To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

RENESAS

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

H8/300L

Implementation of I^2C (Port) to Three I^2C Devices (3I2Cport)

Introduction

This application note provides an overview of the I^2C bus interface. It also demonstrates how to integrate and interface multiple I^2C devices to the H8/38024 SLP series through software control of its two general I/O pins:

- Microchip 24AA16 16K I^2C Serial EEPROM (read/write)
- Maxim MAX6626 12-bit Temperature Sensor (read/write)
- MAX6953EPL 2-wire Interfaced 4-digit 5 x 7 Matrix LED Display Driver (write only)

Target Device

H8/38024 SLP Series

Contents

1. I^{2}C™ Interface Overview

The I^{2}C bus uses a two-wire interface consisting of a serial data line (SDA) and a serial clock line (SCL) to exchange information between devices connected to the bus. Each device on the bus has its own unique address and can operate as a transmitter or receiver (depending on its particular function). Devices are further categorized as Masters or Slaves. A Master is defined as a device that initiates, controls (generates all framing and clock signals), and terminates a transfer whereas a Slave is any device addressed by a Master.

It is noted that different I^{2}C devices will have slightly different protocols. This application note describes an I^{2}C interface comprising of a bus master (H8/38024 SLP MCU) and three slave devices (Microchip 24AA16 16K I^{2}C Serial EEPROM, Maxim MAX6626 12-bit temperature sensor and MAX6953EPL 2-wire interfaced 4-digit 5 x 7 Matrix LED Display Driver). In this simple interface, the temperature converted from the MAX6626 will be displayed and can also be stored in the E^{2}PROM. Note that the I^{2}C interface is simulated using software to control two general I/O pins (P70→SDA & P80→SCL) of the H8/38024 SLP MCU. The SDA and SCL pins of these devices are directly connected to P70 and P80 of the MCU respectively. Figure 1 shows the system block diagram.

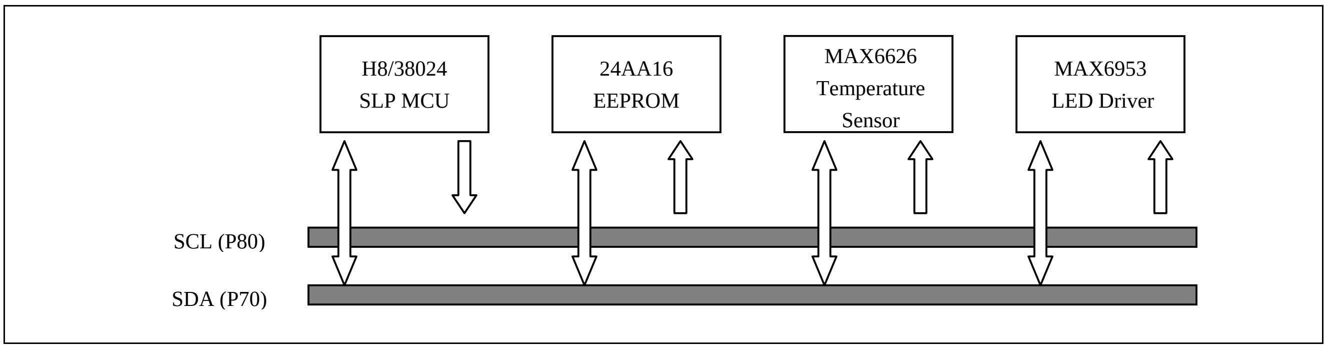


Figure 1 System Block Diagram

The address of each device is summarized in Table 1 (This discussion is limited to the I^{2}C 7-bit addressing mode). Each device has a unique 7-bit I^{2}C address so that the master knows which device it is communicating with. Typically, the upper address lines are fixed while the lower address lines are set by hardware. For the case of three lower address lines (A2, A1, A0), there are up to eight different combinations. Therefore, up to a maximum of eight identical devices can be interfaced on the same bus. To interface another I^{2}C device, the SDA and SCL pins must be connected to the bus with a unique address assigned to it.

Table 1 Device Addresses

Device	Address (Hexadecimal)
Microchip 24AA16 EEPROM	A0 – Block 0 A2 – Block 1 A4 – Block 2 A6 – Block 3 A8 – Block 4 AA – Block 5 AC – Block 6 AE – Block 7
MAX6626 Temperature Sensor	90
MAX6953 LED Driver	B0

The features of these three slave devices and the software description together with the hardware design will be covered in the following sections.

2. Microchip 24AA16 E2PROM

The Microchip 24AA16 is a 16 Kbit EEPROM organized as eight blocks of 256 × 8-bit memory with a 2-wire interface. It also supports page-write up to 16 bytes. Figure 2 shows the block diagram.

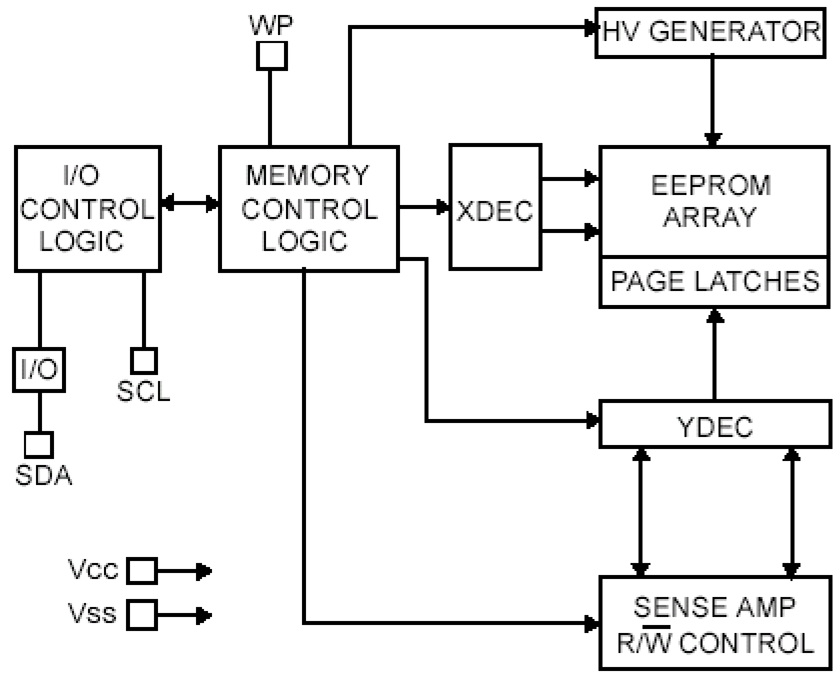


Figure 2 Microchip 24AA16 E2PROM Block Diagram

2.1 Bus Protocol

Data transfers on the I2C bus are controlled (and framed) via two unique bus states generated by the Bus Master. These bus states are the START and STOP bit conditions. When the bus is free, both lines are HIGH.

A START condition is defined as a High-to-Low level transition on SDA while the SCL line is High. A STOP condition is defined as a Low-to-High level transition on SDA while the SCL line is High. Data must always be valid (stable) on the SDA line while SCL is high. The SDA line is only allowed to change during the low period of SCL. One data bit is transmitted per SCL clock pulse.

Following the START condition, the first 8-bit byte sent in a bus message is a 7-bit Slave address field along with a data direction or R/$\overline{W}$ bit. The data direction bit (least significant bit) controls whether the Master transmits (0 = write) or receives (1 = read) data from the addressed Slave.

The acknowledge bit is a low-level signal placed on the SDA line by the receiving device (Master or Slave) during the Master-transmitted acknowledge clock pulse (ninth High SCL clock pulse of the byte transmission). If the Slave is busy and unable to receive data or the Master needs to signal the end of data transfer, a *non-acknowledge* is sent (SDA High during the ninth High SCL clock pulse time).

Following the START and Slave address transmission, data is exchanged between the Master and receiver as required. Upon exchange of the final byte and its acknowledge, the Master issues the STOP condition to end bus usage.

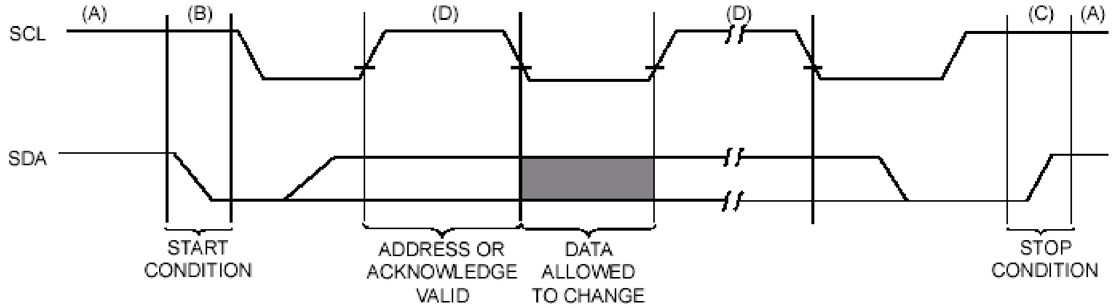
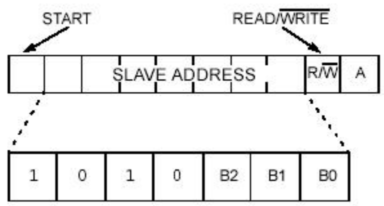


Figure 3 Data Transfer Sequence

2.1.1 Device Addressing

A control byte is the first byte received following the start condition from the master device. For the 24AA16, the first four bits of the control code are set to 1010 binary for both read and write operations. The next three bits are the block select bits (B2, B1, B0), used by the master device to select which 256-word block of memory to be accessed. These bits are in effect the three most significant bits of the word address. It should be noted that the protocol limits the size of the memory to eight blocks of 256 words; therefore the protocol can support only one 24AA16 per system. The last bit of the control byte defines the operation to be performed. When set to '1', a READ operation is selected. When set to '0', a WRITE operation is selected. Following the START condition, the 24AA16 monitors the SDA bus to check the device type identifier being transmitted. Upon reception of the 1010 code, the slave device outputs an acknowledge signal on the SDA line. Depending on the state of the $R/\overline{W}$ bit, the 24AA16 will select a READ or WRITE operation.



