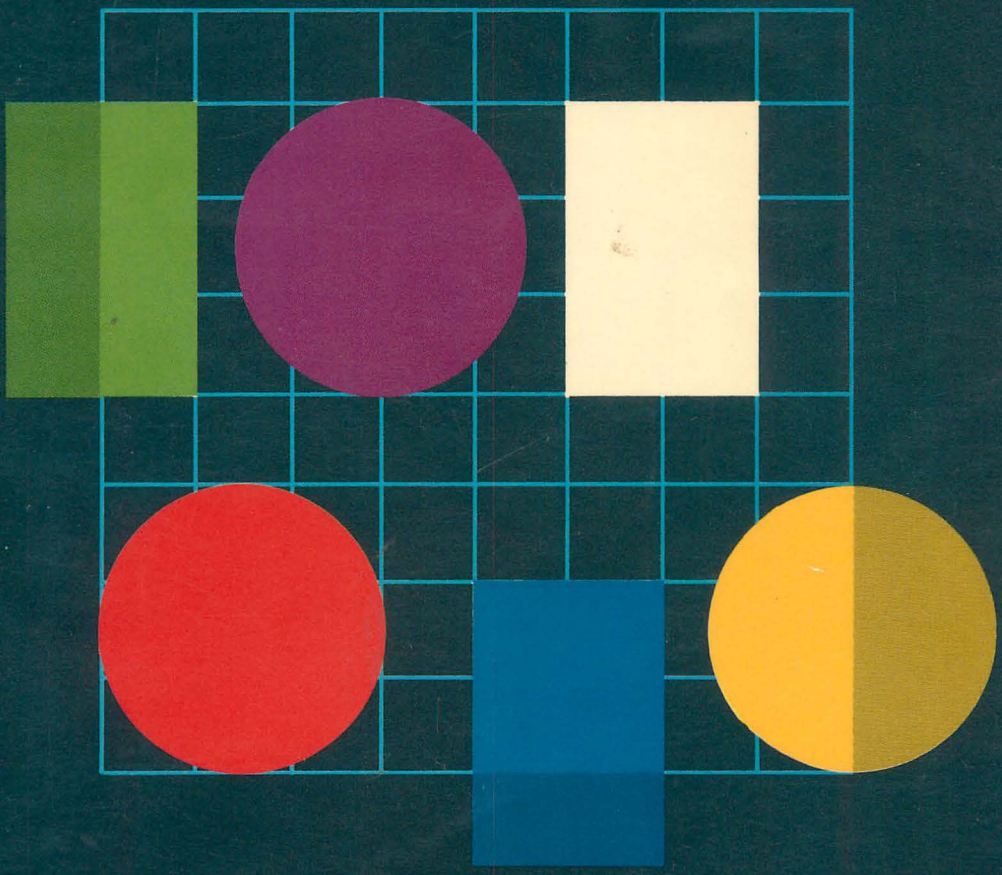
PRENTICE HALL INTERNATIONAL PERSONAL COMPUTER BOOK

COMMODORE 64 ADVANCED USER GUIDE

JOHN GORDON and IAN McLEAN



Commodore 64
Advanced User Guide

Other Commodore books by John Gordon and Ian McLean

100 PROGRAMS FOR THE COMMODORE 64*
100 PROGRAMS FOR THE COMMODORE C16†

* Available from your usual supplier

† Available from your usual supplier, or in case of difficulty from:

Prentice-Hall International (UK) Ltd
66 Wood Lane End
Hemel Hempstead
Herts HP2 4RG
England

Commodore 64
Advanced User Guide

John Gordon and Ian McLean
MEDC, Paisley College, Scotland

Englewood Cliffs, NJ London Mexico New Delhi Rio de Janeiro
Singapore Sydney Tokyo Toronto Wellington

Library of Congress Cataloging in Publication Data

Gordon, John, 1952–
Commodore 64 advanced user guide.

Includes index.
1. Commodore 64 (Computer)—Programming.
I. McLean, Ian, 1946– II. Title.
QA76.8.C64G673 1985 001.64 85-6534
ISBN 0-13-152026-1 (pbk.)

British Library Cataloguing in Publication Data

Gordon, John, 1952–
Commodore 64 advanced user guide.—(Prentice-Hall International personal computer book)
1. Commodore 64 (Computer)
I. Title II. McLean, Ian, 1946–
001.64'04 QA76.8.C84
ISBN 0-13-152026-1

ISBN 0-13-152026-1

Prentice-Hall Inc. *Englewood Cliffs, New Jersey*
Prentice-Hall International (UK) Ltd, *London*
Prentice-Hall of Australia Pty, Ltd, *Sydney*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall Hispanoamericana, SA, *Mexico*
Prentice-Hall of India Private Ltd., *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte, Ltd, *Singapore*
Editora Prentice-Hall do Brasil Ltda, *Rio de Janeiro*
Whitehall Books Ltd, *Wellington, New Zealand*

10 9 8 7 6 5 4 3 2 1

Printed in the United Kingdom by A. Wheaton & Co. Ltd, Exeter

To Teresa and Anne

Contents

Preface

This book is written for those who want to make proper use of the many features and facilities provided by that remarkable computer, the Commodore 64. If you have become bored with the games programs and fret under the limitations and restrictions of BASIC then this book is written for you.

We have taken two distinct approaches to the problem of getting more out of the micro. The first approach is to go deeper into the machine. Machine code and assembly language give you full control, with no BASIC interpreter coming between you and what you wish to do. Machine code programs overcome the speed restrictions so evident in BASIC. The book explains what machine code and assembly language is, and how code can be entered via a monitor. A minimum facility monitor program is given, and all the features of a full facility, commercially available monitor are described. 6510 Assembler is covered in some detail, with examples where appropriate. Machine code subroutines, interrupt routines and operating system calls are covered, enabling complex programs to be built up in simple logical steps.

Armed with a knowledge of assembly language programming, we take another look at BASIC - far too useful a tool to be discarded altogether. We see how the BASIC in the machine can be improved by adding utilities or changing the language itself. We see how machine code and BASIC can be combined, in particular to implement a range of high resolution graphics programs. We see how BASIC programs can be made to run faster and take up less memory by the use of a compiler.

Our second approach is to step back from the machine; to consider it as a tool to run commercial software packages. We look at word processing, spreadsheets and database, with practical examples wherever appropriate.

Common to both approaches is the need for the micro to control peripheral devices. The final chapters of the book discuss the machine's input/output capabilities.

CHAPTER 1

Introduction to Assembly Language

INTRODUCTION

This chapter introduces machine code and assembly language. It examines the 6510 Central Processing Unit from the view point of the assembly language programmer. We look at machine code monitors and discuss the constraints to be considered when locating machine code routines in memory.

MACHINE CODE

Computers work only with ones and zeroes. When you tell the computer to do something, using words which are comprehensible to humans (PRINT, READ etc), these commands are translated into strings of binary numbers. There is a group of programs held permanently in computer memory which does this - this group of programs is known as the operating system.

The binary numbers which the computer understands are called 'machine code'. If we know which numbers do what, we can communicate directly with the machine in its own language.

Hexadecimal notation is a convenient shorthand method of writing binary numbers. When we are working in machine code we normally use hexadecimal format. The symbol $ before a number is used to indicate that the number is hexadecimal.

To help us remember what each number does we give each one a name, or mnemonic.

Thus (for example):

1110 1000 (=$E8) is given the mnemonic INX

A program to change these mnemonics back into machine code is called an assembler. A program to change a high level (English-like) language such as BASIC into machine code is called a compiler or an interpreter.

The programs held within a computer's operating system to

perform the latter function are usually interpretive. A compiler converts a BASIC program to machine code before that program is loaded into the computer.

THE 6510 MICROPROCESSOR

The Commodore 64 is controlled by the 6510 microprocessor system. The microprocessor is working all the time while the machine is switched on. It reads binary numbers from memory, interprets these numbers as instructions, data or memory addresses, and carries out operations on the basis of this information.

To do this the microprocessor uses special number stores contained in the Central Processing Unit (CPU) and known as registers.These are given in figure 1.

The accumulator is involved in most of the arithmetic and logical operations carried out in the machine. The result of such an operation is usually placed in the accumulator.

The index registers are used to store values for counting, timing and 'indexing' (identifying a memory address or series of addresses with reference to a 'base' address).

The program counter holds the memory address of the instruction currently being carried out.

The stack pointer holds the address of the start of a section of memory known as the stack. The stack is used to hold information when the microprocessor is diverted from its main task to go and do something else. For example subroutine return addresses are stored in the stack.

The status register is a collection of single bits or 'flags' which may be set (to 1) or reset (to 0) depending on the result of operations within the machine. We shall return to each of these flags when we discuss in detail the operations which affect it. At present it is sufficient to list the flags and give brief details of their main functions.

The Accumulator (A)

The Index Registers (X and Y)

The Program Counter (PC)

The Stack Pointer (S)

The Status Register (P)

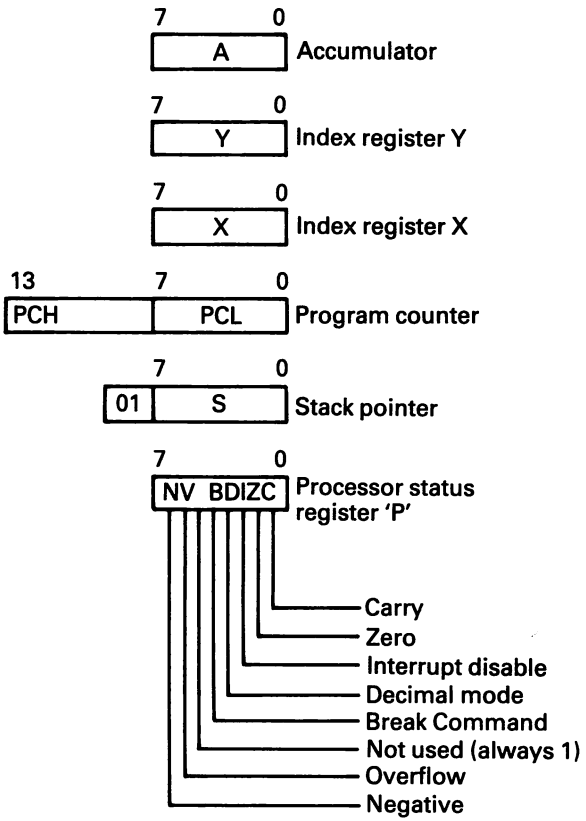


Figure 1

The carry flag is set when the result of an arithmetic operation is greater than 255 and reset otherwise. This flag may also be set or reset directly by a machine code routine (see chapter 2).

The zero flag is set or reset depending on whether the result of an operation is zero or non zero (see chapter 2).

The interrupt flag determines if a program currently running in the processor can be interrupted on receipt of a signal from a peripheral device (see chapter 3).

The decimal flag is set when arithmetic operations are required to produce a decimally adjusted result (see chapter 2).

The break flag indicates whether an interrupt routine (see chapter 3) has been entered because of a signal from a peripheral device or because of an instruction in the main program.

The overflow flag is set or reset during arithmetic operations when the result of such operations cannot be contained in the seven least significant bits of the accumulator. This flag is mostly used during signed arithmetic operations (see chapter 2).

The negative flag contains the value of the most significant bit of the accumulator, which is considered as the sign bit in signed arithmetic operations (see chapter 2).

the status register is sometimes known as the Processor Status Word (PSW).

ASSEMBLY LANGUAGE

Let us look again at the names or 'mnemonics' we give to machine code instructions.

For example:

110 1000 is given the mnemonic INX

This instruction causes the number held in index register X to be incremented by 1. That is the new number held in X after the instruction is carried out is equal to one more than the number which was previously held in X before the instruction was carried out.

INX is short for INcrement X.

the number 1110 1000 is known as the operation code or op-code for the instruction INX. Op-codes are normally written in hexadecimal.

Thus:

$E8 is the op-code for INX.

In general, the mnemonics used to represent numbers which the microprocessor interprets as instructions refer to the various registers and flags. For example:

STA is STore the Accumulator in memory.

LDA is LoaD the Accumulator.

SEC is Set the Carry Flag.

The group of mnemonics thus generated is known as the Assembly Language for the 6510 microprocessor. A list of 6510 Assembly Language instructions is given in Appendix A.

'Machine code programmers' do not actually program in machine code. They write their programs in assembly language and then convert this to machine code. The assembly language program is known as the 'source code'. The resulting machine code program is known as the 'object code'.

Not all the machine code numbers are op-codes. Consider, for example, the instruction:

LDA #$21

- this means LoaD the Accumulator with the value $21.

Note - The # symbol means that the accumulator is being loaded directly with a number rather than with the contents of another register or a memory location.

The machine code for LDA #$21 is:

$A9 $21

$A9 is the op-code for LDA #, but $21 is simply a number, which is loaded into A.

Another example is:

```
STA $0402
```

- this means STore the Accumulator in the memory location whose address is $0402 - i.e. change the number held in that memory location so that it is equal to the number held in the accumulator.

The machine code for this is:

```
$8D $02 $04
```

$8D is the op-code for STA.

$02 and $04 give the address at which the accumulator is to be stored.

Addresses in machine code are written least significant byte first.

Machine code numbers may be op-codes, data or addresses. The microprocessor interprets them depending on what has come before.

SYS

SYS enables a machine code routine to be entered from BASIC. SYS is a BASIC instruction. It addresses the memory location where the first instruction of the machine code is stored. If, for example, a machine code routine starts at memory address 49152 ($C000), then the command:

```
SYS 49152
```

will transfer control to the machine code routine. SYS must have a decimal rather than a hexadecimal arguement.

RTS

RTS is an assembly language instruction. RTS stands for Return from Subroutine and its op-code is $60. If the machine code routine is called from a BASIC program (using the SYS instruction), RTS will return control to the next instruction in that program.

If SYS is used as a direct command, RTS returns the machine to direct mode.

RTS is often the last instruction of any user generated machine code routine.

PUTTING MACHINE CODE INTO MEMORY

Machine code is a series of numbers. These can be placed in memory locations using the POKE command directly or in a BASIC program.

This is best illustrated by an example. The next routine places the number $41 in memory location $C010. The routine is to be placed in memory starting at location $C000.

```
LDA #$41
STA $C010
RTS
```

Assembling this program in machine code gives:

```
$A9 $41
$8D $01 $C0
$60
```

The POKE command may be used to put this routine into memory. Unfortunately POKE cannot use hexadecimal numbers; so the machine code must be converted to decimal.

This gives: 169,65,141,1,192,96.

We could POKE 169 into memory location 49152 ($C000), 65 into memory location 49153 and so on. Alternatively we could use:

```
10 FIRST=49152
20 NUM=6
30 FOR K=0 TO NUM
40 READ X
50 POKE FIRST+K,X
60 NEXT:END
70 DATA 169,65,141,1,192,96
```

RUN this program. It will place the machine code into memory, but will not execute it.

If, however, you enter the command:

```
SYS 49152
```

the computer will execute the machine code routine and then return to the 'Ready' state. Memory location $C010 will now contain $41.

ADVANTAGES OF MACHINE CODE

Machine code routines normally take up less memory space than do the equivalent programs in BASIC. Also, if the BASIC interpreter is not required it can be switched out of memory, leaving more space for user programs. Machine code routines operate very much faster than BASIC programs.

A machine code routine will remain in memory until it is overwritten by some other routine or until the machine is switched off. It is not cleared by the NEW command. Thus we can enter and subsequently clear any number of BASIC programs which use a particular machine code routine, but need load that routine only once.

MACHINE CODE MONITORS

A monitor is a program which allows machine code routines to be entered into memory directly from the keyboard, allows memory contents to be examined, and allows machine code routines to be saved to and loaded from a storage medium.

MINIMUM FACILITY MONITOR

The program listed in Appendix K provides the minimum facilities required for a machine code monitor. Addresses and byte values may be specified in either decimal or hex. If hex is chosen the $ symbol is understood and should not be typed in.

Wherever necessary in program listings we have used the following convention to represent the Commodore 64 screen editing characters.

All strings of the screen positioning characters start with the character [and finish with the character].

cursor-home is replaced by CH
clear-screen is replaced by CS
cursor-down is replaced by CD
cursor-up is replaced by CU
cursor-left is replaced by CL
cursor-right is replaced by CR
reverse-on is replaced by RON
reverse-off is replaced by ROF

COMMERCIAL MONITORS

The monitor given (in Appendix K) enables an assembled machine code routine to be placed in memory. It is necessary before using the routine to assemble the source code by hand to produce the object code. This is a tedious and error-prone process for routines of any length. Even more time consuming is changing the numbers read from memory back into assembly language in order to debug a routine. This latter process is known as disassembling.

A commercial monitor should allow the user to work in assembly language. Assembly and disassembly should be automatic.

A typical commercial machine code monitor will probably be in plug-in cassette form and will provide this and a number of additional facilities, for example:

1. Placing a breakpoint in memory.
2. Comparing two sections of memory.
3. Filling a section of memory with a specified value.
4. Starting execution of a machine code routine.
5. Searching memory for a hex value or values.
6. Displaying the ASCII content of specified memory locations.
7. Loading a program from a storage device.
8. Displaying memory content in both hex and ASCII.
9. Sending output to printer.
10. Tracing a program.
11. Displaying the contents of the processor registers.
12. Removing a breakpoint from memory.
13. Saving a program to a storage device.
14. Transferring a section of memory from one location to another and relocating the code.
15. Allowing step by step execution of a program (walkmode).
16. Displaying the contents of a specified location during walkmode.
17. Allowing access to the Disk Operating System commands.
18. Switching memory banks between RAM or ROM.
19. Dumping screen to printer.
20. Changing the content of the registers.
21. Implementing auto repeat on all keys.
22. Cancelling auto repeat on all keys except CRSR and SPACE.
23. Displaying a helpscreen.
24. Converting hexadecimal to decimal.
25. Converting decimal to hexadecimal.

Some monitors also allow memory addresses to be given labels rather than address numbers, and allow instructions to be used in groups known as 'macros'.

MON64

We use the MON64 (tm) monitor from Handic Software AB. All commercial monitors come with instruction booklets and it is pointless to reproduce details of operation here. A summary of the commands is, however, of value as it indicates the command format of a typical monitor.

```
A $$$$ MNEMONIC          ASSEMBLE
B(1,2) $$$$ NNNN         SET BREAKPOINTS
D $$$$ ($$$$)            DISASSEMBLE
F $$$$ $$$$ NN           FILL MEMORY
G ($$$$)                 GO
H $$$$ $$$$ NN           HUNT
I $$$$ ($$$$)            INTERPRET
L "NAME",NN              LOAD PROGRAM
M $$$$ ($$$$)            DISPLAY MEMORY
N A1 A2 OFF A4 A5 (W)    RELOCATE
O#X                      OUTPUT
Q ($$$$)                 QUICKTRACE
R                        DISPLAY REGISTERS
RB (1,2)                 REMOVE BREAKPOINT
S "NAMN",NN,$$$$,$$$$    SAVE
T $$$$ $$$$ $$$$         TRANSFER
W ($$$$)                 WALK
V $$$$                   SHOW ADDRESS
X                        EXIT TO BASIC
@ (I,$,V,N,S,#X)         DOS COMMANDS
*                        RAM/ROM MODE
F1                       DUMP SCREEN
F3                       ALTER REGISTERS
F5/F6                    REPEAT ON/OFF
F8                       LOAD (MENU)
#$$$$ / $NNNNN           HEX<->DECIMAL
```

WHERE TO PUT CODE

Because there is RAM behind (accessed by the same memory addresses as) all the ROM in the Commodore 64, all 64K of memory is, in theory, available to the programmer. There are, however, some areas which are easier to use than others, and some which should not be used.

The easiest section of memory to use is between $C000 and

$CFFF. This gives four kilobytes of free RAM which is unaffected by other operations (e.g BASIC program storage, input/output operations etc).

The BASIC program space between $0800 and $9FFF is also easy to use, particularly if we are using only machine code and not mixing machine code and BASIC. In this case the monitor should be located between $A000 and $BFFF.

Note, however, that if we are using high resolution graphics we shall have to allocate 8K of the BASIC memory to the high resolution screen (see chapter 5). This memory is then no longer available for programs.

If we mix BASIC and machine code it is necessary to ensure that BASIC programs and data storage do not overwrite the machine code. In this case the pointers to the highest address used by BASIC (held in addresses $0039 and $003A) and to the bottom of string storage (held in addresses $0033 and $0034), must be altered. This is normally done in the first instruction of the BASIC program - for example:

```
POKE 52,48:POKE56,48:CLR
```

will set the highest memory location which can be used by BASIC programs to $2FFF, leaving $3000 to $9FFF free for machine code.

Code may be placed in the RAM behind the BASIC ROM ($A000 to $BFFF) and behind the KERNAL ROM ($E000 to $FFFF).

Bit 0 of the byte in memory location $0001 (LORAM) controls the BASIC area, and bit 1 of this byte (HIRAM) controls the KERNAL area. Normally both ROM areas are switched in and both these bits are set to one. Setting either bit to zero will switch out the corresponding ROM and switch in the RAM.

The RAM behind these ROMs may be used for machine code routines. It is, however, convenient to be able to use the routines in BASIC ROM and particularly useful to be able to use the KERNAL ROM (see chapter 3). Thus care must be taken in switching this RAM in and out. It can be used, but is not as easy to use as those areas previously mentioned. If a choice has to be made it is possibly preferable to use $A000 to $BFFF.

The memory area between $D000 contains both the character ROM

and the memory mapped I/O devices. This area of memory is controlled by bit 2 of the byte in $0001 (CHAREN). Normally this bit is 1 and the I/O devices are switched in. Setting CHAREN to zero switches in the character ROM.

This area cannot normally be used for machine code. Some cartridge based programs use this memory, in which case both the character ROM and the I/O devices are switched out and RAM or ROM in the cartridges switched in by means of the signals GAME and EXPROM on the cartridge parts (see appendix C).

Do not use $0000 to $03FF for routines. This area contains the system data. There are some spare locations (see memory map in appendix B), but not sufficient to hold any significant routines. The spare locations on page zero of memory $00FB to $00FE are useful for storing base addresses for page zero indirect addressing (see chapter 2).

Screen memory can be relocated by using the video bank selection facilities of the VIC II chip (see chapter 5). If this is done the default screen memory ($0400 to $07FF) may be used to hold code.

Care must be taken not to use addresses required by the monitor being used - for example MON64 uses the following RAM addresses:

$00AE - $00AF
$00B2 - $00B3
$00C0 - $00C4
$0200 - $0258
$02A7 - $02BD
$0334 - $033B
$BC00 - $BFFF

Routines to be run from MON64 should not use these locations.

SUMMARY

Machine code is a series of numbers loaded into computer memory. These numbers cause the computer to perform a specified series of operations. Assembly language is a series of mnemonic codes which represent these numbers.

Machine code routines are much faster and use memory more

efficiently than the equivalent BASIC programs. A monitor allows machine code routines to be entered directly from the keyboard.

The easiest area of memory to use for machine code routines is $C000 to $CFFF.

The BASIC program area $0800 to $9FFF may also be used for machine code provided that care is taken that BASIC routines do not overwrite the machine code.

The RAM area behind the ROM at $A000 to $BFFF and $E000 to $FFFF may be used, but not as easily as the other areas.

$D000 to $DFFF and $0000 to $07FF are not normally used to hold machine code routines.

CHAPTER 2

Data Handling Instructions

INTRODUCTION

This chapter discusses the movement and manipulation of data within the 6510 microprocessor system, and how the flags are affected by the various operations.

The following instructions are covered:

ADC - Add memory to accumulator with carry
AND - Logical AND memory and accumulator
ASL - Shift memory or accumulator left one bit
BRK - Break to monitor
CLC - Clear carry flag
CLD - Clear decimal mode
CLV - Clear overflow flag
DEC - Decrement memory
DEX - Decrement the X register
DEY - Decrement the Y register
EOR - Exclusive OR memory and accumulator
INC - Increment memory
INX - Increment X register
INY - Increment Y register
LDA - Load accumulator from memory
LDX - Load X register from memory
LDY - Load Y register from memory
LSR - Shift memory or accumulator left one bit
NOP - No operation
ORA - Logical OR memory and accumulator
ROL - Rotates memory or accumulator one bit to the left
ROR - Rotates memory or accumulator one bit to the right
SBC - Subtract memory from accumulator with borrow
SEC - Set carry flag
SED - Set decimal mode
STA - Store accumulator to memory
STX - Store X register to memory
STY - Store Y register to memory
TAX - Transfer accumulator to X register
TAY - Transfer accumulator to Y register
TXA - Transfer X register to accumulator
TYA - Transfer Y register to accumulator

The instructions JMP, BEQ and CMP are used in this chapter to enable the effects of the other instructions to be examined. These instructions are, however, explained more fully in chapter 3.

LOADING THE ACCUMULATOR

A value may be loaded into the accumulator using the LDA instruction. There are several versions of this instruction.

1. IMMEDIATE: In this case a value specified in the instruction is loaded,overwriting the previous contents. Thus

```
LDA #$56
```

loads the value $56 to the accumulator.

2. ABSOLUTE: In this case the value held in a specified memory location is loaded. Thus if (for example) memory location $5000 holds the value $4A, then

```
LDA $5000
```

loads the value $4A into the accumulator. The contents of the specified memory location are unaltered.

3. ZERO PAGE: This is a special case of absolute addressing. If the address of the memory location from which a data item is taken lies between $0000 and $00FF (known as the zero page) then it may be specified by its least significant byte. Thus if, for example, memory location $00FB contains $B8, then

```
LDA $FB
```

loads $B8 into the accumulator. The same task could, of course, be carried out by absolute addressing - LDA $00FB does exactly the same as LDA

$FB. The advantages of zero page addressing are that a zero page instruction takes up one less byte in memory and is faster in execution.

4. ABSOLUTE INDEXED: In this type of memory addressing the contents of an index register - either X or Y is added to the specified address to calculate the actual memory address from which data is loaded. Suppose, for example, the Y register contained the value $55, then

```
LDA $2000,Y
```

would load the value of the byte contained in memory location $2055 ($2000+$55) into the accumulator.

5. ZERO PAGE INDEXED: Zero page indexing may also be used, but only with the X register. Thus if X contains $02,

```
LDA $61,X
```

will load the accumulator with the value of the byte in memory location $0063.

6. INDIRECT INDEXED: Indirect addressing is an important assembly language concept. In indirect addressing the microprocessor does not take data directly from the address specified in the instruction. Instead it finds at that address a second address and from there it gets data.

This seems a complicated procedure, but it has a purpose. The address held in memory (sometimes called the pointer or vector) may be altered so that one instruction can call up, through the pointer, a number of addresses.

Some areas in computer memory are set aside to hold a series of addresses. Such an area is referred to as a wordtable or look up table.

One problem that springs to mind is that addresses require two bytes in a 64K system such as the 6510. A memory location can hold only a single byte. Thus two memory locations are required to hold an

address. If an indirect instruction specifies location $00FB the actual address of the data is held in locations $00FB and $00FC with the least significant byte in the first of these locations - i.e. if $00FB holds $00 and $00FC holds $CD, then the address pointed to by these locations is $CD00.

The LDA instruction uses only zero page addresses for indirect addressing. Indirect addresses are also indexed. There are two forms of indirect addressing associated with this instruction. The first uses the Y register and is known as indirect indexed. The contents of the Y register are added to the address found at the specified location so as to define the address from which a data item is read. Let us take two examples.

Suppose $00FB holds $40, $00FC holds $74, and the Y register holds $65. Then

```
LDA ($FB),Y
```

will read the contents of the memory location $74A5 ($7440+$65) into the accumulator.

Suppose $0002 holds $00, $0003 holds $60 and the Y register holds $F2. Then

```
LDA ($02),Y
```

will read the contents of memory location $60F2 ($6000+$F2) into the accumulator.

Indirect indexed is used when there are a number of target addresses - the addresses which contain the actual data - contiguous in memory. It gives access to as many sequential locations as required, whereas normal indexing gives access to only 255 ($FF) instructions from the base addresses. In indirect indexing the most significant byte of the base address can be incremented by 1 to point to a new page in memory each time the Y register reaches its limit.

7. INDEXED INDIRECT: The second form of indirect addressing uses the X register. In this case the

value in the index register is added to the address in the instruction rather than the pointer read from that address. For example, if the X register contains $04 then:

```
LDA ($16,X)
```

will load into the accumulator the value contained in that memory location whose address is held in locations $001A ($16+$04) and $001B.

Indexed indirect is used when pointer addresses are contiguous in memory (rather than the target addresses); its main funtion is to work with wordtables.

The accumulator may also be loaded from the X and Y registers by the instructions TXA and TYA respectively. These instructions do not affect the source register. They are said to use IMPLIED ADDRESSING. This means that both the source and destination registers are implied by the instruction and no other form of addressing is necessary.

Instructions which load the accumulator affect the zero flag (Bit 1) and the negative flag (Bit 7) of the status register.

If the number loaded into the accumulator is $00 the zero flag is set. Otherwise it is reset.

If the number loaded into the accumulator lies between $80 and $FF the negative flag is set. Otherwise it is reset.

LOADING THE X REGISTER

Index register X may be loaded from memory using the LDX instruction. As with the LDA instruction there are several addressing modes:

```
1.  Immediate          -e.g.   LDX #$6A
2.  Absolute           -e.g.   LDX $6000
3.  Zero Page          -e.g.   LDX $FD
4.  Absolute Indexed   -e.g.   LDX $2000,Y
5.  Zero Page Index    -e.g.   LDX $60,Y
```

This instruction does not use indirect addressing. Only the Y register (for obvious reasons) may be used as an index.

The X register may also be loaded from the accumulator by the TAX instruction.

The X register may be decremented by 1 and incremented by 1 by the DEX and INX instructions respectively.

TAX, DEX and INX are implied address instructions.

The instructions which load, increment and decrement the X register affect the negative flag (set if the value placed in the X register lies between $80 and $FF - otherwise reset) and the zero flag (set if the value placed in the X register is $00, othewise reset).

Note that increment and decrement instructions do not affect the carry flag. Thus if the X register holds $FF, then

```
INX
```

will change the value of the X register to $00, will set the zero flag and reset the negative flag, but will NOT affect the carry flag.

LOADING THE Y REGISTER

The Y register may be loaded, incremented and decremented in exactly the same way as the X register, except of course that the X register is now used in the indexed instructions. Thus we may have, for example:

```
LDY    #$F6
LDY    $5000
LDY    $FE
LDY    $4000,X
LDY    $61,X
TAY
DEY
INY
```

These instructions may affect the negative and zero flags, but NOT the carry flag.

STORING TO MEMORY

Placing a value into a memory location is known as 'storing'. The contents of the accumulator, the X register and the Y register may be stored in memory. Note that a memory location cannot be loaded immediately with a value.

THE ACCUMULATOR may be stored in the memory using absolute, zero page, absolute indexed (X and Y), zero page indexed (X), indirect indexed and indexed indirect.

Thus we may have, for example:

```
STA     $1000
STA     $FD
STA     $2000,X
STA     $4000,Y
STA     ($02),Y
STA     ($FE,X)
```

The next three programs illustrate the various methods of addressing memory for load and store instructions. They all perform the same task - to transfer the data held in memory locations $C100 onwards into memory locations $C200 onwards.

The first example uses direct addressing to transfer two bytes. This technique is NOT recommended for large quantities of data. The BRK instruction returns control to the monitor.

```
C000 LDA $C100
C003 STA $C200
C006 LDA $C101
C009 STA $C201
C00C BRK
```

The next example uses absolute indexed addressing and transfers 255 ($FF) bytes. The BNE instruction will cause the program to loop back to the address specified until the INY instruction increments the value in Y from $FF to $00, hence setting the carry flag.

```
C000 LDY #$00
C002 LDA $C100,Y
C005 STA $C200,Y
C008 INY
C009 BNE $C002
C00B BRK
```

A similar program may be written using the X rather than the Y register as an index. This is left as an exercise to the reader.

The next program does the same as the last, but uses indirect indexed addressing. While this technique is rather more complex than absolute indexed addressing, we shall see later in this chapter that it is very powerful when large blocks of data have to be transferred.

```
C000 LDA #$C1       ;Set base address
C002 STA $03        ;$C100 in $02 and
C004 LDA #$C2       ;$03, and $C200
C006 STA #$FC       ;in $FB and $FC.
C008 LDA #$00       ;
C00A STA $02        ;
C00C STA $FB        ;
C00E LDY #$00       ;Set pointer.
C010 LDA ($02),Y    ;Transfer
C012 STA ($FB),Y    ;data.
C014 INY            ;Increment pointer.
C015 BNE $C010      ;Loop until pointer zero.
C017
```

THE X AND Y REGISTERS may be stored using absolute, zero page and zero page indexed addressing.

Thus we may have, for example:

```
STX     $2F00
STY     $46A3
STX     $02
STY     $FD
STX     $80,Y
STY     $61,X
```

The STA, STX and STY instructions do not affect any flags.

The value in a RAM memory location may be decremented and indecremented by 1 by the DEC and INC instructions. These instructions may use absolute, zero page, absolute indexed (X register only) and zero page indexed (X register only). Thus we may have, for example:

```
DEC     $7000
INC     $4600
DEC     $61
INC     $FD
DEC     $2C00,X
INC     $36AB,X
DEC     $F0,X
INC     $62,Y
```

The DEC and INC instructions may affect the zero and negative flags but do not affect the carry flag.

The next program shows how indirect addressing is used to transfer a block of memory from $C100 - $C4FF to $0400 - $07FF. As the latter area is screen memory this program enables messages held in memory to be put on screen. Test the program initially by filling $C100 - $C4FF with a code between $41 and $5A before running it. Note that both the program and the data may be saved to disk or tape, as the same file or separately.

The CMP instruction is used in this program to compare the value in a zero page memory location with the value in the accumulator. If these are equal, the zero flag is set. This condition is tested by the BNE instruction.

```
C000   LDA #$17     ;Screen display codes,
C002   STA $D018    ;set 2.
C005   LDA #$00     ;Set up least significant
C007   STA $02      ;bytes of base addresses
C009   STA $FB      ;in $0002 and $00FB.
C00B   LDA #$04     ;Base address $0400 in
C00D   STA $03      ;$0002 and $0003.
C00F   LDA #$C1     ;Base address $C100 in
C011   STA $FC      ;$00FB and $00FC.
C013   LDY #$00     ;Zero in index register.
C015   LDA ($FB),Y  ;Indirect indexed load.
C017   STA ($02),Y  ;Indirect indexed store.
C019   INY          ;Increment Y to point to
                     next address.
```

```
C01A   BNE $C015    ;If non zero Y, loop back to $C015.
C01C   INC $03      ;Increment base addresses
C01E   INC $FC      ;to point to next pages.
C020   LDA #$08     ;If the full transfer
C022   CMP $03      ;has not yet taken place,
C024   BNE $C015    ;loop back to $C015.
C026   LDA #$D8     ;Base address for color map
C028   STA $FC      ;in $00FB and $00FC.
C02A   LDA #$01     ;Foreground color white.
C02C   STA ($FB),Y  ;Indirect index store.
C02E   INY          ;Increment Y to point to
                     next address.
C02F   BNE $C02C    ;If non zero Y, loop back to $C02C.
C031   INC $FC      ;Point to next page.
C033   LDA #$DC     ;If full color map has
C035   CMP $FC      ;not yet been set to
C037   BNE $C02A    ;$01, then loop back to $C02A.
C039   JMP $C039    ;Loop continuously.
```

There are a few points of interest in this program.

Firstly, set 2 of the screen display codes is chosen, because these codes are the same as ASCII codes for upper case letters A through Z. This enables messages to be typed in directly in Interpret Mode (if the monitor has this facility), provided that only these characters and spaces are used.

Secondly the loop to set the character color is separated from the main program loop. Instructions could be saved by incorporating this in the main loop. The reasons for keeping this as a separate part of the program, is that, at a later time, we may wish to loop continuously changing color at each loop, or to assign different colors to each row, each column or each character. Having the loop separate also allows us to use the page zero locations $FB and $FC in both loops. The number of page zero locations available to the programmer, is unfortunately, rather limited in the Commodore 64.

This program loops continuously in the final instruction so as not to spoil the screen display.

DATA MANIPULATION

We have covered instructions which move data around - from register to register and between registers and memory, and

which increment and decrement data bytes.

Normally we wish to do rather more with data, particularly numeric data. We wish to add and subtract and perform logic operations on such data. The 6510 microprocessor has a number of data manipulation instructions.

ADD WITH CARRY

The ADC instruction adds together the contents of a memory location, the contents of the accumulator, and the value (1 or 0) in the carry flag. The result is placed in the accumulator.

Thus if the specified location holds, for example, $2A, the accumulator holds $37 and the carry flag is set (to 1), then the resultant value in the accumulator would be $62 ($2A+$37+1). The contents of the memory location would be unaltered.

Why add the carry flag? This seems to complicate an otherwise simple instruction. To see the rationale behind this we consider a normal decimal addition,

```
    say 684 + 237

carry : 110
-----------
        684
        237
       ----
        921
       ----
```

Think about what is happening here. Every addition of a column includes the carry value from the previous column. The carry value for the first (units) column is of course zero. In fact the normal everyday method of addition is add with carry!

The ADC instruction has the same addressing modes as has the LDA instruction. Thus we may have, for example:

```
ADC #$62      (immediate)
ADC $2000     (absolute)
ADC $FD       (zero page)
```

```
ADC $3C00,X  (absolute indexed,X)
ADC $4800,Y  (absolute indexed,Y)
ADC $6A,X    (zero page indexed)
ADC ($7B),Y  (indirect indexed)
ADC ($02,X)  (indexed indirect)
```

If the ADC instruction results in a zero value in the accumulator then the zero flag is set - otherwise it is reset; if the result lies between $80 and $FF then the negative flag is set, otherwise reset; if the result of the addition is greater than $FF then the carry flag is set - otherwise reset, and if there is a carry from the second most significant to the most significant bit the overflow flag is set, otherwise reset.

This is best illustrated by an example. Suppose the accumulator holds $6E and the carry flag is set, then

```
ADC  #$DC
```

would result in the addition (in binary form):

```
carry:1 1111 1001
      ----------
        0110 1110
        1101 1100
        ---------
        0100 1011
```

Thus the accumulator will hold $4B, the overflow and carry flag will be set and the negative and zero flags reset.

CLEAR CARRY

The setting of the carry flag when the result of adding two bytes together is greater than $FF enables the carry to be included in the addition of the next more significant bytes. This enables large numbers several bytes long to be added together. Sometimes, however, such as when we are starting an addition at the least significant byte, we wish to ensure the carry flag is reset to zero.

The instruction

```
CLC
```

clears the carry flag. This instruction normally precedes an addition routine.

QUIZ 2.1

Write a routine, starting at $C000, which will add together two two-byte hexadecimal numbers starting at $2000 and $2100, and store the results starting at $2200. Numbers should be stored least significant byte first.

Note that three bytes are required to hold the result. End the routine with the BRK instruction.

SUBTRACTION

Until now we have been dealing with simple unsigned numbers, although we have remarked that the flag which reflects the value in the most significant bit of the accumulator is known as the 'negative' flag. Once we come to subtraction, however, things get rather more complicated.

Subtraction may be thought of as the addition of a negative number. Thus

$74 - $3A

may be written

$74 + (-$3A).

We represent a negative number by its 'twos complement'. To calculate this we express the number in a binary form, complement each bit and add 1.

Thus to find - $3A

$3A = 0011 1010

complement 1100 0101

Add 1 1100 0110

-$3A may be represented by $C6.

Let us check this. Taking an eight bit addition we should have $3A+(-$3A)=$00

```
   $3A = 0011 1010
  -$3A   1100 0110
       ----------
     + 0000 0000
```

To return to our original problem, $74 - $3A is equivalent to $74 + $66 which equals $3A.

Check this result by manual hexadecimal or binary arithmetic. Note that a carry into the ninth column occurs both for $74 - $3A and for $3A - $3A. Let us consider $01 - $3A. This is equivalent to $01 + $C6 which equals $C7. This result represents a negative number ($01 - $3A is obviously negative) so that taking its twos complement will check our answer:

```
      $C7   1100 0111
complement  0011 1000
                  +1
            ---------
            0011 1001
```

Thus we have $01 - $3A = -$39, which checks out in standard hexadecimal arithmetic. Note that no carry into the ninth column occurs in this case.

Note - an alternative way of taking twos complement is to subtract the number from $100. This is a faster method for those thoroughly familiar with hexadecimal arithmetic.

When a smaller number was subtracted from a larger number, or when the numbers were equal, a final carry took place. When a larger number was subtracted from a smaller there was no final carry. Now in normal subtraction we borrow from the next column when the number subtracted is larger. This borrow then affects the calculation in the next column. We see here that borrow may be considered the inverse of carry. The situation where we require the borrow was the only situation we did not set the carry.

The microprocessor uses the inverse or complement of the carry flag as a borrow flag in its subtract instruction. The instruction SBC will subtract the contents of a memory location and the complement of the carry flag from the value in the accumulator, placing the resulting value in the accumulator.

