
Programming the X16 for beginners
The definitive guide to BASIC on the Commander X16
Justin Baldock

Programming the X16 for beginners

The Definitive Guide to BASIC On The Commander X16

Justin Baldock

This book is dedicated to Phil and Mary Baldock.
More loving and supportive parents I could not ask for.

Table of Contents

Introduction

"Ripples are made by those reckless enough to jump into the ocean." - Michael Bassey Johnson

Acknowledgements

This book began development in January 2020. Special thanks are due to my parents who have supported me my entire life. Without them, I would not have spent countless hours tinkering and playing on the Commodore 64, Amiga 500, a PC 486-SX and gaining a career in IT.

What is this book about?

The main goals are to teach the BASIC programming language on the Commander X16 and provide a handy reference book about the Commander X16. It will cover programming in BASIC as well as some design theory. An emphasis will be placed on good programming practices such as easy to read code that can be enhanced quickly and is well documented.

For these reasons, this book has two kinds of programs. First, short programs are intended to be easy to understand to teach a concept. Second, more extended programs that use graphics, sound, etc. The shorter programs will often show how BASIC commands are used. The more extended programs are designed to be practical applications for you to use in your future work and study.

Who is this book for?

This book is for you if you can answer 'yes' to any of the following questions.

- Do you want to learn how to program BASIC on the Commander X16?
- Do you want to learn more about how the Commander X16 works?
- Do you want a handy book to refer to when programming?
- Do you want to improve your problem-solving skills?
- Are you interested in creating games on the Commander X16?

Who should walk away from this book?

If you can answer 'yes' to any of these questions.

- Are you an expert in programming BASIC on the Commander X16?
- Do you want to learn Assembly programming on the Commander X16?
- Are you looking for a reference book that covers all the technical details in excruciating detail?
- Would you rather eat rotten eggs than learn something new?

Prerequisite Knowledge

This book combines everything I have learned about BASIC programming. I have tried to write it for the average person. The hope is that this book is suitable for anyone to pick up and understand. Earlier programming experience is not needed but is helpful. It is assumed you are familiar with using computers, connecting to power and display, and typing on a keyboard.The content covers a broad range of topics to hopefully provide a solid base of skills for those studying it.

How you learn

There is a discussion between academics about learning styles [1]. Amongst the various learning style models, there does appear to be some commonality. It comes down to how you prefer information to be presented, how you like to process new information, how you take in information and how you organise and progress towards understanding. The only certainty is that we each learn in our way and at our speed.
TODO improve this section

How to use this book

I have tried to create a super helpful learning resource for you, and I have a few recommendations on how to get the most from it.

- Slow down. The better you understand the concepts, the less effort will bed needed to memorise them
- Limit your reading to 30-45 minutes. Once you start to lose focus, you will not have quality learning
- If it is helpful for you, write notes. There is evidence that is writing your notes while reading can increase the learning experience
- If it is helpful for you, read the book aloud. There is evidence that hearing information can increase the learning experience
- Write down your ideas! Once you start to explore new concepts, you will begin to create ideas for your programs. Capture these ideas as you have them
- Create something. Do something beyond the examples in this book. Experiment! The best way to learn programming techniques is to try things out. Once you have seen how I present a solution to a problem, try, and solve the same problem a different way
- You are learning a foreign language, one of the many different computer languages. To learn any language, you must practice, practice, practice

- There will be elements that are repeated. This book aims to help you learn how to program the Commander X16. Repetition helps with retention
- There may be elements that appear but are not explained until further in the book
- Follow up on the recommended reading once you have finished reading this book. You will gain understanding by seeing how other people solve problems. You may notice patterns or techniques that you can apply. Like with the example in this book, you should experiment with these

TODO note about summary sections.
TODO should I include practice code questions with examples in the back?

How this book is organised

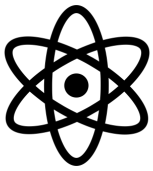
This book is made up of 13 chapters split into three main parts.
Part 1: Chapter 1, 2, 3, 4
These cover the basics of programming and the BASIC programming language.
Part 2: Chapter 5,6,7,8
These chapters cover more advanced concepts and theory and the use of the Commander X16 support chips for graphics and sound.
Part 3: Chapter 9,10,11, 12
These chapters provide the capstone of the book. Theory on the design of games and a complete case study in creating a game and ending in some final words and pointers for extra resources.

Conventional Terms

The Commander X16 may be shortened to X16.
The Commodore 64 may be shortened to C64.
The special Commander X16 logo keys, found on the bottom row next to the Ctrl key, will be called the 'X16 Key'.
Hexadecimal numbers are represented with a leading dollar sign $. There is hexadecimal to decimal conversion table in Appendix XXX
Binary numbers are represented with a leading percentage sign %. If you are not familiar with hexadecimal or binary, do not worry. That will be covered.

Icons used to highlight tips

	This icon is used to highlight a few points which you should help you understand the concept.

	This icon is used to highlight an exercise you can do. Answers or possible solutions to the exercises are in Appendix X **UPDATE**
	This icon is used to highlight technical issues.
	This icon is used to highlight potential problems or issues.

How to type in Programs

Some of the programs in this book will have special control characters. These characters enable the computer to change the colour of text, move the cursor, etc. To make it easier I have used the following conventions.

Program listings will contain descriptive words within square braces which indicate special characters are needed. For example [8 SPACE] would mean press the space bar 8 times, or [2 UP] would mean press and hold the shift key and then the up-cursor key twice.

Table 1 - Control Key Conventions

[SPACE]	Press the space bar key
[UP]	SHIFT key and UP Cursor key
[DOWN]	SHIFT key and DOWN Cursor key
[LEFT]	SHIFT key and LEFT Cursor key
[RIGHT]	SHIFT key and RIGHT Cursor key
[BLACK]	CTRL key and 1 key
[WHITE]	CTRL key and 2 key
[RED]	CTRL key and 3 key
[CYAN]	CTRL key and 4 key
[PURPLE]	CTRL key and 5 key
[GREEN]	CTRL key and 6 key
[BLUE]	CTRL key and 7 key
[YELLOW]	CTRL key and 8 key
[REVERSE]	CTRL key and 9 key

Book website

The code listed in this book is available from the GitHub website.
https://github.com/abc

This web site address will be referred to as the book website. The book website will contain all the program listings as well as errata. This is the web site for this book and will be updated with content when appropriate.

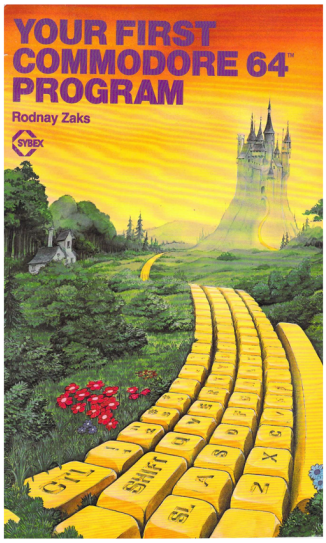
Related Texts

A lot of research was done in creation of this book. Many Commodore 64 programming books from the 1980s were useful. To help the reader I have included the full biblography of all the books, articles, research papers I used.

Some of the chapters touch on modern program design theory. I have included sections discussing these theories, methods and models to assist with BASIC programming. Many of these concepts have entire books dedicated to them.

Many chapters will include a section at the end labelled as "Further Reading". There will be recommendations for books or websites which may cover concepts or code in far more detail.

About the Author



When I was eight years old, my father brought home a Commodore 64. We had a tape deck, and it was connected to an old black and white TV. Dad taught computing in the military and thought they would become essential, and he wanted his kids to understand them. I remember borrowing some of the Usborne children's programming books from the local library in Darwin. These books are now free from the publisher, https://usborne.com/au/books/computer-and-coding-books

My first serious programming book was 'Your First Commodore 64 Program' by Rodnay Zaks. It was colourful and had beautiful drawings, making it very approachable for me. This book helped me learn the fundamentals.

I would spend many hours typing up programs from the 'Compute' magazines. As I got older, I played more computer games. In the 90s, I upgraded to the Amiga 500, then on a 486 PC and so on. During that time, there was still this foundation of wanting to create and make my own stories.

I naturally started working in IT when I was 19 and have continued ever since. I have been a hobby programmer most of my life, and I have published several programming tutorials online.

PART I:
A Call to Adventure

"Knowledge is an unending adventure at the edge of uncertainty." – Jacob Bronowski

Chapter 1 – The Commander X16

"All our dreams can come true, if we have the courage to pursue them."
Walt Disney.

An Overview

The Commander X16 is a modern retro computer envisioned by David Murray. David saw a need for a modern retro-styled computer made with current off the shelf components. A group of another interested enthusiasts joined him in the project. Collecting and programming on real retro computers are fun, but there are drawbacks. The hardware is old and is starting to fail. Few of the old components are made anymore, and finding parts means stripping down old computers. The old computers were designed to plug into monochrome monitors or analogue antenna connectors on TVs. These old monitors and TVs are also becoming hard to collect. Typically, the old computers had design drawbacks to save money or address features of CPU or support chips. These old computers have been studied, documented and highly clever people have worked out ways of stretching the hardware to create programs no one at the time could have imagined. This has raised the standard for new software. To release a game on the C64, you must be a true expert in all programming tricks.

8-bit computers are great for learning about computer architecture and programming because they are small and simple enough to understand with time and study. David's design is based on the Commodore VIC-20 and C64 computers of the 1980s but made simpler, faster, and more capable. Unfortunately, while the Commander X16 is based on Commodore VIC-20/C64, it is not fully compatible with those original computers.

The hardware specifications are.

- A real 65C02 Microprocessor running at 8Mhz
- 64 KB of main RAM
- 512 KB or 2048 KB of banked RAM
- **CHECK** 128 or 512 KB of banked ROM
- VERA Video controller in FPGA.
- 128 KB of Video RAM
- 640x480 or 320x240 pixel
- 256 colours out of a palette of 4096 colours
- 2 layers supporting tiles and bitmap modes
- 128 sprites, the limit of **CHECK** per line
- Sound **TODO**
- Two real 6522 VIA I/O Controllers

The hardware connections are.
VGA (480p) and Composite (480i)
PS/2 keyboard
PS/2 mouse

Two NES/SNES controller
SD Card
Commodore IEC
RS-232
3.5mm audio

To see David's original presentation

The 8-Bit Guy: My dream computer – Part 1
The 8-Bit Guy: My dream computer – Part 2

TODO compute first book of C64 – more than just another computer
The 65C02 CPU can only address 64 Kb of memory. The X16 uses a memory banking scheme to allow it to access much more RAM and ROM memory.

VERA
A discussion about VERA and graphics

References

The Hardware

Central Processing Unit (CPU)

At the heart of the Commander X16 is the Western Design Centre 65C02S microprocessor. Before you begin programming the computer you should become familiar with the processor itself.

The 65C02 is an enhanced version of the 8-bit MOS Technology 6502. The 6502 was originally released in 1975 and sold for 1/6 the price of competing processors from Intel and Motorola. The clock rate was 1 MHz to 3 Mhz. The 6502 was used in many computers and home consoles of the time. Along with the Zilog Z80 processor, it started the home computer revolution of the 1980s.

The main advantages of the 65C02 are significantly reduced power usage, a clock rate up to 14 MHz's The Commander X16 only uses the 65C02 up to 8 MHz due to timing and the use of other support chips. The 65C02 also has an increased number of processor commands, and it also fixes several bugs found in the original 6502. Most importantly for the Commander X16, the 65C02 is still in production today. Microprocessors have a set of instructions that they understand and perform tasks with. Some instructions have different modes, resulting in different operation codes inside the central processing unit (CPU). Each instruction and mode have a collection of electronic circuits specific for that instruction. The 65C02 has 212 operation codes which implement 69 instructions. A modern Intel X86 64-bit microprocessor could have over 3600 operation codes and implement over 980 instructions.

Modern microprocessors are incredibly complex. However, with some time and study, anyone can understand a 65C02 CPU.

The 65C02 has the following features:

- 8-bit data bus
- 16-bit address bus (supplying the address space of 64 KB)
- 8-bit arithmetic logic unit
- 8-bit processor registers
- Accumulator
- Stack pointer
- Index registers
- X register
- Y register
- Status registers
- 16-bit program counter
- 69 instructions implemented by 212 operation codes
- 16 addressing modes, including zero page addressing

How much memory does the X16 have?

There are two types of memory in the X16. Volatile and non-volatile. Volatile memory can only store information when there is power. When it loses power, it loses everything stored in it. Non-volatile memory can keep its contents without power. The X16 has non-volatile memory that is read-only memory called ROM. It also has volatile random-access memory called RAM.

The ROM in the X16 is 64KB in size, which is 64 x 1024 bits, or 65536 bits. **TODO** or is ROM 128KB? The X16 only views 16KB of this ROM at any one time. Currently, about half of the ROM is used to control the computer providing the KERNAL software and BASIC interpreter. The ROM is permanent unless you have extra electronic hardware to write a new ROM chip. The RAM memory is used by us to load and run programs.

Computer memory is made up of memory locations. Each memory location has an address. Each address is a single byte, 8 bits of storage. A bit is the smallest unit of storage, it represents a 1 or a 0. The bit is either on or off. In the X16 memory is arranged into pages. The memory addresses $0000 to $00FF, which has a total of 256 bytes, is page 0. Page 1 is memory address $0100 to $01FF, and so on up to $FF00-$FFFF for Page 255. The CPU in the X16, the 65C02 has a 16-bit address bus which means it can access up to 64kB, 65536 bytes of memory.

The X16 has a total maximum of 2MB of RAM memory, which is 2048KB. Some models have 512KB of RAM memory. The 64KB of memory is available all the time, but there is an 8 KB part of memory that can be swapped or banked out. This trick allows a programmer access

to the remaining 448KB-1992KB. How the Commander X16 memory is mapped out will be discussed in a later chapter.

Memory and storage

The memory hierarchy pyramid helps explain how memory and storage are separated and related. The different layers in the pyramid show how the response time, cost, capacity is related.

TODO compare modern computer memory registers/cache L0-4/RAM//Virtual memory/SDD storage /HDD Disk storage/network storage/Tape storage/offline. Vs Registers / RAM/ ROM/SD storage
**REF Wikipedia / Write Great Code Vol1

Storage

The X16 has a modern industry-standard SD flash card reader built-in as well as a Commodore serial IEEE-488 (IEC BUS) connector. The SD card must be formatted **TODO**

TODO The SD card reader can allow a transfer speed up to 50 MB/s or 400,000,000 bit/s, far more than what the processor can handle.

If you would like a more retro experience the Commodore IEC bus connector allows you to connect a vintage Commodore floppy disk drive to the Commander X16. In theory, the IEC connector can allow up to 50,000 bit/s transfer. Significantly slower than the modern SD card reader.

TODO check compatible 1541 / 1541-2 / etc

Keyboard

The Commander X16 comes with a custom PS/2 keyboard with PETSCII characters printed on the top of the keyboard. PS/2 is an industry standard originally created by IBM and released in 1987. The Commander X16 can work with any PS/2 keyboard.

What accessories are needed for the Commander X16?

You can do a lot on the Commander X16 with any modern TV or LCD monitor which has a VGA connection. When you start to write programs, you will want to save them, so you do not need to re-type them each time you want to run them. To save them you will need an SD card, a 4MB or larger will be suitable.

Exercise - Hardware Components

1. Match the word to the phrase
    a. CPU. Is removable media
    b. SD Card. Stores a program
    c. Keyboard. Is an 8-bit processor
    d. Memory. Executes a program in memory
    e. 65C02. Allows a user to type commands
2. Fill in the blanks.
    a. CPU stands for _____ Processor ____.
    b. RAM stands for _____ _____ Memory.
    c. The CPU can ____ and _____ to RAM.
    d. ROM stands for ____ ____ _____.
3. True or False
    a.

Exercise 1 - Hardware Components

The Software

BASIC

The Commander X16 is using a version of BASIC based on the classic Commodore BASIC V2. The operating system is called the KERNAL and is based on the Commodore 64 KERNAL, with significant improvements and new functions. With ROM banking it is possible for the X16 to have different operating systems.

KERNAL

The KERNAL is the operating system for the computer, and it is stored in ROM. The KERNAL manages the computer and its standard features. It has the software which controls the 40/80-character screen, the keyboard, the mouse, the joypad, the clock, RS-232 serial port and IEC port. It also supplies simple memory management.

Text Graphics

The text display of the Commander X16 which appears when you turn on the computer is 60 lines with 80 characters on each line. It is possible to select different screen modes allowing for 30 lines with 40 characters. There is a screen setting that is like the Commodore 64 which gives 25 lines with 40 characters on each line. The X16 supplies two-character sets. The first set supplies all upper-case letters and graphic characters as well. The second set supplies upper-case and lower-case letters. These characters are based on PETSCII, the PET Standard Code of Information Interchange. Using these characters creatively it is possible to create reasonable graphics without using more advanced features.

Bitmap Graphics

Bitmap graphics allows each dot, or pixel, to be controlled. The X16 has several different graphic modes.
TODO
640x480, 320x240, 320x200, number of colours in mode, palette, layers, etc.

Sprite Graphics

The VERA graphics system in the X16 excels with sprites. Sprites are special graphic objects which can be moved around the screen independently. There can be up to 128 sprites. Each sprite has a width and height, and these can be any combination of four values: 8,16,32,64 pixels. Sprites can be one of two colour modes, 4 bits per pixel (bpp) (16 distinct colours) or 8 bpp (256 different colours). **CHECK** Up to ??? sprites can be displayed on any horizontal line. Each sprite has a pointer that points to a location in memory with the graphical data. The graphic appearance of the sprite. It is possible to change this pointer to a new location. By doing this it is easy to create animation. There are settings to flip the sprite vertically, horizontally or both quickly and easily. Also, it is possible to set if a sprite moves in front of or behind other screen graphics. With these options, sprites on the X16 are extremely flexible.

Music and Sound

TODO

Connectivity

To round out the feature set the Commander X16 has 4 **CHECK** expansion slots. The VERA video card has an SD card slot. On the mainboard, there is a PS2 keyboard/mouse connector, a 6 pin IEC connector as well as a 3.5mm audio jack. There are two W65C22 Complex Interface Adapter (CIA) microprocessors on the mainboard. The CIA chips are used for the keyboard, IEC connector and expansion slots.

The W65C22 CIA microprocessors are made by Western Design Center the same manufacturer as the CPU and are fully compatible with the 65C02 CPU. These chips each have two 8-bit bi-directional peripheral I/O ports, two 16-bit programmable interval timer/counters as well. Versatility is increased with the inclusion of various control registers, interrupt flag register, interrupt enable register and two function control registers.

How to program the X16?

