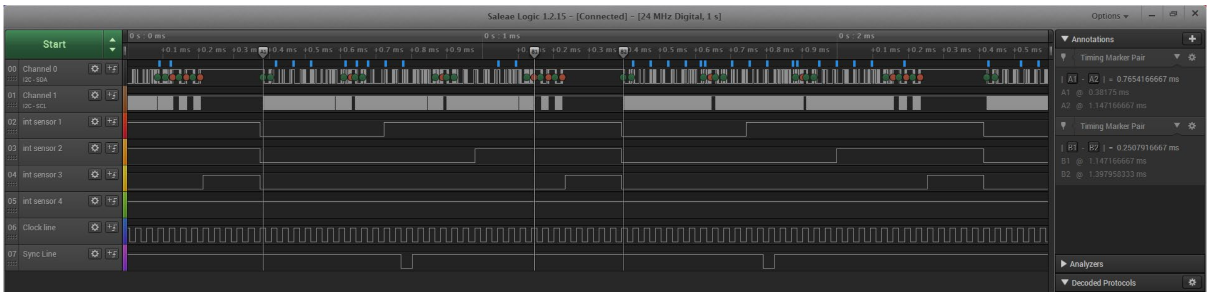
However, the GUI application reports a data rate of 464 Hz. When the data is collected into a CSV file, half of the packets of the sampling are lost. Due to the processing overhead that takes place between the firmware and the GUI and the time the next sample data becomes available.

To achieve the best compromise between number of sensors and maximum number of channels sampled, several configurations of number of active channels and number of sensors where tested.

By changing the configuration of the firmware and observing the data rate reported by the USEQ*S GUI 15 channels could be sampled with the following distribution:

Sensor	Channels enabled
1	5
2	5
3	5
4	0

The collected data is shown on the logic analyser below.



| A1 - A2 | = 0.7654166667 ms
A1 @ 0.38175 ms
A2 @ 1.147166667 ms

| B1 - B2 | = 0.2507916667 ms
B1 @ 1.147166667 ms
B2 @ 1.397958333 ms

Figure 2 – Logic Capture with no Frames Lost on the GUI

The first maker reports 0.76 milliseconds for the 15 channels read. Two false reads take place that the firmware execute as part of the normal execution loop, when there is no data on the sensors. The last marker shows the gap between reads, when no data is available, which amounts to 0.25 milliseconds.

3.4 Conclusion: Maximum Throughput

From the perspective of collecting data with no frames lost and transmitting this data to the existing GUI and the current USEQ*S API based firmware can record data from **15 channels across 3 sensors**.

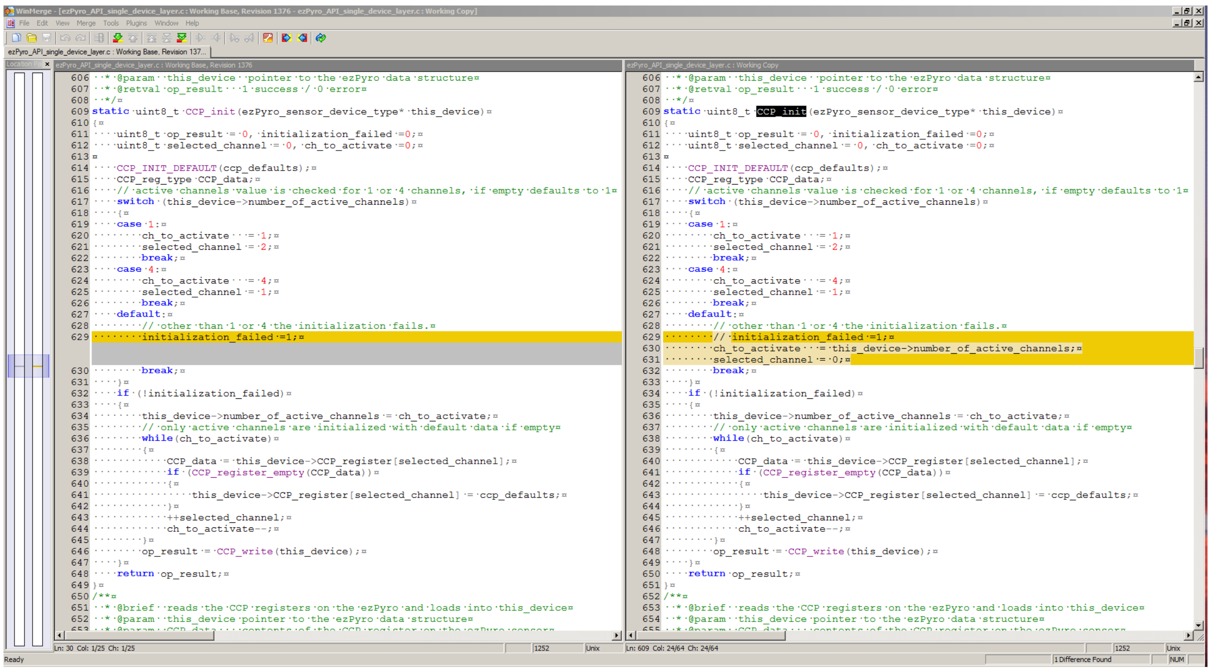
If a mechanism to store the collected data in a separate thread is used, then the **theoretical maximum of 20 channels across 4 sensors could be achieved.**

