

BACHELORTHESIS

Jacob Seal

I2C Bus communication for distributed embedded applications running in a network of ARM-based microcontrollers

FACULTY OF ENGINEERING AND COMPUTER SCIENCE

Department of Information and Electrical Engineering

Fakultät Technik und Informatik

Department Informations- und Elektrotechnik

Jacob Seal

I2C-bus communication for distributed embedded
applications running in a network of ARM-based
microcontrollers

Bachelor Thesis based on the examination and study regulations for the
Bachelor of Engineering degree programme

Information Engineering

at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner: Prof. Dr. Pawel Buczek

Second examiner: Prof. Dr. Lutz Leutelt

Day of delivery: 25. August 2020

ABSTRACT

Jacob Seal

Title of the paper

I2C-bus communication for distributed embedded applications running in a network of ARM-based microcontrollers

Keywords

Microcontroller, ARM, I2C, bus communication, embedded systems

Abstract

I2C is a two-wire serial bus communications protocol. The purpose of the I2C bus is to allow robust and efficient communication between I2C enabled devices on the same bus. Each device on the bus has its own unique address and will be configured as either the master or the slave. A practical example of an I2C bus would be a microcontroller as the master, and several sensors as slaves with single a LED readout to display sensor data.

A driver is required to configure the operating parameters and access the memory registers of an I2C enabled IC. A driver exists as an interface between the main program and the hardware. It is not desirable for the main program to touch the memory registers on the microcontroller directly. The driver is written in such a way that it provides a layer of abstraction between the user/program and the registers. Development and testing of such a driver is the technical goal of this thesis.

Thema der Bachelorthesis

I2C-Bus-Kommunikation für dezentrale eingebettete Applikationen, die in einem Netzwerk von ARM-basierten Mikrocontrollern laufen

Stichworte

Mikrocontroller, ARM, I2C, Buskommunikation, eingebettete Systeme

Kurzzusammenfassung

I2C ist ein serielles Zweidraht-Bus-Kommunikationsprotokoll. Der Zweck des I2C-Busses besteht darin, eine robuste und effiziente Kommunikation zwischen I2C-fähigen Geräten am selben Bus zu ermöglichen. Alle Geräte am Bus haben ihre eigene eindeutige Adresse und werden entweder als Master oder als Slave konfiguriert. Ein praktisches Beispiel für einen I2C-Bus wäre ein Mikrocontroller als Master und mehrere Sensoren als Slaves, mit einer einzigen LED-Anzeige zur Anzeige von Sensordaten.

Ein Treiber ist erforderlich, um die Betriebsparameter zu konfigurieren und auf die Speicherregister eines I2C-fähigen ICs zuzugreifen. Ein Treiber existiert als Schnittstelle zwischen dem Hauptprogramm und der Hardware. Es wäre unvorteilhaft, dass das Hauptprogramm die Speicherregister auf dem Mikrocontroller direkt berührt. Der Treiber ist so geschrieben, dass er eine Abstraktionsschicht zwischen dem Benutzer/Programm und den Registern bietet. Die Entwicklung und Erprobung eines solchen Treibers ist das technische Ziel dieser Arbeit. Für das Testen werden die wichtigsten Anwendungsfälle mit Ausgaben dokumentiert, die belegen, dass der Treiber gemäß den funktionalen Anforderungen arbeitet.

*Dedicated to my loving wife, Christy.
I absolutely could not have done this without you.
Here's to the next adventure.....*

Table of Contents

ABSTRACT	<i>i</i>
List of Figures	<i>vi</i>
List of equations	<i>vii</i>
Acronyms and abbreviations	<i>viii</i>
Glossary	<i>ix</i>
CHAPTER 1	1
INTRODUCTION	1
1.1 Scope of thesis	2
1.2 Functional Requirements	2
1.3 Software Requirements	3
1.4 required hardware	3
1.5 software utilized	4
CHAPTER 2	5
I2C IN DETAIL	5
2.1 Introduction to I2C	6
2.2 Hardware Design	6
2.3 Master Slave relationship	7
2.4 Arbitration	9
2.5 Start and Stop bit	10
2.6 I2C Addresses (7-bit vs 8-bit)	10
2.7 Acknowledge or Not Acknowledged	11
2.8 Complete I2C Data Packet	12
2.9 Read and Write	12
2.9.1 Write Operation	13
2.9.2 Read Operation	13
2.10 FIFO usage description	14
CHAPTER 3	15
I2C DRIVER DESIGN	15
3.1 HAL (Hardware Abstraction Layer)	16
3.2 Assumptions	17
3.3 Driver Architecture	17
3.3.1 Organizational Structure	17
3.3.2 I2cCores.h	18
3.3.3 I2cPinouts.h	18

3.3.4 <i>hall2c.h</i>	19
3.3.5 <i>I2c.c</i>	20
3.4 Circular Buffer Interface	20
3.5 NVIC integration	21
3.6 User Accessible Functions	22
3.6.1 <i>hall2cDriverInit()</i>	22
3.6.2 <i>hall2cCoreInit()</i>	22
3.6.3 <i>hall2cTx()</i>	23
3.6.4 <i>hall2cRx()</i>	24
3.7 Interrupts	24
3.7.1 <i>Triggering the ISR</i>	24
3.7.2 <i>ISR functional description</i>	25
CHAPTER 4	26
I2C DRIVER IMPLEMENTATION	26
4.1 Configuration Registers	27
4.1.1 <i>GPIO registers</i>	27
4.1.2 <i>I2C Master config registers</i>	28
4.1.3 <i>I2C Slave config registers</i>	30
4.1.4 <i>I2C FIFO config registers</i>	30
4.1.5 <i>Block Diagram for I2C module</i>	32
4.2 Configuration routine	32
4.2.1 <i>Functional Description of configuration function</i>	32
4.2.2 <i>Required data structures in the main program</i>	33
4.2.3 <i>example code for master and slave configuration with descriptors</i>	34
4.3 Send and Receive data	34
4.3.1 <i>Functional description of TX and RX functions</i>	34
4.3.2 <i>Required data structures in the main program (for TX and RX only)</i>	35
4.3.2 <i>Example code for master send and slave receive with descriptors</i>	35
4.4 Main program to configure and send data	36
4.4.1 <i>Steps to send a data transmission</i>	36
4.4.2 <i>Example of a complete main program code</i>	37
CHAPTER 5	40
TESTING	40
5.1 Test Case 1: 32 byte message.	42
5.2 Test Cast 2: 0 byte message.	43
5.3 Test Case 3: 1 byte message.	44
5.4 Test Case 4: 16 byte message.	45
5.5 Test Case 5: 35 byte message.	46
5.6 Test Case 6: Second Transmission	48
CHAPTER 6	50
CONCLUSION	50

6.1 Conclusion	51
6.2 Design goals met or unmet	51
6.3 Recommendations for expanding the driver	51
<i>ADDENDUM</i>	53
<i>References</i>	54
<i>Declaration</i>	55

List of Figures

Figure 1: Example of an I2C bus 2 wire configuration with 1 master and 2 slaves	6
Figure 2: Simplified steps for data transmission	8
Figure 3: Arbitration procedure with 2 masters	9
Figure 4: START and STOP conditions	10
Figure 5: 7-bit slave address plus read/write bit	10
Figure 6: I2CMSA (master slave address) calculation process	11
Figure 7: Example NACK waveform	12
Figure 8: Complete data transfer packet	12
Figure 9: Data write with 8-bit register address and data byte	13
Figure 10: Example of a data read	13
Figure 11: Example of a data read with register pointer	14
Figure 12: Application design diagram	16
Figure 13: Complete driver architecture diagram	17
Figure 14: Circular buffer logical implementation diagram	21
Figure 15: I2C module block diagram	32
Figure 16: Basic steps for data transmission in the main program	36
Figure 17: Input data for test case 1	42
Figure 18: Output for test case 1	42
Figure 19: Input data for test case 2	43
Figure 20: Output for test case 2	43
Figure 21: Input data for test case 3	44
Figure 22: Output for test case 3	44
Figure 23: Input data for test case 4	45
Figure 24: Output for test case 4	45
Figure 25: Input data for test case 5	46
Figure 26: Output data for test case 5 with 3 lost bytes	46
Figure 27: Output data for test case 5 with no data loss	47
Figure 28: Input data for test case 6	48
Figure 29: Output for test case 6 with data overwrite	48
Figure 30: Output for test case 6 with all bytes stored	49

List of equations

<i>Equation 1: $R_{p,min}$</i>	<u>7</u>
<i>Equation 2: $R_{p,max}$</i>	<u>7</u>
<i>Equation 3: SCL period</i>	<u>28</u>

Acronyms and abbreviations

SDA - Serial Data Line

SCL - Serial Clock Line

IC - Integrated Circuit

LSB – Least Significant Bit

MSB – Most Significant Bit

ACK – Acknowledged

NACK – Not Acknowledged

TX – Transmit

RX – Receive

ISR - Interrupt Service Routine

FIFO - First In First Out hardware data buffer

Glossary

Transmitter - the device that sends data to the bus

Receiver - the device that receives data from the bus

Master - the device that initiates/terminates a transfer

Slave - the device addressed by the master

Multi-Master - more than one master can attempt to control the bus at the same time

Arbitration - procedure to ensure that only one master device is controlling the bus at a time

Synchronization - procedure to synchronize the clock signals of 2 devices

Bus Idle - the bus is idle when both the SDA and SCL lines are high

START Bit - defined as SDA driven low while SCL is high

STOP Bit - defined as SDA pulled high while SCL is high

Interrupt – hardware triggered software function

Circular Buffer - a data structure that uses a single, fixed-sized buffer “connected” end to end

CHAPTER 1

INTRODUCTION

1.1 Scope of thesis

The purpose of this thesis is to learn about hardware driver design using the I2C bus as the basis. To expand this knowledge, I have designed an I2C driver using embedded C programming that can be used to send and receive messages over the I2C bus of the Texas Instruments TM4C1294XL microcontroller. This is an ARM Cortex M4 microcontroller. For the purposes of this thesis, I will send data from the I2C master to a single I2C slave. The I2C master is an I2C module (I2C0) on the TM4C that has been configured for master operation. The I2C slave is another I2C module (I2C3) on the TM4C microcontroller that has been configured for slave operation. It is also possible to send messages from the slave to the master using this driver, however, for testing purposes this functionality has been ignored.

This thesis contains 6 chapters plus an addendum. Chapter 1 is a basic overview of the thesis. Chapter 2 is an in-depth discussion of the I2C bus and how it works. In Chapter 3, I will provide a theoretical overview of the driver design. After the theoretical discussion, chapter 4 will get into the nuts and bolts of the I2C configuration. I will discuss each configuration register in detail. Chapter 4 also contains code for a full program that can be used to demonstrate the basic functionality of the driver. Chapter 5 will cover several testing scenarios with a brief discussion regarding the output for each test. Chapter 6 will provide a concluding discussion and some ideas for expanding the driver. The addendum will discuss a few additional functionalities included in the driver that may be of use for future projects.

This thesis should serve as a “how-to” guide for any future students who need to use this driver. I will fully discuss how to use the driver, and how to expand it with further functionality in the future. There are also some hidden functionalities that may be of use in certain scenarios, and those will be briefly covered as well.

1.2 Functional Requirements

-the driver shall facilitate every configuration requirement to configure the needed I2C module:

- GPIO
- master or slave
- sender or receiver
- configure the hardware FIFO
- configure the ISR using the provided NVIC driver

-the internal FIFO's of the TM4C shall be utilized as the data transfer medium.

-the Circular Buffer provided by the university will be utilized as the buffer between the user data and the FIFO (i.e. the user will never directly access the hardware FIFO).

-an interrupt will Trigger when the receiving FIFO has received data. The appropriate ISR will transfer this data into the circular buffer, making room for the next data transfer into the FIFO.

-the driver shall contain functionality to send and receive data from a single byte, up to an arbitrary size limited only by the size of the circular buffer. Caveat: the FIFO is limited to 8 bytes per transfer. This must be managed using the circular buffer and programming logic.

1.3 Software Requirements

-the driver shall be written in the C programming language.

-the code shall be re-usable and scalable (multiple identical hardware modules must be serviced using the same code).

-fully commented, clear coding style shall be used. Comments must include a brief description of the function as well as an explanation for each argument and return value.

-a layer of abstraction must exist between the user and the registers of the TM4C (i.e. the user will not have any access directly to the data or configuration registers).

-the driver shall contain all required definitions and enumerations (ex: base address of I2C module and offsets for individual registers, commands for the master controller, etc.).

-once received by the I2C slave, data should be accessible by the user so it can be processed or sent to another peripheral device.

-the user shall only have access to a limited function block allowing full configuration and data transmission without the ability to access dangerous functions which could break the functionality of the device.

-an I2C module should be able to be configured with a single, re-usable function call.

-data transmission is accomplished with a single, re-usable function call.

-the driver should be designed with ease of use as a major design goal.

1.4 required hardware

-Texas Instruments TM4C1294XL Microcontroller

-RS232 USB serial communications cable

-Basic Breadboard

-4.7kOhm pull-up resistors

-Connector cables

1.5 software utilized

- Texas Instruments Code Composer Studio (Eclipse based IDE)
- Notepad++
- REALTerm

CHAPTER 2

I2C IN DETAIL

2.1 Introduction to I2C

The I2C bus is a bi-directional 2-wire communications bus that requires only an SDA and SCL line for robust and efficient communication. In layman's terms, the I2C bus requires very few parts and facilitates easy communication between I2C compatible IC's. Phillips Semiconductors developed the I2C bus with the purpose of efficient inter-IC control. The I2C bus is now a standard feature in thousands of IC's produced by dozens of companies.

The purpose of the I2C bus is to maximize hardware efficiency and circuit simplicity for system designers and manufacturing companies. All IC'S that include an on chip I2C interface can easily communicate with each other using the I2C bus. A standardized protocol for addressing, data transmission, and line arbitration makes this somewhat of a plug and play solution. The need to design bus interfaces is essentially eliminated, speeding up development time and saving on engineering costs (NXP Semiconductors, 1982, p. 5).