Thus if the accumulator holds $6B, the memory location holds $3A and the carry flag is 0 (i.e. borrow = 1), then the SBC instruction will place the value $30 ($6B - $3A - 1) in the accumulator.

If the accumulator holds $7A, the memory location holds $2B and the carry flag is 1 (i.e. borrow = 0), then SBC will place $4F ($7A - $2B - 0) in the accumulator.

Neither of these two examples require a borrow as the smaller number is subtracted from the larger. As they set borrow to zero (no borrow) it follows that they must set the carry flag to 1.

If the accumulator holds $01, the memory location holds $26 and the carry flag is 0 (borrow = 1), then the result of the SBC instruction is that $DA (equivalent to -$26, which equals $01-$26-1) is placed in the accumulator. In this case there will be a borrow as the larger number is subtracted from the smaller. As borrow = 1 the carry flag is set to 0.

The SBC instruction has the same addressing modes as have the ADC and LDA instructions. Thus we may have ; for example:

```
SBC   #$63       (immediate)
SBC   $2630      (absolute)
SBC   $61        (zero page)
SBC   $3000,X    (absolute indexed X)
SBC   $4C00,Y    (absolute indexed Y)
SBC   $55,X      (zero page indexed)
SBC   ($01),Y    (indirect indexed)
SBC   ($60,X)    (inedxed indirect)
```

SET CARRY

In the same way as we wish to clear the carry flag before the start of an addition sequence, we wish to clear the borrow at the start of a subtraction sequence. Clearing the borrow is the same as setting the carry flag. The instruction which sets the carry flag is

```
SEC
```

This instruction should be executed before a subtraction program to ensure that no residual borrows from previous

operations affect the result.

SIGNED NUMBERS

A byte can hold 256 ($00 to $FF) different values. If, however, we define some of the values as negative - that is to say as twos complement values - then we change the range which can be held. If we define $DA to mean -$26 (-38 decimal) for an operation, then we cannot also have it meaning +$DA (218 decimal).

For signed arithmetic we define byte values $01 to $7F as being positive and $80 to $FF as being twos complement negative numbers. Thus if the most significant bit in the accumulator is one (10000000 to 11111111 binary) the number is a twos complement negative. The negative flag in the status register takes the same value (1 or 0) as the most significant bit in the accumulator and hence indicates whether the number in the accumulator, if signed, is positive or negative. Zero is of course unsigned (+0=-0) but could be considered positive as the negative flag is reset when the accumulator holds $00.

Thus the range of decimal values which can be held in a signed byte is -128 to +127.

```
$80 = 1000 0000 = -128
$81 = 1000 0001 = -127
$82 = 1000 0010 = -126

$FE = 1111 1110 = -2
$FF = 1111 1111 = -1
$00 = 0000 0000 =  0
$01 = 0000 0001 = +1

$7E = 0111 1110 = +126
$7F = 0111 1111 = +127
```

QUIZ 2.2

Write a program similar to that for the previous quiz, but this time subtract the number starting at $2000 from that starting at $2100. For the sake of simplicity at this stage assume that both numbers contain only positive bytes and that the number to be subtracted will be the smaller. Store the

answer starting at $2200.

OVERFLOW

We have seen how signed numbers may be defined and used in arithmetic routines. Twos complement allows us to define a positive complement to a negative number and vice-versa. Once we have started an arithmetic routine using signed numbers we must keep to the same convention. Also it is sometimes the case that a number is subtracted from a value which is already negative. It could therefore be the case that we have to perform, with signed numbers, calculations whose decimal equivalents could be (for example)

```
123+8
-83-62
```

Let us follow the conventions we have been using and work out these calculations in binary form, as would the microprocessor

```
123:0111 1011
  8:0000 1000
   ----------
    1000 0011
```

A 1 in the eighth bit of the answer indicates a negative; so take twos complement

```
            1000 0011
complement  0111 1100
                  +1
            ---------
            0111 1101:125
```

Thus we have 123+8=-125!

The other example gives an equally strange result.

```
-83: 1010 1101
-62:+1100 0010
     ----------
     0110 1111  :111
```

Thus -83-62=111!

What has happened here is that the sum of the value bits (0 to 6) has overflowed these bits and affected the sign bit. This happens when the absolute value of the result is greater than 127.

Whenever this happens the microprocessor informs us by setting the overflow flag (V) to 1. The condition for this flag to be set is that the sign of the result is different from the sign of the two numbers being (algebraicly) added. Note that overflow cannot occur if these two numbers have opposite signs.

We shall see how to test this flag in the next chapter. The flag may be cleared using the instruction CLV.

DECIMAL ARITHMETIC

The 6510 microprocessor can perform ADC and SBC operations on decimal numbers. Such numbers are held in memory such that each nybble (4 bits) contain a number from zero to nine.

Thus a byte could contain a value between

```
      0000 0000     (0)
 and  1001 1001     (99)decimal.
```

A two byte (16 bit) number in this mode has a value between 0 and 9999.

This form of notation is known as Binary Coded Decimal or BCD. The microprocessor will carry out decimal arithmetic if the decimal flag (Bit 3 of the status register) is set, and will perform normal binary arithmetic if this flag is reset. The decimal flag is controlled by the instructions:

```
SED  -  Set Decimal Flag
CLD  -  Clear Decimal Flag
```

The decimal flag is reset on power on.

QUIZ 2.3

Adapt the program written for Quizzes 2.1 and 2.2 to work in decimal mode. This time store the numbers most significant byte first (i.e. in the way humans, not computers would read

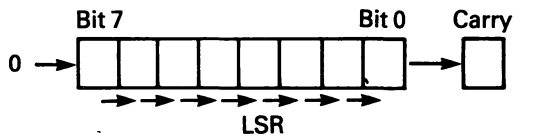
them).

ROTATE AND SHIFT

The 6510 microprocessor can rotate or shift the accumulator or a memory location left or right through the carry flag. This gives access to the contents one bit at a time.

LOGICAL SHIFT RIGHT

The LSR instruction shifts the accumulator or selected memory location to the right. The least significant bit goes into the carry flag and zero is shifted into the most significant bit.

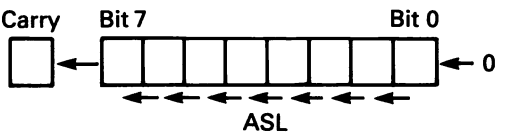


The LSR instruction may operate on the accumulator, or upon a memory location identified by absolute, zero page, absolute indexed X and zero page indexed addressing. Thus we may have, for example

```
LSR  A        (accumulator)
LSR  $4000    (absolute)
LSR  $FD      (zero page)
LSR  $6400,X  (absolute indexed X)
LSR  $50,X    (zero page indexed)
```

ARITHMETIC SHIFT LEFT

The ASL instruction shifts the accumulator or selected memory location to the left. The most significant bit moves into the carry flag and zero is shifted into the least significant bit.

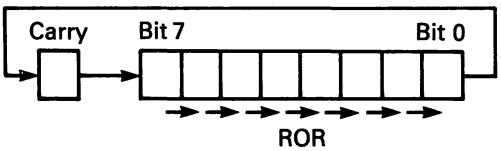


The ASL instuction operates on the accumulator or a memory location identified in the same way as for the LSR instruction. Thus we may have, for example

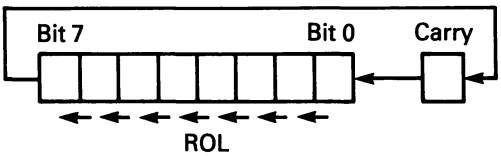
```
ASL   A         (accumulator)
ASL   $2000     (absolute)
ASL   $61       (zero page)
ASL   $3C00,X   (absolute indexed X)
ASL   $43,X     (zero page indexed)
```

ROTATE RIGHT AND LEFT

The ROR instruction rotates the accumulator or a selected memory location to the right. The least significant bit moves into the carry flag and the carry flag value moves into the most significant bit.



The ROL instruction rotates the accumulator or a memory location through the carry flag in the opposite direction to ROR.



Both ROR and ROL have the same addressing modes as the LSR and ASL instructions. Thus we may have, for example:

```
ROR A
ROL A
ROR $3FC0
ROL $2CCF
```

```
ROR $FD
ROL $FE
ROR $1FC0,X
ROL $5CDE,X
ROR $02,X
ROL $61,X
```

The LSR, ASL, ROR and ROL instructions all affect the carry flag. When the accumulator is used the negative and zero flags may be affected.

The next program demonstrates how the ROL instruction may be used to perform an operation on each bit of a memory location in turn. It displays the contents of memory location $0002 in binary on the screen. The program uses the fact that the display code for 1 is one more than the display code for zero. Thus each bit in turn is shifted into the carry flag where it can be added using the ADC instruction to the display code $30.

Ensure your computer is in upper case mode before executing the program. The routine uses the instruction CPY #$08. This sets the zero flag if the contents of the Y register are $08.

```
C000   LDY #$00    ;Reset memory index pointer.
C002   LDA #$08    ;Set base addresses to
C004   STA $FB     ;point to an area
C006   STA $FD     ;in screen control
C008   LDA #$06    ;memory and the
C00A   STA #FC     ;corresponding color
C00C   LDA #$DA    ;control memory.
C00E   STA $FE     ;
C010   LDA #$20    ;Display code for space
C012   STA ($FB),Y;into screen memory.
C014   LDA #$01    ;Color code for white
C016   STA ($FD),Y;into color memory.
C018   INY         ;Repeat until area in middle
C019   BNE $C010  ;of screen is cleared.
C01B   LDA #$68    ;Change base address to point
C01D   STA $FB     ;to middle of cleared area.
C01F   LDA #$30    ;Display code for zero.
C021   ROL $02     ;Put MSB of $0002 into
C023   ADC #$00    ;carry flag and add to display code.
C025   STA ($FB),Y;Display code on screen.
C027   INY         ;Point to next screen location.
C028   CPY #$08    ;Continue until all bits
```

```
C02A   BNE $C01F  ;of $0002 are displayed.
C02C   BRK        ;Return to monitor.
```

This program overwrites the memory location with the value $80. This represents eight carry flag values before the memory rotations. If it is desirable to retain the same value in $0002 at the end of this routine as was present at the beginning then this value should first be stored (in the X register or in another location) before the routine is entered and placed back in $0002 after the routine is completed.

MULTIPLICATION

One use of the ASL operation is in multiplication. Shifting one bit to the left multiplies by two. For example

00101101=45(decimal)

shift left:

01011010=90(decimal)

shift left

10110100=180(decimal)

An overflow will result in the carry flag being set.

To multiply by five, for example, we would shift left twice and then add the original number. Two memory locations are required to hold the result in case of an overflow.

The next program multiplies the number in $0002 by five and places the results in $007B and $007C (most significant byte first).

```
LDA  $02   ;Value to be multiplied
STA  $7C   ;in least significant byte.
LDA  #$00  ;Zero in most
STA  $7B   ;significant byte.
ASL  $7C   ;Multiply LSB by 2.
ADC  #$00  ;If there is a carry,
STA  $7B   ;add 1 to MSB.
ASL  $7B   ;Multiply MSB by 2.
ASL  $7C   ;Multiply LSB by 2.
```

```
LDA  $7B   ;If there is a carry,
ADC  #$00  ;add it to
STA  $7B   ;MSB.
LDA  $02   ;Add the original
ADC  $7C   ;number to
STA  $7C   ;the LSB.
LDA  $7B   ;If there is a
ADC  #$00  ;carry, add it
STA  $7B   ;to the MSB.
BRK        ;Return to monitor.
```

LOGICAL OPERATIONS

Logical operations allow us to carry out operations on single bits rather than on bytes. These operations implement the Boolean arithmetic functions AND, OR and XOR (EOR).

AND

The AND function produces a bit value 1 as its output only if both input bits are 1. Otherwise the output is zero.

```
1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0
```

The instruction AND performs a bitwise logical AND between the value in a memory location and the value in the accumulator. The result of this operation is placed in the accumulator. Thus if the accumulator holds $6A and the memory location holds $7D, the result will be:

```
 0110 1010
 0111 1101
----------
 0110 1000
```

that is $68 in the accumulator.

The AND instruction is used to 'mask off' a bit or bits - that is set to zero every bit in which we are not interested. Suppose, for example, we are interested only in bit 2 of memory location $D019, then

```
LDA  #$04
```

```
AND  $D019
```

will set the value in the accumulator to zero if bit 2 in $D019 is zero. Otherwise the value in the accumulator will remain at $04.

Checking this out:

```
Accum:  0000 0100
$D019:  1011 0100     (Bit 2 set)
        ---------
        0000 0100
```

```
Accum:  0000 0100
$D019:  1101 1010     (Bit 2 reset)
        ---------
        00000000
```

It may also be used to reset a single bit or bits. For example:

```
AND   #$FE
```

will ensure that bit 0 in the accumulator is reset without affecting any of the other bits.

OR

The OR function produces a bit value 1 as its output if either or both of the input bits are 1.

```
1 OR 1 = 1
1 OR 0 = 1
0 OR 1 = 1
0 OR 0 = 0
```

The instructions ORA performs a bitwise OR between a byte in memory and the contents of the accumulator, placing the result in the accumulator. One use of this instruction is to ensure that one or more bits in a memory location are set to 1 (whatever their value beforehand) without affecting any other bits.

For example

```
LDA  #$D0
ORA  $6000
STA  $6000
```

will set to 1 bits 6 and 7 of memory location $6000, whatever their value was previously. None of the other bits of $6000 will be affected.

EOR

The EOR (exclusive or) function (more commonly written as XOR in logic texts) produces a 1 at its output if either but not both of the input bits are 1.

```
1  EOR  1 = 0
0  EOR  1 = 1
1  EOR  0 = 1
0  EOR  0 = 0
```

The instruction EOR performs a bitwise exclusive OR between a byte in memory and the value held in accumulator, placing the result in the accumulator. One use of this instruction is to toggle a bit - that is to set it to 0 if it is a 1 and vice versa.

For example

```
LDA   $4000
EOR   #$80
STA   $4000
```

will toggle bit 7 of memory location $4000 without affecting any other bits.

NOT

The NOT function complements a binary number. All the 1s become 0s and vice versa.

```
NOT 1 = 0
NOT 0 = 1
```

There is no C-64 instruction which performs a NOT operation directly. However EOR #$FF will complement the value in the

accumulator. For example

```
LDA   #$4C
EOR   #$FF
```

will place the value $B3 (which is the complement of $4C) in the accumulator.

The AND, ORA and EOR instructions have the same addressing modes as the LDA and ADA instructions - that is immediate, absolute, zero page, absolute indexed (X and Y), zero page indexed (X), indirect indexed and indexed indirect.

QUIZ 2.4

Write a routine which negates (twos complements) the value held in memory location $62C0. End the routine with a BRK instruction.

BRK

We have already used the BRK instruction at the end of routines to force a return of control to the monitor. We shall discuss this instruction further when we look at interrupts in the next chapter.

NOP

This is the easiest instruction to understand - it does nothing whatsoever! The instruction is often used in delay routines when we wish the microprocessor to wait for a specified time before carrying out the next operation.

SUGGESTED SOLUTIONS

2.1.
```
C000  CLC
C001  LDA  $2000 ;Add least
C004  ADC  $2100 ;significant bytes
C007  STA  $2200 ;and store.
C00A  LDA  $2001 ;Add most significant
C00D  ADC  $2101 ;bytes and carry
```

```
     C010  STA  $2201 ;flag and store.
     C013  LDA  #$00  ;Add carry flag to
     C015  ADC  #$00  ;zero value in acc.
     C017  STA  $2202 ;Store in MSB of answer.
     C01A  BRK        ;Return to monitor.
```

```
2.2. C000  SEC
     C001  LDA  $2100 ;Subtract least
     C004  SBC  $2000 ;significant bytes
     C007  STA  $2200 ;and store.
     C00A  LDA  $2101 ;Subtract most significant
     C00D  SBC  $2001 ;bytes with borrow
     C010  STA  $2201 ;and store.
     C013  BRK        ;Return to monitor.
```

Note - this program could result in a negative value in the least significant byte of the answer.

```
2.3a.C000  SED
     C001  SEC
     C002  LDA  $2001
     C005  ADC  $2101
     C008  STA  $2202
     C00B  LDA  $2000
     C00E  ADC  $2100
     C011  STA  $2201
     C014  LDA  #$00
     C016  ADC  #$00
     C018  STA  $2200
     C01B  CLD
     C01C  BRK
```

For comments see suggested solution 2.1.

```
2.3b.C000  SED
     C001  SEC
     C002  LDA  $2101
     C005  SBC  $2001
     C008  STA  $2201
     C00B  LDA  $2100
     C00E  SBC  $2000
     C011  STA  $2200
     C014  CLD
     C015  BRK
```

For comments see suggested solution 2.2.

```
2.4. LDA  #$FF  ;Complement of $62C0
     EOR  $62C0 ;contents in acc.
     CLC
     ADC  #$01  ;Add 1 to get twos
     STA  $62C0 ;complement, then store.
     BRK
```

SUMMARY

Instructions LDA, LDX, LDY, STA, STX, STY, TAX, TAY, TYA, TXA move bytes of data around the microprocessor system.

Instructions ADC, SBC, INX, INY, DEX, DEY, INC, DEC alter the value of bytes held within the microprocessor registers or memory.

Arithmetic operations may use unsigned, signed or decimal (BCD) numbers. The Carry, Overflow and Decimal flags may affect or be affected by such operations. These flags may be controlled directly by the SEC, CLC, CLV, SED and CLD instructions.

The contents of the accumulator or a memory location may be rotated or shifted one bit using the ASL, LSR, ROL and ROR instructions. All these instructions shift the contents of either the most significant or least significant bit into the carry flag.

Instructions AND, ORA, and EOR perform bitwise logical operations using the value in the accumulator and the value in a specified memory location.

Data movement and manipulation operations may affect the negative and zero flags. These flags are set and reset depending upon the results of such operations. There are no instructions which control them directly.

BRK returns control to the monitor. NOP does nothing.

CHAPTER 3

Program Control

INTRODUCTION

This chapter discusses how decisions may be made in 6510 assembly language routines by conditional branch instructions, and how program loops can be implemented using both conditional branches and unconditional jumps. The use of subroutines is investigated, as are other stack operations. We shall also see how interrupts are used in the Commodore 64, and how the KERNAL jump table gives access to operating system routines.

The following instructions are introduced.

BCC Branch on Carry Clear
BCS Branch on Carry Set
BEQ Branch on Result Zero
BIT Test Bits in Memory With Accumulator
BMI Branch on Result Minus
BNE Branch on Result not Zero
BPL Branch on Result Plus
BVC Branch on Overflow Clear
BVS Branch on Overflow Set
CLI Clear Interrupt Disable flag
CMP Compare Memory and Accumulator
CPX Compare Memory and Index X
CPY Compare Memory and Index Y
JMP Jump to New Location
JSR Jump to New Location Saving Return Address
PHA Push Accumulator on Stack
PHP Push Processor Status on Stack
PLA Pull Accumulator from Stack
PLP Pull Processor Status from Stack
RTI Return from Interrupt
RTS Return from Subroutine
SEI Set Interrupt Disable Status
TSX Transfer Stack Pointer to Index X
TXS Transfer Index X to Stack Pointer

MEMORY JUMPS

The JMP instruction directs the program to a memory location,

whereupon it will start executing the code it finds there. This is somewhat similar to GOTO in BASIC.

JMP has two addressing modes, absolute and indirect. Absolute is very simple:

```
JMP $3060
```

causes the program control to be switched to the instruction starting at location $3060. This is accomplished by the microprocessor putting $3060 into the program counter.

Indirect addressing (unlike the indirect addressing we have seen used previously with, for example, the LDA instruction) is neither zero page nor indexed. Thus:

```
JMP ($2FCD)
```

will cause the program to jump to the memory location whose least significant byte is held in $2FCD and whose most significant byte is held in $2FCE. These two bytes are placed in the (16 bit) program counter.

SUBROUTINE JUMPS

The JSR instruction directs the program to a specified memory location in the same way as JMP. This instruction also stores the address of the memory location directly following its own final location in an area of memory known as the stack. This address is known as the return address. Thus JSR is a subroutine jump (similar to GOSUB in BASIC). At the end of the subroutine called by JSR the RTS instruction (similar to RETURN in BASIC) will take the return address from the stack and direct program control to that address.

The JSR instruction uses absolute addressing. RTS is an implied address instruction.

The next program is the multiplication program in the previous chapter amended to use subroutines. Note that subroutines can call other subroutines, as in BASIC, so that we can have subroutine 'nesting'.

```
C500 LDA $02      ;INITIAL VALUE
C502 STA $7C      ;IN LSB.
C504 LDA #$00     ;ZERO IN
C506 STA $7B      ;MSB.
C508 JSR $C600    ;MULTIPLY BY TWO.
C50B JSR $C600    ;BY TWO AGAIN.
C50E JSR $C700    ;ADD ORIGINAL VALUE.
C511 BRK
```

Multiply subroutine

```
C600 ASL $7B      ;MULTIPLY
C602 ASL $7C      ;BY TWO.
C604 JSR $C800    ;ADD CARRY TO MSB.
C607 RTS
```

Add Subroutine

```
C700 LDA $02      ;ADD ORIGINAL
C702 ADC $7C      ;NUMBER
C704 STA $7C      ;TO LSB.
C706 JSR $C800    ;ADD CARRY TO MSB.
C709 RTS
```

Carry Subroutine

```
C800 LDA $7B      ;ADD
C802 ADC #$00     ;CARRY TO
C804 STA $7B      ;MSB.
C806 RTS
```

Use of subroutines can make programming more flexible. We can amend the main program in the last example to:

```
C500 LDA $02
C502 STA $7C
C504 LDA #$00
C506 STA $7B
C508 JSR $C600
```

```
C50B JSR $C600
C50E JSR $C600
C511 JSR $C600
C514 JSR $C600
C517 JSR $C700
C51A JSR $C700
C51D JSR $C700
C520 BRK
```

The subroutines remain the same. This now gives multiplication by thirty five.

We mentioned RTS briefly in chapter 1. This instruction at the end of a routine called by the BASIC instruction SYS returns control to the BASIC program (or BASIC direct mode if SYS is a direct command).

Thus if we are working with machine code and BASIC we have a choice any time we call a subroutine. If we call the subroutine in the normal way with JSR then program control will return to the machine code routine. If we enter the subroutine with the JMP instruction then the RTS instruction will return program control to BASIC.

MAKING DECISIONS

The main function of flags in a microprocessor system is to enable decisions to be made. The conditional branch instructions will cause control to branch to another part of the program if a specified flag condition is met and continue with the next instruction if it is not.

The branch instructions are:

```
BNE - Branch if zero flag = 0
BEQ - Branch if zero flag = 1
BCC - Branch if carry flag = 0
BCS - Branch if carry flag = 1
BPL - Branch if negative flag = 0
BMI - Branch if negative flag = 1
BVC - Branch if overflow flag = 0
BVS - Branch if overflow flag = 1
```

Thus for example

```
C000 DEY
```

```
C001 BNE $C000
C003 BRK
```

will decrement the Y register until it holds $00, then break to the monitor.

The next program counts the number of memory locations between $2060 and $20FF which do not hold the code $00. This number is placed in the X register.

```
C000 LDX #$00      ;RESET COUNT.
C002 LDY #$60      ;SET MEMORY POINTER.
C004 LDA $2000,Y   ;LOAD ACC FROM $2000+POINTER.
C007 BEQ $C00A     ;SKIP COUNT IF CONTENTS ZERO.
C009 INX           ;INCREMENT POINTER.
C00A BNE $C004     ;REPEAT UNTIL POINTER ZERO.
C00C BRK
```

This program can be amended by altering the instruction starting at location $C007.

```
$C007 BNE $C00A - count locations which hold code $00.
$C007 BMI $C00A - count locations which hold a non
                  negative number.
$C007 BPL $C00A - count locations which hold a negative
                  number.
```

The next program uses the condition of the overflow flag to correct for overflow when adding together two signed numbers held in memory locations $2000 and $2001. The result is stored, most significant byte first, in memory locations $2010 and $2011.

```
4000 CLC
4001 LDA $2000 ; ADD THE
4004 ADC $2001 ; NUMBERS.
4007 BVC $400C ; IF NO OVERFLOW SKIP CORRECTION.
4009 SEC       ; IF OVERFLOW, CORRECT VALUE
400A SBC #$80  ; AND SIGN. ALSO SET CARRY.
400C STA $2011 ; STORE LSB OF ANSWER.
400F LDA #$00  ; STORE CARRY
4011 ADC #$00  ; VALUE IN MSB
4013 STA $2010 ; OF ANSWER.
4016 BRK
```

We shall look more closely at the BCS and the BCC instructions later in this chapter, after we have discussed CMP.

RELATIVE ADDRESSING

All branching instructions use a form of addressing known as relative addressing. Consider a full disassembly of the last program but one:

```
C000 A2 00     LDX #$00
C002 A0 60     LDY #$60
C004 B9 00 20  LDA $2000,Y
C007 F0 01     BEQ $C00A
C009 E8        INX
C00A C8        INY
C00B D0 F7     BNE $C004
C00D 00        BRK
```

Both of the branch instructions are two byte. As one byte is required for the op-code it follows that the address is specified in a single byte. This is accompleshed by specifying the branch address relative to the instruction address. The instruction, rather than holding an absolute branch address, instructs the program to branch backwards or forwards by a specified number of locations if the branch condition is met.

Thus the addressing byte may be considered as a displacement or difference. It can be defined as the displacement between where program control goes if the branch condition is not satisfied, and where program control goes if the condition is satisfied.

Take the instruction at $C008, BEQ $C00A. If the branch condition (zero flag = 1) is not satisfied program control goes to location $C009.

If the branch condition is satisfied program control skips location $C009 and goes to location $C00A - a branch forward of one memory location. Thus the address value of the branch

instruction in $C008 (the displacement byte) is $01.

Now consider the instruction at $C00B and $C00C, BNE $C004. If the branch condition is not satisfied program control goes to location $C00D.

If the branch condition is satisfied program control goes to location $C004, a displacement of minus nine ($C004 - $C00D) locations or a backward displacement of nine locations from where it would otherwise have gone.

Thus the value in the displacement byte is minus nine, or $F7 in signed number notation.

Looking at things from the hardware point of view, the operand of the branch instruction is added to the program counter after the P.C. has been incremented to point to the location following the branch instruction.

RELOCATABLE CODE

The most obvious advantage of relative addressing is that branch instructions take up only two memory locations. They are therefore faster to execute and more economical of memory space than jump instructions.

There is, however, another major advantage. Because the branching is relative a machine code routine which uses branch rather than jump instructions may be relocated to any suitable area in RAM without any alteration in the object code. Code which has this facility is called relocatable code.

Relative addressing is restricted in that branch instructions have a range of 128 locations backwards or 127 locations forwards. If it is necessary to branch to an instruction outside this range the branch must be to a jump (JMP, JSR or RTS) instruction. In this case the branch is usually to the exit point of the routine. As with BASIC, machine code routines should have only one exit point and a single routine should not be too large.

RELATIVE BRANCH TABLES

If you are using a monitor such as MON64 then the value of the displacement byte will be automatically calculated when

assembling, and the branch address when disassembling.

If, however, you are using a minimum facility monitor you will have to work this out. The relative branch tables below should assist in this.

BACKWARD RELATIVE BRANCH TABLE

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	47	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

FORWARD RELATIVE BRANCH TABLE

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

To use the tables when assembling, count the number of locations backwards or forwards between the memory location following the branch instruction and the branch address. Locate this number in the (decimal) body of the appropriate

table and read of the (hexadecimal) value of the displacement byte. The opposite procedure is used to find the backward or forward jump when disassembling.

COMPARE

The CMP instruction is used to compare accumulator contents with the contents of a memory location. It does not alter either of these. CMP can be thought of as a 'dummy subtract'. Its effect is to set whatever flags would have been set if the memory location contents were subtracted from the accumulator (ignoring the borrow).

If the value in the memory location is the same as that in the accumulator the notional result of the subtraction would have been zero. In this case the zero flag is set - otherwise it is reset.

If the value in the memory location is larger than that in the accumulator the subtraction would have required a borrow. In this case the carry flag would have been reset (borrow = not carry). Otherwise this flag is set.

Thus we have four possibilities:

1. CMP....
   BEQ....
   Branch if value in accumulator equals value in memory.

2. CMP....
   BNE....
   Branch if value in accumulator is not equal to value in memory.

3. CMP....
   BCC....
   Branch if value in accumulator is less than value in memory.

4. CMP
   BCS
   Branch if value in accumulator is greater than or equal to value in memory.

To test for value in accumulator is less than or equal to value in memory and value in accumulator is greater than value in memory we require BEQ followed by BCC and BEQ followed by BCS respectively.

CPX and CPY work in the same way as CMP except that the value in the specified memory location is compared with the value in the X and Y registers respectively. These instructions have only three addressing modes - immediate, absolute and zero page.

QUIZ 3.1

Write a program to put the Commodore 64 into bit map mode, clear the high resolution screen and set black foreground and white background colors.

Hints: Set bit map mode by setting bit 3 of $D018 to 1. Locate bit mapped screen at $2000 to $3F3F by setting bit 5 of $D011 to 1.

Clear screen by loading $00 into all bit mapped screen memory. Set foreground and background colors by loading all video memory ($0400 to $07E8) with $01.

BIT

The BIT instruction allows us to test the two most significant bits of a memory location without altering the data in that location or any other data registers. This instruction 'echoes' the value in the most significant bit (bit 7) in the negative flag and the second most significant bit (bit 6) in the overflow flag. The BIT instruction will also set the zero flag if the number in the memory location is the direct complement (not the twos complement) of the value in the accumulator. Otherwise this flag is reset.

BIT has two addressing modes, absolute and zero page.

THE STACK

The stack consists of a page (256 bytes) of memory between $0100 and $01FF. This is known as page 1 of memory. There is nothing special about stack memory itself; it is just

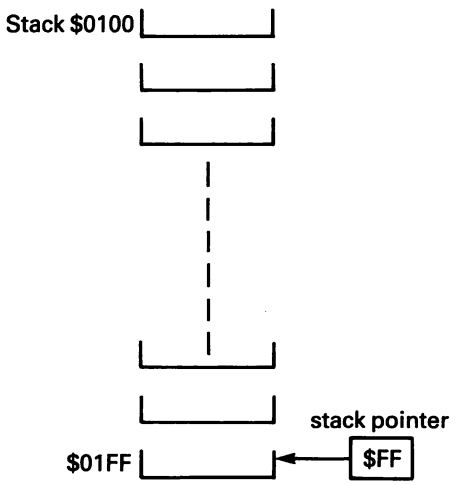
ordinary RAM. What makes the stack different is that additional address decoding circuitry in the microprocessor allows this page of RAM to be addressed using the value in the stack pointer register in addition to all the normal memory addressing modes.

The stack pointer always points to (holds the final byte of the address of) the first memory location in the stack which is available for storing information. When the stack is used and data are stored in that location the stack pointer decrements to point to the next location. If data are read from the stack the stack pointer increments to point to the location from which data has been read, and which is now a free location.

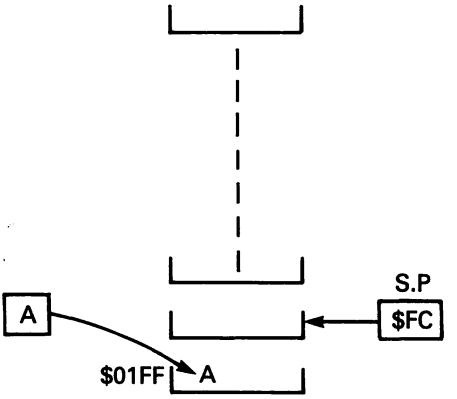
When the stack is empty the stack pointer holds $FF. When the stack is fully loaded the stack pointer holds $00.

To see how the stack works, suppose we were to store and then retrieve four data items (bytes) which we shall designate A,B,C and D.

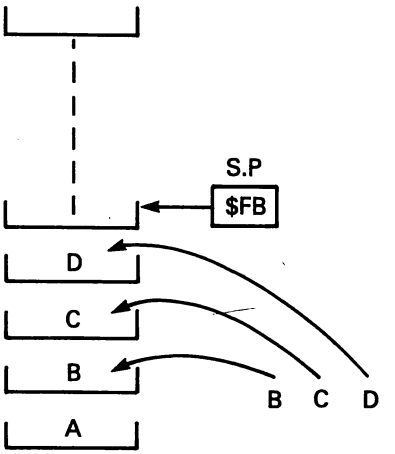
Suppose the stack is empty initially and the stack pointer holds $FF, pointing to memory location $01FF.



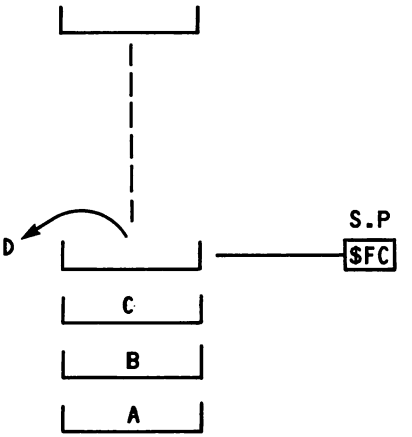
Load in data item A. The stack pointer decrements to $FE.



Data items B,C and D are loaded in that order:



Now an item is read out of the stack. This item will be the item 'below' the first free location. That is item D. The stack pointer increments.



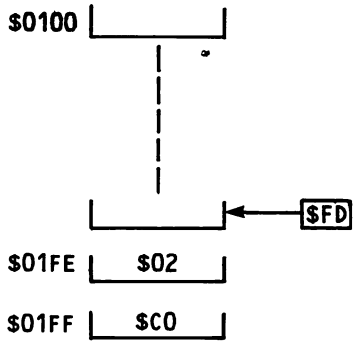
This process is repeated so that the other data items are read off in the order C,B,A. The stack is then empty and the stack pointer holds $FF.

Thus the stack is what is known as a Last In - First Out (LIFO) memory. We have seen one use of the stack - storing subroutine return addresses. We shall now look at this in detail.

Consider the simple program:

```
C000  JSR  C010
C003  BRK
----------
C010  RTS
```

Assume the stack is empty - the stack pointer holds $FF. When the program is executed the first thing that happens is that the first instruction is fetched from memory so that it can be decoded and acted upon. This is a three byte instruction located in $C000 - $C002. Thus when the instruction is executed the program counter holds $C002. The execution of the instruction causes this number to be stored in stack.



The jump address in the JSR instruction is then loaded into the program counter resulting in program control going to location $C010. The single byte instruction code for RTS is fetched and decoded.

On execution this instruction retrieves the address data from the stack, least significant byte first, increments this number, and stores it in the program counter. Thus the program counter holds $C003 and program control goes to that location.

OTHER STACK OPERATIONS

Often when we 'call' a subroutine we find that it uses and hence corrupts the data in the accumulator and index registers, and alters the status register. In this case we wish to store all this data so that we can retrieve it later and restore the registers to their original conditions. The stack offers us a quick and easy way to accomplish this.

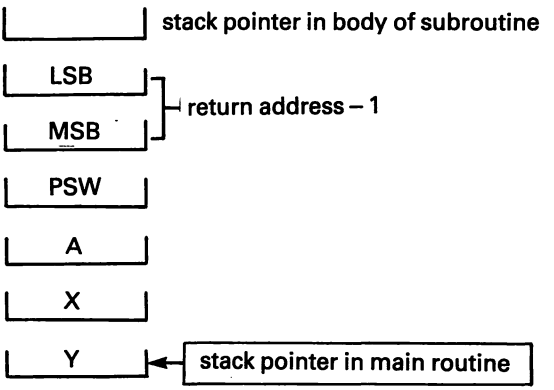
The instructions PHP and PHA store the status register and accumulator contents respectively on to the stack (push instructions). The instructions PLP and PLA load the status register and accumulator respectively from the stack (pull instructions).

We can save and retrieve the registers by pushing them on to and pulling them back off the stack either within the subroutine or in the main program.

a) Within the subroutine

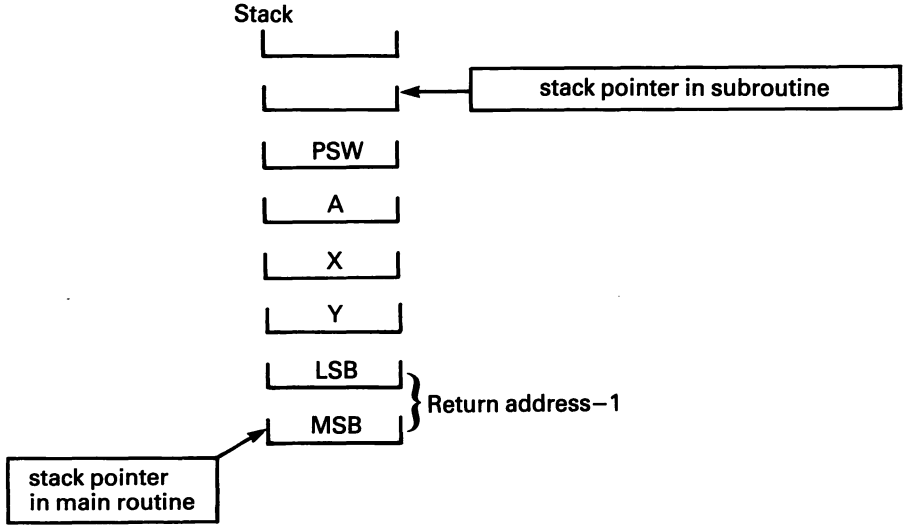
```
        Main routine
              .
              .
              .
           JSR label
              .
              .
              .

        Subroutine

Label      PHP
           PHA
           TXA
           PHA
           TYA
           PHA
             .
             .
             .
           PLA
           TAY
           PLA
           TAX
           PLA
           PLH
           RTS
```

Stack



b) Within the main routine

```
Main routine
      .
      .
      .
     PHP
     PHA
     TXA
     PHA
     TYA
     PHA
     JSR  label
     PLA
     TAY
     PLA
     TAX
     PLA
     PLP
      .
      .
      .

Subroutine

label ____
      ____
       RTS
```



In the best of all possible worlds all subroutines would store and retrieve the values in the registers they use. We could then make all subroutine calls in the confidence that our data would not be corrupted. Sometimes, however, we shall be using subroutines written by others, or subroutines which pass information to the main routine by the values in the A, X and Y registers or the condition of the flags. Subroutines should always be documented, and on the documentation should be listed the registers corrupted by the subroutine, so that users can save their important data.

The PHP and PLA instructions can be used to transfer the flag register contents to the accumulator. Such transfers are done via the stack. PHA and PLP can transfer the value in the accumulator into the status register.

RELOCATING STACK

It is often useful to have stack memory available for use within a subroutine. If, however, we have our working area right next to where we have stored our return address and register information we have to take great care not to overwrite this information.

In this case it is usually easiest to charge the value in the stack pointer and point to a new area of stack 'out of range' of data.

The stack pointer may be loaded via the X register with the TXS instruction.

Before doing this, however, we require to save the current value of the stack pointer. Otherwise we will never be able to retrieve what is currently on stack. The current stack pointer value can be read via the X register with the TSX instruction.

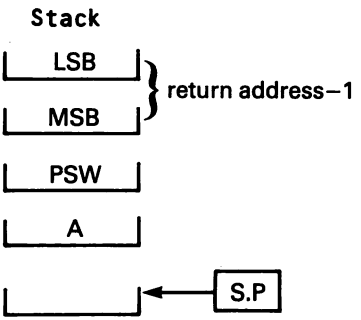
Thus
```
TSX
TXA
LDX #$55
TXS
PHA
```

will set the stack pointer to $55 so that memory location $0155 will be the first stack location used. The first thing

which is pushed on to the new stack is the value which was in the original stack pointer, which will be retrieved when we wish to use the original area of stack.

The stack is easy to use provided care is taken that a return address is in the first two locations whenever an RTS is encountered. Suppose for example we have a subroutine where we saved the status register and accumulator and then omitted to pull them back off the stack :

```
PHP
PHL
 ..
 ..

RTS
```



Then the RTS, instead of pulling the correct return address, will pull the accumulator and flag data off the stack and on to the program counter. This will result in program control flying off into the wide blue yonder, and almost certainly in the program 'hanging'. Always ensure, particularly in subroutines, that your pushes balance your pulls.

The SYS command places on the stack an address which will return control to BASIC routines when placed in the program counter (and incremented) by an RTS instruction. It is important not to 'lose the place' in the stack when working between machine code and BASIC.

WORDTABLES AND JUMP TABLES

Indirect addressing enables a set of addresses to be held as a block in memory. This technique gives us a method of

writing code which can be used by other programs and other programmers, but which we can amend or relocate if necessary.

To do this we fix an area in memory in which we will place the addresses of entry points into our routines. The location of this area in memory remains constant as does the memory locations which hold the start address (start vector) for a particular routine. As long as the addresses of the memory locations which held the vector remain unchanged we can relocate the start address and change the 'wordtable' accordingly without having to alter any other programs which use the routine.