There is an overhead due to normal execution of the firmware that limits the number of channels that can be collected without affecting data collection from the user perspective. How much can this be improved by optimizing the structure of the current firmware could be investigated if required.

4. TEST SETUP

Data was collected using Saleae logic analyser, with the I²C decoder set to 7-bit addresses [2].

Special modification to the API and the messaging application.



WinMerge - [ezPyro_API_single_device_layer.c : Working Base, Revision 1376 - ezPyro_API_single_device_layer.c : Working Copy]

ezPyro_API_single_device_layer.c : Working Base, Revision 1376

```c
  * @param  this_device  pointer to the ezPyro data structure
  * @retval op_result   1 success / 0 error
  */
static uint8_t CCP_init(ezPyro_sensor_device_type* this_device)
{
    uint8_t op_result = 0, initialization_failed =0;
    uint8_t selected_channel = 0, ch_to_activate =0;

    CCP_INIT_DEFAULT(ccp_defaults);
    CCP_reg_type CCP_data;
    // active channels value is checked for 1 or 4 channels, if empty defaults to 1
    switch (this_device->number_of_active_channels)
    {
    case 1:
        ch_to_activate   = 1;
        selected_channel = 2;
        break;
    case 4:
        ch_to_activate   = 4;
        selected_channel = 1;
        break;
    default:
        // other than 1 or 4 the initialization fails.
        initialization_failed =1;


        break;
    }
    if (!initialization_failed)
    {
        this_device->number_of_active_channels = ch_to_activate;
        // only active channels are initialized with default data if empty
        while(ch_to_activate)
        {
            CCP_data = this_device->CCP_register[selected_channel];
            if (CCP_register_empty(CCP_data))
            {
                this_device->CCP_register[selected_channel] = ccp_defaults;
            }
            ++selected_channel;
            ch_to_activate--;
        }
        op_result = CCP_write(this_device);
    }
    return op_result;
}
/**
  * @brief  reads the CCP registers on the ezPyro and loads into this_device
  * @param  this_device pointer to the ezPyro data structure
```