2.2 Hardware Design

The I2C bus consists of 2 wires, the SDA line and the SCL line. Each line must be pulled up with a resistor to a true "high" level. This is accomplished by using a pullup resistor connected to a Vcc of 3.3V or 5V. Each device on the I2C bus can be directly connected to the bus by its own SDA and SCL device pins.

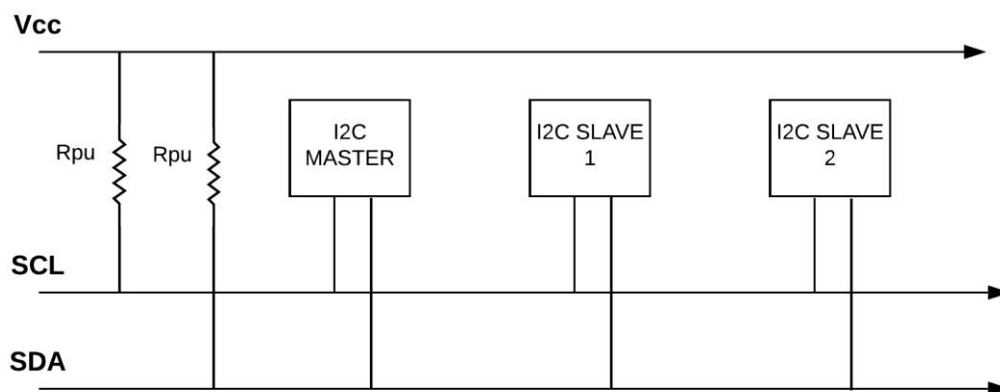


Figure 1: Example of an I2C bus 2 wire configuration with 1 master and 2 slaves (based on NXP Semiconductors, 1982, pg. 8)

The bus is IDLE when the SDA and SCL lines are high. The pullup resistors (R_{pu}) are what allows the lines to be driven high. I2C is an open drain system, so it can only pull down. This means that a transistor pulls the lines low when a device is ACTIVE, and when the device is off the line will be high and IDLE. In this case, the line would be pulled high by the pullup resistors connected to the Vcc. The value of the pullup resistors is an important choice and can be determined by the following equations:

Equation 1: $R_{p,min}$

$$R_{p, \min} = \frac{V_{cc} - Vol(\max)}{I_{ol}} \quad (\text{Arora, 2015, p. 2})$$

Vol is the voltage that will be read as a Valid Logic Low and I_{ol} is its corresponding current draw.

Equation 2: $R_{p,max}$

$$R_{p, \max} = \frac{tr}{0.8473 \times C_p} \quad (\text{Arora, 2015, p. 2})$$

tr is the standard I2C rise time and the C_p is the line capacitance. These values can be found on the datasheet for each IC or by taking measurements on a scope.

In this project, to calculate the $R_{p, \min}$ I looked at the data sheet to find the values for V_{cc} , Vol , and I_{ol} .

$$R_{p, \min} = \frac{3.3V - .4V}{2mA} = \mathbf{1.45k\Omega}$$

To calculate the $R_{p, \max}$ I had to get values from the datasheet for capacitance and from the Phillips/NXP spec sheet for SDA and SCL rise time. The calculation is as follows:

$$R_{p, \max} = \frac{300ns}{0.8473 \times 50pF} = \mathbf{7.08k\Omega}$$

For the pullup resistors I have selected 4.7k Ω . This falls within the min/max range and the functionality is verified by the output results. Unfortunately, due to the COVID-19 lockdown, I was unable to travel to Hamburg to use the oscilloscope in the lab for proper verification of these values.

2.3 Master Slave relationship

The master-slave relationship is what establishes the hierarchy on the bus. Every device on the bus has a unique address. Any device addressed by the master is considered a

slave. The master initiates all data transfers on the bus and provides the clock signal that the slave device must synchronize with (NXP Semiconductors, 1982, p. 6). The ONLY device on the bus that will receive the data is the one with the unique slave address referenced by the master. It is not possible to send a message to all slaves at the same time. This also does not mean that the master can ONLY send data. On the contrary, the master can also receive data from the slave device.

A data transfer sequence looks like this:

Master sending data to slave

1	Microcontroller(master) addresses slave with START condition and provides a clock signal
2	Master sends data to slave
3	Microcontroller terminates transmission with a STOP condition

Slave sending data to master

1	Microcontroller(master) addresses slave with START condition and provides a clock signal
2	Master receives data from slave
3	Microcontroller terminates transmission with a STOP condition

Figure 2: Simplified steps for data transmission (based on Valdez, 2015, pg. 3)

The key point is that the master ALWAYS initiates the transfer whether it is sending or receiving. The master must also provide a clock signal for every transmission. This is an integral part of any I2C driver and there are dedicated registers that must be written to enable the required clock speed.

The I2C protocol also allows for multiple masters to exist on the same bus. Therefore, it is possible that 2 masters could try and send data at the same time. For example, imagine the digital LCD display on a car. It can display information from multiple systems in the car: perhaps the outside temperature, the speed, and the current mileage on the car are all displayed. These 3 systems are controlled by separate Microcontrollers which are all Masters in their smaller, buffered section of the I2C bus. However, they must all 3 report data to the LCD screen, which is only a slave device. There is an arbitration process that will allow them to co-exist peacefully.

2.4 Arbitration

Arbitration is required if there is more than one master in the system. The I2C protocol only allows a master to initiate a transfer if the bus is clear. When a master initiates a transfer with a START condition there is a small delay before the I2C line is “busy” for this transfer. This time is referred to as the minimum hold time ($t_{hd;STA}$). During this short time period, it is possible for another master to initiate a START (Texas Instruments, 2013, p. 1282). The arbitration process is designed to maintain data integrity and successfully complete both transfers when this happens.

The arbitration process proceeds bit by bit (NXP Semiconductors, 1982, p. 11). During each high phase of the SCL, each master checks the current value of the SDA line against the current bit in the data byte that it transmitted. The first master to send a high but find the SDA line to be low has lost the arbitration and turns off its SDA driver. The winning master can complete its transfer. No information will be lost. The master that lost the arbitration will restart its transfer when the bus is free.

The following figure shows an arbitration process by 2 masters. When DATA1 has a different value from the SDA, then master 1 shuts down its SDA line and waits for the bus to be free. This allows master 2 to complete its transmission.

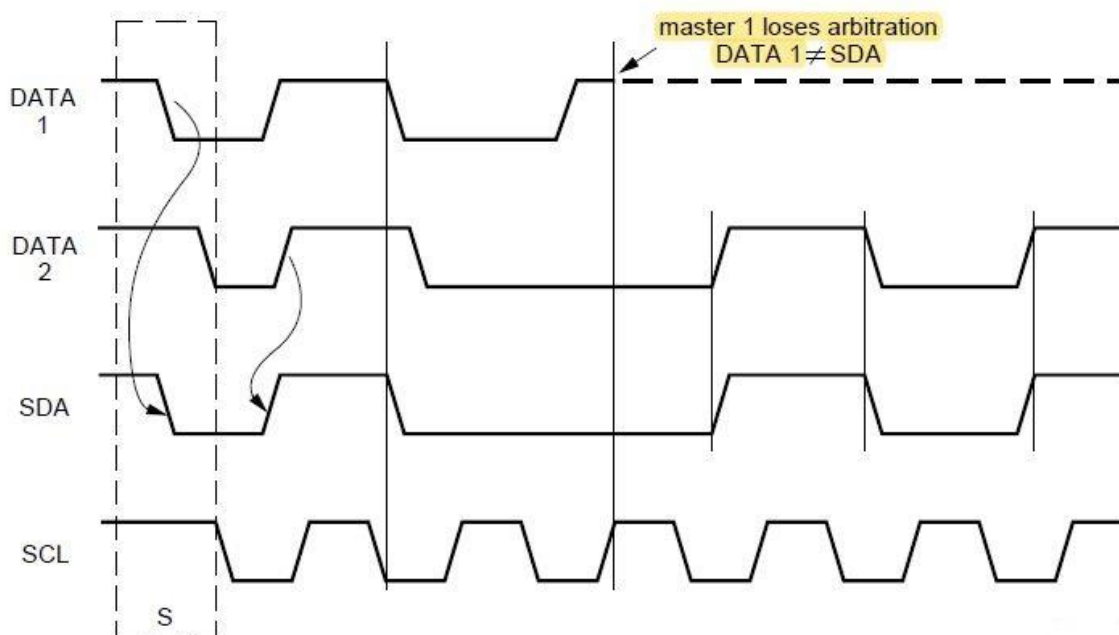


Figure 3: Arbitration procedure with 2 masters (NXP Semiconductors, 1982, pg. 12)

2.5 Start and Stop bit

Every I2C transaction is begun by the master sending a START bit and ended by the master sending a STOP bit (Valdez, 2015, p. 4). A START condition is when the SDA transitions from high to low while the SCL remains high. A STOP condition is when the SDA transitions from low to high while the SCL remains high. After a START condition the bus is considered busy. If another master tries to transmit while the bus is busy, then the arbitration process outlined in section 2.4 must take place.

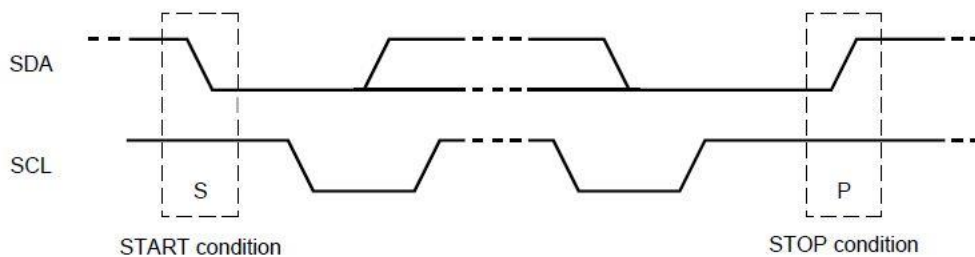


Figure 4: START and STOP conditions (NXP Semiconductors, 1982, pg. 9)

2.6 I2C Addresses (7-bit vs 8-bit)

Each device on the I2C bus has a unique address. When the master wants to transmit to or receive from the slave, it must address the specific slave that it wants to communicate with. These addresses are 8 bits in total. However, it is not as simple as just picking a random 8-bit value. The actual slave address is only 7 bits. The extra bit, or the LSB, serves as the read or write bit. A '0' in the LSB means that the master will write to the slave, and a '1' in the LSB means that the master will read from the slave.



Figure 5: 7-bit slave address plus read/write bit

For testing the driver, I have assigned the slave device with the hex address 0x76. This address is saved by the slave as its own address. However, when the master stores the slave address it will store the address appended with the read/write bit. The following code snippet from the I2C driver will explain further:

```
I2CMaster_MSA = (SLAVE_ADDRESS << 1) | RW;
```

The MSA, or Master Slave Address register, is where the slave address that the master will place on the I2C bus is stored. To get this result, the 8-bit slave address is bit shifted left by a single bit, then the read/write bit is appended to the LSB with a bitwise OR operator. The first 7 bits will be re-interpreted as the slave address while the LSB will determine if the master performs a read or a write.

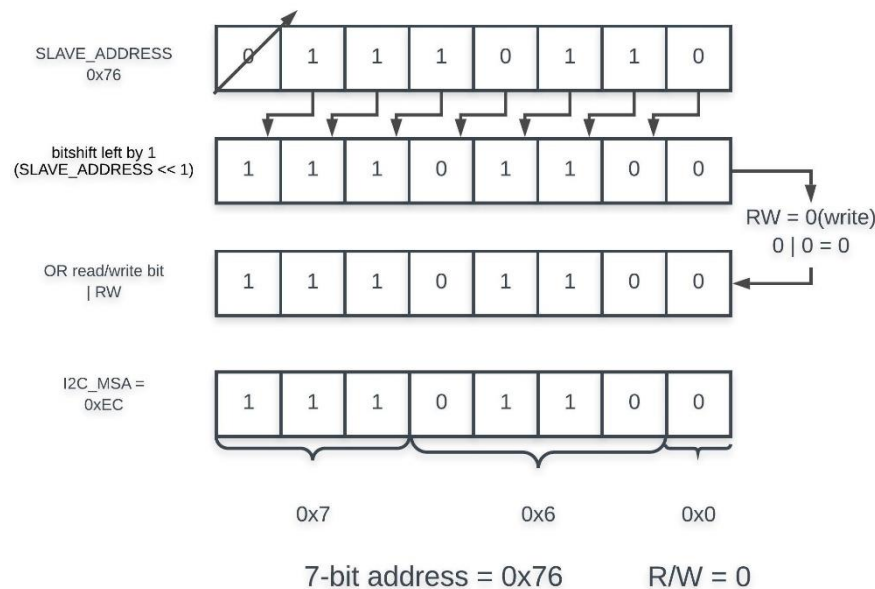


Figure 6: I2CMSA (master slave address) calculation process

For a write operation, the *I2C_MSA* register will store the hex value 0xEC. For a read operation, the method is the same except the RW bit is 1. The result is *I2C_MSA* = 0xED. The 7-bit slave address of 0x76 will remain the same in either case. The only change is the read or write bit.

2.7 Acknowledge or Not Acknowledged

After each byte transferred, there is an ACK or NACK bit to indicate that the byte was successfully received (ACK) or not (NACK). This includes the slave address byte as well. The transmitter has control of the SDA line. During the 9th clock cycle, the transmitter will release the SDA line so the addressed receiver can control it for this clock cycle (Valdez, 2015, p. 5). If the SDA is pulled low by the receiver this is considered an ACK by the transmitter. If the SDA remains high, then this is a NACK indication. In the event of a NACK the transmitter can issue a STOP bit to cancel the transmission, or a repeated START to try it again. An interrupt can be triggered when a NACK is received so that the user can configure a response that is appropriate for their project.

There are several conditions that can result in a NACK:

- 1) *There is no device on the bus with this slave address.*
- 2) *The receiver is busy with another function and cannot accept the transmission currently.*
- 3) *The receiver does not understand the data or commands.*

4) The receiver cannot accept any more bytes.

