

HEALTH MONITORING SYSTEM

Venetia Furtado

Final Project Report
ECEN 5613 Embedded System Design
University of Colorado, Boulder
May 7, 2024

Contents

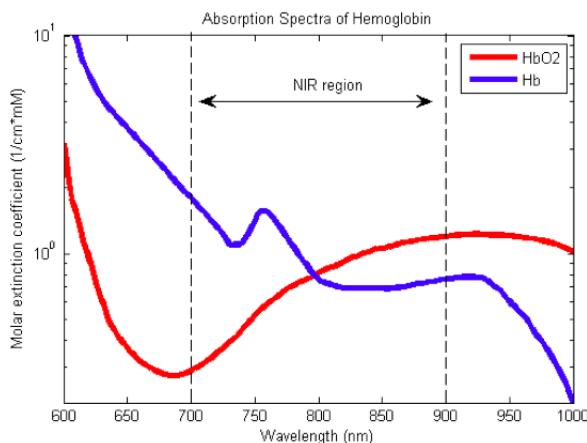
1	Introduction	1
1.1	Basic principles of Operation	1
1.2	Ambient Light Cancellation	2
2	Technical Description	2
2.1	Board Design	2
2.2	New Hardware Description	3
2.2.1	MAX30101WING	3
2.2.2	Voltage Regulator	5
2.2.3	Logic Level Translator	5
2.2.4	Pulse oximeter module by SparkFun (MAX30101 & MAX32664)	6
2.2.5	Graphical LCD	6
2.3	Firmware Design	7
2.3.1	First In First Out (FIFO) implementation(MAX30101):	7
2.3.2	Graphical LCD Device Driver	8
2.3.3	Bresenham's Line Algorithm Implementation	10
2.4	Testing Process & Error Analysis & Result	10
2.4.1	Unit Testing	10
2.4.2	Integration Testing	13
2.5	Result	13
3	Conclusion	14
4	Future Development Ideas	14
5	Acknowledgment	14
6	Appendices	15
6.0.1	Appendix - Bill of Materials	15
6.0.2	Appendix - Schematics	15
6.0.3	Appendix - Firmware Source Code (8051)	17
6.0.4	Appendix - Firmware Source Code (ARM)	97
6.0.5	Appendix - Firmware Source Code (MAXIM Algorithm)	146
6.0.6	Appendix - Firmware Source Code (Makefile)	159
6.0.7	Appendix - Software Source Code	162
6.0.8	Appendix - Datasheets and Application Notes	166

1 Introduction

Measuring oxygen saturation and heart rate is essential for continuous monitoring of our health. In this context, The MAX30101 sensor stands out in its application to be used for both SP02(oxygen) and heart rate measurements . This report puts forward the fundamental principles behind the working of a pulse oximeter and then dives into the detailed hardware implementation of MAX30101, the firmware and software design to capture the SpO2 and heart rate readings.

1.1 Basic principles of Operation

In very simple terms, a pulse oximeter is a device that clips onto a finger to measure the amount of oxygen level in the blood and heart rate of the user. Pulse oximetry is based on the principle that oxygenated blood (O_2Hb) and deoxygenated blood (HHb) absorb light differently. The pulse oximeter shines two types of lights - red light at 660 nm wavelength and infrared light at 940 nm wavelength as shown in Figure 1. Oxygenated blood absorbs higher amounts of IR light and lower amount of red light as compared to deoxygenated blood. On the other hand, deoxygenated blood absorbs lower amounts of IR light and higher amount of red light. This difference in the amount of light absorbed is used to measure the oxygen level in the blood (1) (6).



By Adrian Curtin (Own work) [CC BY-SA 3.0 (<http://creativecommons.org/licenses/by-sa/3.0>)], via Wikimedia Commons

Figure 1: Oxygenated and deoxygenated hemoglobin light absorption

The amount of light that is transmitted(or not absorbed) is captured by a photodiode which in turn feeds this signal to an ADC for it to process and give to the processor. The ratio of IR light and red light measurements (representing oxygenated and deoxygenated blood respectively) also called as R ratio ($R = (A_{red,AC}/A_{red,DC})/(A_{IR,AC}/A_{IR,DC})$) Figure 2 is what is used for the calculation of blood oxygen level (1). The ratio is converted to SPO₂ measurements by the processor using a look-up table based on the Beer-Lambert Law(1).

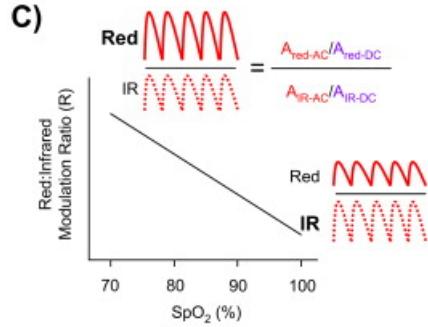


Figure 2: Calculation of R ratio (1)

1.2 Ambient Light Cancellation

The light intensity measured by the pulse oximeter includes both a pulsatile(AC) component caused by the pulsating blood through the arteries and steady(DC) component caused by ambient light and tissue absorption(Figure 3). The oximeter's signal processing algorithm separate the AC and DC components. In order to cancel the effect of ambient light, the sensor's algorithms continuously monitor the changes in the DC component caused by ambient light. This is done by periodically measuring the DC component during periods when the AC component is at its minimum (for example when the user is exhaling) and then using this as a baseline to subtract it from subsequent measurements (1).

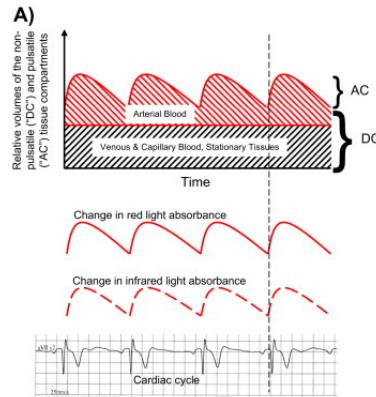


Figure 3: AC and DC components (light absorbance) by a pulse oximeter (1)

2 Technical Description

2.1 Board Design

As per the intial plan for this project, a MAX30101WING sensor was integrated with the Atmel 8051 MCU. Since the MAX30101 works on a 3.3V power supply, a 3.3V voltage level translator was used. This sensor also works on I2C communication protocol for which a 5V to 3.3V logic level converter was used. After facing significant challenges integrating this module with the 8051, a breakout board from SparkFun was used. This module was different from the MAX30101WING in that in addition to the MAX30101 sensor, it also had a sensor hub (MAX32664) which consisted of proprietary algorithm from the manufacturer to process the IR

LED and red LED values for the calculation of SpO₂ and heart rate. This module was initially integrated with the 8051, however due to challenges faced in this design as well, details of which will be discussed in a later section, it was successfully integrated with ARM STM32 board. The measured SpO₂ and heart rate values received from ARM were then transferred to the 8051 (via serial communication: a simple python script was written to transfer the data received at COM5 to COM4) and print it onto a 128x64 Graphical LCD. A block diagram of the board design has been shown in Figure 4.

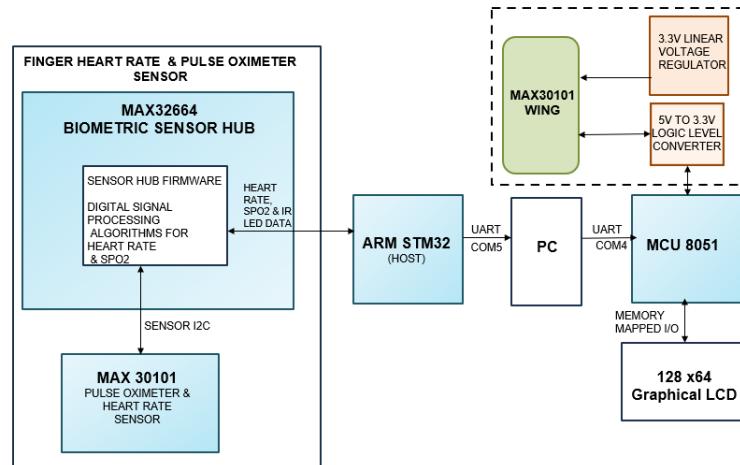


Figure 4: Board Design Block Diagram

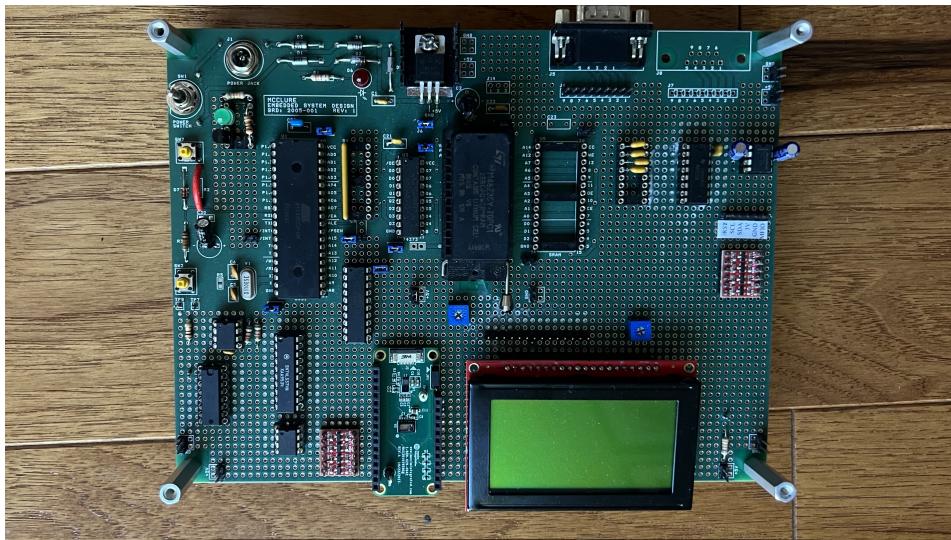


Figure 5: Board Design

2.2 New Hardware Description

2.2.1 MAX30101WING

The MAX30101 is an integrated pulse oximetry and heart-rate monitor module. It includes the IR and red LEDs, photodiode, and low-noise electronics with ambient light rejection. A block diagram has been shown in Figure 6 (3). When the user places their finger on the cover glass, the MCU signals the LED driver via

I²C communication protocol to activate the Red and IR LED at the programmed pulse width setting. The amount of light that has not been absorbed(or transmitted) is signaled to the photodiode which in turn feeds this signal to the ADC after canceling out the ambient light. The digital signal received at the output of the ADC is fed to the FIFO register. The implementation of this register is similar to that of a circular buffer. The host (ARM/8051) reads the FIFO register sequentially.

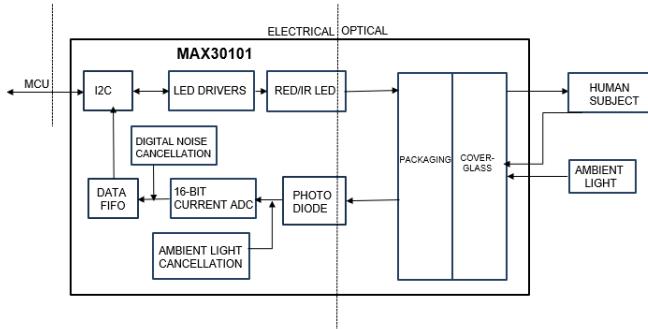


Figure 6: MAX30101 System Diagram (3)

In this particular module, the maximum wavelength for the IR LED is 900 nm and for the Red LED is 670 nm (3). The MAX30101 can fully be adjusted through software registers and the digital output of the ADC can be stored in a 32-deep FIFO within the IC. This FIFO allows the MAX30101 to be connected to a microcontroller. The MAX30101 has two modes of operation: the SpO₂ mode and the HR Mode. The difference between these two modes is that in SpO₂ mode both the IR light and red light values are received which can further be used to calculate the R value whereas HR mode only measure the IR light absorption value. The HR mode was not used in this project. All the information that follows is with respect to the operation in SpO₂ mode. The SpO₂ subsystem of the MAX30101 consists of the Ambient Light Cancellation (ALC), a continuous-time sigma-delta ADC and proprietary discrete time-filter. The ALC internally has a Track/Hold circuit which out ambient light. The ADC is programmable from 2 μ A to 16 μ A. The internal ADC has a 18-bit resolution. The maximum sample rate of the ADC depends on the selected pulse width, which in turn determines the ADC resolution. The MAX30101 LED driver consists of red, green and IR LEDs to modulate pulses for SpO₂ and Heart Rate(HR) measurements. The LED pulse width can be programmed between 69 μ s to 411 μ s which allows the algorithm to optimize the accuracy of the measured SpO₂ and HR and power consumption based on the application it is being used for (3). For this project, the pulse width selected was 118 μ s which can give upto a 1000 samples per second which in turn sets the ADC resolution to 16-bit. The MAX30101 also has an on-chip temperature sensor which wasn't used in this project.



Figure 7: MAX30101WING (Source: Digikey)

2.2.2 Voltage Regulator

In this project two implementations were attempted to build the pulse oximeter and heart rate monitor- the first was using the MAX30101WING and the second using the breakout board by SparkFun. Both these pulse oximeter modules ran at a lower voltage(3.3V) which was lower than any other device used with the 8051 during the semester. Therefore, a 5V to 3.3V voltage regulator (MAX604 in particular) was introduced on the board. The pin diagram of the MAX604 has been shown in Figure 8. The IN is the regulated 5V input from the voltage regulator for the 8051. /OFF when activated is used to switch off the LDO, which was not used in this project. The pre-selected output voltage (3.3V) of the MAX604 is set by connecting SET to ground. A stable 3.3V was received at the output. In order to provide stable operation, improved load-transient response and power-supply rejection 10 μ F capacitors were used in the input and output. The schematic of this voltage regulator circuit has been given in Section 6.0.2 2.

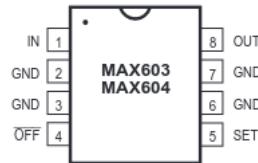


Figure 8: MAX604 pin diagram (2)

2.2.3 Logic Level Translator

A logic level translator was introduced in order to step down the voltage levels (to 3.3V)of the SCL, SDA lines for the I2C transaction and the EN and INT lines of the MAX30101WING that was connected to Port 1.5 and Port 3.4 of the ATMEL AT89C51RC2 respectively.

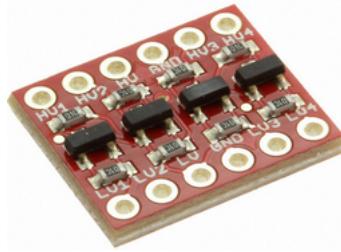


Figure 9: Logic Level Converter (Source: DigiKey)

2.2.4 Pulse oximeter module by SparkFun (MAX30101 & MAX32664)

This pulse oximeter module by SparkFun has a MAX32664 along with the MAX30101 sensor discussed in Section 2.2.1. The MAX32664 is a pre-programmed microcontroller with firmware drivers and proprietary algorithms. When combined with sensor devices like the MAX30101, the MAX32664 acts as a sensor hub and provides processed data to the host device(STM32 in this project). This module allows users to receive raw and/or calculated data seamlessly. The generic pin connections between the MAX32664 and the host has been shown in Figure 10. The detailed connections have been shown in the schematic in Section 6.0.2. The RSTN pin along with MFIO pin is used to control whether the MAX32664 starts in Bootloader mode or Application mode. The MAX32664 was used in application mode for most of this project. The host acts as an I2C master when it is communicating with the MAX32664.

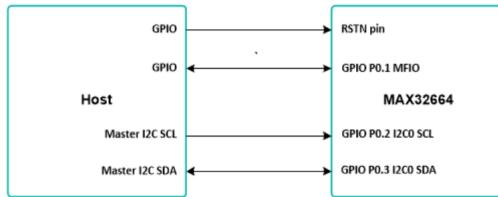


Figure 10: Pin connections between host and MAX32664 (4)

2.2.5 Graphical LCD

A 128x64 graphical LCD was added to the board using a male-female strip header pin similar to what was done in Lab 4. The Graphical LCD is a 20-pin module consisting of 8 data line (D0 - D7). The interface pin connections have been shown in Figure 11. More details of the pin connections has been shown in the schematic in Section 6.0.2. The Graphical LCD was memory-mapped in the address space 8000H to FFFFH. One mistake that was made in this particular hardware implementation was that the LCD was soldered upside-down to the board. This took some time to realize, since all the wire wrapping had been reversed which had to be re-done. After reading through the datasheet multiple times the error was caught while noticing the mechanical diagram that was provided in the datasheet.

Pin No.	Symbol	Level	Description
1	Vdd	5.0V	Supply voltage for logic and LCD (+)
2	Vss	0V	Ground
3	V0	-	Operating voltage for LCD (variable)
4-11	DB0-DB7	H/L	Data bit 0-7
12	CS2	L	Chip select signal for IC2
13	CS1	L	Chip select signal for IC1
14	/RES	L	Reset signal
15	R/W	H/L	H: read (MUP< - module),L: write (MPU ->module)
16	D/I	H/L	H: data, L: instruction code
17	E	H, H L	Chip enable signal
18	VEE	-	Operating voltage for LCD (variable)
19	A	4.2V	Backlight power supply
20	K	0V	Backlight power supply

Figure 11: Pin description of the Graphical LCD (9)

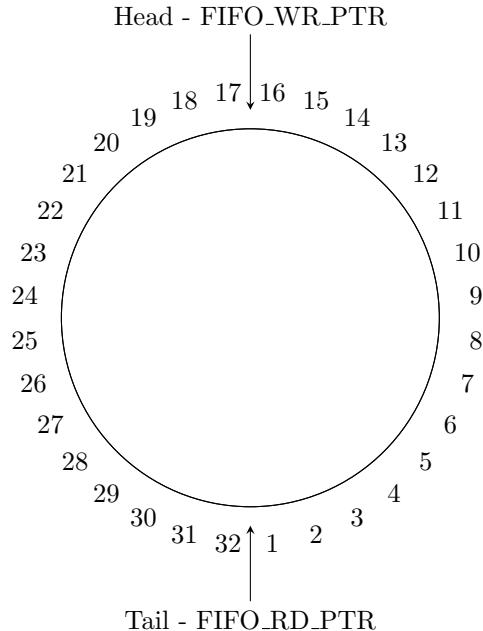
2.3 Firmware Design

2.3.1 First In First Out (FIFO) implementation(MAX30101):

The register map of the FIFO is shown in Figure 12. The implementation of the FIFO is based upon a circular buffer design explained as follows.

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
FIFO Write Pointer				FIFO_WR_PTR[4:0]				0x04	0x00	R/W	
Over Flow Counter				OVF_COUNTER[4:0]				0x05	0x00	R/W	
FIFO Read Pointer				FIFO_RD_PTR[4:0]				0x06	0x00	R/W	
FIFO Data Register	FIFO_DATA[7:0]				0x07	0x00	R/W				

Figure 12: FIFO register map (3)



The FIFO Write Pointer(FIFO_WR_PTR) points to the location of the next sample and advances for each sample pushed onto the FIFO. Therefore, it acts as the HEAD of the circular buffer. Similarly, the FIFO Read Pointer(FIFO_RD_PTR) points to the location from where the next sample is read. This advances

each time a sample is popped from the FIFO. Therefore, it acts as the TAIL of the circular buffer. The depth of the circular FIFO is 32, which means it can hold up to 32 samples of data. The sample size depends on the number of LED channels activated. Each channel signal is stored as a 3-byte data signal. The FIFO width can be 3 bytes, 6 bytes, 9 bytes or 12 bytes in size. In this project the FIFO width was 6 bytes since only two LED channels were activated (red and IR) as shown in Figure 13 (9).

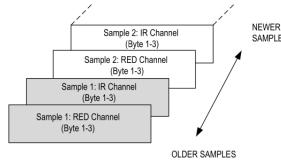


Figure 13: IR and Red LED sample in SpO2 Mode (3)

A pseudo-code of the implementation of the sample read transaction done on the 8051 is as follows.

Procedure: Read Sensor FIFO

1. Read FIFO Write Pointer(FIFO_WR_PTR) → HEAD
2. Read FIFO Read Pointer(FIFO_RD_PTR) → TAIL
3. if(HEAD > TAIL): Number of samples to read = HEAD - TAIL
4. if(HEAD < TAIL): Number of samples to read = (32 - TAIL)+HEAD
5. Number of bytes to read = (Number of samples to read) * (Size per channel) * (Number of LEDs)
6. Burst Read FIFO Data Register for the number of samples to read (6 in this project implementation)

2.3.2 Graphical LCD Device Driver

Graphical LCD screen layout:

The graphical LCD screen is divided into two halves (column number starting from 0 to 63 on each half): the controller $\overline{CS1}$ controls the left half and the controller $\overline{CS2}$ controls the right half of the screen (9). As mentioned in Section 2.2.5, the Graphical LCD was memory mapped from in the address space for peripheral devices(between 8000H to FFFFH). Table 1 shows the address mapping that was used for the different display instructions. In the Graphical LCD driver code, glcdWrite() was used to write data to the Graphical LCD and glcdRead() was used to read data from the Graphical LCD.

Table 1: Memory Mapping of the Graphical LCD

Function	$\overline{CS1}$ (A3)	$\overline{CS2}$ (A2)	R/W(A0)	RS(A1)	Status Address
Status Read	1	1	1	0	800E
Data Write(Left)	1	0	0	1	8009
Data Write(Right)	0	1	0	1	8005
Column Select(Left)	1	0	0	0	8008
Column Select(Right)	0	1	0	0	8004
Command Write	1	1	0	0	800C
Data Read(Left)	1	0	1	1	800B
Data Read(Right)	0	1	1	1	8007

Accessing pixels in the Graphical LCD:

The GDM12864HLCM LCD has a 128x64 resolution. The pixels are arranged in 128 columns (0 to 127) and 64 rows (0 to 63). The display configuration of the gLCD has been shown in Figure 14 . A memory block stores the value for each pixel on the screen. A 1 written to a pixel location darkens the screen and a 0 clears it. The left and right controller has control over 64 pixels (0 to 63 on each side). Vertically the screen is divided into 8 pages (Page 0 to Page 7) and each page consists of 8 pixels (9).

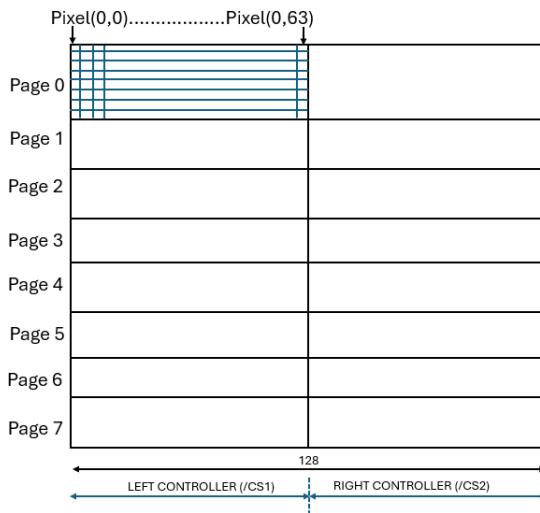


Figure 14: Graphical LCD screen display structure

In order to access a pixel within a page, the page address (0 to 7) and Y address(0 to 63) are set. In the LCD device driver code, the `setPixel()` function provides this functionality. The following pseudo-code explains this working.

Procedure: `setPixel(row, column, value)`

1. `row = 63 -row; // sets the Y-axis from Cartesian coordinate system`
2. `page = row/8; // finds the page number`
3. `pixelPosition = row%8; // gets the pixel position within the page`
4. `glcdGoToAddr(column,page); //moves the cursor to the corresponding column and page`
5. As per the datasheet, in order to read the contents of the display data RAM, the read instruction needs to be accessed twice. In the first read, the data in the display data RAM is latched into an internal output register of the LCD. In the second read the microcontroller can read this data that has been latched.
`pageByte = glcdRead(column);`
`pageByte = glcdRead(column);`
6. `if(value = 1) → pageByte = pageByte |(1 << pixelPosition);`
7. `if(value = 0) → pageByte = pageByte & (~((1 << pixelPosition)));`
8. `glcdWrite(column, page, pageByte); //Writes the pixel value to the column and page`

2.3.3 Bresenham's Line Algorithm Implementation

Bresenham's Line Algorithm is a line drawing algorithm used to draw straight lines on a grid, like a computer screen, by finding the approximate point between two points (8). In this project, it was used to plot the values of the IR LED which represent the pulsatile flow of blood representing the heart beat cycle. When plotting on the LCD screen all the IR values were scaled to fit between 0 and 32 on the y-axis of the LCD. The pseudo-code below explains the implementation of the line algorithm.

Procedure: Plot IR value

```

1. dx = x2 - x1; dy = y2 -y1;//gets absolute difference between (x2 ,x1 )and (y2,y1)
2. if(dx >dy)
   error = dx >> 1; //Sets error to approximately half of dx
   y =y1; //set y to y1
   for(x = x1; x <= x2; x++);//Iterates from x1 to x2
   error = error - dy;
   if(error < 0) → y = y+1; error = error + dx;
3. else
   error = dy >> 1; //Set error to approximately half of dy
   x = x1; //Set x to x1
   for(y = y1; y <= y2; y + +) //Iterates from y1 to y2
   error = error - dx;
   if(error < 0) → x = x+1; error = error + dy;
```

The line drawn between two points has a positive or negative slope depending on the dx and dy values and the error margin(error in the pseudo-code). When (dx >dy), the value of y changes only when the error goes negative, this is adjusted by incrementing y to the next point and adding dx to the error. Similarly, when (dy >dx) x is incremented only when when error goes negative and dy is added to the error value. An example of this code is shown in Table 2. In the example, x1 = 2, y1 = 7, x2 = 7, y2 = 9. Therefore, dx = 5 and dy = 2. The value of y can be seen changing only when error goes negative.

Table 2: Example of Bresenham's Line algorithm

x	y	Error
2	7	0
3	7	-2
4	8	1
5	8	-1
6	9	2
7	9	0

2.4 Testing Process & Error Analysis & Result

2.4.1 Unit Testing

MAX30101WING:

After the initialization code was written, a logic analyzer was used to verify the I2C read and write transaction. A screenshot of the I2C sequential read sequence is shown in Figure 15. The values of the IR LED

being read from the FIFO was then printed on the terminal. A simple python script was used to plot these values as shown in Figure 16.

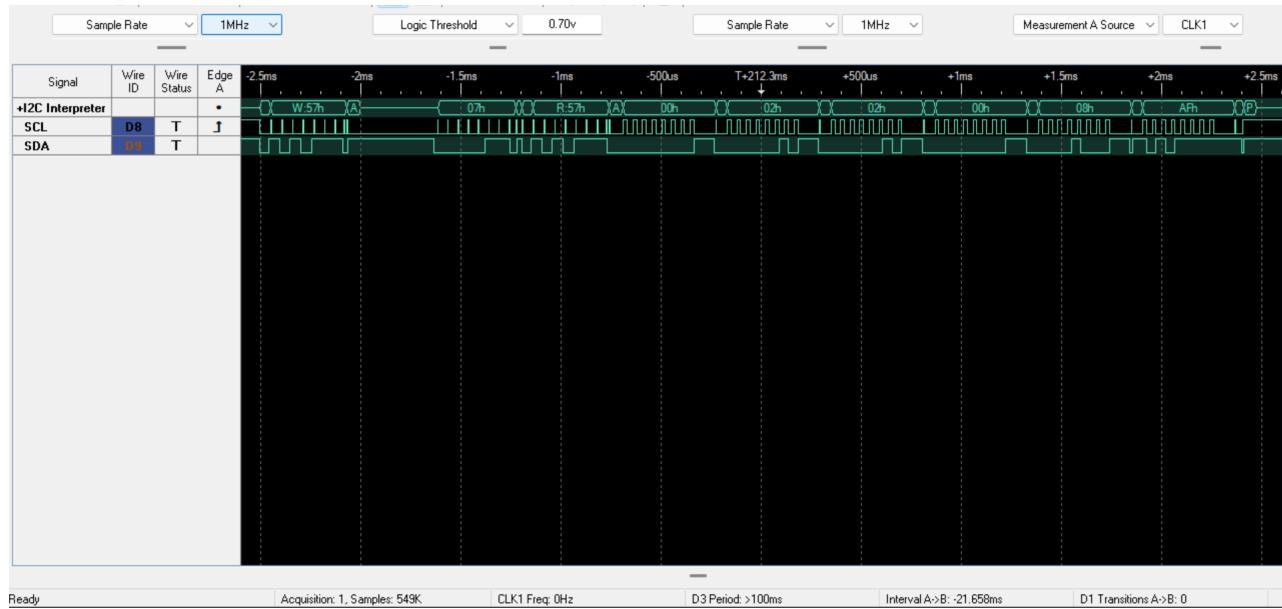


Figure 15: I2C sequential read transaction of the MAX30101WING

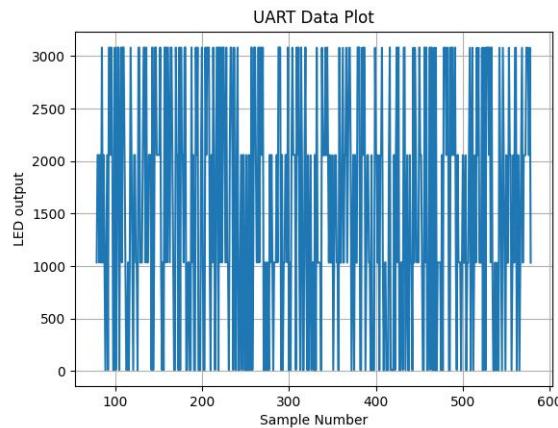
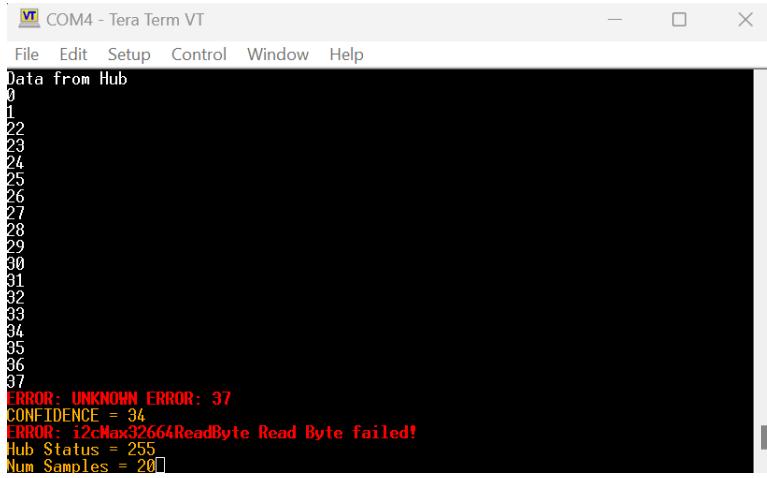


Figure 16: Plot of the IR LED values from MAX30101WING module

A reference algorithm from the manufacturer Maxim was also used to cross-verify if the values received could be used for the calculations of SpO₂ and Heart Rate. This algorithm calculated the values of SpO₂ to be 82% and heart rate value to be 122 which did not match the off-the-shelf oximeter which was being used to verify the readings.

MAX32664 sensor hub(breakout board by SparkFun):

Similar to the MAXWING module, after the initialization code for the MAX32664 sensor hub was written, a logic analyser was used to study the I2C read and write transactions. This board was initially integrated with the 8051 MCU. The data read from the FIFO of the sensor hub has been shown in Figure 17.



```

COM4 - Tera Term VT
File Edit Setup Control Window Help
Data from Hub
0
1
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
ERROR: UNKNOWN ERROR: 37
CONFIDENCE = 34
ERROR: i2cMax32664ReadByte Read Byte failed!
Hub Status = 255
Num Samples = 20
  
```

Figure 17: Values read from FIFO by 8051

Error 37 shown in Figure 17 was not listed in the datasheet. The status hub should have returned 0 if no errors were encountered which was not true in this case. The number of samples were 20 however the data was not valid. The expected values are shown in Figure 18. The data read should have been in the format shown in Figure 18.

SENSOR	SIZE(BYTES)	DESCRIPTION
WHRM/MaximFast	2	Heart Rate (bpm): 16-bit, LSB = 0.1 bpm
	1	Confidence level (0 - 100%): 8-bit, LSB = 1%
	2	SpO ₂ value (0 - 100%): 16-bit, LSB = 0.1%
	1	WHRM State Machine Status Codes MaximFast State Machine Status Codes:

Figure 18: Expected values from the sensor hub (4)

The MAX32664 sensor hub was then integrated with the STM32 board using the same initialization sequence. The logic analyzer capture of the read sequence has been shown in Figure 19. Here, the status hub returned 00 indicating no errors. The IR LED value was printed out on the terminal and the python plotter was used to compare with the plot from the MAX30101WING module (Figure 20).

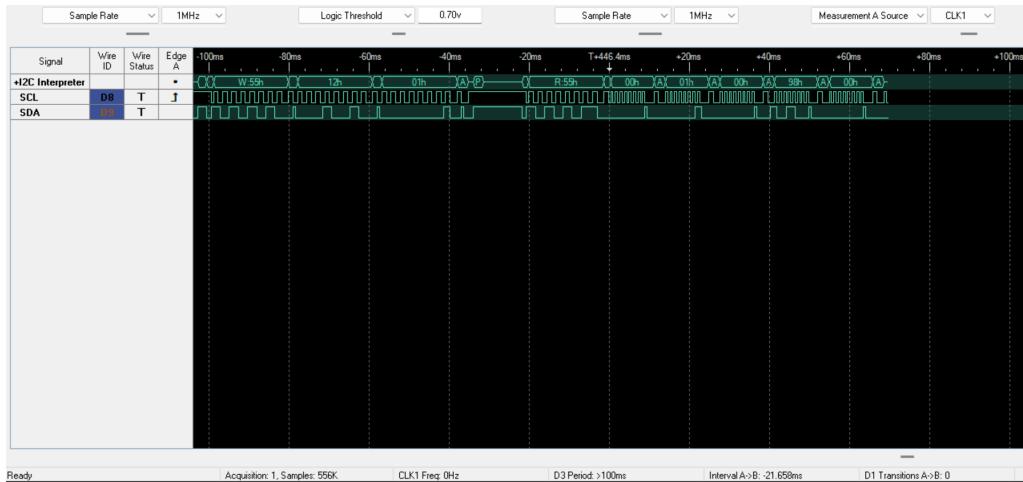


Figure 19: MAXHUB sequential read with STM32 as host

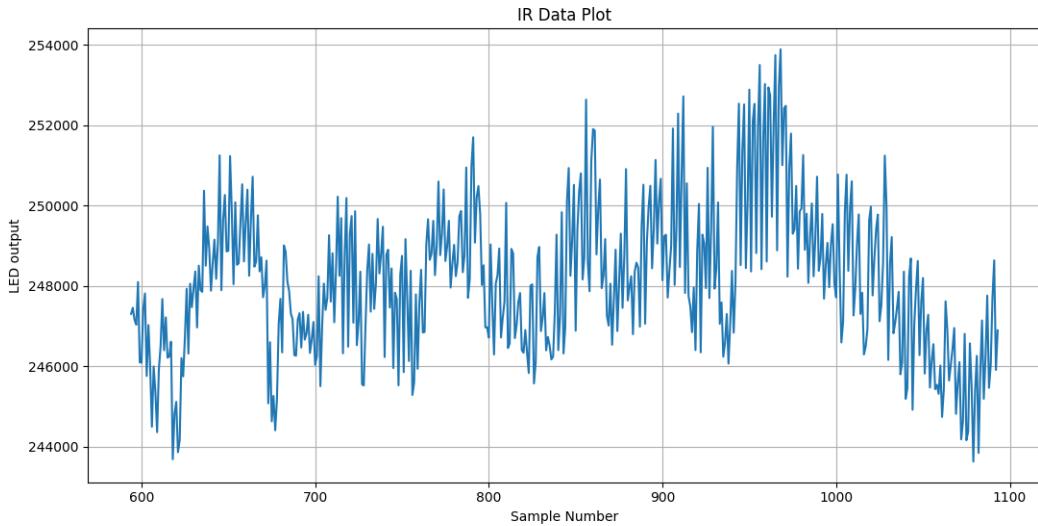


Figure 20: Plot of the IR LED values received from MAX32664 sensor hub

Graphical LCD testing:

A test function (`lcdTest()`) was written to test out the LCD. The LCD was tested in stages. First, `glcdWriteChar()` was used to write one character onto the screen. Following this `glcdWriteStr()` wrote a string on to the LCD. In order to test out plotting on the LCD screen `generateRandom()` was written to generate random values and plot those points on the screen.

2.4.2 Integration Testing

Since MAX30101 module did not yield any successful data, the integration testing was performed using the MAX32664 module. All the components used for this is shown in the block diagram Figure 4. The SpO₂, heart rate and IR LED values read by the STM32 board (COM5)was then transferred to the 8051 (COM4) via serial communication. A python script was written to achieve this. The data was then displayed onto the LCD screen and the graph of the IR LED values against number of samples was plotted.

2.5 Result

The SpO₂ and heart rate readings obtained from the MAX32664 sensor hub was the same as the reading observed on the off-the-shelf oximeter. There was a slight delay in obtaining the readings which was because the sampling rate was set slow because the 8051 was not able to read the incoming serial data at a fast rate.

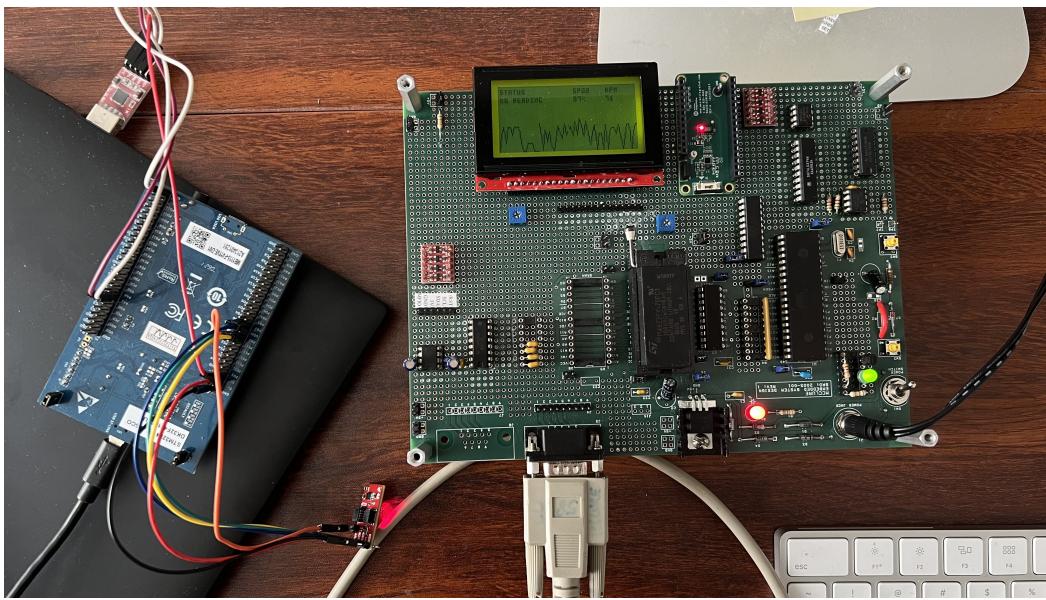


Figure 21: Integration of all components

3 Conclusion

In this project, while there initially were failed attempts to interface the MAX30101WING with the 8051 MCU, the SpO₂, heart rate and IR LED values were successfully calculated using the MAX32663 sensor hub. The key learnings in this project were:

- Understanding the calculations behind SpO₂ and heart rate readings in a pulse oximeter.
- Exploring the inner working of the algorithm was both time consuming and rewarding.
- Working of the FIFO register of the MAX30101.
- Tuning the pulse oximeter modules with the correct pulse width and sampling rate.
- Interfacing the Graphical LCD with the 8051 MCU and understanding the relation between page and pixels.
- Learning how Bresenham's Line Algorithm is used to plot graphs on a screen.

4 Future Development Ideas

One improvement of the project which would have been attempted if not for the limitation of time was to introduce a Real Time Clock(RTC) on the board for storing the readings and retrieving the last shown data with the date and time the user last used it. Another improvement would be to use a TFT LCD instead of the Graphical LCD used in this project so that the display would be more colorful and descriptive.

5 Acknowledgment

I would like to thank Prof. Linden McClure for offering a well-structured and rigorous course. My understanding of the subject, has certainly improved since the beginning of the semester. Additionally, I have also

developed debugging skills and gained valuable experience in reading datasheets, skills which I will surely carry forward with me.

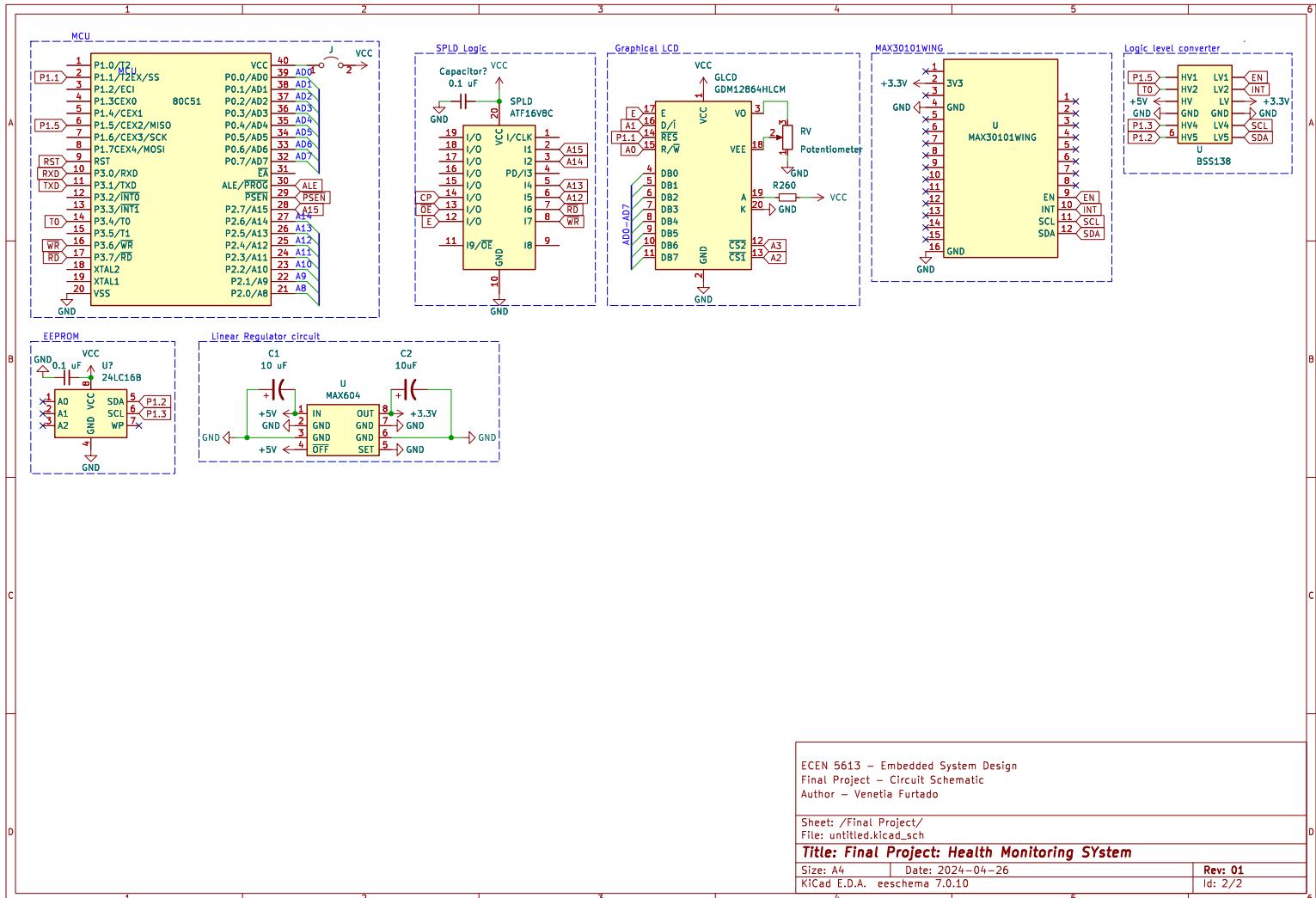
I would also like to extend my thanks to the TAs Amey More, Parth Kharade and Aneesh Deshpande for always being willing to help out and offer guidance with the course materials.

6 Appendices

6.0.1 Appendix - Bill of Materials

Part Description	Source	Cost
MAX30101WING	DigiKey	\$29.11
MAX604	DigiKey	\$7.33
BOB-12009 Logic level Converter	DigiKey	\$3.5
SEN-15219 Pulse Oximeter and Heart Rate Sensor	DigiKey	\$42.95
LCD-00710 Graphical LCD	Mouser Electronics	\$23.95
Jumper wires	ITLL	\$0.00

6.0.2 Appendix - Schematics



6.0.3 Appendix - Firmware Source Code (8051)

The following is the firmware source code for both the MAX30101WING module and the MAX32664.

```
*****  
* Author - Venetia Furtado  
* Final Project: Health Monitoring System  
* ECEN 5613 - Spring 2024 - Prof. McClure  
* University of Colorado Boulder  
* Revised 04/26/2024  
*  
*****
```

```
* This file contains the main function for the initialization of the MAHHUB and  
* the sensor modules
```

```
******/
```

```
*****/  
//           INCLUDES  
*****/
```

```
#include <mcs51/8051.h>  
#include "at89c51ed2.h"  
#include <stdio.h>  
#include <stdint.h>  
#include <stdlib.h>  
#include <stdbool.h>  
#include "init.h"  
#include "io.h"  
#include "log.h"  
#include "i2c.h"  
#include "lcd.h"  
#include "datareceiver.h"
```

```
//#define MANUAL_ENTRY_MODE 1
```

```
*****/  
//           FUNCTION DEFINITIONS & FORWARD DECLARATIONS  
*****/
```

```
**  
* @brief Main function contains function calls to get user input and evaluate inputs received  
*
```

```

* @return int
*/
int main()
{
    hardware_init();           // Function call to initialize hardware
    //lcdinit();
    lcdInit();
    //lcdTest();
    testMaxHub();

    //testMaxWing();

    while (1)
    {
        char input[20] = {0};
        for(uint8_t i = 0; i < 20; i++)
        {
            input[i] = '|';
        }

        uint8_t idx = 0;
#ifdef MANUAL_ENTRY_MODE
        INFO_LOG("Enter sample");
#endif
        while(1)
        {
            char c = getchar();

#ifdef MANUAL_ENTRY_MODE
            printf_tiny("%c", c);
#endif
            if (c == '\n' || c == '\r')
            {
                break;
            }

            input[idx] = c;
            idx++;
        }
    }
}

```

```

#ifndef MANUAL_ENTRY_MODE
    INFO_LOG("Printing data");
    for(int i = 0; i < idx; i++)
    {
        printf_tiny("%d ", input[i]);
    }
#endif

BioData data;
readBiodata(input, &data);
INFO_LOG("BPM= %d SPO2=%d\n\r", data.bpm, data.spo2);

glcdUpdateStatus(getStatusString(data.status));
if (data.status == FINGER_DETECTED)
{
    glcdUpdateSpo2(data.spo2);
    glcdUpdateBpm(data.bpm);
    glcdPlotIrValue(data.irled);
}
}

//testMaxWing();

return 0;
}

/*
 * Author - Venetia Furtado
 * Final Project: Health Monitoring System
 * ECEN 5613 - Spring 2024 - Prof. McClure
 * University of Colorado Boulder
 * Revised 04/18/2024
 *
 */

```

```

* This file contains the configuartion of the registers to setup the MAXHUB WING
* module to calculate the SPO2, heart rate and IR LED values.
* References: https://os.mbed.com/teams/Maxim-Integrated/code/MAX30205_Demo/
* https://os.mbed.com/teams/Maxim-Integrated/code/MAX30101WING_HR_SPO2/
***** */
/* **** */

// INCLUDES
/* */

#include <mcs51/8051.h>
#include "at89c51ed2.h"
#include "i2c.h"
#include "log.h"

#define PMIC_ENABLE      P1_5

// PMIC Registers
#define BBB_EXTRA_ADRS  0x1C
#define BOOST_VOLTAGE   0x05

#define NUM_LEDS         2 // 2 LEDs for SpO2 mode
#define SIZE_PER_CHANNEL 3 // (3 bytes per channel - LED or IR)
#define BUFFER_SIZE      32
#define MAX_FIFO_BYTES   NUM_LEDS *SIZE_PER_CHANNEL *BUFFER_SIZE // Max # Bytes in FIFO

/// MAX30101 Register Map
typedef enum
{
    InterruptStatus1 = 0x00,
    InterruptStatus2 = 0x01,
    InterruptEnable1 = 0x02,
    FIFO_WritePointer = 0x04,
    OverflowCounter = 0x05,
    FIFO_ReadPointer = 0x06,
    FIFO_DataRegister = 0x07,
    FIFO_Configuration = 0x08,
    ModeConfiguration = 0x09,
    SpO2Configuration = 0x0A,
    LED1_RED_PA = 0x0C,
    LED2_IR_PA = 0x0D,
}

```

```

ModeControlReg1 = 0x11,
ModeControlReg2 = 0x12,
} MAX30101Registers;

/*****************/
// FUNCTION DEFINITIONS & FORWARD DECLARATIONS
/*****************/

/**
 * @brief Logic to provide delay
 *
 * @param val
 */
void sensor_delay(uint16_t val)
{
    for (uint16_t i = 0; i < val; i++)
    ;
}

/**
 * @brief Logic for initialization of the onboard PMIC
 *
 */
void pmicInit()
{
    // enable passive pull down.
    writeRegisterMAXWing(PMIC, BBB_EXTRA_ADRS, 0x40);

    // set voltage to 4.5V
    PMIC_ENABLE = 0;
    writeRegisterMAXWing(PMIC, BOOST_VOLTAGE, 0x08);
    PMIC_ENABLE = 1;
}

/**
 * @brief Logic to write into a register and read from the same to cross-check if
 * correct value has been set by the host.
 * @param deviceId
 * @param regAddr

```

```

* @param regVal
* @return int
*/
int configureSensorRegister(uint8_t deviceId, uint8_t regAddr, uint8_t regVal)
{
    int status = writeRegisterMAXWing(deviceId, regAddr, regVal);
    if (status != SUCCESS)
    {
        ERROR_LOG("Write to Register %u Failed", regAddr);
        return FAIL;
    }

    sensor_delay(500);

    // check if register is set
    uint8_t val = readRegisterMAXWing(deviceId, regAddr);
    if (val != regVal)
    {
        ERROR_LOG("Register %d Expected %d and actual %d config does not match!", regAddr, regVal, val);
        return FAIL;
    }

    return SUCCESS;
}

/**
 * @brief Function containing the initialization sequence of the MAX30101
 *
 */
void sensorInit()
{
    int status;

    // reset configuration
    uint8_t modeConfig = 0;
    modeConfig |= 0x01 << 6; // Set RESET bit (B6) of register ModeConfiguration
    modeConfig |= 0x03; // Sets to SPO2 mode
    status = writeRegisterMAXWing(MAX30101, ModeConfiguration, modeConfig);
    if (status != SUCCESS)

```

```

{

    ERROR_LOG("1 Reset Conguration Failed");

    return;
}

sensor_delay(500);

#ifndef MAX30101_INTERRUPT_ENABLE_REG
#define MAX30101_INTERRUPT_ENABLE_REG 0x10
#endif

#ifndef MAX30101_FIFO_CONFIG_REG
#define MAX30101_FIFO_CONFIG_REG 0x11
#endif

#ifndef MAX30101_SPO2_CONFIG_REG
#define MAX30101_SPO2_CONFIG_REG 0x12
#endif

uint8_t interruptEnableRegister = 0;

//sends an interrupt if the FIFO of the MAX30101 is full
interruptEnableRegister |= 0x01 << 7; // A_FULL_EN

//sends an interrupt if a sample is available to be read
interruptEnableRegister |= 0x01 << 6; // PPG_RDY_EN

status = configureSensorRegister(MAX30101, InterruptEnable1, interruptEnableRegister);

if (status != SUCCESS)
{
    ERROR_LOG("2 Interrupt Enable Register Failed");

    // return;
}

#endif

// configure FIFO
uint8_t fifoConfig = 0;

fifoConfig |= 0x2;      // FIFO_A_FULL, interrupt when 30 samples in FIFO
//fifoConfig |= 0x01 << 4; // Fifo rollover enable
//fifoConfig |= 0x5 << 5; // Average out 32 samples
status = configureSensorRegister(MAX30101, FIFO_Configuration, fifoConfig);

if (status != SUCCESS)
{
    ERROR_LOG("3 FIFO Config Failed");

    // return;
}

// set spo2 config
uint8_t spo2Config = 0;

spo2Config |= 0x01 << 5; // sets resolution to 4096 nAfs
spo2Config |= 0x01 << 2; // sets sample rate; SpO2 SR = 100Hz
spo2Config |= 0x01;      // 16-bit ADC resolution ~118 us
status = configureSensorRegister(MAX30101, SpO2Configuration, spo2Config);

if (status != SUCCESS)
{
    ERROR_LOG("4 SpO2 Configuration Failed");
}

```

```

{

    ERROR_LOG("4 SpO2 Config Failed");

    // return;

}

#endif 0

uint8_t multiLedControlRegister2 = 0;

multiLedControlRegister2 |= 0x03; // Slot 3 = green LED

writeRegisterMAXWing(MAX30101, ModeControlReg2, multiLedControlRegister2);

if (status != SUCCESS)

{

    ERROR_LOG("5 Multi LED Control 2 Failed");

    return;

}

#endif

// Set LED drive currents

// Red LED; Pulse amp. = ~50mA (0xFF)

status = configureSensorRegister(MAX30101, LED1_RED_PA, 0xFF);

if (status != SUCCESS)

{

    ERROR_LOG("6 Red LED Drive Current Configuration Failed");

    // return;

}

// IR LED; Pulse amp. = ~50mA (0xFF)

status = configureSensorRegister(MAX30101, LED2_IR_PA, 0xFF);

if (status != SUCCESS)

{

    ERROR_LOG("7 IR LED Drive Current Configuration Failed");

    // return;

}

uint8_t multiLedControlRegister1 = 0;

multiLedControlRegister1 |= 0x01;    // Slot 1 = Red LED

multiLedControlRegister1 |= 0x02 << 4; // Slot 2 = IR LED

writeRegisterMAXWing(MAX30101, ModeControlReg1, multiLedControlRegister1);

if (status != SUCCESS)

{

```

```

        ERROR__LOG("5 Multi LED Control Failed");

    return;
}

#endif

// Resetting the write pointer to 0 as per datasheet
status = writeRegisterMAXWing(MAX30101, FIFO__WritePointer, 0);
if (status != SUCCESS)
{
    ERROR__LOG("8 Resetting Write Pointer Failed");
    return;
}
// Resetting the read pointer to 0 as per datasheet
status = writeRegisterMAXWing(MAX30101, FIFO__ReadPointer, 0);
if (status != SUCCESS)
{
    ERROR__LOG("9 Resetting Read Failed");
    return;
}
#endif

// Set operating mode
modeConfig = 0;
modeConfig |= 0x03; // Set SP02 mode
status = configureSensorRegister(MAX30101, ModeConfiguration, modeConfig);
if (status != SUCCESS)
{
    ERROR__LOG("10 Setting operating mode Failed");
    // return;
}

sensor_delay(500);

}

/***
 * @brief Functions to get the size of the circular buffer (FIFO)
 *
 * @param head
 * @param tail
 */

```

```

* @return uint8_t
*/
uint8_t getBufferSizeInBytes(uint8_t head, uint8_t tail)
{
    uint8_t numBytes = 0;

    // head > tail
    if (head > tail) // Head is greater than tail
    {
        numBytes = head - tail;
    }
    // tail > head
    else if (head < tail) // Tail is greater than head
    {
        numBytes = (BUFFER_SIZE - tail) + head;
    }
    else
    {
        numBytes = 0;
    }

    numBytes = numBytes * SIZE_PER_CHANNEL * NUM_LEDS;

    return numBytes;
}

/**
* @brief Function to read the number of Bytes in the FIFO
*
* @param fifoData
* @return int
*/
int readSensorFifo(uint8_t fifoData[])
{
    int result;

    // Get overflow counter
    int overflowCnt = readRegisterMAXWing(MAX30101, OverflowCounter);
    if (overflowCnt == FAIL)

```

```

{
    ERROR_LOG("%s: Reading overflow counter fail", __func__);
    // return FAIL;
}

INFO_LOG("Overflow Count = %d", overflowCnt);
sensor_delay(500);

// Get write pointer (head)
int writeIndex = readRegisterMAXWing(MAX30101, FIFO_WritePointer);
if (writeIndex == FAIL)
{
    ERROR_LOG("%s: Reading Write Index fail", __func__);
    // return FAIL;
}

INFO_LOG("Write ptr = %d", writeIndex);
sensor_delay(500);

// Get read pointer (tail)
int readIndex = readRegisterMAXWing(MAX30101, FIFO_ReadPointer);
if (readIndex == FAIL)
{
    ERROR_LOG("%s: Reading Read Index fail", __func__);
    // return FAIL;
}

INFO_LOG("Read ptr = %d", readIndex);
sensor_delay(500);

#if 0
// Calculate num bytes to read
uint8_t numBytes = getBufferSizeInBytes(writeIndex, readIndex);

INFO_LOG("%s: Numbytes = %u", __func__, numBytes);
if (numBytes == 0)
{
    INFO_LOG("Nothing to read");
    return 0;
}
#endif

// read FIFO, numBytes-> number of bytes to be read
result = readBytesMAXWing(MAX30101, FIFO_DataRegister, fifoData, 6);

```

```

if (result == FAIL)
{
    ERROR_LOG("%s: Reading FIFO fail", __func__);
    return FAIL;
}

return 6;
}

/***
 * @brief Gets the status of the interrupt registers
 *
 */
void getInterruptStatus()
{
    int status = readRegisterMAXWing(MAX30101, InterruptStatus1);
    INFO_LOG("Interrupt status1 = %d", status);
    status = readRegisterMAXWing(MAX30101, InterruptStatus2);
    INFO_LOG("Interrupt status2 = %d", status);
}

/***
 * @brief Test code for MAXWING working
 *
 */
void testMaxWing()
{
    pmicInit();
    sensorInit();

    __xdata uint8_t fifo[MAX_FIFO_BYTES];

    uint16_t counter = 0;

    while (1)
    {
        SECTION;
        uint8_t deviceID = readRegisterMAXWing(MAX30101, 0xFF);
        //INFO_LOG("Device ID: %d", deviceID);
    }
}

```

```

getInterruptStatus();

int nreadBytes = readSensorFifo(fifo);

//INFO_LOG("nReadBytes = %d", nreadBytes);
if (nreadBytes != FAIL && nreadBytes > 0)
{
    uint16_t ledRed = fifo[2];
    ledRed = (ledRed << 8) | fifo[1];

    uint16_t ledIr = fifo[5];
    ledIr = (ledIr << 8) | fifo[4];

/*
    if (ledIr == 0)
    {
        continue;
    }*/
}

INFO_LOG("%u,red = %u ir = %u", counter++,ledRed,ledIr);
/*
for (int i = 0; i < nreadBytes; i++)
{
    printf("i = %d, fifo[i] = %x \n\r", i, fifo[i]);
}*/
}

// delay
#if 1
    for (uint16_t i = 0; i < 100; i++)
{
    sensor_delay(500);
}
#endif
}

}

```

```

*****  

* Author - Venetia Furtado  

* Final Project: Health Monitoring System  

* ECEN 5613 - Spring 2024 - Prof. McClure  

* University of Colorado Boulder  

* Revised 04/26/2024  

*  

*****  

* This file contains the initialization and configuration of the MAXHUB module.  

* References:  

https://github.com/sparkfun/SparkFun\_Bio\_Sensor\_Hub\_Library/blob/master/src/SparkFun\_Bio\_Sensor\_Hub\_Library.cpp  

*  

******/  

*****/  

//           INCLUDES  

*****/  

#include <mcs51/8051.h>  

#include <stdint.h>  

#include "at89c51ed2.h"  

#include "i2c.h"  

#include "io.h"  

#include "log.h"  

#include "maxhub.h"  

#define MAX32664_ADDRESS 0b1010101 // 0b1010101  

#define CMD_DELAY      20  

#define RSTN P3_4  

#define MFIO P1_4  

#define XDATA __xdata  

#define STATUS_LOG  

*****/

```

```

//          FUNCTION DEFINITIONS & FORWARD DECLARATIONS
/********************************************/


/***
 * @brief Function to provide a hubDelayMs of timeMs
 *
 * @param timeUs
 */
void hubDelayMs(const uint8_t delayMs)
{
    for (uint8_t i = 0; i < delayMs; i++)
    {
        // delay of 1 ms
        for (uint16_t j = 0; j < 1000; j++)
        {
            __asm__ ("nop");
        }
    }
}

/***
 * @brief Logic for the I2C write transaction for the MAX32664 for multiple bytes
 *
 * @param familyByte
 * @param indexByte
 * @param dataArray
 * @param size
 * @return int
 */
int i2cMax32664WriteBytes(uint8_t familyByte, uint8_t indexByte, uint8_t dataArray[], uint8_t size)
{
    start();
    sendControlByteNoAddr(WRITE, MAX32664_ADDRESS);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s:%d Write address failed!", __func__, __LINE__);
        return FAIL;
    }
}

```

```

// send family byte
i2cByteWrite(familyByte);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Family Byte Write address failed!", __func__);
    return FAIL;
}

// send index byte
i2cByteWrite(indexByte);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Index Byte Write address failed!", __func__);
    return FAIL;
}

for (uint8_t i = 0; i < size; i++)
{
    uint8_t dataByte = dataArray[i];
    i2cByteWrite(dataByte);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Data Byte Write %d failed!", __func__, i);
        return FAIL;
    }
}
stop();

hubDelayMs(CMD_DELAY);

start();
sendControlByteNoAddr(READ, MAX32664_ADDRESS);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Read Byte failed!", __func__);
    return FAIL;
}

uint8_t statusByte = i2cByteRead();

```

```

sendNAcknowledge();

stop();

return statusByte;
}

/** 
 * @brief Logic for the I2C write transaction for writing 1 byte
 *
 * @param familyByte
 * @param indexByte
 * @param dataByte
 * @return int
 */
int i2cMax32664WriteByte(uint8_t familyByte, uint8_t indexByte, uint8_t dataByte)
{
    int status;

    do
    {
        status = i2cMax32664WriteBytes(familyByte, indexByte, &dataByte, 1);
        INFO_LOG("retrying write");
    } while (ERR_DEVICE_BUSY == status);

    return status;
}

/** 
 * @brief Logic for the I2C Read Transaction(specific to MAX32664)
 *
 * @param familyByte
 * @param indexByte
 * @param writeByte
 * @return int
 */
int i2cMax32664ReadByte(uint8_t familyByte, uint8_t indexByte, uint8_t *writeByte)
{
    start();
    sendControlByteNoAddr(WRITE, MAX32664_ADDRESS);
}

```

```

if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Write address failed!", __func__);
    return FAIL;
}

// send family byte
i2cByteWrite(familyByte);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Family Byte Write address failed!", __func__);
    return FAIL;
}

// send index byte
i2cByteWrite(indexByte);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Index Byte Write address failed!", __func__);
    return FAIL;
}

// send write byte if necessary
if (writeByte != NULL)
{
    i2cByteWrite(*writeByte);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Write Byte Write address failed!", __func__);
        return FAIL;
    }
}
stop();

hubDelayMs(CMD_DELAY);

start();
sendControlByteNoAddr(READ, MAX32664_ADDRESS);

```

```

if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Read Byte failed!", __func__);
    return FAIL;
}

uint8_t statusByte = i2cByteRead();
sendAcknowledge();

if (statusByte != SFE_BIO_SUCCESS)
{
    return statusByte;
}

uint8_t readByte = i2cByteRead();
// send nack
sendNAcknowledge();

stop();

return readByte;
}

/**
 * @brief Logic for implementing the Sequential Read for MAX23664
 *
 * @param familyByte
 * @param indexByte
 * @param readArr
 * @param size
 * @return int
 */
int i2cMax32664SequentialReadByte(uint8_t familyByte, uint8_t indexByte, uint8_t readArr[], uint8_t size)
{
    for (uint8_t i = 0; i < size; i++)
    {
        readArr[i] = 0;
    }
}

```

```

start();
sendControlByteNoAddr(WRITE, MAX32664_ADDRESS);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Write address failed!", __func__);
    return FAIL;
}

// send family byte
i2cByteWrite(familyByte);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Family Byte Write address failed!", __func__);
    return FAIL;
}

// send index byte
i2cByteWrite(indexByte);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Index Byte Write address failed!", __func__);
    return FAIL;
}

stop();

hubDelayMs(CMD_DELAY);

start();
sendControlByteNoAddr(READ, MAX32664_ADDRESS);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Read Byte failed!", __func__);
    return FAIL;
}

uint8_t statusByte = i2cByteRead();
sendAcknowledge();

```

```

if (statusByte != SFE_BIO_SUCCESS)
{
    ERROR_LOG("%s Status Byte = %d", __func__, statusByte);
    return statusByte;
}

for (uint8_t i = 0; i < size; i++)
{
    readArr[i] = i2cByteRead();
    sendAcknowledge();
    hubDelayMs(CMD_DELAY);

#ifndef O
    if (i < size - 1)
    {
        sendAcknowledge();
    }
    else
    {
        // send nack after reading last byte
        sendNAcknowledge();
    }
#endif
}

stop();
return SUCCESS;
}

/***
 * @brief Family Byte: WRITE_REGISTER (0x40), Index Byte: WRITE_MAX30101 (0x03), Write Bytes:
 * Register Address and Register Value
 * This function writes the given register value at the given register address
 * for the MAX30101 sensor and returns a boolean indicating a successful or
 * non-successful write.
 * @param regAddr
 * @param regVal
 * @return int
 */
int writeRegisterMAX30101(uint8_t regAddr, uint8_t regVal)

```

```

    uint8_t dataArr[2];
    dataArr[0] = regAddr;
    dataArr[1] = regVal;

    return i2cMax32664WriteBytes(WRITE_REGISTER, WRITE_MAX30101, dataArr, 2);
}

/** @brief Function to write a register value to the MAX30101 sensor.
 * @param regAddr Register Address
 * This function writes the given register address for the MAX30101 Sensor and
 * returns the values at that register.
 * @param regAddr
 * @return uint8_t
 */
uint8_t writeRegisterMAX30101(uint8_t regAddr)
{
    uint8_t data = i2cMax32664ReadByte(READ_REGISTER, READ_MAX30101, &regAddr);
    return data;
}

/** @brief Reads what mode the MAX32664 is to
 * @return uint8_t
 */
uint8_t readSensorHubMode()
{
    int mode = i2cMax32664ReadByte(READ_DEVICE_MODE, 0x00, NULL); // family and index byte.

    if (mode == FAIL)
    {
        ERROR_LOG("Error reading device mode!");
    }

    return mode; // Will return 0x00
}

```

```

/**
 * @brief Sets MAXHUB in application mode
 *
 */
void setApplicationMode()
{
    INFO_LOG("Setting to Application Mode...");
    // i2cSetPin(RSTN, 0);
    RSTN = 0;
    // i2cSetPin(MFIO, 1);
    MFIO = 1;
    hubDelayMs(10*2);
    // i2cSetPin(RSTN, 1);
    RSTN = 1;
    hubDelayMs(50*2);
    // approx 1 sec delay
    for (uint8_t i = 0; i < 5; i++)
    {
        hubDelayMs(100);
    }

    if (readSensorHubMode() != APPLICATION_MODE)
    {
        ERROR_LOG("Application Mode not set!!!");
        return;
    }
    INFO_LOG("Application Mode Set");
}

/**
 * @brief Sets MAXHUB in Bootloader Mode
 *
 */
void setBootloaderMode()
{
    // i2cSetPin(RSTN, 0);
    RSTN = 0;
    // i2cSetPin(MFIO, 0);
    MFIO = 0;
}

```

```

hubDelayMs(10);
// i2cSetPin(RSTN, 1);
RSTN = 1;
hubDelayMs(50);
}

/***
* @brief Family Byte: HUB_STATUS (0x00), Index Byte: 0x00, No Write Byte.
* The following function checks the status of the FIFO.
* @return uint8_t
*/
uint8_t readSensorHubStatus()
{

    uint8_t statusByte = i2cMax32664ReadByte(0x00, 0x00, NULL); // family and index byte.
    return statusByte; // Will return 0x00
}

/***
* @brief Family Byte: OUTPUT_MODE (0x10), Index Byte: SET_FORMAT (0x00),
*Write Byte : outputType (Parameter values in OUTPUT_MODE_WRITE_BYTE)
*
* @param outputType
* @return uint8_t
*/
uint8_t setOutputMode(uint8_t outputType)
{

    if (outputType > SENSOR_ALGO_COUNTER) // Bytes between 0x00 and 0x07
        return FAIL;

    // Check that communication was successful, not that the IC is outputting
    // correct format.

    int statusByte = i2cMax32664WriteByte(OUTPUT_MODE, SET_FORMAT, outputType);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        ERROR_LOG("%s: Received Error Status = %d", __func__, statusByte);
        return statusByte;
    }
}

```

```

    return SFE_BIO_SUCCESS;
}

/***
 * @brief Family Byte: OUTPUT_MODE(0x10), Index Byte: WRITE_SET_THRESHOLD (0x01), Write byte: intThres
 * This function changes the threshold for the FIFO interrupt bit/pin. The
 * interrupt pin is the MFIO pin which is set to INPUT after initialization.
 *
 *
 * @param intThresh
 * @return uint8_t
 */

uint8_t setFifoThreshold(uint8_t intThresh)
{

    // Checks that there was successful communication, not that the threshold was
    // set correctly.
    uint8_t statusByte = i2cMax32664WriteByte(OUTPUT_MODE, WRITE_SET_THRESHOLD, intThresh);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        ERROR_LOG("%s: Received Error Status = %d", __func__, statusByte);
        return statusByte;
    }
    return SFE_BIO_SUCCESS;
}

/***
 * @brief Family Byte: ENABLE_ALGORITHM (0x52), Index Byte:ENABLE_AGC_ALGO (0x00)
 * This function enables (one) or disables (zero) the automatic gain control algorithm.
 * @param enable
 * @return uint8_t
 */

int agcAlgoControl(uint8_t enable)
{
    if (enable > 1)
    {
        return FAIL;
    }

```

```

int statusByte = i2cMax32664WriteByte(ENABLE_ALGORITHM, ENABLE_AGC_ALGO, enable);
if (statusByte != SFE_BIO_SUCCESS)
{
    ERROR_LOG("%s: Received Error Status = %d", __func__, statusByte);
    return statusByte;
}
return SFE_BIO_SUCCESS;
}

/**
 * @brief Family Byte: ENABLE_SENSOR (0x44), Index Byte: ENABLE_MAX30101 (0x03), Write
 * Byte: senSwitch (parameter - 0x00 or 0x01).
 * This function enables the MAX30101.
 * @param senSwitch
 * @return uint8_t
 */
uint8_t max30101Control(uint8_t senSwitch)
{
    if (senSwitch > 1)
    {
        return FAIL;
    }

    // Check that communication was successful, not that the sensor is enabled.
    uint8_t statusByte = i2cMax32664WriteByte(ENABLE_SENSOR, ENABLE_MAX30101, senSwitch);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        ERROR_LOG("%s: Received Error Status = %d", __func__, statusByte);
        return statusByte;
    }
    return SFE_BIO_SUCCESS;
}

/**
 * @brief Family Byte: ENABLE_ALGORITHM (0x52), Index Byte:ENABLE_WHRM_ALGO (0x02)
 * This function enables (one) or disables (zero) the wrist heart rate monitor
 * algorithm.
 * @param mode
 * @return uint8_t

```

```

*/
uint8_t maximFastAlgoControl(uint8_t mode)
{
    if (mode > 2)
    {
        return FAIL;
    }

    int statusByte = i2cMax32664WriteByte(ENABLE_ALGORITHM, ENABLE_WHRM_ALGO, mode);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        ERROR_LOG("%s: Recieved Error Status = %d", __func__, statusByte);
        return statusByte;
    }
    return SFE_BIO_SUCCESS;
}

/**
 * @brief Logic for configuring the MAXHUB only for algo data(heart rate & SPO2)
 *
 * @param mode
 */
void configBpm(uint8_t mode)
{
    uint8_t statusByte;

    statusByte = setOutputMode(ALGO_DATA); // Just the data
    INFO_LOG("setOutputMode Status = %d", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        // return;
    }

    statusByte = setOutputMode(ALGO_DATA); // Just the data
    INFO_LOG("setOutputMode Status 2 = %d", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        // return;
    }
}

```

```

statusByte = setFifoThreshold(0x01); // One sample before interrupt is fired.
INFO_LOG("setFifoThreshold Status = %d", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    // return;
}

statusByte = agcAlgoControl(ENABLE);
INFO_LOG("agcAlgoControl Status = %d", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    // return;
}

statusByte = max30101Control(ENABLE);
INFO_LOG("max30101Control Status = %d", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    // return;
}

hubDelayMs(100);

statusByte = maximFastAlgoControl(mode);
INFO_LOG("maximFastAlgoControl Status = %d", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    // return;
}

hubDelayMs(20);
INFO_LOG("%s: configuration success", __func__);
}

/**
 * @brief Logic for configuring the MAX32664 with MAX30101 sensor data
 *
 * @param mode

```

```

*/
void configSensorBpm(uint8_t mode)
{
    int statusByte;

    statusByte = setOutputMode(SENSOR_AND_ALGORITHM);
    INFO_LOG("setOutputMode Status = %d", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        // return;
    }

    statusByte = setOutputMode(SENSOR_AND_ALGORITHM);
    INFO_LOG("setOutputMode Status = %d", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        // return;
    }

    statusByte = setFifoThreshold(0x01); // One sample before interrupt is fired.
    INFO_LOG("setFifoThreshold Status = %d", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        // return;
    }

    statusByte = agcAlgoControl(ENABLE);
    INFO_LOG("agcAlgoControl Status = %d", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        // return;
    }

    statusByte = max30101Control(ENABLE);
    INFO_LOG("max30101Control Status = %d", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {

```

```

// return;
}

statusByte = maximFastAlgoControl(mode);
INFO_LOG("maximFastAlgoControl Status = %d", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    // return;
}

hubDelayMs(2000);
INFO_LOG("%s: configuration success", __func__);
}

/***
 * @brief Family Byte: READ_DATA_OUTPUT (0x12), Index Byte: NUM_SAMPLES (0x00), Write
 * Byte: NONE
 * This function returns the number of samples available in the FIFO.
 * @return uint8_t
 */
uint8_t numSamplesOutFifo()
{
    int sampAvail = i2cMax32664ReadByte(READ_DATA_OUTPUT, NUM_SAMPLES, NULL);
    if (sampAvail == FAIL)
    {
        ERROR_LOG("Error reading numSamplesOutFifo");
        return 0;
    }
    return sampAvail;
}

/***
 * @brief Logic similar to readSensor Bpm function except that this function only read the
 * SP02 and heart rate values (algo data)
 *
 * @param mode
 * @param data
 * @return int
 */

```

```

int readBpm(uint8_t mode, BioData *data)
{
    uint8_t hubStatus = readSensorHubStatus();
    if (hubStatus == 1)
    {
        ERROR_LOG("%s: Hub Status Error : %d", __func__, hubStatus);
        return FAIL;
    }

    INFO_LOG("Status Byte = %d", hubStatus);

    uint8_t numSamples = numSamplesOutFifo();
    INFO_LOG("numSamples = %d", numSamples);
    if (numSamples == 0)
    {
        INFO_LOG("%s: No samples to read", __func__);
        return FAIL;
    }

    int statusByte;
    XDATA uint8_t dataArr[MAXFAST_ARRAY_SIZE + MAXFAST_EXTENDED_DATA];
    if (mode == 1)
    {
        statusByte = i2cMax32664SequentialReadByte(READ_DATA_OUTPUT, READ_DATA, dataArr, MAXFAST_ARRAY_SIZE);
        printf_tiny("\n\rData from Hub\n\r");
        for (int i = 0; i < MAXFAST_ARRAY_SIZE; i++)
        {
            printf_tiny("%d\n\r", dataArr[i]);
        }
    }
    else if (mode == 2)
    {
        statusByte = i2cMax32664SequentialReadByte(READ_DATA_OUTPUT, READ_DATA, dataArr, MAXFAST_ARRAY_SIZE +
MAXFAST_EXTENDED_DATA);
    }

    if (statusByte != SFE_BIO_SUCCESS)
    {
        ERROR_LOG("%s: Status Error : %d", __func__, statusByte);
    }
}

```

```

    return FAIL;
}

// Heart Rate formatting
uint16_t heartRate = dataArr[0];
heartRate = (heartRate << 8) | (dataArr[1]);
heartRate /= 10;

// Confidence formatting
uint8_t confidence = dataArr[2];

// Blood oxygen level formatting
uint16_t oxygen = dataArr[3];
oxygen = (oxygen << 8) | dataArr[4];
oxygen /= 10;

// "Machine State" - has a finger been detected?
uint8_t status = dataArr[5];

data->heartRate = heartRate;
data->confidence = confidence;
data->oxygen = oxygen;
data->status = status;

if (mode == 2)
{
    // SpO2 r Value formatting
    uint16_t temp = dataArr[6];
    temp = (temp << 8) | dataArr[7];
    float rValue = temp;
    rValue = rValue / 10.0;

    // Extended Machine State formatting
    uint8_t extStatus = dataArr[8];

    data->rValue = rValue;
    data->extStatus = extStatus;
}

#ifndef DEBUG

```

```

switch (status)
{
    case NO_READING:
        INFO_LOG("%s: Finger Status = NO_READING", __func__);
        break;
    case NOT_READY:
        INFO_LOG("%s: Finger Status = NOT_READY", __func__);
        break;
    case OBJECT_DETECTED:
        INFO_LOG("%s: Finger Status = OBJECT_DETECTED", __func__);
        break;
    case FINGER_DETECTED:
        INFO_LOG("%s: Finger Status = FINGER_DETECTED", __func__);
        break;
    default:
        ERROR_LOG("Unknown case!");
}
#endif
return SUCCESS;
}

/***
 * @brief Logic to read the status of the FIFO register (refer to Table 7, pg.25)
 * of the MAX32664
 * @param mode
 * @param data
 * @return int
 */
int readSensorBpm(uint8_t mode, BioData *data)
{
    uint8_t hubStatus = readSensorHubStatus();
    if (hubStatus == 1)
    {
        ERROR_LOG("%s: Hub Status Error : %d", __func__, hubStatus);
        return FAIL;
    }

    INFO_LOG("Hub Status = %d", hubStatus);
}

```

```

uint8_t numSamples = numSamplesOutFifo();
if (numSamplesOutFifo() == 0)
{
    INFO_LOG("%s: No samples to read", __func__);
    return FAIL;
}

INFO_LOG("Num Samples = %d", numSamples);

int statusByte;
XDATA uint8_t sensorData[MAXFAST_ARRAY_SIZE + MAX30101_LED_ARRAY + MAXFAST_EXTENDED_DATA];
if (mode == 1)
{
    statusByte = i2cMax32664SequentialReadByte(READ_DATA_OUTPUT, READ_DATA, sensorData, MAXFAST_ARRAY_SIZE +
MAX30101_LED_ARRAY);
    #if 0
    printf_tiny("\n\rData from Hub");
    for (int i = 0; i < MAXFAST_ARRAY_SIZE + MAX30101_LED_ARRAY; i++)
    {
        printf_tiny("\n\r%d", sensorData[i]);
    }
    #endif
}
else if (mode == 2)
{
    statusByte = i2cMax32664SequentialReadByte(READ_DATA_OUTPUT, READ_DATA, sensorData, MAXFAST_ARRAY_SIZE +
MAX30101_LED_ARRAY + MAXFAST_EXTENDED_DATA);
}

if (statusByte != SFE_BIO_SUCCESS)
{
    ERROR_LOG("%s: Status Error : %d", __func__, statusByte);
    return FAIL;
}

// Value of LED one....
uint32_t irLed = sensorData[0];
irLed = (irLed << 8) | sensorData[1];
irLed = (irLed << 8) | sensorData[2];

```

```

// Value of LED two...
uint32_t redLed = sensorData[3];
redLed = (irLed << 8) | sensorData[4];
redLed = (irLed << 8) | sensorData[5];

// -- What happened here? -- There are two uint32_t values that are given by
// the sensor for LEDs that do not exists on the MAX30101. So we have to
// request those empty values because they occupy the buffer:
// bpmSenArr[6-11].
```

```

// Heart Rate formatting
uint16_t heartRate = sensorData[12];
heartRate = (heartRate << 8) | (sensorData[13]);
heartRate /= 10;

// Confidence formatting
uint8_t confidence = sensorData[14];

// Blood oxygen level formatting
uint16_t oxygen = sensorData[15];
oxygen = (oxygen << 8) | sensorData[16];
oxygen /= 10;

// "Machine State" - has a finger been detected?
uint8_t status = sensorData[17];

data->irLed = irLed;
data->redLed = redLed;

data->heartRate = heartRate;
data->confidence = confidence;
data->oxygen = oxygen;
data->status = status;

if (mode == 2)
{
    // SpO2 r Value formatting
    uint16_t temp = sensorData[18];
}

```

```

temp = (temp << 8) | sensorData[19];
float rValue = temp;
rValue = rValue / 10.0;

// Extended Machine State formatting
uint8_t extStatus = sensorData[20];

data->rValue = rValue;
data->extStatus = extStatus;
}

#ifndef STATUS_LOG
switch (status)
{
case NO_READING:
INFO_LOG("NO_READING");
break;

case NOT_READY:
INFO_LOG("NOT_READY");
break;

case OBJECT_DETECTED:
INFO_LOG("OBJECT_DETECTED");
break;

case FINGER_DETECTED:
INFO_LOG("FINGER_DETECTED");
break;

case PRESSING_TOO_HARD:
INFO_LOG("PRESSING_TOO_HARD");
break;

default:
ERROR_LOG("UNKNOWN ERROR: %d", status);
}
#endif

return SUCCESS;
}

```

```
/*
 * @brief This function modifies the pulse width of the MAX30101 LEDs. All of the LEDs
 * are modified to the same width. This will affect the number of samples that
 * can be collected and will also affect the ADC resolution.

