

Lawless

Elf wizard

Computer Systems

Incremental Systems Engineering

From a blinking LED
to a video game

Sean Lawless

```
static void  
// Roll  
int d20  
// Sim  
d20roll  
if (d20roll < 20)  
d20roll = 20  
Remote target  
bt  
creature_attack (creature, &TheCharacter, 1, range = 1)  
stack=0x19b54 <TheCharacter>  
virtual_world.c:748  
0x00013564 in play_move  
virtual_world.c:1730  
0x000146c0 in North (command=0x19b54)  
0x0000bc80 in ShellRoll (data=0x19b54)  
0x00009ce8 in OsTick (data=0x19b54)  
0x00009d8c in OsStack (data=0x19b54)  
0x00014acc in main (data=0x19b54)  
p *attack  
(SpriteFile *) 0x19b48 <TheCharacter>
```

Computer Systems

Incremental Systems Engineering

Sean Lawless

This book is for sale at <http://leanpub.com/computersystems>

This version was published on 2019-08-26



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2018 - 2019 Sean Lawless

Contents

Chapter 1: Binary Computers	1
1.1 Introduction	1
1.2 Binary Numbers	2
1.3 Large Binary Numbers	8
1.4 Binary Computers	11
1.5 Bit Manipulation	13
Chapter 1 Glossary	14
Chapter 1 Exercises	16
Chapter 2: Machine Language	18
2.1 Algorithms and Machine Language	18
2.2 Bit Endianess	19
2.3 Assembly Language	20
2.4 Create a Program in Assembly Language	21
2.5 Variable sizes and roll over	22
2.6 Branching and Loops in Assembly Language	24
Chapter 2 Glossary:	25
Chapter 2 Exercises:	26
Chapter 3: Compiled Systems	28
3.1 Origin of the C Language	28
3.2 C Language Basics	29
3.3 C Language Data Types and Sizes	32
3.4 C Language Math and Bit Manipulation	34
3.5 C Language Functions	38
3.6 C Language Organization	39
3.7 The C main() function	40
3.8 C Language Variable Scope and Volatility	41
Chapter 3 Glossary:	42

CONTENTS

Chapter 3 Exercises:	44
Chapter 4: System Architecture	45
4.1 Address Space and Software Memory Map	45
4.2 Memory Address and Pointers	46
4.3 Using C Pointers with Peripheral Registers	47
4.4 Create Software with an Editor and Compiler	49
4.5 Creating an Executable with the Linker	50
4.6 Configuring General Purpose Input Output (GPIO) pins	51
4.7 Debugging	53
Chapter 4 Glossary:	55
Chapter 4 Exercises:	56
Chapter 5: Timer Design	57
5.1 Hardware Clocks	57
5.2 System Software for Hardware Clocks	58
5.3 Software Interface for Clocks	59
5.4 Software timer uses	63
5.5 Building Software with Make	64
5.6 Project Management	67
5.7 System Software Organization	68
Chapter 5 Glossary:	69
Chapter 5 Exercises:	70
Chapter 6: Universal Asynchronous Receiver Transmitter (UART)	71
6.1 UART Introduction	71
6.2 UART Hardware	71
6.3.1 Configure the Development PC	73
6.3.2 Configure the System Software	73
6.4 Send Data with Software	76
6.5 Receive data with Software	79
6.6 System Shell	80
Chapter 6 Glossary:	87
Chapter 6 Exercises:	88

Chapter 1: Binary Computers

1.1 Introduction

This journey into the world of understanding computer systems encourages creativity and attention to detail. Modern computer systems can be overwhelming to understand, however when concepts are introduced carefully and cumulatively, the passage of knowledge has its best chance. Every concept learned will build upon the last, creating a knowledge base. Open your mind as you embark on an incredible, no experience necessary, journey to bring a computer to life with software! Computer hardware is nothing but hazardous waste without software, while software is meaningless text and gobbledygook without working computer hardware. Only by combining these two incredible technologies can something meaningful be achieved.

Computers can only compute, they cannot think. This statement may sound simple, or even untrue with the impressive capabilities of modern handhelds and the cloud. In order to understand how to communicate with CPU's, hardware **peripherals** and other hardware resources with software, one must learn to think like a computer and to understand the limitations and capabilities. To direct this journey, the first major goal will be to activate an LED light on physical hardware. Then an interactive communications interface will be established between a PC and the system software we create and **execute** on computer hardware. This communications interface we create will allow rapid software development.

The concept of writing software to drive/control hardware is often described as system engineering. The software interface between the hardware and the user software application(s) is the system software. A portion of this software interface to hardware is often described as a **driver**, and the initial attempt to execute system software or drivers on new hardware is often described as a **bring up** or 'power on' experience. System software is the collection of all the drivers needed for a particular hardware and application. This software needs to be reliable and optimized for the CPU and specific hardware **peripherals**, or the end application and/or user experience will suffer.

All software created in this book, as well as the companion laboratory book, are available to the public at the authors GitHub page, <https://github.com/sean-lawless/-computersystems>. This repository holds the latest source code, chapter by chapter, which can and should be used to follow along with the chapters of this book.

1.2 Binary Numbers

The first step to think like a computer is to understand **binary** numbers. At its core, a computer can only understand off and on, or zero (0) and (1) respectfully. This zero or one number system is known as **binary**, and a single zero or one digit is called a **bit**. Modern computers can **operate** on many **bits** at a time and in fact are defined by how many **bits** they can **operate** upon per **instruction**. Older computers are 8 or 16 **bit**, while today's computers (PCs, laptops, tablets, phones, etc.) are either 32 or 64 **bit**. When a computer is **executing**, it is performing a single **operation** every **clock cycle**.

The power of a computer is in its ability to quickly **operate** on multiple **bits** at a time, but what does this mean? **Binary** numbers, when combined, represent an **integer** number as a series of **bits** that each represent a binary digits represented as escalating power of twos. Mathematically **binary** can be explained as a sequence of (2^n) additions that represent the 0-n **bits**. A mathematical series is a sequence of additions, so whole **integer** numbers are represented in **binary** as a sequence of power of twos. Remember from math class that any number to the power of 0 is one, so 2 to the power of zero is 1. Two to the power of 1 (2^1) is $2 * 1$ or 2, and two to the power of two (2^2) is $2 * 2$ or 4. Two to the power of three (2^3) is $2 * 2 * 2$ or 8, two to the power of four (2^4) is $2 * 2 * 2 * 2$ or 16, is the pattern identified yet? Combinations of these binary values can represent any whole number or **integer** value. Let us review more examples that show how to use a series (addition) of power of two's to represent any larger number.

Fig 1: Binary as power of two

Binary	Integer	Binary Calculation
0000	0	0
0001	1	(2^0)
0010	2	(2^1)
0011	3	$(2^1) + (2^0)$
0100	4	(2^2)
0101	5	$(2^2) + (2^0)$
0110	6	$(2^2) + (2^1)$
0111	7	$(2^2) + (2^1) + (2^0)$
...
1111	15	$(2^3) + (2^2) + (2^1) + (2^0)$

It may help to compare binary numbers to the commonly used base 10 number system everyone uses. The common base 10 number system is of the same design except each base 10 digit has ten possibilities so each digit represents the power of 10. Each digit in the common integer is representing in the value 0 through 9 multiplied by the digit value. Binary digits can hold only two values vs the base 10 number holding 10 values. But we can describe those digits with a similar equation as with binary above. The difference being that for binary numbers the digit value multiplier is one so the equation is reduced. The elimination of the digit multiplier and reduction of this equation for binary numbers has led to efficiencies within hardware that allow calculations to be performed very quickly.

Fig 1: Base 10 as power of ten

Base 10	Integer	Base 10 digit Calculation
0000	0	0
0001	1	$(10^0) * 1$
0002	2	$(10^0) * 2$
0010	10	$(10^1) * 1$
0020	20	$(10^1) * 2$
0100	100	$(10^2) * 1$
0200	200	$(10^2) * 2$
1000	1000	$(10^3) * 1$
2000	2000	$(10^3) * 2$
...
1111	1111	$(10^3)1 + (10^2)1 + (10^1)1 + (10^0)1$

Fig 1: Base 10 as power of ten

Base 10	Integer	Base 10 digit Calculation
2222	2222	$(10^3)2 + (10^2)2 + (10^1)2 + (10^0)2$

One way to better understand **binary** numbers is to count in **binary** with your fingers. Each hand has five fingers to represent 5 **bits**, **integer** value 0 through 31, so both hands can represent a total of 10 **bits**, which can represent the **integer** numbers 0 through 1023. This is a bit different than traditional finger counting. This hand representation of **binary** can be helpful in later sections to help understand **binary operations**. With your right hand and palm facing toward yourself, start the counting by first extending your thumb (least significant **bit**) and ending with the pinkie finger (most significant **bit**) to represent **bits** 0 through 5 respectfully. So to represent the number two is the index finger (**bit** 1, or 2^1). Thumb and index finger represent the number 3 (**bits** 0 and 1). To represent the number 4 is the middle finger. Notice in the table below that the **binary** calculation correlates to the finger representation in that thumb = 2^0 , index = 2^1 , middle = 2^2 , ring = 2^3 and pinkie = 2^4 . To calculate the total number of the hand sign, add up all the **bits** as represented in all of the extended fingers.

Fig 2: Binary finger counting

Binary	Integer	Finger Representation
00000	0	fist, no fingers

00000	0	fist, no fingers
-------	---	------------------

00000

Fig 2: Binary finger counting

Binary	Integer	Finger Representation
--------	---------	-----------------------


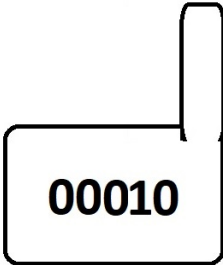
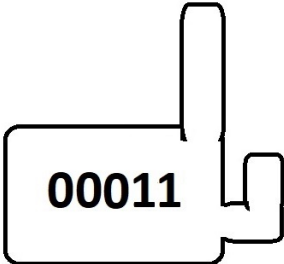
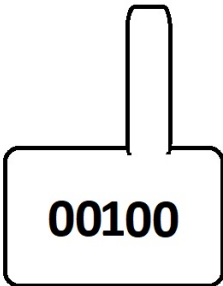
00001	1	thumb	
00010	2	index	
00011	3	index and thumb	
00100	4	middle	

Fig 2: Binary finger counting

Binary	Integer	Finger Representation
00101	5	middle and thumb
00110	6	middle and index
00111	7	middle, index, and thumb
01000	8	ring
...

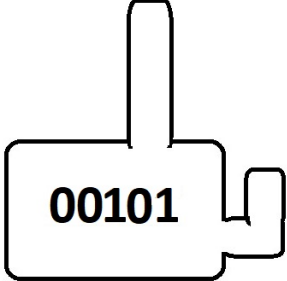
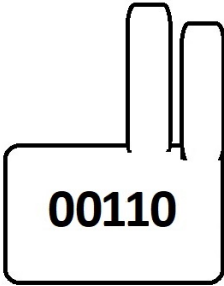
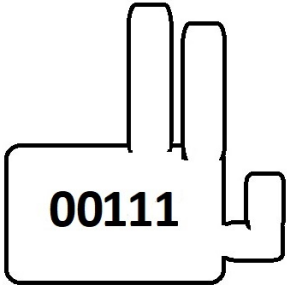
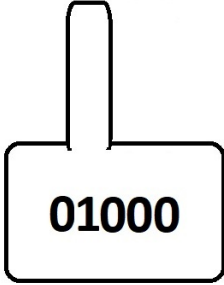
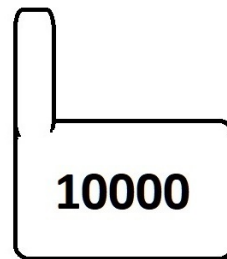
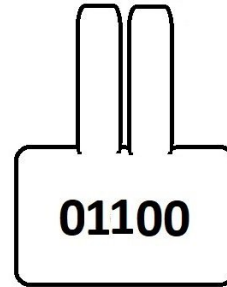
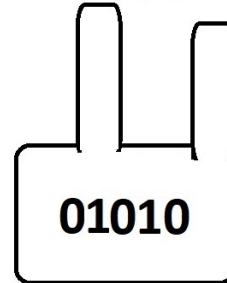
Binary	Integer	Finger Representation
00101	5	middle and thumb 
00110	6	middle and index 
00111	7	middle, index, and thumb 
01000	8	ring 
...

Fig 2: Binary finger counting

Binary	Integer	Finger Representation
01010	10	ring and index
...
01100	12	ring and middle
...
10000	16	pinkie
...

Binary	Integer	Finger Representation
01010	10	ring and index
...
01100	12	ring and middle
...
10000	16	pinkie
...



If using both hands, this convention defines to use your right hand for bits 0 through 4, while the left hand faces away from yourself (palms out) to represents bits 5 through 9 using the same finger representation as above. The thumb always represents the lowest power of two for each hand so the right hand thumb is 2^0 , while the left thumb is 2^5 . This orientation may appear arbitrary or confusing but it proves beneficial later when performing **binary operations** with our hands.

1.3 Large Binary Numbers

The **bit** size that a computer can compute upon is commonly 32 **bits**. A 32 **bit binary** number takes 32 digits to represent in human readable form. To more compactly visualize **binary** numbers, the base 16 **hexadecimal** numbering system was created and is commonly used. To represent a **binary** number (base 2) in **hexadecimal** (base 16), combine each 4 **binary bits** into a **nibble** which is then represented in **hexadecimal** as a value of 0 through 15. A **nibble** is used in the human readable form of a **hexadecimal** digit. To represent a **nibble** are the numbers 0-9 plus the characters A-F, or 0 through F. For example, 9 is 9, A is 10, B is 11, C is 12, D is 13, E is 14 and F is 15. **Hexadecimal** is a natural numbering system to use to represent **binary** because base 16 is a power of base 2. **Hexadecimal** numbers in print or display are encouraged to begin with the 0x syntax, identifying the number that follows as **hexadecimal**. This 0x has no value, it is just an indicator to the reader (and some software) that the number is **hexadecimal** instead of an **integer**.

Two 4 **bit nibbles** combined is an 8 **bit** number, commonly referred to as a **byte**. A **byte** is represented as two **hexadecimal** digits, or **nibbles**, for a value of 0 through 0xFF (255). Larger **hexadecimal** numbers are represented as multiple **bytes**, for example, a 32 **bit** number is represented as 4 **bytes** in **hexadecimal**, for a value range of 0x0 to 0xFFFFFFFF (8 **nibbles** or **hexadecimal** digits). Some examples follow.

Fig 3: Binary as hexadecimal

Binary	Integer	Hexadecimal
0000	0	0x0
0001	1	0x1
0010	2	0x2
0011	3	0x3
...
00001000	8	0x8
00001001	9	0x9
00001010	10	0xA
00001011	11	0xB
00001100	12	0xC
00001101	13	0xD
00001110	14	0xE
00001111	15	0xF

Fig 3: Binary as hexadecimal

Binary	Integer	Hexadecimal
00010000	16	0x10
00010001	17	0x11
00010010	18	0x12
00010011	19	0x13
00010100	20	0x14
...
00011111	31	0x1F
00100000	32	0x20
00100001	33	0x21
...
11111000	248	0xF8
11111001	249	0xF9
11111010	250	0xFA
11111011	251	0xFB
11111100	252	0xFC
11111101	253	0xFD
11111110	254	0xFE
11111111	255	0xFF

A 32 **bit binary** number can be used to represent an unsigned **integer** value from 0 to 4,294,967,295, or 0 to $2^{32} - 1$. An **integer** is a whole number (no decimal or fraction) represented internally as a **binary** number. Common **integer** sizes CPU's understand are 8, 16, 32 or 64 **bits** in length. Software can represent these **binary** numbers as signed or unsigned **integers**. A 32 **bit signed integer** uses one **bit** for the sign and thus has a value range from -2^{30} to 2^{30} , which represents the whole numbers -2,147,483,648 to 2,147,483,648.

Adding and subtracting **hexadecimal** numbers is the same as with the common 10 based number system, except each digit increments up or down in value from zero (0) to fifteen (15 or 0xF), instead of zero (0) to nine (9). It is often very helpful to understand hexadecimal numbers by adding and subtracting numbers as in the example below.

Fig 4: Hexadecimal arithmetic examples

Binary	Integer	Hexadecimal
0001 + 0001 = 00010	1 + 1 = 2	0x1 + 0x1 = 0x2
0011 + 0111 = 01010	3 + 7 = 10	0x3 + 0x7 = 0xA
0111 + 0111 = 01110	7 + 7 = 14	0x7 + 0x7 = 0xE
1110 + 1110 = 11100	14 + 14 = 28	0xE + 0xE = 0x1C

To perform addition, subtraction and other **operations** on large **hexadecimal** numbers with your fingers, it can be helpful to use only four **bits**, or a **nibble**, for each hand. This allows for more easily performing **operations** and frees up the thumb(s) for any carry **bit(s)**. With each hand representing a single **hexadecimal** digit, this **nibble** based arithmetic can be exercised on large **hexadecimal** numbers using hands and a carry bit. To be comfortable comparing the **binary** digits of each **nibble** with ones hands, the reverse orientation is helpful so that each index finger, for example, is the first **bit** in that hand.