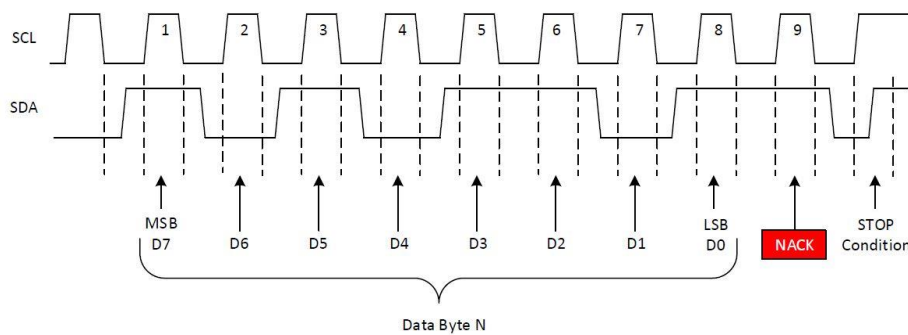


Figure 7: Example NACK waveform (Valdez, 2015, pg. 6)

In the above figure, during clock cycle 9 of the SCL the SDA remains high, which indicates a NACK condition. A stop condition is immediately sent by the master to stop the transfer.

2.8 Complete I2C Data Packet

Now that all the relevant parts have been defined, I will briefly discuss the data packet format for I2C transmission. Firstly, each byte transmitted must be exactly 8 bits long and it will always be followed by an ACK or NACK bit. An unlimited number of bytes can be transmitted per transfer, but the slave must ACK each byte. The transfer is ended by a stop bit. The slave can hold the SCL line low between bytes if it is not ready to receive more information. In this manner, the slave can force the master into a “wait” status. Once the slave has released the SCL then the transfer will continue.

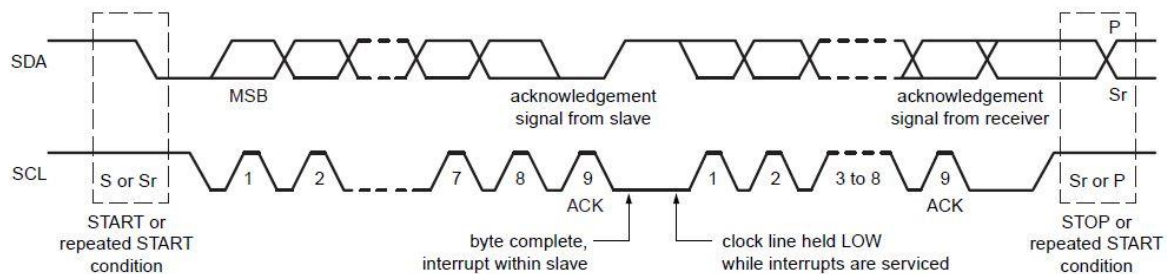


Figure 8: Complete data transfer packet (NXP Semiconductors, 1892, pg. 10)

2.9 Read and Write

Whether the master is writing to the slave or the slave is writing to the master, there is a similar process that facilitates the transfer. This is based upon the configuration of the master, which must be set with read or write bit (discussed in section 2.6). When there is data available to transmit or receive, a command is written to the master device’s control register. This command tells the master to initiate a transfer to or to read data from the slave. First, the master will write a START bit to the bus to make the bus busy for the

transfer. Then the 7-bit slave address is placed on the bus along with the read/write bit for a total of 8 bits. When the transfer is over, the master places a STOP bit on the bus and the bus is then released.

2.9.1 Write Operation

After the START bit, the master transmits the 7-bit slave address and the read/write bit, which is '0' for a write operation. In the 9th clock cycle, the slave will acknowledge. With a successful acknowledge, the master can send 8 more bits, and the slave must again acknowledge. This 2nd byte could be raw data or a register address. If the slave device is, for instance, an IC that controls charging of a battery, then the 2nd byte could be a register address and the 3rd byte could contain configuration data. This process continues until the master issues a STOP on the bus.

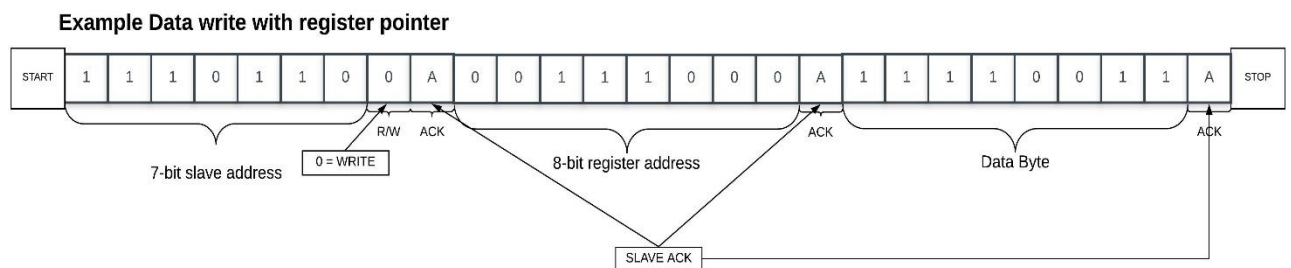


Figure 9: Data write with 8-bit register address and data byte (based on Valdez, 2015, pg. 7)

2.9.2 Read Operation

To initiate a read operation there is a similar process. After the START bit, the master transmits the 7-bit slave address and the read/write bit, which is '1' for a read operation. In the 9th clock cycle, the slave will acknowledge. Once the slave has acknowledged, the slave then transfers 8 bits of data to the master. At this point, roles are reversed so the master must then acknowledge to the slave that the transfer was successful.

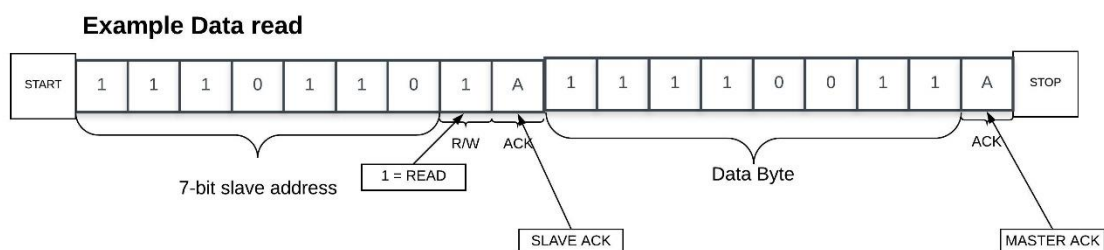


Figure 10: Example of a data read (based on Valdez, 2015, pg. 7)

There is a second type of read operation that is possible when there is a specific register in memory that the data should be stored in. For this case, the master initiates a START as if it's going to write data (i.e. read/write bit set to 0). It writes the register address

to the bus, then instead of a STOP condition the master sends a repeated START with the read/write bit set to 1 for a read operation.

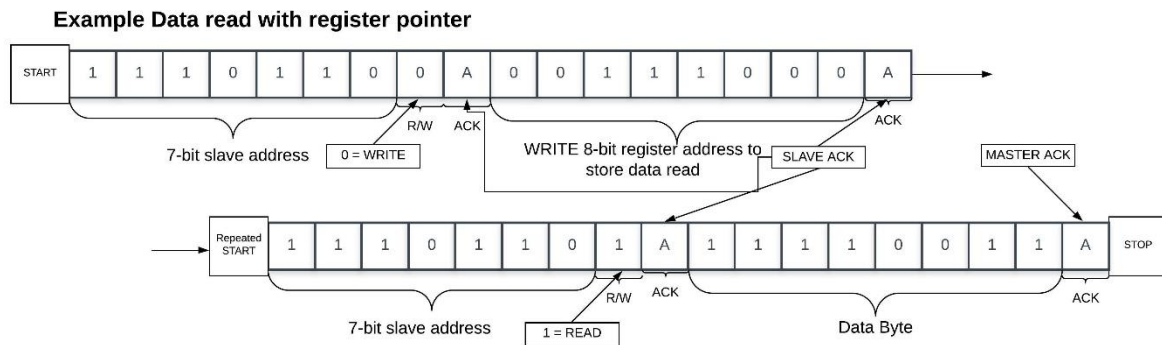


Figure 11: Example of a data read with register pointer (based on Valdez, 2015, pg. 7)

2.10 FIFO usage description

The hardware FIFO is a memory buffer that is dedicated to I2C communication. On the TM4C, the FIFO can hold 8 bytes of data at any given time. TX interrupts can be triggered when the buffer is full or empty, but not at any point in between. There is a “trigger value” that can be set, but this functionality is not available on the TM4C when filling the TX FIFO. That is not normal, but it is a limitation of the TM4C. The RX FIFO can react to the trigger value with no problems. So, it is possible to run the ISR after any number of bytes has filled the RX FIFO, from 1 to 8.

FIFO simply means data that is put into the buffer first, is also the data that is read from the buffer first. The FIFOs can be assigned to the master or the slave. The boundary conditions that have been used in testing the driver are that the TX (transfer) FIFO is assigned to the master, and the RX (receive) FIFO is assigned to the slave. These can be re-assigned and re-configured for different operating modes including the slave sending data to the master. In many cases, both FIFO’s would be assigned to a single I2C device since the device being communicated with is not on the same Microcontroller. For driver development, the only device available for testing was the TM4C. Therefore, the FIFO had to be shared amongst two I2C modules on the same device.

CHAPTER 3

I2C DRIVER DESIGN

3.1 HAL (Hardware Abstraction Layer)

This driver was designed with a layer of abstraction in mind. This is formally known as a HAL, or Hardware Abstraction Layer, driver. This means that the calling program can interact with the hardware device in a very general, abstract way. The driver acts as an interface between the program and the hardware that the program needs to interact with. The application layer is where the programs run, and logically sits above the driver. Therefore, hardware drivers are referred to as “HAL low-level drivers.” The HAL is the bridge that allows the applications to send and receive from the hardware.

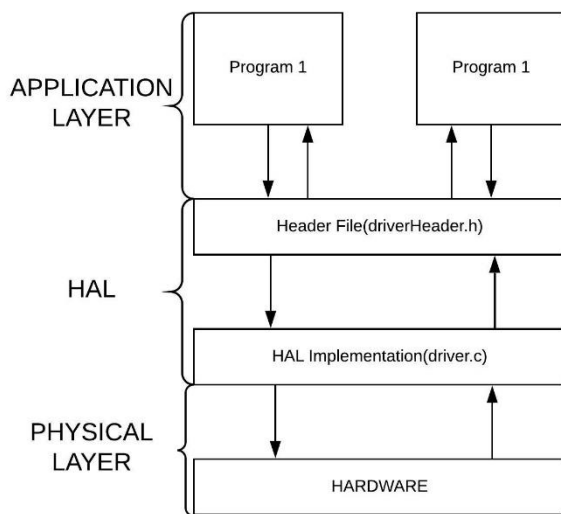


Figure 12: Application design diagram

There are several reasons for this abstraction. First, it protects against misuse. The data registers that are being accessed by the driver are hidden from the calling program. Only very specific functions can be called by the program. Interacting directly with the hardware is only possible for the driver. This allows for full functionality with zero direct register access, protecting against unwanted values in configuration or memory registers.

The second reason is for ease of use. To interact with something like an I2C or UART device is complicated. There are a lot of registers to configure for reliable data transfer. Without this kind of abstraction, a programmer would have to understand how each hardware device communicates with the rest of the system. To spend hours with a data sheet figuring out which registers need to be written for a certain configuration takes a lot of time. Abstraction allows this process to be simplified. In fact, the programmer can use the driver as an interface without really having any knowledge of how it works.

A third reason for HAL drivers is compatibility. The code for a specific program functionality is re-usable. For instance, imagine 2 different ARM M4 Microcontrollers that want to communicate over an I2C bus. The same application and driver code can be used on both controllers. In this way, the same few function calls can travel across multiple

Microcontrollers. It is also scalable, meaning that multiple I2C modules can be configured with the same re-usable function call.

3.2 Assumptions

For development purposes the following assumptions were made:

1. Only 2 I2C devices will be used, I2C0 and I2C3.
2. I2C0 will be configured as the master and I2C3 will be configured as the slave.
3. Data will only travel in a single direction, from master to slave.