This book will teach you the fundamentals of programming the X16 using the BASIC programming language. BASIC is easier to read and understand than machine code or assembly. Below is an example of a BASIC program on the X16.

```
10 LET X=1
20 PRINT "THE X16 IS THE BEST! TIMES"X
30 LET X=X+1
40 GOTO 20
```

Listing 1 - Example of a BASIC program

Learning how to use the Commander X16

The best way to learn how to use the Commander X16 is to start using it. Use the guide that came with the X16 and become familiar with typing commands. You cannot damage the computer by playing around on the keyboard. Once you feel familiar with the X16 it is time to start reading this book in earnest.

Emulator

While the Commander X16 was being developed there needed to be a way for programmers to gain experience on the platform and for the community to get involved. The developers created the X16 emulator, which is available from www.commanderx16.com. The X16 emulator allows Commander X16 software to be run on Windows, Linux and macOS.

Setup X16 emulator on Windows

Setup X16 emulator on Linux

Setup X16 emulator on macOS

The Commander X16 emulator can be downloaded from.
https://www.commanderx16.com/forum/index.php?/files/file/25-commander-x16-emulator-winmaclinux/&do=download&r=405&confirm=1&t=1&csrfKey=df160969460fb8d40b63549bfaad4f41

The zip file will be downloaded to your download folder and macOS will unzip the file.

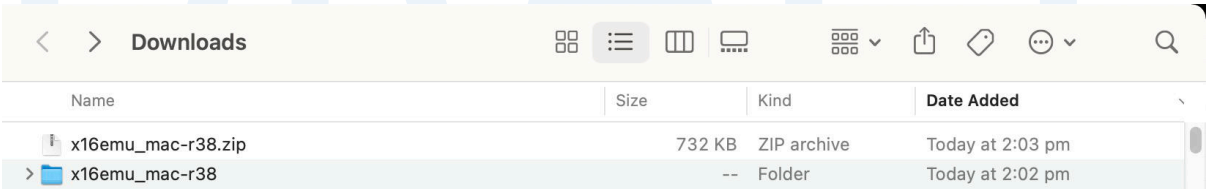


Figure 1 - X16 r38 emulator downloaded on macOS

The folder can be expanded, and you should see the x16emu executable file.

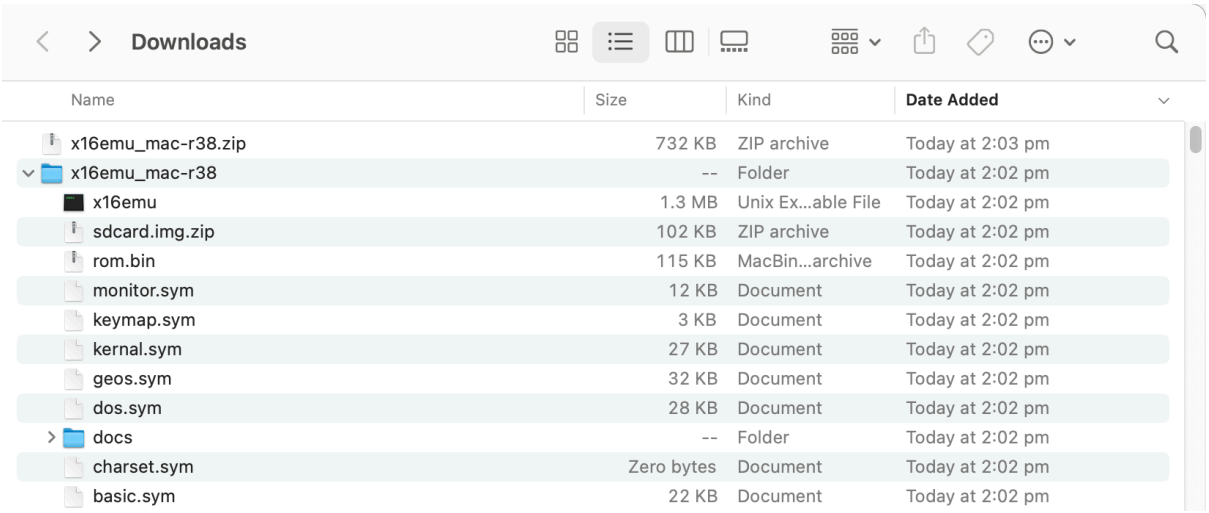


Figure 2 - X16 r38 emulator executable

Newer versions of macOS have a security feature which requires software be digital signed. A pop up will appear letting you know the x16emu software is not signed and cannot be opened.

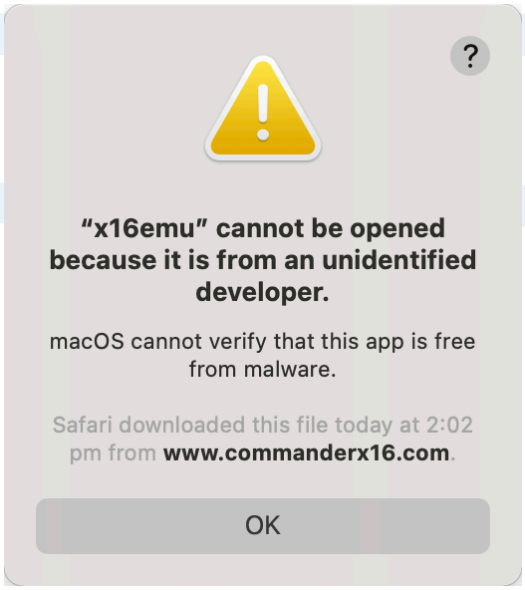


Figure 3 - macOS unable to verify X16 emulator

To run this software, you will need to open the system preferences and go to the "Security & Privacy" control panel.

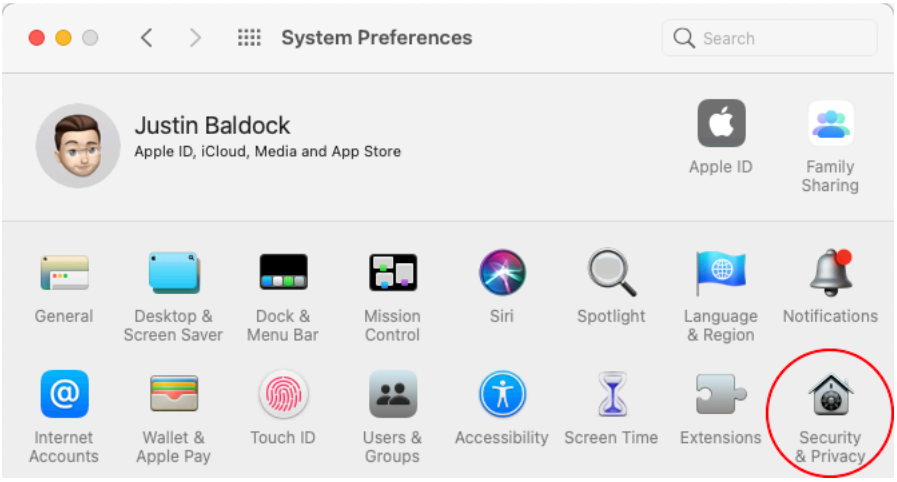


Figure 4 - macOS System Preferences

When you open the “Security & Privacy” control panel you should see a message like the image below. Click on the “Open Anyway”

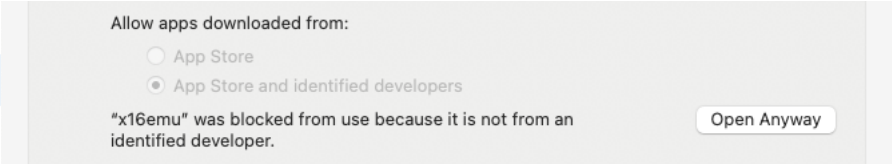


Figure 5 - macOS blocked app

Click on the “Open Anyway” button. Another pop-up will appear which is the final warning, click on the “Open” button to start the X16 emulator.

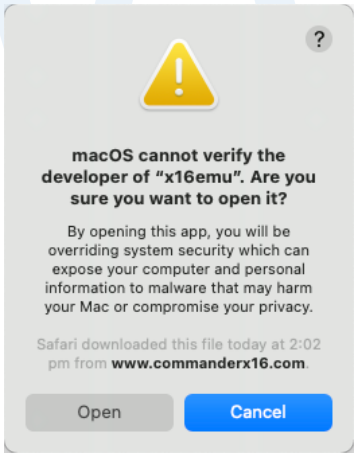


Figure 6 - macOS final enable X16 emulator

The Commander X16 emulator will then open a terminal window and the emulator will start.

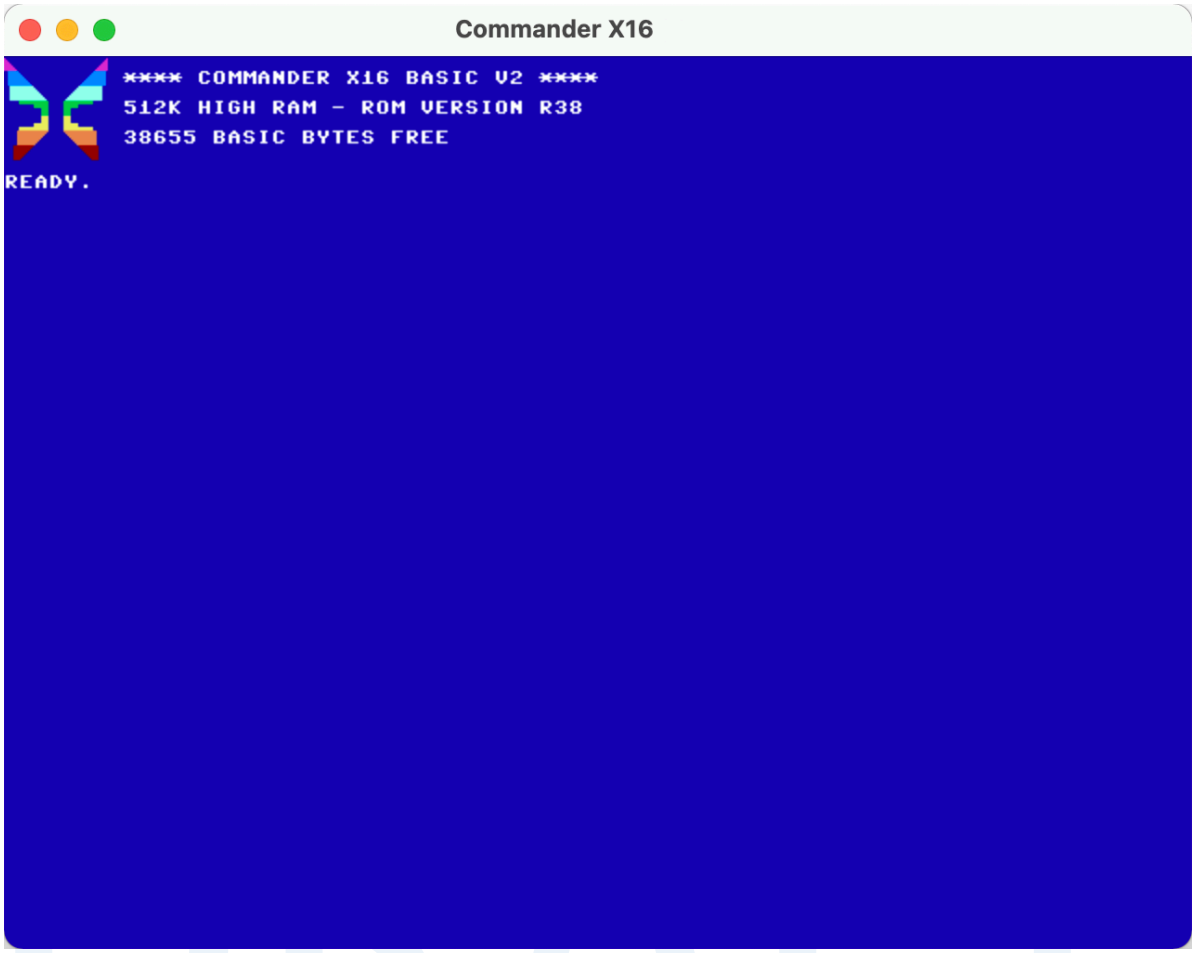


Figure 7 - X16 r38 emulator running on macOS

Software Development Environment

Programming on the X16

TODO

Cross-Platform Development

TODO setting up development tools on Windows / Mac and using emulator/hardware

Further Reading

The Century Computer Programming Course for the Commodore C64 [2]
Compute!'s First Book of Commodore 64 [3]
Programming the 6502 [4]
W65C02S 8-bit Microprocessor [5]
The 65C02 Microprocessor
Wikipedia – PETSCII

X86 Opcode and Instruction Reference [6]

Chapter 2 – First Steps

"The first step is you have to say that you can." - Will Smith

What is Programming?

Today we are surrounded by the results of programming. When you use your computer, use your phone, respond to a social media post on a website, even look at your watch, or read this book on a tablet. People had to create programs on these devices to perform tasks.

But what is programming? Programming is the process of converting an idea into a sequence of instructions or code that a computer can execute. These instructions are specific, and the computer will execute the commands precisely as we arranged them. The sequence is the program. Sometimes the program creates undesired outcomes; these are bugs. Bugs can cause minor problems or crashes. If we make a program with bugs, it is because we misunderstood the problem or the correct use of the instructions. Programming is as much about finding and preventing bugs as it is about converting ideas into cool programs.

For us, programming is problem-solving using two fancy tools, the X16 and our brain. Solving any problem requires three steps.

```
1. Define the goal. Clearly identify the problem
2. What resources are available
3. Apply resources to reach the goal
```

The big trick is you must produce a solution for a given problem. In programming, this solution is referred to as an algorithm and it is a step-by-step method. The process of creating this algorithm is called structured programming. This involved breaking the problem up into smaller and smaller sub-tasks. This process will be covered in much more detail in chapter 6. The algorithm can be expressed in any language.
Consider the below example.

```
1. Remove Commander X16 from the box
2. Remove the power cable from the box
3. Plug the power cable into the wall outlet
4. Plug the power cable into Commander X16
5. Turn on Commander X16
6. Have fun
```

The above algorithm is simple for a human to follow. Unfortunately, 8-bit computers are unable to understand or run ordinary written English. The computer can only “understand” a well-defined subset of language. There are several programming languages available for

the X16. Converting an algorithm into a particular programming language is called programming or coding.

Efficient programming includes the extensive usage of all computing tools including internal registers, memory storage, external equipment, as well as the use of suitable data structures.

Programming also requires a strict discipline of documentation to enable the program to be understood by others and by the author. Internal and external documents must be provided for the software. Internal documentation refers to comments or remark statements made in the body of a program that describes the process. You should make sure there are no "magic boxes". Magic boxes are those processes, or pieces of software where things happen but no one really understands. This is not necessarily a problem, however, when the magic stops, it can make troubleshooting and fix very difficult.

"Any code of your own that you haven't looked at for six or more months might as well have been written by someone else." Eagleson's Law

What is a Programming Language?

Computers understand their processor instruction set or machine code. A computers instruction set is unique to the type of computer processor. When you want to program instructions into a computer, you use a programming language. There are hundreds of different programming languages. Each language has its syntax, rules, and keywords with specific meanings. Some languages are only available on certain devices. The program you create with a language then gets converted into the computer's instruction set, or machine code so that it can be executed. Programming languages can be high level, low level or the grey level in-between.

A high-level language is closer to a human language like English. A high-level language will provide abstraction. The goal of abstraction is to hide the intricate details of something to make it easier to understand and use. Typically, a high-level language does not deal with memory management. High-level languages need a compiler to convert the high-level language into machine code that the computer can execute.

The lowest level of language is machine code. Low-level languages typically do not look like human language at all. Machine code is unreadable to all but experts since it is merely a stream of binary numbers. Low-level languages do not provide abstraction and typically require the user to manage memory. The primary low-level language is assembly. It offers human-readable mnemonics and hexadecimal numbers to represent the program. The code is then assembled into machine code.

The Commander X16 has a few languages available. Assembly is available for the Commander X16. The programming language C is also available, and it is often considered a grey level language as it provides abstraction but requires memory management. The high-level language we will be using in this book is BASIC. The primary reason we will use BASIC is

it comes standard on the Commander X16. No tools are needed to start programming it. The Commander X16 has a BASIC interpreter which will allow our programs to execute.

Machine Code, and Assembly

These binary values stored in memory can then be read by the CPU to perform work. The computer CPU can interpret these values to perform many different functions. This is machine code.

%1010 1001 1100 0000 1010 1010 1110 1000 0110 1001 1100 0100 0000 0000

This does not look easy to read or understand for a person! To make machine code easier for a person to work with we have another number system called hexadecimal which is base 16. Hexadecimal numbers have a $ prefix. When we convert the binary stream into hexadecimal it looks like

$a9 c0 aa e8 69 c4 00

This *still* does not look easy to read or understand for a person! To make machine code easier to read and program the code representing CPU functions or opcodes were given mnemonics. Code written with these mnemonics would then be loaded into an assembler which could convert it into machine code. This language was then known as assembly language. Assembly language is specific to the type or family of CPU. Assembly language on the 6502 families of CPU is different from the x86 or ARM family of CPU. Below is an example of a very short machine code/assembly language program loaded into the X16 memory address $0800

0800: a9 c0 aa e8 69 c4 00

```
LDA $c0 ; Load hex value $c0 into A register
TAX ; transfer the value in A register to X
INX ; increment the value in the x register
ADC $c4 ; add the hex value $c4 to A register
BRK ; we are done
```

TODO What is this translated into BASIC and C?

Memory TODO
Memory stores everything – 64k of low memory, 512kb of high memory, a byte is 8 bits. Discuss bits and binary. Discuss bits vs bytes and kb. Discuss binary vs decimal and hexadecimal.

What is BASIC

BASIC was designed initially in Dartmouth College by John G Kemeny and Thomas E Kurtz, in 1964. BASIC is an acronym that stands for Beginners All-purpose Symbolic Instruction Code. At the time computers required software to be written to do anything. This was a task often done by mathematicians and early computer scientists. The authors of BASIC wanted students at any school of study to be able to access and use computers. The main philosophy of BASIC is the ease of use. With computers becoming more popular in the late 1970s, Microsoft developed Microsoft BASIC 1975. Commodore licensed Microsoft BASIC for use on their range of computers starting in 1977. The Commodore 64 came with Commodore BASIC v2.0. The Commander X16 is using a version of BASIC which is mostly compatible with Commodore BASIC 2.0.

Commodore released seven versions of Commodore BASIC. There were almost 30 different variants of Microsoft BASIC. Also, at the time, there were complete other families of BASIC from Atari, BBC etc. We will focus on the Commander X16 BASIC V2 which incorporates all the features of version 2 with some extra commands and features.

Fundamentally BASIC is a kind of translator between your words and the instructions that the computer can understand. Once you know what you want the computer to do, it is simple to create BASIC instructions for it to follow. By following the standards of the language, the computer deciphers your program and carries out those instructions.

Reasons to learn BASIC.

There are a few good reasons to learn BASIC. The main reason is that learning how to programs helps you get more out of your Commander X16. You will be able to modify existing programs to suit your needs better. It is a steppingstone to more advanced programming languages.