Ln: 30 Col: 1/25 Ch: 1/25 | 1252 | Unix

ezPyro_API_single_device_layer.c : Working Copy

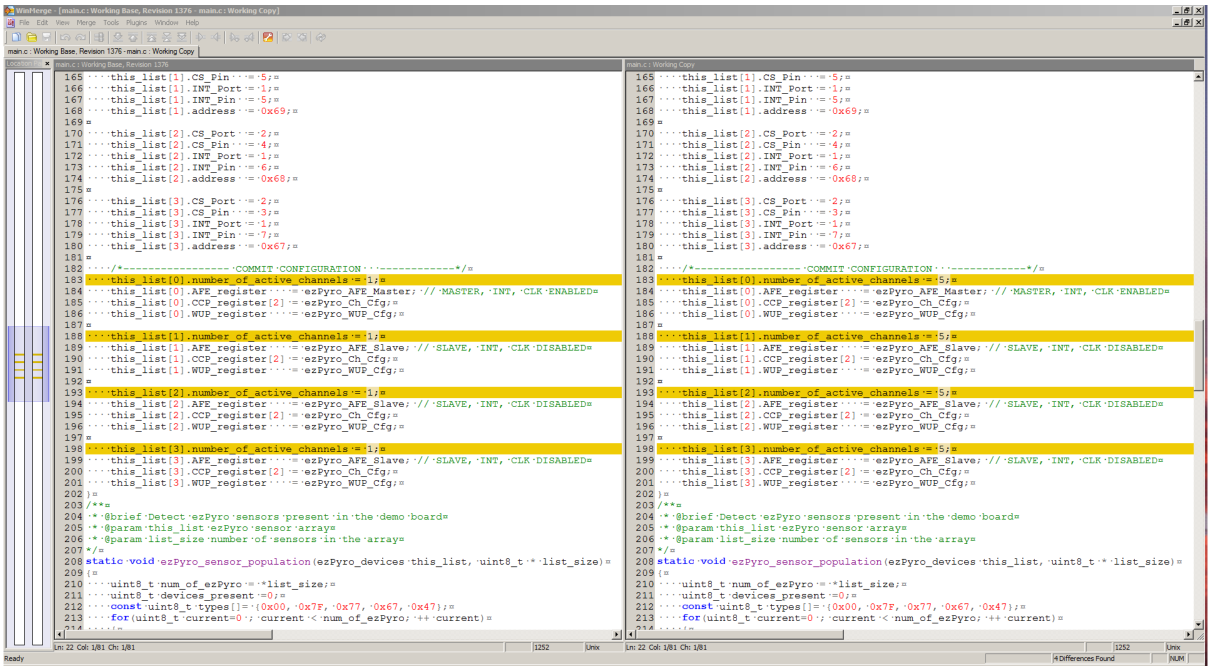
```c
  * @param  this_device  pointer to the ezPyro data structure
  * @retval op_result   1 success / 0 error
  */
static uint8_t CCP_init(ezPyro_sensor_device_type* this_device)
{
    uint8_t op_result = 0, initialization_failed =0;
    uint8_t selected_channel = 0, ch_to_activate =0;

    CCP_INIT_DEFAULT(ccp_defaults);
    CCP_reg_type CCP_data;
    // active channels value is checked for 1 or 4 channels, if empty defaults to 1
    switch (this_device->number_of_active_channels)
    {
    case 1:
        ch_to_activate   = 1;
        selected_channel = 2;
        break;
    case 4:
        ch_to_activate   = 4;
        selected_channel = 1;
        break;
    default:
        // other than 1 or 4 the initialization fails.
        // initialization_failed =1;
        ch_to_activate   = this_device->number_of_active_channels;
        selected_channel = 0;
        break;
    }
    if (!initialization_failed)
    {
        this_device->number_of_active_channels = ch_to_activate;
        // only active channels are initialized with default data if empty
        while(ch_to_activate)
        {
            CCP_data = this_device->CCP_register[selected_channel];
            if (CCP_register_empty(CCP_data))
            {
                this_device->CCP_register[selected_channel] = ccp_defaults;
            }
            ++selected_channel;
            ch_to_activate--;
        }
        op_result = CCP_write(this_device);
    }
    return op_result;
}
/**
  * @brief  reads the CCP registers on the ezPyro and loads into this_device
  * @param  this_device pointer to the ezPyro data structure
```

Ln: 609 Col: 24/64 Ch: 24/64 | 1252 | Unix

Ready | 1 Difference Found | NUM

Figure 3 – CCP_init() Modification



WinMerge - [main.c : Working Base, Revision 1376 - main.c : Working Copy]