3.3 Driver Architecture

The driver is organized with a total of 4 files:

1. I2cCores.h
2. I2cPinouts.h
3. hall2c.h
4. I2c.c

3.3.1 Organizational Structure

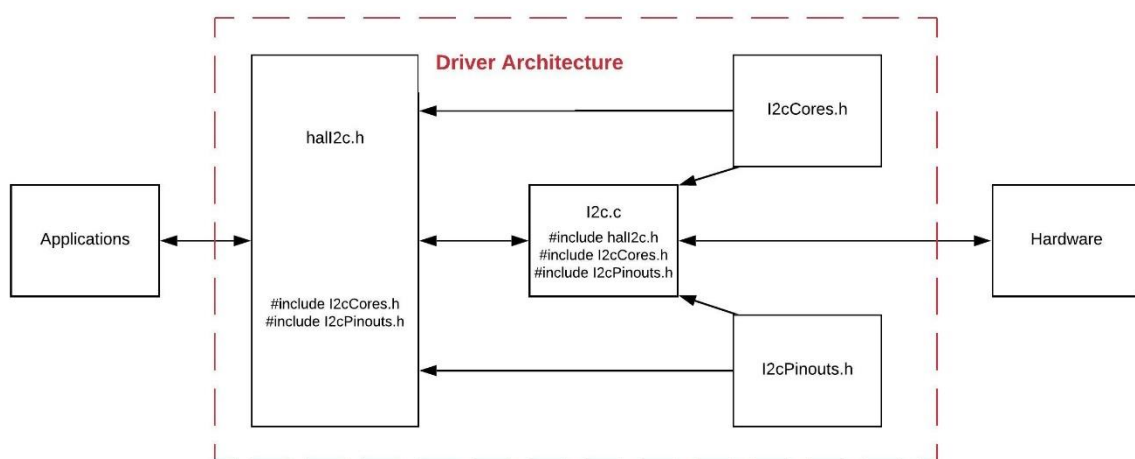


Figure 13: Complete driver architecture diagram

3.3.2 I2cCores.h

This header file provides an enumeration to make it easier to address a specific I2C device onboard the TM4C. There are 9 I2C devices on the Microcontroller. Here, 8 have been enumerated and are accessible to the driver:

```
enum i2cCores {
    i2c0Core = 0,
    i2c1Core = 1,
    i2c2Core = 2,
    i2c3Core = 3,
    i2c4Core = 4,
    i2c5Core = 5,
    i2c6Core = 6,
    i2c7Core = 7
};

enum i2cCoreCount {
    i2cCoreCount = 8
};
```

3.3.3 I2cPinouts.h

This header file establishes a struct for each I2C module on the TM4C to describe their pinouts. This struct will be used by the driver in I2C.c to configure each I2C module. The SCL and SDA pins are explicitly defined. One “entry” is needed for each desired core. As per the assumptions mentioned earlier, only 2 cores are currently defined.

```
enum i2cPinoutCount_ {
    i2cPinoutCount_ = 2 //the number of I2C cores currently defined
};

static const struct i2cPinoutSetup_ i2cPinoutSetups_[i2cPinoutCount_] =
{
    //hali2c0sclPB2scaPB3 [0]
    {
        .i2cCore_ = i2c0Core,
        .txPresent_ = true,
        .sclPin_ = halGpioI2c0SCLPB2,
        .rxPresent_ = false,
        .sdaPin_ = halGpioI2c0SDAPB3
    },
    //hali2c3sclPK4scaPK5 [1]
    {
        .i2cCore_ = i2c3Core,
        .txPresent_ = false,
        .sclPin_ = halGpioI2c3SCLPK4,
        .rxPresent_ = true,
        .sdaPin_ = halGpioI2c4SDAPK5
    }
};
```

3.3.4 hall2c.h

This is the header file that interfaces the application with the driver. All functions that are available in the driver are prototyped here including the ISR's. Most functions are hidden from the main program/application by declaring them as static. There are only 4 functions required by the main program to provide fully functional I2C communication. The functions are organized and grouped in a logical way with user facing functions listed first, then configuration functions, the communication functions, then the ISR's.

Also included in the header file are important definitions and enumerations that are used throughout the driver. These definitions are used in the driver for more efficient programming. For instance, a command of 0x07 must be written to the Master Control Register to command the master to send a single byte to the slave. To make this more logical to the programmer, the Master Control commands are defined in the header file as follows:

```
#define I2C_MASTER_CMD_SINGLE_SEND 0x00000007
//master will send a single byte
```

These definitions save time because the programmer will only have to read the definitions and choose the command that he or she needs instead of pouring over a data sheet to figure out the correct command sequence. That work has already been done and included here. Similarly, many other helpful enumerations and definitions are contained in this header file. Other examples include enumerations to state whether an I2C core is the slave or the master, and whether that I2c Core will read or write. It is recommended for anyone using this driver in the future to read the header file completely.

```
/**
 * 1 for Master. 0 for Slave. to be used in function calls.
 */

enum i2cMasterSlave_ {
    master = 1,
    slave = 0
};

/**
 * 1 for read 0 for write. to be used in function calls.
 */
enum i2cReadWrite_ {
    read = 1,
    write = 0
};
```

The other important need that this header fills is to provide a “handle” back to the calling program. This is accomplished with a constant pointer to a struct, which is defined as follows in the header:

```
typedef struct HalI2cCoreState * const HalI2cCoreStateHandle;
```

The typedef struct creates a new name for an already existing element. It does not create a new “type” of *HalI2cCoreStateHandle*. The *HalI2cCoreStateHandle* is what will be used by the main program to interact with the I2C module. When the I2C configuration function has completed it returns the *HalI2cCoreStateHandle* to the calling program. This handle points to a struct of type *HalI2cCoreState*, which holds configuration data about the I2C module. The struct is defined in the main driver I2c.c. This handle will be needed to write or read data using the provided user facing functions.

3.3.5 I2c.c

The file I2c.c is the actual driver. This file contains all functions that are required for configuration and data transmission over I2C. This is where all communication with the hardware takes place. The driver will often reference the pinouts, i2cCore enumerations, and definitions from the header files. Therefore, each of these files is incorporated in the “#include” section of the driver. The driver also assigns the ISR to the NVIC and initiates the circular buffer for each I2C Core. The ISR is fully contained inside this file as well. The struct mentioned in section 3.3.4, *HalI2cCoreState*, is also instantiated here:

```
// struct definition to hold dynamic core properties used to configure each
utilized I2C core
struct HalI2cCoreState {
    enum i2cCores i2cCore_;
    bool i2cCoreInitalized_;
    struct UtilsCircularBufferState * circularBufferRxHandle_;
    struct UtilsCircularBufferState * circularBufferTxHandle_;
    bool forceTx_;
    bool forceRx_;
};
```

3.4 Circular Buffer Interface

A circular buffer is a single, fixed sized buffer that is connected end to end. The code for the circular buffer was provided by the university so that it could be incorporated into the I2C driver. It is simply a buffer that exists between the main program and the FIFO of the I2C core. The program will not place data directly in the I2C FIFO's, but instead data will go into the circular buffer first before transferring into the FIFO.

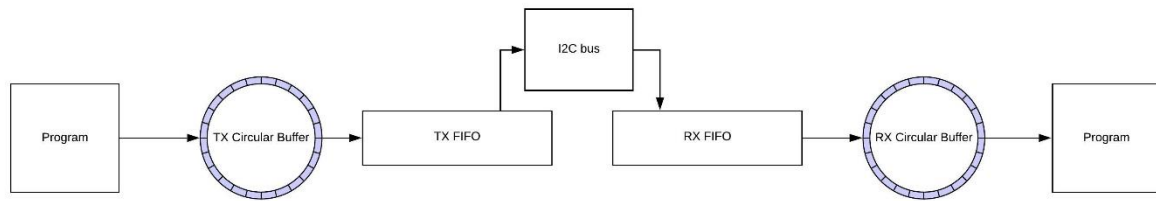


Figure 14: Circular buffer logical implementation diagram

To initialize the buffer, in the main program there must exist several char arrays that are sent to the configuration routine. These arrays will be used to allocate the memory space for the buffer. The circular buffer will be initialized and attached to the given I2C core during the I2C configuration routine with the following code:

```

    /*
    * Initialization of Rx and Tx Circular Buffer for this I2ccore
    * i2cMemoryAreaRx and i2cMemoryAreaTx are char arrays
    *passed from main
    */
    struct UtilsCircularBufferState * circularBufferRxHandle =
utilsCircularBufferInit(i2cMemoryAreaRx, i2cMemoryAreaRxSize);
    struct UtilsCircularBufferState * circularBufferTxHandle =
utilsCircularBufferInit(i2cMemoryAreaTx, i2cMemoryAreaTxSize);

    i2cCoreStates_[i2cCore].circularBufferRxHandle_ =
circularBufferRxHandle;
    i2cCoreStates_[i2cCore].circularBufferTxHandle_ =
circularBufferTxHandle;

```

3.5 NVIC integration

The NVIC, or Nested Vector Interrupt Control, is a method of prioritizing and dealing with interrupts. The NVIC allows a certain ISR to be assigned to a given peripheral device or pin on the microcontroller. To assign interrupts in the driver, the university provided an NVIC interface that needed to be expanded to include I2C. To accomplish this, I had to extend the enumerations in the NVIC to include the I2C exceptions found in the data sheet:

```

enum HalNvicException {
    //UART Interrupts
    halNvicUart0Exception = 21,    // UART0 interrupt INT_UART0
    halNvicUart1Exception = 22,    // UART1 interrupt INT_UART1
    halNvicUart2Exception = 49,    // UART2 interrupt INT_UART2
    halNvicUart3Exception = 72,    // UART3 interrupt INT_UART3
    halNvicUart4Exception = 73,    // UART4 interrupt INT_UART4
    halNvicUart5Exception = 74,    // UART5 interrupt INT_UART5
    halNvicUart6Exception = 75,    // UART6 interrupt INT_UART6
    halNvicUart7Exception = 76,    // UART7 interrupt INT_UART7
    //I2C Interrupts
    halNvicI2C0Exception = 24,     // I2C0 interrupt INT_I2C0
    halNvicI2C1Exception = 53,     // I2C1 interrupt INT_I2C1
    halNvicI2C2Exception = 77,     // I2C2 interrupt INT_I2C2
    halNvicI2C3Exception = 78,     // I2C3 interrupt INT_I2C3

```

```

    halNvicI2C4Exception = 86,    // I2C4 interrupt INT_I2C4
    halNvicI2C5Exception = 87,    // I2C5 interrupt INT_I2C5
    halNvicI2C6Exception = 118,   // I2C6 interrupt INT_I2C6
    halNvicI2C7Exception = 119    // I2C7 interrupt INT_I2C7
};

```

To point the NVIC interrupt response to the correct ISR requires only a few lines of code. First, a struct must be created to assign the exception for the I2C module with the correct ISR function. The following code snippet defines the ISR for I2C0:

```

static const struct i2cCoreSetup_ i2cCoreSetup_[i2cCoreCount] =
{
    // I2C0
    {
        .interruptId_ = halNvicI2C0Exception,    / Exception number
        .isr_ = i2c0ISR_                        // pointer to corresponding ISR method
    },
};

```

During the I2C configuration routine, the following function calls will register and activate the ISR using the NVIC driver:

```

halNvicInstallISR(i2cCoreSetup_[i2cCore].isr_,
i2cCoreSetup_[i2cCore].interruptId_);
halNvicEnableInterrupt(i2cCoreSetup_[i2cCore].interruptId_);

```

From now on, any condition from I2C0 that triggers an interrupt will call the function i2c0ISR_. Which events can trigger an interrupt is fully configurable.

3.6 User Accessible Functions

There are only 4 functions that are accessible to the programmer. This was by design to make the driver extremely user friendly.

3.6.1 hali2cDriverInit()

This function must be run first by the main program. It has no arguments or return values. It accomplishes several important functions. First, it sets the i2cCoreInitialized flag to false for each available core. This essentially turns off all I2C modules. The programmer can be sure that only I2C cores which are meant to be initialized will be active since they all begin with this flag set to false. This function also initializes the NVIC driver and the Circular Buffer with specific function calls. After calling this function the foundation is laid so that I2C modules can be initialized.

3.6.2 hali2cCoreInit()

This is the function that will initialize and configure an I2C module on the TM4C. It returns a handle, or a pointer to a struct, that can be used to interact with the I2C driver during data transmission. Master or slave? Sender or receiver? The same function is used to

configure the I2C module no matter what parameters are chosen. The arguments will determine the behavior of the I2C module:

enum Hali2cPinout i2cPinout - this is the pinout for this I2cCore. enumerated in hall2c.h header file.

enum i2cMasterSlave_ MS - 1 for master 0 for slave. Enumerated in hall2c.h header file.

enum i2cReadWrite_ RW - 1 for read, 0 for write. Enumerated in hall2c.h header file.

uint8_t SLAVE_ADDRESS - the 8-bit address for the slave device on the I2C bus. Defined in the main program.

uint32_t SysClock - the system clock speed. It is required to configure the master I2cCore. Calculated in the main program.

char * const i2cMemoryAreaTx - a pointer to the transmit memory area of the circular buffer this core will use. Declared in the main program.

unsigned int const i2cMemoryAreaTxSize - size in Bytes of the circular buffer transmit memory area. Enumerated in the main program.

char * const i2cMemoryAreaRx - a pointer to the receive memory area of the circular buffer this core will use. Declared in the main program.

unsigned int const i2cMemoryAreaRxSize - size in Bytes of the circular buffer receive memory area. Enumerated in the main program.

Inside this function the GPIO, I2C module, FIFO, NVIC/Interrupts, and Circular Buffer will all be initialized for the chosen I2C module.

3.6.3 hall2cTx()

This function transmits data over the I2C bus. First, the data is placed in the TX circular buffer. From there, the data is sent to the TX FIFO. When the FIFO is full or when there is no more data to send, a command is sent to the master control register to send the contents of the FIFO to the receiver over the I2C bus. The return value is the number of bytes that were sent. After this function is complete the RX circular buffer will contain the bytes that were sent.

The arguments are:

char const * const txBuffer - array containing the bytes to be transmitted

unsigned int const txBufferLength - the length of TxBuffer

Hali2cCoreStateHandle I2cCoreStateHandle - (pointer) handle provided by the main program to interact with the I2cCore

3.6.4 hall2cRx()

This function moves the data bytes from the RX circular buffer into a char array that is accessible by the main program. From this array, what happens next depends on the application. The data could be used for calculations, or sorted, or just sent to another peripheral device. The return value is the number of Bytes that were read from the Circular Buffer.

The arguments are:

char const * const txBuffer - pointer to a character array containing the bytes to be transmitted

unsigned int const txBufferLength - the length of TxBuffer

HallI2cCoreStateHandle I2cCoreStateHandle - (pointer) handle provided by the main program to interact with the I2cCore

3.7 Interrupts

The ISR's are configured by using the NVIC driver. This process is detailed in section 3.5. The specific register configurations for the required behavior will be covered in Chapter 4.

3.7.1 Triggering the ISR

The main ISR is the function:

```
i2cISR_ (i2cCore);
```

It accepts a single I2C core as its argument. However, this function is not called directly from the NVIC. There is an intermediate step. Each I2C core has its own ISR, which in turn calls the main ISR function. I2C0, for example, has the following function call assigned in the NVIC when an interrupt is triggered:

```
static void i2c0ISR_(void) {  
    i2cISR_(i2c0Core);  
}
```

Each I2C core has a similar function. When called, the ISR for each I2C core only calls the main ISR function with its own core value as an argument. This multi-step process is required because the main ISR needs the I2C core as an argument for internal function calls.

There is no way to include this argument directly from the NVIC. This functionality allows re-use of the same ISR function by all I2C modules.