Computers play a massive role in our society today. They are in every electrical consumer device we use, from building air conditioning management systems to our doorbells and our children's toys. Knowing how they work will make adapting to our changing world easier for you. You are furthering your education by learning about computers, and you will have an advantage over those people who are not keeping up with progress.

In the end, computers can be fun. Your Commander X16 is a great entertainment machine, with fast action, colour display with excellent resolution (for a modern retro 8-bit computer), and music. However, if you stop and think about it, every time you play a game, you are enjoying the product of someone else's imagination. A computer can perform math calculations, display colourful images, and produce sounds. But to transform these computational operations into an entertaining game, a person with original thinking is essential. The computer is lacking in imagination. That imagination can come from a professional game programmer, or it can come from you. By learning to program, you can turn your thoughts and ideas into reality in ways that are not possible before. This computer enables you to express your creativity.

Interacting with the X16

Unlike modern computers, which use a mouse or a touch screen, most of your interaction with the X16 will require the use of a keyboard. When you turn on the Commander X16, a message is displayed in letters and numbers to welcome you. These and other symbols which the machine can display are called characters. You will also see a square that flashes, known as the cursor. By typing into the keyboard, you can put your characters on the screen.

The corresponding character appears on the screen, at the cursor's location, each time you press a key. Afterwards, the cursor travels one spot to the right. To type in the punctuation characters shown on the digit keys above the numbers, hold down one of the SHIFT keys while typing the digit keys. Using SHIFT with letter key triggers characters in the graphics.

You will have one screen row filled out after you type 80 characters. When the cursor moves off the screen's right edge, it will reappear at the left side, one row lower. That is called wrap-around. Typically, you only type a few characters on a line, then click the RETURN key. Pressing the RETURN key makes the cursor move back to the screen's left edge, one row lower once again.

When you press RETURN without typing any characters, it merely moves one row downwards. Do this as many times as you can, and the entire display will move upwards. This is called scrolling, and the horizontal wrap-around is the vertical counterpart. The screen scrolls up. The characters that scroll off the screen cannot be retrieved.

If you type something and then you press RETURN, the computer displays the message "?SYNTAX ERROR", do not worry about it at this point. Syntax refers to BASIC-language grammar. When you type something that does not belong in the language, the computer displays this error message.
You can get some additional graphics characters by holding down the Commander key while entering the letter keys. Entering a digit key while holding down the Commander key will change the colour of the cursor. Now all typing is in the new colour. Keep down the key labelled CTRL when typing a digit key to get more colours.

Some keys do not put characters on the screen. Instead, they are making the cursor move. The four cursor keys can move the cursor in all four directions, the space bar key creates a space, etc.

Numbers

Decimal

We learn to count on our hands, and we use the numbers 0 to 9. We use decimal which has 10 different symbols to represent numbers, this is base 10. We learned at a young age that the position of the numbers represents a different power of 10 values. E.g., How many 1s, how many 10s, how many 100s. Decimal is a positional number system.

10^3 (1000)	10^2 (100)	10^1 (10)	1	Notes
1	0	2	4	The number 1024 has 1 x 1000 + 0 x 100 + 2 x 10 + 4 x 1.

0	0	1	6	The number 16, has 0 x 1000 + 0 x 100 + 1 x 10 + 6 x 1

Table 2 – Example of Decimal numbers

Binary

Most modern computers are binary computers and contain many electronic circuits and transistors. All these components can be in one of two states. High power or low power. 'On' or 'off'. True or false. Everything that happens in the computer is based on the possibilities of the components being in those states. True or False, 1 or 0, Binary. With binary, there are two numbers, so it is base 2. The position of the number represents different values than what we are used to. The position represents a different power of 2 values.

2^7 (128)	2^6 (64)	2^5 (32)	2^4 (16)	2^3 (8)	2^2 (4)	2^1 (2)	1	Notes
0	0	0	1	0	0	0	0	The number is 16. We see there is 0 x 128 + 0 x 64 + 0 x 32 + 1 x 16 + 0 x 8 + 0 x 4 + 0 x 2 + 0 x 1
1	1	1	0	0	1	1	0	The number is 230. We see there is 1 x 128 + 1 x 64 + 1 x 32 + 0 x 16 + 0 x 8 + 1 x 4 + 1 x 2 + 0 x 1.

Table 3 – Example of Binary numbers

When we look at a small part of the computer's memory, we have 8 transistors. We can describe the state of those electronic components with numbers. Each of the transistor circuits or "bits" are 'on' and some are 'off'. E.g., 11100110. That looks like a number, and it is. It looks like a decimal number but in this case, it is a binary number. To prevent confusion in this book binary numbers will have a % prefix.

Converting decimal to binary is simply a matter of subtraction. We simply see if we can subtract the binary representative values from our decimal number. Consider the decimal number 175. If we subtracted the binary value of 256 it would give us a negative, so that does not work. But we can subtract 128 from 175 and that leaves us 47. The value 64 is too large, but we can subtract 32. 16 is too large but we can subtract 8 leaving 7. We can see from here we can subtract 4,2 and 1 leaving us zero. We can now see the binary number %10101111.

2^7 (128)	2^6 (64)	2^5 (32)	2^4 (16)	2^3 (8)	2^2 (4)	2^1 (2)	1	Notes
1	0	0	0	0	0	0	0	175 – 128 = 47. We place a 1 under 128.
1	0	1	0	0	0	0	0	47 – 32 = 15
1	0	1	0	1				15 – 8 = 7
1	0	1	0	1	1	1	1	7 – 4 – 2 – 1 = 0

Table 4 – Convert decimal to binary

Hexadecimal

Often when working with computers we use hexadecimal numbers. Hexadecimal numbers are base 16. Hexadecimal uses the symbols 0 to 9 like decimal, then it uses A to F from the alphabet to represent values 10 to 15. To prevent confusion in this book hexadecimal numbers will have a $ prefix. Often in other texts, you will also see the prefix of 0x. Hexadecimal is also a position value system.

16^3 (4096)	16^2 (256)	16^1 (16)	1	Notes

0	0	1	6	The number $16 has the value of 1 x 16 + 6 x 1 = 22
0	A	C	E	The number $ACE has the value of A (10) x 256 + C (12) x 16 + E (14) x 1 = 2766

Table 5 – Example of Hexadecimal numbers

Nowadays it is easy to open a calculator app and use programming mode which offers an easy way to convert decimal to hexadecimal to binary. However, it can be useful to understand how to convert manually. Consider that hexadecimal base 16 and first 4 value positions in binary. The below example will convert a decimal number to binary and then to hexadecimal.

Consider the decimal number 213. To convert to binary, we look at the represented values of binary. 256 is greater than 213. However, 128 can be subtracted from 213 and leaves 85. We can subtract 64 and it leaves 21. We cannot subtract 32, but we can subtract 16 leaving 5. We can then subtract 4 which leaves 1. We see this gives us the binary number of %11010101. Now to convert to hexadecimal we focus on the first 4 values, $0101. This has a value of 5, which is our first hexadecimal number. We then look at the next 4 values, %1101, but we consider them to represent values of 1,2,4 and 8. This part then represents a value of 13, which is $D. Converting 213 to hexadecimal value $D5. We can convert binary to hexadecimal very easily now that we understand each hexadecimal number represents 4-bits of the binary number.

2^7 (128)	2^6 (64)	2^5 (32)	2^4 (16)	2^3 (8)	2^2 (4)	2^1 (2)	1	Notes
1	1	0	1	0	1	0	1	Decimal 213 converted to %11010101
				2^3 (8)	**2^2 (4)**	**2^1 (2)**	**1**	
				0	1	0	1	These 4-bits have a value of 5. Our first hexadecimal number is 5.
				5				
2^3 (8)	**2^2 (4)**	**2^1 (2)**	**1**					We then look at the next 4-bits, however we consider them to have values of 1,2,4 and 8.
1	1	0	1					These 4-bit have a value of 13. Our next hexadecimal number is D.
D				5				213 converted to hexadecimal is $D5

Table 6 – Convert Decimal to Binary to Hexadecimal

Octal

Another numeral system seen with computers is Octal, which is base 8. It uses the digits 0 to 7 just like decimal. This system was used before computers by several different cultures who counted on the spaces between fingers. It was also seen as a handy system for dividing items into halves and quarters. The system became more widely used in computer science with the early mainframes which used 6-bit, 12-bit, 24-bit and 36-bit words. The use faded away when 8-bit and 16-bit computing became the norm. To prevent confusion octal numbers in this book will have a & prefix. In other texts, you may also see the prefix 0o (Zero, small letter o)

Octal has a close relationship with binary like hexadecimal. We can convert binary to Octal easily when we consider that each octal number represent 3-bit of the binary number. While we will not use Octal in this book it is still handy to be aware of. It is still used today in some programming languages.

2^8 (256)	2^7 (128)	2^6 (64)	2^5 (32)	2^4 (16)	2^3 (8)	2^2 (4)	2^1 (2)	1	Notes
0	1	1	0	1	0	1	0	1	Decimal 213 converted to %11010101
						2^2 (4)	**2^1 (2)**	**1**	
						1	**0**	**1**	These 3-bits have the value of 5. The first octal number is 5.
						5			
			2^2 (4)	**2^1 (2)**	**1**				
			0	**1**	**0**				These 3-bits have the value of 2. The second octal number is 2
			2						
2^2 (4)	**2^1 (2)**	**1**							
0	**1**	**1**							These 3-bits have the value of 3. The third octal number is 3.
3									
			2			**5**			213 converted to octal is &325

Table 7 – Convert Decimal to Binary to Octal

Summary - Numbers

- Humans normally use Decimal
- Decimal has ten digits. The digit position in the number represents a value power of ten
- Computers normally use Binary
- Binary has two digits. The digit position in the number represents a different value of the power of two.
- Octal has eight digits.
- Hexadecimal is particularly useful for converting between binary and decimal
- Hexadecimal has sixteen digits.

Exercise - Numbers

4. Convert the following binary numbers to hexadecimal, then to decimal.
    a. 0011 1101
    b. 0110 1011
    c. 0010 0101
    d. 0110 1110
    e. 1110 0100
5. Convert the following decimal numbers to hexadecimal, then to binary.
    a. 16
    b. 100
    c. 7
    d. 229
    e. 131

Exercise 2 - Numbers

Bits, Bytes and Words

Bit

A ‘Bit’ is the smallest unit of storage, it represents a 1 or a 0. We know that this is a binary value.

Byte

Early computers typically used 8-bits to encode a text character. The term ‘Byte’ was coined and represented 8-bits. A byte being equal to 8-bits was not always the case, in the 1960s there were computers that had 6-bit or even 9-bit bytes. However, a byte being 8-bits became the de facto standard partly because it is a convenient power of two. As a result, the majority of computing architectures use 8-bit bytes. The symbol used to represent a byte is an upper-case B. With 8 bits in binary it is possible to represent the values from zero to 255, or 256 different numbers. The value of a byte is often written as a decimal number (from 0 to 255) or as a hexadecimal number (from $00 to $FF).

Nibble

When you only take half a bite of a sandwich, you have had a nibble. When you only access 4 bits or half a byte, you are working on a nibble. A byte is made of a low nibble and a high nibble. The low nibble is the 4 least significant bits. The high nibble is the 4 most significant bits.

kB vs KiB?

During the 70s as computers storage and memory increased the industry added the metric term of kilo (meaning 1000) to Byte to represent 1024 bytes since 1000 is close to the binary 1024. Then there was some interesting marketing. Companies could release a device that could store 256,256 bytes and it would be advertised as 256 kB. While another vendor would have a device advertised as 256 kB and it held 262,144 bytes. To solve this issue in the late 90s updated terms were created by the IEC: kibibyte, mebibyte, gibibyte.

Value	Metric	Value	IEC
1,000 Bytes	1 kilobyte = 1 kB	1,024 Bytes	1 kibibyte = 1 KiB
1,000,000 Bytes	1 megabyte = 1 MB	1,048,576 Bytes	1 mebibyte = 1 MiB
1,000,000,000 Bytes	1 gigabyte = 1 GB	1,073,741,824 Bytes	1 gibibyte = 1 GiB

Table 8 – Metric vs IEC multiple-byte units

Word

When reading about bit and bytes you may also see the term 'Word.' A word represents a certain number of bits, and it is dependent on the computer architecture. A word is a fixed size piece of data used by the CPU. Typically, it is the size of the largest piece of data that can be stored in the CPUs (Central Processing Unit) internal registers.

As computers became more powerful, going from 16-bit to 32-bit and then 64-bit, the hardware developers wanted to maintain backwards compatibility. The concept of doublewords and quadwords were created. A word was kept at 16-bit and a doubleword was 32-bit and a quadword was 64-bit. The value of a word is often written as a decimal or hexadecimal based on how the value is going to be used. E.g., When dealing with a word that contains a memory address is it common to use hexadecimal.

Term	Bits	Hexadecimal range	Number of values
Word	16-bits	$0000 - $FFFF	65,535
Doubleword	32-bits	$0000 0000 - $FFFF FFFF	4,294,967,295
Quadword	64-bits	$0000 0000 0000 0000 - $FFFF FFFF FFFF FFFF	281,474,976,710,655

The CPU used in the X16 is, for the most part, an 8-bit computer. Because the internal registers are 8-bit technically the word size is 8-bit. However, it does have a 16-bit address bus and a 16-bit program counter. So, the term 'Word' may be used to describe a memory address or a piece of data in the program counter register. Often when dealing with an 8-bit word it is just easier to refer to the data as a byte.

Endianness

When we read this sentence, we read from left to right. It is a convention for many of us as English is written in lines starting on the left moving to right, progressing from top to bottom. But consider other written languages. Arabic script is written from right to left, top

to bottom. Japanese kanji is written in columns from top to bottom, then left to right. People have diverse ways to read and write text.

Computers, as it turns out can store and read data in different ways as well. How it does stores numbers is based on its CPU. When a computer stores a byte, it stores it the way we think it should, from largest value bit to smallest. The value 255, $FF, would be stored in memory as %11111111. The value 16, $10, would be stored in memory as %00010000.

But when dealing with a word that has two bytes, which byte should be written first? Consider the value 65,000. The hexadecimal value is $FDE8 and it is made of two bytes. Which byte should be written first? Should $FD, being the most significant part of the number be written first in a lower memory location? Or should it be written in a higher memory location? The byte $E8 is the least significant part of the number. But where is the least significant or 'end' of the number going to be stored?

Endianness is simply the order in which bytes in a word are sorted. Big-endian store the most significant byte first in the lower memory address. Little-endian store the least significant byte first in the lower memory address. Most modern computers with CPUs from Intel and AMD are Big-endian. The X16 with the 65C02 CPU is a little-endian system.

Variables

Programming involves handling and processing information and data. When we make a game, we need to track many pieces of data, like the player's score, how many lives they have, the player's location on screen etc. A variable is simply a container that stores a value. When we create a variable, the computer allocates part of its memory to store it. A variable is a symbol used to represent a piece of data. A variable's value can change based on what operations the program does to it. The contents of the variable's container are variable.

Data Types

When we create a variable, it is important to tell the computer what type of data will be stored. Distinct types of data require different amounts of memory and allow different types of operations. Adding two numbers together is different from adding two strings together. In modern programming, there are many different types of data. Some common types are listed in the below table.

Table 9 - Common Data Types

Integer (int)	The most basic type of number data. It is used to store integer numbers, numbers without a fraction component. E.g., 16 or 2021.
Floating Point (float)	This is also a type of number data. A floating-point number has a fractional component. E.g., 128.5 or 3.14159265
Character (char)	Used to store a single letter, symbol, digit.
String (str)	A string is a sequence of characters. Normally it is used to store text. A string can contain numbers; however, it will be treated as text.
Boolean (bool)	A simple type that can only hold one of two values, true or false

Enumerated type (enum)	Contains a set of predefined values. The values can be text or numerical and are known as elements. Typically, values can be stored and retrieved using a number index.
Array	Also known as a list. Array's stores a number of elements, normally of the same type of data. E.g., an array of strings. Each element in the array is accessed using an integer index (0,1,2…n). The total number of elements is the size of the array.
Date	Most languages will store a date in YYY-MM-DD, ISO 8601 format.
Time	Some languages can store time in a HH:MM: SS format.

There are also four other terms relating to data types; Strongly typed, weakly typed, statically typed, dynamically typed. These four terms all relate to how strict a programming language is about data typing. The concept is not new, people have been writing about these concepts since 1974. Strongly typed has been defined as "Strong type checking means that whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function."[7] A weakly typed language makes conversion between unrelated data types when required. Statically typed means the language will check the data type rules at compile time. Dynamically typed means the language will check the data type rules as it is running. BASIC is not compiled so it is not a statically typed language. These concepts are not hard and fast. In fact, they are often presented as a graph.

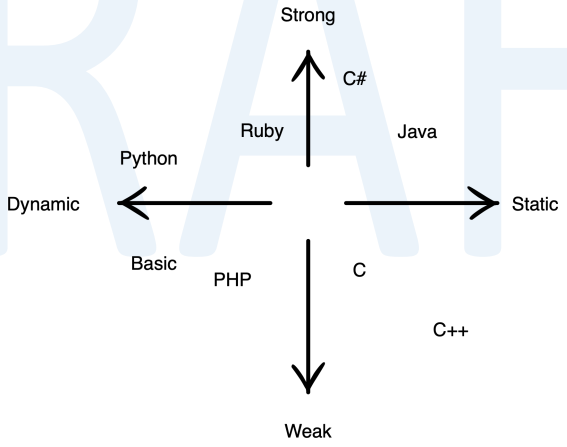


Figure 8 - Classification of data typing

In other words, some programming languages require that you define the exact variable type when you create it, and the type of data cannot be changed. These are languages are strongly typed. Other languages can be weakly typed, and you can create a variable and store any type of data you want in it. Some languages are stricter or more relaxed than others.

BASIC Data Types

BASIC only understands a few different types of variable data. BASIC can work with strings of text, integer numbers, float numbers and arrays. BASIC does impose limits on how large a variable can be based on the type. A string can contain up to 255 characters. An integer is any whole number between -32768 and + 32767. Float numbers can be very large between

+- 2.93873588E-39 and +-1.70141183E38. 1.70141183E38 is written as a standard number. To write it out for a non-programmer to see it would look like; 170,141,183,000,000,000,000,000,000,000,000,000. Needless to say, but floats can be very large numbers. Array are lists of data. When an array is created its size and the data type it will hold are declared.

Table 10 - BASIC Data Types

Integer (int)	Any whole number between -32768 and + 32767.
Floating Point (float)	Very large numbers between +- 2.93873588E-39 and +- 1.70141183E38.
String	String can contain up to 255 characters.
Array	Can store a single type of data, Integer, Float or string.

Variable Rules

With any language, there are rules about variables. In BASIC the names of variables must follow the below rules.