Operation	Control Code	Block Select	$R/\overline{W}$
Read	1010	Block Address	1
Write	1010	Block Address	0

Figure 4 Control Byte

2.1.2 Bit Transfer and Data Validity

The number of data bytes transferred from the transmitter to receiver between the START and STOP conditions is determined by the Master. Each byte (eight bits) is transferred serially with the most significant bit first followed by an acknowledge bit. The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the HIGH period of the clock signal. The data on the line must be changed during the LOW period of the clock signal. There is one clock pulse per bit of data.

2.1.3 Acknowledge

Each receiving device, when addressed, is obliged to generate an acknowledge after the reception of each byte. The master device must generate an extra clock pulse which is associated with this acknowledge bit. For 24AA16, it does not generate any acknowledge bits if an internal programming cycle is in progress.

2.2 WRITE Operation

Write operations are initiated when the R/$\overline{W}$ bit of the slave address is set to '0'. There are two types of write operation: byte and page writes.

2.2.1 Byte Write

Byte operations allow a random EEPROM address to be written. Byte-write operations require the following transmission:

- START condition (Master)
- EEPROM Device Address with R/$\overline{W}$ = 0 (Master)
- Acknowledge Bit (EEPROM)
- Target EEPROM Word Address to be Written (Master)
- Acknowledge Bit (EEPROM)
- Data Byte to be Written (Master)
- Acknowledge bit (EEPROM)
- STOP Condition (Master)

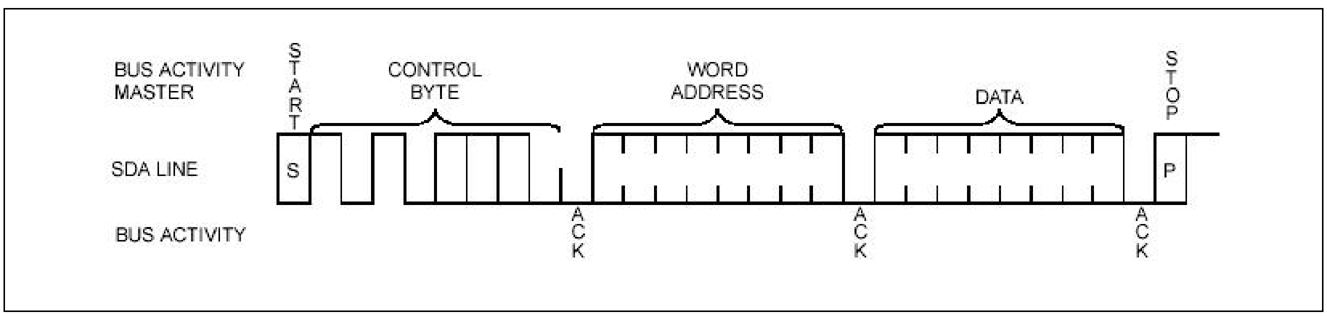


Figure 5 Byte Write

2.2.2 Page Write

Page write operations allows up to 16 bytes to be written to the EEPROM. During these page writes, the EEPROM automatically increments its internal address pointer between bytes.

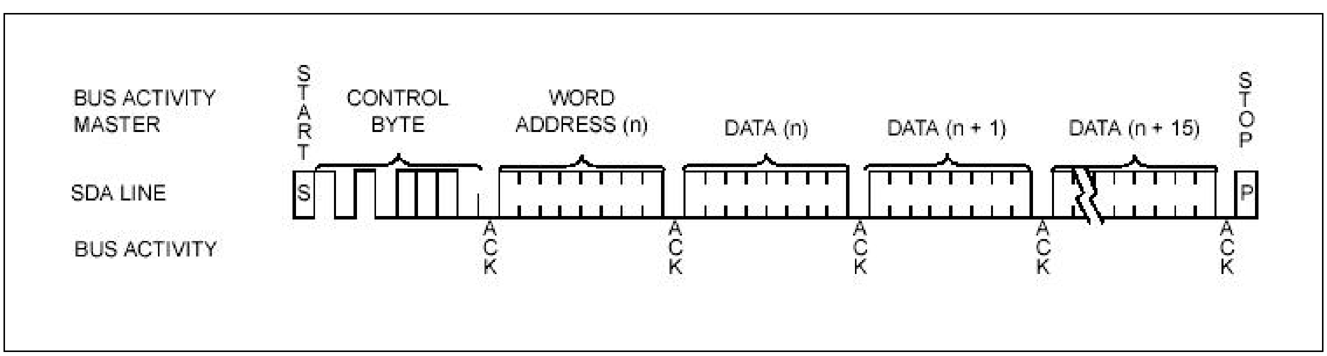


Figure 6 Page Write

2.3 READ Operation

Three types of read operations are supported: Current Address, Random, and Sequential Read. Read operations begin just like write operations, except that the R/$\overline{W}$ bit is set to 1 for the device address byte.

2.3.1 Current Address Read

In the Current Address Read mode, the data is read from the location of the most recent access. This read transmission type sequence appears as:

- START condition (Master)
- EEPROM device address with R/$\overline{W}$ = 1 (Master)
- Acknowledge bit (EEPROM)
- Data byte to be read (EEPROM bytes sent from the addressed Slave's most recent pointed-to memory location incremented by 1)
- Non-acknowledge bit (Master)
- STOP condition (Master)

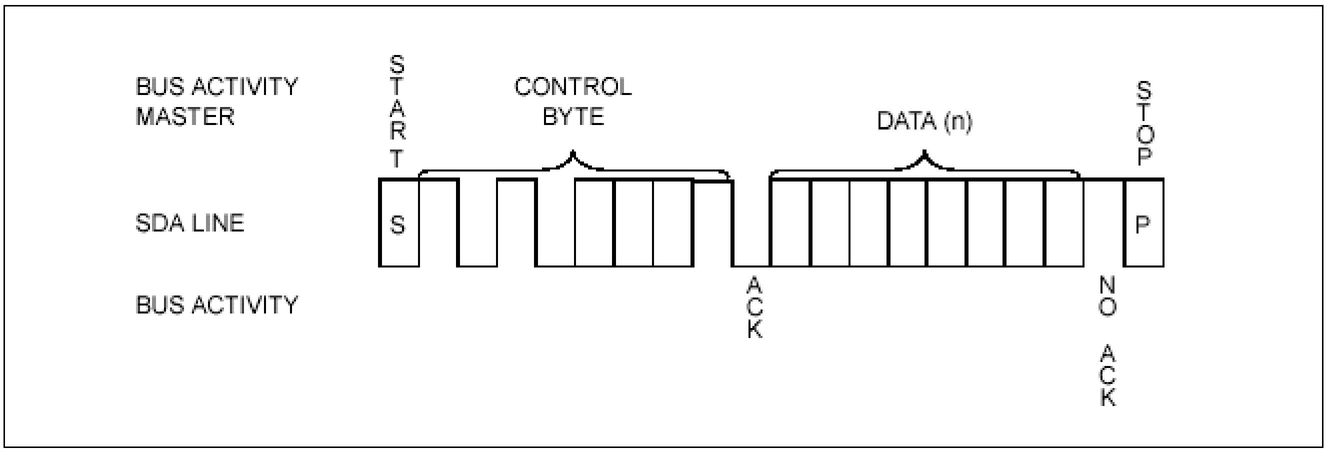


Figure 7 Current Address Read

2.3.2 Random Read

The Random Read mode is begun with a dummy byte write cycle (Master sends a START condition followed by the device address and target word address) followed by a Current Address Read mode cycle as described previously.

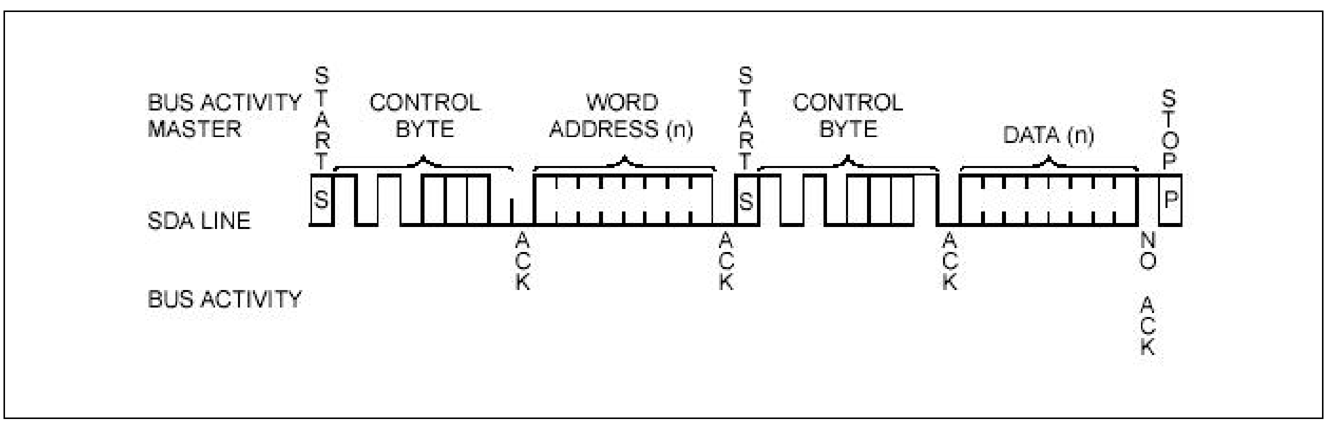


Figure 8 Random Read

2.3.3 Sequential Read

The Sequential Read mode is initiated with a Random Read. Instead of the Master terminating the read after a single byte exchange (with a non-acknowledge), the Master responds with a valid acknowledge after each received data byte. This acknowledge instructs the Slave EEPROM to continue the read operation and transmit out the next data byte. Sequential reads continue until terminated by the Master via issuance of a non-acknowledge on the most recent byte read followed by the STOP condition.