This is an important concept. Let us illustrate it with an example.

Consider a block of useful routines (utilities) one of which reads the keyboard and places the ASCII value of any key detected in the accumulator.

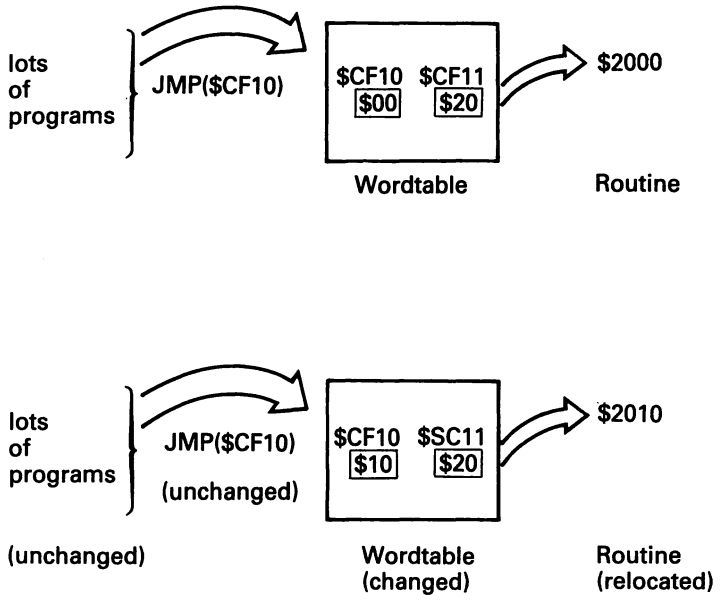
Let us say that this routine starts at $2000, and that the start addresses of all the utility routines are held in a wordtable located at address $CF00 onwards. Both the routines and the wordtable are loaded in whenever they are used. Suppose $CF10 is defined as the vector address for the routine described. This means that location $CF10 holds the value $00 and location $CF11 holds the value $20.

Thus to call up the routine and scan the keyboard we use

```
JMP ($CF10)
```

This results indirectly in a jump to $2000.

Suppose, say, that we generate twenty programs which use the routine. Then, for some reason, our utilities have to be amended. The keyboard scan routine now starts at location $2010. The corresponding wordtable is changed such that locations $CF10 and $CF11 hold $10 and $20 respectively. Thus the instruction JMP ($CF10) will do exactly what it did before - call up the routine to scan the keyboard. As a result we do not have to change any of the twenty programs which use the utilities.



Using a Wordtable

A jump table holds jump instructions rather than address words. Thus if the example given the vector at $CF10 would initially be implemented by the instruction JMP $2000 held in locations $CF10, $CF11 and $CF12. As before if the routine is relocated the jump table is altered. Jump tables are called up by direct rather than indirect jumps. The advantage of jump tables is that they may be accessed from BASIC using the SYS command, or from machine code using JSR.

The BASIC and KERNAL ROMs contain a large number of machine code routines which make up the machine's operating system. Many of these routines may be accessed by the machine code programmer and jump tables and wordtables are provided. For example, ROM memory locations $FF81 - $FFF5 contains a jump table used by the KERNAL routines; (see later in this chapter) $A000 - $A09E hold wordtables for ROM control vectors, keyword action vectors, function vectors and operator vectors.

The next diagram shows the KERNAL jump table. Note that some of the instructions refer in turn to a wordtable in $031A to $032D. Wordtables are often loaded into RAM from ROM on power up in microcomputer systems. This provides flexibility in that the wordtables can be changed by the programmer.

Address	Contents	Routine Name
FF81	JMP $FF5B	CINT-initialize screen editor
FF84	JMP $FDA3	IOINIT - initialize input/output
FF87	JMP $FD50	RAMTAS - initialize RAM, allocate screen and tape buffer
FF8A	JMP $FD15	RESTOR - restore default I/O vectors
FF8D	JMP $FD1A	VECTOR - read/set vector I/O
FF90	JMP $FE18	SETMSG - control KERNAL messages
FF93	JMP $EDB9	SECOND - send secondary address after LISTEN
FF96	JMP $EDC7	TKSA - send secondary address after TALK
FF99	JMP $FE25	MENTOP - read /set top of memory
FF9C	JMP $FE34	MENBOT - read /set bottom of memory
FF9F	JMP $EA87	SCNKEY - scan keyboard
FFA2	JMP $FE21	SETTNO - set time out on serial bus
FFA5	JMP $EE13	ACPTR - input byte from serial port
FFA8	JMP $EDDD	CIOUT - output byte to serial port
FFAB	JMP $EDEF	UNTLK - command serial bus to UNTALK
FFAE	JMP $EDFE	UDTIM - implement real time clock
FFB1	JMP $ED0C	LISTEN - command serial bus devices to LISTEN
FFB4	JMP $ED09	TALK - command serial bus devices to TALK
FFB7	JMP $FE07	READST - read I/O status word
FFBA	JMP $FE00	SETLFS - set logical first and second address
FFBD	JMP $FDF9	SETNAM - set file name
FFC0	JMP ($031A)	OPEN - open a logical file
FFC3	JMP ($031C)	CLOSE - close a specified logical file
FFC6	JMP ($031E)	CHKIN - open channel for input
FFC9	JMP ($0320)	CHKOUT - open channel for output
FFCC	JMP ($0322)	CLRCHN - close input and output channels
FFCF	JMP ($0324)	CHRIN - input character from channel
FFD2	JMP ($0326)	CHROUT - output character from channel
FFD5	JMP $F49E	LOAD - load RAM from a device
FFD8	JMP $F5DD	SAVE - save RAM to device
FFDB	JMP $F6E4	SETTIM - set real time clock
FFDE	JMP $F6DD	RDTIM - read real time clock
FFE1	JMP ($0328)	STOP - scan STOP key
FFE4	JMP ($032A)	GETIN - get character from keyboard buffer
FFE7	JMP ($032C)	CLALL - close all channels and files
FFEA	JMP $F69B	UNLSN - command serial bus to UNLISTEN
FFED	JMP $E505	SCREEN - return XY organization of screen
FFF0	JMP $E50A	PLOT - read/set cursor position
FFF3	JMP $E500	IOBASE - return base address of I/O devices

KERNAL JUMP TABLE

KERNAL PROGRAMMING

The KERNAL is an 8K ROM located at $E000 to $FFFF, which contains the operating system for the C-64. In this ROM there are routines for input/output, memory management, and so on. The KERNAL is accessed via a jump table located at $FF81 to $FFF5. Commodore have indicated that all future upgrades of the C-64 will use the same jump table (given on previous page).

The KERNAL consists of 39 routines which are used as follows:

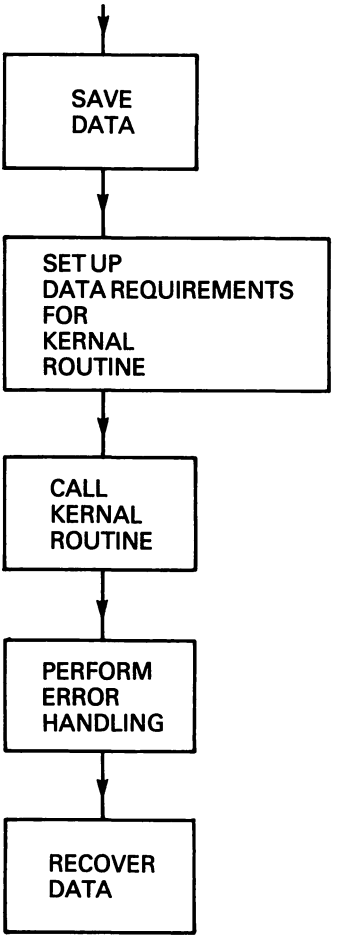
1. Set up parameters which are required by routine.

2. Call the subroutine by using JSR.

3. Use data returned by KERNAL subroutine, some of which are error data.

Of course, it is possible to call the KERNAL routine directly, without using the jump table. If, however, we call the routine directly, we cannot be sure our program will work on any upgrade.

KERNAL routines were written by a programmer using the same processor as the main program. Therefore, KERNAL routines could destroy data which are left in registers. Before calling the KERNAL then, we have to save any data that are required for later use. This can be done by pushing register values onto the stack and pulling them after the KERNAL routine returns control to the main program. Of course enough room must be left on the stack for use by the KERNAL routines themselves. If there is not enough stack space then RAM must be used for saving data. A flowchart for using KERNAL routines would be:



Things are slightly different if we are making calls to routines within the BASIC ROM. These are known as 'illegal' calls (see later in this chapter). The most important factor in any program using these routines is documentation. The program should inform any user that it is probably restricted for use on a C-64 running BASIC V2. Remember that a BASIC V2 routine could destroy register data and gobble up stack - so the programmer should be aware and be careful.

THE KERNAL ROUTINES

The KERNAL routines, documented in Appendix J are available,

to use with confidence. The appendix gives the following information

ROUTINE NAME
DESCRIPTION
DATA REQUIRED
REGISTERS USED
STACK
CALL ADDRESS

ILLEGAL CALLS TO THE OPERATING SYSTEM

In the BASIC and KERNAL ROMs there are 16K of machine code routines performing many of the tasks we would wish our own code to perform. It seems pointless and wasteful to write our own routine to clear the screen or read the keyboard when this has already been done for us. Instead subroutine calls may be used to access operating system subroutines.

Commodore has provided this facility through the KERNAL jump table, giving the guarantee that whatever changes may be made to the operating system in the future, KERNAL calls will always work (in fact VIC 20 KERNAL calls are the same as the Commodore 64 KERNAL calls - some surety that these will not change). Calls to the KERNAL are known as legal operating system calls.

There are, however, many useful routines which are not called by the KERNAL. These routines may be called from machine code or BASIC. Such calls are illegal. This means that if Commodore bring out a new version of the operating system, then programs using illegal calls may not run on the new machines.

To use calls to the operating system we first disassemble the routine in which we are interested to see what it does and which registers it corrupts. Appendix H gives a list of useful ROM addresses. It is important to ensure that the routine ends eventually with RTS. Calling the routine with a JSR instruction results in control returning to the machine code program; calling it with JMP results in control returning to BASIC.

Illegal calls are used at the programmer's own risk. We must ensure that we know exactly what a routine does before using it.

The next program uses both legal and illegal calls to print the alphabet on the screen, with a one second delay between each letter.

```
        JSR $E544     ;CLEAR SCREEN (ILLEGAL).
        LDY #$41      ;ASCII CODE FOR A.
  PRT:  TYA           ;PUT ASCII CODE IN ACC.
        JSR $FFD2     ;PRINT TO SCREEN (LEGAL).
        JSR ASEC      ;ONE SECOND DELAY.
        INY           ;NEXT ASCII CODE.
        CPY #$5B      ;HAS Z BEEN PRINTED?
        BNE PRT       ;IF NOT, PRINT NEXT CHARACTER.
        BRK

 ASEC:  LDA #$00      ;SET MILLISECOND
        STA $FB       ;COUNTER TO
        STA $FC       ;ZERO.
COUNT:  JSR $EEB3     ;DELAY ONE MILLISECOND (ILLEGAL).
        INC $FB       ;THIS LOOP COUNTS
        BNE COUNT     ;256 MILLISECONDS.
        INC $FC       ;THIS LOOP COUNTS
        LDA #$03      ;THE PREVIOUS LOOP
        CMP $FC       ;THREE TIMES, GIVING
        BNE COUNT     ;768 MILLISECONDS.
FINAL:  JSR $EEB3     ;DELAY ONE MILLISECOND.
        INC $FB       ;A FINAL COUNT
        LDA #$E8      ;OF 232 MILLISECONDS
        CMP $FB       ;GIVES ONE
        BNE FINAL     ;SECOND DELAY.
        RTS
```

FLOATING POINT OPERATIONS

So far we have dealt with integer arithmetic - that is using whole numbers. Numbers with figures after the decimal point are much more difficult to deal with. Such numbers are described as 'floating point', and are held in designated areas of page 1 memory known as floating point accumulators. There are two of these in the Commodore 64, FAC#1 at $0061 - $0066 and FAC#2 at $0069 - $006E. These accumulators are used by the floating point arithmetic routines.

PASSING PARAMETERS FROM BASIC

Floating point numbers can be passed to machine code routines

by USR function. For example the command:

```
PRINT USR(62.4)
```

will:

a) Place the value 62.4 in FAC#1.

b) Transfer control to a machine code routine pointed to by the vector held in $0311 and $0312 (785 and 786). This vector may be entered from BASIC using POKE commands. The machine code routine will normally perform floating point arithmetic operations upon the value in FAC#1 and store the result in FAC#1.

c) Print the value stored in FAC#1 by the machine code routine on the screen.

USR can be used in the same way as any other BASIC function. Thus we may have for example:

```
A=6+USR(4.2)
PRINT SIN(USR(60.3))
C=(USR(5.3)+1)/(USR(3.6)-1)
```

The next program allows us to experiment with various USR functions, the machine code for which starts at memory location $C0C0.

```
100 REM USR FUNCTION
110 POKE 785,192
120 POKE 786,192
130 INPUT"PARAMETER";X
140 PRINT USR(X)
150 GOTO 130
```

Try the following examples:

1. Multiply by 10

```
JMP $BAE2
```

2. Absolute value of nearest integer (useful after INPUT statements)

```
JSR $BC58     ;ABS
JSR $B849     ;+.5
```

```
        JMP $BCCC      ;INT
```

3. SIN(TAN(X))

```
        JSR $E2B4      ;TAN
        JMP $E26B      ;SIN
```

PROGRAM STYLE

Program style is important in machine code programming - possibly more than when using high level languages. The difficulty in understanding machine code - especially when hand disassembly is necessary - makes clear documentation and a logical approach essential.

Whenever possible code should be relocatable - apart from anything else this saves having to put memory locations in documentation. This means using branch rather than jump instructions inside a routine. Programs should be kept as short as possible and written as subroutines. Branches in the program are to labels - where possible give these meaningful names. Routines should have only one entry and one exit point.

BASIC programs can be made self documenting due to REM statements. Thus when the program is LISTed the description is there to be read. Machine code does not have this facility. Therefore the documentation produced when the program is being designed (flowchart, pseudocode or whatever) should be of a reasonable standard (no scribbles on scrap paper). A listing of the disassembled routine with labels and remarks written in should be included. Remarks should be sensible and helpful - do not just interpret the mnemonic code. For example:

```
    LDX #$60     ;INITIAL VALUE IN COUNTER.
    LDY #$00     ;POINT TO FIRST LOCATION IN PAGE.
```

is helpful, whereas

```
    LDY #$60     ;LOAD X WITH 60 HEX.
    LDY #$00     ;SET Y TO ZERO.
```

is not. Traditionally the semicolon is used to indicate a remark.

Some monitors let the user assign values to variables or which are then used in the program. This operation is usually denoted by = or EQUI. Unfortunately MON 64 does not allow this. Nevertheless it is useful in program documentation. For example:

```
        FIRST = $41
        LAST  = $5B
        PRNT  = $FFD2
        LDX #FIRST      ;ASCII CODE FOR 'A' IN X.
PRLOOP: TXA             ;PUT CHARACTER CODE INTO
        JSR PRNT        ;ACC AND PRINT TO SCREEN.
        INX             ;GET NEXT CHARACTER CODE
        CPX #LAST       ;HAS 'Z' BEEN PRINTED?
        BNE PRLOOP      ;IF NOT LOOP BACK TO PRINT.
        RTS             ;EXIT TO MAIN PROGRAM.
```

The appropriate numbers for FIRST, LAST etc will be used when assembling the program.

The other information which should be given in any documentation is the name of the routine and if necessary a description of its function. The registers used (corrupted) by the routine must also be listed. Thus for the above example:

ALPHAPRINT

A SUBROUTINE TO PRINT THE ALPHABET ON THE SCREEN.

REGISTERS CORRUPTED A,X,P

would be an appropriate heading.

INTERRUPTS

Interrupt routines are similar to subroutines in that control is switched from the main routine to the interrupt routine, with the return address information being stored on the stack. The ReTurn from Interrupt (RTI) instruction at the end of the interrupt routine returns control to the main program.

While subroutines are called up by software at points specified by the programmer using the JSR instruction, a jump to an interrupt routine may occur anywhere in a program and

is initiated by a hardware signal, usually from a peripheral device. Interrupt routines are used to control peripheral devices which require attention immediately, or to perform a function periodically (such as updating the real time clock) independent of any other programs being executed.

NON MASKABLE INTERRUPT

The non maskable interrupt takes place when the signal on the NMI line goes from high (+5V) to low (0V). This is known as an edge triggered signal. The microprocessor will not detect another NMI until this line has gone high and then low again. NMI must stay low for at least two clock cycles in order to be recognized. This enables execution of the current instruction to be completed before the interrupt occurs.

Non maskable interrupt is used when it is essential that a peripheral be serviced urgently on request - as for example when data is coming in at high speed from an RS232 link. In this case it is essential that the interrupt routine is completed before the next negative going NMI edge is received. Programs written with this sort of time restraint are known as 'real time' programs.

When a non maskable interrupt is received, the program counter and status register are saved to stack and a jump is forced to the location pointed to by the vector address in locations $FFFA and $FFFB. The code starting at this location will first set the interrupt flag using instruction SEI. This ensures the lower priority IRQ interrupt (see below) cannot affect the routine. The program then jumps to a routine pointed to by the vector held in memory locations $0318 and $0319, which are loaded from ROM by an earlier set up program.

It is possible to change the vector in $0318 and $0319 so that it points to a user generated interrupt routine. This might be done if it were necessary to interface the computer with a non standard data transfer device or real time controller. In general this is not advisable until you are thoroughly familiar with the operating system and have had considerable practice in assembly language programming. The MON64 monitor cartridge, for example, generates a non maskable interrupt to implement the 'Walk' facility.

MASKABLE INTERRUPT

A maskable interrupt is generated by a OV signal on the IRQ (Interrupt Request) line. This is a level sensitive rather than an edge triggered signal. The IRQ line must be held low until the interrupt request is recognised.

This interrupt is maskable. While the interrupt disable flag is set then the IRQ interrupt from an external device will be ignored. The instructions SEI and CLI set and clear this flag. The interrupt disable flag is automatically set when an interrupt request is accepted so that further maskable interrupts are disabled. A non maskable interupt can interrupt a routine initiated by a maskable interrupt and is therefore said to have a higher priority.

When the interrupt is recognized the program counter and status register are stored on stack. A jump is forced to the IRQ entry routine pointed to by the vector in $FFFE and $FFFF.

The BRK instruction forces an interrupt similar to IRQ. BRK works whether the interrupt disable flag is set or not. It sets the break flag, stores the program counter and status register on the stack and forces a jump to the IRQ entry routine.

The IRQ entry routine (at $FF48 in the current operating system) pushes the accumulator, X and Y registers on to the stack. It then tests if the break flag was set before the routine was entered. If this flag is set the program branches to the address pointed to by the vector in $0316 and $0317. Otherwise control is switched to the address pointed to by the vector in $0314 and $0315.

In the Commodore 64 maskable interrupts are generated every 1/60th of a second, synchronized by the TOD clock. The interrupt routine will increment the real time clock counter and scan the keyboard, cassette buffer and data port buffers in Complex Interface Adaptor (CIA) chip #1 at $DC00 to $DCFF in the memory map. When these tasks are complete the data registers are restored. The interrupt disable flag is reset and the RTI instruction restores the status register and returns control to the main program.

The vector in memory locations $0314 and $0315 may be altered by the operating system programs, so that there are several

alternative entries to the interrupt handling routine. In the current operating system the values for the various vectors are held in $FD9B - $FDA1 and the subroutine to set up the vector is in $FCB8 - $FCC9. The interrupt exit is in $FEBC - $FEC1. The normal entry to the interrupt handling routine is $EA31.

It is possible to change the vector in $0316 and $0317 and write your own routine for BRK. Rather more useful is the facility for changing the vector in $0316 and $0317 to insert your own interrupt routine. This lets your code perform a function at regular intervals irrespective of your main program. It is advisable to return via the normal IRQ routines which carry out some important regular tasks (house keeping) rather than directly to the main program. Note that the interrupt disable flag must be set while the vectors are being altered.

The next two programs demonstrate the independent action which can be generated using interrupts. The first program generates a bat shaped sprite and places it on the screen. This program also changes the interrupt vector to point to $C040.

Note - Do not attempt to execute this program until the second program has been keyed in.

First program - key in instructions starting at $C000 ; data starting at $3000. All data items are hexadecimal.

Program

```
LDA #$D0      ;PLACE BASE ADDRESS
STA $FC       ;$D000 (START OF VIC CHIP)
LDA #$00      ;IN PAGE ZERO.
STA $FB       ;LOCATIONS $FB AND $FC
LDY #$27      ;POINT TO COLOR CONTROL LOCATION.
STA ($FB),Y   ;SPRITE COLOR BLACK (ZERO).
LDY #$1D      ;POINT TO X-EXPANSION LOCATION.
STA ($FB),Y   ;NO X EXPANSION.
LDY #$17      ;POINT TO Y-EXPANSION LOCATION.
STA ($FB),Y   ;NO Y EXPANSION.
LDA #$01      ;SPECIFY SPRITE IN LEFT
LDY #$10      ;HALF OF SCREEN BY PUTTING
STA ($FB),Y   ;$01 IN MSB-X LOCATION.
LDA #$41      ;SPECIFY THE
LDY #$01      ;INITIAL
```

```
      STA ($FB),Y  ;Y POSITION.
      LDA #$32     ;SPECIFY THE
      DEY          ;INITIAL
      STA ($FB),Y  ;X POSITION.
      LDA #$C0     ;SPECIFY THAT SRITE SHAPE
      STA $07F8    ;DATA STARTS AT LOCATION $3000.
      LDA #$01     ;ENABLE
      LDY #$15     ;SPRITE
      STA ($FB),Y  ;ZERO.
      STA $FD      ;$FD REGISTER HOLDS $01-DOWN MOVE.
      SEI          ;CHANGE
      LDA #$C0     ;INTERRUPT
      STA $0315    ;VECTOR
      LDA #$43     ;TO
      STA $0314    ;$C045.
      CLI          ;ENABLE INTERRUPT.
      RTS          ;RETURN TO BASIC.
```

Data

```
      :3000 00 00 00 00 00 00 08 24
      :3008 10 08 24 10 08 18 10 0C
      :3010 3E 30 0C 7E 30 0C 3E 30
      :3018 0E 18 70 0E 18 70 0E 7E
      :3020 70 0F FF F0 0F FF F0 0F
      :3028 FF 70 0E 7E 70 0C 5A 30
      :3030 08 99 10 09 C3 90 00 00
      :3038 00 00 00 00 00 00 00 00
```

The second program contains the additional code to be executed at each interrupt. This moves the sprite up and down the screen.

Key this in starting at the new interrupt entry address $C045. The label values are MOVUP = $C057, EXIT = $C065.

```
        LDA #$01     ;IS THE BAT
        CMP $FD      ;MOVING DOWNWARDS ?
        BNE MOVUP    ;IF NOT BRANCH TO UP MOVEMENT.
        INC $D001    ;MOVE BAT DOWN.
        LDA $FC      ;IS IT AT THE
        CMP $D001    ;BOTTOM ?
        BNE EXIT     ;IF NOT, GO TO NORMAL IRQ HANDLING.
        STA $FD      ;PUT UP CODE IN $FD.
MOVUP:  DEC $D001    ;MOVE BAT UP.
        LDA #$31     ;IS IT AT THE
        CMP $D001    ;TOP ?
```

```
        BNE EXIT      ;IF NOT GO TO NORMAL IRQ HANDLING.
        LDA #$01      ;PUT DOWN CODE
        STA $FD       ;IN $FD.
EXIT:   JMP $EA31     ;GO TO NORMAL IRQ HANDLING.
```

Return to BASIC and enter SYS 49152. The bat appears and moves up and down one side of the screen. However, we still have full control of BASIC direct mode. We can type in immediate commands, enter and run programs, and even switch back to running the machine code monitor.

The bat will continue to flit serenely up and down the screen independent of all these actions.

The sprite data may be protected from being overwritten by BASIC programs by entering:

```
POKE 48,64:POKE 56,48:CLR
```

Experiment with this program. The bat can be moved horizontally as well as vertically, for example, or several bats can be placed on the screen at the same time.

SUGGESTED SOLUTIONS

3.1

```
        LDA #$08      ;SWITCH ON
        ORA $D018     ;BITMAP
        STA $D018     ;MODE.
        LDA #$20      ;LOCATE BITMAP
        ORA $D011     ;SCREEN AT
        STA $D011     ;$2000-$3F3F.
        LDA #$00      ;SET BASE
        STA $02       ;ADDRESS $2000
        LDA #$20      ;IN PAGE ZERO LOCATIONS
        STA $03       ;$02 AND $03.
        LDY #$00      ;RESET IN-PAGE POINTER.
OUTLP:  LDA #$00      ;THIS LOOP
 INLP:  STA ($02),Y   ;LOADS ONE PAGE
        INY           ;OF HIRES SCREEN
        BNE $INLP     ;MEMORY WITH ZERO.
        INC $03       ;THIS LOOP REPEATS
        LDA #$3F      ;THE PREVIOUS LOOP
        CMP $03       ;FOR EACH PAGE OF SCREEN
        BNE $OUTLP    ;UP TO $3EFF.
```

```
        LDA #$00      ;ZERO VALUE FOR SCREEN.
EXTRA:  STA ($02),Y   ;THIS LOADS THE FINAL
        INY           ;$40 LOCATIONS OF
        CPY #$3F      ;SCREEN MEMORY
        BNE EXTRA     ;WITH 0.
        LDY #$00      ;RESET IN-PAGE POINTER.
        LDA #$04      ;CHANGE BASE ADDRESS
        STA $03       ;TO $0400 FOR VIDEO MATRIX.
BIGLP:  LDA #$01      ;CODE FOR WHITE PAPER, BLACK INK.
LITLP:  STA ($02),Y   ;THIS LOOP FILLS A
        INY           ;PAGE OF VIDEO MATRIX
        BNE LITLP     ;WITH COLOR CODE $01.
        INC $03       ;THIS LOOP REPEATS
        LDA #$07      ;THE PREVIOUS LOOP
        CMP $03       ;UNTIL THE VIDEO MATRIX
        BNE BIGLP     ;IS FILLED UP TO $06FF.
        LDA #$01      ;PAPER/INK CODE.
FINAL:  STA ($02),Y   ;THIS FILLS THE
        INY           ;REMAINDER OF
        CPY #$E8      ;THE VIDEO MATRIX
        BNE FINAL     ;UP TO $07E8.
        RTS
```

SUMMARY

The BCC, BCS, BEQ, BNE, BMI, BVC and BVS instructions allow decisions to be made in a routine depending on the condition of a status flag. The BIT, CMP, CPX and CPY instructions allow flags to be set without altering any data.

The JMP instruction switches program control to a specified location unconditionally. The indirect form of this instruction allows word tables to hold a series of jump addresses. Jump tables can also be used for this purpose. The JSR instruction allows subroutines to be called. RTS returns from a subroutine to the main routine, or from a machine code routine to BASIC. KERNAL routines in operating system ROM (and other routines in BASIC ROM) may be called from user programs.

Interrupt routines are similar to subroutines in that the are accessed from and on completion, return control to a main routine with RTI. They are initiated, however, by means of a hardware signal rather than a program instruction. Maskable interrupts may be disabled by setting the interrupt disable flag with the SEI instruction. CLI clears this flag. The

BRK instruction causes an interrupt and sets the break flag.

Subroutines and interrupt routines use a LIFO register called the stack to hold the value in the program counter when the routine was accessed. This enables control to be switched back to the correct place in the program when the routine is complete. The accumulator and status register may also be stored to or loaded from the stack by the PHA, PHP, PLA and PLP instructions. The first free memory location in stack is pointed to by the stack pointer register. This may be stored in or loaded from the X register by the instructions TSX and TXS.

Parameters may be passed to machine code routines from BASIC by the USR funtion. This function takes the value generated by passing its argument to a machine code routine specified by a memory vector.

CHAPTER 4

Software Design and Implementation

INTRODUCTION

The purpose of this chapter is to discuss programming the Commodore 64 in BASIC V2. It is not an introduction to BASIC. For those with no programming experience we suggest that they consult TIM ONOSKO'S book "Commodore 64: Getting the Most From It", Prentice/Hall International, 1983.

PROGRAM DESIGN

It is obvious that computer programming is a different task from simply coding a computer program. If we have to solve a problem we do not take the specification to our computer and start to code our solution. We have to think first - and code later.

What then is computer programming?

We can define it as the complete set of actions which have to be undertaken in order to solve a problem using a computer.

These actions are:

(a) Analyzing the problem - it may seem that this is an obvious or trivial step in writing a program. But it is all too easy to start an exercise without seeing all its ambiguities or even impossibilities. For example, we may wish to write some real time program only to realize later that our processor's clock is not fast enough to solve our problem.

    No matter what the project or exercise is, a clear understanding of exactly what is required must be attained in order to end up with a solution which works.

(b) Designing the program - this can be considered as the main part of the exercise, this action can be broken down into a series of sub-actions:

    (i) Solution outline - it is necessary to have a broad

idea of the solution to a software problem. Perhaps our problem would be best solved by a series of programs, e.g., a payroll system. Or, perhaps, we have to work in a team - we have to decide who does what. Either way, we have to know how the various parts of our solutions are interrelated.

(ii) Programming style - this is a further refinement on sub-action (i). Here we specify how we are to write, or structure, each part of our solution outline, noting whether algorithms exist or not.

(iii) Representing the details of the solution - this is the kernal of computer programming. If this action is done well, the task thereafter is straightforward. There are various aids in finding the details of the solution, e.g.

- flowcharts
- segmented flowcharts
- decision tables
- pseudo codes

These aids allow us to design the details of our program, but are still independent of the machine and the language used on the machine.

(c) Coding our program - It is only at this point that we implement the design in BASIC V2.

(d) Debugging our code - it is a mistake to hand over our program immediately it is written. We must first of all find and correct all errors. This can take quite a bit of time. But time taken will be minimized if we have a well written program. As well as finding syntax and run time errors, the debugging action includes testing the program - making sure that the results output from the program are valid.

(e) Documentation - documentation should be an ongoing process. If we have taken notes as each action is undertaken, then we will have part of the documentation of our program already written. However, once we have brought our project to a successful conclusion, we have to ensure that our documentation is complete:

- can other programmers build on our work?

- can users perform their tasks from the documentation alone?

There is one final point which has to be made - these various actions do not necessarily occur one after the other. We shall find that we continually go back from one action to a previous one, refining our solution as we go.

We shall now discuss these actions in some detail.

PROBLEM ANALYSIS

During this phase of software development, we should develop a clear specification of the details of the project.

The starting point will be the specification recieved from the customer. If the customer is not a professional systems analyst, this specification will tend to be rather vague, ill-defined, or ambiguous.

For example, a business executive might ask the following type of question:

1. Could we provide an inventory control program?
2. I would like a system for feeding in sales to the computer and getting a sales ledger, invoices, etc out.
3. How would I automate the finance section?

A scientist might ask:

1. I would like a program to break this data down into something manageable.
2. I would like a program to simulate the human respiratory system.
3. Can a program be provided to control my perpetual motion machine?

Nowadays, with so much interest in microprocessors and microcomputers, we could be asked to:

1. Provide a program to print labels for cans.
2. Write a program to produce logos.
3. Construct a software device which will allow access to bank accounts, and allow updates of those

accounts without the bank's knowledge.

If we are experienced in the area, then perhaps that is all the information that we need. But more often we have to tie the project down more precisely.

This will normally be done by working with the project originator, having a dialogue wherein we can outline the capabilities of the computer system and the project originator can make decisions when options are available.

At the end of this dialogue, we should have arrived at a problem specification, in the following form:

1. Input Specifications - this should be a list of all the inputs that may be received by the computer. For example:

   - data blocks from transmission lines
   - data from A/D converters
   - hours worked from a time card
   - results from a questionaire

   For all our inputs we should be able to answer the following questions:

   i. Where is the input coming from?
   ii. How will the processor recognize it? (At assembly level this will involve timing problems etc, at a higher level this will involve format problems).
   iii. Will the processor be able to spot errors (out of valid range, data transmission errors, etc)?
   iv. What does the processor do with the data; can it be altered or discarded?
   v. How is the end of the data recognized - a special signal or symbol, perhaps?

2. Output Specification - the outputs required from the program must also be laid out in some detail. For example, outputs could be:

   - data blocks to transmission lines
   - data to A/D converters
   - payslips
   - opinion poll results

   We should be able to answer the following questions:

i. Where is the output going to?
ii. What is its format?
iii. How does the peripheral device know that it is coming?
iv. How are errors avoided, (transmission errors etc)?
v. How much output is a report; are there any specific annotations, headers, etc?

3. Special Processing Specification - in the special processing section we list any special conditions which are to be considered. We do this from the point of view that we must know what to do if something goes wrong. For example, an LED fails, the wrong tape is mounted, a nasty human gets access to our program, or two numbers are added giving a negative result.

Every variation from the norm should be specified. If it can go wrong, it will.

MONOLITHIC PROGRAMMING

It is possible at this point in program design to sit down and write our program. We would consider the task to be one logical unit, starting at the beginning and finishing at the end.

In this style of programming, we are free to design our work to our own standards. Our only restrictions are those of the language and hardware.

It is a style of programming which allows us to be "clever", and makes programming a difficult but satisfying art. When people are left to their own devices, programs tend to have this monolithic style. And when they crash, it is a terrible crash.

The disadvantages of monolithic programming are:

1. Difficult to construct.
2. Difficult to amend.
3. Expensive to write and maintain.
4. Difficult and expensive to test.
5. Difficult and expensive to document.

The advantages of monolithic programming are:

1. Can be highly efficient at RUN time.
2. Can stretch the hardware to its limits.
3. Can be very satisfying to write.

For examples of monolithic programs see some of the games programs in personal computer magazines.

MODULAR PROGRAMMING

In modular programming, we attempt to break the problem down into a hierarchy of simpler tasks, each of which can be written in a self-contained manner. To break down the tasks, we perform what is known as top down analysis.

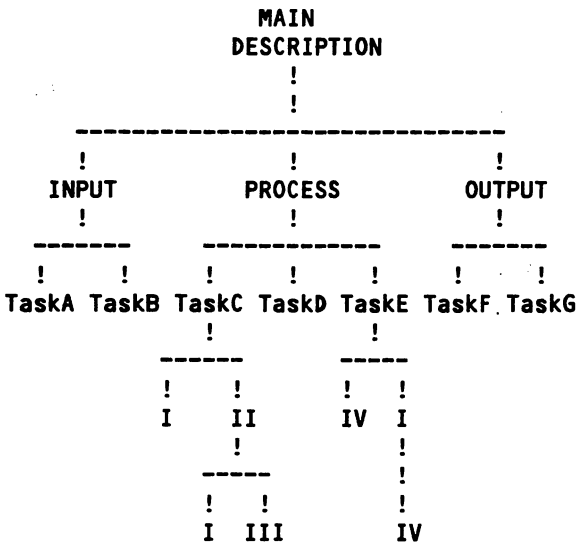
Initially, we describe the problem we are working on at the highest level - this will normally be the problem definition. This will break down naturally to perhaps three lower levels of task - INPUT, PROCESS, OUTPUT.

We then take each of these tasks in turn and break them down into simpler tasks, continuing in this manner until we are in a position to code each module directly.

We can then form a module relationship chart which shows how the different modules of program will call each other. A typical relationship chart is shown overleaf. Notice that it is possible for a module to be reused elsewhere in the problem.

Once we have gone through the process of top down analysis, we can then code each module. At this point we must restrict ourselves as programmers; we must follow the following rules:

1. All modules must have one entry and one exit point only. In particular, jumps or GOTO's can only occur within a module.
2. A module cannot call itself as a lower level module.
3. All modules should be independent of each other and can thus be coded and tested separately.



MODULE RELATIONSHIP CHART

The advantages of modular programming are:

1. Can be written by a team of programmers.
2. Each module can be tested independently.
3. Since each task is simple, the programmer does not need to be highly skilled. (If the top down analysis is done by a chief programmer).
4. Can be maintained easily.
5. Organization can build up library of modules for program construction.

The disadvantages are:

1. We can end up with repeated code.
2. Leads to programmer dissatisfaction - especially if specialization is developed.

STRUCTURED PROGRAMMING (Program Design Language)

Let us consider the problems which arise when trying to understand an unknown programmer's code

We start at the first line of the code with pencil at the ready. Everything can go well, but we hit problems at the first jump instruction, or a call to a subroutine. We can get extremely muddled when these jumps or calls are followed by other jumps, perhaps conditional jumps.

We would then use the pencil and attempt to produce a diagrammatic representation of the code - but this could be extremely hard work.

If, however, the code had been designed from a structured point of view, a fully documented, easily understood, program could have been left for us.

Consider the following sets of code - which is the best? (Assume that the initial value of A is zero in each case).

```
(A) 100 REM UNTIL A=10
    110 :    A=A+1
    120 :    PRINT A
    130 IF A<10 THEN 110: REM ENDDO

(B) 100 IF A<10 THEN A=A+1:PRINT A:GOTO 110

(C) 100 IF NOT(A>10 OR A=10) THEN GOTO 200
    110 A=A+1
    120 PRINT A
    130 GOTO 100
    200 REM REST OF THE PROGRAM

(D) 100 IF A<0 THEN 120
    110 GOTO 150
    120 A=A+1
    130 PRINT A
    140 GOTO 100
    150 REM REST OF THE PROGRAM
```

In the first case we can see easily that the computer has to execute two lines of code UNTIL A=10. Forgetting the details of BASIC V2, we could write this as:

```
DO UNTIL A=10
   A=A+1
   PRINT A
ENDDO
```

Notice that we have indented the set of code which has to be DOne.

In the structured programming approach to software design, we attempt to achieve our solution in a piecewise fashion. In fact, our first description of the eventual software would not be written in BASIC V2, but in a Program Design Language (PDL) or Pseudo Code. The PDL or pseudo code is the means whereby we can show the functions and features of the software in an English-like text.

A PDL exposition of a piece of proposed software breaks it down into segments in a similar fashion to modular programming. These segments could be interpreted as subroutines or subprograms in a BASIC V2 environment. The PDL segment statements could be included in the final program as REM statements. We can standardize the English-like text that we use as follows.

(a) SEGMENT,ENDSEGMENT,INVOKE.

These statements handle the top level structure of the program. For example:

```
VERIFY SEGMENT    )
    :             )
    :             )  logical block
    :             )  of code.
END SEGMENT       )
```

to call the VERIFY segment, we would use the keywords:

```
INVOKE VERIFY
```

(b) DO WHILE.....ENDDO
FOR.....ENDFOR
DO UNTIL.....ENDDO

These statements allow us to perform a set of code cyclicly. For example:

```
DO WHILE A<10
   PRINT A
ENDDO

FOR A=1 TO N DO
  PRINT A
ENDFOR
```

(c) IF...ELSE...ENDIF
SELECT...WHEN...OTHERWISE...ENDSELECT

These statements are used in conditional processing. For example:

```
IF A<10
   PRINT A
ELSE
   A=A+1
ENDIF

SELECT A
  WHEN A=1
       PRINT A
  WHEN A=2
       PRINT 2*A
  OTHERWISE
       PRINT 3*A
ENDSELECT
```

The appearance of PDL code is all important. We must observe the following rules:

1. One statement per line only is allowed.
2. Indentation to show structure must be undertaken. Normally statements are lined up one below another. However statements subordinate to a SEGMENT, DO, IF, SELECT, WHEN, or OTHERWISE clause must be indented.
3. In conditional statements, the ELSE must be placed in line below the corresponding IF (similarly with the SELECT clause).

EXAMPLE OF PDL

Let us construct the PDL statements to allow us to build a program to display the current time at the center of the computer screen. The format of the display is to be

HH:MM:SS, with the clock accurate to one second.

```
INITIALIZE SEGMENT
   SETUP SCREEN AND INITIALIZE GLOBAL VARIABLES
   DO UNTIL HOUR VALID
      WRITE 'ENTER HOUR'
      INPUT HOUR
      VALIDATE HOUR (-1<HOUR<24)
   ENDDO
   DO UNTIL MIN VALID
      WRITE 'ENTER MIN'
      INPUT MIN
      VALIDATE MIN (-1<MIN<60)
   ENDDO
   DO UNTIL SEC VALID
      WRITE 'ENTER SEC'
      INPUT SEC
      VALIDATE SEC (-1<SEC<60)
   ENDDO
   SET INTERNAL TIMER
ENDSEGMENT

DISPLAY SEGMENT
   DO UNTIL STOPPED
      DO WHILE HOUR<24
         READ INTERNAL TIMER
         DISPLAY TIME
      ENDDO
      SET INTERNAL CLOCK TO ZERO
   ENDDO
ENDSEGMENT
```

IMPLEMENTING STRUCTURED PROGRAMMING ON THE COMMODORE 64

Once we have designed the solution to our problem, we have to implement it using the software available - BASIC V2.