- The first character must be alphabetic. (Range A-Z)
- The 2nd character may be alphanumeric. (Range A-Z and 0-9)
- Any more alphanumeric characters are allowed but are not considered part of the variable name
- The 2nd or 3rd character can be % to represent an integer or $ to represent a string variable
- The 2nd or 3rd character can be (to represent a subscripted variable
- A name cannot include reserved words since the BASIC interpreter will treat them as keywords.

BASIC can-do automatic conversion between different types of numeric variables. However, converting a string to a number and vs versa requires a function.

From the rules we see variables can have suffixes to distinguish what type of variable they hold. % for integer data or $ for string datae. A numeric variable has no suffix and can hold many values. E.g., A variable for Players Gold, PG holds a value of 123456789.1234
An integer variable has the suffix of % sign. An integer variable can only hold whole numbers, no decimal places. They use less memory in the computer. E.g., PL% or S7% holding a value of 32000. A string variable is used to store 'strings' of letters or numbers. This is the variable type used for handling text. It has the suffix $ sign. E.g., A variable, string 1, as S1$ holding a value of "Ready Player One"

It is worth noting that naming variables in a meaningful way is considered difficult with modern languages. Typically, you try to create short, descriptive names which make the code easier to understand. With BASIC this is not possible. I recommend creating variables at the beginning of the program and describing them with a REM statement. A detailed

comment about what the variable is for and how it is used. This will take up more space, however, while the software is being developed it is very useful.

Before, We see a list of variables where we may guess what they are used for.

```
10 X=100; A=200; L=3; W$="WELCOME"; PN$="COMMANDER JAMESON"
```

Listing 2 – Example of variables

After We see a list of variables with some description, so we know exactly what the variables are for.

```
10 LET X=100: REM PLAYER CURRENT X ORDINATE
15 LET A=200: REM PLAYER CURRENT AMMO
20 LET L=3: REM PLAYER LIVES REMAINING
25 LET W$="WELCOME": REM MESSAGE TO DISPLAY
30 LET PN$="COMMANDER JAMESON": REM THE PLAYERS NAME
```

Listing 3 – Example of variables with description

Constants

A constant is similar to a variable. However, the value is set at the beginning of the program and remains constant. They are often used for testing or setting boundaries. An example from a game could be the constant max bullets, LET MB=10 so before that spaceship fires another bullet it checks to see how many bullets are flying around and makes sure it is less than MB before creating another one.
There are also integer constants. Integer constants are whole numbers (numbers without decimal points) Integer constants must be between -32768 and 32767. Integer constants do not have decimal points or commas between the numbers. If the minus (-) sign is left out, the constant is assumed to be a positive number. Zeros coming before a constant are ignored and should not be used as they will use memory.

Do not use a comma in a number.

```
LET X = 32,767
?SYNTAX ERROR
READY.
```

Virtual screen 1 - Syntax Error with LET

Using a comma will create a ?SYNTAX ERROR, as seen in the above example. Without the comma it will work.

```
LET X = 32767
READY.
```

Virtual screen 2 - Example of LET

The value of an integer cannot be changed. The integer constant 5 has a value of 5.

```
LET 5 = 2
?SYNTAX ERROR
READY.
```

Virtual screen 3 - Constants cannot be changed

The above example generates a ?SYNTAX ERROR

True and False

In BASIC on the Commander certain operations will return a True or False. The value of -1 represents True and the value of zero represents False. This is very different than other programming languages.

To see this behaviour, we can do some basic maths with the PRINT command. We will print the result of 1 is greater than zero, which is true. We see the value -1 appear. When we print the result of 1 is smaller than zero, which is false. We see the value zero appear.

```
READY.
PRINT 1>0
-1

READY.
PRINT 1<0
 0

READY.
```

Virtual screen 4 - Example of True and False

In BASIC on the X16, the value -1 represents True.

Mathematical Operators and Precedence

The computer supports a number of mathematical operations. The symbols used to indicate the operation is called an operator. + Addition, - Subtraction, * Multiplication, / Division and ^ Exponentiation. Computers just like people cannot figure out how to divide by zero so if you try this in BASIC you will get the message “?DIVISION BY ZERO ERROR”

Operators can be mixed on a single line as the below example.

```
PRINT 2*3+4
 10
READY.
```

Virtual screen 5 – Multiple operators on single line

However, it is important to remember that certain operators have precedence over others. Operators which have the same level of precedence are checked in order of left to right.

^ Exponentiation	
+ positive number	• Negative number

• Multiplication	/ Division
+ Addition	• Subtraction
<= Less than, equal to	• Greater than
NOT logical NOT	
AND logical AND	
OR logical OR	

The table lists the three binary operators NOT, AND, and OR which we will discuss later.

Functions

A function is sometimes called a subroutine, method, procedure or subprogram. With BASIC we will use two terms, function and subroutine. Both are created from lines of code and serve the same purpose, to save time and space by allowing us to reuse code.

A function is a sequence of programming instructions that perform a specific task and return a value. This function can then be called whenever that task should be done. A function may require variables to be given to it to perform the particular task. For an example a function that when given a number value as a variable will calculate and return the amount of government tax required.

A subroutine is a sequence of instructions that perform the same task. For an example, clear the screen of text and display a menu.

In BASIC some functions return a value dependent on the variable given to them. For example, the numeric function SQR (number) returns the square root of the number passed to it. The string function LEFT$(“string”,2) will return the left 2 characters from string.

TODO

Expressions

A numeric expression is an arrangement of numbers, number functions, integer and real variables, operators or logical expressions.
E.g., PRINT 2+3*4
Because multiplication has higher precedence than addition the computer would multiply 3 and 4 and then add 2 giving us the result of 14.

We can change the order of precedence by using parentheses (). Parts of an expression in parentheses are evaluated first.
E.g., PRINT (2+3)*4
This time the computer will add 2 and 3 then multiply by 4 giving us the result of 20

Numbers
TODO Need to write something about Decimal numbers, vs $ hex, vs % binary

Learn the language

If you are truly interested in programming, then you will end up learning many different languages over the years and decades to come. But for now, read and understand all of the BASIC keywords. Create little programs to explore and use them all. Exploring and reading other books, websites, tutorials on BASIC and other languages will help you. You will expand your skills and gain different views and ideas on how to accomplish tasks. You may find, like I have that when returning to BASIC you see new ways of using this old language.

Join the community

There are no programming gurus. A 'guru' is someone who analyses problems systematically. Often, they are people who have spent years learning programming languages and patterns. It is from this experience they can appear to magically know solutions. To receive help from these 'guru's you need to include your code, explain what you've tried and understand that they are human to. You will not necessarily get answers. You may work things out yourself when you sit down to explain to others what is happening. The more you learn and practice the better you will become, moving along the continuum from 'newbie' towards 'guru' yourself.

Chapter 3 – Introduction to BASIC

"Programs must be written for people to read, and only incidentally for machines to execute." - Harold Abelson

Beginning BASIC Statements

Numbers, Operators, Expressions and Precedence.

When you see the prompt READY, it means that the computer is waiting for you to type something. In the previous section, you always type a line and press RETURN. This means you type something that your PC cannot understand. A syntax error is a problem. Here is a line the computer is going to follow. Type this line and press RETURN

```
PRINT 64 + 16
 80
READY.
```

Virtual screen 6 – Example of addition

The computer has responded by displaying the answer, 80, on the screen. The computer can also handle more difficult problems.

```
PRINT 64 + 16 + 512 + 128 + 4096
 4816
READY.
```

Virtual screen 7 – Example of multiple additions

Now you see why you need the RETURN key. It allows the computer to know when a line is complete. It also allows you to fix errors before the X16 interprets the line. From now on, it means you must type a line into it and press the RETURN key. The computer supports adding, subtracting, multiplying and division operations. One of these operations is referred to as an operator as a symbol. The plus sign was used in the examples. Enter these three lines to view the others

```
PRINT 64 - 16
 48
READY.
PRINT 16 * 4
 64
READY.
PRINT 512 / 64
 8
READY.
```

Virtual screen 8 - Example of subtraction, multiplication and division

Since alphabet letters have a particular computer meaning, X or x cannot be used for multiplication so; instead, the* asterisk is used. For division, the slash symbol is used. Division by zero, incidentally, is illegal, so if you try to divide the computer by zero, you will get the Division by Zero error.

There is another operation, which is exponentiation. On the X16 keyboard is the symbol (up arrow) which is SHIFT + 6. It is not the Up-arrow cursor key on a PC keyboard is it the ^ symbol.

```
PRINT 2 ^ 4
 16
READY.
```

Virtual screen 9 – Example of exponentiation

Exponentiation shall be used to increase a number to a power. The line above, as in 2 * 2 * 2 * 2, means multiplying the number 2 by itself four times.

On a single line, operators may be mixed. The result is referred to as an expression. The computer evaluates an expression to produce a response or value.
2* 3 + 4 When using expressions, it is important to remember those individual operators have precedence over others. Here is a list showing the operator hierarchy, beginning with the highest precedence. The operators with the same precedence are assessed in order, from left to right.
^ exponentiation
*,/ multiplication, division
+,— addition, subtraction

Please enter this line to see precedence in action.

```
PRINT 2 + 3 * 4
 14
READY.
```

Virtual screen 10 – Example of precedence

Since multiplication takes precedence over addition, the computer first multiplies 3 and 4, and then adds 2 to the result. The proper response to this expression is 14.

Using parentheses, you can change the order in which an expression is assessed. Parts of the expression in parenthesis are always evaluated first.

```
PRINT (2 + 3) * 4
 20
READY.
```

Virtual screen 11 – Example of parenthesis

This time, the computer will add 2 and 3, then multiply the result by 4, and display the answer, 20.

Keywords and statements

Now we should take a moment to notice the organisation of statements. The computer scans first thing in the line when entering a line and pressing Return to see if the word is

recognised as a statement or command. If so, the computer does what it is commanded by the statement.

There are certain statements that require additional information after the declaration name. For PRINT, one example of this additional information is an expression.

Here is a statement typical of PRINT

```
PRINT 16 * 64 + 128
 1152

READY.
```

Virtual screen 12 – Example of PRINT

PRINT is the keyword for this statement. Every word BASIC knows is called a keyword, and all BASIC statements must start with one keyword.

If the computer does not recognise the first word in a line, it will report a "?SYNTAX ERROR". For example, if you type a line that starts with the word "SHUTDOWN", the computer will respond with the error message. The X16 does not have a keyword of SHUTDOWN.

The X16 expects perfect spelling. If you make a mistake by typing a keyword, even though you only have one letter off, the computer will respond to the problem with a "SYNTAX ERROR". "PRITN64" won't work, therefore, and "PRINU64" will not work either. For the computer to recognise the keyword, every letter must be correct.

You do not have to be so fussy in spaces, on the other hand. The following are legal and workable lines.

```
PRINT64
 64

READY.
PRINT          64
 64

READY.
PRINT 2 + 3*4
 14

READY.
```

Virtual screen 13 – Example of legal whitespace

The computer does not recognise and ignores any space. However, it cannot be used inside a keyword. "P R I N T" will not be interpreted by the computer. Using the spacing carefully, simply makes reading the line easier. The second line below is easier to read than the first.

```
PRINT(64+16)*(128+512)
 51200

READY.
PRINT (64 + 16) * (128 + 512)
 51200

READY.
```

Virtual screen 14 – Example of white space

It is possible to abbreviate most keywords. Knowing these short-cuts can make typing BASIC programs quicker. Personally, I only know a few short-cuts. The X16 will convert the short-cut before the statement is stored in memory or processed.

Summary - Statements

- Every statement begins with a keyword. Some statements need to know more about the keyword Maximum file size is 4GB
- The PRINT keyword requires a number or expression which tells the computer what to print on the screen
- Some keywords can be abbreviated
- Keywords should be correctly spelled. You cannot insert spaces inside a keyword
- Spaces can be put in a line virtually everywhere else. Spaces are used to make reading a line easier

PRINT statement

So far, we have used the print statement with numbers and operators to display the answer on the screen. The term print remains in place from the days before video display when the display device on the computer was a printer. Nowadays, we can use or view the information using our HDTV or monitor, and PRINT means “display on a screen.” When the computer displays to the screen, it is said to be printing to it. The PRINT statement has a short form which is the question mark character (?). We can retry an earlier example to see it work.

```
? 64 + 64
 128
READY.
```

Virtual screen 15 – Example of short form PRINT

Many BASIC statements have short forms. The purpose is the ease of typing. By typing “?” you are typing four fewer letters.

Summary - PRINT

- The PRINT statement enables the computer to display on the screen
- The abbreviated form is "?"
- This is a statement commonly used in BASIC programming
- The computer sees the question marking as the same as the PRINT declaration

POKE Statement

POKE is often used as it allows the contents of a memory location within X16 to be changed. You are in effect 'poking' at a value in memory to change it. A variety of things can be done by POKE. You use POKE, for example, to change your X16 register or variable values. The syntax of POKE as a keyword is (POKE), followed by a number (or expression), a comma, and finally, the second number (or expression). The first number indicates the computer's memory or hardware location. Memory settings are used to store information such as software, display screen and system software.

On vintage computers like the Commodore 64 or VIC 20, the POKE command was especially powerful as it allowed you to change values in memory to create graphics. On the Commander X16 we cannot POKE into video memory, we have the VPOKE command, which will be discussed later.

The Commander X16 can access a total of 65536 addresses. The data or value in each address is between 0 and 65535. Position 0 is the first address. This idea may be a little unfamiliar because usually, you start with 1 when you count. Remember, zero is also a number. And you start with zero when you count memory addresses or the values in them.

The purpose of each address can be different. For example, address 40800 is used by the VIA#1 I/O Controller. Another example is address 40739, which is the data port 0 on the VERA graphics module.

Generally, the use of the POKE statement will be used for I/O programming or for creating cheats for games.

You will receive an "ILLEGAL QUANTITY" error when trying to POKE an address with a value outside the range of 0 to 255. This is because POKE can only affect a single byte of data at a memory location. A single byte can only have a value between 0 and 255.

A word of caution is essential here. The POKE statement is extremely powerful, but additional responsibility is needed. Unknown addresses should not be changed carelessly. You cannot physically damage your Commander X16 by changing the values of random addresses, but you can accidentally force your computer to ignore your keyboard entries so

that the computer seems to stop working entirely. It is called a system crash. Crashing the computer is not a problem. Simply turn it off and back on again.

POKE could cause a system crash. Crashing the Commander X16 is not a problem. Simply restart it

Summary - POKE

- There are 65536 memory addresses in the Commander X16. The numbers vary between 0 and 65535
- Some addresses are used to contain information. Other addresses are hardware related
- The value is between 0 and 255 for every location
- The POKE statement replaces an address's contents with a new value
- The POKE statement replaces an address's contents with a new value
- The POKE Statement syntax is the POKE Keyword, a number or expression to show where to change the position, a comma, and the second or new value in the address
- An abbreviated form is, P shift-O

VPOKE Statement

Similar to the POKE statement, VPOKE allows the contents of a memory location within VERA to be changed. The Video Enhanced Retro Adapter (VERA) graphics module has three banks, each having 65536 addresses. The first two banks are used for graphics, and the third bank is for controlling VERA. The syntax of VPOKE is the keyword VPOKE, followed by a number (or expression), a comma, a second number (or expression), a comma and finally the third number (or expression). The first number is the VERA bank (valid range is 0,1,15). The second number is the address in that bank (valid range is 0 to 65536). The third number is the new value to be stored (the valid range is 0 to 255).

The VERA module is an impressive piece of technology and is covered in detail in a following chapter. For now, we will experiment by changing a few basic values.

```
VPOKE 0,0,1
VPOKE 0,2,49
VPOKE 0,4,54
```

Virtual screen 16 – Example of VPOKE and placing characters

Displayed in the top left-hand corner is X16. The values that we stored (1,49,54) are screen display codes. To see all of the Commander X16 screen codes, check out Appendix **TODO** X16 Screen Display Codes.

Now we will change the colours of those letters.

```
VPOKE 0,1,1
VPOKE 0,3,2
VPOKE 0,5,3
```

Virtual screen 17 – Example of VPOKE changing a character colour

The X16 in the top left-hand corner should now be coloured white, red and cyan. There are 16 colours, in the range 0-15. We can change the background colour for each individual character.

```
VPOKE 0,1,16
VPOKE 0,1,17
VPOKE 0,1,18
```

Virtual screen 18 – Example of VPOKE changing background colour

TODO work out a good way to explain how background changed by poking
TODO work out a way to change screen background and border

While in our POKE statements we used only numbers, expressions can be used instead of numbers, so the statement shown below is an alternative way to set the background colour to red.

```
VPOKE 0,1,16/8
```

Virtual screen 19 – Example of VPOKE changing background colour with an expression

Maybe you want to experiment further and see what other colours can be put on the screen. Nevertheless, bear in mind that your characters are white. So, if you are going to change the background of the screen to white, first change the colour of the text character by holding down CTRL and pressing the key 1 (for black). The text characters, otherwise, will be the same colour as the background, and the screen will appear to be blank.

Summary - VPOKE

- There are three banks of 65536 memory addresses in VERA
- Some places are used to contain information. Other addresses are hardware-related
- The value is between 0 and 255 for every location
- The VPOKE statement replaces a location's contents with a new value
- The VPOKE Statement syntax is the VPOKE Keyword, a number or expression to select the bank, a comma, a number or expression to show where to change the position, a comma, and number or expression which will become the new value in the location
- There is no abbreviated form

Multiple Statements

Now that you have given a workout to the PRINT and VPOKE statements, you have seen how the computer accepts a line you enter, searches for a keyword it recognises and executes the statement. You have just placed one statement on each line so far. But you will often want to put multiple statements on one line. This is done by means of the colon (:). The colon's purpose is to tell the computer where one statement ends, and the next one starts. Try this example with several statements on one line.

```
? 64: ? 16: ? 128: ? 512: ? 2048
```

Virtual screen 20 – Example of multiple short form commands

After pressing the RETURN key, a column of numbers should have been printed before the READY prompt was printed.

By default, the X16 uses 80 characters per row and 60 rows per screen. A line in BASIC cannot be longer than 80 characters. If you type in 81 or more characters without pressing the RETURN key, the computer will ignore the first 80 characters and accept the remainder. If the remainder occurs in the middle of a statement, the computer will generate a "SYNTAX ERROR". Type lines short enough, so you do not use more than 80 characters to prevent problems.

Summary - Colon

- A colon : is used on one line to separate multiple statements
- The computer can handle a maximum line length of 80 characters

Introduction to Variables

So far, only PRINT and POKE statements have been used. And we have re-typed the line every time we wanted to change the number we print or to change the colour of a character. However, numbers can be represented more easily.