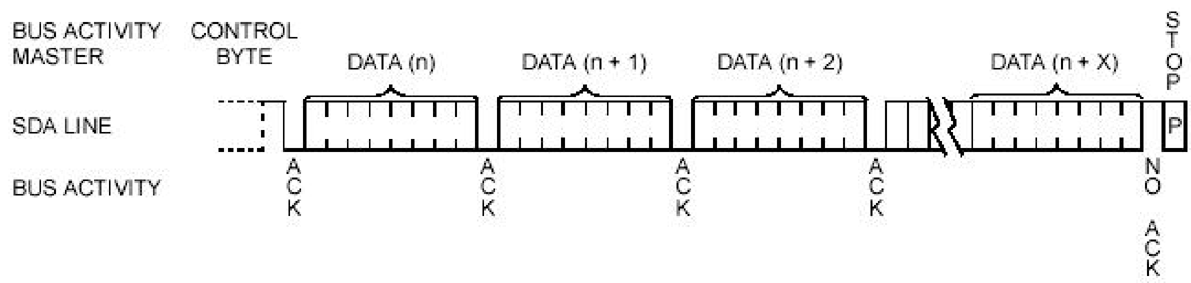


Figure 9 Sequential Read

3. I²C Temperature Sensor

The MAX6626 comprises a temperature sensor, programmable over-temperature alarm and an I²C-compatible serial interface. The temperature of the die is converted into digital values using the internal analog-to-digital converter. The converted result is stored in a temperature register, readable at any time through the serial interface. A dedicated alarm output (OT) is activated if the conversion result exceeds the value in the programmable high-temperature register. It also comes with a programmable fault queue, which sets the number of faults that must occur before the alarm activates. This prevents spurious alarms in noisy environments. The device functions as a slave and supports byte/word-read/write commands. The functional block diagram is shown in Figure 10.

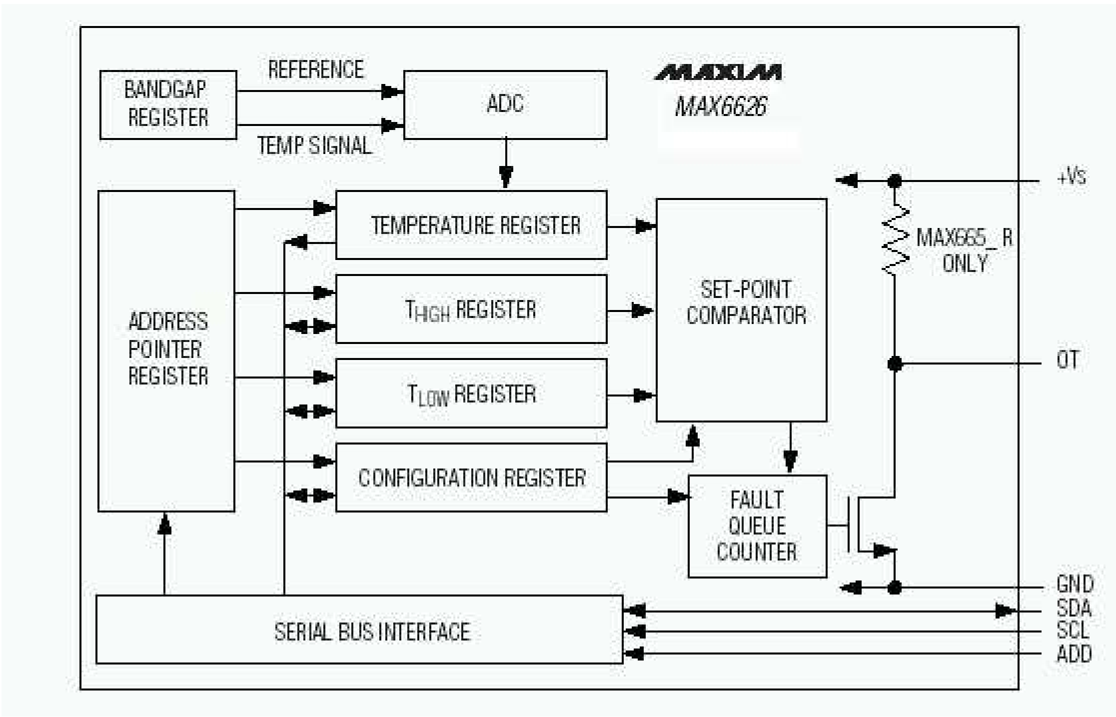


Figure 10 Block Diagram of Temperature Sensor

3.1 Addressing

Four separate addresses can be configured with the ADD pin, allowing up to four MAX6626s to be connected on the same bus. The table summarizes the different address selection. In this interface, the ADD pin is connected to GND and the address is set to 90 (hexadecimal).

Table 2 ADD Connection

ADD Connection	I²C-Compatible Address
GND	100 1000
V_S	100 1001
SDA	100 1010
SCL	100 1011

3.2 Control Registers

The operation is defined by the following registers:

a. The ***Pointer*** register is addressed first to determine the register to be acted on.

Table 3 Pointer Register

D7	D6	D5	D4	D3	D2	D1	D0	Register
0	0	0	0	0	0	0	0	Temperature
						0	1	Configuration
						1	0	T_{LOW}
						1	1	T_{HIGH}

b. The ***Temperature*** (TEMP) register is 12-bit, read-only and contains the latest temperature data. The register length is 16 bits with the unused bits masked to 0. The digital temperature is in °C using a two's complement format with the LSB corresponding to 0.0625°C.

Table 4 Temperature Register

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2-D0
MSB (Sign)	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	LSB	Unused 0

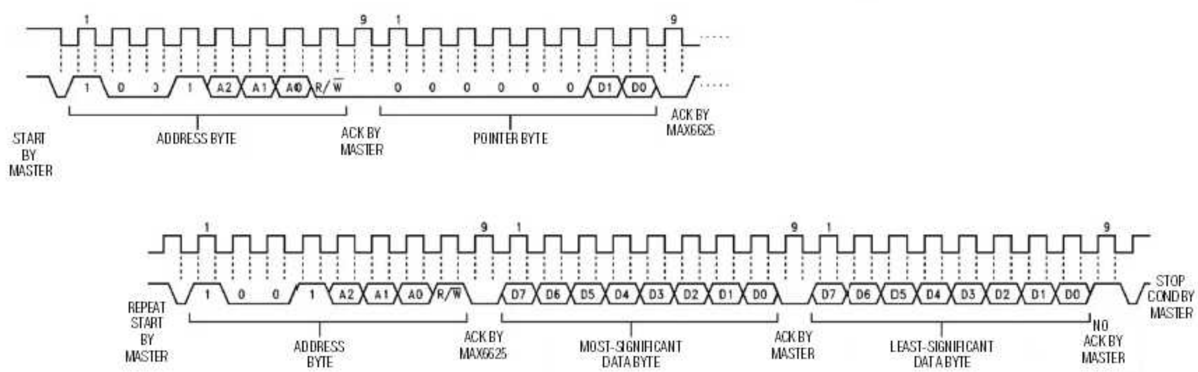


Figure 11 Reading of 2-byte Registers (TEMP, T_{HIGH} and T_{LOW})

c. The ***Configuration*** register is 8-bit, read/write and contains the fault queue depth, temperature alarm polarity select, interrupt mode select and shutdown control bits. Refer to Table 5 for the bit structure.

Table 5 Configuration Register

D7	D6	D5	D4	D3	D2	D1	D0	No. of Faults
			Fault Queue Depth		OT Polarity	Comparator or Interrupt Mode	Shutdown	
0	0	0	0	0	0: Active low 1: Active high	0: Comparator 1: Interrupt	0: Normal Operation 1: Shutdown	1
			0	1				2
			1	0				4
			1	1				6

Figures 12 and 13 show the timing diagrams for a read from and write to the configuration register respectively.

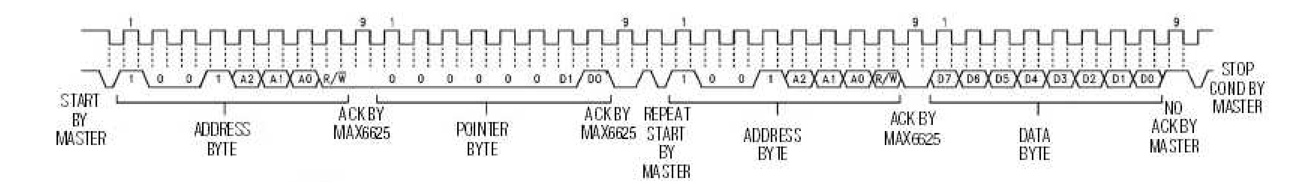


Figure 12 Read from Configuration Register

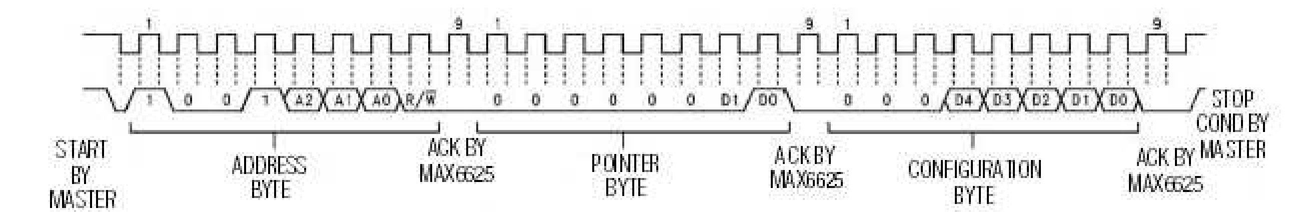


Figure 13 Configuration Register Write

c. The ***High-Temperature*** (T_{HIGH}) register is 9-bit, read/write and contains the value that triggers the over-temperature alarm. The ***Low-Temperature*** (T_{LOW}) register is 9-bit, read/write. It contains the value to which the temperature must fall before the over-temperature alarm is de-asserted in comparator mode. Refer to Table 6 for the bit structure of these two registers. The timing diagram for reading from and writing to are shown in Figures 11 and 14 respectively.

Table 6 T_{HIGH} and T_{LOW} Registers

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MSB	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	LSB	0	0	0	0	0	0	0

Notes: 1. D15: MSB is the sign bit
2. D6 to D0: Will read all zeros, cannot be written
3. LSB = 0.5°C

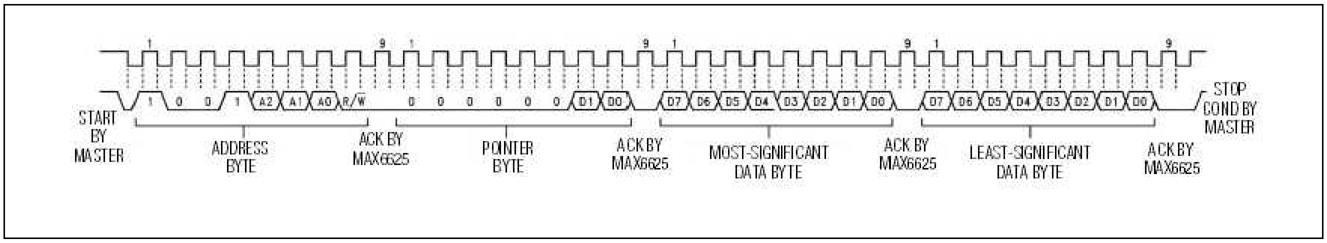


Figure 14 T_{HIGH} and T_{LOW} Write

3.3 Temperature Conversion

An on-chip bandgap reference produces a signal proportional to absolute temperature (PTAT), as well as the temperature-stable reference voltage necessary for the analog-to-digital conversion. The resolution of the digitized PTAT signal is 0.0625°C with the rate of conversion at 133ms. The temperature register contains the value of the most recently completed conversion.