Binary and **hexadecimal** math can be performed similar to standard base 10 math calculations. The **operation** can be calculated from right to left a **nibble** at a time with out hands, adding each column being **operated** on together with any overflow resulting in a carry over to the next column on the left. If a carry over, the thumb should be used going into the next calculation to remind the counter that there is a carry over for the first **bit** in the next **nibble**. Let us end this section with three identical examples showing how to add the same number in **binary**, **integer** and **hexadecimal**.

Fig 5: Hexadecimal large integer addition first example

Binary	Integer	Hexadecimal
0000 0001 0000 0110 0100	04196	0x00001064
0000 0010 0000 1001 0110	08342	0x00002096
0000 0011 0000 1111 1010	12538	0x000030FA

In the example the **binary** digits are stacked on top of each other and compared digit by digit from right to left. If both bits are one (1), then the result is binary zero (1) but a **bit** is carried over to the addition of next digit on the left. If next digits to add are also both one (1) and a bit has been carried to it, the result is 1 and another bit is carried to the left. Just like adding digits in base 10 numbers results in carrying over digits to the next digit. This shifting carry or shifting of **bits** is exactly how the computer **operates** internally to the user.

The **hexadecimal** digits/**nibbles** for the addition **operation** look similar to the standard 10 based number system and may be familiar. Hexadecimal is base 16, binary is base 2 and standard base 10 should all be recognized by the reader as familiar mathematical systems that can be used to accomplish the same mathematical goals. The only difference between the three are in their perspective of the numbers.

Fig 6: Hexadecimal large integer second addition example

Binary	Integer	Hexadecimal
0000 1111 1111 1011 0100	65460	0x0000FFB4
0000 0100 1101 1100 0101	19909	0x00004DC5
0001 0100 1101 0111 1001	85369	0x00014D79

To summarize the sections, large **binary** numbers (represented as **hexadecimal**) can represent any ten (10) based number and that the mathematics is unchanged as long as the numbers do not exceed the **CPU bit** size. The only change in the perspective is to the data that enters the mathematical equation; the human number (base 10), and the computer (binary or base 2). Computers and hardware **peripherals** compute, calculate, act, and perform upon **binary** data. To communicate with **binary** hardware requires a number system to represent **binary**, and **hexadecimal** and **nibbles** do this.

1.4 Binary Computers

Computers **execute** a series of **CPU instructions** and computations upon **binary** numbers stored in **registers** to perform a **program** or computation. Every CPU has hardware **registers** that are the same **bit** size as the CPU. For example, a 32 **bit** CPU will have 32 **bit registers**. This allows a single **CPU operation** to perform a computation on all 32 **bits** of the **register** in a single **clock cycle**. There are a limited number of **registers** within the CPU to hold the data to be calculated upon during a **CPU operation**. Computer systems consist of a CPU, **RAM** and **peripherals**. Many CPU's cannot perform calculations on **RAM** directly, only hardware **registers**. Calculations that are to be performed on data in **RAM** may first require **CPU operations** to move the value from **RAM** into a **register** before the calculation (another **operation**) can be performed by the CPU. Then another **operation** may be required to move the result back into **RAM** after the calculation. Every CPU has only a limited number of hardware **registers** so **executables** must move data back and forth between **RAM** and hardware **registers** often.

While each **operation** can operate on data in **registers**, some **operations** require one **register** be designated the destination **register** that will contain the result of the **operation** or calculation. A computer computes when it is provided an **operation** and the **registers** to use for the **operator** inputs and outputs. For example, if one number is added to another, one **operation** and at least two **registers** are used. Both are added together but only one will contain the result of the **operation** after completion at the end of the **clock cycle**. The exact behavior of **operations** are CPU specific. A sequence of **operations** or **instructions** that performs a meaningful goal or computation is called a **program**, or **software**.

The reader is now prepared with enough vocabulary to precisely describe how a modern computer works. A computer is a **motherboard** with **RAM**, input and output **peripherals** and a **CPU** with a limited number of hardware **registers**. The **CPU instruction set** is all the available **CPU operations** that can be used by **programs** to perform a calculation or sequence of calculations for that **CPU**. The computer **CPU** can begin **executing** an **operation** every **clock cycle**. How many **operations** that can be performed per second is often referred to as the **CPU frequency**. The higher the frequency of the **CPU clock**, the more **operations** or **instructions** that can be performed per second. Modern computer **CPU's** have frequencies in the GHz range, or billions of times a second. For this simplified discussion, let us assume **operations** are **executed** one before the other or sequentially, and each **operation** completes in a single **clock cycle**.

The **CPU executes** software, or runs a **program**, by reading a sequence of **binary operations** from **RAM**. The **Program Counter**, or **PC**, is the location in **RAM** that is considered by the **CPU** as the current location to **execute** the next **operation**. Essentially, the **CPU** moves the **PC** through the **program** in **RAM**, **executing** the **operations** one at a time. At the end of each **clock cycle**, the **CPU** moves the **PC** to the next **operation** in the binary executable stored in **RAM**, which is then **executed** during the next **clock cycle**. Oftentimes the next **operation** in **RAM** is moved to a special hardware **register** at the end of a clock cycle so that it can be **executed** next cycle, but this depends on the **CPU** and is conceptually the same.

It is not possible to complete our knowledge without quickly discussing why computers can only compute with **binary** numbers. This has to do with the physical nature in which computers are built, namely the transistor. The first computers used vacuum tubes to maintain **bits** and hardware state machine circuits to perform **binary operations**. Vacuum tubes can maintain the state of on or off, that is one (1) or

zero (0) or a **binary** digit. Racks of vacuum tubes were used for **RAM** in the first computers, while the walls had the variable routing **circuitry** of state machines to compute or **execute** a limited number of **operations** or **instructions**. Vacuum tubes were expensive and prone to failure, and in fact the term ‘bug’ in software engineering came from a physical bug whose body had shorted a switch which led to intermittent bouts of unexplained behavior. The size, expense and fragility of vacuum tubes led to the invention of the practical transistor, and later to further miniaturization and eventually the spawn of the computer age. The simplicity and scalability of the transistor to large **binary** numbers ensures that as long as humanity pushes electrons around to compute, the fastest computers will use **binary**.

1.5 Bit Manipulation

Since CPU **operations** are all performed on **binary** numbers, a few **bit operators** have evolved and are essential to systems engineering. A common **bit operation** is the shift operator. The shift operator moves, or shifts, every **bit** of a larger **binary** number in one direction or the other. Shift right and shift left move the existing **bit** pattern of a larger **binary** number one or more **bits** in the right or left direction respectfully. This is helpful as it is simple to perform shift **operations** with hardware and this shifting serves a mathematical purpose. A right shift of one divides a large **binary** number by 2, while a left shift multiplies a large **binary** number by 2. Another simple **operation** involving a single **binary** number is the one’s complement, which **flips** every bit in a large **binary** number. For example, each one (1) becomes zero (0), and zeros (0) become ones (1), for every corresponding **bit** in the number.

Other **bit operations** involve operating upon two large **binary** numbers to create a result dependent upon the **operation** and the two **binary** numbers. The **operations And, Or, and Xor** (exclusive **Or**) are all easy to compute **bit by bit** for two large **binary** numbers with a CPU. The first two **operators** mentioned are somewhat easy to explain in that for **And**, if both **bits** are one (1), then the result is one (1), otherwise the result is zero (0). For example $A \text{ And } B \text{ equals } C$, where C has only **bits** set to one (1) if the corresponding **bit** in both A And B are set to one (1). **Or** is the same thing but the result is set to one (1) if either the corresponding **bit** in A Or B is set to one (1). For **Xor**, or exclusive **Or**, the result is set to one (1) for every corresponding **bit** where the corresponding **bit** in A is different from that in B.

So how does a computer use these **binary operations** to perform calculations? Besides

multiplying by two and divide by two, the **And** and **Xor bit operations** can be used together for addition. Let's end this chapter with an example of how a computer adds two **binary bytes** together.

Fig 6: Binary addition using And and Xor

Binary Add	And	Xor
1011 0100	1011 0100	1011 0100
1100 0101	1100 0101	1100 0101
1 0111 1001	1000 0100	0111 0001

See the pattern and how this can be used for addition? Not many do at first. Let's take the results from the table above one step further and shift the And result one **bit** to the left, so 1000 0100 in the table above becomes 1 0000 1000. Now **Or** this result with the **Xor** result above so 1 0000 1000 **Or** 0111 0001 becomes 1 0111 1001, or the result of the addition above. To complete our knowledge let us write this as a **binary equation**. $A + B = ((A \text{ And } B) \text{ Shift left 1 bit}) \text{ Or } (A \text{ Xor } B)$. If you did not understand this no worries, just remember that a computer performs mathematical computations using combinations of **binary operations**.

Chapter 1 Glossary

And - a binary **operation** supported by CPU's to compare each bit of two numbers and output a one (1) for each corresponding bit if both bits are one (1).

Bare Metal - a computer **software execution** environment that requires **software** to initialize all hardware required by the **program** that is to **execute**.

Binary - a numerical system used by computers where each digit is either zero or one.

Bit - a single binary digit holding the value of zero or one.

Bring Up - A phrase used to mean creating a software driver for a peripheral. Board **bring up** refers to creating software for all the peripherals of a system. Oftentimes the software drivers already exist and the act of bring up is modifying the old driver for the new hardware.

Byte - eight (8) bits of **binary** data, typically represented as two **hexadecimal** digits, or **nibbles**.

Circuit - an electrical connection between two points. For example, **circuits** on a **motherboard** connect the **CPU** to the **peripherals**. **Circuits** can contain resistors, capacitors, gates and other electronic components to manipulate the electrical signal through them.

CPU - Central Processing Unit, the core of a computer that performs **operations** or **instructions**.

Clock cycle - a complete interval of a clock where the rate of **clock cycles** is the frequency. The **CPU execution** clock performs a single **CPU operation** every **clock cycle**. Systems can have more than one clock in which case each may have their own **clock cycle** frequency.

Execute - when a **CPU** performs the **operations** of a **program** until it ends. The **Program Counter (PC)** maintains the location within the **program** that is currently **executing**.

Hardware - a general term used to describe any physical computer component, such as a **CPU**, **motherboard** or **peripherals**.

Hexadecimal - a base 16 (or 24) numbering system used to visually represent large **binary** numbers. **Hexadecimal** digits are represented as 0 through 9, and the letters A, B, C, D, E and F. These letters represent the **integer** values 10, 11, 12, 13, 14 and 15 respectfully.

Instruction - see **Operation**. Besides calculations, **instructions** can compare, branch or perform other available **CPU operations**.

Instruction Set - the complete set of available **operations** that a **CPU** can perform.

Integer - any whole number represented as a **binary** number.

Motherboard - usually a thin fiberglass type material of a green or brown color where the **CPU**, **peripherals** and other physical pieces that make up the computer are located. The **motherboard** contains all the **circuits** and electrical components necessary to connect the **CPU** to the **peripherals**.

Nibble - four (4) **bits** of **binary** data, typically represented as a single **hexadecimal** digit.

Operation - a single calculation or other **instruction** performed by the **CPU** during a **clock cycle**.

Operators or Operands - data in **RAM** or hardware **registers** that are calculated upon during a **CPU operation** or calculation.

Or - a **binary operation** supported by **CPU's** to compare each **bit** of two numbers and output a one (1) for each corresponding **bit** if either of the **bits** being compared are one (1).

Peripheral - a portion of computer hardware separate from the **CPU**, but connected to the **CPU** through **circuits** on the computer **motherboard**.

Program - a series of computer **operations/instructions** that performs a specific computation.

Program Counter (PC) - The memory address or location of the **binary operation** that the **CPU** is currently **executing**. This is often the current location within **RAM** of a **program** or **software** previously copied into **RAM**.

RAM - Random Access Memory, the computer memory, is often located outside of the **CPU** and connected with **circuits** on the **motherboard** to the **CPU**. Commonly referred to simply as memory.

Register - A limited number of high speed data storage resources within the **CPU** and physically near the computational **circuitry** of a **CPU**. Data to be computed is often required to be in a **register** for it to be used in **operations** that compute or compare. Data to be computed must often first be moved from **RAM** memory to registers before the computation or comparison **instruction** is executed.

Software - **executable** binary or machine code that performs a solution to a task or user request. A generic term that can refer to multiple **programs** or a portion of a single **program**.

Xor - a **binary operation** supported by **CPU's** that compares each **bit** of two numbers and output a one (1) for each corresponding **bit** only if the **bits** being compared do not equal each other (one **bit** is zero and other one, or vice versa).

Chapter 1 Exercises

1. Using your fingers to represent each **binary** number, count to ten (10) on one hand.

2. What is the **hexadecimal** representation of the **binary** number 10110011? Hint, split into **nibbles** 1011, and 0011 and do not forget to start the **hexadecimal** result with '0x'.
3. Using your fingers to represent each **binary** number, count to fifty (50).
4. What is the **binary** representation of the **hexadecimal** value 0xF63? Hint, split into **nibbles** and then **binary**.
5. What are the **integer**, **hexadecimal** and **binary** representations of the individual decimal numbers 24, 28, 213, and 216? Please compute the sum of these numbers, $24 + 28 + 213 + 216$, and present in **integer**, **hexadecimal** and **binary** form.
6. A 32 **bit** system has 1,000,000 **bytes** of **RAM** filled with a **binary executable**. On startup the **Program Counter (PC)** begins **execution** of these **instructions** at the start of the **RAM** so that all the **RAM** may be utilized during **executing** the **operations**. Assume the entire **RAM** is filled with valid **operations** that **execute** in order from beginning to end. Also assume all **operations** are the same size and move the **Program Counter (PC)** ahead 32 **bits** every **clock cycle**, from the beginning of **RAM** to the end. What is the number of **operations** that can be **executed** before the end of the **RAM** is reached?
7. If the above system runs at 1 Megahertz (MHz), or 1,000,000 **clock cycles** per second, how long will it take to execute all instructions in **RAM**?
8. Describe your understanding of the differences and similarities between a **program** and **software**.

Chapter 2: Machine Language

2.1 Algorithms and Machine Language

The previous chapter discussed how computers operate on binary numbers. Now we can begin to discuss how computer operations can be combined to perform complex calculations using **algorithms**. An **algorithm** is a sequence of computations that performs a solution to a problem. A program typically utilizes one or more **algorithms** which may need to communicate with one or more hardware components to arrive at a solution. A CPU executes operations in sequential order until a **branch** operator is executed, which modifies the execution location register or Program Counter (PC). Once a **branch** operator changes the PC, the CPU immediately continues execution from the new location in the program until it ends, or the next **branch** operation. Binary computer operations are referred to as **machine language**. This **machine language** is directly executable by the computer and is not human readable. It can be created by hand with a binary/hexadecimal editor but is usually generated from **assembly language** with an **assembler**.

Below is an example of the **assembly language** flow to continually loop on a sequence of four operations and not continue to the fifth operation until a comparison becomes equal.

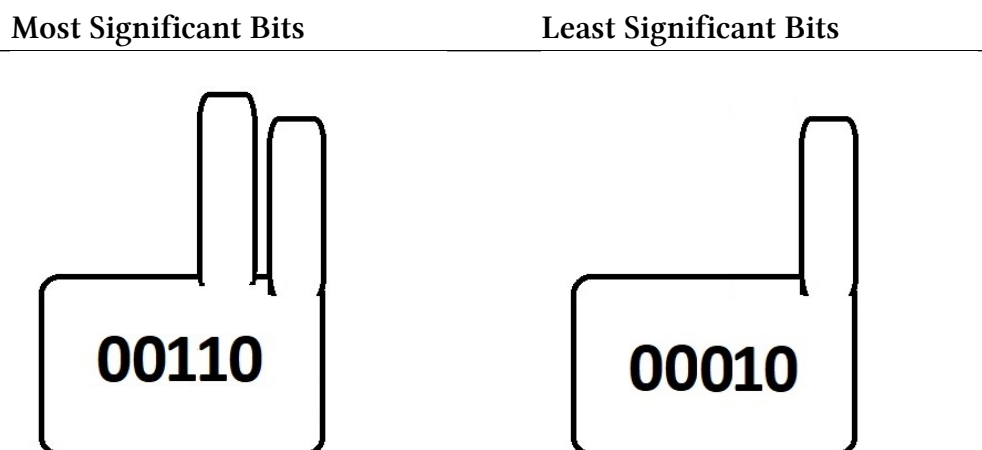
```
loop: operation1
      operation2
      operation3
      operation4
      compare
      if not equal branch to loop:
      operation5
```

2.2 Bit Endianness

An important concept to understand when writing system software is **endianness**. CPU's represent large binary numbers in registers and **memory** in two different ways. Least significant bit (LSB) first is little **endian**, while most significant bit (MSB) first is big **endian**. For example, ARM and x86 CPU's are both little endian while PowerPC CPU's are big **endian**. Peripherals and software may also have interfaces which require a specific bit size and **endianness**, which may or may not match the CPU.

To better understand **endianness** see the pictures of a reverse hand method (right hand on right side, palm facing you, left hand on left side, palm facing out) below. Represent the binary number 00110 00010, or hexadecimal 0xC2, or integer (decimal) value 194. The right hand thumb is the least significant bit in binary format, the left hand pinkie finger is the most significant bit. This is called little **endian** format as the smallest bit is on the far right (end of the bits). See the pictures below to see this visually.