During the PRINT and POKE demonstrations, you may have wondered if you type anything other than a number like an alphabet letter what would have happened? Just try now.

```
PRINT X
 0
READY.
```

Virtual screen 21 – Example of printing a variable

A number zero is printed on the screen instead of an X. Try a few more letters.

```
PRINT Y: PRINT Z
 0
 0
READY.
```

The computer prints zero each time.

This does not mean that every letter is interpreted by the computer as number 0. Instead, the computer considers that the letter is a variable, and it is the variable that has the value of 0.

A variable is composed of two things, a name and a value. It is known as a variable because the value can be changed. The number 5 is always 5, and PRINT 5 always prints 5 on the screen. However, a numerical variable can have different values to represent any number at different times.

All of the variables you saw so far (X, Y, and Z) had a value of 0. You did not give them value, so the computer assigns a default value of 0.

LET statement

To assign a value to a variable, we use the LET statement. The LET syntax is very distinct from the PRINT or VPOKE syntax. The syntax is the LET keyword, a name for the variable, the equal symbol then a value or expression.

```
LET X = 16
READY.
```

Virtual screen 22 – Assigned a variable value

The first step that the computer does is evaluate the part of the statement to the right of the equal sign to compute a value. Then the LET statement is performed. If an expression appears, it must be evaluated. When the computer has a value, it is assigned into the variable on the left of the equal symbol.

Now the variable can be used, for example with a PRINT statement.

```
PRINT X
 16
READY.
```

The computer answered the usual READY prompt when you entered the LET state. There is nothing to be printed by the LET Statement itself. The value assignment is done in the memory of the computer.

Assigning a value to one variable does not affect other variables. The variables Y and Z still have the default value of zero. However, we can change that quickly.

```
LET Y = 64: LET Z = 128
```

As we can see, every variable is independent.

```
PRINT X: PRINT Y: PRINT Z
```

Once a value is assigned to a variable, it does not mean that the value is permanently set. Another LET statement can change the value. While a variable value may be altered, a variable at any time can have just one value.

```
LET X = 64: PRINT X: LET X = 16: PRINT X
```

Furthermore, once a variable has been set, that value remains unchanged. Additionally, without creating conflict, two separate variables can have the same value.

```
LET X = 16: LET Y = 16: PRINT X: PRINT Y
```

It is illegal for the LET statement to change the value of a constant. For example, the statement below.

```
LET 5 = A
```

Will create a “?SYNTAX ERROR” as the number 5 is an integer constant and its value cannot be changed.

Summary - LET

- The LET statement replaces the value of a variable with a new value
- The syntax for LET is the keyword (LET), a variable name, an equal sign, and a number or expression
- Abbreviated form, L + Shift-E

Rules of Variables in BASIC

In BASIC, the names of variables must follow the rules.

- The first character must be alphabetic. (Range A-Z)
- The 2nd character may be alphanumeric. (Range A-Z and 0-9)
- Any more alphanumeric characters are allowed but are not considered part of the variable name
- The second character may be % to represent an integer or $ to represent a string variable
- The second character may be (to represent a subscripted variable
- A name cannot include reserved words since the BASIC interpreter will treat them as keywords

Variable names can be of any length but BASIC only deals with the first two characters. All variable names cannot have the same first two characters.

```
LET PLAYER1LIVES = 3: PRINT PLAYER1LIVES
LET PLAYER1PTS = 12345: PRINT PLAYER1PTS
PRINT PLAYER1LIVES
```

Virtual screen 23 – Example of conflicting variable names

In the above code, the variable PLAYER1LIVES was assigned a value of 3. BASIC refers to that variable by the name PL. We then assign the value of 12345 to PLAYER1PTS. However, this variable has the same first two letters. The BASIC interpreter will store the value 12345 into the variable PL. This has just overwritten our PLAYER1LIVES from the value 3 to 12345.

Variable names *CANNOT* be identical to BASIC keywords and may not contain keywords anywhere in the variable name. All BASIC commands, statements, function names and logical operator names are covered as keywords.

```
LET PLAYER1PRINT = "YES"
```

Virtual screen 24 – Example of the variable name with a keyword

If you use a keyword within the variable name by mistake, the BASIC error message "?SYNTAX ERROR" will appear. In the above example, we used the keyword PRINT.

Summary - Introductory BASIC Variables

- A variable consists of a name and a value
- Variable names must be unique
- All variables have a default value of zero
- You can modify the value of a variable. This is the difference between a number and a variable
- Variables are independent. Changing a variable does not change another
- A variable keeps its value until it is changed
- A variable can only be one value at a time
- Letters of the alphabet may be used as names for variables
- There may be any value valid for a number in a numeric variable

Functions

Functions are not statements. Functions are not operators. Functions are not variables. Functions are a class by themselves. Functions return values that are used by other functions or statements. You *CANNOT* use a function by itself. Functions are used in statements where a number or value would be. It is said that the statement calls the function.

Functions have names just like variables but are they are not assigned values. Functions normally require that a value be passed to them. They then perform a process and return a value. This chapter will introduce several functions. Functions can be important tools for creating good programming logic.

Mathematical functions can be written using the notation F(X)=Y. Y is the result when the value X is given to the function F. Functions are good examples of the concept input/output. A value is an input to the function, it processes it, and a value is output from the function. Different functions follow different processes and will return different values.

BASIC functions pretty much work in the same way. Here are several examples.

```
LET X = RND(10)
LET X=8: PRINT ABS(X-9)
LET X=2.2: PRINT INT(X)
```

A function performs its process on a single value unlike operators (+ - * /) which require at least two values to create a result.

At least three letters are used in the names of BASIC functions. The name of a function is immediately followed by a pair of parentheses containing a value that can be a number, a variable, or an expression. This value for the function in parentheses is the value or argument. Spaces in the name of the function or between the name and the opening parenthesis are not permitted.

Summary - Functions

- A function is not used independently. A function is called by a statement
- The function has no value. Rather, a function is a value-generating process. The resulting value may then be used as a number. A function will return the created value to the statement which called it
- A BASIC function has a name composed of at least three alphabetic characters
- The function name is followed by a pair of parentheses. There can be no spaces in the function name
- Inside the parentheses, the input value or argument for the function is stored. A function can only have one argument

Absolute Value (ABS) function

There are two sections of a number: an absolute, and a sign (plus or minus). The ABS function removes the sign from a number. When the sign is positive, the absolute value is returned unchanged. When the sign is negative, it will be changed to positive, and then the absolute value is returned. The absolute value function returns a positive value every time.

In BASIC, the Absolute Value function is called ABS.

```
PRINT ABS(10)
PRINT ABS(-10)
```

In the example, the value which is printed to the screen in both instances is 10.

The real utility of this function is that many other functions and statements can only accept positive values. Using this function will ensure that the values being used are positive.

Summary - ABS

- The syntax for ABS is the keyword (ABS), parentheses, the argument (a variable name or value), closing parentheses
- The ABS function always returns a positive value
- Abbreviated form, A + Shift-B

Integer (INT) function

An integer is a number that can be written without a fractional component. As a reminder, BASIC has three variable types. String variables, which hold characters. Integer variables which can hold -32768 to +32767. Finally, real variables which can hold +-2.93873588 X 10-38 to +-1.70141183x1038. When a value is passed to the INT function, it will return the nearest integer that is less than or equal to the argument value.

```
PRINT INT(10)
PRINT INT(2.9387358800000001)
```

In the example, the argument of 10 is already an integer, so there is no change. The long number is not an integer, so the INT function returns the value 2.

It might appear that the integer function simply removes the fractional component. However, when we use a negative number as the argument, the value returned is the greatest integer less than the argument value.

```
PRINT INT(-10)
PRINT INT(-2.93873588)
```

In the example, the argument of -10 is already an integer, so there is no change. The long negative number is not an integer and the next integer less than -2.93873588 is not -2 but -3.

The integer function will never return a value larger than the argument provided.

Summary - INT

- The syntax for INT is the keyword (INT), parentheses, the argument (a variable name or value), closing parentheses
- The INT function returns a value less than or equal to the argument
- The INT function never returns a value larger than the argument
- Abbreviated form, none

Nesting of functions

Now that we have seen a few functions, we will look at a useful feature of functions. That is the returned value can be used like any other number. With this, it is possible to use the value returned from one function as the argument for another. This is called nesting the functions.

```
PRINT ABS(INT(64))
PRINT INT(ABC(2.56))
PRINT ABS(INT(-128))
PRINT INT(ABC(-5.12))
```

In the example, you should notice the two pairs of parentheses are required. Each opening parenthesis must have a closing parenthesis. Otherwise, a "?SYNTAX ERROR" will be created.

The order the functions are nested will affect the result. The functions deepest inside the nest are processed first. The value returned from them is then passed as the argument for the next function, and so on.

```
PRINT ABS(INT(-128.256))
PRINT ABS(SGN(INT(-128.256)))
```

In the first line of the example, the INT function returns a value of -129 which is passed to the ABS function which returns a value of 129.

In the second line of the example, the INT function returns a value of -129 which is passed to SGN which returns the value of -1, and this value is then passed to the ABS function which returns a value of 1.

As we can see, it is possible to build complex logic from a few simple functions.

Summary - Nested functions

- The value returned by a function can be used as an argument to another function
- Each opening parenthesis must be matched with a closing parenthesis
- The deepest function is evaluated first

Random (RND) function

The RND function returns a floating-point number between the range of 0.0 to 1.0. The RND function is somewhat unique in the way the argument affects the returned value. When an argument is a positive number, it will return a random number from a predetermined sequence.

```
PRINT RND(10)
 .185564016
PRINT RND(10)
 .0468986348
```

Virtual screen 25 - Example of RND from a sequence

In the example, we see random numbers being returned. However, these are from the sequence. When the computer is restarted, and the same example is entered, you will see the exact same random numbers. This feature can be useful when creating procedurally created games. The procedurally created data could be the same each reboot and across the entire Commander X16 platform.

When the argument passed to RND is the value zero, the X16 will generate the random number from the internal clock. Because the number range from the clock is 0-60, the RND(0) function might not be suitable for creating large ranges of numbers. It is possible that patterns may become apparent. It is still useful for games, just maybe not applications that require truly random numbers.
TODO check GIT source for RND

```
PRINT RND(0)
 .74621658

READY.
PRINT RND(0)
 .404580259

READY.
```

Virtual screen 26 - Example of Random RND

The random numbers you see on your X16 should not match the above example. They should be completely random.

When the argument passed to RND is negative, it will return a particular number in the determined sequence. Repeatedly calling RDN with the same negative number will result in the same value being returned.

```
PRINT RND(-10)
 3.73729563E-08

READY.
PRINT RND(-10)
3.73729563E-08

READY.
```

Virtual screen 27 - Example of RND to return a particular number

The random numbers you see on your X16 should match the above example.

The RND function is very useful in games and applications. It can be used to creating some random numbers to seed the locations of enemies in games and used for simulating rolled dice. The drawback with the function is it returns decimal numbers between 0.0 and 1.0. However, with nested functions, we can create the number of values we want.

```
PRINT INT(RND(0)*128)
PRINT INT(RND(0)*6)+1
```

In the example, we could get a random number between 0 and 127. Then we could get a random number between 1 and 6.

Summary - RND

- The syntax for RND is the keyword (RND), parentheses, the argument (a variable name or value), closing parentheses
- The RND function returns a random value from a predetermined sequence if the argument is positive
- The RND function returns a random value generated from the internal clock if the argument is zero
- The RND function returns a particular value from a predetermined sequence if the argument is negative. Repeated calls with the same negative argument will result in the same 'random' number in the predetermined sequence
- Use nested functions to create useful random ranges
- Abbreviated form, R + Shift N

Free Memory (FRE) function

This function will return how much free memory is available for use by BASIC in the base memory space. As a refresher, the Commander X16 base memory is 64 x 1024 = 65536 bytes. Measuring space and memory in computers 1024 bytes is called 1 K. The X16 has 64k of base memory and 512k of high memory. Different portions of memory are reserved for the Kernal, BASIC interpreter, sound and external input/output. Once all the reserved locations are accounted for, there is around 38K left.

The FRE function is somewhat unique that while an argument is required, it is a dummy argument and will not affect the value returned by the function. If the argument is missing, it will create a "?SYNTAX ERROR".

The Commander X16 team have kept the BASIC commands backwards compatible with the older Commodore BASIC v2. As a result, FRE still shares an odd behaviour. If the amount of free memory is more than 32K, it will return a negative number that you need to add 65536 to get the actual free memory amount.

Summary - FRE

- The syntax for FRE is the keyword (FRE), parentheses, any value or argument (zero is ok), closing parentheses
- If free memory is less than 32k, it will return the correct amount
- If free memory is greater than 32k, it will return a negative amount which 65536 must be added to for the correct result
- Abbreviated form, F + Shift R

PEEK function

The PEEK function is for peeking at a location in memory and reading its value without changing anything. The PEEK function operates the same as others function. The argument which is passed to PEEK is the memory location that is to be viewed. The valid options are 0 to 65535. From this, we can see that PEEK can only view the base memory. You *CANNOT* directly peek into the 512k -2048k of high memory. The value returned by PEEK will be an integer number in the range of 0-255.

TODO PEEK into memory banks by poking value and switching bank

Summary - PEEK

- The syntax for PEEK is the keyword (PEEK), parentheses, argument, or value in the range of 0-65535, closing parentheses
- PEEK will inspect the memory location passed in the argument and return a value representing the contents at that location. It will return an integer in the range 0-255
- PEEK only inspects locations in the X16 64k base memory
- Abbreviated form, P + Shift E

VPEEK function

The VPEEK function is similar to the PEEK function. VPEEK views the memory contents of a memory location within VERA. As a refresher, the Video Enhanced Retro Adapter (VERA) graphics module has three banks, each with 65536 addresses. The first two banks are used for graphics, and the third bank is for controlling VERA.

The VPEEK function is different than other BASIC functions in that it requires two arguments. The first argument is the VERA bank. The valid options are $0,$1 and $F. The second argument is the address in the selected bank. The valid options are 0 – 65535.

Turn on or reset your Commander X16 and type in the example below.

```
PRINT VPEEK(0,2048):PRINT VPEEK(0,2050):PRINT
VPEEK(0,2052):PRINT VPEEK(0,2054)
```

Virtual screen 28 – Example of VPEEK

The numbers printed out are 18,5,1,4, which are the letters R, E, A, D. It is the ready prompt which appears at the top of the screen. To check for yourself, have a look at Appendix **TODO** X16 Screen Display Codes. If you type, return multiple times, so the first screen scrolls off the top and try this example again, you will get different values. The contents of those memory locations have changed. A more detailed explanation of VERA will follow in another chapter.

Summary - VPEEK

- VPEEK needs two arguments
- View the contents of VERA memory
- There are three banks of 65536 memory locations in VERA
- Some places are used to contain information. Other places are hardware-related
- The value is between 0 and 255 for every location
- The syntax for VPEEK is the keyword (VPEEK), parentheses, first argument which is the VERA bank value $0 $1 $F, the second argument which is the location in the bank, and it is in the range of 0-65535, closing parentheses
- Abbreviated form, none

String functions

There are several functions which are for processing string variables. It is quite simple to concatenate strings. It is just a matter of adding the strings together.

```
LET A$="COMMANDER X16"
LET B$="CONCATENATE"
LET C$="STRINGS"
LET D$=A$+" "+B$+" "+C$
PRINT D$
```

The functions LEFT$, RIGHT$ and MID$ will break up a string into a substring. A continuous sequence of characters in a string is called a substring.

The entire string								
S	U	B	S	T	R	I	N	G
SUB			STRING					
A 3-character substring starting from the left			A 6-character substring starting from the right.					

Figure 9 – Explaining substring

There are other functions that perform some work on a string. Such as working out which character code is being used in the string or the numerical value of a string.

LEFT$ function

The LEFT$ function creates a substring counting from the left side of the original string. The syntax is the keyword LEFT$ followed by parenthesis, followed by a string or string variable,

followed by a comma, then an integer or integer variable, then the closing parenthesis. The string can have 255 characters and the integer can range from 0 to 255.

```
LET X$="COMMANDER X16"

READY.
L$=LEFT$(X$,9)

READY.
PRINT L$
COMMANDER

READY.
```

Virtual screen 29 – Example of LEFT$

RIGHT$ function

The RIGHT$ function creates a substring counting from the right side of the original string. The syntax is the keyword RIGHT$ followed by parenthesis, followed by a string or string variable, followed by a comma, then an integer or integer variable, then the closing parenthesis.

```
LET X$="BASIC PROGRAMMING CAN BE FUN"

READY.
R$=RIGHT$(X$,10)

READY.
PRINT R$
CAN BE FUN
```

Virtual screen 30 – Example of RIGHT$

MID$ function

The MID$ function creates a certain length substring starting from a position on the left. The syntax is the keyword MID$ followed by a parenthesis, followed by a string parameter, followed by a comma, then the start position integer parameter, followed by a comma, then the length of the substring parameter then the closing parenthesis. The start position parameter integer must be greater than zero.

```
LET X$="MID$ FUNCTION DEMO"

READY.
PRINT MID$(X$,15,4)
DEMO

READY.
PRINT MID$(X$,1,4)
MID$

READY.
PRINT MID$(X$,6,8)
FUNCTION

READY.
```

Virtual screen 31 – Example of MID$

Length (LEN) function

When working with strings it may be important to know how many characters are in the string. The LEN function returns an integer number value of the total of all characters in the string. Non-printed and special characters are counted as well as blanks. The syntax is the keyword LEN, followed by a parenthesis and the string or string variable parameter, followed by the closing parenthesis.

```
LET T$"COMMANDER X16 IS SUPER"
READY.
PRINT LEN(T$)
22
READY.
```

Virtual screen 32 – Example of LEN

ASC function

Every character that appears on screen has a number value ranging from 0 to 255. The ASC function when passed a character or string will return the ASCII value of the first character.

Example of ASC

PRINT ASC(“X”)
A=ASC(“1”): PRINT A
PRINT ASC(“6”)

The abbreviated form is, A + Shift-S.

Summary ASC
The syntax is keyword ASC, parentheses, a parameter, closing parentheses. The parameter must be a string variable which has a minimum of one character.
Abbreviated form, A + Shift-S

Mathematical Functions

Sine (SIN) function

The SIN function returns the sine of the radian value passed to it. The syntax is the keyword SIN, followed by an open parenthesis, a numeric parameter, and the closing parenthesis. The numeric parameter must be in radians, not in degrees.

Cosine (COS) function

The COS function returns the cosine of the radian value passed to it. The syntax is the keyword COS, followed by an open parenthesis, a numeric parameter, and the closing parenthesis. The numeric parameter must be in radians, not in degrees.

Tangent (TAN) function

The TAN function returns the tangent of the radian value passed to it. The syntax is the keyword COS, followed by an open parenthesis, a numeric parameter, and the closing parenthesis. The numeric parameter must be in radians, not in degrees.

Π (Pi) constant

The X16 has several reserved variables. The Pi variable is a special case and is treated as a constant. It cannot be changed. Pi has its own character reserved for it as well. Press Shift-~ to display the π symbol.