3.4 Over-temperature Alarm

The polarity and modes (interrupt and comparator) of the dedicated over-temperature output pin (OT) are programmable through the configuration register. Fault queue depth defines the alarm activity.

- The programmable fault queue eliminates spurious alarm activity in noisy environments by setting the number of consecutive out-of-tolerance temperature readings that must occur before the OT alarm is triggered. Out-of-tolerance refer to a temperature reading above T_{HIGH} or below T_{LOW}.
- In comparator mode, OT is asserted when the number of consecutive conversions exceeding the value of the T_{HIGH} register is equal to the fault queue depth. OT will be de-asserted when the number of consecutive conversions below T_{LOW} is equal to the fault queue depth. For example, T_{HIGH}, T_{LOW} and fault queue depth are set to +75°C, +50°C and 4 respectively. OT will not assert until four consecutive conversions exceed +75°C. Similarly, OT will not be de-asserted until four consecutive conversions are below +50°C. Comparator mode allows autonomous clearing of an OT fault without the intervention of a master and is ideal for driving a cooling fan.
- In interrupt mode, the OT pin asserts an alarm for an under-temperature as well as an over-temperature fault, depending on certain conditions. If the fault queue is cleared at power-up, the IC looks for a T_{HIGH} fault after which it will then monitor for a T_{LOW} fault. After T_{LOW} fault, it will then monitor for a T_{HIGH} fault. This process will be repeated if OT is properly de-asserted each time. Once either fault has occurred, it remains active until de-asserted by a read of any register. The device will then monitor for a fault of the opposite type. For example, T_{HIGH}, T_{LOW} and fault queue depth are set to +75°C, +50°C and 4 respectively. OT will not assert until four consecutive conversions exceed +75°C. OT will then de-assert upon reading the temperature register. OT will then assert again after four consecutive conversions below +50°C.

3.5 Shutdown

In shutdown mode, the temperature register is set to H’8000 and the ADC is turned off (reducing the device current to 1µA). Upon exiting from shutdown, the value of the temperature register is H’8000 until the completion of the first temperature conversion.

4. I^{2}C 4-Digit 5 × 7 Matrix LED Display Driver

The MAX6953 is a serially interfaced display driver that can drive four digits of 5 × 7 cathode-row dot-matrix displays. It includes an ASCII 104-character font, multiplex scan circuitry, column and row drivers, static RAM to store each digit as well as font data for 24 user-definable characters. The segment current for the LEDs is set by an internal digit-by-digit digital brightness control. It also features a low-power shutdown mode, segment blinking, and a test mode that forces all LEDs to be on. Figure 15 shows the functional block diagram for the LED driver.

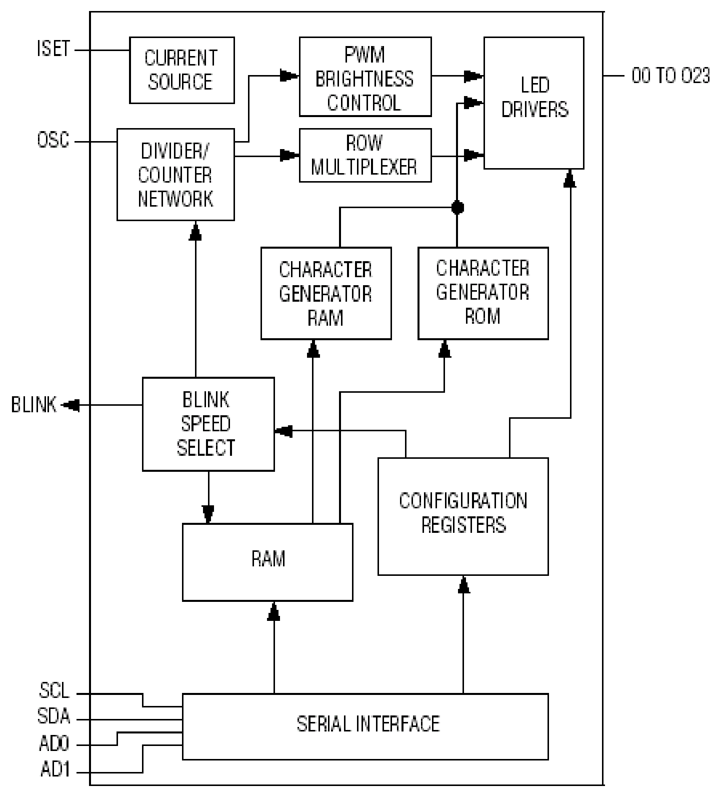


Figure 15 Block Diagram for MAX6953

4.1 Serial Addressing

The MAX6953 operates as a slave that sends and receives data through an I^{2}C-compatible 2-wire interface. The serial data (SDA) and clock (SCL) lines are used to achieve bi-directional communication between master (H8/38024) and slave (MAX6953). The master initiates all data transfers to and from the MAX6953, and also generates the SCL clock for synchronization of data transfer.

4.2 Start and Stop Conditions

Both SCL and SDA remain high when the interface is not busy. A master signals the beginning of a transmission with a START (S) condition by a high to low transition on SDA while SCL is high. When the master has finished communicating with the slave, it issues a STOP (P) condition by a low to high transition on SDA while SCL is high. Another bus transmission can then begin.

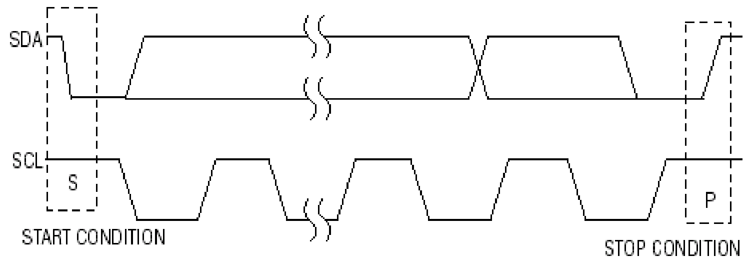


Figure 16 Start and Stop Conditions

4.3 Bit Transfer

One data bit is transferred during each clock pulse. The data on the SDA line must remain stable while SCL is high.

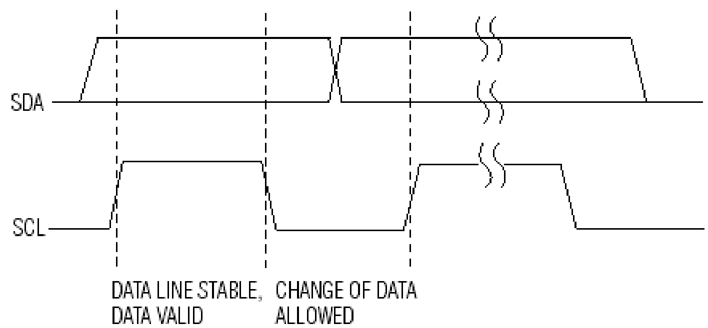


Figure 17 Bit Transfer

4.4 Acknowledge

The acknowledge bit is a clocked 9^{th} bit that the recipient uses to handshake receipt of each data byte. Refer to Figure 18. Thus, each transferred byte effectively requires 9 bits. The master generates the 9^{th} clock pulse, and the recipient pulls down SDA such that the SDA line is stable low during the high period of the acknowledge clock pulse. When the master is transmitting to the MAX6953, the acknowledge bit is generated by the MAX6953. When the MAX6953 is transmitting to the master, the master generates the acknowledge bit because the master is the recipient.

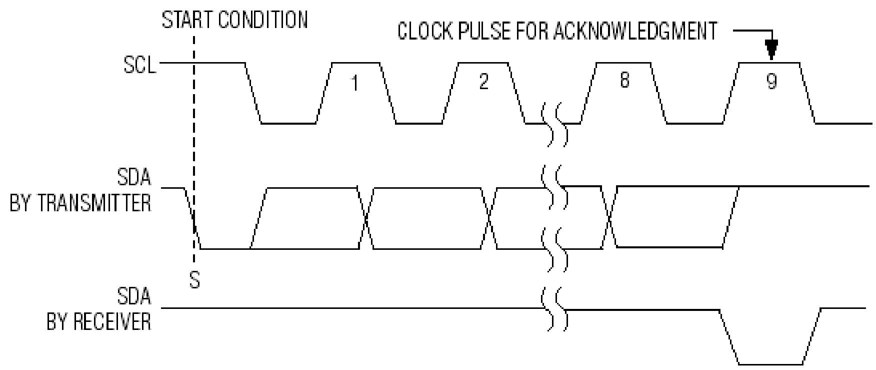


Figure 18 Acknowledge

4.5 Slave Address

The MAX6953 has a 7-bit slave address. The 8th bit following the 7-bit slave address is the R/$\overline{W}$ bit. It is low for a write command and high for a read command. The first 3 bits (A6, A5 & A4) of the MAX6953 slave address are always 101. The address input pins AD1 and ADO determine the slave address bits A3, A2, A1 and A0. These two input pins can be connected to GND, V+, SDA or SCL. Table 7 lists all the possible connections for AD1 and AD0 and the correspondingly addresses assigned to the MAX6953. Note that addresses A0 to AE (hexadecimal) have already been assigned to the EEPROM, the address B0 is allocated to the MAX6953.