We can implement the basic structures as follows:

SIMPLE PROCESS:

Straight forward lines of BASIC code, perhaps called as subroutine using GOSUB...RETURN

IF..THEN..ELSE:

As there is no ELSE in BASIC V2, we have to use GOTO.

```
100 IF A<10 THEN 200
110 REM 'ELSE' CODE
120 :   PRINT A
130 :   A=A+1
140 :   GOTO 300
200 REM 'THEN' CODE
210 :   PRINT A
220 :   A=A-1
300 REM ENDIF
```

Notice that to implement the structure fully, we require two 'GOTO' statements.

Note that if the conditional statement is simple, we can implement it as one line of code:

```
100 IF A<10 THEN PRINT A : A=A+1
```

DO UNTIL LOOP

We implement the DO UNTIL LOOP by using IF THEN and GOTO

```
100 REM DO UNTIL A=10
110 :   PRINT A
120 :   A=A+1
130 :   IF A<10 THEN 110
140 REM ENDDO
```

DO WHILE LOOP

A variation of the DO UNTIL is the DO WHILE, this is implemented by using IF..THEN

```
100 REM DO WHILE A<10
110 :   IF A=10 THEN 200
120 :   PRINT A
130 :   A=A+1
140 :   GOTO 110
150 REM ENDDO
```

SELECT

The SELECT structure is implemented directly in CBM BASIC V2 as the ON..GOSUB construct. However the selecting expression can have integer values only.

```
100 REM SELECT ON A%
110 :   REM WHEN A%=1,2,3
120 :   ON A% GOSUB 1000,2000,3000
130 :   REM OTHERWISE
140 :   IF A%<1 OR A%>3 THEN GOSUB 4000
```

FOR

The FOR stucture is implemented directly in CBM BASIC V2 as the FOR...NEXT loop.

```
100 FOR A=1 TO 5
120 :   PRINT A
130 NEXT A: REM ENDFOR
```

It is, therefore, possible to write well designed and well structured code in BASIC V2.

It is up to the PROGRAMMER to design code, and not up to the LANGUAGE. All the points we have made in writing structured code apply equally well when writing in assembler.

EXAMPLE

Let us implement the clock example in BASIC V2 to run on a Commodore 64.

```
 10 REM DIGITAL CLOCK
 20 REM *************
 30 REM
 80 REM INITIALIZE SEGMENT
 90 :   REM SET UP SCREEN AND GLOBAL VARIABLES
100 :   CL$=CHR$(147):REM TO CLEAR SCREEN
110 :   UP$=CHR$(145):REM UP CURSOR
120 :   S$=CHR$(32)  :REM SPACE CHARACTER
121 :
125 :   REM ROUTINE TO SET UP 27 SPACES
130 :     FOR N=0 TO 2
132 :        S$=S$+S$+S$
133 :     NEXT N
134 :     REM ENDFOR
135 :
138 :   REM COLOR SCREEN
140 :     POKE 53280,0:REM BLACK BORDER
150 :     POKE 53281,0:REM BLACK SCREEN
```

```
160 :     PRINT CHR$(30):REM GREEN INK
170 :   REM PLACE BACKGROUND INFORMATION
180 :     PRINT CL$:PRINT:PRINT
190 :     PRINT TAB(10)"24 HOUR DIGITAL CLOCK"
200 :     PRINT TAB(10)"++++++++++++++++++++"
210 :     PRINT:PRINT
220 :     PRINT TAB(11)"ENTER INITIAL TIME"
230 :     PRINT
231 :   REM DO UNTIL HOUR VALID
240 :     PRINT TAB(11):INPUT"HOURS";H%
250 :     IF H%<0 OR H%>23 THEN
          PRINT UP$+S$+UP$:GOTO 240
255 :   REM ENDDO
256 :   REM CORRECT FOR LENGTH
260 :     A$=STR$(H%):IF LEN(A$)=2 THEN
          A$="0"+RIGHT$(A$,1)
270 :     IF LEN(STR$(H%))=3 THEN
          A$=RIGHT$(STR$(H%),2)
280 :   PRINT
285 :   REM DO UNTIL MIN VALID
290 :     PRINT TAB(11):INPUT"MINUTES";M%
300 :     IF M%<0 OR M%>59 THEN
          PRINT UP$+S$+UP$:GOTO 290
305 :   REM ENDDO
306 :   REM CORRECT FOR LENGTH
310 :     B$=STR$(M%):IF LEN(B$)=2 THEN
          B$="0"+RIGHT$(B$,1)
320 :     IF LEN(STR$(M%))=3 THEN
          B$=RIGHT$(STR$(M%),2)
330 :   PRINT
335 :   REM DO UNTIL SEC VALID
340 :     PRINT TAB(11):INPUT"SECONDS";S%
350 :     IF S%<0 OR S%>59 THEN
          PRINT UP$+S$+UP$:GOTO 340
355 :   REM ENDDO
356 :   REM CORRECT FOR LENGTH
360 :     C$=STR$(S%):IF LEN(C$)=2 THEN
          C$="0"+RIGHT$(C$,1)
370 :     IF LEN(STR$(S%))=3 THEN
          C$=RIGHT$(STR$(S%),2)
375 :   REM SET INTERNAL TIMER
380 :     TI$=A$+B$+C$:REM SET CLOCK
381 REM ENDSEGMENT
382 :
385 REM DISPLAY SEGMENT
390 :   PRINT CL$:FOR N=0 TO 10:PRINT:NEXT
400 :   REM DO UNTIL STOPPED USING 'STOP'
```

```
405 :     REM DO WHILE HOUR<24
410 :       IF LEFT$(TI$,2)="24" THEN
            TI$="000000"
420 :       D$=LEFT$(TI$,2)+":"+MID$(TI$,3,2)+
               ":"+RIGHT$(TI$,2)
430 :       PRINT TAB(15)D$:PRINT UP$+UP$
440 :       GOTO 410
450 :     REM ENDDO
460 :   REM ENDDO
470 REM ENDSEGMENT
480 END
```

EXPLORING THE COMMODORE 64

As well as solving problems in the external world, BASIC V2 can be used to explore the innards of the Commodore 64.

With the C-64, you do not have to be an assembly language programmer, you can use BASIC to POKE about in RAM and ROM.

In Appendix K we presented a monitor program written entirely in BASIC. We used this program extensively when we were first introduced to the Commodore 64. You can use BASIC to build utilities which are useful for program development. We have included listings of two such programs in Appendix L. These utilities allow us to renumber programs, and to delete portions of programs.

RENUMBER

This is a very useful routine to have available when developing programs. A program can be renumbered by simply calling this routine. The program will renumber both GOTOs and GOSUBs, but will not renumber an IF...THEN line number unless the GOTO is present. The routine deletes itself after the renumber has been completed.

A short sample program is shown in the code. To use the program, load the program before writing code. When code has been fully developed, call the subroutine by typing GOTO 10000.

DELETE

This program can be used in program development to allow us to delete portions of a program when we no longer require them. Typically we would use extra lines of code when debugging a program and, after the program has been debugged, we would delete our debug lines.

The routine is appended to our program when we are at the development stage. When we execute the routine it deletes our section of code and then removes itself. Thus in the present state of the routine we can use it only once.

The program makes use of some of the special memory locations in the Commodore 64 to carry out the delete. When about to develop a new program load this routine first, then enter code. To delete a section of program then type GOTO 60000.

QUIZ 4.1

Use the program listings in Appendix L to discover how a BASIC program is laid out in memory.

IMPROVING BASIC

One of the main advantages of the Commodore 64 is its flexibility. It allows us to change the BASIC on board at will. It is possible to change the vectors into BASIC rerouting them to our own routines.

Those who have a disk drive will probably have used the DOS SUPPORT PROGRAM which changes the BASIC V2 disk input/output commands. This program introduces a wedge into the CHARGET routine. If the first character in a command line is '/', control is vectored to the DOS SUPPORT routines.

As well as introducing a wedge, the Commodore 64 allows us to modify the BASIC itself.

The BASIC ROM is the 8K starting at $A000. The keyboard action vectors are in a block of ROM at $A00C to $A051. Following the keyword vectors are the function vectors and the operator vectors.

To improve the BASIC we proceed as follows:

1. Copy the 8K ROM into the RAM behind it.
2. Switch out BASIC ROM.
3. Replace the keyword action vectors with our own.
4. Write the new keyword routines.

Notice that when using this procedure we do not increase the commands available, but we do change them. We can still, of course, use CHARGET to add to our commands.

To copy the 8K BASIC ROM we need a program. The structure of the program is quite simple:

```
FOR I=40960 TO 49151
    POKE I, PEEK(I)
NEXT I
```

To switch out the BASIC ROM we set bit 0 of the MOS 6510 Microprocessor On Chip I/O Port at location 0001. This is done as follows

```
POKE 1, PEEK(1) AND 254
```

We would, of course, write the above routines in assembler for the sake of speed.

The BASIC keyword action vectors are presented in the following table.

BASIC KEYWORD ACTION VECTORS

ADDRESS	CONTENTS DEC	CONTENTS HEX	KEYWORD
40972	48	30	
40973	168	A8	END
40974	65	41	
40975	167	A7	FOR
40976	29	1D	
40977	173	AD	NEXT

!40978	247	F7	!
!40979	168	A8	DATA !
!40980	164	A4	!
!40981	171	AB	INPUT# !
!40982	190	BE	!
!40983	171	AB	INPUT !
!40984	128	80	!
!40985	176	B0	DIM !
!40986	5	5	!
!40987	172	AC	READ !
!40988	164	A4	!
!40989	169	A9	LET !
!40990	159	9F	!
!40991	168	A8	GOTO !
!40992	112	70	!
!40993	168	A8	RUN !
!40994	39	27	!
!40995	169	A9	IF !
!40996	28	1C	!
!40997	168	A8	RESTORE!
!40998	130	82	!
!40999	168	A8	GOSUB !
!41000	209	D1	!
!41001	168	A8	RETURN !
!41002	58	3A	!
!41003	169	A9	REM !
!41004	46	2E	!
!41005	168	A8	STOP !
!41006	74	4A	!
!41007	169	A9	ON !
!41008	44	2C	!
!41009	184	B8	WAIT !

41010	103	67	
41011	225	E1	VERIFY
41012	85	55	
41013	225	E1	SAVE
41014	100	64	
41015	225	E1	LOAD
41016	178	B2	
41017	179	B3	DEF
41018	35	23	
41019	184	B8	POKE
41020	127	7F	
41021	170	AA	PRINT#
41022	159	9F	
41023	170	AA	PRINT
41024	86	56	
41025	168	A8	CONT
41026	155	9B	
41027	166	A6	LIST
41028	93	5D	
41029	166	A6	CLR
41030	133	85	
41031	170	AA	CMD
41032	41	29	
41033	225	E1	SYS
41034	189	BD	
41035	225	E1	OPEN
41036	198	C6	
41037	225	E1	CLOSE
41038	122	7A	
41039	171	AB	GET
41040	65	41	

```
!41041        166      A6       NEW     !
!-----------------------------------------!
```

Use the Monitor to verify any of these numbers before use. Suppose we wish to change the INPUT routine to allow for the use of cursor positioning. We would have to:

1. Change vector at 40982/40983 to point to new version of INPUT.
2. Write new INPUT routine.
3. If first character was @, say, perform cursor position subroutine.
4. Reroute to old INPUT routine ($ABBF).

The above procedure would allow us to change the syntax of the INPUT statement to:

```
INPUT [@A,B,]"string";variable list
```

or similar.

Routines to implement the above are:

```
POKE 40982,XX
POKE 40983,XX
```

and in assembler;

```
XXXX:       JSR $0079    ; Call CHRGOT routine
                         ; to get next character
            CMP #$40     ; is it '@'
            BEQ LBL      ; if it is, jump to LBL
            JMP $ABBF    ; jump to old INPUT
                         ; routine
LBL:        JSR CRSPOS   ; call cursor positioning
                         ; subroutine
            JMP $ABBF    ; jump to old INPUT
                         ; routine
            ; cursor position subroutine
CRSPOS      JSR CHRGET   ; CHRGET=$0073, gets
                         ; character from input stream
            JSR $B79E    ; input byte subroutine
            STX $57      ; store byte to memory
            JSR $B7F1    ; get next byte
            STX $58      ; store byte to memory
            CPX #$28     ; >40
            BCS ERROR    ; branch to error message
```

```
            LDA $57     ; load row number
            BCC OKDATA  ; if ok, then proceed
ERROR:      JMP $B248   ; bad subscript message
OKDATA:     JSR $E566   ; home cursor
            LDA $57     ; load row number
NEXT ROW:   LDA #$11    ; load accumulator with
                        ; cursor down character
            JSR $E716   ; output character to screen
            DEC $57
            BNE NEXT ROW
COLUMN:     LDA $58     ; load column number
            BEQ FINISH
NEXTCOL:    LDA #$1D    ; cursor right character
            JSR $E716   ; output character
            DEC $58
            BNE NEXTCOL
FINISH:     JSR $AEFD
            RTS
```

The above program is based on a listing which appeared in "Commodore Computing", February, 1984 pp 53-55. Locations $57 and $58 form part of the page zero data area available to the programmer. Note that the program makes heavy use of BASIC ROM routines, see Appendix H.

QUIZ 4.2

Amend the PRINT routine to give the user the option of PRINT @.

THE PETSPEED COMPILER

A compiler is a computer program which translates programs written in a high level language such as BASIC into machine code. This allows the programmer to develop code using the BASIC interpreter. Such code can be easily checked by running it. However, the interpreter is inherently slow in executing programs.

PETSPEED takes a Commodore BASIC V2 program and translates it into a pseudo code which the PETSPEED run time unit can execute quickly.

PETSPEED's sole aim is to provide the maximum execution speed for Commodore-64 programs, while still producing reasonably

small run time programs.

The compiler was designed to compile programs written in Commodore BASIC to run on Commodore hardware. It is therefore straightforward to use and requires very little learning. There are also very few differences between PETSPEED BASIC and BASIC V2.

PETSPEED requires a minimum of a C-64 computer and a 1541 disk drive. It can also run on a system with two 1541's and a printer, or a Commodore dual disk drive using a IEEE interface.

DIFFERENCES BETWEEN PETSPEED BASIC AND BASIC V2

The PETSPEED program is an attempt to provide a maximum compatability with BASIC V2. There are, however, some restrictions and extensions.

The restrictions are as follows:

1. When RUN is used within a program, it may not appear with a line number as its parameter - e.g. RUN 1000 is invalid.

2. LIST and SAVE are not implemented within a program.

3. Overlaid programs cannot share the same variables.

4. PETSPEED cannot handle dynamic arrays. DIM A(N) is illegal. The programmer must set aside space for all variables at compile time.

5. Programs which alter BASIC pointers in page zero may have to be changed. This may give problems when compiling games programs which use joysticks and graphics.

The extensions are as follows:

1. User defined string functions and mixed functions are permitted in PETSPEED.

   For example, both

   ```
   DEF FNA$(X) and DEF FNA(X$)
   ```

are valid in PETSPEED.

The PETSPEED manual gives a rather neat string function for right justifying numbers:

```
10 DEF FNJUST$(NUM)=RIGHT$("    "+STR$(NUM),10)
```

this right justifies the numeric NUM into a field of 10 characters.

In Commodore BASIC, this would be written as a subroutine

```
10      NUM = X
20      GOSUB 100
30      PRINT NUM$
40      END
100     NUM$ = STR$ (NUM):N = LEN(NUM$)
120     NUM$ = LEFT$("    ",10-N) + NUM$
130     RETURN
```

2. Integer FOR loops are implemented in PETSPEED. This brings COMMODORE BASIC into line with other BASIC. Therefore programmers who are in the habit of using integer loops can continue to do so with PETSPEED. PETSPEED, however, will automatically convert variables to integer type when they contain integer values, and therefore this extension is of no real value.

3. PETSPEED, at the program's discretion, can disable the STOP key. This is to make sure that all exits from the software are legal. This is of particular importance in financial software, where the program can be involved in updating ledger information when it is broken. The STOP key is disabled by default. Where the programmer needs the STOP key, however, it can be enabled as follows:

```
10 REM!es       this enables STOP key
.
.
.
.
50 REM!DS       this disables STOP key
```

While in the development phase of a program, this is required. (We've found this out to our cost!!)

PROGRAM DEVELOPMENT WITH PETSPEED

The first thing to point out about program development, is that we can now throw away all "clever tricks". PETSPEED will take away all our attempts at optimization.

All code can now be developed with plenty of remarks; code can be laid out in a reasonable fashion; subroutines can be placed out of the way at the end of the program. Thus we may develop programs using the interpreter, without really worrying about the speed of the code under execution. We can therefore hold our PDL statements as REMs without any cost.

Consider the following example program:

```
100 TI$= "000000"
110 FOR I= 1 TO 1000
120 NEXT I
130 PRINT TI$
```

When we run this program under the interpreter the final value of TI$ is 20.

Once the progam is compiled and run, however, the final value of TI$ is 2. This is a tenfold improvement in speed.

This is not really a fair comparison, because there is practically no screen output.

We can change the program as follows:

```
100 REM TEST PROGRAM FOR PETSPEED
110 TI$= "000000"
120 FOR I=1 TO 10000
130 :  PRINT CHR$(147)
140 :  PRINT CHR$(19);I
150 NEXT I
160 PRINT TI$
```

When this program is run under the interpreter the final value of TI$ is 237. When it is compiled the final value is 45.

Thus, even with screen I/O, we still have an improvement of over 5 times as fast. Therefore speed is one thing we can

forget about when developing our own programs.

The PETSPEED compiler consists of various subprograms which will carry out specific taks, as detailed below.

PETSPEED

This program asks the user to define the hardware environment and to name the program which is to be compiled. This information is passed onto the compiler proper which consists of the programs PASS 1 to PASS 4.

PASS 1

The object of PASS 1 is to scan the source program and to build a symbol table. A symbol table is a dictionary of all objects used in the program. These objects are all the variables, arrays and functions defined by the user.

Once the symbol table has been set up, it is sorted according to the frequency of use at run time. This will have the effect of saving on run time.

As well as building the symbol table, PASS 1 analyzes all DATA statements, to make the most efficient storage thereof. PASS 1 creates two files whose names are based on the source program's name. Suppose the source program was MONSTER; then PASS 1 creates:

MONSTER.A - this file contains the symbol table and a table of user defined functions, with some other system data.

MONSTER.B - this file contains the processed DATA statements.

PASS 2

PASS 2 is where PETSPEED checks the syntax of the source program. If the source program has any syntax errors they will be reported at this stage.

The main object of this pass is the building of the parse tree representation of the source program.

The parse tree is the outcome of parsing the source program. For those who have forgotten what parsing is about, we quote from the Oxford English Dictionary:

- to parse is to "resolve (a sentence, etc.) into its component parts of speech and describe them grammatically".

The parse tree therefore, reflects the structure of the program according to the grammar rules of PETSPEED BASIC.

The parse tree is written to disk and can occupy hundreds of blocks. After the parse tree has been created, the original source program is no longer required by PETSPEED. The parse tree is held in a file with extension .T. In our example the file would be called MONSTER.T.

PASS 3

PASS 3 simplifies and optimizes the parse tree created in PASS 2. When this simplification is complete, PASS 3 uses the new parse tree to build the Speedcode.

PASS 4

PASS 4 effectively builds the run time code. All code, data, GOTO/GOSUB references are linked together and a LOADable file is written to the disk. The LOADable file is given the extension .WOW. At this point, PASS 4 scratches all work files used during the compilation.

These additional workfiles are as follows (using MONSTER as the source file name):

MONSTER.C - an intermediate version of the code, fairly close to Speedcode.

MONSTER.G - this is a file of all GOTO/GOSUB references, in the form of line number of occurrence and the line number of target reference. This is the file used in PASS 4.

There are two files, which are not scratched by PASS 4, left on the disk. These files have extensions .WOW and .W. The .WOW is the LOADable file and the .W contains a list of all BASIC line numbers with their SPEEDCODE addresses. This

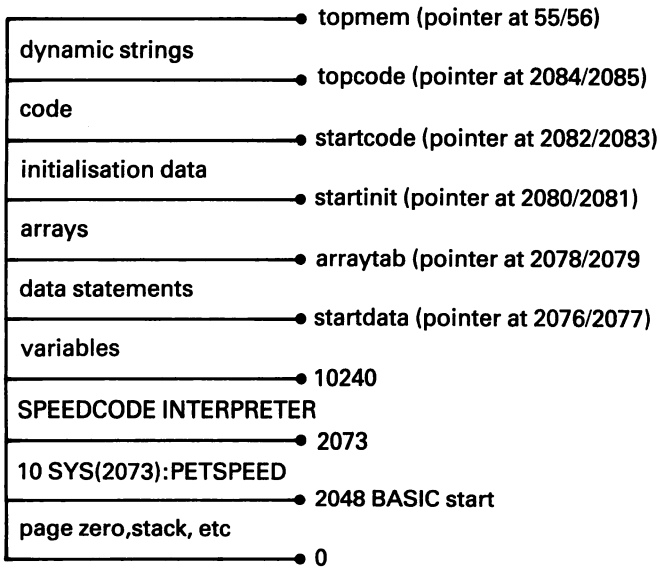
allows the PETSPEED system to give debug information.

The last file left on disk is the original source program. We can see, from the above, that the PETSPEED system is rather greedy on disk space. If we are going to run out of disk space, and if we are beyond PASS 2, we are given the option of scratching the source program, thus releasing some disk space.

THE PETSPEED OBJECT PROGRAM

At run time the PETSPEED program makes maximum use of integer arithmetic. This results in a great increase in speed. PETSPEED, however, does not use a standard approach to integers.

We must bear in mind where PETSPEED places the Speedcode. The position of the SPEEDCODE is given by the PETSPEED memory map, as below:



From the above memory map, it is seen that PETSPEED uses up at least 8K of memory for the SPEEDODE interpreter, starting at location 2073. The compiled program's Speedcode uses up memory from 10240 upwards. This has implications if we wish

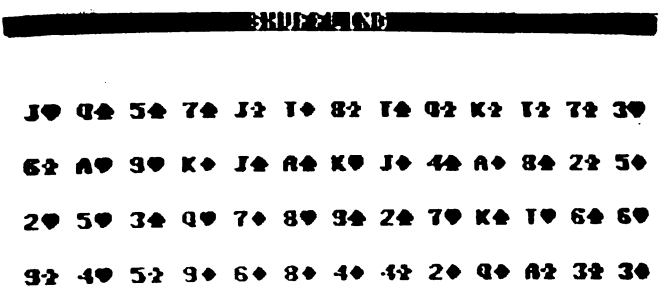
to set aside space for graphics or user defined characters.

For example, in our collection of programs 100 PROGRAMS FOR THE COMMODORE 64 (Prentice/Hall International, 1985), we had some problems with our graphics and our games programs. The problem with the graphics was that we used a wedge to change the SYS command which led to syntax errors when compiling. The problem with the games was that the graphics of the games destroyed the Speedcode program.

CASE STUDY

As an example of a compilation, we compiled the following program from our book "100 PROGRAMS FOR THE COMMODORE 64". The program is called "Shuffle", and it shuffles a deck of cards on the screen. The program uses the Commodore card characters, and so we do not need to change it at all in order to compile it. This program could be considered as the kernal of any card based game.

The screen display is as follows when the shuffle is in progress:



The program listing is as follows:

```
10 REM PROGRAM  -  SHUFFLE
20 CL$=CHR$(147):HOME$=CHR$(19):
   DOWN$=CHR$(17):ACROS$=CHR$(29)
30 H$=CHR$(115):C$=CHR$(120):
   D$=CHR$(122):S$=CHR$(97)
40 PRINT CL$;
```

```
50 X=10:Y=16:GOSUB 5000
60 PRINT "SHUFFLE"
70 A=TI
80 IF TI<A+150 THEN GOTO 80
90 REM SET UP SCREEN DISPLAY
100 BRDER=53280:SCREEN=53281
110 PURPLE=4:YELLOW=7:RED$=CHR$(28):
    BLACK$=CHR$(144)
120 CASE$=CHR$(142):UNCASE$=CHR$(14)
130 POKE BRDER,YELLOW
140 POKE SCREEN,PURPLE
150 FOR I=2 TO 9
160 HR$=HR$+MID$(STR$(I),2)+H$
170 CB$=CB$+MID$(STR$(I),2)+C$
180 DI$=DI$+MID$(STR$(I),2)+D$
190 SP$=SP$+MID$(STR$(I),2)+S$
200 NEXT I
210 HR$="A"+H$+HR$+"T"+H$+"J"+H$+"Q"+H$+"K"+H$
220 CB$="A"+C$+CB$+"T"+C$+"J"+C$+"Q"+C$+"K"+C$
230 DI$="A"+D$+DI$+"T"+D$+"J"+D$+"Q"+D$+"K"+D$
240 SP$="A"+S$+SP$+"T"+S$+"J"+S$+"Q"+S$+"K"+S$
250 PACK$=HR$+CB$+DI$+SP$
260 PRINT CL$+BLACK$
270 PRINT "               SHUFFLING"
280 FOR Z=1 TO 10:PRINT DOWN$:NEXT Z
290 PRINT
300 SHUFFLED$=""
310 FOR I=1 TO 50
320 P=(INT(RND(0)*(53-I))+1)*2-1
330 SHUFFLED$=SHUFFLED$+MID$(PACK$,P,2)
340 PACK$=LEFT$(PACK$,P-1)+MID$(PACK$,P+2)
350 T$=SHUFFLED$+PACK$
360 PRINT HOME$+DOWN$+DOWN$+DOWN$+DOWN$+DOWN$
370 FOR J=1 TO 4
380 HN$=MID$(T$,(J-1)*26+1,26)
390 PRINT " ";
400 FOR K=0 TO 12
410 CD$=MID$(HN$,K*2+1,2)
420 S$=RIGHT$(CD$,1)
430 IF S$=H$ OR S$=D$ THEN PRINT RED$;CD$;" ";:GOTO 450
440 IF S$=C$ OR S$=S$ THEN PRINT BLACK$;CD$;" ";
450 NEXT K
460 PRINT:PRINT
470 NEXT J
480 FOR Z=1 TO 50:NEXT Z
490 NEXT I
500 END
```

```
5000 PRINT HOME$;
5010 IF X=1 THEN 5030
5020 FOR R=1 TO X:PRINT DOWN$;:NEXT R
5030 IF Y=1 THEN 5050
5040 FOR C=1 TO Y:PRINT ACROS$;:NEXT C
5050 RETURN
```

Using the interpreter, the program took 159 jiffies to complete. The compiled version took 22 jiffies. The REPORT generated by the system was:

```
PROGRAM:-    SHUFFLE
-------------------
SIMPLE VARIABLES:-
-----------------

S ........ 10240    I ........ 10248    DO$ ...... 10256    H$ ....... 10264
C$ ....... 10272    D$ ....... 10280    PA$ ...... 10288    HR$ ...... 10296
CB$ ...... 10304    DI$ ...... 10312    SP$ ...... 10320    Z ........ 10328
SH$ ...... 10336    P ........ 10344    CD$ ...... 10352    CL$ ...... 10360
HO$ ...... 10368    X ........ 10376    Y ........ 10384    BL$ ...... 10392
J ........ 10400    K ........ 10408    TI$ ...... 10416    AC$ ...... 10424
A ........ 10432    TI ....... 10440    BR ....... 10448    SC ....... 10456
PU ....... 10464    YE ....... 10472    RE$ ...... 10480    T$ ....... 10488
HN$ ...... 10496    R ........ 10503    CA$ ...... 10510    UN$ ...... 10517
C ........ 10524

SUBSCRIPTED VARIABLES:-
----------------------


SYSTEM MEMORY MAP:-

VARTAB ........ 10240  (SYSTEM CONSTANT)
STARTDATA ..... 10531
ARRAYTAB ...... 10532
STARTINIT ..... 10533
STARTCODE ..... 10542
```

SUMMARY

In this chapter we discussed the concepts of program design with particular emphasis on implementing structured programming on the Commodore 64. We showed how BASIC can be used to explore the innards of the Commodore 64 and also how to change the BASIC. The chapter was completed by a case study of the use of a compiler for BASIC V2.

CHAPTER 5

Programming SID and VIC

INTRODUCTION

This chapter discusses the sound and graphics facilities of the computer. Dynamic sound techniques, user defined characters, sprites and bit mapped graphics are investigated.

SOUND

The Commodore 64 has an intelligent sound generator chip. This means that we can specify the sound we require in terms of volume, pitch and ADSR (Attack, Decay, Sustain, Release) envelope and waveform. The 6521 "SID" chip will then generate all the audio signals required. Thus the program can load SID and then go away and do something else while the sound is being generated.

The SID chip has three voices. Each voice is controlled by seven write only memory locations. The SID chip is located in the Commodore 64 memory map starting at location $D400. Thus locations $D400-$D406 control voice 1.

$D400-$D401 control the frequency of the sound. The most significant byte of the frequency control word is in $D401. $D402 and the four least significant bytes of $D403 determine mark to space ratio when a square waveform is selected-i.e.

Low mark to space ratio 

High mark to space ratio 

$D404 holds the control bytes for voice 1. We shall deal with this in detail shortly.

$D405 holds the attack and decay times for the ADSR envelope. The four most significant bits control the attack rate, the four least significant bits the decay rate.

$D406 holds the sustain value and the release time in its four most and four least significant bits respectively.

The control byte in $D404 must be considered a bit at a time. Bit 0 is the gate bit. When this is set to 1 the Attack/Decay/Sustain part of the envelope is initiated. When it is set to zero the Release section is initiated. This signal is edge triggered and a positve going edge is required at the start of each sound. Thus if the Release was not switched in on the previous sound this bit must be set to zero and back to one before the current sound.

Bit 1 is the synchronization bit. If set, this bit synchronizes the fundamental frequencies of oscillators 1 and 3. Varying the former frequency produces a range of harmonic structures at voice 1 output.

Bit 2 is the ring modulation bit. If set, this replaces the triangular waveform output of oscillator 1 with a 'ring modulated' combination of oscillator 1 and 3. Varying the frequency of oscillator 3 produces a range of non harmonic structures which create bell or gong-like sounds at voice 1 output.

Bit 3 is the test bit. When set, this bit inhibits voice 1 output. While normally used for testing, this facility may also be used to synchronize voice 1 output to an external signal.

Bits 4, 5 and 6 select the triangle, sawtooth and pulse waveforms respectively. If more than one of these bits are set the selected waveforms are ANDed.

Bit 7 selects the noise waveform. Do not set bits 4, 5 or 6 if bit 7 is set.

Memory locations $D407-$D40D control voice 2 in a similar fashion, except that the synchronization bit and the ring modulation bit in $D40A combine oscillator 2 with oscillator 1.

Memory locations $D40E-$D414 control voice 3. In this case oscillators 3 and 2 are combined for synchronization and ring modulation.

The SID chip contains a further eight registers. Four of these ($D415-$D418) are write only and the other four are

($D419-$D41C) are read only.

Memory locations $D415-$D417 control the filter. This can be a high pass, low pass or notch filter. The filter cut-off frequency is determined by the contents of $D416 and bits 0-2 of $D415. Bits 0-2 of $D417 determine whether the outputs of oscillators 1-3 respectively pass through the filter (if bit =1 output is filtered). Bit 3 determines whether an external audio input signal will be filtered. Bits 4-7 determine the resonance - or the effectiveness - of the filter.

Memory location $D418 is multi-purpose. Setting one of bits 4-6 determines whether the filter is high pass, notch (or band pass) or low pass. Setting bit 7 disconnects tho output of oscillator 3 from the audio out, enabling this voice to be used for modulation. The value in bits 0-3 determines the overall volume of sound.

Memory locations $D419-$D41A implement the microcomputer's analog to digital converters. These are not specifically associated with sound (although any external signal read by the computer could be used by the program for controlling sound) and are discussed in chapter 9.

Register $D41B holds a value which is determined by the waveform of voice 3. If, for example, this waveform is sawtooth the value in $D41B will climb steadily to $FF and then fall quickly to $00. By adding the value in $D41B to the frequency or pulse ratio control of the other two voices, a wide range of dynamic effects may be generated. If this technique is used bit 7 of $D418 should be set.

NOTE - If the noise waveform of voice 3 is selected $D41B may be used as a random number generator in games programs.

Register $D41C holds the numerical output of the ADSR envelope generator for voice 3. This may be added to the filter frequency to produce 'wah wah' effects or to the frequency control values to produce 'phaser' sounds.

The next program demonstrates how a variety of sounds may be generated. The program is written in BASIC, which is fast enough for sound generation.

Machine code can be used as easily as BASIC for simple sound effects, but where dynamic techniques are used, involving feedback from $D41B or $D41C, the arithmetic involved can

make machine code routines rather tedious.

```
10 REM SOUNDS
20 REM ******
30 REM
100 CL$=CHR$(147):REM TO CLEAR SCREEN
110 POKE 53280,3:REM CYAN BORDER
120 POKE 53281,6:REM BLUE SCREEN
130 PRINT CHR$(5):REM WHITE INK
140 S=54272:REM START OF SOUND CHIP
150 REM**************************
160 PRINT CL$:PRINT
170 PRINT TAB(7)
    "SELECT SOUND BY PRESSING:"
180 PRINT:PRINT
190 PRINT TAB(9)CHR$(18)" KEY 1 "
    CHR$(146)" - MACHINE GUN"
200 PRINT
210 PRINT TAB(9)CHR$(18)" KEY 2 "
    CHR$(146)" - EXPLOSION"
220 PRINT
230 PRINT TAB(9)CHR$(18)" KEY 3 "
    CHR$(146)" - SHOT AND RICOCHET"
240 PRINT
250 PRINT TAB(9)CHR$(18)" KEY 4 "
    CHR$(146)" - ALARM BELL"
260 PRINT
270 PRINT TAB(9)CHR$(18)" KEY 5 "
    CHR$(146)" - BOUNCE"
280 PRINT
290 PRINT TAB (9)CHR$(18)"KEY 6"
    CHR$(146)"-BREAKING WAVE"
300 PRINT
310 PRINT TAB (9)CHR$(18)"KEY 7"
    CHR$ (146)"-SURPRISE"
320 PRINT
330 PRINT TAB (9)CHR$(18)"KEY 8"
340 PRINT
350 PRINT TAB (9)CHR4(18)"KEY 9"
    CHR$(146)"-STOP PROGRAM"
370 REM**********************
380 FOR N=0 TO 23:POKE S+N,0:NEXT
390 POKE S+24,15
400 GET A$:IF A$=""THEN 400:REM
    NO SPACE BETWEEN INVERTED COMMAS
410 IF ASC(A$)<49 OR ASC (A$)>57 THEN 400
```

```
420 IF ASC(A$)=57 THEN POKE S+24,0:END
430 SL=ASC(A$)-48
440 ON SL GOSUB 1000,2000,3000,4000,5000,6000,7000,8000
450 GOTO 380
460 REM*************************
470 REM*************************
1000 REM MACHINE GUN
1010 POKE S,0:POKE S+1,100
1020 POKE S+5,5:POKE S+6,19
1030 FOR N=0 TO 10
1040 POKE S+4,0:POKE S+4,129
1050 FOR DE=0 TO 50:NEXT
1060 POKE S+4,128
1070 FOR DE=0 TO 20:NEXT
1080 NEXT
1090 RETURN
1100 REM*************************
1110 REM*************************
2000 REM EXPLOSION
2010 POKE S,0:POKE S+1,30
2020 POKE S+5,12:POKE S+6,5
2030 POKE S+4,0:POKE S+4,129
2040 FOR DE=0 TO 2000:NEXT
2050 POKE S+4,128
2060 FOR DE=0 TO 400:NEXT
2070 RETURN
2080 REM*************************
2090 REM*************************
3000 REM SHOT AND RICOCHET
3010 POKE S,0:POKE S+1,100
3020 POKE S+5,9:POKE S+6,0
3030 POKE S+4,0:POKE S+4,129
3040 FOR DE=0 TO 150:NEXT
3050 POKE S+4,128
3060 FOR DE=0 TO 50:NEXT
3070 POKE S+4,33
3080 FOR DE=0 TO 700:NEXT
3090 RETURN
3100 REM*************************
3110 REM*************************
4000 REM ALARM BELL
4010 POKE S,0:POKE S+1,100
4020 POKE S+5,9:POKE S+6,0
4030 POKE S+7,0:POKE S+8,200
4040 POKE S+12,8:POKE S+13,0
4050 POKE S+15,30
4060 FOR N=0 TO 9
```

```
4070 POKE S+4,0:POKE S+4,21
4080 POKE S+11,0:POKE S+13,0
4090 FOR DE=0 TO 30:NEXT
4100 POKE S+11,20
4110 FOR DE=0 TO 30:NEXT
4120 POKE S+11,0:POKE S+11,21
4130 FOR DE=0 TO 30:NEXT
4140 POKE S+4,20:POKE S+11,20
4150 FOR DE=0 TO 30:NEXT
4160 NEXT
4170 RETURN
4180 REM************************
4190 REM************************
5000 REM BOUNCE
5010 POKE S+1,50
5020 POKE S+5,141:POKE S+6,134
5030 POKE S+4,0:POKE S+4,33
5040 FOR FR=240 TO 0 STEP-10
5050 POKE S,FR
5060 NEXT
5070 POKE S+1,30
5080 FOR FR=240 TO 0 STEP -10
5090 POKE S,FR
5100 NEXT
5110 POKE S+4,32
5120 FOR DE=0 TO 500:NEXT
5130 RETURN
5140 REM************************
5150 REM************************
6000 REM BREAKING WAVE
6010 POKE S,0:POKE S+1,80
6020 POKE S+5,205:POKE S+6,17
6030 POKE S+15,20
6040 POKE S+4,0:POKE S+4,131
6050 FOR DE=0 TO 1500:NEXT
6060 POKE S+1,15
6070 FOR DE=0 TO 1000: NEXT
6080 POKE S+1,60
6090 FOR DE=0 TO 1500:NEXT
6100 POKE S+4,130
6110 FOR DE= 0 TO 200:NEXT
6120 RETURN
6130 REM************************
6140 REM************************
7000 REM SURPRISE
7010 POKE S,0:POKE S+1,50
7020 POKE S+5,12:POKE S+6,70
```

```
7030 POKE S+4,0:POKE S+4,65
7040 FOR PL =0 TO 255
7050 POKE S+3,PL
7060 NEXT
7070 POKE S+4,64
7080 FOR DE=0 TO 200:NEXT
7090 RETURN
7100 REM***********************
7110 REM***********************
8000 REN ALIEN SPACESHIP
8010 POKE S+1,45:POKE S+3,2
8020 POKE S+5,193:POKE S+6,133
8030 POKE S+14,200
8040 POKE S+4,0:POKE S+4,65
8050 POKE S+18,0:POKE S+18,16
8060 FOR K=0 TO 150
8070 A%=PEEK(S+27)*5:B%=A%/256
8080 POKE S,A%-B%*256
8090 POKE S+1,45+B%
8100 NEXT
8110 POKE S+4,64
8120 FOR DE=0 TO 300:NEXT
8130 RETURN
8140 REM************************
8150 REM************************
```

LOW RESOLUTION GRAPHICS

Display on the Commodore 64 is implemented by the VIC II chip. This chip can access 16K of memory at a time. The 64K memory is therefore divided into four banks of 16K. Normally bank 0 is accessed ($0000-$3FFF). The bank accessed by the VIC II chip should contain all character and sprite information and the screen memory. If, for example, we wish to put our sprite data in bank 3 ($C000-$FFFF) out of the way of BASIC programs, then we can change to this bank.

The bank selected is controlled by the 6526 Complex Interface Adapter (CIA) #2 chip starting at $DD00 in memory. We shall look more closely at this chip in chapter 9. The bank is selected by bits 0 and 1 of port A ($DD00). These bits must first be set to outputs by setting bits 0 and 1 of control register $DD02 to 1.

Thus:

```
LDA  $DD02
ORA  #$03
STA  $DD02
LDA  $DD00
AND  #$FC
ORA  #CONST
STA  $DD00
```

will set bank 0 to 3 depending on the value of CONST.

CONST	BANK	VIC RANGE
$00	3	$C000-$FFFF
$01	2	$8000-$8FFF
$02	1	$4000-$7FFF
$03	0	$0000-$3FFF

Note that the Commodore 64 character set is not avaliable to the VIC II chip in banks 1 and 3.

Screen memory may be located at any one of sixteen areas within a bank - provided that the location is sensible (for example it would be most unwise to start the screen on page zero). The location of memory is determined by the value in the four most significant bits of $D018. The other bits in this memory location control the character set and should not be disturbed.

Thus

```
LDA  $D018
AND  #$0F
ORA  #CONST
```

will set the screen in a position within the bank selected. Add the bank start address to get the absolute address.