```

```

* Default: 69us - 15 resolution - 50 samples per second.
* Register: 0x0A, bits [1:0]
* Width(us) - Resolution - Sample Rate
* 69us   -  15   -  <= 3200 (fastest - least resolution)
* 118us  -  16   -  <= 1600
* 215us  -  17   -  <= 1600
* 411us  -  18   -  <= 1000 (slowest - highest resolution)
* @param width
* @return int
*/
int setPulseWidth(uint16_t width)
{
    uint8_t bits;
    uint8_t regVal;

    // Make sure the correct pulse width is selected.
    if (width == 69)
        bits = 0;
    else if (width == 118)
        bits = 1;
    else if (width == 215)
        bits = 2;
    else if (width == 411)
        bits = 3;
    else
        return FAIL;

    // Get current register value so that nothing is overwritten.
    regVal = readRegisterMAX30101(CONFIGURATION_REGISTER);
    regVal &= PULSE_MASK;           // Mask bits to change.
    regVal |= bits;                // Add bits
    writeRegisterMAX30101(CONFIGURATION_REGISTER, regVal); // Write Register

    return SFE_BIO_SUCCESS;
}

/**
* @brief This function changes the sample rate of the MAX30101 sensor. The sample

```

```

* rate is affected by the set pulse width of the MAX30101 LEDs.
* Default: 69us - 15 resolution - 50 samples per second.
* Register: 0x0A, bits [4:2]
* Width(us) - Resolution - Sample Rate
* 69us   - 15   - <= 3200 (fastest - least resolution)
* 118us  - 16   - <= 1600
* 215us  - 17   - <= 1600
* 411us  - 18   - <= 1000 (slowest - highest resolution)
* @param sampRate
* @return int
*/
int setSampleRate(uint16_t sampRate)
{
    uint8_t bits;
    uint8_t regVal;

    // Make sure the correct sample rate was picked
    if (sampRate == 50)
        bits = 0;
    else if (sampRate == 100)
        bits = 1;
    else if (sampRate == 200)
        bits = 2;
    else if (sampRate == 400)
        bits = 3;
    else if (sampRate == 800)
        bits = 4;
    else if (sampRate == 1000)
        bits = 5;
    else if (sampRate == 1600)
        bits = 6;
    else if (sampRate == 3200)
        bits = 7;
    else
        return FAIL;

    // Get current register value so that nothing is overwritten.
    regVal = readRegisterMAX30101(CONFIGURATION_REGISTER);

```

```

regVal &= SAMP_MASK; // Mask bits to change.
regVal |= (bits << 2); // Add bits but shift them first to correct position.
writeRegisterMAX30101(CONFIGURATION_REGISTER, regVal); // Write Register

return SFE_BIO_SUCCESS;
}

/**
 * @brief Family Byte: IDENTITY (0x01), Index Byte: READ MCU_TYPE, Write Byte: NONE
 * The following function returns a byte that signifies the microcontroller that
 * is in communication with your host microcontroller. Returns 0x00 for the
 * MAX32625 and 0x01 for the MAX32660/MAX32664.
 *
 * @return int
 */
int checkDeviceType()
{
    int returnByte = i2cMax32664ReadByte(IDENTITY, READ MCU_TYPE, NULL);
    if (returnByte == FAIL)
    {
        ERROR_LOG("Reaed Device Type failed");
        return FAIL;
    }

    // 0x01: MAX32660/MAX32664
    if (returnByte != 0x01)
    {
        ERROR_LOG("Device Type %d not expected", returnByte);
        return FAIL;
    }

    INFO_LOG("Device Type Check Success");
    return SUCCESS;
}

//#define BPM_DATA
#define SENSOR_DATA

// #define PLOT_DATA

```

```

#define SHOW_DATA

void lastTest()
{
    // set output mode
    int status = i2cMax32664WriteByte(0x10, 0x00, 0x03);

    // set fifo threshold
    status = i2cMax32664WriteByte(0x10, 0x01, 0x01);

    // enable max30101
    status = i2cMax32664WriteByte(0x44, 0x03, 0x01);

    // enable maximfast
    status = i2cMax32664WriteByte(0x52, 0x02, 0x01);

    // read hub status
    while(1)
    {
        int readVal = i2cMax32664ReadByte(0x00, 0x00, NULL);
        printf("hub status = %x\n\r", readVal);
        hubDelayMs(100);
    }
}

/***
 * @brief Test code to check if the MAxHub is working as expected
 *
 */
void testMaxHub()
{
    INFO_LOG("Entering Text Max HUB");
    setApplicationMode();
    // setBootloaderMode();

    // i2cSetPin(MFIO, 0);
    MFIO = 0;
}

```

```

//lastTest();
//while(1);

checkDeviceType();

uint8_t mode = 1;

#ifndef BPM_DATA
configBpm(mode);
#endif

#ifndef SENSOR_DATA
configSensorBpm(mode);
setPulseWidth(118);
setSampleRate(1600);
#endif

XDATA BioData data;
uint32_t counter = 0;

#if 0
uint8_t continousMode[2] = {0x0A, 0x00};
uint8_t val = i2cMax32664WriteBytes(CHANGE_ALGORITHM_CONFIG, 0x07, continousMode, 2);

hubDelayMs(10);

// read wearable Algorithm Suite
uint8_t writeByte = 0x0B;
val = i2cMax32664ReadByte(READ_ALGORITHM_CONFIG, 0x07, &writeByte); // family and index byte.
INFO_LOG("ALGO SUITE = %d", val);
#endif

while (1)
{
#ifndef BPM_DATA
int status = readBpm(mode, &data);
if (status == SUCCESS)
{
// INFO_LOG("CONFIDENCE = %u", data.confidence);
if (data.confidence > 50)
{

```

```
INFO_LOG("HEART-RATE = %u", data.heartRate);
INFO_LOG("OXYGEN = %u", data.oxygen);
```

```
if (mode == 2)
{
    INFO_LOG("rValue = %f", data.rValue);
    INFO_LOG("Extended Status = %u", data.extStatus);
}
data.confidence = 0;
}
}

#endif
```

```
#ifdef SENSOR_DATA
int status = readSensorBpm(mode, &data);
if (status == SUCCESS)
{
    INFO_LOG("CONFIDENCE = %u", data.confidence);
    if (data.confidence > 50)
    {

```

```
#ifdef PLOT_DATA
LOG("\n\r%d,%d,%d", counter++, data.irLed, data.redLed);
#endif
#endif SHOW_DATA
INFO_LOG("IR LED = %lu", data.irLed);
INFO_LOG("RED LED = %lu", data.redLed);
```

```
INFO_LOG("HEART-RATE = %u", data.heartRate);
INFO_LOG("OXYGEN = %u", data.oxygen);
```

```
if (mode == 2)
{
    INFO_LOG("rValue = %f", data.rValue);
    INFO_LOG("Extended Status = %u", data.extStatus);
}
data.confidence = 0;
}
```

```

#endif

hubDelayMs(10);

}

// i2cSetPin(MFIO, 1);

MFIO = 1;

}

/*****
 * Author - Venetia Furtado
 * Final Project: Health Monitoring System
 * ECEN 5613 - Spring 2024 - Prof. McClure
 * University of Colorado Boulder
 * Revised 04/26/2024
 *
 *****
 * This file contains the implementation for the LCD device driver.
 * References: https://www.pjrc.com/teensy/td\_libs\_GLCD.html
*****/

/***** INCLUDES & DEFINES *****/
#include <mcs51/8051.h>
#include "at89c51ed2.h"
#include <stdint.h>
#include "log.h"
#include "io.h"
#include "font.h"

// Sending command to LCD
#define GLCD_RESET P1_1

#define STATUS_READ 0x800E
#define DATA_WRITE_LEFT_ADDR 0x8009
#define DATA_WRITE_RIGHT_ADDR 0x8005

#define COMMAND_WRITE_LEFT_ADDR 0x8008
#define COMMAND_WRITE_RIGHT_ADDR 0x8004

```

```
#define COMMAND_WRITE_ADDR 0x800C
```

```
#define DATA_READ_LEFT_ADDR 0x800B  
#define DATA_READ_RIGHT_ADDR 0x800C
```

```
#define MAX_COLUMN_PER_CONTROLLER 64
```

```
#define PAGE_MASK          0xB8 // 1011 1 xxx  
#define Y_MASK            0x40 // 01xx xxxx  
#define BUSY_MASK         0x80
```

```
#define CHARACTER_WIDTH 3
```

```
#define STATUS_X_LOC 2  
#define SPO2_X_LOC 70  
#define BPM_X_LOC 100
```

```
typedef enum
{
    COMMAND_WRITE = 0,
    COMMAND_WRITE_LEFT,
    COMMAND_WRITE_RIGHT,
    COLUMN_SELECT_LEFT,
    COLUMN_SELECT_RIGHT,
    DATA_WRITE_LEFT,
    DATA_WRITE_RIGHT,
    DATA_READ_LEFT,
    DATA_READ_RIGHT
} MemoryAddress;
```

typedef enum

```
{  
    ZERO = 0,  
    ONE = 1  
}; BitValue;
```

```
/******
```

// FUNCTION DEFINITIONS

```
/******|
```

```

/**
 * @brief Function to provide a delay of timeUs
 *
 * @param timeUs
 */
void delay(const uint16_t timeUs)
{
    for (uint16_t i = 0; i < timeUs; i++)
    {
        __asm__("nop");
    }
}

/** 
 * @brief Polls the LCD busy flag. Function does not return until the LCD
 * controller is ready to accept another command.
 */
void busyWait()
{
    // since value at BUSY_ADDRESS can change we use volatile
    volatile __xdata uint8_t *ptr = (volatile __xdata uint8_t *)STATUS_READ;
    // checks and waits until Busy Flag (DB7) is 0
    while ((*ptr & BUSY_MASK) != 0);
}

/** 
 * @brief Logic to write to the requested memory address by the 8051
 *
 * @param addr
 * @param val
 */
void gLcdWriteToRegister(MemoryAddress addr, uint8_t val)
{
    __xdata uint8_t *ptr = NULL;

    if (addr == COMMAND_WRITE)
    {
        //Type-casting COMMAND_WRITE_ADDR to a ptr of type __xdata uin8_t ptr

```

```

ptr = (__xdata uint8_t *)COMMAND_WRITE_ADDR;
}

else if (addr == COMMAND_WRITE_LEFT)
{
    ptr = (__xdata uint8_t *)COMMAND_WRITE_LEFT_ADDR;
}

else if (addr == COMMAND_WRITE_RIGHT)
{
    ptr = (__xdata uint8_t *)COMMAND_WRITE_RIGHT_ADDR;
}

else if (addr == DATA_WRITE_LEFT)
{
    ptr = (__xdata uint8_t *)DATA_WRITE_LEFT_ADDR;
}

else if (addr == DATA_WRITE_RIGHT)
{
    ptr = (__xdata uint8_t *)DATA_WRITE_RIGHT_ADDR;
}

else
{
    ERROR_LOG("Unknown Memory Addr! = %d", addr);
    return;
}

// write at address
*ptr = val;
}

/** 
 * @brief Logic that return which controller needs to be selected based on the
 * column value
 * @param column
 * @return uint8_t
 */
uint8_t glcdControllerSelect(uint8_t column)
{
    // left controller
    if (column < MAX_COLUMN_PER_CONTROLLER)
    {
        return COLUMN_SELECT_LEFT;
    }
}

```

```

}

else // right controller
{
    return COLUMN_SELECT_RIGHT;
}
}

/** 
 * @brief Function that has the logic to select the column and page to be written to
 *
 * @param column
 * @param page
 */
void glcdGoToAddr(uint8_t column, uint8_t page)
{
    uint8_t controllerSelect = glcdControllerSelect(column);

    if (controllerSelect == COLUMN_SELECT_LEFT)
    {
        // Set X Address(Page Mask)
        gLcdWriteToRegister(COMMAND_WRITE_LEFT, PAGE_MASK | page);
        busyWait();
    }

    // Set Y Address(Y Mask)
    gLcdWriteToRegister(COMMAND_WRITE_LEFT, Y_MASK | column);
    busyWait();
}

else if (controllerSelect == COLUMN_SELECT_RIGHT) // right controller
{
    column = column - MAX_COLUMN_PER_CONTROLLER;

    // Set X Address
    gLcdWriteToRegister(COMMAND_WRITE_RIGHT, PAGE_MASK | page);
    busyWait();

    // Set Y Address
    gLcdWriteToRegister(COMMAND_WRITE_RIGHT, Y_MASK | column);
    busyWait();
}

```

```

}

/** 
 * @brief Logic to write value onto the specified address location
 *
 * @param column
 * @param page
 * @param val
 */
void glcdWrite(uint8_t column, uint8_t page, char val)
{
    glcdGoToAddr(column, page);

    // Send character
    uint8_t controllerSelect = glcdControllerSelect(column);

    if (controllerSelect == COLUMN_SELECT_LEFT)
    {
        gLcdWriteToRegister(DATA_WRITE_LEFT, val);
    }
    else
    {
        gLcdWriteToRegister(DATA_WRITE_RIGHT, val);
    }
    busyWait();
}

/** 
 * @brief Logic to read from a specified address location
 *
 * @param column
 * @return uint8_t
 */
uint8_t glcdRead(uint8_t column)
{
    uint8_t controllerSelect = glcdControllerSelect(column);

    __xdata uint8_t *ptr = NULL;

```

```

if (controllerSelect == COLUMN_SELECT_LEFT)
{
    ptr = (__xdata uint8_t *)DATA_READ_LEFT_ADDR;
}

else if (controllerSelect == COLUMN_SELECT_RIGHT)
{
    ptr = (__xdata uint8_t *)DATA_READ_RIGHT_ADDR;
}

else
{
    ERROR_LOG("UNKNOWN CONTROLLER");
}

uint8_t dataByte = *ptr;

return dataByte;
}

/***
 * @brief Logic to set the pixel within a page to be set or reset
 *
 * @param row
 * @param column
 * @param val
 */
void setPixel(uint8_t row, uint8_t column, BitValue val)
{
    //setting y-axis to increment downwards
    row = 63 - row;

    //Dividing by total no. of pages (8) to get the page
    uint8_t page = row / 8;
    //gives pixel positon within the page
    uint8_t pixelPosition = row % 8;

    // set the column and page
    lcdGoToAddr(column, page);

/*dummy read as per datasheet: To read the contents of display data RAM, twice access of read instruction is needed.

```

In first access, data in display data RAM is latched into output register. In second access, MPU can read data which is latched. That is to read the data in display data RAM, it needs dummy read. But status read is not needed dummy read.*/

```

uint8_t pageByte = glcdRead(column);
pageByte = glcdRead(column);

// INFO_LOG("read pageByte = %d", pageByte);
// set or reset the bit
if (val == ONE)
{
    pageByte = pageByte | (1 << pixelPosition);
}
else if (val == ZERO)
{
    // Example: 00000001 << 2 = ~(00000100) = 11111011
    pageByte = pageByte & (~(1 << pixelPosition));
}

// INFO_LOG("write pageByte = %d", pageByte);
glcdWrite(column, page, pageByte);
}

/***
 * @brief Logic to clear GLCD
 *
 */
void glcdClear()
{
    for (uint8_t column = 0; column < 128; column++)
    {
        for (uint8_t page = 0; page < 8; page++)
        {
            glcdWrite(column, page, 0x00);
        }
    }
}

/***
 * @brief Logic to print a character on the screen

```

```

*
* @param column
* @param page
* @param c
*/
void glcdWriteChar(uint8_t column, uint8_t page, char c)
{
    // Subtracting c from ASCII value of SPACE; each character takes three pages (5x3)
    uint8_t offset = (c - 0x20) * CHARACTER_WIDTH;
    for (uint8_t i = 0; i < CHARACTER_WIDTH; i++)
    {
        // INFO_LOG("Column: %u,Page: %u, Value: %u",column, page, font[offset + i] );
        glcdWrite(column, page, font[offset + i]);
        column++;
    }
}

/**
* @brief Function to clear out a string
*
* @param column
* @param page
* @param size
*/
void glcdClearString(uint8_t column, uint8_t page, uint8_t size)
{
    for (uint8_t i = 0; i < size; i++)
    {
        glcdWriteChar(column, page, 0x20);
        column += CHARACTER_WIDTH + 1;
    }
}

/**
* @brief Function to write a string at a location
*
* @param column
* @param page
* @param string

```

```

*/
void glcdWriteString(uint8_t column, uint8_t page, char string[])
{
    uint8_t i = 0;
    while (string[i] != '\0')
    {
        glcdWriteChar(column, page, string[i]);
        // INFO_LOG("%c", string[i]);
        i++;
        column += CHARACTER_WIDTH + 1;
    }
}

/** 
 * @brief Logic to find the absolute between two values
 *
 * @param value1
 * @param value2
 * @return uint8_t
 */
uint8_t absolute(int value1, int value2)
{
    uint8_t result;
    if (value1 > value2)
    {
        result = value1 - value2;
    }
    else
    {
        result = value2 - value1;
    }
    return result;
}

/** 
 * @brief Logic to plot graph on LCD based on Bresenham's Line algorithm
 *
 * @param x1
 * @param y1
 */

```

```

* @param x2
* @param y2
*/
void glcdDrawGraph(int x1, int y1, int x2, int y2)
{
    int slope = absolute(y1, y2) - absolute(x1, x2);
    if (slope > 0)
    {
        // swapping x1 and y1 values
        int temp;
        temp = x1;
        x1 = y1;
        y1 = temp;

        // swaping x2 and y2 values
        temp = x2;
        x2 = y2;
        y2 = temp;
    }

    if (x1 > x2)
    {
        // swapping x1 and x2 values
        int temp;
        temp = x1;
        x1 = x2;
        x2 = temp;

        // swapping y1 and y2 values
        temp = y1;
        y1 = y2;
        y2 = temp;
    }

    int deltaX = x2 - x1;
    int deltaY = absolute(y2, y1);
    int error = deltaX / 2;
    int yStep;
    if (y1 < y2)

```

```

{
    yStep = 1;
}
else
{
    yStep = -1;
}

int y = y1;
for (int x = x1; x <= x2; x++)
{
    if (slope > 0)
    {
        setPixel(x, y, 1);
    }
    else
    {
        setPixel(y, x, 1);
    }

    error = error - deltaY;
    if (error < 0)
    {
        y = y + yStep;
        error = error + deltaX;
    }
}

}

/** 
 * @brief Function that converts an integer value to its ASCII equivalent
 *
 * @param val
 * @param result
 * @return uint8_t Returns the length of the string
 */
uint8_t itoa(uint8_t val, char result[])
{
    if (val == 0)
    {

```

```

    result[0] = '0';
    return 1;
}

uint8_t count = 0;
uint8_t digits[3];
while (val != 0)
{
    //gets LSB
    digits[count] = val % 10;
    //gets MSB
    val = val / 10;
    count++;
}

//feeding the converted string to result[i]
uint8_t i = 0;
for (int j = count - 1; j >= 0; j--)
{
    result[i] = '0' + digits[j];
    i++;
}

return count;
}

/***
 * @brief Prints the Oxygen level SPO2 on the GLCD screen
 *
 * @param oxygen
 */
void glcdUpdateSpo2(uint8_t oxygen)
{
    __xdata char oxygenLevel[5]; // max 3 digits + % + NULL char
    uint8_t size = itoa(oxygen, oxygenLevel);
    oxygenLevel[size] = '%';
    oxygenLevel[size + 1] = '\0';
    glcdClearString(SPO2_X_LOC, 1, 5);
    glcdWriteString(SPO2_X_LOC, 1, oxygenLevel);
}

```

```

}

/** 
 * @brief Prints the BPM level on the GLCD screen
 *
 * @param bpm
 */
void glcdUpdateBpm(uint8_t bpm)
{
    __xdata char bpmString[4]; // max 3 digits + NULL char
    uint8_t size = itoa(bpm, bpmString);
    bpmString[size] = '\0';
    glcdClearString(BPM_X_LOC, 1, 5);
    glcdWriteString(BPM_X_LOC, 1, bpmString);
}

/** 
 * @brief Prints the status of the sensor on the GLCD screen
 *
 * @param status
 */
void glcdUpdateStatus(char* status)
{
    glcdClearString(STATUS_X_LOC, 1, 15);
    glcdWriteString(STATUS_X_LOC, 1, status);
}

/** 
 * @brief Initialization process of the GLCD as per datasheet
 *
 */
void glcdInit()
{
    LCD_RESET = 0;
    delay(1000);
    LCD_RESET = 1;

    gLCDWriteToRegister(COMMAND_WRITE, 0xB8); // select first page 0
    delay(100);
}

```

```

gLcdWriteToRegister(COMMAND_WRITE, 0x40); // select row 0
delay(100);

gLcdWriteToRegister(COMMAND_WRITE, 0x3F); // display on
delay(100);

glcdClear();

glcdWriteString(STATUS_X_LOC, 0, "STATUS");
glcdWriteString(SPO2_X_LOC, 0, "SPO2");
glcdWriteString(BPM_X_LOC, 0, "BPM");
}

// global variables set to zero at init
uint8_t previousReadingX;
uint8_t previousReadingY;

/**
 * @brief Plots the IR LED reading on the GLCD
 *
 * @param IrValue
 */
void glcdPlotIrValue(uint8_t IrValue)
{
    //Scaling a uint8_t value (0-256) to be plotted between 0 to 32 on the LCD screen
    uint8_t y = IrValue >> 3;
    uint8_t x = previousReadingX + 2;

    //after rollover, clears the immediate next section of the graph
    for (uint8_t page = 2; page < 8; page++)
    {
        glcdWrite(previousReadingX + 1, page, 0x00);
        glcdWrite(previousReadingX + 2, page, 0x00);
    }

    // INFO_LOG("x: %d, Previous Y: %d, x+1: %d, y = %d", x, previousReadingY, x+1, y);
    glcdDrawGraph(previousReadingX, previousReadingY, x, y);
}

```

```

if(previousReadingX == 126)
{
    previousReadingX = 0;
    previousReadingY = 0;
}
else
{
    previousReadingX = x;
    previousReadingY = y;
}
}

/***
 * @brief Function was used during testing code to generate a random value
 * @param min
 * @param max
 * @return int
 */
int generateRandom(int min, int max)
{
    return min + rand() % (max - min + 1);
}

/***
 * @brief Function to test the LCD working
 *
 */
void lcdTest()
{
    INFO_LOG("LCD Testing");

    // glcdWriteChar(0,0,'V');
    glcdWriteString(32, 0, "SPO2");
    // glcdWriteString(32,1, "97%");
    glcdWriteString(94, 0, "BPM");
    // glcdWriteString(94,1, "76");

    uint8_t x1 = 0;
    uint8_t y1 = 5;

```

```

#if 0
for(uint8_t i = 0; i < 127; i++)
{
    uint8_t x2 = i;
    uint8_t y2 = generateRandom(0, 40);
    lcdDrawGraph(x1, y1, x2,y2);
    delay(500);

    x1 = x2;
    y1 =y2;
}
#endif

while(1)
{
    lcdPlotIrValue(generateRandom(40,160));
    delay(500);
}

// lcdWriteString()

while (1);
}

/******************
 * Author - Venetia Furtado
 * Final Project: Health Monitoring System
 * ECEN 5613 - Spring 2024 - Prof. McClure
 * University of Colorado Boulder
 * Revised 04/26/2024
 *
***** */

* This file contains the definitions of all the functions that receive an input
* and give an output through the serial port.
***** */

/******************/ INCLUDES
***** */

#include <mcs51/8051.h>

```

```

#include "at89c51ed2.h"
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include "log.h"
#include "io.h"

/*****************/
//          FUNCTION DEFINITIONS
/*****************/

/***
 * @brief Puts a character onto the serial line
 * Note: Assuming hardware is initialized (TI = 1)
 * @param c
 * @return int
 */
int putchar(int c)
{
    while (TI != 1);      // Waits until TI flag is ready to transmit data, set by hardware
    TI = 0;                // Once data has been transmitted, TI flag is cleared by software
    SBUF = c;              // Transmits the ASCII value of c on the serial line
    return c;
}

/***
 * @brief Receives a character from the serial line
 * Note: Assuming hardware is initialized (RI = 1)
 * @param none
 * @return int
 */
int getchar(void)
{
    while (RI != 1);      // Waits until RI has received data
    RI = 0;                // Once data has been received, software clears the RI flag
    return SBUF;            // Returns value in SBUF to the calling function
}

```

```

* @brief Takes a maximum of maxDigits digits (no alphabets or special characters allowed)
* input from the user and converts ASCII to unsigned integer
* @param maxDigits
* @param maxVal Maximum value the user can enter
* @return uint16_t
*/
uint16_t getUserNumberInput(uint8_t maxDigits, uint16_t maxVal) //max number of digits user can input
{
    //take only a maximum n digit input (3 in this case)
    __xdata uint16_t value = 0;
    while (1)
    {
        value = 0;
        bool fail = false;
        printf_tiny("\n\rEnter a maximum %d digit number:", maxDigits);
        for (uint8_t i = 0; i < maxDigits; i++)
        {
            char c = getchar();           // take user input
            if (c == '\r')              //user enters ENTER key
            {
                break;
            }
            printf_tiny("%c", c);        //prints character on screen as user is typing
            if (c >= '0' && c <= '9') // check if input is a number
            {
                uint8_t x = c - '0';     //converts user input(ASCII) to int
                value = value * 10 + x;   //calculates multi-digit ASCII input
                if (value > maxVal)
                {
                    ERROR_LOG("Value entered is beyond range %x", maxVal);
                    fail = true;
                    break;
                }
            }
        }
        else
        {
            ERROR_LOG("Enter only digit!"); //if user enters an other character than a number
            fail = true;
            break;
        }
    }
}

```

```

        }

    }

    if (fail == false)           //if no previous errors have occurred, break while loop
    {
        break;
    }
}

USER_INPUT_CHECK("User entered input = %d", value);
return value;
}

/***
 * @brief Takes a maximum of n character HEX input from user
 * @param maxChar Maximum characters user can enter
 * @param maxVal Maximum value the user can enter
 * @return uint16_t
 */
uint16_t getUserHexInput(uint8_t maxChar,uint16_t maxVal) //max number of digits user can input
{
    //take only a maximum n character input
    __xdata uint16_t value = 0;
    while (1)
    {
        value = 0;
        bool fail = false;
        printf_tiny("\n\rEnter a maximum %d character input:", maxChar);
        for (uint8_t i = 0; i < maxChar; i++)
        {
            char c = getchar();           // take user input
            if (c == '\r')              //user enters ENTER key
            {
                break;
            }
            printf_tiny("%c", c);        //prints character on screen as user is typing
            uint8_t x = 0;
            if (c >= '0' && c <= '9') // check if input is a number
            {
                x = c - '0';           //converts user input(ASCII) to int

```

```

}

else if (c == 'A' || c == 'a')
{
    x = 10;
}

else if (c == 'B' || c == 'b')
{
    x = 11;
}

else if (c == 'C' || c == 'c')
{
    x = 12;
}

else if (c == 'D' || c == 'd')
{
    x = 13;
}

else if (c == 'E' || c == 'e')
{
    x = 14;
}

else if (c == 'F' || c == 'f')
{
    x = 15;
}

else
{
    ERROR_LOG("Enter only values between 0 to 9 and A to F!"); //if user enters an other character than a number
    fail = true;
    break;
}

value = value * 16 + x; //calculates multi-digit ASCII input
if (value > maxVal)
{
    ERROR_LOG("Value entered is beyond range %x", maxVal);
    fail = true;
    break;
}
}

```

```

if (fail == false) //if no previous errors have occurred, break while loop
{
    break;
}
}

USER_INPUT_CHECK("User entered input = %x", value);
return value;
}

```

```
*****
```

```

* Author - Venetia Furtado
* Final Project: Health Monitoring System
* ECEN 5613 - Spring 2024 - Prof. McClure
* University of Colorado Boulder
* Revised 04/26/2024
*
```

```
*****
*This file contains the function for the initialization of hardware and serial
*communication.
```

```
******/
```

```
******/
```

```
//           INCLUDES
```

```
******/
```

```
#include <mcs51/8051.h>
```

```
#include "at89c51ed2.h"
```

```
#include "stdint.h"
```

```
******/
```

```
//           FUNCTION DEFINITIONS
```

```
******/
```

```
/**
```

```
* @brief Implements functionality that needs to be done before the main() function
```

```
*/
```

```
_sdcc_external_startup()
```

```
{
```

```
    AUXR |= 0x0C;           //Setting the XRS0 and XRS1 bits to 1 to enable 1KB of internal XRAM
```

```

    return 0;           //to indicate successful initialization
}

/***
 * @brief Initializing the 8051 serial communication hardware.
 *
 */
void hardware_init()
{
    TMOD = 0x20;      // initialize Timer 1, Mode 2(auto-reload)
    TH1 = 0xFD;        // 9600bps baud rate
    //TH1 = 0xFA;       // 9600bps baud rate in X2 mode
    //CKCON0 |= 0x01;   //Configures system clock in X2 mode
    SCON = 0x50;       /*SCON register is loaded with 50H indicating serial mode 1 where an 8 bit data is framed with
                        start and stop bits,REN enabled*/
    TR1 = 1;          // starts timer 1 to generate clk at baud rate
    TI = 1;           //Set to 1 since int putchar(int c) expects it to be 1 initially
}

```

```
*****
```

```

* Author - Venetia Furtado
* Final Project: Health Monitoring System
* ECEN 5613 - Spring 2024 - Prof. McClure
* University of Colorado Boulder
* Revised 04/26/2024
*
```

```
*****
```

```
* This file contains the logic for i2c driver.
```

```
******/
```

```
*****/
```

```
//           INCLUDES & DEFINES
```

```
*****/
```

```
#include <mcs51/8051.h>
```

```
#include <stdint.h>
```

```
#include "at89c51ed2.h"
```

```
#include "log.h"
```

```
#include "io.h"
```

```

#include "i2c.h"

#define SCL      P1_3
#define SDA      P1_2
#define NOP      __asm__("nop")

#define HEX_DUMP_LINE_SIZE 16

/*****
//          FUNCTION DEFINITIONS
****/

/***
 * @brief Toggles the clock
 *
 */
void clock()
{
    NOP;
    SCL = 1;
    NOP;
    NOP;
    SCL = 0;
    NOP;
}

/***
 * @brief Checks if an acknowledgement has been given by the peripheral
 *
 * @return int SUCCESS or FAIL
 */
int checkAcknowledgement()
{
    //check to see if the device is acking by pulling SDA low
    SDA = 1;
    NOP;
    NOP;
    SCL = 1;
}

```

```

//Capturing value at SDA
uint8_t value = SDA;
NOP;
SCL = 0;
NOP;
if(value != 0)
{
    return FAIL;
}
return SUCCESS;
}

/** @brief Logic for starting the I2C communication
 */
void start()
{
    SDA = 1;
    NOP;
    SCL = 1;
    NOP;
    SDA = 0;
    NOP;
    NOP;
    SCL = 0;
}

/** @brief Logic to stop the I2C communication
 */
void stop()
{
    SDA = 0;
    NOP;
    NOP;
    SCL = 1;
    NOP;
}

```

```

NOP;
SDA = 1;
NOP;
NOP;
}

/***
* @brief Logic to send an Acknowledgement after a read/write transaction
*
*/
void sendAcknowledge()
{
    SDA = 0;
    NOP;
    NOP;
    clock();
    SDA = 1;
    NOP;
    NOP;
}

/***
* @brief Logic to send a Nack at the end of a Read/Write transaction
*
*/
void sendNAcknowledge()
{
    SDA = 1;
    NOP;
    NOP;
    clock();
    NOP;
    NOP;
}

/***
* @brief Logic to reverse a byte; LSB of input become MSB of output.
* @param input
* @return uint8_t

```

```

*/
uint8_t reverse(uint8_t input)
{
    uint8_t output = 0;

    for (int i = 0; i < 8; i++)
    {
        output = output << 1;
        //Bitwise AND-ing input with 0x01 to obtain its LSB and OR with output
        output = output | (input & 0x01);
        input = input >> 1;
    }

    return output;
}

/**
 * @brief Logic for sending the Control Byte
 *
 * @param op enum READ/WRITE
 * @param deviceId device ID to be read/write from
 * @param addr address to be READ/WRITE from
 */
void sendControlByte(const Operation op, uint8_t deviceId, uint16_t addr)
{
    //Operation to get the upper 3 bits (b2,b1,b0)
    uint8_t blockAddress = (addr & 0x700) >> 8;

    uint8_t b0 = blockAddress & 0x01;
    blockAddress >>= 1;
    uint8_t b1 = blockAddress & 0x01;
    blockAddress >>= 1;
    uint8_t b2 = blockAddress & 0x01;
    blockAddress >>= 1;

    DEBUG("Block Bits b2=%u \t b1=%u \t b0=%u\n\r", b2,b1,b0);

    //Sending the device address
    for (int i = 0; i < 4; i++)
    {

```

```

SDA = deviceId & 0x01;
clock();
deviceId = deviceId >> 1;
}

// block address b2
SDA = b2;
clock();

// block address b1
SDA = b1;
clock();

// block address b0
SDA = b0;
clock();

//Read or Write
SDA = op;
clock();
}

/***
 * @brief Logic for sending the Control Byte
 *
 * @param op enum READ/WRITE
 * @param deviceId device ID to be read/write from
 */
void sendControlByteNoAddr(const Operation op, uint8_t deviceId)
{
    // Sending the device address
    for (int i = 0; i < 7; i++)
    {
        SDA = deviceId & 0x01;
        clock();
        deviceId = deviceId >> 1;
    }
    //Read or Write
    SDA = op;
}

```

```

clock();

}

void i2cByteWrite(uint8_t dataByte)
{
    //Sending the data that needs to be written at addr
    uint8_t databyteMsbFirst = reverse(dataByte);
    for (int i = 0; i < 8; i++)
    {
        SDA = databyteMsbFirst & 0x01;
        databyteMsbFirst >>= 1;
        clock();
    }
}

uint8_t i2cByteRead()
{
    uint8_t dataByte = 0;
    for (int i = 0; i < 8; i++)
    {
        SCL = 1;
        // Bit packing each bit into dataByte to give MCU the data from requested address from MSB to LSB(little endian)
        dataByte = (dataByte << 1) | SDA;
        NOP;
        SCL = 0;
        NOP;
    }
    return dataByte;
}

/***
 * @brief Function that contains logic for the Read transaction (one byte) from the MAX30101
 * Refer to page 31, Figure 10 of datasheet
 * @param deviceId
 * @param regAddr
 * @return int
 */
int readRegisterMAXWing(uint8_t deviceId, uint8_t regAddr)
{

```

```

start();

sendControlByteNoAddr(WRITE, deviceId);

if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s: Acknowledgement Failed", __func__);
    return FAIL;
}

uint8_t regAddressMsbFirst = reverse(regAddr);

for (int i = 0; i < 8; i++)
{
    SDA = regAddressMsbFirst & 0x01;

    regAddressMsbFirst >>= 1;

    clock();
}

if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Read address failed!", __func__);
    return FAIL;
}

start();

sendControlByteNoAddr(READ, deviceId);

if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s: Acknowledgement Failed", __func__);
    return FAIL;
}

uint8_t dataByte = 0;

for (int i = 0; i < 8; i++)
{
    SCL = 1;

    // Bit packing each bit into dataByte to give MCU the data from requested address from MSB to LSB(little endian)
    dataByte = (dataByte << 1) | SDA;

    NOP;

    SCL = 0;

    NOP;
}

sendNAcknowledge();

```

```

stop();

return dataByte;
}

/** 
 * @brief Function that contains logic for the Read transaction (multiple bytes) from the MAX30101
 * Refer to page 31, Figure 11 of datasheet
 *
 * @param deviceId
 * @param regAddr
 * @param dataArray
 * @param size
 * @return int
 */
int readBytesMAXWing(uint8_t deviceId, uint8_t regAddr, uint8_t dataArray[], uint8_t size)
{
    start();
    sendControlByteNoAddr(WRITE, deviceId);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s: Acknowledgement Failed", __func__);
        return FAIL;
    }

    uint8_t regAddressMsbFirst = reverse(regAddr);
    for (int i = 0; i < 8; i++)
    {
        SDA = regAddressMsbFirst & 0x01;
        regAddressMsbFirst >>= 1;
        clock();
    }
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Read address failed!", __func__);
        return FAIL;
    }

    start();
}

```

```

sendControlByteNoAddr(READ, deviceId);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s: Acknowledgement Failed", __func__);
    return FAIL;
}

for (uint8_t k = 0; k < size; k++)
{
    uint8_t dataByte = 0;
    for (int i = 0; i < 8; i++)
    {
        SCL = 1;
        // Bit packing each bit into dataByte to give MCU the data from requested address from MSB to LSB(little endian)
        dataByte = (dataByte << 1) | SDA;
        NOP;
        SCL = 0;
        NOP;
    }

    dataArray[k] = dataByte;

    // send ack for all recv bytes except last
    if (k < size - 1)
    {
        sendAcknowledge();
    } // send nack for lasy read byte
    else
    {
        sendNAcknowledge();
    }
}

stop();

return SUCCESS;
}

/**
 * @brief Logic for performing the write transaction for MAX30101

```

```

* Refer to Pg.30 Figure 9 for Write transaction

* @param deviceId
* @param regAddr
* @param regVal
* @return int
*/
int writeRegisterMAXWing(uint8_t deviceId, uint8_t regAddr, uint8_t regVal)
{
    start();
    sendControlByteNoAddr(WRITE, deviceId);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s: Acknowledgement Failed", __func__);
    }
    uint8_t regAddressMsbFirst = reverse(regAddr);
    for (int i = 0; i < 8; i++)
    {
        SDA = regAddressMsbFirst & 0x01;
        regAddressMsbFirst >>= 1;
        clock();
    }
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Read address failed!", __func__);
        return FAIL;
    }
    //Sending the data that needs to be written at addr
    uint8_t databyteMsbFirst = reverse(regVal);
    for (int i = 0; i < 8; i++)
    {
        SDA = databyteMsbFirst & 0x01;
        databyteMsbFirst >>= 1;
        clock();
    }
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Write data failed!", __func__);
        return FAIL;
    }
}

```

```

stop();
}

<**
 * @brief Tests the I2C read and write functionality
 *
 */
int testI2c()
{
    uint8_t regVal;
    // Get current register value so that nothing is overwritten.

    regVal = readRegisterMAXWing(MAX30101, 0x0A);
    INFO_LOG("LOG 1: Value at location: %u", regVal);
    writeRegisterMAXWing(MAX30101, 0x0A, 0x01); // Write Register
    regVal = readRegisterMAXWing(MAX30101, 0x0A);
    INFO_LOG("LOG 2: Value at location: %u", regVal);

}

/*****
 * Author - Venetia Furtado
 * Final Project: Health Monitoring System
 * ECEN 5613 - Spring 2024 - Prof. McClure
 * University of Colorado Boulder
 * Revised 04/26/2024
 *
 *****
 * This file contains the function to parse the received serial data into Bio Data values
*****/

/*****
//           INCLUDES
*****/
#include <mcs51/8051.h>
#include "at89c51ed2.h"
#include <stdio.h>
#include <stdint.h>

```

```

#include <stdlib.h>
#include <stdbool.h>
#include "io.h"
#include "log.h"
#include "datareceiver.h"

/***** FUNCTION DEFINITIONS & FORWARD DECLARATIONS *****/
// FUNCTION DEFINITIONS & FORWARD DECLARATIONS
/***** */

/***
 * @brief Function to convert enum value to string
 *
 * @param status
 * @return const char*
 */
const char* getStatusString(FingerStatus status)
{
    switch (status)
    {
        case NO_READING:
            return "NO READING";
        case NOT_READY:
            return "NOT READY";
        case OBJECT_DETECTED:
            return "OBJECT DETECTED";
        case FINGER_DETECTED:
            return "FINGER DETECTED";
        default:
            return "UNKNOWN";
    }
}

/***
 * @brief Parses biodata values from input string
 *
 * @param input
 * @param data
 */

```

```

void readBiodata(char input[], BioData* data)
{
    uint8_t status = 0;
    uint8_t idx = 0;
    uint8_t bpm = 0;
    uint8_t spo2 = 0;
    uint8_t irled = 0;

    while(input[idx] != '#')
    {
        // gives integer value
        uint8_t digit = input[idx] - '0';
        status = status*10 + digit;
        idx++;
    }

    INFO_LOG("%s: Status = %d", __func__, status);

    while(input[idx] != '#')
    {
        // gives integer value
        uint8_t digit = input[idx] - '0';
        bpm = bpm*10 + digit;
        idx++;
    }

    INFO_LOG("%s: BPM = %d", __func__, bpm);

    while(input[idx] != '#')
    {
        uint8_t digit = input[idx] - '0';
        //ascii to integer conversion
        spo2 = spo2*10 + digit;
        idx++;
    }

    INFO_LOG("%s: SPO2 = %d", __func__, spo2);
}

```

```
while(input[idx] != '#')
{
    uint8_t digit = input[idx] - '0';
    irled = irled*10 + digit;
    idx++;
}
INFO_LOG("%s: IRLED = %d", __func__, irled);

data->status = status;
data->bpm = bpm;
data->spo2 = spo2;
data->irled = irled;
}
```

6.0.4 Appendix - Firmware Source Code (ARM)

The following contains firmware source code for MAX32664 oximeter module

```

/************************************************************
 * Author - Venetia Furtado
 * Final Project: Health Monitoring System
 * ECEN 5613 - Spring 2024 - Prof. McClure
 * University of Colorado Boulder
 * Revised 04/26/2024
 *
 ************************************************************

 * This file contains the main function for the initialization of the functions
 * being used in the MAXHUB module
************************************************************/

```

```
/************************************************************/
```

```
//           INCLUDES
```

```
/************************************************************/
```

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
```

```
#include "main.h"
```

```
#include "define.h"
```

```
#include "init.h"
```

```
#include "io.h"
```

```
#include "buffer.h"
```

```
#include "rtc.h"
```

```
#include "i2c.h"
```

```
#include "maxhub.h"
```

```
/************************************************************/
```

```
//           FUNCTION DEFINITION
```

```
/************************************************************/
```

```
/**
```

```
* @brief Initialization of all functions
```

```
* @retval int
```

```
*/
```

```
int main(void)
```

```
{
```

```

// Initialization
HAL_Init();
SystemClock_Config();
UART2_Init();

printWelcomeMessage();

testI2C();
}

/******************
 * Author - Venetia Furtado
 * Final Project: Health Monitoring System
 * ECEN 5613 - Spring 2024 - Prof. McClure
 * University of Colorado Boulder
 * Revised 04/26/2024
 *
***** This file contains the initialization and configuration of the MAXHUB module.
* References:
https://github.com/sparkfun/SparkFun\_Bio\_Sensor\_Hub\_Library/blob/master/src/SparkFun\_Bio\_Sensor\_Hub\_Library.cpp
****/





//           INCLUDES
****/





#include "main.h"
#include <stdbool.h>
#include "i2c.h"
#include "gpio.h"
#include "io.h"
#include "define.h"
#include "maxhub.h"

#define MAX32664_ADDRESS 0x55

****/





//           FUNCTION DEFINITIONS
****/

```

```

/**
 * @brief Logic for the I2C write transaction for the MAX32664 for multiple bytes
 *
 * @param familyByte
 * @param indexByte
 * @param dataArray
 * @param size
 * @return int
 */

int i2cMax32664WriteBytes(uint8_t familyByte, uint8_t indexByte, uint8_t data[], uint8_t size)
{
    start();
    sendControlByte(WRITE, MAX32664_ADDRESS);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s:%d Write address failed!", __func__, __LINE__);
        return FAIL;
    }

    // send family byte
    i2cByteWrite(familyByte);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Family Byte Write address failed!", __func__);
        return FAIL;
    }

    // send index byte
    i2cByteWrite(indexByte);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Index Byte Write address failed!", __func__);
        return FAIL;
    }

    for (uint8_t i = 0; i < size; i++)
    {
        uint8_t dataByte = data[i];

```

```

i2cByteWrite(dataByte);

if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Data Byte Write %d failed!", __func__, i);
    return FAIL;
}

stop();

delay(CMD_DELAY);

start();
sendControlByte(READ, MAX32664_ADDRESS);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Read Byte failed!", __func__);
    return FAIL;
}

uint8_t statusByte = i2cByteRead();
sendNAcknowledge();
stop();

return statusByte;
}

/**
 * @brief Logic for the I2C write transaction for writing 1 byte
 *
 * @param familyByte
 * @param indexByte
 * @param dataByte
 * @return int
 */
int i2cMax32664WriteByte(uint8_t familyByte, uint8_t indexByte, uint8_t dataByte)
{
    return i2cMax32664WriteBytes(familyByte, indexByte, &dataByte, 1);
}

```

```

/**
 * @brief Logic for the I2C Read Transaction(specific to MAX32664)
 *
 * @param familyByte
 * @param indexByte
 * @param writeByte
 * @return int
 */

int i2cMax32664ReadByte(uint8_t familyByte, uint8_t indexByte, uint8_t *writeByte)
{
    start();
    sendControlByte(WRITE, MAX32664_ADDRESS);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Write address failed!", __func__);
        return FAIL;
    }

    // send family byte
    i2cByteWrite(familyByte);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Family Byte Write address failed!", __func__);
        return FAIL;
    }

    // send index byte
    i2cByteWrite(indexByte);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Index Byte Write address failed!", __func__);
        return FAIL;
    }

    // send write byte if necessary
    if (writeByte != NULL)
    {
        i2cByteWrite(*writeByte);
        if (checkAcknowledgement() == FAIL)

```

```

    {
        ERROR_LOG("%s Write Byte Write address failed!", __func__);
        return FAIL;
    }
}

stop();

delay(CMD_DELAY);

start();
sendControlByte(READ, MAX32664_ADDRESS);
if (checkAcknowledgement() == FAIL)
{
    ERROR_LOG("%s Read Byte failed!", __func__);
    return FAIL;
}

uint8_t statusByte = i2cByteRead();
sendAcknowledge();

if (statusByte != SFE_BIO_SUCCESS)
{
    return statusByte;
}

uint8_t readByte = i2cByteRead();
// send nack
sendNAcknowledge();

stop();

return readByte;
}

/**
 * @brief Logic for implementing the Sequential Read for MAX23664
 *
 * @param familyByte

```

```

* @param indexByte
* @param readArr
* @param size
* @return int
*/
int i2cMax32664SequentialReadByte(uint8_t familyByte, uint8_t indexByte, uint8_t readArr[], uint8_t size)
{
    start();
    sendControlByte(WRITE, MAX32664_ADDRESS);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Write address failed!", __func__);
        return FAIL;
    }

    // send family byte
    i2cByteWrite(familyByte);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Family Byte Write address failed!", __func__);
        return FAIL;
    }

    // send index byte
    i2cByteWrite(indexByte);
    if (checkAcknowledgement() == FAIL)
    {
        ERROR_LOG("%s Index Byte Write address failed!", __func__);
        return FAIL;
    }

    stop();

    delay(CMD_DELAY);

    start();
    sendControlByte(READ, MAX32664_ADDRESS);
    if (checkAcknowledgement() == FAIL)
    {

```

```

    ERROR__LOG("%s Read Byte failed!", __func__);
    return FAIL;
}

uint8_t statusByte = i2cByteRead();
sendAcknowledge();

if (statusByte != SFE_BIO_SUCCESS)
{
    ERROR__LOG("%s Status Byte = %X", __func__, statusByte);
    return statusByte;
}

for (uint8_t i = 0; i < size; i++)
{
    readArr[i] = 0;
}

for (uint8_t i = 0; i < size; i++)
{
    readArr[i] = i2cByteRead();

    if (i < size - 1)
    {
        sendAcknowledge();
    }
    else
    {
        // send nack after reading last byte
        sendNAcknowledge();
    }
}

stop();
return statusByte;
}

/**
 * @brief Family Byte: WRITE_REGISTER (0x40), Index Byte: WRITE_MAX30101 (0x03), Write Bytes:

```

```

* Register Address and Register Value
* This function writes the given register value at the given register address
* for the MAX30101 sensor and returns a boolean indicating a successful or
* non-successful write.
* @param regAddr
* @param regVal
* @return int
*/
int writeRegisterMAX30101(uint8_t regAddr, uint8_t regVal)
{
    uint8_t dataArr[2];
    dataArr[0] = regAddr;
    dataArr[1] = regVal;

    return i2cMax32664WriteBytes(WRITE_REGISTER, WRITE_MAX30101, dataArr, 2);
}

/** 
* @brief Family Byte: READ_REGISTER (0x41), Index Byte: READ_MAX30101 (0x03), Write Byte:
* Register Address
* This function reads the given register address for the MAX30101 Sensor and
* returns the values at that register.
* @param regAddr
* @return uint8_t
*/
uint8_t readRegisterMAX30101(uint8_t regAddr)
{
    uint8_t data = i2cMax32664ReadByte(READ_REGISTER, READ_MAX30101, &regAddr);
    return data;
}

/** 
* @brief Sets MAXHUB in application mode
*
*/
void setApplicationMode()
{

```

```

    i2cSetPin(RSTN, 0);
    i2cSetPin(MFIO, 1);
    delay(10);
    i2cSetPin(RSTN, 1);
    delay(50);
}

/** 
 * @brief Sets MAXHUB in Bootloader Mode
 *
 */
void setBootloaderMode()
{
    i2cSetPin(RSTN, 0);
    i2cSetPin(MFIO, 0);
    delay(10);
    i2cSetPin(RSTN, 1);
    delay(50);
}

/** 
 * @brief Family Byte: HUB_STATUS (0x00), Index Byte: 0x00, No Write Byte.
 * The following function checks the status of the FIFO.
 * @return uint8_t
 */
uint8_t readSensorHubStatus()
{
    uint8_t statusByte = i2cMax32664ReadByte(0x00, 0x00, NULL); // family and index byte.
    return statusByte; // Will return 0x00
}

/** 
 * @brief Family Byte: OUTPUT_MODE (0x10), Index Byte: SET_FORMAT (0x00),
 * Write Byte : outputType (Parameter values in OUTPUT_MODE_WRITE_BYTE)
 *
 * @param outputType
 * @return uint8_t
 */

```

```

uint8_t setOutputMode(uint8_t outputType)
{
    if (outputType > SENSOR_ALGO_COUNTER) // Bytes between 0x00 and 0x07
        return FAIL;

    // Check that communication was successful, not that the IC is outputting
    // correct format.
    int statusByte = i2cMax32664WriteByte(OUTPUT_MODE, SET_FORMAT, outputType);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        ERROR_LOG("%s: Received Error Status = %X", __func__, statusByte);
        return statusByte;
    }
    return SFE_BIO_SUCCESS;
}

/**
 * @brief Family Byte: OUTPUT_MODE(0x10), Index Byte: WRITE_SET_THRESHOLD (0x01), Write byte: intThres
 * This function changes the threshold for the FIFO interrupt bit/pin. The
 * interrupt pin is the MFIO pin which is set to INPUT after initialization.
 *
 *
 * @param intThresh
 * @return uint8_t
 */
uint8_t setFifoThreshold(uint8_t intThresh)
{
    uint8_t statusByte = i2cMax32664WriteByte(OUTPUT_MODE, WRITE_SET_THRESHOLD, intThresh);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        ERROR_LOG("%s: Received Error Status = %X", __func__, statusByte);
        return statusByte;
    }
    return SFE_BIO_SUCCESS;
}

/**
 * @brief Family Byte: ENABLE_ALGORITHM (0x52), Index Byte:ENABLE_AGCA_ALGO (0x00)

```

```

* This function enables (one) or disables (zero) the automatic gain control algorithm.
* @param enable
* @return uint8_t
*/
uint8_t agcAlgoControl(uint8_t enable)
{
    if (enable > 1)
    {
        return FAIL;
    }

    uint8_t statusByte = i2cMax32664WriteByte(ENABLE_ALGORITHM, ENABLE_AGC_ALGO, enable);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        ERROR_LOG("%s: Received Error Status = %X", __func__, statusByte);
    }
    return SFE_BIO_SUCCESS;
}

/** 
* @brief Family Byte: ENABLE_SENSOR (0x44), Index Byte: ENABLE_MAX30101 (0x03), Write
* Byte: senSwitch (parameter - 0x00 or 0x01).
* This function enables the MAX30101.
* @param senSwitch
* @return uint8_t
*/
uint8_t max30101Control(uint8_t senSwitch)
{
    if (senSwitch > 1)
    {
        return FAIL;
    }

    // Check that communication was successful, not that the sensor is enabled.
    uint8_t statusByte = i2cMax32664WriteByte(ENABLE_SENSOR, ENABLE_MAX30101, senSwitch);
    if (statusByte != SFE_BIO_SUCCESS)
    {

```

```

    ERROR__LOG("%s: Recieved Error Status = %X", __func__, statusByte);
}

return SFE_BIO_SUCCESS;
}

/** 
 * @brief Family Byte: ENABLE_ALGORITHM (0x52), Index Byte:ENABLE_WHRM_ALGO (0x02)
 * This function enables (one) or disables (zero) the wrist heart rate monitor
 * algorithm.
 * @param mode
 * @return uint8_t
 */

uint8_t maximFastAlgoControl(uint8_t mode)
{
    if (mode > 2)
    {
        return FAIL;
    }

    uint8_t statusByte = i2cMax32664WriteByte(ENABLE_ALGORITHM, ENABLE_WHRM_ALGO, mode);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        ERROR__LOG("%s: Recieved Error Status = %X", __func__, statusByte);
    }
    return SFE_BIO_SUCCESS;
}

/** 
 * @brief Logic for configuring the MAXHUB only for algo data(heart rate & SPO2)
 *
 * @param mode
 */
void configBpm(uint8_t mode)
{
    uint8_t statusByte;

    statusByte = setOutputMode(ALGO_DATA); // Just the data
    INFO__LOG("setOutputMode Status = %X", statusByte);
}

```

```

if (statusByte != SFE_BIO_SUCCESS)
{
    //return;
}

statusByte = setOutputMode(ALGO_DATA); // Just the data
INFO_LOG("setOutputMode Status 2 = %X", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    //return;
}

statusByte = setFifoThreshold(0x01); // One sample before interrupt is fired.
INFO_LOG("setFifoThreshold Status = %X", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    //return;
}

statusByte = agcAlgoControl(ENABLE);
INFO_LOG("agcAlgoControl Status = %X", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    //return;
}

statusByte = max30101Control(ENABLE);
INFO_LOG("max30101Control Status = %X", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    //return;
}

statusByte = maximFastAlgoControl(mode);
INFO_LOG("maximFastAlgoControl Status = %X", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    //return;
}

```

```

delay(2000);

INFO_LOG("%s: configuration success", __func__);

}

/** 
 * @brief Logic for configuring the MAX32664 with MAX30101 sensor data
 *
 * @param mode
 */
void configSensorBpm(uint8_t mode)
{
    int statusByte;

    statusByte = setOutputMode(SENSOR_AND_ALGORITHM);
    INFO_LOG("setOutputMode Status = %d", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        //return;
    }

    statusByte = setOutputMode(SENSOR_AND_ALGORITHM);
    INFO_LOG("setOutputMode Status = %d", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        // return;
    }

    statusByte = setFifoThreshold(0x01); // One sample before interrupt is fired.
    INFO_LOG("setFifoThreshold Status = %X", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {
        // return;
    }

    statusByte = agcAlgoControl(ENABLE);
    INFO_LOG("agcAlgoControl Status = %X", statusByte);
    if (statusByte != SFE_BIO_SUCCESS)
    {

```

```

// return;
}

statusByte = max30101Control(ENABLE);
INFO_LOG("max30101Control Status = %X", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    // return;
}

statusByte = maximFastAlgoControl(mode);
INFO_LOG("maximFastAlgoControl Status = %X", statusByte);
if (statusByte != SFE_BIO_SUCCESS)
{
    // return;
}

delay(2000);
INFO_LOG("%s: configuration success", __func__);
}

/**
 * @brief Family Byte: READ_DATA_OUTPUT (0x12), Index Byte: NUM_SAMPLES (0x00), Write
 * Byte: NONE
 * This function returns the number of samples available in the FIFO.
 * @return uint8_t
 */
uint8_t numSamplesOutFifo()
{
    uint8_t sampAvail = i2cMax32664ReadByte(READ_DATA_OUTPUT, NUM_SAMPLES, NULL);
    return sampAvail;
}

//#define STATUS_LOG

/**
 * @brief Logic similar to readSensor Bpm function except that this function only read the
 * SP02 and heart rate values (algo data)
 *

```

```

* @param mode
* @param data
* @return int
*/
int readBpm(uint8_t mode, BioData *data)
{
    uint8_t hubStatus = readSensorHubStatus();
    if (hubStatus == 1)
    {
        ERROR_LOG("%s: Hub Status Error : %d", __func__, hubStatus);
        return FAIL;
    }

    if (numSamplesOutFifo() == 0)
    {
        //INFO_LOG("%s: No samples to read", __func__);
        return FAIL;
    }

    int statusByte;
    uint8_t dataArr[MAXFAST_ARRAY_SIZE + MAXFAST_EXTENDED_DATA];
    if (mode == 1)
    {
        statusByte = i2cMax32664SequentialReadByte(READ_DATA_OUTPUT, READ_DATA, dataArr, MAXFAST_ARRAY_SIZE);
    }
    else if (mode == 2)
    {
        statusByte = i2cMax32664SequentialReadByte(READ_DATA_OUTPUT, READ_DATA, dataArr, MAXFAST_ARRAY_SIZE +
MAXFAST_EXTENDED_DATA);
    }

    if (statusByte != SFE_BIO_SUCCESS)
    {
        ERROR_LOG("%s: Status Error : %d", __func__, statusByte);
        return FAIL;
    }

    // Heart Rate formatting
    uint16_t heartRate = dataArr[0];

```

```

heartRate = (heartRate << 8) | (dataArr[1]);
heartRate /= 10;

// Confidence formatting
uint8_t confidence = dataArr[2];

// Blood oxygen level formatting
uint16_t oxygen = dataArr[3];
oxygen = (oxygen << 8) | dataArr[4];
oxygen /= 10;

// "Machine State" - has a finger been detected?
uint8_t status = dataArr[5];

data->heartRate = heartRate;
data->confidence = confidence;
data->oxygen = oxygen;
data->status = status;

if (mode == 2)
{
    // Sp02 r Value formatting
    uint16_t temp = dataArr[6];
    temp = (temp << 8) | dataArr[7];
    float rValue = temp;
    rValue = rValue / 10.0;

    // Extended Machine State formatting
    uint8_t extStatus = dataArr[8];

    data->rValue = rValue;
    data->extStatus = extStatus;
}

#ifndef STATUS_LOG
switch (status)
{
    case NO_READING:
        INFO_LOG("%s: Finger Status = NO_READING", __func__);
        break;
}

```

```

case NOT_READY:
    INFO_LOG("%s: Finger Status = NOT_READY", __func__);
    break;
case OBJECT_DETECTED:
    INFO_LOG("%s: Finger Status = OBJECT_DETECTED", __func__);
    break;
case FINGER_DETECTED:
    INFO_LOG("%s: Finger Status = FINGER_DETECTED", __func__);
    break;
default:
    ERROR_LOG("Unknown case!");
}
#endif
return PASS;
}

```

```

/**
 * @brief Logic to read the status of the FIFO register (refer to Table 7, pg.25)
 * of the MAX32664
 * @param mode
 * @param data
 * @return int
 */

```

```

int readSensorBpm(uint8_t mode, BioData *data)
{
    uint8_t hubStatus = readSensorHubStatus();
    if (hubStatus == 1)
    {
        ERROR_LOG("%s: Hub Status Error : %d", __func__, hubStatus);
        return FAIL;
    }

```

```
// INFO_LOG("Status Byte = %d", hubStatus);
```

```

    uint8_t numSamples = numSamplesOutFifo();
    // INFO_LOG("numSamples = %d", numSamples);
    if (numSamples == 0)
    {
        // INFO_LOG("%s: No samples to read", __func__);

```

```

    return FAIL;
}

int statusByte;
//Refer to Pg. 25 of 32664 datasheet (Table 7)
uint8_t dataArr[MAXFAST_ARRAY_SIZE + MAX30101_LED_ARRAY + MAXFAST_EXTENDED_DATA];
if (mode == 1)
{
    statusByte = i2cMax32664SequentialReadByte(READ_DATA_OUTPUT, READ_DATA, dataArr, MAXFAST_ARRAY_SIZE +
MAX30101_LED_ARRAY);
}
else if (mode == 2)
{
    statusByte = i2cMax32664SequentialReadByte(READ_DATA_OUTPUT, READ_DATA, dataArr, MAXFAST_ARRAY_SIZE +
MAX30101_LED_ARRAY + MAXFAST_EXTENDED_DATA);
}

if (statusByte != SFE_BIO_SUCCESS)
{
    ERROR_LOG("%s: Status Error : %d", __func__, statusByte);
    return FAIL;
}

// Value of LED2(IR)
uint32_t irLed = dataArr[0];
irLed = (irLed << 8) | dataArr[1];
irLed = (irLed << 8) | dataArr[2];

// Value of LED1(RED)
uint32_t redLed = dataArr[3];
redLed = (redLed << 8) | dataArr[4];
redLed = (redLed << 8) | dataArr[5];

/*There are two uint32_t values that are given by
the sensor for LEDs that do not exists on the MAX30101. So we have to
request those empty values because they occupy the buffer:
bpmSenArr[6-11]*/
// Heart Rate formatting

```

```

    uint16_t heartRate = dataArr[12];
    heartRate = (heartRate << 8) | (dataArr[13]);
    heartRate /= 10;

    // Confidence formatting
    uint8_t confidence = dataArr[14];

    // Blood oxygen level formatting
    uint16_t oxygen = dataArr[15];
    oxygen = (oxygen << 8) | dataArr[16];
    oxygen /= 10;

    // "Machine State" - has a finger been detected?
    uint8_t status = dataArr[17];

    data->irLed = irLed;
    data->redLed = redLed;

    data->heartRate = heartRate;
    data->confidence = confidence;
    data->oxygen = oxygen;
    data->status = status;

    if (mode == 2)
    {
        // SpO2 r Value formatting
        uint16_t temp = dataArr[18];
        temp = (temp << 8) | dataArr[19];
        float rValue = temp;
        rValue = rValue / 10.0;

        // Extended Machine State formatting
        uint8_t extStatus = dataArr[20];

        data->rValue = rValue;
        data->extStatus = extStatus;
    }

#endif STATUS_LOG
switch (status)

```

```

{
    case NO_READING:
        INFO_LOG("NO_READING");
        break;
    case NOT_READY:
        INFO_LOG("NOT_READY");
        break;
    case OBJECT_DETECTED:
        INFO_LOG("OBJECT_DETECTED");
        break;
    case FINGER_DETECTED:
        INFO_LOG("FINGER_DETECTED");
        break;
    default:
        ERROR_LOG("UNKNOWN");
}
#endif
return PASS;
}

/** @brief This function modifies the pulse width of the MAX30101 LEDs. All of the LEDs
 * are modified to the same width. This will affect the number of samples that
 * can be collected and will also affect the ADC resolution.
 * Default: 69us - 15 resolution - 50 samples per second.
 * Register: 0x0A, bits [1:0]
 * Width(us) - Resolution - Sample Rate
 * 69us - 15 - <= 3200 (fastest - least resolution)
 * 118us - 16 - <= 1600
 * 215us - 17 - <= 1600
 * 411us - 18 - <= 1000 (slowest - highest resolution)
 * @param width
 * @return int
 */
int setPulseWidth(uint16_t width)
{
    uint8_t bits;
    uint8_t regVal;

```

```

// Make sure the correct pulse width is selected.

if (width == 69)
    bits = 0; // 00
else if (width == 118)
    bits = 1; // 01
else if (width == 215)
    bits = 2; // 10
else if (width == 411)
    bits = 3; // 11
else
    return FAIL;

// Get current register value so that nothing is overwritten.

regVal = readRegisterMAX30101(CONFIGURATION_REGISTER);

regVal &= PULSE_MASK;           // Mask bits to change.

regVal |= bits;                // Add bits

writeRegisterMAX30101(CONFIGURATION_REGISTER, regVal); // Write Register

return SFE_BIO_SUCCESS;
}

/***
 * @brief This function reads the CONFIGURATION_REGISTER (0x0A), bits [1:0] from the
 * MAX30101 Sensor. It returns one of the four settings in microseconds.
 *
 * @return int
 */

int readPulseWidth()
{
    uint8_t regVal;

    regVal = readRegisterMAX30101(CONFIGURATION_REGISTER);

    regVal &= READ_PULSE_MASK;

    if (regVal == 0)
        return 69;
    else if (regVal == 1)

```

```

    return 118;
else if (regVal == 2)
    return 215;
else if (regVal == 3)
    return 411;
else
    return FAIL;
}

/** 
 * @brief This function changes the sample rate of the MAX30101 sensor. The sample
 * rate is affected by the set pulse width of the MAX30101 LEDs.
 * Default: 69us - 15 resolution - 50 samples per second.
 * Register: 0x0A, bits [4:2]
 * Width(us) - Resolution - Sample Rate
 * 69us   - 15   - <= 3200 (fastest - least resolution)
 * 118us  - 16   - <= 1600
 * 215us  - 17   - <= 1600
 * 411us  - 18   - <= 1000 (slowest - highest resolution)
 * @param sampRate
 * @return int
 */
int setSampleRate(uint16_t sampRate)
{
    uint8_t bits;
    uint8_t regVal;

    // Make sure the correct sample rate was picked
    if (sampRate == 50)
        bits = 0;
    else if (sampRate == 100)
        bits = 1;
    else if (sampRate == 200)
        bits = 2;
    else if (sampRate == 400)
        bits = 3;
    else if (sampRate == 800)
        bits = 4;
}

```

```

else if (sampRate == 1000)
    bits = 5;
else if (sampRate == 1600)
    bits = 6;
else if (sampRate == 3200)
    bits = 7;
else
    return FAIL;

// Get current register value so that nothing is overwritten.
regVal = readRegisterMAX30101(CONFIGURATION_REGISTER);
regVal &= SAMP_MASK;                                // Mask bits to change.
regVal |= (bits << 2);                            // Add bits but shift them first to correct position.
writeRegisterMAX30101(CONFIGURATION_REGISTER, regVal); // Write Register

return SFE_BIO_SUCCESS;
}

/** 
 * @brief This function reads the CONFIGURATION_REGISTER (0x0A), bits [4:2] from the
 * MAX30101 Sensor. It returns one of the 8 possible sample rates.
 *
 * @return int
 */
int readSampleRate()
{
    uint8_t regVal;

    regVal = readRegisterMAX30101(CONFIGURATION_REGISTER);
    regVal &= READ_SAMP_MASK;
    regVal = (regVal >> 2);

    if (regVal == 0)
        return 50;
    else if (regVal == 1)
        return 100;
    else if (regVal == 2)
        return 200;
}

```

```

else if (regVal == 3)
    return 400;
else if (regVal == 4)
    return 800;
else if (regVal == 5)
    return 1000;
else if (regVal == 6)
    return 1600;
else if (regVal == 7)
    return 3200;
else
    return FAIL;
}

```

```
#define DATA_SIZE 10
```

```

typedef struct
{
    uint32_t minInput;
    float scaleFactor;
} Scaler;
```

```

/**
 * @brief Logic to calculate the scale factor
 *
 * @param input
 * @param size
 * @param scaleval
 */
void scale_uint32_to_uint8(uint32_t *input, int size, Scaler* scaleval) {
    uint32_t max_input = 0;
    uint32_t minInput = UINT32_MAX;
    float scaleFactor;

    // Find min and max values in the dataset
    for (int i = 0; i < size; i++)
    {
        if (input[i] > max_input)
        {

```

```

        max_input = input[i];
    }
    if (input[i] < minInput)
    {
        minInput = input[i];
    }
}

// Calculate scale factor
scaleFactor = (float)(UINT8_MAX) / (max_input - minInput);

scaleval->minInput = minInput;
scaleval->scaleFactor = scaleFactor;
}

//#define BPM_DATA
#define SENSOR_DATA

#define PLOT_DATA
//#define SHOW_DATA

/**
 * @brief Logic to test out the working of the MAXHUB configurations
 *
 */
void testI2C()
{
    i2cInitGpio();
    delay(1000);
    setApplicationMode();
    // setBootloaderMode();
    delay(1000);
    i2cSetPin(MFIO, 0);
    uint8_t mode = 1;

#ifndef BPM_DATA
    configBpm(mode);
#endif
}

```

```

#ifndef SENSOR_DATA
configSensorBpm(mode);

uint16_t width = 118;
setPulseWidth(width);

uint16_t sampleRate = 1600;
setSampleRate(sampleRate);

#endif
BioData data;

uint32_t irTuningData[DATA_SIZE];
uint32_t counter = 0;
bool tuningComplete = false;
Scaler scaleValue;

while (1)
{
#ifndef BPM_DATA
int status = readBpm(mode, &data);
if (status == PASS)
{
// INFO_LOG("CONFIDENCE = %u", data.confidence);
if (data.confidence > 50)
{
#endif
#ifndef PLOT_DATA
LOG("%d%d|\n", data.heartRate, data.oxygen);
delay(1000);
#endif
#endif
#ifndef SHOW_DATA
INFO_LOG("HEART-RATE = %u", data.heartRate);
INFO_LOG("OXYGEN = %u", data.oxygen);

if (mode == 2)
{
INFO_LOG("rValue = %f", data.rValue);
INFO_LOG("Extended Status = %u", data.extStatus);
}
data.confidence = 0;

```

```

#endif
}

}

#endif

#ifndef SENSOR__DATA
int status = readSensorBpm(mode, &data);
if (status == PASS)
{
    // INFO_LOG("CONFIDENCE = %u", data.confidence);
    if (data.confidence > 50)
    {
#endif

#ifndef PLOT__DATA
if (tuningComplete == false)
{
    if (counter < DATA__SIZE)
    {
        INFO_LOG("Collecting tuning data ,sample = %d, val = %ld", counter, data.irLed);
        irTuningData[counter] = data.irLed;
        counter++;
    }
    else
    {
        INFO_LOG("Calculating tuning data")
        scale_uint32_to_uint8(irTuningData, DATA__SIZE, &scaleValue);
        tuningComplete = true;
    }
}
else
{
    uint8_t irvalue = (uint8_t)((data.irLed - scaleValue.minInput) * scaleValue.scaleFactor);
    LOG("%d%d%d%d\n", data.status, data.heartRate, data.oxygen, irvalue);
}
#endif

#endif

#ifndef SHOW__DATA
INFO_LOG("IR LED = %lu", data.irLed);
INFO_LOG("RED LED = %lu", data.redLed);

INFO_LOG("HEART-RATE = %u", data.heartRate);

```

```

    INFO_LOG("OXYGEN = %u", data.oxygen);

    if (mode == 2)
    {
        INFO_LOG("rValue = %f", data.rValue);
        INFO_LOG("Extended Status = %u", data.extStatus);
    }
#endif
data.confidence = 0;
}
}

#endif
}

i2cSetPin(MFIO, 1);
}

```

```

*****
* Author - Venetia Furtado
* Final Project: Health Monitoring System
* ECEN 5613 - Spring 2024 - Prof. McClure
* University of Colorado Boulder
* Revised 04/26/2024
*
*****
* This file contains code to set the GPIOA registers MODER, OTYPER, OSPEEDR and
* PUPDR to the appropriate functions. These were used by the pulse oximeter
* module pins RST, MFIO, SCL and SDA.
* Reference: https://github.com/embeddedalpha/STM32F407Vx/blob/main/Src/Inc/GPIO/GPIO.c
*****
//           INCLUDES
*****
#include "gpio.h"
#include "stm32f4xx_hal.h"
*****
//           FUNCTION DEFINITION
*****
/**
```

```

* @brief Sets a GPIO pin to the appropriate function
*
* @param Port
* @param pin
* @param function
* @param alternate_function
*/
void GPIO_Pin_Setup(char Port, uint8_t pin, uint8_t function, uint8_t alternate_function)
{
    if (Port == 'A' || Port == 'a')
    {
        RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

        GPIOA->MODER |= ((0xC0 & function) >> 6) << (2 * pin); // MODER sets I/O direction
        GPIOA->OTYPER |= ((0x30 & function) >> 4) << (1 * pin); // Configures the output type of the I/O port
        GPIOA->OSPEEDR |= ((0x0C & function) >> 2) << (2 * pin); // configure the I/O output speed
        GPIOA->PUPDR |= ((0x03 & function) >> 0) << (2 * pin); // configure the I/O pull-up or pull-down

        if (alternate_function != NONE)
        {
            if (pin < 8)
                GPIOA->AFR[0] |= (alternate_function << (4 * (pin)));
            else
                GPIOA->AFR[1] |= (alternate_function << (4 * (pin - 8)));
        }
    }

    if (Port == 'B' || Port == 'b')
    {
        RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;

        GPIOB->MODER |= ((0xC0 & function) >> 6) << (2 * pin);
        GPIOB->OTYPER |= ((0x30 & function) >> 4) << (1 * pin);
        GPIOB->OSPEEDR |= ((0x0C & function) >> 2) << (2 * pin);
        GPIOB->PUPDR |= ((0x03 & function) >> 0) << (2 * pin);

        if (alternate_function != NONE)
        {
            if (pin < 8)
                GPIOB->AFR[0] |= (alternate_function << (4 * (pin)));
            else

```

```

        GPIOB->AFR[1] |= (alternate_function << (4 * (pin - 8)));
    }
}

if (Port == 'C' || Port == 'c')
{
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN;
    GPIOC->MODER |= ((0xC0 & function) >> 6) << (2 * pin);
    GPIOC->OTYPER |= ((0x30 & function) >> 4) << (1 * pin);
    GPIOC->OSPEEDR |= ((0x0C & function) >> 2) << (2 * pin);
    GPIOC->PUPDR |= ((0x03 & function) >> 0) << (2 * pin);

    if (alternate_function == NONE)
    {
        if (pin < 8)
            GPIOC->AFR[0] |= (alternate_function << (4 * (pin)));
        else
            GPIOC->AFR[1] |= (alternate_function << (4 * (pin - 8)));
    }
}
}

```

```

*****
* Author - Venetia Furtado
* Final Project: Health Monitoring System
* ECEN 5613 - Spring 2024 - Prof. McClure
* University of Colorado Boulder
* Revised 04/26/2024
*
*****
* This file contains the function to implement I2C communication for the
* MAXHUB module
*****

```

```

*****
//           INCLUDES
*****
#include "main.h"
#include "i2c.h"
#include "gpio.h"

```

```

#include "io.h"

#define NOP for(int k = 0; k < 5000; k++)

/*MODER[7:6] = 01
OTYPER[5:4] = 01
OSPEEDR[3:2] = 01
PUPDR[1:0] = 01*/
#define GPIO_OUTPUT_OPENDRAIN 0x51
//********************************************************************/
// FUNCTION DEFINITIONS
//********************************************************************/

/***
 * @brief Logic to provide delay
 *
 * @param val
 */
void delay(int val)
{
    HAL_Delay(val);
}

/***
 * @brief Logic to initialize the GPIO pins to be used as SCL, SDA, MFIO, RSTN
 *
 */
void i2cInitGpio()
{
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;

    // changed from ALTERNATE_FUNCTION_OUTPUT_OPENDRAIN to 0x91 (low speed)
    GPIO_Pin_Setup('B', SCL, GPIO_OUTPUT_OPENDRAIN, NONE);
    GPIO_Pin_Setup('B', SDA, GPIO_OUTPUT_OPENDRAIN, NONE);
    GPIO_Pin_Setup('B', MFIO, GPIO_OUTPUT_OPENDRAIN, NONE);
    GPIO_Pin_Setup('B', RSTN, GPIO_OUTPUT_OPENDRAIN, NONE);
}

/***

```

```

* @brief Reads the status of a input GPIO pin
*
* @param pin
* @return uint8_t
*/
uint8_t i2cReadPin(int pin)
{
    uint8_t val = (GPIOB->IDR >> pin) & 0x01;
    //INFO_LOG("GPIOB->IDR = %lx", GPIOB->IDR);
    return val;
}

/** 
* @brief Logic to set/reset the bit (pin) of the BSRR
*
* @param pin
* @param val 1: set; 0:reset
*/
void i2cSetPin(int pin, uint8_t val)
{
    uint32_t pinMask = (0b01 << pin);
    if (val == 1)
    {
        GPIOB->BSRR |= pinMask;
    }
    else
    {
        GPIOB->BSRR |= pinMask << 16;
    }
}

/** 
* @brief Toggles the clock
*
*/
void clock()
{
    NOP;
    // SCL = 1;
}

```

```

i2cSetPin(SCL, 1);
NOP;
NOP;
// SCL = 0;
i2cSetPin(SCL, 0);
NOP;
}

/** 
* @brief Logic for starting the I2C communication
*
*/
void start()
{
    // SDA = 1;
    i2cSetPin(SDA, 1);
    NOP;
    // SCL = 1;
    i2cSetPin(SCL, 1);
    NOP;
    // SDA = 0;
    i2cSetPin(SDA, 0);
    NOP;
    NOP;
    // SCL = 0;
    i2cSetPin(SCL, 0);
}

/** 
* @brief Logic to stop the I2C communication
*
*/
void stop()
{
    // SDA = 0;
    i2cSetPin(SDA, 0);
    NOP;
    NOP;
    // SCL = 1;
}

```

```
i2cSetPin(SCL, 1);
NOP;
NOP;
// SDA = 1;
i2cSetPin(SDA, 1);
NOP;
NOP;
}
```

```
/***
 * @brief Logic to send Acknowledge(from Master)
 *
 */
```

```
void sendAcknowledge()
{
    // SDA = 0;
    i2cSetPin(SDA, 0);
    NOP;
    NOP;
    clock();
    // SDA = 1;
    i2cSetPin(SDA, 1);
    NOP;
    NOP;
}
```

```
/***
 * @brief Logic to send Nack
 *
 */
```

```
void sendNAcknowledge()
{
    // SDA = 0;
    i2cSetPin(SDA, 1);
    NOP;
    NOP;
    clock();
    NOP;
    NOP;
```

```

}

/** 
 * @brief Logic to reverse a byte; LSB of input become MSB of output.
 * @param input
 * @return uint8_t
 */
uint8_t reverse(uint8_t input)
{
    uint8_t output = 0;

    for (int i = 0; i < 8; i++)
    {
        output = output << 1;
        // Bitwise AND-ing input with 0x01 to obtain its LSB and OR with output
        output = output | (input & 0x01);
        input = input >> 1;
    }
    return output;
}

/** 
 * @brief Logic for sending the Control Byte
 *
 * @param op enum READ/WRITE
 * @param deviceId device ID to be read/write from
 */
void sendControlByte(const Operation op, uint8_t deviceId)
{
    // Sending the device address
    for (int i = 0; i < 7; i++)
    {
        // sets SDA pin high
        i2cSetPin(SDA, deviceId & 0x01);
        clock();
        deviceId = deviceId >> 1;
    }

    //Selects Read or Write operation

```

```

    i2cSetPin(SDA, op);
    clock();
}

/** 
 * @brief Checks if an acknowledgement has been given by the peripheral
 *
 * @return int SUCCESS or FAIL
 */
int checkAcknowledgement()
{
    // check to see if the device is acking by pulling SDA low
    i2cSetPin(SDA, 1);
    NOP;
    NOP;
    i2cSetPin(SCL, 1);
    NOP;
    // Capturing value at SDA
    uint8_t value = i2cReadPin(SDA);
    NOP;
    // SCL = 0;
    i2cSetPin(SCL, 0);
    NOP;
    if (value != 0)
    {
        return FAIL;
    }
    return SUCCESS;
}

/** 
 * @brief Write a Byte over I2C
 *
 * @param byte
 */
void i2cByteWrite(uint8_t byte)
{
    uint8_t byteMsbFirst = reverse(byte);

```

```

//Sending byteMsbFirst
for (int i = 0; i < 8; i++)
{
    //SDA = byteMsbFirst & 0x01;
    i2cSetPin(SDA, byteMsbFirst & 0x01);

    byteMsbFirst >>= 1;
    clock();
}

}

/***
 * @brief Reads a byte from the I2C transaction
 *
 * @return uint8_t
 */
uint8_t i2cByteRead()
{
    uint8_t dataByte = 0;

    for (int i = 0; i < 8; i++)
    {
        // SCL = 1;
        i2cSetPin(SCL, 1);

        // Bit packing each bit into dataByte to give MCU the data from requested address from MSB to LSB(little endian)
        uint8_t sdaReadData = i2cReadPin(SDA);
        dataByte = (dataByte << 1) | sdaReadData;

        NOP;
        //SCL = 0;
        i2cSetPin(SCL, 0);
        NOP;
    }

    return dataByte;
}

```

* Author - Venetia Furtado
 * Final Project: Health Monitoring System
 * ECEN 5613 - Spring 2024 - Prof. McClure
 * University of Colorado Boulder
 * Revised 04/26/2024

```

*
*****
*This file contains the initializations of all the functions required for
*error handling and configuring the system clock.
*****/



/***** INCLUDES *****/
// INCLUDES
/*****/



#include "stm32f4xx_hal.h"

/***** FUNCTION DEFINITIONS *****/
// FUNCTION DEFINITIONS
/*****



/**
 * @brief This function is executed in case of error occurrence.
 * @retval None
 */
void Error_Handler(void)
{
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the HAL error return state */
    __disable_irq();
    while (1)
    {
    }
    /* USER CODE END Error_Handler_Debug */
}

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

```



```
* Author - Venetia Furtado
* Final Project: Health Monitoring System
* ECEN 5613 - Spring 2024 - Prof. McClure
* University of Colorado Boulder
* Revised 04/26/2024
```

```
*
```

```
*****
```

```
* This file contains all functions that implement UART functionality.
```

```
******/
```

```
******/
```

```
//           INCLUDES
```

```
******/
```

```
#include <string.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
```

```
#include "define.h"
#include "io.h"
#include "buffer.h"
#include "stm32f4xx_hal.h"
```

```
******/
```

```
//           GLOBAL VARIABLES
```

```
******/
```

```
UART_HandleTypeDef huart2;          //Stores the UART2 configuration
BufferType bufferRx;              //Circular buffer for RX
BufferType bufferTx;              //Circular buffer for TX
```

```
******/
```

```
//           FUNCTION DEFINITIONS
```

```
******/
```

```
/**
```

```
* @brief USART2 Initialization Function
```

```
* @param None
```

```

* @retval None
*/
void UART2_Init(void)
{
#if 1
/* Enable USART2 interrupt */
HAL_NVIC_SetPriority(USART2_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(USART2_IRQn);

/* USER CODE END USART2_Init 1 */

huart2.Instance = USART2;
huart2.Init.BaudRate = 9600;
huart2.Init.WordLength = UART_WORDLENGTH_8B;
huart2.Init.StopBits = UART_STOPBITS_1;
huart2.Init.Parity = UART_PARITY_NONE;
huart2.Init.Mode = UART_MODE_TX_RX;
huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart2.Init.OverSampling = UART_OVERSAMPLING_16;
HAL_UART_Init(&huart2);
#endif

#if 0
NVIC_SetPriority(USART2_IRQn, 1);
NVIC_EnableIRQ(USART2_IRQn);

//Enables USART2 and GPIOA clock
RCC->APB1ENR |= RCC_APB1ENR_USART2EN;
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

GPIOA->MODER &= ~(0xFU << 4); // Reset bits 4:5 for PA2 and 6:7 for PA3
GPIOA->MODER |= (0xAU << 4); //configures PA2 and PA3 as Alternate function pins

GPIOA->AFR[0] |= (0x7 << 8); // Sets pin A2 to USART2 TX alternate function
GPIOA->AFR[0] |= (0x7 << 12); // Sets pin A3 to USART2 RX alternate function

USART2->BRR |= (52 << 4); //Sets Baud rate to 9600
USART2->BRR |= 15;

```

```

// USART2 word length M, bit 12
USART2->CR1 |= (0 << 12); // 0 - 1,8,n

USART2->CR1 |= (1 << 13);      //Enables USART2
USART2->CR1 |= (1 << 3);      //Transmitter enable
USART2->CR1 |= (1 << 2);      //Receiver enable

USART2->CR1 |= USART_CR1_RXNEIE; //enables the receive interrupt
USART2->CR1 |= USART_CR1_TXEIE; //enables the transmit interrupt*/



#endif

USART2->CR1 |= (1 << 13);      //Enables USART2
USART2->CR1 |= (1 << 3);      //Transmitter enable
USART2->CR1 |= (1 << 2);      //Receiver enable

initBufferWithDefaultValues(&bufferRx);      // Initializes circular buffer with default values
initBufferWithDefaultValues(&bufferTx);      // Initializes circular buffer with default values

USART2->CR1 |= USART_CR1_RXNEIE;      // enables the receive interrupt
USART2->CR1 |= USART_CR1_TXEIE;      //enables the transmit interrupt


}

/***
* @brief This function handles ISR for USART2 interrupt
*
*/
void USART2_IRQHandler(void)
{
    if(USART2->SR & USART_SR_RXNE)      //Checks if this is a RX interrupt
    {
        char c = USART2->DR;          //Copies the value form the data register of USART into c; hardware automatically resets the flag
when read
        putCharIntoBuffer(&bufferRx, c); //Puts the value of c into the circular RX buffer
    }
}

```

```

if (USART2->SR & USART_SR_TXE)      //Checks if this is a TX interrupt
{
    char c;

    int status = getCharFromBuffer(&bufferTx, &c); //Checks if the TX circular buffer has any data to be transmitted
    if (status == PASS)
    {
        USART2->DR = c;                      //If there is data, puts it on the terminal (DR is similar to SBUF of 8051)
    }
    else
    {
        USART2->CR1 &= ~(USART_CR1_TXEIE); //Else disable the transmit interrupt
    }
}

/** @brief This function puts one character into the TX circular buffer
 */
/* @param One character from msg
 */
void putChar(char c)
{
    putCharIntoBuffer(&bufferTx, c);
}

/** @brief This function contains an asynchronous (non-blocking) implementation of the printf function
 */
/* @param char buffer to be transmitted
 */
void printfASync(const char* msg)      //puts data on the TX circular buffer
{
    uint16_t len = strlen(msg);          // gets length of string
    for (uint16_t i = 0; i < len; i++)
    {
        putChar(msg[i]);              // puts msg into putCharIntoBuffer
    }
}

USART2->CR1 |= USART_CR1_TXEIE;      //enables the transmit interrupt

```

```

}

/** 
 * @brief This function is called when the user presses the RESET button
 *
 */
void printWelcomeMessage(void)
{
    LOG("\n\r033[48;5;4m" "Welcome to ARM" "\033[0m");
}

/** 
 * @brief Receives one character at a time from RX circular buffer
 *
 * @param pointer to a valid char location
 * @return int
 */
int readUserInput(char *c)
{
    int status = getCharFromBuffer(&bufferRx, c);
    return status;
}

uint16_t getUserNumberInput(uint8_t maxDigits, uint16_t maxVal) //max number of digits user can input
{
    //take only a maximum n digit input (3 in this case)
    uint16_t value = 0;
    while (1)
    {
        value = 0;
        bool fail = false;
        LOG("\n\rEnter a maximum %d digit number:", maxDigits);
        for (uint8_t i = 0; i < maxDigits; i++)
        {
            char c;

            while(1)
            {
                int status = readUserInput(&c);

```

```

if (status == PASS)
{
    break;
}

if (c == '\r')           //user enters ENTER key
{
    break;
}

LOG("%c", c);          //prints character on screen as user is typing

if (c >= '0' && c <= '9') // check if input is a number
{
    uint8_t x = c - '0';   //converts user input(ASCII) to int
    value = value * 10 + x; //calculates multi-digit ASCII input
    if (value > maxVal)
    {
        ERROR_LOG("Value entered is beyond range %x", maxVal);
        fail = true;
        break;
    }
}

else
{
    ERROR_LOG("Enter only digit!"); //if user enters an other character than a number
    fail = true;
    break;
}

if (fail == false)         //if no previous errors have occurred, break while loop
{
    break;
}

INFO_LOG("User entered input = %d", value);

return value;
}

```


6.0.5 Appendix - Firmware Source Code (MAXIM Algorithm)

```
*****
* This code obtained from the MAX30101WING manufacturer MAXIM was used to
* validate the values of the IR and red led values for the SpO2 and Heart rate.
* It was not used further in the project as the values obtained were incorrect.
*
* The file contains calculation for R value which is used for the determining
* the oxygen level in the blood.
*
* -----
*
* This code follows the following naming conventions:
*
* char          ch_pmod_value
* char (array)   s_pmod_s_string[16]
* float         f_pmod_value
* int32_t       n_pmod_value
* int32_t (array) an_pmod_value[16]
* int16_t        w_pmod_value
* int16_t (array) aw_pmod_value[16]
* uint16_t      uw_pmod_value
* uint16_t (array) auw_pmod_value[16]
* uint8_t        uch_pmod_value
* uint8_t (array) auch_pmod_buffer[16]
* uint32_t       un_pmod_value
* int32_t *     pn_pmod_value
*
* -----
*/
*****
* Copyright (C) 2016 Maxim Integrated Products, Inc., All Rights Reserved.
*
* Permission is hereby granted, free of charge, to any person obtaining a
* copy of this software and associated documentation files (the "Software"),
* to deal in the Software without restriction, including without limitation
* the rights to use, copy, modify, merge, publish, distribute, sublicense,
* and/or sell copies of the Software, and to permit persons to whom the
* Software is furnished to do so, subject to the following conditions:
*
* The above copyright notice and this permission notice shall be included
* in all copies or substantial portions of the Software.
*
```

* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
* OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
* IN NO EVENT SHALL MAXIM INTEGRATED BE LIABLE FOR ANY CLAIM, DAMAGES
* OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
* ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
* OTHER DEALINGS IN THE SOFTWARE.

*

* Except as contained in this notice, the name of Maxim Integrated
* Products, Inc. shall not be used except as stated in the Maxim Integrated
* Products, Inc. Branding Policy.

*

* The mere transfer of this software does not imply any licenses
* of trade secrets, proprietary technology, copyrights, patents,
* trademarks, maskwork rights, or any other form of intellectual
* property whatsoever. Maxim Integrated Products, Inc. retains all
* ownership rights.

*/

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
```

```
#define MAX_LINES 500
```

```
#include "algo.h"
```

```
#ifdef ARCH_8051
```

```
// Arduino Uno doesn't have enough SRAM to store 100 samples of IR led data and red led data in 32-bit format
```

```
// To solve this problem, 16-bit MSB of the sampled data will be truncated. Samples become 16-bit data.
```

```
void maxim_heart_rate_and_oxygen_saturation(uint16_t *ir_buffer, int32_t buffer_length, uint16_t *red_buffer, int32_t *pn_spo2,
int8_t *pch_spo2_valid,
```

```
                int32_t *pn_heart_rate, int8_t *pch_hr_valid)
```

```
#else
```

```
void maxim_heart_rate_and_oxygen_saturation(uint32_t *ir_buffer, int32_t buffer_length, uint32_t *red_buffer, int32_t *pn_spo2,
int8_t *pch_spo2_valid,
```

```
                int32_t *pn_heart_rate, int8_t *pch_hr_valid)
```

```
#endif
```

```

/**
 * \brief Calculate the heart rate and SpO2 level
 * \par Details
 * By detecting peaks of PPG cycle and corresponding AC/DC of red/infra-red signal, the an_ratio for the SPO2 is computed.
 * Since this algorithm is aiming for Arm M0/M3, formula for SPO2 did not achieve the accuracy due to register overflow.
 * Thus, accurate SPO2 is precalculated and save longo uch_spo2_table[] per each an_ratio.
 *
 * \param[in] *ir_buffer - IR sensor data buffer
 * \param[in] buffer_length - IR sensor data buffer length
 * \param[in] *red_buffer - Red sensor data buffer
 * \param[out] *pn_spo2 - Calculated SpO2 value
 * \param[out] *pch_spo2_valid - 1 if the calculated SpO2 value is valid
 * \param[out] *pn_heart_rate - Calculated heart rate value
 * \param[out] *pch_hr_valid - 1 if the calculated heart rate value is valid
 *
 * \retval None
 */
{
    uint32_t un_ir_mean;
    int32_t k, n_i_ratio_count;
    int32_t i, n_exact_ir_valley_locs_count, n_middle_idx;
    int32_t n_th1, n_npks;
    int32_t an_ir_valley_locs[15];
    int32_t n_peak_interval_sum;

    int32_t n_y_ac, n_x_ac;
    int32_t n_spo2_calc;
    int32_t n_y_dc_max, n_x_dc_max;
    int32_t n_y_dc_max_idx = 0;
    int32_t n_x_dc_max_idx = 0;
    int32_t an_ratio[5], n_ratio_average;
    int32_t n_nume, n_denom;

    // calculates DC mean and subtract DC from ir
    un_ir_mean = 0;
    for (k = 0; k < buffer_length; k++)
        un_ir_mean += ir_buffer[k];

    un_ir_mean = un_ir_mean / buffer_length;
}