3.7.2 ISR functional description

The ISR has a few key tasks. First, it must determine if this is a master or slave module that has received an interrupt. Therefore, it calls the function

```
Master_Slave(i2cCore);
```

This function returns a 1 if the I2C core is a master, and a 0 if it is a slave.

From there the ISR needs to read the Masked Interrupt Status and service what interrupts are found there. Generally, if the I2C core is a slave, then the RX FIFO needs to be read, and the bytes found there placed in the circular buffer. This is accomplished with the function call:

```
readI2cRxHardwareFifo_(i2cCore);
```

If the I2c Core is a master, then the master can receive data by using the same function call. The functionality is also there for the master to send data when the TX FIFO reaches a certain fill level for triggering. This would work with the following function call:

```
writeI2cRxHardwareFifo_(i2cCore);
```

The ISR is already coded with this functionality. However, as covered in section 2.10 the TM4C does not currently have a working TX trigger, so this code remains untested.

The last task for the ISR is to clear the interrupts by writing into the master or slave Interrupt Clear register. Because of timing requirements, it is important that this is done **after** servicing the interrupts. If the interrupts are cleared too early in the function, then the interrupt may trigger again while the original ISR is still processing. The consequences could be data loss or data corruption.

CHAPTER 4

I2C DRIVER IMPLEMENTATION

4.1 Configuration Registers

This section will cover the specific registers than need to be configured to use the I2C bus.

4.1.1 GPIO registers

The following registers must be configured for the GPIO to function as an I2C bus:

Register 95: Inter-Integrated Circuit Run Mode Clock Gating Control (RCGCI2C)

This register activates an I2C module by providing it with a clock signal. If an I2C module is made active, then a clock signal is provided. Otherwise, there is no clock signal and the I2C module cannot be used.

Register 89: General-Purpose Input/Output Run Mode Clock Gating Control (RCGCGPIO)

This register activates a GPIO port by providing it with a clock signal. If an GPIO port is made active, then a clock signal is provided. Otherwise, there is no clock signal and the GPIO port cannot be used.

Register 10: GPIO Alternate Function Select (GPIOAFSEL)

This register is essentially a control selection register. If the corresponding bit is left at 0, then the pin is used as a standard GPIO pin. If the bit is 1 then the pin will be controlled by a peripheral device.

Register 18: GPIO Digital Enable (GPIODEN)

This register determines whether a pin will be used as an analog input pin or a digital signal is expected. If the bit is left at 0 it behaves as a standard analog I/O pin. If the bit is set at 1 then it will function as a digital I/O pin.

Register 14: GPIO Open Drain Select (GPIOODR)

This register is the open drain control register. Setting the bit to 1 enables the open drain functionality of the pin, which is needed for some peripheral devices like I2C. Only the SDA must be configured as an open drain for I2C.

Register 22: GPIO Port Control (GPIOPCTL)

When using the AFSEL register to set a pin to work with an alternate function, the PCTL register is where the specific peripheral which will be used is set. The value that needs to be written into the PCTL for each pin is different depending on which peripheral device is used. These values can be found in a table in the data sheet. This table for the TM4C is on page 1808. It shows that a value of 0x2 must be written for I2C operation with pins PB2 and PB3 being used for I2C0.

The following code example shows a manual configuration of the GPIO for the I2C0 module. However, it should be noted that this type of manual operation is not used in the I2C driver. I have integrated a GPIO driver that was provided by the university that allows the same functionality without writing directly to the registers.

```

case I2C0_BASE:
    SYSCTL_RCGCI2C_R |= 0x0001;           //activate I2C0
    SYSCTL_RCGCGPIO_R |= 0x0002;         //activate GPIO port B
    while((SYSCTL_PRGPIO_R&0x0002) == 0){}; // ready?
    GPIO_PORTB_AHB_AFSEL_R |= 0x0C;      // 3) enable alt funct on PB2,3
    GPIO_PORTB_AHB_ODR_R |= 0x08;        // 4) enable open drain on PB3
    GPIO_PORTB_AHB_DEN_R |= 0x0C;        // 5) enable digital I/O on PB2,3
    GPIO_PORTB_AHB_PCTL_R |= 0x00002200; // 6) configure PB2,3 as I2C

```

4.1.2 I2C Master config registers

Register 9: I2C Master Configuration (I2CMCR)

This register configures whether the I2C module runs in master or slave mode. If bit 4 is set, then the module runs as master. If bit 5 is set, the module runs as a slave.

Register 4: I2C Master Timer Period (I2CMTPR)

This register is programmed to set the period for the SCL clock. This value will determine the timing of the entire I2C network, since the master always sets the clock value for any transmission. The value of the MTPR needs to be calculated using the following formula found in the datasheet:

Equation 3: SCL period

$$SCL_{PRD} = 2 \times (1 + TPR) \times (SCL_{LP} + SCL_{HP}) \times CLK_{PRD} \text{ (Texas Instruments, 2014, pg. 1285)}$$

SCL_{PRD} is the SCL line period (I2C clock).

TPR is the Timer Period Register Value (range of 1 to 127).

SCL_{LP} is the SCL Low period (fixed at 6).

SCL_{HP} is the SCL High Period (fixed at 4).

CLK_{PRD} is the system clock period in ns.

After doing a bit of algebra, this calculation can be written into the driver with the following code:


```
TPR = ((SysClock + (2 * 10 * SCLFreqLow) - 1) /
      (2 * 10 * SCLFreqLow)) - 1;
I2C_MTPR = TPR;
```

SysClock is the return value of the function *SysCtlClockFreqSet()*, which configures the clock frequency for the entire microcontroller. This function must be run in the main program before configuring of I2C modules is possible. The *SysClock* value is then passed into the function from the main program. The *SCLFreq* can be either 100kbps or 400 kbps. This is a choice made by the programmer. For testing, I have chosen 100kbps (*SCLFreqLow*) since this is the standard transmission speed of the I2C bus. These values are enumerated in the driver as follows:

```
//define baud rate to be used when configuring the I2C modules
enum baud_rate {
    SCLFreqLow = 100000, //SCL baud rate at 100kbps(standard speed)
    SCLFreqHigh = 400000 //SCL baud rate at 400kbps
};
```

Register 1: I2C Master Slave Address (I2CMSA)

This register holds the slave address that the master will send on the I2C bus. Recall that the full address is 8 bits, with 7 of those being the slave address and the LSB holding the read/write bit.

Register 12: I2C Master Burst Length (I2CMBLEN)

This register determines how many Bytes are sent per data transmission when using the internal FIFO. Each FIFO can hold 8 Bytes, so I have chosen the burst length as 8 bytes. The values that can be used as MBLEN are defined in the *halI2c.h* header file.

Register 5: I2C Master Interrupt Mask (I2CMIMR)

This register controls whether or not a raw master interrupt will trigger an ISR in the NVIC. Each possible interrupt source always sets the corresponding bit in the Master Raw Interrupt Status Register(I2CMRIS). It is always possible to read these bits, but an interrupt will only trigger when the corresponding bit in the MIMR is set.

Each possible source of interrupts is defined in the *halI2c.h* header file as follows:

```
#define I2C_MASTER_INT_TX_FIFO_REQ
\          0x00000100 // TX FIFO Request Interrupt
```

Master interrupts are currently disabled since I am only sending data from the master to the slave. To enable a master interrupt, the following function call can be used:

```
Master_Interrupt_Enable(i2cCore, I2C_MASTER_INT_TX_FIFO_REQ);
```

Any combination of interrupts can be enabled by ORing the defined I2C master interrupts together in the function call.

4.1.3 I2C Slave config registers

Register 9: I2C Master Configuration (I2CMCR)

This register must also be configured for the slave I2C module. If bit 4 is set, then the module runs as master. If bit 5 is set, the module runs as a slave.

Register 15: I2C Slave Control/Status (I2CSCSR)

When read, this is a status register that can give pertinent information about that status of a transfer. However, when written to it is a control register that is required to put an I2C module into slave mode. It is also used to turn on or off the slave FIFO.

Register 21: I2C Slave Own Address (I2CSOAR)

This register stores the slave module's own address. The slave address is configurable and can be chosen by the programmer. For the testing of I2C3 as a slave module I have chosen 0x76 as the slave address. This address is defined in the main program as follows:

```
#define SLAVE_ADDRESS 0x76
```

If there is more than a single slave, it may be beneficial to enumerate multiple slave addresses.

Register 17: I2C Slave Interrupt Mask (I2CSIMR)

This register controls whether a raw slave interrupt will trigger an ISR in the NVIC. Each possible interrupt source always sets the corresponding bit in the Slave Raw Interrupt Status Register(I2CSRIS). Each possible source of interrupts is defined in the hall2c.h header file. Configuring the Interrupt Masks works in the same way as the explanation in section 4.1.2 with the exception that the proper function call is

```
Slave_Interrupt_Enable(i2cCore, I2C_SLAVE_INT_RX_FIFO_REQ);
```

For testing purposes, the only bits set in the SIMR are bit 8 and bit 6. Bit 8 triggers the ISR when the RX FIFO is full. Bit 6 triggers the ISR when the fill level reaches its trigger, which is set to 1 byte. This way, even if only a single byte is transferred the ISR will run and the byte is properly handled.

4.1.4 I2C FIFO config registers

The send (TX) FIFO and the receive (RX) FIFO must be configured respectively. In most cases a device (microcontroller, sensor, LED...e.tc) will be only a master or a slave, not both. Therefore, the RX and TX FIFO are normally assigned to the same I2C module. In this case, I only had access to a single device. So, I had to configure one I2C module to be the

master and another I2C module on the same device to be the slave. These devices then share the FIFO. The master gets the TX FIFO and the slave gets the RX FIFO.

To configure the FIFO, the following registers are required:

Register 24: I2C FIFO Control (I2CFIFOCTL)

This is the main control register for the FIFOs. RX and TX are assigned, the FIFO can be flushed, and the FIFO fill trigger level can be set. As mentioned before, the TX FIFO trigger fill level does not function on this version of the TM4C, so I was unable to test this functionality. However, it does work on the RX side. I have set it to react to a single byte to ensure that no data is lost even with single byte transmissions. Bit 31 controls whether the RX FIFO is assigned to the master or slave, and bit 15 controls the TX FIFO in the same way. Setting bit 14 flushes (empties) the FIFO.

Register 15: I2C Slave Control/Status (I2CSCSR)

For the slave only, bit 2 needs to be set to assign the RX FIFO to the slave.

4.1.5 Block Diagram for I2C module

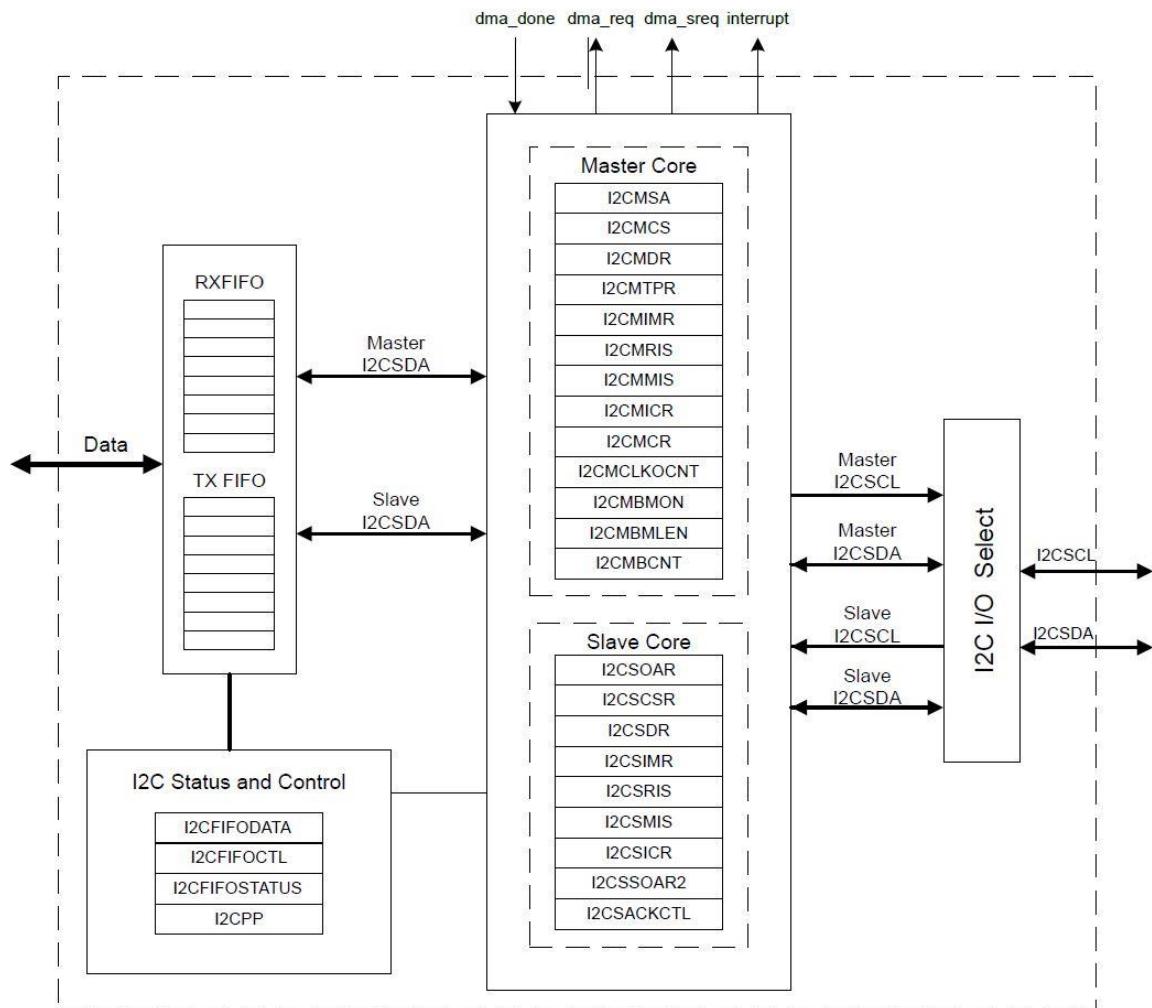


Figure 15: I2C module block diagram (Texas Instruments, 2014, pg. 1276)

4.2 Configuration routine

4.2.1 Functional Description of configuration function

The configuration function performs the following necessary actions:

1. Configure the GPIO pins for the I2C module using the GPIO driver.
2. Enable the I2C module in the SYSCTL register.
3.
 - 3a. If the module is to be a master:
 - 1.Enable the master.