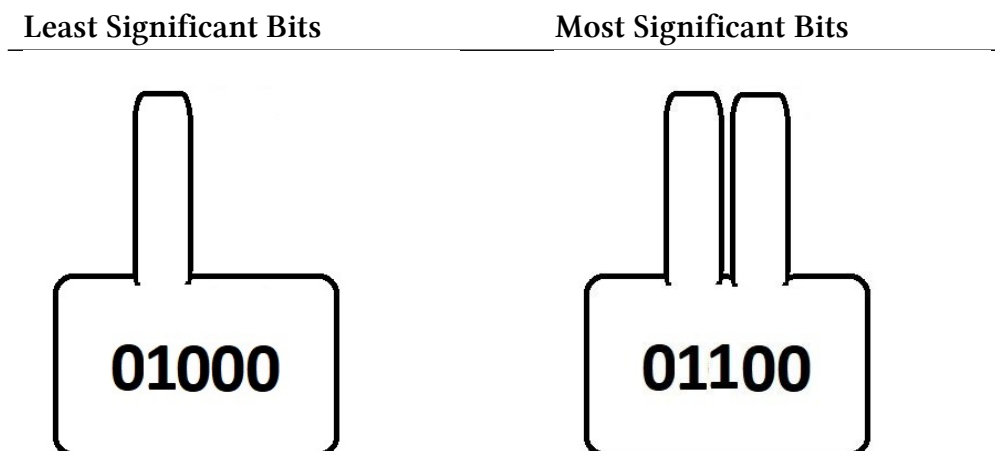
Fig 7: Little endian representation of 0xC2 with fingers



To convert this number to most significant bit format, rotate and swap your hands. In the new orientation the thumb of each hand is still the least significant bit, but the thumb moves from the right side to the left of each hand and the right hand moves to the left side palm facing out, and the left hand moves to the right side palm facing toward you. The last ending bit is the pinkie of the left hand, the most significant bit.

This swapping and rotating results in the same hexadecimal number 0xC2, but it now becomes 01000 01100 when represented as most significant bit, or big **endian**, binary format.

Fig 8: Big endian representation of 0xC2 with fingers



The endianness does not affect the concept of the various binary operations, but does affect the resulting machine language and how those operations are performed by the CPU. For the most part the endianness of a CPU or peripheral is hidden from the user, until it is required that a peripheral of one endianness communicate with a CPU of a different endianness, in which case a translation must be done in software between the two.

2.3 Assembly Language

This section discusses **assembly language**, the readable form of **machine language**. **Machine language** is in binary form and is very difficult for most people to understand in hexadecimal form, but is ready for immediate execution by computers. There are tools called **disassemblers** which can read **machine language** and convert it back into readable **assembly language**. **Disassemblers** are one of many tools used on the journey to understand creating software for computer systems. **Assembly language** is a primitive **assembled** language because it provides a human readable representation of the **machine language**, which may differ by CPU type and model. With a few exceptions, one operation in **assembly** source code **assembles** into one **machine language** operation.

Assembly instructions represent the CPU hardware operations, or instructions, and can be categorized into **move**, compare, **branch** or compute operations. Operations are combined and ordered to form **algorithms**. Most common operations can be classified into similar categories for all CPU's. For this discussion we will use a make believe instruction set that is very similar to modern instruction sets. Every instruction is represented as 4 bytes (32 bits) and contains an operation type (ADD, MOVE, etc.) and up to two registers or values as operands. Let us view an **assembly language** example that adds two numbers.

Fig 9: Assembly to Machine Language

Assembly	Machine Language
ADD R4, R4, #1	0xE2844001

The above **assembly language** ADDs the absolute value (#) of one (1) to register four (R4), putting the result in the destination register (R4). The CPU **machine language** binary on the right is an example of what an **assembler** might create from the instruction on the left. This resulting **machine language** binary number is all that the computer understands. While this binary is all that computers understand, for humans this **machine language** is very difficult to read and understand.

Thankfully it is not required to understand exactly how to convert **assembly** instructions into **machine language** as this is what the **assembler** is for. If the reader is interested there are many good online and print resources that describe **assembly** and **machine language** in detail. What is important for the system engineer to understand is the concept of **assembly language** and machine language and how an **assembler** can be used to convert **assembly** to **machine language**, and a **disassembler** can be used to convert **machine language** back into **assembly** so it can be read and understood again by people.

2.4 Create a Program in Assembly Language

Using the MOVE instruction with the ADD instruction it is possible to create something useful in **assembly language**. Since **variables** are stored in **memory** (RAM), operations on those **variables** must therefore perform computations on locations in **memory**. However, many CPU operations require their operands, or values used in an operation, to be in registers, not **memory**. A hardware register is a special **memory**

location within the CPU very near to the hardware circuitry that performs operations or calculations. To accommodate the need for calculations to be performed only on hardware registers, a MOVE instruction is often needed to copy a value from a variable in **memory** to a register, before a computational operation can be performed on that value. Then after the CPU performs the calculation on the register, another MOVE instruction copies the value of the result from a register back into the **variable in memory**. The following **assembly language** example does exactly this, increasing value in **variable** 'Var1' by one (1).

```
MOVE    R1, =Var1
MOVE    R4, [R1]
ADD     R4, R4, #1
MOVE    [R1], R4
```

The above **assembly language** first **moves** the address in **memory** of **variable** 'Var1' into the first register R1. The second line **moves** the value at **memory** location R1 to another register R4. The brackets around R1 indicate to the compiler to **move** the value at this **memory** location, not the **memory** location itself which is stored in R1. In the third statement, the value one is added (ADD) to the value in R4 and the result saved back to R4. The final statement **moves** the new value of R4 back to the **memory** location of 'Var1', which is still stored in R1. These instructions **move** the **variable** to a register, add one to the register, and **move** the resulting value back to the **variable in memory**. Together, these statements increase the 32 bit value of **variable** 'Var1' by one.

The above **assembly language** is generic and will not execute on any CPU, but is very similar to real **assembly**. **Assembly language** is the direct representation of the CPU **machine language** and is therefore different for every CPU. Oftentimes CPU families such as ARM and x86 have core **assembly language** instructions that exist on all CPU versions. If it is desired for the same **assembly** to **assemble*** and execute on as many CPU's as possible, these core instructions should be preferred.

2.5 Variable sizes and roll over

Variables, such as the 'Var1' used above, must first be defined in the **assembly language** source (.s) file. A common method to do this is to create a data section

at the top of the `.s` file. Then any number of **variables** can be declared below this data section. A common syntax to declare this data section, which may differ between **assemblers**, is the `‘.section .data’` phrase. On the next line after this phrase it is possible to declare the variable name followed by a colon `‘:’`. After the colon it is required have a space character `‘ ’` and then the keyword of the variable type (`‘.int’`, `‘.long’`, etc.) followed by the initial value.

```
.section .data
Var1: .int 0    ;@ declare Var1 as an integer of value zero
```

A CPU uses integers in **assembly** of the size that the CPU can process. The **assembler** accommodates this by creating 32 bit sized `.int`’s if the CPU is 32 bit. To create a larger value such as a 64 bit **variable** often requires defining the **variable** as type `.long`. However, some CPU types define and use `.int` and `.long` the same, others assign `.int` **variables** to be 16 bit and `.long` **variables** as 32 bit. These differences between CPU’s is one of the many reason why **assembly language** does not move well from one computer system to the other. In **assembly** it is required to pay attention to the CPU bit size and to use `.int` and `.long` appropriately. If in doubt, just use `.int` and the **assembler** will optimize the resulting binary code for the particular CPU.

After all **variables** are declared it is required to tell the **assembler** that we are moving out of the data or declaration section. This is done by assigning a new section, for example the text section is reserved for the **assembly** code which usually follows the data section. Similarly this text section is entered with the `‘.section .text’` phrase. Here is the completion of the previous example but with the variable `‘Var1’` declared at the top. Note that a common syntax to define a comment in **assembly** is the `‘;@’` keyword. As is good practice, we use a comment below to help clarify what the code is doing.

```
.section .data
Var1: .int 0    ;@ declare Var1 as an integer of value zero
```

```
.section .text
MOVE  R1, =Var1
MOVE  R4, [R1]
ADD   R4, R4, #1
MOVE  [R1], R4
```

When a binary integer value increases beyond the maximum for the size of the **variable**, it rolls back to zero. For example, if an `.int` of size 32 bits and of value `0xFFFFFFFF` is increased by one, the value becomes `0x00000000`. A similar affect occurs when decreasing a value below zero (0). If a **variable** of size 32 bits and value 0 is decreased by one (subtracted by one), the resulting **variable** value is `0xFFFFFFFF`. The behavior is often described as a ‘roll over’ or more completely as an ‘integer roll over’.

2.6 Branching and Loops in Assembly Language

Assembly language instructions for **branch** and compare operations are different between CPU families. However, most have operations for comparing integer values and **branching** if the comparison was equal, less or greater than. Some CPU’s have a single compare operation that must be followed by one or more of the **branching** operations. Other systems have instructions that combine the compare with a **branch** all in the same operation or instruction. The concept is the same either way.

It is now time to define more generic instructions to compare and **branch** in **assembly language**. For this make believe instruction set we define a single compare operation `CMP`, and four **branch** instructions; unconditional **branch** `B`, **branch** if equal `BEQ`, **branch** if less than `BLT` and **branch** if greater than `BGT`. The instruction `CMP` compares two registers and saves the result internally. **Branch** instructions change the next execution location (program counter or PC) to a new location identified with a label. Labels in the source code are required so the **assembler** knows where to **branch** the execution. These **branch** instructions in the pseudo code examples below do not compare but instead use the result of the last comparison. Comparison **branches** (`BEQ`, `BLT` and `BGT`) require a previous compare instruction (`CMP`) or are considered an invalid instruction. The following is the last example of this chapter, an endless loop that declares a variable in RAM, increments the variable by one each time through the loop, storing the result back into RAM every ten times.

```
.section .data
Count: .int 0    ;@ declare Count as an integer of value zero

.section .text
MOVE r1,=Count ;@ load register 1 with address in RAM of Count
MOVE r3, [r1]  ;@ load R3 with Count by referencing R1 address

_save:
MOVE [r1], r3  ;@ store the total count to Count in RAM
MOVE r2, #1    ;@ initialize the loop count to one

_loop:
ADD r3, r3, #1 ;@ add one to the total count value
ADD r2, r2, #1 ;@ add one to the loop count
CMP r2, #10    ;@ compare the loop count to 10
BGT _save     ;@ branch to _save if greater than 10
B _loop       ;@ branch to _loop
```

Chapter 2 Glossary:

Algorithm - a series of calculations that perform a larger computation or decision making process. Typically used to describe a computer program or portion thereof.

Assembler - a software program that can read and parse assembly language files and create machine language output.

Assembly Language - a human readable text representing the CPU instructions but conforming to the syntax defined by the assembler. When **assembled** with an **assembler**, an **assembly language** file will generate a **machine language** binary for the CPU. Oftentimes referred to simply as **assembly**.

Branch - An operation that jumps execution to a new location so as to continue executing the program from there. **Branch** operations allow programs to move around and loop within the executable code in RAM, performing groups of operations repeatedly and/or out of sequence.

Disassembler - a software program that can read machine language binary and

create human readable assembly language output. This is the opposite process as an assembler.

Endian - the order of binary significance when using large binary numbers. CPU's are classified as either big endian or little endian. The term refers to whether a 32 bit integer uses bit 31 to represent the most significant digit (big endian), or the least significant digit (little endian). In other words, endianness is the bit ordering CPU's use to represent large binary numbers.

Executable - a file of machine language operations resulting after linking software or a program. An executable contains the operations to be performed by the CPU in order, one after the other, until the program ends. Only a branch operation can move the Program Counter (PC) out of sequence and is used for looping or to skip a group of operations.

Machine Language - any software or program in binary format, ready for execution on hardware. Machine language is most often created with an assembler.

Memory - a location in hardware to store data. Typically this is RAM but could be any hardware where data can be stored and accessed later by software, such as memory onboard a CPU, GPU or peripheral.

Move - general term for an assembly operation that copies a value from a register or RAM to another location, such as a register or RAM.

Source Code - any program or software that is human readable. Source code must be parsed and converted to **machine language** before it can be executed on a computer.

Variable - a named location in software that holds data so that programs or algorithms can access and change that data.

Chapter 2 Exercises:

1. What is more readable by a computer, **assembly** or **machine language**? Which is more readable for people?
2. Using the MOVE instruction, write a program in **assembly** language that will assign the value one (1) to 'Var1'.
3. Using the MOVE and ADD operations, write a program in assembly language using the ADD instruction to increase the variable 'Val1' by the variable 'Val2'. Include the declaration and initialization of the variables in **assembly**.

4. Using the MOVE, ADD, COMPARE and BRANCH operations, write a program in **assembly language** that will loop, copying 200 bytes of data beginning at **memory** location 0x4000 to the **memory** location 0x200000. Hint: Access the value at a **memory** location directly with the absolute value (#) operator, instead of using a register to hold the address, for example [#0x4000] and [#0x200000].

Chapter 3: Compiled Systems

3.1 Origin of the C Language

Computers do not understand math equations, only binary operations within their CPU instruction set. With each processor using a different instruction set, assembly language programs are difficult to move to new computer systems. The C language was created so a common language, independent of any CPU, could be used to write software for communicating with hardware.

The C language has become the most common language used for system software because it is independent from the hardware, yet low level and flexible enough to interface with any hardware. The **C compiler** reads a source file (with the `.c` extension), parses the words and numbers into tokens, converts **expressions** and then **assignments**, **conditionals**, and **loops** into operations and then creates a corresponding assembly language file (with the `.s` extension). A **C compiler** is specific to a particular CPU type and converts C source code into assembly language. Then the assembler is used to convert the `.s` file to binary machine language.

The output depends upon the specific **C compiler** used as each is specific to the CPU type. The assembler assembles the assembly language, created by the **C compiler**, into a machine language or binary output (`.o`) file. These `.o` files are referred to as object code and are specific to the hardware the **compiler** created it for. The idea of the C language was to allow the reuse of systems software (written in the C language) across different CPU systems with only a **recompile** of the source code required. A small change to the base address of the peripheral address space, and a CPU specific **compiler** are usually all that is required to move a well designed peripheral software driver to a different CPU.

If there are errors in the C source file, the **compiler** reports the errors and does not attempt to create assembly code. If there are warnings during **compilation** the warnings are reported and assembly code is created based on **compiler** assumptions for those warnings. The **C linker** is used as the last step after assembling and combines all assembled output files (`.o` files) created by the assembler into a single binary file,

ready for execution by the computer. The combination of the **compiler**, assembler and the **linker** allow great flexibility when creating software.

Compiled languages, and C in particular, are responsible for the emergence of the computer age. Hardware and software change, but the C language allows hardware and software to change simultaneously without interfering with each other. Some may argue this point, but time will show that it was the C language, and very little else, that emboldened the Internet age. The C language allowed portability, so engineers can write software that can be **compiled** for multiple hardware platforms. System engineers could then develop and reuse system software optimized for a variety of hardware, allowing the creation of all of the underlying technologies that accelerated the PC age.

With a few exceptions where assembly language is used, the remainder of this book will focus on using the C language to create system software. Unless the goal is to understand a specific CPU instruction set, assembly language should be avoided when writing system software, the disadvantages otherwise are enormous. Every different computer uses a different instruction set, which may require a complete rewrite of the assembly language for the software to **run** on different hardware. Also assembly is very low level and thus has poor readability with more complex algorithms. The last and most important reason to not use assembly is that it is lengthy/tedious, complex and prone to human error. This said, it is impossible for the system engineer to avoid assembly completely and understanding how the computer computes is vital to creating system software that optimizes use of the hardware. In general, assembly is best used for specific operations related to hardware initialization or optimization (to speed it up). The C language is best for general system software development as well those software applications that require or desire speed of execution.

3.2 C Language Basics

It is essential to understand the language, or syntax, of C in order to create a program that can change computer hardware. The structure and words of a language are known as the syntax, and describe the rules of the language. We previously discussed how assembly language instructions are categorized into compare, branch and computational. Similar categories exist for the C language. With C, the corresponding terminology used is **conditional**, **loop** and **assignment**.

The following is not a comprehensive introduction to the C language. Only the

minimal amount of C language rules and syntax needed for each chapter is presented. This introduction to C is from the perspective of controlling hardware with software, and this is not the only use of the C language. The C language is organized as **statements** that are **compiled** to execute left to right and top down, like reading a book. Each **statement** consists of a series of words, or tokens, separated by any amount of white space and concluded with a semicolon. Brackets, '{' and '}', are used to enclose groups or blocks of code as a single **statement**. Parenthesis, '(', ')', are used to order the variables and constants when evaluating or calculating an **expression**.

This order of evaluation/compute is known as precedence to software professionals. Every **expression** in parenthesis is evaluated inside out so the inner most **expression** in parenthesis is calculated first. For example, $(1 + 3 * 2)$ equals 7 as multiplication (*) has a higher precedence than addition (+). To understand and use parenthesis to control the precedence is required if the above **expression** wishes to evaluate the addition first, for example $((1 + 3) * 2) = (4 * 2) = 8$. It is important to remember that calculations in **expressions** are performed inside out with regard to parenthesis and that parenthesis can be used to control the order of evaluation or precedence.

A **conditional statement** in C is defined with the syntax 'if (expression) statement; else statement;'. An 'if' **statement** evaluates, or calculates, the **expression** and if the **expression** is not zero, the **statement** below the 'if' , or code block if in brackets, is executed next. If the **expression** is zero, the code below the 'else' **statement** (if present) is executed.