```

```

// remove DC and invert signal so that we can use peak detector as valley detector
for (k = 0; k < BUFFER_SIZE; k++)
    an_x[k] = -1 * (ir_buffer[k] - un_ir_mean);

// 4 pt Moving Average
for (k = 0; k < BUFFER_SIZE - MA4_SIZE; k++)
{
    an_x[k] = (an_x[k] + an_x[k + 1] + an_x[k + 2] + an_x[k + 3]) / (int)4;
}

// calculate threshold
n_th1 = 0;
for (k = 0; k < BUFFER_SIZE; k++)
{
    n_th1 += an_x[k];
}
n_th1 = n_th1 / (BUFFER_SIZE);
if (n_th1 < 30)
    n_th1 = 30; // min allowed
if (n_th1 > 60)
    n_th1 = 60; // max allowed

for (k = 0; k < 15; k++)
    an_ir_valley_locs[k] = 0;

// since we flipped signal, we use peak detector as valley detector
maxim_find_peaks(an_ir_valley_locs, &n_npks, an_x, BUFFER_SIZE, n_th1, 4, 15); // peak_height, peak_distance,
max_num_peaks
n_peak_interval_sum = 0;
if (n_npks >= 2)
{
    for (k = 1; k < n_npks; k++)
        n_peak_interval_sum += (an_ir_valley_locs[k] - an_ir_valley_locs[k - 1]);
    n_peak_interval_sum = n_peak_interval_sum / (n_npks - 1);
    *pn_heart_rate = (int32_t)((FreqS * 60) / n_peak_interval_sum);
    *pch_hr_valid = 1;
}
else
{

```

```

*pn_heart_rate = -999; // unable to calculate because # of peaks are too small
*pch_hr_valid = 0;
}

// load raw value again for SPO2 calculation : RED(=y) and IR(=X)
for (k = 0; k < buffer_length; k++)
{
    an_x[k] = ir_buffer[k];
    an_y[k] = red_buffer[k];
}

// find precise min near an_ir_valley_locs
n_exact_ir_valley_locs_count = n_npks;

// using exact_ir_valley_locs , find ir-red DC and ir-red AC for SPO2 calibration an_ratio
// finding AC/DC maximum of raw

n_ratio_average = 0;
n_i_ratio_count = 0;
for (k = 0; k < 5; k++)
    an_ratio[k] = 0;

for (k = 0; k < n_exact_ir_valley_locs_count; k++)
{
    if (an_ir_valley_locs[k] > BUFFER_SIZE)
    {
        *pn_spo2 = -999; // do not use SPO2 since valley loc is out of range
        *pch_spo2_valid = 0;
        return;
    }
}

// find max between two valley locations
// and use an_ratio betwen AC component of Ir & Red and DC component of Ir & Red for SPO2
for (k = 0; k < n_exact_ir_valley_locs_count - 1; k++)
{
    n_y_dc_max = -16777216;
    n_x_dc_max = -16777216;
    if (an_ir_valley_locs[k + 1] - an_ir_valley_locs[k] > 3)
    {

```

```

for (i = an_ir_valley_locs[k]; i < an_ir_valley_locs[k + 1]; i++)
{
    if (an_x[i] > n_x_dc_max)
    {
        n_x_dc_max = an_x[i];
        n_x_dc_max_idx = i;
    }
    if (an_y[i] > n_y_dc_max)
    {
        n_y_dc_max = an_y[i];
        n_y_dc_max_idx = i;
    }
}
n_y_ac = (an_y[an_ir_valley_locs[k + 1]] - an_y[an_ir_valley_locs[k]]) * (n_y_dc_max_idx - an_ir_valley_locs[k]); // red
n_y_ac = an_y[an_ir_valley_locs[k]] + n_y_ac / (an_ir_valley_locs[k + 1] - an_ir_valley_locs[k]);
n_y_ac = an_y[n_y_dc_max_idx] - n_y_ac; // subtracting linear DC

components from raw
n_x_ac = (an_x[an_ir_valley_locs[k + 1]] - an_x[an_ir_valley_locs[k]]) * (n_x_dc_max_idx - an_ir_valley_locs[k]); // ir
n_x_ac = an_x[an_ir_valley_locs[k]] + n_x_ac / (an_ir_valley_locs[k + 1] - an_ir_valley_locs[k]);
n_x_ac = an_x[n_x_dc_max_idx] - n_x_ac; // subtracting linear DC components from raw
n_nume = (n_y_ac * n_x_dc_max) >> 7; // prepare X100 to preserve floating value
n_denom = (n_x_ac * n_y_dc_max) >> 7;
if (n_denom > 0 && n_i_ratio_count < 5 && n_nume != 0)
{
    an_ratio[n_i_ratio_count] = (n_nume * 100) / n_denom; // formula is ( n_y_ac * n_x_dc_max ) / ( n_x_ac
*n_y_dc_max );
    n_i_ratio_count++;
}
}

// choose median value since PPG signal may varies from beat to beat
maxim_sort_ascend(an_ratio, n_i_ratio_count);
n_middle_idx = n_i_ratio_count / 2;

if (n_middle_idx > 1)
    n_ratio_average = (an_ratio[n_middle_idx - 1] + an_ratio[n_middle_idx]) / 2; // use median
else
    n_ratio_average = an_ratio[n_middle_idx];

```

```

if (n_ratio_average > 2 && n_ratio_average < 184)
{
    n_spo2_calc = uch_spo2_table[n_ratio_average];
    *pn_spo2 = n_spo2_calc;
    *pch_spo2_valid = 1; // float_SPO2 = -45.060*n_ratio_average* n_ratio_average/10000 + 30.354 *n_ratio_average/100 + 94.845
; // for comparison with table
}
else
{
    *pn_spo2 = -999; // do not use SPO2 since signal an_ratio is out of range
    *pch_spo2_valid = 0;
}
}

void maxim_find_peaks(int32_t *pn_locs, int32_t *n_npks, int32_t *pn_x, int32_t n_size, int32_t n_min_height, int32_t
n_min_distance, int32_t n_max_num)
/**
 * \brief      Find peaks
 * \par       Details
 *             Find at most MAX_NUM peaks above MIN_HEIGHT separated by at least MIN_DISTANCE
 *
 * \retval     None
 */
{
    maxim_peaks_above_min_height(pn_locs, n_npks, pn_x, n_size, n_min_height);
    maxim_remove_close_peaks(pn_locs, n_npks, pn_x, n_min_distance);
    *n_npks = min(*n_npks, n_max_num);
}

void maxim_peaks_above_min_height(int32_t *pn_locs, int32_t *n_npks, int32_t *pn_x, int32_t n_size, int32_t n_min_height)
/**
 * \brief      Find peaks above n_min_height
 * \par       Details
 *             Find all peaks above MIN_HEIGHT
 *
 * \retval     None
 */
{
    int32_t i = 1, n_width;

```

```

*n_npks = 0;

while (i < n_size - 1)
{
    if (pn_x[i] > n_min_height && pn_x[i] > pn_x[i - 1])
    { // find left edge of potential peaks
        n_width = 1;
        while (i + n_width < n_size && pn_x[i] == pn_x[i + n_width]) // find flat peaks
            n_width++;
        if (pn_x[i] > pn_x[i + n_width] && (*n_npks) < 15)
        { // find right edge of peaks
            pn_locs[(*n_npks)++] = i;
            // for flat peaks, peak location is left edge
            i += n_width + 1;
        }
        else
            i += n_width;
    }
    else
        i++;
}
}

void maxim_remove_close_peaks(int32_t *pn_locs, int32_t *pn_npks, int32_t *pn_x, int32_t n_min_distance)
/**
 * \brief Remove peaks
 * \par Details
 * Remove peaks separated by less than MIN_DISTANCE
 *
 * \retval None
 */
{
    int32_t i, j, n_old_npks, n_dist;

    /* Order peaks from large to small */
    maxim_sort_indices_descend(pn_x, pn_locs, *pn_npks);

    for (i = -1; i < *pn_npks; i++)

```

```

{
    n_old_npks = *pn_npks;
    *pn_npks = i + 1;
    for (j = i + 1; j < n_old_npks; j++)
    {
        n_dist = pn_locs[j] - (i == -1 ? -1 : pn_locs[i]); // lag-zero peak of autocorr is at index -1
        if (n_dist > n_min_distance || n_dist < -n_min_distance)
            pn_locs[(*pn_npks)++] = pn_locs[j];
    }
}

// Resort indices int32_to ascending order
maxim_sort_ascend(pn_locs, *pn_npks);
}

```

```

void maxim_sort_ascend(int32_t *pn_x, int32_t n_size)
/**
 * \brief      Sort array
 * \par       Details
 *             Sort array in ascending order (insertion sort algorithm)
 *
 * \retval     None
 */
{
    int32_t i, j, n_temp;
    for (i = 1; i < n_size; i++)
    {
        n_temp = pn_x[i];
        for (j = i; j > 0 && n_temp < pn_x[j - 1]; j--)
            pn_x[j] = pn_x[j - 1];
        pn_x[j] = n_temp;
    }
}

```

```

void maxim_sort_indices_descend(int32_t *pn_x, int32_t *pn_idx, int32_t n_size)
/**
 * \brief      Sort indices
 * \par       Details
 *             Sort indices according to descending order (insertion sort algorithm)

```

```

*
* \retval    None
*/
{
    int32_t i, j, n_temp;
    for (i = 1; i < n_size; i++)
    {
        n_temp = pn_idx[i];
        for (j = i; j > 0 && pn_x[n_temp] > pn_x[pn_idx[j - 1]]; j--)
            pn_idx[j] = pn_idx[j - 1];
        pn_idx[j] = n_temp;
    }
}

#endif ARCH_X86
int read_file(uint32_t *ir_data, uint32_t *red_data)
{
    FILE *file;
    int num_lines = 0;

    // Open file
    file = fopen("data.txt", "r");
    if (file == NULL)
    {
        printf("Error opening the file.\n");
        return 0;
    }

    // Read and process each line
    while (fscanf(file, "%d %d", &ir_data[num_lines], &red_data[num_lines]) == 2 && num_lines < MAX_LINES)
    {
        num_lines++;
    }

    // Close the file
    fclose(file);

    // Print the data
    printf("Data read from the file:\n");
}

```

```

for (int i = 0; i < num_lines; i++)
{
    printf("Line %d: ", i + 1);
    printf("%d\n\r", ir_data[i]);
}
printf("\n");

return num_lines;
}

int main()
{
    uint32_t ir_data[MAX_LINES]; // Array to store data
    uint32_t red_data[MAX_LINES];
    uint32_t buffer_length = read_file(ir_data, red_data);
    printf("Read %d lines\n\r", buffer_length);

    int32_t pn_spo2;
    int8_t pch_spo2_valid;
    int32_t pn_heart_rate;
    int8_t pch_hr_valid;

    // calc spo2
    maxim_heart_rate_and_oxygen_saturation(ir_data, buffer_length, red_data, &pn_spo2, &pch_spo2_valid,
                                             &pn_heart_rate, &pch_hr_valid);

    if (pch_hr_valid == 1)
    {
        printf("Calculated HR = %d\n\r", pn_heart_rate);
    }

    if (pch_spo2_valid == 1)
    {
        printf("Calculated Sp02 = %d\n\r", pn_spo2);
    }

    return 0;
}
#endif

```




6.0.6 Appendix - Firmware Source Code (Makefile)

```

#####
#Author: Venetia Furtado
#ECEN 5613 - Spring 2024 - Prof. McClure
#University of Colorado Boulder
#
#GNU makefile for SDCC base program
#Generates Intel Hex file for C files in this directory using SDCC
#####

CC = sdcc  #Sets the compiler to SDCC
PACKTOOL = packihx
CFLAGS = --model-large --out-fmt-ihx -mmcs51 --opt-code-size --std-sdcc99 --verbose  # Compiler flags
LDFLAGS = --out-fmt-ihx --model-large -mmcs51 --opt-code-size --std-sdcc99 --verbose --code-loc 0x0000 --code-size 0x8000 --xram-loc
0x0000 --xram-size 0x8000 #Linker flags
SRC_DIR = src
BIN_DIR = bin
INCLUDE_DIR = inc
TARGET-IHX = main.ihx          # Output IHX File
TARGET-HEX = main.hex          # Output HEX file

#Searches for all .c files in this directory
SOURCES := $(wildcard $(SRC_DIR)/*.c)

#Creates a .rel file for all the .c files in directory
OBJECTS := $(patsubst $(SRC_DIR)/%.c, $(BIN_DIR)/%.rel, $(SOURCES))

#first command that make executes
all: $(BIN_DIR)/$(TARGET-HEX)

#Compiles object files (.rel) from .c files
$(BIN_DIR)/%.rel: $(SRC_DIR)/%.c
    $(CC) $(CFLAGS) -I $(INCLUDE_DIR) -c $< -o $@

#Link all .rel files to create the .hex files
$(BIN_DIR)/$(TARGET-IHX):$(BIN_DIR)/$(OBJECTS)
    $(CC) $(LDFLAGS) $(BIN_DIR)/main.rel $(filter-out $(BIN_DIR)/main.rel, $(OBJECTS)) -o $@
```

```
#Generates Hex files for the ihx file
$(BIN_DIR)/$(TARGET-HEX):$(BIN_DIR)/$(TARGET-IHX)
$(PACKTOOL) $< > $@

# Phony target for running the command clean
.PHONY: clean
clean:
cd bin; rm -f *.asm *.rel *.lst *.rst *.hex *.mem *.map *.sym *.lnk *.lk *.ihx
```

6.0.7 Appendix - Software Source Code

```

#####
# Author - Venetia Furtado
# Final Project: Health Monitoring System
# ECEN 5613 - Spring 2024 - Prof. McClure
# University of Colorado Boulder
# Revised 04/26/2024
#
#####

# This file contains the code to transfer the data received at COM5 to COM4
#####

import serial

# Define the serial ports and baud rates
input_serial_port = 'COM5' # Change this to match your input serial port
output_serial_port = 'COM4' # Change this to match your output serial port
baud_rate = 9600 # Change this to match your device's baud rate

# Open input and output serial ports
input_serial = serial.Serial(input_serial_port, baud_rate)
output_serial = serial.Serial(output_serial_port, baud_rate)

try:
    # Continuously read data from the input serial port and send it to the output serial port
    while True:
        # Read a line of data from the input serial port
        data = input_serial.readline().decode('ascii')
        print(data)

        if not data[0].isdigit():
            continue

        # If data is received, send it to the output serial port
        if data:
            output_serial.write(data.encode('ascii'))

    # Handle keyboard interrupt (Ctrl+C)
except KeyboardInterrupt:
    print("\nProgram terminated by user.")

```

```

# Close the serial ports
finally:
    input_serial.close()
    output_serial.close()

#####
# Author - Venetia Furtado
# Final Project: Health Monitoring System
# ECEN 5613 - Spring 2024 - Prof. McClure
# University of Colorado Boulder
# Revised 04/26/2024
#
#####

# This file contains the code to plot the values of the IR LED received.

#####

import serial
import matplotlib.pyplot as plt

# Initialize serial communication
ser = serial.Serial('COM4', 9600) # Change 'COM1' to the appropriate port and baud rate

# Initialize empty lists to store data
x_data = []
y_data = []

plot_count = 0
# Infinite loop to continuously receive and plot data
try:
    while plot_count < 500:
        plot_count += 1
        print(plot_count)

        # Read data from UART
        data = ser.readline().decode().strip() # Decode received bytes into string

        # Parse data (assuming it's a comma-separated pair of values)
        if ',' in data:

```

```
x_str, y_str = data.split(',') # Assuming data is in the format "x,y"
x = int(x_str)
y = int(y_str)

x_data.append(x)
y_data.append(y)

# Close serial port and plot on program exit
finally:
    ser.close()

# Plot the data
plt.plot(x_data, y_data)
plt.xlabel('Sample Number')
plt.ylabel('LED output')
plt.title('UART Data Plot')
plt.grid(True)

# Save the plot as a JPG file
plt.savefig('uart_data_plot.jpg')

# Show the plot
plt.show()
```

6.0.8 Appendix - Datasheets and Application Notes



Recommended Configurations and Operating Profiles for MAX30101/MAX30102 EV Kits

UG6409; Rev 0; 3/18

Abstract

This user guide was developed to help users quickly configure the MAX30101 and MAX30102 devices for pulse-ox and/or heart-rate monitoring use using the MAX30101ACCEVKIT and MAX30102ACCEVKIT. Recommended operating profiles are offered as well as a suggested methodology for determining signal optimization and associated power-supply requirements. In addition, heart-rate and pulse-ox calculation examples are given that are representative of basic post-processing operations.

Table of Contents

Introduction	4
Pulse Oximetry and Heart Rate.....	4
Transmissive Pulse Oximetry	4
Reflective Pulse Oximetry	4
Primary Applications	4
Discussion.....	5
Heart Rate	5
SpO ₂	6
Default EV Kit	6
Flowchart for General EV Kit Operation	8
Implementing Specific Configurations.....	9
Expected Heart-Rate (HR) Signals	10
Expected SpO₂ Signals	13
General Recommendations	14
SPO₂ Register-Level Operation:.....	16
Register Map and General Guidelines	16
LED-Pulse Amplitude and Current Control	17
SpO ₂ Recommended Configuration	19
Heart Rate Monitoring for Wrist Application:.....	21
Increasing the Green LED Signal:	23
Register Map	25
Interrupts	25
Data Streaming and FIFO Operation.....	25
Recommended Practices:	27
Proximity Detection	28
Post-Processing.....	29
Heart-Rate Post-Processing Using ADC Code	29
SpO ₂ Post-Processing Using ADC Code.....	30
Enclosure Consideration.....	32
Troubleshooting	33
Firmware	33
Ambient Light.....	35

References.....	36
Figures.....	36
Trademarks	36

List of Figures

Figure 1. DC and AC components of a PPG signal.....	5
Figure 2. Default EV Kit GUI screen.....	7
Figure 3. General setup for using MAX30101.....	8
Figure 4. Accessing registers while the EV kit GUI is in operation.....	10
Figure 5. Data recorded by the GUI when a Green LED is used at 0.04mA.....	11
Figure 6. Data recorded by the GUI when a Green LED is used at 0.13mA.....	11
Figure 7. Data recorded by the GUI when a Green LED is used at 0.20mA.....	12
Figure 8. Data recorded by the GUI when a Green LED is used at 0.41mA.....	12
Figure 9. Data recorded by the GUI when Red and IR LEDs are used at 0.41mA.....	13
Figure 10. Pulse width example.....	14
Figure 11. Allowed settings for heart rate configuration.	14
Figure 12. Allowed settings for SpO2 configuration.....	15
Figure 13. Mode register, available modes, and SpO2 registers.....	16
Figure 14. LED pulse-width control.....	17
Figure 15. LED-pulse amplitude register and multi-LED mode control registers.	18
Figure 16. Red and IR showing SpO2 signals in sync	20
Figure 17a and 17b. SpO2 mode and LED mode (for wrist HR applications)	21
Figure 18. LED-mode timing slots.....	21
Figure 19. LED mode timing slots for wrist HR measurements.	22
Figure 20. Changing the Green LED current.	22
Figure 21. Wrist HR measurement using the MAX30101 EV kit.....	22
Figure 22. Register access while the GUI is operating.....	23
Figure 23. GUI register access.....	24
Figure 24. Input values in LED4GreenPulseAmp register.	24
Figure 25. Interrupt status and interrupt enable registers.....	25
Figure 26. FIFO registers.	26
Figure 27. FIFO_A_FULL registers and how they correspond to number of samples in FIFO.	27
Figure 28. General guidelines for setting up and verifying the proximity detection function.	28
Figure 29. ADC code plotted with respect to time.	29
Figure 30. One period of the ADC code plotted with respect to time.....	29
Figure 31. IR and Red Raw data for SpO2 processing.....	30
Figure 32. Hand calculations needed for SpO2.	31
Figure 33. Where the USB Debut Adapter should be connected for reflashing.....	33
Figure 34. Silicon Laboratories Flash Utility.....	34
Figure 35. Regularity in spikes indicating possible ambient light.....	35
Figure 36. No spikes and low-ADC code indicating a dark room.....	35

Introduction

This user guide presents heart-rate and SpO₂ configurations for the MAX30101/MAX30102 (MAX3010x) product family evaluation kits. The MAX3010x provides a complete optical-module solution to ease the design-in process for mobile and wearable devices for both fitness and health. Because the MAX30101 and MAX30102 work similarly, with the only difference being the LEDs present (see Table 1), this user guide focuses on the MAX30101 device throughout. The user should be aware that the same settings can be applied to MAX30102 devices. Additionally, MAX3010x EV kit boards (MAX30101ACCEVKIT and MAX30102ACCEVIT) also feature an ultra-low-power linear accelerometer that can be used to implement motion detection or compensation algorithms.

Table 1. MAX3010x Product Family and Their Available Modes

DEVICE	LEDs PRESENT	AVAILABLE MODES
MAX30101	Green, Red, Infrared	HR, SpO ₂
MAX30102	Red, Infrared	HR, SpO ₂

Pulse Oximetry and Heart Rate

Pulse oximetry is a noninvasive method of measuring an individual's blood oxygen saturation levels. Oxygen saturation levels, referring to the ratio of oxygenated hemoglobin to total hemoglobin in the blood, can aid in detection of hypoxemia, deteriorating organ function and even cardiac arrest. Therefore, a noninvasive solution to measure oxygen saturation levels, as provided by MAX3010x, is of medical importance. Additionally, there is an inherent heart-rate signal associated with pulse-oximetry measurement, allowing the MAX30101 users to obtain this as well.

Transmissive Pulse Oximetry

LEDs transmit light of specific wavelengths through tissue, which is absorbed by photodetectors on the other end. The change in absorbance of each wavelength determines oxygen saturation levels. This application requires use of a thin test site with adequate blood perfusion like a finger or ear lobe to maximize the transmission of light.

Reflective Pulse Oximetry

In this case, the setup is like the one used for transmissive pulse oximetry except the photodiodes are placed on the same side of the test site as the LEDs. The LED light illuminates the skin while the reflected signal is monitored for changes in light absorption. This is known as photoplethysmography (PPG): the optical measurement of organ volume changes. In the same way, the MAX30101 monitors the perfusion of blood to the dermis and subcutaneous tissue of the skin. Reflective pulse oximeters work well for many mobile fitness applications as the technology is independent of tissue depth, unlike the transmissive method. The MAX3010x product family uses reflective pulse-oximetry technology to determine HR, and SpO₂.

Note: Each type of pulse oximetry method relies on the idea that the light reflected or transmitted from the tissue is a signal strong enough to be captured by the photodiode. Thus, depending on the application MAX3010x is used for, the LED signal strength, pulse width, or sampling rate needs to be changed by the user to optimize its performance.

Primary Applications

- SpO₂ Operation
- Heart-Rate Monitoring
- Proximity Detection

Discussion

Heart Rate

While there are various kinds of hemoglobin compounds present in blood, for SpO₂ calculations it is assumed that oxygenated hemoglobin and deoxygenated hemoglobin are the only significant factors. In a reflective pulse-oximetry set up, LEDs illuminate skin tissue and the reflected signal is detected by the photodiode. This reflected signal contains the light that was optically modulated by the volumetric changes of the arteries and capillaries. This photoplethysmography (PPG) signal is extremely important in determining heart rate and SpO₂ levels. PPG signals have a DC component and an AC component combined with it, as can be seen below in **Figure 1**.

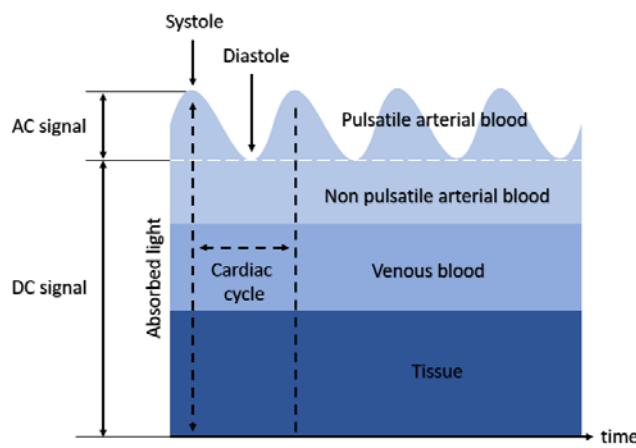


Figure 1. DC and AC components of a PPG signal.

The DC component is attributed to the light absorption of nonpulsatile tissue: venous capillary and arterial blood.^[1] The AC component, on the other hand, is due to the pulsatile nature of arterial blood. Since arteries have a direct connection to the heart, the arterial blood pulsates as the heart pulsates. Instantaneous heart rate can be calculated by measuring the time between consecutive systolic peaks.

It is important to note here that heart rate can be measured by using just one LED, red for instance, as the AC component is the only signal needed.

SpO₂

MAX3010x SpO₂ measurements employ two different wavelength LEDs to identify the ratio of oxygenated hemoglobin to the deoxygenated hemoglobin. Red and IR LEDs are used to determine separate PPG signals. As the DC components and AC components of the two LEDs have different amplitudes, they must be normalized to make useful comparisons. For this comparison, a ratio 'R' is determined, which is directly proportional to SpO₂. Before the equations for R and SpO₂ are introduced, it should be noted that SpO₂ approximates arterial oxygen saturation (SaO₂). SpO₂ measurements are commonly found in fitness/medical applications due to its good approximation of SaO₂.^[2] The following equation for R is known as "ratio of ratios:"

$$R = \frac{\frac{AC_{red}}{DC_{red}}}{\frac{AC_{infrared}}{DC_{infrared}}}$$

Once R is determined, a curvilinear approximation or a lookup table can be used to determine the SpO₂ estimate. This data is typically collected by empirical methods where numerous subjects are used. Age, skin tone, overall health, and medical conditions can affect the accuracy of the SpO₂ measurement. Below is a linear approximation example derived from a best-fit straight-line approximation of SpO₂ vs. R data (S. Prahls^[3]) between the R-range of 0.4 to 3.4:

$$SpO_2 = 104 - 17R$$

Note: The MAX30101 EV kit GUI can log data in CSV format, but post-processing is required when MAX30101 is used in an application to determine HR and/or SpO₂. This user guide provides a brief mention of the challenges associated with post processing and how a user can make sense of the graphical data.

Default EV Kit

When the user initiates the MAX30101 EV kit GUI, **Figure 2** shows what the GUI should look like. Two very basic example algorithms are provided with the GUI. These simple algorithms provide numerical estimates of HR and SpO₂.

The GUI allows the user to log raw data from the ADC. While the Maxim Integrated user interface can be used to visually determine functionality of the MAX3010x device, post processing is required to determine the numerical values of HR and SpO₂.

The main GUI window, on the left, has graphical plots for optical measurements and accelerometer measurements. The optical measurements show data corresponding to the LED mode selected. For example, if LED slot 1 has "3 LED3(Green)" selected and LED slots 2 to 4 have "0 Disabled" selected, then optical measurements only show green measurements when the GUI is operational. The accelerometer measurement plot shows three axes of linear acceleration.

On the left, Mode Configuration, Settings, LED currents, Proximity, and LED Mode Timing slot settings are provided for the user to configure the MAX30101 device.

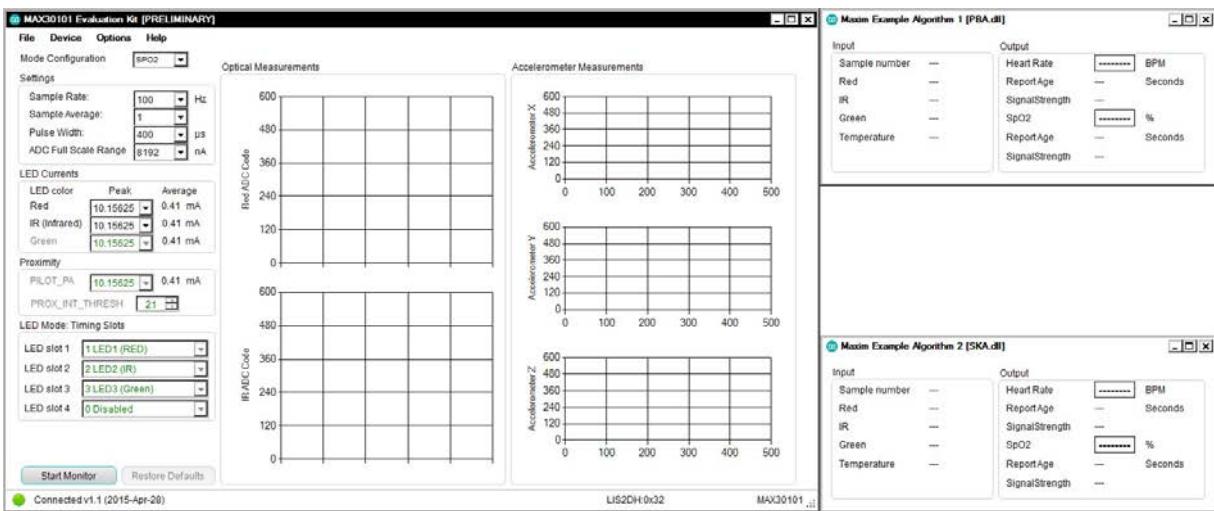


Figure 2. Default EV Kit GUI screen.

Flowchart for General EV Kit Operation

The flowchart below (**Figure 3**) shows a general setup for getting started with MAX30101. The user should make sure that the sensor board and microcontroller are connected properly.

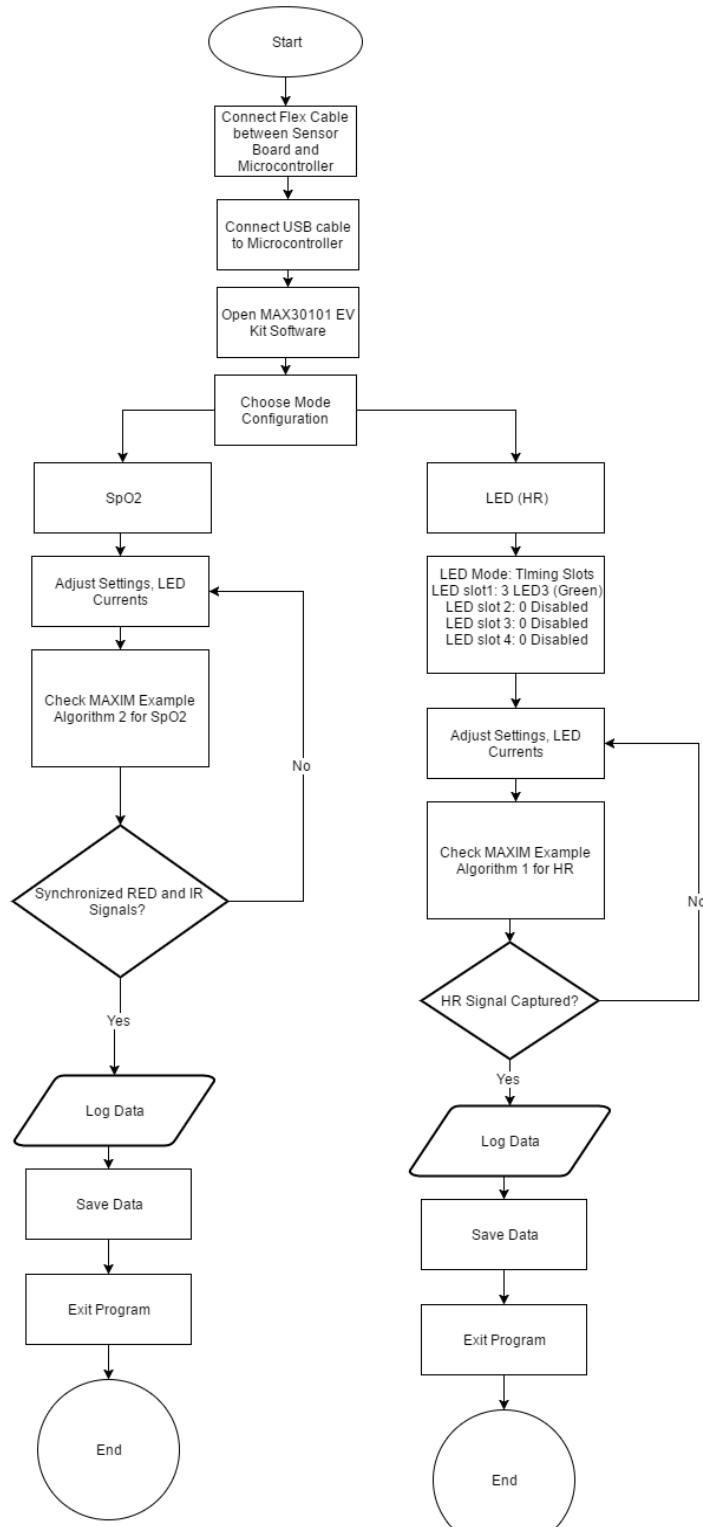
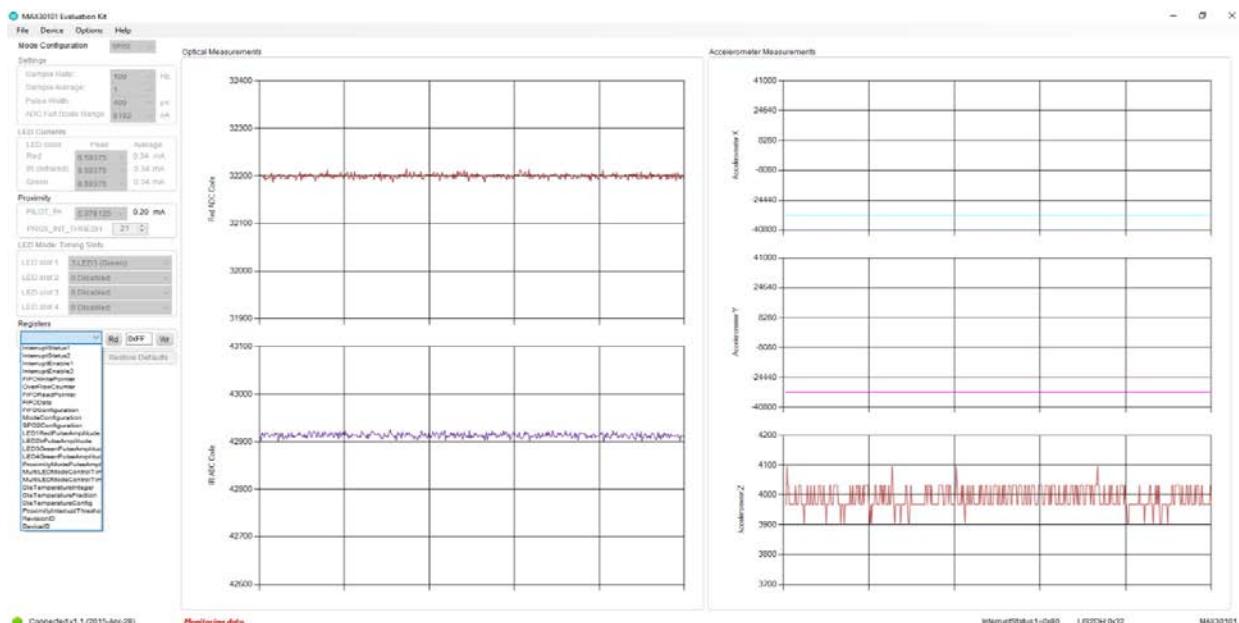
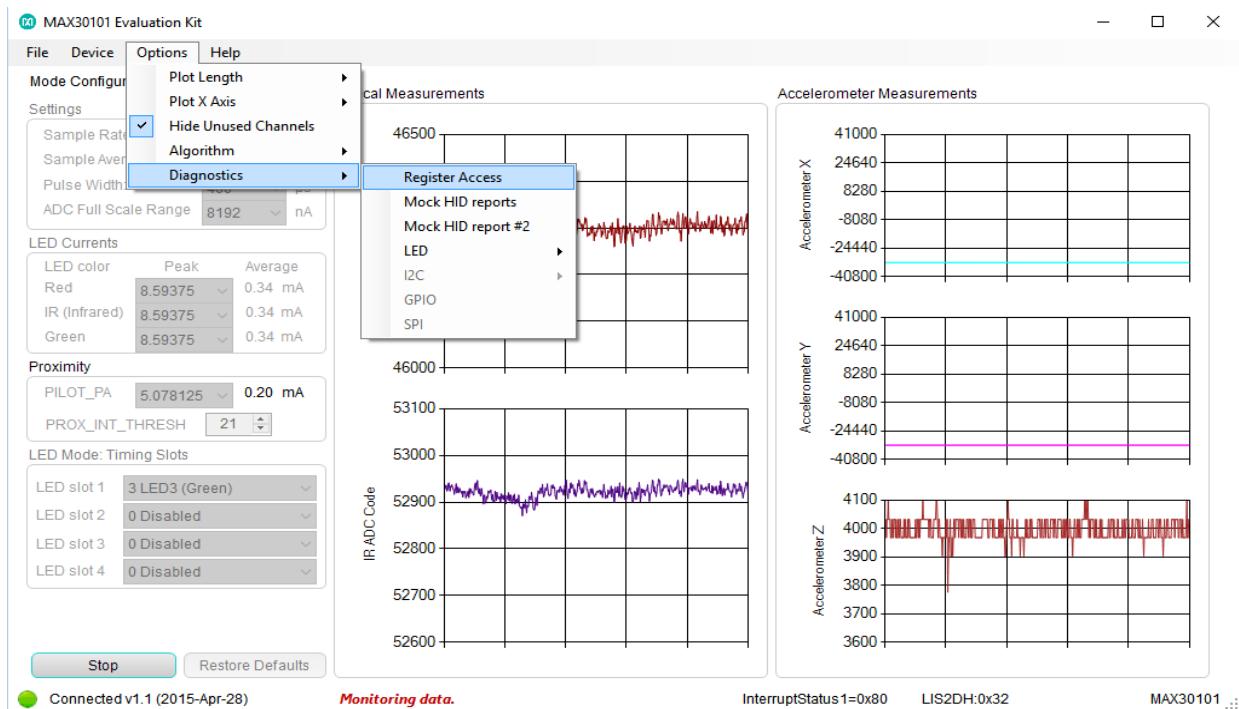


Figure 3. General setup for using MAX30101.

Implementing Specific Configurations

Changing the GUI settings is intuitive and allows the user to experiment with them before use. However, it is important to note that if the user wants to read or write new register values, the MAX30101 GUI needs to be operational. This can be extremely useful as this provides direct access to the registers of the operational MAX30101. The user should study the MAX30101 data sheet in depth and use this user guide to optimize MAX30101 use. **Figure 4** shows how the registers can be accessed during operation followed by a recommended way of using the MAX30101 EV kit for heart rate (HR) and SpO₂.



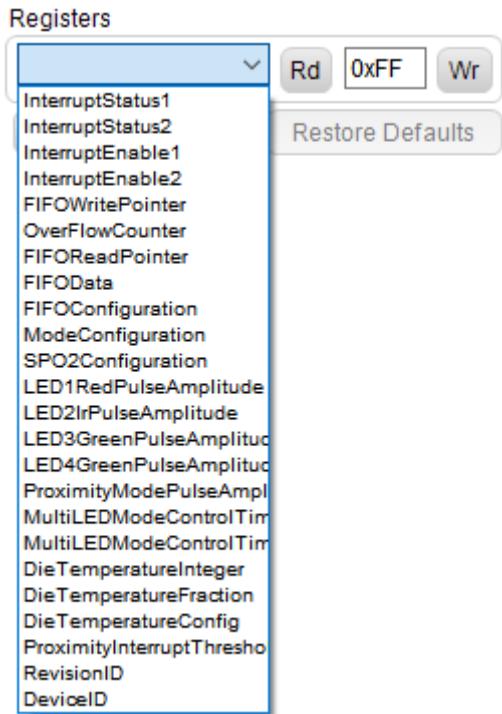


Figure 4. Accessing registers while the EV kit GUI is in operation.

Expected Heart-Rate (HR) Signals

Before discussing the recommended register configurations, **Figure 5**, **Figure 6**, **Figure 7**, and **Figure 8** show what the expected heart-rate signal should look like and what happens when LED-current settings are changed. Ideally, applications requiring a good HR signal require a higher LED current. But if low power consumption is desired, then the user should consider lowering the LED current. The issue with this is that the SNR lowers when reducing pulse width and sample rate. To resolve this issue, the user should try different LED currents to determine an optimal setting. Figure 5 through Figure 8 show HR signals when captured at average currents of 0.04mA (peak = 1), 0.13mA (peak = 3.125), 0.20mA (peak = 5.078125), and 0.41mA (peak = 10.15625). The peak value refers to the peak current level for the LEDs in mA.

Note: Green LEDs are typically used for wrist HR monitoring instead of Red or IR LEDs, sensing blood pulsations where perfusion is reduced. The Green PPG is influenced by DC to a smaller degree.^[4]

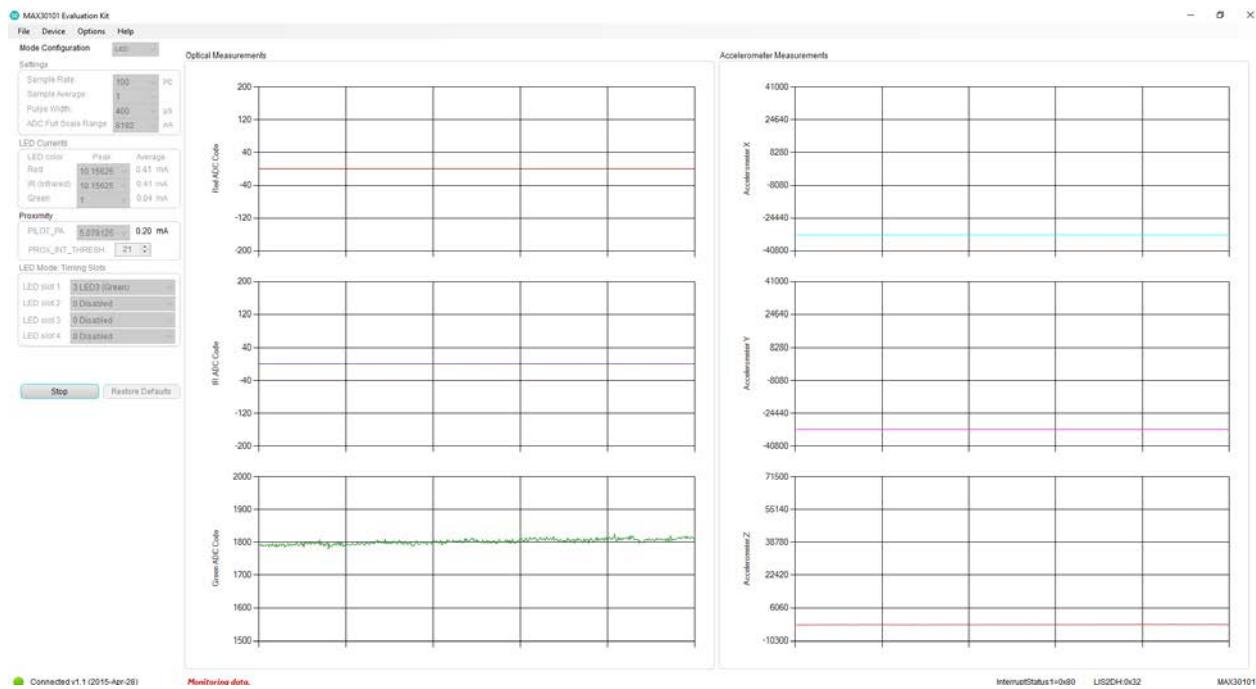


Figure 5. Data recorded by the GUI when a Green LED is used at 0.04mA.

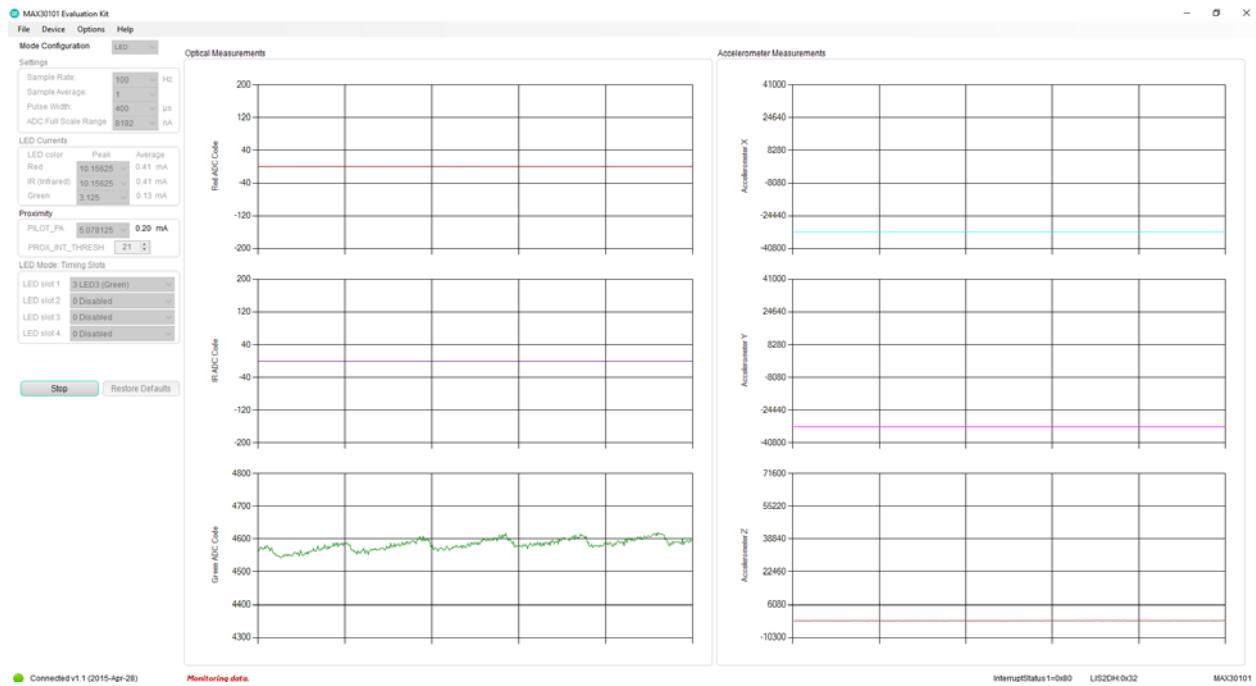


Figure 6. Data recorded by the GUI when a Green LED is used at 0.13mA.

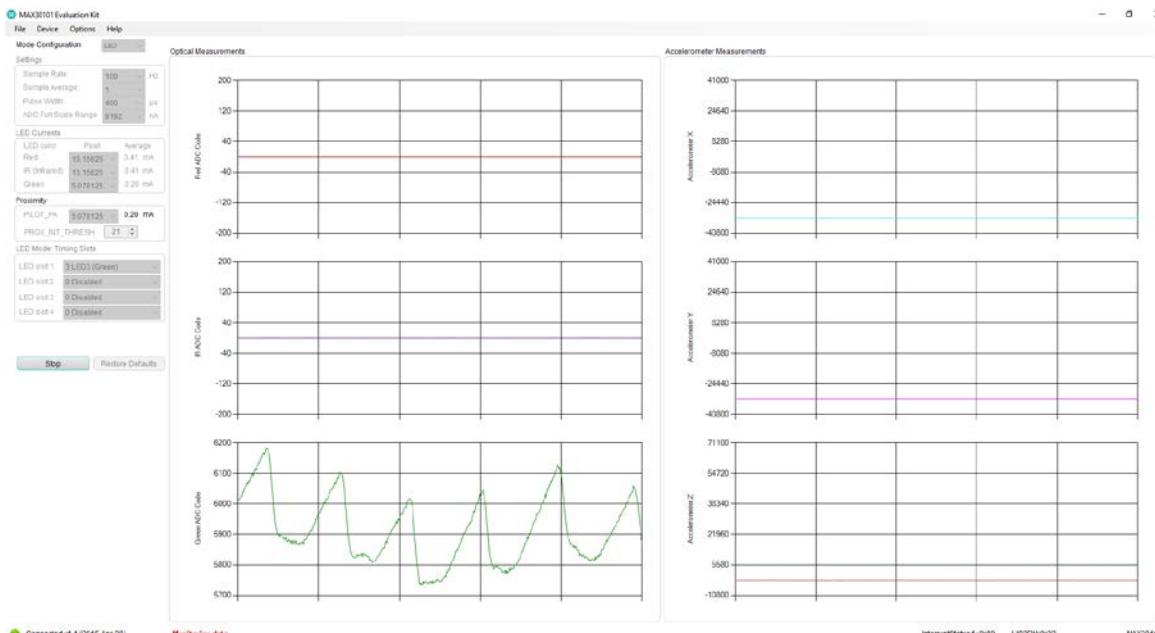


Figure 7. Data recorded by the GUI when a Green LED is used at 0.20mA.

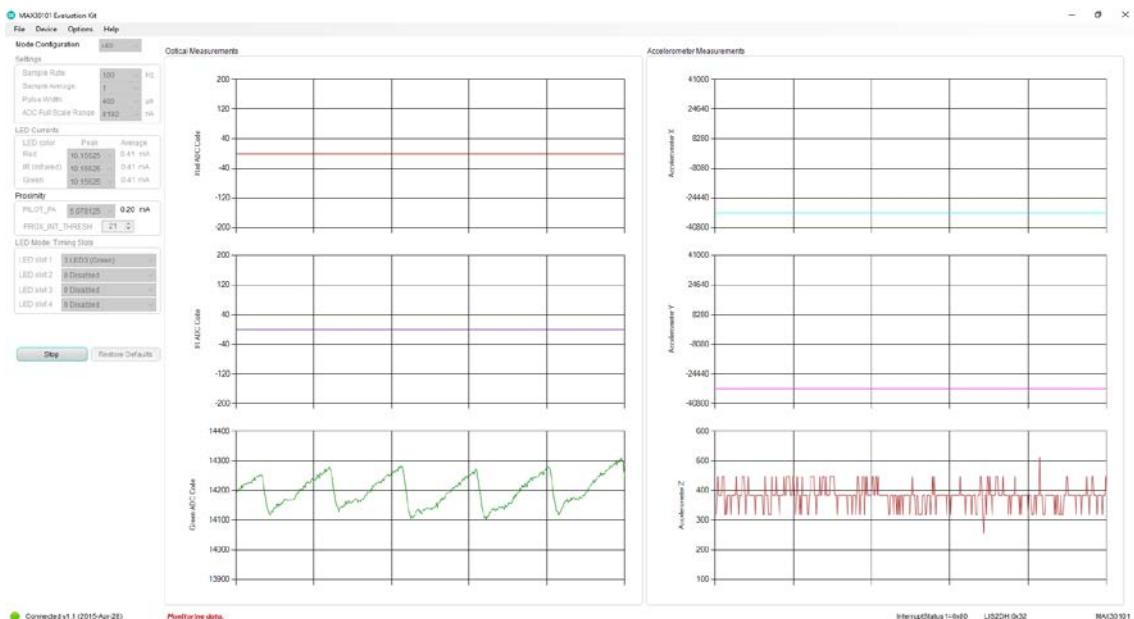


Figure 8. Data recorded by the GUI when a Green LED is used at 0.41mA.

The MAX3010x digitizes the photo-diode signal using an internal ADC. As can be seen in the figures above, Red and IR-ADC code “flatlined” at 0 as these LEDs were disabled. The motion was kept to a minimum, as can be seen by the flat line on the accelerometer axes. While keeping all settings the same in each configuration (Figure 5 to Figure 8), only the LED currents were changed. As the GUI was designed to auto-scale the ADC output signal, the y-axis range changes as displayed in the above figures. The ADC count (on the y axis of Green ADC code) shows the peak-to-peak counts increased as the current was increased. Higher LED current results in a brighter LED. These figures illustrate that no HR signal was captured at 0.04mA (peak = 1). The current had to be about 0.20mA (peak = 5.078125) for MAX30101 to capture the signal.

Note: The user can change settings to measure an improved HR signal if necessary.

Expected SpO₂ Signals

Using the same procedure mentioned above for the **Expected Heart-Rate Signals**, LED currents can be minimized, but need to be kept high enough so that a good SpO₂ signal is captured. Since Red and IR LEDs are being used, the user can configure each LED amplitude (i.e., current) to find the optimal configuration for their application. **Figure 9** shows a possible configuration when SpO₂ mode is being used. A 0.41mA (peak = 10.15625) average signal setting is used for both LEDs. The user should note that the two signals (Red and IR) are in sync while their amplitudes have different strengths. Post processing requires the user to first normalize the data, as mentioned in the **In-Depth Discussion**, and then use the normalized Red and IR to find a ratio corresponding to the SpO₂.

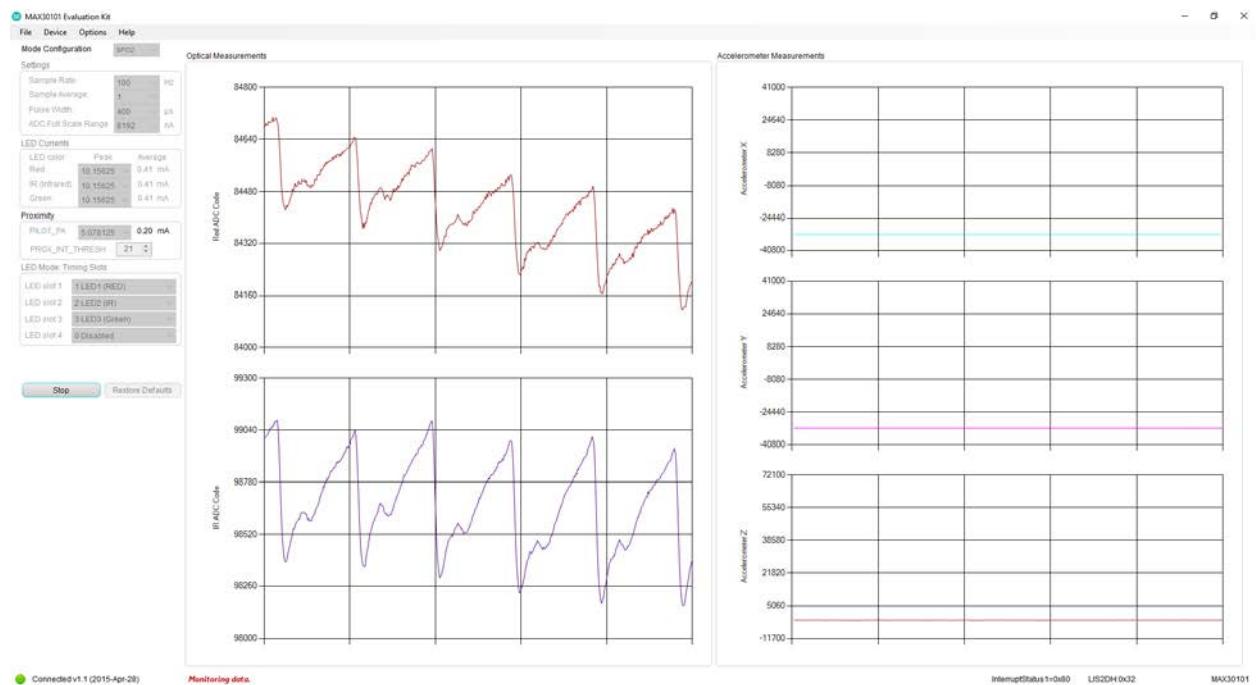


Figure 9. Data recorded by the GUI when Red and IR LEDs are used at 0.41mA.

General Recommendations

Before proceeding, it should be noted that a pulse width tells how long in time a signal is active for. In terms of power consumption, the longer a signal is active, the more energy it consumes. The drive frequency for the LEDs is selected by the sample rate. Correspondingly, a higher sample rate results in a higher drive frequency, thereby consuming more energy.

The ideal option for a wearable application is to choose lower pulse width combined with a lower sample rate to minimize energy consumption. However, this does not work since pulse width has an inverse relationship with the sample rate. This follows from the reasoning that a higher pulse width corresponds to a lower drive frequency and vice versa.

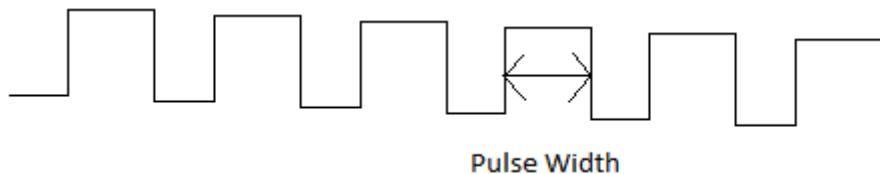


Figure 10. Pulse width example.

Figure 11 and **Figure 12** show what the allowed settings are for HR (single LED mode), and SpO₂ (dual LED mode). The shaded boxes correspond to settings that are not allowed. The user should adjust the pulse width and samples per second to determine the best settings for their application.

SAMPLES PER SECOND	PULSE WIDTH (μs)			
	69	118	215	411
50	O	O	O	O
100	O	O	O	O
200	O	O	O	O
400	O	O	O	O
800	O	O	O	O
1000	O	O	O	O
1600	O	O	O	
3200	O			
Resolution (bits)	15	16	17	18

Figure 11. Allowed settings for heart rate configuration.

SAMPLES PER SECOND	PULSE WIDTH (μs)			
	69	118	215	411
50	O	O	O	O
100	O	O	O	O
200	O	O	O	O
400	O	O	O	O
800	O	O	O	
1000	O	O		
1600	O			
3200				
Resolution (bits)	15	16	17	18

Figure 12. Allowed settings for SpO2 configuration.

SPO2 Register-Level Operation

The default mode is for pulse-oximetry (SpO2) measurements using Red and IR LEDs only. These selections are bolded on the GUI screen.

Register Map and General Guidelines

Figure 13 is also provided in the MAX30101 data sheet. The different registers are grouped together for convenience.

The Mode Configuration (0x09) register is used for shutdown control (SHDN), reset control (RESET), HR mode, SpO2 mode, and multi-LED mode. Shutdown control allows the user to put MAX30101 in power-saving mode. When the reset bit is set to one, all configuration, threshold, and data registers are reset to their power-on state through a power-on reset. Mode [3:0] allows the user to choose the modes such as HR, SpO2, or Multi-LED.

Mode Configuration (0x09)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
Mode Configuration	SHDN	RESET					MODE[3:0]		0x09	0x00	R/W

MODE[2:0]	MODE	ACTIVE LED CHANNELS
000		Do not use
001		Do not use
010	Heart Rate mode	Red only
011	SpO2 mode	Red and IR
100–110		Do not use
111	Multi-LED mode	Green, Red, and/or IR

SpO2 Configuration (0x0A)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
SpO2 Configuration		SPO2_ADC_RGE<1:0>		SPO2_SR[2:0]		LED_PW[2:0]		0x0A	0x00	R/W	

SPO2_ADC_RGE[1:0]	LSB SIZE (pA)	FULL SCALE (nA)
00	7.81	2048
01	15.63	4096
02	31.25	8192
03	62.5	16384

SPO2_SR[2:0]	SAMPLES PER SECOND
000	50
001	100
010	200
011	400
100	800
101	1000
110	1600
111	3200

Figure 13. Mode register, available modes, and SpO2 registers.

SpO2_ADC_RGE corresponds to SpO2 ADC Range Control where bits can set the SPO2 sensor ADC's full-scale range from 2048nA to 16384nA. The default full-scale range is 8192nA.

SPO2_SR corresponds to SpO2 Sample Rate Control where bits can set the effective sampling rate with one sample consisting of one IR pulse/conversion and one Red pulse/conversion. The default sampling rate is 100 samples per second (Hz).

LED_PW corresponds to LED pulse width. IR, Red, and Green have the same pulse width, but this pulse width can be varied from 69 μ s to 411 μ s. ADC resolution is proportional to the pulse width. As pulse width is increased, ADC resolution is also increased. This is shown in **Figure 14**.

LED_PW[1:0]	PULSE WIDTH (μ s)	ADC RESOLUTION (bits)
00	69 (68.95)	15
01	118 (117.78)	16
10	215 (215.44)	17
11	411 (410.75)	18

Figure 14. LED pulse-width control.

In SpO2 mode, each sample consists of 6 bytes of data and each byte requires an I²C read to read a sample. Refer to **FIFO** section for a detailed discussion on FIFO operation.

LED-Pulse Amplitude and Current Control

As SpO2 mode and Heart rate mode both employ LEDs, the following section applies to both modes. It provides register level insight into LED pulse control.

Figure 15 is provided in the MAX30101 data sheet. The different registers are grouped together for convenience.

LED-pulse amplitude registers (0x0C to 0x10) allow the user to set the typical LED current in millamps (mA). This can be adjusted from 0.0mA to 50mA peak amplitude.

Multi-LED mode allows the user to determine which LEDs are active. This mode is divided into a maximum of four FIFO time slots which can be modified using SLOT bits (i.e., SLOT 2 [2:0], SLOT 1 [2:0]). The slots also determine the size of the sample. For example, if one slot is being used, then one sample is 3 bytes long. If two slots are being used, then the sample size becomes 6 bytes long, and so on. The figure below also shows which bit configuration should be used for the LED being used. For instance, to turn on Green LED, the SLOT bit setting should be 011.

Multi-LED mode can also be used in proximity mode. Here the PILOT_PA bits set LED power and SLOT settings 101, 110, and 111 allow the user to choose active LEDs during proximity mode.

LED Pulse Amplitude (0x0C–0x10)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
LED Pulse Amplitude	LED1_PA[7:0]				LED2_PA[7:0]				0x0C	0x00	R/W
									0x0D	0x00	R/W
LED Pulse Amplitude	LED3_PA[7:0]								0x0E	0x00	R/W
RESERVED									0x0F	0x00	R/W
Proximity Mode LED Pulse Amplitude	PILOT_PA[7:0]								0x10	0x00	R/W

LEDx_PA [7:0], RED_PA[7:0], IR_PA[7:0], or G_PA[7:0]	TYPICAL LED CURRENT (mA)*
0x00h	0.0
0x01h	0.2
0x02h	0.4
...	...
0x0Fh	3.1
...	...
0x1Fh	6.4
...	...
0x3Fh	12.5
...	...
0x7Fh	25.4
...	...
0xFFh	50.0

Multi-LED Mode Control Registers (0x11–0x12)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
Multi-LED Mode Control Registers	SLOT2[2:0]			SLOT1[2:0]				SLOT3[2:0]			0x11 0x00 R/W
	SLOT4[2:0]										0x12 0x00 R/W

SLOTx[2:0] Setting	WHICH LED IS ACTIVE	LED PULSE AMPLITUDE SETTING
000	None (time slot is disabled)	N/A (Off)
001	LED1 (RED)	LED1_PA[7:0]
010	LED2 (IR)	LED2_PA[7:0]
011	LED3 (GREEN)	LED3_PA[7:0]
100	None	N/A (Off)
101	LED1 (Red)	PILOT_PA[7:0]
110	LED2 (IR)	PILOT_PA[7:0]
111	LED3 (GREEN)	PILOT_PA[7:0]

Figure 15. LED-pulse amplitude register and multi-LED mode control registers.

SpO₂ Recommended Configuration

A recommended methodology is presented here for determining an appropriate configuration and operating profile for the use of Maxim Integrated's MAX30101 SpO₂ sensor device. While differing applications (e.g., fingertip, forehead, ear, etc.) influence corresponding settings, the basic procedure is similar for most use cases.

Disclaimer: This document addresses only the signals that are derived from the MAX30101 device. That is, only the ADC outputs from the Red LED and Infrared LED channels are considered. Thus, settings are recommended that maximize signal-to-noise ratios, while preserving dynamic range and minimizing operating power. The ultimate test of performance vs. settings should be evaluated using an algorithm designed to process pulse oximeter digital data.

1. Set the SPO₂_ADC_RGE[1:0] to 00 to place the ADC in its highest gain setting (full scale = 2.048μA). Leave the Pulse Amplitude setting at the default value for now (approximately 10mA). Both the Red and IR channels should be set to the same value.

If the channel is saturated, adjust the gain setting to the next lower setting (SPO₂_ADC_RGE[1:0] = 01). Adjust the gain setting (SPO₂_ADC_RGE[1:0]) until the signal is somewhere between ¼ full scale (FS) and ¾ FS. The user should be looking at the average level of the DC + pulsatile signal.

Note: A pulsatile signal is typically less than 1% of the FS range.

Note: A user can use a setting level where operation is outside the ¼ to ¾ FS range. The above procedure is meant as a general guide to make a user aware of the dynamic range available. The use case needs to be fully characterized with multiple test subjects and appropriate ambient conditions. The gain setting (along with the pulse amplitude, width, sample rate, etc.) must be selected so that the sensor has appropriate dynamic range for the use case.

2. Next set the SpO₂ LED current settings (LED1_PA [7:0] and LED2_PA[7:0]). The default value is approximately 10mA. This value can typically be reduced to save power. The amplitude and pulse-width settings can be adjusted to lower power. However, this sets signal-strength level. The user must make the decision of acceptable signal strength vs. power level.

If the signal strength is low enough, the gain setting can be lowered, adjusting the dynamic range. If low power operation is desired, an iterative process between adjusting the dynamic range and the LED amplitude occurs until desired settings are found.

Figure 16 shows the MAX30101 SpO₂ signals. Distinct and synchronized pulsatile signals must exist on both the Red and IR channels for further SpO₂ processing. This graph is in our EV kit data sheet.

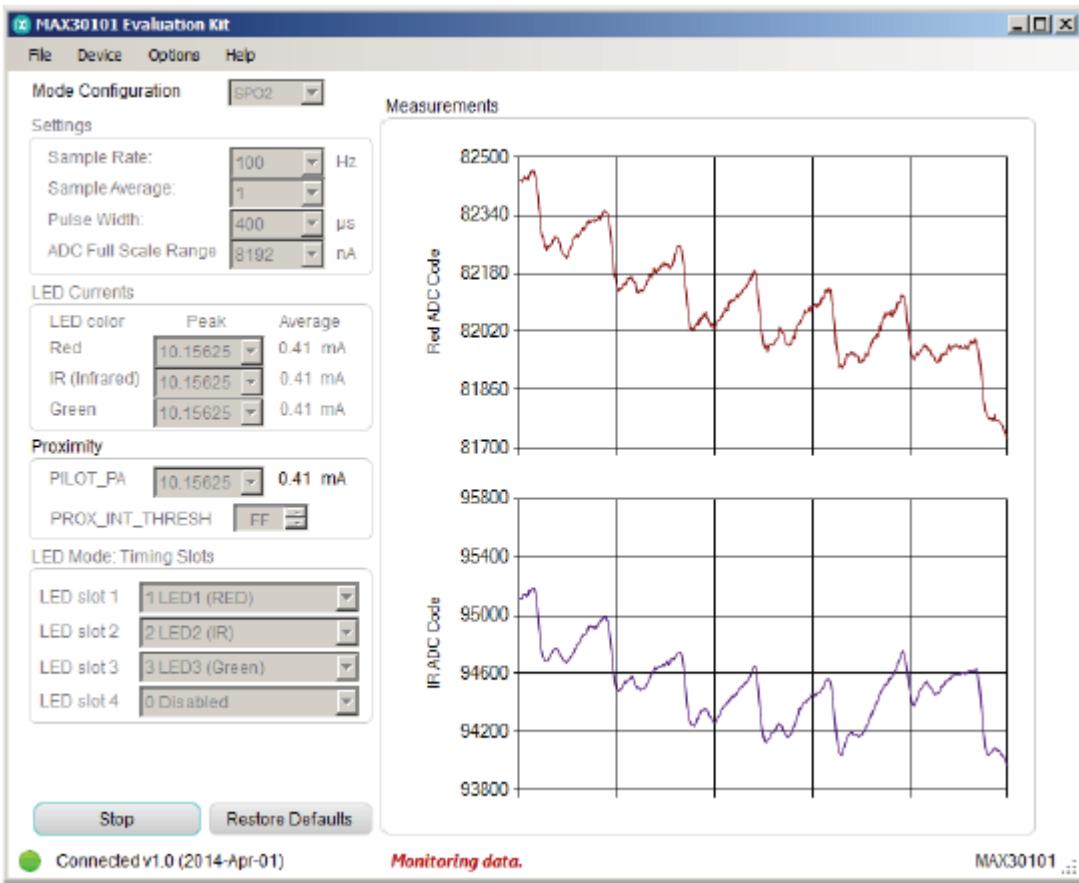


Figure 16. Red and IR showing SpO₂ signals in sync.

Using the EV kit, a user can lower the current amplitudes until the synchronized signals disappear (visually). Putting some margin on this setting (e.g., double the current setting) can give a user a starting point for use case evaluations.

Note: This setting level should not be used for determining the final setting configuration. It is highly recommended that a user collect data and set the performance level with the use of an SpO₂ algorithm. In this manner, a user can optimize the power setting vs. performance metric of choice (e.g., SNR, perfusion index, PPG var, etc.).

Heart Rate Monitoring for Wrist Application:

As mentioned previously, Red and IR LEDs can be used to measure heart rate. However, they are not as effective for wrist measurements. The wrist is generally not a convenient place for capturing HR and/or SpO₂ signals. A higher energy signal (Green LED) is used for wrist applications because of diminished blood perfusion in the wrist. To configure the EV kit for wrist HR measurements, several settings must be changed. The first is to change the Mode Configuration selection to “LED.” **Figure 17** shows the initial SpO₂ screen and the screen when the LED is selected.

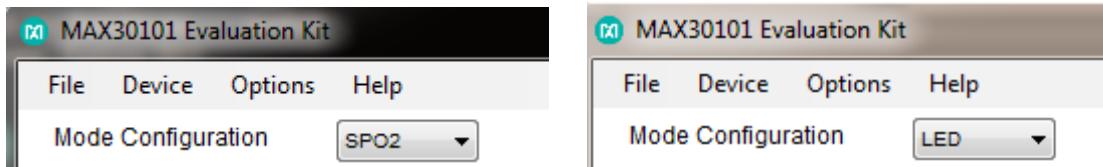


Figure 17a and 17b. SpO₂ mode and LED mode (for wrist HR applications).

When the “LED” Mode Configuration is selected, the LED Mode: Timing Slots window is highlighted. This mode implements all three LEDs in operation. **Figure 18** shows this default configuration.

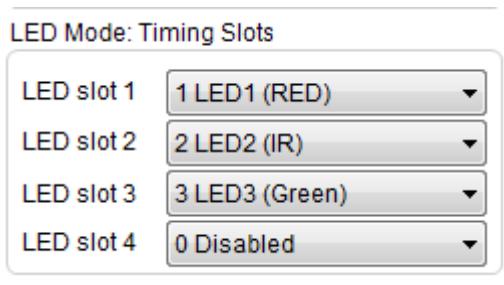


Figure 18. LED-mode timing slots.

Two Maxim Integrated example algorithms are included in the EV kit GUI. These basic algorithms are supplied for demonstration purposes only and Maxim Integrated does not guarantee their accuracy. Further, these algorithms do not provide motion compensation. To use the algorithms for wrist HR measurements, the following modifications are required:

1. Change LED slot 1 to “3LED3 (Green).”
2. Disable LED slot 2.
3. Disable LED slot 3 (automatically disables when slot 2 is disabled).

After these changes are made, the GUI screen looks like **Figure 19**.

LED Mode: Timing Slots

LED slot 1	3 LED3 (Green)
LED slot 2	0 Disabled
LED slot 3	0 Disabled
LED slot 4	0 Disabled

Figure 19. LED mode timing slots for wrist HR measurements.

Wrist HR measurements generally require higher signal levels. This can be accomplished by adjusting the ADC FS range to 4096nA and/or adjusting the LED currents (**Figure 20**).

LED Currents

LED color	Peak	Average
Red	10.15625	0.41 mA
IR (Infrared)	10.15625	0.41 mA
Green	25	1.00 mA

Figure 20. Changing the Green LED current.

While the GUI is not operating, all three plots (Red, IR, and Green) are visible on the GUI screen. As soon as Start Monitor is engaged, only the Green LED measurement is active (**Figure 21**).

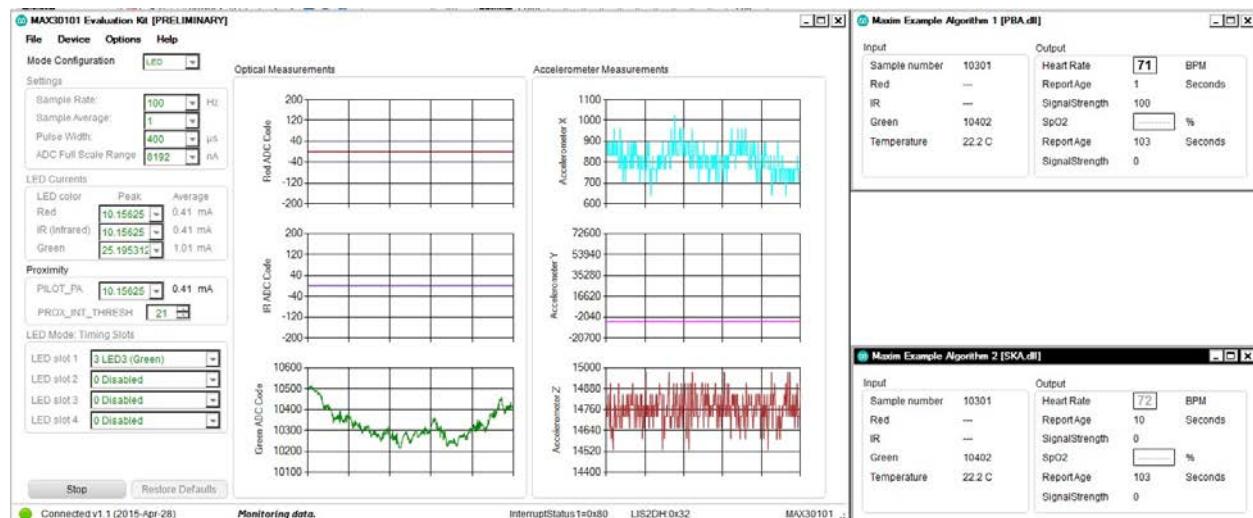


Figure 21. Wrist HR measurement using the MAX30101 EV kit.

The configuration of the LED3 (Green) through LED slot 1 allows the computation of the heart rate using the simple algorithms that are supplied with the EV kit GUI. The next section shows how a user can increase the Green LED signal.

Increasing the Green LED Signal

Within the Integrated Circuit, Channel 3 (i.e., Green LED) is tied internally to Channel 4. Thus, a user can configure the drive signal to a higher level. While the GUI is running, select “Options” from the pull-down tab. Next, select “Register Access” (**Figure 22**).

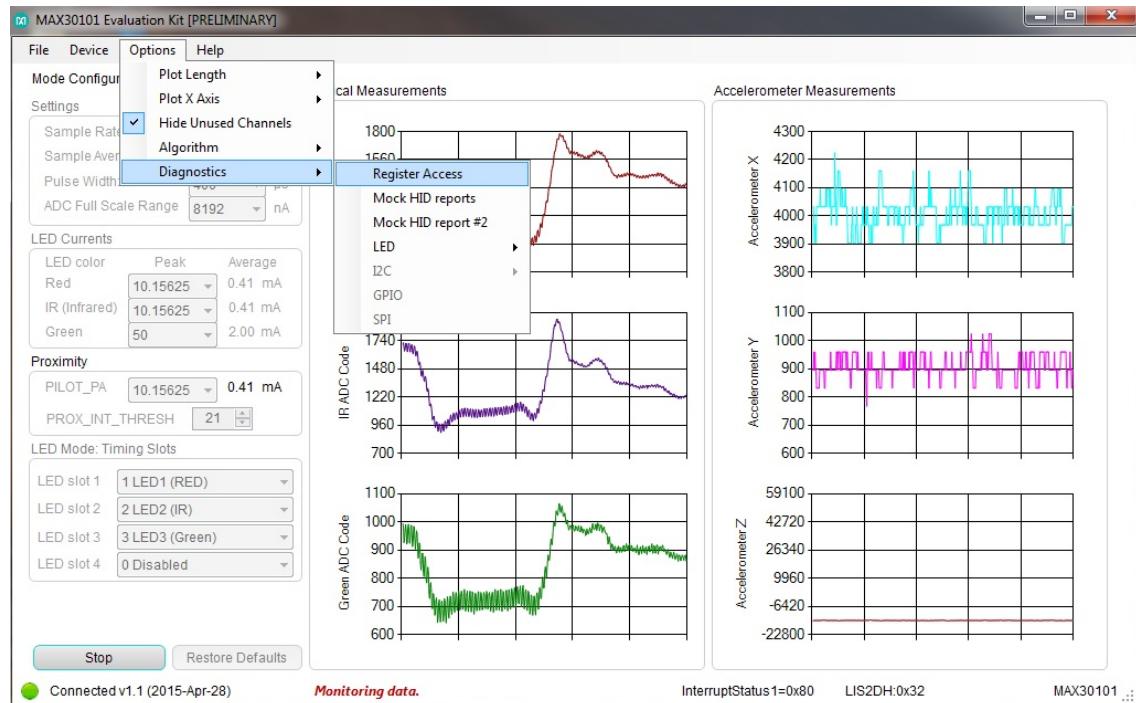


Figure 22. Register access while the GUI is operating.

Four new windows appear showing a “Registers” selection, value, Read Command (“Rd”) button, and a Write Command (“Wr”) button (**Figure 23**).

The changing of register values can only occur while the GUI is operational. As soon as the GUI is stopped, the S/W returns to its default setup configuration.

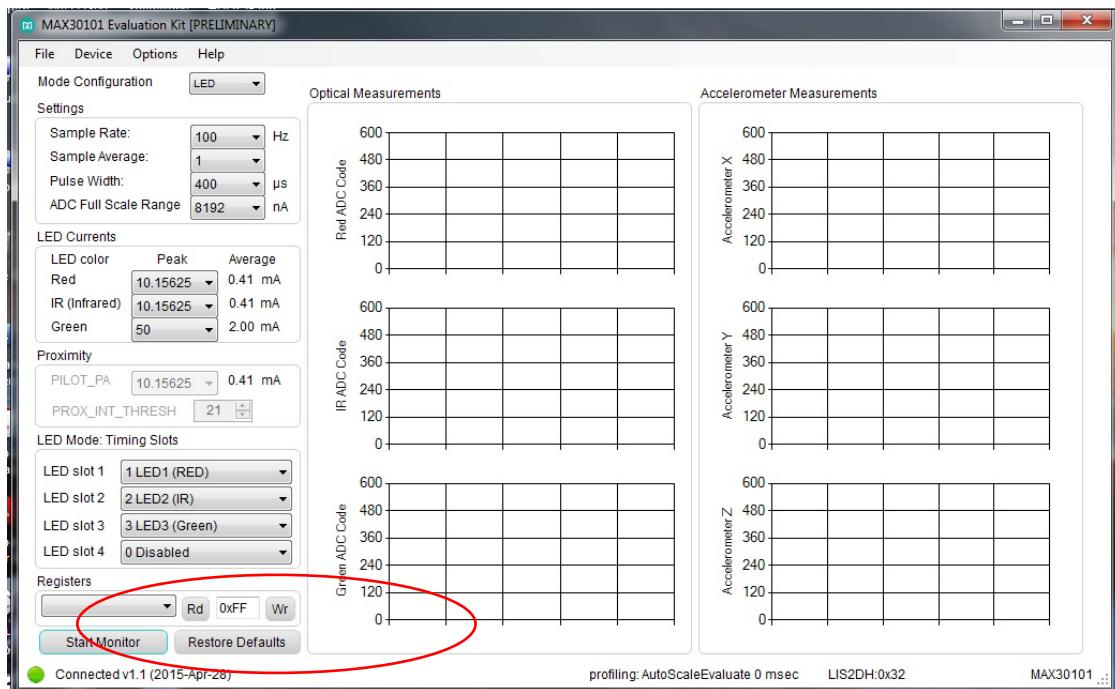


Figure 23. GUI register access.

Next, select the “LED4GreenPulseAmp” register, fill in the value of “0xFF,” and hit “Wr.” The Green LED is configured to its maximum level of approximately 80mA. Any value between “0x00” and “0xFF” can be selected (**Figure 24**).

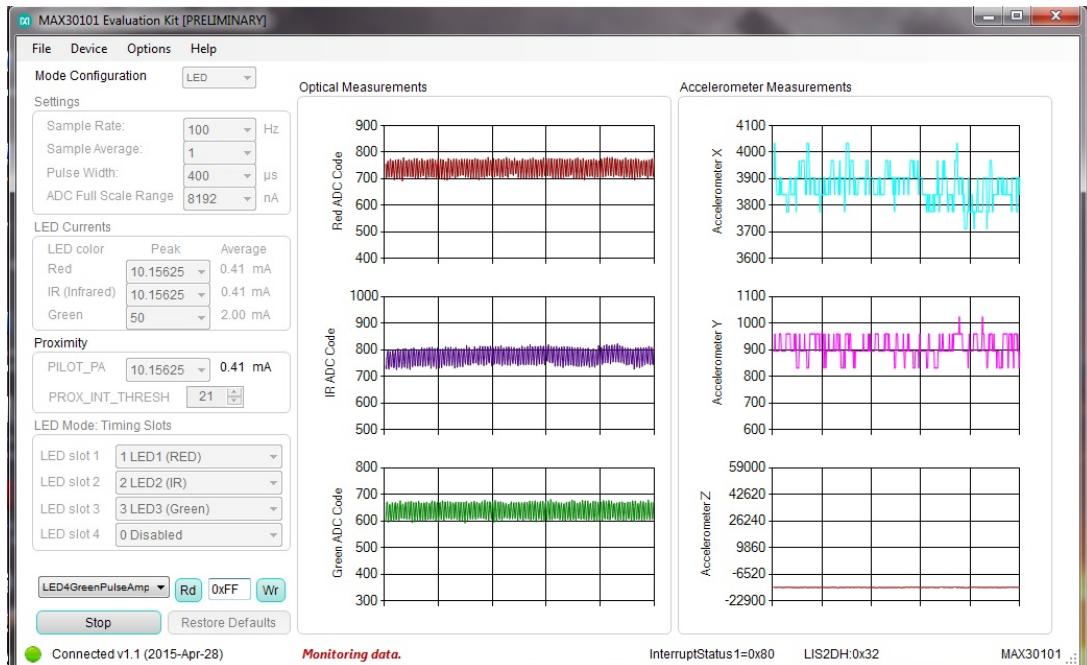


Figure 24. Input values in LED4GreenPulseAmp register.

Register Map

Interrupts

Figure 25 is provided in the MAX30101 data sheet.

Interrupt Status (0x00–0x01)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
Interrupt Status 1	A_FULL	PPG_RDY	ALC_OVF	PROX_INT				PWR_RDY	0x00	0x00	R
Interrupt Status 2							DIE_TEMP_RDY		0x01	0x00	R

Interrupt Enable (0x02-0x03)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
Interrupt Enable 1	A_FULL_EN	PPG_RDY_EN	ALC_OVF_EN	PROX_INT_EN					0x02	0x00	R/W
Interrupt Enable 2							DIE_TEMP_RDY_EN		0x03	0x00	R/W

Figure 25. Interrupt status and interrupt enable registers.

The interrupts triggered by Interrupt status 1 and Interrupt status 2 are as follows: FIFO almost full flag (A_FULL), new FIFO data ready (PPG_RDY), ambient light cancellation overflow (ALC_OVF), proximity threshold triggered (PROX_INT), power ready flag (PWR_RDY).

As ambient light can easily introduce error into the HR and SpO2 measurement, MAX30101 incorporates an Ambient Light Correction (ALC) feature that removes ambient light. Generally, the goal is to minimize the ambient light to isolate the desired signal. As such, ambient light should be kept in mind when designing enclosures and spacing of the photodiodes from the skin. In terms of the register map, it is important to note the ALC_OVF interrupt indicates that ambient light cancellation function has reached its maximum light and is affecting the output of ADC.

Data Streaming and FIFO Operation

Figures 26 and 27 are provided in the MAX30101 data sheet.