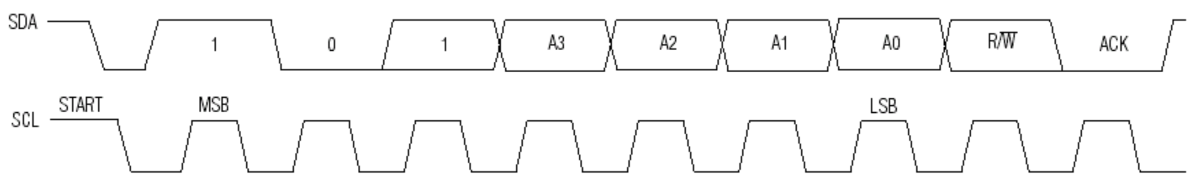


Figure 19 Slave Address

Table 7 MAX6953 Device Map

PIN		DEVICE ADDRESS						
AD1	AD0	A6	A5	A4	A3	A2	A1	A0
GND	GND	1	0	1	0	0	0	0
GND	V+	1	0	1	0	0	0	1
GND	SDA	1	0	1	0	0	1	0
GND	SCL	1	0	1	0	0	1	1
V+	GND	1	0	1	0	1	0	0
V+	V+	1	0	1	0	1	0	1
V+	SDA	1	0	1	0	1	1	0
V+	SCL	1	0	1	0	1	1	1
SDA	GND	1	0	1	1	0	0	0
SDA	V+	1	0	1	1	0	0	1
SDA	SDA	1	0	1	1	0	1	0
SDA	SCL	1	0	1	1	0	1	1
SCL	GND	1	0	1	1	1	0	0
SCL	V+	1	0	1	1	1	0	1
SCL	SDA	1	0	1	1	1	1	0
SCL	SCL	1	0	1	1	1	1	1

4.6 Writing Message Format

A write to the MAX6953 comprises the transmission of the slave address with the R/$\overline{W}$ set to zero followed by at least one byte of information. The first byte of information is the command byte, which determines the register to be written to by the next byte. If a STOP condition is detected after the command byte is received, then no further action other than the storage of the command byte is taken (Figure 20).

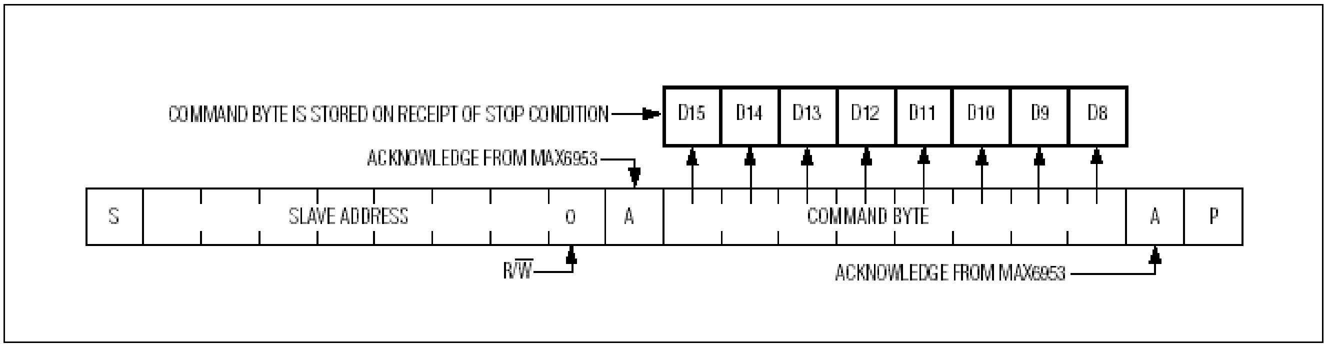


Figure 20 Command Byte Received

All bytes received after the command byte are data bytes. The first data byte goes into the internal register of the MAX6953 selected by the command byte (Figure 21).

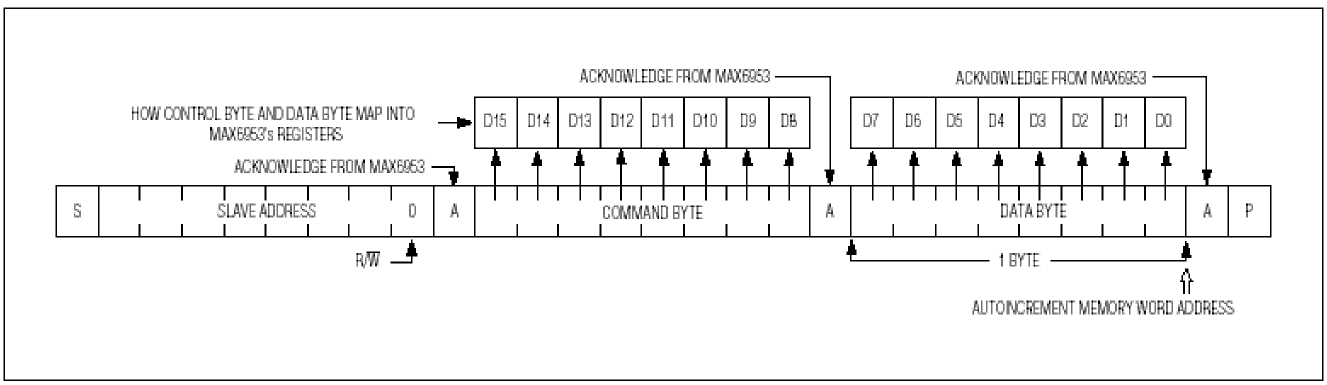


Figure 21 Command and Single Data Byte Received

If multiple data bytes are transmitted before a STOP condition is detected, these bytes are generally stored in subsequent MAX6953 internal registers because the command byte address generally auto-increments (Refer to Table 8 and Figure 22).

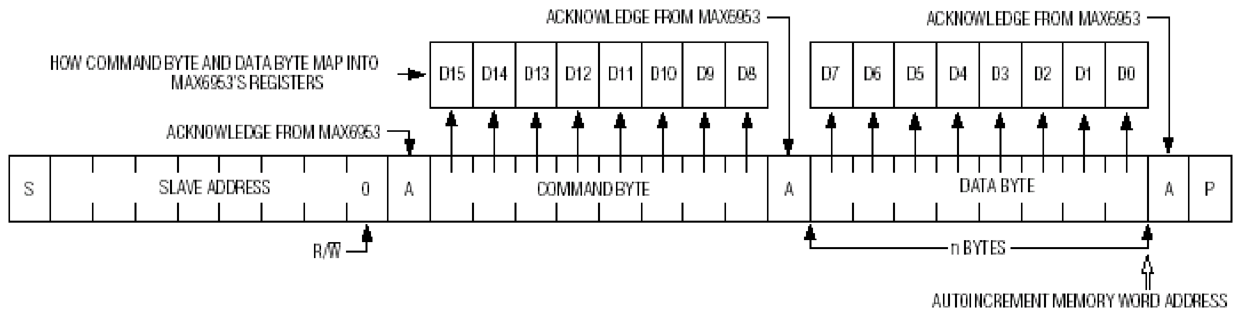


Figure 22 n Data Bytes Received

Table 8 Command Address Auto-increment Rules

COMMAND BYTE ADDRESS RANGE	AUTOINCREMENT BEHAVIOR
x0000000 to x0000100	Command byte address autoincrements after byte read or written.
x0000101	Command byte address remains at x0000101 after byte read or written, but the font address pointer autoincrements.
x0000110	Factory reserved; do not write to this register.
x000111 to x1111110	Command byte address autoincrements after byte read or written.
x1111111	Command byte address remains at x1111111 after byte read or written.

4.7 Reading Message Format

The MAX6953 is read using the MAX6953's internally stored command byte as address pointer, the same way the stored command byte is used as address pointer for a write. The pointer generally auto-increments after each data byte is read using the same rules described in Table 8. Thus, a read is initiated by performing a write (Figure 20). The master can now read n consecutive bytes from the MAX6953, with the first data byte being read from the register addressed by the initialized command byte (Figure 22).

5. Code

The functions are listed in these two C source files:

- I2C.c
  - Contains the main function
  - Performs initialization of the Serial Communication Interface (SCI) and temperature sensor
  - Tests the EEPROM, temperature sensor and the LED driver

- RW.c
  - Contains the general functions to emulate SDA and SCL

The flowchart of the main function is shown in Figure 23. The following steps are performed:

1. Initialization of SCI (2400bps, 1 stop bit, parity disabled), temperature sensor and LED driver.
2. Test EEPROM: Perform byte write, byte read, page write, read from current and sequential addresses. The test results are transmitted to the PC via SCI.
3. Read temperature, transmit to PC via SCI, and then display on Digits 1, 2 and 3.
4. Display '0' to '9' on Digit 0.
5. Repeat steps 2 to 4.