TODO

Logarithm (LOG) function

The LOG function returns a floating-point number which is the natural logarithm (to the base of *e*) of the value passed to the function. The natural logarithm is the power to which *e* would have to be raised to equal the value passed. The approximate value of *e* is 2.718281828. The syntax is the keyword LOG, followed by an opening parenthesis and a number parameter, followed by the closing parenthesis. The number parameter can be an integer or a floating-point number.

TODO better way to explain the use of natural log.

Summary

Square Root (SQR) function

The SQR returns the square root of the number parameter passed to it. The number passed to SQR cannot be less than zero. The syntax is the keyword SQR followed by an opening parenthesis and the number parameter followed by the closing parenthesis.

```
PRINT SQR(16)
 4

READY.
PRINT SQR(25)
 5

READY.
```

Value (VAL) function

It is possible to convert a string into a number that computer can perform mathematical functions on. The VAL function will take a string that starts with a plus sign (+) or a minus sign (-) or a digit and return its value.
TODO

```
LET S$="128"

READY.
PRINT (VAL(S$)-112) * 16
 256

READY.
```

Virtual screen 33 – Example of VAL function

Sign (SGN) function

The SGN function returns one of three values. If the value/argument passed to the function is positive (greater than zero) the function will return the value 1. If the value is zero, the function will return a zero. If the value/argument is negative (less than zero), then the function will return the value -1.

Example of SGN

```
PRINT SGN(256)
PRINT SGN(0)
PRINT SGN(-128)
```

This function can be useful in games. Using the SGN function, it is possible to determine if an object was moving to the left or right, or not moving at all. For example, the variable X1 represents the objects current x co-ordinate and X2 represents the new location. The value returned from SGN(X2-X1) will tell us which direction the object is moving. If the function returns the value 1, then the object is moving to the right. If the function returns the value of 0, then the object has not moved. If the function returns a value of -1, then the object has moved to the left.

Summary - SGN

- The syntax for the Sign function is the keyword (SGN), parentheses, the argument (a variable name or value), closing parentheses
- The SGN function returns a value of 1 if the argument is positive
- The SGN function returns a value of 0 if the argument is zero
- The SGN function returns a value of -1 if the argument is negative
- Abbreviated form, S + Shift-G

Time functions

The X16 has an internal jiffy clock. A jiffy can have several different meanings, but in the context of the Commander X16, it is 1/60 of a second. When the X16 is powered on, it starts to count how many 1/60 seconds have passed. Performing a soft reset does not restart the clock. The clock is initialized or set to zero on power-up. It is possible to reset the clock to any value with the below functions. This can allow a stopwatch like functionality.

Time (TI) function

The TI function reads the internal clock and returns the current jiffy count since the computer was turned on. The TI function returns a value. It must be used with another function such as PRINT or LET.

```
10 PRINT TI,"JIFFYS SINCE POWER ON"
20 PRINT "OR NUMBER OF SECONDS SINCE POWER ON",TI/60
```

It is possible to set the jiffy clock to any value. Setting the value of TI to 0 will reset the clock and can act as a timer.

```
10 PRINT "TIME SINCE CLOCK RESET",TI
20 PRINT "RESETTING CLOCK.."
30 LET TI=0
40 PRINT "CLOCK IS NOW=",TI
```

Time$ (TI$) function

The TI$ function is similar to TI however it performs some maths and returns the number of hours, minutes and seconds since the computer was turned on. The returned value is a single number in the format HHMMSS.

The REM statement

The REM statement lets you place remarks in your program. The modern equivalent is the comment. In this book I use remark and comment interchangeably. When I was new to programming, I did not believe in commenting my code. I quickly discovered that when returning to code I had written only a few months before I had forgotten what I was doing. It would then take valuable time to work out what was happening. This was for my own code! Working in teams made commenting code mandatory. BASIC textbooks from the 1980s may say that remark statements are for users benefit only, do not believe them. REM statements are an important part of documenting your code and enable you to move quickly around the program while you work on it and understand it when you return to a project. Often REM comments are used in the first few lines of a BASIC program to describe a number of different items. The name of the program, what it is going to do, who the author/s are. A blank REM statement can be used to provide some ‘whitespace’ and create clear breaks in the program. White space can help with the aesthetics and overall readability of your code.

The syntax of the REM statement is the REM keyword, followed by optional text. When the X16 finds a REM statement it will ignore the rest of that line, moving on to the next. REM can be used on a line with other statements. However, remember it must be the last statement on the line.

Good Comments

But what to comment? When you are reading code all you see is what is written there. Your comments should explain what the code cannot. Are there special sequences in the code? What do the variables mean? Is this a section of code that needs to be reworked. It is

common to find TODO comments in code, reminding a programmer of something they wanted to do but did not have time or all of the skills to do.

TODO BETTER CODE EXAMPLE HERE
Your comments should clarify the code.
10 LET X=16
20 PRINT X

The code makes sense, but why 16 and why am I printing it out? Your comments should explain what the code is trying to accomplish. There is no point in commenting what the code is doing.

10 REM SET X TO BE 16
20 LET X=16
30 REM PRINT OUT VARIABLE X
40 PRINT X

Do not comment on code which is easy to understand. The above comments do not clarify what the code is trying to accomplish and do not add value.

Writing a comment that explains simple code does not add value. Comments are there to help people understand your code quickly. Comments can be used to explain why certain limits exist.

10 REM CALCULATE VALUE FROM TERRAIN GENERATOR FOR LOCATION X AND Y
20 **TODO** Add code example from terrain/dungeon generators

It can be helpful to record discoveries about the code or your thoughts as you worked, a running programmer's commentary. Include thoughts about problems or improvements that could be done.

```
10 REM DO NOT USE RED ON BROWN AS COLOUR BLIND USERS WON'T BE ABLE TO READ
10 REM *TODO* SORT IS SLOW, RESEARCH QUICKSORT
10 REM *FIX* PLAYER SPRITE WILL FLICKER IF X IS OVER 255 AND PLAYER SHOOTING
10 REM *LIMIT* ENEMY SPRITES LIMITED TO NO MORE THAN 16 OTHERWISE THEY FLICKER
```

Virtual screen 34 – Examples of commentary comments

10 REM **TODO** create examples of good comments

Comment Keywords

It helps if you have a system for how you manage comments. Having a comment start with a keyword helps set the tone of the comment. It also greatly helps with searching for comments in your code. Some of the keywords I use you have seen in earlier examples. I have included a few extra in the table below. The list is not exhaustive, and you can develop your own.

Keyword	Meaning
TODO	Outstanding task that has not been completed.
FIX* / *BUG	A known bug or issue exists in this code.
CLUDGE	A piece of programming that works but is not pretty at all.
XXXX	A fundamental problem exists here. Comments around this code may contain strong language.
ROLE	Describe what the subroutine is providing
INPUT	List the inputs needed for a subroutine.
OUTPUT	Discuss the outputs from a subroutine.
DISCUSS* / *REVIEW	Useful when working on a team and marking code for review.
LIMIT	Comment about the limits of the code.
SUMMARY	Used to provide a general summary of the code that follows.

Table 11 – Comment Keywords

When writing comments, it is very important to think about how others (including your future self) will read and understand your code. Thinking yourself a few questions can help you write better comments.

- `What are the limits of this code?`
- `How might this code be misused?`
- `Are there any surprises in this code?`
- `Are there any design or strange details in this code?`

It can also help to think you have someone sitting next to you and you are explaining what your code does. I often end up working on several projects at once, some at work, some personal and using different languages as well. Some modern comment advise is "Comment the *why*, not the *what*". However, remember the goal of comments to help the reader understand the code easily. This means my comments often end up explaining the why, the what and how!

Remarkable comments

While I believe comments should explain a lot, they need to be concise. You should not have three lines of comments if a single line will do the job. Remember the goal of comments is to explain the code, you must avoid vague sentences. As an example, consider the below sentence.

```
REM INSERT TEXT INTO ARRAY, BUT FIRST CHECK IF IT IS TOO LARGE
```

Virtual screen 35 – Vague comment example

While this comment is grammatically correct, when reading this comment, what does "IT" refer to? The array or the text? The reader would need to then read the following code to

work out exactly what is happening. Consider rewording the comment being less ambiguous and more precise.

```
REM IF TEXT IS LESS THAN 64 CHARACTERS, INSERT INTO ARRAY
```

Virtual screen 36 – Concise comment example

Because BASIC limits the size of variable names to two characters it is important to include comments at the start of a subroutine which explain how variables will be used.

```
10 REM SUBROUTINE PLAYER BULLET COLLISION
20 REM *INPUT* PLAYER SPRITE SCREEN X,Y (PX)(PY) BULLET SPRITE
ARRAY (B$$)
30 REM *OUTPUT* UPDATE PLAYER STATE (PS) IF HIT
40 REM *SUMMARY* CHECK IF PLAYER HAS COLLIDED WITH BULLET
50 REM IF COLLISION REMOVE BULLET FROM ARRAY AND UPDATE PLAYER
STATE
```

Virtual screen 37 – Example of subroutine comments

Summary - REM

- The syntax for REM is keyword (REM) followed by optional text
- The REM statement lets you place comments in your program
- REM statements can be placed on a line after other statements
- When the computer encounters a REM statement the rest of the line is ignored
- Blank REM statements can be used to create 'whitespace' in a program
- Comments need to be concise and help the reader understand the code
- Use of comment keywords can be helpful

Further Reading

All About the commodore 64 Vol 1 [8]
All About the commodore 64 Vol 2 [9]
The Century Computer Programming Course for the Commodore C64 [2]
Programming in BASIC a complete course by Margaret McRitchie [10]

Chapter 4 – Writing BASIC programs

"Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program." – Linus Torvalds

Immediate Mode vs Deferred Mode

So far everything you have typed has been processed at once. Once you pressed the RETURN key, the computer processed the line, presented any output, then the system presented a READY prompt. The X16 has immediately processed the BASIC statements. This is the immediate mode or direct mode. It is possible to enter multiple lines of BASIC to complete simple tasks. As we can see in the below example.

```
REM P1 = PLAYER 1 LIVES
LET P1 = 3
PRINT P1
```

Immediate mode BASIC cannot be saved and executed at a later time. It has to be typed each time. If you clear the screen or reboot the X16, everything is lost.

A BASIC program is stored in memory. You can use the RUN statement to execute or run the program from memory. When the X16 executes a BASIC program, it is working in a deferred mode—also called program mode. To have BASIC statements stored in memory, we have to use line numbers. The line number determines the order of each line in the program. This way, it is possible to enter code out of sequence and have it executed in the correct order.

```
5 REM P1 = PLAYER 1 LIVES, BO = BONUS
10 LET P1 = 3
20 PRINT P1
15 LET P1 = P1 + BO
```

Listing 4 - Simple example of BASIC

Valid line numbers are integers between 0 and 63999. Numbers with fractions are not permitted, e.g., 1.5 or 10.25 are not valid line numbers. There does not have to be a program line for every line number. However, no two lines of a program can have the same line number. There are two problems with large line numbers. The first being, typing them. The second problem is they use more characters on the line. In BASIC, we are limited to 80 characters per line. So, a five-digit line number will use four more characters than a single-digit line number.

We can add lines to our program, and we can place the lines between existing lines of the program. It is a frequent practice to increase the line number by increments of 10. This allows for extra lines of code to be inserted. Otherwise, the program must be renumbered, which can cause other issues as we will see later.

Summary - Immediate Mode vs Deferred Mode

- The X16 works in immediate or direct mode, where the line is executed as soon as it is entered
- When a number is placed at the start of a line, the X16 will store it in memory as a program
- The X16 ignores unused line numbers
- The order of the program is sorted lowest to highest by the line number
- It is a good idea to increment line numbers by 10 to allow the program to be changed easily
- A line number does not represent its actual position. E.g., A numbered line in a program of 5 is not always the fifth line in the program. It could be the first
- Line numbers must be integers in the range of 0 to 63999. Other numbers will create a SYNTAX ERROR
- When the X16 runs a program, it is in deferred or program mode

Commands

LIST command

There is a number of BASIC commands, similar to statements. However, they are used in immediate mode to manage programs. While it is possible to include commands in a program, they may not function.
The LIST command has the X16 print all the lines in the program on the screen. The lines are listed in order from lowest to highest line number.

```
LIST
0 REM EXAMPLE
10 LET PLAYER1LIVES = 3
20 PRINT PLAYER1LIVES
30 IF PLAYER1LIVES=0 THEN PRINT "DEAD": END
40 PLAYER1LIVES=PLAYER1LIVES-1
50 GOTO 20
```

Virtual screen 38 – Example of LIST

The LIST command can take an argument. It is possible to list a single line number or list a range of line numbers. For example.

```
LIST 10
10 LET PLAYER1LIVES = 3
LIST 10-20
10 LET PLAYER1LIVES = 3
20 PRINT PLAYER1LIVES
```

Virtual screen 39 – Example of LIST with parameter

You can use the dash with only one line number for even more versatility. When you select a line number and then a dash, the desired line will be displayed, along with all the lines that come after it. The X16 can display all lines up to and including the requested line if you type a dash and a line number. This is especially useful if you do not recall the first or final line numbers of the program.

```
LIST 30-
30 IF PLAYER1LIVES=0 THEN PRINT "DEAD": END
40 LET PLAYER1LIVES=PLAYER1LIVES-1
50 GOTO 20

LIST -30
0 REM EXAMPLE
10 LET PLAYER1LIVES = 3
20 PRINT PLAYER1LIVES
30 IF PLAYER1LIVES=0 THEN PRINT "DEAD": END
```

Virtual screen 40 – Example of LIST using a range

Due to how the X16 handles line numbers if you LIST 0, the entire program will be displayed. When dealing with large programs, the list will scroll past on the screen too quickly to be read. You can slow the display down by holding down the CTRL key. It does not stop the listing, but it does add a slight pause at the end of each line.

Summary - List

- The LIST command will display the current program
- After LIST you can define a line number range. Place a '-' dash between the start and end line numbers
- LIST number ranges can from the start of the program to a line number or a line number to the end of the program
- Holding CTRL key while LISTing the program will display it slower than normal

RUN command

Once you have a BASIC program in memory, you command the X16 to execute the program by using the RUN command. The RUN command orders the machine to run the program from the first line. During the execution of a program, it is said to be running. When the last line is completed, the execution ends. Then the READY prompt will be displayed.

```
10 PRINT "COMMANDER X16"
20 PRINT "BASIC COMMANDS"
RUN

COMMANDER X16
BASIC COMMANDS

READY.
```

Virtual screen 41 – Example of RUN

You can execute the program again by instructing it to RUN again. This is one of the benefits of using a program. You can run it repeatedly. After a program has been RUN, any variables used will keep their values.

When the RUN command is executed, it will clear all the program variables. Then it will start the program on the first line. Like the LIST command, the RUN command can be given an argument. You can select which line number you want the program to start running from. Once started, the program will run until it is finished, or an error happens. A number range may not be used with this command.

```
10 PRINT "COMMANDER X16"
20 PRINT "BASIC COMMANDS"
RUN 20
BASIC COMMANDS
READY.
```

Virtual screen 42 – Example of RUN with parameter

Normally the RUN command is used in immediate mode. However, it is possible to place the command in a program to re-run the program. This could be useful if your program needed to re-initialise variables or an array between executions.

```
10 PRINT "THIS PROGRAM WILL LOOP WITHOUT A GOTO"
20 RUN
```

Listing 5 – Using RUN inside a program

Summary - RUN

- A program is executed using the RUN command. Execution starts in the first line and ends when no more lines are available
- When a program is being executed, it is said to be running
- The RUN command clears all the variables automatically before beginning program execution
- Typically, the RUN command begins running a program on the first line. The line number can be defined after the keyword RUN to start execution at the specified line number
- RUN can be used as an alternative to the END statement to restart the program

END Statement

The computer ceases working in immediate mode when you enter the RUN command and begins operating in deferred mode. It runs every line of the program sequentially and returns to immediate mode only if an error happens or if the last line of the programme is

executed. The keyword END, which is optional, will cause the computer to stop running the program when it encounters it.

On the surface, the END statement might not appear useful. Placing it on the last line is redundant. Placing it in the middle of the program does not seem to make much sense. However, the END statement can be placed on the line with other statements and can be used with IF-THEN.

```
10 IF X<16 OR X>16 THEN PRINT "ERROR! X IS NOT 16": END
```

Listing 6 – Example of the END statement

In the above example if X does not equal 16 then PRINT a message and END the program as soon as the END statement is processed, the program execution stops. No other lines will be processed.

Summary - END

- The syntax for END is the keyword (END) followed by nothing
- When the END statement is processed, the computer will stop executing the rest of the program
- Using END on the last line of the program is redundant, but it was used as a polite formality
- The END statement can be placed on a line with other commands
- The END statement is often used with the IF-THEN statement, as a means of ending the program if an error has occurred or from user input, i.e., quit from the program
- Statements placed after the END statement on the same line will not be processed

NEW command

The NEW command will remove the current BASIC program from memory.

```
LIST
10 PRINT "COMMANDER X16"
20 PRINT "BASIC COMMANDS"
NEW
LIST

READY.
```

Virtual screen 43 – Example of NEW

As well as removing the program, the NEW command will clear and remove all variables. Now you can enter a new program. Unlike LIST and RUN, NEW does not accept a line number argument. It cannot be used to delete part of a program. This program is useful in direct mode, it can be used in program mode but when run it will remove the existing program from memory.

Summary - NEW

- A simple way to remove a program is to use the NEW command, which deletes every line in the program and, at the same time, removes all variables

OLD command

The OLD command will recover the earlier BASIC program that was removed using the NEW command. It is also possible to recover the earlier BASIC program if the computer was restarted using the RESET command. The program can then be RUN as normal. This command useful in direct mode, it can be used in program mode but cannot restore the previous program as there is currently one in memory. When used in a program it will not generate an error.

Summary - OLD

- A simple command to recover a program that was removed using the NEW command
- Can recover a program after the computer was restarted using RESET

CLR command

Erases the contents of all variables in memory.

RESET command

This is a quite simple command that does a software reset of the X16. The BASIC program and contents of variables are cleared but not lost. It is possible to recover the program and variable contents using the OLD command. This command works in direct and program mode.