This 'if' syntax is more readable if split up onto different lines in the source code file. The example below demonstrates how 'if' **statements** can be stacked to check multiple **expressions** with the else phrase. The 'else' is always optional and you can also use 'else if' **statements** below the original 'if'. This allows the stacking of many 'else if' **statements** between the original 'if' and the final 'else'. Here is an example of pseudo code showing usage of the 'if' **statement**.

```
if (expression1)
    statement1;
else if (expression2)
    statement2;
else
    statement3;
```

A **loop** in C is defined with 'for (assignment; expression; assignment) statement;'.

This 'for' **loop** continues **looping** forever until the **expression** equals false or zero. The first **assignment** is executed only once, before the **loop** is entered and is often used to initialize a variable. The second **assignment** is executed every **loop**, before evaluating the **expression**. In a 'for' **loop**, any of the **assignment** or **expressions** can be omitted if not needed, but the semicolon's are required.

```
for (assign_once; expression; assign_every_loop)
    statement;
```

Using comments to describe C source code is very important to creating readable and reusable source code. The C language allows you to define comments in two ways. One way is with the double slash `'//'` and this keyword tells the **compiler** that the remainder of the line is a comment. The second type of comment is the comment block which can encompass multiple lines. The comment block is begun with the slash star characters and all text between this keyword and the ending comment block keyword of star slash. Let us review the generic example above with multiple **statements** within brackets and a comment added to the for **loop** indicating that since it has no **expression** it will **loop** forever.

```
if (expression1)
{
    /* The good. */
    statement1;
    statement2;
}
else if (expression2)
{
    /* The bad. */
    statement3;
}

/* The ugly. */
else
    statement4;

/* Loop forever. */
```

```
for (;;)
{
    statement1;
    statement2;
}
```

The syntax for an **assignment statement** is ‘variable = expression;’. Variables must be **declared** with a data **type** before use and the variable **types** used in an **expression** must match. Common integer data variable **type** keywords in C are ‘long’, ‘int’, ‘short’ and ‘char’. Each of these keywords can be preceded with ‘unsigned’ if an unsigned version of this integer **type** is desired, for example, ‘unsigned short’, and ‘unsigned char’. This ‘unsigned’ modifier does not change the number of bits used to represent the integer variable, only the interpretation of those bits by the **compiler** when creating assembly language.

3.3 C Language Data Types and Sizes

Depending on the register and CPU bit size (8, 16, 32, or 64 bit), the C **compiler** may use different sizes for ‘int’, ‘short’ and ‘char’. Since the size of the data **type** is commonly of interest to the system engineer, it is prudent to create **types** that allow better software control of a variables size. For this text a signed 8 bit integer value will be defined to be ‘i8’, or (i)nteger with (8) eight bits, while an unsigned 8 bit integer value will be defined as ‘u8’, the same for 16 and 32 bit numbers. This naming convention makes it very easy to know the sign and size of the integer it represents. The C language allows the creation of new **types** with the **type define** command, or ‘typedef’.

Below are an example of **type definitions** for signed and unsigned integers of common sizes on a 32 bit processor. The semicolon on the end indicates the end of each **type definition statement**.

```
typedef long long      i64;
typedef unsigned long long u64;
typedef int            i32;
typedef unsigned int   u32;
typedef short         i16;
typedef unsigned short u16;
typedef char          i8;
typedef unsigned char  u8;
```

Each ‘typedef’ could also be expressed with what is known as a pound define, or ‘#define’. A ‘#define’ defines a keyword that will be replaced by the **compiler** with another keyword or words. When compiling, the **C compiler** makes many ‘passes’ of the source code, morphing the code at each pass closer to assembly language. Typically the first pass for a **compiler** finds all keywords that have been ‘#defined’ and replaces them with the **definition**. For example all ‘u32’ keywords found in the source code will be replaced with ‘unsigned int’. Below are the equivalent **type definitions** above, but ‘#defined’ instead.

```
#define i64      long long
#define u64      unsigned long long
#define i32      int
#define u32      unsigned int
#define i16      short
#define u16      unsigned short
#define i8       char
#define u8       unsigned char
```

Congratulations, we just wrote our first **C statements!** If **compiled** these **definitions** or **type definitions** would create no executable binary code, but it would **compile**.

But why do we need to do this, why not use the standard **C types** ‘int’ and ‘char’ and ‘short’? The standard **types** can have different sizes when **compiled** on different systems. For example, a 16 bit CPU might use a 16 bit ‘int’, while compilers for 32 bit CPU’s would need to **declare** ‘short’ (instead of ‘int’) to get a 16 bit number. Using universal **type definitions** (or #defines) allows the **C software** to **compile** correctly on 8, 16, 32 and 64 bit CPU’s by simply ensuring these size based **type definitions** match the **C compiler** documentation for a particular CPU.

To understand better, remember that the **C compiler** creates assembly language so it must decide ahead of time whether ‘int’, ‘short’ and ‘char’ in C will represent 32, 16 and 8 bit numbers within the assembly language. The **C compiler** for a particular CPU may translate the C integer **types** into the registers and memory locations of whatever bit size the CPU desires. The system software creator must check the documentation for the CPU and change the ‘typedef’ or ‘#define’ of the sized C **types** so that they match. For most CPU’s of 32 and 64 bit, the examples shown will hold true.

Below is an example of C **type definitions** for the same sized signed and unsigned integers as above but for a 16 bit processor. Note that all is the same except ‘int’ which is now 16 bit as it is common for optimization that the ‘int’ size match the CPU bit size.

```
typedef long long      i64;
typedef unsigned long long u64;
typedef long           i32;
typedef unsigned long  u32;
typedef int            i16;
typedef unsigned int   u16;
typedef char           i8;
typedef unsigned char  u8;
```

The understanding and thoughtful use of known sized variables is a core requirement for system software to work across CPU’s of different bit sizes. The debate on whether it is best to use ‘#defines’ or **type definitions** boils down to personal preference usually. There should be no difference introduced into the **compiled** binary, but some **compilers** and existing source code bases are easier to work with one style or the other. If the project involving the systems software effort may require supporting multiple **compilers** as well as hardware systems, ‘#define’s are often easier to work with. To check that the **compiler** is doing the right thing, **build** two versions of the software (one with ‘#define’ and the other ‘typedef’) and compare the binaries. Most if not all current C **compilers** will generate identical code.

3.4 C Language Math and Bit Manipulation

Before we can operate hardware with software it is necessary to learn how to modify the individual bits of an integer data **type** in C. To create a series of binary **expressions**

using integer numbers requires adding individual bits together using an unsigned integer **expression** in C. This is the mathematical series representation of the binary number. To create this **expression** we need to understand how to represent the value of an individual bit of a binary integer. The C language allows you to do this easily with a concept called shifting.

Each bit can be represented in C by shifting left (\ll) a one (1) by the number of the bit offset. Shifting a bit left represents multiplying by a power of two. For example, the **expression** $(1 \ll 0)$ is a binary one shifted zero bits to the left, which is equal to 2^0 or 1. The **expression** $(1 \ll 5)$ is a binary one shifted five bits to the left, which is equal to 2^5 , 100000 binary, 32 integer, and 0x20 hexadecimal. To summarize, zero is the first binary digit and if set it equals one, meaning $(1 \ll 0)$ is 1, $(1 \ll 1)$ is 2, $(1 \ll 2)$ is 4, etc., each increment a power of two. System software creators use this shifting method to access and modify single bits when communicating with hardware registers. Shifting can also be performed to the right with the (\gg) **expression**, which divides an integer by a power of two.

Fig 10: Binary represented as C series expression

Binary	Integer	C Expression
0000	0	0
0001	1	$(1 \ll 0)$
0010	2	$(1 \ll 1)$
0011	3	$(1 \ll 1) + (1 \ll 0)$
0100	4	$(1 \ll 2)$
0101	5	$(1 \ll 2) + (1 \ll 0)$
0110	6	$(1 \ll 2) + (1 \ll 1)$
0111	7	$(1 \ll 2) + (1 \ll 1) + (1 \ll 0)$
...	...	
1111	15	$(1 \ll 3) + (1 \ll 2) + (1 \ll 1) + (1 \ll 0)$

Another binary operation involving a single binary number is the ones complement, which “flips” every bit in a large binary number. For example, all ones (1) become zeros (0) and zeros (0) becomes ones (1) for every bit in the large binary number. The ones complement is represented in the C language with the tilde ‘ \sim ’ character.

The remaining bit operations involve performing bit operations upon two large binary numbers, creating a result dependent upon the operator. The operators ‘And’, ‘Or’, and ‘Xor’ (exclusive Or) are represented in C with ampersand (&), bar (|) and caret

(^) characters respectfully within an **expression**. To recap from chapter one, these operations compare two large binary numbers bit by bit. The first two operators mentioned are somewhat easy to explain. For ‘And’ (&), if both corresponding bits of the two large binary numbers are one (1), then the result is one (1) for that bit, otherwise the result is zero (0) for that bit. For example A ‘And’ (&) B equals C, where C has only bits set to one (1) if the corresponding bit in both A and B are set to one (1). ‘Or’ (|) is the same thing but the result is set to one (1) if either the corresponding bit in A ‘Or’ (|) B is set to one (1). For Xor, or exclusive Or (^), the result is set to one (1) for every corresponding bit whenever the bit in A is different from that in B. Below is the same example from above but using or (|) instead of addition (+). Or (|) can also be used in place of addition when all the bits are independent between the two values.

Fig 11: Binary represented as C Or expression

Binary	Integer	C Expression
0000	0	0
0001	1	(1 << 0)
0010	2	(1 << 1)
0011	3	(1 << 1) (1 << 0)
0100	4	(1 << 2)
0101	5	(1 << 2) (1 << 0)
0110	6	(1 << 2) (1 << 1)
0111	7	(1 << 2) (1 << 1) (1 << 0)
...
1111	15	(1 << 3) (1 << 2) (1 << 1) (1 << 0)

At the end of Chapter 1 we described how a computer does addition using binary operations. Let us rewrite that equation using C syntax as ‘a + b = ((a & b) << 1) | (a ^ b)’. Remember again that an **assignment statement** in C uses the equal sign (=) to assign a variable to a value. First it is required to **declare** a variable and then that variable can be assigned a value. Let us look at a simpler example.

```
u8 data;
```

```
data = (1 << 0);
```

This example **declares** an 8 bit unsigned integer named ‘data’ and then sets the first bit, bit 0, to 1. If later in the program other bits need to be set while keeping the

original bits unchanged, the C “or equal” (`|=`) **assignment statement** can and is often used. The “or equal” performs a binary ‘Or’ between both sides of the assignment, turning on, or enabling, new bits of a variable, without turning off any bits already set. It is equivalent to using the equal sign and including the variable name on the right, for example ‘`data |= (1 << 0)`’ is equivalent to ‘`data = data | (1 << 0)`’. Here is the next example that will set bit 3 in ‘`data`’, without changing any other bits.

```
data |= (1 << 3);
```

Oftentimes it is required to set a bit to zero that was previously one, or whose value is unknown. This can be done using the ‘and equal’ (`&=`) **assignment statement**. The ‘and equal’ **assignment** sets each bit only if the bit is currently set and also included in the new value, otherwise the bit is cleared, or set to zero. In other words, the variable is bit wise ANDed with the assigned value (**expression** right of `&=`). To turn off specific bits in a variable, this ‘and equal’ **assignment** is used in conjunction with the ones complement operation (`~`). The ones complement operator (`~`) flips every bit of the **expression**, so `0x00` becomes, `0xFF`, etc. Let us show an example of identical **expressions** that will turn off bit 3 in ‘`data`’ above.

```
data &= ~(1 << 3);
data = data & (~8)
data = data & 0xFFFFFFFF7
```

The ones complement sets all bits to one (1) except bit 3 (`1 << 3`) that we wish to remove. Then this value is ANDed with the existing ‘`data`’ value, to effectively turn off bit 3 while leaving the other bits unchanged. Let us review this binary operation above with a table, assuming that `data` has an original value of `00001001` or 9, meaning bit 3 is set as well as bit zero.

Fig 12: C expression represented as binary in little endian format

Variable or Expression	Binary Value
<code>(1 << 3)</code>	00001000
<code>~(1 << 3)</code>	11110111
<code>(data)</code>	00001001
<code>(data & ~(1 << 3))</code>	00000001

After the **assignment**, the variable 'data' has a value of 1 (binary 00000001), so this **expression** only cleared the single bit 3, leaving all other bits of the variable 'data' unchanged.

3.5 C Language Functions

As code grows it can benefit from thoughtful organization. C programs are organized into **functions**. **Functions**, sometimes referred to as procedures, are **declared** with a name and consist of a group of **statements** that performs a specific task or computation. All code in C must exist within a **function** and the first **function** to be executed in a program or software is often named `main()`. **Functions** can be passed variables, at which point these variables are described as **parameters** to that **function**. To reiterate, **functions** are passed an optional list of operands, or **parameters**, and then the **statements** within the **function** perform a specific task or computation on or with those **parameters**, thereafter optionally returning a value. **Functions** can be called from other **functions** and can return a value. Let us put together the code to turn on bits 0 and 3 into a function.

```
u32 set_bits(u32 data)
{
    data |= (1 << 0);
    data |= (1 << 3);

    return data;
}
```

The first `u32`, before the **function** name `set_bits()`, is the **return value type**, and the declaration in parenthesis is the **parameter** to the **function**. This function sets bit 0 and 3 of the value in the **parameter** 'data', and then returns the new value. The C language allows many ways to do the same thing. For example, the function below computes the identical result to the one above.

```
u32 set_bits(u32 data)
{
    data |= (1 << 0) | (1 << 3);

    return data;
}
```

Which computes and returns the identical result to the function below.

```
u32 set_bits(u32 data)
{
    return data | (1 << 0) | (1 << 3);
}
```

To use this function to add these bits to a value would look like this.

```
data = set_bits(data);
```

3.6 C Language Organization

This is a good time to pause and conclude on the organizational structure of the C source code we will create, starting with file naming and usage. C source code consists of header and source files. Header files in C are identified with the '.h' file extension, while source files in C are identified with the '.c' extension. Source code is not typically defined within header (.h) files, only **declarations** and **#defines**. Anything **defined** or **declared** in a header file can also be done so privately at the top of a source file (.c). Therefore header files (.h) should only be used for **definitions** and **declarations** that must be shared across different source files (.c). To summarize, all variable and **type declarations** should go at the top of the source (.c) file. Only when there is a clear need to share a **definition** and/or **declaration** with multiple source (.c) files should a header (.h) file be used.

One consistent challenge with creating software with C is organization, as the software developer is responsible for everything. However, if some simple rules are followed, most common organization problems can be avoided. First, any **functions**

that are only needed within a single source file (**private functions**) should be located at the top of the C file before their usage and **declared** with the 'static' keyword.

If a function requires calling a private static **function** in another source file, first consider moving this **function** to the other source file so the **function** can access the private static **function**. If this is not possible or creates other problems, the **function declaration** must have a prototype of the **function declared** in a C language header file, or .h file. **Functions declared** in a header file are called global **functions**. Only create global **functions** when necessary, since header file management becomes an issue as the system grows. Only **functions** required by multiple source files should be made global and the **function declared** in a header file.

It is always best practice to minimizing the number of required header files so it is easier to migrate the system source code to other systems. Group functionality into header files that can be shared by any number of underlying C source files. With this in mind, let us create our first header file that includes the **type definitions declared** in section 3.3. These common **types** can then be used by all source files we create, so let's name this header file 'system.h'.

3.7 The C main() function

The C main() function is a special **function** that is the entry point into a software program. To clarify, the main() **function** is the first **function** that is executed by a program when it starts **running**. Let us create a simple main() program, one that **loops** forever, adding one to a variable every **loop**. On Windows development systems, Notepad can be used to open, edit and create source files. On Linux, use the editor for your windowing system, such as gedit for Gnome, or kedit for KDE. If the user is in a command line only environment, 'pico' is the simplest editor to use, for example 'pico main.c'.

```
/*.....*/
/*      main: Application Entry Point      */
/*                                          */
/*      Returns: Exit error                */
/*.....*/
int main(void)
{
    int loop_count = 0;

    /* Loop forever. */
    for (;;)
        loop_count = loop_count + 1;
}
```

Please create this main.c file, copy the above source code to the file and save it.

3.8 C Language Variable Scope and Volatility

The last concept for this chapter is **variable scope**. There are important differences in the locations where variables and functions are **declared** that determine the scope or persistence of a variable, or **function**. **Variable scope** can also be described as where in a program a variable exists. Where a variable is **declared** in a source file determines its scope, or accessibility by other portions of the program. Global variables are variables **declared** outside of any **function** and can be accessible by any **function** or other portion of the C file that follows the **declaration**. Variables **declared** within **functions** are only accessible within that **function**, meaning variables needed by many different **functions** must be global, or passed as **parameters** to the **functions** that need access to their values. It is considered best practice to minimize the use of global variables and use **function parameters** instead.

So **compiled** systems use **functions** for organization, manipulate variables by **assigning** them to **expressions**, and use **loops** and **conditional statements** to modify the flow of execution of the program. This is a big step from assembly language, but still close enough to the way a computer operates to be used to directly control hardware resources. One missing piece is how to represent a location in the hardware memory map of a hardware register with a C variable? For example, how can we

tell the **compiler** that a memory location contains a value that can simultaneously be changed by physical hardware?