FIFO (0x04–0x07)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
FIFO Write Pointer					FIFO_WR_PTR[4:0]						0x04 0x00 R/W
Over Flow Counter					OVF_COUNTER[4:0]						0x05 0x00 R/W
FIFO Read Pointer					FIFO_RD_PTR[4:0]						0x06 0x00 R/W
FIFO Data Register				FIFO_DATA[7:0]						0x07 0x00 R/W	

ADC Resolution	FIFO_DATA[17]	FIFO_DATA[16]	⋮	FIFO_DATA[12]	FIFO_DATA[11]	FIFO_DATA[10]	FIFO_DATA[9]	FIFO_DATA[8]	FIFO_DATA[7]	FIFO_DATA[6]	FIFO_DATA[5]	FIFO_DATA[4]	FIFO_DATA[3]	FIFO_DATA[2]	FIFO_DATA[1]	FIFO_DATA[0]
18-bit																
17-bit																
16-bit																
15-bit																

BYTE 1										FIFO_DATA[17]	FIFO_DATA[16]
BYTE 2	FIFO_DATA[15]	FIFO_DATA[14]		FIFO_DATA[13]	FIFO_DATA[12]	FIFO_DATA[11]	FIFO_DATA[10]	FIFO_DATA[9]	FIFO_DATA[8]		
BYTE 3	FIFO_DATA[7]	FIFO_DATA[6]		FIFO_DATA[5]	FIFO_DATA[4]	FIFO_DATA[3]	FIFO_DATA[2]	FIFO_DATA[1]	FIFO_DATA[0]		

FIFO Configuration (0x08)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
FIFO Configuration	SMP_AVE[2:0]			FIFO_ROL LOVER_EN	FIFO_A_FULL[3:0]				0x08	0x00	R/W

SMP_AVE[2:0]		NO. OF SAMPLES AVERAGED PER FIFO SAMPLE					
000		1 (no averaging)					
001		2					
010		4					
011		8					
100		16					
101		32					
110		32					
111		32					

Figure 26. FIFO registers.

FIFO_A_FULL[3:0]	ALMOST FULL INTERRUPT TRIGGER (NO. OF SAMPLES IN THE FIFO)
0x0h	0
0x1h	1
0x2h	2
0x3h	3
...	...
0xFh	15

Figure 27. FIFO_A_FULL registers and how they correspond to number of samples in FIFO.

FIFO_WR_PTR (FIFO write pointer) points to where MAX30101 writes the next sample. Every time a new sample is pushed onto the FIFO, this pointer advances. FIFO_RD_PTR (FIFO read pointer) points to where the processor gets the next sample from FIFO. Every time a sample is pushed from the FIFO, this pointer advances. OVF_COUNTER (FIFO overflow counter) counts number of samples lost when the FIFO can no longer accept new samples. It is reset to zero when a complete sample is pushed from the FIFO.

Note: Often in applications, the user might want to reread their samples from FIFO in case their data gets corrupted. Decrease the value of FIFO_RD_PTR by one to reread the FIFO_DATA register.

While the FIFO can hold up to 32 samples of data, the FIFO depth can be adjusted depending on the number of (LED) channels used. FIFO can store Green, IR, and Red ADC data. Since each sample consists of three channels and each channel signal is stored as a 3-byte data signal, this equates to 9 bytes of data per sample.

Note: Reading or burst reading the FIFO_DATA register does not automatically advance it. Similarly, reading bytes after 0xFF does not reset the address pointer to 0x00.

SMP_AVE (bits 7:5 in 0x08) are used to select number of samples averaged per FIFO sample. Adjacent samples in each channel can be averaged by writing to this.

FIFO_ROLLOVER_EN bit allows the user to update the FIFO. When the FIFO is completely full, room for new data can only be made if FIFO_DATA is read or if FIFO_WR_PTR/FIFO_RD_PTR positions are changed. In this case, the purpose of the FIFO_ROLLOVER_EN bit is that when it is set, the FIFO address rolls to zero and allows new data to fill the FIFO.

FIFO_A_FULL bits allow the user to determine the number of samples in the FIFO by adjusting when to trigger A_FULL interrupt (bit 7 in Interrupt Status 1 register).

Recommended Practices:

- Reset (0x00) FIFO_WR_PTR, FIFO_RD_PTR, and OVF_COUNTER when starting a new heart rate or SpO2 conversion.
- Try to avoid writing to FIFO_WR_PTR, OVF_COUNTER, and FIFO_DATA. They automatically advance when data is inputted by MAX30101. Only FIFO_RD_PTR should be written if needed (to reread samples for instance).

Proximity Detection

Note: Entering or exiting the proximity mode clears the FIFO. A flowchart (Figure 28) is provided for setting up proximity detection. This provides a general guideline and the user should use their own code for implementing proximity detection while keeping this in mind. Although the flowchart stops at “end,” it should be noted that “end” means that proximity detection has been verified.

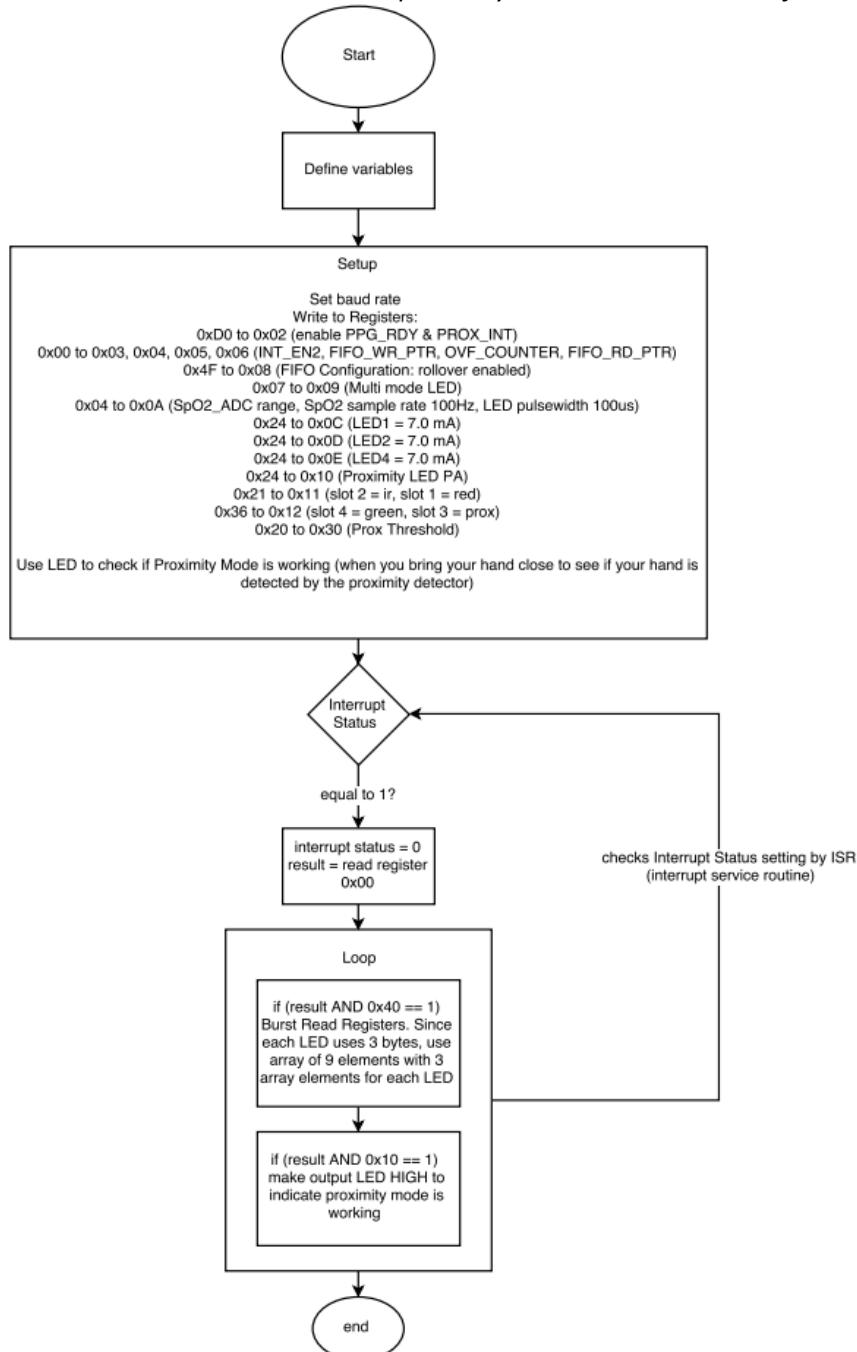


Figure 28. General guidelines for setting up and verifying the proximity detection function.

Note: Proximity detection usually uses IR for detection, thereby avoiding visual detection of visible light.

Post-Processing

Post-processing requires several steps which include filtering, peak-to-peak detection, normalization, and digital signal processing. Many users usually want a simpler way of understanding the raw data that they are capturing. A basic method shows the use of the ADC code that is visible in the GUI (**Figure 29**).

Note: For useful measurements, use the appropriate post-processing techniques.

Heart-Rate Post-Processing Using ADC Code

Go to the file, choose “log,” and save the file. Then, with your finger touching the sensor, click on “start monitor.” The heart-rate signal can then be verified visually from the graphical plot on the GUI. Collect the data for a few seconds, then go back to file and click on “log” to finish saving it. Next, open the file in Microsoft Excel, or any spreadsheet software, and graph “Raw data” on the y axis and the time on the x axis. The user should get a graph like the one in **Figure 29**.

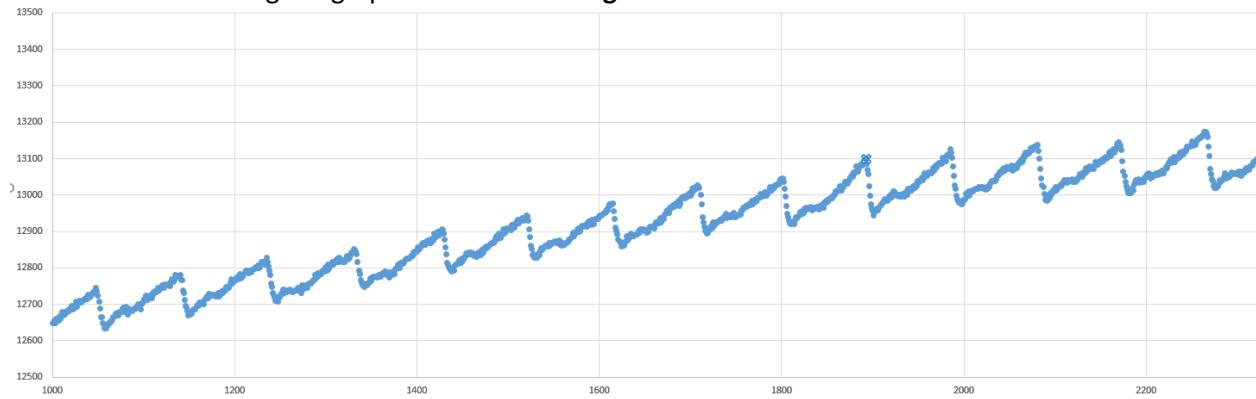


Figure 29. ADC code plotted with respect to time.

One period is shown in **Figure 30**. The numbers 1 and 2 identify the two peaks (1 = systolic, 2 = diastolic) in the signal. Heart rate is found by measuring the time between consecutive systolic peaks and then multiplying it by 60. It should be noted that often the time (on the horizontal axis) is not in seconds which is why the user should convert it before calculating their heart rate. For this example, HR was calculated to be 53 bpm.



Figure 30. One period of the ADC code plotted with respect to time.

SpO₂ Post-Processing Using ADC Code

A similar procedure was followed to log Red and IR data from the GUI. **Figure 31** was plotted in Microsoft® Excel®.

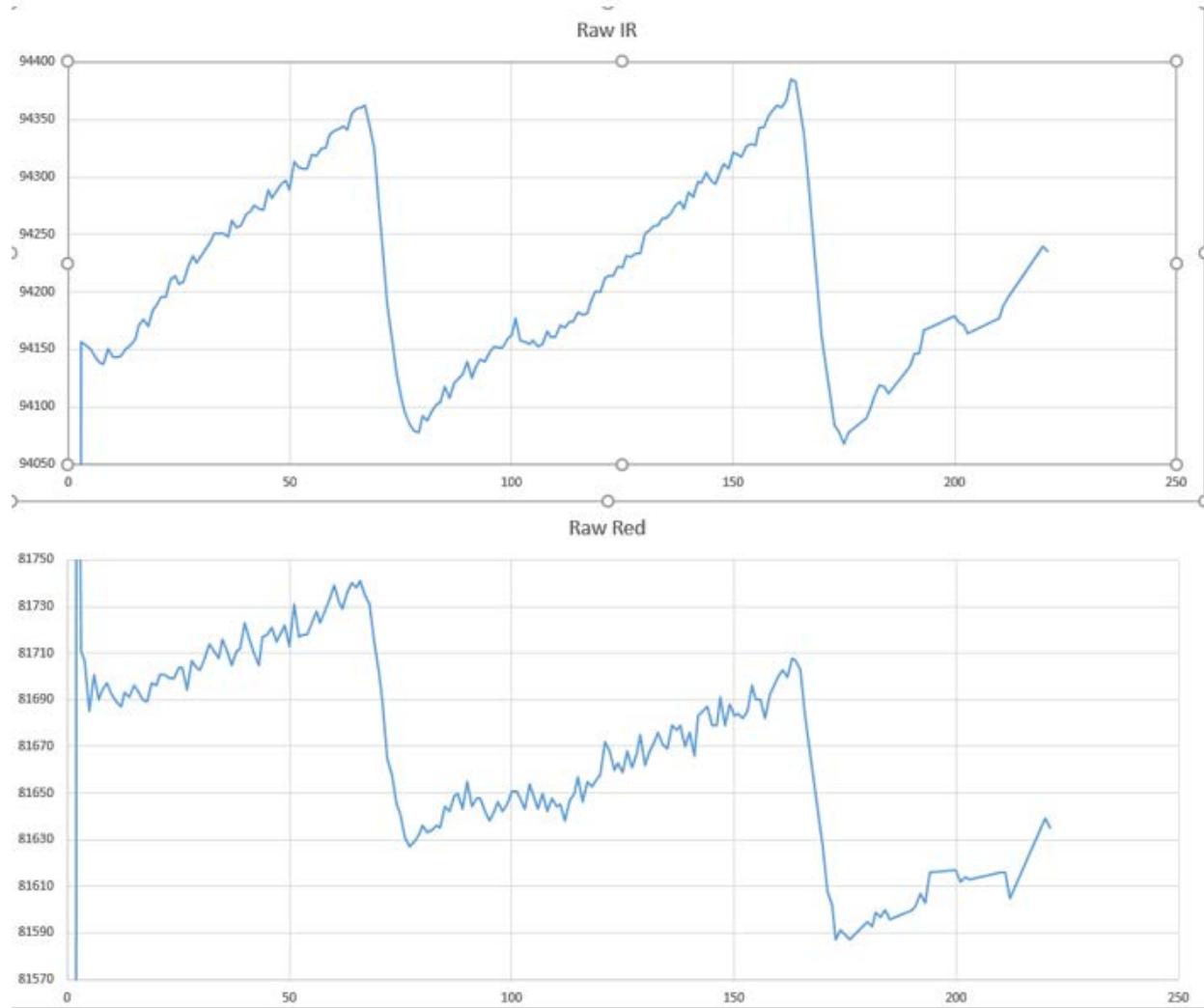


Figure 31. IR and Red Raw data for SpO₂ processing.

As mentioned in the *In-Depth Discussion*, DC and AC components should be verified first. The next step is to normalize the two (by taking the ratio of AC to DC) and then finding a value “R,” which is the ratio of normalized Red to normalized IR data. Finally, a linear approximation can be used to find SpO₂.

In Figure 31, the DC component was found by using a straight-line approximation between two valleys. Let's label the points first. Call the valley between 50 to 100 “1” and the valley between 150 to 200 “3.” The peak between 150 and 200 is called “2.”

The DC component is the offset of the AC signal, which can be found by first drawing a line between 1 and 3 and then drawing a line parallel to the y axis from 2 to the line connecting 1 and 3. The DC value is the point where the two lines intersect.

The AC component on the other hand is the distance between the peak (“2”) and the DC value. These calculations are shown in **Figure 32**.

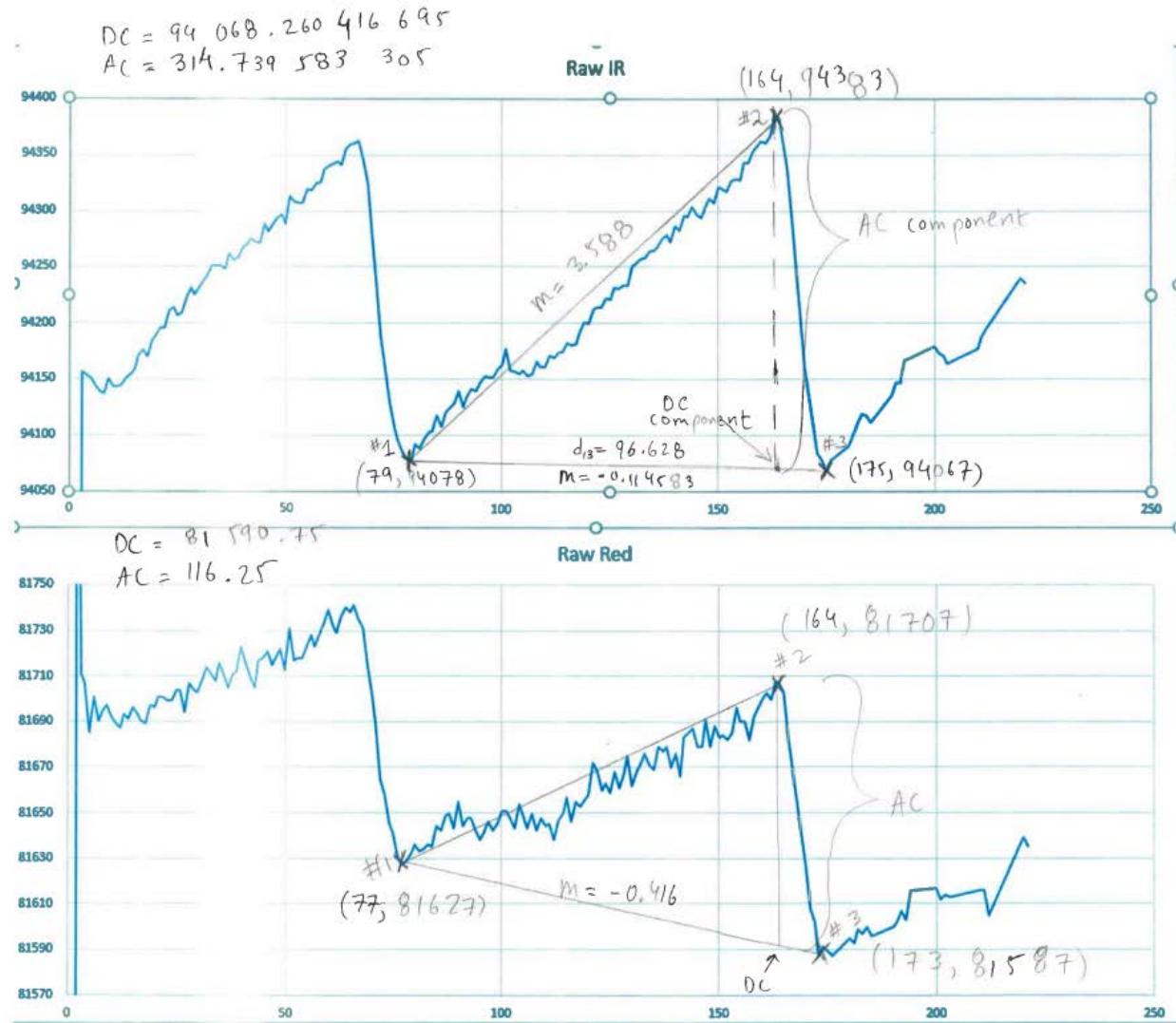


Figure 32. Hand calculations needed for SpO₂.

The AC and DC components are labeled above each graph. The following equations were used to first find "R" and then SpO₂.

$$R = \frac{\frac{AC_{red}}{DC_{red}}}{\frac{AC_{IR}}{DC_{IR}}} = \frac{\frac{116.25}{81590.75}}{\frac{314.739583305}{94068.260416695}} = 0.42583738237923$$

$$SpO_2 = 104 - 17R = 104 - 17(0.43) = 96.8\%$$

Enclosure Consideration

Since the MAX3010x is used for optical applications, the user must minimize the amount of ambient light exposure to maximize its performance. Ambient light consists of all the light that is not part of the intended signal (i.e., sunlight, room lights, etc.). Additionally, as the enclosure can be in contact with human skin for long periods of time (e.g., fitness applications), an enclosure made of biocompatible material should be used. It should be kept in mind that while MAX30101 works the best when it is in direct contact with human skin. The enclosure typically incorporates some distance between MAX30101 and skin which causes measurement degradation.

With these factors in mind, the following recommendations are given:

- 1) Use optical walls to minimize crosstalk between LEDs and photodiodes.
- 2) Create enclosure designs that minimize ambient light.
- 3) Use biocompatible materials for the enclosure.
- 4) Use optical computer simulations to help guide the design of the sensor-tissue interface. The use of optical resins or optical-grade lens films (with air gaps) for water (i.e., sweat) resistance can greatly affect the optical system performance.

Troubleshooting

Firmware

If MAX30101 is disconnected without first exiting the GUI, the firmware on the microcontroller might get corrupted. This can be confirmed by disconnecting and then connecting the microcontroller. If the two lights (green and red) on the microcontroller do not turn on or if they keep flashing without stopping, then the firmware might be corrupted. In this scenario, the firmware must be reflashed and tested again using the GUI. The suggested software for this operation is Silicon Labs Flash Programming Utility. It is not provided with the EV kit but can be found on the Silicon Labs website.

A step-by-step procedure is shown in **Figure 33** for reflashing firmware by connecting a Silicon Labs USB Debug Adapter to pins (J2) and PC. It should be noted, however, that if the pins (J2) are not provided, as they are removed sometimes when packaging the microcontroller in an enclosure, the user cannot flash new firmware.

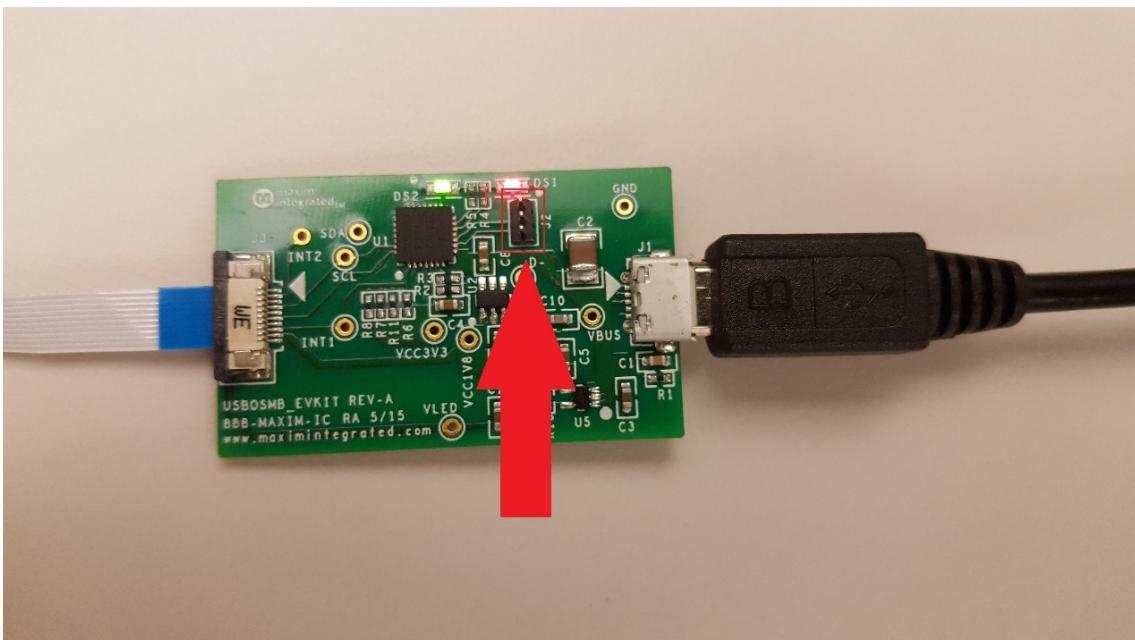


Figure 33. Where the USB Debut Adapter should be connected for reflashing.

Download Silicon Laboratories Flash Utility (**Figure 34**).

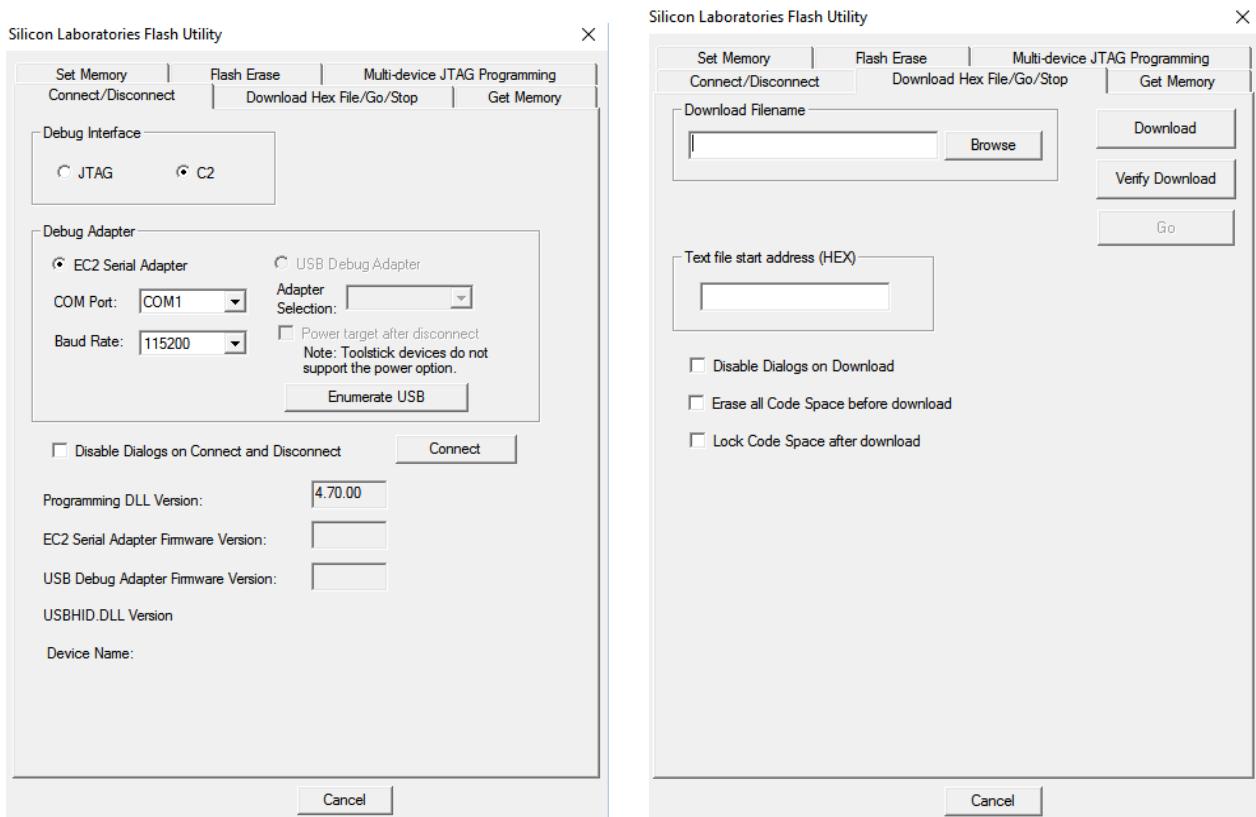


Figure 34. Silicon Laboratories Flash Utility.

The following steps flash the firmware on the microcontrollers:

- **Enumerate USB** (since Silicon Labs USB Debug Adapter is being used).
- **Select USB Debut Adapter.**
- **Select Connect.**
- Click on the tab **Download HEX File/Go/Stop**.
- Browse and select the hex file for firmware.
- **Select Download.**
- Go back to the tab **Connect/Disconnect**.
- **Select Disconnect.**

Ambient Light

Sometimes the ambient light in the room can also affect the HR-SpO₂ measurements. The flickering of lights in a room or light coming from a monitor or laptop can contribute noise to the intended measurement.

The ‘spikes’ that can be seen below are characteristic of flickering lights. As such, if the user notices a regularity in spikes in the intended signal (**Figure 35**), they should try to go in a darker room or cover MAX30101 to minimize the ambient light. For comparison, **Figure 36** shows what the signal would look like if MAX30101 was used in a completely dark room.

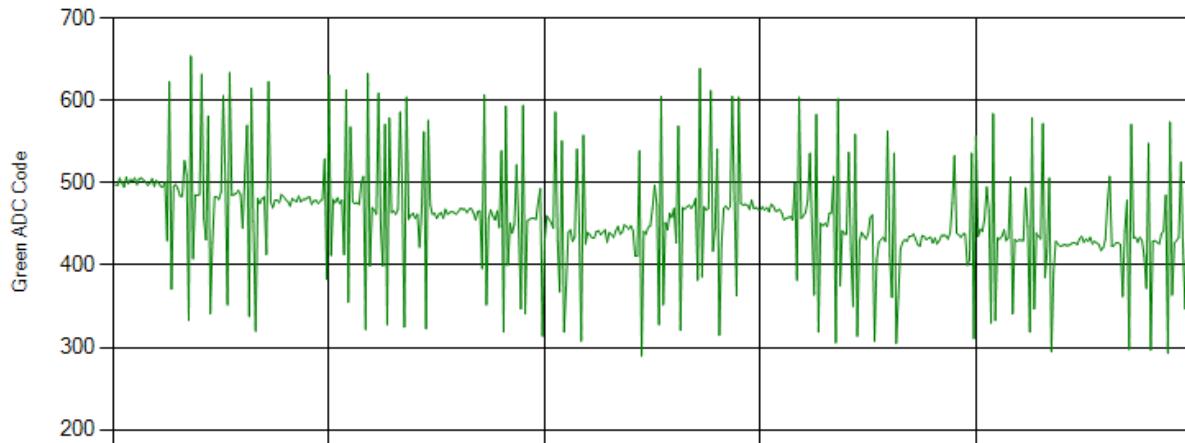


Figure 35. Regularity in spikes indicating possible ambient light.

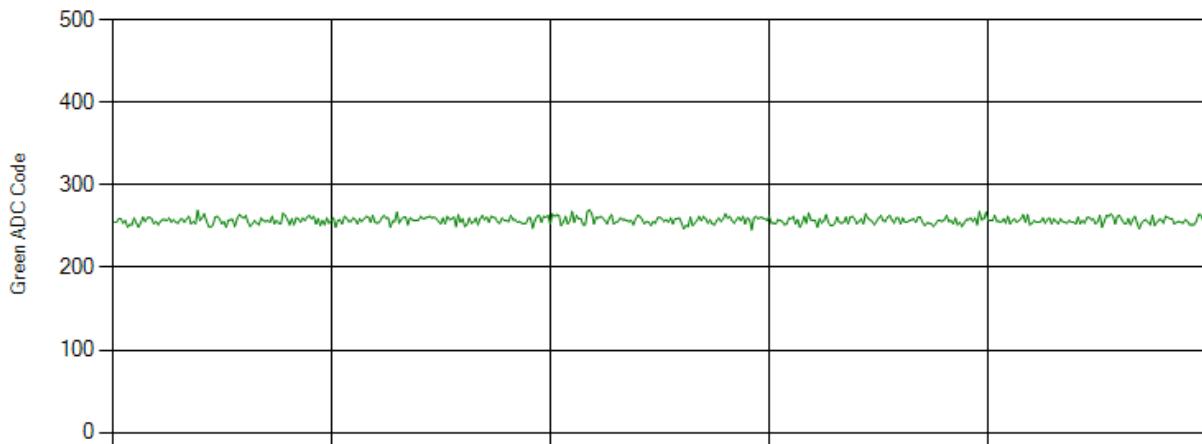


Figure 36. No spikes and low-ADC code indicating a dark room.

References

- [1] Wukitsch, M. W., Petterson, M. M. T., Tobler, D. R., & Pologe, J. A. (1988). Pulse oximetry: analysis of theory, technology, and practice. *Journal of Clinical Monitoring*, 4(4), 290-301.
- [2] Rusch, T. L., Sankar, R., & Scharf, J. E. (1996). Signal processing methods for pulse oximetry. *Computers in biology and medicine*, 26(2), 143-159.
- [3] Tabulated Molar Extinction Coefficient for Hemoglobin in Water, Scott Prahl, <http://omlc.org/spectra/hemoglobin/summary.html>
- [4] Vizbara, V., Solosenko, A., Stankevicius, D., Marozas, V. (2013) Comparison of green, blue, and infrared light in wrist and forehead photoplethysmography.

Figures

Figure 1: J. G. Webster, "Design of Pulse Oximeters", Series in Medical Physics and Biomedical Engineering, Taylor & Francis, New York, USA, 1997.

Trademarks

Excel and Microsoft are a registered trademarks and Microsoft is a registered service mark of Microsoft Corporation.

©2018 by Maxim Integrated Products, Inc. All rights reserved. Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. MAXIM INTEGRATED PRODUCTS, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. MAXIM ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering or registered trademarks of Maxim Integrated Products, Inc. All other product or service names are the property of their respective owners.



MAX32664 User Guide

UG6806; Rev 0; 01/19

Abstract

The MAX32664 user guide provides flow charts, timing diagrams, GPIOs/pin usage, I²C interface protocol, and annotated I²C traces between the host microcontroller and the MAX32664. Typical application uses the MAX32664 as a low-power microcontroller in a sensor hub configuration to provide processed data such as heart rate and SpO₂.

Table of Contents

Introduction.....	4
MAX32664 Variants	5
Maxim Reference Designs with MAX32664	6
MAXREFDES220#	6
MAXREFDES101#.....	7
MAX32664 GPIOs and RSTN Pin	8
MAX32664 Bootup and Application Mode	9
MAX32664 Bootloader Mode.....	9
MAX32664 Application Mode.....	9
Communications to the MAX32664 over I ² C.....	10
Bit Transfer Process.....	10
I ² C Write	12
I ² C Read.....	13
MAX32664 I ² C Message Protocol Definition.....	14
MAX32664 I ² C Annotated Application Mode Example	27
I ² C Commands to Flash the Application Algorithm.....	29
In-Application Programming of the MAX32664	32
MAX32664 APIs and Methods for Reset, Sleep, Status, Heartbeat.....	34
Revision History.....	35

List of Figures

Figure 1. MAX32664 block diagram.....	6
Figure 2. MAXREFDES101# block diagram.....	7
Figure 3. Pin connections between the host and the MAX32664.....	8
Figure 4. Entering bootloader mode using the RSTN pin and the MFIO GPIO pin.....	9
Figure 5. Entering application mode using the RSTN pin and MFIO pin.....	10
Figure 6. I ² C Write/Read data transfer from host microcontroller.....	10
Figure 7. Sequence to enter bootloader mode.....	29
Figure 8. Page number byte 0x44 from the .msbl file.....	29
Figure 9. Initialization vector bytes 0x28 to 0x32 from the .msbl file.....	29
Figure 10. Authentication bytes 0x34 to 0x43 from the .msbl file.....	30
Figure 11. Send page bytes 0x4C to 0x205B from the .msbl file.....	30
Figure 12. Sequence to enter application mode.....	31
Figure 13. MAX32664 in-application programming flowchart.....	33

List of Tables

Table 1. MAX32664 Variants, Matching Algorithms, and Reference Designs.....	5
Table 2. RSTN Pin and GPIOs Pins	8
Table 3. Additional GPIOs Used on the MAX32664 for the MAXREFDES220#	9
Table 4. Read Status Byte Value.....	11
Table 5. MAX32664 I ² C Message Protocol Definitions.....	14
Table 6. Sensor Hub Status Byte.....	25
Table 7. Output FIFO Format Definitions	25
Table 8. Sequence of Commands to Write External Accelerometer Data to the Input FIFO	27
Table 9. MAX32664GWEA I ² C Annotated Application Mode Example	27
Table 10. Annotated I ² C Trace for Flushing the Application	29
Table 11. MAX32664 I ² C Message Protocol Definitions	34

Introduction

The MAX32664 is a pre-programmed microcontroller with firmware drivers and algorithms. Combined with the appropriate sensor devices, the MAX32664 acts as a sensor hub to provide processed data to a host device. This solution seamlessly enables customers to receive raw and/or calculated data from Maxim's optical sensor solutions, while keeping overall system power consumption in check. The tiny form factor (1.6mm x 1.6mm 16-bump WLP) allows for integration into extremely small applications. The MAX32664 is integrated into Maxim's complete reference design solutions, which shortens the time to market.

The MAX32664 is the same hardware as the MAX32660 but with a pre-programmed bootloader that accepts in-application programming (IAP) of Maxim supplied algorithms and sensor drivers. The MAX32664 provides a fast-mode, I²C slave interface to a microcontroller host. A second I²C interface is dedicated to communicating with sensors.

For further details on memory, register mapping, system clocks, reset, power management, GPIOs/alternate functions, DMA controller, UART, RTC, timers, WDT, I²C, and SPI, see the MAX32660 User Guide.

For ordering information, mechanical and electrical characteristics, and the pinout for the MAX32664 family of devices, refer to the MAX32664 data sheet.

For information on the Arm® Cortex®-M4 with FPU core, refer to the Cortex-M4 with FPU Technical Reference Manual.

Arm is a registered trademark and registered service mark of Arm Limited.

Cortex is a registered trademark of Arm Limited.

MAX32664 Variants

The MAX32664 is pre-programmed with bootloader software that accepts in-application programming of Maxim supplied algorithms and the associated sensor drivers. The MAX32664 is used as a sensor hub controller.

The algorithm/application code provides processed and/or raw data through the I²C interface. Several variants of the MAX32664 exist based on the target application. These variants come pre-programmed with a bootloader that only accepts the matching encryption keys for the part (e.g., the MAX32664GWEA bootloader is pre-programmed with the A encryption key, reference designs are programmed with Z keying, etc.). Designers should use the table below in order to select the correctly keyed part.

Table 1. MAX32664 Variants, Matching Algorithms, and Reference Designs

PART NUMBER	APPLICATION FIRMWARE	BOOTLOADER KEY	MAXIM REFERENCE DESIGN
MAX32664GWEA	<p>MaximFast: Maxim Integrated® finger-based heart-rate and SpO₂ monitoring algorithm (100Hz sampling). The MaximFast algorithm is compatible with the sensor hub combination of the MAX32446GWEA, MAX30101 AFE, and KX-122 accelerometer. It is recommended, but not mandatory, to use an accelerometer with the MaximFast algorithm. Do not enable the accelerometer if there is no accelerometer in your design. If the KX-122 accelerometer is not installed in the design and external accelerometer data is supplied, then the accelerometer should use the 100Hz sampling rate.</p> <p>Automatic gain control (AGC) for MaximFast. If the AGC is enabled, the LED currents and pulse width are automatically determined by the algorithm. If the AGC is not enabled, the LED currents and pulse width registers should be configured by the host software.</p>	A	MAXREFDES220#

For all the MAX32664 parts, the algorithm (.msbl file) with the corresponding bootloader key must be downloaded, and these parts must be programmed using the in-application programming feature of the bootloader. The MAX32664 is only pre-programmed with the bootloader software.

Maxim Integrated is a trademark of Maxim Integrated Products, Inc.

Maxim Reference Designs with MAX32664

Maxim provides multiple reference designs to its customers to enable quick and effective adoption of MAX32664 and fastest time to market. For detailed schematics, refer to the user guide of each reference design.

MAXREFDES220#

The MAXREFDES220# reference design provides everything you need to quickly prototype your product to measure finger-based heart rate and blood oxygen saturation level (SpO_2).

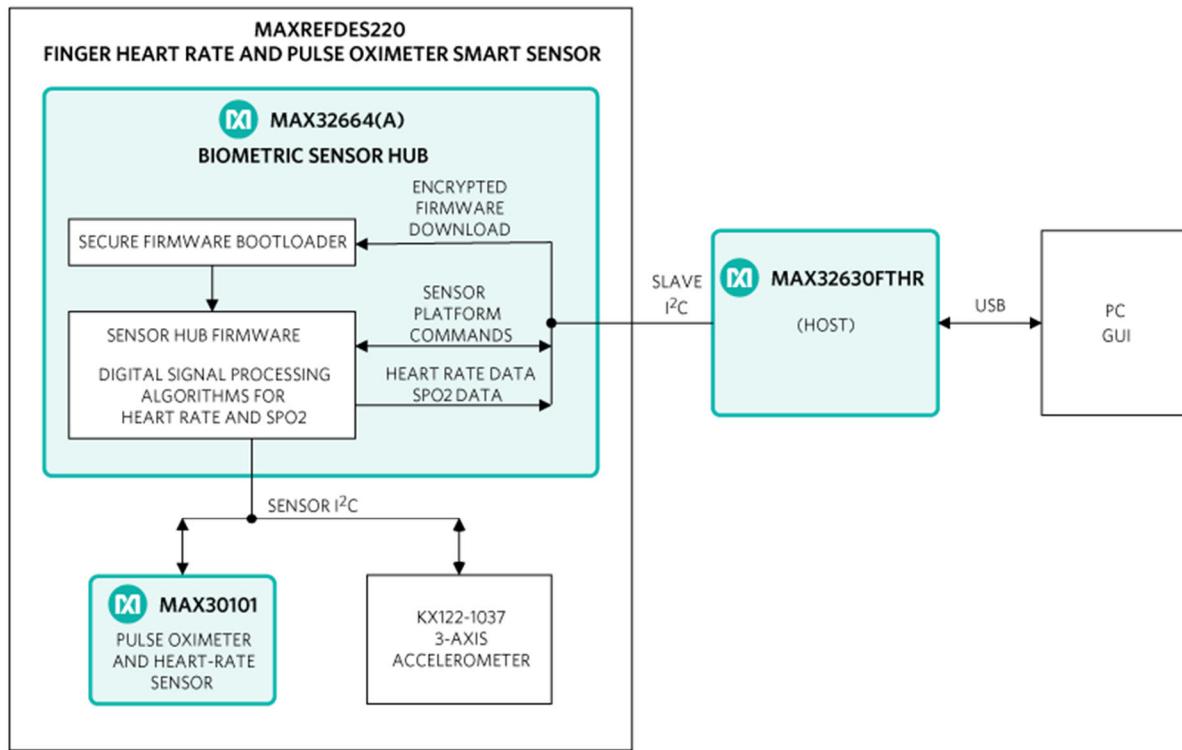


Figure 1. MAX32664 block diagram.

The MAXREFDES220# solution, which includes the MAX30101 and the MAX32664, provides an integrated hardware and software solution for finger-based applications. The MAX32664 is used as a sensor hub to collect data from the MAX30101 analog front end (AFE). The reference design also includes a tri-axis accelerometer (KX-122) to compensate for motion artifacts. (Accelerometer support in the MAXREFDES220# is optional.)

The MAX32630FTHR is used as a sample host is included in MAXREFDES220# reference design.

MAXREFDES101#

The MAXREFDES101# is a unique evaluation and development platform in a wrist-worn wearable form factor that demonstrates the functions of a wide range of Maxim's products for health-sensing applications.

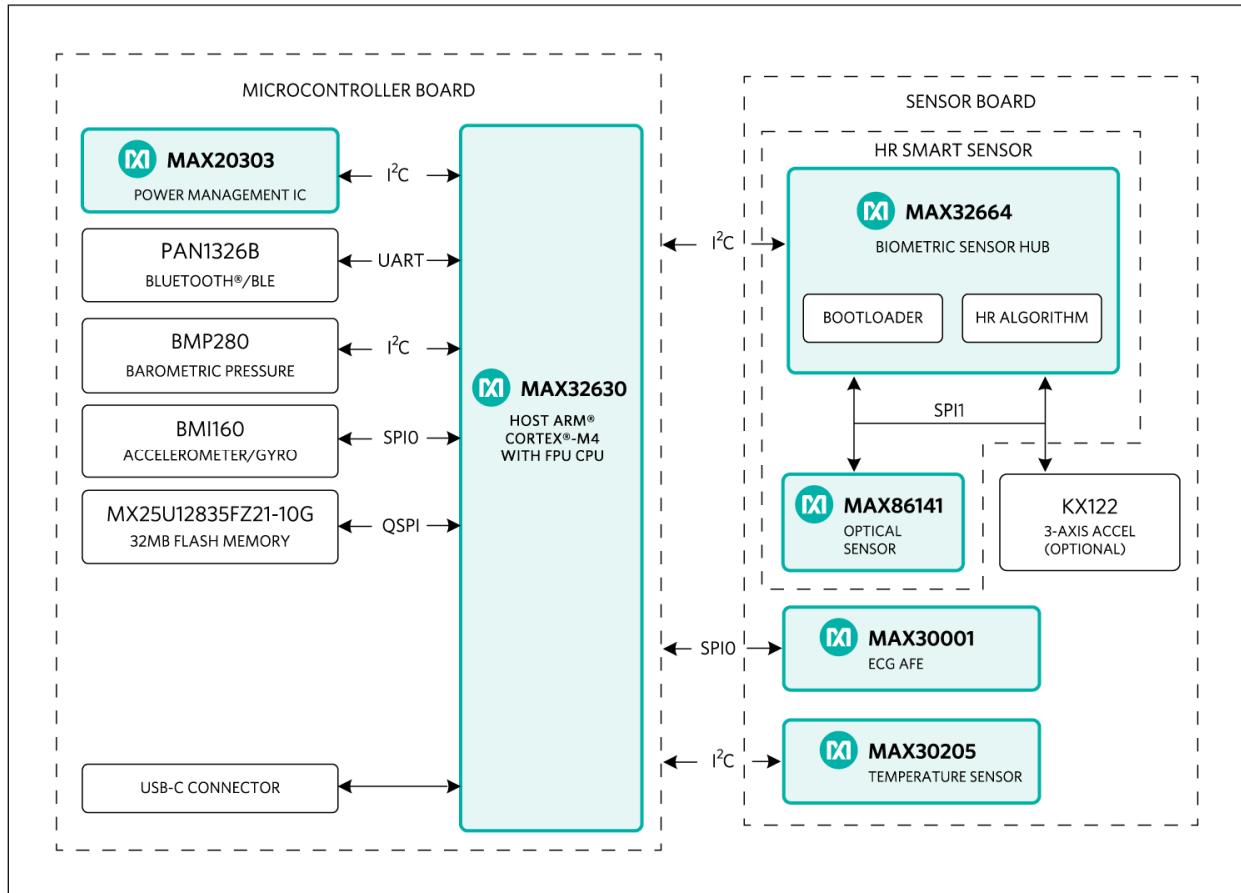


Figure 2. MAXREFDES101# block diagram.

This second-generation health sensor platform (a follow-on to the MAXREFDES100#) integrates a PPG AFE sensor (MAX86141), a biopotential AFE (MAX30001), a human body temperature sensor (MAX30205), a microcontroller (MAX32630), a power-management IC (MAX20303), and a 6-axis accelerometer/gyroscope. The complete platform includes a watch enclosure and a biometric sensor hub with an embedded heart-rate algorithm (MAX32664). Algorithm output and raw data can be streamed through Bluetooth® to an Android® application or PC GUI for demonstration, evaluation, and customized development.

Android is a registered trademark of Google Inc.

The Bluetooth word mark and logos are registered trademarks owned by Bluetooth SIG, Inc. and any use of such marks by Maxim is under license.

MAX32664 GPIOs and RSTN Pin

To control and communicate with the MAX32664, the RSTN pin and GPIOs P0.1, P0.2, P0.3 of the MAX32664 are connected to the host as pictured in Figure 3.

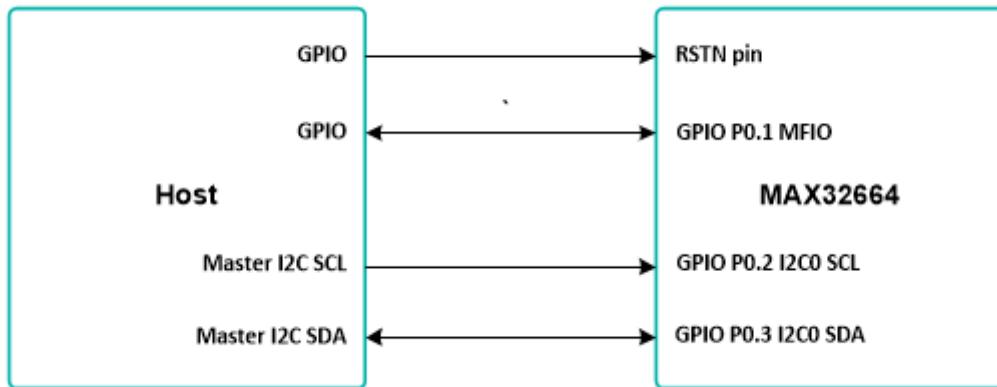


Figure 3. Pin connections between the host and the MAX32664.

The RSTN pin is used in conjunction with the GPIO P0.1 MFIO pin to control whether the MAX32664 starts up in Application mode or Bootloader mode. While in application mode, the MFIO pin can be configured to provide an interrupt signal to the host.

The host acts as an I²C master to communicate with the MAX32664. GPIO P0.2 is used as the SCL line and GPIO P0.3 is used as the SDA line.

Table 2. RSTN Pin and GPIOs Pins

MAX32664	DESCRIPTION	DIRECTION FROM THE MAX32664 SIDE
Pin RSTN	Reset_N	Input
GPIO P0.1	GPIO MFIO interrupt to host, wake from host, bootloader/application on powerup,	Input/Output
GPIO P0.2	I ² C0_Host SCL	Input
GPIO P0.3	I ² C0 Host SDA	Input/Output

Variations of the MAX32664 use additional GPIO pins in order to communicate and control sensor devices. For example, in the MAXREFDES220#, the additional GPIOs listed in Table 3 are used to control the sensors used.

Table 3. Additional GPIOs Used on the MAX32664 for the MAXREFDES220#

MAX32664	DESCRIPTION	DIRECTION FROM THE MAX32664 SIDE
GPIO P0.6	KX122 ACCEL Interrupt	Input
GPIO P0.7	MAX30101 Interrupt	Input
GPIO P0.8	MAX30101, KX122 I2C1_SCL	Output
GPIO P0.9	MAX30101, KX122 I2C1_SDA	Input/Output

MAX32664 Bootup and Application Mode

The MAX32664 is programmed to enter either bootloader mode or application mode at the start-up based on the state of the MFIO pin.

Variations of the MAX32664 part are pre-programmed with the different algorithms and application firmware. Check with your Maxim representative.

MAX32664 Bootloader Mode

The MAX32664 enters bootloader mode based on the sequencing of the RSTN pin and the MFIO pin. The necessary sequence is as follows:

- Set the RSTN pin low for 10ms.
- While RSTN is low, set the MFIO pin to low.
- After the 10ms has elapsed, set the RSTN pin high.
- After an additional 50ms has elapsed, the MAX32664 is in bootloader mode.

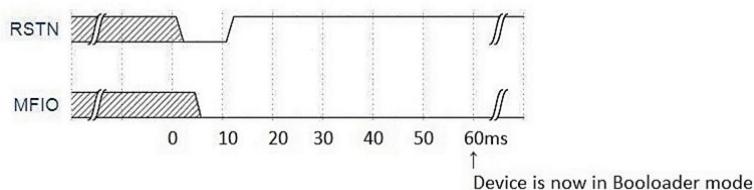


Figure 4. Entering bootloader mode using the RSTN pin and the MFIO GPIO pin.

MAX32664 Application Mode

The MAX32664 enters application mode based on the sequencing of the RSTN pin and the MFIO pin. The necessary sequence is as follows:

- Set the RSTN pin low for 10ms.
- While RSTN is low, set the MFIO pin to high.
- After the 10ms has elapsed, set the RSTN pin high.
- After an additional 50ms has elapsed, the MAX32664 is in application mode.

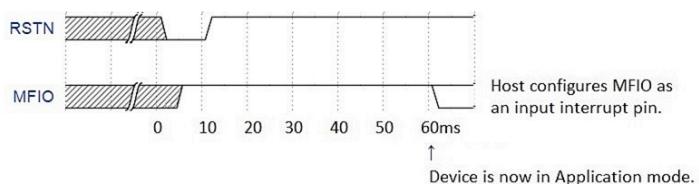


Figure 5. Entering application mode using the RSTN pin and MFIO pin.

Communications to the MAX32664 over I²C

The host communicates to the MAX32664 through the I²C bus. The MAX32664 uses 0xAA as the I²C 8-bit slave write address and 0xAB is used as the I²C 8-bit slave read address. The maximum I²C data rate supported is 3400 Kbps.

Bit Transfer Process

Both SDA and SCL signals are open-drain circuits. Each has an external pullup resistor that ensures each circuit is high when idle. The I²C specification states that during data transfer, the SDA line can change state only when SCL is low, and that SDA is stable and able to be read when SCL is high. Typical I²C write/read transactions are shown in Figure 6.

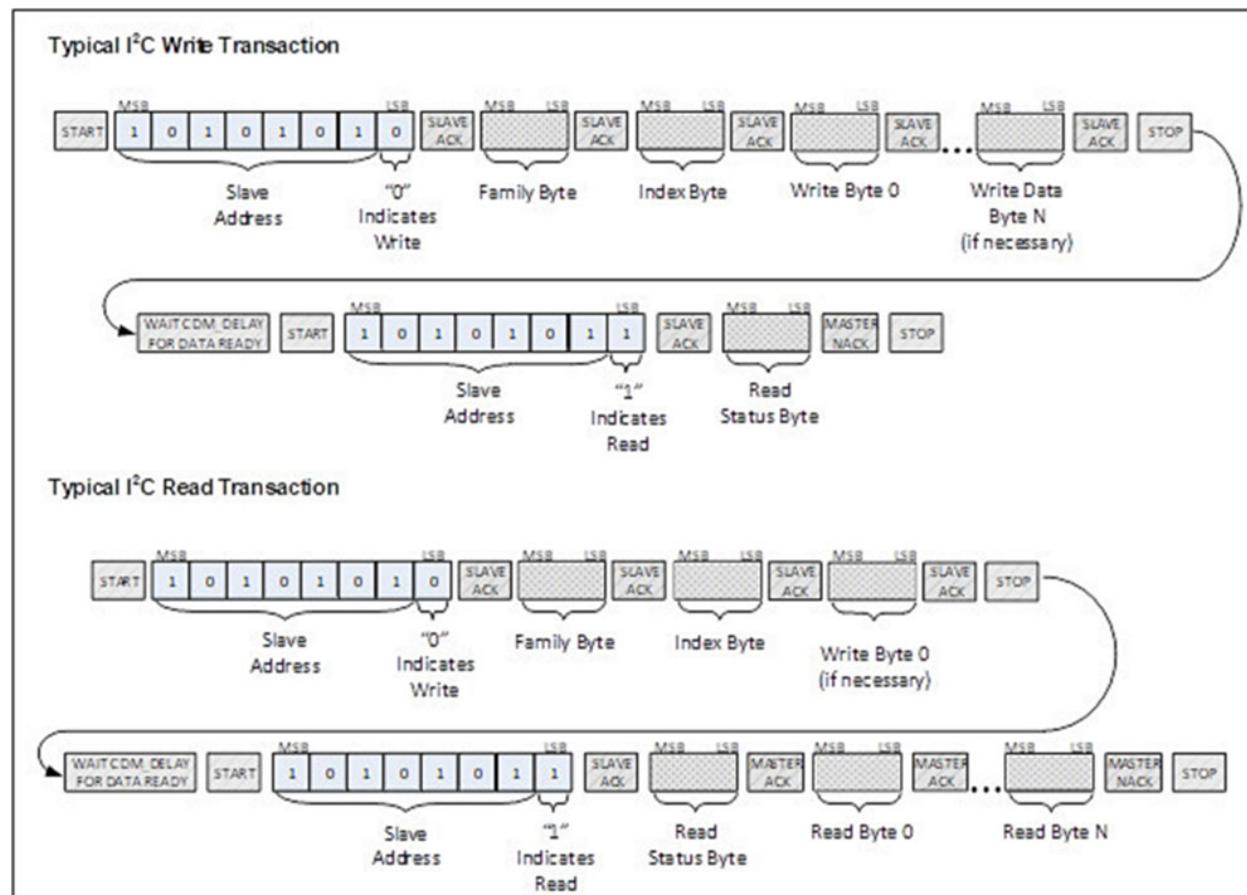


Figure 6. I²C Write/Read data transfer from host microcontroller.

The read status byte is an indicator of the success or failure of the Write Transaction. The read status byte must be accessed after each write transaction to the device. This ensures that write transaction processing is understood and any errors in the device command handling can be corrected. The value of the read status byte is summarized in Table 4.

Table 4. Read Status Byte Value

STATUS BYTE VALUE	DESCRIPTION
0x00	SUCCESS. The write transaction was successful.
0x01	ERR_UNAVAIL_CMD. Illegal Family Byte and/or Command Byte was used.
0x02	ERR_UNAVAIL_FUNC. This function is not implemented.
0x03	ERR_DATA_FORMAT. Incorrect number of bytes sent for the requested Family Byte.
0x04	ERR_INPUT_VALUE. Illegal configuration value was attempted to be set.
0x05	ERR_TRY AGAIN. Device is busy. Try again.
0x80	ERR_BTLDL_GENERAL. General error while receiving/flashing a page during the bootloader sequence.
0x81	ERR_BTLDL_CHECKSUM. Checksum error while decrypting/checking page data.
0x82	ERR_BTLDL_AUTH. Authorization error.
0x83	ERR_BTLDL_INVALID_APP. Application not valid.
0xFF	ERR_UNKNOWN. Unknown Error.

I²C Write

The process for an I²C write data transfer is as follows:

1. The bus master indicates a data transfer to the device with a START condition.
2. The master transmits one byte with the 7-bit slave address (most significant 7 bits of the 8-bit address) and a single write bit set to zero. The eight bits to be transferred as a slave address for the MAX32664 is 0xAA for a write transaction.
3. During the next SCL clock following the write bit, the master releases SDA. During this clock period, the device responds with an ACK by pulling SDA low.
4. The master senses the ACK condition and begins to transfer the Family Byte. The master drives data on the SDA circuit for each of the eight bits of the Family byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
5. The master senses the ACK condition and begins to transfer the Index Byte. The master drives data on the SDA circuit for each of the eight bits of the Index byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
6. The master senses the ACK condition and begins to transfer the Write Data Byte 0. The master drives data on the SDA circuit for each of the eight bits of the Write Data Byte 0, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
7. The master senses the ACK condition and can begin to transfer another Write Data Byte if required. The master drives data on the SDA circuit for each of the eight bits of the Write Data Byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication. If another Write Data Byte is not required, the master indicates the transfer is complete by generating a STOP condition. A STOP condition is generated when the master pulls SDA from a low to high while SCL is high.
8. The master waits for a period of CMD_DELAY (60μs) for the device to have its data ready.
9. The master indicates a data transfer to a slave with a START condition.
10. The master transmits one byte with the 7-bit slave address and a single write bit set to one. This is an indication from the master of its intent to read the device from the previously written location defined by the Family Byte and the Index Byte. The master then floats SDA and allows the device to drive SDA to send the Status Byte. The Status Byte reveals the success of the previous write sequence. After the Status Byte is read, the master drives SDA low to signal the end of data to the device.
11. The master indicates the transfer is complete by generating a STOP condition.
12. After the completion of the write data transfer, the Status Byte must be analyzed to determine if the write sequence was successful and the device has received the intended command.

I²C Read

The process for an I²C read data transfer is as follows:

1. The bus master indicates a data transfer to the device with a START condition.
2. The master transmits one byte with the 7-bit slave address and a single write bit set to zero. The eight bits to be transferred as a slave address for the MAX32664 is 0xAA for a write transaction. This write transaction precedes the actual read transaction to indicate to the device what section is to be read.
3. During the next SCL clock following the write bit, the master releases SDA. During this clock period, the device responds with an ACK by pulling SDA low.
4. The master senses the ACK condition and begins to transfer the Family Byte. The master drives data on the SDA circuit for each of the eight bits of the Family byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
5. The master senses the ACK condition and begins to transfer the Index Byte. The master drives data on the SDA circuit for each of the eight bits of the Index byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
6. The master senses the ACK condition and begins to transfer the Write Data Byte if necessary for the read instruction. The master drives data on the SDA circuit for each of the eight bits of the Write Data byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
7. The master indicates the transfer is complete by generating a STOP condition.
8. The master waits for a period of CMD_DELAY (60μs) for the device to have its data ready.
9. The master indicates a data transfer to a slave with a START condition.
10. The master transmits one byte with the 7-bit slave address and a single write bit set to one. This is an indication from the master of its intent to read the device from the previously written location defined by the Family Byte and the Index Byte. The master then floats SDA and allows the device to drive SDA to send the Status Byte. The Status Byte reveals the success of the previous write sequence. After the Status Byte is read, the master drives SDA low to acknowledge the byte.
11. The master floats SDA and allows the device to drive SDA to send Read Data Byte 0. After Read Data Byte 0 is read, the master drives SDA low to acknowledge the byte.
12. The master floats SDA and allows the device to drive SDA to send the Read Data Byte N. After Read Data Byte N is read, the master drives SDA low to acknowledge the Read Data Byte N. This process continues until the device has provided all the data that the master expects based upon the Family Byte and Index Byte definition.
13. The master indicates the transfer is complete by generating a STOP condition.

MAX32664 I2C Message Protocol Definition

Table 5 defines the I2C message protocol for the MAX32664.

Table 5. MAX32664 I2C Message Protocol Definitions

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Read Sensor Hub Status	Read sensor hub status	0x00	0x00	-	Err0[0]: 0 = No Error; 1 = Sensor Communication Problem Err1[0]: Not used Err2[0]: Not used DataRdyInt[3]: 0 = FIFO below threshold; 1 = FIFO filled to threshold or above. FifoOutOvrInt[4]: 0 = No FIFO overflow; 1 = Sensor Hub Output FIFO overflowed, data lost. FifoInOvrInt[5]: 0 = No FIFO overflow; 1 = Sensor Hub Input FIFO overflowed, data lost. DevBusy[6]: 0 = Sensor Hub ready; 1 = Sensor Hub is busy processing. See Table 6 for the for the bit field table.
Device Mode	Select the device operating mode. The application must implement this.	0x01	0x00	0x00: Exit bootloader mode. 0x02: Reset. 0x08: Enter bootloader mode.	-
Device Mode	Read the device operating mode.	0x02	0x00	-	0x00: Application operating mode. 0x08: Bootloader operating mode.
Set Output Mode	Set the output format of the sensor hub.	0x10	0x00	0x00: Pause (no data) 0x01: Sensor Data 0x02: Algorithm Data 0x03: Sensor Data and Algorithm Data 0x04: Pause (no data) 0x05: Sample Counter byte, Sensor Data 0x06: Sample Counter byte, Algorithm Data 0x07: Sample Counter byte, Sensor Data and Algorithm Data	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Set Output Mode	Set the threshold for the FIFO interrupt bit/pin. The MFIO pin is used as the interrupt and the host should configure this pin as an input interrupt pin. The status bit DataRdyInt is set when this threshold is reached.	0x10	0x01	0x01 to 0xFF: Sensor Hub Interrupt Threshold for FIFO almost full.	-
Read Output FIFO	Get the number of samples available in the FIFO.	0x12	0x00	-	Number of samples available in the FIFO.
Read Output FIFO	Read data stored in output FIFO.	0x12	0x01	-	See Table 7, Output FIFO Format Definitions. The internal FIFO read pointer increments once the sample size bytes have been read.
Read Input FIFO for External Sensors (e.g., systems that have an externally supplied accelerometer)	Read the sensor sample size.	0x13	0x00	0x04: Accelerometer	0x06: Bytes per sample for the external accelerometer. Three 16-bit 2's complement with LSB = 0.001g. See Table 8 for an example.
Read Input FIFO for External Sensors	Read the input FIFO size for the maximum number of samples that the input FIFO can hold (16-bit).	0x13	0x01	-	MSB, LSB
Read Input FIFO for External Sensors	Read the sensor FIFO size for the maximum number of samples that the sensor FIFO can hold (16-bit).	0x13	0x02	0x04: Accelerometer	MSB, LSB
Read Input FIFO for External Sensors	Read the number of samples currently in the input FIFO (16-bit).	0x13	0x03	0x04: Accelerometer	MSB, LSB
Read Input FIFO for External Sensors	Read the number of samples currently in the sensor FIFO (16-bit).	0x13	0x04	-	MSB, LSB
Write Input FIFO for External Sensors	Write data to the input FIFO.	0x14	0x04	Number of Samples, Sample 1 values, ..., Sample n values See Table 8 for an example.	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Write Register	Write a value to a writable MAX86140/MAX86141 register.	0x40	0x00	Register address, Register value	-
Write Register	Write a value to a writable MAX30205 register.	0x40	0x01	Register address, Register value	-
Write Register	Write a value to a writable MAX30001 register.	0x40	0x02	Register address, Register value	-
Write Register	Write a value to a writable MAX30101 register.	0x40	0x03	Register address, Register value	-
Write Register	Write a value to a writable accelerometer sensor register.	0x40	0x04	Register address, Register value	-
Read Register	Read the value of a MAX86140/MAX86141 register.	0x41	0x00	Register Address	Register value
Read Register	Read the value of a MAX30205 register.	0x41	0x01	Register Address	Register value
Read Register	Read the value of a MAX30001 register.	0x41	0x02	Register Address	Register value
Read Register	Read the value of a MAX30101 register.	0x41	0x03	Register Address	Register value
Read Register	Read the value of an accelerometer sensor register.	0x41	0x04	Register Address	Register value
Get Attributes of the AFE	Retrieve the attributes of the MAX86140/MAX86141 AFE.	0x42	0x00	-	Number of bytes in a word for this sensor, Number of registers available for this sensor.
Get Attributes of the AFE	Retrieve the attributes of the MAX30205 AFE.	0x42	0x01	-	Number of bytes in a word for this sensor, Number of registers available for this sensor.
Get Attributes of the AFE	Retrieve the attributes of the MAX30001 AFE.	0x42	0x02	-	Number of bytes in a word for this sensor, Number of registers available for this sensor.
Get Attributes of the AFE	Retrieve the attributes of the MAX30101 AFE.	0x42	0x03	-	Number of bytes in a word for this sensor, Number of registers available for this sensor.
Get Attributes of the AFE	Retrieve the attributes of the accelerometer sensor AFE.	0x42	0x04	-	Number of bytes in a word for this sensor, Number of registers available for this sensor.
Dump Registers	Read all the MAX86140/MAX86141 registers.	0x43	0x00	-	Register address 0, register value 0, register address 1, register value 1, ..., register address n, register value n
Dump Registers	Read all the MAX30205 registers.	0x43	0x01	-	Register address 0, register value 0, register address 1, register value 1, ..., register address n, register value n

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Dump Registers	Read all the MAX30001 registers.	0x43	0x02	-	Register address 0, register value 0, register address 1, register value 1, ..., register address n, register value n
Dump Registers	Read all the MAX30101 registers.	0x43	0x03	-	Register address 0, register value 0, register address 1, register value 1, ..., register address n, register value n
Dump Registers	Read all the accelerometer sensor registers.	0x43	0x04	-	Register address 0, register value 0, register address 1, register value 1, ..., register address n, register value n
Sensor Mode Enable	Enable the MAX86140/MAX86141 sensor.	0x44	0x00	0x00: Disable 0x01: Enable	-
Sensor Mode Enable	Enable the MAX30205 sensor.	0x44	0x01	0x00: Disable 0x01: Enable	-
Sensor Mode Enable	Enable the MAX30001 sensor.	0x44	0x02	0x00: Disable 0x01: Enable	-
Sensor Mode Enable	Enable the MAX30101 sensor.	0x44	0x03	0x00: Disable 0x01: Enable	-
Sensor Mode Enable	Enable the external accelerometer sensor. This command is used when host accelerometer data is supplied to the sensor hub.	0x44	0x04	0x00: Disable 0x01: Enable	-
Algorithm Configuration	Automatic Gain Control (AGC) algorithm: Set the target percentage of the full-scale ADC range that the automatic gain control (AGC) algorithm uses.	0x50	0x00	0x00, 0 to 100 percent	-
Algorithm Configuration	AGC algorithm: Set the step size toward the target for the AGC algorithm.	0x50	0x00	0x01, 0 to 100 percent	-
Algorithm Configuration	AGC algorithm: Set the sensitivity for the AGC algorithm.	0x50	0x00	0x02, 0 to 100 percent	-
Algorithm Configuration	AGC algorithm: Set the number of samples to average for the AGC algorithm.	0x50	0x00	0x03, Number of samples to average (range is 0 to 255).	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	Wrist Heart Rate Monitor (WHRM) algorithm: Set the sample rate for the WHRM algorithm. The WHRM algorithm is configured to use the LED1 and Photodiode 1 (25Hz sample rate) and is compatible with the MAX86141 AFE, KX-122 accelerometer.	0x50	0x02	0x00, MSB of sample rate, LSB of sample rate (16-bit unsigned integer)	-
Algorithm Configuration	WHRM algorithm: Set the maximum allowed height (cm).	0x50	0x02	0x01, MSB of height, LSB of height (16-bit unsigned integer)	-
Algorithm Configuration	WHRM algorithm: Set the maximum allowed weight (kg).	0x50	0x02	0x02, MSB of weight, LSB of weight (16-bit unsigned integer)	-
Algorithm Configuration	WHRM algorithm: Set the maximum allowed age (years).	0x50	0x02	0x03, Age	-
Algorithm Configuration	WHRM algorithm: Set the minimum allowed height (cm).	0x50	0x02	0x04, MSB of height, LSB of height (16-bit unsigned integer)	-
Algorithm Configuration	WHRM algorithm: Set the minimum allowed weight (kg).	0x50	0x02	0x05, MSB of weight, LSB of weight (16-bit unsigned integer)	-
Algorithm Configuration	WHRM algorithm: Set the minimum allowed age (years).	0x50	0x02	0x06, Age	-
Algorithm Configuration	WHRM algorithm: Set the default height (cm).	0x50	0x02	0x07, MSB of height, LSB of height (16-bit unsigned integer)	-
Algorithm Configuration	WHRM algorithm: Set the default weight (kg).	0x50	0x02	0x08, MSB of weight, LSB of weight (16-bit unsigned integer)	-
Algorithm Configuration	WHRM algorithm: Set the default age (years).	0x50	0x02	0x09, Age	-
Algorithm Configuration	WHRM algorithm: Set the initial heart-rate value in bpm which might speed up the algorithm.	0x50	0x02	0x0A, Heart rate (bpm)	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	MaximFast algorithm: Set the MaximFast SpO ₂ coefficients.	0x50	0x02	0x0B, four bytes signed integer A, four bytes signed integer B, four bytes signed integer C (32-bit integers which are the coefficients times 100,000) The MAXREFDES220# without the cover glass uses the following coefficients as the default values: A = 159584 B = -3465966 C = 11268987	-
Algorithm Configuration	WHRM algorithm: Enable automatic exposure control (AEC) algorithm.	0x50	0x02	0x0B, 0x00: Disable 0x0B, 0x01: Enable	-
Algorithm Configuration	WHRM algorithm: Enable skin contact detection (SCD) algorithm.	0x50	0x02	0x0C, 0x00: Disable 0x0C, 0x01: Enable	-
Algorithm Configuration	WHRM algorithm: Set adjusted target photo detector (PD) current period in seconds.	0x50	0x02	0x0D, MSB, LSB (unsigned 16-bit integer)	-
Algorithm Configuration	WHRM algorithm: Set SCD debounce window.	0x50	0x02	0x0E, MSB, LSB (unsigned 16-bit integer)	-
Algorithm Configuration	WHRM algorithm: Set motion magnitude threshold in 0.1g.	0x50	0x02	0x0F, MSB, LSB (unsigned 16-bit integer)	-
Algorithm Configuration	WHRM algorithm: Set the minimum PD current in 0.1mA.	0x50	0x02	0x10, MSB, LSB (unsigned 16-bit integer)	-
Algorithm Configuration	WHRM algorithm: Configure the source of the PPG signal for the PD.	0x50	0x02	0x11, 0x01: PD1 0x11, 0x02: PD2 0x11, 0x03: PD1 and PD2	-
Algorithm Configuration	Blood Pressure Trending (BPT) algorithm: Set if the user is on blood pressure medication.	0x50	0x04	0x00, 0x00: Not using blood pressure (BP) medication 0x00, 0x01: Using BP medication	-
Algorithm Configuration	BPT algorithm: Write the three samples of the diastolic BP byte values needed by the calibration procedure.	0x50	0x04	0x01, diastolic value 1, diastolic value 2, diastolic value 3	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	BPT algorithm: Write the three samples of the systolic BP byte values needed by the calibration procedure.	0x50	0x04	0x02, systolic value 1, systolic value 2, systolic value 3	-
Algorithm Configuration	BPT algorithm: Calibrate the BPT algorithm using the result of the calibration procedure. (Use the data from the 0x51 0x04 0x03 command).	0x50	0x04	0x03, 608 bytes of calibration data	-
Algorithm Configuration	BPT algorithm: Set the estimation date, using the current date. The year is sent as a 16-bit unsigned integer, and the month and day are bytes.	0x50	0x04	0x04, year MSB, year LSB, month, day	-
Algorithm Configuration	BPT algorithm: Configure whether the user is not resting or resting.	0x50	0x04	0x05, 0x00: Resting 0x05, 0x01: Not resting	-
Algorithm Configuration	BPT algorithm: Set the SpO ₂ coefficients A, B, C. Defaults: C = 0x42E16137 B = 0xC20AA37F A = 3FCC448F	0x50	0x04	0x06, four bytes float C, four bytes float B, four bytes float A	-
Algorithm Configuration	SpO ₂ on the wrist (WSpO ₂) algorithm: Set the WSpO ₂ coefficients. WSpO ₂ is a Maxim supplied algorithm for measuring SpO ₂ on the wrist. Defaults: A = 159584 B = -3465966 C = 11268987	0x50	0x05	0x00, four bytes signed integer A, four bytes signed integer B, four bytes signed integer C (32-bit integers which are the coefficients times 100,000).	-
Algorithm Configuration	WSpO ₂ algorithm: Set the sample rate.	0x50	0x05	0x01, 0x00: 100Hz 0x01, 0x01: 25Hz	-
Algorithm Configuration	WSpO ₂ algorithm: Set the algorithm run mode.	0x50	0x05	0x02, 0x00: Continuous 0x02, 0x01: One-shot	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	WSpO ₂ algorithm: Set the AGC mode for the WSpO ₂ algorithm.	0x50	0x05	0x03, 0x00: Disable 0x03, 0x01: Enable	-
Algorithm Configuration	WSpO ₂ algorithm: Set motion detection.	0x50	0x05	0x04, 0x00: Disable 0x04, 0x01: Enable	-
Algorithm Configuration	WSpO ₂ algorithm: Set the motion detection period.	0x50	0x05	0x05, MSB of Period, LSB of Period (16-bit unsigned integer)	-
Algorithm Configuration	WSpO ₂ algorithm: Set the motion threshold for the WSpO ₂ algorithm.	0x50	0x05	0x06, 4 bytes (a 32-bit signed integer, which is the motion threshold value times 100,000)	-
Algorithm Configuration	WSpO ₂ algorithm: Set WSpO ₂ AGC Timeout (sec)	0x50	0x05	0x07, 8-bit unsigned value	-
Algorithm Configuration	WSpO ₂ algorithm: Set WSpO ₂ Algorithm Timeout (sec)	0x50	0x05	0x08, 8-bit unsigned value	-
Algorithm Configuration	WSpO ₂ algorithm: Set WSpO ₂ PD configuration (source of PPG signal)	0x50	0x05	0x09, 0x01: PD1 0x09, 0x02: PD2	-
Algorithm Configuration Read	Automatic Gain Control (AGC) algorithm: Read the target percentage of the full-scale ADC range that the AGC algorithm is using.	0x51	0x00	0x00	0 to 100 Percent
Algorithm Configuration Read	AGC algorithm: Read step size toward the target.	0x51	0x00	0x01	0 to 100 Percent
Algorithm Configuration Read	AGC algorithm: Read the sensitivity for the AGC algorithm.	0x51	0x00	0x02	0 to 100 Percent
Algorithm Configuration Read	AGC algorithm: Read the number of samples to average for the AGC algorithm.	0x51	0x00	0x03	Number of samples to average (range is 0 to 255)
Algorithm Configuration Read	Wrist Heart Rate Monitor (WHRM) algorithm: Read the sample rate.	0x51	0x02	0x00	MSB of sample rate, LSB of sample rate (16-bit unsigned integer)
Algorithm Configuration Read	WHRM algorithm: Read the maximum allowed height (cm).	0x51	0x02	0x01	MSB of height, LSB of height (16-bit unsigned integer)
Algorithm Configuration Read	WHRM algorithm: Read the maximum allowed weight (kg).	0x51	0x02	0x02	MSB of weight, LSB of weight (16-bit unsigned integer)

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration Read	WHRM algorithm: Read the maximum allowed age (years).	0x51	0x02	0x03	Age
Algorithm Configuration Read	WHRM algorithm: Read the minimum allowed height (cm).	0x51	0x02	0x04	MSB of height, LSB of height (16-bit unsigned integer)
Algorithm Configuration Read	WHRM algorithm: Read the minimum allowed weight (kg).	0x51	0x02	0x05	MSB of weight, LSB of weight (16-bit unsigned integer)
Algorithm Configuration Read	WHRM algorithm: Read the minimum allowed age (years).	0x51	0x02	0x06	Age
Algorithm Configuration Read	WHRM algorithm: Read the default height (cm).	0x51	0x02	0x07	MSB of height, LSB of height (16-bit unsigned integer)
Algorithm Configuration Read	WHRM algorithm: Read the default weight (kg).	0x51	0x02	0x08	MSB of weight, LSB of weight (16-bit unsigned integer)
Algorithm Configuration Read	WHRM algorithm: Read the default age (years).	0x51	0x02	0x09	Age
Algorithm Configuration Read	WHRM algorithm: Read the initial heart rate value in bpm which can speed up the algorithm.	0x51	0x02	0x0A	Heart rate (bpm)
Algorithm Configuration Read	MaximFast algorithm: Read the coefficients for the MaximFast SpO ₂ algorithm.	0x51	0x02	0x0B	Four bytes signed integer A, Four bytes signed integer B, Four bytes signed integer C (32-bit integers which are the coefficients times 100,000).
Algorithm Configuration Read	WHRM algorithm: Read the AEC algorithm enable status.	0x51	0x02	0x0B	0x00: Disabled 0x01: Enabled
Algorithm Configuration Read	WHRM algorithm: Read the SCD algorithm enable status.	0x51	0x02	0x0C	0x00: Disabled 0x01: Enabled
Algorithm Configuration Read	WHRM algorithm: Read the adjusted target PD current period in seconds.	0x51	0x02	0x0D	MSB, LSB (16-bit unsigned integer)
Algorithm Configuration Read	WHRM algorithm: Read the SCD debounce window.	0x51	0x02	0x0E	MSB, LSB (16-bit unsigned integer)
Algorithm Configuration Read	WHRM algorithm: Read the motion magnitude threshold in 0.1g.	0x51	0x02	0x0F	MSB, LSB (16-bit unsigned integer)
Algorithm Configuration Read	WHRM algorithm: Read the minimum PD current in 0.1mA.	0x51	0x02	0x10	MSB, LSB (16-bit unsigned integer)

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration Read	WHRM algorithm: Read the PD configuration of the PPG signal source.	0x51	0x02	0x11	0x01: PD1 0x02: PD2 0x03: PD1 and PD2
Algorithm Configuration Read	BPT algorithm: Read the calibration data results from the calibration procedure.	0x51	0x04	0x03	608 bytes of calibration data
Algorithm Configuration Read	WSPO ₂ algorithm: Read coefficients A, B, C.	0x51	0x05	0x00	Four bytes signed integer A, Four bytes signed integer B, Four bytes signed integer C
Algorithm Configuration Read	WSPO ₂ algorithm: Read the sample rate.	0x51	0x05	0x01	0x00: 100 Hz 0x01: 25 Hz
Algorithm Configuration Read	WSPO ₂ algorithm: Read the algorithm run mode.	0x51	0x05	0x02	0x00: Continuous 0x01: One-shot
Algorithm Configuration Read	WSPO ₂ algorithm: Read the AGC mode status.	0x51	0x05	0x03	0x00: Disabled 0x01: Enabled
Algorithm Configuration Read	WSPO ₂ algorithm: Read the motion detection status.	0x51	0x05	0x04	0x00: Disabled 0x01: Enabled
Algorithm Configuration Read	WSPO ₂ algorithm: Read the motion detection period.	0x51	0x05	0x05	MSB of Period, LSB of Period (16-bit unsigned integer)
Algorithm Configuration	WSPO ₂ algorithm: Read the motion threshold for the WSPO ₂ algorithm.	0x51	0x05	0x06	4 bytes (32-bit signed integers which are the motion threshold times 100,000)
Algorithm Configuration Read	WSPO ₂ algorithm: Read the AGC timeout (sec).	0x51	0x05	0x07	AGC timeout (8-bit unsigned)
Algorithm Configuration Read	WSPO ₂ algorithm: Read Algorithm Timeout (sec)	0x51	0x05	0x08	WSPO ₂ algorithm timeout (8-bit unsigned)
Algorithm Configuration Read	WSPO ₂ algorithm: Read the PD configuration (source of PPG signal).	0x51	0x05	0x09	0x01: PD1 0x02: PD2
Algorithm Mode Enable	AGC: Enable the AGC algorithm.	0x52	0x00	0x00: Disable 0x01: Enable	-
Algorithm Mode Enable	AEC: Enable the AEC algorithm.	0x52	0x01	0x00: Disable 0x01: Enable	-
Algorithm Mode Enable	WHRM, MaximFast: Enable the WHRM, MaximFast algorithm.	0x52	0x02	0x00: Disable 0x01: Enable	-
Algorithm Mode Enable	Electrocardiogram (ECG): Enable the ECG algorithm.	0x52	0x03	0x00: Disable 0x01: Enable	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Mode Enable	Blood Pressure Trending (BPT): Enable the BPT algorithm.	0x52	0x04	0x00: Disable 0x01: Enable	-
Algorithm Mode Enable	WSpO ₂ : Enable the algorithm.	0x52	0x05	0x00: Disable 0x01: Enable	-
Bootloader Flash	Set the initialization vector (IV) bytes. This is not required for a non-secure bootloader.	0x80	0x00	Use bytes 0x28 to 0x32 from the .msbl file as the IV bytes	-
Bootloader Flash	Set the authentication bytes. This is not required for a non-secure bootloader.	0x80	0x01	Use bytes 0x34 to 0x43 from the .msbl file.	-
Bootloader Flash	Set the number of pages.	0x80	0x02	0x00, Number of pages located at byte 0x44 from the .msbl file.	-
Bootloader Flash	Erase the application flash memory.	0x80	0x03		-
Bootloader Flash	Send the page values.	0x80	0x04	The first page is specified by byte 0x4C from the .msbl file. The total bytes for each message protocol are the page size + 16 bytes of CRC.	-
Bootloader Information	Get bootloader version.	0x81	0x00	-	Major version byte, Minor version byte, Revision byte
Bootloader Information	Get the page size in bytes.	0x81	0x01	-	Upper byte of page size, Lower byte of page size
Identity	Read the MCU type.	0xFF	0x00	-	0x00: MAX32625 0x01: MAX32660/MAX32664
Identity	Read the sensor hub version.	0xFF	0x03	-	Major version byte, Minor version byte, Revision byte
Identity	Read the algorithm: version.	0xFF	0x07	-	Major version byte, Minor version byte, Revision byte

Table 6 defines the bit fields of the sensor hub status byte.

Table 6. Sensor Hub Status Byte

BIT	7	6	5	4	3	2	1	0
Field	Reserved	DevBusy	FifoInOverInt	FifoOutOvrlnt	DataRdyInt	Err2	Err1	Err0

Table 7 defines the output FIFO format for the Read Output FIFO I²C message.

Table 7. Output FIFO Format Definitions

SENSOR OR ALGORITHM	SIZE (BYTES)	DESCRIPTION	
MAX86140/MAX86141	3	LED1 value: 24-bit.	
	3	LED2 value: 24-bit.	
	3	LED3 value: 24-bit.	
	3	LED4 value: 24-bit.	
	3	LED5 value: 24-bit.	
	3	LED5 value: 24-bit.	
MAX30205	2	Temperature value: 16 bits, LSB = 0.00390625°C.	
MAX30001	3	ECG: 24-bit.	
	3	R-to-R: 24-bit.	
	3	BIOZ: 24-bit.	
	3	PACE: 24-bit.	
	3	LED1 value: 24-bit.	
MAX30101	3	LED2 value: 24-bit.	
	3	LED3 value: 24-bit.	
	3	LED4 value: 24-bit.	
	3	LED4 value: 24-bit.	
Accelerometer	2	X: 16-bit two's compliment. LSB = 0.001g	
	2	Y: 16-bit two's compliment. LSB = 0.001g	
	2	Z: 16-bit two's compliment. LSB = 0.001g	
AGC Algorithm	0	No output from algorithm.	
AEC Algorithm	0	No output from algorithm.	
WHRM/MaximFast	2	Heart Rate (bpm): 16-bit, LSB = 0.1 bpm	
	1	Confidence level (0 - 100%): 8-bit, LSB = 1%	
	2	SpO ₂ value (0 - 100%): 16-bit, LSB = 0.1%	
	1	WHRM State Machine Status Codes (signed 8-bit): 0: Success +1: Not ready -1: Object detected -2: Excessive sensor device motion -3: No object detected -4: Pressing too hard -5: Object other than finger detected -6: Excessive finger motion	MaximFast State Machine Status Codes: 0: No object detected. 1: Object detected. 2: Object other than finger detected 3: Finger detected.
	1	Calibration/estimation mode status.	
BPT	1	Calibration mode progress. Value ranges from 0 to 100. 100 meaning calibration is complete.	
	2	Heart rate (bpm). LSB = 0.1 bpm.	
	1	Systolic BP value. Only valid once progress = 100.	
	1	Diastolic BP value. Only valid once progress = 100.	

SENSOR OR ALGORITHM	SIZE (BYTES)	DESCRIPTION
WSPO ₂	2	r: 10x SpO ₂ r value.
	1	SpO ₂ confidence in %.
	2	10x the measured value of SpO ₂ .
	1	SpO ₂ calculation progress (%) - only in one-shot mode.
	1	lowSNRflag: A flag showing low SNR .
	1	Motion flag: A flag showing high motion.
	1	IsWSpo2Calculated: Algorithm reported status.

Table 8 provides the sequence of commands for writing external accelerometer data to the input FIFO.