Summary - RESET

- A simple command to perform a soft reset of the X16
- Previous BASIC program can be recovered using OLD

Editing a program

Entering a program

When you turned on your Commander X16 and entered some of the examples in this chapter, you may have noticed a difference from earlier chapters. When you entered the following line.

```
10 PRINT "COMMANDER X16"
```

Listing 7 – Creating a program

Nothing was displayed to the screen, not even a READY prompt. The cursor appeared on the next line when you pressed the ENTER key. The line you entered was not executed because it has a line number. It was stored in memory as a program line. We can now use the LIST command to view the program. We can now execute the program with a RUN command to see the information printed to the display. We can now add a second line to our program by entering the following line.

```
10 PRINT "COMMANDER X16"
20 PRINT "BASIC COMMANDS"
```

Listing 8 – Building on a program

We can continue to add extra lines to our program.

```
10 PRINT "COMMANDER X16"
20 PRINT "BASIC COMMANDS"
30 ? "STATEMENT SHORTFORM"
```

Listing 9 – Three line program

Program lines are always put in memory, with two exceptions, exactly as you type them, including spacing. The first exception is if the BASIC statement short form is entered, it will be converted to the full statement. For example, using the “?” character instead of PRINT. This “?” is converted to PRINT before stored in memory. If you use LIST, BASIC statement short form will not appear. The second exception is that additional spaces between the line number and the first statement are ignored. Those are referred to as leading spaces. When the line is stored, only one leading area is kept.

```
10 ? "SHORTFORM WITH LEADING SPACES"
LIST
10 PRINT "SHORTFORM WITH LEADING SPACES"
```

Virtual screen 44 – Short form commands listed

Editing a program

After you have entered a line, you may need to modify it, correct a mistake, or add an extra statement. You can change a line by retyping the line, using the same line number.

```
LIST
10 PRINT "COMMANDER X16"
20 PRINT "BASIC COMMANDS"
READY.
20 PRINT "EDIT THIS LINE"
LIST
10 PRINT "COMMANDER X16"
20 PRINT "EDIT THIS LINE"
```

Virtual screen 45 – Example of editing a program

The original line 20 with '20 PRINT "BASIC COMMANDS"' has been replaced with '20 PRINT"EDIT THIS LINE"'. As mentioned earlier, no two program lines can have the same line number.

Edit a line

If you only require making a small edit, then it is possible to use the cursor keys. Using the up/down left/right cursor keys, you can position the cursor key and type over the top of the existing line. Once you have finished and the cursor is still on the same line, hit the ENTER key, and the line will be updated.

```
LIST
10 PRINT "COMMANDER X16"
<UP CURSOR KEY, RIGHT CURSOR KEY, -, ENTER>
LIST
10 PRINT "COMMANDER X-16"
```

Virtual screen 46 – Example of editing a line

Delete a line

To delete a line simply type the line number and press ENTER. It will remove the line.

```
LIST
10 PRINT "COMMANDER X16"
20 PRINT "DELETE LINE"
READY.
20
LIST
10 PRINT "COMMANDER X16"
READY.
```

Virtual screen 47 – Example of deleting a line

Move a line

It is possible to move an entire line of code to a different line number. Simply edit the line number and press ENTER to update the line. This will, in effect, copy the line to the new location. Then simply delete the original line.

```
LIST
10 PRINT "COMMANDER X16"
20 PRINT "BASIC EDIT"
READY.
<UP CURSOR, UP CURSOR, 3, ENTER>
LIST
10 PRINT "COMMANDER X16"
20 PRINT "BASIC EDIT"
30 PRINT "COMMANDER X16"
READY.
10
LIST
20 PRINT "BASIC EDIT"
30 PRINT "COMMANDER X16"
READY.
```

Virtual screen 48 – Example of moving a line

Reminder that when editing a line, you must remember to press the ENTER key while the cursor is still on the line you are editing. Otherwise, the changes will not be saved to memory.

Summary - Editing a program

- When a line is stored in the memory, the cursor moves onto the next line. No READY message is displayed
- A program line is stored in memory exactly as it was typed, except when leading spaces or short form are used. E.g., short form "?" is converted to PRINT. E.g., 10<5 leading spaces>PRINT is converted to 10<1 leading space>PRINT
- A program line can be replaced simply by typing the line number and new BASIC statements
- A program line can be edited by using the CURSOR keys to position the cursor and typing to modify the line. The ENTER key must be pressed while the cursor is on the same line for the changed to be updated in memory
- To delete a program line, simply type the line number and press ENTER. It will replace the contents of the line with nothing, and blank lines are ignored
- Errors are not detected when a line is entered and saved to memory. Only when the program is executed can BASIC detect errors
- When executed if an error is encountered, BASIC will display an error message and the line number which triggered the error

Storage

The X16 has fast and reliable SD card storage. SD cards are much quicker and have a larger capacity than storage used by vintage computers from the 1970s, 80s and 90s. During the late 1970s, a 5.25” floppy disk could hold 110kB. In the early 1980s, as computers became more mainstream, a 5.25” floppy could hold 340kB. By the late 1980s, as technology changed, a 3.5” floppy could hold 1440kB. The 1990s saw CDs become readily available, having a storage capacity of 700,000kB. Each technology change saw an increase in storage capability and performance. With the X16, we can use a standard SD card. A small 16GB SD card holds 16,777,216 kB and is lightning fast compared to a floppy disk. For a retro computer such as the X16, this is truly a massive amount of available space.

When using an SD card, the X16 uses the FAT32 file system. FAT32 is a 1996 extension to the FAT file system, first introduced in 1977. This upgrade allows for large volumes, and Windows supports up to 32GB. However, the maximum individual file size is 4GB in size, which is unlikely to be used on the X16.

Summary - Storage

- SD Card will use FAT32 file system
- Maximum file size is 4GB

SAVE command

Saving your data is essential and makes using the X16 more convenient. If we turned on the X16 and had to re-type our programs or data, that would be frustrating very quickly. Instead, we can store our programs on to some removable media. Most often, it will be an SD card. The Commander X16 supports an IEC connector, so connecting a vintage floppy disk drive and saving data onto a floppy disk is possible. The SD card media is faster, smaller and is more reliable.

To keep our data, we will use the SAVE command. The syntax is the keyword SAVE followed by a string parameter representing the filename you want to use for the file. While the file name can be 60 characters long with letters, numbers, and spaces, be aware that only the first 16 characters of a filename will be displayed when viewing a directory. Also, only the first 16 characters will be used when LOADing a file. Using a long file name may make the file inaccessible on the X16. After the filename, we use a comma to separate optional parameters. The first optional parameter is the device number. The second optional parameter is called the secondary number. Device numbers allow us to communicate with different devices. The device numbers 8 to 11 are for SD/Disk drives. If a device number is not provided, the X16 will default to the last device communicated with, typically 8. The secondary number generally with disk drives only has two options; the value zero, which indicates a BASIC program will be saved, and value one, which indicates a machine language program is being written to the media. The secondary number will default to zero if not provided.

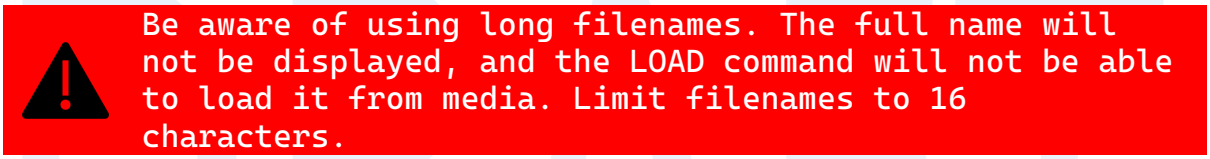
```
SAVE "FILE-NAME.PRG",8

SAVING FILE-NAME.PRG
READY.
DOS"$"

0 "X16 DISK       " FAT32
1    "FILE-NAME.PRG". PRG
98 MB FREE
```

Virtual screen 49 – Example of SAVE command

Be careful when using the space in a file name. It can be difficult to read. Also, I recommend against starting a file name with a space.


```
SAVE "IDEA 1",8
SAVE " IDEA1",8
```

Virtual screen 50 - Example of file misreadable filename



VERIFY command

Older magnetic media such as floppy disks, while reasonably reliable, are not perfect and do degrade over time. SD cards are significantly more reliable, but they can wear out. You can use the VERIFY command to ensure the removable media has the file correctly stored. The VERIFY command will compare the contents of a BASIC program file on removable media with the program stored in memory. Usually, a VERIFY command will be used after a successful SAVE command. The VERIFY command is only helpful in saving and comparing BASIC program files.

The syntax is the keyword VERIFY followed by a string parameter representing the filename you want to check. The filename can only be 16 characters long, consisting of letters, numbers and spaces. After the filename, we use a comma to separate the optional device number parameter.

As the VERIFY command executes, it would respond with "OK" if the file stored on the media matches the one in memory. However, if the program in memory does not match the file on the media, a "?VERIFY ERROR" will be shown on screen.

```
10 PRINT "SAVE AND VERIFY"
20 GOTO 10
SAVE "IDEA-1.PRG",8

READY.
VERIFY "IDEA-1.PRG",8

OK

READY.
```

Virtual screen 51 - Example of VERIFY command

LOAD command

We need an easy way to execute our programs or access the data stored on removable media. We can load our program using the LOAD command.

TODO

```
LOAD"FILE-NAME.PRG",8

SEARCHING FOR FILE-NAME.PRG
LOADING FROM $0801 TO $0822
READY.
```

Virtual screen 52 – Example of LOAD command

DOS command

The DOS command allows access to the SD card. The syntax is the keyword DOS followed by a string. The string is processed by the DOS command to perform several actions.

Finding the current status of the SD card is simple, no string is required.

```
DOS
73,CMDR-DOS V1.0 X16,00,00
```

Virtual screen 53 - Example of DOS command showing status

To perform a directory listing and view a list of contents on the SD card you can perform the following command.

```
DOS"$"
0 "X16 DISK      " FAT32
98 MB FREE
```

Virtual screen 54 – Example of DOS Directory listing

Deleting a file is possible and to do so type

DOS”S:BAD_FILE”

ST variable

Floppy Disk

TODO discuss floppy disk IEC / commodore 1541 etc.

Interacting with the user

As a programmer, you need to think about what task your software will perform for the user. Generally speaking, a program must allow a user to make choices. But when the user runs the software, you cannot read their mind. The software will have to interact with the user in some fashion. On the X16, we can use the keyboard and various peripherals such as a mouse or joystick to get input from the user. The joystick is a great way to get feedback from a user. It has limited choices, and people are familiar with them nowadays. People also understand how to use a mouse. However, programming for mouse use creates extra work as you will need to think about the user interface and process mouse clicks and movement. It is a good idea when using a peripheral to provide a keyboard option. For example, when designing a game, offer keyboard controls and the joystick. Or when developing an application that uses a mouse also allows keyboard hotkeys. On the X16, the keyboard can supply a lot of information. The function keys, cursor keys and number pad can all provide a quick and easy way for people to use your software.

There are some basic principles for user interaction. When designing your software, think about how the user will use it. Clarity is key. The program flow should be streamlined, and tasks should take as few steps as possible. Each screen should have one primary job. For example, a help screen will provide information, and a load file screen will allow file selection and loading. Keep the appearance consistent and straightforward. When a design is consistent, it becomes easier to use. Think about providing feedback to the user via audio or on-screen. User action should get feedback to show it was successful or not. For example, a happy ding sound when an action is accepted, or a buzzing sound when an error has happened. Feedback can also help answer questions before the user even thinks about it. Location, where is the user in the software? For example, the load file screen will clearly show the load file screen, not the save file screen. Current status, what is happening? Is it still going on? For example, a game loading screen may have a percentage. The user can see the game is loading and 50% complete. Future status, what will be happening next? The same game loading screen should have a message letting the player know which level is about to start. Try and reduce the user's cognitive load. Use common names, screen locations and images/icons to help users identify functionality. Make sure that information is accessible. The modern "3 click rule" says it should not take more than three mouse clicks to find any piece of information. Think about accessibility, if you have a button for a mouse to click on provide a keyboard shortcut for it. When using colour, think about how it will look to people. 8% of men and 0.5% of women have a form of colour blindness. Do not rely on colour to communicate information. Lastly, provide a clear next step. The user should not be left guessing what to do next.

There are often complications when software interacts with a user. The screen's instructions may be vague or open to interpretation, or the user may not have read them. When you get input from a user it often needs to be placed into a variable. The user may hit the enter key without having entered any data. When prompted for a choice between 1,2 or 3 the user may have typed "All". The software could be expecting an answer between 1 and 10 and the

value “-500” is entered. When getting input from a user it is important to check the type of data you are getting and make sure it is inside the scope of what you expect.

GET statement

The GET statement will read a single character from the keyboard buffer and transfer it to the named variable. The syntax is the keyword GET followed by a variable. Every time the statement is processed it will get a single entry from the buffer. Normally the buffer is empty nothing is actually getting stored in the variable. If you run the below example, you will see a lot of -48 scrolling by.
TODO is -48 a bug?

```
10 GET A
20 PRINT A
30 GOTO 10
```

Listing 10 – Example of GET statement

If the variable you specified was a float or integer you can only assign values between 0 and 9. If a non-numeric key is pressed such as X, then a “SYNTAX ERROR” will be triggered. To capture most of the keys you would specify a string variable such as A$. A number of the keys will not be captured; RUN/STOP, ESCAPE, SHIFT, X16, CTRL and RESTORE
TODO verify non captured keys.

The GET statement has limited use unlike its related statement GET# which will be discussed later. Another limitation of the GET statement is it can only be used in a program. Trying to use it in direct mode will result in an “ILLEGAL DIRECT” error.

Summary - GET

- GET statement retrieves a single keypress from the buffer and assigned it to a variable
- Syntax is GET keyword followed by a variable name or a list of variables
- Can only be used in program mode. If used in direct mode, the computer will respond with an “ILLEGAL DIRECT” error

INPUT statement

The INPUT statement is the preferred way of getting keyboard input from a user. Similar to the GET it can take keyboard input and assign it to a variable or a list of variables. Also, the INPUT statement can only be used in program mode. However, that is where similarities end. The INPUT statement can display a prompt with a question mark as well as displaying the text a user enters. It is possible for a user to correct the input using the delete or cursor keys. The program waits for data and the on-screen input can be edited until the RETURN key is pressed. Once RETURN is pressed the statement processes the input and the program resumes. The syntax is the keyword INPUT followed by an optional prompt, if the prompt is

used it must be followed by a semicolon, then a variable name or a list of variables separated by commas. When the simple example below is run you should notice the “?” and the flashing cursor. The program is waiting for input.

```
10 LET X$="X16"
20 INPUT "ENTER THE COMPUTER MODEL NUMBER";X$
30 PRINT X$
```

Listing 11 – Example of INPUT Statement

The INPUT statement is programmer and somewhat user friendly too. In the above example if you enter a string of text over 255 characters, longer than allowed for a string variable, the command truncates the text. When truncating text, no error will be created so your program can continue running. The way the text can get truncated can be **TODO** how is text truncated **TODO** If nothing is entered and the RETURN key is pressed the contents of the variable will be left as is. The statement also has basic data type checking. If the variable passed to it a numeric variable, then the data entered by the user must be numeric data as well.

```
10 INPUT "WHAT IS THE NUMBER AFTER X IN X16";X
20 PRINT X
```

Listing 12 – Example of INPUT statement

In the above example if you type text and press enter a “REDO FROM START” error will be displayed, and the prompt will be displayed on a new line again. The correct data type must be entered.

The INPUT statement can be passed a list of variables for input. There is a major difference when there are multiple variables with respect to variable existing value being replaced. If no value is entered at all and the RETURN key is pressed the INPUT statement will process it as zero value and replace the existing value in the variables with zero. If a single value is entered and RETURN key is pressed the INPUT statement processes the input and realises that other inputs are required. The program will then prompt with a “??” for the next value. The double? Represents that more input is required from the user.

```
10 INPUT "ENTER IN VALUES FOR X,Y AND Z";X,Y,Z
20 PRINT "X IS =",X," AND Y=",Y," AND Z=",Z
```

Listing 13 – Example of INPUT statement

Another important feature is when using multiple variables, the comma character becomes the separator between values. The below example shows three variables being input from a user.

```
10 INPUT "ENTER IN VALUES FOR X,Y AND Z";X,Y,Z
20 PRINT "X IS =",X," AND Y=",Y," AND Z=",Z
RUN
ENTER IN VALUES FOR X,Y AND Z? 1,10,100
X IS =  1   AND Y =  10.  AND Z=  100
```

Listing 14 – Example of INPUT statement

Be aware of the comma and the type and amount of data the user may want to enter. For example, if the values of X, Y or Z had been 1000 or 10000 the user may have entered 1,000

or 10,000 as the values. The INPUT statement will interpret the comma as the separator. So, if the user had entered “10,000, 1,000, 10” the INPUT statement will have stored the value “10” in the first variable, “000” in the second variable and “1” in the third variable. The other values would have triggered the “?EXTRA IGNORED” warning message. Whenever extra input is sent to the INPUT statement it will ignore the extra data and display this warning message. If the incorrect type of data is entered, then the “?REDO FROM START” error message will be displayed. When this error is displayed none of the variables will be updated and the user will have to input all of the data again from the start.

TODO check
Normally it is possible to press RUN/STOP to stop processing of a BASIC program. However, while the INPUT statement is waiting for input this is not possible. To break out of the program you must press RUN/STOP and RESTORE.

Summary - INPUT

- The syntax is the keyword INPUT, followed by an optional prompt with a semicolon. Then a variable or variable list
- The program will wait for a user input
- When there is more than a single variable used the comma in the input is treated as a separator
- The optional prompt will be displayed with a question mark after it
- If too much data is entered a warning message “?EXTRA IGNORED”
- If not, enough data is entered “??” will be displayed on a new line with the cursor waiting for new input
- If too much data is entered a warning message “?EXTRA IGNORED” will be displayed and extra data will be ignored
- If the wrong data type is entered as data, then the error message “?REDO FROM START” will be displayed and all data must be re-entered
- The INPUT statement cannot be used in immediate mode, if attempted a “?ILLEGAL DIRECT ERROR” will be displayed

MOUSE command

The X16 has support for mice built in unlike many of the early 1980s computers. The simple command MOUSE can enable or disable the mouse pointer. With this command it is possible to create a basic GUI for your program or use the mouse for a game. The syntax is

the keyword MOUSE followed by 1 to enable the mouse or 0 to disable it. When the mouse is enabled, the kernel will update several special integer variables.
TODO $FF option, configure custom mouse pointer.

```
MOUSE 1
```

Virtual screen 55

Summary - MOUSE

- The X16 supports a mouse
- Syntax is the keyword MOUSE followed by a 1 to enable or 0 to disable

Mouse Detail (MX/MY/MB) Integer function

The MX, MY and MB functions will read information about the mouse. The MB function will return a value ranging from 0 to 4. If the left mouse button is currently pressed the value will be one, if the right mouse button is pressed it will be 2, the third button will return the value of 4, otherwise it will be zero. The MX function will return a value ranging from 0 to 639 which represents its X axis location on screen. The MY function will return a value ranging from 0 to 479 which represents its Y axis location on screen. These integer functions cannot be used as statements by themselves, doing so will create a "?SYNTAX ERROR". These functions must be used by other statements to provide information about the mouse pointer. The example below demonstrates enabling the mouse pointer, using MX and MY to display its location and using MB to work out if a mouse button is being pressed.

```
10 MOUSE 1
20 PRINT "MOUSE IS CURRENTLY AT:",MX,MY
30 IF MB=1 THEN PRINT "LEFT MOUSE BUTTON PRESSED"
40 GOTO 20
```

Listing 15

Joypad (JOY) statement

The X16 comes with two NES/SNES style joypad/joystick ports. This type of joypad enables more button options than many vintage computers. Compared to the old Commodore 64 that had two joystick ports, each with four directions and fire buttons 1 and 2. However, very few games used fire button 2. Each of the X16 two joypad ports has four directions and four function buttons, allowing a lot of flexibility. The function buttons are "A", "B", "Select" and "Start".