main.c : Working Base, Revision 1376

```c
    this_list[1].CS_Pin   = 5;
    this_list[1].INT_Port = 1;
    this_list[1].INT_Pin  = 5;
    this_list[1].address  = 0x69;

    this_list[2].CS_Port  = 2;
    this_list[2].CS_Pin   = 4;
    this_list[2].INT_Port = 1;
    this_list[2].INT_Pin  = 6;
    this_list[2].address  = 0x68;

    this_list[3].CS_Port  = 2;
    this_list[3].CS_Pin   = 3;
    this_list[3].INT_Port = 1;
    this_list[3].INT_Pin  = 7;
    this_list[3].address  = 0x67;

    /*----------------- COMMIT CONFIGURATION  ------------*/
    this_list[0].number_of_active_channels = 1;
    this_list[0].AFE_register    = ezPyro_AFE_Master; // MASTER, INT, CLK ENABLED
    this_list[0].CCP_register[2] = ezPyro_Ch_Cfg;
    this_list[0].WUP_register    = ezPyro_WUP_Cfg;

    this_list[1].number_of_active_channels = 1;
    this_list[1].AFE_register    = ezPyro_AFE_Slave; // SLAVE, INT, CLK DISABLED
    this_list[1].CCP_register[2] = ezPyro_Ch_Cfg;
    this_list[1].WUP_register    = ezPyro_WUP_Cfg;

    this_list[2].number_of_active_channels = 1;
    this_list[2].AFE_register    = ezPyro_AFE_Slave; // SLAVE, INT, CLK DISABLED
    this_list[2].CCP_register[2] = ezPyro_Ch_Cfg;
    this_list[2].WUP_register    = ezPyro_WUP_Cfg;

    this_list[3].number_of_active_channels = 1;
    this_list[3].AFE_register    = ezPyro_AFE_Slave; // SLAVE, INT, CLK DISABLED
    this_list[3].CCP_register[2] = ezPyro_Ch_Cfg;
    this_list[3].WUP_register    = ezPyro_WUP_Cfg;
}
/**
 * @brief Detect ezPyro sensors present in the demo board
 * @param this_list ezPyro sensor array
 * @param list_size number of sensors in the array
*/
static void ezPyro_sensor_population(ezPyro_devices this_list, uint8_t * list_size)
{
    uint8_t num_of_ezPyro = *list_size;
    uint8_t devices_present =0;
    const uint8_t types[]= {0x00, 0x7F, 0x77, 0x67, 0x47};
    for(uint8_t current=0 ; current < num_of_ezPyro; ++ current)
```

Ln: 22 Col: 1/81 Ch: 1/81 | 1252 | Unix

main.c : Working Copy

```c
    this_list[1].CS_Pin   = 5;
    this_list[1].INT_Port = 1;
    this_list[1].INT_Pin  = 5;
    this_list[1].address  = 0x69;

    this_list[2].CS_Port  = 2;
    this_list[2].CS_Pin   = 4;
    this_list[2].INT_Port = 1;
    this_list[2].INT_Pin  = 6;
    this_list[2].address  = 0x68;

    this_list[3].CS_Port  = 2;
    this_list[3].CS_Pin   = 3;
    this_list[3].INT_Port = 1;
    this_list[3].INT_Pin  = 7;
    this_list[3].address  = 0x67;

    /*----------------- COMMIT CONFIGURATION  ------------*/
    this_list[0].number_of_active_channels = 5;
    this_list[0].AFE_register    = ezPyro_AFE_Master; // MASTER, INT, CLK ENABLED
    this_list[0].CCP_register[2] = ezPyro_Ch_Cfg;
    this_list[0].WUP_register    = ezPyro_WUP_Cfg;

    this_list[1].number_of_active_channels = 5;
    this_list[1].AFE_register    = ezPyro_AFE_Slave; // SLAVE, INT, CLK DISABLED
    this_list[1].CCP_register[2] = ezPyro_Ch_Cfg;
    this_list[1].WUP_register    = ezPyro_WUP_Cfg;

    this_list[2].number_of_active_channels = 5;
    this_list[2].AFE_register    = ezPyro_AFE_Slave; // SLAVE, INT, CLK DISABLED
    this_list[2].CCP_register[2] = ezPyro_Ch_Cfg;
    this_list[2].WUP_register    = ezPyro_WUP_Cfg;

    this_list[3].number_of_active_channels = 5;
    this_list[3].AFE_register    = ezPyro_AFE_Slave; // SLAVE, INT, CLK DISABLED
    this_list[3].CCP_register[2] = ezPyro_Ch_Cfg;
    this_list[3].WUP_register    = ezPyro_WUP_Cfg;
}
/**
 * @brief Detect ezPyro sensors present in the demo board
 * @param this_list ezPyro sensor array
 * @param list_size number of sensors in the array
*/
static void ezPyro_sensor_population(ezPyro_devices this_list, uint8_t * list_size)
{
    uint8_t num_of_ezPyro = *list_size;
    uint8_t devices_present =0;
    const uint8_t types[]= {0x00, 0x7F, 0x77, 0x67, 0x47};
    for(uint8_t current=0 ; current < num_of_ezPyro; ++ current)
```

Ln: 22 Col: 1/81 Ch: 1/81 | 1252 | Unix

Ready | 4 Differences Found | NUM

Figure 4 – main.c Modification

[2] Saleae. (2018, January 16). *Learn I²C - Inter-Integrated Circuit*. Retrieved from Saleae.com: https://support.saleae.com/tutorials/learning-portal/learning-resources/learn-i2c#learn-i-2-c-inter-integrated-circuit