CONST	BANK POSITION
$00	$0000
$10	$0400
$20	$0800
$30	$0C00
$40	$1000
$50	$1400
$60	$1800

$70	$1C00
$80	$2000
$90	$2400
$A0	$2800
$B0	$2C00
$C0	$3000
$D0	$3400
$E0	$3800
$F0	$3C00

The page in which the absolute address of the screen starts, that is the two most significant digits of the address, must be loaded into memory location $0288 (HIBASE). Color memory is fixed at $D800-$DBE7. This cannot be moved.

CHARACTER MEMORY

The full character set takes up 2K bytes of data. It is held in character ROM at $D000-$DFFF. Normally this ROM is switched out as these locations are used for I/O devices. The Commodore 64 gets round this very cleverly. The VIC II chip points to character ROM by pointing to locations $1000-$1FFF within its current memory bank. Hardware signals then cause this address to 'echo' in the character ROM so that VIC II reads character information from (say) $D008-$D00F when it is (apparently) addressing $1008-$1FFF. This operation does not affect $1000 to $1FFF in any way. Only banks 0 and 2 have this 'echo' or 'imaging' facility.

If we wish to alter the characters in any way, or to read them from other banks, it is necessary to transfer all or part of the character set from ROM to RAM. This is implemented as follows:

1. Turn off keyboard scan interrupt timer.
2. Switch out I/O and switch in character ROM.
3. Copy the contents of character ROM to a suitable area of RAM.
4. Switch in I/O and switch out character ROM.
5. Turn keyboard scan interrupt timer back on.
6. Change the character set pointer register to point to the start of the designated RAM area.

If we are using BASIC programs it is necessary to lower the top of BASIC memory to protect the character set from being overwritten.

The next program will transfer the first 64 characters to RAM starting at $3000.

```
     LDA  $DC0E       ;Switch off
     AND  #$FE        ;keyscan interrupt
     STA  $DC0E       ;timer.
     LDA  $01         ;Switch in
     ADD  #SFB        ;character
     STA  $01         ;ROM.
     LDA  #$30        ;Set base
     STA  $FC         ;addresses
     LDA  #$30        ;$3000 &
     STA  $FE         ;$D000 in
     LDA  #$00        ;$FB-$FC
     STA  $FB         ;and
     STA  $FD         ;$FD-$FE.
     LDY  #$00        ;Initialize pointer.
LOOP:LDA  ($FB), Y    ;Transfer
     STA  ($FD), Y    ;memory.
     INY              ;]
     BNE  LOOP        ;]
     INC  $FC         ;]
     INC  $FE         ;] 64X8 bytes
     LDX  #S04        ;]
     CPX  $FE         ;]
     BNE  LOOP        ;]
     LDA  $01         ;Switch
     ORA  #S04        ;in
     STA  $01         ;I/O.
     LDA  $DC0E       ;Switch on
     ORA  $#01        ;key scan interrupt
     STA  $DC0E       ;timer.
     LDA  $D018       ;Change character.
     AND  #$F0        ;Set pointer
     CLC              ;to point
     ADC  #$0C        ;to $3000.
     STA  $D018       ;
```

Once the characters are stored in RAM they may be redefined. For example the remainder of this program could be:

```
     LDA  #$30      ;Restore base address
     STA  $FE       ;to $3000
INVT:LDA  ($FD),Y   ;
     EOR  #$FF      ;
     STA  ($FD), Y  ;complement all memory contents
```

```
                ;in $3000-$31FF
INY             ;
BNE  INVT       ;
INC  $FE        ;
CPX  $FE        ;
BNE  INVT       ;
RTS
```

This would change all characters to their inverse.

SPRITES

Sprite shapes are held in memory in what are known as Moveable Object Blocks (MOB'S). Each MOB is made up of 64 bytes. 63 bytes define the sprite shape (see appendix G), the final bit is spare. One 16K bank of memory can hold up to 256 MOBs, although not all of these could be used for sprites. Sprites may be held in any of the four memory banks. For ease of explanation we shall assume that bank 0 (the default bank) is used.

Up to eight sprites may be enabled at any one time. Each sprite may take its shape from any one MOB. The sprite data pointers in $07F8-$07FF each point to one of the possible 256 MOB locations in the bank to indicate the shape of sprites 0 to 7 respectively.

Changing the value in any one of these registers will cause the shape of the corresponding sprite to change.

Sprites are controlled by memory mapped registers in the VIC II chip, starting at $D000. The functions of various sprite registers are as follows:

```
$D000 - sprite 0 X position
$D001 - sprite 0 Y position
$D002 - sprite 1 X position
$D003 - sprite 1 Y position
$D004 - sprite 2 X position
$D005 - sprite 2 Y position
$D006 - sprite 3 X position
$D007 - sprite 3 Y position
$D008 - sprite 4 X position
$D009 - sprite 4 Y position
$D00A - sprite 5 X position
$D00B - sprite 5 Y position
```

$D00C - sprite 6 X position
$D00D - sprite 6 Y position
$D00E - sprite 7 X position
$D00F - sprite 7 Y position
$D010 - most significant bit of X position
$D012 - raster register(MSB in bit 7 of $D011)
$D015 - sprite enable
$D017 - sprite expand Y
$D019 - interrupt status
$D01A - interrupt enable
$D01B - background priority
$D01C - multicolor select
$D01D - sprite expand X
$D01E - sprite-sprite collision
$D01F - sprite-background collision
$D021 - $D024 background colors 0-3
$D025 - $D026 sprite multicolor 0-1
$D027 - $D02E sprite color 0-7

Some of these are reasonably self explanatory. Others require more detail.

The raster register $D012 returns the lower eight bits of the raster position. Bit 7 of $D011 holds the most significant raster position bit. The raster scan is the device which creates the picture on the television or monitor screen. To prevent flicker, changes should be made to the screen display when the raster is off the visible area - i.e. raster values between 51 and 251. If the raster register (including the most significant bit) is written to, then a raster compare is initiated. This results in an interrupt pulse being generated (provided the interrupt is enabled - see below). An interrupt routine is entered with a bit set in the interrupt register to indicate that this is a raster interrup and not a normal keyboard scan interrupt. The interrupt routine could, for example, reposition eight sprites normally on the top half of the screen on to the botton half. This will happen with high frequency, resulting in sixteen sprites being seen on the screen.

A sprite to sprite collision will result in a bit being set for each sprite involved in the collision. Thus a collision between sprite zero and sprite six will result in bits zero and six being set to 1 in $D01E. A sprite to sprite collision will also generate an interrupt if the appropriate interrupt enable bit is set.

Sprites may move in front of or behind background shapes. This is determined by the condition of the corresponding bit in the background priority register $D01B. For example if bit 0 in this register is set (1), sprite 0 will move behind background shapes. If this bit is reset (0) sprite 0 will move in front of these shapes.

The interrupt enable register $D01A determines whether an interrupt from raster compare, sprite to sprite collision or sprite to background collision generates a maskable interrupt signal. If the appropriate bit is set to one the interrupt is enabled. Four bits of the register are used.

Bit 0 - enables raster compare interrupt
Bit 1 - enables sprite/background collision interrupt
Bit 2 - enables sprite/sprite collision interrupt
Bit 3 - enables light pen control interrupt (see chapter 9)

The interrupt status register is used by interrupt routines to determine which condition caused the interrupt. The interrupt routine can then take the appropriate action. Five bits of the register are used.

Bit 0 - set by raster compare
Bit 1 - set by sprite/data collision
Bit 2 - set by sprite/sprite collision
Bit 3 - set by negative transition of light pen (see chapter 9)
Bit 7 - set by normal IRQ interrupt

This register should be cleared by the interrupt routine once the interrupt source has been identified.

Sprite control programs are best written in machine code. Moving sprites smoothly, one pixel at a time, results in very slow movement if BASIC is used. Sprite handling using interrupt routines enables a great deal to be happening on the screen, apparently simultaneously. As sprites are controlled by loading values into VIC II registers sprite handling programs are usually very straightforward. A sprite creation and control program was given when we discussed interrupts in chapter 3.

DRAWING AND WRITING ON THE BIT MAP

The Commodore 64 can produce high resolution graphics and place text on to the graphics screen. This is done by first of all writing a machine code program to set up a high resolution bit mapped screen. The program can be entered using our assembler or using a BASIC loader routine as below.

```
100  REM program to set up bit map
110  REM M/C routine located at 49152
120  FOR N=49152 TO 49221
130  READ A
140  POKE N, A
150  NEXT N
1900 REM MACHINE CODE ROUTINE
1910 REM SCREEN LOCATED AT 49152
1920 REM BIT MAP AT 8192
1930 REM
1940 REM            ********
1950 REM            *      *
1960 REM            * DATA *
1970 REM            *      *
1980 REM            ********
1990 REM
2000 REM DATA FOR M/C CODE ROUTINE
2010 DATA 169,8:REM          LDA #8
2020 DATA 13,24,208:REM      ORA 53272
2030 DATA 141,24,208:REM     STA 53272
2040 DATA 169,32:REM         LDA #32
2050 DATA 13,17,208:REM      ORA 53265
2060 DATA 141,17,208:REM     STA 53265
2070 DATA 169,0:REM          LDA #00
2080 DATA 133,2:REM          STA 02
2090 DATA 169,32:REM         LDA #32
2100 DATA 133,3:REM          STA 03
2110 DATA 160,0:REM          LDY #00
2120 DATA 169,0:REM       L1:LDA #00
2130 DATA 145,2:REM       L2:STA(02),Y
2140 DATA 200:REM            INY
2150 DATA 208,251:REM        BNE L2
2160 DATA 230,3:REM          INC 03
2170 DATA 169,64:REM         LDA #64
2180 DATA 197,3:REM          CMP 03
2190 DATA 208,241:REM        BNE L1
2200 DATA 169,4:REM          LDA #4
2210 DATA 133,3:REM          STA 03
```

```
2220 DATA 169,1:REM      L3:LDA #01
     WHITE SCREEN, BLACK INK
2230 DATA 145,2:REM      L4:STA(02),Y
2240 DATA 200:REM           INY
2250 DATA 208,251:REM       BNE L4
2260 DATA 230,3:REM         INC 03
2270 DATA 169,7:REM         LDA #7
2280 DATA 197,3:REM         CMP 03
2290 DATA 208,241:REM       BNE L3
2300 DATA 169,1:REM         LDA #01
2310 DATA 145,2:REM      L5:STA(02),Y
2320 DATA 200:REM           INY
2330 DATA 192,232:REM       CPY #232
2340 DATA 208,249:REM       BNE L5
2350 DATA 96:REM            RTS
```

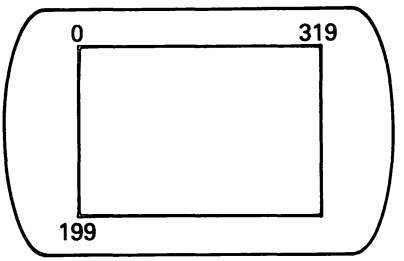
This program is called by the BASIC line

```
350 SYS 49152
```

We can then draw graphics on the screen by setting the appropriate bit to on or off.

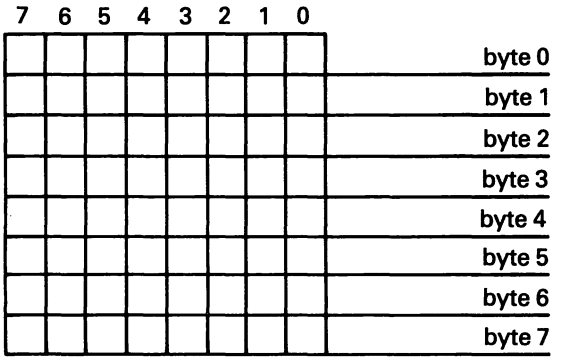
To set a picture element (pixel) on or off, we have to set or reset the corresponding bit in the high resolution RAM.

In bit map mode, the Commodore screen consists of an array of pixels, 320 to a row and 200 rows to the screen



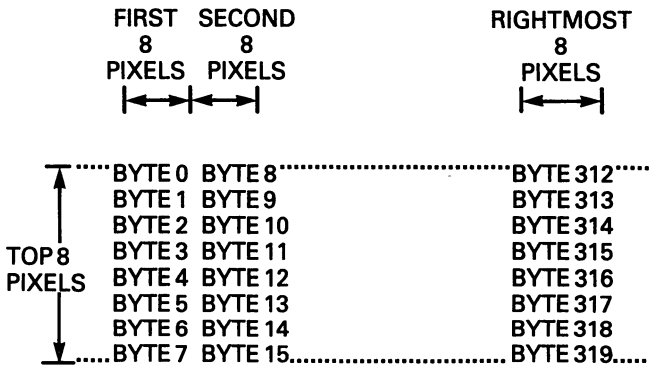
Thus the screen consists of 64000 pixels. These pixels, however, are not arranged in a straightforward manner in memory. Effectively, the bit map screen is a set of 1000 programmable characters arranged in 25 rows of 40 characters each.

To define a character we require 8 bytes as in the following grid.



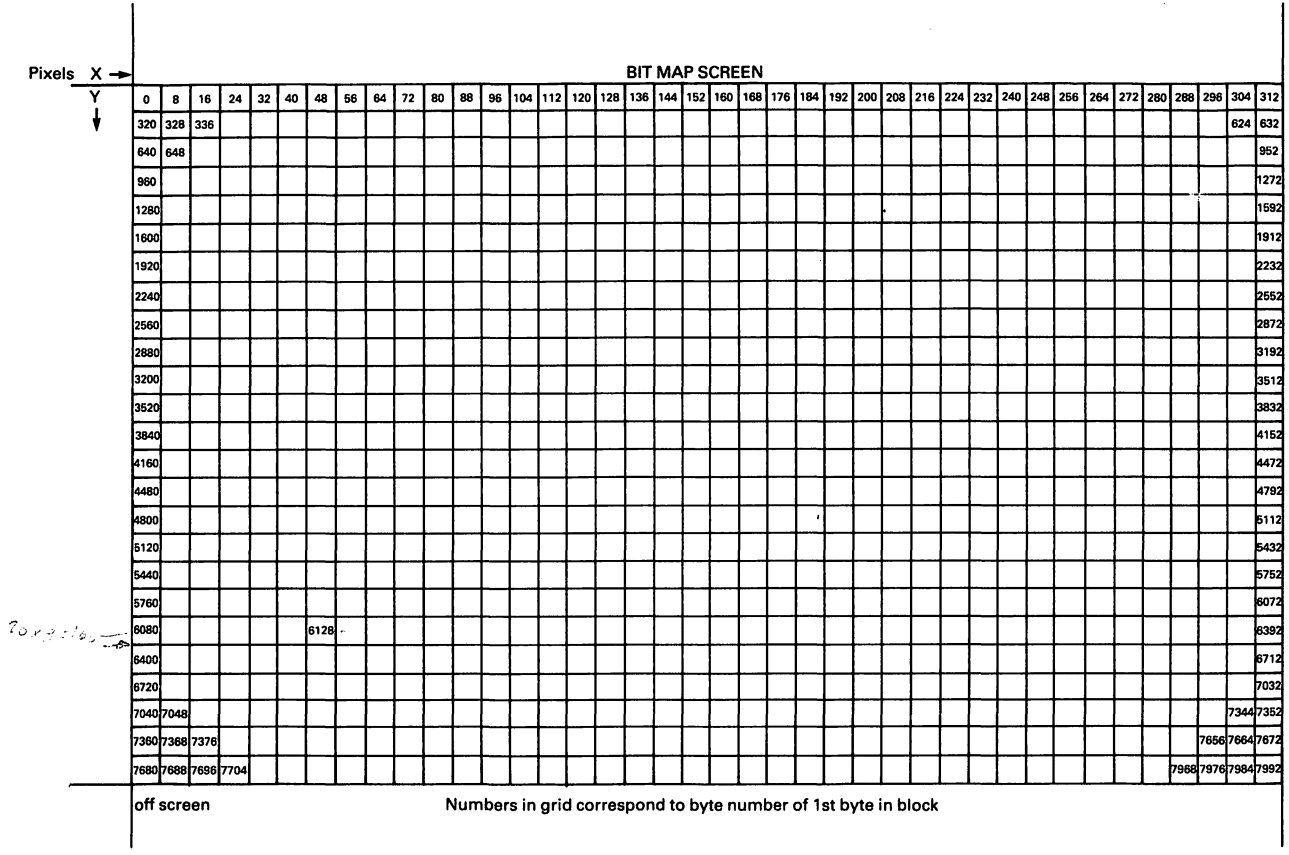
Suppose we wish to set a pixel two pixels down from the top of the screen and 4 pixels from the left edge of the screen. Then we would have to set bit 4 of byte 1 of the character at the top left of bit map screen.

Our 64000 pixels of bit map are arranged in 8000 bytes of screen RAM. The first eight bytes cover the top left hand portion of the screen. The rest of the bytes are arranged as follows:



We thus have on 8000 bytes of RAM allocated as in the following diagram. Each square makes up 8 contiguous bytes of our 8000 bytes of RAM.

From the diagram we can see that, if we wish to set the pixel with coordinate X=54 and Y=154, we have to set a


Pixels X →
Y ↓
BIT MAP SCREEN
0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 128 136 144 152 160 168 176 184 192 200 208 216 224 232 240 248 256 264 272 280 288 296 304 312
320 328 336 624 632
640 648 952
960 1272
1280 1592
1600 1912
1920 2232
2240 2552
2560 2872
2880 3192
3200 3512
3520 3832
3840 4152
4160 4472
4480 4792
4800 5112
5120 5432
5440 5752
5760 6072
6080 6128 6392
6400 6712
6720 7032
7040 7048 7344 7352
7360 7368 7376 7656 7664 7672
7680 7688 7696 7704 7968 7976 7984 7992
off screen
Numbers in grid correspond to byte number of 1st byte in block

bit in the 8 bytes block starting at 6128th byte of our 8000 bytes of bit map RAM.

There is a straightforward formula to set such a bit. Suppose our screen RAM starts at address START and the screen is made up of 25 rows of 40 blocks of 8 bytes.

Then the row number of pixel (X, Y) is

```
INT (Y/8)
```

since there are 8 bytes per row. The block number is

```
INT (X/8)
```

The byte number within the block is given by

```
Y AND 7
```

The bit of that byte is given by

```
7-(X AND 7)
```

Putting all this together we have

```
BYTE=START + (INT(Y/8))*320+8*INT(X/8)+Y AND 7
```

and to set the bit, we have

```
BIT = 7 - (X and 7)
POKE BYTE,PEEK(BYTE) OR 2^BIT
```

We can now draw on our bit map screen. The following subroutine draws a sine wave on the bit map at 8192.

```
4980  REM SAMPLE PIECE OF GRAPHIC.
4990  REM SINE CURVE
5000  FOR X=0 TO 319 STEP 5
5010  Y=INT(90+80*SIN(X/10))
5020  BY=8192 +(INT(Y/8))*320+8*INT (X/8)+(Y AND 7)
5030  BI=7-(X AND 7)
5040  POKE BY, PEEK(BY) OR (2^BI)
5050  NEXT I
5060  RETURN
```

To clear the bit map and draw this curve we would write

```
350 SYS 49152:GOSUB 5000
```

If we want to label this curve we proceed by grabbing the character ROM to allow us to copy characters from ROM to the bit map at 8192 onwards.

To get access to the character ROM, we must switch out the I/O register. This allows us access to the character ROM. We can then switch in the character ROM, and copy in text.

If we do this, no interrupts can be allowed to take place. This is because the I/O registers are needed to service the interrupts. To turn off interrupts we need the following BASIC line

```
360 POKE 56334, PEEK(56334) AND 254
```

This sets bit 0 of CIA control register to 0.

To switch in character ROM we set bit 2 of location 1 to 0 thus:

```
370 POKE 1, PEEK(1) AND 251
```

Now that we have our ROM avaliable, we can copy any appropriate character from ROM to the bit map RAM at 8192. The following subroutine accomplishes that task:

```
3980  REM SUBROUTINE TO PRINT STRING S$
3990  REM AT ROW R AND CHARACTER C
3995  REM OF THE BIT MAP SCREEN
4000  GM=8192+(R-D*320+(C-1)*8
4005  REM FIND APPROPRIATE BLOCK
4010  FOR I=1 TO LEN(S$)
4020  K=ASC(MID$(S$,I,1)):REM CONSIDER CHARACTER
4030  REM CONVERT FROM ASCII TO CODE
4040  IF K>63 AND K<91 THEN K=K-64
4050  K=K*8+53248:REM FIND CHARACTER BIT PATTERN
4060  FOR J=0 TO 7
4070  BT=PEEK(K+J):REM GET BYTE
4080  POKE(GM+J), BT:REM PLACE BYTE ON BIT MAP
4090  NEXT J
4100  GM=GM+8:REM NEXT CHARACTER
4110  NEXT I
4120  RETURN
```

We can use this subroutine to label our graph.

```
470  S$="SINE CURVE"
480  R=25: C=4
490  GOSUB 4000
```

After labelling our high resolution screen, we must reset the machine.

```
500  POKE 1, PEEK(1) OR 4
510  POKE 56334,PEEK(56334) OR 1
```

Once this has been done, we can finish

```
520  END
```

QUIZ 5.1

Adapt the program which has just been developed to produce a labelled cosine curve, and allow the user to place a piece of text anywhere on the screen.

SUMMARY

Sound in the Commodore 64 is controlled by the SID chip. The sound control registers appear on the memory map between locations $D400 and $D41C.

Display on the micro is implemented by the VIC II chip. This can access 16K of memory at a time, resulting in memory being arranged in four 16K blocks for display purposes. User defined graphics may be generated by transferring character set data from ROM to RAM, switching control to RAM and redefining characters.

Sprite shapes are held in Moveable Object Blocks (MOBs) each of which consists of 64 bytes. Up to eight sprites can be enabled at any one time. Sprites are controlled by memory mapped registers in $D000 to $D02E.

High resolution graphics are implemented by putting the machine into bit map mode. This results in 8 kilobytes of memory being dedicated to the control of the high resolution screen.

CHAPTER 6

High Resolution Graphics

INTRODUCTION

We have seen that it is possible, but awkward, to produce high resolution graphics on the C-64.

We know, however, that the C-64 is an infinitely flexible machine, and we can therefore adapt it to allow us to explore graphics without being bogged down too much.

We can write a high resolution graphics utility of our own, or buy something like "SIMON'S BASIC" to extend our capabilities.

We have used the "MINIGRAPH" program. This program was written by PAUL SCHATZ and can be found in TIM ONOSKO'S book "Commodore 64: getting the most from it", which is published by Prentice/Hall International.

The program loads a set of machine code subroutines into memory locations 36883 to 37681. Using these routines we can reduce our own programs dramatically.

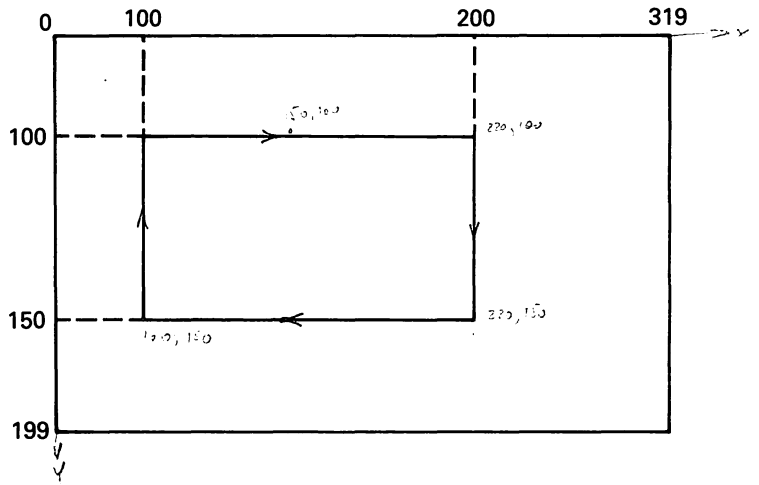
Once the routines have been installed, we can do the following.

1. Bit map is enabled by issuing the command SYS 50195.
2. Bit map is disabled with SYS 50198.
3. Bit map is cleared with SYS 50207
4. Colors displayed on the bit map are set with SYS 50210, F, B. F and B are the Foreground and Background numbers, and have values between 0 and 15.
5. A point is plotted on the bit map with SYS 50201, X, Y, M. X and Y are the horizontal and vertical positions of the point. The horizontal position can be from 0 to 319 and the vertical position can be from 0 to 199. The upper left hand corner is 0, 0. The last variable M, sets the plotting mode. If M=1 the foreground color is used. If M=2 then background color is used. If M=3 then the color of the point is flipped.
6. A line is drawn with the command SYS 50204, X, Y, M. X and Y are the coordinates of the end point of the line,

the color of the line is determined by M, as before.

The program has some error correcting features to help us to enter the data correctly. The listing and dissassembly of the program is given in Appendix I.

If we wish to draw a rectangle on the screen, we must plan where it is to go and then write the program to place it there.



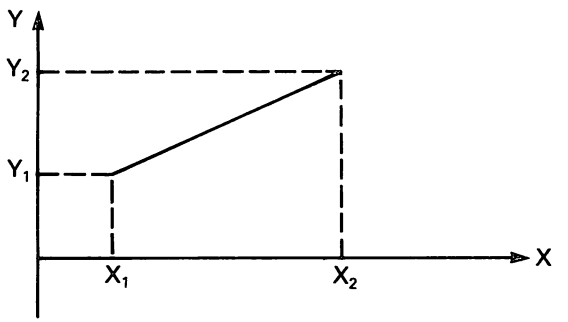
The program to draw the above rectangle is as follows

```
 10 REM ENSURE GRAPHICS HAS BEEN LOADED
 20 SYS 50207:REM CLEAR BIT MAP
 30 SYS 50195:REM TURN ON BIT MAP
 40 SYS 50210, 0, 1:REM SELECT COLORS
 50 SYS 50201, 100, 100:REM PLOT POINT X=100, Y=100
 60 SYS 50204, 200, 100:REM DRAW LINE TO X=200, Y=100
 70 SYS 50204, 200, 150:REM DRAW LINE TO X=200, Y=150
 80 SYS 50204, 100, 150:REM DRAW LINE TO X=100, Y=150
 90 SYS 50204, 100, 100:REM DRAW LINE TO X=100, Y=100
100 GET A$:IF A$ =""THEN 100
110 REM PRESS ANY KEY TO FINISH
120 SYS 50198:REM TURN OFF BIT MAP
130 END
```

That, roughly, is the idea of using MINIGRAPH. We can now try out some graphics ideas.

SHADING

Graphics can be used to shade objects on the computer's screen. We can shade a solid object by placing dots over its surface. Before considering a surface, let us consider drawing a straight line, but using the idea of shading.



From high school mathematics the equation of a straight line is given by

```
Y = MX+C
```

where, from the above diagram:

```
M = (Y2-Y1)/(X2-X1)
```

and `C = Y1-M*X1.`

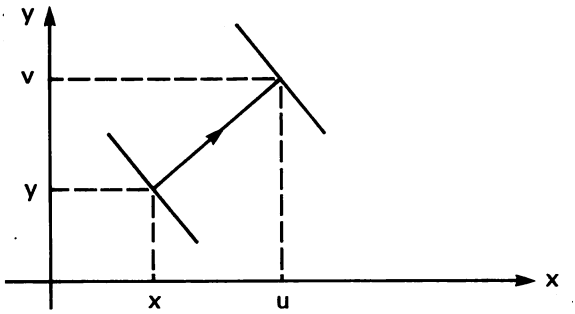
The number of dots on the line is given by the number of dots per unit length (illumination) and the length of the line, where the length of the line is given by `SQR((Y2-Y1)^2+(X2-X1)^2)`. The number of dots to be plotted is then given by length/illumination. If the illumination is even along the line, then the distance between the dots is constant.

Using MINIGRAPH, its fairly easy to construct a shading routine for shading along a straight line. We use the plot routine of MINIGRAPH to draw a line by placing a series of dots along it. The density of the dots gives a measure of the illumination of the line.

```
100  I=.1:REM ILLUMINATION
110  X1=100:Y1=100:REM START POINT OF LINE
120  X2=200:Y2=150:REM FINISH POINT OF LINE
130  SYS 50207:REM CLEAR BIT MAP
140  SYS 50195:REM TURN ON BIT MAP
150  SYS 50210, 0, 1:REM SELECT COLORS
160  GOSUB 1000:REM DRAW LINE
170  GET A$:IF A$="" THEN 170
180  REM PRESS ANY KEY TO STOP
190  SYS 50198:REM TURN OFF BIT MAP
200  END
985  REM SHADE LINE SUBROUTINE
990  REM LINE IS DRAWN FROM X1, Y1 TO X2, Y2
995  REM WITH ILLUMINATION CONSTANT 1
1000 IF X2=X1 THEN GOTO 1120
1010 M=(Y2-Y1)/(X2-X1)
1020 C=Y1-M*X1
1030 D=SQR(X1-X2)^2+(Y2-Y1)^2)
1040 N=D*I
1050 DX=(X2-X1)/N
1060 FOR K=1 TO N
1070 J=INT(X1+K*DX)
1080 L=INT(M*J+C)
1090 SYS 50201, J, L, 1
1100 NEXT K
1110 RETURN
1120 REM TAKE CARE OF VERTICAL LINES
1130 D=ABS(Y2-Y1)
1140 N=D*I
1150 IF Y2<Y1 THEN Y2=T:Y2=Y1:Y1=T
1160 DY=(Y2-Y1)/N
1170 FOR K=1 TO N
1180 J=INT(Y1+K*DY)
1190 L=INT(X1)
1200 SYS 50201, L, J, 1
1210 NEXT K
1220 RETURN
```

MOVING A LINE (Translation)

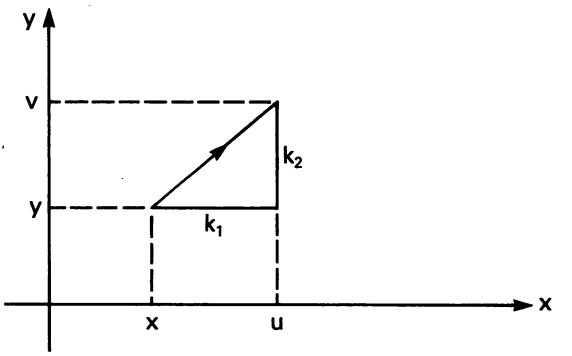
To translate a line, we must compute the new end points of the line and then draw it.



A translation can be represented by a pair of numbers which is added to the number pair representing each point in the line. In vector notation

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} k1 \\ k2 \end{pmatrix}$$

Diagramatically, we have



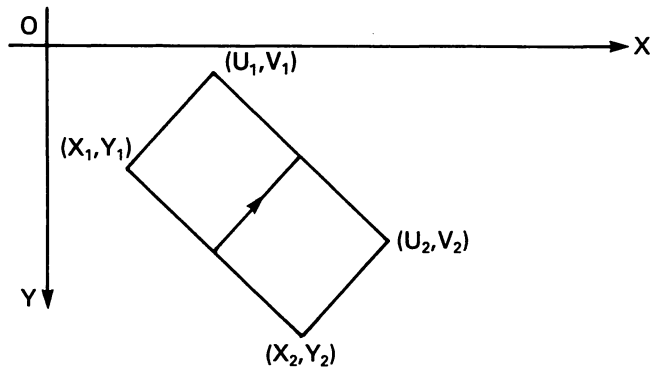
If we can move a single line, then we have the capability of moving line drawings about the screen. We can implement a translation subroutine as follows:

```
1960  REM TRANSLATION SUBROUTINE
1970  REM LINE FROM X1,Y1, TO X2,Y2, IS
1980  REM MOVED K1 IN X-DIRECTION
```

```
2000  X1=X1+K1:Y1=Y1+K2
2010  X2=X2+K1:Y2=Y2+K2
2020  GOSUB 1000
2030  RETURN
```

Using line drawing and translation subroutines, we can develop a routine for shading a parallelogram.

The parallelogram is drawn by taking a vector (a straight line) and moving it to a new position, drawing many intermediate lines between the starting and the finishing vectors. We can now use the shading routine to mimic illumination, this time in two directions. To draw the parallelogram we need to know the position of the generating vector and the distance to be moved in both the X and Y directions.



The program to construct the parallelogram would be as follows

```
100  X1=100:Y1=100:REM FIRST POINT
110  X2=150:Y2=150:REM SECOND POINT
120  I1=5:REM X-ILLUMINATION CONSTANT
130  I2=5:REM Y-ILLUMINATION CONSTANT
140  R1=50:REM X MOVEMENT
150  R2=50:REM Y MOVEMENT
160  SYS 50207:REM CLEAR BIT MAP
170  SYS 50195:REM TURN ON BIT MAP
180  SYS 50210,0,1:REM SELECT COLORS
300  GOSUB 3000:REM PARALLELOGRAM SHADING ROUTINE
310  GET A$:IF A$=""THEN 310
320  SYS 50198:REM TURN OFF BIT MAP
```

```
330  END
1000 REM LINE SHADING SUBROUTINE AS BEFORE
.
.
.
2000 REM TRANSLATION SUBROUTINE AS BEFORE
.
.
.
2960 REM PARALLELOGRAM SHADING ROUTINE
2970 REM GENERATING VECTOR IS X1,Y1, TO X2,Y2
2980 REM ILLUMINATION CONSTANTS I1, I2
2990 REM X MOVEMENT R1;Y MOVEMENT R2
3000 I=I1
3100 GOSUB 1000:REM DRAW VECTOR
3110 LN=SQR(R1*R1+R2*R2)
3120 M=LN*#2 I2
3130 D1R=R1/M
3140 D2R=R2/M
3150 FOR P=1 TO M
3160 K1=D1R
3170 K2=D2R
3180 I=I1
3190 GOSUB 2000
3200 NEXT P
3210 RETURN
```

QUIZ 6.1

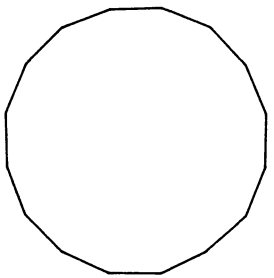
Use the illumination factors to draw other regular shapes with variable illumination. Note that we have used a rather crude method of varying illumination. A better means of illuminating an object would be to imagine that our flat object is illuminated by a point source. We could then cut up our object into a rectangular grid. By taking the center of each rectangle and calculating its distance from the point source, we would have a better determination of the illumination required for that rectangle

The brightness (number of pixels set on per unit area) will be inversely proportional to the square of the distance between the center of the rectangle and the point source.

DRAWING CIRCLES

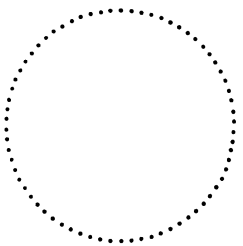
When using the C-64 we always have to make decisions when implementing graphics. If, for example, we wish to draw a circle, we could use one of the following methods.

1. Short line segments:



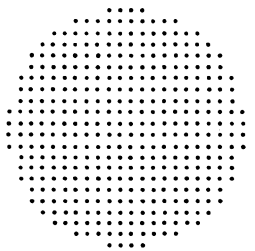
In this routine the circumference of the circle is approximated by joining up a series of short line segments. The shorter the segments, the better the approximation.

2. Dots distributed round the circumference:



In this routine, the circumference of the circle is approximated by distributing a number of dots evenly along it.

3. Solid shaded circle:



In this routine, a circle is filled with shaded straight lines

When involved in such routines, we find that we are continually recalculating trigonometic functions. It can, therefore, be useful to set up an array of values of such functions.

```
980  REM SUBROUTINE TO SET UP ARRAYS OF
990  REM SINES AND COSINES
1000 REM DATA HELD IN S(I), C(I)
1010 FOR I=1 TO 200
1020 S(I)=SIN(I*2*[PI]/200)
1030 C(I)=COS(I*2*[PI]/200)
1040 NEXT I
1050 RETURN
```

Using these arrays we can write circle drawing routines.

CIRCLE DRAWING WITH SHORT STRAIGHT LINES

```
1980 REM SUBROUTINE TO DRAW CIRCLES
1990 REM USING SHORT LINES
2000 PRINT
2010 INPUT "ENTER RADIUS OF CIRCLE";RD
2020 PRINT "THE CIRCLE WILL BE CENTERED ON THE SCREEN
2030 PRINT "PRESS ANY KEY TO DRAW CIRCLE"
2040 PRINT "WHEN CIRCLE HAS BEEN DRAWN, PRESS KEY"
2045 GET A$:IF A$=""THEN 2045
2050 SYS 50195:REM TURN ON BIT MAP
2060 X=INT(RD*S(1)+159)
2070 Y=INT(RD*C(1)+100)
2080 SYS 50201, X, Y, 1:REM PLOT POINT X, Y
2100 FOR I=1 TO 200
2110 X=RD*S(I):Y=RD*C(I)
2120 X=X+159:Y=Y+100
2130 X=INT(X):Y=INT(Y)
2140 SYS 50204, X, Y, 1:REM DRAW TO POINT X, Y
2150 NEXT I
2160 RETURN
```

CIRCLE DRAWING WITH DOTS AROUND CIRCUMFERENCE

```
2980 REM SUBROUTINE TO DRAW CIRCLES
2990 REM USING DOTS
3000 PRINT "[CS]"
3010 INPUT "ENTER RADIUS OF CIRCLE";RD
3020 PRINT "THE CIRCLE WILL BE CENTERED ON THE SCREEN"
3030 INPUT "ENTER NUMBER OF DOTS 10-200";DT
3040 PRINT "PRESS ANY KEY TO DRAW CIRCLE"
3050 PRINT "WHEN CIRCLE HAS BEEN DRAWN, PRESS KEY"
3055 GET A$:IF A$=""THEN 3055
3060 SYS 50195:REM TURN ON BIT MAP
```

```
3070  FOR I=1 TO 200 STEP 200/DT
3080  X=RD*S(I):Y=RD*C(I)
3090  X=X+159:Y=Y+100
3100  X=INT(X):Y=INT(Y)
3110  SYS 50201, X, Y, 1:REM PLOT POINT X, Y
3120  NEXT I
3130  RETURN
```

CIRCLE DRAWING WITH SHADING

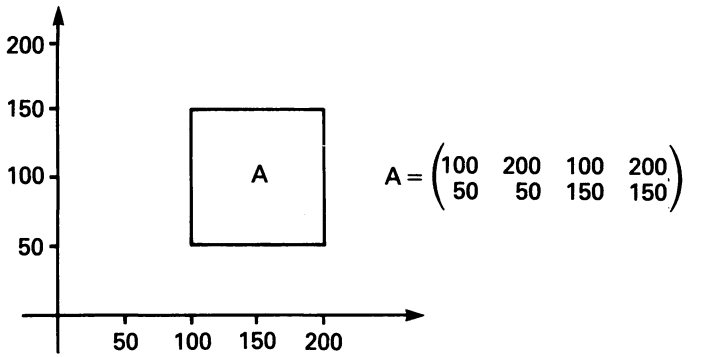
```
3980  REM SUBROUTINE TO DRAW CIRCLES
3990  REM USING DOT SHADING
4000  PRINT
4010  INPUT "ENTER RADIUS OF CIRCLE".  RD
4020  PRINT "THE CIRCLE WILL BE CENTERED ON THE SCREEN"
4030  PRINT "ENTER THE ILLUMINATION CONSTANT"
4040  INPUT "(.05 TO .9)";I
4050  PRINT "PRESS ANY KEY TO DRAW CIRCLE"
4060  PRINT "WHEN CIRCLE HAS BEEN DRWN, PRESS KEY"
4065  GET A$:IF A$="" THEN 4065
4070  SYS 50195:REM TURN ON BIT MAP
4080  LN=2*RD:REM CALCULATE LENGTH OF DIAMETER
4090  N=LN*I:REM NUMBER OF LINES TO BE DRAWN
4100  DX=2*RD/N:REM DISTANCE BETWEEN LINES OF DOTS
4120  X=RD+DX:REM STARTING POINT
4130  REM LOOP BACK FOR REPEAT
4140  X=X-DX
4150  T=RD*RD-X*X:
4160  IF T<0 THEN DTS=1
4170  Y=SQR(T):REM HEIGHT OF LINE, USING CIRCLE FORMULA
4180  D=2*Y:REM ABOVE AND BELOW AXIS
4190  DTS=D*I:REM NUMBER OF DOTS TO BE DRAWN
4200  IF DTS=0 THEN DTS=1
4210  DY=2*Y/DTS:REM DISTANCE BETWEEN DOTS
4220  FOR K=1 TO DTS
4230  J=-Y+K*DY
4240  X1=INT(X+159):J=INT(J+100):
      REM ADJUST TO CENTER OF SCREEN
4250  SYS 50201, X1, J, 1
4260  NEXT K
4270  IF X>-RD THEN GOTO 4140
4280  RETURN
```

These routines can be called from a main module program, which has the following features:

```
100  GOSUB 1000:REM SET UP TRIG TABLE
110  SYS 50207:REM CLEAR BIT MAP
120  SYS 50207, 0, 1:REM COLOR CODE
130  REM ENTER 1 FOR LINE SEGMENTS,
         2 FOR DOTS, 3 FOR SHADING
140  INPUT C%
150  ON C% GOSUB 2000, 3000, 4000
160  GET A$:IF A$="" THEN 160
170  GOTO 140
```

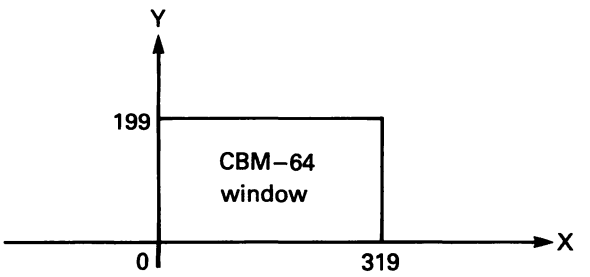
SHAPE AS AN ARRAY OF POINTS

Line drawings can be recorded as an array of points. For example, a square has 4 points joined by straight lines.