Table 8. Sequence of Commands to Write External Accelerometer Data to the Input FIFO

HOST COMMAND	COMMAND DESCRIPTION	MAX32664 RESPONSE	RESPONSE DESCRIPTION
0xAA 0x44 0x04 0x1	Enable the input FIFO for host supplied accelerometer data.	0xAB 0x00	No error.
0xAA 0x13 0x00 0x04	Read the sensor sample size for the accelerometer. (optional)	0xAB 0x00 0xN	No error. N is the number of samples in the FIFO.
0xAA 0x14 0x04 Number of Samples, Sample 1 values, ..., Sample N values	Write data to the input FIFO.	0xAB 0x00	No error.
0xAA 0x00 0x00	Read the sensor hub status to verify that the FifoOutOvrlnt bit 4 is not set in the Sensor Hub Status byte. (optional)	0xAB 0x00 0x88	No error. Sensor hub is not busy.

MAX32664 I2C Annotated Application Mode Example

Table 9 shows a capture of the I2C traffic between the example host microcontroller (MAX32630FTHR) and the MAX32664GWEA for commanding the MAX32664GWEA to stream raw and algorithm data. The MAXREFDES220# is used for this example.

Table 9. MAX32664GWEA I2C Annotated Application Mode Example

HOST COMMAND	COMMAND DESCRIPTION	MAX32664 RESPONSE	RESPONSE DESCRIPTION
0xAA 0x02 0x00†	Read device mode.	0xAB 0x00 0x00	No error. Mode is application operating mode.
0xAA 0xFF 0x03	Read the sensor hub version.	0xAB 0x00 0x01 0x09 0x00	No error. Version is 1.9.0
0xAA 0x42 0x03†	Get the MAX30101 register attributes.	0xAB 0x00 0x01 0x24	No error. Attributes are 1 byte, 0x24 registers available.
0xAA 0x43 0x03	Read all the MAX30101 registers.	0xAB 0x00 0x00 0x00 0x00 0x01 0x00 0x02 0x00 ... 0xFF 0x15	No error. Reg 0x00 = 0, Reg 0x01 = 0, Reg 0x02 = 0, ..., Reg 0xFF = 0x15
0xAA 0x41 0x03 0x07†	Read the MAX30101 register 7.	0xAB 0x00 0x60	No error. Register 7 is 0x60.
0xAA 0x10 0x00 0x03*	Set output mode to raw and algorithm data.	0xAB 0x00	No error.
0xAA 0x10 0x01 0x0F*	Set FIFO threshold as almost full at 0x0F. Increase or decrease this value if you want more or fewer samples per interrupt.	0xAB 0x00	No error.
0xAA 0x44 0x03 0x01*	Enable the MAX30101 sensor.	0xAB 0x00	No error.
0xAA 0x44 0x04 0x01†	Enable the accelerometer. (Only enable if the board has an accelerometer.)	0xAB 0x00	No error.
0xAA 0x52 0x02 0x01*	Enable WHRM/MaximFast 10.x algorithm.	0xAB 0x00	No error.
0xAA 0x00 0x00*	Read the sensor hub status.	0xAB 0x00 0x08	No error. DataRdyInt bit is set
0xAA 0x12 0x00*	Get the number of samples in the FIFO.	0xAB 0x00 0x0F	No error. 0x0F samples are in the FIFO.

HOST COMMAND	COMMAND DESCRIPTION	MAX32664 RESPONSE	RESPONSE DESCRIPTION
0xAA 0x12 0x01*	Read the data stored in the FIFO.	0xAB 0x00 0x03 0x6A 0x43 0x03 0x04 0x92 0x00 0x00 0x00 0x00 0x2E 0x15x 0x02 0x76 0x63 0x03 0xE4 0x03, data for fourteen other samples	No error. IR counts = 223811, Red counts = 19778, LED3 = 0, LED4 = 11797, Heart Rate = 63.0, Confidence = 99, SpO ₂ = 99.6, MaximFast State Machine Status = 3, data for fourteen other samples.
0xAA 0x40 0x03 0x0D 0x32	Set the MAX30101 register 0xD (LED2, pulse amplitude) to 0x32.	0xAB 0x00	No error.
0xAA 0x40 0x03 0x0C 0x32	Set the MAX30101 register 0xC (LED1, pulse amplitude) to 0x32.	0xAB 0x00	No error.
0xAA 0x40 0x03 0x0A 0x23	Set the MAX30101 register 0xA (ADC Range, Sample Rate, Pulse Width) to 0x23.	0xAB 0x00	No error.
0xAA 0x40 0x03 0x08 0x2F	Set the MAX30101 register 0x08 (Sample Average, FIFO Full) to 0x2F.	0xAB 0x00	No error.
0xAA 0x00 0x00*	Read the Sensor Hub Status.	0xAB 0x00 0x08	No error. DataRdyInt bit is set.
0xAA 0x12 0x00*	Get the number of samples in FIFO.	0xAB 0x00 0x0F	No error. Fifteen samples available.
0xAA 0x12 0x01*	Read the data stored in FIFO.	0xAB 0x00 0x03 0x6A 0x43 0x03 0x04 0x92 0x00 0x00 0x00 0x00 0x2E 0x15x 0x02 0x76 0x63 0x03 0xE4 0x03, data for fourteen other samples	No error. IR counts = 223811, Red counts = 19778, LED3 = 0, LED4 = 11797, Heart Rate = 63.0, Confidence = 99, SpO ₂ = 99.6, MaximFast State Machine Status = 3, data for fourteen other samples.
0xAA 0x00 0x00*	Read the Sensor Hub Status.	0xAB 0x00 0x08	No error. DataRdyInt bit is set.
0xAA 0x12 0x00*	Get the number of samples in FIFO.	0xAB 0x00 0x0F	No error. Fifteen samples available.
0xAA 0x12 0x01*	Read the data stored in FIFO.	0xAB 0x00 ...	No error. Data read.

*Mandatory

?Recommended

I²C Commands to Flash the Application Algorithm

The MAX32664 is pre-programmed with bootloader firmware which accepts in-application programming of the Maxim supplied algorithm file (.msbl). Table 10 is a capture of the I²C commands that are necessary to flash the application algorithm to the MAX32664.

IMPORTANT: Do not enable the accelerometer if your board does not have the accelerometer.

This example was captured with the MAX32630FTHR acting as the host microcontroller. The MAX32664 uses the 8-bit slave address of 0xAA. The example encrypted algorithm file used was the MAX32660_SmartSensor_OS24_MaximFast_1.8.2a.msbl (26 pages, 8196 bytes for the page size). Each page sent includes 16 CRC bytes for that page, so there are 8208 bytes per page sent in the payload of the message. The number of pages is located at address 0x44 in the .msbl file.

Table 10. Annotated I²C Trace for Flashing the Application

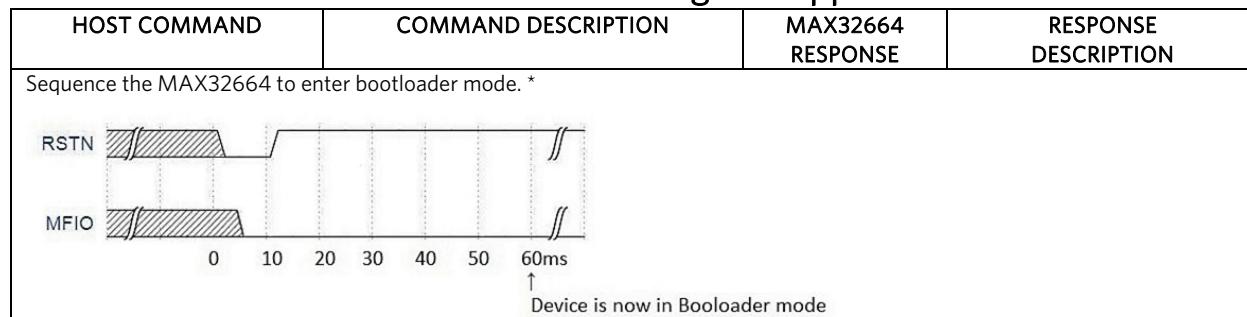


Figure 7. Sequence to enter bootloader mode.

0xAA 0x01 0x00 0x08*	Set mode to 0x08 for bootloader mode.	0xAB 0x00	No error.
0xAA 0x02 0x00	Read mode.	0xAB 0x00 0x08	No error. Mode is bootloader.
0xAA 0xFF 0x00+	Get ID and MCU type.	0xAB 0x00 0x01	No error. MCU is MAX32660/MAX32664.
0xAA 0x81 0x00	Read bootloader firmware version.	0xAB 0x00 0x03 0x00 0x00	No error. Version is 3.0.0.
0xAA 0x81 0x01	Read bootloader page size.	0xAB 0x00 0x20 0x00	No error. Page size is 8192.
0xAA 0x80 0x02 0x00 0x1A*	Bootloader flash. Set the number of pages to 31 based on byte 0x44 from the application .msbl file, which is created from the user application .bin file.	0xAB 0x00	No error.

Figure 8. Page number byte 0x44 from the .msbl file.

0x00000044 02 ed 27 af 1a 00 00 20 04 00 00 00 c2 31 90 2c	Bootloader flash. Set the initialization vector bytes 0x28 to 0x32 from the .msbl file.	0xAB 0x00	No error.
00000000 6d 73 62 6c 00 00 00 00 4d 41 58 33 32 36 36 30 00000010 00 00 00 00 00 00 00 00 41 45 53 2d 32 35 36 00 00000020 00 00 00 00 00 00 00 00 1a db e5 0d 90 79 e6 c6 00000032 13 87 b9 00 2b f5 ad cd 2e 47 d2 83 23 88 37 63 00000040 02 ed 27 af 1a 00 00 20 04 00 00 00 c2 31 90 2c 00000050 e4 c8 37 e9 18 92 ad 3b 64 e7 0a ed eb 40 c1 66 00000060 e2 23 4f 71 d4 6b 98 e3 a7 f9 85 80 7a 4e 17 e7			

Figure 9. Initialization vector bytes 0x28 to 0x32 from the .msbl file.

0xAA 0x80 0x01 0x2B 0xF5 0xAD 0xCD 0x2E 0x47 0xD2 0x83 0x23 0x88 0x37 0x62 0x02 0xED 0x27 0xAF*	Bootloader flash. Set the authentication bytes 0x34 to 0x43 of the .msbl file.	0xAB 0x00	No error.
--	--	-----------	-----------

HOST COMMAND	COMMAND DESCRIPTION	MAX32664 RESPONSE	RESPONSE DESCRIPTION
00000030 13 87 b9 00 2b f5 ad cd 2e 47 d2 83 23 88 37 63 00000043 02 ed 27 af 1a 00 00 20 04 00 00 00 c2 31 90 2c			
Figure 10. Authentication bytes 0x34 to 0x43 from the .msbl file.			
0xAA 0x80 0x03*	Bootloader flash. Erase application.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xC2 0x31 0x90 ... 0x9E 0x6A 0x0E*	Bootloader flash. Send page bytes 0x4C to 0x205B from the .msbl file.	0xAB 0x00	No error.
00000040 02 ed 27 af 1a 00 00 20 04 00 00 00 c2 31 90 2c 00000050 e4 c8 37 e9 18 92 ad 3b 64 e7 0a ed eb 40 c1 66 0000006f e2 23 4f 71 d4 6b 98 e3 a7 f9 85 80 7a 4e 17 e7 00002040 9e 7c c0 3c 47 81 91 35 27 4c be cc 2a 7f ab 1f 0000205b 00 0d d6 ce 6f d4 ee cc b2 9e 6a 0e cc c5 68 92			
Figure 11. Send page bytes 0x4C to 0x205B from the .msbl file.			
0xAA 0x80 0x04 0xCC 0xC5 0x68 ... 0xF7 0xD6 0x4C*	Bootloader flash. Send page bytes 0x205C to 0x406B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x2E 0xA6 0x13 ... 0x84 0xF7 0xCF*	Bootloader flash. Send page bytes 0x406C to 0x607B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xD7 0x1F 0x7F ... 0x55 0xAB 0xB8*	Bootloader flash. Send page bytes 0x607C to 0x808B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xC4 0x63 0x2B ... 0x48 0xCD 0x52*	Bootloader flash. Send page bytes 0x808C to 0xA09B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x89 0x33 0x22 ... 0x31 0xAD 0x19*	Bootloader flash. Send page bytes 0xA09C to 0xCOAB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x8B 0x97 0x18 ... 0xF3 0xCF 0x90*	Bootloader flash. Send page bytes 0xCOAC to 0xE0BB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xD0 0x78 0x38 ... 0x1F 0x7F 0x92*	Bootloader flash. Send page bytes 0xE0BC to 0x100CB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xB1 0xE9 0x8F ... 0xF4 0x23 0xD8*	Bootloader flash. Send page bytes 0x100CC to 0x120DB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xF8 0xC6 0x83 ... 0xF4 0x24 0xE2*	Bootloader flash. Send page bytes 0x120DC to 0x140EB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x1F 0x4F 0x5C ... 0xCC 0x2E 0xCD*	Bootloader flash. Send page bytes 0x140EC to 0x160FB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x40 0x1F 0x03 ... 0x26 0xEB 0xB9*	Bootloader flash. Send page bytes 0x160FC to 0x1810B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x2F 0xD9 0xB2 ... 0xEE 0x2A 0x8F*	Bootloader flash. Send page bytes 0x1810C to 0x1A11B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x51 0x32 0x47 ... 0x41 0xE6 0x47*	Bootloader flash. Send page bytes 0x1A11C to 0x1C12B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x22 0xA6 0x06 ... 0x2A 0xCB 0x44*	Bootloader flash. Send page bytes 0x1C12C to 0x1E13B from the .msbl file.	0xAB 0x00	
0xAA 0x80 0x04 0x68 0x9E 0x1E ... 0x53 0x89 0xE8*	Bootloader flash. Send page bytes 0x1E13C to 0x2014B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x5F 0x1A 0x6A ... 0x14 0xA1 0x85*	Bootloader flash. Send page bytes 0x2014C to 0x2215B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xE8 0xDE 0xC9 ... 0x81 0xD8 0x00*	Bootloader flash. Send page bytes 0x2215C to 0x2416B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x0E 0xD2 0x16 ... 0x8D 0x69 0xEE*	Bootloader flash. Send page bytes 0x2416C to 0x2617B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x2F 0x4B 0x38 ... 0x02 0xA7 0xDC*	Bootloader flash. Send page bytes 0x2617C to 0x2818B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xA5 0xFE 0xFD ... 0xE3 0x38 0x89*	Bootloader flash. Send page bytes 0x2818C to 0x2A19B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x52 0x88 0x9A ... 0xF0 0xC5 0x9D*	Bootloader flash. Send page bytes 0x2A19C to 0x2C1AB from the .msbl file.	0xAB 0x00	No error.

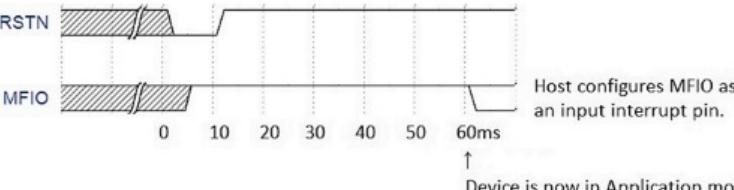
HOST COMMAND	COMMAND DESCRIPTION	MAX32664 RESPONSE	RESPONSE DESCRIPTION
0xAA 0x80 0x04 0xA3 0xA6 0x92 ... 0xA0 0x4D 0xBE*	Bootloader flash. Send page bytes 0x2C1AC to 0x2E1BB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x47 0x09 0x75 ... 0x24 0xBD 0x3D*	Bootloader flash. Send page bytes 0x2E1BC to 0x301CB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x44 0xEC 0xE6 ... 0xBC 0xC9 0x5E*	Bootloader flash. Send page bytes 0x301CC to 0x321DB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xD3 0x58 0x34 ... 0x62 0x00 0x37*	Bootloader flash. Send page bytes 0x321DC to 0x341EB from the .msbl file.	0xAB 0x00	No error.
Sequence the MAX32664 to enter application mode. *			
 <p>RSTN</p> <p>MFIO</p> <p>Host configures MFIO as an input interrupt pin.</p> <p>Device is now in Application mode.</p>			

Figure 12. Sequence to enter application mode.

0xAA 0x02 0x00+	Read mode.	0xAB 0x00 0x00	No errors. Mode is application.
0xAA 0xFF 0x00	Get ID and MCU type.	0xAB 0x00 0x01	No error. MCU is MAX32660/MAX32664
0xAA 0xFF 0x03	Get Sensor Hub version.	0xAB 0x00 0x01 0x08 0x02	No error. Version is 1.8.2.
0xAA 0x42 0x03+	Get the MAX30101 AFE register attributes.	0xAB 0x00 0x01 0x24	No error. Attributes are 1 byte, 0x24 registers available.
0xAA 0x43 0x03	Read all the MAX30101 registers.	0xAB 0x00 0x00 0x00 0x01 0x00 0x02 0x40...	No error. Reg 0x00=0, reg 0x01=0, reg0x02=0x40, ... Returns the Read Status Byte and 36 pairs of numbers.
0xAA 0x41 0x03 0x07	Read the MAX30101 register 7.	0xAB 0x00 0x00	No error. Register 0x07 is 0.

*Mandatory

+Recommended

It is recommended to program the latest version of the MAX32664 sensor hub algorithm .msbl file into the MAX32664 chip. Check the version that is programmed into the chip by using the command "Identity, Read sensor hub version." The latest sensor hub algorithm is available for download for the MAX32664, MAXREFDES220#, and MAXREFDES101# from the Maxim website.

In-Application Programming of the MAX32664

The MAX32664 allows for in-application programming of the firmware.

In-application programming allows for the programming of the sensor hub application firmware during manufacturing and for allowing over-the-air (OTA) updates of the application firmware in the product.

Figure 13 is a flowchart of the in-application programming.

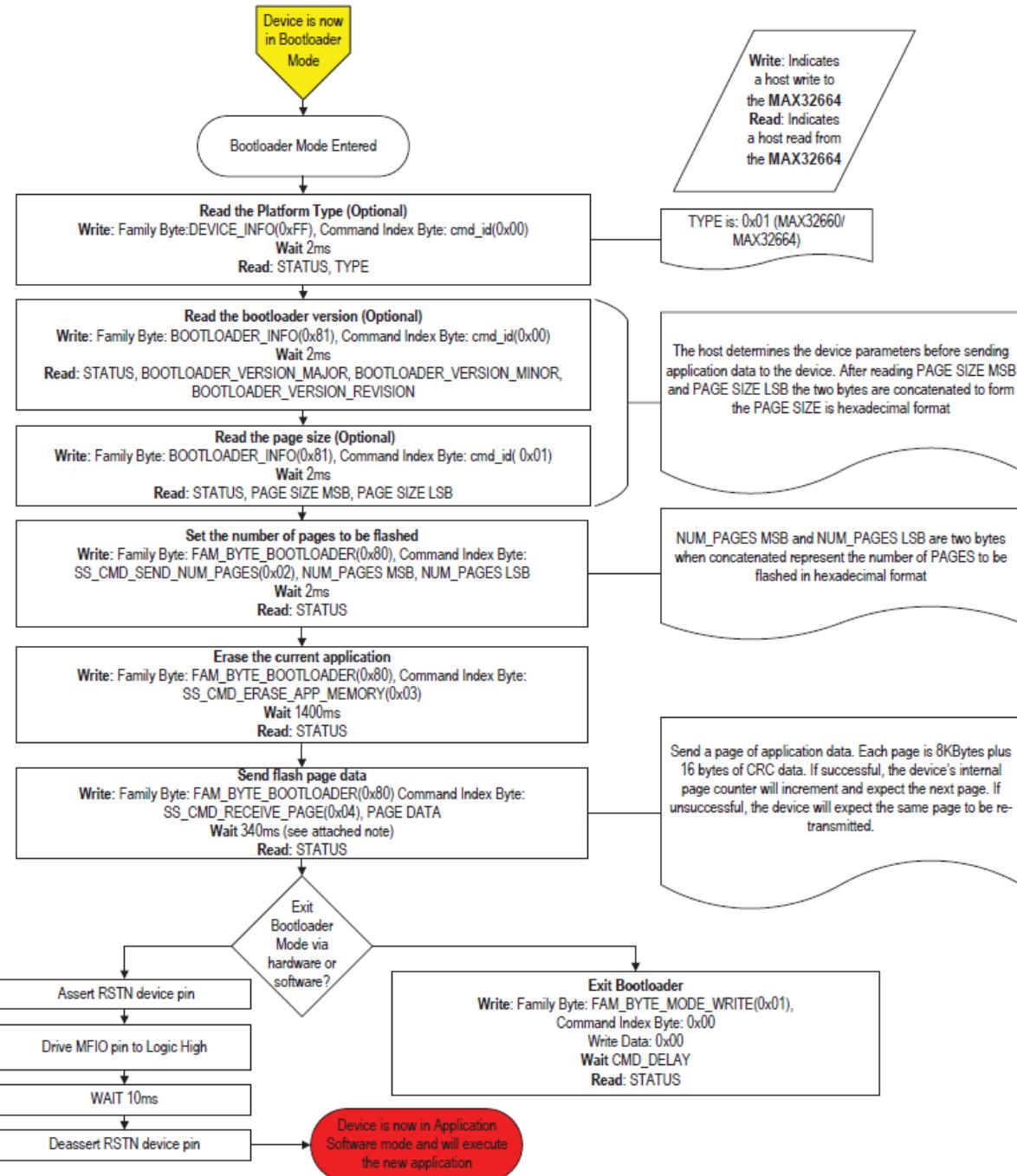


Figure 13. MAX32664 in-application programming flowchart.

MAX32664 APIs and Methods for Reset, Sleep, Status, Heartbeat

Table 11 summarizes the commands and methods to place the MAX32664 into reset or sleep, to interrogate its status, or to generate the “heartbeat” (a periodic signal generated by the software to indicate normal operation).

Table 11. MAX32664 I²C Message Protocol Definitions

COMMAND NAME	HOST COMMAND TO MAX32664	DESCRIPTION
MAX32664 Soft Reset	0xAA 0x01 0x00 0x02	Puts MAX32664 into reset.
MAX30101 AFE Soft Reset by Write Register to AFE	0xAA 0x40 0x03 0x09 0x40	Write 0x40 to MAX30101 register 0x09 to issue a soft reset to the MAX30101. The AFE must be enabled using the enable command.
MAX32664 Sleep	0xAA 0x01 0x00 0x01	To be implemented in the future.
MAX32664 Sleep between Interrupts		To be implemented in the future.
MAX30101 AFE Sleep, Use Write Reg to AFE	0xAA 0x40 0x03 0x09 0x80	Write 0x80 to MAX30101 register 0x09 to put the MAX30101 into shutdown mode. The AFE must be enabled using the enable command.
MAX32664 Hard Reset	Use MFIO and RSTN pins according to Figure 4 and Figure 5.	
WDT in MAX32664 Bootloader Mode		Not implemented.
WDT in MAX32664 .msbl Application mode		Not implemented.
Bootloader or Application Status	0XAA 0x02 0x00	Send the read mode command. Response is 0xAB 0x00 0x08 if in bootloader mode or 0xAB 0x00 0x00 if in application mode.
Heartbeat for Application Mode		Sample source and .msbl file to toggle P0.9 is provided.

Revision History

REVISION NUMBER	REVISION DATE	DESCRIPTION	PAGES CHANGED
0	01/19	Initial release	—

©2019 by Maxim Integrated Products, Inc. All rights reserved. Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. MAXIM INTEGRATED PRODUCTS, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. MAXIM ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering or registered trademarks of Maxim Integrated Product

User's Guide

GDM12864HLCM

(Liquid Crystal Display Module)

For product support, contact

XIAMEN OCULAR OPTICS CO.,LTD.

厦门高卓立光电有限公司

South2/F.,Guangxia Building Torch Hi-tech

Industrial Development Area Xiamen, China

36100 中国厦门火炬高技术产业开发区光厦楼南二楼

Tel: 86-592-5650516 Fax: 86-592-5650695

CONTENTS

Mechanical diagram

Absolute maximum ratings

Interface pin connections

Optical characteristics

Electrical characteristics

 KS0107B

 KS0108B

 Write or read cycle

Timing characteristics

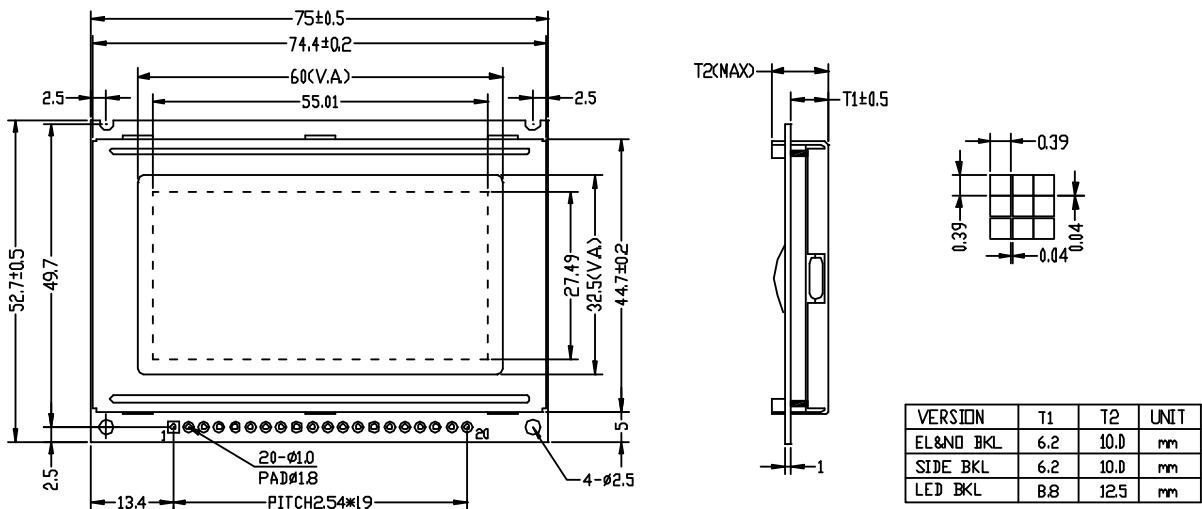
Block diagram

Display commands

Reliability and lift time

Operating Principles & Methods

➤ Mechanical diagram



➤ Absolute maximum ratings

Item	Symbol	Min.	Max.	Unit
Supply voltage for logic	Vdd - Vss	0	6.5	V
Input voltage	Vin	0	Vdd	
Operating temperature range	T0p	-20	70	℃
Storage temperature range	Tst	-25	75	

➤ Interface pin connections

Pin No.	Symbol	Level	Description
1	Vdd	5.0V	Supply voltage for logic and LCD (+)
2	Vss	0V	Ground
3	V0	-	Operating voltage for LCD (variable)
4~11	DB0~DB7	H/L	Data bit 0~7
12	CS2	L	Chip select signal for IC2
13	CS1	L	Chip select signal for IC1
14	/RES	L	Reset signal
15	R/W	H/L	H: read (MUP<- module), L: write (MPU->module)
16	D/I	H/L	H: data, L: instruction code
17	E	H, H L	Chip enable signal
18	VEE	-	Operating voltage for LCD (variable)
19	A	4.2V	Backlight power supply
20	K	0V	Backlight power supply

XIAMEN OCULAR OPTICS CO., LTD.

3

SOUTH2/F, GUANGXIA BUILDING, TORCH HIGH-TECH DEVELOPMENT AREA,
XIAMEN 361006.P.R.CHINA TEL: 86-592-5650516 FAX: 86-592-5650695

➤ Optical characteristics

STN Type display module (Ta=25°C, Vdd=5.0V)

Item	Symbol	Condition	Min.	Typ.	Max.	Unit
Viewing angle	θ	$Cr \geq 2$	-60	-	35	deg
			-40	-	40	
Contrast ratio (rise)	Cr		-	6	-	
Response time (fall)	Tr	-	-	150	250	ms
	Tr	-	-	150	250	ms

➤ Electrical characteristics

Item	Symbol	Condition	Standard value			Unit	
			Min.	Typ.	Max.		
Supply voltage for	Logic	Vdd - Vss	-	4.75	5.0	5.25	
	LCD	Vdd-V0	-	-	9.5	-	
Supply current for	Logic	Idd	-	-	2.5	-	
	LCD	Iee	-	-	1.0	-	
Operating voltage for LCD (Recommended)		Vdd-v0	-	-	-	V	
			25°C	-	9.5		
			-	-	-		
Input voltage	H: level	Vih	High level	0.7Vdd	-	Vdd	
	L: Level	Vil	Low level	0	-	0.3Vdd	

Electrical Absolute Maximum Ratings (KS0107B)

Parameter	Symbol	Rating	Unit	Note
Operating voltage	V _{DD}	-0.3 ~ +7.0	V	*1
Supply voltage	V _{EE}	V _{DD} -19.0 ~ V _{DD} +0.3	V	*4
Driver supply voltage	V _B	-0.3 ~ V _{DD} +0.3	V	*1,2
	V _{LCD}	V _{EE} -0.3 ~ V _{DD} +0.3	V	*3,4

*Notes:

- *1. Based on V_{SS} = 0V
- *2. Applies to input terminals and I/O terminals at high impedance. (Except V0L, V1L, V4L, and V5L)
- *3. Applies to V0L, V1L, V4L, and V5L.
- *4. Voltage level: V_{DD} ≥ V0 ≥ V1 ≥ V2 ≥ V3 ≥ V4 ≥ V5 ≥ V_{EE}

DC Electrical Characteristics (KS0107B)

(VDD= 4.5 to 5.5V, VSS=0V, VDD-VEE=8~17V, Ta= -30 to +85°C)

Item	Symbol	Condition	Min.	Typ.	Max.	Unit	Note
Operating voltage	V _{DD}	-	4.5	-	5.5	V	
Input voltage	V _{IH}	-	0.7 _{VDD}	-	V _{DD}		*1
	V _{IL}	-	V _{SS}	-	0.3V _{DD}		
output voltage	V _{OH}	I _{OH} = -0.4mA	V _{DD} -0.4	-	-	*2	
	V _{OL}	I _{OL} = 0.4mA	-	-	0.4		
Input leakage current	I _{LKG}	V _{IN} = V _{DD} ~ V _{SS}	-1.0	-	+1.0	μA	*1
OSC Frequency	fosc	Rf=47kΩ ±2% Cf=20pF±5%	315	450	585	kHz	
On Resistance (Vdiv-Ci)	R _{ONS}	V _{DD} -V _{EE} =17V Load current±150μA	-	-	1.5	kΩ	
Operating current	I _{DD1}	Master mode 1/128 Duty	-	-	1.0	mA	*3
	I _{DD2}	Master mode 1/128 Duty	-	-	0.2		*4
Supply Current	I _{EE}	Master mode 1/128 Duty	-	-	0.1		*5
Operating	f _{op1}	Master mode External Duty	50	-	600	kHz	
Frequency	f _{op2}	Slave mode	0.5	-	1500		

Notes

- *1. Applies to input terminals FS, DS1, DS2, CR, SHL, MS and PCLK2 and I/O terminals DIO1, DIO2, M, and CL2 in the input state.
- *2. Applies to output terminals CLK1, CLK2 and FRM and I/O terminals DIO1, DIO2, M, and CL2 in the output state.
- *3. This value is specified about current flowing through Vss.
Internal oscillation circuit: Rf=47kΩ , cf=20pF
- Each terminals of DS1, DS2, FS, SHL, and MS is connected to VDD and out is no load.
- *4. This value is specified about current flowing through Vss.
Each terminals is DS1, DS2, FS, SHL, PCLK2 and CR is connected to VDD,MS is connected to Vss and CL2, M, DIO1 is external clock.
- *5. This value is specified about current flowing through VEE,Don't connect to VLCD(V1~V5).

Electrical Absolute Maximum Ratings (KS0108B)

Parameter	Symbol	Rating	Unit	Note
Operating voltage	V _{DD}	-0.3 ~ +7.0	V	*1
Supply voltage	V _{EE}	V _{DD} -19.0 ~ V _{DD} +0.3	V	*4
Driver supply voltage	V _B	-0.3 ~ V _{DD} +0.3	V	*1,3
	V _{LCD}	V _{EE} -0.3 ~ V _{DD} +0.3	V	*2

***Notes:**

- *1. Based on V_{SS} = 0V
- *2. Applies the same supply voltage to V_{EE}. V_{LCD}=V_{DD}-V_{EE}.
- *3. Applies to M, FRM, CLK1, CLK2, CL, RESETB, ADC, CS1B, CS2B, CS3, E, R/W, RS and DB0~DB7.
- *4. Applies V_{0L}, V_{2L}, V_{3L} and V_{5L}.

Voltage level: V_{DD}≥V₀≥V₁≥V₂≥V₃≥V₄≥V₅≥V_{EE}

DC Electrical Characteristics (KS0108B)

(V_{DD}= 4.5 to 5.5V, V_{SS}=0V, V_{DD}-V_{EE}=8~17V, Ta= -30 to +85°C)

Item	Symbol	Condition	Min.	Typ.	Max.	Unit	Note
Operating voltage	V _{DD}	-	4.5	-	5.5		
Input High voltage	V _{IH1}	-	0.7 _{VDD}	-	V _{DD}	V	*1
	V _{IH2}	-	2.0	-	V _{DD}		*2
Input Low voltage	V _{IL1}	-	0	-	0.3V _{DD}		*1
	V _{IL2}	-	0	-	0.8		*2
Output High Voltage	V _{OH}	I _{OH} = -0.2mA	2.4	-	-		*3
Output Low Voltage	V _{OL}	I _{OL} = 1.6mA	-	-	0.4		*3
Input leakage current	I _{LKG}	V _{IN} = V _{SS} ~ V _{DD}	-1.0	-	+1.0	μA	*4
Three-state (OFF) Input Current	I _{TSL}	V _{IN} = V _{SS} ~ V _{DD}	-5.0	-	5.0		*5
Driver Input leakage current	I _{DIL}	V _{IN} = V _{EE} ~ V _{DD}	-2.0	-	2.0		*6
On Resistance (V _{div} -C _i)	R _{ONS}	V _{DD} -V _{EE} =15V Load current±100μA	-	-	7.5	kΩ	*8
Operating current	I _{DD1}	During Display	-	-	0.1	mA	*7
	I _{DD2}	During Access Access Cycle=1MHz	-	-	0.5		*7

Notes

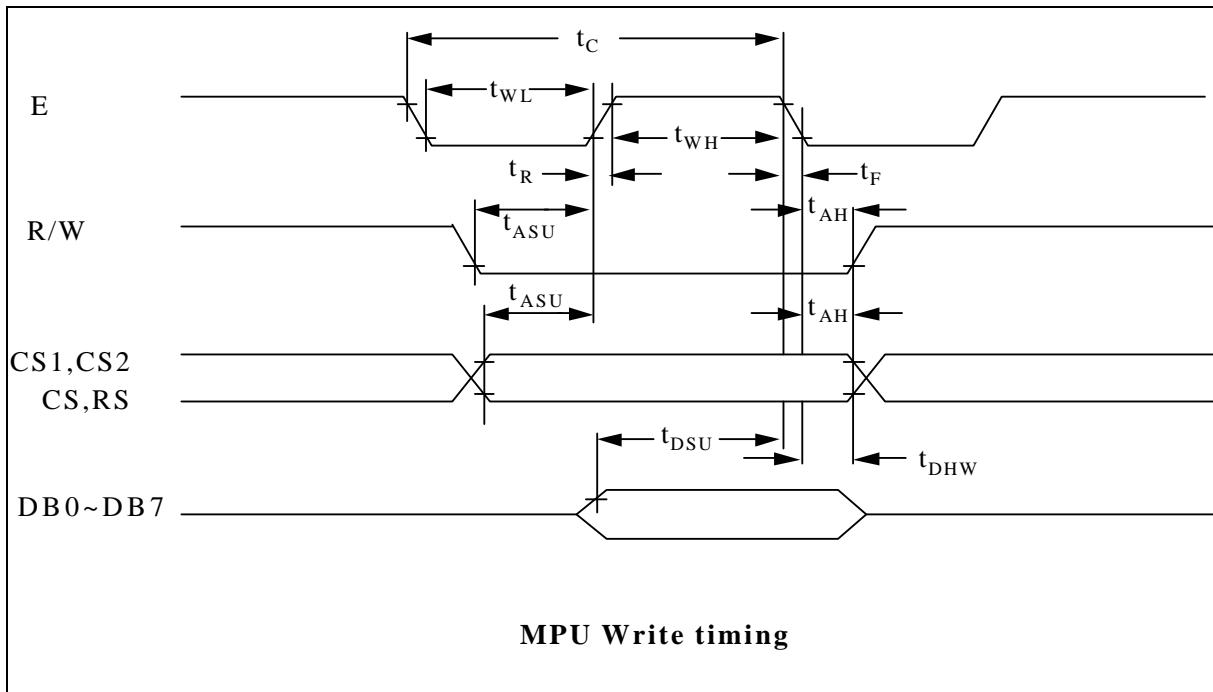
- *1. CL, FRM, M, RSTB, CLK1, CLK2
- *2. CS1B, CS2B, CS3, E, R/W, RS, DB0~DB7
- *3. DB0~DB7
- *4. Except DB0~DB7
- *5. DB0~DB7 at high impedance
- *6. V₀, V₁, V₃, V₃, V₄, V₅
- *7. 1/64 duty, FCLK=250KHZ, Frame Frequency=70HKZ, Output: No Load
- *8. V_{DD}-V_{EE}=15.5V

$$V_{0L} > V_{2L} \geq V_{DD} - 2/7(V_{DD} - V_{EE}) > V_{3L} = V_{EE} + 2/7(V_{DD} - V_{EE}) > V_{5L}$$

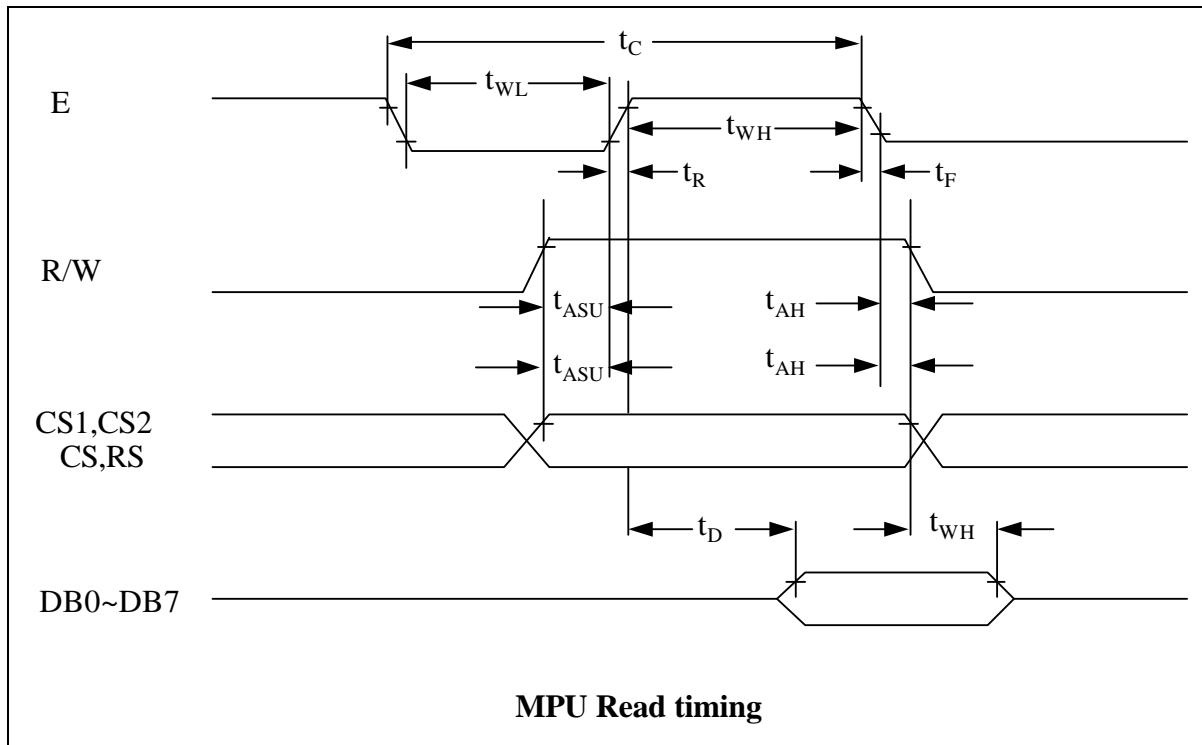
➤ Write or read cycle

Characteristic	Symbol	Min.	Typ.	Max.	Unit
E cycle	Tc	1000	-	-	ns
E high level width	Twh	450	-	-	ns
E low level width	Twl	450	-	-	ns
E rise time	Tr	-	-	25	ns
E fall time	Tf	-	-	25	ns
Address set-up time	Tasu	140	-	-	ns
Address hold time	Tah	10	-	-	ns
Data set-up time	Tdsu	200	-	-	ns
Data delay time	Td	-	-	320	ns
Data hold time (write)	Tdhw	10	-	-	ns
Data hold time (read)	Tdhr	20	-	-	ns

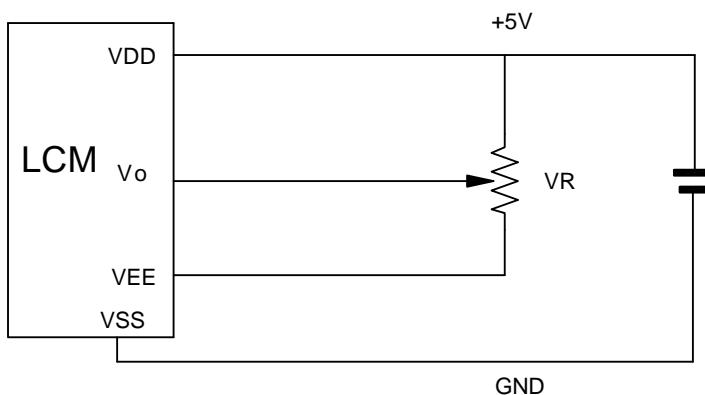
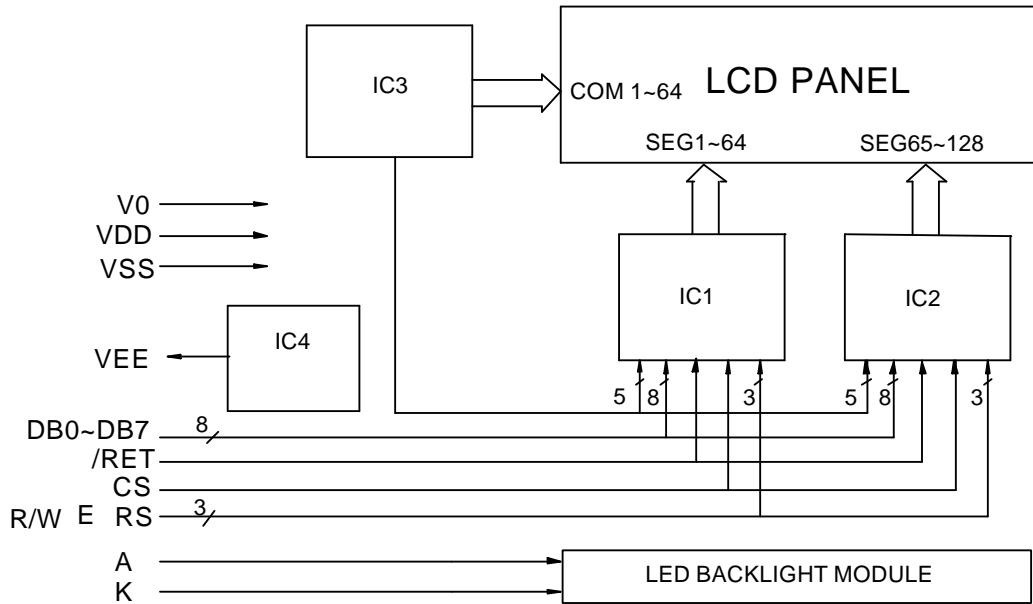
✧ Write timing



✧ Read timing



✧ Block diagram



VDD-Vo:LCD DRIVING VOLTAGE

VR:10K~20K

*Note

1/64 duty, 1/9 bias
 $V_{DD} > V_1 > V_2 > V_3 > V_4 > V_5 > V_{EE}$

✧ Display Control Instruction

The display control instructions control the internal state of the KS0108B. Instruction is received from MPU to KS0108B for the display control. The following table shows various instructions.

Instruction	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Function
Read Display Data	1	1	Read data						Reads data (DB[7:0])from display data RAM to the data bus.		
Write Display Data	1	0	Write data						Writes data (DB[7:0]) into display data RAM. After writing instruction, Y address is incremented by 1 automatically		
Status Read	0	1	Busy	0	ON/OFF	Re-set	0	0	0	0	Reads the internal status BUSY 0: Ready 1: In operation ON/OFF 0: Display ON 1: Display OFF RESET 0: Normal 1: Reset
Set Address (Y address)	0	0	0	1	Y address (0~63)						Sets the Y address in the Y address counter
Set Display Start Line	0	0	1	1	Display start line (0~63)						Indicates the display data RAM displayed at the top of the screen.
Set Address (X address)	0	0	1	0	1	1	1	Page (0~7)			Sets the X address at the X address register.
Display On/off	0	0	0	0	1	1	1	1	1	0/1	Controls the display ON or OFF. The internal status and the DDRAM data is not affected. 0: OFF, 1: ON

1. Display On/Off

The display data appears when D is 1 and disappears when D is 0.

Though the data is not on the screen with D=0, it remains in the display data RAM.

Therefore, you can make it appear by changing D=0 into D=1.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	1	1	1	1	D

2. Set Address (Y Address)

Y address (AC0~AC5) of the display data RAM is set in the Y address counter.

An address is set by instruction and increased by 1 automatically by read or write operations of display data.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0

3. Set Page (X Address)

X address (AC0~AC2) of the display data RAM is set in the X address register.

Writing or reading to or from MPU is executed in this specified page until the next page is set.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	0	1	1	1	AC2	AC1	AC0

4. Display Start Line (Z Address)

Z address (AC0~AC5) of the display data RAM is set in the display start line register and displayed at the top of the screen.

When the display duty cycle is 1/64 or others (1/32~1/64), the data of total line number of LCD screen, from the line specified by display start line instruction, is displayed.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	1	AC5	AC4	AC3	AC2	AC1	AC0

5. Status Read

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	0	BUSY	0	ON/OFF	RESET	0	0	0	0

- BUSY

When BUSY is 1, the Chip is executing internal operation and no instructions are accepted.
When BUSY is 0, the Chip is ready to accept any instructions.

- ON/OFF

When ON/OFF is 1, the display is on.
When ON/OFF is 0, the display is off.

- RESET

When RESET is 1, the system is being initialized.
In this condition, no instructions except status read can be accepted.
When RESET is 0, initializing has finished and the system is in the usual operation condition.

6. Write Display Data

Writes data (D0~D7) into the display data RAM.

After writing instruction, Y address is increased by 1 automatically.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	0	D7	D6	D5	D4	D3	D2	D1	D0

7. Read Display Data

Reads data (D0~D7) from the display data RAM.

After reading instruction, Y address is increased by 1 automatically.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	1	D7	D6	D5	D4	D3	D2	D1	D0

❖ Operating principles & methods

1. I/O Buffer

Input buffer controls the status between the enable and disable of chip. Unless the CS1B to CS3 is in active mode, Input or output of data and instruction does not execute. Therefore internal state is not change. But RSTB and ADC can operate regardless CS!B-CS3.

2. Input register

Input register is provided to interface with MPU which is different operating frequency. Input register stores the data temporarily before writing it into display RAM.

When CS1B to CS3 are in the active mode, R/W and RS select the input register. The data from MPU is written into input register. Then writing it into display RAM. Data latched for falling of the E signal and write automatically into the display data RAM by internal operation.

3. Output register

Output register stores the data temporarily from display data RAM when CS1B, CS2B and CS3 are in active mode and R/W and RS=H, stored data in display data RAM is latched in output register. When CS1B to CS3 is in active mode and R/W=H , RS=L, status data (busy check) can read out.

To read the contents of display data RAM, twice access of read instruction is needed. In first access, data in display data RAM is latched into output register. In second access, MPU can read data which is latched. That is to read the data in display data RAM, it needs dummy read. But status read is not needed dummy read.

RS	R/W	Function
L	L	Instruction
	H	Status read (busy check)
H	L	Data write (from input register to display data RAM)
	H	Data read (from display data RAM to output register)

4. Reset

The system can be initialized by setting RSTB terminal at low level when turning power on, receiving instruction from MPU. When RSTB becomes low, following procedure is occurred.

1. Display off

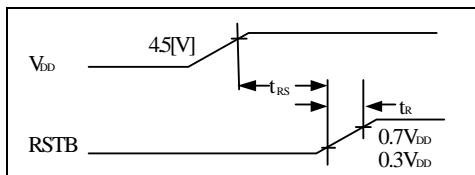
2. Display start line register become set by 0. (Z-address 0)

While RSTB is low, No instruction except status read can be accepted. Therefore, execute other instructions after making sure that DB4= (clear RSTB) and DB7=0 (ready) by status read instruction.

The conditions of power supply at initial power up are shown in table 1.

Table 1. Power Supply Initial Conditions

Item	Symbol	Min	Typ	Max	Unit
Reset Time	t_{RS}	1.0	-	-	us
Rise Time	t_R	-	-	200	ns

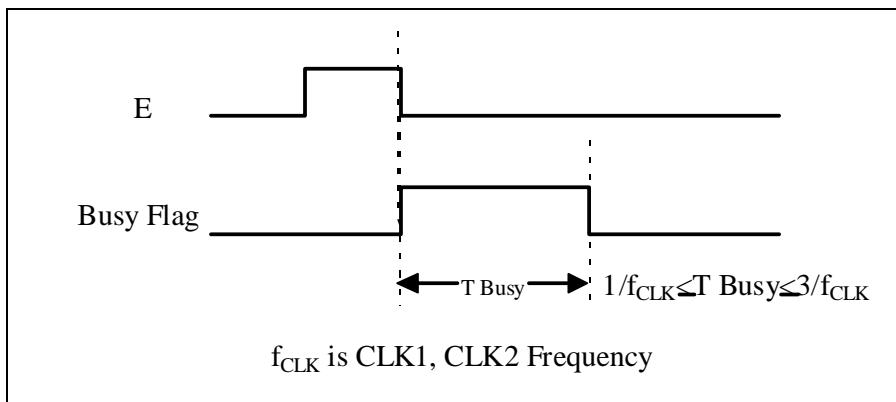


5. Busy flag

Busy flag indicates that KS0108B is operating or no operating. When busy flag is high, KS0108B is in internal operating .

When busy flag is low, KS0108B can accept the data or instruction.

DB7indicates busy flag of the KS0108B.



6. Display On/Off Flip-Flop

The display on/off flip-flop makes on/off the liquid crystal display. When flip-flop is reset (logical low), selective voltage or non-selective voltage appears on segment output terminals. When flip-flop is set (logic high), non selective voltage appears on segment output terminals regardless of display RAM data.

The display on/off flip-flop can changes status by instruction. The display data at all segments disappear while RSTB is low.

The status of the flip-flop is output to DB5 by status read instruction.

The display on/off flip-flop synchronized by CL signal.

7. X Page Register

X page register designates pages of the internal display data RAM.

Count function is not available. An address is set by instruction.

8. Y address counter

Y address counter designates address of the internal display data RAM. An address is set by instruction and is increased by 1 automatically by read or writes operations of display data.

9. Display Data RAM

Display data RAM stores a display data for liquid crystal display. To indicate on state dot matrix of liquid crystal display, write datra1. The other way, off state, writes 0.

Display data RAM address and segment output can be controlled by ADC signal.

ADC=H => Y-address 0: S1~Y address 63: S64

ADC=L => Y-address 0: S64~Yaddress 63: S1

ADC terminal connect the V_{DD} or V_{SS}.

10. Display Start Line Register

The display start line register indicates of display data RAM to display top line of liquid crystal display.

Bit data (DB<0.5>) of the display start line set instruction is latched in display start line register. Latched data is transferred to the Z address counter while FRM is high, presetting the Z address counter.

It is used for scrolling of the liquid crystal display screen.

MAX603/MAX604

5V/3.3V or Adjustable, Low-Dropout, Low I_Q , 500mA Linear Regulators

General Description

The MAX603/MAX604 low-dropout, low quiescent current, linear regulators supply 5V, 3.3V, or an adjustable output for currents up to 500mA. They are available in a 1.8W SO package. Typical dropouts are 320mV at 5V and 500mA, or 240mV at 3.3V and 200mA. Quiescent currents are 15 μ A typ and 35 μ A max. Shutdown turns off all circuitry and puts the regulator in a 2 μ A off mode. A unique protection scheme limits reverse currents when the input voltage falls below the output. Other features include foldback current limiting and thermal overload protection.

The output is preset at 3.3V for the MAX604 and 5V for the MAX603. In addition, both devices employ Dual Mode™ operation, allowing user-adjustable outputs from 1.25V to 11V using external resistors. The input voltage supply range is 2.7V to 11.5V.

The MAX603/MAX604 feature a 500mA P-channel MOSFET pass transistor. This transistor allows the devices to draw less than 35 μ A over temperature, independent of the output current. The supply current remains low because the P-channel MOSFET pass transistor draws no base currents (unlike the PNP transistors of conventional bipolar linear regulators). Also, when the input-to-output voltage differential becomes small, the internal P-channel MOSFET does not suffer from excessive base current losses that occur with saturated PNP transistors.

Features

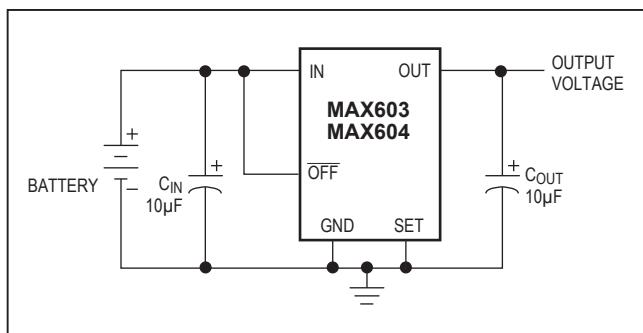
- 500mA Output Current, with Foldback Current Limiting
- High-Power (1.8W) 8-Pin SO Package
- Dual Mode™ Operation: Fixed or Adjustable Output from 1.25V to 11V
- Large Input Range (2.7V to 11.5V)
- Internal 500mA P-Channel Pass Transistor
- 15 μ A Typical Quiescent Current
- 2 μ A (Max) Shutdown Mode
- Thermal Overload Protection
- Reverse-Current Protection

Applications

- 5V and 3.3V Regulators
- 1.25V to 11V Adjustable Regulators
- Battery-Powered Devices
- Pagers and Cellular Phones
- Portable Instruments
- Solar-Powered Instruments

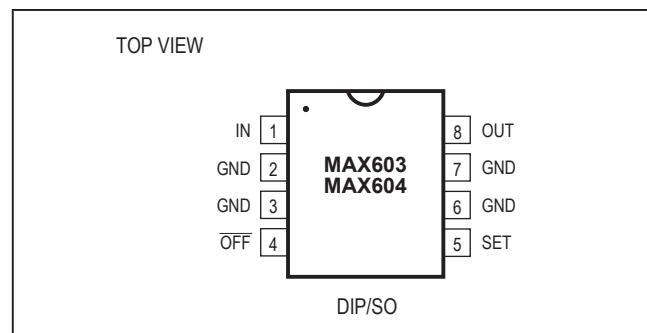
[Ordering Information](#) appears at end of data sheet.

Typical Operating Circuit



Dual Mode is a trademark of Maxim Integrated Products.

Pin Configuration



Absolute Maximum Ratings

Supply Voltage (IN or OUT to GND)	-0.3V to +12V
Output Short-Circuit Duration	1 min
Continuous Output Current	600mA
SET, \overline{OFF} Input Voltages.....	-0.3V to the greater of (IN + 0.3V) or (OUT + 0.3V)
Continuous Power Dissipation ($T_A = +70^\circ C$)	
Plastic DIP (derate 9.09mW/ $^\circ C$ above $+70^\circ C$).....	727mW
SO (derate 23.6mW/ $^\circ C$ above $+70^\circ C$).....	1.8W
CERDIP (derate 8.00mW/ $^\circ C$ above $+70^\circ C$).....	640mW

Operating Temperature Ranges

MAX60_C_A.....	0°C to $+70^\circ C$
MAX60_E_A.....	-40°C to $+85^\circ C$
MAX60_MJA.....	-55°C to $+125^\circ C$
Junction Temperature.....	$+150^\circ C$
Storage Temperature Range.....	-65°C to $+160^\circ C$
Lead Temperature (soldering, 10sec)	+300°C

Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated in the operational sections of the specifications is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Electrical Characteristics

($V_{IN} = 6V$ (MAX603) or 4.3V (MAX604), $C_{IN} = C_{OUT} = 10\mu F$, $\overline{OFF} = V_{IN}$, SET = GND, $T_J = T_{MIN}$ to T_{MAX} , unless otherwise noted. Typical values are at $T_J = +25^\circ C$.) (Note 1)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
Input Voltage	V_{IN}	SET = OUT, $R_L = 1k\Omega$	MAX60_C	2.7	11.5	V
			MAX60_E	2.9	11.5	
			MAX60_M	3.0	11.5	
Output Voltage (Note 2)	V_{OUT}	$I_{OUT} = 20\mu A$ to 500mA, $6.0V < V_{IN} < 11.5V$	MAX603	4.75	5.00	5.25
		$I_{OUT} = 20\mu A$ to 300mA, $4.3V < V_{IN} < 11.5V$	MAX604	3.15	3.30	3.45
Load Regulation	ΔV_{LDR}	$I_{OUT} = 1mA$ to 500mA	MAX603C/E	60	100	mV
			MAX603M		150	
		$I_{OUT} = 1mA$ to 300mA	MAX604		30	100
Line Regulation	ΔV_{LNR}	$(V_{OUT} + 0.5V) \leq V_{IN} \leq 11.5V$, $I_{OUT} = 25mA$		7	40	mV
Dropout Voltage (Note 3)	ΔV_{DO}	$I_{OUT} = 200mA$	MAX603	130	220	mV
		$I_{OUT} = 500mA$		320	550	
		$I_{OUT} = 200mA$	MAX604	240	410	
		$I_{OUT} = 400mA$		480	820	
Quiescent Current	I_Q	$3.0V \leq V_{IN} \leq 11.5V$, SET = OUT	MAX60_C/E	15	35	μA
			MAX60_M		40	
OFF Quiescent Current	$I_{Q OFF}$	$\overline{OFF} \leq 0.4V$, $R_L = 1k\Omega$, $(V_{OUT} + 1V) \leq V_{IN} \leq 11.5V$	MAX60_C	0.01	2	μA
			MAX60_E		10	
			MAX60_M		20	
Minimum Load Current	$I_{OUT MIN}$	$V_{IN} = 11.5V$, SET = OUT	MAX60_C		2	μA
			MAX60_E		6	
			MAX60_M		20	
Foldback Current Limit (Note 4)	I_{LIM}	$V_{OUT} < 0.8V$		350		mA
		$V_{OUT} > 0.8V$ and $V_{IN} - V_{OUT} > 0.7V$		1200		
Thermal Shutdown Temperature	T_{SD}			160		$^\circ C$
Thermal Shutdown Hysteresis	ΔT_{SD}			10		$^\circ C$

Electrical Characteristics (continued)

($V_{IN} = 6V$ (MAX603) or 4.3V (MAX604), $C_{IN} = C_{OUT} = 10\mu F$, $\bar{OFF} = V_{IN}$, SET = GND, $T_J = T_{MIN}$ to T_{MAX} , unless otherwise noted. Typical values are at $T_J = +25^\circ C$.) (Note 1)

PARAMETER	SYMBOL	CONDITIONS		MIN	TYP	MAX	UNITS
Reverse-Current Protection Threshold (Note 5)	ΔV_{RTH}	$V_{OUT} = 4.5V$	MAX603	6	20		mV
		$V_{OUT} = 3.0V$	MAX604	6	20		
Reverse Leakage Current	I_{RL}	$V_{IN} = 0V$, $V_{OUT} = 4.5V$ (MAX603) $V_{OUT} = 3.0V$ (MAX604)	MAX60_C	0.01	10		μA
			MAX60_E		20		
			MAX60_M		100		
Start-Up Overshoot	V_{OSH}	$R_L = 1k\Omega$, $C_{OUT} = 10\mu F$, \bar{OFF} rise time $\leq 1\mu s$		2			% V_{OUT}
Time Required to Exit Shutdown	t_{START}	$V_{IN} = 9V$, $R_L = 18\Omega$, V_{OFF} switched from 0V to V_{IN} , time from 0% to 95% of V_{OUT}		200			μs
Dual-Mode SET Threshold	$V_{SET\ TH}$	For internal feedback		80	30		mV
		For external feedback		150	80		
SET Reference Voltage	V_{SET}	$SET = OUT$, $R_L = 1k\Omega$		1.16	1.20	1.24	V
SET Input Leakage Current	I_{SET}	$V_{SET} = 1.5V$ or 0V			± 0.01	± 10	nA
OUT Leakage Current	$I_{OUT\ LKG}$	$V_{IN} = 11.5V$, $V_{OUT} = 2V$, $SET = OUT$	MAX60_C	0.01	2		μA
			MAX60_E		6		
			MAX60_M		20		
OFF Threshold Voltage	$V_{IL\ OFF}$	Off			0.4		
	$V_{IH\ OFF}$	On, $SET = OUT$, $V_{IN} = 4V$		2.0			V
		On, $SET = OUT$, $V_{IN} = 6V$		3.0			
		On, $SET = OUT$, $V_{IN} = 11.5V$		4.0			
OFF Input Leakage Current	I_{OFF}	$V_{OFF} = V_{IN}$ or GND			± 0.01	± 10	nA
Output Noise (Note 6)	e_n	10Hz to 10kHz, $SET = OUT$, $R_L = 1k\Omega$, $C_{OUT} = 10\mu F$		250			μV_{RMS}

Note 1: Electrical specifications are measured by pulse testing and are guaranteed for a junction temperature (T_J) equal to the operating temperature range. C and E grade parts may be operated up to a T_J of $+125^\circ C$. Expect performance similar to M grade specifications. For T_J between $+125^\circ C$ and $+150^\circ C$, the output voltage may drift more.

Note 2: $(V_{IN} - V_{OUT})$ is limited to keep the product ($I_{OUT} \times (V_{IN} - V_{OUT})$) from exceeding the package power dissipation limits.

Note 3: Dropout Voltage is $(V_{IN} - V_{OUT})$ when V_{OUT} falls to 100mV below its nominal value at $V_{IN} = V_{OUT} + 2V$. For example, the MAX603 is tested by measuring the V_{OUT} at $V_{IN} = 7V$, then V_{IN} is lowered until V_{OUT} falls 100mV below the measured value. The difference $(V_{IN} - V_{OUT})$ is then measured and defined as ΔV_{DO} .

Note 4: Foldback Current Limit was characterized by pulse testing to remain below the maximum junction temperature.

Note 5: The Reverse-Current Protection Threshold is the output/input differential voltage ($V_{OUT} - V_{IN}$) at which reverse-current protection switchover occurs and the pass transistor is turned off.

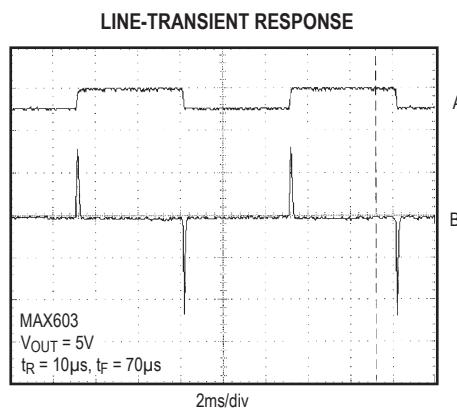
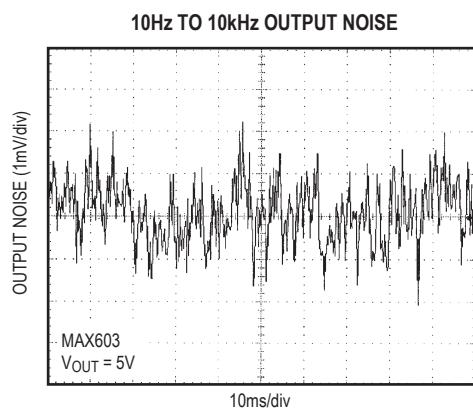
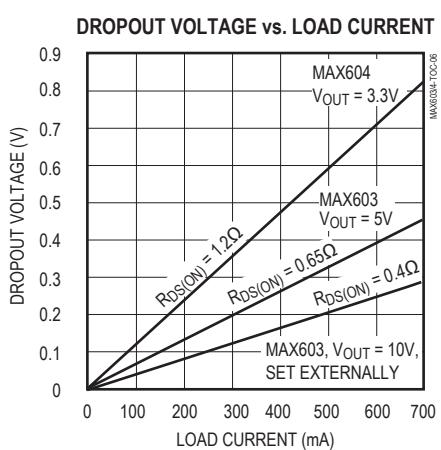
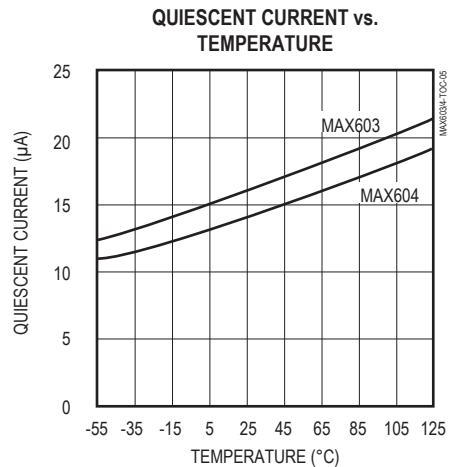
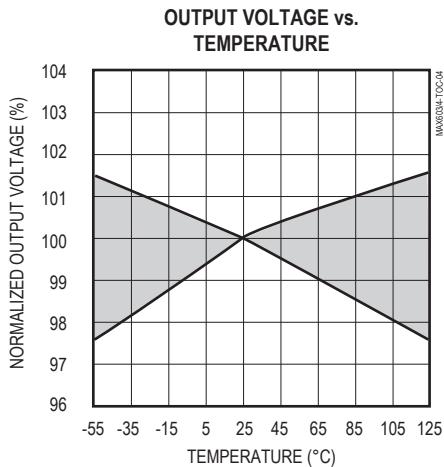
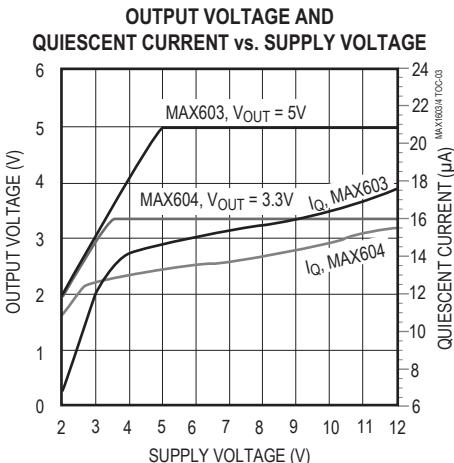
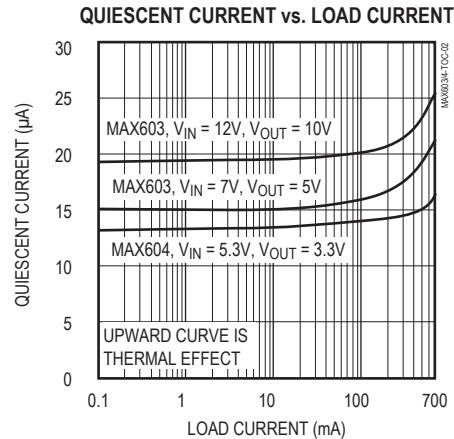
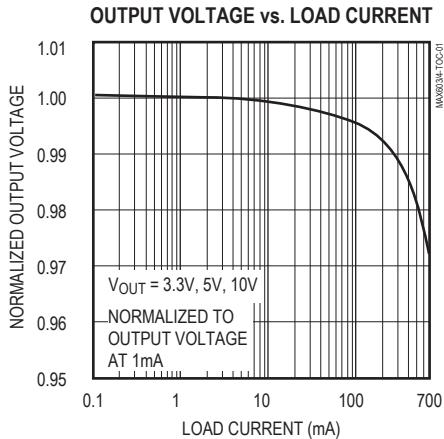
Note 6: Noise is tested using a bandpass amplifier with two poles at 10Hz and two poles at 10kHz.

MAX603/MAX604

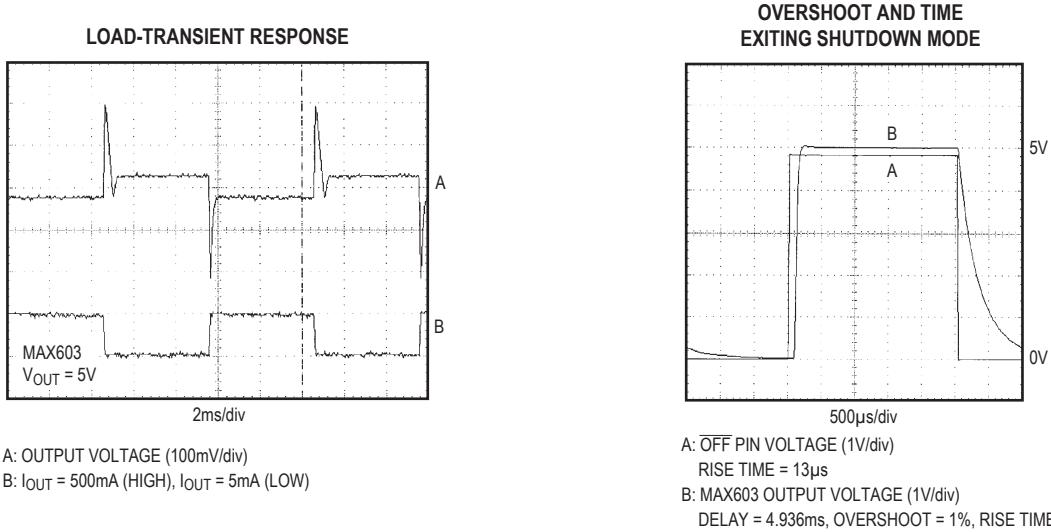
5V/3.3V or Adjustable, Low-Dropout,
Low I_Q , 500mA Linear Regulators

Typical Operating Characteristics

($V_{IN} = 7V$ for MAX603, $V_{IN} = 5.3V$ for MAX604, $\bar{OFF} = V_{IN}$, $SET = GND$, $C_{IN} = C_{OUT} = 10\mu F$, $R_L = 1k\Omega$, $T_J = +25^\circ C$, unless otherwise noted.)



A: $V_{IN} = 8V$ (HIGH), $V_{IN} = 7V$ (LOW)
B: OUTPUT VOLTAGE (50mV/div)

Typical Operating Characteristics (continued)(V_{IN} = 7V for MAX603, V_{IN} = 5.3V for MAX604, $\bar{O}FF$ = V_{IN}, SET = GND, C_{IN} = C_{OUT} = 10µF, R_L = 1kΩ, T_J = +25°C, unless otherwise noted.)**Pin Description**

PIN	NAME	DESCRIPTION
1	IN	Regulator Input. Supply voltage can range from 2.7V to 11.5V.
2, 3, 6, 7	GND	Ground. These pins function as heatsinks, only in the SOIC package. All GND pins must be soldered to the circuit board for proper power dissipation. Connect to large copper pads or planes to channel heat from the IC.
4	$\bar{O}FF$	Shutdown, active low. Switch logic levels in less than 1µs with the high level above the $\bar{O}FF$ threshold.
5	SET	Feedback for Setting the Output Voltage. Connect to GND to set the output voltage to the preselected 3.3V or 5V. Connect to an external resistor network for adjustable output operation.
8	OUT	Regulator Output. Fixed or adjustable from 1.25V to 11.0V. Sources up to 500mA for input voltages above 4V.

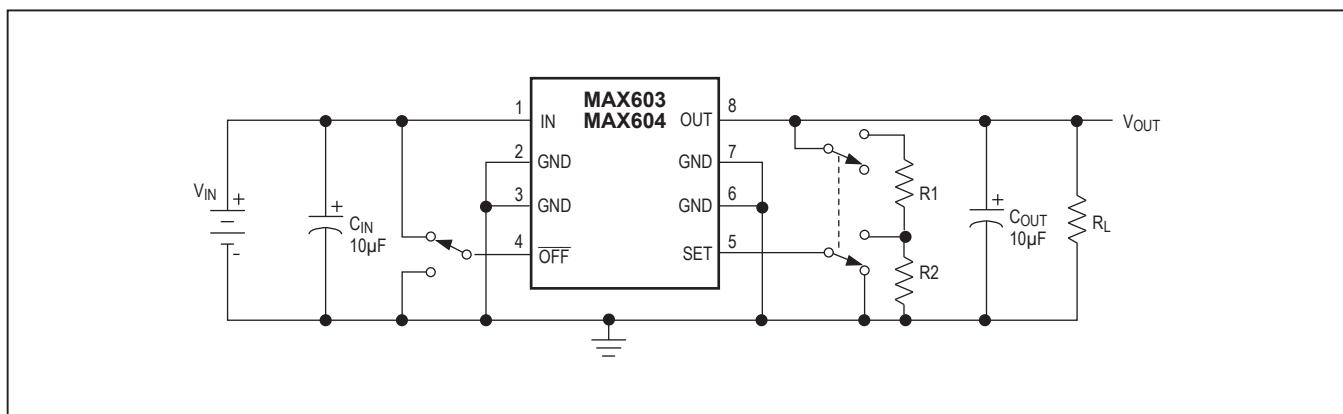


Figure 1. Test Circuit

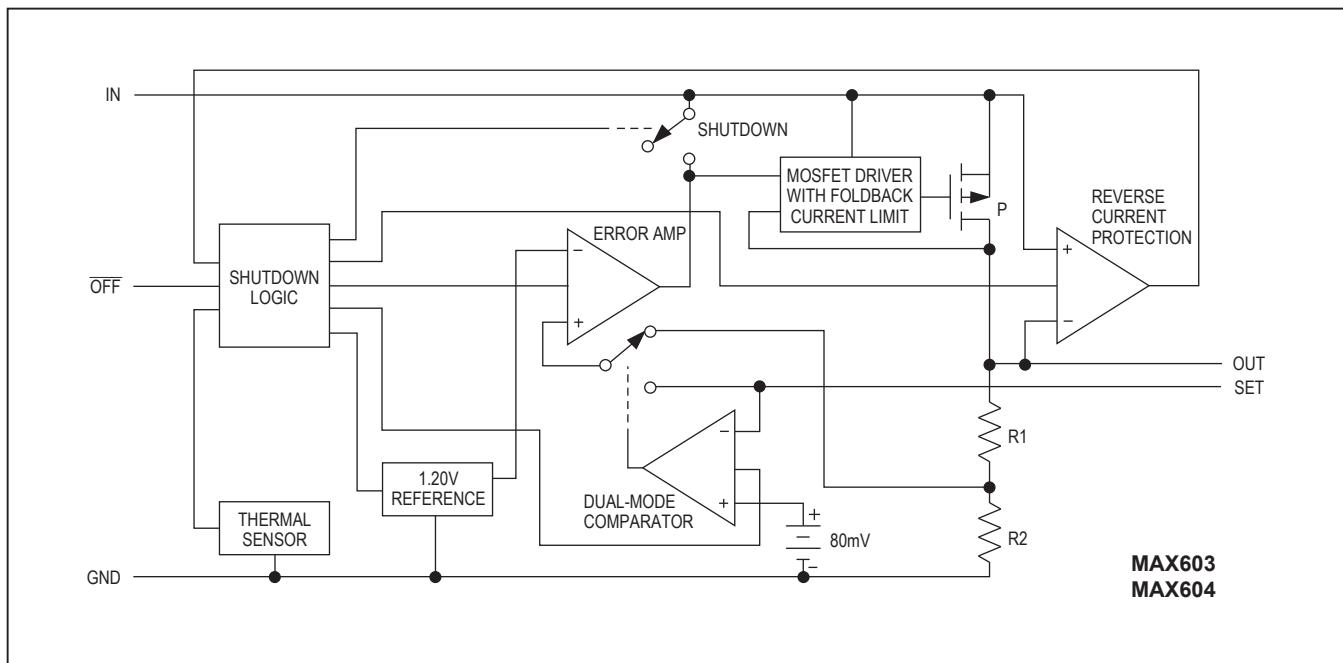


Figure 2. Functional Diagram

Detailed Description

The MAX603/MAX604 are low-dropout, low-quiescent-current linear regulators designed primarily for battery-powered applications. They supply an adjustable 1.25V to 11V output or a preselected 5V (MAX603) or 3.3V (MAX604) output for load currents up to P-channel. As illustrated in Figure 2, they consist of a 1.20V reference, error amplifier, MOSFET driver, P-channel pass transistor, dual-mode comparator, and internal feedback voltage divider.

The 1.20V bandgap reference is connected to the error amplifier's inverting input. The error amplifier compares this reference with the selected feedback voltage and amplifies the difference. The MOSFET driver reads the error signal and applies the appropriate drive to the P-channel pass transistor. If the feedback voltage is lower than the reference, the pass transistor gate is pulled lower, allowing more current to pass and increasing the output voltage. If the feedback voltage is too high, the pass transistor gate is pulled up, allowing less current to pass to the output.

The output voltage is fed back through either an internal resistor voltage divider connected to the OUT pin, or an external resistor network connected to the SET pin. The dual-mode comparator examines the SET voltage and selects the feedback path used. If SET is below 80mV, internal feedback is used and the output voltage is regulated to 5V for the MAX603 or 3.3V for the MAX604. Additional blocks include a foldback current limiter, reverse current protection, thermal sensor, and shutdown logic.

Internal P-Channel Pass Transistor

The MAX603/MAX604 feature a 500mA P-channel MOSFET pass transistor. This provides several advantages over similar designs using PNP pass transistors, including longer battery life.

The P-channel MOSFET requires no base drive, which reduces quiescent current considerably. PNP based regulators waste considerable amounts of current in dropout when the pass transistor saturates. They also use high base-drive currents under large loads. The MAX603/MAX604 do not suffer from these problems and consume only 15 μ A of quiescent current under light and heavy loads, as well as in dropout.

MAX603/MAX604

5V/3.3V or Adjustable, Low-Dropout, Low I_Q , 500mA Linear Regulators

Output Voltage Selection

The MAX603/MAX604 feature dual-mode operation. In preset voltage mode, the output of the MAX603 is set to 5V and the output of the MAX604 is set to 3.3V using internal, trimmed feedback resistors. Select this mode by connecting SET to ground.

In adjustable mode, an output between 1.25V and 11V is selected using two external resistors connected as a voltage divider to SET (Figure 3). The output voltage is set by the following equation:

$$V_{OUT} = V_{SET} \left(1 + \frac{R_1}{R_2} \right)$$

where $V_{SET} = 1.20V$. To simplify resistor selection:

$$R_1 = R_2 \left(\frac{V_{OUT}}{V_{SET}} - 1 \right)$$

Since the input bias current at SET is nominally zero, large resistance values can be used for R_1 and R_2 to minimize power consumption without losing accuracy. Up to $1.5M\Omega$ is acceptable for R_2 . Since the V_{SET} tolerance is less than $\pm 40mV$, the output can be set using fixed resistors instead of trim pots.

In preset voltage mode, impedances between SET and ground should be less than $10k\Omega$. Otherwise, spurious conditions could cause the voltage at SET to exceed the 80mV dual-mode threshold.

Shutdown

A low input on the \overline{OFF} pin shuts down the MAX603/MAX604. In the off mode, the pass transistor, control circuit, reference, and all biases are turned off, reducing the supply current below $2\mu A$. \overline{OFF} should be connected to IN for normal operation.

Use a fast comparator, Schmitt trigger, or CMOS or TTL logic to drive the \overline{OFF} pin in and out of shutdown. Rise times should be shorter than $1\mu s$. Do not use slow RC circuits, leave \overline{OFF} open, or allow the input to linger between thresholds; these measures will prevent the output from jumping to the positive supply rail in response to an indeterminate input state.

Since the \overline{OFF} threshold varies with input supply voltage (see *Electrical Characteristics*), do not derive the drive voltage from 3.3V logic. With V_{IN} at 11.5V, the high \overline{OFF} logic level needs to be above 4V.

Foldback Current Limiting

The MAX603/MAX604 also include a foldback current limiter. It monitors and controls the pass transistor's gate voltage, estimating the output current and limiting it to 1.2A

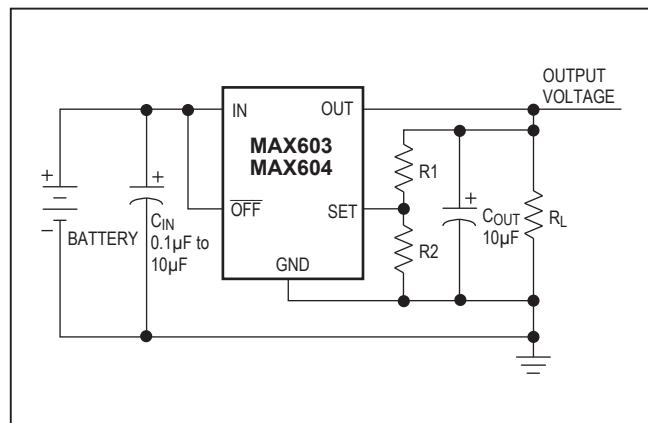


Figure 3. Adjustable Output Using External Feedback Resistors

for output voltages above 0.8V and $V_{IN} - V_{OUT} > 0.7V$. For $V_{IN} - V_{OUT} < 0.7V$ (dropout operation), there is no current limit. If the output voltage drops below 0.8V, implying a short-circuit condition, the output current is limited to 350mA. The output can be shorted to ground for one minute without damaging the device if the package can dissipate $V_{IN} \times 350mA$ without exceeding $T_J = +150^{\circ}C$.

Thermal Overload Protection

Thermal overload protection limits total power dissipation in the MAX603/MAX604. When the junction temperature exceeds $T_J = +160^{\circ}C$, the thermal sensor sends a signal to the shutdown logic, turning off the pass transistor and allowing the IC to cool. The thermal sensor will turn the pass transistor on again after the IC's junction temperature cools by $10^{\circ}C$, resulting in a pulsed output during thermal overload conditions.

Thermal overload protection is designed to protect the MAX603/MAX604 in the event of fault conditions. For continual operation, the absolute maximum junction temperature rating of $T_J = +150^{\circ}C$ should not be exceeded.

Operating Region and Power Dissipation

Maximum power dissipation of the MAX603/MAX604 depends on the thermal resistance of the case and circuit board, the temperature difference between the die junction and ambient air, and the rate of air flow. The power dissipation across the device is $P = I_{OUT} (V_{IN} - V_{OUT})$. The resulting maximum power dissipation is:

$$P_{MAX} = \left(\frac{(T_J - T_A)}{(\theta_{JB} + \theta_{BA})} \right)$$

where $(T_J - T_A)$ is the temperature difference between the MAX603/MAX604 die junction and the surrounding air, θ_{JB}

MAX603/MAX604

5V/3.3V or Adjustable, Low-Dropout, Low I_Q , 500mA Linear Regulators

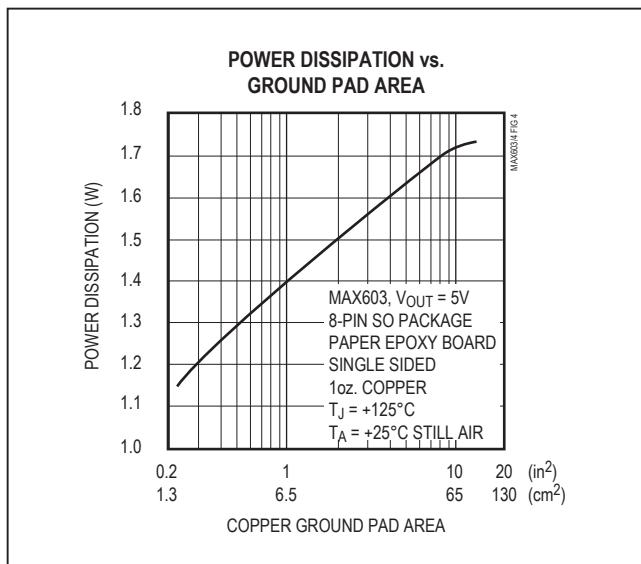


Figure 4. Typical Maximum Power Dissipation vs. Ground Pad Size.

(or θ_{JC}) is the thermal resistance of the package chosen, and θ_{BA} is the thermal resistance through the printed circuit board, copper traces and other materials to the surrounding air. The 8-pin SOIC package for the MAX603/MAX604 features a special lead frame with a lower thermal resistance and higher allowable power dissipation. The thermal resistance of this package is $\theta_{JB} = 42^\circ\text{C}/\text{W}$, compared with $\theta_{JB} = 110^\circ\text{C}/\text{W}$ for an 8-pin plastic DIP package and $\theta_{JB} = 125^\circ\text{C}/\text{W}$ for an 8-pin ceramic DIP package.

The GND pins of the MAX603/MAX604 SOIC package perform the dual function of providing an electrical connection to ground and channeling heat away. Connect all GND pins to ground using a large pad or ground plane. Where this is impossible, place a copper plane on an adjacent layer. The pad should exceed the dimensions in Figure 4.

Figure 4 assumes the IC is an 8-pin SOIC package, is soldered directly to the pad, has a $+125^\circ\text{C}$ maximum junction temperature and a $+25^\circ\text{C}$ ambient air temperature, and has no other heat sources. Use larger pad sizes for other packages, lower junction temperatures, higher ambient temperatures, or conditions where the IC is not soldered directly to the heat-sinking ground pad.