Well the C language has the 'volatile' keyword which should be used for **declaring** variable that can be changed by hardware. Volatile means the variable can be changed outside of the immediate program or software under execution, for example, by the hardware. Volatile variables are required when a peripheral in hardware is accessed directly by the software interface. The volatile keyword lets the **compiler** know that hardware can change the variable without software knowing. This keyword ensures that the value in memory is not cached in software (in a hardware register for example) in between usage, as is common during **compiler** optimizations. Here is an example of the **declaration** of a pointer to a volatile variable of **type** 'unsigned integer'.

```
volatile u32 data;
```

Chapter 3 Glossary:

Assignment – A source code **statement** that assigns the value of a variable to an **expression**.

Build – the act of creating software using **compiler** tools such as a **compiler**, assembler and **linker**.

C – A human readable source code language used for software development. Designed for developing software to control hardware, as is common in system engineering.

Compiler - A software program that can parse C source code files and convert them into assembly language. **Compiling** is the act of executing a **compiler** with a source code (.c) file as input and assembly (or binary) being the output.

Conditional – a type of **statement** that performs a comparison of values, branching the execution order depending on the comparison result.

Declare – to define a variable with a specific type. Variables must be **declared** before than can be used in **C statements**.

Definition – a #define, whether for a type or function. Function #defines are also called macros.

Expression – a portion of a **statement** that defines the mathematical equation to use for **assignment** or comparison purposes. Used to assign a new value or within a **conditional statement** to determine whether to branch or not.

Function – a group of **statements** that performs a specific computation or task. A **function** can be passed variables, known as parameters and can also return a value to the caller. **Declared** with a name, return type and parameter list, **functions** can be used within **conditional** and **assignment statements**.

Linker - a software program that converts one or more machine language output files into a single executable binary file.

Loop – a series of **statements** that are repeated until a **conditional statement** breaks execution out of the **loop**.

Parameter – a variable that is passed to a **function**. **Functions** define the required **parameters** which the caller must provide.

Return Value – a value returned by a **function** after execution is complete. The **return value** is passed back to the caller of the **function**.

Run - to execute a program or software in binary machine language format.

Statement – a complete line of source code, whether **assignment**, **conditional** or **loop**. **C statements** end with a semicolon (;) or an end bracket (}). **Statements** are similar to CPU operations in that they are ordered, but a single **statement** can be much more complex, **compiling** into many CPU operations.

Type – A particular representation of a variable, such as an integer. All variables are binary data, but the **type** signifies to the **compiler** the size and how the binary data is interpreted.

Type Definition – A language dependent **definition** of a new **type** of variable. A new **type** must first be defined before it can be used in a declaration **statement**. Represented in the C language with the keyword 'typedef', or as a #define to a previously defined **type**.

Variable Scope – the position within a source file or **function** that a variable was **declared**. Variables exist only in the scope they are **declared**, so variables declared in a **function** exist only for that **function** and are called local variables. Variables **declared** outside of **functions** are global variables and can be accessed in any **function**.

Chapter 3 Exercises:

1. In your own words, what is the difference between a variable and a **parameter**?
2. In your own words, what is the difference between an **expression** and a declaration?
3. Write a C **expression** that assigns the binary value 0101 to the variable “var”.
4. Write a C **function** that takes one **parameter** and returns zero or one. The **function** must check if bit 0 of the **parameter** is set, returning one (1) if set, or zero (0) if the bit is not set. Hint: this will require use of the C **conditional statement** ‘if’. Name your **function** and variables whatever you please.
5. Write a C **function** that takes no **parameters** and returns zero or one. The **function** loops 10000 times, or until bit 0 of a volatile global variable is set. The **function** must return one (1) if the bit is set, or zero (0) if the bit was not set once after 10000 tries. Hint: this will require use of the C **conditional statement** ‘if’ and the C **loop statement** ‘for’. Name your **function** and variables whatever you please.

Chapter 4: System Architecture

4.1 Address Space and Software Memory Map

Computer systems have an **address space**, which is a range of accessible memory **addresses**. Within this **address space** exists the system **memory map**. There are generally two types of **memory maps** per system, sharing the same **address space**. These are the software **memory map** and the hardware **memory map**. The software **memory map** defines how the compiled software is structured internally as a binary executable. The hardware **memory map** defines how peripheral **address spaces** can be accessed with software. Let us take a look at an example linker script that defines a software **memory map**. Note that both the ORIGIN field identifies the start of the software executable and LENGTH is the length of available RAM. Both depend on the specific hardware the software will execute on.

```
MEMORY
{
    ram : ORIGIN = 0x0, LENGTH = 0x1000000
}

SECTIONS
{
    .text : { *(.text*) } > ram
    .bss : { *(.bss*) } > ram
    .rodata : { *(.rodata*) } > ram
    .data : { *(.data*) } > ram
}
```

This linker script defines a software **memory map** entirely within RAM, including the origin, or beginning, as well as the length of available RAM for the software program. Additionally, the linker script allows the program to define the order in the software **memory map** for the executable code (.text), uninitialized global variables (.bss), read

only variables (.rodata) as well as initialized global variables (.data). In this example, the .text, .bss, .rodata and .data are all placed in the defined RAM region starting at 0x0000, one section after another as defined in the order of SECTIONS.

In other words, the .text is placed in RAM beginning at location 0x0000 and the other software portions (.bss, .rodata and .data) are located immediately following, in the respective order defined in the linker script. The highest valid **address** of RAM is defined as LENGTH + ORIGIN. If there is not enough available RAM defined in the **memory map** for the compiled executable, there should be a link error. Let us save the software **memory map** linker script above to a file named 'memory.map' for use when linking the compiled code of our first software program in the next laboratory assignment.

4.2 Memory Address and Pointers

Computers have a CPU that can perform operations upon data in registers or RAM. Registers are limited, the exact number dependent upon the CPU. RAM exists as a consecutive region of memory **addresses** within the **memory map**. Most peripherals have registers for status and control that are **mapped** in hardware to a different location within the **address space**. These locations are defined in the hardware **memory map**.

The C language is ideal for communicating with the status and control registers of peripherals within the hardware **memory map** because of C's ability to **reference** any memory location using a **pointer**. A **pointer** is assigned the **address** in memory of a particular data type. This allows a **pointer** to be assigned to the **address** of a hardware register and then the register can be changed with a C assignment statement. **Pointers** are very powerful and dangerous as the entire **address space** is available to the C programmer, but only **addresses** in the **memory map** are valid.

This is too important not to repeat in a different way to ensure the reader understands. A **pointer** is a C language variable declaration that contains the **address** of a variable of that type located in RAM, not the variable itself. C requires **pointers** to be declared with a particular type so the compiler knows how to **reference** the underlying value being pointed too. A **pointer** to an integer is the location in the **address space** that holds an integer value. A **pointer** represents a location in the **address space**, so **referencing** the **pointer** is required to access the value stored at this memory location.

To **reference** a variable, C uses the asterisk ‘*’ symbol. Add this symbol before a **pointer** variable and the value of that type stored at that memory location is the result of the expression. Otherwise the **address** of the location in memory is the normal result when using a **pointer** in an expression. Also, the symbol ‘&’ can be used to get the **address** of a variable, allowing any normal variable to be converted to a **pointer**. **Pointers** need to be used with caution as it is easy to make a mistake when using them and do something unintended. However, **pointers** are absolutely essential to controlling peripherals from software, as the registers for status and control of hardware are typically **referenced** from somewhere within the **memory map**. From a historical perspective, this necessity to **reference** peripheral registers/memory was the primary reason for the **pointer** data type. Let us review a quick example function that uses pointers, with comments. Remember that comments are important to creating readable code and start with */** and end with **/*.

```
int doubled(int value)
{
    int *ptr_value;

    /* assign ptr_value to address of "value" */
    ptr_value = &value;

    /* double the value */
    value = *ptr_value + value;

    /* return the value doubled */
    return value;
}
```

4.3 Using C Pointers with Peripheral Registers

The **address** of the peripheral registers defined in the hardware **memory map** can be assigned to **pointers** in C and directly accessed (**referenced**) from C programs. There are some precautions needed when accessing hardware with software, the most important is to be sure to use the volatile keyword. Volatile is needed because these are hardware registers and can be changed by the hardware at any time without software

knowing. The `volatile` keyword ensures that the compiled C program **references** the **address** location of the **pointer** every time, and never **caches** the value in a register, as the compiler will often try to do when optimizing operations with a nonvolatile variable.

The below picture is an example hardware **memory map** to help visualize the concept of both hardware and software memory.

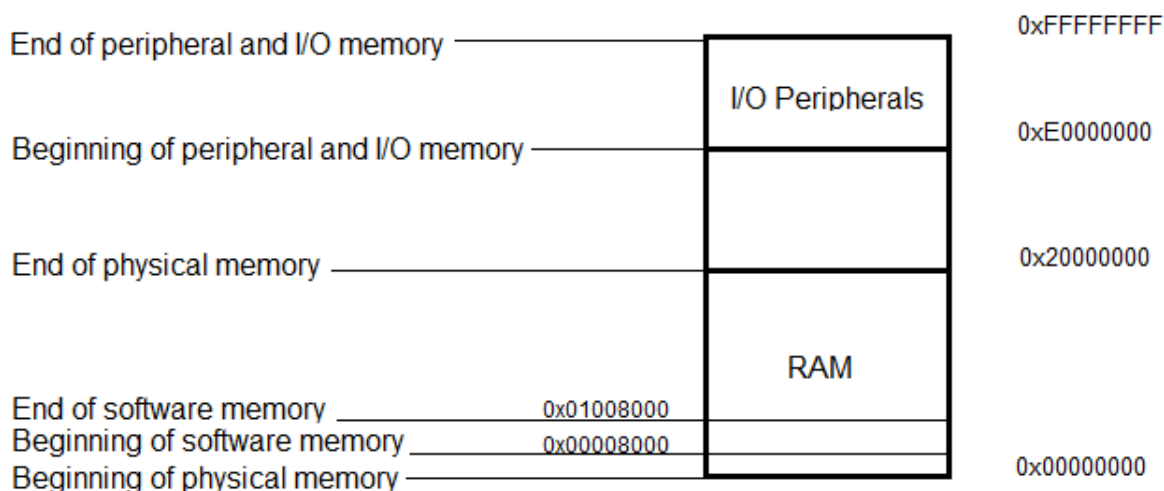


Fig 13: The hardware memory map

Some hardware registers in the **memory map** can only be accessed (read only), while other registers can only be assigned (written), while others allow both read and write access. It is required to consult documentation for the motherboard and peripherals to determine the overall **memory map** as well as the detailed registers for each peripheral. Combining information from multiple documents to correctly operate the hardware through the use the registers in the **memory map** is the first step to writing system software for a hardware platform. The hardware and peripherals must first be documented, and the documentation understood, before system software can be written.

As described above, it will be required for system software to access a hardware register in the **memory map**. There are many ways to accomplish this but it is best that the system software provide a consistent way to do this. Creating a `#define` function, commonly referred to as a **macro** definition, will aid readability and avoid mistakes accessing and assigning these hardware **pointers**. Using `#define` functions, or **macro**'s, will also allow the system source code to be easily ported from one system to another.

```
/*  
 * Register manipulation macros  
*/  
#define REG8(address) (*(volatile u8*)(address))  
#define REG16(address) (*(volatile u16*)(address))  
#define REG32(address) (*(volatile u32*)(address))
```

The `REG32()` **macro** is defined to access 32 bit hardware registers, which are by definition volatile as they are controlled by hardware. This **macro** can also be used to assign a 32 bit hardware register in the **memory** map to a new value. However, this **macro** should never be used in a hybrid assignment such as `'|=', '&=',` etc. This is because the compiler can often get sloppy in how it performs these operations and has been known to change bits temporarily before arriving at the final result, which can be catastrophic when these temporary bit changes control the hardware in unforeseen ways.

Defining the `REG32()` and similar **macros** helps the systems engineer to remember this volatile requirement, but not the assignment requirement of no hybrid assignments. To mitigate this issue many OS software packages include functions such as `iowrite32()` that enforce two parameters and thus do not allow hybrid assignments. Compiler optimizations have continuously caused problems for system engineers, leading to the development of the high level functions/**macros** such as `iowrite32()`. However, `REG32()` is a primitive lower than `iowrite32()`, allowing more freedom and flexibility. The goal of this book is to inform the reader, not hide the details. The freedom of `REG32()` comes with the requirement that the software creator understand that the assignment of hardware **mapped** registers must be precise.

4.4 Create Software with an Editor and Compiler

The system engineer frequently uses software tools in order to create system software. The most frequently used tool is a source code editor, while the most important tool of the system engineer is the compiler. The linker is the command line tool needed to create the final binary software program by combining (linking) all the output (`.o`) files generated by the compiler into a single executable. Let us discuss these software tools in more detail now.

The source code editor is used to read, modify and create source code. While any text editor can be used to create source code, many are designed for the C programming language and have user friendly color schemes. From a command line, the simplest editor is pico or nano. The author frequently uses Gvim, a GUI interface based on vim, a classic command line editor. Vim allows custom color schemes for displaying C source code and file search tools can be integrated into it. Other common editors are Visual Studio, Eclipse, Emacs, Geany and many others. System engineers spend the majority of their time using editors, but other tools are required behind the scenes. It is important for readers to use a source code editor that is simple and familiar to the user in order to ease the initial learning curve. A source code editor must save the text in raw text (ASCII or UTF-8) format or the compiler cannot read it.

To compile a file, invoke the compiler at the command line with the file to compile as an argument to the command. Typically 'cc' is the C language compiler command line tool but this can vary by development and whether or not a cross compiler is being used to compile a source code for hardware different than the development system. Let us review a simple command to compile the file main.c into the output file main.o, searching for any #include files '-I' in the current '.' directory.

```
cc -I. -o main.o main.c
```

4.5 Creating an Executable with the Linker

The compiler is the bedrock of systems engineering. It is vital to the success of any project that the compiler be tested and verified for the hardware system it is to be used for. Compilers execute from the command line or system shell, accepting a list of options followed by the files to compile. Each source code file is compiled into an associated binary output (.o) file. After all files are compiled they can be linked together into a single executable binary file. The **linking** command provides system options and accepts a software **memory map** file to use when creating the layout of the binary software.

Every program typically has one function named main(), which the linker uses by default as the beginning of the program execution. It is important to remember that when main() finishes, the program will end and the hardware will halt. Programs require loops if they are to service peripherals, provide a user interface, or otherwise need to wait for data or an event.

The commands and options for compiling and linking can vary from one tool set to another, as well as from one system to another. Historically the C compiler command is “cc”, while the linker command is “ld”. Compiler and linker commands can be executed from the command line, such as a Terminal window in Linux or the Command Prompt in Windows. Here are two commands, including options, which compile the file “main.c” and then link “main.o” into an Executable and Linkable Format, or .elf file.

```
cc -I. -o main.o main.c
ld -T memory.map -o led.elf main.o
```

The -I option is used to include a path to search for header files, the dot meaning to look in the current directory. The -o option to “cc” indicates the output file, in this case, main.o and the last parameter is the input file “main.c”. The “ld” command links output files into Executable and Linkable Format (.elf), the -T option instructs the linker to create an executable according to the **memory map** defined in file “memory.map”.

While ELF is a standard and convenient way to format executables, CPU’s cannot execute this format directly. Software boot loaders and debuggers often parse the .elf file and move the binary regions to RAM for execution. It is not necessary to know the details of ELF format beyond that it is the standard linker output file and required to be used for disassembly with ‘objdump’. A tool to convert the ELF to a raw executable binary is often required before the application is ready for execution on a CPU. Most tool sets that include a C compiler include the “objcopy” or similar command to convert .elf into raw binary. The command line below shows how to convert an .elf file into the binary output file led.bin, ready for execution.

```
objcopy led.elf -O binary led.bin
```

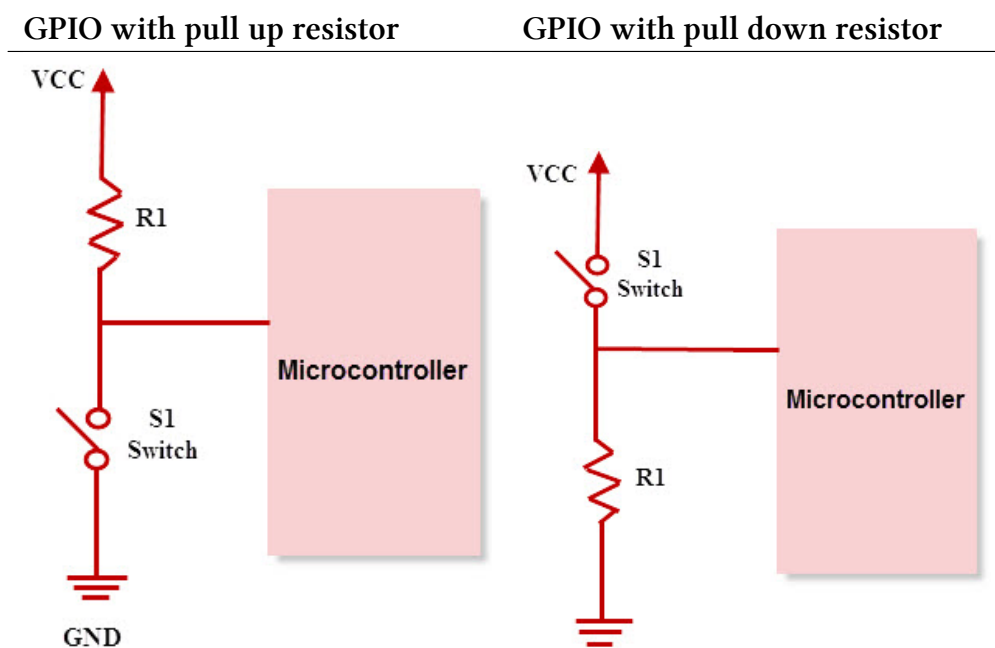
4.6 Configuring General Purpose Input Output (GPIO) pins

GPIO pins are simply electrical circuits that are connected to and controlled by a computer system. These GPIOs are typically of three different types, pull up, pull down or floating. A pull up, or pull high, type of GPIO puts a resistor between the

voltage and the **GPIO** pin, while the software controllable switch to set or clear the **GPIO** opens or closes the ground connection. A pull down, or pull low, type of **GPIO** puts a resistor between the ground source and the **GPIO** pin, while the software controllable switch to set or clear the **GPIO** opens or closes the voltage connection. The resistor limits the amount of current that can flow through the **GPIO** pin, either too the voltage source (pull up) or ground (pull down).