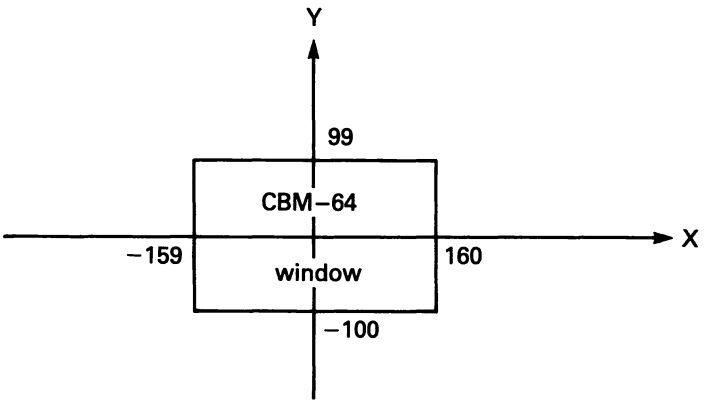


The array A is known as a matrix. Using matrix arithmetic it is possible to represent many transformations in the plane.

Before looking at transformations, let us construct a shape drawing routine. The next program allows us to enter a shape, which is then drawn. At this point we should recall the format of the C-64 screen. The default coordinate system allows us to draw only part of the full X-Y plane, thus:



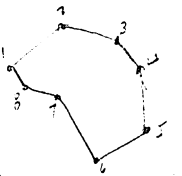
We can adjust our coordinates so that the center of the X-Y plane is approximately in the center of the screen. In this case the C-64 will show only the following window:



Thus if any part of our shape goes outside this window, do not expect to see it!!!

If we wish to see more of the X-Y plane, then we must scale our data.

SHAPE GRABBING ROUTINE



```
150  SYS 50207:REM CLEAR BIT MAP
160  SYS 50210, 0, 1:REM COLOR BIT MAP
170  PRINT"SHAPE DRAWING ROUTINE"
180  PRINT"ENTER NUMBER OF POINTS IN SHAPE"
190  INPUT N
200  DIM S(2,N)
210  PRINT "ENTER THE COORDINATES OF THE POINTS"
220  PRINT "IN THE SHAPE"
230  FOR I=1 TO N
240  PRINT "POINT";I;"X, Y";
250  INPUT S(1,I), S(2,I)
260  NEXT I
270  PRINT "THE POINTS ARE:"
280  L=1
290  FOR I=1 TO N
300  PRINT S(1,I), S(2,I)
310  L=L+1
320  IF L<20 THEN GOTO 360
330  PRINT "PRESS ANY KEY TO CONTINUE"
```

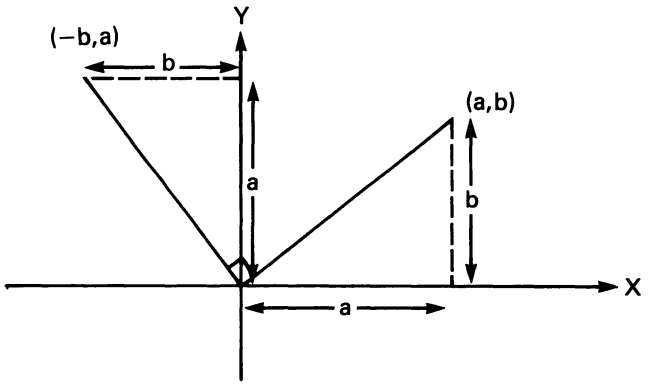
```
340  GET A$:IF A$=""THEN GOTO 340
350  L=0
360  NEXT I
370  PRINT "PRESS ANY KEY TO SEE DRAWING"
380  PRINT "ANOTHER KEY TO RELEASE DRAWING"
390  GET A$:IF A$="" THEN GOTO 390
400  SYS 50207:REM CLEAR BIT MAP
410  SYS 50210,1,0:REM COLOR BIT MAP
420  SYS 50195:REM TURN ON BIT MAP
430  X=INT(S(1,1)):Y=INT(S(2,1))
440  SYS 50201,X,Y,1:REM PLOT POINT
450  FOR I=2 TO N
460  X=INT(S(1,I)):Y=INT(S(2,I))
470  SYS 50204,X,Y,1:REM DRAW TO POINT x, Y
480  NEXT I
490  X=INT(S(1,1)):Y=INT(S(2,1))
500  SYS 50204,X,Y,1:REM DRAW TO FIRST POINT
510  GET A$:IF A$="" THEN GOTO 510
520  SYS 50198
530  END
```

NOTES

1. The order of the drawing is important when using this routine.
2. Another way of constructing drawings is to store the end points of all straight lines. In this routine order would not matter.

ROTATION

Rotation of P1/2 Radians about Origin



A rotation of PI/2 radians about the origin maps any point (a,b) to the corresponding point (-b,a).

$$(a,b) \rightarrow (-b,a)$$

If we write the point (a,b) as a vector b then we have.

$$\begin{pmatrix} a \\ b \end{pmatrix} \rightarrow \begin{pmatrix} -b \\ a \end{pmatrix}$$

Mapping is achieved by the following matrix multiplication:

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \rightarrow \begin{pmatrix} -b \\ a \end{pmatrix}$$

Thus if we have a shape held as an array or matrix we can compute the new coordinates by simply performing a matrix multiplication.

```
  :
  :REM enter shape coordinates
  :
  :
DIM newshape (2,N)
Call routine to draw old shape
Call routine to compute new shape
The following routine swaps old
    and new shapes
FOR I=1 TO N
   S(1,I)=newshape(1,I)
   S(2,I)=newshape(2,I)
NEXT I
Call routine to draw new shape
END

Routine to calculate new shape
REM old shape held in S(2,N)
REM new shape to be held in newshape (2,N)
FOR I=1 TO N
   newshape (1,I) = -S(2,I)
   newshape (2,I) = S(1,I)
NEXT I
RETURN
```

QUIZ 6.2

Implement this routine in BASIC V2.

Use the routine developed to rotate a triangle by PI/2 about the origin.

OTHER TRANSFORMATIONS

Other useful transformations are given in the following table.

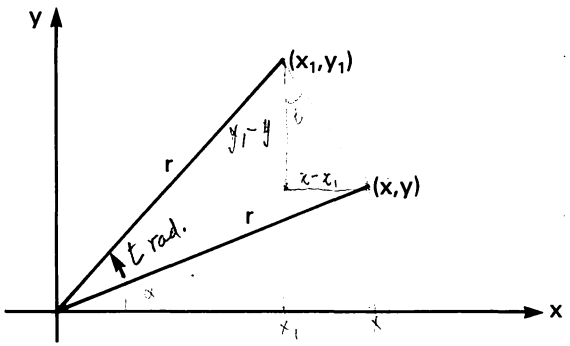
Transformation	Matrix
Identity	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
Reflection in y=x	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Half turn about 0	$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$
Reflection in x-axis	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
Rotation about 0 of PI/2 rad	$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$
Reflection in y-axis	$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$
Rotation about 0 of -PI/2 rad	$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$

QUIZ 6.3

Change the program in Quiz 6.2 to try out the above transformations.

THE GENERAL TRANSFORMATION OF ROTATION

Let us consider a rotation of t radians about the origin.



From the above diagram we can show that:

```
X1=X*COS(t)-Y*SIN(t)
Y1=X*SIN(t)+Y*COS(t)
```

thus we can write

$$\begin{pmatrix} X1 \\ Y1 \end{pmatrix} = \begin{pmatrix} COS(t) & -SIN(t) \\ SIN(t) & COS(t) \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix}$$

We can now amend our rotation routine to produce a general purpose rotation routine.

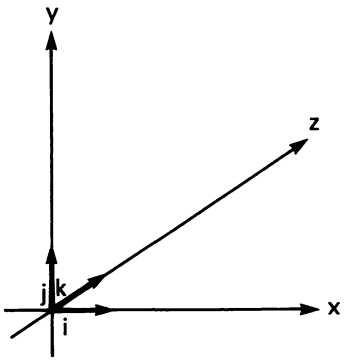
```
2000  REM old shape in S(2,N)
      REM new shape in newshape (2,N)
      REM angle of rotation theta
      sint = SIN(theta): cost = COS(theta)
      For I = 1 to N
        newshape (1,I)=cost*S(1,I)-sint*S(2,I)
        newshape (2,I) =sint*S(1,I)+cost*S(2,I)
      NEXT I
      RETURN
```

QUIZ 6.4

Write a program to allow the user to enter a shape and rotate it about the origin.

3-D ROTATION ABOUT Y-AXIS

The next diagram shows the frame of reference for drawing objects in three dimensions.

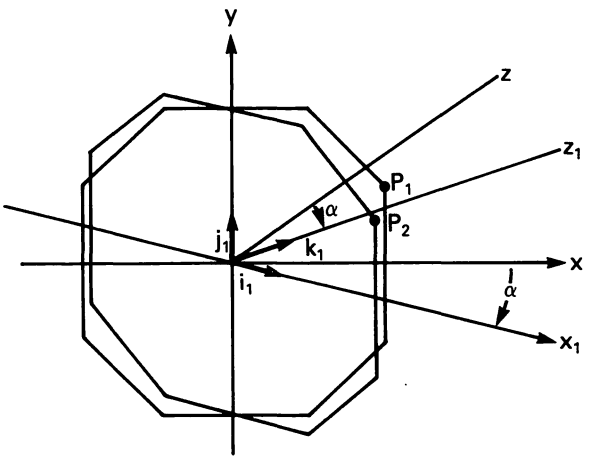


x y form plane of screen.
z is depth.
i, j, k are unit vectors in the x, y, z directions respectively.

What we plan to do in this section is to rotate a 2-D shape in the plane of the screen. As before, the shape will be held in a matrix, which will then be multiplied by an appropriate rotation matrix and the new shape drawn.

The shape chosen is a regular polygon. (Note that in the limit the polygon becomes a circle.)

The polygon is centered at x0,y0,z0 and has NS sides. The geometry of the drawing is as follows:



NOTES

P1 is point before rotation.
P2 is point after rotation.
x1,y1, z1 are the rotated axes.
i1,j1, k1 are the new unit vectors.

The origin of the drawing is the point x0, y0, z0.

P1 is the point xr, yr, zr.

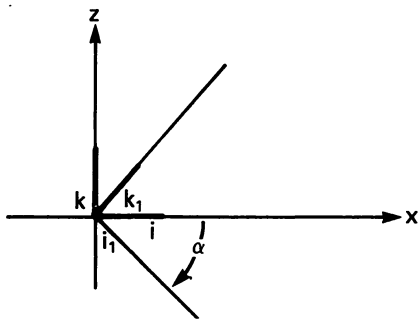
```
P1=(xr)*i + (xr)*j + (xr)*k
```

In terms of the new unit vectors,then:

```
P2=(xr)*i1 + (xr)*j1 + (xr)*k1
```

all we have to find now are the new unit vectors.

Consider the following diagram:



Neither i1 nor k1 have a component in the y direction (j component=0)

x component of i_1=i*cos α
z component of i_1k*(−sin α)

$$i_1 = \cos\alpha * i + 0 * j + (-\sin\alpha) * k$$

x component of k_1=(sin α)*i
z component of k_1=(cos α)*k

$$k_1 = \sin\alpha * i + 0 * j + \cos\alpha * k$$
$$j_1 = j$$

P2 then becomes the point:

$$P2=xr*[\cos\alpha\ .i-\sin\alpha\ .k]+yr*[j]+zr*[\sin\alpha\ .i+\cos\alpha\ .k]$$

We can now multiply out the expression for P2 and examine the coefficient of i, j, and k.

$$P2 = i(xr\ \cos\alpha + zr\ \sin\alpha)+j(yr)+k(-xr\ \sin\alpha + zr\ \cos\alpha)$$

$$XP = xr*(\cos\alpha)+yr*(0)+zr*(\sin\alpha)$$
$$YP = xr*(0)\ \ +yr*(1)+zr*(0)$$
$$ZP = xr*(-\sin\alpha+yr*(0)+zr*(\cos\alpha)$$

In matrix notation, we then have

$$[XP\ YP\ ZP]=[xr\ yr\ zr]\begin{bmatrix}\cos\alpha & 0 & -\sin\alpha\\ 0 & 1 & 0\\ \sin\alpha & 0 & \cos\alpha\end{bmatrix}$$

QUIZ 6.5

Construct a program to implement the above algorithm. Note that you might have to re-center the origin.

SOLUTION OUTLINE

The object of this program is to rotate a two dimensional shape in the plane of the screen.

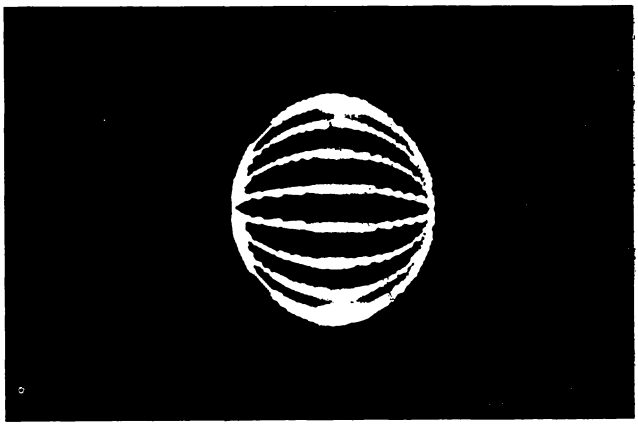
```
10 REM PROGRAM - 3D ROTATION
240 DIM S(3,50),NS(3,50),C(3,3)
250 INPUT "ENTER POLYGON RADIUS";R
260 INPUT "ENTER NUMBER OF SIDES";N
270 INPUT "ENTER Y-AXIS ROTATION (DEGS)";B
280 B=B*[PI]/180
290 SYS 50207:REM CLEAR BIT MAP
300 SYS 50210,1,0:REM COLOR BIT MAP
310 TH=B
320 GOSUB 1000:REM ROTATION-X MATRIX CALCULATION
330 GOSUB 2000:REM SHAPE ROUTINE
340 SYS 50195:REM TURN ON BIT MAP
350 GOSUB 3000:REM DRAW SHAPE
360 FOR TH=B TO 2*[PI] STEP B
370 GOSUB 4000:REM CALCULATE NEW VIEW
380 GOSUB 3000:REM DRAW SHAPE
```

```
390 NEXT TH
400 END
990 REM X-ROTATION MATRIX
1000 C(1,1)=1
1010 C(1,2)=0
1020 C(1,3)=0
1030 C(2,1)=0
1040 C(2,2)=COS(TH)
1050 C(2,3)=SIN(TH)
1060 C(3,1)=0
1070 C(3,2)=-SIN(TH)
1080 C(3,3)=COS(TH)
1090 RETURN
1990 REM SHAPE SUBROUTINE
2000 DA=2*[PI]/N
2010 A=-DA
2020 FOR I=1 TO N
2030 A=A+DA
2040 S(1,I)=R*COS(A)
2050 S(2,I)=R*SIN(A)
2060 S(3,I)=0
2070 NEXT I
2080 RETURN
2990 REM DRAW SHAPE ROUTINE
3000 X=INT(S(1,1)+160):Y=INT(S(2,1)+100)
3010 SYS 50201,X,Y,1
3020 FOR I=2 TO N
3030 X=INT(S(1,I)+160):Y=INT(S(2,I)+100)
3040 SYS 50204,X,Y,1
3050 NEXT I
3060 X=INT(S(1,1)+160):Y=INT(S(2,1)+100)
3070 SYS 50204,X,Y,1
3080 RETURN
3990 REM NEW VIEW CALCULATION
4000 FOR I=1 TO N
4010 NS(1,I)=S(1,I)*C(1,1)+S(2,I)*C(2,1)+S(3,I)*C(3,1)
4020 NS(2,I)=S(1,I)*C(1,2)+S(2,I)*C(2,2)+S(3,I)*C(3,2)
4030 NS(3,I)=S(1,I)*C(1,3)+S(2,I)*C(2,3)+S(3,I)*C(3,3)
4040 NEXT I
4050 FOR J=1 TO N
4060 FOR K=1 TO 3
4070 S(K,J)=NS(K,J)
4080 NEXT K
4090 NEXT J
4100 RETURN
```

The above program is from our companion book 100 PROGRAMS FOR

THE COMMODORE 64 (Prentice/Hall International 1984). All the solutions to the Quizzes in this chapter can be found there.

The following photograph is taken from the above program.



3-D ROTATION ABOUT THE X-AXIS

If we perform the same analysis as before for rotation about the x-axis, say an angle t, then the rotation matrix is found to be:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(t) & \sin(t) \\ 0 & -\sin(t) & \cos(t) \end{bmatrix}$$

Notice that the structure of this matrix is very similar to that for y-axis rotation.

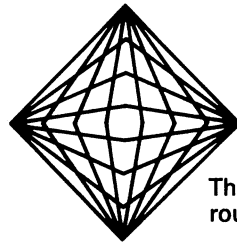
QUIZ 6.6

Amend the last program to show rotation about the x-axis.

SOLID DRAWING

We can now rotate a line drawing round both the X and Y axes. Using these capabilities we can draw regular polyedra.

QUIZ 6.7



This is a square rotated round both axes.

Construct a program to draw the above shape. The program would be constructed as follows:

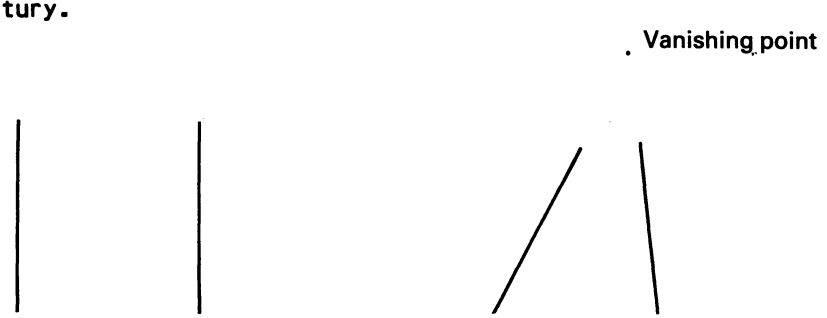
1. Decide how many views are to be shown (say 6 in each direction)
2. Using y-rotation program, plot the 6 views using an angle of rotation of 20 degrees.
3. Using x-rotation program, plot the 6 views using an angle of rotation of 20 degrees.

SOLUTION OUTLINE

```
REM SOLID DRAWING
SET UP GRAPHICS
     DIM XR(50),YR(50),ZR(50)
     DIM XP(50),YP(50),ZP(50)
     DIM C(3,3):REM ROTATION MATRIX
     GOSUB CIRCLE:REM SET UP ARRAYS OF XR,YR,ZR
     DALPHA = 20*PI/180
     ALPHA = - DALPHA
     FOR J=1 TO 5
        ALPHA=ALPHA+DALPHA
        GOSUB ROTATION Y:REM ROTATE IN y DIRECTION
        GOSUB NEWVIEW:REMOVE COORDS TO SHAPE DRAWING
        GOSUB DRAW:REM DRAW SHAPE
     NEXT J
     DBETTA= 20*PI/180
     BETA= -DBETA
     FOR J=1 TO 5
        BETA= BETA+DBETA
        GOSUB ROTATION.X
        GOSUB NEWVIEW
        GOSUB DRAW
     NEXT J
```

PERSPECTIVE

Most people will remember from school days about perspective. The method we remember is to locate a vanishing point. All parallel lines should converge to that point. This method resulted from a technique developed in Italy in the fifteenth century.



The technique is to place a wire mesh between the artist and his subject, and a corresponding sheet of graph paper on the drawing surface. A pin before the artist served to give a fixed viewing point. A drawing of the subject was made with reference to the mesh grid. See Fig 1 and Fig 2.

FIG 1

To quote Leonardo 'Set a frame with a network of threads between your eye and the nude model you are drawing, and draw

these same squares on the paper. Place a spot on the net to serve as a fixed point'.

The above drawing (Fig 1) is Durers' drawing of a reclining nude, shown in J.Bronowski's "The Ascent of Man", BBC Publications, 1973.

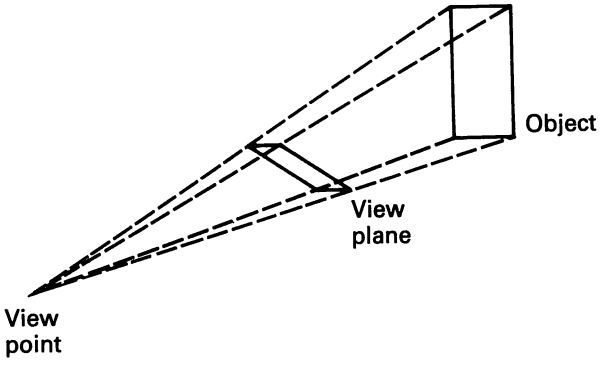


FIG 2

SOME MATHEMATICS

From Fig 2 it is seen that we have to find the intersection of a line drawn between a point on the object and the view point and the image plane. Normally, we consider the view screen to be the Z=0 plane, but for the sake of simplicity let us translate our coordinate system to be centered at the view point and let us find the projection of a point on the plane Z=K.

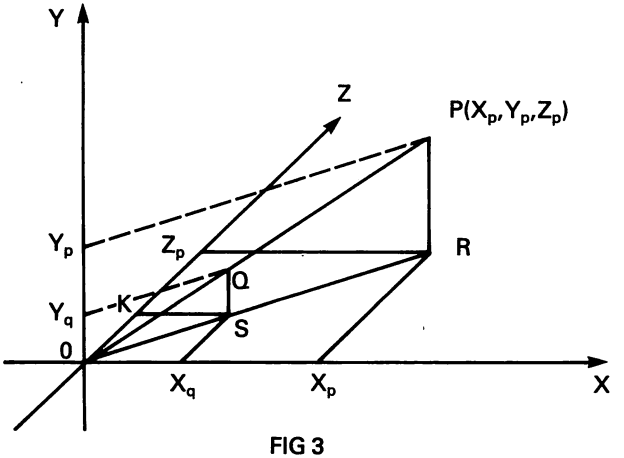


FIG 3

R is the projection of P in X,Z plane, angles <OSQ and <ORP are right angles. K is known, Xp, Yp and Zp are also known. We thus have to find Xq and Yq.

It can be shown using simple geometry that:

$$XQ = \frac{K}{ZP} * XP$$

$$YQ = \frac{K}{ZP} * YP$$

$$ZQ = K$$

Thus the point (XP,YP,ZP) intersects the view plane at the point:

$$\frac{K}{ZP}.XP, \frac{K}{ZP}.YP, K$$

If we perform this transformation on all points of a 3D shape, then we will have a perspective view of the object.

PERSPECTIVE ALGORITHM

Normally,we do not wish to view from (0,0,0) and with an image plane at Z=K. We will normally have an arbitrary view point (Vx,Vy,Vz) and use the plane Z=0 as the image plane.

The algorithm to find the coordinates of an image point is then:

1. Rewrite the coordinates of the point with respect to the view point

   (X,Y,Z) --> (X-Vx,Y-Vy,Z-Vz)
   = (Px,Py,Pz)

2. Calculate the coordinates of the projection in the plane Z=-Vz

   $$QX = -\frac{VZ}{PZ}.PX$$

   $$QY = -\frac{VZ}{PZ}.PY$$

   $$QZ = -VZ$$

3. Rewrite the coordinates with respect to the old coordinates

$$(QX, QY, QZ) \rightarrow (QX + VX, QY + VY, QZ + VZ)$$
$$= \left(-\frac{VZ}{PZ}.PX + VX, -\frac{VZ}{PZ}.PY + VY, 0\right)$$

Draw the shape with the new coordinates.

QUIZ 6.8

Implement the above algorithm using the following routine to draw a house as the shape.

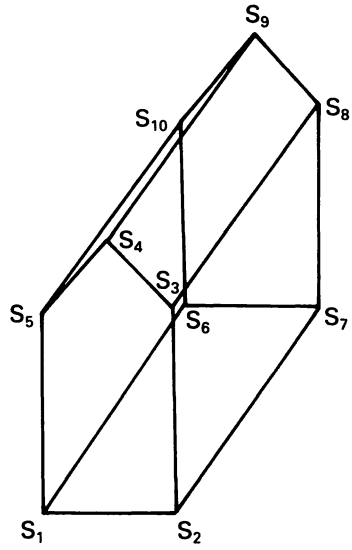
ROUTINE TO DRAW HOUSE

```
REM HOUSE PROCEDURE

     REM front of house

     MOVE S1
     DRAW S2
     DRAW S3
     DRAW S4
     DRAW S5
     DRAW S1

REM BACK OF HOUSE

     MOVE S6
     DRAW S7
     DRAW S8
     DRAW S9
     DRAW S10
     DRAW S6

REM CONNECTING LINES

     MOVE S6
     DRAW S1
     MOVE S10
     DRAW S5
     MOVE S9
     DRAW S4
```

Possible Co-ordinates

```
S1=0,0,-100
S2=50,0,-100
S3=50,40,-100
S4=25,25,-100
S5=0,40,-100
S6=0,0,-25
S7=50,0,-25
S8=50,40,-25
S9=25,500,-25
S10=0,40,-25
```

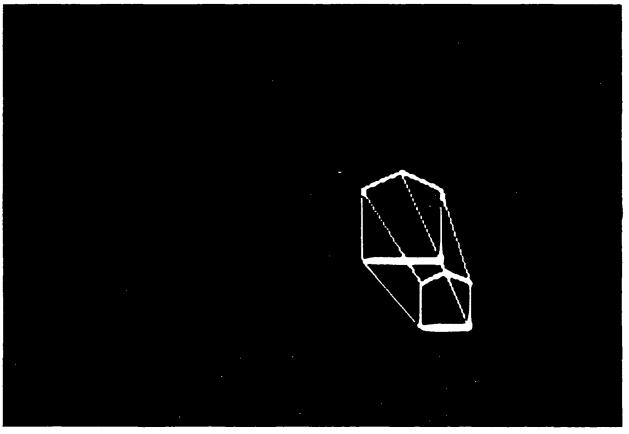
```
      MOVE S8
      DRAW S3
      MOVE S7
      DRAW S2
RETURN

REM PERSPECTIVE ROUTINE

      PX = X-VX
      PY = Y VY
      PZ = Z-VZ
       R = -VZ/PZ
      QX = R*PX+VX
      QY = R*PY+VY
RETURN


SOLUTION
```




```
      10 REM PROGRAM - PERSPECTIVE
      170 DIM H(10,3),S(10,3),P(10,3)
      180 FOR I=1 TO 10
      190 FOR J=1 TO 3
      200 READ H(I,J):S(I,J)=H(I,J)
      210 NEXT J
      220 NEXT I
      230 REM DATA FOR HOUSE
      240 DATA 0,0,-100,50,0,-100,50,-40,-100,25,-50
      250 DATA -100,0,-40,-100,0,0,-25,50 ,0,-25,50
      260 DATA -40,-25,25,-50 ,-25,0,-40,-25
      270 REM END OF DATA
      280 SYS 50207:REM CLEAR BIT MAP
      290 SYS 50210,1,0:REM COLOR BIT MAP
      300 GOSUB 1000:REM CLEAR MAP AND DRAW HOUSE
      310 REM LOOP BACK POINT
      320 GET A$:IF A$="" THEN GOTO 320
      330 SYS 50198:REMBIT MAP OFF
      340 INPUT "VIEW POINT (X,Y,Z)";VX,VY,VZ
      345 IF VZ=-100 THEN PRINT "CANNOT SEE SHAPE FROM INSIDE
          WALL":GOTO 340
      350 REM PERSPECTIVE ROUTINE
      360 FOR I=1 TO 10
      370 PX=S(I,1)-VX
      380 PY=S(I,2)-VY
      390 PZ=S(I,3)-VZ
      400 R=-VZ/PZ
      410 QX=R*PX+VX
```

```
420 QY=R*PY+VY
430 P(I,1)=QX
440 P(I,2)=QY
450 P(I,3)=0
460 NEXT I
470 FOR I=1 TO 10
480 FOR J=1 TO 3
490 H(I,J)=P(I,J)
500 NEXT J
510 NEXT I
520 GOSUB 1000:REM CLEAR BIT MAP AND DRAW HOUSE
530 GOTO 310
990 REM CLEAR BIT MAP AND DRAW HOUSE
1000 SYS 50207
1010 SYS 50195
1020 K=1:GOSUB 2000
1030 SYS 50201,X,Y,1
1040 FOR J=2 TO 5
1050 K=J:GOSUB 2000
1060 SYS 50204,X,Y,1
1070 NEXT J
1080 K=1:GOSUB 2000
1090 SYS 50204,X,Y,1
1100 REM NEXT THE BACK OF THE HOUSE
1110 K=6:GOSUB 2000
1120 SYS 50201,X,Y,1
1130 FOR J=7 TO 10
1140 K=J:GOSUB 2000
1150 SYS 50204,X,Y,1
1160 NEXT J
1170 K=6:GOSUB 2000
1180 SYS 50204,X,Y,1
1190 REM NEXT JOIN THE BACK TO FRONT
1200 K=6:GOSUB 2000
1210 SYS 50201,X,Y,1
1220 K=1:GOSUB 2000
1230 SYS 50204,X,Y,1
1240 K=10:GOSUB 2000
1250 SYS 50201,X,Y,1
1260 K=5:GOSUB 2000
1270 SYS 50204,X,Y,1
1280 K=9:GOSUB 2000
1290 SYS 50201,X,Y,1
1300 K=4:GOSUB 2000
1310 SYS 50204,X,Y,1
1320 K=8:GOSUB 2000
1330 SYS 50201,X,Y,1
```

```
1340 K=3:GOSUB 2000
1350 SYS 50204,X,Y,1
1360 K=7:GOSUB 2000
1370 SYS 50201,X,Y,1
1380 K=2:GOSUB 2000
1390 SYS 50204,X,Y,1
1400 RETURN
1990 REM CLIPPING ROUTINE
2000 X=INT(H(K,1)+160)
2010 IF X>319 THEN X=319
2020 IF X<0 THEN X=0
2030 Y=INT(H(K,2)+100)
2040 IF Y>199 THEN Y=199
2050 IF Y<0 THEN Y=0
2060 RETURN
```



ROTATING HOUSE

Using the techniques developed in the previous programs we present here a program which shows an object continuously rotating about the origin. The object chosen is a line drawing of a house.

```
10 REM PROGRAM - ROTATING HOUSE
60 DIM H(10,3),S(10,3),P(10,3)
70 FOR I=1 TO 10
80 FOR J=1 TO 3
90 READ H(I,J):S(I,J)=H(I,J)
```

```
100 NEXT J
110 NEXT I
120 REM DATA FOR HOUSE
130 DATA 0,0,-100,50,0,-100,50,-40,-100,25,-50
140 DATA -100,0,-40,-100,0,0,-25,50 ,0,-25,50
150 DATA -40,-25,25,-50 ,-25,0,-40,-25
160 REM END OF DATA
170 SYS 50207:REM CLEAR BIT MAP
180 SYS 50210,1,0:REM COLOR BIT MAP
190 SYS50195 :REM BIT MAP ON
200 GOSUB 1000:REM CLEAR MAP AND DRAW HOUSE
210 VX=500:VY=-500:VZ=800:REM VIEW POINT
220 GOSUB 3000 :REM ROTATION MATRIX
230 REM LOOP BACK POINT
240 DATA 0,0,-100,50,0,-100,50,-40,-100,25,-50
250 REM PERSPECTIVE ROUTINE
260 FOR I=1 TO 10
270 PX=S(I,1)-VX
280 PY=S(I,2)-VY
290 PZ=S(I,3)-VZ
300 R=-VZ/PZ
310 QX=R*PX+VX
320 QY=R*PY+VY
330 P(I,1)=QX
340 P(I,2)=QY
350 P(I,3)=0
360 NEXT I
370 FOR I=1 TO 10
380 FOR J=1 TO 3
390 H(I,J)=P(I,J)
400 NEXT J
410 NEXT I
420 GOSUB 1000:REM CLEAR BIT MAP AND DRAW HOUSE
430 GOSUB 4000 :REM ROTATE HOUSE
440 GOTO 230
450 END
900 READ H(I,J):S(I,J)=H(I,J)
990 REM CLEAR BIT MAP AND DRAW HOUSE
1000 REM CALLS SIMPLE CLIPPING SUBROUTINE
1010 SYS 50207
1020 K=1:GOSUB 2000
1030 SYS 50201,X,Y,1
1040 FOR J=2 TO 5
1050 K=J:GOSUB 2000
1060 SYS 50204,X,Y,1
1070 NEXT J
1080 K=1:GOSUB 2000
```

```
1090 SYS 50204,X,Y,1
1100 REM NEXT THE BACK OF THE HOUSE
1110 K=6:GOSUB 2000
1120 SYS 50201,X,Y,1
1130 FOR J=7 TO 10
1140 K=J:GOSUB 2000
1150 SYS 50204,X,Y,1
1160 NEXT J
1170 K=6:GOSUB 2000
1180 SYS 50204,X,Y,1
1190 REM NEXT JOIN THE BACK TO FRONT
1200 K=6:GOSUB 2000
1210 SYS 50201,X,Y,1
1220 K=1:GOSUB 2000
1230 SYS 50204,X,Y,1
1240 K=10:GOSUB 2000
1250 SYS 50201,X,Y,1
1260 K=5:GOSUB 2000
1270 SYS 50204,X,Y,1
1280 K=9:GOSUB 2000
1290 SYS 50201,X,Y,1
1300 K=4:GOSUB 2000
1310 SYS 50204,X,Y,1
1320 K=8:GOSUB 2000
1330 SYS 50201,X,Y,1
1340 K=3:GOSUB 2000
1350 SYS 50204,X,Y,1
1360 K=7:GOSUB 2000
1370 SYS 50201,X,Y,1
1380 K=2:GOSUB 2000
1390 SYS 50204,X,Y,1
1400 RETURN
1990 REM CLIPPING ROUTINE
2000 X=INT(H(K,1)+160)
2010 IF X>319 THEN X=319
2020 IF X<0 THEN X=0
2030 Y=INT(H(K,2)+100)
2040 IF Y>199 THEN Y=199
2050 IF Y<0 THEN Y=0
2060 RETURN
2990 REM ROTATION MATRIX
3000 C(1,1)=COS([PI]/10)
3010 C(1,2)=0
3020 C(1,3)=-SIN([PI]/10)
3030 C(2,1)=0
3040 C(2,2)=1
3050 C(2,3)=0
```

```
3060 C(3,1)=SIN([PI]/10)
3070 C(3,2)=0
3080 C(3,3)=COS([PI]/10)
3090 RETURN
3990 REM ROTATE HOUSE
4000 FOR I=1 TO 10
4010 T1=C(1,1)*S(I,1)+C(2,1)*S(I,2)+C(3,1)*S(I,3)
4020 T2=C(1,2)*S(I,1)+C(2,2)*S(I,2)+C(3,2)*S(I,3)
4030 T3=C(1,3)*S(I,1)+C(2,3)*S(I,2)+C(3,3)*S(I,3)
4040 S(I,1)=T1:S(I,2)=T2:S(I,3)=T3
4050 NEXT I
4060 RETURN
```

SUMMARY

In this chapter we looked at how the C-64 could be used to develop high resolution graphics routines. Routines were developed to show:

shading
translation
circle drawings
line drawings
matrix transformations
3-D transformations
perspective.

These routines were used to develop a program to show a rotating house always drawn in perspective.

CHAPTER 7

Data Processing in BASIC

INTRODUCTION

In this chapter we discuss the design of files of information held within a microcomputer based business system. Whenever a business decides to move its traditional files from a paper based system, the personnel involved in the change will be faced with a minefield of jargon. This jargon will appear fairly similar to that which they are used to, but the words will be used in a more rigorous fashion. The word 'file' itself becomes one of a hierarchy of data structures. The form of this hierarchy depends on the hardware as well as the actual data, since some forms of data organization cannot be implemented on some hardware configurations.

We also discuss data structures, the types of processing which can be carried out on these data structures and the hardware is necessary to implement these structures. Finally we discuss how to define and specify files.

DATA STRUCTURES

In a microcomputer based filing system, there is a hierarchy of data types. At the very bottom of this hierarchy is the data item or elementary field. This is the smallest piece of data which has a meaning to its user. For example, in a personnel file an employee's first name will have a meaning, but the individual characters which make up that name do not have a meaning. In this case we would say that "first-name" is a data item. Of course, if we were not really interested in the data item first-name, we could have decided that name was a data item, with first-name only being a part.

Groups of data items can be placed together to form group items. For example, the group item Address could be made up of street, town, region, zip-code, each of which could be considered to be a data item in its own right. Of course to some users the group item Address could be considered to be simply a string of characters, and thus be used as an elementary data item. We can see therefore that even at this stage in the hierarchy, different users will see data in

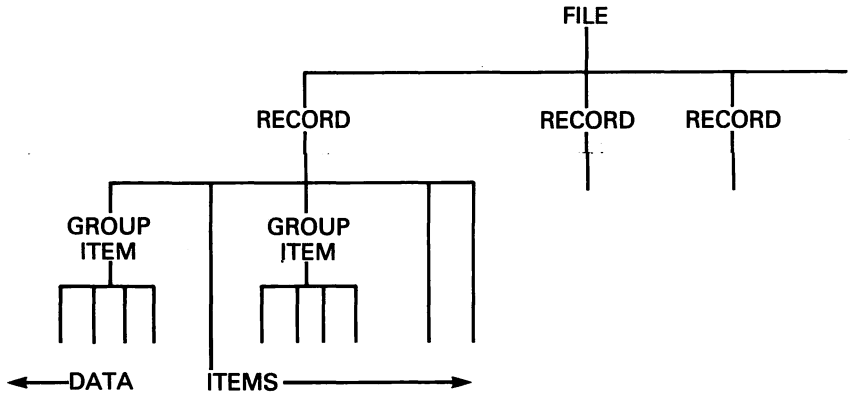
different ways, this problem must be bourne in mind at all times by the system designer.

At the next stage up in the hierarchy we have the record, this has a similar meaning to the traditional paper based idea. A record is a collection of group or data items which are logically related to some specified identifier. For example in a personnel system, we could have personnel records which consist of all data items which relate to individual employees. Another example could be stock records which would group together all data items referring to particular items of stock. Here again we see that idea of a record is user or application oriented. It is a logical relationship between data items.

Moving up the hierarchy, we have the idea of a file. This can be losely defined as a collection of records, which are grouped together for a particular application. For example, a personnel file consists of a set of personnel records; a stock file is made up of a collection of stock records. Thus in a microcomputer based system a file will be designed with a particular application in view. This can lead to problems when we wish to use the information held in a file for an application for which it was not designed.

Taking note of the problems involved in files, there has been a major effort in designing DATABASE SYSTEMS. A database is a collection of interrelated data stored together to serve more than one application. The data are stored to make them independent of the application program, thus if another user wants to use data already stored in the system this can be done without problems. We will be covering a commercial database system in Chapter 8, but we note here that true database systems running on microcomputers are few and far between.

Thus we have the following information tree:



DATA STORAGE

As well as defining the logical meaning of our data structures, we must also consider what they are like physically, and how we are going to store this data.

Since data items are the basis of our information, we must be able to determine when one item ends and another starts. For example we can distinguish the boundaries between the data items (words) which make up this text. In this case the data items end with either a space or a punctuation mark. If all the words were of the same length then we would have been able simply to count the characters which make up these words. For example the following text is made up of three letter words:

onecanaskhimwhy

It is possible to interpret this. It would have slightly more difficult if we did not know the length of each data item. For example, the following set of numeric data items are each of different lengths:

36212974392114225

It is impossible to determine where one item ends and another begins.

Thus if our data items are of fixed lengths then we do not need to record a separating character along with the data.

On the other hand if the data are of variable length then we have to record some means of specifying end of item. This is done by adopting an item separator. We decide on special character which we use to separate our items. For example:

621,97,3921,4,25

Here we have used the comma as an item separator. This is a fairly common choice in microcomputers using the BASIC language.