2. Set the master timer period.
 3. Set the slave address and the read/write bit.
 4. Set the master burst length
- 3b. If the module is a slave:
1. Enable the slave.
 2. Set the slave address.
4. Enable and configure the FIFOs.
 5. Configure the interrupt masks.
 6. Clear all interrupts.
 7. Register the ISR using the NVIC driver.
 8. Initialize the circular buffer for this I2C module.
 9. Setup the handle for the I2C module.
 10. Return the handle to the main program.

4.2.2 Required data structures in the main program

The following data structures, enums, and defines are needed in the main program for the configuration of the I2C module and circular buffer:

```
#define SLAVE_ADDRESS 0x76 //Assigned slave address for the I2C Slave

//these are for the circular buffer
enum i2cAreaMemorySize_ {
    i2cMasterTxAreaMemorySize_ = 9,
    i2cMasterRxAreaMemorySize_ = 33,
    i2cSlaveTxAreaMemorySize_ = 9,
    i2cSlaveRxAreaMemorySize_ = 33
};

//Initialization parameters for the circular buffer
//the master and slave each need their own TX and RX buffer
static char i2cMasterMemoryAreaTx_[i2cMasterTxAreaMemorySize_];
static char i2cMasterMemoryAreaRx_[i2cMasterRxAreaMemorySize_];
static char i2cSlaveMemoryAreaTx_[i2cSlaveTxAreaMemorySize_];
static char i2cSlaveMemoryAreaRx_[i2cSlaveRxAreaMemorySize_];
```

4.2.3 example code for master and slave configuration with descriptors

To configure the master or the slave I2C module, only a single function call is required for each:

```
//configure i2c0 Master to write
HalI2cCoreStateHandle i2cMasterHandle =
hali2cCoreInit(hali2c0sclPB2sdaPB3,
               master,
               write,
               SLAVE_ADDRESS,
               SysClock,
               &i2cMasterMemoryAreaTx_[0],
               i2cMasterTxAreaMemorySize_,
               &i2cMasterMemoryAreaRx_[0],
               i2cMasterRxAreaMemorySize_);

//configure i2c3 slave to receive
HalI2cCoreStateHandle i2cSlaveHandle =
hali2cCoreInit(hali2c3sclPK4sdaPK5,
               slave,
               read,
               SLAVE_ADDRESS,
               SysClock,
               &i2cSlaveMemoryAreaTx_[0],
               i2cSlaveTxAreaMemorySize_,
               &i2cSlaveMemoryAreaRx_[0],
               i2cSlaveRxAreaMemorySize_);
```

The specifics of the function arguments are covered in section 3.6.2. The same function call is used to configure a master or a slave. It is only the arguments that change.

4.3 Send and Receive data

4.3.1 Functional description of TX and RX functions

To send data over the I2C bus, the function that must be called from the main is:

```
halI2cTx();
```

This function will accept a pointer to a char array that contains the bytes to be sent as an argument. These bytes are first placed inside the TX circular buffer, 8 bytes at a time. From the circular buffer, the bytes are put into the TX FIFO. When the TX FIFO is full, the command to transmit the bytes to the slave is sent to the master control register. The 8 bytes from the TX FIFO are then transmitted over the I2C bus and received in the slave RX FIFO. The function returns the number of bytes that were sent, which is used by the main program to update a counter that is needed to transfer larger messages.

To retrieve this data from the FIFO, an ISR is called which places the transferred bytes into the RX circular buffer. In order to maintain a level of abstraction, there exists a function that only serves to pull the data from the circular buffer into a data array in the main program:

```
halI2cRx();
```

This function accepts a pointer to a char array as an argument and simply uses the circular buffer driver's internal read function to place the available bytes into the given array.

4.3.2 Required data structures in the main program (for TX and RX only)

The following are required in the main program for sending and receiving data:

```
/**
 * Constant specifying the length of the message, i.e.
 * number of bytes to be received (Rx) or send (Tx).
 */
enum i2cMessageLength_ {
    i2cMessageLength_ = 32
};

//Length of the read buffer
enum I2cReadBufferSize_ {
    I2cReadBufferSize_ = 32
};

//Array to store returned values from the RX circular buffer
char i2cCircularBufferMessageRead_[I2cReadBufferSize];

//Message to be transmitted
static char const i2cMessage_[i2cMessageLength_ + 1] = "message";
```

4.3.2 Example code for master send and slave receive with descriptors

To send a message that is stored in the `i2cMessage_[]` array:

```
//TRANSMIT THE CONTENTS OF "i2cMessage_" OVER THE I2C BUS
//the message will end up in the RX circular buffer of the I2C
slave(I2c3)
sentBytesCount = 0;
while (sentBytesCount < i2cMessageLength_) {
    sentBytesCount += halI2cTx(
        &i2cMessage_[sentBytesCount],
        i2cMessageLength_ - sentBytesCount,
        i2cMasterHandle
    );
}
```

Since the FIFOs can only hold 8 bytes it is necessary to use some programming logic to send larger messages. The total message length is arbitrary, but we want to send only 8 bytes at a time. In this case the `i2cMessageLength` is 32 bytes. The variable `sentBytesCount` is

updated during each iteration with the total number of bytes sent thus far. The value of *sentBytesCount* is used to determine the current position in the *i2cMessage* array. A pointer to this memory location is sent as a function argument so the function knows where to begin reading from in the following iteration. This occurs until the number of bytes sent is equal to or greater than the *i2cMessageLength*.

To read the contents of the circular buffer after transmission is complete:

```

//READ THE MESSAGE BACK FROM THE I2C3 RX Circular Buffer into the
array "i2cCircularBufferMessageRead_"
//receivedBytesCount should be equal to sentBytesCount after
reading the message
    unsigned int receivedBytesCount = 0;
    receivedBytesCount =
halI2cRx(&i2cCircularBufferMessageRead_[receivedBytesCount],
        i2cMessageLength_,
        i2cSlaveHandle);

```

The data should already exist in the RX circular buffer for the slave. Depending on the needs of the program, it may be valuable to transfer the data into a standard char array so it can be processed in some way or sent to another peripheral. A pointer to a data array is sent as a function argument and the function places the entire contents of the circular buffer into that data array. Take care that the array is large enough to receive the entire circular buffer or some data may be lost.

4.4 Main program to configure and send data

4.4.1 Steps to send a data transmission

1	Create all required Data Structures, enums, and defines
2	Configure Master I2C Module
3	Configure Slave I2C Module
4	Send Data
5	Read Data

Figure 16: Basic steps for data transmission in the main program

4.4.2 Example of a complete main program code

```
/*
 * main.c
 *
 * The main program simply instantiates 2 I2C handles. One for the master
and one for the slave.
 * Then a message of Bytes is placed in the circular buffer, and then
transmitted
 * from the master to the slave over the I2C bus.
 * This data is then read from the RX I2C FIFO and placed into the RX
circular buffer, then stored in a buffer array
 * visible to the main program
 * Created on: June 28, 2020
 * Author: Jacob Seal
 */

#include <stdint.h>
#include <stdbool.h>

// HAL libraries
#include "hal/uart/halUart.h"
#include "hal/i2c/halI2c.h"
#include "C:/ti/TivaWare_C_Series-2.1.4.178/driverlib/sysctl.h"
#include "utils/circularBuffer/utilsCircularBuffer.h"

#define SLAVE_ADDRESS 0x76 //Assigned slave address for the I2CSlave

/**
 * Constant specifying the length of the message, i.e.
 * number of bytes to be received (Rx) or send (Tx).
 */
enum i2cMessageLength_ {
    i2cMessageLength_ = 32
};

//these are for the circular buffer
//memory size needs to be 1 bigger than transmission size. ex: 8 bytes
transmitted needs memory size of 9
enum i2cAreaMemorySize_ {
    i2cMasterTxAreaMemorySize_ = 9,
    i2cMasterRxAreaMemorySize_ = 33,
    i2cSlaveTxAreaMemorySize_ = 9,
    i2cSlaveRxAreaMemorySize_ = 33
};

enum I2cReadBufferSize_ {
    I2cReadBufferSize_ = 32
};

/**
 * Static variables will not be seen outside
 * of their compilation unit.
 */
//Message to be transmitted
static char const i2cMessage_[i2cMessageLength_ + 1] =
"TEST1I2CTEST2I2CTEST3I2CTEST4I2C";
```

```

//Initialization parameters for the circular buffer
static char i2cMasterMemoryAreaTx_[i2cMasterTxAreaMemorySize_];
static char i2cMasterMemoryAreaRx_[i2cMasterRxAreaMemorySize_];
static char i2cSlaveMemoryAreaTx_[i2cSlaveTxAreaMemorySize_];
static char i2cSlaveMemoryAreaRx_[i2cSlaveRxAreaMemorySize_];

int main(void) {
    char i2cCircularBufferMessageRead_[I2cReadBufferSize_];
    //set system clock for I2C - required to initialize the I2C master
module
    uint32_t SysClock = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
SYSCTL_OSC_MAIN |
                                                    SYSCTL_USE_PLL |
SYSCTL_CFG_VCO_480), 120000000);

//*****
//*****
    //Initialize and configure I2C0(writer) as master and I2C3(reader) as
slave

//*****
//*****
    hali2cDriverInit();

    //configure i2c0 Master to write
    HalI2cCoreStateHandle i2cMasterHandle =
hali2cCoreInit(hali2c0sclPB2sdaPB3,
                master,
                write,
                SLAVE_ADDRESS,
                SysClock,
                &i2cMasterMemoryAreaTx_[0],
                i2cMasterTxAreaMemorySize_,
                &i2cMasterMemoryAreaRx_[0],
                i2cMasterRxAreaMemorySize_);

    //configure i2c3 slave to receive
    HalI2cCoreStateHandle i2cSlaveHandle =
hali2cCoreInit(hali2c3sclPK4sdaPK5,
                slave,
                read,
                SLAVE_ADDRESS,
                SysClock,
                &i2cSlaveMemoryAreaTx_[0],
                i2cSlaveTxAreaMemorySize_,
                &i2cSlaveMemoryAreaRx_[0],
                i2cSlaveRxAreaMemorySize_);

//*****
//*****
    //Configuration complete

//*****
//*****

    unsigned int sentBytesCount;

    //TRANSMIT THE CONTENTS OF "i2cMessage_" OVER THE I2C BUS

```

```

//the message will end up in the RX circular buffer of the I2C
slave(I2c3)
sentBytesCount = 0;
while (sentBytesCount < i2cMessageLength_) {
    sentBytesCount += halI2cTx(
        &i2cMessage_[sentBytesCount],
        i2cMessageLength_ - sentBytesCount,
        i2cMasterHandle
    );
}

//READ THE MESSAGE BACK FROM THE I2C3 RX Circular Buffer into the
array "i2cCircularBufferMessageRead_"
//receivedBytesCount should be equal to sentBytesCount after
reading the message
unsigned int receivedBytesCount = 0;
receivedBytesCount =
halI2cRx(&i2cCircularBufferMessageRead_[receivedBytesCount],
        i2cMessageLength_,
        i2cSlaveHandle);

return 0;
}

```

CHAPTER 5

TESTING

In order to properly test the driver, I determined the most important test cases. The size of the RX circular buffer is 32 bytes. This is an arbitrary size that is configurable in the main program. This means the maximum single message size is also only 32 bytes. The example code from section 4.4.2 was used as the base code for all testing.

The test cases are as follows:

Test Case 1: 32 byte message

Test Case 2: 0 byte message

Test Case 3: 1 byte message

Test Case 4: 15 byte message

Test Case 5: 35 byte message

Test Case 6: second transmission

5.1 Test Case 1: 32 byte message.

A “full” message buffer:

```
56 static char const i2cMessage_[i2cMessageLength_ + 1] =  
57     "TEST1I2CTEST2I2CTEST3I2CTEST4I2C"; //32 byte message  
58
```

Figure 17: Input data for test case 1

Figure 18: Output for test case 1

(x)= Variables		Registers	
Name	Type	Value	
▼ i2cCircularBufferMessageRead_	unsigned char[40]	[84 'T', 6	
(x)= [0]	unsigned char	84 'T'	
(x)= [1]	unsigned char	69 'E'	
(x)= [2]	unsigned char	83 'S'	
(x)= [3]	unsigned char	84 'T'	
(x)= [4]	unsigned char	49 'I'	
(x)= [5]	unsigned char	73 'I'	
(x)= [6]	unsigned char	50 '2'	
(x)= [7]	unsigned char	67 'C'	
(x)= [8]	unsigned char	84 'T'	
(x)= [9]	unsigned char	69 'E'	
(x)= [10]	unsigned char	83 'S'	
(x)= [11]	unsigned char	84 'T'	
(x)= [12]	unsigned char	50 '2'	
(x)= [13]	unsigned char	73 'I'	
(x)= [14]	unsigned char	50 '2'	
(x)= [15]	unsigned char	67 'C'	
(x)= [16]	unsigned char	84 'T'	
(x)= [17]	unsigned char	69 'E'	
(x)= [18]	unsigned char	83 'S'	
(x)= [19]	unsigned char	84 'T'	
(x)= [20]	unsigned char	51 '3'	
(x)= [21]	unsigned char	73 'I'	
(x)= [22]	unsigned char	50 '2'	
(x)= [23]	unsigned char	67 'C'	
(x)= [24]	unsigned char	84 'T'	
(x)= [25]	unsigned char	69 'E'	
(x)= [26]	unsigned char	83 'S'	
(x)= [27]	unsigned char	84 'T'	
(x)= [28]	unsigned char	52 '4'	
(x)= [29]	unsigned char	73 'I'	
(x)= [30]	unsigned char	50 '2'	
(x)= [31]	unsigned char	67 'C'	

As expected, all 32 bytes which were sent over the bus ended up in the *i2cCircularBufferMessageRead* buffer.