Each type of **GPIO** has its uses, typically a pull up is used for outputs such an LED, while pull downs are typically used for inputs. Floating **GPIOs** have no resistor in the circuit and can easily drift between high and low depending upon static discharges. Floating **GPIOs** are typically used for data transfer pins going to a peripheral data line, where the electrical activity occurring on the **GPIO** is controlled by that peripheral.

Fig 14: GPIO pull up/down resistor hardware configuration



From a software perspective the location of the switch in a pull up or pull down **GPIO** changes the way it should use the switch within that circuit in order to set the circuit high or low. For a pull up **GPIO** the switch connects to ground, so when the switch is closed (on) the **GPIO** pin will be grounded and driven low (0). For a pull down **GPIO**, the switch connects to the voltage source so when the switch is closed (on) the **GPIO** pin will be provided voltage and driven high (1). Regardless of the details, the software creator must understand that based on the type of **GPIO**, the internal circuitry and

usage change. This means that to change the **GPIO** type requires interfacing with the control registers for the **GPIO** pins in order to change this circuitry. The details are board specific and will be covered in the laboratory companion book.

4.7 Debugging

Besides the compiler, the most important tool for software creators is the **debugger**. Defects in software have been called bugs as a carry over from mechanical computers of the past. Thomas Edison is credited as the first documented use of the term “**bug**” to describe a technical malfunction in 1878. Grace Hopper is credited as giving the term popularity with software. When operators at Harvard Computation Laboratory repeatedly gave her the results of her software execution, the results were incorrect. She questioned the hardware while computer operators faulted her software. Eventually she convinced some of the operators that the software was correct and they double checked the hardware. Sure enough they eventually found a moth trapped in a mechanical relay, shorting it out and causing the incorrect computation.

A **bug** is defined as any unknown defect in the execution of a computer system or application. It might be a hardware bug but is most often a software bug. Once an unknown defect becomes known, it is no longer a **bug** but a known defect. The term **bug** should be reserved for unexpected or explained behavior. Missing features, known defects, or incompatibilities are not **bugs**. A faulty cable could be a **bug**, and even appear to be a software **bug**, at least until the problem is known. The term ‘**bug**’ has risen to popularity as it is a simple way to describe the unknown or unexplained.

Most **bugs** in computer systems are in software. No matter how experienced, software creators write software that often has problems or defects that cause it to not work as expected. Sometimes observing the behavior of the software and reviewing the written source code is enough to understand and fix the problem. Other times the problem is not obvious and to understand requires more visibility into how the software is behaving while it is running.

A **debugger** is a software tool that allows software creators to pause their software creations at run time and review the value of variables and other details to see if everything is working as expected. **Debuggers** typically allow a software creator to set a **breakpoint** at a location in the source code and then execute until it stops, or

breaks, at this point in the execution. A **breakpoint** is a point in the source code to break the execution for examination with a **debugger**.

Debuggers do not change the source code, even when setting a **breakpoint**. **Debuggers** communicate with the CPU while it is executing. To use a **debugger** effectively the source code is often compiled with a flag so that the resulting executable contains symbols for the **debugger**. These symbols allow the **debugger** to know which part of the software is executing so that when the the executable is paused, **the debugger** knows what line in the source code is being executed next. This allows the **debugger** to display the line of source code that is ready for execution and without symbols compiled into the executable, the **debugger** can only show the disassembled assembly language of the raw executable.

Debuggers, like compilers, are typically used to **debug** programs running on the same system they are created or develop on. For the purposes of systems engineering we need to debug a different, or **remote system**, than the one that runs the **debugger**. To clarify, we want to execute the **debugger** on the development PC in order to **debug** the **remote system**. This means that, like the compiler, we must have a special version of the **debugger** that is built to run on one system while **debugging** a different system. This special version of a **debugger** is also known as a cross compiled **debugger**. Like the compiler, a cross compiled **debugger** is one built to run and be used on the development PC while communicating with and **debugging** a potentially different **remote system**.

There are different types of remote **debugging** technologies, but one in wide use is JTAG. JTAG (developed by the Joint Test Action Group) is a serial bus technology that can be chained together to support any number of attached systems to a single JTAG bus controller. This makes it ideal to debug complex hardware systems that contain multiple CPUs, or even processors of different types. The details of JTAG are beyond the scope of this book and the particular JTAG controller hardware that is used to connect the development PC to the remote device to be **debugged** is specific to the development PC and the **remote system** being **debugged**. Details on the JTAG controller/adaptor such as how to connect it, configure and use it, etc. are discussed in the companion laboratory book.

To summarize, **debugging** is the process of pausing and resuming software execution in order to examine the software behavior at run time. **Debugging a remote system** is only possibly by using a physical connection (JTAG, etc.) to the development PC executing the **debugger** with the **remote system** being **debugged**. Only once the

development PC and **remote system** are connected through the JTAG controller interface can the **remote system** be **debugged**.

Chapter 4 Glossary:

Address - A generic term for a location in RAM. Can also be the location within the **address space** of a peripheral defined in the **memory map**.

Address Space - The range of valid memory locations usable by software, including any peripherals that software can control. Usually defined in the motherboard manufacturer **data sheet**.

Breakpoint - A location, or point, in the source code that a **debugger** will recognize and stop execution (break) so the software creator can examine the state of the software such as the value of variables.

Bug - Any unknown defect or problem in the execution of a computer system.

Cache - when a value of a variable in memory is moved to a register within the CPU to speed up one or more operations. By default the compiler **caches** variables in registers during compilation to speed up execution. The volatile keyword can be used when declaring a variable so the compiler knows not to do this and is required when C source code accesses a peripheral in the memory map.

Data Sheet - A document created by a manufacturer of a circuit board that describes the usage of that circuit board. This is a historic term and not used by all manufacturers, but the idea is the same. Without the **data sheet** naming the peripheral register **addresses** and system memory map, system software development cannot begin.

Debug - The act of examining a running system to determine the reason for unexpected behavior.

GPIO - General Purpose Input Output is a physical connection on a circuit board that has a binary state, that is, on or off. **GPIOs** can be turned on and off with software and are often used to enable peripherals, or for use by the system or user to control LED's, motors, or other hardware or peripherals.

Macro - A C #define that takes one or more parameters and evaluates an expression using those parameters. The C compiler replaces **macros**, substituting the **macro** with the defined expression during the first stage of compilation.

Mapped - A hardware peripheral that is represented within the Memory Map.

Memory Map - A table of where hardware and peripherals can be accessed by software within the **address space**. This layout of the **address space** is usually fixed, declaring the **address** range for RAM as well as the **address** range(s) to use for operating hardware peripherals.

Pointer - A programming language concept where a variable represents the **address** location of the data. A **pointer** allows a variable to be accessed and assigned by **referencing the pointer**. Using **pointers**, operational registers of peripherals that are in the **memory map** can be accessed, assigned and used in expressions within a C program.

Reference - the act of accessing or assigning a value stored at a **pointer** location. **Pointers** must be declared with a specific type before **referencing**.

Remote System - a physical computer system that is electronically connected to the development PC.

Chapter 4 Exercises:

1. What is the difference between a **pointer** and a **reference**?
2. What is the difference between a variable and a **pointer**?
3. What is the difference between the **address space** and the **memory map** of a system?
4. Draw the hardware **memory map** of a system with the following specifications: 1GB of RAM starting at physical location 0x10000000 and a peripheral **address space** of 256MB beginning at location 0x80000000. Hint, 1GB is 0x40000000 bytes and 256MB is 0x10000000 bytes.
5. In the above system, what **address** ranges are not valid? That is, what **address** ranges do not map into the **memory map**?
6. Author a linker script to describe the software **memory map** for this new system. It should start at the beginning of RAM and utilize all 1GB of RAM. This project also requires that the read only variables must be at the end of the software **memory map**.

Chapter 5: Timer Design

5.1 Hardware Clocks

CPU's operate at a certain **frequency**, or number of operations per second. One common way for a hardware system to keep track of time is to increment a counter every so many CPU clock ticks. If this counter value is saved, it can be compared to a future value to measure the change in time, or **time delta**. The equation to calculate this can be written as ' $\text{delta} = (\text{time_now} - \text{saved_time})$ ' and is equal to the number of clock **ticks** between the two time samples. If the **frequency** in seconds of the clock is known, converting the **time delta** to seconds can be performed with the C statement ' $\text{seconds} = \text{delta} / \text{frequency}$ '. Computer clocks can run at different speeds so a **microsecond** representation of time is a high level of precision that can represent any computer system and is commonly used when creating generic system software. **Microseconds** automatically allows more precision if the systems supports it, yet also works just as well with slower clocks.

Using **microseconds** for a clock means that the numbers get big very quickly, as one million is added to the clock every second. For example, a hardware register represented as an unsigned integer counting **microseconds** will reach the 32 bit size limit in 4,294 seconds, or every 71 minutes. To keep track after the 71 minute range, a system clock may have a second overflow counter that increments every time the main clock register overflows. This second clock register is referred to as the high register, as it contains the high order bits for the clock. The primary clock register is often referred to as the low register, as it contains the low order bits for the clock. If they are present in hardware, software can use both the high and low hardware clock registers (32 bit each) to represent any day or time. These 64 bits allow **microsecond** precision or less, depending on what the hardware supports and/or is configured for.

Systems have a hardware memory map with peripheral registers accessible with volatile pointers in software. System clocks may use a different speed than **microseconds** and if so this must be accounted for in system software. To convert to seconds, divide the clock **ticks** by the clock **frequency**. Clocks with a **frequency** less than **microseconds** are common and converting to **microseconds** involves dividing the

time value by a fraction. Dividing by a fraction is multiplication, so converting these clocks to the **microsecond** range is as simple as multiplying the number of **ticks** by the difference in frequency from microseconds.

5.2 System Software for Hardware Clocks

A true **microsecond** clock requires 64 bits to represent any day in the past and future. Some hardware supports two 32 bit clock registers, other hardware a single 64 register, and other hardware only a single 32 bit clock register. If only a single 32 bit clock register is available with a frequency of **microseconds**, software must manage the roll over. Let us now demonstrate how to do this in software.

Any software algorithm that must handle roll over must first be able to detect an overflow of the 32 bits register. With overflow properly supported in software, any period of time can be maintained. Let us start with a look at a typical **microsecond** clock representation within the hardware memory map. This single register interface to the timer is accessible as an address in the memory map. This software interface to hardware is a representation of a simple hardware clock, and the idea is the same whether this clock uses **microsecond** or less precision.

```
#define TIMER_CLOCK (TIMER_BASE | 0x00)
```

The software interface to the hardware clock has only this single `TIMER_CLOCK` to keep track of time changes or deltas. Since the clock is always increasing, we can detect a roll over of the clock if a read of the clock register results in a value less than the previously read value. If the timer algorithm compares the new value with the previous time value and it is less, then a roll over occurred. The first step to create the time interface is to define an expiration time.

For our timer design, what is needed is an unsigned 64 bit timeout value. For this discussion we will use **microsecond** precision, but this could be a translation to **microseconds** of any frequency defined by the hardware. In addition to the expiration time in clock ticks, we need to detect how many clock roll overs are required before the expiration time is achieved.

To avoid mistakes and make the code easier to read, let us define some additional constants in `system.h`. These defines will allow us to better use the timer functions as well as to visually use `TRUE` and `FALSE` instead of 1 and 0.

```

/*.....*/
/* Symbol Definitions */
/*.....*/
/*
** Common definitions
*/
#define TRUE          1
#define FALSE        0

/*
** Time definitions
*/
#define MICROS_PER_SECOND      1000000 /* Microseconds per second */
#define MICROS_PER_MILLISECOND 1000    /* Microseconds per millisecond */

```

5.3 Software Interface for Clocks

As discussed, a fundamental requirement for creating software for hardware is that the software needs to know how to accurately wait for the hardware. To accurately wait for any amount of time in **microseconds** requires a 64 bit variable as previously discussed. The goal is to create a **microsecond** sleep function that will wait for the requested number of **microseconds** before returning. There are three key requirements to waiting, to know what time it is now, to know how long to wait for, and a way to keep track of time.

The timer software interface this book proposes consists of two functions. The first is `TimerRegister()`, which requires as a parameter the number of **microseconds** in the future to expire, and returns a 'u64' representing this future time, relative to the current time. The second function to this software timer interface is `TimerRemaining()`, which takes as a parameter the expiration time returned in a previous call to `TimerRegister()`. `TimerRemaining()` compares the expiration time to the current time and returns the amount of time that the timer has left until expiration, or zero if it has already expired. To clarify, `TimerRegister()` takes the wait time and calculates and returns the system time that corresponds to this time in the future, based on the wait time.

The `usleep()` function, pronounced micro sleep based on the Latin letter mu or μ , takes a parameter of the timeout value in microseconds and passes this value to `TimerRegister()`. `TimerRegister()` then returns the system time of expiration which should be saved by the caller in a variable. Then a loop can check the previously returned expiration time with the system time with the `TimerRemaining()` function, until it returns a zero value indicating the current timer has expired. The `usleep()` function is a common system software function to sleep for the number of **microseconds** passed as a parameter. Note how the return value from `TimerRemaining()` is used to exit the loop once the remaining time is zero.

```

/*.....*/
/*    usleep: Wait or sleep for an amount of microseconds    */
/*                                          */
/*    Input: microseconds to sleep          */
/*.....*/
void usleep(u64 microseconds)
{
    struct timer tw;

    /* Create a timer that expires 'microseconds' from now. */
    tw = TimerRegister(microseconds);

    /* Loop checking the timer until it expires. */
    for (;TimerRemaining(&tw) > 0;)
        ; // Do nothing
}

```

This `usleep()` function depends on two timer functions, `TimerRegister()` and `TimeRemaining()`. The `TimerRegister()` calculates and returns the expiration time based on the time right now, plus the number of **microseconds** defined in the timeout parameter 'microseconds'. The `TimeRemaining()` function will compare the expiration time (returned from `TimerRegister()`) with the current time.

Let us create these functions now. The comments should help the reader understand the code. Please read the code top down, using the comment above each code line to help understand the code itself. One point of potential confusion with this algorithm should be briefly discussed ahead of reviewing the code. The hardware timer in this

example is a single 32 bit register that may roll over. How roll over is detected is with the `TimerRemaining()` function. `TimerRemaining()` must detect when the clock register rolls over. This function takes as a parameter the last time it returned (or 1 for the first call) and returns the current time. Using the last time the function can detect if a roll over occurred.

```
/*.....*/
/* TimerRegister: Register an expiration time */
/* */
/*      Input: microseconds until timer expires */
/* */
/*      Returns: resulting timer structure */
/*.....*/
struct timer TimerRegister(u64 microseconds)
{
    struct timer tw;
    u64 now;

    /* Retrieve the current time ticks of T1_CLOCK_SECOND frequency. */
    now = REG32(TIMER_CLO); /* 32 bits of time */

    /* Scale hardware time to microseconds by multiplying by the */
    /* magnitude (scale) of the difference in time frequency. */
    if (MICROS_PER_SECOND > T1_CLOCK_SECOND)
        now *= (MICROS_PER_SECOND - T1_CLOCK_SECOND);

    /* Calculate and return the expiration time of the new timer. */
    tw.expire = now + microseconds;

    /* Return the created timer. */
    return tw;
}

/*.....*/
/* TimerRemaining: Check if a registered timer has expired */
/* */
```

```

/*      Inputs: expire - clock time of expiration in microseconds      */
/*                  last - the last time this function was called      */
/*
/*      Returns: Zero (0) or current time if unexpired                */
/*.....*/
u64 TimerRemaining(struct timer *tw)
{
    u64 now;

    /* Retrieve the current time from the 1MHZ hardware clock. */
    now = REG32(TIMER_CLO); /* 32 bits of time */

    /* Scale hardware time to microseconds by multiplying by the */
    /* magnitude (scale) of the difference in time frequency. */
    if (MICROS_PER_SECOND > T1_CLOCK_SECOND)
        now *= (MICROS_PER_SECOND - T1_CLOCK_SECOND);

    /* If low order 32 bits of 'last' > 'now' a rollover occurred. */
    if ((tw->last & 0xFFFFFFFF) > now)
        now += ((u64)1 << 32);

    /* Add saved high order bits from the last. */
    now += tw->last & 0xFFFFFFFF00000000;

    /* Return zero if timer expired. */
    if (now > tw->expire)
        return 0;

    /* Return time until expiration if not expired. */
    tw->last = now;
    return tw->expire - now;
}

```

These functions will often be used by other components of the system software and user applications, so let us declare the functions globally in `system.h`. Note the additional macro definitions below that define `Sleep()` using the `usleep()` function and

multiplying the parameter. Defining functions as a macro that uses another function keeps the software small yet easy to use and read by others. The `usleep()` function is defined so that we can **mask** the input parameter to a 64 bit value, allowing the caller to use a smaller sized integer if desired. Another important concept introduced below are C data **structures**, which combine a collection of C types into a single **structure** or 'struct'. We define the timer as a **structure** that holds both the expiration time as well as the last time the timer was checked.

```
/*
 * Timer structures
 */
struct timer
{
    u64 expire;
    u64 last;
};

...

/*
 * Timer interface
 */
#define Sleep(a) usleep((u64)(a) * MICROS_PER_SECOND)
void usleep(u64 microseconds);
struct timer TimerRegister(u64 microseconds);
u64 TimerRemaining(struct timer *tw);
```

5.4 Software timer uses

In software design, it can be desired to use a **naming convention** to identify the scope of a function or variable through the use of letter capitalization. The authors preference is that global variables and functions should capitalize the words without spaces, while local variables and functions should be lower case and separate words with the underscore `_` character. This way it is possible to know the scope of a

variable or function just by the name of the variable or function, greatly increasing the readability and understanding of the source code.