Reading and processing user input from a joypad is essential for games. The JOY(N) statement allows us to query the state of either joystick port. It will return the status of all of the buttons. The value returned is a decimal value. The syntax is JOY(N), where N is either 1 or 2. 1 represents joypad port one and the value 2 represents port two. The abbreviated

form of the command is J and Shift-O. This command works in immediate and program mode.

```
PRINT JOY(1)
 16

READY.
PRINT JOY(2)
 0
```

Virtual screen 56 – Example of JOY command

The value 16 may appear odd. When we look at the below table, we see that there are some keyboard keys that also map to the joypad actions. The return key maps to the Start button for Joy 1 and has a value of 16.

Each button pressed on the joypad is equal to a particular value.

Table 12 – Joypad button values

Value	1	2	4	8	16	32	64	128
Joypad 1 or 2	Right	Left	Down	Up	Start	Select	B	A
Keyboard Mapping for Joy 1	Cursor Right	Cursor Left	Cursor Down	Cursor Up	Return	Space	Alt	Ctrl
Keyboard Mapping for Joy 2	None	None	None	None	None	None	None	None

The different combinations of these values let us know which buttons are currently pressed. For example, when the Right, Up, and B buttons are pressed simultaneously, the value would be 1 + 8 + 64 = 73. Another example could be Left, Down and A, the value being 2 + 4 + 128 = 134. Using the different combinations of these values lets us program all the possible valid combinations the player can use.

```
10 J=JOY(1)
15 PRINT J
20 IF J = 0 THEN GOTO 100
25 IF J = 1 THEN PRINT "RIGHT":GOTO 100
30 IF J = 2 THEN PRINT "LEFT":GOTO 100
35 IF J = 3 THEN PRINT "RIGHT AND LEFT":GOTO 100
40 IF J = 4 THEN PRINT "DOWN":GOTO 100
45 IF J = 8 THEN PRINT "UP":GOTO 100
50 IF J = 9 THEN PRINT "UP AND RIGHT":GOTO 100
55 IF J = 10 THEN PRINT "UP AND LEFT":GOTO 100
60 IF J = 16 THEN PRINT "START":GOTO 100
65 IF J = 24 THEN PRINT "START AND UP":GOTO 100
70 IF J = 32 THEN PRINT "SELECT":GOTO 100
75 IF J = 40 THEN PRINT "SELECT AND UP":GOTO 100
80 IF J = 64 THEN PRINT "B":GOTO 100
85 IF J = 68 THEN PRINT "B AND DOWN":GOTO 100
90 IF J = 72 THEN PRINT "B AND UP":GOTO 100
95 IF J = 128 THEN PRINT "A":GOTO 100
100 REM UPDATE PLAYER
110 REM UPDATE OTHER ELEMENTS
120 REM UPDATE GRAPHICS AND DRAW TO SCREEN
130 GOTO 10
```

Listing 16 – Example program using JOY command

Summary - JOY

- The X16 supports two NES/SNES joypads
- The syntax is JOY, followed by a (with the value 1 or 2, followed with a closing)
- Abbreviated JOY command is J Shift-O

Error reporting

In immediate mode when a mistake was made, the X16 would report a SYNTAX ERROR straight away.
IF X=1 PRINT “X EQUALS 1”

?SYNTAX ERROR
However, in the deferred mode, you will not receive an error until the program is executed. The X16 will provide extra information than the immediate mode. The error message will report the line number that has the error. This becomes very important in larger programs.
LIST
10 IF X=1 PRINT “X EQUALS 1”
RUN

?SYNTAX ERROR IN 10
READY.
We can see in the above example that it is missing the THEN keyword from the IF statement.

Error Messages

TODO Page 69 from Computes Programming The commodore 64 – the definitive guide revised edition.

Interacting with devices

Easily interacting with a device can be done with a file. The file is then read or written to, performing input or output. The X16 has four main file types: PRG, SEQ, REL, USR.

PRG, are generally program files, which contain executable code. When the X16 LOADs a PRG file it will read the first two bytes. The first two bytes indicate the location in memory the executable code should be loaded into. PRG files can also be used for data files.

REL is a relative file. These files do have a basic file structure. They contain an index. Each record can be 254 bytes in size. With the index is it possible to read/write to any part of the file.

SEQ, are sequential data files. Because these files are stored sequentially, they are read from start to end. They are often used for storing a single file such as a text file document, a graphic file or other such user or data file. SEQ files are flat and contain no structure. This means it is not possible to move to a particular location inside the files.

USR is a user-specified file. They are the same as SEQ files in that they are plain data formats. No part of the file has a specified meaning unlike with REL and PRG file types.

OPEN command

Like modern computers, when accessing a file, the X16 must create a unique variable to manage the file. These special variables are called file descriptors or filehandles. In BASIC the filehandles are called logical file numbers. Logical file numbers must be created before other commands can access a file. The OPEN statement makes these logical file numbers. The syntax is the OPEN keyword followed by the logical file number parameter, a comma, the device number parameter, a comma, an options secondary address parameter, a comma, and an optional command string.

The logical file number can be from 1 to 255. **TODO** CHECK IF FILES OVER 128 USED BY X16 ** It is a good idea to keep your file numbers from 1 to 127. When there is more than one file open the logical file numbers used must be unique. All input and output statements use the logical file number to access that particular file. The device number can be 0 to 15, see the below table.

All logical file numbers used must be unique.

The secondary address number can be 0 to 15. This parameter passes different information depending on the device. The value of 15 is reserved as the command channel to the device controller.

Device	Device#	Secondary Address Number	String
Keyboard	0		
?was cassette	1		
Modem	2	0	Control registers
Screen	3		
Printer	4 or 5		
Disk	8 to 11	0 = LOAD program file 1 = SAVE program file 2-14 = Data channel 15 = Command channel	

Table 13 – Device and address codes.

The command string is interpreted by the device being accessed. As a result, the command string can be varied. When dealing with the SD card it can have three parts. The filename and then the optional parameters of type and mode. The type can be left blank but if done so the mode parameter should be omitted. Type can be PRG, SEQ, REL, USR or ID. **TODO** ID is a new type which is…. The mode parameter can be R for read access, W for write access and **TODO**

```
10 OPEN 10,8,
```

Summary - OPEN

- OPEN command used to create logical file number which enables input/output
- The syntax is the OPEN keyword followed by the logical file number parameter, a comma, the device number parameter, a comma, an options secondary address parameter, a comma and then an optional command string
- Logical file numbers are used to differentiate between currently open files
- Logical file numbers can be from 1 to 255

INPUT# statement

Just like the earlier INPUT statement INPUT# will

TODO

Because INPUT# is for us on files or devices it is possible to do a little trick to get input from the user without an? appearing on screen similar to INPUT. We open the keyboard like any other device. The keyboard is device 0. Using the OPEN command, we open file 1 from device 0. Consider the below example.

```
10 OPEN 1,0
20 PRINT "ENTER A WORD:";
30 INPUT#1,A$
40 PRINT
50 PRINT "YOU ENTERED=";A$
```

Listing 17 – I/O example using OPEN and INPUT#

GET# statement

Sometimes reading in a single character at a time is what is required. For this task, the GET# statement is just like the GET statement. However, instead of getting a single character from

the keyboard, it will get the data from a logical file number. The syntax is the keyword GET followed immediately by the hash symbol (#) then a comma followed by a list of variables separated by commas.

```
10 OPEN 1,8,2,"HIGHEST-SCORE"
20 GET#1,A$: PRINT A$
30 CLOSE 1
```

Virtual screen 57 – Example of GET# statement

Status (ST) reserved variable

When performing any input/output action on an open file the KERNAL will store a status code in the reserved variable ST. The status code is an integer that can have several different values based on the outcome of the I/O action. While the status code can be read as an integer, the KERNAL is simply setting a single bit value.

ST Bit Set	ST Value	Meaning
%0000 0001	1	
%0000 0010	2	
%0000 0100	4	
%0000 1000	8	
%0001 0000	16	
%0010 0000	32	Checksum error
%0100 0000	64	End of File (EOF)
%1000 0000	128	Device not present?

Table 14 – Status codes

PRINT# statement

The PRINT# is a more flexible version of the PRINT statement. It is possible to write data to any device not just the video display. The syntax is the keyword PRINT followed immediately by the hash symbol (#) then the logical file number followed by a list of variables separated by commas.

The below example will save the contents of three variables to disk.

```
10 OPEN 1,8,2,"PRINT-FILE-DATA,USR,W"
20 A$="COMMANDER X16"
30 B$="BASIC"
40 C$="PRINT# DATA TO FILE"
50 PRINT#1,A$,B$,C$
60 CLOSE 1
```

Listing 18 – I/O Example saving data to SD card using PRINT#

The below example will read back the three variables from disk and display.

```
10 OPEN 1,8,2,"PRINT-FILE-DATA,USR,R"
20 INPUT#1,A$,B$,C$
30 PRINT "DATA=",A$,B$,C$
40 CLOSE 1
```

Listing 19 – I/O Example reading data from SD card using INPUT#

CMD command

TODO By default, the X16 displays output to the screen. We can change that using the CMD command. We can redirect output from the screen to a logical file. The actual file can be on disk, sent to the printer or other I/O device.

CLOSE command

After work has been completed on a file it should be closed so that all buffered data is written to the disk or device. Once a file is closed the logical file no longer exists. Memory that was being used is then released. The logical file number can now be reused if required. The syntax is the keyword CLOSE followed by the logical file number.

Summary - CLOSE

- It is important to CLOSE a file so that buffered data is written to the file
- Syntax is the keyword CLOSE followed by the logical file number

TODO demo open file and keyboard, get input and save to disk and close file and keyboard.

WAIT statement

The WAIT statement causes the current program processing to wait until a location in memory matches a specific bit pattern. The memory location is often pointed towards a register used by a device. The device could be the keyboard, internal timers or an external device via the IEC or serial port.
TODO

Debugging

"Computers are like Old Testament gods; lots of rules and no mercy." V-Joseph Campbell, The Power of Myth

Debugging is the process of finding and resolving errors in your program. The chances of your program running perfectly the first time are not good. It is easy to create a bug in software without realising it. Simple typing mistakes, missing a semicolon, forgetting a statement. Then the more difficult bugs where there is a logical error, normally based on false assumptions. When debugging you need to find and fix all of the problems with your software. With modern computers and software there are many tools and techniques to

assist debugging. BASIC has a few commands which can be used for debugging. We will cover the techniques or post-mortem debugging and print debugging.

Post-mortem debugging

Post-mortem debugging is the process of debugging a program that has stopped and reported an error message. Generally fixing these sorts of bugs is the easiest. Errors are inevitable, even for experienced programmers. The error message being displayed is a diagnostic tool pointing you to the error. Normally the error messages will not only tell you the error but the line in the program which is triggering the error. Unfortunately, BASIC error messages can be a bit cryptic. The below table includes common errors and possible causes for the error.

Error Message	Possible Cause
?BAD SUBSCRIPT	An array has been dimensioned improperly. Check the DIM statement and confirm number of variables and elements in the array.
?BREAK	The program was stopped. It is possible to use the CONT command to continue from the break point. It is possible to use the RUN command to re-run the entire program.
?CAN'T CONTINUE	The program cannot be continued using the CONT command. You will this error if the program has not been RUN yet. The error can be caused if program was edited after it was stopped, which erases variables. If the CLR command was processed it has erased the variables. Or a direct mode error has occurred which the system cannot distinguish from a program error.
?FILE OPEN ERROR IN <LINE>	An OPEN statement is attempting to use a logical file number which is currently open and in use.
EXTRA IGNORED	Too much data was entered in response to an INPUT statement
ILLEGAL QUANTITY	The number or input used as part of a statement or function does not make sense.
?ILLEGAL DIRECT	A statement or command has been used which is not valid in direct mode. The statement or command can only be used in program mode. For example, INPUT#
NEXT WITHOUT FOR	There is an error with a FOR…NEXT loop structure. Check to see if the variable used for the NEXT statement is correct.
OUT OF DATA	The program attempted to READ data, however there was none left to read.
OUT OF MEMORY	The X16 has no memory available for the program. Or the program has run out of stack space from using to many subroutines (GOSUB statements) or to many FOR…NEXT loops.
REIM'D ARRAY	The DIM statement has been used incorrectly. Or you have tried to use an array (with a subscript size larger than 10) before it was dimensioned.
REDO FROM START	A character or string was input to the computer when a number was required.

RETURN WITHOUT GOSUB	A RETURN statement has been used without a matching GOSUB command.
?SYNTAX ERROR	Most likely there is a punctuation or spelling error which breaks the syntax of a command.
UNDEF'D STATEMENT	The program flow has been directed to a non-existent line number by GOTO or GOSUB.

Print debugging

Print debugging a simple method and uses the print statement. It is simply a matter of placing print commands in your code so you can see the progress of the code. The print statement should be providing as much information as possible. Print statements should display some location data as well as the value of variables to assist with your understanding of what is happening. Modern programming tools have the idea of debug flags and code. BASIC does not have this, but we can recreate it using a variable to set if we want to display our debug code or not. We can use a variable to set a flag and have different parts of our problem response to the debug flag. For example, debug=1 could mean all debug code in module 1 runs, debug=8 meaning all debug code in module 8 runs and so on. This method might be handy is large programs so you might not need all the debugging code to be running, just the troublesome section. It is worth noting before you release your program it should be optimised to remove the debug code. Optimization is covered in a later chapter. Consider the below example.

```
10 DB=1 REM SET DEBUG LEVEL TO 1
20 FOR I=1 TO 10
30 FOR J=1 TO 10
40 REM MAJOR TASK 1
50 REM PRINT DEBUG
60 IF DB=1 THEN PRINT "TASK 1 - DEBUG: I=",I,"J=",J
70 NEXT J
80 NEXT I
```

Listing 20

Another method of print debugging is having a subroutine that displays information. Using RUN/STOP to break out of a loop or the program and call the subroutine. Consider the below example, using GOSUB 60000 to display debugging information.

```
10 FOR I=1 TO 100
20 FOR J=1 TO 100
30 REM MAJOR TASK 1
40 LET X=X+J*I: LET Y=X/2
50 NEXT J
60 NEXT I
70 END
60000 REM DEBUG INFO
60010 PRINT "-DEBUG INFO -"
60020 PRINT "LOOP COUNTERS: I=",I,"J=",J,"K=",K
60030 PRINT "VARIABLES:   X=",X,",Y=",Y
60040 RETURN
```

Listing 21

Breakpoints

Breakpoints are not a modern idea; they were invented for the ENIAC computer in 1945. They are simply intentionally pausing or stopping a program. During this pause, the computer's memory and CPU can be inspected and modified if required. Modern software

development environments allow for breakpoints to be configured easily and for the programmer to step thru the execution of the program. Using breakpoints is a very important technique for a programmer. BASIC provides two statements that give us some of the functionality, the STOP and CONT statements.

STOP statement

The STOP statement is used to stop the currently running program and switch modes from program mode to direct mode. Using the RUN/STOP key has the same effect as a STOP statement, except you can place it in the program exactly where you want it to stop. Multiple STOP statements can be placed strategically around your program to aid in debugging it. When used in your code it stops the program and displays the warning message "BREAK IN XX" where XX is the line number of the program. All the currently used variables stay in memory, and any open files remain open. It is then possible to view the contents of variables. Once a program has been stopped this way it is possible to continue the program running by using a GOTO line number, to have the continue running from that point. It is often used in conjunction with the CONT command.

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 LET X=X+J*I: LET Y=X/2
40 NEXT J
50 STOP
60 NEXT I
```

Listing 22

When faced with a difficult bug remember you can create a conditional breakpoint by using some simple IF…THEN statements. The bug may only be occurring under a specific condition. Consider the below code.

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 LET X=X+J*I: LET Y=X/2
40 IF J=5 AND X>10000 THEN STOP
40 NEXT J
50 IF I>7 THEN STOP
60 NEXT I
```

Listing 23

Combining the idea of debug levels and conditional breakpoints we can create very flexible debugging code to help troubleshoot those different to solve problems.

CONT statement

This statement allows a program to continue after a STOP statement, an END statement or RUN/STOP. This command is useful for programmers and is often used with the STOP statement. Using the CONT statement the program will continue from the exact location it was stopped. When a program has been stopped a programmer may view the listing of the program. Editing any line of the program, even pressing return on an unmodified line, results in a change, will prevent the CONT statement from working. If the program is modified in anyway and the CONT statement is processed a "CAN'T CONTINUE" will be displayed. Experiment with the code listed below. Run the program and press RUN/STOP to stop the program. Just like with the STOP statement you will be able to view variables and list the program code.

```
10 LET PI=0: LET C=1
20 PI = PI+4/C-4/(C+2)
30 PRINT PI
40 LET C=C+1
50 GOTO 20
```

Listing 24

Use the CONT statement and notice that the program continues from where it left off.

If the program is modified in anyway, it will prevent the CONT statement from working.

Summary

- Post-mortem debugging involved reviewing the error messages from the system and working out what caused it.
- Print debugging requires extra code to be placed in the program to print out messages and variables to assist with troubleshooting.
- Breakpoints allow program to be paused at a position that interests the programmer. While paused the contents of the variables and code can be reviewed.
- It is possible to continue after a pause so long as no program code is modified. The value of variables can be modified.
- The STOP statement is used to create breakpoints, pausing the program
- The CONT statement is used to continue processing the program
- It is possible to review the contents and status of the CPU and memory using the MONITOR command.

BASIC Errors

In BASIC you cannot really ignore errors because the program stops. However, when you experience them you should stop and pull part from is happening. Making a quick fix to get around the error is lazy programming and not a good strategy for quality code.

You want to make sure your code is solid and easy to read. Otherwise, you could end up with brittle code. Code which you will be excited to search for hours on hard-to-find bugs.

Getting a lot of errors, your code could be poorly structured. It might be time to review your design. Your design should be able to handle all conditions. Everything a user could do, but also all the possible values that could be given to it. Asking a user, a ‘Yes’ or ‘No’ question, you should also check for other input.

Why people do not try to prevent errors?

"It's extra work."
"This function will never get that input."
"It's a toy program so it doesn't matter."

However, I want you to create good code. To learn good habits now so you carry them over to your future programming projects.

TODO list BASIC errors, about 30+ of them.