The MAX603/MAX604 can regulate currents up to 500mA and operate with input voltages up to 11.5V, but not simultaneously. High output currents can only be sustained when input-output differential voltages are low, as shown

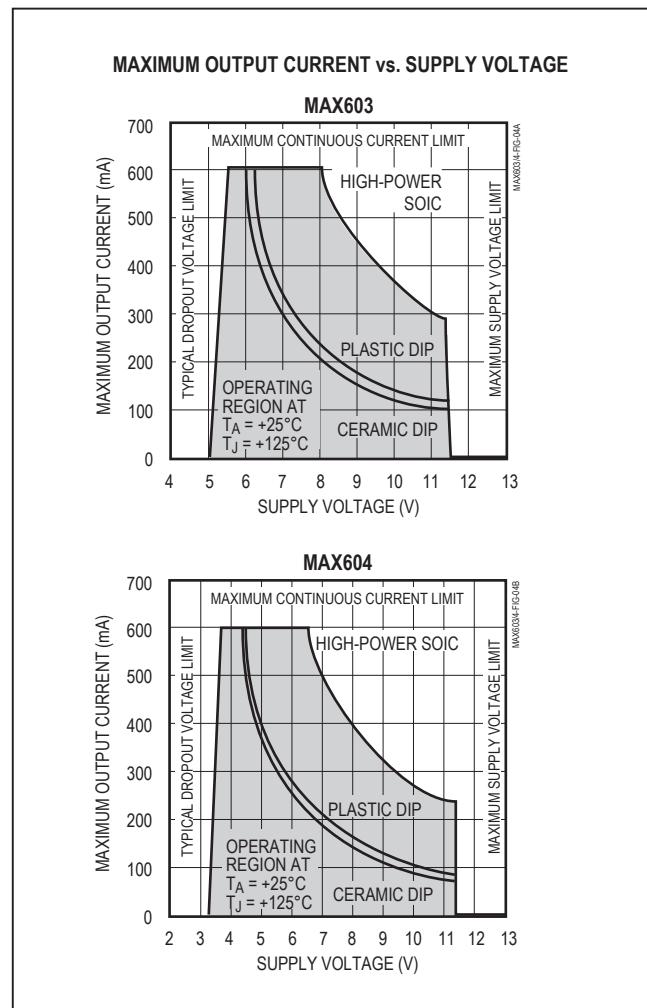


Figure 5. Power Operating Regions: Maximum Output Current vs. Differential Supply Voltage

in Figure 5. Maximum power dissipation depends on packaging, board layout, temperature, and air flow. The maximum output current is:

$$I_{\text{OUT}(\text{max})} = \frac{P_{\text{MAX}} \times (T_J - T_A)}{(V_{\text{IN}} - V_{\text{OUT}}) \times 100^\circ\text{C}}$$

where P_{MAX} is derived from Figure 4.

Reverse-Current Protection

The MAX603/MAX604 has a unique protection scheme that limits reverse currents when the input voltage falls below the output. It monitors the voltages on IN and OUT

MAX603/MAX604

5V/3.3V or Adjustable, Low-Dropout,
Low I_Q , 500mA Linear Regulators

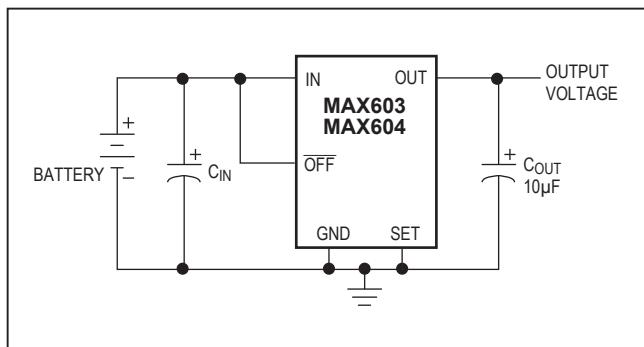


Figure 6. 3.3V or 5V Linear-Regulator Application

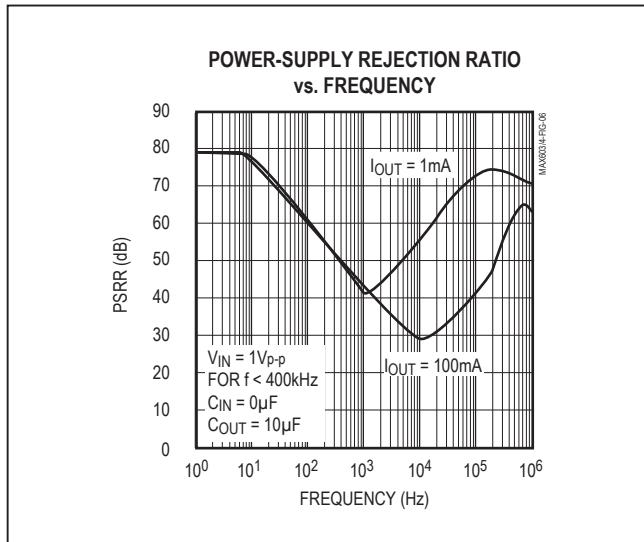


Figure 7. Power-Supply Rejection Ratio vs. Ripple Frequency

and switches the IC's substrate and power bus to the more positive of the two. The control circuitry can then remain functioning and turn the pass transistor off, limiting reverse currents back through the device. This feature allows a backup regulator or battery pack to maintain V_{OUT} when the supply at IN fails.

Reverse-current protection activates when the voltage on IN falls 6mV (20mV maximum) below the voltage on OUT. Before this happens, currents as high as several milliamperes can flow back through the device. After switchover, typical reverse currents are limited to 0.01µA for as long as the condition exists.

Applications Information

Figure 6 illustrates the typical application for the MAX603/MAX604.

Capacitor Selection and Regulator Stability

Normally, use 0.1µF to 10µF capacitors on the input and 10µF on the output of the MAX603/MAX604. The larger input capacitor values provide better supply-noise rejection and line-transient response. Improve load-transient response, stability, and power-supply rejection by using large output capacitors. For stable operation over the full temperature range and with load currents up to 500mA, 10µF is recommended. Using capacitors smaller than 3.3µF can result in oscillation.

Noise

The MAX603/MAX604 exhibit 3mV_{P-P} to 4mV_{P-P} of noise during normal operation. This is negligible in most applications. When using the MAX603/MAX604 in applications that include analog-to-digital converters of greater than 12 bits, consider the ADC's power-supply rejection specifications. Refer to the output noise plot in the *Typical Operating Characteristics*.

PSRR and Operation from Sources Other than Batteries

The MAX603/MAX604 are designed to deliver low dropout voltages and low quiescent currents in battery-powered systems. Achieving these objectives requires trading off power-supply noise rejection and swift response to supply variations and load transients. Power-supply rejection is 80dB at low frequencies and rolls off above 10Hz. As the frequency increases above 10kHz, the output capacitor is the major contributor to the rejection of power-supply noise (Figure 7). Do not use power supplies with ripple above 100kHz, especially when the ripple exceeds 100mV_{P-P}. When operating from sources other than batteries, improved supply-noise rejection and transient response can be achieved by increasing the values of the input and output capacitors, and through passive filtering techniques. The *Typical Operating Characteristics* show the MAX603/MAX604 supply and load-transient responses.

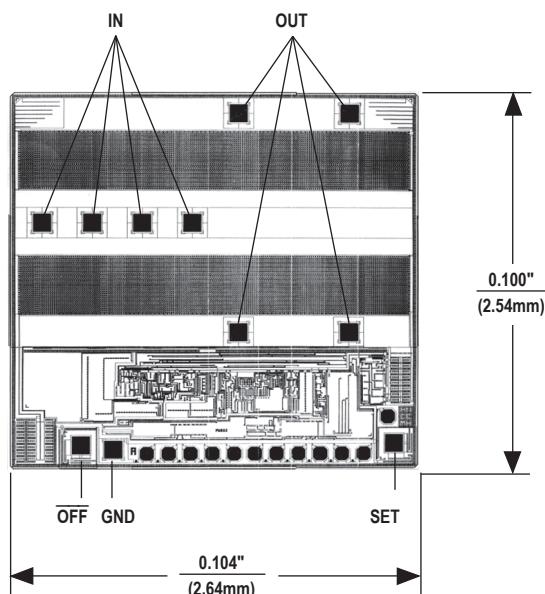
Transient Considerations

The Typical Operating Characteristics show the MAX603/MAX604 load-transient response. Two components of the output response can be observed on the load-transient graphs—a DC shift from the output impedance due to the different load currents, and the transient response. Typical transients for step changes in the load current from 5mA to 500mA are 0.2V. Increasing the output capacitor's value attenuates transient spikes.

Input-Output (Dropout) Voltage

A regulator's minimum input-output voltage differential, or dropout voltage, determines the lowest usable supply voltage. In battery-powered systems, this will determine the useful end-of-life battery voltage. Because the MAX603/MAX604 use a P-channel MOSFET pass transistor, their dropout voltage is a function of $r_{DS(ON)}$ multiplied by the load current (see *Electrical Characteristics*).

Quickly stepping up the input voltage from the dropout voltage can result in overshoot. This occurs when the pass transistor is fully on at dropout and the IC is not given time to respond to the supply voltage change. Prevent this by slowing the input voltage rise time.

Chip Topography

TRANSISTOR COUNT: 111

NO DIRECT SUBSTRATE CONNECTION. THE N-SUBSTRATE IS INTERNALLY SWITCHED BETWEEN THE MORE POSITIVE OF IN OR OUT.

Package Information

For the latest package outline information and land patterns (footprints), go to www.maximintegrated.com/packages. Note that a "+", "#", or "-" in the package code indicates RoHS status only. Package drawings may show a different suffix character, but the drawing pertains to the package regardless of RoHS status.

PACKAGE TYPE	PACKAGE CODE	OUTLINE NO.	LAND PATTERN NO.
8 PDIP	P8+3	21-0043	—
8 SOIC	S8-7F	21-0041	90-0096
8 CDIP	J8-3	—	—

Ordering Information

PART	TEMP. RANGE	PIN-PACKAGE
MAX603CPA	0°C to +70°C	8 Plastic DIP
MAX603CSA	0°C to +70°C	8 SO
MAX603C/D	0°C to +70°C	Dice*
MAX603EPA	-40°C to +85°C	8 Plastic DIP
MAX603ESA	-40°C to +85°C	8 SO
MAX603MSA/PR+T	-55°C to +125°C	8 SO
MAX604CPA	0°C to +70°C	8 Plastic DIP
MAX604CSA	0°C to +70°C	8 SO
MAX604C/D	0°C to +70°C	Dice*
MAX604EPA	-40°C to +85°C	8 Plastic DIP
MAX604ESA	-40°C to +85°C	8 SO

* Dice are tested at $T_A = +25^\circ\text{C}$, DC parameters only.

** Contact factory for availability.

Revision History

REVISION NUMBER	REVISION DATE	DESCRIPTION	PAGES CHANGED
0	9/94	Initial Release	—
1	4/17	Updated <i>Ordering Information</i> table	10

For pricing, delivery, and ordering information, please contact Maxim Direct at 1-888-629-4642, or visit Maxim Integrated's website at www.maximintegrated.com.

Maxim Integrated cannot assume responsibility for use of any circuitry other than circuitry entirely embodied in a Maxim Integrated product. No circuit patent licenses are implied. Maxim Integrated reserves the right to change the circuitry and specifications without notice at any time. The parametric values (min and max limits) shown in the Electrical Characteristics table are guaranteed. Other parametric values quoted in this data sheet are provided for guidance.



MAX32664 User Guide

UG6806; Rev 3; 8/20

Abstract

The MAX32664 user guide provides flow charts, timing diagrams, GPIOs/pin usage, I²C interface protocol, and annotated I²C traces between the host microcontroller and the MAX32664. Typical application uses the MAX32664 as a low-power microcontroller in a sensor hub configuration to provide processed data such as heart rate and SpO₂.

Table of Contents

Introduction	4
MAX32664 Variants	5
Reference Designs with the MAX32664	9
MAXREFDES220#	9
MAXREFDES101#	10
MAXREFDES103#	11
Additional Sensor Hub Products	12
MAXM86161 Integrated Optical Module for In-Ear HR and SpO ₂ Measurement	12
MAXM86146 Integrated Sensor Hub with AFE and Two Integrated Photodiodes for Wrist-Based HR and SpO ₂ Measurements	12
MAX32664 GPIOs and RSTN Pin	14
MAX32664 Bootup and Application Mode	16
MAX32664 Bootloader Mode	16
MAX32664 Application Mode	17
Communications to the MAX32664 over I ² C	18
Bit Transfer Process	18
I ² C Write	20
I ² C Read	21
MAX32664 I ² C Message Protocol Definition	22
MAX32664 I ² C Annotated Application Mode Example and Output FIFO Format	51
I ² C Commands to Flash the Application Algorithm/Firmware	51
In-Application Programming of the MAX32664	57
MAX32664 APIs and Methods for Reset, Sleep, Status, Heartbeat	58
Default Application .msbl Versions Pre-Programmed on the MAX32664A/B/C/D	59
MAX32664 Processing Capabilities	59
References	60
Trademarks	60
Revision History	61

List of Figures

Figure 1. MAXREFDES220# block diagram.....	9
Figure 2. MAXREFDES101# block diagram.....	10
Figure 3. MAXREFDES103# block diagram.....	11
Figure 4. Block diagram for Host, MAX32664C Sensor Hub, and MAXM86161.....	12
Figure 5. Block diagram for Host, MAXM86146 (Sensor Hub with Integrated AFE and Photodiodes).....	13
Figure 6. Pin connections between the host and the MAX32664.....	14
Figure 7. Host sets MFIO low to wake up the low-powered versions of the MAX32664.....	15
Figure 8. Entering bootloader mode using the RSTN pin and the MFIO GPIO pin.	17
Figure 9. Entering application mode using the RSTN pin and MFIO pin.....	17
Figure 10. I ² C Write/Read data transfer from host microcontroller.....	18
Figure 11. Using the MAX32630FTHR to flash the application .msbl to the MAX32664.	52
Figure 12. Sequence to enter bootloader mode.	53
Figure 13. Page number byte 0x44 from the .msbl file.	53
Figure 14. Initialization vector bytes 0x28 to 0x32 from the .msbl file.	53
Figure 15. Authentication bytes 0x34 to 0x43 from the .msbl file.....	53
Figure 16. Send page bytes 0x4C to 0x205B from the .msbl file.	54
Figure 17. Sequence to enter application mode.	55
Figure 18. MAX32664 in-application programming flowchart.....	57

List of Tables

Table 1. MAX32664 Variants, Matching Algorithms, and Reference Designs.....	5
Table 2. RSTN Pin and GPIOs Pins.....	14
Table 3. Additional MAX32664 GPIOs for the MAXREFDES220#.....	16
Table 4. Additional MAX32664 GPIOs or the MAXREFDES101#.....	16
Table 5. Read Status Byte Value	19
Table 6. MAX32664 I ² C Message Protocol Definitions	22
Table 7. Sensor Hub Status Byte	50
Table 8. Sequence of Commands to Write External Accelerometer Data to the Input FIFO	50
Table 9. Annotated I ² C Trace for Flashing the Application.....	53
Table 10. MAX32664 I ² C Message Protocol Definitions	58
Table 11. MAX32664A/B/C/D/MAXM86146 Pre-Programmed .msbl Version.....	59

Introduction

The MAX32664 is a pre-programmed microcontroller with firmware drivers and algorithms. Combined with the appropriate sensor devices, the MAX32664 acts as a sensor hub to provide processed data to a host device. This solution seamlessly enables customers to receive raw and/or calculated data from Maxim optical sensor solutions, while keeping overall system power consumption in check. The tiny form factor (1.6mm x 1.6mm 16-bump WLP) allows for integration into extremely small applications. The MAX32664 is integrated into Maxim's complete reference design solutions, which shortens the time to market.

The MAX32664 is the same hardware as the MAX32660 but with a pre-programmed bootloader that accepts in-application programming (IAP) of Maxim supplied algorithms and sensor drivers. The MAX32664 provides a fast-mode, I²C slave interface to a microcontroller host. A second I²C interface is dedicated to communicating with sensors.

The MAXM86146 is an additional sensor hub product that integrates two photodiodes, the MAX32664C sensor hub, and the bio-sensing analog front end (AFE) into one compact IC package.

For further details on memory, register mapping, system clocks, reset, power management, GPIOs/alternate functions, DMA controller, UART, RTC, timers, WDT, I²C, and SPI, see the MAX32660 User Guide.

For ordering information, mechanical and electrical characteristics, and the pinout for the MAX32664 family of devices, refer to the MAX32664 data sheet.

For information on the Arm® Cortex®-M4 with FPU core, refer to the Cortex-M4 with FPU Technical Reference Manual.

MAX32664 Variants

The MAX32664 is pre-programmed with bootloader software that accepts in-application programming of Maxim application code which consists of algorithms and the associated sensor driver. The MAX32664 is used as a sensor hub controller.

The algorithm/application code provides processed and/or raw data through the I²C interface. Several variants of the MAX32664 exist based on the target application. These variants come pre-programmed with a bootloader that only accepts the matching encryption keys for the part (e.g., the MAX32664A bootloader is pre-programmed with the A encryption key, reference designs are programmed with Z keying, etc.). Designers should use the table below in order to select the correctly keyed part.

Table 1. MAX32664 Variants, Matching Algorithms, and Reference Designs

MAX32664 VARIANT	APPLICATION ALGORITHM/FIRMWARE	BOOTLOADER KEY	MAXIM REFERENCE DESIGN
MAX32664A	MaximFast: Maxim Integrated finger-based heart-rate and SpO ₂ monitoring algorithm (100Hz sampling). The MaximFast algorithm is compatible with the sensor hub combination of the MAX32664A, MAX30101 AFE, and KX-122 accelerometer. It is recommended, but not mandatory, to use an accelerometer with the MaximFast algorithm. Do not enable the accelerometer if there is no accelerometer in your design. If the KX-122 accelerometer is not installed in the design and external accelerometer data is supplied, then the accelerometer should use the 100Hz sampling rate. Automatic gain control (AGC): If the AGC is enabled, the LED currents and pulse width are automatically determined by the algorithm. If the AGC is not enabled, the LED currents and pulse width registers should be configured by the host software.	A	MAXREFDES220#

MAX32664 VARIANT	APPLICATION ALGORITHM/FIRMWARE	BOOTLOADER KEY	MAXIM REFERENCE DESIGN
MAX32664B	<p>Wearable heart-rate monitoring (WHRM) algorithm: The WHRM algorithm is configured to use LED1 and photodiode (PD) 1, and it is compatible with the sensor hub combination of the MAX32664B, MAX86141 AFE, and KX-122 accelerometer. Using the KX-122 accelerometer or external accelerometer data with the WHRM algorithm is required to compensate motion artifacts. If the KX-122 accelerometer is not connected to the MAX32664, then the external accelerometer data should be supplied at the 25Hz sampling rate.</p> <p>The WHRM algorithm includes automatic exposure control (AEC) and skin control detection (SCD). If AEC is enabled, the LED current, pulse width, and sample rate are automatically determined by the algorithm. If AEC is disabled, the LED current, LED current range, pulse width, and ADC range registers are set to default and can be updated by the host software.</p>	B	MAXREFDES101#

MAX32664 VARIANT	APPLICATION ALGORITHM/FIRMWARE	BOOTLOADER KEY	MAXIM REFERENCE DESIGN
MAX32664C MAX32664C MAXM86146	<p>Wearable heart-rate monitoring and wearable oxygen saturation (WHRM+WSPO₂) algorithm version 3x.xx.x, where xx is 3 or greater: The wearable algorithm suite can monitor heart rate and SpO₂ simultaneously. It is configured to use LED1 (green), LED2 (IR), LED3 (red), and photodiode 1 and 2, and it is compatible with the sensor hub combination of the MAX32664C, MAX86141 (or MAXM86161/MAXM86146) AFE, and KX-122 accelerometer. Using the KX-122 accelerometer or external accelerometer data with the WHRM algorithm is required to compensate for motion artifacts. If the KX-122 accelerometer is not connected to the MAX32664, then the external accelerometer data should be supplied at the 25Hz sampling rate.</p> <p>The wearable algorithm suite includes automatic exposure control (AEC) and skin control detection (SCD). If AEC is enabled, the LED currents, pulse width, and sample rate are automatically determined by the algorithm. The AEC algorithm adjusts averaging and sample rates for an effective rate of 25Hz. If AEC is not enabled, the rates are set to default and can be updated by the host software.</p> <p>Low power mode is enabled in the firmware. Normally, when the MAX32664 is idle, it switches to the Deep Sleep state to save power. An external interrupt like a sensor, the host MFIO, or RTC alarm forces the MAX32664 to wake up.</p>	C C C	MAXREFDES103# MAXM86161EVSYS# MAXM86146EVSYS#

MAX32664 VARIANT	APPLICATION ALGORITHM/FIRMWARE	BOOTLOADER KEY	MAXIM REFERENCE DESIGN
MAX32664D	<p>Finger-based blood pressure trending (BPT), heart-rate, and SpO₂ monitoring algorithm (100Hz Sampling). The algorithm is compatible with the sensor hub combination of the MAX32664D and MAX30101 AFE. No accelerometer is required for this algorithm.</p> <p>The BPT algorithm includes automatic gain control to adjust the LED currents. Prior to running the algorithm, a calibration procedure is required to determine blood pressure and SpO₂ calibration coefficients.</p> <p>Automatic gain control (AGC). If the AGC is enabled, the LED currents and pulse width are automatically determined by the algorithm. If the AGC is not enabled, the LED currents and pulse width registers should be configured by the host software.</p>	D	MAXREFDES220#
MAX32664Z	The algorithms listed in this table have a corresponding algorithm/application Z-keyed .msbl file, which can be flashed to the MAX32664Z using in-application programming on the MAX32664Z.		MAXREFDES220# MAXREFDES101# MAXREFDES103#

For all the MAX32664 parts, the latest algorithm (.msbl file) with the corresponding bootloader key must be downloaded, and these parts must be programmed using the in-application programming feature of the bootloader.

Reference Designs with the MAX32664

Maxim provides multiple reference designs to its customers to enable quick and effective adoption of the MAX32664 and fastest time to market. For detailed schematics, refer to the user guide of each reference design.

MAXREFDES220#

The MAXREFDES220# reference design provides everything you need to quickly prototype your product to measure finger-based heart rate and blood oxygen saturation level (SpO_2).

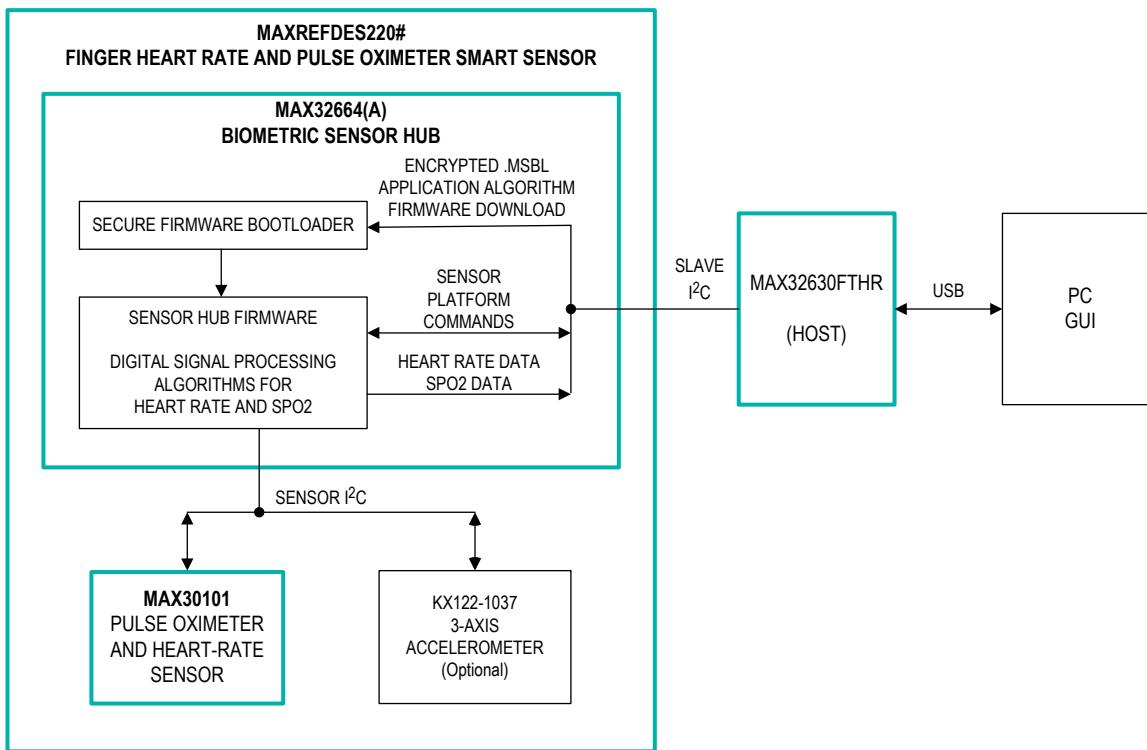


Figure 1. MAXREFDES220# block diagram.

The MAXREFDES220# solution, which includes the MAX30101 and the MAX32664, provides an integrated hardware and software solution for finger-based applications. The MAX32664 is used as a sensor hub to collect data from the MAX30101 analog front end (AFE). The reference design also includes a tri-axis accelerometer (KX-122) to detect motion artifacts. (Accelerometer support in the MAXREFDES220# is optional.)

The MAX32630FTHR is used as a sample host is included in MAXREFDES220# reference design.

MAXREFDES101#

The MAXREFDES101# is a unique evaluation and development platform in a wrist-worn wearable form factor that demonstrates the functions of a wide range of Maxim's products for health-sensing applications.

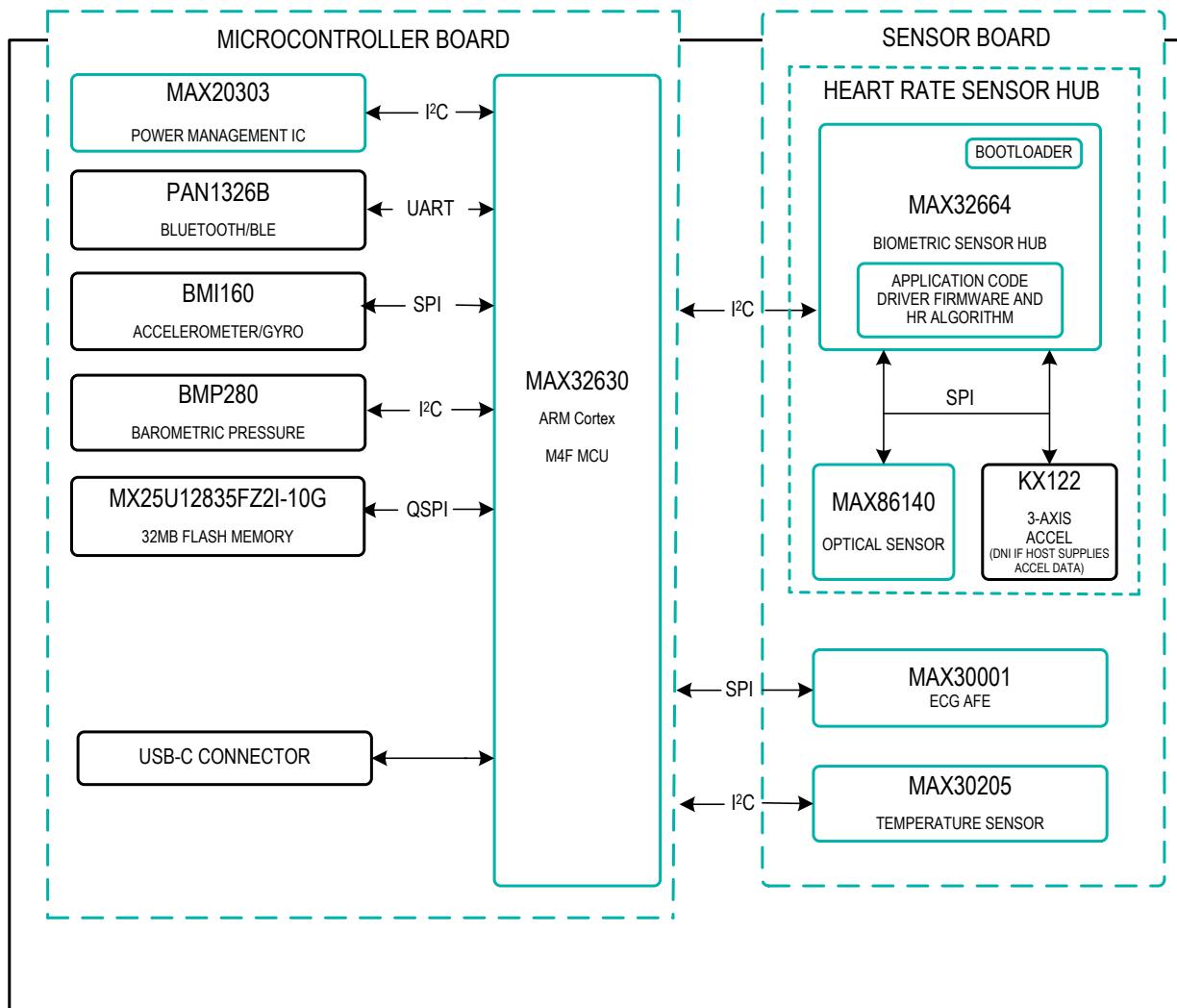


Figure 2. MAXREFDES101# block diagram.

This second-generation health sensor platform (a follow-on to the MAXREFDES100#) integrates a PPG AFE sensor (MAX86141), a biopotential AFE (MAX30001), a human body temperature sensor (MAX30205), a microcontroller (MAX32630), a power-management IC (MAX20303), and a 6-axis accelerometer/gyroscope. The complete platform includes a watch enclosure and a biometric sensor hub with an embedded application code for heart-rate algorithm and AFE drivers (MAX32664). Algorithm output and sensor data can be streamed through Bluetooth® to an Android® application or PC GUI for demonstration, evaluation, and customized development.

MAXREFDES103#

The MAXREFDES103# is a wrist-worn wearable form factor that demonstrates the high-sensitivity and algorithm processing functions for health-sensing applications. This health sensor band platform includes an enclosure and a biometric sensor hub with an embedded algorithm for heart rate and SpO₂ (MAX32664C) which processes PPG signals from the analog-front-end (AFE) sensor (MAX86141). Algorithm output and raw data can be streamed through Bluetooth® to an Android® app or PC GUI for demonstration, evaluation, and customized development.

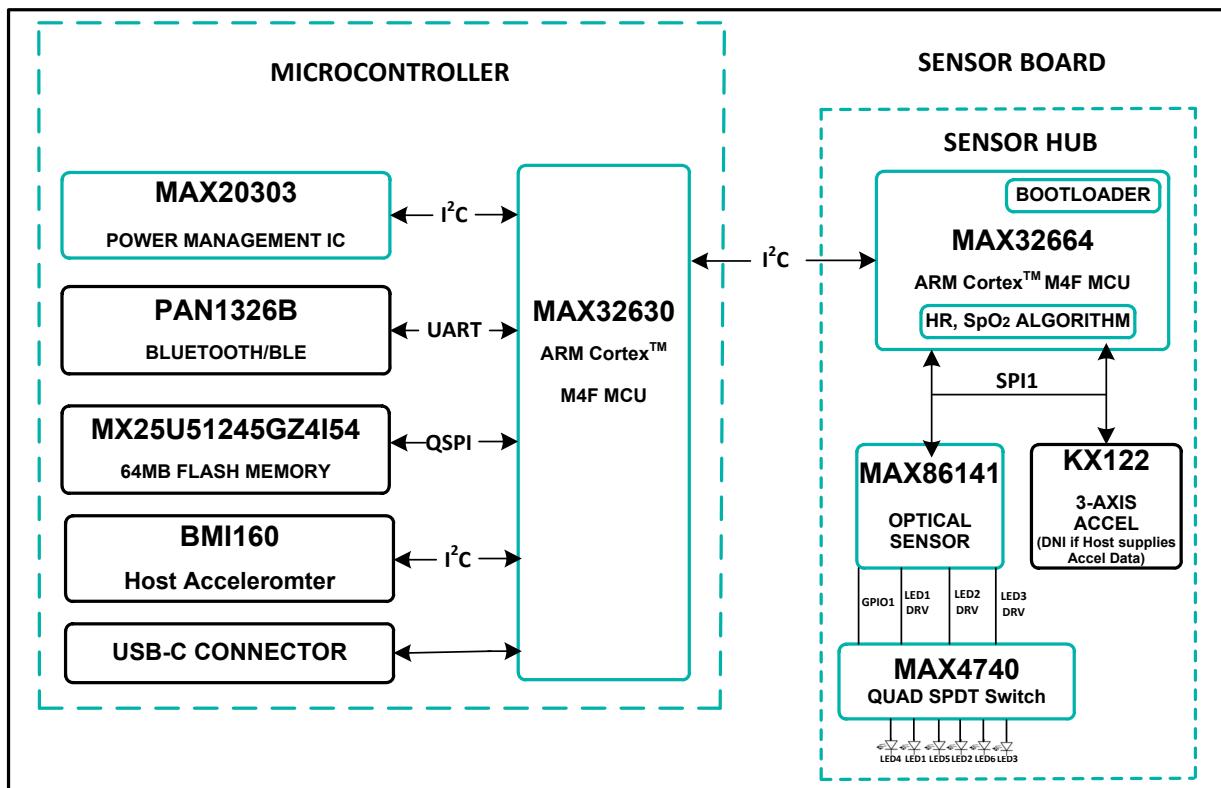


Figure 3. MAXREFDES103# block diagram.

Additional Sensor Hub Products

MAXM86161 Integrated Optical Module for In-Ear HR and SpO₂ Measurement

The MAXM86161 is an ultra-low-power, completely integrated, optical data-acquisition system ideally suited for in-ear products. On the transmitter side, the MAXM86161 has three programmable high-current LED drivers. On the receiver side, MAXM86161 consists of a high-efficiency PIN photo-diode and an optical readout channel. The optical readout has a low-noise signal conditioning analog front-end (AFE), including 19-bit ADC, an industry-lead ambient light cancellation (ALC) circuit, and a picket fence detect-and-replace algorithm. Due to the low power consumption, compact size, easy, flexible-to-use, and industry-lead ambient light rejection capability of the MAXM86161, the device is ideal for a wide variety of optical sensing applications such as heart-rate detection and pulse oximetry.

MAXM86161 MAX32664C application .msbl are versioned as 32.x.x. If using the MAXM86161EVSYS#, the MAX32664 must be flashed with the compatible .msbl application file that matches the Nordic binary.

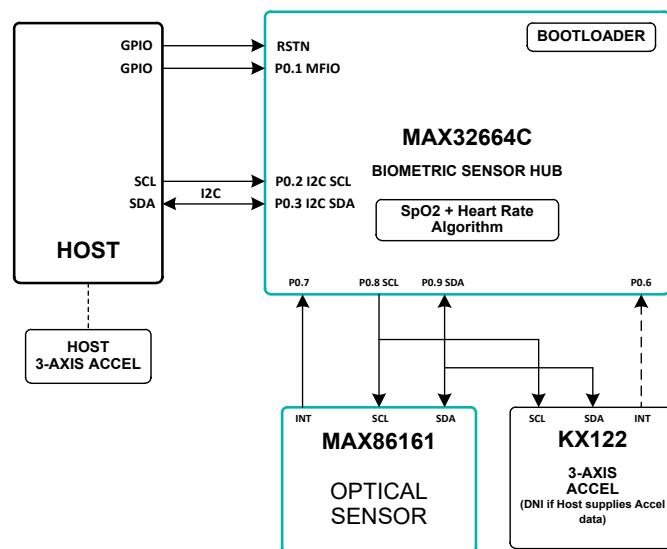


Figure 4. Block diagram for Host, MAX32664C Sensor Hub, and MAXM86161.

MAXM86146 Integrated Sensor Hub with AFE and Two Integrated Photodiodes for Wrist-Based HR and SpO₂ Measurements

The MAXM86146 is an ultra-low-power, completely integrated, optical data acquisition system specifically designed for battery-powered devices and wireless sensors. It combines Maxim's best in class optical bio-sensing analog front end (AFE) with the powerful Arm Cortex-M4 sensor hub microcontroller and two high sensitivity photo diodes, all in a compact 4.5mm x 4.1mm x 0.88mm, 38-pin OLGA package with commercial operating temperature range of 0°C to +70°C. The AFE has two, low-noise, optical readout channels. Both channels have independent 19-bit ADCs, industry leading ambient-light cancellation (ALC) circuit, and a picket fence detect-and-replace algorithm. The AFE includes three programmable high-current LED drivers and operates on a 1.8V main supply voltage and a 3.1V-5.5V LED driver supply voltage.

The sensor hub MCU within the MAXM86146 is factory programmed with the sensor hub bootloader; the application algorithm .msbl is not included in the factory programming. The latest

MAX32664C MAXM86146 application algorithm .msbl from the MAX32664 website must be flashed to the MAXM86146 MCU.

MAXM86146 MAX32664C application .msbl are versioned as 33.x.x. If using the MAXM86146EVSYS#, the MAX32664 must be flashed with the compatible .msbl application file that matches the Nordic binary.

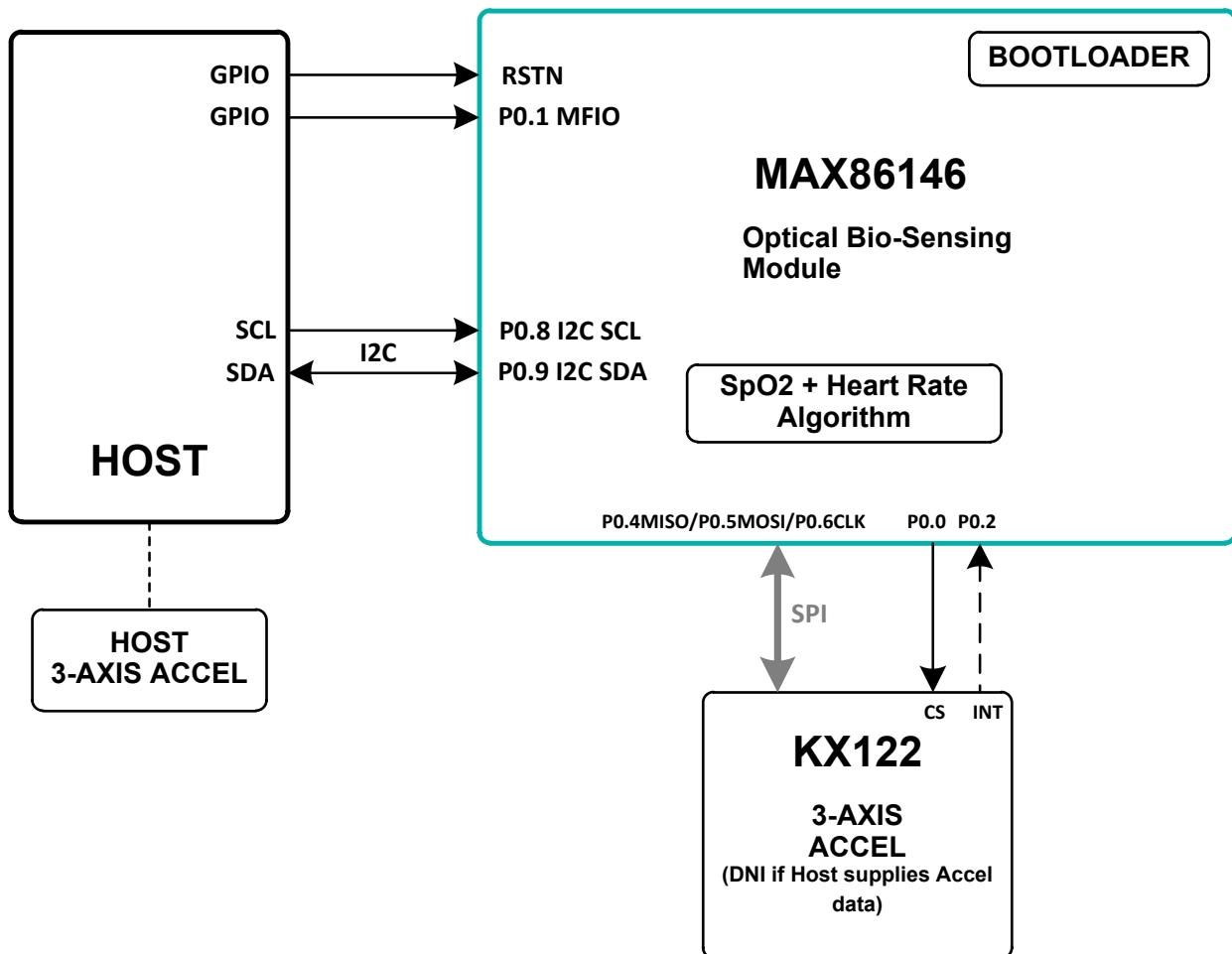


Figure 5. Block diagram for Host, MAXM86146 (Sensor Hub with Integrated AFE and Photodiodes).

MAX32664 GPIOs and RSTN Pin

To control and communicate with the MAX32664, the RSTN pin and GPIOs P0.1, P0.2, P0.3 of the MAX32664 are connected to the host as pictured in Figure 6.

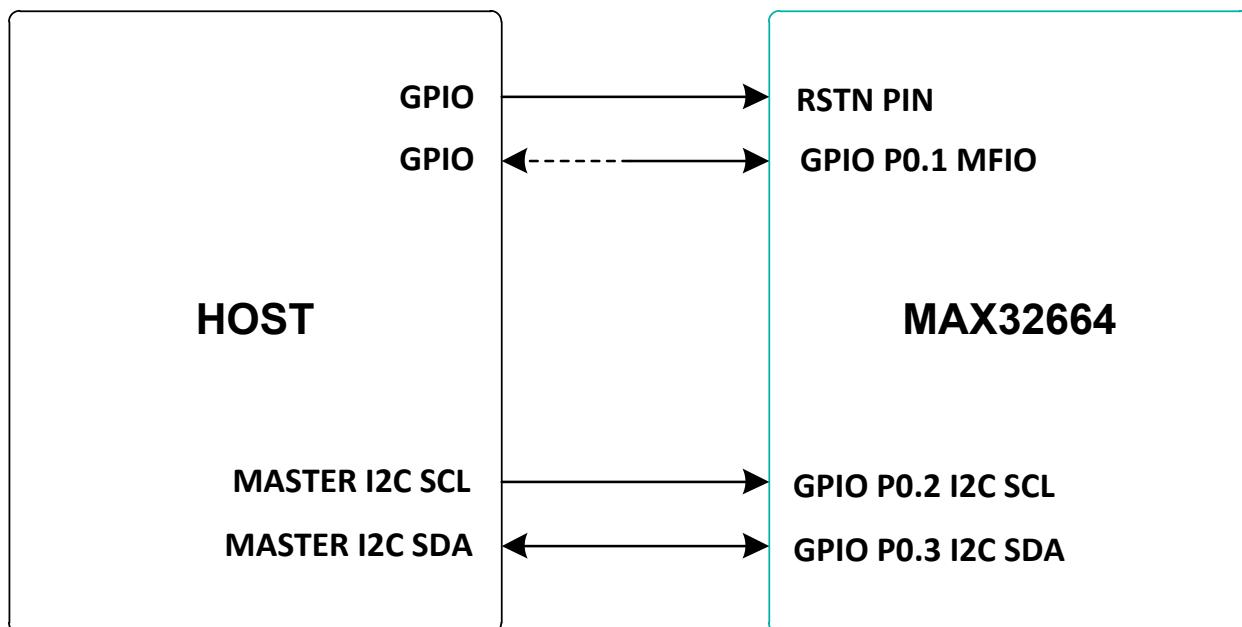


Figure 6. Pin connections between the host and the MAX32664.

The RSTN pin is used in conjunction with the GPIO P0.1 MFIO pin to control whether the MAX32664 starts up in Application mode or Bootloader mode. While in application mode, the MFIO pin is configured to provide an interrupt signal to the host, or the host can use it to wake the MAX32664 when using the low-powered firmware.

The host acts an I²C master to communicate with the MAX32664. GPIO P0.2 is used as the SCL line and GPIO P0.3 is used as the SDA line.

Table 2. RSTN Pin and GPIOs Pins

MAX32664	DESCRIPTION	DIRECTION FROM THE MAX32664 SIDE
Pin RSTN	Reset_N	Input
GPIO P0.1	GPIO MFIO interrupt to host, wake from host, bootloader/application on power up. Interrupt to host is not used on the following: <ul style="list-style-type: none">● MAX32664B WHRM v20.2.0+● MAX32664C WHRM+WSpO₂ v30.2.4+, v32.1.2+, v33.6.0+	Input/Output Input only for the following: <ul style="list-style-type: none">● MAX32664B WHRM v20.2.0+● MAX32664C WHRM+WSpO₂ v30.2.4+, v32.1.2+, v33.6.0+
GPIO P0.2	I ² C0 Host SCL	Input
GPIO P0.3	I ² C0 Host SDA	Input/Output

To achieve a lower power profile, the following versions of the .msbl algorithm use a polling method instead of the MFIO pin as an interrupt to the host:

- MAX32664B WHRM v20.2.0+
- MAX32664C WHRM WHRM+WSpO₂ v30.2.4+, v32.1.2+, v33.6.0+

For these versions of the algorithm, the MAX32664B/C switches to “Deep Sleep” state to save power. The MAX32664B/C can be woken from deep sleep by the internal RTC, the connected sensor, or the MFIO pin. The host is required to wake up the MAX32664B/C prior to any I²C communication by performing the following:

- Setting MFIO to low at least 250μsec before the beginning of an I²C communication to make sure the MAX32664B/C is awake
- Keeping MFIO low until the end of the I²C communication to make sure the MAX32664B/C does not switch to “Deep Sleep” state
- Setting MFIO to high after the end of I²C communication to allow the MAX32664B/C to switch back to “Deep Sleep” state

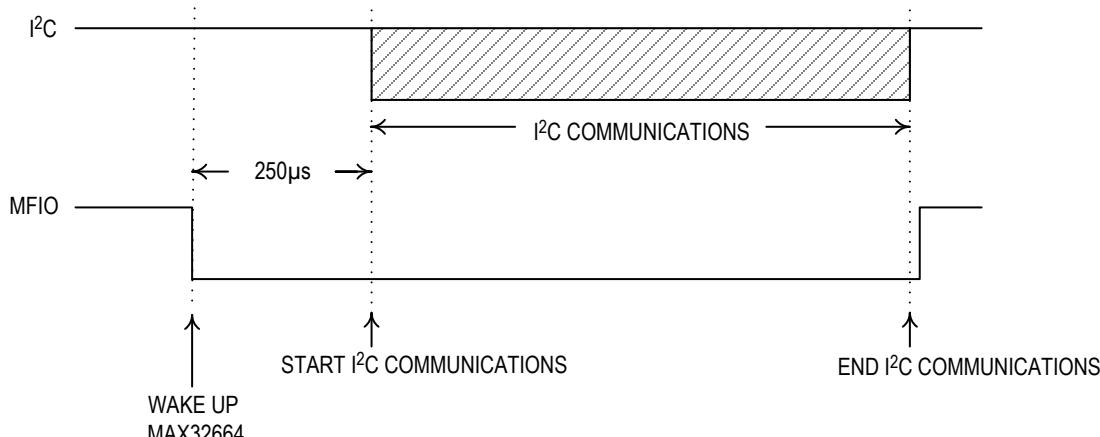


Figure 7. Host sets MFIO low to wake up the low-powered versions of the MAX32664.

For the WHRM (v20.2.0+) and the WHRM+WSpO₂ (v30.2.4+, v32.1.2+, v33.6.0+), the host is required to regularly poll the MAX32664B/C to read the measurement data. The host is required to regularly empty the measurement data in the MAX32664B/C FIFO at a periodic rate. The periodic rate depends on the rate that the MAX32664B/C samples report is generated. By reducing the samples report period, the FIFO does not need to be emptied as often.

The host can read samples in the output FIFO at a period (host reading FIFO period) five times the length of the samples report period to avoid FIFO overflow. In this example, an average of five samples is in the output FIFO.

By default, the samples report period (read samples report period, 0x11 0x02) is set to 40ms. In this case, it is recommended that the host read samples from the output FIFO every 200ms (host reading period). At these rates, on average there will be five samples in the output FIFO for the host to read.

Variations of the MAX32664 use additional GPIO pins in order to communicate and control sensor devices. For example, in the MAXREFDES220#, the additional GPIOs listed in Table 3 are used to control the sensors used.

Table 3. Additional MAX32664 GPIOs for the MAXREFDES220#

MAX32664	DESCRIPTION	DIRECTION FROM THE MAX32664 SIDE
GPIO P0.6	KX122 ACCEL Interrupt	Input
GPIO P0.7	MAX30101 Interrupt	Input
GPIO P0.8	MAX30101, KX122 I2C1_SCL	Output
GPIO P0.9	MAX30101, KX122 I2C1_SDA	Input/Output

Table 4. Additional MAX32664 GPIOs or the MAXREFDES101#

MAX32664	DESCRIPTION	DIRECTION FROM THE MAX32664 SIDE
GPIO P0.0	KX122 ACCEL Select	Output
GPIO P0.4	SPI MISO: MAX86141, KX122	Input
GPIO P0.5	SPI MOSI: MAX86141, KX122	Output
GPIO P0.6	SPI CLK: MAX86141, KX122	Output
GPIO P0.7	MAX86141 Select	Output
GPIO P0.8	MAX86141 Interrupt	Input
GPIO P0.9	KX122 Interrupt (N/A for polling versions 30.2.3+ for the MAX32664C and v20.2.x+ for the MAX32664B)	Input

MAX32664 Bootup and Application Mode

The MAX32664 is programmed to enter either bootloader mode or application mode at the start-up based on the state of the MFIO pin.

Variations of the MAX32664 part are pre-programmed with the different algorithms and application firmware. Table 11 details the applications firmware that are pre-programmed. It is strongly recommended that the application firmware be updated to the latest version.

MAX32664 Bootloader Mode

The MAX32664 enters bootloader mode based on the sequencing of the RSTN pin and the MFIO pin. The necessary sequence is as follows:

- Set the RSTN pin low for 10ms.
- While RSTN is low, set the MFIO pin to low (MFIO pin should be set low at least 1ms before RSTN pin is set high.)
- After the 10ms has elapsed, set the RSTN pin high.
- After an additional 50ms has elapsed, the MAX32664 is in bootloader mode.
- If the enter bootloader mode command, 0x01 0x00 0x08, is not received within the first approximately 780ms and there is a valid .msbl application that has been flashed to the MAX32664, then the mode changes to the application mode automatically.

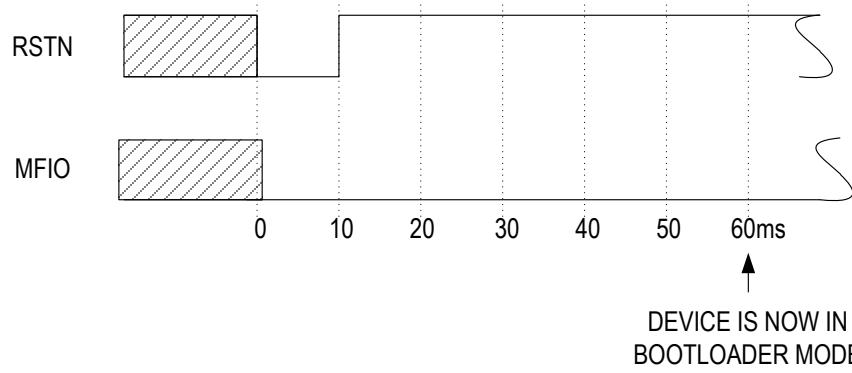


Figure 8. Entering bootloader mode using the RSTN pin and the MFIO GPIO pin.

MAX32664 Application Mode

The MAX32664 enters application mode based on the sequencing of the RSTN pin and the MFIO pin. The necessary sequence is as follows:

- Set the RSTN pin low for 10ms.
- While RSTN is low, set the MFIO pin to high.
- After the 10ms has elapsed, set the RSTN pin high. (MFIO pin should be set high at least 1ms before RSTN pin is set high.)
- After an additional 50ms has elapsed, the MAX32664 is in application mode and the application performs its initialization of the application software.
- Approximately 1.5 second after the RSTN is set to high, the application completes the initialization and the device is ready to accept I²C commands. (For MAX32664A and MAX32664D, the startup time is 1.0 second).

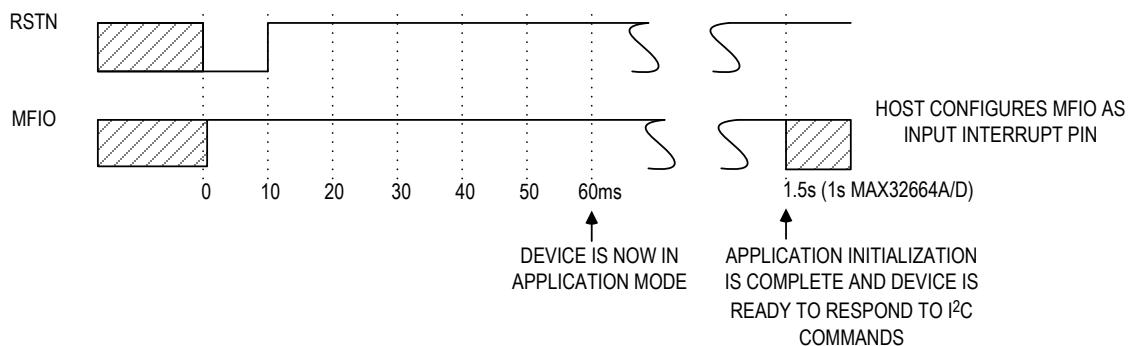


Figure 9. Entering application mode using the RSTN pin and MFIO pin.

Communications to the MAX32664 over I²C

The host communicates to the MAX32664 through the I²C bus. The MAX32664 uses 0xAA as the I²C 8-bit slave write address and 0xAB is used as the I²C 8-bit slave read address. The maximum I²C data rate supported is 3400Kbps.

Bit Transfer Process

The defined bit transfer process is described below. It is recommended that I²C GPIO 'bit-bang' software be implemented on the host if the host MCU I²C hardware/HAL is not compatible with sensor hub protocol.

Both SDA and SCL signals are open-drain circuits. Each has an external pullup resistor that ensures each circuit is high when idle. The I²C specification states that during data transfer, the SDA line can change state only when SCL is low, and that SDA is stable and able to be read when SCL is high. Typical I²C write/read transactions are shown in Figure 10.

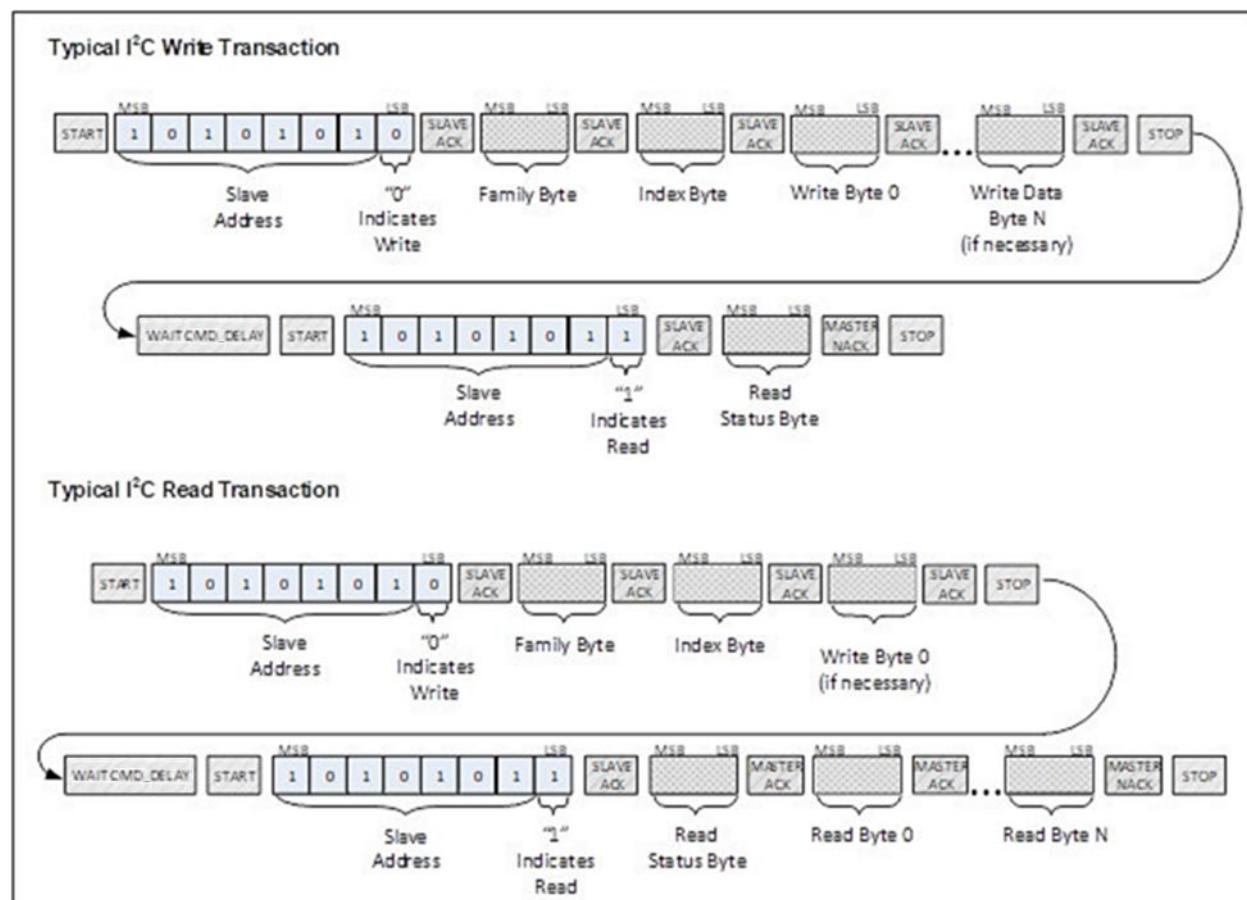


Figure 10. I²C Write/Read data transfer from host microcontroller.

The read status byte is an indicator of the success or failure of the Write Transaction. The read status byte must be accessed after each write transaction to the device. This ensures that write transaction processing is understood and any errors in the device command handling can be corrected. The value of the read status byte is summarized in Table 5.

Table 5. Read Status Byte Value

STATUS BYTE VALUE	DESCRIPTION
0x00	SUCCESS. The write transaction was successful.
0x01	ERR_UNAVAIL_CMD. Illegal Family Byte and/or Index Byte was used. Verify that the Family Byte, Index Byte are valid for the host command sent. Verify that the latest .msbl is flashed.
0x02	ERR_UNAVAIL_FUNC. This function is not implemented. Verify that the Index Byte and Write Byte(s) are valid for the host command sent. Verify that the latest .msbl is flashed.
0x03	ERR_DATA_FORMAT. Incorrect number of bytes sent for the requested Family Byte. Verify that the correct number of bytes are sent for the host command. Verify that the latest .msbl is flashed.
0x04	ERR_INPUT_VALUE. Illegal configuration value was attempted to be set. Verify that the Index Byte is correct for Family Byte 0x44. Verify that the report period is not 0 for host command 0x10 0x02. Verify that the Write byte for host command 0x10 0x03 is in the valid range specified. Verify that the latest .msbl is flashed.
0x05	Application mode: ERR_INVALID_MODE. Not used in application mode. Bootloader mode: ERR_BTLDTRY AGAIN. Device is busy. Insert delay and resend the host command.
0x80	ERR_BTLDGENERAL. General error while receiving/flashing a page during the bootloader sequence. Not used.
0x81	ERR_BTLDRCHECKSUM. Bootloader checksum error while decrypting/checking page data. Verify that the keyed .msbl file is compatible with MAX32664A/B/C/D.
0x82	ERR_BTLDRAUTH. Bootloader authorization error. Verify that the keyed .msbl file is compatible with MAX32664A/B/C/D.
0x83	ERR_BTLDINVALIDAPP. Bootloader detected that the application is not valid.
0xFE	ERRTRY AGAIN. Device is busy, try again. Increase the delay before the command and increase the CMD_DELAY.
0xFF	ERRUNKNOWN. Unknown Error. Verify that the communications to the AFE/KX-122 are correct by reading the PART_ID/WHO_AM_I register. For MAX32664B/C, the MAX32664 is in deep sleep unless the host sets the MFIO pin low 250µs before and during the I2C communications.

I²C Write

The process for an I²C write data transfer is as follows:

1. The bus master indicates a data transfer to the device with a START condition.
2. The master transmits one byte with the 7-bit slave address (most significant 7 bits of the 8-bit address) and a single write bit set to zero. The eight bits to be transferred as a slave address for the MAX32664 is 0xAA for a write transaction.
3. During the next SCL clock following the write bit, the master releases SDA. During this clock period, the device responds with an ACK by pulling SDA low.
4. The master senses the ACK condition and begins to transfer the Family Byte. The master drives data on the SDA circuit for each of the eight bits of the Family byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
5. The master senses the ACK condition and begins to transfer the Index Byte. The master drives data on the SDA circuit for each of the eight bits of the Index byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
6. The master senses the ACK condition and begins to transfer the Write Data Byte 0. The master drives data on the SDA circuit for each of the eight bits of the Write Data Byte 0, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
7. The master senses the ACK condition and can begin to transfer another Write Data Byte if required. The master drives data on the SDA circuit for each of the eight bits of the Write Data Byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication. If another Write Data Byte is not required, the master indicates the transfer is complete by generating a STOP condition. A STOP condition is generated when the master pulls SDA from a low to high while SCL is high.
8. The master waits for a period of CMD_DELAY (2ms is the default) for the device to have its data ready.
9. The master indicates a data transfer to a slave with a START condition.
10. The master transmits one byte with the 7-bit slave address and a single write bit set to one. This is an indication from the master of its intent to read the device from the previously written location defined by the Family Byte and the Index Byte. The master then floats SDA and allows the device to drive SDA to send the Status Byte. The Status Byte reveals the success of the previous write sequence. After the Status Byte is read, the master drives SDA low to signal the end of data to the device.
11. The master indicates the transfer is complete by generating a STOP condition.
12. After the completion of the write data transfer, the Status Byte must be analyzed to determine if the write sequence was successful and the device has received the intended command.

I²C Read

The process for an I²C read data transfer is as follows:

1. The bus master indicates a data transfer to the device with a START condition.
2. The master transmits one byte with the 7-bit slave address and a single write bit set to zero. The eight bits to be transferred as a slave address for the MAX32664 is 0xAA for a write transaction. This write transaction precedes the actual read transaction to indicate to the device what section is to be read.
3. During the next SCL clock following the write bit, the master releases SDA. During this clock period, the device responds with an ACK by pulling SDA low.
4. The master senses the ACK condition and begins to transfer the Family Byte. The master drives data on the SDA circuit for each of the eight bits of the Family byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
5. The master senses the ACK condition and begins to transfer the Index Byte. The master drives data on the SDA circuit for each of the eight bits of the Index byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
6. The master senses the ACK condition and begins to transfer the Write Data Byte if necessary for the read instruction. The master drives data on the SDA circuit for each of the eight bits of the Write Data byte, and then floats SDA during the ninth bit to allow the device to reply with the ACK indication.
7. The master indicates the transfer is complete by generating a STOP condition.
8. The master waits for a period of CMD_DELAY (2ms is the default) for the device to have its data ready.
9. The master indicates a data transfer to a slave with a START condition.
10. The master transmits one byte with the 7-bit slave address and a single write bit set to one. This is an indication from the master of its intent to read the device from the previously written location defined by the Family Byte and the Index Byte. The master then floats SDA and allows the device to drive SDA to send the Status Byte. The Status Byte reveals the success of the previous write sequence. After the Status Byte is read, the master drives SDA low to acknowledge the byte.
11. The master floats SDA and allows the device to drive SDA to send Read Data Byte 0. After Read Data Byte 0 is read, the master drives SDA low to acknowledge the byte.
12. The master floats SDA and allows the device to drive SDA to send the Read Data Byte N. After Read Data Byte N is read, the master drives SDA low to acknowledge the Read Data Byte N. This process continues until the device has provided all the data that the master expects based upon the Family Byte and Index Byte definition.
13. The master indicates the transfer is complete by generating a STOP condition.

MAX32664 I²C Message Protocol Definition

Table 6 defines the I²C message protocol for the MAX32664.

Table 6. MAX32664 I²C Message Protocol Definitions

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Read Sensor Hub Status	Read sensor hub status (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x00	0x00	-	Err0[0] : 0 = No Error; 1 = Sensor Communication Problem Err1[0] : Not used Err2[0] : Not used DataRdyInt[3] : 0 = FIFO below threshold; 1 = FIFO filled to threshold or above. FifoOutOvrlnt[4] : 0 = No FIFO overflow; 1 = Sensor Hub Output FIFO overflowed, data lost. FifoInOvrlnt[5] : 0 = No FIFO overflow; 1 = Sensor Hub Input FIFO overflowed, data lost. HostAccelUflnt[6] : 0 = No underflow; 1 = Host data to input FIFO is slow and the input FIFO has underflowed. See Table 7 for the bit field table.
Device Mode	Select the device operating mode. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x01	0x00	0x00 : Exit bootloader mode, enter application mode. 0x01 : Shutdown the MAX32664B/C. Restart by power cycling or pulsing RSTN. 0x02 : Reset. 0x08 : Enter bootloader mode.	-
Device Mode	Read the device operating mode. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x02	0x00	-	0x00 : Application operating mode. 0x02 : Reset. 0x08 : Bootloader operating mode.

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Set Output Mode	Set the output format of the sensor hub. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x10	0x00	0x00 : Pause (no data) 0x01 : Sensor Data 0x02 : Algorithm Data 0x03 : Sensor Data and Algorithm Data 0x04 : Pause (no data) 0x05 : Sample Counter byte, Sensor Data 0x06 : Sample Counter byte, Algorithm Data 0x07 : Sample Counter byte, Sensor Data and Algorithm Data	-
Set Output Mode	Set the threshold for the FIFO interrupt bit/pin. The MFIO pin is used as the interrupt and the host should configure this pin as an input interrupt pin. The status bit DataRdyInt is set when this threshold is reached. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x10	0x01	0x01 to 0xFF : Sensor Hub Interrupt Threshold for FIFO.	-
Set Output Mode	Set the samples report period (e.g., a value of 25 means a samples report is generated once every 25 samples). (MAX32664C)	0x10	0x02	0x01 to 0xFF : LSB is 40ms. N, where a samples report is generated once every N samples.	-
Set Output Mode	Change I ² C address of the MAX32664. (MAX32664B, MAX32664C)	0x10	0x03	0x02 to 0xFF : New I ² C address (8-bit I ² C write address)	
Set Output Mode	Set the sensor hub counter. (MAX32664B, MAX32664C)	0x10	0x04	0x00 to 0xFF : Counter	

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Read Output Mode	Read the output format of the sensor hub. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x11	0x00	-	0x00: Pause (no data) 0x01: Sensor Data 0x02: Algorithm Data 0x03: Sensor Data and Algorithm Data 0x04: Pause (no data) 0x05: Sample Counter byte, Sensor Data 0x06: Sample Counter byte, Algorithm Data 0x07: Sample Counter byte, Sensor Data, and Algorithm Data
Read Output Mode	Read the threshold for the FIFO interrupt bit/pin. The MFIO pin is used as the interrupt and the host should configure this pin as an input interrupt pin. The status bit DataRdyInt is set when this threshold is reached. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x11	0x01	-	0x01 to 0xFF: Sensor Hub Interrupt Threshold for FIFO.
Read Output Mode	Read the samples reporting period (e.g., a value of 25 means a report is generated once every 1s. The default of 1 is one report is generated once per sample or every 40ms). (MAX32664C)	0x11	0x02	-	0x01 (default) to 0xFF: LSB is 40ms. N, where a samples report is generated once every N samples.
Read Output Mode	Read the I ² C address of the MAX32664. (MAX32664B, MAX32664C)	0x11	0x03		0x00 to 0xFF: I ² C address
Read Output Mode	Read the sensor hub counter. (MAX32664B, MAX32664C)	0x11	0x04		0x00 to 0xFF: Counter

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Read Output FIFO	Get the number of samples available in the FIFO. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x12	0x00	-	Number of samples available in the FIFO.
Read Output FIFO	Read data stored in output FIFO. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x12	0x01	-	See Table 8, Output FIFO Format Definitions. The internal FIFO read pointer increments once the sample size bytes have been read.
Read Input FIFO for External Sensors ¹	Read the sensor sample size. (MAX32664A, MAX32664B, MAX32664C)	0x13	0x00	0x04 : Accelerometer	0x06 : Bytes per sample for the external accelerometer. Three 16-bit 2's complement with LSB = 0.001g. See Table 9 for an example.
Read Input FIFO for External Sensors	Read the input FIFO size for the maximum number of samples that the input FIFO can hold (16-bit). (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x13	0x01	-	MSB, LSB
Read Input FIFO for External Sensors	Read the sensor FIFO size for the maximum number of samples that the sensor FIFO can hold (16-bit). (MAX32664A, MAX32664B, MAX32664C)	0x13	0x02	0x04 : Accelerometer	MSB, LSB
Read Input FIFO for External Sensors	Read the number of samples currently in the input FIFO (16-bit). (MAX32664A, MAX32664B, MAX32664C)	0x13	0x03	0x04 : Accelerometer	MSB, LSB

¹ Systems that have an externally supplied accelerometer.

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Read Input FIFO for External Sensors	Read the number of samples currently in the sensor FIFO (16-bit). (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x13	0x04	-	MSB, LSB
Write Input FIFO for External Sensors	Write data to the input FIFO. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x14	0x00	Sample one value, ..., Sample N values See Table 9 for an example.	-
Write Register	Write a value to a writable MAX86140/ MAX86141/ MAXM86161 register. (MAX32664B, MAX32664C)	0x40	0x00	Register address, Register value	-
Write Register	Write a value to a writable MAX30205 register. (MAX32664B)	0x40	0x01	Register address, Register value	-
Write Register	Write a value to a writable MAX30001 register. (MAX32664B)	0x40	0x02	Register address, Register value	-
Write Register	Write a value to a writable MAX30101/ MAX30102 register. (MAX32664A, MAX32664D)	0x40	0x03	Register address, Register value	-
Write Register	Write a value to a writable accelerometer sensor register. (MAX32664A, MAX32664B, MAX32664C)	0x40	0x04	Register address, Register value	-
Read Register	Read the value of a MAX86140/ MAX86141/ MAXM86161 register. (MAX32664B, MAX32664C)	0x41	0x00	Register Address	Register value
Read Register	Read the value of a MAX30205 register. (MAX32664B)	0x41	0x01	Register Address	Register value

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Read Register	Read the value of a MAX30001 register. (MAX32664B)	0x41	0x02	Register Address	Register value
Read Register	Read the value of a MAX30101/ MAX30102 register. (MAX32664A, MAX32664D)	0x41	0x03	Register Address	Register value
Read Register	Read the value of an accelerometer sensor register. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x41	0x04	Register Address	Register value
Get Attributes of the AFE	Retrieve the attributes of the MAX86140/ MAX86141/ MAXM86146/ MAXM86161 AFE. (MAX32664B, MAX32664C)	0x42	0x00	-	Number of bytes in a word for this sensor, Number of registers available for this sensor.
Get Attributes of the AFE	Retrieve the attributes of the MAX30205 AFE. (MAX32664B)	0x42	0x01	-	Number of bytes in a word for this sensor, Number of registers available for this sensor.
Get Attributes of the AFE	Retrieve the attributes of the MAX30001 AFE. (MAX32664B)	0x42	0x02	-	Number of bytes in a word for this sensor, Number of registers available for this sensor.
Get Attributes of the AFE	Retrieve the attributes of the MAX30101/ MAX30102 AFE. (MAX32664A, MAX32664D)	0x42	0x03	-	Number of bytes in a word for this sensor, Number of registers available for this sensor.
Get Attributes of the AFE	Retrieve the attributes of the accelerometer sensor AFE. (MAX32664A, MAX32664B, (MAX32664C, MAX32664D)	0x42	0x04	-	Number of bytes in a word for this sensor, Number of registers available for this sensor.
Dump Registers	Read all the MAX86140/ MAX86141/ MAXM86161 registers. (MAX32664B, MAX32664C)	0x43	0x00	-	Register address 0, register value 0, register address 1, register value 1, ..., register address n, register value n

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Dump Registers	Read all the MAX30205 registers. (MAX32664B)	0x43	0x01	-	Register address 0, register value 0, register address 1, register value 1, ..., register address n, register value n
Dump Registers	Read all the MAX30001 registers. (MAX32664B)	0x43	0x02	-	Register address 0, register value 0, register address 1, register value 1, ..., register address n, register value n
Dump Registers	Read all the MAX30101/ MAX30102 registers. (MAX32664A, MAX32664D)	0x43	0x03	-	Register address 0, register value 0, register address 1, register value 1, ..., register address n, register value n
Dump Registers	Read all the accelerometer sensor registers. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x43	0x04	-	Register address 0, register value 0, register address 1, register value 1, ..., register address n, register value n
Sensor Mode Enable	Enable the MAX86140/ MAX86141/ MAXM86146/ MAXM86161 sensor. CMD_DELAY = 250ms (MAX32664B, MAX32664C)	0x44	0x00	0x00: Disable 0x01: Enable	-
Sensor Mode Enable	Enable the MAX30205 sensor. CMD_DELAY = 20ms (MAX32664B)	0x44	0x01	0x00: Disable 0x01: Enable	-
Sensor Mode Enable	Enable the MAX30001 sensor. CMD_DELAY = 20ms (MAX32664B)	0x44	0x02	0x00: Disable 0x01: Enable	-
Sensor Mode Enable	Enable the MAX30101/ MAX30102 sensor. CMD_DELAY = 40ms (MAX32664A, (MAX32664D)	0x44	0x03	0x00: Disable 0x01: Enable	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Sensor Mode Enable	Enable the accelerometer sensor. CMD_DELAY = 20ms (MAX32664A, MAX32664B, MAX32664C)	0x44	0x04	0x00, 0x00: Disable sensor hub accelerometer 0x00, 0x01: Disable external host accelerometer 0x01, 0x00: Enable sensor hub accelerometer 0x01, 0x01: Enable external host accelerometer	-
Sensor Mode Enable	Single command to enable multiple sensors. CMD_DELAY = 20ms For the MAX30101/MAX30102 sensor, the CMD_DELAY = 40ms Use the total CMD_DELAY of all the sensors that are enabled. Exceptions: 1. If any sensor in the list is already enabled, it turns off and enables again. 2. If enabling one of the sensors in the list fails, the sensor hub disables all the sensors in the command list. 3. All sensors in this command list must be valid available hardware, otherwise, the sensor hub disables all the sensors listed in this command. (MAX32664B, MAX32664C)	0x44	0xFF	N, SI, SM, SE, ... SI, SM, SE: Enable multiple sensors, where: <ul style="list-style-type: none"> • N is the number of sensors • SI is the sensor index • SM is the sensor mode • SE is 1 if the sensor is an external host or 0 if the sensor is connected to the sensor hub Sensor indices are defined as: 0x00: MAX86140/ MAX86141/ MAXM86146/ MAXM86161 0x01: MAX30205 0x02: MAX30001 0x03: MAX30101/MAX30102 0x04: Accelerometer Sensor modes are defined in the first byte of Write Bytes field of the Sensor Mode Enable commands, 0x44 0x00 to 0x44 0x04	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Sensor Mode Read	Read the MAX86140/ MAX86141/ MAXM86146/ MAXM86161 sensor mode. (MAX32664B, MAX32664C)	0x45	0x00	-	0x00: Disabled 0x01: Enabled
Sensor Mode Read	Read the MAX30205 sensor mode. (MAX32664B)	0x45	0x01	-	0x00: Disabled 0x01: Enabled
Sensor Mode Read	Read the MAX30001 sensor mode. (MAX32664B)	0x45	0x02	-	0x00: Disabled 0x01: Enabled
Sensor Mode Read	Read the MAX30101/ MAX30102 sensor mode. (MAX32664A, (MAX32664D)	0x45	0x03	-	0x00: Disabled 0x01: Enabled
Sensor Mode Read	Read the external accelerometer sensor mode. (MAX32664A, MAX32664B, MAX32664C)	0x45	0x04	-	0x00, 0x00: Sensor hub accelerometer disabled 0x00, 0x01: External host accelerometer disabled 0x01, 0x00: Sensor hub accelerometer enabled 0x01, 0x01: External host accelerometer enabled

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Sensor Configuration	<p>Write the sensor configuration for the MAX86140/ MAX86141/ MAXM86146/ MAXM86161.</p> <p>The LSb0 of the Write Byte is the firmware_default bit.</p> <p>The LSb1 of the Write Byte is the dac_calib bit.</p> <p>CMD_DELAY = 220ms if the first Write Byte is 0x02 or 0x03.</p> <p>(MAX32664C with MAXM86161)</p>	0x46	0x00	<p>First Byte</p> <p>0x00: Do not use firmware default register settings, and do not run DAC calibration when the algorithm/sensor is enabled. The sensor hub does not overwrite the user settings when the algorithm/sensor is enabled. If the user does not disable AEC, then the sample rate, pulse interval, and LED current are managed by the algorithm. AEC disable is a separate command. Ignore the second byte ppg_cfg1 value.</p> <p>0x01: Use firmware default register settings and disable DAC calibration. As soon as the algorithm runs, it uses the firmware defaults. Do not run DAC calibration. Ignore the second byte ppg_cfg1 value.</p>	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
	(Continued)			<p>0x02: Immediately run DAC calibration but do not use the firmware default. Instead use the second byte ppg_cfg1 value and wait for the user settings to directly write user-defined register values to the MAX8614x. This mode does not use the firmware default. The algorithm does not run the calibration again, because it was run when the command was received. Only this mode immediately runs calibration and uses the second byte ppg_cfg1 value.</p> <p>0x03 (Default): Use the firmware default register settings and run DAC calibration when the algorithm/sensor is enabled. Ignore the second byte ppg_cfg1 value and use the firmware default ppg_cfg1.</p> <p>Second Byte: The ppg_cfg1 value.</p>	

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Sensor Configuration	Enable/disable wake up on motion. (MAX32664C)	0x46	0x00	<p>First Byte: 0x00: Disable (1st byte) 0x01: Enable</p> <p>Second Byte: 0x01 to 0xFE: Wake up filter period (seconds). Motion must be present during this period time before a wake-up is generated. 0xFF: Disable (2nd byte)</p> <p>Third Byte: 0x01 to 0x80 LSB = 0.0625g (1/16g. For example 0x08 is 0.5g.</p>	

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Sensor Configuration	<p>Read the sensor configuration for the MAX86140/ MAX86141/ MAXM86146/ MAXM86161.</p> <p>The LSb0 of the Write Byte is the firmware default bit. The LSb1 of the Write Byte is the dac_calib bit. (MAX32664C with MAXM86161)</p>	0x47	0x00	-	<p>First Byte 0x00: Do not use the firmware default register settings, and do not run DAC calibration when the algorithm/sensor is enabled. The sensor hub does not overwrite the user settings when the algorithm/sensor is enabled. If the user does not disable AEC, then the sample rate, pulse interval, and LED current are managed by algorithm. AEC disable is a separate command. Ignore the ppg_cfg1 value.</p> <p>0x01: Use the firmware default register settings and disable DAC calibration. As soon as the algorithm is run, it uses the firmware defaults. Do not run DAC calibration. Ignore the ppg_cfg1 value.</p> <p>0x02: Immediately run DAC calibration using the ppg_cfg1 value, and wait for the user settings to directly write the user-defined register values to the MAX8614x. This mode does not use the firmware default. The algorithm does not run calibration again, because it was run when the command was received. Only this mode immediately runs calibration. Use the ppg_cfg1 value.</p> <p>0x03 (Default): Use the firmware default register settings and run DAC calibration when the algorithm/sensor is enabled. Ignore the ppg_cfg1 value and use the firmware default ppg_cfg1.</p> <p>Second Byte: The ppg_cfg value.</p> <p>Third Byte: The DAC calibration register value.</p>

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	Automatic Gain Control (AGC) algorithm: Set the target percentage of the full-scale ADC range that the automatic gain control (AGC) algorithm uses. (MAX32664A, MAX32664D)	0x50	0x00	0x00, 0 to 100 percent	-
Algorithm Configuration	AGC algorithm: Set the step size toward the target for the AGC algorithm. (MAX32664A, MAX32664D)	0x50	0x00	0x01, 0 to 100 percent	-
Algorithm Configuration	AGC algorithm: Set the sensitivity for the AGC algorithm. (MAX32664A, MAX32664D)	0x50	0x00	0x02, 0 to 100 percent	-
Algorithm Configuration	AGC algorithm: Set the number of samples to average for the AGC algorithm. (MAX32664A)	0x50	0x00	0x03, Number of samples to average (range is 0 to 255).	-
Algorithm Configuration	Blood Pressure Trending (BPT) algorithm: Set if the user is on blood pressure medication. (MAX32664D)	0x50	0x04	0x00, 0x00: Not using blood pressure (BP) medication 0x00, 0x01: Using BP medication	-
Algorithm Configuration	BPT algorithm: Write the three samples of the systolic BP byte values needed by the calibration procedure. (MAX32664D)	0x50	0x04	0x01, systolic value 1, systolic value 2, systolic value 3	-
Algorithm Configuration	BPT algorithm: Write the three samples of the diastolic BP byte values needed by the calibration procedure. (MAX32664D)	0x50	0x04	0x02, diastolic value 1, diastolic value 2, diastolic value 3	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	BPT algorithm: Write the calibration data for this user. (Use the data from the 0x51 0x04 0x03 command). CMD_DELAY = 30ms. (MAX32664D)	0x50	0x04	0x03, 824 bytes of calibration data	-
Algorithm Configuration	BPT algorithm: Configure whether the user is not resting or resting. (MAX32664D)	0x50	0x04	0x05, 0x00: Resting 0x05, 0x01: Not resting	-
Algorithm Configuration	BPT algorithm: Set the SpO ₂ coefficients A, B, C. (MAX32664D)	0x50	0x04	0x0B, 4 bytes signed integer A, 4 bytes signed integer B, 4 bytes signed integer C (32-bit integers which are the coefficients times 100,000) The MAXREFDES220# without the cover glass uses the following coefficients as the default values: A = 159584 B = -3465966 C = 11268987	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the SpO ₂ coefficients A, B, and C. (MAX32664C)	0x50	0x07	0x00, 4 bytes signed integer A, 4 bytes signed integer B, 4 bytes signed integer C The MAXREFDES103# uses the following coefficients as the default values: A = 0 (0x00000000) B = -25.224999 (0xFFD7FBDD) C = 112.317421 (0x00AB61FE)	

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the motion detection period in seconds. The algorithm considers the state to be motionless if the motion is below the threshold for this duration of time. (MAX32664C)	0x50	0x07	0x01, MSB of period, LSB of period (16-bit unsigned integer, seconds)	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the motion threshold for the WSpO ₂ algorithm. (MAX32664C)	0x50	0x07	0x02, 4 bytes (32-bit signed integers which are the motion threshold in milli-Gs times 100,000. For example, 0x1C9C380 is 0.3G)	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the WSpO ₂ AGC timeout (seconds). (MAX32664C)	0x50	0x07	0x03, WSpO ₂ AGC timeout (8-bit unsigned)	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the SpO ₂ algorithm timeout (seconds). (MAX32664C)	0x50	0x07	0x04, SpO ₂ algorithm timeout (8-bit unsigned)	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the initial heart-rate setting. (MAX32664C)	0x50	0x07	0x05, Initial heart-rate setting (8-bit unsigned)	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the user's height. (MAX32664C)	0x50	0x07	0x06, Height (16-bit unsigned integer which is the height in centimeters times 256)	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the user's weight. (MAX32664C)	0x50	0x07	0x07, Weight (16-bit unsigned integer which is the weight in kilograms times 256)	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the user's age. (MAX32664C)	0x50	0x07	0x08, Age in years (8-bit unsigned)	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the user's gender. (MAX32664C)	0x50	0x07	0x09, 0x00: Male 0x09, 0x01: Female	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the algorithm run mode. (MAX32664C)	0x50	0x07	0xA, 0x00: Continuous HRM, continuous SpO ₂ 0xA, 0x01: Continuous HRM, one-shot SpO ₂ 0xA, 0x02: Continuous HRM 0xA, 0x03: Sampled HRM 0xA, 0x04: Sampled HRM, one-shot SpO ₂ 0xA, 0x05: Activity tracking only 0xA, 0x06: SpO ₂ calibration	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Enable the AEC algorithm. (MAX32664C)	0x50	0x07	0xB, 0x00: Disable 0xB, 0x01: Enable	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Enable the SCD algorithm. (MAX32664C)	0x50	0x07	0xC, 0x00: Disable 0xC, 0x01: Enable	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the target PD current period (period to update the target PD current with the AEC formula). (MAX32664C)	0x50	0x07	0xD, Target PD current period (16-bit unsigned integer)	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the motion magnitude threshold. (MAX32664C)	0x50	0x07	0xE, Motion magnitude threshold (16-bit unsigned integer, 0.001g. For example, 0x0032 is 0.05g)	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the minimum PD current setting. (MAX32664C)	0x50	0x07	0x0F, Minimum PD current setting (16-bit unsigned integer, 0.1mA) This is the target PD current you would like AEC algorithm to maintain initially. It does not correspond to any register. Once you set what PD current you need, the algorithm calculates what LED current should be.	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the initial PD current setting. (MAX32664C)	0x50	0x07	0x10, Initial PD current setting (16-bit unsigned integer, 0.1mA)	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the target PD current setting. (MAX32664C)	0x50	0x07	0x11, Target PD current setting (16-bit unsigned integer, 0.1mA)	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Enable the auto target PD current calculation. (MAX32664C)	0x50	0x07	0x12, 0x00: Value of target PD current is used (AGC functionality) 0x12, 0x01: Target PD current is calculated automatically	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the minimum integration time. (MAX32664C)	0x50	0x07	0x13, 0x00: 14.8μs (default) 0x13, 0x01: 29.4μs 0x13, 0x02: 58.7μs 0x13, 0x03: 117.3μs	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the minimum frequency sampling. (MAX32664C)	0x50	0x07	0x14, 0x00: 25sps, averaging = 1 (default) 0x14, 0x01: 50sps, averaging = 2 0x14, 0x02: 100sps, averaging = 4 0x14, 0x03: 200sps, averaging = 8 0x14, 0x04: 400sps, averaging = 16	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the maximum integration time. (MAX32664C)	0x50	0x07	0x15, 0x00: 14.8μs (default) 0x15, 0x01: 29.4μs 0x15, 0x02: 58.7μs 0x15, 0x03: 117.3μs	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set the maximum frequency sampling. (MAX32664C)	0x50	0x07	0x16, 0x00: 25sps, averaging = 1 (default) 0x16, 0x01: 50sps, averaging = 2 0x16, 0x02: 100sps, averaging = 4 0x16, 0x03: 200sps, averaging = 8 0x16, 0x04: 400sps, averaging = 16	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set which Slots and PDs are used for input 1 and 2 of the WHRM (Heart Rate) algorithm. Default: 0x0001 MAXM86146 Default: 0x0073 (MAX32664C)	0x50	0x07	0x17, 0xWX, 0xYZ WX is input 1 of the WHRM algorithm. W = 0 for Slot 1 W = 1 for Slot 2 W = 2 for Slot 3 W = 3 for Slot 4 W = 4 for Slot 5 W = 5 for Slot 6 W = 7 for Slot not used X = 0 for PD1 X = 1 for PD2 X = 3 for PD not used. YZ is input 2 of the WHRM algorithm. Y = 0 for Slot 1 Y = 1 for Slot 2 Y = 2 for Slot 3 Y = 3 for Slot 4 Y = 4 for Slot 5 Y = 5 for Slot 6 Y = 7 for Slot not used Z = 0 for PD1 Z = 1 for PD2 Z = 3 for PD not used.	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set which LEDs and PDs are used for IR and red LEDs for the input to the WSpO ₂ algorithm. Default: 0x1020 MAXM86146 Default: 0x2111 (MAX32664C)	0x50	0x07	0x18, 0xWX, 0xYZ WX is the LED/PD used for IR for the WSpO ₂ algorithm. W = 0 for Slot 1 W = 1 for Slot 2 W = 2 for Slot 3 W = 3 for Slot 4 W = 4 for Slot 5 W = 5 for Slot 6 W = 7 for Slot not used X = 0 for PD1 X = 1 for PD2 X = 3 for PD not used. YZ is the LED/PD used for red for the WSpO ₂ algorithm and WHRM algorithm. Y = 0 for Slot 1 Y = 1 for Slot 2 Y = 2 for Slot 3 Y = 3 for Slot 4 Y = 4 for Slot 5 Y = 5 for Slot 6 Y = 7 for Slot not used Z = 0 for PD1 Z = 1 for PD2 Z = 3 for PD not used	-
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Set which slots are used for the firing of which LED. Default: 0x12 0x30 0x00 (slot 1:LED1, slot 2: LED2, slot 3: LED3, slot 4-6: Not used) MAXM86146 Default: 0x123456 (MAX32664C)	0x50	0x07	0x19, UV, WX, YZ U is Slot 1 V is Slot 2 W is Slot 3 X is Slot 4 Y is Slot 5 Z is Slot 6 U, V, W, X, Y, Z are defined as: 0: No LED firing 1: LED1 firing 2: LED2 firing 3: LED3 firing 4: LED4 firing 5: LED5 firing 6: LED6 firing 7: LED1 and LED2 firing 8: LED1 and LED3 firing 9: LED2 and LED3 firing	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration Read	Automatic Gain Control (AGC) algorithm: Read the target percentage of the full-scale ADC range that the AGC algorithm is using. (MAX32664A, MAX32664D)	0x51	0x00	0x00	0 to 100 Percent
Algorithm Configuration Read	AGC algorithm: Read step size toward the target. (MAX32664A, MAX32664D)	0x51	0x00	0x01	0 to 100 Percent
Algorithm Configuration Read	AGC algorithm: Read the sensitivity for the AGC algorithm. (MAX32664A, MAX32664D)	0x51	0x00	0x02	0 to 100 Percent
Algorithm Configuration Read	AGC algorithm: Read the number of samples to average for the AGC algorithm. (MAX32664A, MAX32664D)	0x51	0x00	0x03	Number of samples to average (range is 0 to 255)
Algorithm Configuration Read	BPT algorithm: Read the calibration data results from the calibration procedure. Host can use this for saving the user calibration data when switching users or for writing user calibration data after a reset. (MAX32664D)	0x51	0x04	0x03	824 bytes of calibration data
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the SpO ₂ coefficients A, B, and C. (MAX32664C)	0x51	0x07	0x00	4 bytes signed integer A, 4 bytes signed integer B, 4 bytes signed integer C
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the motion detection period in seconds. (MAX32664C)	0x51	0x07	0x01	MSB of period, LSB of period (16-bit unsigned integer)