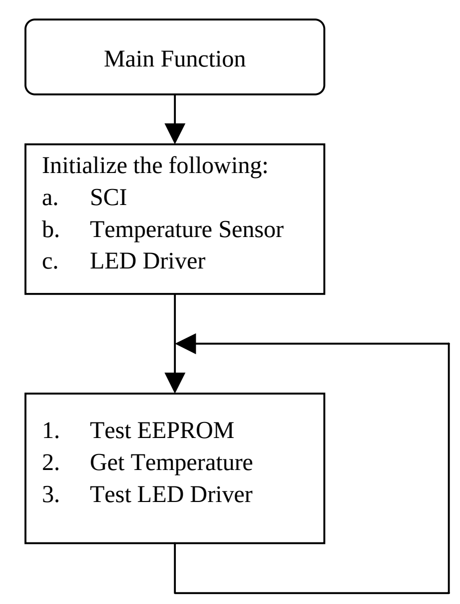


Figure 23 Flowchart of Main Function

```
/**********************************************************************/
/*                                                                    */
/*  FILE        :I2C.c                                                */
/*  DATE        :Fri, Dec 27, 2002                                    */
/*  DESCRIPTION :Main Program                                         */
/*  CPU TYPE    :H8/38024F                                            */
/*                                                                    */
/*  This file is generated by Hitachi Project Generator (Ver.2.1).    */
/*                                                                    */
/**********************************************************************/

#include "iodefine.h"
#include "i2c.h"
#include <stdio.h>
#include <machine.h>

//---------------------------------------------------------------------

//Device Addresses
#define  EEPROM_ADDR         0xA0  //B'10100000x
#define  T_SENSOR_ADDR       0x90  //B'10010000x
#define  LED_DRIVER_ADDR     0xB0  //B'10110000x

//---------------------------------------------------------------------

//LED Driver Registers
#define  DIGIT_0             0x60
#define  DIGIT_1             0x61
#define  DIGIT_2             0x62
#define  DIGIT_3             0x63

#define  DIGIT_0_1_INT_REG   0x01
#define  DIGIT_2_3_INT_REG   0x02

//---------------------------------------------------------------------

/*
   main()

   a. Initializes Serial Communication Interface (SCI) for debugging
   b. Initializes temperature sensor
   c. Initializes LED driver
   d. Repeat the following
      1. Test the EEPROM
      2. Obtain temperature reading
      3. Test the LED Driver
*/

void main(void)
{
   init_sci();

   init_temp_sensor();
```

```
    init_led_driver();

    PutStr("\r\nBeep Beep Beep");

    while(1)
    {
       test_eeprom();
       test_temp_sensor();
       test_led_matrix();
       wait(5); //short delay
    }
}

//------------------------------------------------------------------------

/*
    test_led_matrix() – display 0 to 9 on Digit 0
*/

void test_led_matrix(void)
{
    char  display_char;

    for (display_char = '0' ; display_char <= '9' ; display_char++)
    {
       LEDprint(display_char, DIGIT_0);
       wait(10);   //short delay
    }
}

//------------------------------------------------------------------------

/*
    test_eeprom()

    a. byte write
    b. byte read
    c. page write
    d. current address read
    e. sequential address read
*/

void test_eeprom(void)
{
    unsigned char  buf[16] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
                              0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};

    unsigned char  return_byte;
    unsigned char  *ptr;

    PutStr("\r\n\nEEPROM Testing:");

    //Byte Write
    PutStr("\r\nByte Write");
```

```
    if (I2cWrite(EEPROM_ADDR, buf, 1, 0x00) != OP_DONE)
       PutStr(" -> Fail!");
    else
       PutStr(" -> OK");

    //Need to check if write is complete
    if (CheckWriteReady() == 1)
    {
       //Byte Read
       PutStr("\r\nByte Read");
       return_byte = I2cRead(EEPROM_ADDR, ptr, 1, 0x00);
    }

    //Page Write
    PutStr("\r\nPage Write");
    if (I2cWrite(EEPROM_ADDR, buf, 16, 0x00) != OP_DONE)
       PutStr(" -> Fail!");
    else
       PutStr(" -> OK");

    //Need to check if write is complete
    if (CheckWriteReady() == 1)
    {
       //Current Address Read
       PutStr("\r\nCurrent Address Read");
       return_byte = I2cCurrentRead(EEPROM_ADDR, buf, 0x00);

       //Sequential Read
       PutStr("\r\nSequential Read");
       return_byte = I2cRead(EEPROM_ADDR, ptr, 16, 0x00);
    }
}

//-----------------------------------------------------------------------

/*
    test_temp_sensor()

    a. Get temperature reading
    b. Convert from binary to floating point
    c. Transmit temperature to PC via SCI
    d. Display temperature on Digits 1, 2 and 3
*/

void test_temp_sensor(void)
{
    unsigned char  return_byte;
    unsigned char  tens, ones, tenths;

    unsigned int   return_code;

    float          degree;

    //read from temperature sensor
```

```
   return_code = I2cRead_T_Sensor(T_SENSOR_ADDR, 0x00);

   if (return_code == 0x8000)
      PutStr("SHUT DOWN!");
   else
      PutStr("\r\n\nTemperature : ");

   degree = ConvertBinary2Temp(return_code);

   //For example, temperature = 37.1 degree
   //tens = 3, ones = 7 & tenths = 1
   tens = 0;
   ones = 0;
   tenths = 0;

   while (degree >= 10)
   {
      tens++;
      degree -= 10;
   }

   while (degree >= 1)
   {
      ones++;
      degree -= 1;
   }

   while (degree >= 0.1)
   {
      tenths++;
      degree -= 0.1;
   }

   //Transmit to PC via SCI
   char_put(tens + 0x30);
   char_put(ones + 0x30);
   char_put(0x2E);          //decimal point
   char_put(tenths + 0x30);

   //Display on dot-matrix LED
   LEDprint(tens + 0x30, DIGIT_1);
   LEDprint(ones + 0x30, DIGIT_2);
   LEDprint(tenths + 0x30, DIGIT_3);
}

//----------------------------------------------------------------------

/*
   init_temp_sensor()
*/

void init_led_driver(void)
{
   unsigned char  return_byte;
```

```
    //Configure MAX6953 driver: wake from shutdown mode
    SendStartBit();
    SendByte((LED_DRIVER_ADDR) & 0xfe);
    SendByte(0x04);   //configuration register
    SendByte(0x01);   //select normal operation as power-on default -> shutdown
    SendStopBit();

    //Set the intensity register for digits 0 & 1 to 6/16 duty cycle
    //Set the intensity register for digits 2 & 3 to 6/16 duty cycle
    //Write 0x66 to both 0x01 and 0x02 reg of MAX6953EPL
    SendStartBit();
    SendByte((LED_DRIVER_ADDR) & 0xfe);   //Send Slave Address byte
    SendByte(DIGIT_0_1_INT_REG);          //Send COMMAND byte
    SendByte(0x66);                       //Send data byte 00-min, FF-max
    SendStopBit();                        //Send stop bit
    SendStartBit();
    SendByte((LED_DRIVER_ADDR) & 0xfe);   //Send Slave Address byte
    SendByte(DIGIT_2_3_INT_REG);          //Send COMMAND byte
    SendByte(0x66);                       //Send data byte 00-min, FF-max
    SendStopBit();                        //Send stop bit
}

//-----------------------------------------------------------------------

/*
    init_temp_sensor()
*/

void init_temp_sensor(void)
{
    unsigned char  return_byte;

    SendStartBit();
    SendByte((T_SENSOR_ADDR) & 0xfe);     //Send slave address byte
    SendByte(0x01);                       //Configuration Register of sensor
    SendByte(0x00);                       //Wake up device
    SendStopBit();                        //Send stop bit

    SendStartBit();
    SendByte((T_SENSOR_ADDR) & 0xfe);     //Send slave address byte
    SendByte(0x01);                       //Configuration Register of sensor
    SendStopBit();

    SendStartBit();
    SendByte((T_SENSOR_ADDR) | 0x01);     //Read from Configuration Register
    return_byte = GetByte();
    SendStopBit();

    //THIGH = 80 degrees
    //TLOW  = 0 degrees
    SendStartBit();
    SendByte((T_SENSOR_ADDR) & 0xfe);     //Send Slave Address byte
    SendByte(0x03);                       //Set Max Temperature of Sensor
    SendByte(0x50);                       //msbByte
```

```
   SendByte(0x00);                       //lsbByte
   SendStopBit();                        //Send stop bit

   SendStartBit();
   SendByte((T_SENSOR_ADDR) & 0xfe);     //Send slave address byte
   SendByte(0x02);                       //Set Min Temperature of Sensor
   SendByte(0x00);                       //msbByte
   SendByte(0x00);                       //lsbByte
   SendStopBit();                        //Send stop bit
}

//----------------------------------------------------------------------

/*
   This routine is written for MAXIM 12-bit Temperature Sensors.
   MAX6626 is a 12-bit i2c compatible sensors.

   input:
   unsigned char slave_addr - refer to the address preset
   unsigned char ptr_reg - refer to the pointer register
      0x00 temperature
      0x01 configuration
      0x02 high-temperature
      0x03 low-temperature

   return:
   unsigned int - current temperature in 16bits
*/

unsigned int I2cRead_T_Sensor(unsigned char slave_addr, unsigned char ptr_reg)
{
   unsigned int theWORD;
   unsigned char msbBYTE, lsbBYTE;

   if (CheckBusState() != TRUE)
      return(BUS_BUSY);

   SendStartBit();

   //Send slave address with write command
   if (SendByte((slave_addr) & 0xfe) != LOW)              return(NO_RESPONSE);

   //Send Pointer byte
   if (SendByte(ptr_reg) != LOW)
      return(NO_RESPONSE);

   SendStopBit();     //Send STOP bit

   SendStartBit();

   //Send slave address with read command
   if (SendByte((slave_addr) | 0x01) != LOW)
      return(NO_RESPONSE);
```

```
   msbBYTE = GetByte();

   SendBit(LOW);              //Ack it low!

   lsbBYTE = GetByte();

   SendStopBit();             //Send STOP bit

   theWORD = (unsigned int)msbBYTE << 8;
   theWORD = theWORD + lsbBYTE;

   return(theWORD);
}

//----------------------------------------------------------------------

/*
   ConvertBinary2Temp() Converts temperature from binary to floating point
*/

float ConvertBinary2Temp(unsigned int temp)
{
   float degree;
   float scaleMX;
   int   temp1;

   scaleMX = 0.0625;

   temp1 = temp & 0x7FFF;    //throw away signed bit
   temp1 = temp1>>4;   //get rid of last 4 bits(lsb)

   degree = (float)temp1 * scaleMX;

   return(degree);
}

//----------------------------------------------------------------------

/*
   LEDprint(): Display on the matrix LED.
*/

void LEDprint(char character, unsigned char digit_position)
{
   unsigned char  error_code;

   error_code = I2cMatrixLEDdriver(LED_DRIVER_ADDR, digit_position,
                                   character);
}

//----------------------------------------------------------------------

/*
   This routine is used for MAXIM Matrix LED Display Driver.
```

```
   MAX6953 is a 2-wire I2C interface driver.

   slave_addr is the address preset for MAX6953.

   command_byte refer to the command instruction to be given to MAX6953.

   data_byte refer to the 8-bit data
*/

unsigned char I2cMatrixLEDdriver(unsigned char slave_addr, unsigned char
                                  command_byte, unsigned char data_byte)
{
   /*
   Command Address:

   StartBit [S] -> Slave Address (7bit + 1 R/W bit) -> ACK (MAX6953) ->
   COMMAND Byte -> ACK (MAX6953) -> DATA byte -> ACK (MAX6953) -> StopBit [P]

   Refer to MAX6953 data sheet for command and data instruction
   */

   //Check if I2C bus is busy
   if (CheckBusState() != TRUE)
      return(BUS_BUSY);

   SendStartBit();                      //Send start bit
   //Send slave address and write command
   if (SendByte((slave_addr) & 0xfe) != LOW)
      return(NO_RESPONSE);

   if (SendByte(command_byte) != LOW)   //Send COMMAND byte
      return(NO_RESPONSE);

   if (SendByte(data_byte) != LOW)      //Send DATA byte
      return(NO_RESPONSE);

   SendStopBit();                       //Send stop bit
}

//-----------------------------------------------------------------------

/*
   init_sci() : Sets up the Serial Communication Interface for debugging
*/

void init_sci(void)
{
   //SCR3 : |TIE|RIE|TE|RE|MPIE|TEIE|CKE1|CKE0|
   //TIE : Transmit interrupt enable
   //RIE : Receive interrupt enable
   //TE  : Transmit enable
   //RE  : Receive enable
   //MPIE : Multiprocessor interrupt enable
   //TEIE : Transmit end interrupt enable
```

```
    //CKE1 : Clock enable 1
    //CKE0 : Clock enable 0

    //CKE1 = CKE0 = 0
    //asynchronous mode, internal clock source, SCK32 functions as I/O port
    P_SCI3.SCR3.BYTE &= 0x00; //clear TE & RE

    //SMR : |COM|CHR|PE|PM|STOP|MP|CKS1|CKS0| : |0|0|0|0|0|0|0|0|
    //COM : Communication Mode  : 0 : asynchronous mode
    //CHR : Character Length    : 0 : character length = 8 bits
    //PE  : Parity Enable       : 0 : parity bit addition and checking disabled
    //PM  : Parity Mode         : 0 : even parity (no effect since no parity)
    //STOP: Stop Bit Length     : 0 : 1 stop bit
    //MP  : Multiprocessor Mode : 0 : multiprocessor comm function disabled
    //|CKS1|CKS0| : Clock Select: |0|0| : clock source for baud rate gen = clk
    P_SCI3.SMR.BYTE = 0x00;

    //For clk = 10MHz, bit rate = 2400 bps, n = 0, N = 64
    P_SCI3.BRR = 64;

    //minimum of 1-bit delay = 417ns
    nop();
    nop();
    nop();

    //SPCR : |---|---|SPC32|---|SCINV3|SCINV2|---|---| : |1|1|1|0|0|0|0|0|
    //SPC32 = 1 : P42 functions as TXD32 output pin
    //need to set TE bit in SCR3 after setting this bit to 1
    //SCINV3 = 0 : TXD32 output data is not inverted
    //SCINV2= 0 : RXD32 input data is not inverted
    //Bits 7 and 6 are reserved and always read as 1
    //Bits 4, 1 and 0 are reserved and only 0 can be written to these bits
    P_SCI3.SPCR.BYTE = 0xE0;

    P_SCI3.SCR3.BYTE |= 0x30; //Set TE & RE
}

//-----------------------------------------------------------------------

/*
    char_put() : Transmits a character to the PC for debugging purposes.
*/

void char_put(char OutputChar)              //Serial Port
{
    //SSR  : |TDRE|RDRF|OER|FER|PER|TEND|MPBR|MPBT|
    //TDRE : transmit data register empty
    //RDRF : receive data register full
    //OER  : overrun error
    //FER  : framing error
    //PER  : parity error
    //TEND : transmit end
    //MPBR : Multiprocessor bit receive
    //MPBT : Multiprocessor bit transfer
```

```
   while ((P_SCI3.SSR.BIT.TDRE) == 0);        //Wait for TDRE = 1

   P_SCI3.TDR = OutputChar;
}

//------------------------------------------------------------------------

/*
   PutStr() : Transmits a string of characters to the PC for debugging
purposes.
*/

void PutStr(char *str)
{
   while (*str != 0)
   {
      char_put(*str++);
   }
}

//------------------------------------------------------------------------

/*
   wait(): Generates a software delay.
*/

void wait(unsigned int time)
{
   unsigned int   i, j;

   for (i = 0 ; i < time ; i++)
   {
      for (j = 0 ; j < 3500 ; j++)
      {
      }
   }
}

//------------------------------------------------------------------------
```

```
/*****************************************************************/
/*                                                                */
/*  FILE        :RW.c                                             */
/*  DATE        :Fri, Dec 27, 2002                                */
/*  DESCRIPTION :Function Program                                 */
/*  CPU TYPE    :H8/38024F                                        */
/*                                                                */
/*  This file is generated by Hitachi Project Generator (Ver.2.1).    */
/*                                                                */
/*****************************************************************/

//-----------------------------------------------------------------

#include "i2c.h"
#include "iodefine.h"

//-----------------------------------------------------------------

/*
   SclIn()
   Defines the SCL as an input pin and checks the port status (low or high).
*/

unsigned char SclIn(void)
{
   SCL_IO_REG &= SCL_IO_RESET_BIT;      //Set to Input

   if (SCL_DATA_REG & SCL_DATA_SET_BIT) //Check pin status
   {
         return(HIGH);
   }
   else
   {
      return(LOW);
   }
}

//-----------------------------------------------------------------

/*
   SdaIn()
   Defines the SDA as an input pin and checks the port status (low or high).
*/

unsigned char SdaIn(void)
{
   SDA_IO_REG &= SDA_IO_RESET_BIT;            //Set to Input

   if (SDA_DATA_REG & SDA_DATA_SET_BIT)
   {
      return(HIGH);                           //Check pin status
   }
   else
   {
```

```
      return(LOW);
   }
}

//-----------------------------------------------------------------------

/*
   SclOut()
   Defines the SCL pin as an output pin and sets it to the level
   determined by the parameter.
*/

void SclOut(unsigned char status)
{
   if (status == LOW)
   {
      SCL_DATA_REG = 0;                  //Drive Port LOW
   }
   else
   {
      SCL_DATA_REG = 1;                  //Drive Port High
   }

   SCL_IO_REG |= SCL_IO_SET_BIT;         //Set to output
}

//-----------------------------------------------------------------------

/*
   SdaOut()
   Defines the SDA as an output pin and sets it to the level determined by the
   parameter.
*/

void SdaOut(unsigned char status)
{
   if (status == LOW)
   {
      SDA_DATA_REG = 0;                  //Drive Port LOW
   }
   else
   {
      SDA_DATA_REG = 1;                  //Drive Port High
   }

   SDA_IO_REG |= SDA_IO_SET_BIT;         //Set to output
}

//-----------------------------------------------------------------------

/*
   Delay()
   Provide an internal minimum delay time to bridge the undefined
   region of a falling edge of SCL to avoid unintended generation
```

```
   of unwanted signal.
*/

void Delay(void)
{
   unsigned char  i = 0;

   while (i < 20)
   {
      i++;
   }
}

//-----------------------------------------------------------------------

void Delay2x(void)
{
   Delay();
   Delay();
}

//-----------------------------------------------------------------------
//All codes below here are independent with hardware, such as microprocessor,
//I/O port, or etc.

/*
   CheckBusState()
Determine whether the I2C bus is free (both SCL and SDA = HIGH) or in busy
state.
*/

unsigned char CheckBusState(void)
{
   if ((SclIn() == HIGH) && (SdaIn() == HIGH))
   {
      return(TRUE);
   }
   else
   {
      return(FALSE);
   }
}

//-----------------------------------------------------------------------

/*
   SendStartBit(): Issues a START condition
*/

void SendStartBit(void)
{
   Delay();
   SdaOut(LOW);
   Delay2x();
```

```
   Delay2x();
   Delay2x();
   Delay2x();
   SclOut(LOW);
   Delay();
}

//----------------------------------------------------------------------

/*
   SendBit(): Send out data in bit format
*/

void SendBit(unsigned char data_byte)
{
   SclOut(LOW);

   Delay();

   if (data_byte != 0)
   {
      SdaOut(HIGH);
   }
   else
   {
      SdaOut(LOW);
   }

   Delay();

   SclOut(HIGH);

   while (SclIn() != HIGH) {}  //wait for slow device to release clock

   Delay2x();
}

//----------------------------------------------------------------------

/*
   GetBit(): Receive data input in bit format
*/

unsigned char GetBit(void)
{
   unsigned char  temp;

   SclOut(LOW);
   temp = SdaIn();
   Delay2x();
   SclOut(HIGH);
   while (SclIn() != HIGH) {}  //wait for slow device to release clock
   Delay();
   temp = SdaIn();
```

```
   Delay();
   return(temp);
}

//-------------------------------------------------------------------------

/*
   GetAck():
   Getting ACK is similar to GetBit, but this is critical operation since
   master must pull SDA high before it finds out whether there is a ACK
   (SDA is low) or not.
*/

unsigned char GetAck(void)
{
   unsigned char  temp;

   SclOut(LOW);
   Delay();
   SdaOut(HIGH);
   temp = SdaIn();
   Delay();
   SclOut(HIGH);
   while (SclIn() != HIGH) {}  //wait for slow device to release clock
   Delay();
   temp = SdaIn();
   Delay();
   return(temp);
}

//-------------------------------------------------------------------------

/*
   SendByte(): Send out a byte starting with most significant bit (MSB) first.
*/

unsigned char SendByte(unsigned char data_byte)
{
   unsigned char  i;
   unsigned char  mask;

   mask = 0x80;        //send out MSB first

   for (i = 0 ; i < 8 ; i++)
   {
      SendBit(data_byte & mask);
      mask >>= 1;
   }

   return(GetAck());
}

//-------------------------------------------------------------------------
```

```
/*
   GetByte(): Get a byte of data starting with most significant bit (MSB)
*/

unsigned char GetByte(void)
{
   unsigned char  temp1, temp2;
   unsigned char  i,mask;

   mask = 0x80;

   temp2 = 0;

   for (i = 0; i < 8 ; i++)
   {
      temp1 = GetBit() * mask;
      temp2 += temp1;
      mask >>= 1;
   }
   return(temp2);
}

//-----------------------------------------------------------------------

/*
   SendStopBit(): Send a STOP condition to terminate the operation
*/

void SendStopBit(void)
{
   SclOut(LOW);
      Delay();
      SdaOut(LOW);
      Delay();
      SclOut(HIGH);
      Delay2x();
      SdaOut(HIGH);
}

//-----------------------------------------------------------------------

/*
   I2cWrite()

   a. Byte Write
      1. Start Bit
      2. Control Byte
      3. Ack
      4. Word Address
      5. Ack
      6. Data
      7. Ack
      8. Stop Bit
```