It is common in system software to have to wait for a particular hardware condition to be true before continuing. Hardware peripherals can become unresponsive due to internal processing and thus require a wait in the system software before further software control can be applied. To aid the need to wait on a condition, a hardware register or software variable, we end this discussion on timers with an example of a conditional wait loop. The loop looks similar to `usleep()` except that the loop exits as soon as a conditional statement is true or the timer expires. The variable 'microseconds' should contain the time out value in **microseconds** and an expression 'condition_is_true' that must evaluate to true before breaking out of the loop before expiration. After this loop finishes, software must check if a timeout occurred, possibly with another `TimerRemaining()` call if needed.

```
struct timer expire = TimerRegister(microseconds);

/* Loop checking the condition and timer until true or expiration. */
for (;condition_is_true || (TimerRemaining(&expire) > 0);)
{
    // Do work/check peripheral
}
```

To summarize, in this section we created a global software interface containing timer and sleep functions. We examined these functions and provided examples how to use them. Congratulations, we have created a system software library for timers that will be the foundation of all algorithms to follow.

5.5 Building Software with Make

So far we have been using the command line to compile our software, and have used shell scripts to automate and eliminate errors when using these commands. The software package **make** is a common and very helpful tool that can be used to manage the compiling, linking and cleaning needed to create and update libraries and final binary images to be executed by hardware. As programs grow in source files, it is cumbersome to maintain the build scripts that compile each individual C file and then link the software.

Make is historically a tool for compiling on Unix systems, but is now available on all modern operating systems such as Windows, Mac OS and Linux. The idea behind **make** is that each application or solution has its own **Makefile**, which defines a specific number of source code files, and build options, that are to be used to compile and link the software solution. In one analogy, the **Makefile** is the recipe while **Make** is the chef.

Ending this chapter will be an example of a **Makefile** for an application that has a single source file, main.c. It is not important to understand every line of this **Makefile**, just the concept of what it is doing. For example, to add additional source code files, include them in the OBJECTS definition. Also note that the 'memory.map' linker script created earlier is used as a parameter to the linker. This is needed for the linker to create a resulting binary that is in the proper position to execute on the system. Organizing source code into multiple C header files and C source files is easy when using a **Makefile**. This ease of management eliminates the bad practice of putting everything into the main.c file to avoid updating the build scripts.

Presented below is an example of a **Makefile** configured for the default GNU C compiler on a PC. Note that the pound symbol '#' is used to indicate a comment.

```
#
# Makefile for application
#

##
## Commands:
##
CP   = cp
RM   = rm
C    = gcc
CC   = gcc
LINK= ld
PLINK= objcopy

##
## Definitions:
##
APPNAME = app
```

```
EXTRAS = -ffreestanding
CFLAGS = -Wall -O2 $(EXTRAS)
ASFLAGS =
INCLUDES = -I.

##
## Application
##
OBJECTS = main.o \

##
## Define build commands for the C and assembly source file types
##
.c.o:
    $(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $<

.s.o:
    $(AS) $(ASFLAGS) -o $@ $<

##
## Targets
##
all: object

object: $(OBJECTS)
    $(LINK) -T memory.map -o $(APPNAME).elf $(OBJECTS)
    $(PLINK) $(APPNAME).elf -O binary $(APPNAME).bin

clean:
    rm -f $(OBJECTS)
    rm -f *.bin
    rm -f *.elf
```

This **Makefile** can perform two actions, to build the software or clean out the previously created software. Let us take a look at running the above **Makefile** for the main.c application, first to clean out the old output files and then to compile and

link a fresh version of the software. This process of creating software is referred to as a build of the software. Let us review the output of the above Makefile to see what it is doing.

```
$ make clean
rm -f main.o
rm -f *.bin
rm -f *.elf

$ make
gcc -c -Wall -O2 -DRPI=3 -ffreestanding -I. -o main.o main.c
ld -T memory.map -o led.elf main.o
objcopy led.elf -O binary led.bin

$
```

5.6 Project Management

It is good project management to build software in increments and validate the changes by compiling and testing the software often, after each change. This repeated testing style of development is sometimes referred to as incremental, or agile. The repetition helps isolate problems early to avoid introducing multiple problems at once. Many system software problems tend to leave symptoms rather than an easy to identify source of the problem. Daily and incremental backups should be performed whenever you begin development for the day, as well as at each successful point. The simplest form of version control is to copy yesterday's source folder(s) to a backup folder (preferably a different hard drive or removable flash drive). Rename the backup folder with today's date so it is easy to find and go back to previous versions.

Backups are vital to recovering from a catastrophic bug or problem that cannot be easily understood, fixed or otherwise root caused. The procedure when confronted with this obstacle is to go back to an older version of the software and slowly start adding in the changes, testing often. Oftentimes the problem has existed for some time but was not exposed or detected until some recent change. Having an archive trail can be very helpful and encourages risk taking during development. Want to try an off the

wall idea? Archive and hack away safely. Achieve a milestone? Back up immediately before success becomes but a fleeting moment in time.

Commercial and open source tools exist to aid the task of maintaining versions of source code files. The details of the different version control systems will be specific to the tool used, but the concepts are the same. To simplify the concept, version control systems contain source files with the mechanism to add new versions of a file at any time. Since each new version includes a comment about the change, it can aid debugging to compare with a previous version. Version control systems also allow rolling back to a previous version if needed. Common version control lingo uses library terms; to modify a file requires first a `check out` and then a `check in` once the file has been changed and tested successfully. Keeping each small change in a version control system allows the developer to quickly isolate a new problem and can allow rapid recovery from newly introduced bugs.

5.7 System Software Organization

Now that we have created a system software library for the timer, let us organize the source code into directories so we can easily expand the system software in the future. First, in the source code tree 'source', create three directories, `applications`, `boards` and `include`. In the `applications` directory create another directory named `errorcodes` and copy the source code files from the previous and current chapter to `errorcodes` (`main.c`, **Makefile**, `memory.map`). Copy the `system.h` file from the previous chapter to the `include` directory. In the `boards` directory, create a directory for the hardware and then create two files in that directory, `board.h` and `board.c`.

Open `board.h` and `main.c`, cut and pasting the definitions from the top of `main.c`, added in the previous chapter of the companion laboratory book, to `board.h`. With the definitions removed from `main.c`, the `#include <system.h>` line must be added to the top of `main.c` before it will compile successfully. Next open `board.c` and cut and paste the time functions created in the previous section to this file. `System.h` should already contain the prototypes for these functions. Here is a directory map of the new organization.


```
Source\applications\errorcodes
    main.c
    Makefile
    memory.map
Source\boards\<>board_name>
    board.c
    board.h
Source\include
    system.h
```

The last step needed is to modify the **Makefile** to use the new directory hierarchy above. Open the file `Source/applications/errorcodes/Makefile` and replace the old `INCLUDES` line with this one. Replace `<board_name>` with the name of the board used in the laboratory assignments.

```
INCLUDES = -I. -I../..../include -I../..../boards/<board_name>
```

Next, modify the `OBJECTS` line to include the files and paths.

```
OBJECTS = main.o \
    ../..../boards/<board_name>/board.o \
```

Finally, define a function prototype ‘`void BoardInit(void)`’ in `system.h`. Open `board.c` and create this `BoardInit()` function, copying the LED GPIO initialization from `main.c` to this `BoardInit()` function. This code is hardware dependent and will not be reproduced here.

If the above changes were completed correctly the application should now compile with **Make**. If this reorganization is not compiling, see the laboratory assignment following this chapter for more details. This new software organization will be the starting point for the following chapter, as well as all future chapters. This organization allows the system software to be easily expanded in future chapters or changed to use a different board.

Chapter 5 Glossary:

Frequency - how often an event occurs, measured in times per second. For example, the number of CPU operations a second is the CPU **frequency**.

Make - a command line program used to organize and automate the compilation and linking of software source code into an executable form.

Makefile - a human readable file conforming to the **Make** syntax that describes the source files that are to be compiled and linked together into an executable. By default, the **Makefile** in the current directory is parsed and executed by the **make** command line application.

Mask - A technique to change a variable from one declared type to another. Often used with pointers when the raw hardware register of the hardware peripheral must be converted to another type. Use **masks**, particularly pointer **masks** cautiously and appropriately. Whenever possible change the variable declarations to match instead of using a **mask**.

Microsecond - one millionth of a second is $1/1000000$ or 10^6 seconds.

Millisecond - one thousandth of a second is $1/1000$ or 10^3 seconds.

Naming Convention - a defined style for naming variables and functions of a particular type or scope.

Structure - an ordered group of C variables that is defined as a new type of variable. Once a **structure** is defined, variables can be then be declared with that type.

Ticks - the number of times a hardware or software clock has counted. To convert **ticks** to seconds requires knowing the clock **frequency**, defined as the number of **ticks** per second.

Time Delta - the change or difference between two time values. A **time delta** can be represented as clock **ticks** or some form of seconds, such as **microseconds**.

Chapter 5 Exercises:

1. How many **microseconds** does it take to equal one **millisecond**? How many **microseconds** equal 1.25 seconds?
2. Ensure the timer performs the roll over correctly by expanding on the source code from previous labs to create a program that uses the `TimerRegister()` and `TimerRemaining()` functions to turn the LED on after 75 minutes.

Chapter 6: Universal Asynchronous Receiver Transmitter (UART)

6.1 UART Introduction

The Universal Asynchronous Receive Transmitter, or **UART**, can transfer or **receive** bits of **data** across a physical cable connected between two computer systems. The **UART**, pronounced 'you art', is a great way to directly connect the development PC used to create system software with the hardware that executes this system software. With a **UART** software **driver**, the system software can print information such as error **strings** to the development PC during run time, over the **UART**. The system software application could present a menu of commands or even provide a command line interface (system shell) over the **UART**. This familiar interactive and debugging interface speeds up development of more complex **drivers** making the **UART** the usual first choice of **drivers** to bring up on a new hardware system.

6.2 UART Hardware

The simplest **UART** is a two pin connection, **Rx** and **Tx**. Historically a **UART Serial Port** had a 9 pin connection named **DB9** and powered by 5 volts. More recently the **UART** pins are exposed as **Rx** and **Tx** (and optional **RTS** and **CTS** for hardware flow control) pins and powered at 3.3 volts. This lower voltage configuration is referred to as a **UART TTL**, where **TTL** stands for Transistor-Transistor Logic. Both **UART Serial Ports** and **UART TTLs** appear as **COM** ports to the Windows user, and as a **TTY** device (`/dev/ttyX`) on Linux.

It is required to find and read the hardware specifications for the **UART** to be sure of the voltage, pin layout and/or cable. Typically **UART** connections that have **DB9** pin connections are 5 volt **Serial Ports**, while those **UARTs** that connect directly to pins

on the motherboard are often 3.3 volt TTLs. But bare pins can also be 5 volt so read the documentation. The term TTL is always 3.3 volt so look for this distinction.



Important

If there is only a USB **UART** interface on the board to be used for system software development it cannot be used with this book. The USB **UART** requires the system software creator first write a USB driver before creating a **UART** driver.

UART interfaces must be physically connected properly between the two systems or communications will not work. This physical connection consists of at least **transmit** and **receive** wires, commonly referred to as **Tx** and **Rx**. The **UART** is designed so that the **receive (Rx)** on one system is wired to the **transmit (Tx)** of the other system. One hardware option with the **UART** is flow control, which adds wires so the **transmitter** can ask if the **receiver** is ready before sending **data**. Intended to decrease **data** loss, hardware flow control requires two extra **UART** wires, Request to Send (RTS) and Clear to Send (CTS).

Hardware flow control can minimize **data** loss but adds hardware complexity as it requires extra wires connected to a compatible **UART** controller on both sides of the communication. Not all **UART** hardware interfaces support hardware flow control. Flow control is generally not needed for common uses of the **UART** (error reports, system shell, etc.), but is needed for uses that do not verify **data** integrity with software, and cannot accept **data** loss during transfer. For example, when transferring a binary image to another system to execute, a single incorrect bit can cause undefined behavior during execution.

It is very important to find documentation and understand the **UART** physical interface to the system that is be brought up with software. For example, be sure to purchase a USB to **Serial (UART)** cable where the **UART** side will connect properly with the board under development. Check that the USB side connected to the development PC has a good **driver** for your development OS. The wrong connection on either end will result in frustration or could even supply too much electricity (over voltage) and destroy the hardware. If in doubt, please follow the instructions in the laboratory assignments exactly.

6.3.1 Configure the Development PC

The most common use of **UART** communication in system engineering is for sending debugging information or providing a system shell interface. It is important to provide both interfaces to the system developer so that error information can be sent and commands can be received. It is helpful to split this larger goal into stages, so first let us focus on creating an application to initialize and configure the **UART** interface.

Start on the development PC and install an application that can be used to connect to a **serial UART** port. The recommendation for Windows PCs is TeraTerm, or ExtraPuTTY, and Minicom on Linux PCs; all are free downloads or package upgrades. The laboratory assignments will discuss the details of how to install and use these applications depending on hardware. This said, configuring **serial UART** connections have common features which are discussed below.

6.3.2 Configure the System Software

The software configuration must be synchronized on each side of the **UART** communications. The most important and commonly modified configuration is the **baud rate**. The **baud rate** is the speed in bits at which the **UART** transfers **data** and common settings are 115200 or 9600 bits per second (bps). If one side is configured for a different **baud rate** than the other, no communication will occur or garbage will appear on either end.

Other software configuration options for the **UART** will be mentioned but rarely changed beyond the default. These are **data size**, parity and stop bit. The **data size** is the number of bits sent before the stop bit, typically 8 bits (a byte at a time). The **UART** hardware sends a start bit, the **data** bits followed by optional parity bit and then one or more stop bit(s). Adding a parity option and/or a second stop bit can increase **data** transfer reliability, especially when used as in the past across long distances. Most modern PC **UART** adapters are for short range and use 8 bit **data**, No parity and one (1) stop bit, often referred to as 8N1. Use other settings only if you have a reason and know both sides of the **UART** interface support and are configured for it.

The **UART** software interface can be separated into two groups, initialization and **data** transfer. The main thing the software must do to initialize the **UART** hardware

is to configure the the **baud rate**. Once the **baud rate**, and hardware GPIO pins if necessary, are configured on the system the **UART** communication (**Rx** and **Tx**) can begin. A software interface must be created to **transmit (Tx)** as well as **receive (Rx) data**.

As previously mentioned, most modern **UART** software interfaces have a default of 8 bits, no parity and one stop bit (8N1) so the only setting that often changes is the **baud rate**, or **data** bits per second or transfer speed. To configure the **baud rate** with system software is not as simple as it sounds. Systems often share a common clock with the **UART** and other peripherals, while the **UART** requires a specific clock rate in order to transfer **data** at a specific **baud rate**. Each system shares a different clock with a different frequency, yet the same **UART** peripheral can work across all of the different systems using what is known as a **clock divisor**. Most peripherals, including **UART**, have a circuit to a clock on the motherboard. This system clock then becomes the **UART** clock, but this is hardly ever the correct frequency for the desired **baud rate**. The **UART** peripheral includes a **clock divisor** that can be configured in software to divide the system clock in order to achieve the correct **baud rate**.