The system designer will normally have to take into account the peculiarities of a particular system when deciding how to separate fields. In BASIC, for example, variable length data items can be handled extremely easily using commas. With COBOL, on the other hand, it is easier to have the item

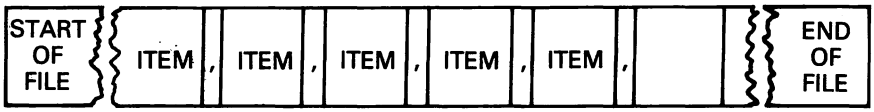
length specified. In Commodore-64 BASIC we can use comma, semicolon, colon or carriage return.

FILE PROCESSING CONCEPTS

There are basically two main divisions in file processing. Records can be processed one after the other in sequence, or the file device can be used to process the records in a random fashion. These two types of processing are known as sequential (serial) and random processing respectively. Some microcomputer systems allow only sequential processing, and others only allow a restricted version of random processing. The Commodore 64 allows for both sequential and random processing.

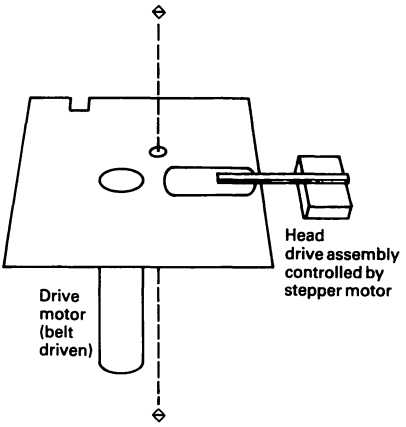
SEQUENTIAL OR SERIAL PROCESSING

This is the traditional method of file processing where the main file storage medium is magnetic tape. Using magnetic tape we can read records only one block at a time; it is impossible to jump about in the tape. Therefore records are accessed in the order in which they are stored. The structure of a file can then be described in the following diagram:



RANDOM PROCESSING

This is a disk based technique which recognizes that, as the computer system has the ability to control the position of the record on the disk, it is possible to move the read/write head to the correct position on the disk thus reading or writing a particular record or block of records. Consider, for example, the operation of a mini floppy disk:



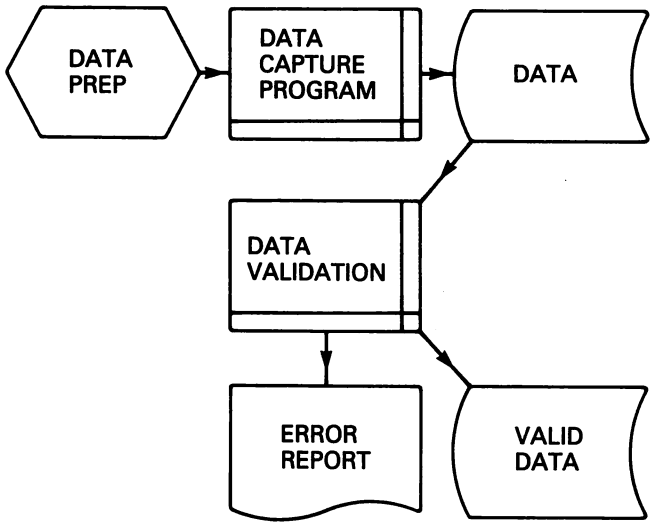
It can be seen from the above diagram that if we can move the heads about under software control, we can access the disk in a random fashion.

These then are the two main types of file processing, but what kinds of processing are carried out using these systems?
We can now consider a few examples of common data processing applications.

INPUT AND DATA VALIDATION

When using a microcomputer system, the data to be manipulated has first of all to be INPUT to the system. The method of input will normally be via the keyboard using a data capture programs to place the data on to a disk or tape file. The purpose of the data validation is to locate and detect, as far as is possible, errors in data presented to the microcomputer system. Note that validation does not ensure that the data is 100% correct, but only that errors have been minimized.

A typical validation run would have the following run diagram.



The data capture and data validation can be contained in the same program, thus releasing a data file.

The types of error checks which can be carried out within a validation run would be:

1. Format Check: this type of check ensures that the data item entered is composed of the correct characters. Thus we can ensure that, for example, there are numeric characters in a numeric data item.
2. Range checks: this type of check ensures that the data entered are within a reasonable range.
3. Check Digit Verification: it can sometimes be of use to append an extra check digit at the end of an important number. For example, we could use a single digit checksum, thus to the number 54321 we would add the digit 6 (5+4+3+2+1=15, 1+5=6), giving the new number 123456
4. Consistency Check: a consistency check would ensure that the data were consistent.
5. Hash Totals: it can happen that we have a particularly important field, such as account number. If this is the case then we could total this field over the whole file to give a number which has no intrinsic meaning. But we could use this number to ensure that reports are correct or that field values have been correctly input.

There are two ways of handling data errors, both of which provide the user with some sort of error report, and both of which handle error records.

Traditionally, transactions entered are collected into a file for data validation, and any records failing any of the data checks are ignored for this processing run, and are recorded on an error report. This printed error report is returned to the human subsystem for investigation and correction.

Using this traditional technique data can be resubmitted by one of the following methods:

- Hold data over until input run.
- Replace corrected data and rerun data vet.
- Add corrected data to end of original output file.

Another approach is to stop data entry when an error is recognized and force the user to re-enter the data immediately.

DATA FILE MAINTENANCE

Once we have a data file set up on either disk or tape, we have to have the capability of maintaining our information. We have to be able to

1. Change the contents of a data record.
2. Delete a data record.
3. Add a data record.

We also have to be able to sort a sequential file into order, to keep processing to a minimum. We will now discuss each of these options in turn.

1. Change the contents of a data record.

If the file was sequential, and on tape, we would have to

- Read file into memory.
- Find correct record.
- Change it.
- Write file back to tape.

If the file was sequential, and on disk, we could

- Open file for input.
- Open new file for output.
- Read record from input.
- If not required record then, correct and write to output.
- Continue reading and writing until we reach the end of the input file.
- Close both files.

2. Delete a data record

Again we have to have two distinct routines depending on whether we are using tape or disk. For a sequential file on disk, we read from one file to another, and simply don't write the deleted record.

3. Add a data record

Similarily, we have to have two distinct routines for tape and disk. In the case of a disk file we would read from one file to another and insert the new record in the correct place in the new file. If we take this approach of inserting a record then we do not need to sort files.

REPORT PROGRAMS

Once data is in the system and is being maintained, we need to write programs to provide users with reports on their data.

Generally speaking, report programs will process records sequentially, extracting and manipulating the data as required.

COMMODORE 64 FILE COMMANDS IN BASIC (Sequential Files)

In order to process data held in a sequential file, we have to able to "open" it. This is done by using the OPEN statement. There are two formats of the OPEN statement depending on whether our file is on tape or disk.

In the tape case, the format of the OPEN statement is:

OPEN file#, 1, numb, filename

The file# is used to identify the file to which we are PRINTing or from which we are INPUTing.

The parameter numb has the following meanings

If numb=0, file is OPEN for INPUT.

If numb=1, file is OPEN for OUTPUT.

If numb=2, file is OPEN for OUTPUT and an "end-of-tape" marker is placed at the end of the file. This prevents a program accidentally reading past the end of the data.

Example OPEN 1,1,2,"TEST"

This opens a sequential file on tape with an EOT marker placed at the end.

In the disk case, the format of the OPEN statement is:

OPEN file#, device#, channel#,"0: name,SEQ,direction"

1. File# is used to identify the file for I/O commands.
2. Device# is usually 8 for a single drive system, but if we have a second drive we can change its device number to another number (see chapter 9).
3. Channel# is a data channel, which can have any value from 2 to 14. Programmers normally keep both the device# and channel# the same.
4. Name is the name of our file.
5. Direction can be READ or WRITE.
6. The '0:' is an attempt at compatability with those Commodore systems with more than one drive built in.
7. The SEQ is short for SEQUENTIAL and it declares the file type. Other file types are PRG - program, USR - user and REL - relative.

EXAMPLES OF OPEN STATEMENT

1. OPEN 4,8,4,"0:TEST,S,W"

Notice from the above example that we only need to write the

first character of SEQ and WRITE. The file is file number 4.

2. OPEN A,8,B,"0:"+A$+"R"

If we wish to overwrite information held on a file, the OPEN statement is of the form:

OPEN 2,8,2,"@0:FILE,S,W"

the @ sign tells the disk operating system to replace the file.

Once a file has been OPENed, we have to be able to PRINT data on to it and to INPUT data from it. This is done by means of the PRINT# and INPUT# statements. The formats of these statements are:

PRINT# filenumber, printdata

INPUT# filenumber, inputdata

The PRINT# command acts in exactly the same way as the PRINT statement, except that the output is redirected to the file OPENed with the same filenumber. All the PRINT formatting capabilities are present when writing to a sequential file.

MUITO IMPORTANTE

The INPUT# command inputs the data from the file with filenumber. The INPUT# assumes that a variable is finished when it reads a:

RETURN (CHR$(13))
COMMA (,)
SEMICOLON (;)
COLON (:)

When numeric data is written to a disk file it is terminated automatically with a cursor right character (->). This gives INPUT# enough information to separate numeric data.

Once we have finished with a file, we have to CLOSE it. To close a file with filenumber N, the command is

CLOSE N

EXAMPLE PROGRAMS

PROGRAM 1

This program writes some characters to a disk file.

```
10 OPEN 2,8,2,"0:DATA,S,W"
20 PRINT#2,"ABC","def","HIJ"
30 CLOSE 2
```

After CLOSEing the file the contents of "DATA" will be

```
Character!         1         2
Number    !12345678901234567890123456789
----------!-----------------------------
Value     !ABC       def        HIJ/@
```

In the above diagram '/' stands for CHR$(13), CARRIAGE RETURN, and '@' for EOF, the end of file marker. Notice the amount of wasted space in the file. Now, how can we write a program to read this back? Notice that we do not know where one item ends and another starts.

PROGRAM 2

This program writes fields to a disk file and separates them with a carriage return

```
10 OPEN 2,8,2,"0:DATA,S,W"
20 PRINT#2,"ABC"
30 PRINT#2, "def"
40 PRINT#2, "HIJ"
50 CLOSE 2
```

After CLOSEing the file, the contents are as follows

```
Character!         1         2
Number    !12345678901234567890123456789
----------!-----------------------------
Value     !ABC/def/HIJ/@
```

PROGRAM 3

We can read this data file by using the following program:

```
10 OPEN 2,8,C,"0:DATA,S,R"
20 INPUT#2,A$
```

```
30 INPUT#2,B$
40 INPUT#2,C$
50 PRINT A$,B$,C$
60 CLOSE 2
```

PROGRAM 4

The next program allows us to create and maintain a mailing list of up to 50 names and addresses. The program is cassette based, but could be transferred easily to disk.

The mailing list is held in the form of a sequential file on tape, but is read into memory for processing.

```
10 REM MAILING LIST SYSTEM
20 DIM NM$(50),AD$(50,4):REM NAME + 4 LINES OF ADDRESS
30 PRINT CHR$(147):REM CLEAR SCREEN
40 PRINT:PRINT:PRINT
50 PRINT "MAILING LIST SYSTEM"
70 PRINT:PRINT:PRINT
80 PRINT "1. CREATE MAILING LIST"
90 PRINT "2. UPDATE MAILING LIST"
100 PRINT "3. PRINT MAILING LIST"
110 PRINT "4. FINISH THE PROGRAM"
120 PRINT:PRINT:PRINT
130 INPUT "CHOOSE";K:IF K=4 THEN END
140 ON K GOSUB 1000,2000,3000
150 GOTO 30
999 REM MAILING LIST CREATION
1000 PRINT CHR$(147):REM CLEAR SCREEN
1010 PRINT:PRINT:PRINT
1020 PRINT "WHAT IS THE NAME OF THE MAILING LIST":INPUT L$
1030 INPUT "HOW MANY NAMES ARE THERE";N
1050 PRINT CHR$(147):REM CLEAR SCREEN
1060 PRINT "ENTER INFORMATION :"
1070 PRINT:PRINT:PRINT
1080 INPUT "NAME       :";NM$(I)
1090 INPUT "ADDRESS    :";AD$(I,1)
1100 INPUT "           :";AD$(I,2)
1110 INPUT "           :";AD$(I,3)
1120 INPUT "           :";AD$(I,4)
1130 NEXT I
1135 PRINT CHR$(147):REM CLEAR SCREEN
1140 PRINT:PRINT:PRINT:PRINT "ABOUT TO SAVE DATA"
1150 PRINT "PRESS ANY KEY WHEN READY"
1160 GET Z$:IF Z$="" THEN 1160
```

Sem leitor de disco não é necessário !

```
1170 GOSUB 4000
1180 RETURN
1999 REM UPDATE MAILING LIST
2000 PRINT CHR$(147):REM CLEAR SCREEN
2010 PRINT:PRINT:PRINT "THIS PROGRAM ALLOWS YOU TO"
2020 PRINT:PRINT:PRINT
2030 PRINT "1. ADD A NEW RECORD"
2040 PRINT "2. DELETE A RECORD"
2050 PRINT "3. CHANGE A RECORD"
2060 PRINT:PRINT:PRINT
2070 INPUT "CHOOSE";K
2080 ON K GOSUB 2200,2500,2700
2090 GOSUB 4000:REM WRITE A NEW DATA FILE
2100 RETURN
2199 REM ADD A RECORD
2200 GOSUB 5000:REM READ DATA FILE INTO MEMORY
2210 N=N+1
2220 PRINT CHR$(147):REM CLEAR SCREEN
2230 PRINT "ENTER NEW RECORD":PRINT:PRINT:PRINT
2240 PRINT:PRINT:PRINT
2250 INPUT "NAME       :";NM$(N)
2260 INPUT "ADDRESS    :";AD$(N,1)
2270 INPUT "           :";AD$(N,2)
2280 INPUT "           :";AD$(N,3)
2290 INPUT "           :";AD$(N,4)
2295 PRINT:PRINT:PRINT
2300 INPUT "ANOTHER (Y/N)";AN$
2310 IF AN$="Y" THEN 2210
2320 GOSUB 4000:REM WRITE DATA FILE TO TAPE
2330 RETURN
2499 REM DELETE A RECORD
2500 GOSUB 5000:REM READ DATA FILE INTO MEMORY
2510 PT=1:REM POINTER FOR NEXT LIVE RECORD TO BE WRITTEN
2520 K=1:REM POINTER FOR NEXT RECORD TO BE READ
2530 PRINT CHR$(147):REM CLEAR SCREEN
2540 PRINT NM$(K)
2550 FOR J=1 TO 4:PRINT AD$(K,J):NEXT J
2570 INPUT "DELETE (Y/N)";AN$
2580 IF AN$="Y" THEN K=K+1:GOTO 2620
2590 NM$(PT)=NM$(K)
2600 FOR J=1 TO 4:AD$(PT,J)=AD$(K,J):NEXT J
2610 PT=PT+1:K=K+1
2620 IF K<N+1 THEN 2530
2630 N=PT
2640 GOSUB 4000:REM WRITE FILE TO TAPE
2650 RETURN
2699 REM CHANGE A RECORD
```

```
2700 GOSUB 5000:REM READ FILE INTO MEMORY
2710 FOR I=1 TO N
2720 PRINT CHR$(147):REM CLEAR SCREEN
2730 PRINT NM$(I):FOR J=1 TO 4:PRINT AD$(I,J):NEXT J
2740 PRINT:PRINT:PRINT
2750 INPUT "CHANGE (Y/N) ";AN$
2760 IF AN$="N" THEN 2820
2770 PRINT CHR$(147):REM CLEAR SCREEN
2780 INPUT "NAME  ";NM$(I)
2790 FOR J=1 TO 4:PRINT "ADDRESS ";J;
     " ";:INPUT AD$(I,J):NEXT J
2800 NEXT I
2810 GOSUB 4000:REM WRITE FILE TO TAPE
2820 RETURN
2999 REM PRINT MAILING LIST
3000 GOSUB 5000:REM READ FILE INTO MEMORY
3010 PRINT CHR$(147):REM CLEAR SCREEN
3020 INPUT "HOW MANY LINES ON LABEL ";LB:IF LB<6 THEN 3020
3030 OPEN 2,4
3040 FOR I=1 TO N
3050 PRINT#2,NM$(I)
3060 FOR J=1 TO 4:PRINT#2,AD$(I,J):NEXT J
3070 FOR J=5 TO LB:PRINT#2:NEXT J
3080 NEXT I
3090 CLOSE 2
3100 RETURN
3999 REM WRITE FILE TO TAPE
4000 OPEN 1,1,1,L$
4010 PRINT#1,N
4020 FOR I=1 TO N
4030 PRINT#1,NM$(I)
4040 FOR J=1 TO 4:PRINT#1,AD$(I,J):NEXT J
4050 NEXT I
4060 CLOSE 1
4070 RETURN
4999 REM READ FILE INTO MEMORY
5000 PRINT CHR$(147):REM CLEAR SCREEN
5010 INPUT "WHAT IS THE NAME OF TAPE FILE";L$
5020 OPEN 3,1,0,L$
5030 INPUT#3,N
5040 FOR I=1 TO N
5050 INPUT#3,NM$(I)
5060 FOR J=1 TO 4:INPUT#3,AD$(I,J):NEXT J
5070 NEXT I
5080 CLOSE 3
5085 GET A$:IF A$="" THEN 5085
5090 RETURN
```

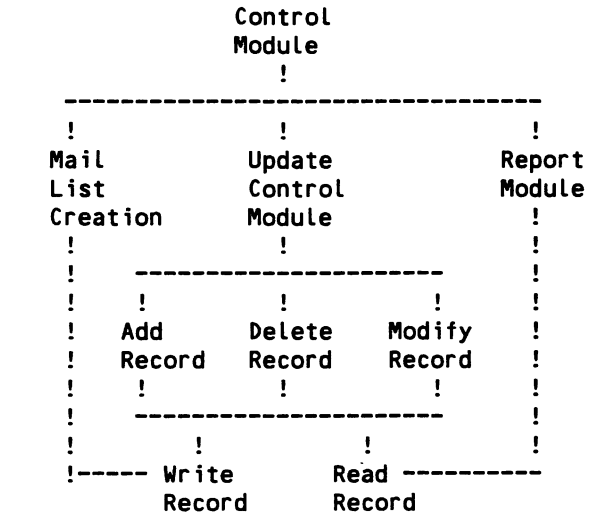
Porquê esta frase!

QUIZ 7.1

Amend the mailing list program to allow it to be disk based.

Hint: Only two subroutines need to be changed.

This is an example of a program with a modular design. The various modules in the program are related as follows.



It is sometimes useful to be able to extract information from a file a character at a time. This can be done by using the GET# command.

PROGRAM 5

The following program can be used to dump a file to either screen or printer. It is very useful for checking that a datafile has been set up correctly.

However, it can be useful on other occasions as well. The program should find a place in the library of anyone with a disk drive. The program can be amended to examine tape files as well.

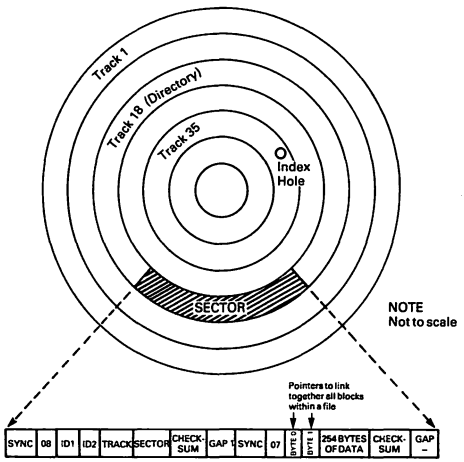
```
20 REM PROGRAM - FILE DUMP UTILITY
```

```
 40 PRINT CHR$(147)
 50 PRINT "FILE DUMP UTILITY"
 60 :
 70 PRINT:PRINT
 80 PRINT "THIS PROGRAM CAN BE USED TO OBTAIN A"
 90 PRINT "DUMP OF A DISK FILE TO SCREEN OR PRINTER"
100 PRINT "THE OPTIONS AVAILABLE ARE"
110 PRINT:PRINT
120 PRINT "1. DUMP A FILE TO SCREEN"
130 PRINT "2. DUMP A FILE TO PRINTER"
140 PRINT "3. FINISH THE PROGRAM"
150 PRINT
160 PRINT "ENTER NUMBER CORRESPONDING TO CHOICE":INPUT A
170 IF A<>1 AND A<>2 AND A<>3 THEN GOTO  160
180 IF A=3 THEN END
190 PRINT "ENTER THE NAME OF THE FILE TO BE"
200 INPUT "DUMPED";A$
205 INPUT "FILE TYPE(S,PORV)";T$
210 OPEN 5,8,5,"0:"+A$","+T$+",R"
215 IF A=1 THEN OPEN1,4
220 GET#5,A$
230 IF ST<>0 THEN PRINT:
    PRINT"SYSTEM STATUS IS "ST:GOTO260
240 IF A=1 THEN PRINT A$;
245 IF A=2 THEN PRINT#1,A$;
250 GOTO 220
260 IF A=2 THEN PRINT#1:CLOSE 1
270 CLOSE 5
280 PRINT "PRESS ANY KEY TO RETURN TO MENU"
290 GET A$:IF A$="" THEN GOTO 290
300 GOTO 10
310 END
```

RANDOM FILES

Random files on the Commodore disk drive can access any one of the 256 byte data block on the disk. Each disk, when formatted, has 664 blocks of data free.

The disk is organized as follows:



1. Each disk is formatted as a set of concentric tracks (35 in all).
2. Each track is organized into a set of sectors, with each sector containing one block of data and some operating system information.
3. Tracks contain from 17 to 21 sectors.
4. Track 18 holds the directory.

When we are processing random files, the format of the OPEN statement is slightly different from that of a sequential file. We have to OPEN two channels

- one for Disk Operating System (DOS) commands
- one for data

The format is then:

```
OPEN 15,8,15 for DOS commands
OPEN file number, device number, channel number, "#"
```

where "#" states that we are opening a data channel for random access. When data is moved between the disk and the computer, it is placed in a special area of RAM known as a "buffer". The OPEN statement allows us to select the buffer used as follows:

```
OPEN fno, devno, channo, "#bufferno"
```

To get data from the disk drive to the buffer, the program must execute a BLOCK-READ. The format is

```
PRINT#fnumb,"BLOCK-READ:" channel,drive,track,block
```

or

```
PRINT#fnumb,"B-R:" channel,drive,track,block
```

Once the data is in the buffer, we can use

```
INPUT#fnumb
```

or

```
GET#fnumb
```

to process the data as in a sequential file.

The equivalent command to write data to a block on the disk is the BLOCK-WRITE. The format is

```
PRINT#fnumb,"BLOCK-WRITE:"drive,channel,track,block
```

or

```
PRINT#fnumb,"B-W:drive,channel,track,block
```

To put data into the buffer, we would use

```
PRINT#fnumb
```

as in sequential processing.

DOS maintains a buffer pointer while data are being entered into the buffer. When the block is written, the value of that pointer is also written to disk. This buffer pointer can also be handled under program control. It would be very unusual for a disk file record to be exactly 256 bytes long. Thus if we had one record per block, we could have a large amount of wasted space.

To set the buffer pointer before reading from or writing to the buffer, the command is

```
PRINT#15, "B-P:" channel;location
```

EXAMPLE

```
OPEN 15,8,15
OPEN 6,8,6,"#
PRINT#15,"B-P:"6;10
```

This sets the buffer pointer to 10.

It would be rather silly to write blocks all over the disk without taking cognisance of other data on the disk. For example, using random files, we could overwrite programs or

data files. To get round this problem, DOS maintains a Block Allocation Map (BAM). So, before writing to a block, a program should check that the block is free, and if it is, then allocate it for use.

If the block is not available, then DOS will inform the program via the error channel of this fact and also the address of the next free block.

The format of the B-A is

```
PRINT#15, "B-A:"0,T,S
```

where 0 is drive number, T is track number and S is block (sector) number.

After this command, the program must check the error status

```
INPUT#15,A,B$,C,D
```

If the block is in use then:

```
      A = 65
      C is track
and   D is sector of next available block.
```

Once we have the address of the next available block, we can write to it.

To free blocks for further use we can use the BLOCK-FREE command:

```
PRINT #15, "B-F:"drive,track,block
```

The BLOCK-READ and WRITE commands have a slightly modified form which is more reliable in data processing. These are the USER commands, if the B-R and B-W are replaced by U1 and U2 respectively, the block I/O commands are executed without changing the buffer pointer.

RANDOM FILE PROCESSING

In random file processing we can write to anywhere on the disk. We must, however, also remember where we wrote the data to. There are many ways of doing this:

1. INDEX FILE

We can set up a sequential file on the disk holding the following information.

KEY VALUE OF RECORD
TRACK NUMBER
BLOCK NUMBER

The KEY VALUE could simply be the record number, or some unique identifier. The track and block numbers locate the record. Using indexed processing the programmer would search the index file before reading and write to the index file before writing.

2. INDEX ARRAY

To cut down the amount of disk activity, we could read our index file into an array in memory.

3. ADDRESS GENERATING ALGORITHM

An address generating algorithm takes a record number and computes its address on the disk. If we use an AGA we do not need to maintain an index. However, we still have to consider the problem of block allocation.

PROGRAM 6 - DATA CAPTURE PROGRAM

This program writes a set of records on tothe disc.

```
 10 OPEN 15,8,15, : REM COMMAND CHANNEL
 20 OPEN 5,8,5,"#" :REM RANDOM DATA CHANNEL
 30 OPEN 6,8,6 "INDEX,S,W" :REM INDEX FILE
 35 R=Q
 40 PRINT "ENTER DATA, USE 'END' TO FINISH"
 50 INPUT "RECORD CONTENTS=";R$
 60 R=R+1
 70 PRINT#5,R","R$ :REM RECORD RECORD# PLUS RECORD
 80 T=1 : B=1 :REM SET UP DUMMY ADDRESS
 90 PRINT#15, "B-A:"0,T,B :REM ATTEMPT TO ALLOCATE
100 INPUT#15,A,B$,C,D :REM READ ERROR
110 IF A=65 THEN T=C:B=D:GOTO 90
120 PRINT#15,"B-W:"5,0,T,B :REM WRITE DATA
130 PRINT#16, R;T;B :REM WRITE KEY VALUE+ADDRESS
140 IF R$<>"END"THEN GOTO 40
150 CLOSE 5 : CLOSE 6 : CLOSE 15
```

PROGRAM 7 - DATA REPORT PROGRAM

This program allows the user to read a selected record from a random file.

```
10 OPEN 15,8,15
20 OPEN 5,8,5,"#"
30 OPEN 6,8,6,"INDEX,S,R"
40 INPUT "RECORD NUMBER" N
50 FOR I=1TON:INPUT#6,R,T,B:NEXTI
60 PRINT#15,"B-R:"5,0,T,B
70 INPUT#5,X,R$
80 PRINT "RECORD IS:" R$
90 CLOSE 5:CLOSE6:CLOSE15
```

Notice that, in the above program, we have not included any error checking.

RELATIVE FILES

A DOS relative file uses the first two bytes of a data record block as pointers to other records.

The DOS keeps tracks of all the blocks used, and can indeed keep track of records, even if they cross over a block boundary.

This is done by establishing a set of SIDE SECTORS for the beginning of each record in the file. There are a maximum of 6 SIDE SECTORS to a FILE. Each SIDE SECTOR can address up to 120 RECORDS, and there can be up to 720 records of up to 254 characters. The maximum file size is then 720*254 characters. That is 180 kilobytes, approximately.

File processing commands are as follows:

OPEN

```
    OPEN fnum,devnum,channum,"name,L,"+CHR$(record length)
or
    OPEN fnum,devnum,channum,"name"
```

The first format is used when initially setting up the file, the second for any subsequent file processing.

When processing a relative file, we first of all position a file pointer to the correct record. The record number is a two byte value, and is calculated as follows:

```
Record number = High byte *256+low byte.
```

POSITION COMMAND

The format for setting the file pointer is:

```
PRINT#15,"P"CHR$(channum)CHR$(reclo)CHR$(rechi)
```

EXAMPLE

```
PRINT#15,"P"CHR$(4)CHR$(5)CHR$(1)
```

This positions the record pointer to record 1*256+5 = 261 in the file using channel 4. We can then use INPUT# and PRINT# as before.

When processing relative files, we still have to compute the record number. However, this no longer has to be related to the physical address of the record. We can therefore use any of the techniques of random processing, without worrying about block allocation.

QUIZ 7.2

Convert the mailing list system to relative file format. Use a data array for holding SURNAME and RECORD NUMBER as an index to a relative file. Add a feature which would allow the user to selectively print out the mailing list. This would involve setting up other indexes, perhaps zip-code and record number, or town and record number.

SUMMARY

In this chapter we discussed file handling on the Commodore 64 from both a theoretical and practical point of view. The following commands are available using BASIC and 1541 DOS:

```
OPEN     - opens file
PRINT#   - writes to file
INPUT#   - reads from file
CLOSE    - closes file
```

```
      GET#      - gets a byte
```

Using DOS, we can access data using

```
      BLOCK-READ
      BLOCK-WRITE
      BLOCK-ALLOCATE
      BLOCK-FREE
      BUFFER-POINTER
      USER1
      USER2
```

Using DOS relative files, we can use the

```
      POSITION
```

command to select data.

CHAPTER 8

Application Packages

INTRODUCTION

In this chapter we discuss the use of three commercial packages available on the Commodore 64. These packages show the use of databases, wordprocessing and spreadsheets. The packages are fairly typical of those available on the market, and show the use of the Commodore 64 in an office environment.

DATABASE PACKAGE - MAGPIE

In this section we discuss a commercial database package. The package we have selected is MAGPIE, it is manufactured and distributed in the U.K. by Audiogenic Ltd., P.O. Box 88, Reading, England.

Magpie is an example of a "card box" database system. Each database record is seen as entries on a "form". The database record is a sequential file, with Magpie providing indexes to allow fast access to the data.

Magpie provides the following capabilities:

1. Customized form design. The package allows the programmer to use the screen as a window on to a form. The form is designed by drawing fields on to the screen, declaring their type and writing any background information required.

2. Programmable query language. The programmer can use a built in query language to write an application, such as a stock control system.

3. Automatic data search and retrieval. The package can carry out complex searches as part of a system. Users are allowed to set up a search mask, thus giving them the capability of making flexible demands on the system. For example, in a personnel system, a user could form a search to ask a question such as:

"Search for all sales personnel with a name like Jones, living in New York, earning more than $25000 per annum"

4. Data extraction and printout. An output form may be designed for displaying retrieved data. The package allows two forms to be manipulated at the one time, with information being posted from one to the other. The forms can be displayed on the screen or on the printer.

5. Graphics output. It is possible to process data from numerical fields and display the result in the form of a bar chart.

6. Pop up menus. A rather nice feature of Magpie is its POP UP MENUS. These are menus which pop up on to the screen whenever a selection has to be made, and disappear when not required, leaving whatever text was behind them intact. The main pop up menu appears after signing on. It is as follows:

```
+------------------+
!Magpie Date-Base  !
!------------------!
!Run Procedure     !
!Use Calculator    !
!Get System        !
!Create System     !
!Load and Run      !
+------------------!
```

All pop up menus have the same format as the main menu.

CASE STUDY - MAILING LIST SYSTEM

A system in Magpie is defined as a complete application such as STOCK CONTROL, INVOICING, or in this case a MAILING LIST SYSTEM.

When we select "Get System" from the main pop up menu. We are presented with a menu of systems available on disk. We can then load a system. Magpie comes with a demonstration database mailing list system, which we shall discuss in the next few paragraphs.

When the mailing list system is loaded, control returns to

the main menu. We can then select the option to "run procedure". A new pop up menu appears:

```
+----------------+
!Run Procedure   !
!----------------!
!EXIT            !
!Add Names       !
!Modify Name     !
!Delete Name     !
!                !
!Print Labels    !
!Selective Print !
!                !
!Create New File !
+----------------+
```

It can be seen from the above menu that we have procedures which can handle most requests that we could make of a mailing list system.

The "Create New File" option allows the user to format a disk and set it up to receive new data.

The data is input via the "Add Names" procedure, which displays a form on the screen as below.

```
Mailing-List
------------


Name      :----------------
Address   :----------------
          :----------------
Town      :----------------
State     :------------
Zipcode   :----------
Phone No  :--------
Comments  :------------------------
```

WE enter names into the system using the above form.

The "Modify Name" procedure allows us to search for a particular record using a user defined search mask. We can use the usual wild card characters ? and * for unknown parts of a field.

WE enter as much information as necessary on to the form and then issue the GO command. Once a match has been found, the package asks the question:

"Is this the one? Enter Yes or No"

When the correct record has been found, it can be edited and control returned to the system menu.

The "Delete Name" procedure is similar to Modify Name, except that when a record is found, it is deleted.

There are two print commands in the system. "Print Labels" prints all the records on to sticky labels. The "Selective Print" option allows the us to set up a search mask, and only those records fitting the mask are printed.

Programming in Magpie is different from traditional programming, in that all commands are inserted into a procedure by selecting them from pop up menus.

As an example of a procedure, we include the "Add Names" procedure of the mailing list system.

```
'Add Names
Name & Address .D
begin
Name & Address .F
enter fields
save record
Enter Yes or No
Enter Another?
if no skip
repeat
End of Procedure
```

In the above procedure, the .D refers to the datafile, .F to the form. As well as data input and output, the package allows us to process data and output results.

Any user of such a package must be prepared to spend some time exploring the system. Such time spent will be worth it!!

In conclusion, a commercial database package such as Magpie provides us with a database management system and a query

programming language.

Using these features, we can:

- design input/output forms
- construct procedures
- search and retrieve data
- extract and output data.

By manipulating these features, we can build fairly sophisticated applications.

WORD PROCESSING - EASYSCRIPT

Possibly the first and certainly one of the most useful pieces of software for the business environment is a word processor. An increasing number of home users are also learning to appreciate the facilities provided by word processing packages. The Commodore recommended word processor, Easyscript, is provided free of charge at the time of writing on purchase the 1541 disk drive. Easyscript has been around for some time and is fairly well established piece of software. It is, somewhat unfortunately, still only available as a floppy disk. It is possible that a plug-in module containing Easyscript will be available at some future time but there is no sign of it as yet. Easyscript works with both disk and tape as storage media.

Easyscript loads in about two minutes. While it is loading, the screen display shows a number of color combinations. These color combinations can later be selected by the Easyscript user. However, in our experience the default of white lettering on a dark gray background is both the easiest to read and the kindest to the eyes.

After loading, Easyscript allows us to select the text width. This is the width of the text as it is typed on to the screen. It does not determine the width of the text printed on to a printer. In most cases the screen text width will be chosen as 40 columns so that the entire text can be seen on the screen at once. Alternatively a screen width equal to the print width could be chosen. Widths up to 240 columns may be chosen, with the screen acting as a window on the text.

We are then asked to select disk or tape storage and to choose from a list of printer types.

Easyscript drives a wide range of printers. After we have specified the text width, the storage medium and the printer type, Easyscript enters edit mode.

At this point the advantages of a wordprocessor become apparent to the user. Text may be typed in and is automatically aligned, without the need to press carriage return at the end of a line. Errors may be corrected, text rearranged, copied, right, left or center justified, printed bold or underlined, superscripted or subscripted (if the printer has this facility) by pressing a few keys. Files may be saved and recalled at a later date, or merged with other files. There is a mail merge facility allowing 'personalized' letters to be created.

A reasonably comprehensive manual is provided with the software, and there is little point attempting to reproduce this in detail here. We do, however, consider that a brief list of the available commands is valuable at this point, both as a reference and as an indication of what is in the package for those who are considering obtaining it.

COMMAND INSTRUCTIONS

[ ] underline or enhance
{ } bold or reverse
& % shadow
' superscript
, subscript
; : bold
0-9 special character
↑ escape character
- soft hyphen
D delete
E erase
I insert
R range
X transfer
A duplicate
S search definction
@ implement search(and replace)
U uppercase
F save
L load
O output V video

```
                P printer
                C continuous
                L linked
                X multiple copies
                F fill
     B   variable block
     M   (following B) measured block
     T   set tabs   H  horizontal
                    V  vertical
     P   view tabs
     C   delete a tab
     Z   delete all tabs


FORMAT INSTRUCTIONS

     lm  left margin
     rm  right margin
     ma  margin release
     sp  line spacing
     ju  right justification
     ra  right alignment
     cn  centering
     pl  paper length
     vp  vertical offset
     of  horizontal offset
     tl  text length
     fp  forced page
     ln  blank lines
     hd  heading
     ft  footing
     hl  heading margin
     hr  footing margin
     p#  page numbering
     lp  lines per inch
     lf  line feed
     pt  characters per inch
     nb  comments


FUNCTION KEYS

     F1  command
     F2  copies characters enclosed in quotes to status line
     F3  format
     F4  disk operations
     F5  capital lock
```

F6
F7 horizontal tab key
F8 vertical tab key

DISK OPERATIONS

N format new disk
S delete
R rename
V validate
= copy
$ directory

BASIC PROGRAMS AS TEXT

This is a useful feature for those who wish to document or publish, say as magazine articles, program descriptions with program listings. The Easyscript manual (or at least the copies we have seen), has a misprint in the section dealing with this topic. Therefore we consider it valuable to repeat the instructions here.

1. Load the program.
2. Insert a formatted disk into the disk drive to save the sequential text file that this procedure creates. We consider it bad practice to hold BASIC and text versions of the same program on the same disk.
3. Enter:

```
OPEN 8,8,8,"0:FILENAME,S,W":CMD8:LIST
PRINT#8
CLOSE8
```

   where 'FILENAME' is the chosen name for the file.
4. Wait until the disk drive stops. The disk should now hold a sequential text file which may be read by EASYSCRIPT.

LIMITATIONS OF EASYSCRIPT

There are, in our opinion, two major limitations in the use of EASYSCRIPT. The first is a hardware rather than a software limitation. As the micro is limited to a forty column screen display, what we see on the screen is not

necessarily what we get on the printer. This limitation is partly, but not wholly, circumvented by the video output facility.

Another limitation of EASYSCRIPT is that files can have a maximum length of just over 760 lines - about 14 pages of script. While adequate for most uses this makes the package difficult to use for long documents. Files may be linked, but editing a linked file can be time consuming.

These limitations must, however, be taken in context. On balance we consider EASYSCRIPT to be a valuable and useful item in any software library.

OTHER WORDPROCESSORS

Other wordprocessors available for the Commodore 64 include:

Quick Brown Fox - Quick Brown Fox Inc.
WordPro 3 Plus/64 - Professional Software Inc.
PaperClip 64 - Batteries Included (Canada)

SPREADSHEETS

In the beginning, when microcomputers were being introduced into business, the only systems development tool available was BASIC. Thus traditional systems such as payroll or accounts were devloped and sold as specific solutions to specific business problems. If a business executive wished to use a computer to help make a planning decision, or whatever, he or she had to learn COMPUTING; this takes time and dedication.

Unfortunately, BASIC was not designed for business personnel to "knock up" an application quickly, and so it is not the easiest of tools to use.

On the other hand, when business people are planning some venture, they will use sheets of paper, calculators and pencils, and develop the options available to them using these tools. Typically, they will plan a cash flow, say, and then see WHAT happens to cash flow IF they change some of their assumptions. This involves many repeated recalculations of the same set of mathematical problems, and as any human being knows, this inevitably leads to incorrect answers.

It was to meet this requirement of the market that spreadsheet programs were developed. A spreadsheet program provides us with an electronic equivalent of a sheet of paper, and also gives us an accurate calculator.

A typical spreadsheet is that marketed by HANDIC SOFTWARE AB of Sweden, called CALC RESULT.

The CALC RESULT program allows us to

- lay out spreadsheets in a convenient manner.
- carry out any calculations which we once undertook with paper and pencil.
- carry out a number of calculations quickly, and recalculate our results with different data.
- build professional standard reports using our own designs.
- report data in a graphical manner.

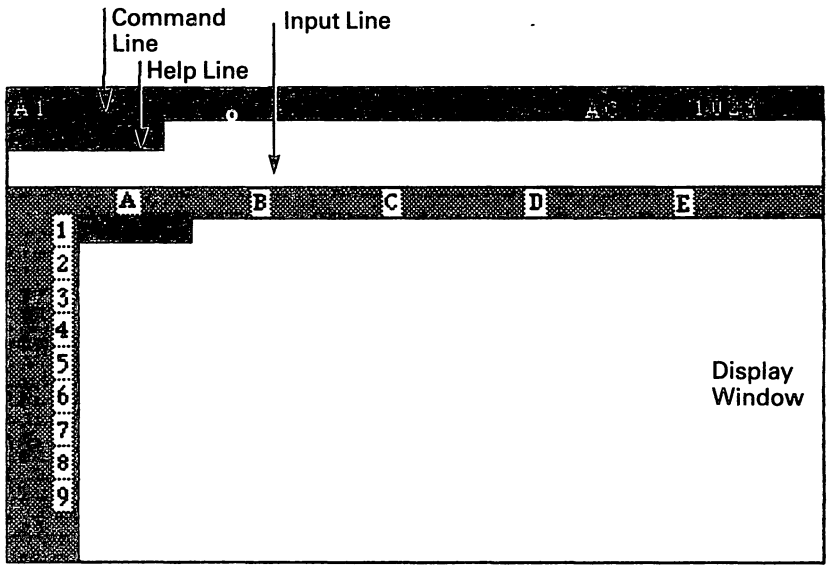
The spreadsheet consists of a two dimensional grid, with CELLS at the intersection of each row and column. The cell is referenced by two coordinates, a row number and a column letter. The row numbers range from 1 to 254, and the column letters range from A to BK (ie. A - Z, AA - AZ, BA - BK), 63 columns in all.