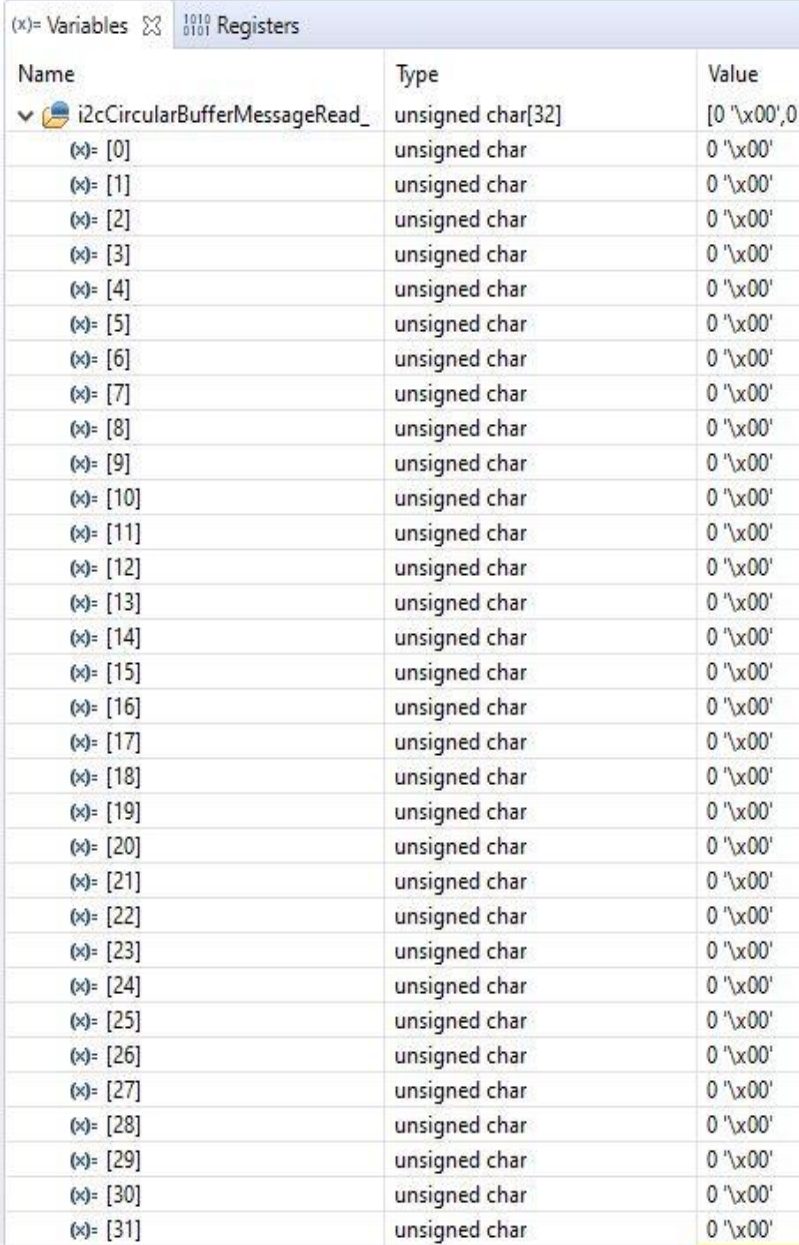
5.2 Test Cast 2: 0 byte message.

An “empty” 32 byte message buffer:

```
56 static char const i2cMessage_[i2cMessageLength_ + 1] =  
57     "" ; //0 byte message
```

Figure 19: Input data for test case 2

Figure 20: Output for test case 2



Name	Type	Value
▼ i2cCircularBufferMessageRead_	unsigned char[32]	[0 '\x00',0
(x)= [0]	unsigned char	0 '\x00'
(x)= [1]	unsigned char	0 '\x00'
(x)= [2]	unsigned char	0 '\x00'
(x)= [3]	unsigned char	0 '\x00'
(x)= [4]	unsigned char	0 '\x00'
(x)= [5]	unsigned char	0 '\x00'
(x)= [6]	unsigned char	0 '\x00'
(x)= [7]	unsigned char	0 '\x00'
(x)= [8]	unsigned char	0 '\x00'
(x)= [9]	unsigned char	0 '\x00'
(x)= [10]	unsigned char	0 '\x00'
(x)= [11]	unsigned char	0 '\x00'
(x)= [12]	unsigned char	0 '\x00'
(x)= [13]	unsigned char	0 '\x00'
(x)= [14]	unsigned char	0 '\x00'
(x)= [15]	unsigned char	0 '\x00'
(x)= [16]	unsigned char	0 '\x00'
(x)= [17]	unsigned char	0 '\x00'
(x)= [18]	unsigned char	0 '\x00'
(x)= [19]	unsigned char	0 '\x00'
(x)= [20]	unsigned char	0 '\x00'
(x)= [21]	unsigned char	0 '\x00'
(x)= [22]	unsigned char	0 '\x00'
(x)= [23]	unsigned char	0 '\x00'
(x)= [24]	unsigned char	0 '\x00'
(x)= [25]	unsigned char	0 '\x00'
(x)= [26]	unsigned char	0 '\x00'
(x)= [27]	unsigned char	0 '\x00'
(x)= [28]	unsigned char	0 '\x00'
(x)= [29]	unsigned char	0 '\x00'
(x)= [30]	unsigned char	0 '\x00'
(x)= [31]	unsigned char	0 '\x00'

The process completed without error and the *i2cCircularBufferMessageRead* buffer is empty. Since no bytes were in the message buffer this is the expected output.

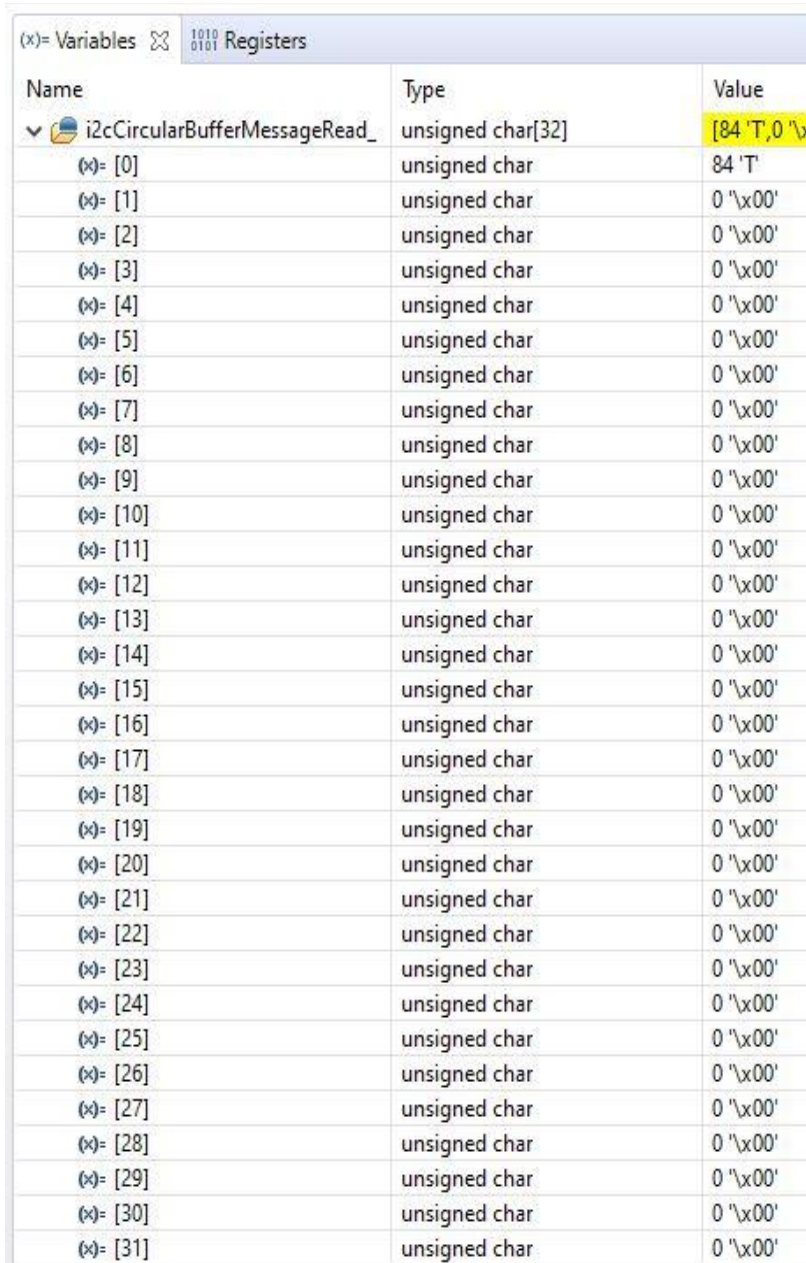
5.3 Test Case 3: 1 byte message.

A single byte in a 32 byte message buffer:

```
56 static char const i2cMessage_[i2cMessageLength_ + 1] =  
57 "T"; //1 byte message  
58
```

Figure 21: Input data for test case 3

Figure 22: Output for test case 3



Name	Type	Value
✓ i2cCircularBufferMessageRead_	unsigned char[32]	[84 'T', 0 '\x00', ...]
(x)- [0]	unsigned char	84 'T'
(x)- [1]	unsigned char	0 '\x00'
(x)- [2]	unsigned char	0 '\x00'
(x)- [3]	unsigned char	0 '\x00'
(x)- [4]	unsigned char	0 '\x00'
(x)- [5]	unsigned char	0 '\x00'
(x)- [6]	unsigned char	0 '\x00'
(x)- [7]	unsigned char	0 '\x00'
(x)- [8]	unsigned char	0 '\x00'
(x)- [9]	unsigned char	0 '\x00'
(x)- [10]	unsigned char	0 '\x00'
(x)- [11]	unsigned char	0 '\x00'
(x)- [12]	unsigned char	0 '\x00'
(x)- [13]	unsigned char	0 '\x00'
(x)- [14]	unsigned char	0 '\x00'
(x)- [15]	unsigned char	0 '\x00'
(x)- [16]	unsigned char	0 '\x00'
(x)- [17]	unsigned char	0 '\x00'
(x)- [18]	unsigned char	0 '\x00'
(x)- [19]	unsigned char	0 '\x00'
(x)- [20]	unsigned char	0 '\x00'
(x)- [21]	unsigned char	0 '\x00'
(x)- [22]	unsigned char	0 '\x00'
(x)- [23]	unsigned char	0 '\x00'
(x)- [24]	unsigned char	0 '\x00'
(x)- [25]	unsigned char	0 '\x00'
(x)- [26]	unsigned char	0 '\x00'
(x)- [27]	unsigned char	0 '\x00'
(x)- [28]	unsigned char	0 '\x00'
(x)- [29]	unsigned char	0 '\x00'
(x)- [30]	unsigned char	0 '\x00'
(x)- [31]	unsigned char	0 '\x00'

A single byte, 'T', was sent over the I2C bus, and the single byte exists in the *i2cCircularBufferMessageRead* buffer after transmission. This is the expected behavior.

5.4 Test Case 4: 16 byte message.

A “half full” 32 byte message buffer:

```
56 static char const i2cMessage_[i2cMessageLength_ + 1] =  
57 "TEST1I2CTEST2I2C"; //16| byte message
```

Figure 23: Input data for test case 4

Figure 24: Output for test case 4

Name	Type	Value
▼ i2cCircularBufferMessageRead_	unsigned char[32]	[84 'T', 69 'E'
(x)- [0]	unsigned char	84 'T'
(x)- [1]	unsigned char	69 'E'
(x)- [2]	unsigned char	83 'S'
(x)- [3]	unsigned char	84 'T'
(x)- [4]	unsigned char	49 '1'
(x)- [5]	unsigned char	73 'I'
(x)- [6]	unsigned char	50 '2'
(x)- [7]	unsigned char	67 'C'
(x)- [8]	unsigned char	84 'T'
(x)- [9]	unsigned char	69 'E'
(x)- [10]	unsigned char	83 'S'
(x)- [11]	unsigned char	84 'T'
(x)- [12]	unsigned char	50 '2'
(x)- [13]	unsigned char	73 'I'
(x)- [14]	unsigned char	50 '2'
(x)- [15]	unsigned char	67 'C'
(x)- [16]	unsigned char	0 '\x00'
(x)- [17]	unsigned char	0 '\x00'
(x)- [18]	unsigned char	0 '\x00'
(x)- [19]	unsigned char	0 '\x00'
(x)- [20]	unsigned char	0 '\x00'
(x)- [21]	unsigned char	0 '\x00'
(x)- [22]	unsigned char	0 '\x00'
(x)- [23]	unsigned char	0 '\x00'
(x)- [24]	unsigned char	0 '\x00'
(x)- [25]	unsigned char	0 '\x00'
(x)- [26]	unsigned char	0 '\x00'
(x)- [27]	unsigned char	0 '\x00'
(x)- [28]	unsigned char	0 '\x00'
(x)- [29]	unsigned char	0 '\x00'
(x)- [30]	unsigned char	0 '\x00'
(x)- [31]	unsigned char	0 '\x00'

16 bytes were sent over the I2C bus, and the 16 bytes exists in the *i2cCircularBufferMessageRead* buffer after transmission. This is the expected behavior.