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the motion threshold for the WSpO ₂ algorithm. (MAX32664C)	0x51	0x07	0x02	4 bytes (32-bit signed integers which are the motion threshold times 100,000)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the WSpO ₂ AGC timeout (seconds). (MAX32664C)	0x51	0x07	0x03	WSpO ₂ AGC timeout (8-bit unsigned)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the SpO ₂ algorithm timeout (seconds). (MAX32664C)	0x51	0x07	0x04	SpO ₂ algorithm timeout (8-bit unsigned)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the initial heart-rate setting. (MAX32664C)	0x51	0x07	0x05	Initial heart-rate setting (8-bit unsigned)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the user's height. (MAX32664C)	0x51	0x07	0x06	Height (16-bit unsigned integer which is the height in centimeter times 256)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the user's weight. (MAX32664C)	0x51	0x07	0x07	Weight (16-bit unsigned integer which is the weight in kilograms times 256)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the user's age. (MAX32664C)	0x51	0x07	0x08	Age in years (8-bit unsigned)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the user's gender. (MAX32664C)	0x51	0x07	0x09	0x00: Male 0x01: Female

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the algorithm run mode. (MAX32664C)	0x51	0x07	0x0A	0x00: Continuous HRM, continuous SpO ₂ 0x01: Continuous HRM, one-shot SpO ₂ 0x02: Continuous HRM 0x03: Sampled HRM 0x04: Sampled HRM, one-shot SpO ₂ 0x05: Activity tracking only 0x06: SpO ₂ calibration
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the AEC algorithm enable. (MAX32664C)	0x51	0x07	0x0B	0x00: Disabled 0x01: Enabled
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the SCD algorithm enable. (MAX32664C)	0x51	0x07	0x0C	0x00: Disabled 0x01: Enabled
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the target PD current period (period to update the target PD current with the AEC formula). (MAX32664C)	0x51	0x07	0x0D	Target PD current period (16-bit unsigned integer)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the motion magnitude threshold. (MAX32664C)	0x51	0x07	0x0E	Motion magnitude threshold (16-bit unsigned integer, 0.001g)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the minimum PD current setting. (MAX32664C)	0x51	0x07	0x0F	Minimum PD current setting (16-bit unsigned integer, 0.1mA)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the initial PD current setting. (MAX32664C)	0x51	0x07	0x10	Initial PD current setting (16-bit unsigned integer, 0.1mA)

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the target PD current setting. (MAX32664C)	0x51	0x07	0x11	Target PD current setting (16-bit unsigned integer, 0.1mA)
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the auto target PD current calculation enable (MAX32664C)	0x51	0x07	0x12	0x00: Value of target PD current is used (AGC functionality) 0x01: Target PD current is calculated automatically
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the minimum integration time setting. (MAX32664C)	0x51	0x07	0x13	0x00: 14.8µs (default) 0x01: 29.4µs 0x02: 58.7µs 0x03: 117.3µs
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read minimum frequency sampling setting. (MAX32664C)	0x51	0x07	0x14	0x00: 25sps, averaging = 1 (default) 0x01: 50sps, averaging = 2 0x02: 100sps, averaging = 4 0x03: 200sps, averaging = 8 0x04: 400sps, averaging = 16
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the maximum integration time setting. (MAX32664C)	0x51	0x07	0x15	0x00: 14.8µs (default) 0x01: 29.4µs 0x02: 58.7µs 0x03: 117.3µs
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the maximum frequency sampling setting. (MAX32664C)	0x51	0x07	0x16	0x00: 25sps, averaging = 1 (default) 0x01: 50sps, averaging = 2 0x02: 100sps, averaging = 4 0x03: 200sps, averaging = 8 0x04: 400sps, averaging = 16

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read the slots and PD configuration. (MAX32664C)	0x51	0x07	0x17	<p>0xWX, 0xYZ</p> <p>WX is the LED/PD used for IR for the WSpO₂ algorithm.</p> <p>W = 0 for Slot 1 W = 1 for Slot 2 W = 2 for Slot 3 W = 3 for Slot 4 W = 4 for Slot 5 W = 5 for Slot 6 W = 7 for Slot not used</p> <p>X = 0 for PD1 X = 1 for PD2 X = 3 for PD not used.</p> <p>YZ is the LED/PD used for red for the WSpO₂ algorithm.</p> <p>WHRM algorithm.</p> <p>Y = 0 for Slot 1 Y = 1 for Slot 2 Y = 2 for Slot 3 Y = 3 for Slot 4 Y = 4 for Slot 5 Y = 5 for Slot 6 Y = 7 for Slot not used</p> <p>Z = 0 for PD1 Z = 1 for PD2 Z = 3 for PD not used</p>

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read which LEDs and PDs are used for IR and red LEDs for the input to the WSpO ₂ algorithm. (MAX32664C)	0x51	0x07	0x18	<p>0xWX, 0xYZ</p> <p>WX is the LED/PD used for IR for the WSpO₂ algorithm.</p> <p>W = 0 for Slot 1 W = 1 for Slot 2 W = 2 for Slot 3 W = 3 for Slot 4 W = 4 for Slot 5 W = 5 for Slot 6 W = 7 for Slot not used</p> <p>X = 0 for PD1 X = 1 for PD2 X = 3 for PD not used.</p> <p>YZ is the LED/PD used for red for the WSpO₂ algorithm.</p> <p>WHRM algorithm.</p> <p>Y = 0 for Slot 1 Y = 1 for Slot 2 Y = 2 for Slot 3 Y = 3 for Slot 4 Y = 4 for Slot 5 Y = 5 for Slot 6 Y = 7 for Slot not used</p> <p>Z = 0 for PD1 Z = 1 for PD2 Z = 3 for PD not used</p>
Algorithm Configuration Read	Wearable Algorithm Suite (WHRM+WSpO ₂): Read which slots are used for the firing of which LED. (MAX32664C)	0x51	0x07	0x19	<p>0xUV, 0xWX, 0xYZ</p> <p>U is Slot 1 V is Slot 2 W is Slot 3 X is Slot 4 Y is Slot 5 Z is Slot 6</p> <p>U, V, W, X, Y, Z are defined as:</p> <ul style="list-style-type: none"> 0: No LED firing 1: LED1 firing 2: LED2 firing 3: LED3 firing 4: LED4 firing 5: LED5 firing 6: LED6 firing 7: LED1 and LED2 firing 8: LED1 and LED3 firing 9: LED2 and LED3 firing
Algorithm Mode Enable	AGC: Enable the AGC algorithm. CMD_DELAY = 20ms (MAX32664A)	0x52	0x00	0x00: Disable 0x01: Enable	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Algorithm Mode Enable	AEC: Enable the AEC algorithm.	0x52	0x01	0x00: Disable 0x01: Enable	-
Algorithm Mode Enable	MaximFast: Enable the WHRM, MaximFast algorithm. CMD_DELAY = 40ms (MAX32664A, MAX32664B)	0x52	0x02	0x00: Disable 0x01: Enable Mode 1 Report 0x02: Enable Mode 2, Extended Report	-
Algorithm Mode Enable	Electrocardiogram (ECG): Enable the ECG algorithm.	0x52	0x03	0x00: Disable 0x01: Enable	-
Algorithm Mode Enable	Blood Pressure Trending (BPT): Enable the BPT algorithm. CMD_DELAY = 20ms (MAX32664D)	0x52	0x04	0x00: Disable 0x01: Enable Calibration Mode 0x02: Enable Estimation Mode	-
Algorithm Mode Enable	Wearable Algorithm Suite (WHRM+WSPO ₂): Enable the algorithm. (MAX32664C)	0x52	0x07	0x00: Disable (CMD_DELAY = 120ms) 0x01: Enable Mode 1 (CMD_DELAY = 320ms) 0x02: Enable Mode 2 (CMD_DELAY = 320ms)	-
Bootloader Flash	Set the initialization vector (IV) bytes. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x80	0x00	Use bytes 0x28 to 0x32 from the .msbl file as the IV bytes.	-
Bootloader Flash	Set the authentication bytes. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x80	0x01	Use bytes 0x34 to 0x43 from the .msbl file.	-
Bootloader Flash	Set the number of pages. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x80	0x02	0x00, Number of pages located at byte 0x44 from the .msbl file.	-
Bootloader Flash	Erase the application flash memory. CMD_DELAY = 1400ms. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x80	0x03	-	-

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Bootloader Flash	Send the page values. CMD_DELAY = 680ms. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x80	0x04	The first page is specified by byte 0x4C from the .msbl file. The total bytes for each message protocol are the page size plus 16 bytes of CRC.	-
Bootloader Information	Get bootloader version. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x81	0x00	-	Major version byte, Minor version byte, Revision byte
Bootloader Information	Get the page size in bytes. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0x81	0x01	-	Upper byte of page size, Lower byte of page size
Sensor Hub AFE Authentication Read	Sensor hub authentication AFE sequence: (Wellness App library, MAX32664C)	0xB2	-	-	Twelve bytes of authentication data
Sensor Hub AFE Initialization Vector Read	Get the sensor hub authentication AFE initialization vector (IV) bytes. (MAX32664C)	0xB3	-	-	Twelve bytes of initialization vector data
Sensor Hub AFE Authentication Public Key Write	Set the sensor hub authentication AFE public key: Wellness app library uses this to authenticate the Maxim AFE. (Wellness App library, MAX32664C)	0xB4	-	Public key: twelve bytes	
Sensor Hub authentication AFE Public Key Read	Get the sensor hub authentication AFE public key: Wellness app library uses this to authenticate the Maxim AFE. (Wellness App library, MAX32664C)	0xB5	-	-	Twelve bytes of the public key

HOST COMMAND					MAX32664
FAMILY NAME	DESCRIPTION	FAMILY BYTE	INDEX BYTE	WRITE BYTES	RESPONSE BYTES
Identity	Read the MCU type. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0xFF	0x00	-	0x00: MAX32625 0x01: MAX32660/MAX32664
Identity	Read the sensor hub version. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0xFF	0x03	-	Major version byte, Minor version byte, Revision byte
Identity	Read the algorithm: version. Deprecated - no longer supported. (MAX32664A, MAX32664B, MAX32664C, MAX32664D)	0xFF	0x07	-	Major version byte, Minor version byte, Revision byte. Deprecated

Table 7 defines the bit fields of the sensor hub status byte.

Table 7. Sensor Hub Status Byte

BIT	7	6	5	4	3	2	1	0
Field	Reserved	HostAccelUflnt	FifoInOverInt	FifoOutOvrlnt	DataRdyInt	Err2	Err1	Err0

Table 8 provides the sequence of commands for writing external (host connected) accelerometer data to the input FIFO for the MAX32664A. The KX-122 connected to the MAX32664 is not used. The MAX32664B and MAX32664C implementations are similar and require a couple of commands to be added to the setup sequence.

Table 8. Sequence of Commands to Write External Accelerometer Data to the Input FIFO

HOST COMMAND	COMMAND DESCRIPTION	READ MAX32664 RESPONSE	RESPONSE DESCRIPTION
0xAA 0x10 0x00 0x03†	Set output mode to sensor and algorithm data.	0xAB 0x00	No error.
0xAA 0x10 0x01 0x0F‡	Set the threshold for the FIFO to 0x0F.	0xAB 0x00	No error.
0xAA 0x44 0x03 0x01*	Enable the MAX30101 sensor. (MAX32664A)	0xAB 0x00	No error.
0xAA 0x44 0x04 0x01 0x01*	Enable the input FIFO for host supplied accelerometer data.	0xAB 0x00	No error.
0xAA 0x52 0x02 0x01*	Enable MaximFast algorithm mode 1. (MAX32664A)	0xAB 0x00	No error.

HOST COMMAND	COMMAND DESCRIPTION	READ MAX32664 RESPONSE	RESPONSE DESCRIPTION
0xAA 0x13 0x00 0x04†	Read the sensor sample size for the accelerometer. (optional)	0xAB 0x00 0x06	No error. 6 bytes is the sample size.
0xAA 0x14 0x00 Sample 1 value to Sample N value*‡	Write data to the input FIFO. 6 bytes per accelerometer sample.	0xAB 0x00	No error.
0xAA 0x00 0x00*	Read the sensor hub status.	0xAB 0x00 0x08	No error. DataRdyInt bit is set
0xAA 0x12 0x00*	Get the number of samples in the FIFO.	0xAB 0x00 0x0F	No error. 0x0F samples are in the FIFO.
0xAA 0x12 0x01*	Read the data stored in the FIFO.	0xAB 0x00 0x03 0x6A 0x43 0x03 0x04 0x92 0x00 0x00 0x00 0x00 0x2E 0x15 0xFC 0xD8 0x00 0x04 0x02 0x3e 0x02 0x76 0x63 0x03 0xE4 0x03, data for fourteen other samples	No error. IR counts = 223811, Red counts = 19778, LED3 = 0, LED4 = 11797, X accelerometer = -0.808, Y accelerometer = 0.004, Z accelerometer = 0.574, Heart Rate = 63.0, Confidence = 99, SpO ₂ = 99.6, MaximFast State Machine Status = 3, data for fourteen other samples.

*Mandatory

†Recommended

‡Required for the MAX32664B and MAX32664C setup sequence

MAX32664 I²C Annotated Application Mode Example and Output FIFO Format

Refer to the following documents for example I²C sequences that the host microcontroller can use to configure the MAX32664 for data streaming. The output FIFO format for the sensors and algorithms are described in these documents.

- [User Guide 7087: Measuring Heart Rate and SpO₂ Using the MAX32664A](#)
- [User Guide 6922: Measuring Heart Rate Using MAX32664B](#)
- [User Guide 6924: Measuring SpO₂ and Heart Rate Using MAX32664C](#)
- [User Guide 6921: Measuring Blood Pressure, Heart Rate, and SpO₂ Using MAX32664D](#)

I²C Commands to Flash the Application Algorithm/Firmware

The MAX32664 is pre-programmed with bootloader firmware which accepts in-application programming of the Maxim supplied application algorithm/firmware file (.msbl).

To program the MAX32664 .msbl, the host microprocessor can implement the software to flash the .msbl file or the MAX32630FTHR can be used as a programmer. To use the MAX32630FTHR as a programmer, the following four MAX32630FTHR pins should be connected to the MAX32664

pins: P3.4 to SLAVE_SDA, P3.5 to SLAVE_SCL, P5.4 to MFIO, P5.6 to RSTN. The programming instructions and software needed are available in the HR, SpO₂ software download package on the [MAXREFDES220](#) site.

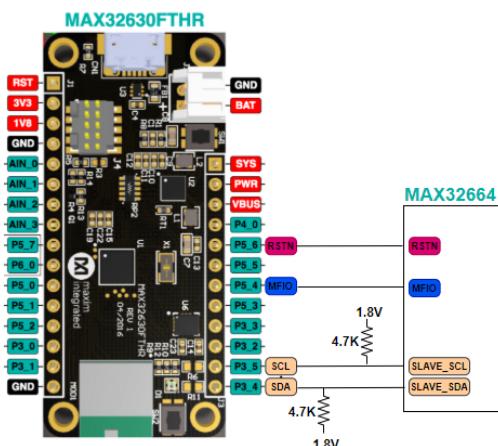


Figure 11. Using the MAX32630FTHR to flash the application .msbl to the MAX32664.

Sample host code to flash the .msbl can be found in the bootloader download package at the [MAX32660](#) website. Another example is located at this Mbed website: [Host Software MAX32664GWEC](#). SpO₂ (the host code that interfaces with the MAX32664 on the mbed site is dated. For the latest sample host code to interface with the MAX32664, use the compatible version of sample host code available in the download package on the [MAX32664](#) website). The source code to the python .msbl download file is in the download package at the [MAXREFDES220](#) website. The following constants in the sample .msbl code should be updated to reflect the latest CMD_DELAY definitions:

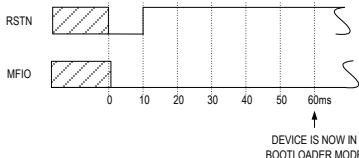
```
#define SS_BOOTLOADER_ERASE_DELAY 1400
#define PAGE_WRITE_DELAY_MS 680
```

Table 9 is a capture of the I²C commands that are necessary to flash the application algorithm/firmware to the MAX32664.

IMPORTANT: Do not enable the accelerometer if your board does not have the accelerometer.

This example was captured with the MAX32630FTHR acting as the host microcontroller. The MAX32664 uses the 8-bit slave address of 0xAA. The example encrypted algorithm file used was the MAX32660_SmartSensor_OS24_MaximFast_1.8.2a.msbl (26 pages, 8196 bytes for the page size). Each page sent includes 16 CRC bytes for that page, so there are 8208 bytes per page sent in the payload of the message. The number of pages is located at address 0x44 in the .msbl file. Values for the number of pages, initialization vector, and authorization bytes, might be different for the latest .msbl, but the locations of these values in the .msbl file remain the same. There are additional bytes in the .msbl past the last page; these are the file checksum bytes. Since the bootloader uses the commands listed below and it does not accept files, the file checksum bytes are not used by the bootloader.

Table 9. Annotated I2C Trace for Flashing the Application

HOST COMMAND	COMMAND DESCRIPTION	READ MAX32664 RESPONSE	RESPONSE DESCRIPTION
Sequence the MAX32664 to enter bootloader mode. *			
			
0xAA 0x01 0x00 0x08*	Set mode to 0x08 for bootloader mode.	0xAB 0x00	No error.
0xAA 0x02 0x00	Read mode.	0xAB 0x00 0x08	No error. Mode is bootloader.
0xAA 0xFF 0x00+	Get ID and MCU type.	0xAB 0x00 0x01	No error. MCU is MAX32660/MAX32664.
0xAA 0x81 0x00	Read bootloader firmware version.	0xAB 0x00 0x03 0x00 0x00	No error. Version is 3.0.0.
0xAA 0x81 0x01	Read bootloader page size.	0xAB 0x00 0x20 0x00	No error. Page size is 8192.
0xAA 0x80 0x02 0x00 0x1A*	Bootloader flash. Set the "number of pages" to 31 based on the value at byte 0x44 from the application .msbl file.	0xAB 0x00	No error.
00000044 02 ed 27 af 1a 00 00 20 04 00 00 00 c2 31 90 2c	Figure 13. Page number byte 0x44 from the .msbl file.		
0xAA 0x80 0x00 0x1A 0xDB 0xE5 0x0D 0x90 0x79 0xE6 0xC6 0x13 0x87 0xB9*	Bootloader flash. Set the initialization vector bytes to the 0x28 to 0x32 values from the .msbl file.	0xAB 0x00	No error.
00000000 6d 73 62 6c 00 00 00 00 4d 41 58 33 32 36 36 30 00000010 00 00 00 00 00 00 00 00 41 45 53 2d 32 35 36 00 00000020 00 00 00 00 00 00 00 00 1a db e5 0d 90 79 e6 c6 00000032 13 87 b9 00 2b f5 ad cd 2e 47 d2 83 23 88 37 63 00000040 02 ed 27 af 1a 00 00 20 04 00 00 00 c2 31 90 2c 00000050 e4 c8 37 e9 18 92 ad 3b 64 e7 0a ed eb 40 c1 66 00000060 e2 23 4f 71 d4 6b 98 e3 a7 f9 85 80 7a 4e 17 e7	Figure 14. Initialization vector bytes 0x28 to 0x32 from the .msbl file.		
0xAA 0x80 0x01 0x2B 0xF5 0xAD 0xCD 0x2E 0x47 0xD2 0x83 0x23 0x88 0x37 0x62 0x02 0xED 0x27 0xAF*	Bootloader flash. Set the authentication bytes to the 0x34 to 0x43 values from the .msbl file.	0xAB 0x00	No error.
00000030 13 87 b9 00 2b f5 ad cd 2e 47 d2 83 23 88 37 63 00000043 02 ed 27 af 1a 00 00 20 04 00 00 00 c2 31 90 2c	Figure 15. Authentication bytes 0x34 to 0x43 from the .msbl file.		
0xAA 0x80 0x03*	Bootloader flash. Erase application.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xC2 0x31 0x90 ... 0x9E 0x6A 0x0E*	Bootloader flash. Send page bytes 0x4C to 0x205B from the .msbl file.	0xAB 0x00	No error.
00000040 02 ed 27 af 1a 00 00 20 04 00 00 00 c2 31 90 2c 00000050 e4 c8 37 e9 18 92 ad 3b 64 e7 0a ed eb 40 c1 66 0000006f e2 23 4f 71 d4 6b 98 e3 a7 f9 85 80 7a 4e 17 e7			

HOST COMMAND	COMMAND DESCRIPTION	READ MAX32664 RESPONSE	RESPONSE DESCRIPTION
00002040 0000205b	9e 7c c0 3c 47 81 91 35 27 4c be cc 2a 7f ab lf 00 0d d6 ce 6f d4 ee cc b2 9e 6a 0e cc c5 68 92	Figure 16. Send page bytes 0x4C to 0x205B from the .msbl file.

HOST COMMAND	COMMAND DESCRIPTION	READ MAX32664 RESPONSE	RESPONSE DESCRIPTION
0xAA 0x80 0x04 0x0E 0xD2 0x16 ... 0x8D 0x69 0xEE*	Bootloader flash. Send page bytes 0x2416C to 0x2617B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x2F 0x4B 0x38 ... 0x02 0xA7 0xDC*	Bootloader flash. Send page bytes 0x2617C to 0x2818B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xA5 0xFE 0xFD ... 0xE3 0x38 0x89*	Bootloader flash. Send page bytes 0x2818C to 0x2A19B from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x52 0x88 0x9A ... 0xF0 0xC5 0x9D*	Bootloader flash. Send page bytes 0x2A19C to 0x2C1AB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xA3 0xA6 0x92 ... 0xA0 0x4D 0xBE*	Bootloader flash. Send page bytes 0x2C1AC to 0x2E1BB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x47 0x09 0x75 ... 0x24 0xBD 0x3D*	Bootloader flash. Send page bytes 0x2E1BC to 0x301CB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0x44 0xEC 0xE6 ... 0xBC 0xC9 0x5E*	Bootloader flash. Send page bytes 0x301CC to 0x321DB from the .msbl file.	0xAB 0x00	No error.
0xAA 0x80 0x04 0xD3 0x58 0x34 ... 0x62 0x00 0x37*	Bootloader flash. Send page bytes 0x321DC to 0x341EB from the .msbl file.	0xAB 0x00	No error.

Sequence the MAX32664 to enter application mode.*

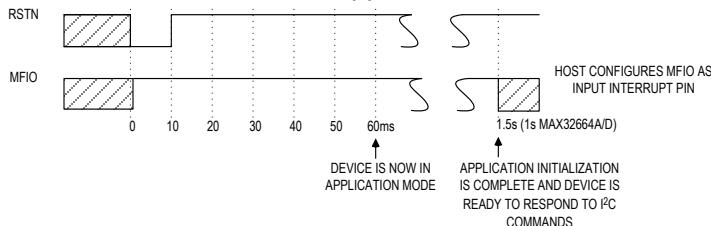


Figure 17. Sequence to enter application mode.

Alternately, the MAX32664 can be commanded to application mode.+

0xAA 0x01 0x00 0x00+	Set mode to 0x00 for bootloader mode.	0xAB 0x00	No error.
0xAA 0x02 0x00+	Read mode.	0xAB 0x00 0x00	No errors. Mode is application.
0xAA 0xFF 0x00	Get ID and MCU type.	0xAB 0x00 0x01	No error. MCU is MAX32660/MAX32664
0xAA 0xFF 0x03	Get Sensor Hub version.	0xAB 0x00 0x01 0x08 0x02	No error. Version is 1.8.2.
0xAA 0x42 0x03+	Get the MAX30101 AFE register attributes.	0xAB 0x00 0x01 0x24	No error. Attributes are 1 byte, 0x24 registers available.
0xAA 0x43 0x03	Read all the MAX30101 registers.	0xAB 0x00 0x00 0x00 0x01 0x00 0x02 0x40... Returns the Read Status Byte and 36 pairs of numbers.	No error. Reg 0x00=0, reg 0x01=0, reg0x02=0x40, ... Returns the Read Status Byte and 36 pairs of numbers.

HOST COMMAND	COMMAND DESCRIPTION	READ MAX32664 RESPONSE	RESPONSE DESCRIPTION
0xAA 0x41 0x03 0x07	Read the MAX30101 register 7.	0xAB 0x00 0x00	No error. Register 0x07 is 0.

*Mandatory

+Recommended

It is recommended to program the latest version of the MAX32664 sensor hub application algorithm/firmware .msbl file into the MAX32664 chip. Check the version that is programmed into the chip by using the command “Identity, Read sensor hub version.” The latest sensor hub algorithm/firmware is available for download for the MAX32664, MAXREFDES220#, and MAXREFDES101# from the Maxim website.

In-Application Programming of the MAX32664

The MAX32664 allows for in-application programming of the application algorithm/firmware.

In-application programming allows for the programming of the sensor hub application firmware during manufacturing and for allowing over-the-air (OTA) updates of the application firmware in the product. **Figure 18** is a flowchart of the in-application programming.

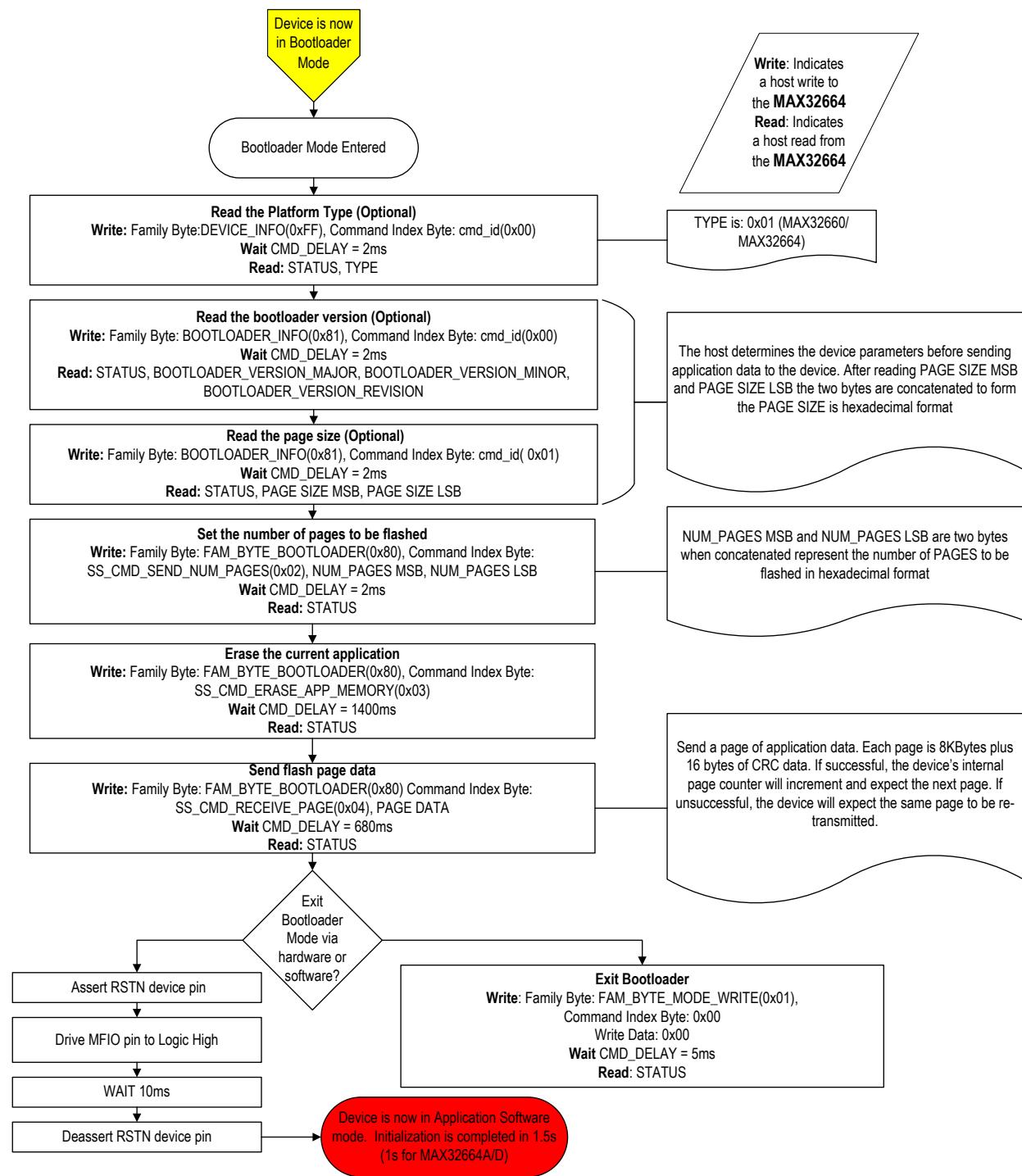


Figure 18. MAX32664 in-application programming flowchart.

MAX32664 APIs and Methods for Reset, Sleep, Status, Heartbeat

Table 10 summarizes the commands and methods to place the MAX32664 into reset or sleep, to interrogate its status, or to generate the “heartbeat” (a periodic signal generated by the software to indicate normal operation).

Table 10. MAX32664 I2C Message Protocol Definitions

COMMAND NAME	HOST COMMAND TO MAX32664	DESCRIPTION
MAX32664 Soft Reset	0xAA 0x01 0x00 0x02	Puts MAX32664 into reset.
MAX30101/MAX30102 AFE Soft Reset by Write Register to AFE	0xAA 0x40 0x03 0x09 0x40	Write 0x40 to MAX30101/MAX30102 register 0x09 to issue a soft reset to the MAX30101. The AFE must be enabled using the enable command.
MAX32664A/B/C/D Shutdown	0xAA 0x01 0x00 0x01	Place the MAX32664 into shutdown (MAX32660 "Backup" mode with RAM disabled). Restart by power cycling or pulsing RSTN.
MAX32664 Sleep between Interrupts		V20.2.0+, v30.2.4+, v32.1.2+, v33.6.0+ use sleep/deep-sleep for low-powered mode between polling periods.
MAX86140/MAX86141/MAXM86146/MAXM86161 AFE Shutdown. Use Write Reg to AFE	0xAA 0x40 0x00 0x0D 0x02	Write 0x06 to MAX86140/1, MAXM86146/61 register 0x0D (System Control) to put the MAX86140/1, MAXM86146/61 into shutdown (SHDN) mode. The AFE must be enabled using the enable command when using the read, write AFE register command)
MAX30101/MAX30102 AFE Sleep, Use Write Reg to AFE	0xAA 0x40 0x03 0x09 0x80	Write 0x80 to MAX30101/MAX30102 register 0x09 (Mode Configuration) to put the MAX30101/MAX30102 into shutdown mode. The AFE must be enabled using the enable command when using the read, write AFE register command)
KX122 Standby. Use Write Reg to Sensor	0xAA 0x40 0x04 0x18 0x00	Write 0x00 to KX122 register 0x018 (CNTL1) to put the KX122 into "Standby" mode. The KX122 must be enabled using the enable command when using the read, write KX122 register command)
MAX32664 Hard Reset	Use MFIO and RSTN pins according to Figure 5 and Figure 6.	
WDT in MAX32664 Bootloader Mode		Not implemented.

WDT in MAX32664 .msbl Application mode		Not implemented.
Bootloader or Application Status	0XAA 0x02 0x00	Send the read mode command. Response is 0xAB 0x00 0x08 if in bootloader mode or 0xAB 0x00 0x00 if in application mode.
Heartbeat (signal to signify that the sensor hub firmware is not stuck) for Application Mode		Not implemented.

Default Application .msbl Versions Pre-Programmed on the MAX32664A/B/C/D

The MAX32664A/B/C/D are pre-programmed with the bootloader and the application .msbl application/sensor hub version listed in Table 11. The pre-programmed application .msbl versions are not updated by Maxim. The pre-programmed parts may not be programmed with the latest version of the .msbl application. It is recommended that the sensor hub be updated with the latest application .msbl available on the Maxim Integrated website in order to be compatible with the latest sensor hub documentation.

Table 11. MAX32664A/B/C/D/MAXM86146 Pre-Programmed .msbl Version

MAXIM PART	PRE-PROGRAMMED .msbl APPLICATION/SENSOR HUB VERSION
MAX32664A	Version 1.9.1 (deprecated)
MAX32664B	Version 20.1.2 (deprecated)
MAX32664C	Version 30.2.2 (deprecated)
MAX32664D	Version 40.2.2 (deprecated)
MAXM86146	Application not pre-programmed.

MAX32664 Processing Capabilities

The MAX32664 IC hardware is the same as the MAX32660.

1. MIPS: Arm Cortex-M4 with FPU: 1.27 Dhystone MIPS/MHz
2. RAM: 96kB SRAM
3. Flash: 256kB Flash Memory
4. CPU Frequency: 96MHz

References

MAX32664 website: MAX32664 user guides; C-keyed .msbl for MAX32664A, MAX32664D; sample host code: [MAX32664 Design Resources Website](#).

Application Note 7148, protocol definition between sample host (MAX32630) and PC UART/BLE: [Interface Guide for MAX32664 Sensor Hub-Based Reference Design Platforms](#)

Frequently Asked Questions: [Maxim Support Center](#)

MAXREFDES101# hardware, software files: [MAXREFDES101#: Health Sensor Platform 2.0](#)

MAXREFDES103# hardware, software files: [MAXREFDES103#: Wrist-Based SpO₂, HR, and HRV Health Sensor Platform](#)

MAXREFDES220# hardware, software files: [MAXREFDES220#: Finger Heart Rate and Pulse Oximeter Smart Sensor with Digital Signal Processing](#)

Trademarks

Android is a registered trademark of Google Inc.

Arm and Cortex are registered trademarks of Arm Limited.

Bluetooth is a registered trademark owned by Bluetooth SIG, Inc. and any use of such marks by Maxim is under license.

Revision History

REVISION NUMBER	REVISION DATE	DESCRIPTION	PAGES CHANGED
0	01/19	Initial release	—
1	06/19	<p>Added the MAX32664B/C/D application firmware descriptions, Table 4 for additional GPIOs for the MAXREFDES101#, and 0x11 to the commands. Changed 0x14 0x04 to 0x14 0x00. Updated the 0x44 0x04 command and Sequence of Commands for the external host. Added PD1, PD2 to 0x51/0 0x05 0x09, and additional modes to 0x52 0x02 and 0x52 0x04. Labeled commands in Table 6 and Table 8 as MAX32664A/B/C/D. Added mode 2 for MaximFast in Table 8. Updated Table 9. Added Table 12, Table 13, and Table 14 for annotated I²C traces for MAX32664C and MAX32664D. Corrected the timing for CMD_DELAY and the application response time to I²C commands. Added descriptions for the Wearable Algorithm Suite (WHRM+WSpO₂) and updated MFIO for WHRM+WSpO₂. Added sensor configuration, MAXM86161 calibration, Table 17 and mode 2 for the WHRM+WSpO₂ algorithm. Updated host command 0x51/0 0x07 0x17/8 for LED/PD configuration and the TRY AGAIN values in Table 5.</p>	1–72
2	11/19	<p>Updated Table 1 by removing content for older MAX32664C WHRM and SpO₂ algorithms. Updated section MAX32664 GPIOs and RSTN Pin and Table 2 by adding content that the MAX32664B WHRM v20.2.x+ uses a polling method for the MFIO pin. Updated section MAX32664 Bootloader Mode. Updated Table 6 by adding shutdown command to Device Mode (0x01 0x00 0x01), removing the WHRM command (0x50) and WSpO₂ command (0x51), adding commands to change and read the I²C address (0x10/11 0x03), set and read the sensor hub counter (0x10/11 0x04), and a single command to enable multiple sensors (0x44 0xFF), reversing the systolic command (0x50 0x04 0x01) and the diastolic command (0x50 0x04 0x02), updating 0x50/51 0x07 0x017 for which slots and PDs are used, updating 0x50/51 0x07 0x018 for which LEDs and PDs are used for IR and red LEDs, and adding 0x50/51 0x07 0x18 for which slots are used for the firing of each LED. Removed Table 8 and Table 10 through Table 14. Updated section MAX32664 I²C Annotated Application Mode Example and Output FIFO Format. Updated Table 10 by adding the Shutdown command.</p>	7, 12–13, 19–21, 26, 32–33, 38–39, 46–47, 57, 63

3	8/20	<p>Updated the <i>Introduction</i>, <i>MAX32664 Variants</i>, <i>MAX32664 GPIOs and RSTN Pin</i>, <i>MAX32664 Bootup and Application Mode</i>, <i>MAX32664 Application Mode</i>, <i>Bit Transfer Process</i>, <i>I²C Write</i>, <i>I²C Read</i>, <i>I²C Commands to Flash the Application Algorithm/Firmware</i>, and <i>Default Application .msbl Versions Pre-Programmed on the MAX32664A/B/C/D sections</i>; updated Tables 1–2, 4–6, 8, and 10–11; replaced Figures 1–2, 6, 9–10, and 18; added Figures 3–5, and renumbered subsequent figures; added <i>MAXREFDES103#</i>, <i>Additional Sensor Hub Products</i>, <i>MAX32664 Processing Capabilities</i>, <i>References</i>, and <i>Trademarks</i> sections</p>	4–5, 10–11, 14–21, 23, 52–53, 59–60, 62
---	------	--	--

©2020 by Maxim Integrated Products, Inc. All rights reserved. Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. MAXIM INTEGRATED PRODUCTS, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. MAXIM ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering or registered trademarks of Maxim Integrated Product

MAX30101

High-Sensitivity Pulse Oximeter and Heart-Rate Sensor for Wearable Health

General Description

The MAX30101 is an integrated pulse oximetry and heart-rate monitor module. It includes internal LEDs, photodetectors, optical elements, and low-noise electronics with ambient light rejection. The MAX30101 provides a complete system solution to ease the design-in process for mobile and wearable devices.

The MAX30101 operates on a single 1.8V power supply and a separate 5.0V power supply for the internal LEDs. Communication is through a standard I²C-compatible interface. The module can be shut down through software with zero standby current, allowing the power rails to remain powered at all times.

Applications

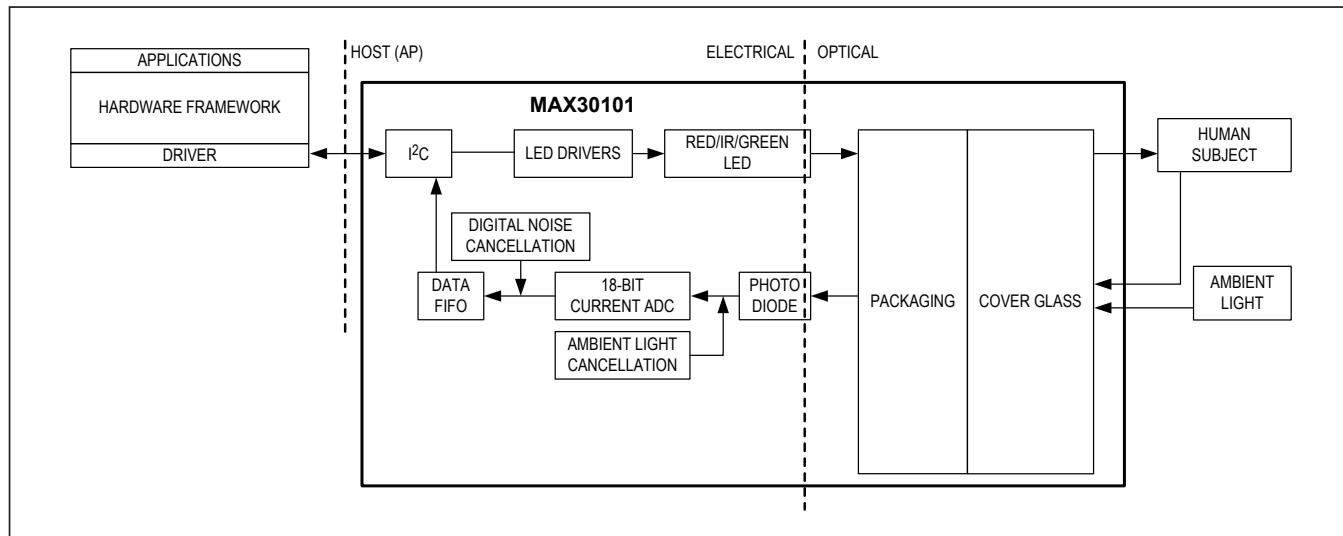
- Wearable Devices
- Fitness Assistant Devices
- Smartphones
- Tablets

Benefits and Features

- Heart-Rate Monitor and Pulse Oximeter Sensor in LED Reflective Solution
- Tiny 5.6mm x 3.3mm x 1.55mm 14-Pin Optical Module
 - Integrated Cover Glass for Optimal, Robust Performance
- Ultra-Low-Power Operation for Mobile Devices
 - Programmable Sample Rate and LED Current for Power Savings
 - Low-Power Heart-Rate Monitor (< 1mW)
 - Ultra-Low Shutdown Current (0.7µA, typ)
- Fast Data Output Capability
 - High Sample Rates
- Robust Motion Artifact Resilience
 - High SNR
- -40°C to +85°C Operating Temperature Range

Ordering Information appears at end of data sheet.

System Diagram



Absolute Maximum Ratings

V _{DD} to GND	-0.3V to +2.2V	Continuous Power Dissipation (T _A = +70°C)
GND to PGND	-0.3V to +0.3V	OESIP (derate 5.5mW/°C above +70°C)
V _{LED+} to PGND	-0.3V to +6.0V	440mW
All Other Pins to GND	-0.3V to +6.0V	Operating Temperature Range
Output Short-Circuit Current Duration	Continuous	-40°C to +85°C
Continuous Input Current into Any Terminal	±20mA	Junction Temperature
		+90°C
		Soldering Temperature (reflow)
		+260°C
		Storage Temperature Range
		-40°C to +105°C

Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. These are stress ratings only; functional operation of the device at these or any other conditions beyond those indicated in the operational sections of the specifications is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Package Information

PACKAGE TYPE: 14 OESIP	
Package Code	F143A5+1
Outline Number	21-1048
Land Pattern Number	90-0602
THERMAL RESISTANCE, FOUR-LAYER BOARD	
Junction to Ambient (θ _{JA})	180°C/W
Junction to Case (θ _{JC})	150°C/W

Package thermal resistances were obtained using the method described in JEDEC specification JESD51-7, using a four-layer board. For detailed information on package thermal considerations, refer to [www.maximintegrated.com/thermal-tutorial](#).

For the latest package outline information and land patterns (footprints), go to [www.maximintegrated.com/packages](#). Note that a "+", "#", or "-" in the package code indicates RoHS status only. Package drawings may show a different suffix character, but the drawing pertains to the package regardless of RoHS status.

Electrical Characteristics

(V_{DD} = 1.8V, V_{LED+} = 5.0V, T_A = +25°C, min/max are from T_A = -40°C to +85°C, unless otherwise noted. Typical values are at T_A = 25°C.) (Note 1)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
POWER SUPPLY						
Power-Supply Voltage	V _{DD}	Guaranteed by RED and IR count tolerance	1.7	1.8	2.0	V
LED Supply Voltage V _{LED+} to PGND	V _{LED+}	Guaranteed by PSRR of LED driver (RED and IR LED only)	3.1	3.3	5.0	V
		Guaranteed by PSRR of LED driver (GREEN LED only). T _A = 25°C	4.5	5.0	5.5	
Supply Current	I _{DD}	SpO ₂ and HR mode, PW = 215μs, 50spS	600	1100		μA
		IR only mode, PW = 215μs, 50spS	600	1100		
Supply Current in Shutdown	I _{SHDN}	T _A = +25°C, MODE = 0x80	0.7	2.5		μA

Electrical Characteristics (continued)

($V_{DD} = 1.8V$, $V_{LED+} = 5.0V$, $T_A = +25^\circ C$, min/max are from $T_A = -40^\circ C$ to $+85^\circ C$, unless otherwise noted. Typical values are at $T_A = 25^\circ C$.) (Note 1)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
PULSE OXIMETRY/HEART-RATE SENSOR CHARACTERISTICS						
ADC Resolution			18			bits
Red ADC Count (Note 2)	REDC	LED1_PA = 0x0C, LED_PW = 0x01, SPO2_SR = 0x05, ADC_RGE = 0x00	65536			Counts
IR ADC Count (Note 2)	IRC	LED2_PA = 0x0C, LED_PW = 0x01, SPO2_SR = 0x05 ADC_RGE = 0x00	65536			Counts
Green ADC Count (Note 2)	GRNC	LED3_PA = LED4_PA = 0x24, LED_PW = 0x01, SPO2_SR = 0x05, ADC_RGE = 0x00	65536			Counts
Dark Current Count	LED_DCC	LED1_PA = LED2_PA = 0x00, LED_PW = 0x03, SPO2_SR = 0x01	30	128		Counts
		ADC_RGE = 0x02	0.01	0.05		% of FS
DC Ambient Light Rejection (Note 3)	ALR	ADC counts with finger on sensor under direct sunlight (100K lux), ADC_RGE = 0x3, LED_PW = 0x03, SPO2_SR = 0x01	Red LED	2		Counts
			IR LED	2		Counts
ADC Count—PSRR (V_{DD})	PSRR _{V_{DD}}	1.7V < V_{DD} < 2.0V, LED_PW = 0x00, SPO2_SR = 0x05	0.25	1		% of FS
		Frequency = DC to 100kHz, 100mV _{P-P}	10			LSB
ADC Count—PSRR (LED Driver Outputs)	PSRR _{LED}	3.1V < V_{LED+} < 5.0V, LED1_PA = LED2_PA = 0x0C, LED_PW = 0x01, SPO2_SR = 0x05	0.05	1		% of FS
		4.5V < V_{LED+} < 5.5V, $T_A = 25^\circ C$ LED3_PA = LED4_PA = 0x24, LED_PW = 0x01, SPO2_SR = 0x05				
		Frequency = DC to 100kHz, 100mV _{P-P}	10			LSB
ADC Clock Frequency	CLK		10.2	10.48	10.8	MHz
ADC Integration Time (Note 3)	INT	LED_PW = 0x00	69	μs		
		LED_PW = 0x01	118			
		LED_PW = 0x02	215			
		LED_PW = 0x03	411			
Slot Timing (Timing Between Sequential Channel Samples; e.g., Red Pulse Rising Edge To IR Pulse Rising Edge)	INT	LED_PW = 0x00	427	μs		
		LED_PW = 0x01	525			
		LED_PW = 0x02	720			
		LED_PW = 0x03	1107			
COVER GLASS CHARACTERISTICS (Note 3)						
Hydrolytic Resistance Class		Per DIN ISO 719		HGB 1		

Electrical Characteristics (continued)

($V_{DD} = 1.8V$, $V_{LED+} = 5.0V$, $T_A = +25^\circ C$, min/max are from $T_A = -40^\circ C$ to $+85^\circ C$, unless otherwise noted. Typical values are at $T_A = 25^\circ C$.) (Note 1)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
IR LED CHARACTERISTICS (Note 3)						
LED Peak Wavelength	λ_P	$I_{LED} = 20mA$, $T_A = +25^\circ C$	870	880	900	nm
Full Width at Half Max	$\Delta\lambda$	$I_{LED} = 20mA$, $T_A = +25^\circ C$		30		nm
Forward Voltage	V_F	$I_{LED} = 20mA$, $T_A = +25^\circ C$		1.4		V
Radiant Power	P_O	$I_{LED} = 20mA$, $T_A = +25^\circ C$		6.5		mW
RED LED CHARACTERISTICS (Note 3)						
LED Peak Wavelength	λ_P	$I_{LED} = 20mA$, $T_A = +25^\circ C$	650	660	670	nm
Full Width at Half Max	$\Delta\lambda$	$I_{LED} = 20mA$, $T_A = +25^\circ C$		20		nm
Forward Voltage	V_F	$I_{LED} = 20mA$, $T_A = +25^\circ C$		2.1		V
Radiant Power	P_O	$I_{LED} = 20mA$, $T_A = +25^\circ C$		9.8		mW
GREEN LED CHARACTERISTICS (Note 3)						
LED Peak Wavelength	λ_P	$I_{LED} = 50mA$, $T_A = +25^\circ C$	530	537	545	nm
Full Width at Half Max	$\Delta\lambda$	$I_{LED} = 50mA$, $T_A = +25^\circ C$		35		nm
Forward Voltage	V_F	$I_{LED} = 50mA$, $T_A = +25^\circ C$		3.3		V
Radiant Power	P_O	$I_{LED} = 50mA$, $T_A = +25^\circ C$		17.2		mW
PHOTODETECTOR CHARACTERISTICS (Note 3)						
Spectral Range of Sensitivity	$\lambda > 30\% QE$	QE: Quantum Efficiency	640	980		nm
Radiant Sensitive Area	A			1.36		mm ²
Dimensions of Radiant Sensitive Area	L x W			1.38 x 0.98		mm x mm
INTERNAL DIE TEMPERATURE SENSOR						
Temperature ADC Acquisition Time	T_T	$T_A = +25^\circ C$		29		ms
Temperature Sensor Accuracy	T_A	$T_A = +25^\circ C$		± 1		°C
Temperature Sensor Minimum Range	T_{MIN}			-40		°C
Temperature Sensor Maximum Range	T_{MAX}			85		°C
DIGITAL INPUTS (SCL, SDA)						
Input Logic-Low Voltage	V_{IL}			0.3 x V_{DD}		V
Input Logic-High Voltage	V_{IH}			0.7 x V_{DD}		V
Input Hysteresis	V_{HYS}			0.5 x V_{DD}		V
Input Leakage Current	I_{IN}			± 0.1	± 1	µA
Input Capacitance	C_{IN}			10		pF
DIGITAL OUTPUTS (SDA, INT)						
Output Low Voltage	V_{OL}	$I_{SINK} = 3mA$		0.4		V

Electrical Characteristics (continued)

($V_{DD} = 1.8V$, $V_{LED+} = 5.0V$, $T_A = +25^\circ C$, min/max are from $T_A = -40^\circ C$ to $+85^\circ C$, unless otherwise noted. Typical values are at $T_A = 25^\circ C$.) (Note 1)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
I²C TIMING CHARACTERISTICS						
I ² C Write Address				AE		Hex
I ² C Read Address				AF		Hex
SCL Clock Frequency	f _{SCL}	Lower limit not tested	0	400		kHz
Bus Free Time Between STOP and START Condition	t _{BUF}		1.3			μs
Hold Time (Repeated) START Condition	t _{HD,STA}		0.6			μs
SCL Pulse-Width Low	t _{LOW}		1.3			μs
SCL Pulse-Width High	t _{HIGH}		0.6			μs
Setup Time for a Repeated START Condition	t _{SU,STA}		0.6			μs
Data Hold Time	t _{HD;DAT}		0	0.9		μs
Data Setup Time	t _{SU;DAT}		100			ns
Setup Time for STOP Condition	t _{SU;STO}		0.6			μs
Pulse Width of Suppressed Spike	t _{SP}			50		ns
Bus Capacitance	C _b			400		pF
SDA and SCL Receiving Rise Time	T _r	(Note 4)	20	300		ns
SDA and SCL Receiving Fall Time	t _{Rf}	(Note 4)	20 x V _{DD} /5.5	300		ns
SDA Transmitting Fall Time	t _{of}		20 x V _{DD} /5.5	250		ns

Note 1: All devices are 100% production tested at $T_A = +25^\circ C$. Specifications over temperature limits are guaranteed by Maxim Integrated's bench or proprietary automated test equipment (ATE) characterization.

Note 2: Specifications are guaranteed by Maxim Integrated's bench characterization and by 100% production test using proprietary ATE setup and conditions.

Note 3: For design guidance only. Not production tested.

Note 4: These specifications are guaranteed by design, characterization, or I²C protocol.

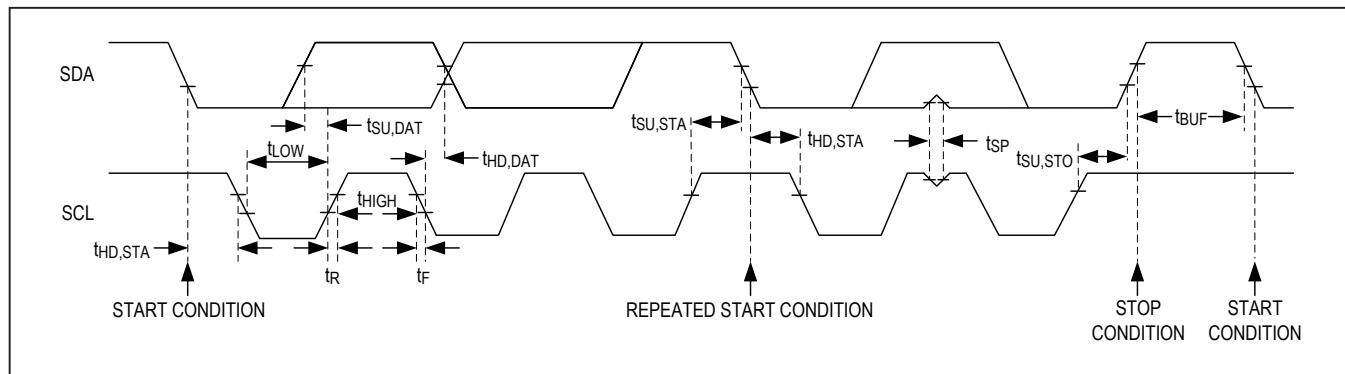
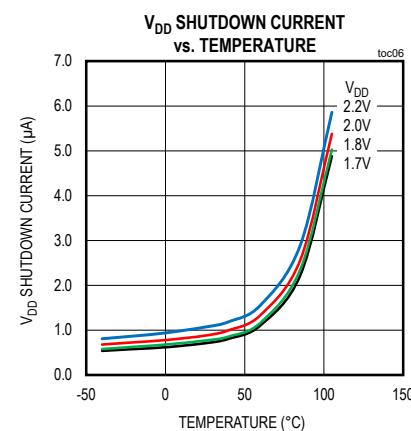
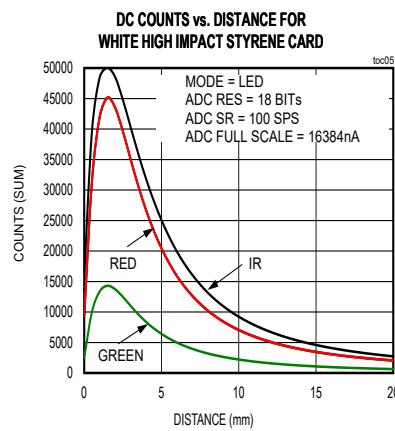
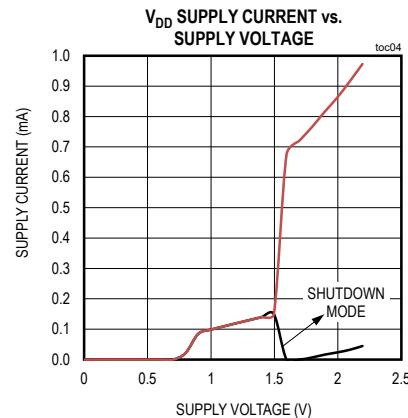
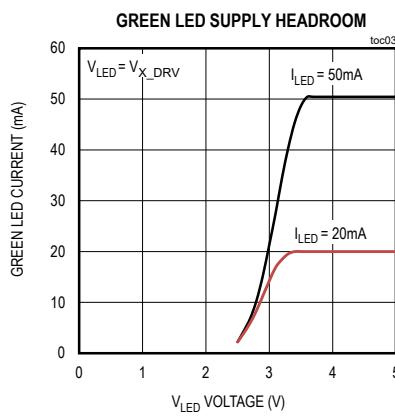
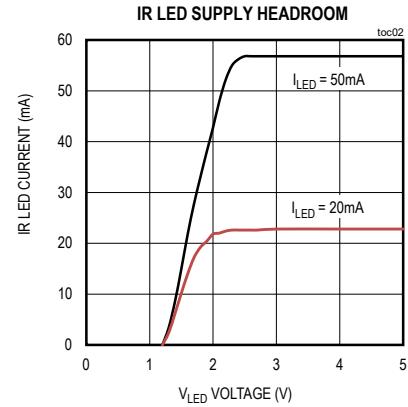
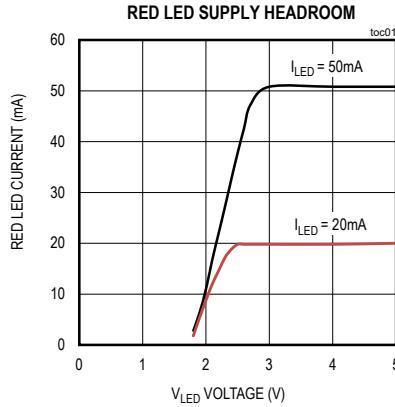
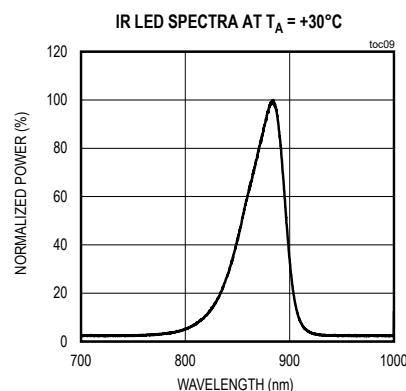
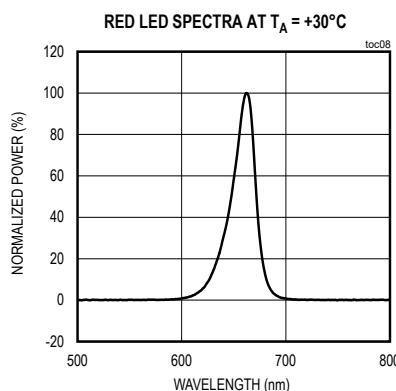
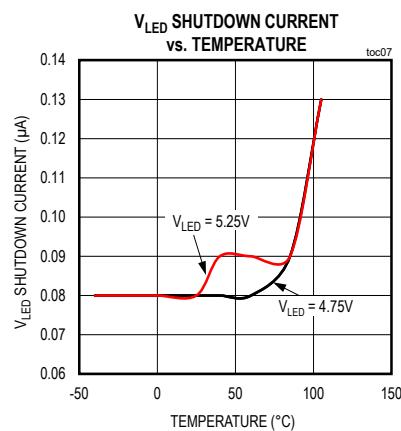
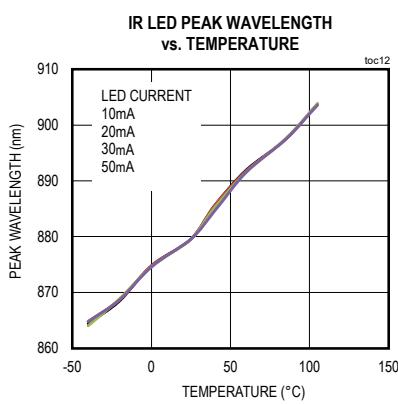
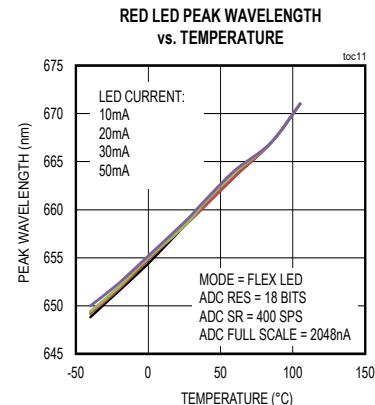
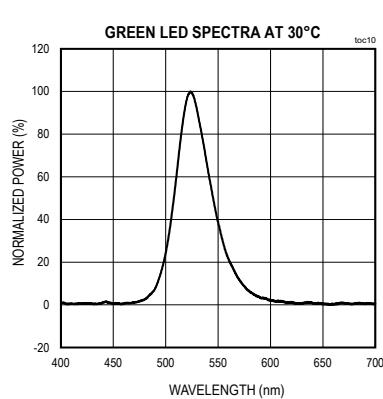
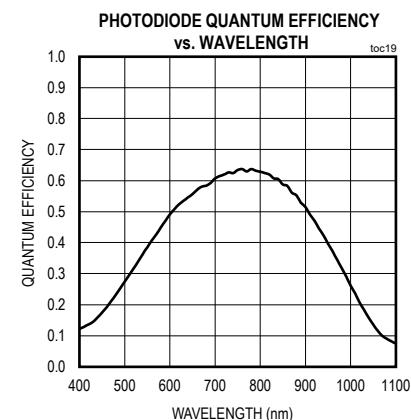
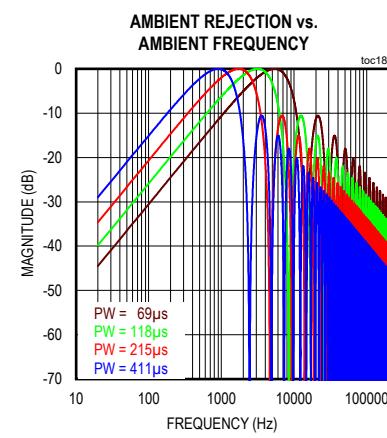
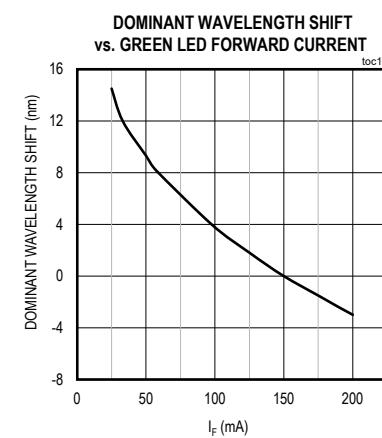
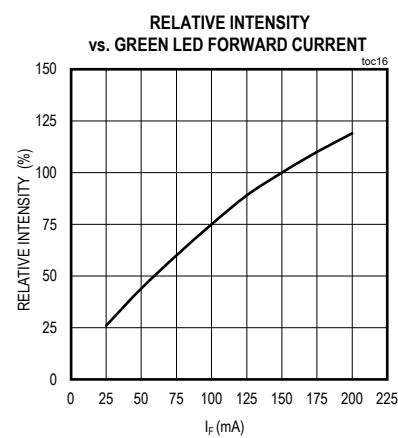
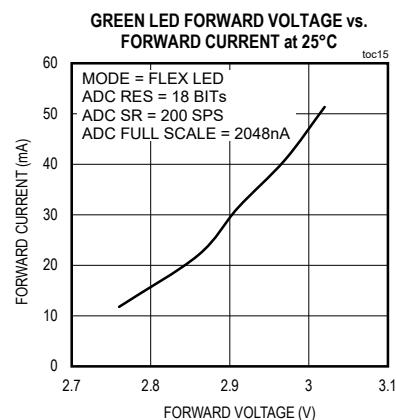
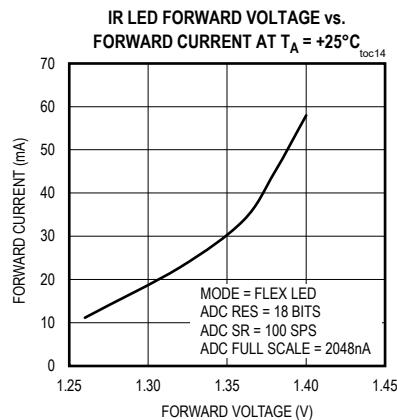
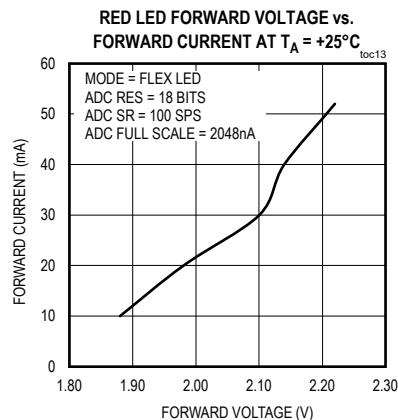
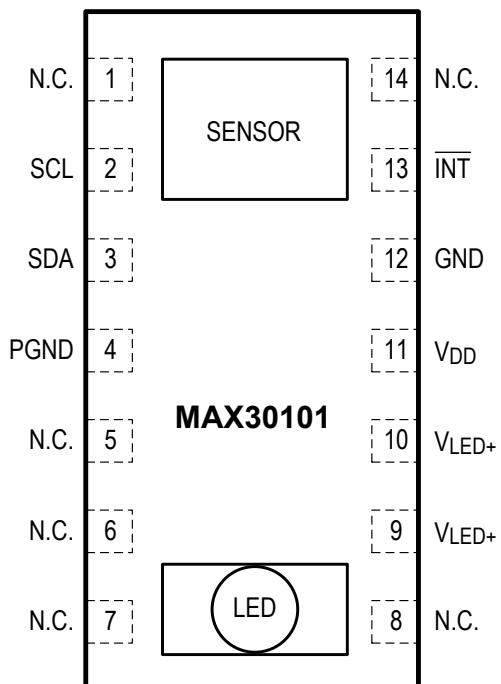


Figure 1. I²C-Compatible Interface Timing Diagram

Typical Operating Characteristics(V_{DD} = 1.8V, V_{LED+} = 5.0V, T_A = +25°C, RST, unless otherwise noted.)

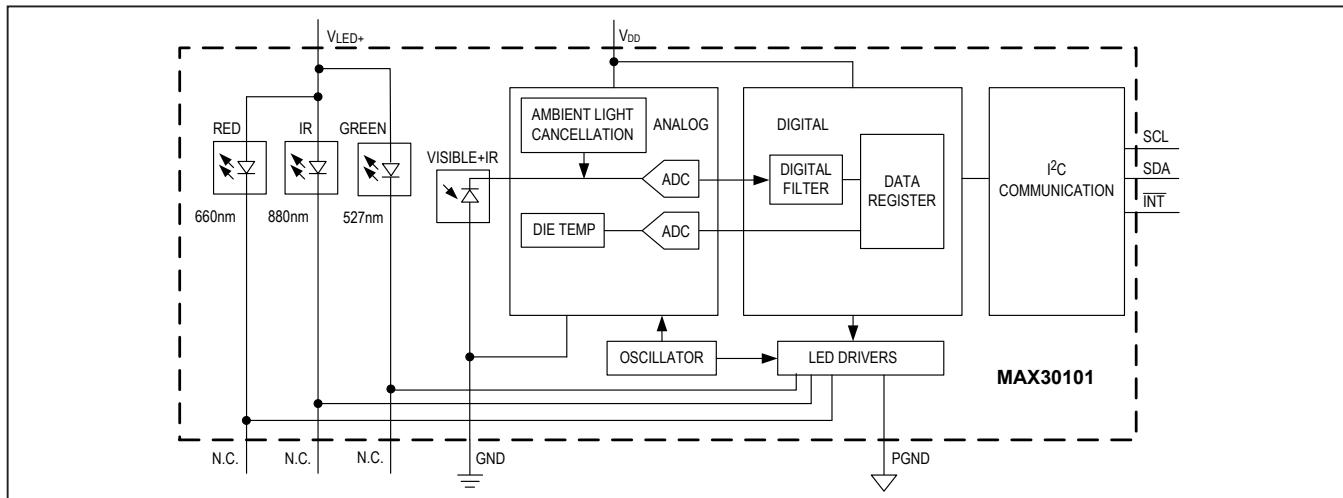
Typical Operating Characteristics (continued)(V_{DD} = 1.8V, V_{LED+} = 5.0V, T_A = +25°C, RST, unless otherwise noted.)

Typical Operating Characteristics (continued)(V_{DD} = 1.8V, V_{LED+} = 5.0V, T_A = +25°C, RST, unless otherwise noted.)

Pin Configuration**Pin Description**

PIN	NAME	FUNCTION
1, 5, 6, 7, 8, 14	N.C.	No Connection. Connect to PCB pad for mechanical stability.
2	SCL	I ² C Clock Input
3	SDA	I ² C Clock Data, Bidirectional (Open-Drain)
4	PGND	Power Ground of the LED Driver Blocks
9	V _{LED+}	LED Power Supply (anode connection). Use a bypass capacitor to PGND for best performance.
10	V _{LED+}	Analog Power Supply Input. Use a bypass capacitor to GND for best performance.
11	V _{DD}	
12	GND	Analog Ground
13	INT	Active-Low Interrupt (Open-Drain). Connect to an external voltage with a pullup resistor.

Functional Diagram



Detailed Description

The MAX30101 is a complete pulse oximetry and heart-rate sensor system solution module designed for the demanding requirements of wearable devices. The MAX30101 maintains a very small solution size without sacrificing optical or electrical performance. Minimal external hardware components are required for integration into a wearable system.

The MAX30101 is fully adjustable through software registers, and the digital output data can be stored in a 32-deep FIFO within the IC. The FIFO allows the MAX30101 to be connected to a microcontroller or processor on a shared bus, where the data is not being read continuously from the MAX30101's registers.

SpO₂ Subsystem

The SpO₂ subsystem contains ambient light cancellation (ALC), a continuous-time sigma-delta ADC, and proprietary discrete time filter. The ALC has an internal Track/Hold circuit to cancel ambient light and increase the effective dynamic range. The SpO₂ ADC has a programmable full-scale ranges from 2µA to 16µA. The ALC can cancel up to 200µA of ambient current.

The internal ADC is a continuous time oversampling sigma-delta converter with 18-bit resolution. The ADC sampling rate is 10.24MHz. The ADC output data rate can be programmed from 50sps (samples per second) to 3200sps.

Temperature Sensor

The MAX30101 has an on-chip temperature sensor for calibrating the temperature dependence of the SpO₂ subsystem. The temperature sensor has an inherent resolution 0.0625°C.

The device output data is relatively insensitive to the wavelength of the IR LED, where the red LED's wavelength is critical to correct interpretation of the data. An SpO₂ algorithm used with the MAX30101 output signal can compensate for the associated SpO₂ error with ambient temperature changes.

LED Driver

The MAX30101 integrates red, green, and IR LED drivers to modulate LED pulses for SpO₂ and HR measurements. The LED current can be programmed from 0 to 50mA with proper supply voltage. The LED pulse width can be programmed from 69µs to 411µs to allow the algorithm to optimize SpO₂ and HR accuracy and power consumption based on use cases.

Register Maps and Descriptions

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W					
STATUS																
Interrupt Status 1	A_FULL	PPG_RDY	ALC_OVF					PWR_RDY	0x00	0X00	R					
Interrupt Status 2							DIE_TEMP_RDY		0x01	0x00	R					
Interrupt Enable 1	A_FULL_EN	PPG_RDY_EN	ALC_OVF_EN						0x02	0X00	R/W					
Interrupt Enable 2							DIE_TEMP_RDY_EN		0x03	0x00	R/W					
FIFO																
FIFO Write Pointer					FIFO_WR_PTR[4:0]				0x04	0x00	R/W					
Overflow Counter					OVF_COUNTER[4:0]				0x05	0x00	R/W					
FIFO Read Pointer					FIFO_RD_PTR[4:0]				0x06	0x00	R/W					
FIFO Data Register	FIFO_DATA[7:0]								0x07	0x00	R/W					
CONFIGURATION																
FIFO Configuration	SMP_AVE[2:0]			FIFO_ROLL_OVER_EN	FIFO_A_FULL[3:0]				0x08	0x00	R/W					
Mode Configuration	SHDN	RESET				MODE[2:0]			0x09	0x00	R/W					
SpO ₂ Configuration	0 (Reserved)	SPO2_ADC_RGE[1:0]		SPO2_SR[2:0]			LED_PW[1:0]		0xA0	0x00	R/W					
RESERVED									0xB0	0x00	R/W					
LED Pulse Amplitude	LED1_PA[7:0]								0xC0	0x00	R/W					
	LED2_PA[7:0]								0xD0	0x00	R/W					
	LED3_PA[7:0]								0xE0	0x00	R/W					
	LED4_PA[7:0]								0xF0	0x00	R/W					
Multi-LED Mode Control Registers		SLOT2[2:0]				SLOT1[2:0]			0x11	0x00	R/W					
		SLOT4[2:0]				SLOT3[2:0]			0x12	0x00	R/W					

Register Maps and Descriptions (continued)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
RESERVED									0x13– 0x17	0xFF	R/W
RESERVED									0x18– 0x1E	0x00	R
DIE TEMPERATURE											
Die Temp Integer					TINT[7:0]				0x1F	0x00	R
Die Temp Fraction						TFRAC[3:0]			0x20	0x00	R
Die Temperature Config								TEMP _EN	0x21	0x00	R/W
RESERVED									0x22– 0x2F	0x00	R/W
PART ID											
Revision ID					REV_ID[7:0]				0xFE	0xXX*	R
Part ID					PART_ID[7]				0xFF	0x15	R

*XX denotes a 2-digit hexadecimal number (00 to FF) for part revision identification. Contact Maxim Integrated for the revision ID number assigned for your product.

Interrupt Status (0x00–0x01)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
Interrupt Status 1	A_FULL	PPG_RDY	ALC_OVF					PWR_RDY	0x00	0X00	R
Interrupt Status 2							DIE_TEMP_RDY		0x01	0x00	R

Whenever an interrupt is triggered, the MAX30101 pulls the active-low interrupt pin into its low state until the interrupt is cleared.

A_FULL: FIFO Almost Full Flag

In SpO₂ and HR modes, this interrupt triggers when the FIFO write pointer has a certain number of free spaces remaining. The trigger number can be set by the FIFO_A_FULL[3:0] register. The interrupt is cleared by reading the Interrupt Status 1 register (0x00).

PPG_RDY: New FIFO Data Ready

In SpO₂ and HR modes, this interrupt triggers when there is a new sample in the data FIFO. The interrupt is cleared by reading the Interrupt Status 1 register (0x00), or by reading the FIFO_DATA register.

ALC_OVF: Ambient Light Cancellation Overflow

This interrupt triggers when the ambient light cancellation function of the SpO₂/HR photodiode has reached its maximum limit, and therefore, ambient light is affecting the output of the ADC. The interrupt is cleared by reading the Interrupt Status 1 register (0x00).

PWR_RDY: Power Ready Flag

On power-up or after a brownout condition, when the supply voltage V_{DD} transitions from below the undervoltage lockout (UVLO) voltage to above the UVLO voltage, a power-ready interrupt is triggered to signal that the module is powered-up and ready to collect data.

DIE_TEMP_RDY: Internal Temperature Ready Flag

When an internal die temperature conversion is finished, this interrupt is triggered so the processor can read the temperature data registers. The interrupt is cleared by reading either the Interrupt Status 2 register (0x01) or the TFRAC register (0x20).

The interrupts are cleared whenever the interrupt status register is read, or when the register that triggered the interrupt is read. For example, if the SpO₂ sensor triggers an interrupt due to finishing a conversion, reading either the FIFO data register or the interrupt register clears the interrupt pin (which returns to its normal HIGH state). This also clears all the bits in the interrupt status register to zero.

Interrupt Enable (0x02-0x03)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
Interrupt Enable 1	A_FULL_EN	PPG_RDY_EN	ALC_OVF_EN						0x02	0X00	R/W
Interrupt Enable 2							DIE_TEMP_RDY_EN		0x03	0x00	R/W

Each source of hardware interrupt, with the exception of power ready, can be disabled in a software register within the MAX30101 IC. The power-ready interrupt cannot be disabled because the digital state of the module is reset upon a brownout condition (low power supply voltage), and the default condition is that all the interrupts are disabled. Also, it is important for the system to know that a brownout condition has occurred, and the data within the module is reset as a result.

The unused bits should always be set to zero for normal operation.

FIFO (0x04–0x07)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W	
FIFO Write Pointer				FIFO_WR_PTR[4:0]						0x04	0x00	R/W
Over Flow Counter				OVF_COUNTER[4:0]						0x05	0x00	R/W
FIFO Read Pointer				FIFO_RD_PTR[4:0]						0x06	0x00	R/W
FIFO Data Register	FIFO_DATA[7:0]								0x07	0x00	R/W	

FIFO Write Pointer

The FIFO Write Pointer points to the location where the MAX30101 writes the next sample. This pointer advances for each sample pushed on to the FIFO. It can also be changed through the I²C interface when MODE[2:0] is 010, 011, or 111.

FIFO Overflow Counter

When the FIFO is full, samples are not pushed on to the FIFO, samples are lost. OVF_COUNTER counts the number of samples lost. It saturates at 0x1F. When a complete sample is “popped” (i.e., removal of old FIFO data and shifting the samples down) from the FIFO (when the read pointer advances), OVF_COUNTER is reset to zero.

FIFO Read Pointer

The FIFO Read Pointer points to the location from where the processor gets the next sample from the FIFO through the I²C interface. This advances each time a sample is popped from the FIFO. The processor can also write to this pointer after reading the samples to allow rereading samples from the FIFO if there is a data communication error.

FIFO Data Register

The circular FIFO depth is 32 and can hold up to 32 samples of data. The sample size depends on the number of LED channels (a.k.a. channels) configured as active. As each channel signal is stored as a 3-byte data signal, the FIFO width can be 3 bytes, 6 bytes, 9 bytes, or 12 bytes in size.

The FIFO_DATA register in the I²C register map points to the next sample to be read from the FIFO. FIFO_RD_PTR points to this sample. Reading FIFO_DATA register, does not automatically increment the I²C register address. Burst reading this register, reads the same address over and over. Each sample is 3 bytes of data per channel (i.e., 3 bytes for RED, 3 bytes for IR, etc.).

The FIFO registers (0x04–0x07) can all be written and read, but in practice only the FIFO_RD_PTR register should be written to in operation. The others are automatically incremented or filled with data by the MAX30101. When starting a new SpO₂ or heart rate conversion, it is recommended to first clear the FIFO_WR_PTR, OVF_COUNTER, and FIFO_RD_PTR registers to all zeroes (0x00) to ensure the FIFO is empty and in a known state. When reading the MAX30101 registers in one burst-read I²C transaction, the register address pointer typically increments so that the next byte of data sent is from the next register, etc. The exception to this is the FIFO data register, register 0x07. When reading this register, the address pointer does not increment, but the FIFO_RD_PTR does. So the next byte of data sent represents the next byte of data available in the FIFO.

Reading from the FIFO

Normally, reading registers from the I²C interface autoincrements the register address pointer, so that all the registers can be read in a burst read without an I²C start event. In the MAX30101, this holds true for all registers except for the FIFO_DATA register (register 0x07).

Reading the FIFO_DATA register does not automatically increment the register address. Burst reading this register reads data from the same address over and over. Each sample comprises multiple bytes of data, so multiple bytes should be read from this register (in the same transaction) to get one full sample.

The other exception is 0xFF. Reading more bytes after the 0xFF register does not advance the address pointer back to 0x00, and the data read is not meaningful.

FIFO Data Structure

The data FIFO consists of a 32-sample memory bank that can store GREEN, IR, and RED ADC data. Since each sample consists of three channels of data, there are 9 bytes of data for each sample, and therefore 288 total bytes of data can be stored in the FIFO.

The FIFO data is left-justified, as shown in [Table 1](#); in other words, the MSB bit is always in the bit 17 data position, regardless of ADC resolution setting. See [Table 2](#) for a visual presentation of the FIFO data structure.

Table 1. FIFO Data is Left-Justified

ADC Resolution	FIFO_DATA[17]	FIFO_DATA[16]	...	FIFO_DATA[12]	FIFO_DATA[11]	FIFO_DATA[10]	FIFO_DATA[9]	FIFO_DATA[8]	FIFO_DATA[7]	FIFO_DATA[6]	FIFO_DATA[5]	FIFO_DATA[4]	FIFO_DATA[3]	FIFO_DATA[2]	FIFO_DATA[1]	FIFO_DATA[0]
18-bit																
17-bit																
16-bit																
15-bit																

FIFO Data Contains 3 Bytes per Channel

The FIFO data is left-justified, meaning that the MSB is always in the same location regardless of the ADC resolution setting. FIFO DATA[18] – [23] are not used. [Table 2](#) shows the structure of each triplet of bytes (containing the 18-bit ADC data output of each channel).

Each data sample in SpO₂ mode comprises two data triplets (3 bytes each). To read one sample, requires an I²C read command for each byte. Thus, to read one sample in SpO₂ mode, requires 6 I²C byte reads. To read one sample with three LED channels requires 9 I²C byte reads. The FIFO read pointer is automatically incremented after the first byte of each sample is read.

Write/Read Pointers

Write/Read pointers are used to control the flow of data in the FIFO. The write pointer increments every time a new sample is added to the FIFO. The read pointer is incremented every time a sample is read from the FIFO. To reread a sample from the FIFO, decrement its value by one and read the data register again.

The FIFO write/read pointers should be cleared (back to 0x00) upon entering SpO₂ mode or HR mode, so that there is no old data represented in the FIFO. The pointers are automatically cleared if V_{DD} is power-cycled or V_{DD} drops below its UVLO voltage.

Table 2. FIFO Data (3 Bytes per Channel)

BYTE 1							FIFO_DATA[17]	FIFO_DATA[16]
BYTE 2	FIFO_DATA[15]	FIFO_DATA[14]	FIFO_DATA[13]	FIFO_DATA[12]	FIFO_DATA[11]	FIFO_DATA[10]	FIFO_DATA[9]	FIFO_DATA[8]
BYTE 3	FIFO_DATA[7]	FIFO_DATA[6]	FIFO_DATA[5]	FIFO_DATA[4]	FIFO_DATA[3]	FIFO_DATA[2]	FIFO_DATA[1]	FIFO_DATA[0]

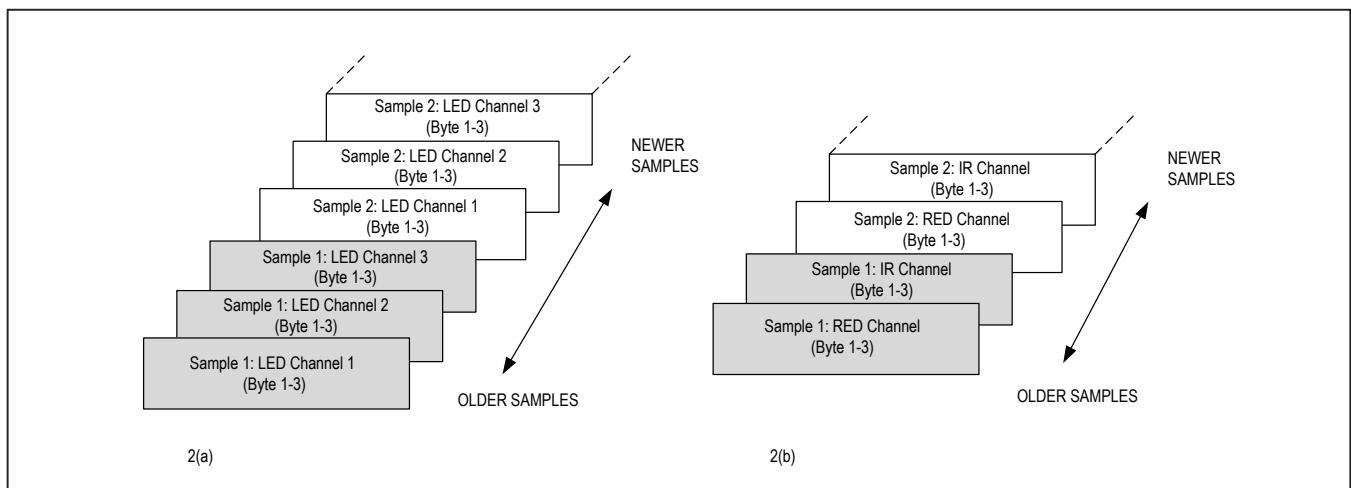


Figure 2a and 2b. Graphical Representation of the FIFO Data Register. The left shows three LEDs in multi-LED mode, and the right shows IR and Red only in SpO₂ Mode.