The **clock divisor** interface is typically one or two registers, an integer **divisor** and possibly a fractional remainder. Let us review the general equation now.

```
baud_rate = system_clock / divisor;
```

Systems with only an integer **divisor** require a compatible system clock to achieve common **UART baud rates** (9600, 115200, etc.). Many **UART** hardware to software interface supports an integer as well as a fractional **baud rate divisor**. The fractional **divisor** is an integer representation of the remainder value. The hardware documentation should provide the specific equations needed to calculate the integer and fractional **baud rate divisors**. If in doubt, review the hardware documentation and pay close attention. The real life equations usually include constant **divisor** multipliers added in hardware to make the **UART** peripheral compatible with the clock. These need to be reflected in software, similar to this example equation.

```
baud_rate = system_clock / (16 * divisor)
divisor = system_clock / (baud_rate * 16)
fractional_divisor = (fractional_remainder * 64) + .5)
```

In the above example equation, the **divisor** is an integer, but the result of the division is typically not a whole number but has a remainder. The fractional remainder is

then multiplied by another constant, rounded up and then converted to an integer (remainder dropped). Here is an example where we configure the integer and fractional **divisor** configuration for 115200 baud where the external UART clock is 3MHz (3000000). This example is tailored to 16C650 compatible UART interfaces, but is applicable to any UART software interface that has both an Integer Baud Rate Divisor (UART0_IBRD) and Fractional Baud Rate Divisor (UART0_FBRD).

```

/*
   divisor = 3000000 / (115200 * 16) = 1.627 = 1
   fractional divisor = (.627 * 64) + .5) = 40
*/
REG32(UART0_IBRD) = 1; /* Integer baud rate divisor */
REG32(UART0_FBRD) = 40; /* Fractional baud rate divisor */

```

It is always best to do the calculation on the development PC and include the equation and results in the comments so others can see the algorithm and are able to change the **baud rate** if needed. The last configuration step that UART software should do is to enable the UART controller. Conversely, before configuring the UART, it should first be disabled. Here is an example UART initialization function for 16C650 compatible UART interfaces that disables, configures and then enables the UART peripheral number zero (UART0) to use the 115200 **baud rate**.

```

/*.....*/
/*   UartInit: Initialize the UART           */
/*                                           */
/*.....*/
void UartInit(void)
{
    /* Disable the UART before configuring. */
    REG32(UART0_LINE_CTRL) = 0;
    REG32(UART0_CONTROL) = 0;

    /* divisor = 3000000 / (115200 * 16) = 1.627 = 1
       fractional divisor = (.627 * 64) + .5) = 40 */
    REG32(UART0_IBRD) = 1; /* Integer baud rate divisor */
    REG32(UART0_FBRD) = 40; /* Fractional baud rate divisor */

```

```
/* Enable the UART and clear the received error count. */
REG32(UART0_LINE_CTRL) = (BYTE_WORD_LENGTH | ENABLE_FIFO);
REG32(UART0_CONTROL) = ENABLE | TX_ENABLE | RX_ENABLE
}
```

There are often other steps to initialize the **UART**, such as configuring the **UART** to use the system clock and enabling the **Rx** and **Tx** GPIO pins if the hardware interface pins out through the GPIO interface. Consult the laboratory assignment for any extra configuration steps that are needed. Any extra software configuration steps must be added to `UartInit()`, in between disabling the **UART** and enabling it.

6.4 Send Data with Software

The **UART** sends, or **transmits**, **data** at the configured **baud rate**. Typically 8 bits (a byte) at a time can be sent by writing into a hardware First In First Out (**FIFO**) queue. Our first thought after reading the hardware register interface might be to just write to the output **data** register, like this.

```
void UartPutc(u8 character)
{
    /* Send the character. */
    REG32(UART0_DATA) = character;
}
```

But the software interface should check the **Tx** status bits to ensure the **Tx FIFO** is not full before attempting to send **data**. If the **Tx FIFO** is full, the outgoing byte written will not be sent and lost/dropped instead. Once the hardware **Tx FIFO** is not full, software may assign the byte to be sent to the **data** register for sending. Here is an example function to send a byte over the **UART**. Some **UARTs** without **FIFO** may have a general bit in the status to communicate that it can send no more at that time. A solution for sending **data** with software over a **UART** with **FIFO** support is commonly similar to this.


```

/*.....*/
/*   UartPutc: Output one character to the UART   */
/*
/*       Input: character to output
/*.....*/
void UartPutc(char character)
{
    unsigned int status;

    /* Read the UART status. */
    status = REG32(UART0_STATUS);

    /* Loop until UART transmission line has room to send. */
    while (status & TX_FIFO_FULL)
        status = REG32(UART0_STATUS);

    /* Send the character. */
    REG32(UART0_DATA) = character;
}

```

Some **UARTs** may not have **FIFO** full bit and this check will be different, but software must check some bit to be sure the **UART** can send more **data** before writing to the **data** register or **Tx data** may be lost. Also of note in the above code is the use of ‘char’ instead of ‘u8’. This is because ‘char’ can be different sizes depending on the language type. Using ‘char’ instead of ‘u8’ allows the **UART** system software to work for any language.

It is often helpful to print out an entire **string** of characters at once, such as a debug or error statement. To do this requires using an **array** of characters, or a **string**. Let us use `UartPutc()` to create a print **string** function now. Note that a **string** end is defined by a zero byte or character `\\0`. This zero value is commonly referred to as **NULL**. **Arrays** in C are declared and elements within the array are referenced with brackets `[]`. In the declaration of a variable type, an **array** of the same type can be declared by including brackets surrounding the size of the **array**. For example ‘`u8 string[20]`’, defines an 8 bit unsigned integer **array** of length 20. When referencing an **array**, the number in the brackets is the specific element within the **array**, starting from zero. When using **arrays** as **strings** be sure to leave room in the **array** size for the **NULL**

character at the end of the **string**.

It is also possible to use **string** constants in C by simply enclosing the **string** in double quotes (“”). Similarly, individual characters can be represented in C with single quotes (‘’). Special characters of note are the carriage return (‘\r’), which brings the cursor down one line, and the new line (‘\n’), which moves the cursor to the left edge of the screen. Let us review the function that takes a **string** as a parameter and puts the **string** (Puts) to the to **UART**, appending a carriage return and new line to the end of the **string** so the next output to the **UART** appears on the left side of the next line.

```

/* ..... */
/*   UartPuts: Output a string to the UART   */
/* ..... */
/*           Input: string to output         */
/* ..... */
void UartPuts(const char *string)
{
    int i;

    /* Loop until the string buffer ends. */
    for (i = 0; string[i] != '\0'; ++i)
        UartPutc(string[i]);

    /* The puts() command must end with new line and carriage return. */
    UartPutc('\n');
    UartPutc('\r');
}

```

The above code outputs each character in the **array** to the **UART** controller, one at a time, until the end of the **string** (the **NULL** or `\0` character). Then to complete the line requires the carriage return (`\r`) and new line (`\n`) characters to move the cursor to the beginning of the next line, so the next character will be one line down and aligned on the far left side.

6.5 Receive data with Software

The **UART** receives data at the configured **baud rate**, usually 8 bits (a byte) at a time into a **FIFO** queue in hardware. Software checks the **Rx** status bits to see if the **Rx FIFO** holds data. If so, software should copy out the bytes in the **FIFO** and then clear the **Rx** status bits so hardware can send more. Here is an example function to check the **UART** to see if a character has been **received**.

```

/*.....*/
/* UartRxCheck: Return true if any character is waiting in UART FIFO */
/*
/* Returns: one '1' if UART receive has a character
/*.....*/
unsigned int UartRxCheck(void)
{
    /* If RX FIFO is empty return zero, otherwise one. */
    if (REG32(UART_STATUS) & RX_FIFO_EMPTY)
        return 0;
    else
        return 1;
}

```

The above function reads the **UART** status register to check if the **Rx FIFO** is empty before returning this result. Programs that do not or cannot wait should first use `UartRxCheck()` to ensure a character is ready to be read.

The next function needed is to read (get) a character from the **UART**, waiting forever until a character arrives. The function design is to wait until the **UART** has **received** a character by looping on the status register (`UART0_STATUS`) until the **Rx FIFO** is not empty. Oftentimes the **UART** can experience errors **receiving** and sometimes these errors require acknowledgment by software for the hardware to continue. Let us review this function that waits for a character to arrive and then acknowledges any **Rx** errors before reading the character from hardware and returning it to the caller.

```
/*.....*/
/*   UartGetc: Receive one character from the UART   */
/*                                                    */
/*       Input: character to output                  */
/*                                                    */
/*.....*/
char UartGetc(void)
{
    u32 character, status;

    /* Loop until UART Rx FIFO is no longer empty. */
    while (!Uart0RxCheck());

    /* Read the character. */
    character = REG32(UART0_DATA);

    /* Read Rx status to acknowledge character and check for error. */
    status = REG32(UART0_RX_STATUS);

    /* Check for and clear any read status errors. */
    if (status & RX_ERROR)
        REG32(UART0_RX_STATUS) = status;

    /* Return the character read. */
    return (RX_DATA & character);
}
```

6.6 System Shell

A serial UART peripheral can be used by system software to provide a shell interface to the system. This is typically a command line interface similar to the development environment PC, but typically more limited. A command line interface must have the following basic principles. First it must present a visible prompt to the user and then wait at this prompt until a command is entered by the user. This can be as simple as

presenting a numbered menu (1 through 5 for example) and then waiting for a valid number to be pressed. Here is an example of a simple menu interface.

```
/*.....*/
/*   OsMenu: present menu and execute selection   */
/*.....*/
void OsMenu(void)
{
    u8 command;

    /* Loop forever. */
    for (;;)
    {
        /* Print menu. */
        UartPuts("Welcome! Please choose a command (1-3):");
        UartPuts("  1. Exit/Goodbye");
        UartPuts("  2. Turn on LED");
        UartPuts("  3. Turn off LED");

        /* Wait for the user to enter the command. */
        command = UartGetc();

        /* Exit command */
        if (command == '1')
        {
            UartPuts("Until we meet again");
            return;
        }

        /* LED on command */
        else if (command == '2')
            LedOn();

        /* LED off command */
        else if (command == '3')
```

```

    LedOff();
}

```

Often times the menu interface is limiting for applications and a full command line interface is desired. In order to accomplish this we will need an algorithm that will compare character **strings**. This is needed to match the user input to the available command list to find the matching command. Let us create a 'system/shell.c' file now, entering the source code below into it. Afterward we will have an expandable system shell that can be used as the foundation for future software features. In this initial version there are three commands, 'ledon', 'ledoff' and 'echo'.

This design uses a ShellCmd structure type that stores the command **string** and the associated function pointer for this command. There is then a global **array** of 'ShellCommands' defined of this 'ShellCmd' type. This allows the system software creator to quickly add new shell commands by adding an entry to the 'ShellCommands' **array**. Also worth discussing is that the 'ledon' and 'ledoff' commands use the same underlying registered shell function led(). Since this shell interface passes the command **string** to the registered function, the led() function can use this command **string** to determine if the command is to turn on or off the LED. Please read and digest the code below, paying attention to the comments and the associated code below them.

```

/*.....*/
/* Configuration */
/*.....*/
#define COMMAND_LENGTH 80

/*.....*/
/* Type definitions */
/*.....*/
typedef struct
{
    char *command;
    int (*function)(const char *command);
} ShellCmd;

/*.....*/
/* Function prototypes */

```

```
/*.....*/
/*
** Shell Functions
*/
static int echo(const char *command);
static int led(const char *command);

/*.....*/
/* Global Variables */
/*.....*/
static ShellCmd ShellCommands[] =
{
    {"echo", echo},
    {"ledon", led},
    {"ledoff", led},
    {0, 0},
};

/*.....*/
/* Local function definitions */
/*.....*/

/*.....*/
/*      echo: Perform the echo command */
/*      */
/*      input: command = the entire command */
/*.....*/
static int echo(const char *command)
{
    /* Echo the command to the console. */
    UartPuts(command);
    return 0;
}

/*.....*/
/*      led: Command to turn the LED on or off */
/*      */
```

```

/*                                                                    */
/*      input: command = the entire command                            */
/*                                                                    */
/*.....                                                                    */
static int led(const char *command)
{
    /* Check the last letter of the command */

    /* If 'ledon' then last array element 4 is 'n' */
    if (command[4] == 'n')
        LedOn();

    /* Otherwise if 'ledoff' then array element 4 is 'f' */
    else if (command[4] == 'f')
        LedOff();
    else
        UartPuts("error - shell mishandled command?");

    return 0;
}

/*.....                                                                    */
/*      shell: Run a system shell command                            */
/*                                                                    */
/*      input: command = the entire command                            */
/*.....                                                                    */
static int shell(const char *command)
{
    int i, j;

    /* If question '?', print out list of available commands. */
    if (command[0] == '?')
    {
        UartPuts("Available commands are:");
        for (i = 0; ShellCommands[i].command; ++i)
            UartPuts(ShellCommands[i].command);
    }
}

```



```

    return 0;
}

/* Search the command table and run any installed commands. */
for (i = 0; ShellCommands[i].command; ++i)
{
    /* Loop through length of command, comparing each character */
    /* until either string ends with the NULL character '\0'. */
    for (j = 0; (command[i] != '\0') &&
           (ShellCommands[i].command[j] != '\0') ; ++j)
    {
        /* If character does not match then command does not so break. */
        if (command[j] != ShellCommands[i].command[j])
            break;
    }

    /* If end of shell command reached then all characters match. */
    if (ShellCommands[i].command[j] == '\0')
    {
        /* Execute and return the command result. */
        return ShellCommands[i].function(command);
    }
}

/* Return command not found. */
return -1;
}

/*.....*/
/* Global function definitions */
/*.....*/

/*.....*/
/* SystemShell: system shell executes commands until 'quit' */
/*
/*
/*.....*/

```

```
void SystemShell(void)
{
    int result = 0, i;
    char character, command[COMMAND_LENGTH];

    /* Loop to read and perform the boot loader commands from UART. */
    for (;;)
    {
        /* Output the shell prompt '>'. */
        UartPutc('>');

        /* Loop reading user input characters to build 'command' array. */
        /* Break out when 'Enter' key pressed or command length reached. */
        for (i = 0, character = 0; (i < COMMAND_LENGTH) &&
            (character != '\r'); ++i)
        {
            /* Read a character and put it in the command array. */
            character = UartGetc();
            command[i] = character;

            /* Echo the character back to the user. */
            UartPutc(character);
        }

        /* Output new line and NULL terminate the command string. */
        UartPutc('\n');
        command[i] = 0;

        /* If an empty command entered, loop back to give a new prompt. */
        if ((i == 1) && (character == '\r'))
            continue;

        /* Perform the command. */
        result = shell(command);

        /* Report any errors. */
    }
}
```

```
    if (result > 0)
        UartPuts("command error");
    else if (result == -1)
        UartPuts("command unknown\n");
}
}
```

Did you notice that the `shell()` and `led()` functions are declared as static? The static function declaration means the function is only referenced from functions defined within the same C file. Maximizing the use of static functions to avoid global clutter is fundamental to good systems engineering. Typically a static function will not need a function prototype if declared before use, however because the `ShellCommands` array assigns the function pointers before the functions are declared, these static functions need to be declared ahead of time in the function prototypes section.

Chapter 6 Glossary:

Array - A series of variables of the same type declared at the same time and referenced one after another in order in memory. A common **array** is the human readable string, which is defined as an **array** of characters.

Asynchronous Communication - A technique to send and receive data that is not synchronized to a common clock signal. These types of communications are the simplest way to transfer data between computer systems or peripherals, but often suffer from data loss and corruption, especially compared to synchronous interfaces.

Baud Rate - the rate or speed of data transfer (Rx and Tx) for a peripheral. This is measured in bits per second (bps). Both ends of a serial interface must configure the same **baud rate** or communication will fail.

Clock Divisor - the value used to divide the external clock frequency in order to achieve the desired internal clock frequency. Used by peripherals to interface with a shared system clock.

Data - A general term used for any information stored or transferred in digital/binary format.

Driver - A component of a software system that provides a software interface to a hardware peripheral.

FIFO - A First In First Out buffer interface stores **data** in order so that the first **data** in will be the first **data** out. Used between hardware and software interfaces to buffer data, speeding transfers and reducing **data** loss.

NULL - A value most often defined as zero, the **NULL** value is used for pointers to indicate no pointer is assigned. **NULL** is also commonly used to terminate **arrays** such as **strings**.

Receive - The act of accepting **data** from a sender. Abbreviated as **Rx**.

Rx - An abbreviation for the **receive** side of a **serial UART** or other communications interface.

Transmit - The act of sending **data** to a **receiver**. Abbreviated as **Tx**.

Tx - An abbreviation for the **transmit** side of a **serial UART** or other communications interface.

Serial - A communication interface that is designed for simplicity by sending **data** one after the other, or serially. The JTAG controller is an example of a synchronous **serial** communication interface, while the UART is an **asynchronous serial** communication interface.

String - A sequence or **array** of characters that ends in the zero byte. The zero byte is commonly referred to as **NULL** and **strings** are commonly referred to as **NULL** terminated.

UART - Universal **Asynchronous Receiver Transmitter** is a simple peripheral that can be used to connect systems. Typically the first peripheral a system engineer will bring up, a **UART** allows diagnostic **data** to be sent to the development PC, as well as a command line shell interface to issue commands to, and receive responses from, the remote system.

Chapter 6 Exercises:

1. Define a structure that contains two unsigned 32 bit integers containing the high and low 32 values of an unsigned 64 bit integer. Assign high and low values to this structure and then mask this structure to a u64 integer. Are the u64 integer and structure the same value?

2. See Lab 6 assignments for additional exercises. The best way to understand all the new system code is to complete the hardware-to-software interface within the lab to create and use a powerful command line shell interface!