5.5 Test Case 5: 35 byte message.

Extended message buffer with a 35 byte message. The RX circular buffer is still 32 bytes.

```
56 static char const i2cMessage_[i2cMessageLength_ + 1] =
57 "TEST1I2CTEST2I2CTEST3I2CTEST4I2CTES"; //35 byte message
58
```

Figure 25: Input data for test case 5

Figure 26: Output data for test case 5 with 3 lost bytes

Name	Type	Value
▼ i2cCircularBufferMessageRead_	unsigned char[35]	[84 'T', 69 'E
(*) [0]	unsigned char	84 'T'
(*) [1]	unsigned char	69 'E'
(*) [2]	unsigned char	83 'S'
(*) [3]	unsigned char	84 'T'
(*) [4]	unsigned char	49 'I'
(*) [5]	unsigned char	73 'I'
(*) [6]	unsigned char	50 '2'
(*) [7]	unsigned char	67 'C'
(*) [8]	unsigned char	84 'T'
(*) [9]	unsigned char	69 'E'
(*) [10]	unsigned char	83 'S'
(*) [11]	unsigned char	84 'T'
(*) [12]	unsigned char	50 '2'
(*) [13]	unsigned char	73 'I'
(*) [14]	unsigned char	50 '2'
(*) [15]	unsigned char	67 'C'
(*) [16]	unsigned char	84 'T'
(*) [17]	unsigned char	69 'E'
(*) [18]	unsigned char	83 'S'
(*) [19]	unsigned char	84 'T'
(*) [20]	unsigned char	51 '3'
(*) [21]	unsigned char	73 'I'
(*) [22]	unsigned char	50 '2'
(*) [23]	unsigned char	67 'C'
(*) [24]	unsigned char	84 'T'
(*) [25]	unsigned char	69 'E'
(*) [26]	unsigned char	83 'S'
(*) [27]	unsigned char	84 'T'
(*) [28]	unsigned char	52 '4'
(*) [29]	unsigned char	73 'I'
(*) [30]	unsigned char	50 '2'
(*) [31]	unsigned char	67 'C'
(*) [32]	unsigned char	0 '\x00'
(*) [33]	unsigned char	0 '\x00'
(*) [34]	unsigned char	0 '\x00'

In this case, the circular buffer is only 32 bytes long. So, the maximum transmission size is 32 bytes. The *i2cCircularBufferMessageRead* buffer array contains the first 32 bytes, but the last 3 bytes are lost.

To get an output that includes all 35 bytes from the transmitted message, the user must simply change the size of the circular buffer to be at least 35 bytes:

```
enum i2cAreaMemorySize_ {
    i2cMasterTxAreaMemorySize_ = 9,
    i2cMasterRxAreaMemorySize_ = 36,
    i2cSlaveTxAreaMemorySize_ = 9,
    i2cSlaveRxAreaMemorySize_ = 36
};
```

After making this change, there is room for all 35 bytes and the transmission will complete successfully.

Figure 27: Output data for test case 5 with no data loss

Name	Type	Value
▼ i2cCircularBufferMessageRead_	unsigned char[44]	[84 'T', 69
(x)= [0]	unsigned char	84 'T'
(x)= [1]	unsigned char	69 'E'
(x)= [2]	unsigned char	83 'S'
(x)= [3]	unsigned char	84 'T'
(x)= [4]	unsigned char	49 'I'
(x)= [5]	unsigned char	73 'I'
(x)= [6]	unsigned char	50 '2'
(x)= [7]	unsigned char	67 'C'
(x)= [8]	unsigned char	84 'T'
(x)= [9]	unsigned char	69 'E'
(x)= [10]	unsigned char	83 'S'
(x)= [11]	unsigned char	84 'T'
(x)= [12]	unsigned char	50 '2'
(x)= [13]	unsigned char	73 'I'
(x)= [14]	unsigned char	50 '2'
(x)= [15]	unsigned char	67 'C'
(x)= [16]	unsigned char	84 'T'
(x)= [17]	unsigned char	69 'E'
(x)= [18]	unsigned char	83 'S'
(x)= [19]	unsigned char	84 'T'
(x)= [20]	unsigned char	51 '3'
(x)= [21]	unsigned char	73 'I'
(x)= [22]	unsigned char	50 '2'
(x)= [23]	unsigned char	67 'C'
(x)= [24]	unsigned char	84 'T'
(x)= [25]	unsigned char	69 'E'
(x)= [26]	unsigned char	83 'S'
(x)= [27]	unsigned char	84 'T'
(x)= [28]	unsigned char	52 '4'
(x)= [29]	unsigned char	73 'I'
(x)= [30]	unsigned char	50 '2'
(x)= [31]	unsigned char	67 'C'
(x)= [32]	unsigned char	84 'T'
(x)= [33]	unsigned char	69 'E'
(x)= [34]	unsigned char	83 'S'

5.6 Test Case 6: Second Transmission

Send a second message directly after the first message using the same code, but with a different message buffer containing a new message “ANOTHER1”.

```
56 static char const i2cMessage_[i2cMessageLength_ + 1] =
57     "TEST1I2CTEST2I2CTEST3I2CTEST4I2C"; //32 byte message
58 static char const i2cMessage2_[i2cMessage2Length_ + 1] =
59     "ANOTHER1"; //8 byte array - will be sent as 2nd transmission
```

Figure 28: Input data for test case 6

Figure 29: Output for test case 6 with data overwrite

Name	Type	Value
✓ i2cCircularBufferMessageRead_	unsigned char[35]	[65 'A',7
(x)= [0]	unsigned char	65 'A'
(x)= [1]	unsigned char	78 'N'
(x)= [2]	unsigned char	79 'O'
(x)= [3]	unsigned char	84 'T'
(x)= [4]	unsigned char	72 'H'
(x)= [5]	unsigned char	69 'E'
(x)= [6]	unsigned char	82 'R'
(x)= [7]	unsigned char	49 'I'
(x)= [8]	unsigned char	84 'T'
(x)= [9]	unsigned char	69 'E'
(x)= [10]	unsigned char	83 'S'
(x)= [11]	unsigned char	84 'T'
(x)= [12]	unsigned char	50 '2'
(x)= [13]	unsigned char	73 'I'
(x)= [14]	unsigned char	50 '2'
(x)= [15]	unsigned char	67 'C'
(x)= [16]	unsigned char	84 'T'
(x)= [17]	unsigned char	69 'E'
(x)= [18]	unsigned char	83 'S'
(x)= [19]	unsigned char	84 'T'
(x)= [20]	unsigned char	51 '3'
(x)= [21]	unsigned char	73 'I'
(x)= [22]	unsigned char	50 '2'
(x)= [23]	unsigned char	67 'C'
(x)= [24]	unsigned char	84 'T'
(x)= [25]	unsigned char	69 'E'
(x)= [26]	unsigned char	83 'S'
(x)= [27]	unsigned char	84 'T'
(x)= [28]	unsigned char	52 '4'
(x)= [29]	unsigned char	73 'I'
(x)= [30]	unsigned char	50 '2'
(x)= [31]	unsigned char	67 'C'
(x)= [32]	unsigned char	0 '\x00'
(x)= [33]	unsigned char	0 '\x00'
(x)= [34]	unsigned char	0 '\x00'

A second message can be sent over the I2C bus as soon as the current transmission is complete. The problem comes when retrieving that message from the RX circular buffer. As

shown above in figure 29: if the *hall2cRx()* function is called after the second transmission, the new values will over-write the current values in the *i2cCircularBufferMessageRead* array. This may or may not be OK, depending on the use case. There is not a problem with the I2C bus communication. It is only a problem of retrieving the data after it is successfully sent over the bus.

If it is necessary to store all the transmitted values from multiple transmissions a small change in programming logic is required in the main program. The programmer simply needs to allow the variable *receivedBytesCount* to accumulate with each call of *hall2cRx()* instead of resetting it to 0 for each read operation. Then make sure the *i2cCircularBufferMessageRead* array is large enough to hold both transmissions. Then the full output from both transmissions can be stored.

Figure 30: Output for test case 6 with all bytes stored

Name	Type	Value
▼ i2cCircularBufferMessageRead_	unsigned char[40]	[84 'T', 69 'E', ...]
(x)- [0]	unsigned char	84 'T'
(x)- [1]	unsigned char	69 'E'
(x)- [2]	unsigned char	83 'S'
(x)- [3]	unsigned char	84 'T'
(x)- [4]	unsigned char	49 'I'
(x)- [5]	unsigned char	73 'I'
(x)- [6]	unsigned char	50 '2'
(x)- [7]	unsigned char	67 'C'
(x)- [8]	unsigned char	84 'T'
(x)- [9]	unsigned char	69 'E'
(x)- [10]	unsigned char	83 'S'
(x)- [11]	unsigned char	84 'T'
(x)- [12]	unsigned char	50 '2'
(x)- [13]	unsigned char	73 'I'
(x)- [14]	unsigned char	50 '2'
(x)- [15]	unsigned char	67 'C'
(x)- [16]	unsigned char	84 'T'
(x)- [17]	unsigned char	69 'E'
(x)- [18]	unsigned char	83 'S'
(x)- [19]	unsigned char	84 'T'
(x)- [20]	unsigned char	51 '3'
(x)- [21]	unsigned char	73 'I'
(x)- [22]	unsigned char	50 '2'
(x)- [23]	unsigned char	67 'C'
(x)- [24]	unsigned char	84 'T'
(x)- [25]	unsigned char	69 'E'
(x)- [26]	unsigned char	83 'S'
(x)- [27]	unsigned char	84 'T'
(x)- [28]	unsigned char	52 '4'
(x)- [29]	unsigned char	73 'I'
(x)- [30]	unsigned char	50 '2'
(x)- [31]	unsigned char	67 'C'
(x)- [32]	unsigned char	65 'A'
(x)- [33]	unsigned char	78 'N'
(x)- [34]	unsigned char	79 'O'
(x)- [35]	unsigned char	84 'T'
(x)- [36]	unsigned char	72 'H'
(x)- [37]	unsigned char	69 'E'
(x)- [38]	unsigned char	82 'R'
(x)- [39]	unsigned char	49 'I'

CHAPTER 6

CONCLUSION

6.1 Conclusion

I2C communication is currently working in a predictable and reliable way. The input data is consistently seen intact at the output. The test results show exactly the desired behavior. The successful testing indicates that the configuration of the I2C modules by the driver are correct and working up to the desired specification. Future students who need to use I2C communications for their projects should not need to do a “deep dive” into the data sheet or a textbook. Everything they need to send and receive data is here with just a few simple function calls. This thesis serves as a manual for this driver that can quickly teach a person all that they would need to know about I2C communications.

6.2 Design goals met or unmet

The driver currently fulfills every functional requirement listed in Section 1.2. An arbitrary number of bytes can be transmitted by the master and received by the slave. This number is configurable by the user based on the maximum size of the chosen data array in the main program. The FIFO is being used as the transfer medium, the interrupts are configured by the NVIC, and the circular buffer is integrated into the system.

Each software requirement listed in section 1.3 has also been fulfilled. First and foremost, the driver is easy to use. Full functionality can be demonstrated with just 4 function calls. All the mentioned protections are also included to prevent the user from making a dangerous mistake and corrupting data or memory. The user is provided with only a “handle (a pointer to a struct),” which is what allows interactions with the I2C module. This handle prevents the user from gaining direct access to any registers. The user can only interact with the driver in a very abstract way. Finally, when the data is received by the I2C slave, there is functionality to transfer that data to a buffer so that it can be processed or used in some other way by the main program.

6.3 Recommendations for expanding the driver

1. Extend the driver to work with more I2C modules.

The TM4C includes 9 I2C modules. I have only included configuration options for 2 of those 9. The driver is, however, easily scalable to include all 9.

2. Sending data from the Slave to the master

When calling the `hall2cCoreinit()` function from the main, the read or write bit allows each module to read or write whether they are the master or slave. For this project, data has only been sent from the master to the slave. Some small extension of the ISR and master control command may be necessary but sending from the slave to the master should also be possible with this driver.

3. uDMA functionality

The DMA is very fast and may be desired in the future project. The configuration function could include an argument to select FIFO or DMA and perform a different config routine for each situation.

4. de-configuration

Currently, there is no mechanism to de-configure an I2C module that is no longer in use.

ADDENDUM

Many functions exist in the driver that are not needed for Master to Slave communication via the FIFO. Some of these functions may prove to be useful, depending on project needs. These functions are prototyped in the hall2c.h header file as static, which needs to be changed if they are to be accessed by a program or application.

It is also possible to send single byte transmissions without using the FIFO. The required steps are outlined in the datasheet starting on page 1297, and the required functions already exist in the driver. These functions are a good place to start learning about I2C communications.

Manual configuration is also possible if there is a specific need that falls outside of what I have provided here. The modular design of the driver would allow a programmer to configure in any way they see fit without having to add any new functions to the driver. At the least, there is a function to configure most registers dealing with the I2C module and the FIFO.

References

- [1] Valvano, J. W. (2017). *Embedded Systems: Introduction to ARM Cortex-M microcontrollers*. USA: Jonathan W. Valvano. <http://users.ece.utexas.edu/~valvano/>
- [2] Texas Instruments, *Tiva™ TM4C1294NCPDT Microcontroller*, TM4C1294NCPDT datasheet, Oct 2013 [revised June 2014]. <https://www.ti.com/lit/gpn/tm4c1294ncpdt>
- [3] NXP Semiconductors, *I2C-bus specification and user manual*, 1982 [revised April 2014]. <https://www.nxp.com/docs/en/user-guide/UM10204.pdf>
- [4] Valdez, J., & J. B. (2015). *Understanding the I2C Bus* (pp. 1-8, Tech. No. SLVA704). Dallas, TX: Texas Instruments. <https://www.ti.com/lit/pdf/slva704>
- [5] Arora, R. (2015). *I2C Bus Pullup Resistor Calculation* (pp. 1-5, Tech. No. SLVA689). Dallas, TX: Texas Instruments. <https://www.ti.com/lit/pdf/slva689>
- [6] Ashara, A. (2015). *Using Feature Set of I2C Master on TM4C129x Microcontrollers* (pp. 1-21, Tech. No. SPMA073). Dallas, TX: Texas Instruments. <http://www.ti.com/lit/zip/spma073>
- [7] Techopedia.com. *What is Hardware Abstraction Layer (HAL)? – Definition from Techopedia*. [online]. Available at: <https://www.techopedia.com/definition/4288/hardware-abstraction-layer-hal>

Declaration

I hereby declare that I have written this work independently without outside help and that I have only used the specified aids.

City, Date

Signature