Pseudo-Code Example of Reading Data from FIFO

First transaction: Get the FIFO_WR_PTR:

```
START;  
Send device address + write mode  
Send address of FIFO_WR_PTR;  
REPEATED_START;  
Send device address + read mode  
Read FIFO_WR_PTR;  
STOP;
```

The central processor evaluates the number of samples to be read from the FIFO:

```
NUM_AVAILABLE_SAMPLES = FIFO_WR_PTR - FIFO_RD_PTR  
(Note: pointer wrap around should be taken into account)  
NUM_SAMPLES_TO_READ = < less than or equal to NUM_AVAILABLE_SAMPLES >
```

Second transaction: Read NUM_SAMPLES_TO_READ samples from the FIFO:

```
START;  
Send device address + write mode  
Send address of FIFO_DATA;  
REPEATED_START;  
Send device address + read mode  
for (i = 0; i < NUM_SAMPLES_TO_READ; i++) {  
    Read FIFO_DATA;  
    Save LED1[23:16];  
    Read FIFO_DATA;  
    Save LED1[15:8];  
    Read FIFO_DATA;  
    Save LED1[7:0];  
    Read FIFO_DATA;  
    Save LED2[23:16];  
    Read FIFO_DATA;  
    Save LED2[15:8];  
    Read FIFO_DATA;  
    Save LED2[7:0];  
    Read FIFO_DATA;  
    Save LED3[23:16];  
    Read FIFO_DATA;  
    Save LED3[15:8];  
    Read FIFO_DATA;  
    Save LED3[7:0];  
    Read FIFO_DATA;  
}  
STOP;
```

```

START;
Send device address + write mode
Send address of FIFO_RD_PTR;
Write FIFO_RD_PTR;
STOP;

```

Third transaction: Write to FIFO_RD_PTR register. If the second transaction was successful, FIFO_RD_PTR points to the next sample in the FIFO, and this third transaction is not necessary. Otherwise, the processor updates the FIFO_RD_PTR appropriately, so that the samples are reread.

FIFO Configuration (0x08)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
FIFO Configuration		SMP_AVE[2:0]		FIFO_ROL LOVER_EN		FIFO_A_FULL[3:0]			0x08	0x00	R/W

Bits 7:5: Sample Averaging (SMP_AVE)

To reduce the amount of data throughput, adjacent samples (in each individual channel) can be averaged and decimated on the chip by setting this register.

Table 3. Sample Averaging

SMP_AVE[2:0]	NO. OF SAMPLES AVERAGED PER FIFO SAMPLE
000	1 (no averaging)
001	2
010	4
011	8
100	16
101	32
110	32
111	32

Bit 4: FIFO Rolls on Full (FIFO_ROLLOVER_EN)

This bit controls the behavior of the FIFO when the FIFO becomes completely filled with data. If FIFO_ROLLOVER_EN is set (1), the FIFO Address rolls over to zero and the FIFO continues to fill with new data. If the bit is not set (0), then the FIFO is not updated until FIFO_DATA is read or the WRITE/READ pointer positions are changed.

Bits 3:0: FIFO Almost Full Value (FIFO_A_FULL)

This register sets the number of data samples (3 bytes/sample) remaining in the FIFO when the interrupt is issued. For example, if this field is set to 0x0, the interrupt is issued when there is 0 data samples remaining in the FIFO (all 32 FIFO words have unread data). Furthermore, if this field is set to 0xF, the interrupt is issued when 15 data samples are remaining in the FIFO (17 FIFO data samples have unread data).

FIFO_A_FULL[3:0]	EMPTY DATA SAMPLES IN FIFO WHEN INTERRUPT IS ISSUED	UNREAD DATA SAMPLES IN FIFO WHEN INTERRUPT IS ISSUED
0x0h	0	32
0x1h	1	31
0x2h	2	30
0x3h	3	29
...
0xFh	15	17

Mode Configuration (0x09)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
Mode Configuration	SHDN	RESET						MODE[2:0]	0x09	0x00	R/W

Bit 7: Shutdown Control (SHDN)

The part can be put into a power-save mode by setting this bit to one. While in power-save mode, all registers retain their values, and write/read operations function as normal. All interrupts are cleared to zero in this mode.

Bit 6: Reset Control (RESET)

When the RESET bit is set to one, all configuration, threshold, and data registers are reset to their power-on-state through a power-on reset. The RESET bit is cleared automatically back to zero after the reset sequence is completed.

Note: Setting the RESET bit does not trigger a PWR_RDY interrupt event.

Bits 2:0: Mode Control

These bits set the operating state of the MAX30101. Changing modes does not change any other setting, nor does it erase any previously stored data inside the data registers.

Table 4. Mode Control

MODE[2:0]	MODE	ACTIVE LED CHANNELS
000	Do not use	
001	Do not use	
010	Heart Rate mode	Red only
011	SpO ₂ mode	Red and IR
100–110	Do not use	
111	Multi-LED mode	Green, Red, and/or IR

SpO₂ Configuration (0x0A)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
SpO ₂ Configuration		SPO ₂ _ADC_RGE[1:0]		SPO ₂ _SR[2:0]		LED_PW[1:0]			0x0A	0x00	R/W

Bits 6:5: SpO₂ ADC Range Control

This register sets the SpO₂ sensor ADC's full-scale range as shown in [Table 5](#).

Table 5. SpO₂ ADC Range Control (18-Bit Resolution)

SPO ₂ _ADC_RGE[1:0]	LSB SIZE (pA)	FULL SCALE (nA)
00	7.81	2048
01	15.63	4096
02	31.25	8192
03	62.5	16384

Bits 4:2: SpO₂ Sample Rate Control

These bits define the effective sampling rate with one sample consisting of one IR pulse/conversion and one RED pulse/conversion.

The sample rate and pulse width are related in that the sample rate sets an upper bound on the pulse width time. If the user selects a sample rate that is too high for the selected LED_PW setting, the highest possible sample rate is programmed instead into the register.

Table 6. SpO₂ Sample Rate Control

SPO ₂ _SR[2:0]	SAMPLES PER SECOND
000	50
001	100
010	200
011	400
100	800
101	1000
110	1600
111	3200

See [Table 11](#) and [Table 12](#) for Pulse Width vs. Sample Rate information.

Bits 1:0: LED Pulse Width Control and ADC Resolution

These bits set the LED pulse width (the IR, Red, and Green have the same pulse width), and, therefore, indirectly sets the integration time of the ADC in each sample. The ADC resolution is directly related to the integration time.

Table 7. LED Pulse Width Control

LED_PW[1:0]	PULSE WIDTH (μs)	ADC RESOLUTION (bits)
00	69 (68.95)	15
01	118 (117.78)	16
10	215 (215.44)	17
11	411 (410.75)	18

LED Pulse Amplitude (0x0C–0x0F)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
LED Pulse Amplitude									0x0C	0x00	R/W
					LED1_PA[7:0]				0x0D	0x00	R/W
					LED2_PA[7:0]				0x0E	0x00	R/W
					LED3_PA[7:0]				0x0F	0x00	R/W
					LED4_PA[7:0]						

These bits set the current level of each LED as shown in [Table 8](#).

Table 8. LED Current Control

LEDx_PA [7:0]	TYPICAL LED CURRENT (mA)*
0x00h	0.0
0x01h	0.2
0x02h	0.4
...	...
0x0Fh	3.0
...	...
0x1Fh	6.2
...	...
0x3Fh	12.6
...	...
0x7Fh	25.4
...	...
0xFFh	51.0

*Actual measured LED current for each part can vary significantly due to the trimming methodology.

Multi-LED Mode Control Registers (0x11–0x12)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
Multi-LED Mode Control Registers			SLOT2[2:0]			SLOT1[2:0]			0x11	0x00	R/W
			SLOT4[2:0]			SLOT3[2:0]			0x12	0x00	R/W

In multi-LED mode, each sample is split into up to four time slots, SLOT1 through SLOT4. These control registers determine which LED is active in each time slot, making for a very flexible configuration.

Table 9. Multi-LED Mode Control Registers

SLOTx[2:0] Setting	WHICH LED IS ACTIVE	LED PULSE AMPLITUDE SETTING
000	None (time slot is disabled)	N/A (Off)
001	LED1 (RED)	LED1_PA[7:0]
010	LED2 (IR)	LED2_PA[7:0]
011*	LED3 (GREEN)	LED3_PA[7:0]
	LED4 (GREEN)	LED4_PA[7:0]
100	None	N/A (Off)
101	RESERVED	RESERVED
110	RESERVED	RESERVED
111	RESERVED	RESERVED

Each slot generates a 3-byte output into the FIFO. One sample comprises all active slots, for example if SLOT1 and SLOT2 are non-zero, then one sample is $2 \times 3 = 6$ bytes. If SLOT1 through SLOT3 are all non-zero, then one sample is $3 \times 3 = 9$ bytes.

The slots should be enabled in order (i.e., SLOT1 should not be disabled if SLOT2 or SLOT3 are enabled).

*Both LED3 and LED4 are wired to Green LED. Green LED sinks current out of LED3_PA[7:0] and LED4_PA[7:0] configurationin Multi-LED Mode and SLOTx[2:0] = 011.

Temperature Data (0x1F–0x21)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
Temp_Integer	TINT[7]								0x1F	0x00	R/W
Temp_Fraction	TFRAC[3:0]								0x20	0x00	R/W
Die Temperature Config								TEMP_EN	0x21	0x00	R/W

Temperature Integer

The on-board temperature ADC output is split into two registers, one to store the integer temperature and one to store the fraction. Both should be read when reading the temperature data, and the equation below shows how to add the two registers together:

$$T_{\text{MEASURED}} = T_{\text{INTEGER}} + T_{\text{FRACTION}}$$

This register stores the integer temperature data in 2's complement format, where each bit corresponds to 1°C.

Table 10. Temperature Integer

REGISTER VALUE (hex)	TEMPERATURE (°C)
0x00	0
0x00	+1
...	...
0x7E	+126
0x7F	+127
0x80	-128
0x81	-127
...	...
0xFE	-2
0xFF	-1

Temperature Fraction

This register stores the fractional temperature data in increments of 0.0625°C. If this fractional temperature is paired with a negative integer, it still adds as a positive fractional value (e.g., -128°C + 0.5°C = -127.5°C).

Temperature Enable (TEMP_EN)

This is a self-clearing bit which, when set, initiates a single temperature reading from the temperature sensor. This bit clears automatically back to zero at the conclusion of the temperature reading when the bit is set to one.

Applications Information

Sampling Rate and Performance

The maximum sample rate for the ADC depends on the selected pulse width, which in turn, determines the ADC resolution. For instance, if the pulse width is set to 69 μ s then the ADC resolution is 15 bits, and all sample rates are selectable. However, if the pulse width is set to 411 μ s, then the samples rates are limited. The allowed sample rates for both SpO₂ and HR Modes are summarized in the [Table 11](#) and [Table 12](#):

Power Considerations

The LED waveforms and their implication for power supply design are discussed in this section.

The LEDs in the MAX30101 are pulsed with a low duty cycle for power savings, and the pulsed currents can cause ripples in the V_{LED+} power supply. To ensure these pulses do not translate into optical noise at the LED outputs, the power supply must be designed to handle these. Ensure that the resistance and inductance from the power supply (battery, DC/DC converter, or LDO) to the pin is much smaller than 1 Ω , and that there is at least 1 μ F of power supply bypass capacitance to a good ground plane. The capacitance should be located as close as physically possible to the IC.

Table 11. SpO₂ Mode (Allowed Settings)

SAMPLES PER SECOND	PULSE WIDTH (μ s)			
	69	118	215	411
50	O	O	O	O
100	O	O	O	O
200	O	O	O	O
400	O	O	O	O
800	O	O	O	
1000	O	O		
1600	O			
3200				
Resolution (bits)	15	16	17	18

Table 12. HR Mode (Allowed Settings)

SAMPLES PER SECOND	PULSE WIDTH (μ s)			
	69	118	215	411
50	O	O	O	O
100	O	O	O	O
200	O	O	O	O
400	O	O	O	O
800	O	O	O	O
1000	O	O	O	O
1600	O	O	O	
3200	O			
Resolution (bits)	15	16	17	18

SpO₂ Temperature Compensation

The MAX30101 has an accurate on-board temperature sensor that digitizes the IC's internal temperature upon command from the I²C master. The temperature has an effect on the wavelength of the red and IR LEDs. While the device output data is relatively insensitive to the wavelength of the IR LED, the red LED's wavelength is critical to correct interpretation of the data.

[Table 13](#) shows the correlation of red LED wavelength versus the temperature of the LED. Since the LED die heats up with a very short thermal time constant (tens of microseconds), the LED wavelength should be calculated according to the current level of the LED and the temperature of the IC. Use [Table 13](#) to estimate the temperature.

Table 13. RED LED Current Settings vs. LED Temperature Rise

RED LED CURRENT SETTING	RED LED DUTY CYCLE (% OF LED PULSE WIDTH TO SAMPLE TIME)	ESTIMATED TEMPERATURE RISE (ADD TO TEMP SENSOR MEASUREMENT) (°C)
00000001 (0.2mA)	8	0.1
11111010 (50mA)	8	2
00000001 (0.2mA)	16	0.3
11111010 (50mA)	16	4
00000001 (0.2mA)	32	0.6
11111010 (50mA)	32	8

Red LED Current Settings vs. LED Temperature Rise

Add this to the module temperature reading to estimate the LED temperature and output wavelength. The LED temperature estimate is valid even with very short pulse widths, due to the fast thermal time constant of the LED.

Interrupt Pin Functionality

The active-low interrupt pin pulls low when an interrupt is triggered. The pin is open-drain, which means it normally requires a pullup resistor or current source to an external voltage supply (up to +5V from GND). The interrupt pin is not designed to sink large currents, so the pullup resistor value should be large, such as 4.7kΩ.

Timing for Measurements and Data Collection

Slot Timing in Multi-LED Modes

The MAX30101 can support up to three LED channels of sequential processing (Red, IR, and Green). In multi-LED modes, a time slot or period exists between active sequential channels. [Table 14](#) displays the four possible channel slot times associated with each pulse width

Table 14. Slot Timing

PULSE-WIDTH SETTING (μs)	CHANNEL SLOT TIMING (TIMING PERIOD BETWEEN PULSES) (μs)	CHANNEL-CHANNEL TIMING (RISING EDGE-TO-RISING EDGE) (μs)
69	358	427
118	407	525
215	505	720
411	696	1107

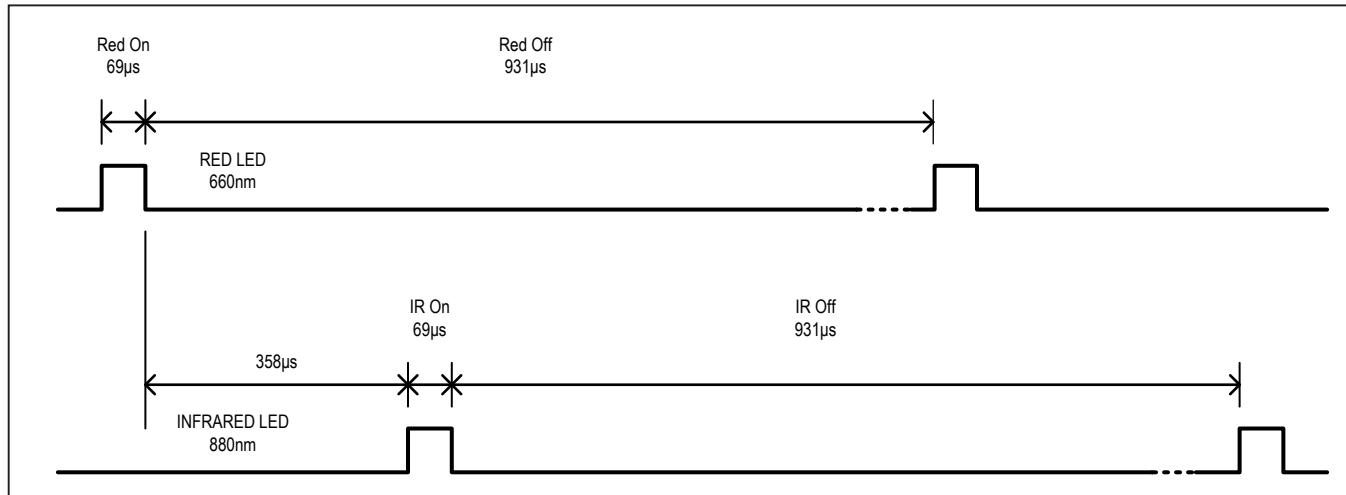


Figure 3. Channel Slot Timing for the SpO₂ Mode with a 1kHz Sample Rate

Timing in SpO₂ Mode

The internal FIFO stores up to 32 samples, so that the system processor does not need to read the data after every sample. SpO₂ can be calibrated using temperature

data. In this case, the temperature does not need to be sampled very often – once a second or every few seconds should be sufficient.

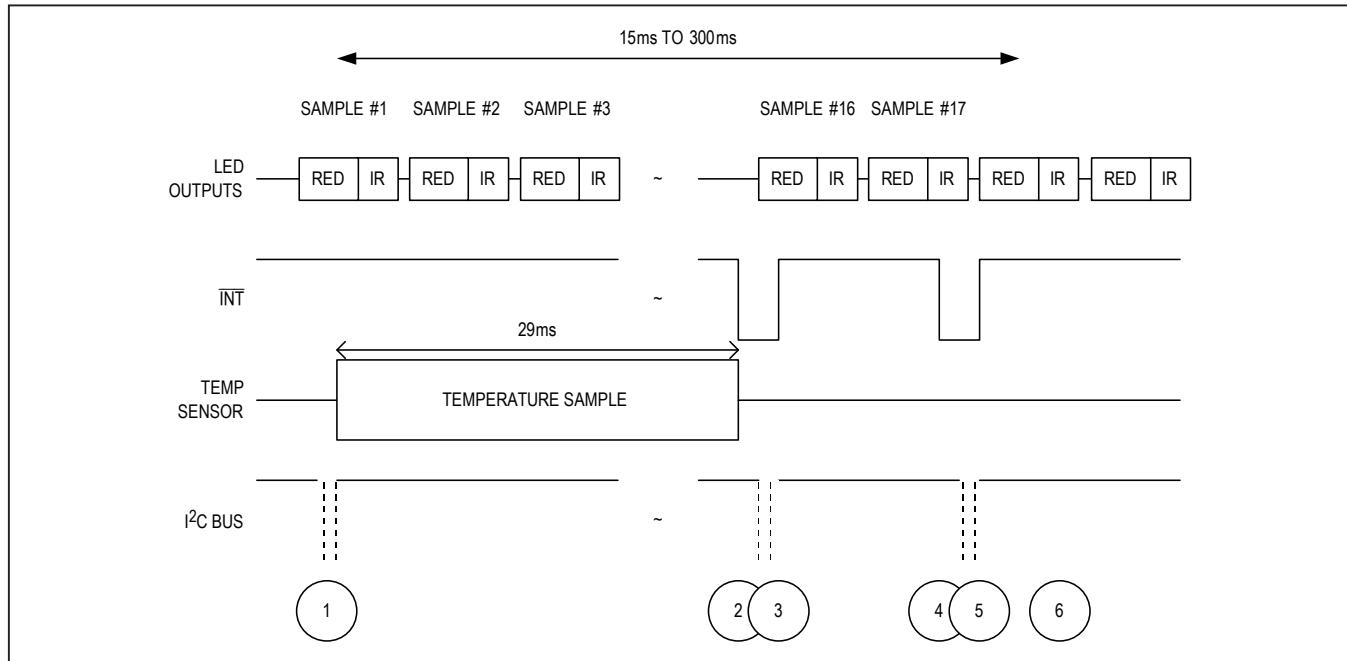


Figure 4. Timing for Data Acquisition and Communication When in SpO₂ Mode

Table 15. Events Sequence for Figure 4 in SpO₂ Mode

EVENT	DESCRIPTION	COMMENTS
1	Enter into SpO ₂ Mode. Initiate a Temperature measurement.	I ² C Write Command sets MODE[2:0] = 0x03 and set A_FULL_EN. Then, to enable and initiate a single temperature measurement, set TEMP_EN and DIE_TEMP_RDY_EN.
2	Temperature Measurement Complete, Interrupt Generated	DIE_TEMP_RDY interrupt triggers, alerting the central processor to read the data.
3	Temp Data is Read, Interrupt Cleared	
4	FIFO is Almost Full, Interrupt Generated	Interrupt is generated when the FIFO almost full threshold is reached.
5	FIFO Data is Read, Interrupt Cleared	
6	Next Sample is Stored	New Sample is stored at the new read pointer location. Effectively, it is now the first sample in the FIFO.

Timing in HR Mode

The internal FIFO stores up to 32 samples, so that the system processor does not need to read the data after every sample. In HR mode (Figure 5), unlike in SpO₂

mode, temperature information is not necessary to interpret the data. The user can select either the Red, IR, or Green LED channel for heart rate.

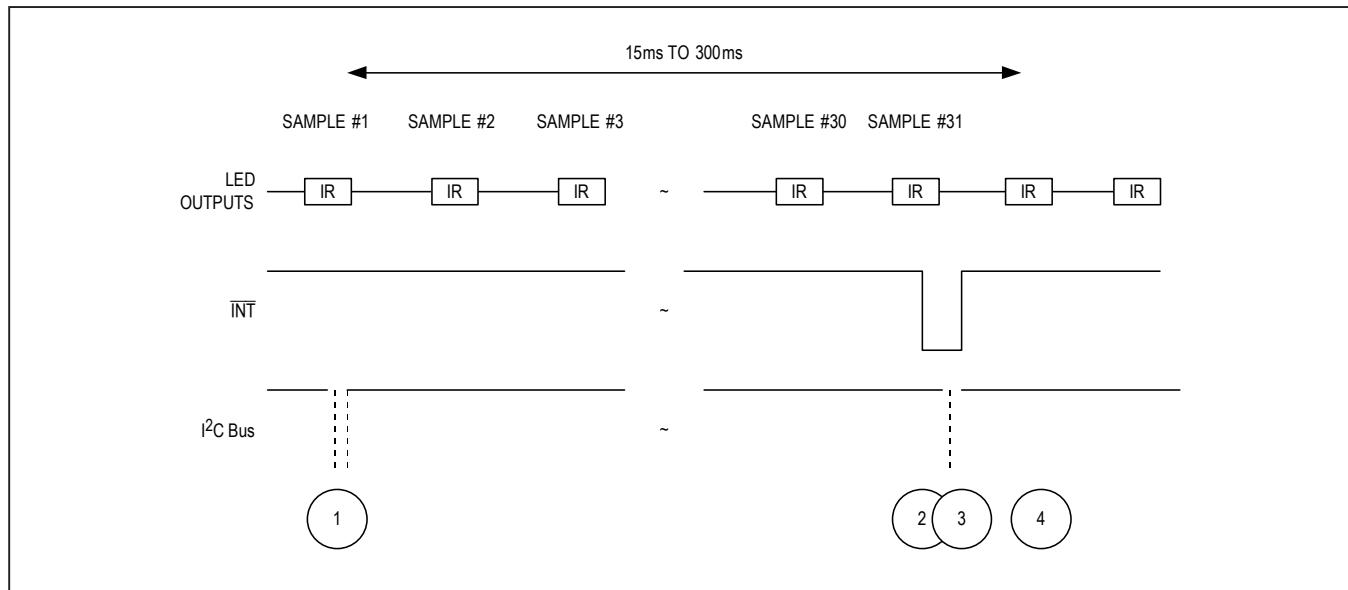


Figure 5. Timing for Data Acquisition and Communication When in HR Mode

Table 16. Events Sequence for Figure 5 in HR Mode

EVENT	DESCRIPTION	COMMENTS
1	Enter into Mode	I ² C Write Command sets MODE[2:0] = 0x02. Mask the A_FULL_EN Interrupt.
2	FIFO is Almost Full, Interrupt Generated	Interrupt is generated when the FIFO has only one empty space left.
3	FIFO Data is Read, Interrupt Cleared	
4	Next Sample is Stored	New sample is stored at the new read pointer location. Effectively, it is now the first sample in the FIFO.

Power Sequencing and Requirements

Power-Up Sequencing

[Figure 6](#) shows the recommended power-up sequence for the MAX30101.

It is recommended to power the V_{DD} supply first, before the LED power supplies (V_{LED+}). The interrupt and I²C pins can be pulled up to an external voltage even when the power supplies are not powered up.

After the power is established, an interrupt occurs to alert the system that the MAX30101 is ready for operation. Reading the I²C interrupt register clears the interrupt, as shown in the [Figure 6](#).

Power-Down Sequencing

The MAX30101 is designed to be tolerant of any power supply sequencing on power-down.

I²C Interface

The MAX30101 features an I²C/SMBus-compatible, 2-wire serial interface consisting of a serial data line (SDA) and a serial clock line (SCL). SDA and SCL facilitate communication between the MAX30101 and the master at clock rates up to 400kHz. [Figure 1](#) shows the 2-wire interface timing diagram. The master generates SCL and initiates data transfer on the bus. The master device writes data to the MAX30101 by transmitting the proper slave address followed by data. Each transmit sequence is framed by a START (S) or REPEATED START (Sr) condition and a STOP (P) condition. Each word transmitted to the MAX30101 is 8 bits long and is followed by an acknowledge clock pulse. A master reading data from the MAX30101 transmits the proper slave address followed by a series of nine SCL pulses.

The MAX30101 transmits data on SDA in sync with the master-generated SCL pulses. The master acknowledges receipt of each byte of data. Each read sequence is framed by a START (S) or REPEATED START (Sr) condition, a not acknowledge, and a STOP (P) condition. SDA operates as both an input and an open-drain output. A pullup resistor, typically greater than 500Ω, is required on SDA. SCL operates only as an input. A pullup resistor, typically greater than 500Ω, is required on SCL if there are multiple masters on the bus, or if the single master has an open-drain SCL output. Series resistors in line with SDA and SCL are optional. Series resistors protect the digital inputs of the MAX30101 from high voltage spikes on the bus lines and minimize crosstalk and undershoot of the bus signals.

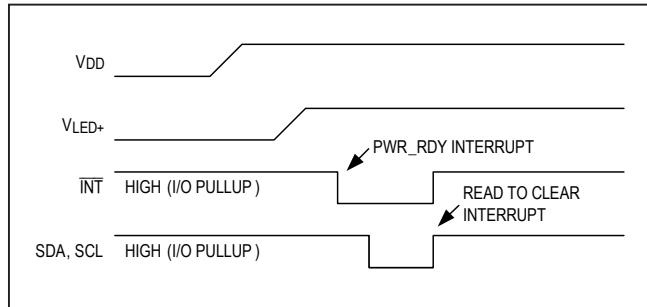


Figure 6. Power-Up Sequence of the Power Supply Rails

Bit Transfer

One data bit is transferred during each SCL cycle. The data on SDA must remain stable during the high period of the SCL pulse. Changes in SDA while SCL is high are control signals. See the [START and STOP Conditions](#) section.

START and STOP Conditions

SDA and SCL idle high when the bus is not in use. A master initiates communication by issuing a START condition. A START condition is a high-to-low transition on SDA with SCL high. A STOP condition is a low-to-high transition on SDA while SCL is high ([Figure 7](#)). A START condition from the master signals the beginning of a transmission to the MAX30101. The master terminates transmission, and frees the bus, by issuing a STOP condition. The bus remains active if a REPEATED START condition is generated instead of a STOP condition.

Early STOP Conditions

The MAX30101 recognizes a STOP condition at any point during data transmission except if the STOP condition occurs in the same high pulse as a START condition. For proper operation, do not send a STOP condition during the same SCL high pulse as the START condition.

Slave Address

A bus master initiates communication with a slave device by issuing a START condition followed by the 7-bit slave ID. When idle, the MAX30101 waits for a START condition followed by its slave ID. The serial interface compares each slave ID bit by bit, allowing the interface to power down and disconnect from SCL immediately if an incorrect slave ID is detected. After recognizing a START condition followed by the correct slave ID, the MAX30101 is programmed to accept or send data. The LSB of the slave ID word is the read/write (R/W) bit. R/W indicates whether the master is writing to or reading data from the MAX30101 (R/W = 0 selects a write condition, R/W = 1 selects a read condition). After receiving the proper slave

ID, the MAX30101 issues an ACK by pulling SDA low for one clock cycle.

The MAX30101 slave ID consists of seven fixed bits, B7–B1 (set to 0b1010111). The most significant slave ID bit (B7) is transmitted first, followed by the remaining bits. [Table 17](#) shows the possible slave IDs of the device.

Acknowledge

The acknowledge bit (ACK) is a clocked 9th bit that the MAX30101 uses to handshake receipt each byte of data when in write mode ([Figure 8](#)). The MAX30101 pulls down SDA during the entire master-generated 9th clock pulse if the previous byte is successfully received. Monitoring ACK allows for detection of unsuccessful data transfers. An unsuccessful data transfer occurs if a receiving device is busy or if a system fault has occurred. In the event of an unsuccessful data transfer, the bus master retries communication. The master pulls down SDA

during the 9th clock cycle to acknowledge receipt of data when the MAX30101 is in read mode. An acknowledge is sent by the master after each read byte to allow data transfer to continue. A not-acknowledge is sent when the master reads the final byte of data from the MAX30101, followed by a STOP condition.

Write Data Format

For the write operation, send the slave ID as the first byte followed by the register address byte and then one or more data bytes. The register address pointer increments automatically after each byte of data received, so for example the entire register bank can be written by at one time. Terminate the data transfer with a STOP condition. The write operation is shown in [Figure 9](#).

The internal register address pointer increments automatically, so writing additional data bytes fill the data registers in order.

Table 17. Slave ID Description

B7	B6	B5	B4	B3	B2	B1	B0	WRITE ADDRESS	READ ADDRESS
1	0	1	0	1	1	1	RW	0xAE	0xAF

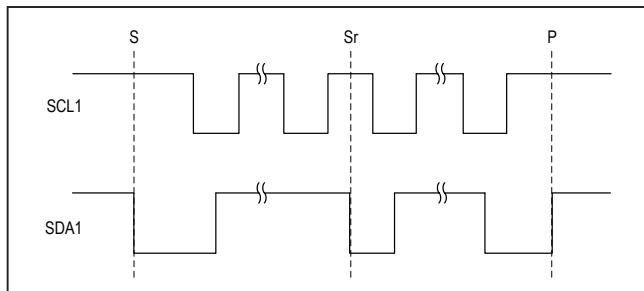


Figure 7. START, STOP, and REPEATED START Conditions

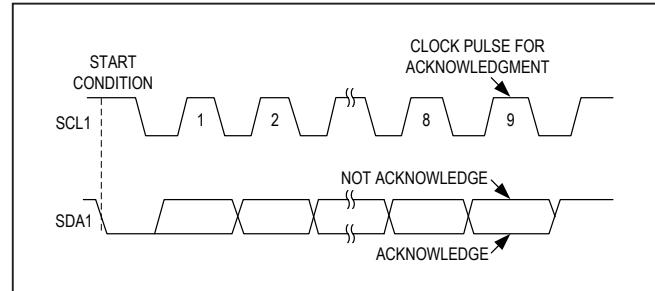


Figure 8. Acknowledge

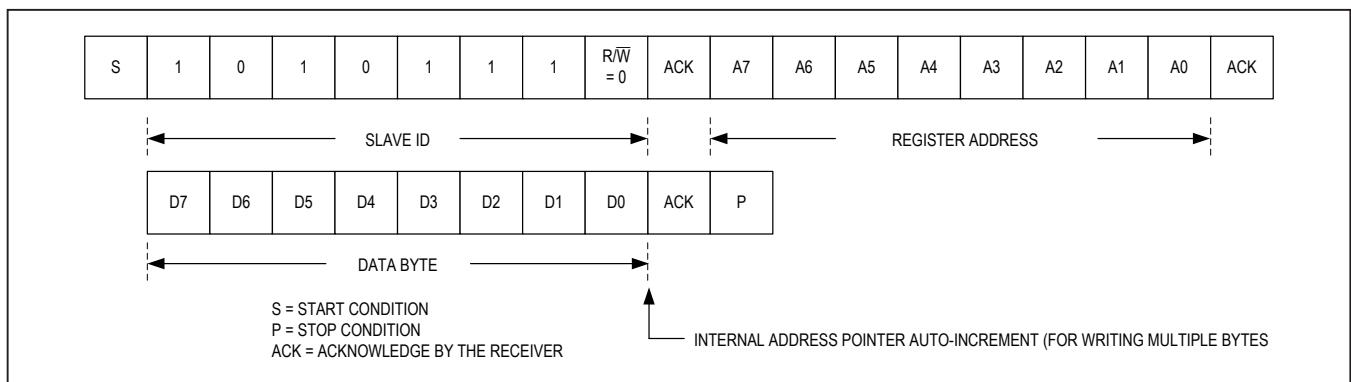


Figure 9. Writing One Data Byte to the MAX30101

Read Data Format

For the read operation, two I²C operations must be performed. First, the slave ID byte is sent followed by the I²C register that you wish to read. Then a REPEAT START (Sr) condition is sent, followed by the read slave ID. The MAX30101 then begins sending data beginning with the register selected in the first operation. The read pointer increments automatically, so the MAX30101 continues sending data from additional registers in sequential order until a STOP (P) condition is received. The exception to this is the FIFO_DATA register, at which the read pointer no longer increments when reading additional bytes. To

read the next register after FIFO_DATA, an I²C write command is necessary to change the location of the read pointer.

[Figure 10](#) show the process of reading one byte or multiple bytes of data.

An initial write operation is required to send the read register address.

Data is sent from registers in sequential order, starting from the register selected in the initial I²C write operation. If the FIFO_DATA register is read, the read pointer will not automatically increment, and subsequent bytes of data will contain the contents of the FIFO.

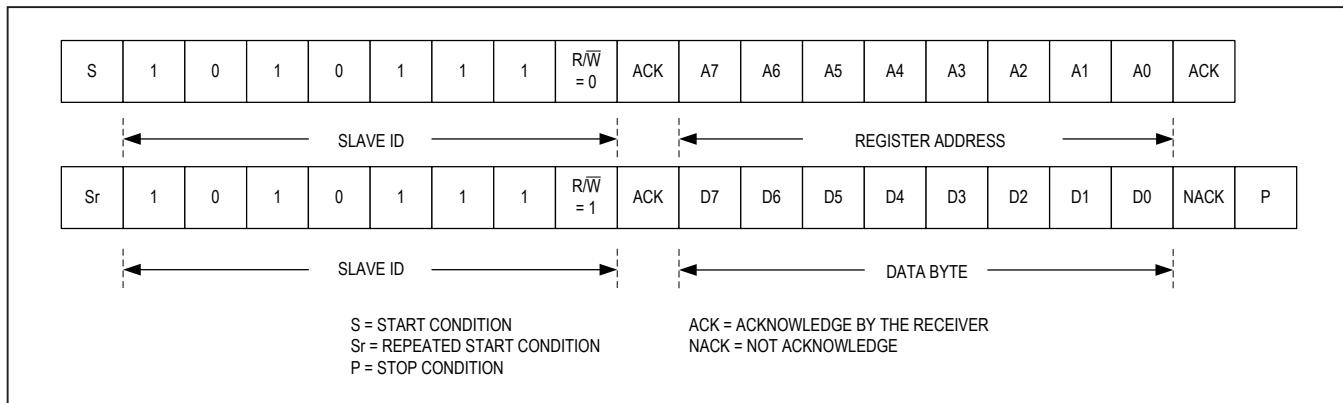


Figure 10. Reading one byte of data from MAX30101

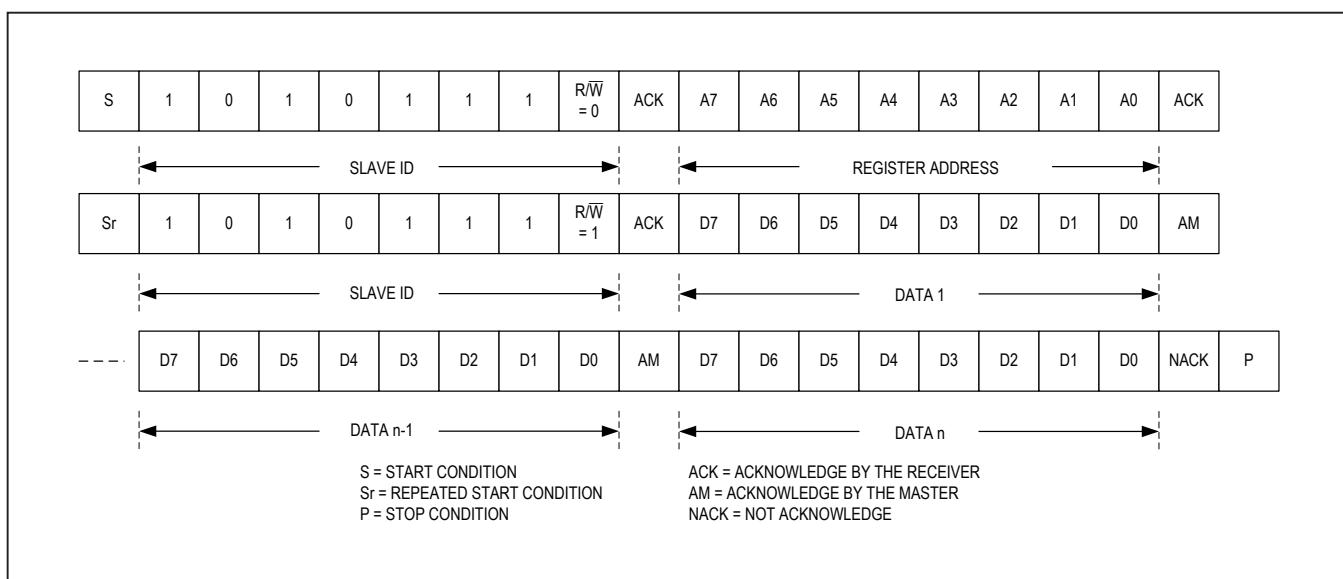
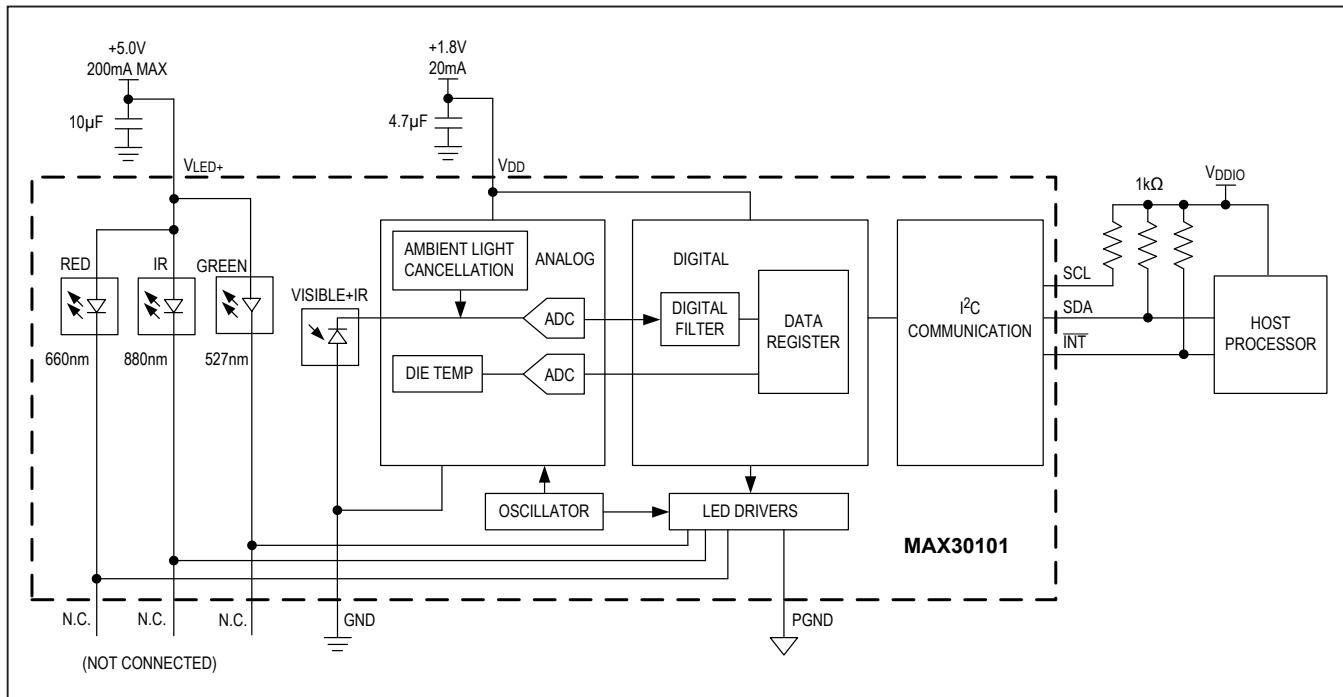


Figure 11. Reading multiple bytes of data from the MAX30101

Typical Application Circuit**Ordering Information**

PART	TEMP RANGE	PIN-PACKAGE
MAX30101EFD+T	-40°C to +85°C	14 OESIP (0.8mm Pin Pitch)

+Denotes lead(Pb)-free/RoHS-compliant package.

T = Tape and reel.

Revision History

REVISION NUMBER	REVISION DATE	DESCRIPTION	PAGES CHANGED
0	3/16	Initial release	—
1	6/18	Changed register descriptions, updated tables 8,9,13,15,16, removed Proximity function, updated FIFO_A_FULL description table	10–15, 18, 21–25, 27, 28
2	9/18	Updated the <i>Applications</i> , <i>Absolute Maximum Ratings</i> , <i>Electrical Characteristics</i> , <i>Pin Description</i> , and <i>Power-Up Sequencing</i> sections; updated the <i>System Diagram</i> , <i>Pin Configuration</i> , <i>Functional Diagram</i> , and <i>Typical Application Circuit</i> ; updated the <i>Register Maps and Descriptions</i> , <i>Mode Configuration (0x09)</i> , <i>SpO₂ Configuration (0x0A)</i> , <i>LED Pulse Amplitude (0x0C–0x0F)</i> , Table 8, and Table 9.	1–5, 9–11, 19, 21–22, 29, 32

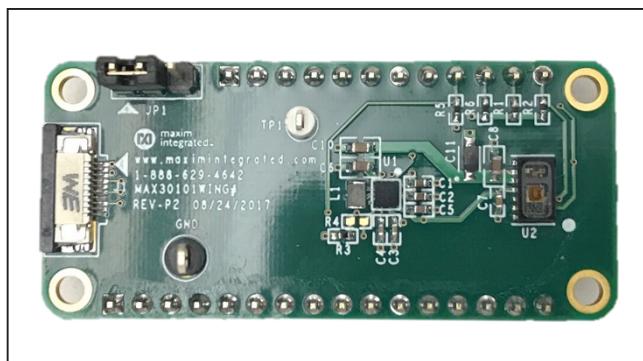
For pricing, delivery, and ordering information, please visit Maxim Integrated's online storefront at <https://www.maximintegrated.com/en/storefront/storefront.html>.

Maxim Integrated cannot assume responsibility for use of any circuitry other than circuitry entirely embodied in a Maxim Integrated product. No circuit patent licenses are implied. Maxim Integrated reserves the right to change the circuitry and specifications without notice at any time. The parametric values (min and max limits) shown in the Electrical Characteristics table are guaranteed. Other parametric values quoted in this data sheet are provided for guidance.

General Description

The MAX30101WING board is a rapid development board designed to quickly develop application firmware for the MAX30101 pulse oximetry sensor. The MAX30101WING contains the MAX14750A power management IC (PMIC) that supplies a +1.8V power rail (1V8) to the MAX30101 along with a programmable +2.5V to +5V rail (VLED) to drive the MAX30101 internal LEDs. The board is compatible with +1.8V and +3.3V logic (VIO), selectable through jumper JP1. The ZIF Flat Flexible Cable Connector allows integration of the MAX30205EVSYS, which features the MAX30205 silicon-based human body temperature sensor. The board seamlessly integrates with the MAX32630FTHR to enable rapid prototyping and development.

MAX30101WING Board

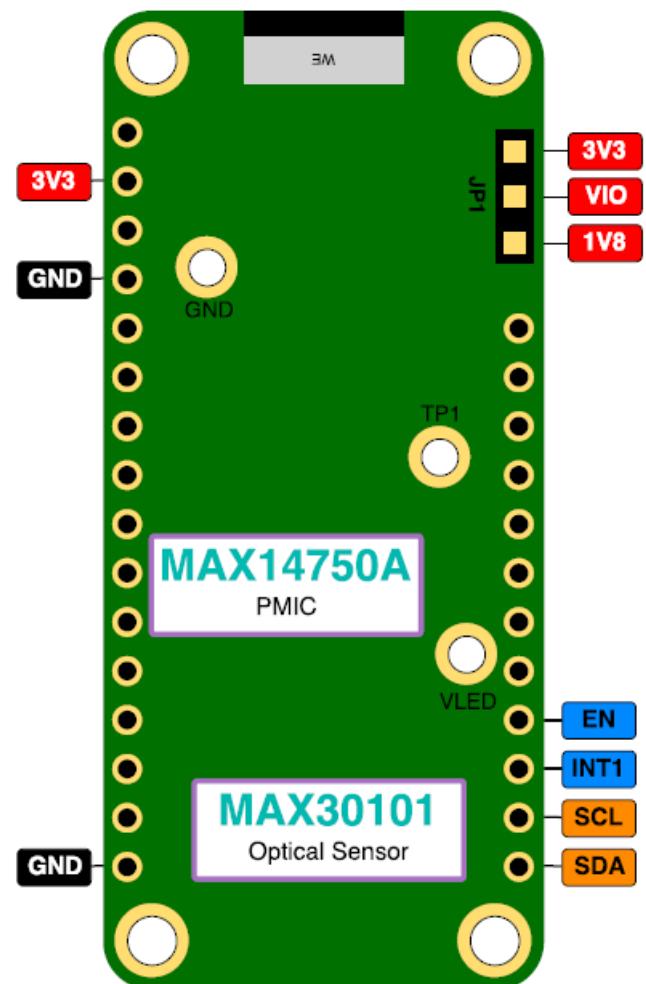


[Ordering Information](#) appears at end of data sheet.

Features

- Convenient Expansion Board
 - 0.9in x 2.0in DIP Form Factor
 - Breadboard Compatible
 - Feather Compatible
- Integrated PMIC to Supply LED Drive Voltage
- +1.8V and +3.3V Logic VIO Compatibility
- ZIF Flat Flex Cable Connector
 - Compatible with the MAX30205EVSYS
- MAX30101 High-Sensitivity Pulse Oximeter and Heart-Rate Sensor Features
 - Noninvasive Reflective LED Solution
 - 5.6mm x 3.3mm x 1.55mm 14-Pin Optical Module
 - Integrated Cover Glass for Optimal, Robust Performance
 - Ultra-Low Power Operation for Mobile Devices
 - Programmable Sample Rate and LED Current for Power Savings
 - Low-Power Heart-Rate Monitor (< 1mW)
 - Ultra-Low Shutdown Current (0.7 μ A, typ)
 - Fast Data Output Capability
 - Sample Rates up to 3200sps
 - Robust Motion Artifact Resilience
 - -40°C to +85°C Operating Temperature Range
- MAX14750A PMIC Features
 - Micro-IQ 250mW Buck-Boost Regulator
 - Supplies VLED to the MAX30101
 - Output Voltage Programmable from 2.5V to 5V
 - 1.1 μ A Quiescent Current
 - Programmable Current Limit
 - Input Voltage from 1.8V to 5.5V
 - Micro-IQ 100mA LDO
 - Supplies 1V8 Rail for the MAX30101
 - Input Voltage From 1.71V to 5.5V
 - Output Programmable from 0.09V to 4.0V
 - 0.9 μ A Quiescent Current
 - Configurable as a Load Switch
 - Individual Enable Pins
 - I²C Control Interface

<https://os.mbed.com/teams/MaximIntegrated/code/MAX30101/>



 maxim
integrated.

0.9" x 2.0"

Figure 1. DIP Pinout

Detailed Description

The MAX30101WING is designed to enable engineers to rapidly prototype and test functionality of the MAX30101 optical sensor. The on-board MAX14750A PMIC sets the LED drive voltage of the optical sensor and is programmable through the I²C interface. The PMIC supplies the 1V8 rail that is used to power the optical sensor along with the communication lines when 1V8 logic is selected. Test point TP1 connects to the monitor multiplexer output of the MAX14750A that allows input and output voltage monitoring through software configuration. Jumper JP1 changes the logic level between 3V3 and 1V8. The ZIF Flat Flex Cable Connector provides a terminal to integrate the MAX30205EVSYS.

The dual inline pinout and form factor for this board is based on the Adafruit® feather series of boards. It is designed for implementation with the MAX32630FTHR and is intended to be compatible with many of the Adafruit peripheral wings, but is not guaranteed to work with all feathers/wings.

Firmware Development

The simplest way to develop firmware for the MAX30101WING board is through the Mbed™ development environment. On the Mbed website, users can import examples into their online IDE, then edit, compile, and load binaries into the board without installing any software. Go to <https://os.mbed.com/platforms/MAX32630FTHR/> to start programming with Mbed.

Table 1. DIP 16-Pin Header

PIN	PORT	DESCRIPTION
1, 3, 5–15	N.C.	Not Connected
2	+3V3	+3V3 Voltage Supply Rail
4, 16	GND	Ground

To support software development, an example library for the MAX30101 and the MAX30205 are found at the following links:

- <https://os.mbed.com/teams/MaximIntegrated/code/MAX30101/>
- <https://os.mbed.com/teams/MaximIntegrated/code/MAX30205/>

In addition, demo programs for the MAX30101WING and MAX30105EVSYS have been developed to expedite prototyping. Visit the links below for access to the source code. Refer to Application Note 6557: *How to Interface the MAX30101WING Pulse Oximeter with the MAX32630FTHR* and Application Note 6558: *How to Interface the MAX30205EVSYS with the MAX32630FTHR* for additional instructions on getting started.

- https://os.mbed.com/teams/Maxim-Integrated/code/MAX30101WING_HR_SPO2/
- https://os.mbed.com/teams/Maxim-Integrated/code/MAX30205_Demo/

Expansion Connectors

The MAX30101WING includes a ZIF Flat Flex Cable Connector to integrate the MAX30205EVSYS. The exposed metallic side of the ribbon cable should face down, blue side up, when inserted. Insert the ribbon cable into ZIF Flat Flex Cable Connector on the MAX30101WING and MAX30205EVSYS to integrate the MAX30205. See the [Firmware Development](#) section for links to development firmware for the MAX30205.

Table 2. DIP 12-Pin Header

PIN	PORT	DESCRIPTION
1–8	N.C.	Not Connected
9	EN	Active-High Enable for VLED Referenced to the Microcontroller's Logic
10	INT	Interrupt for the MAX30101 and MAX30205 Referenced to VIO
11	SCL	I ² C SCL Reference to VIO
12	SDA	I ² C SDA Referenced to VIO

Adafruit is a registered trademark of Adafruit Industries.

Mbed is a trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Table 3. ZIF Flat Flex Cable Connector

PIN	PORT	DESCRIPTION
1	+3V3	+3V3 Voltage Supply Rail
2, 9	N.C.	Not Connected
3, 5, 7, 10	GND	Ground
4	SCL	I ² C SCL
6	SDA	I ² C SDA
8	INT	Interrupt for the MAX30101 and MAX30205

Table 4. Jumpers

PIN	PORT	DESCRIPTION
JP1	1-2*	Sets the logic level to +3V3
	2-3	Sets the logic level to +1V8
R3	Installed*	Connects +3V3 rail to LEN to enable the +1V8 rail
	Not installed	Permits user supplied logic for LEN
R4	Installed	Shorts HVEN and LEN together
	Not installed*	Disconnects HVEN from LEN

*Denotes default positions.

Table 5. Test Point

PIN	DESCRIPTION
GND	Ground test point
VLED	LED drive voltage test point
TP1	Monitor multiplex output test point

Ordering Information

PART	TYPE
MAX30101WING#	Development Board

#Denotes RoHS compliant.

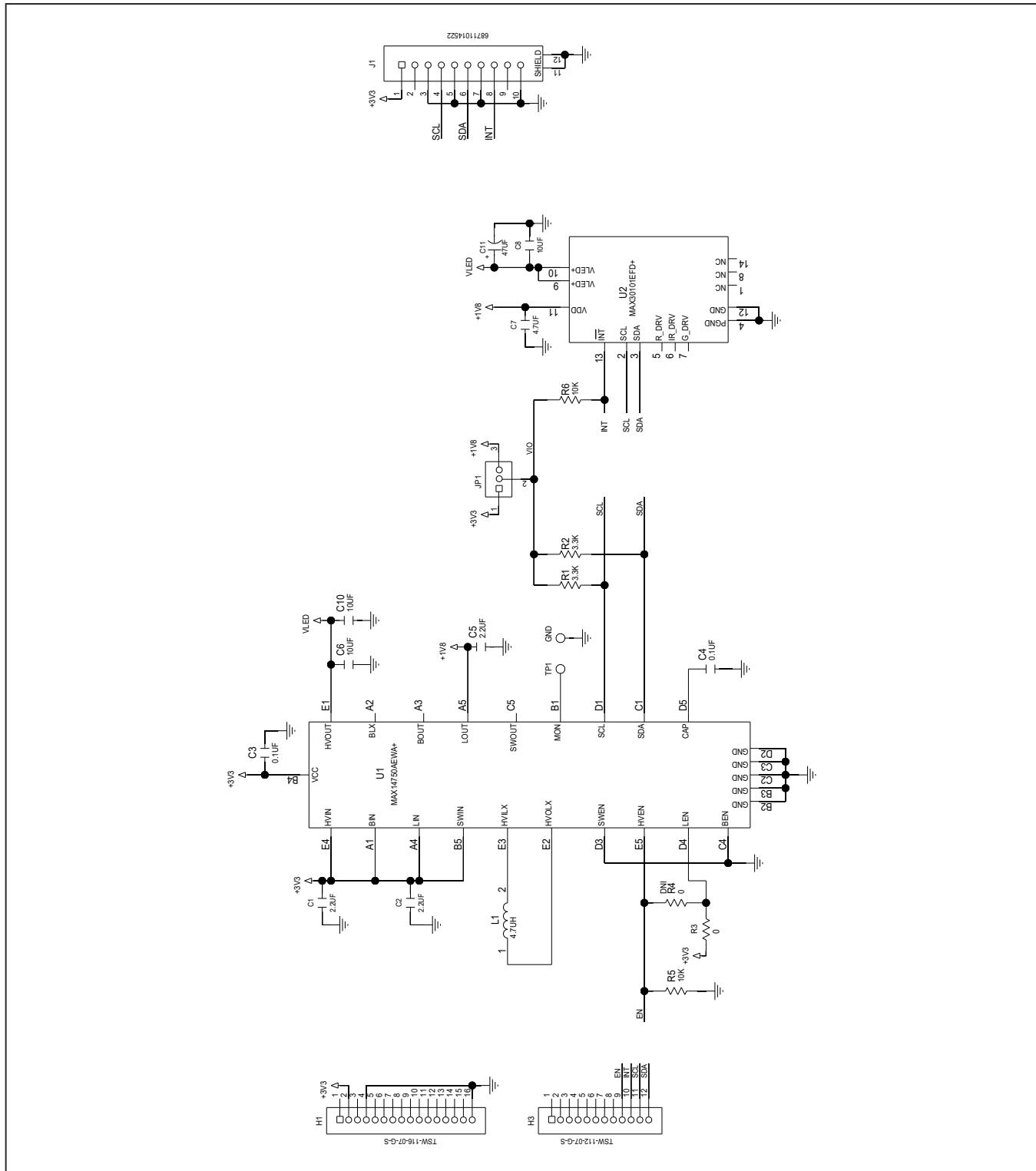
MAX30101WING EV Bill of Materials

ITEM	REF_DES	QTY	MFG PART #	MANUFACTURER	VALUE	DESCRIPTION
1	C1, C2, C5	3	GRM155R61C225KE44	MURATA	2.2UF	CAPACITOR; SMT (0402); CERAMIC CHIP; 2.2UF; 16V; TOL=10%; TG=-55 DEGC TO +85 DEGC; TC=X5R
2	C3, C4	2	GRM155R70J104KA01	MURATA	0.1UF	CAPACITOR; SMT (0402); CERAMIC CHIP; 0.1UF; 6.3V; TOL=10%; TG=-55 DEGC TO +125 DEGC; TC=X7R
3	C6, C8, C10	3	GRM188R61C106KAAL	MURATA	10UF	CAPACITOR; SMT (0603); CERAMIC CHIP; 10UF; 16V; TOL=10%; TG=-55 DEGC TO +85 DEGC; TC=X5R
4	C7	1	GRM155R61A475MEAA	MURATA	4.7UF	CAPACITOR; SMT (0402); CERAMIC CHIP; 4.7UF; 10V; TOL=20%; TG=-55 DEGC TO +85 DEGC; TC=X5R
5	C11	1	F380J476MSAAH1	AVX	47UF	CAPACITOR; SMT (0805); TANTALUM CHIP; 47UF; 6.3V; TOL=20%
6	GND	1	5011	KEYSTONE	N/A	TEST POINT; PIN DIA=0.125IN; TOTAL LENGTH=0.445IN; BOARD HOLE=0.063IN; BLACK; PHOSPHOR BRONZE WIRE SILVER PLATE FINISH;
7	H1	1	TSW-116-07-G-S	SAMTEC	TSW-116-07-G-S	CONNECTOR; MALE; SMT; 0.25INCH SQ POST HEADER; STRAIGHT; 16PINS
8	H3	1	TSW-112-07-G-S	SAMTEC	TSW-112-07-G-S	CONNECTOR; MALE; THROUGH-HOLE .025INCH SQ POST HEADER; STRAIGHT; 12PINS

MAX30101WING EV Bill of Materials (continued)

ITEM	REF_DES	QTY	MFG PART #	MANUFACTURER	VALUE	DESCRIPTION
9	J1	1	6.87E+10	WURTH ELECTRONICS INC.	6.87E+10	CONNECTOR; FEMALE; SMT; 0.5MM ZIF HORIZONTAL BOTTOM CONTACT WR-FPC; RIGHT ANGLE; 10PINS
10	JP1	1	TSW-103-07-T-S	SAMTEC	TSW-103-07-T-S	CONNECTOR; THROUGH HOLE; TSW SERIES; SINGLE ROW; STRAIGHT; 3PINS
11	L1	1	VLS201610CX-4R7M	TDK	4.7UH	INDUCTOR; SMT (2016); FERRITE CORE; 4.7UH; TOL=+/-20%; 0.92A
12	R1, R2	2	ERJ2GEJ332	PANASONIC	3.3K	RESISTOR; 0402; 3.3K OHM; 5%; 200PPM; 0.10W; THICK FILM
13	R3	1	CRCW04020000Z0EDHP; RCS04020000Z0	VISHAY DRALORIC; VISHAY DALE	0	RESISTOR; 0402; 0 OHM; 0%; JUMPER; 0.2W; THICK FILM
14	R5, R6	2	ERJ-2RKF1002	PANASONIC	10K	RESISTOR; 0402; 10K OHM; 1%; 100PPM; 0.10W; THICK FILM
15	TP1	1	5002	KEYSTONE	N/A	TEST POINT; PIN DIA=0.1IN; TOTAL LENGTH=0.3IN; BOARD HOLE=0.04IN; WHITE; PHOSPHOR BRONZE WIRE SILVER;
16	U1	1	MAX14750AEWA+	MAXIM	MAX14750AEWA+	IC; PWRM; POWER-MANAGEMENT SOLUTION; WLP25
17	U2	1	MAX30101EFD+	MAXIM	MAX30101EFD+	IC; SNSR; HIGH-SENSITIVITY PULSE OXIMETER AND HEART-RATE SENSOR IC FOR WEARABLE HEALTH; OLGA14 3.3X5.6
18	PCB	1	MAX	MAXIM	PCB	PCB:MAX

MAX30101WING EV Schematic



Revision History

REVISION NUMBER	REVISION DATE	DESCRIPTION	PAGES CHANGED
0	3/18	Initial release	—

For pricing, delivery, and ordering information, please contact Maxim Direct at 1-888-629-4642, or visit Maxim Integrated's website at www.maximintegrated.com.

Maxim Integrated cannot assume responsibility for use of any circuitry other than circuitry entirely embodied in a Maxim Integrated product. No circuit patent licenses are implied. Maxim Integrated reserves the right to change the circuitry and specifications without notice at any time.

MAX32664

Ultra-Low Power Biometric Sensor Hub

General Description

The MAX32664 is a sensor hub family with embedded firmware and algorithms for wearables. It seamlessly enables customer-desired sensor functionality, including communication with Maxim's optical sensor solutions and delivering raw or calculated data to the outside world. This is achieved while keeping overall system power consumption in check.

The MAX32664 Version A supports the MAX30101/MAX30102 high-sensitivity pulse oximeter and heart rate sensor for wearable health for finger-based applications.

The MAX32664 Version B supports the MAX86140/MAX86141 for wrist-based monitoring.

The sensor hub interface provides a fast-mode, I²C slave interface to a microcontroller host. A second I²C interface is dedicated to communicating with sensors.

The tiny form factor (1.6mm x 1.6mm, 16-bump WLP) allows for integration into extremely small application devices.

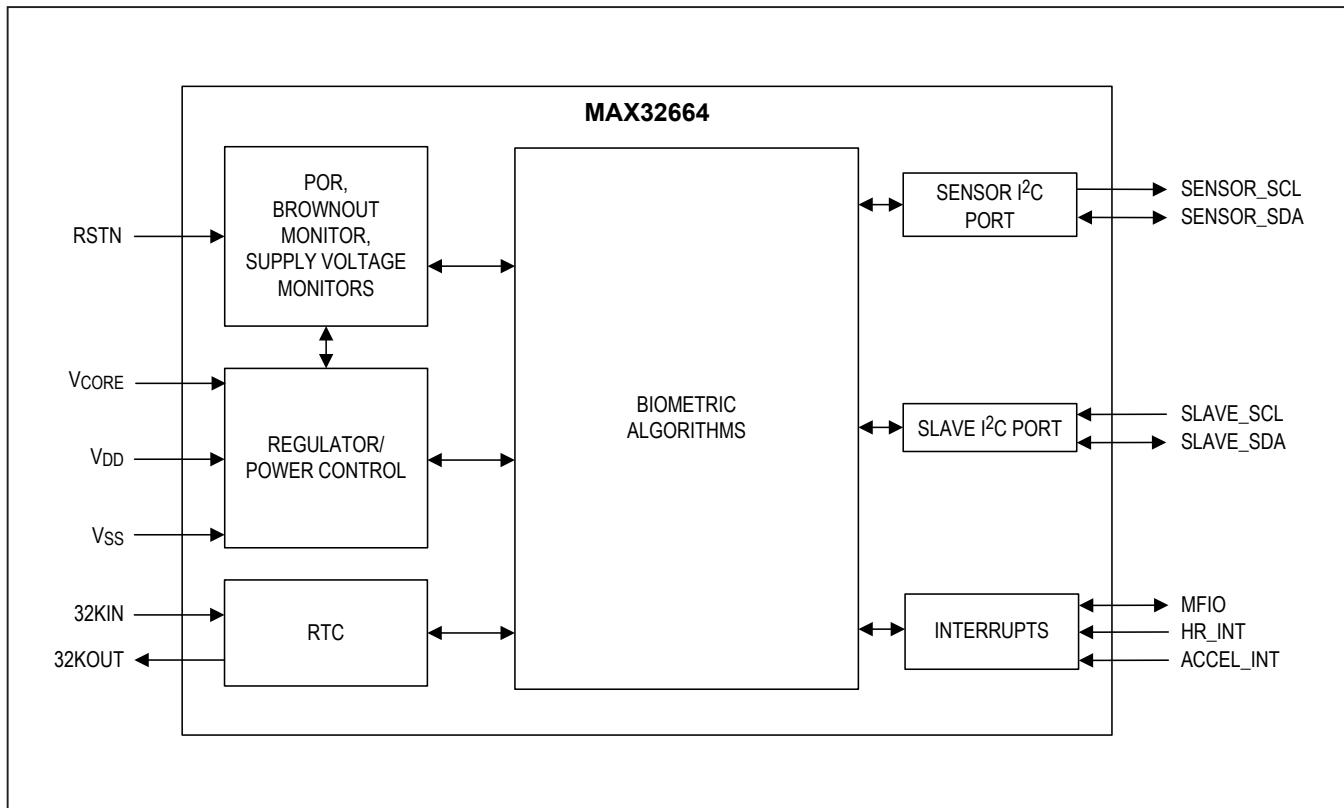
Applications

- Wearable Fitness
- Hearables
- Wearable Medical
- Portable Medical
- Mobile Devices

Benefits and Features

- Biometric Sensor Hub Solution Enables Faster Time to Market
- Finger-Based (Version A) Algorithms Measure
 - Pulse Heart Rate
 - Pulse Blood Oxygen Saturation (SpO₂)
- Both Raw and Processed Data Are Available
- Basic Peripheral Mix Optimizes Size and Performance
 - One Slave I²C for Communication to a Host Microcontroller
 - One Master I²C for Communication with Sensors
 - 32.768kHz RTC
 - FIFO Provides Minimal Host Processor Interaction
 - Secure, Authenticated Firmware Upgrades

[Ordering Information](#) appears at end of data sheet.

Simplified Block Diagram

Absolute Maximum Ratings

(All voltages with respect to V _{SS} , unless otherwise noted.)	
V _{CORE}	-0.3V to +1.21V
V _{DD}	-0.3V to +3.63V
32KIN, 32KOUT.....	-0.3V to V _{DD} + 0.3V
RSTN.....	-0.3V to V _{DD} + 0.3V
Total Current into All Digital Pins Combined (sink)	100mA
V _{SS}	100mA

Output Current (sink) by Any Digital Pin	25mA
Output Current (source) by Any Digital Pin.....	-25mA
Operating Temperature Range.....	-40°C to +105°C
Storage Temperature Range.....	-65°C to +150°C
Soldering Temperature (reflow).....	+260°C

Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated in the operational sections of the specifications is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Package Information

16 WLP

Package Code	W161K1+1
Outline Number	21-100241
Land Pattern Number	Refer to Application Note 1891
Thermal Resistance, Four-Layer Board:	
Junction to Ambient (θ_{JA})	66.34 °C/W
Junction to Case (θ_{JC})	N/A

For the latest package outline information and land patterns (footprints), go to [www.maximintegrated.com/packages](#). Note that a "+", "#", or "-" in the package code indicates RoHS status only. Package drawings may show a different suffix character, but the drawing pertains to the package regardless of RoHS status.

Package thermal resistances were obtained using the method described in JEDEC specification JESD51-7, using a four-layer board. For detailed information on package thermal considerations, refer to [www.maximintegrated.com/thermal-tutorial](#).

Electrical Characteristics

(Limits are 100% tested at $T_A = +25^\circ\text{C}$ and $T_A = +105^\circ\text{C}$. Limits over the operating temperature range and relevant supply voltage range are guaranteed by design and characterization. Specifications marked GBD are guaranteed by design and not production tested. Specifications to the minimum operating temperature are guaranteed by design and are not production tested.)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
POWER SUPPLIES						
Supply Voltage	V_{DD}		1.71	1.8	3.63	V
Power-Fail Reset Voltage	V_{RST}	Monitors V_{DD}	1.63		1.71	V
Power-On Reset Voltage	V_{POR}	Monitors V_{DD}		1.4		V
DIGITAL I/O						
Input Low Voltage for RSTN, SLAVE_SCL	V_{IL_RSTN}			0.3 × V_{DD}		V
Input High Voltage for All GPIO, RSTN	V_{IH_RSTN}			0.7 × V_{DD}		V
Output Low Voltage for SENSOR_SDA, SENSOR_SCL, SLAVE_SDA	V_{OL_I2C}	$V_{DD} = 1.71\text{V}$, $I_{OL} = 2\text{mA}$		0.2	0.4	V
Output High Voltage for SENSOR_SDA, SENSOR_SCL, SLAVE_SDA	V_{OH_I2C}	$V_{DD} = 1.71\text{V}$, $I_{OH} = 2\text{mA}$,		$V_{DD} - 0.4$		V
Input Hysteresis (Schmitt)	V_{IHYS}		300			mV
Input/Output Pin Capacitance for All Pins	C_{IO}		4			pF
Input Leakage Current Low	I_{IL}	$V_{IN} = 0\text{V}$, internal pullup disabled	-500		+500	nA
Input Leakage Current High	I_{IH}	$V_{IN} = 3.6\text{V}$, internal pulldown disabled	-500		+500	nA
Input Pullup Resistor to RSTN	R_{PU_VDD}	Pullup to $V_{DD} = 3.63\text{V}$		10.5		kΩ
CLOCKS						
System Clock Frequency	f_{SYS_CLK}		96			MHz
RTC Input Frequency	$f_{32\text{KIN}}$	32.768kHz watch crystal, $C_L = 6\text{pF}$, ESR < 90kΩ, $C_0 < 2\text{pF}$		32.768		kHz
RTC Operating Current	I_{RTC}	All power modes, RTC enabled		0.57		µA
RTC Power-Up Time	t_{RTC_ON}			250		ms

Electrical Characteristics—I²C

(Limits are 100% tested at $T_A = +25^\circ\text{C}$ and $T_A = +105^\circ\text{C}$. Limits over the operating temperature range and relevant supply voltage range are guaranteed by design and characterization. Specifications marked GBD are guaranteed by design and not production tested. Specifications to the minimum operating temperature are guaranteed by design and are not production tested.)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
FAST MODE						
Output Fall Time	t_{OF}	From $V_{OL_I^2C(MIN)}$ to $V_{OL_I^2C(MAX)}$		150		ns
Pulse Width Suppressed by Input Filter	t_{SP}			75		ns
SCL Clock Frequency	f_{SCL}		0	400		kHz
Low Period SCL Clock	t_{LOW}		1.3			μs
High Time SCL Clock	t_{HIGH}		0.6			μs
Setup Time for Repeated Start Condition	$t_{SU;STA}$		0.6			μs
Hold Time for Repeated Start Condition	$t_{HD;STA}$		0.6			μs
Data Setup Time	$t_{SU;DAT}$			125		ns
Data Hold Time	$t_{HD;DAT}$			10		ns
Rise Time for SDA and SCL	t_R		30			ns
Fall Time for SDA and SCL	t_F		30			ns
Setup Time for a Stop Condition	$t_{SU;STO}$		0.6			μs
Bus Free Time Between a Stop and Start Condition	t_{BUS}		1.3			μs
Data Valid Time	$t_{VD;DAT}$		0.9			μs
Data Valid Acknowledge Time	$t_{VD;ACK}$		0.9			μs

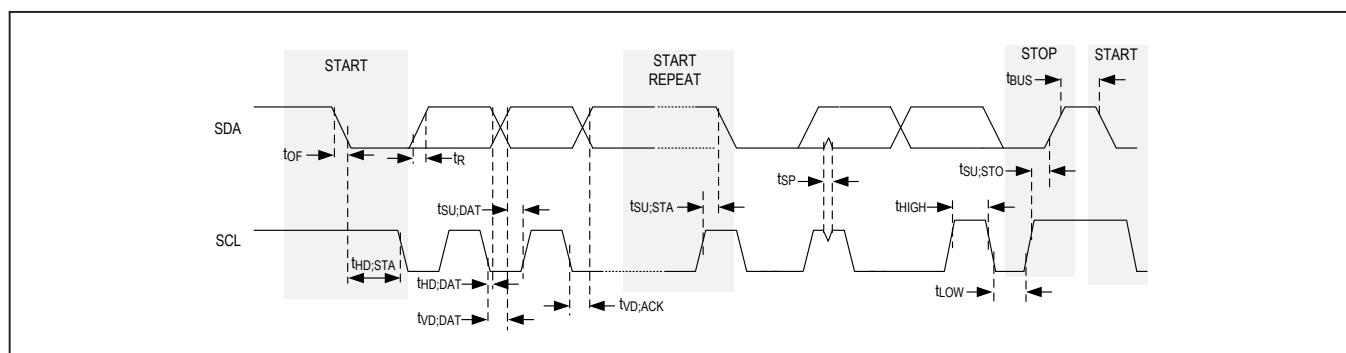
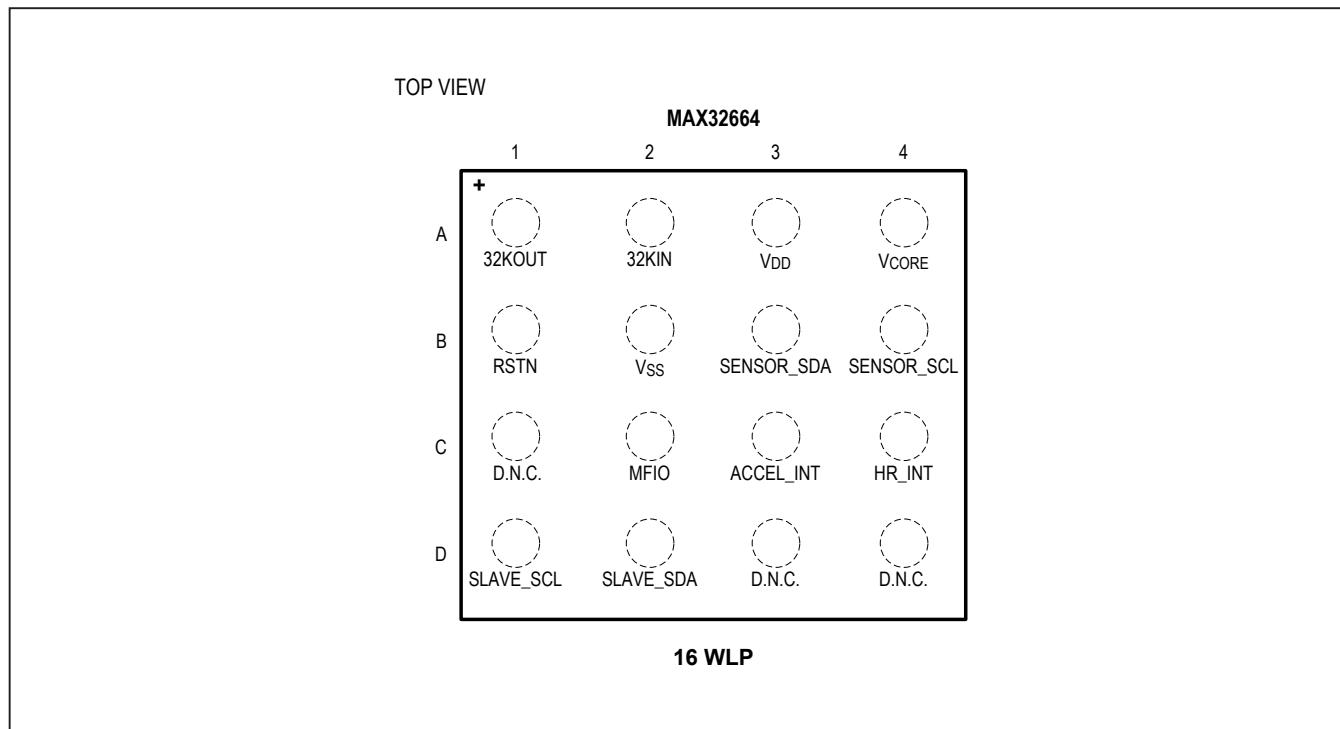


Figure 1. I²C Timing Diagram

Bump Configuration**Bump Description**

BUMP	NAME	FUNCTION
POWER		
A3	V _{DD}	Digital Supply Voltage. This pin must be bypassed to V _{SS} with a 1.0µF capacitor as close as possible to the package. The device can operate solely from this one power supply pin.
A4	V _{CORE}	Core Supply Voltage. This pin must be left open-circuit. This pin must always be bypassed to V _{SS} with a 1.0µF capacitor as close as possible to the package.
B2	V _{SS}	Digital Ground
CLOCK		
A2	32KIN	32.768kHz Crystal Oscillator Input. Connect a 32.768kHz crystal between 32KIN and 32KOUT for RTC operation. Optionally, an external clock source can be driven on 32KIN if the 32KOUT pin is left unconnected.
A1	32KOUT	32.768kHz Crystal Oscillator Output
RESET		
B1	RSTN	Hardware Power Reset (Active-Low) Input. The device remains in reset while this pin is in its active state. When the pin transitions to its inactive state, the device performs a POR reset (resetting all logic on all supplies except for real-time clock circuitry) and begins execution. This pin is internally connected with an internal pullup to the V _{DD} supply as indicated in the Electrical Characteristics table. Add and place a noise snubber circuit as close as possible to the device, with component values shown in the Typical Application Circuit .

Pin Description (continued)

BUMP	NAME	FUNCTION
I²C		
D1	SLAVE_SCL	Slave SCL. This is the I ² C slave SCL that should be connected to the host microprocessor I ² C master SCL.
D2	SLAVE_SDA	Slave SDA. This is the I ² C slave SDA that should be connected to the host microprocessor I ² C master SDA.
B4	SENSOR_SCL	Sensor SCL. This is the I ² C master SCL that should be connected to the I ² C slave SCL on the slave sensors.
B3	SENSOR_SDA	Sensor SDA. This is the I ² C master SDA that should be connected to the I ² C slave SDA on the slave sensors.
INTERRUPTS		
C4	HR_INT	Heart Rate Monitor Interrupt Input. This pin connects to the heart rate monitor sensor interrupt output.
C3	ACCEL_INT	Accelerometer Interrupt Input. This pin connects to the accelerometer sensor interrupt output.
C2	MFIO	Multifunction IO. This pin provides different functions: MFIO asserts low as an output when the sensor hub needs to communicate with the host controller; the pin acts as an input and causes the sensor hub to enter bootloader mode if the MFIO pin is low during a reset.
DO NOT CONNECT		
C1, D3, D4	D.N.C.	Do Not Connect. Internally connected. Do not make any electrical connection, including V _{SS} , to these pins.

Detailed Description

The MAX32664 is a sensor hub family with embedded firmware and algorithms for wearables. It seamlessly enables customer-desired sensor functionality, including communication with Maxim's optical sensor solutions and delivering raw or calculated data to the outside world. This is achieved while keeping overall system power consumption in check.

The MAX32664 Version A supports the MAX30101/MAX30102 high-sensitivity pulse oximeter and heart-rate sensor for wearable health for finger-based applications.

The sensor hub interface provides a fast-mode I²C slave interface to transmit commands and data to a microcontroller host. A second I²C interface is dedicated to communication with sensors.

The sampling of the sensors is derived from the 32.768kHz real-time clock. The sampling rate is user-configurable to minimize power consumption.

Algorithmic Processing**Finger Heart Rate Algorithms**

The MAX32664 Version A performs finger-based heart rate and blood oxygen saturation (SpO₂) monitoring. The embedded algorithm uses digital filtering, pressure/position compensation, advanced R-wave detection, and automatic gain control to determine the heart rate in beats per minute while minimizing power. Also, the Maxim Integrated sensor hardware has built-in ambient light rejection to minimize background noise. SpO₂ results are reported as percentage of hemoglobin that is saturated with oxygen. The calibration values for SpO₂ configuration should be performed while using the end product.

I²C Interface

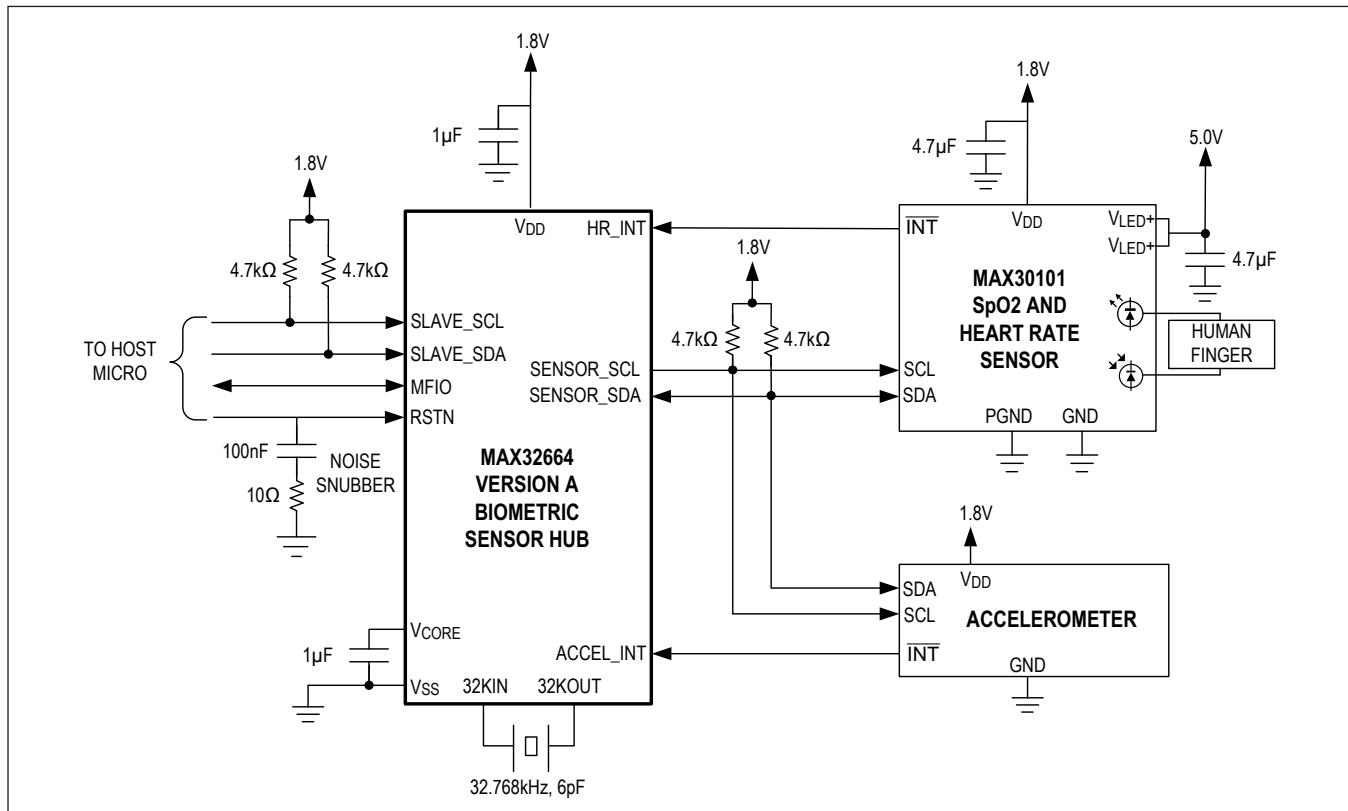
The I²C interface is a bidirectional, two-wire serial bus that provides a medium-speed communications network. The device provides two I²C busses.

The I²C sensor bus is dedicated to communication with the sensors. The characteristics of this bus are not user-configurable.

The I²C slave bus connects the sensor hub to a host system or microcontroller that acts as an I²C master. It provides the following features:

- Slave address 0x55
- Fast-mode (400kbps) transfer rate

- Supports standard 7-bit addressing or 10-bit addressing
- RESTART condition
- Interactive receive mode
- Tx FIFO preloading
- Support for clock stretching to allow slower slave devices to operate on higher speed busses
- Internal filter to reject noise spikes
- Receiver FIFO depth of 8 bytes
- Transmitter FIFO depth of 8 bytes

Typical Application Circuit

Ordering Information

PART	VERSION	BOOT LOADER	PIN-PACKAGE
MAX32664GWEA+	A	Yes	16 WLP (1.6mm x 1.6mm x 0.65mm, 0.35mm pitch)
MAX32664GWEA+T	A	Yes	16 WLP (1.6mm x 1.6mm x 0.65mm, 0.35mm pitch)
MAX32664GWEB+*	B	Yes	16 WLP (1.6mm x 1.6mm x 0.65mm, 0.35mm pitch)
MAX32664GWEB+T*	B	Yes	16 WLP (1.6mm x 1.6mm x 0.65mm, 0.35mm pitch)

+Denotes a lead(Pb)-free/RoHS-compliant package.

T = Tape and reel. Full reel.

*Future product—contact factory for availability.

Revision History

REVISION NUMBER	REVISION DATE	DESCRIPTION	PAGES CHANGED
0	4/18	Initial release	—
0.1		Added future product references to MAX32664GWEB+ and MAX32664GWEB+T	9

For pricing, delivery, and ordering information, please contact Maxim Direct at 1-888-629-4642, or visit Maxim Integrated's website at www.maximintegrated.com.

Maxim Integrated cannot assume responsibility for use of any circuitry other than circuitry entirely embodied in a Maxim Integrated product. No circuit patent licenses are implied. Maxim Integrated reserves the right to change the circuitry and specifications without notice at any time. The parametric values (min and max limits) shown in the Electrical Characteristics table are guaranteed. Other parametric values quoted in this data sheet are provided for guidance.

References

- [1] Edward D. Chan, Michael M. Chan, and Mallory M. Chan. Pulse oximetry: Understanding its basic principles facilitates appreciation of its limitations. *Respiratory Medicine*, 107(6):789–799, 2013. URL: <https://www.sciencedirect.com/science/article/pii/S09546111300053X>, <https://doi.org/https://doi.org/10.1016/j.rmed.2013.02.004>. doi:<https://doi.org/10.1016/j.rmed.2013.02.004>.
- [2] Maxim Integrated. *5V/3.3V or Adjustable, Low-Dropout, Low IQ, 500mA Linear Regulators*.
- [3] Maxim Integrated. *High-Sensitivity Pulse Oximeter and Heart-Rate Sensor for Wearable Health*.
- [4] Maxim Integrated. *MAX32664 User Guide*.
- [5] Maxim Integrated. *Recommended Configurations and Operating Profiles for MAX30101/MAX30102 EV Kits, AN6409*.
- [6] University of Iowa Health Care. Pulse oximetry basic principles and interpretation. URL: <https://medicine.uiowa.edu/iowaproocols/pulse-oximetry-basic-principles-and-interpretation>.
- [7] David Thompson, Austin Wareing, Dwight Day, and Steve Warren. Pulse oximeter improvement with an adc-dac feedback loop and a radial reflectance sensor. In *2006 International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 815–818, 2006. <https://doi.org/10.1109/IEMBS.2006.259501> doi:10.1109/IEMBS.2006.259501.
- [8] Wikipedia. Bresenham's line algorithm. URL: https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm.
- [9] LTD. XIAMEN OCULAR OPTICS CO. *GDM12864HLCM Liquid Crystal Display, User's Guide*.