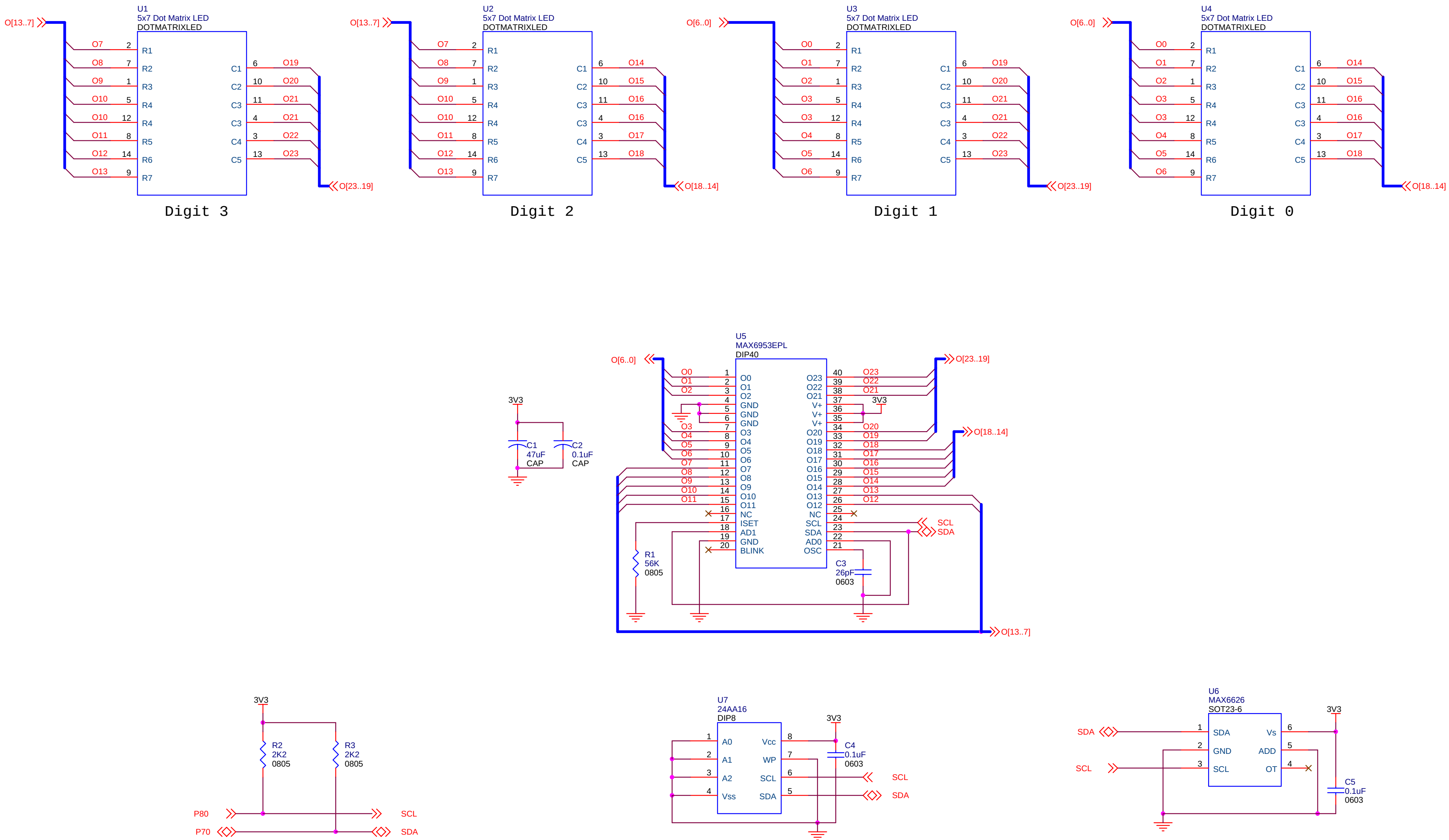
```
   b. Page Write
      1. Start Bit
      2. Control Byte
      3. Ack
      4. Word Address
      5. Ack
      6. Data(n)
      7. Ack
      8. Data(n + 1)
      9. Ack
         ...
      10.   Data(n + 15)
      11.   Ack
      12.   Stop Bit
*/

unsigned char I2cWrite(unsigned char slave_addr, unsigned char *buf_ptr,
                      unsigned char length, unsigned char word_addr)
{
   unsigned int   i;

   if (CheckBusState() != TRUE)
   {
      PutStr(" -> BUS_BUSY!");
      return(BUS_BUSY);
   }
   SendStartBit();

   //Send address and write command
   if (SendByte((slave_addr) & 0xfe) != LOW)
   {
      PutStr(" -> NO_RESPONSE-1");
      return(NO_RESPONSE);
   }

   //Send word address
   if (SendByte(word_addr) != LOW)
   {
      PutStr(" -> NO_RESPONSE-2");
      return(NO_RESPONSE);
   }

   for (i = 0 ; i < length ; i++)
   {
      //Write data
      if (SendByte(*buf_ptr++) != LOW)
      {
         PutStr(" -> ERR_RESPONSE");
         return(ERR_RESPONSE);
      }
   }

   SendStopBit();
```

```
   return(OP_DONE);
}

//-----------------------------------------------------------------------

unsigned char I2cRead(unsigned char slave_addr, unsigned char *buf_ptr,
                  unsigned char length, unsigned char word_addr)
{
   unsigned char  i = 0, j = 0;
   unsigned char  ref_data[16] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66,
                                 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD,
                                 0xEE, 0xFF};
   unsigned char  DataBuffer[256];
   unsigned char  error = 0;

   if (CheckBusState() != TRUE)
   {
      PutStr(" -> BUS_BUSY");
      return(BUS_BUSY);
   }

   SendStartBit();

   //Send dummy address and write command
   if (SendByte((slave_addr) & 0xfe) != LOW)
   {
      PutStr(" -> NO_RESPONSE-1!");
      return(NO_RESPONSE);
   }

   //Send high word address
   if (SendByte(word_addr) != LOW)
   {
      PutStr(" -> NO_RESPONSE-2!");
      return(NO_RESPONSE);
   }

   SdaOut(HIGH);                    //Pull-up SDA line

   SendBit(HIGH);

   SendStartBit();

   //Send address and read command
   if (SendByte((slave_addr) | 0x01) != LOW)
   {
      PutStr(" -> NO_RESPONSE-3!");
      return(NO_RESPONSE);
   }

   for (i = 0 ; i < length - 1 ; i++)
   {
      DataBuffer[i] = GetByte();   //read data
      SendBit(LOW);               //ack it low
```

```
   }

   //Get last data byte and ack high
   DataBuffer[length - 1] = GetByte();
   SendBit(HIGH);
   SendStopBit();

   for (i = 0 ; i < length ; i++)
   {
      if (DataBuffer[i] != ref_data[word_addr + i])
      {
         error++;
      }
   }

   if (error)
   {
      PutStr(" -> Incorrect Data!");
   }
   else
   {
      PutStr(" -> OK");
   }

   return(OP_DONE);
}

//-----------------------------------------------------------------------

unsigned char I2cCurrentRead(unsigned char slave_addr,
                     unsigned char *buf_ptr,
                     unsigned char word_addr)
{
   unsigned char  ref_data[16] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66,
                                  0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD,
                                  0xEE, 0xFF};

   SendStartBit();

   //Send address and read command
   if (SendByte((slave_addr) | 0x01) != LOW)
   {
      PutStr(" -> NO_RESPONSE!");
      return(NO_RESPONSE);
   }

   *buf_ptr = GetByte();     //get data and ack high

   SendBit(HIGH);
   SendStopBit();

   if (*buf_ptr != ref_data[word_addr])
   {
      PutStr(" -> Incorrect Data!");
```

```
    }
    else
    {
       PutStr(" -> OK");
    }

    return(OP_DONE);
}

//-----------------------------------------------------------------------

/*
    Since Microchip devices such as 24AA16 will not acknowledge during
    the internal write cycle, this can be used to determined when this
    cycle is complete so that the master can proceed with next operation.

    Acknowledge Polling
    a. Send write command
    b. Send stop condition to initiate write cycle
    c. Send start bit
    d. Send control byte with r/w_n = 0
    e. If device acknowledge, goto f. Else go to c
    f. Ready for next operation

    Note that (c) to (e) - internal write cycle
*/

char CheckWriteReady(void)
{
    unsigned int   i = 0;

    while (i < 4)
    {
       SendStartBit();

       if (SendByte((0xa0) | 0x00) == LOW)
       {
          SendStopBit();
          return (1);
       }

       SendStopBit();

       i++;
    }

    return (0);
}

//-----------------------------------------------------------------------
```

6. Hardware Design



7. References

1. *The I²C-Bus Specification (Version 2.1)*, January 2000, Philips Semiconductor.

2. *H8/38024, H8/38024S, H8/38024F-ZTAT Group Hardware Manual*, Revision 4, 26 May 2003, Renesas Technology Corporation.

3. *24AA16/ 24LC16B 16K I²C Serial EEPROM*, 2002, Microchip Technology Inc.

4. *MAX6626 12-bit Temperature Sensor with I2C-compatible Serial Interface*, 2002, Maxim Integrated Products.

5. *MAX6953 2-wire Interfaced 4-digit 5x7 Matrix LED Display Driver*, 2002, Maxim Integrated Products.

6. *Application Note on Serial Peripheral Interface (SPI™) & Inter-IC (I²C™)*, 2003, Renesas Technology Corporation.

7. *Application Note on Interfacing to EEPROM with I²C™ Emulation (Port)*, 2003, Renesas Technology Corporation.

Note: I²C is a registered trademark of Philips.

Revision Record

Rev.	Date	Description: Page	Description: Summary
1.00	Sep.18.03	—	First edition issued

Keep safety first in your circuit designs!

1. Renesas Technology Corporation puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage.
   Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corporation product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation or a third party.
2. Renesas Technology Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor for the latest product information before purchasing a product listed herein.
   The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.
   Please also pay attention to information published by Renesas Technology Corporation by various means, including the Renesas Technology Corporation Semiconductor home page (http://www.renesas.com).
4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
5. Renesas Technology Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
6. The prior written approval of Renesas Technology Corporation is necessary to reprint or reproduce in whole or in part these materials.
7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.
   Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
8. Please contact Renesas Technology Corporation for further details on these materials or the products contained therein.