CALC RESULT allows us to enter information into these cells and provides us with a set of powerful logical commands and mathematical functions for manipulating this information.

Because the arithmetic is performed so quickly, we can fairly easily set up WHAT-IF modelling spreadsheets.

The CALC RESULT manual itself contains a tutorial guide, which is fairly good. This section will by means of a set of graded examples, introduce you to some serious applications of the package in the business area.

When we power up the CALC RESULT package, we are presented with a screen as in the following photograph.



We can identify the following features of the CALC RESULT screen:

DISPLAY WINDOW

Since the full spreadsheet is too large to be seen on the computer's display, CALC RESULT presents us with a WINDOW that moves over the spreadsheet, showing a portion at a time. The display window is delineated by the SCREEN BORDER. The top border contains column letters and the left side border contains row numbers. These two sets of numbers tell us where we are on the spreadsheet.

COMMAND LINE

This is the topmost line of the CALC RESULT page, and shows the CALC RESULT commands and the present cursor coordinates.

HELP LINE

The second line from the top shows the three functions which

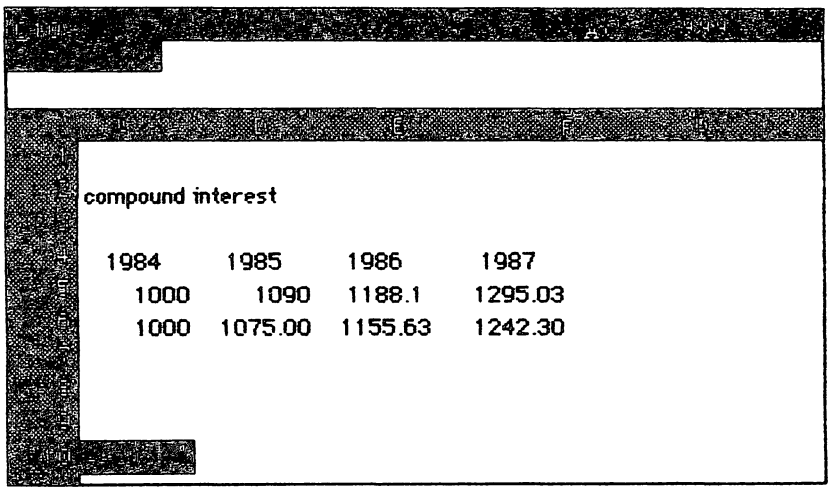
always follow the command choices on the COMMAND LINE. These functions are:

F3 for GOTO - which allows us to move about the sheet
F6 for PRINT - which dumps the display window to printer
CLR for CLEAR - which clears the present spreadsheet.

Note that CALC RESULT makes use of the function keys. The HELP LINE is also used for various other functions.

INPUT LINE

This line, third from the top, is used for accepting input and displaying characters that are entered into a cell. To enter values into a cell, we simply type them in. To indicate text, the first character should be a space.



PRACTICAL EXAMPLE 1

In this, our first example of a spreadsheet, we show the effects of compound interest. The sheet plots the growth of $1000 over the years, with the interest being compounded annually. Two different interest rates are shown.

The report is to have the following format:

Compound Interest

Rate %	1984	1985	1986	1987
9	1000	xxxx	xxxx	xxxx
7.5	1000	xxxx	xxxx	xxxxx

The report is to be placed at the top left of the spreadsheet. The formula to be used for calculating is:

```
(1 + R/100)*P
```

where R is interest Rate
P is the Principal.

Notice that we are not using the compound interest formula.

Using the above design, we can implement the spreadsheet to give us the following report:

COMPOUND INTEREST

RATE	1984	1985	1986	1987
9.00	1000.00	1090.00	1188.10	1295.03
7.50	1000.00	1075.00	1155.63	1242.30

The spreadsheet was developed using the following contents:

CELL REFERENCE	CONTENTS	TYPE
C2	COMPOUND	LABEL
D2	INTEREST	LABEL
E2	T	LABEL
A4	RATE	LABEL
C4	1984	LABEL
D4	1985	LABEL
E4	1986	LABEL
F4	1987	LABEL
A6	9	VALUE
C6	1000	VALUE
D6	(1+A6/100)*C6	VALUE
E6	(1+A6/100)*D6	VALUE
F6	(1+A6/100)*E6	VALUE
A7	7.5	VALUE
C7	1000	VALUE
D7	(1+A7/100)*C7	VALUE
E7	(1+A7/100)*D7	VALUE
F7	(1+A7/100)*E7	VALUE

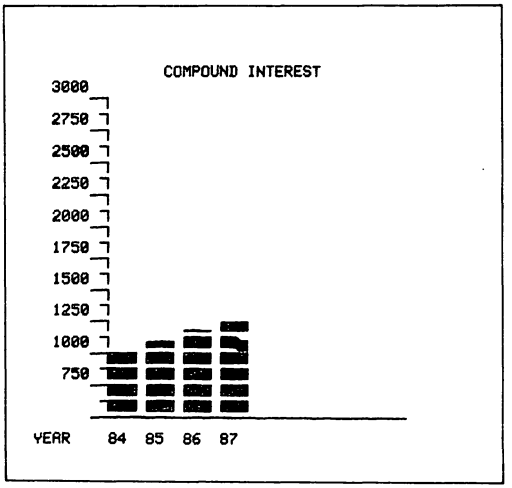
Notice from the report that the format of the numbers has been selected to be in dollar, cent format. When entering the formulae into the sheet we make full use of the replicate command. This allows us to enter a formula into a cell and to copy the cell entry across a range, adjusting the formula as we go.

We can manipulate the entries in A6, C6, A7 and C7 to experiment with the inferred values in the spreadsheet, as shown below.

COMPOUND INTEREST

RATE	1984	1985	1986	1987
8.5	1000	1085	1177.23	1277.29
11	1000	1110.00	1232.1	1367.63

We can see here how CALC RESULT can give us a flexible system of playing about with figures and text. We can use CALC RESULT as a planning tool as well as a general purpose computing tool. One very useful feature of CALC RESULT is its ability to produce graphical output as in the following printout.



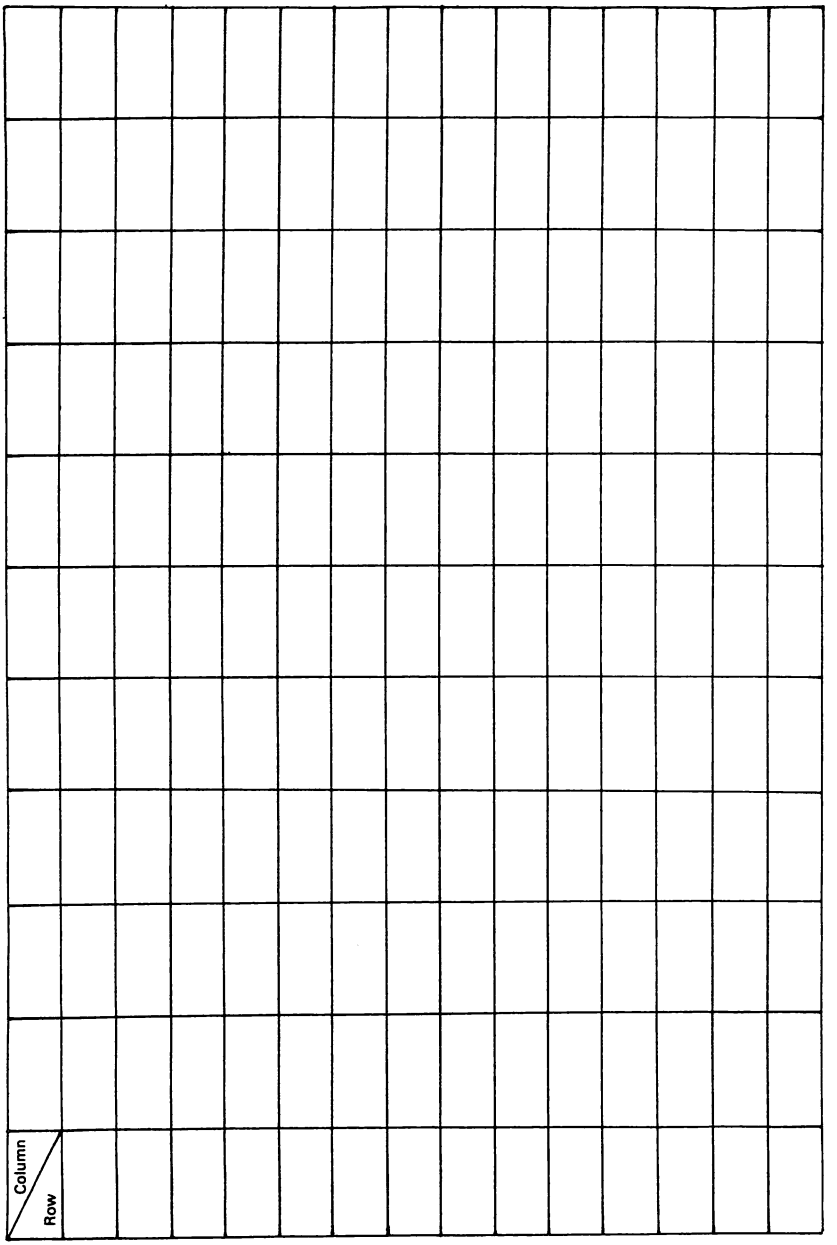
PLANNING AN APPLICATION

It is very important to consider the planning of a spreadsheet before using the software package itself. There is a proverb in computing which states that "the sooner you start to code, the sooner you start to create errors".

If we cannot describe our problem in non-computer terms, we will find it difficult to generate a computer solution.

However, CALC RESULT is normally used to provide solutions to fairly standard business problems. The central idea is to provide a report on some business activity, with the computer taking the place of paper and pencil. One of the most important aspects of such a report is its layout. It can help to use a standard form to plan your layouts. A standard layout form is given in Appendix G, and is used in the following spreadsheets. All we have to do is write in full all headings, etc. We should count the number of characters in each cell, noting its format, and any formulae used in its calculation. Once this has been done, we can create the model by a straightforward transfer from the form to the

CALC RESULT LAYOUT FORM



spreadsheet.

USING DATA DICTIONARIES IN CALC RESULT

When using CALC RESULT, it can be useful to place all parameters in a table in one place in the form. Thus when a parameter changes, for example, a tax rate change, we only change the parameter once rather than in every position where it is used on the sheet. Such a table is known as a data dictionary.

Similarly, it can be useful to use sections of the sheet for inputting data and other sections for reporting. This minimizes operator movement over the sheet.

PRACTICAL EXAMPLE 2

In this spreadsheet we forecast the future sales of a product, given that we expect the sales to increase at a constant rate. Thus the spreadsheet requires two parameters:

Sales in first year
Expected rate of interest per annum

This is quite a simple model, and not very realistic, but it shows some of the features of CALC RESULT.

Using the layout chart we have the following layout:

CALC RESULT LAYOUT

Row \ Column	A	B	C	D	E	F
1						
2						
3						
4	DATA DICTIONARY					
5	YEAR 1 SALES		1000			
6	RATE		10			
7						
8						
9	SALES		1984	1985	1986	
10			(1+C6/100)* C5	(1+C6/100)* C10	(1+C6/100)* D10	

In this model, year 1 is 1983 and the formula used to forecast sales is:

Sales this year = Sales last year *(1+Rate of increase%)

This model was implemented in CALC RESULT and gave us the following printout.

```
DATA DICTIONARY
YEAR 1 SALES      1000.00
RATE                10.00


SALES                 1984     1985     1986
                   1100.00 1210.00 1331.00
```

we will now show a spreadsheet that we use in our day to day work.

PRACTICAL EXAMPLE 3

SCOTTISH SOFTWARE FACTORY INVOICE PREPARATION

The following spreadsheet is used in the Scottish Software Factory to produce invoices for its products. The totals were produced by using the SUM function, with the rest of the model being self explanatory. The formulae were entered using the REPLICATE feature of CALC RESULT. An invoice prepared by the system is given below.

```
                                        SCOTTISH SOFTWARE FACTRY
                                        30 SANDILANDS STREET
                                        GLASGOW
                                        041-778 9009

INVOICE #          25.00                DATE
-------------------------------------------------------------------------

CUSTOMER                                CUSTOMER CODE       3.00
NOVOTECH MICROSYSTEMS                   MACHINE TYPE    APRICOT
30 SANDILANDS ST                        OP. SYS.        MSDOS
GLASGOW

-------------------------------------------------------------------------

PRODUCT#        DESCRIPTION     QUANTITY UNIT PR   PRICE      TAX    TOTAL
    1.00        PNS ACCOUNTS        3.00  330.00  990.00   148.50 1138.50
    5.00        PAYROLL             3.00  270.00  810.00   121.50  931.50
    8.00        FEE ESTIMATING      4.00  150.00  600.00    90.00  690.00
    6.00        PRINTERS ESTIM.     7.00  720.00 5040.00   756.00 5796.00
                                    0.00    0.00    0.00     0.00    0.00
                                    0.00    0.00    0.00     0.00    0.00
                                    0.00    0.00    0.00     0.00    0.00
                                    0.00    0.00    0.00     0.00    0.00
                                    0.00    0.00    0.00     0.00    0.00
                                    0.00    0.00    0.00     0.00    0.00
                                    0.00    0.00    0.00     0.00    0.00
                                    0.00    0.00    0.00     0.00    0.00
                                    0.00    0.00    0.00     0.00    0.00
                                    0.00    0.00    0.00     0.00    0.00
                                    0.00    0.00    0.00     0.00    0.00
                                                          ==========

                                TOTALS    7440.00 1116.00 8556.00

                                TOTAL DUE                 8556.00

-------------------------------------------------------------------------

TAX RATE    0.15
```

SUMMARY OF CALC RESULT FUNCTIONS

The following is a brief catalog of the features available to you as a CALC RESULT programmer.

FILE FUNCTIONS	- allow saving and loading
FORMATTING FUNCTIONS	- allows different means of presentation.

Formats available are:

COLUMN WIDTH	- default 8 characters, can be 5 to 18.
COLOR	- all CBM colors are available
MAXIMUM PRECISION	- decimals are shown
INTEGER FORMAT	- only uses whole numbers
$ FORMAT	- two decimal places
LEFT/RIGHT ADJUST	- allows entries to be left or right justified

MATHEMATICS FUNCTIONS

COUNT	- gives number of cells with constant or valid formula
MAX	- largest value in range
MIN	- least value in range
MEAN	- average value for area
STDDEN	- standard deviation for area
SUM	- sum of an area
NPV	- Net Present Value for a range
NA	- The NOT AVAILABLE function
ABS	- Absolute value,
INT	- Integer part
LN	- Natural Logarithm
LOG10	- Logarithm to base 10
SQRT	- square root
RND	- random numbers
IF..THEN..ELSE	- decision structure
OR,AND and NOT	- Boolean functions used with IF..THEN..ELSE

There are also a host of management functions for presenting graphics and editing the spreadsheet.

In conclusion, we believe that a spreadsheet program such as CALC RESULT is a necessary program in any serious computer

user's library. With a little perseverence a user can build fairly complex models of business aspects using the CALC RESULT program.

CHAPTER 9

Input/Output Devices

INTRODUCTION

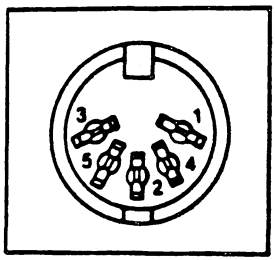
One of the most important considerations to be borne in mind when looking at any computer is its ability to communicate with the outside word. The Commodore 64 is well endowed in this respect, with two games parts, a user port, a serial bus, a cartridge expansion slot, an audio video socket, a cassette slot and an R.F. modulated output (for television). This allows a wide range of peripherals to be connected, although it must be said that - like most Commodore machines - the Commodore 64 will not easily interface with non Commodore devices.

AUDIO VIDEO

The audio video socket provides signals for driving a monitor - audio out, video out and luminance. More unusually there is also an audio-in connection. This input has an impedance of 100 kilohm and can handle an audio signal of up to 3 volts peak to peak. The signal is A.C. coupled to the SID input pin by a 10 microfarad electrolytic capacitor and can ride at a zero volt level - enabling the computer to be connected to the output of an audio amplifier. The signal received may be mixed with computer generated SID output or filtered. The SID chip has unity gain at maximum volume setting. An interesting application is to feed the output from one Commodore 64 into the input of another. This 'daisy chaining' can result in some most unusual effects. The SID volume control setting affects the external signal as well as the three internally generated voices.

The pinout of the audio/video connector is

Pin	Signal
1	LUMINANCE
2	GND
3	AUDIO OUT
4	VIDEO OUT
5	AUDIO IN



THE GAMES PORTS

The games ports each have five digital inputs which can detect a four position joystick plus a fire button. In addition they each have two analog inputs which connect to the POTX and POTY pins on the SID chip via a 4066 interface chip. This gives an analog to digital conversion facility so that the machine can work with analog paddles as well as a digital joystick.

JOYSTICK CONTROL

A joystick connected to port A controls the value of the four least significant bits in $DC00. The condition of the fire button controls the fifth least significant bit. A zero in the appropriate bit indicates the switch is made. Thus:

Joystick up - Bit 0 zero
Joystick down - Bit 1 zero
Joystick left - Bit 2 zero
Joystick right - Bit 3 zero
Fire button pressed - Bit 4 zero

A joystick connected to port B affects memory location $DC01 in a similar fashion.

The next program moves a sprite around the screen under joystick control. Pressing the fire button causes the sprite to change color.

DATA

```
3000 00 00 00 00 00 00 00 00
3008 00 00 18 00 00 18 00 FF
3010 FF FF 44 18 22 44 BD 22
3018 44 7E 22 4F E7 F2 4F DB
3020 F2 44 DB 22 FF E7 FF 00
3028 7E 00 00 BD 00 00 24 00
3030 00 66 00 00 42 00 00 00
3038 00 00 00 00 00 00 00 00
```

PROGRAM

```
JSR $E544      ;Clear screen.
LDA #$D0       ;Address of
STA $FC        ;start of VIC
LDA #$00       ;chip in $FB
STA $FB        ;and $FC.
LDY #$00       ;Reset memory pointer.
LDA #$9B       ;Initial X position
STA ($FB), Y   ;of sprite.
LDA #$7D       ;Initial Y position
INY            ;of
STA ($FB), y   ;sprite.
LDY #$10       ;Reset MSB of
LDA #$00       ;X position.
STA ($FB), Y   ;
LDY #$15       ;Enable
LDA #$01       ;Sprite 0.
STA (FB), Y    ;
INY            ;Expand in
INY            ;Y-direction.
STA ($FB), Y   ;
LDY #$1D       ;Expand in
STA ($FB), Y   ;X direction.
LDA #$C0       ;Point to MOB
STA $07F8      ;at $3000.
LDY #$20       ;Brown
LDA #$20       ;border.
STA ($FB), Y   ;
INY            ;Blue
LDA #$0E       ;screen.
STA ($FB), Y   ;
LDY #$27       ;Set initial
LDA #$00       ;sprite
STA ($FB), Y   ;color.
```

```
LOOP:  LDA $DC00      ;Get joystick control value.
       LSR A          ;Test bit
       BSC TSTDN      ;zero.
       JSR UP         ;
TSTDN: LSR A          ;Test bit
       BCS TSTL       ;one.
       JSR DOWN       ;
TSTL:  LSR A          ;Test bit
       BCS TSTR       ;three.
       JSR LEFT       ;
TSTR:  LSR A          ;Test bit
       BCS TSTFR      ;four.
       JSR RIGHT      ;
TSTFR: LSR A          ;Test bit
       BCS DEL        ;five.
       JSR COLOR      ;
DEL:   JSR $EEB3      ;One millisecond delay.
       CLC            ;Go back and test
       BCC LOOP       ;games port again.


SUBROUTINES

UP:    PHA            ;Ensure accumulator not corrupted.
       LDA $D001      ;Is sprite
       CMP #$32       ;at top of
       BEQ EXUP       ;screen?
       SEC            ;If not move
       SBC #$01       ;it up one
       STA $D001      ;position.
EXUP:  PLA            ;Restore accumulator value.
       RTS

DOWN:  PHA
       LDA $D001      ;Is sprite
       CMP #$D0       ;at bottom
       BEQ EXDN       ;of screen?
       CLC            ;If not move
       ADC #$01       ;it down one
       STA $D001      ;position.
EXDN:  PLA
       RTS

LEFT:  PHA
       LDA $D000      ;If least significant byte
       CMP #$18       ;of sprite position does not
       BNE SUBR       ;indicate farthest left position
```

```
        LDY $D010     ;or if most significant bit is
        BEQ EXLF      ;not zero,
SUBR:   DEC $D000     ;then decrement X position.
        LDA $D00      ;If least significant byte
        CMP #$FF      ;holds $FF then transition
        BNE EXLF      ;point has been passed and
        DEC $D010     ;least significant bit is reset.
EXLF:   PLA
        RTS

RIGHT:  PHA
        LDA $D000     ;If least significant byte
        CMP #$28      ;of sprite position does not
        BNE ADDN      ;indicate farthest right position
        LDY $D010     ;or if most significant
        BNE EXRT      ;bit is one
ADDN:   INC $D000     ;then increment X position.
        BNE EXRT      ;If at transition point
        INC $D010     ;then set most significant bit.
EXRT:   PLA
        RTS

COLOR:  INC $D027     ;Change color.
        LDA #$0E      ;Is sprite color
        CMP $D027     ;same as screen color?
        BNZ EXCOL     ;If not, exit subroutine.
        LDA #$00      ;If so, change sprite back
        STA $D027     ;to first color.
EXCOL:  RTS
```

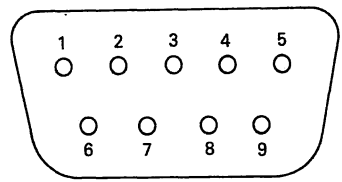
NOTE - Some monitors (such as MON64) accept LSR for LSR A.

PADDLES

A paddle is a analog device - basically a variable resistor. The paddle position is indicated by a digital value in a SID register - $D419 for port A and $D41A for port B. This value could, for example, be transferred to the Y position register of a sprite , causing the sprite to move up and down the screen under control of the paddle. Two paddles controlling two sprites in this way is the basis of many bat'n'ball type games.

The interrupt disable flag should be set when paddle position is being read. Reading paddle positions from BASIC is unreliable.

The games port pinouts are:



LIGHT PEN

A light pen is an input device which detects the raster scan. That is the beam of electrons which continuously sweeps the television or VDU screen creating a picture. A signal from the light pen causes the Y position of the raster scan to be latched into memory location $D014 and the eight most significant bits of the raster X position to be latched onto $D013, giving a resolution to two dots of the 512 dots across the screen. A light pen signal will also cause an interrupt if bit 3 of the interrupt enable register $D01A is set to one. A light pen interrupt will set bit 3 of the interrupt status register $D019. Only one light pen interrupt can occur per frame. The light pen is connected to games port A.

DATA DIRECTION REGISTERS

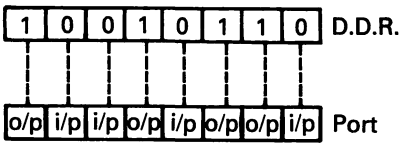
Each of the I/O ports in the Commodore 64 is controlled by a data direction register. This register determines whether an individual port line is an input or an output. If a bit in the data direction register is set to 1 the corresponding line in the port is an output, and vice versa, as in the diagram below.

Port 1

Pin	Signal
1	JOYA0
2	JOYA1
3	JOYA2
4	JOYA3
5	POTAY
6	BUTTON A/LP
7	+5V
8	GND
9	POTAX

Port 2

Pin	Signal
1	JOYB0
2	JOYB1
3	JOYB2
4	JOYB3
5	POTBY
6	BUTTON B
7	+5V
8	GND
9	POTBX



The data direction register for games port A ($DC00) is $DC02, and the D.D.R. for games port B ($DC01) is $DC03.

ON CHIP PORT

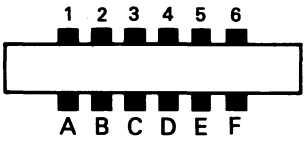
The 6510 on-chip port $0001 controls the cassette recorder and ROM switching circuitry. The bit assignment of this port is

Bit 0 - LORAM (switches BASIC ROM)
Bit 1 - HIRAM (switches KERNAL ROM)
Bit 2 - CHAREN (switches character ROM)
Bit 3 - cassette data output
Bit 4 - cassette switch sense
Bit 5 - cassette motor control.

The data direction register for this port is $0000. The six least significant bits in this register are set to 101111.

The pinout of the cassette connector is:

Pin	Signal
A-1	GND
B-2	+5V
C-3	CASSETTE MOTOR
D-4	CASSETTE READ
E-5	CASSETTE WRITE
F-6	CASSETTE SENSE



SERIAL I/O

A serial IEEE connection is provided by the serial I/O socket. (Some sources describe this as an RS232C connection,

but an interface device is required to drive RS232C peripherals from this connection.) This drives the Commodore MPS 81 printer, the 1541 disk drive, and any other IEEE serial devices connected to the computer. Bits 3-7 of data port A ($DD00) control this connection as follows:

Bit 7 - serial data input
Bit 6 - serial clock input
Bit 5 - serial data output
Bit 4 - serial clock output
Bit 3 - serial ATN output

The data direction register for data port A is $DD02. A channel to a printer on the serial connector may be opened by:

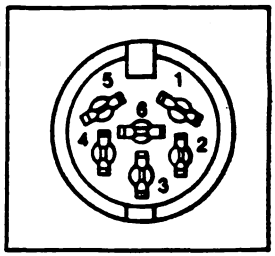
OPEN<channel number>,<primary address>,<secondary address>

The channel number may be between 1 and 255. Channel numbers greater than 128 generate a line feed automatically for each carriage return. The primary address for a printer is normally 4, although 5 can also be used provided the printer is switched to address 5. Only two secondary addresses are used. Zero (optional) gives cursor up mode for capitals and graphics. Seven gives cursor down mode for capitals and lower case.

The instruction CMD <channel number> will cause all output normally sent to the screen to be directed to the opened serial channel. PRINT#<channel number> works like the PRINT command except that output goes to the specified channel. The command CLOSE <channel number> (usually preceded by PRINT #) closes the channel.

The pinout of the serial I/O connector is:

Pin	Signal
1	SERIAL $\overline{\text{SRQIN}}$
2	GND
3	SERIAL ATN IN/OUT
4	SERIAL CLK IN/OUT
5	SERIAL DATA IN/OUT
6	$\overline{\text{RESET}}$



CENTRONICS

There is no centronics connector on the Commodore 64. Centronics devices may be connected using an adaptor such as the VICSPRINT 64 cartridge which converts serial IEEE to centronics.

RS232

Connection to RS232 devices may be implemented using a serial IEEE to RS232 adaptor such as Interpod. There are, however, standard RS232C signals avaliable at the User Port, although these are TTL signals and require voltage conversion before connection to an RS232 device. $DD01 is controlled by data direction register $DD03. RS232 handling routines also use the timers and NMI flags in CIA registers $DD04 - $DD0F. These routines may be called from BASIC or from machine code routines using the KERNAL entries CHKIN ($FFC6), GETIN ($FFE4), CHRIN ($FFCF), CHKOUT ($FFC9), and CHROUT ($FFD2).

Where high speed RS232 devices are used machine code may be necessary as otherwise the two 256-byte send and receive buffers (located at the top of user memory) may overflow.

To open an RS232 channel the syntax is:

```
OPEN<lfn>,2,0,"<control register>
    <command register(optional)> <baud low(optional)>
    <baud high(optional)>"
```

<Logical file number> is a number between 1 and 255. If a number greater than 127 is specified a line feed will be transmitted after each carriage return.

<Control register> is a single byte (0-255) number made up as follows:

Bit 7 specifies stop bits. If this bit is zero there is one stop bit, if it is 1 there are two.

Bits 6, 5 specify data word length:

00 - 8 bits
01 - 7 bits
10 - 6 bits
11 - 5 bits

Bit 4 is unused

```
Bits 3-0   specify baud rate
                  0000 - user rate
                  0001      50  baud
                  0010      75  baud
                  0011     110  baud
                  0100   134.5  baud
                  0101     150  baud
                  0110     300  baud
                  0111     600  baud
                  1000    1200  baud
                  1001    1800  baud
                  1010    2400  baud
                  1011    3600  baud
                  1100    4800  baud
                  1101    7200  baud
                  1110    9600  baud
                  1111   19200  baud
```

If the user rate is specified then baud rate is specified by the <baud low> and <baud high> numbers. These are calculated as follows:

```
<baud low> = system frequency/baud rate/2-100
             -<baud high>*256
<baud high>= INT((system frequency/baud rate/2-100)/256)
```

System frequency is 1.02273E6 in the USA and 0.98525E6 in the UK and most of Europe. This figure is linked to the TV standard.

<Command register> is an optional byte (0-255) character defining other terminal parameters as follows:

```
Bits 7-5   define parity options:
           XX0 parity disabled
           001 odd parity
           010 even parity
           101 mark transmitted, parity disabled
           111 space transmitted, parity disabled
Bit 4      defines transmission:
           0 - full duplex
           1 - half duplex
Bit 3-1    are unused
Bit 0      defines handshaking:
           0 - 3 line
           1 - X line
```

The open command for an RS232 channel clears all variables and arrays.

Data is read from an RS232 channel using either

```
GET#lfn, <string variable>
```

in BASIC or KERNAL routines:

```
CHKIN
GETIN
```
and CHRIN.

Data is sent to an RS232 channel using

```
CMD lfn
```
and PRINT#lfn

in BASIC or KERNAL routines:

```
CHKOUT
```
and CHROUT.

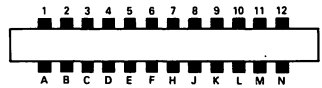
The channel is closed by CLOSE lfn or by KERNAL routine CLOSE.

DIGITAL INFORMATION

The user port lines can be connected to a digital circuit directly. In this mode they provide eight signal lines connecting to port B of the CIA #1 chip. The lines may each be configured as input or output by the appropriate bit of the data direction register $DD03, and will either place the value of the corresponding bit of $DD01 onto the port line, or will read the port line value into the bit as appropriate.

Two handshaking signals are also provided for communication with digital devices. The FLAG 1 signal is negative edge triggered. If the FLAG interrupt is enabled (Bit 4 of $DD0D) then this signal generates an interrupt request. FLAG 1 is used to allow an external device connected to the user port access to the commputer. PA2 is controlled by bit 2 of CIA Port A ($DD00). This signal may be used to indicate to the external device that the FLAG 1 signal has been received and that communication can commence.

The pinout of the user port is:



Pin	Signal
1	GND
2	+5V
3	RESET
4	CNT1
5	SP1
6	CNT2
7	SP2
8	PC2
9	SER.ATN IN
10	9VAC
11	9VAC
12	GND

Pin	Signal
A	GND
B	FLAG2
C	PB0
D	PB1
E	PB2
F	PB3
H	PB4
J	PB5
K	PB6
L	PB7
M	PA2
N	GND

INTERPOD

Interpod is an interface which plugs into the serial port and provides connections for up to thirty serial or parallel IEEE devices, or a mixture of both, and one RS232C device. Interpod does not utilize the user port and the method of setting up RS232 options is different from that previously described. The ability provided by Interpod to interface to parallel IEEE devices enables the 4040 or 8050 dual disk drive to be connected to the Commodore 64. We have used this device fairly often and have found it to be extremely useful, and well worth further investigation.

CARTRIDGE EXPANSION PORT

The cartridge expansion port is the largest connector on the Commodore 64. This port provides connections to all the microprocessor address and data lines. Also at this connector are lines which control the machine's memory configuration and provides direct memory access (DMA) enabling an external processor to take control of the system busses. The NMI, IRQ and Reset lines appear at this connector, as does the 8.18 MHz video dot clock from which all system timing is derived, and the bus available signal from the VIC II chip.

PROM cartridges may be connected to the machine via this port, giving 'instant' games and business programs, and utilities such as MON64 monitor. The port lines also allow the 6510 microprocessor to be switched out altogether and control of the machine taken over by another processor, such as, for example, a Z-80 located in the cartridge. This

enables the Commodore 64 to be converted to a CP/M machine, giving access to a very wide range of software.

The pinout of the cartridge expansion port is as follows:

Pin	Signal	Pin	Signal
1	GND	A	GND
2	+5V	B	ROMH
3	+5V	C	RESET
4	IRQ	D	NMI
5	CR/W	E	SO2
6	Dot clock	F	CA15
7	I/O 1	H	CA14
8	GAME	J	CA13
9	EXROM	K	CA12
10	I/O 2	L	CA11
11	ROML	M	CA10
12	BA	N	CA9
13	DMA	P	CA8
14	CD7	R	CA7
15	CD6	S	CA6
16	CD5	T	CA5
17	CD4	U	CA4
18	CD3	V	CA3
19	CD2	W	CA2
20	CD1	X	CA1
21	CDO	Y	CAO
22	GND	Z	GND

PRINTERS

Any serious user of the Commodore 64 must consider long and hard about what printer or printers to buy. We have, in the past, used printers which ranged in complexity from simple dot matrix printers to high speed laser printers. In producing this book we have used quite a few printers. Only one of which was a Commodore printer, the MPS 801. In the main, however, we have used printers from the BROTHER range.

DOT MATRIX PRINTERS

In a dot matrix printer, the characters are formed by means of a matrix of dots, as are screen characters. Typically the character will be formed by a 9*9 array of dots. The actual

print head on the printer will only have one column of pins, with the characters being formed by 9 firings of the pins.

A dot matrix printer forms the character M, by using the following pattern

```
.........
.O.....O.
.OO...OO.
.O.O.O.O.
.O..O..O.
.O.....O.
.O.....O.
.O.....O.
.........
```

Notice that this arrangement leaves one row for descenders, such as 'y'.

There are a number of types of dot matrix printers. Those most commonly used with microcomputers are needle printers and electrothermal printers.

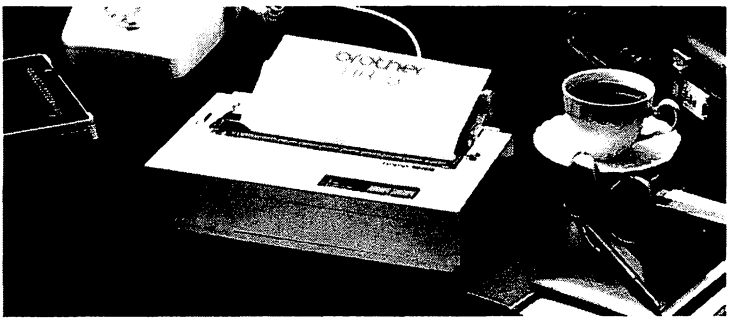
Needle printers are impact printers. The print head is made up of an array of needles or pins, which are hammered against an ink ribbon to place the dots on to the paper. Print speeds range from 12 to 160 characters per second. Needle printers are relatively inexpensive and can be fairly fast. They use ordinary paper and can make carbon copies.

Electrothermal printers produce dot matrix characters on specially coated paper. In this case the image is produced by heat, burning the coating away. Print speed is typically 30 characters per second. Electrothermal printers are relatively inexpensive and quiet, but thermal paper is expensive and unsuitable for some applications. Carbon copies cannot be produced.

We have used the following two dot matrix printers:-

The BROTHER HR5 is a low cost dot matrix recieve-only (RO) printer which can use thermal paper, thus there is no need for a ribbon. It is possible to use a ribbon cassette with ordinary paper.

The HR5 can be used with batteries or a mains adaptor.



The BROTHER HR5. Photograph courtesy of Jones & Brother®, a member of the Brother Industries Group.

We used the HR5 with a Centronics interface, but a direct connection to the C-64 is now available, as is an RS232C interface.

The HR5 prints text at 30 characters per second in a bi-directional mode, using either plain or thermal paper. The character set available is shown in the following sample.

The character pitch can be either 10 or 12 characters per inch.

Sample print from HR5 printer

Going up market a little is the M-1009. This again is a 9*9 dot matrix printer, but now printing at 50 CPS with the options of printing on to single sheets or fanfold continuous stationery. The M-1009 can be used as a standard business printer, printing out invoices (up to 3 part stationery).

Since a dot matrix printer uses hammers to hit pins on toa ribbon, the patterns produced by the printer can be controlled by the computer. In line with other dot matrix printers, the M-1009 has been set up for the computer to produce set effects.

The most interesting effects are produced by sending "control codes" or "escape sequences" to the printer before the characters to be printed. Some of the codes used by the M-1009 are:

Symbol	CHR value	Effect
SO	14	Enter enlarged character set
SI	15	Enter condensed character set
DC2	18	Reset for condensed characters
DC4	20	Reset for enlarged characters
ESC-n	27,45,1	Set underline on
	27,45,0	Set underline off
ESC 0	27,0	8 lines per inch
ESC 1	27,1	10.3 lines per inch
ESC 3n	27,51,n	Line spacing n/216 inch
ESC Cn	27,67,n	Page length =n lines
ESC E	27,69	Set Emphasized characters
ESC F	27,70	Reset Emphasized

The following program and its output shows the effects of such codes.

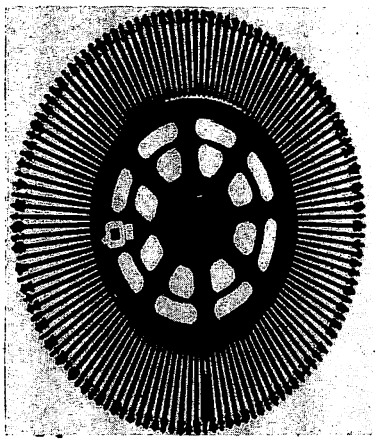
```
10 REM BROTHER M-1009 PRINTER TEST
20 ESC$=CHR$(27)
30 OPEN 1,4
40 PRINT "ENSURE THAT PRINTER IS CONNECTED"
50 PRINT "AND PAPER, ETC IS IN PLACE"
60 PRINT#1,"ENLARGED CHARACTERS"
70 PRINT#1,CHR$(14);"ABCDEF";CHR$(20);"GHIJ"
80 PRINT#1
90 PRINT#1,"CONDENSED CHARACTERS"
100 PRINT#1,CHR$(15);"ABCDEF";CHR$(18);"GHIJ"
110 PRINT#1
120 PRINT#1,"UNDERLINED CHARACTERS"
130 PRINT#1,ESC$+CHR$(45)+CHR$(1); "ABCDEF";
    ESC$+CHR$(45)+CHR$(0); "GHIJ"
140 PRINT#1
150 PRINT#1,"SET LINES TO 10.3 PER INCH"
155 PRINT#1,ESC$+CHR$(49)
160 FOR I=1 TO 10:PRINT#1,"ABCDEFGHI":NEXT I
170 PRINT#1
180 PRINT#1,"SET LINES TO 8 PER INCH"
185 PRINT#1,ESC$+CHR$(48)
190 FOR I=1 TO 10:PRINT#1,"ABCDEFGHI":NEXT I
200 PRINT#1
210 PRINT#1,"EMPHASIZED MODE"
220 PRINT#1,ESC$+CHR$(69);"ABCDEF";ESC$+CHR$(70);"GHIJ"
230 PRINT#1
```

```
240 PRINT#1,"DOUBLE STRIKE"
250 PRINT#1,ESC$+CHR$(71);"ABCDEF";ESC$+CHR$(72);"GHIJ"
260 PRINT#1
```

```
ENLARGED CHARACTERS
ABCDEFGHIJ

CONDENSED CHARACTERS
ABCDEFGHIJ

UNDERLINED CHARACTERS
ABCDEFGHIJ

SET LINES TO 10.3 PER INCH

ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI

SET LINES TO 8 PER INCH

ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI
ABCDEFGHI

EMPHASIZED MODE
ABCDEFGHIJ

DOUBLE STRIKE
ABCDEFGHIJ
```

DAISY WHEEL PRINTERS

In a daisy wheel printer, the characters are formed by a hammer hitting a preformed character against an ink ribbon, thus transferring the character to the paper. Daisy wheel printers can thus produce a high quality output. The figure below shows an example of a daisy wheel cassette. The characters are at the end of each petal of the daisy.



Courtesy of Qume (UK) Ltd®, an ITT subsidiary.

Daisy wheel printers are slow (15 to 50 characters per second), noisy and relatively expensive. To alter the character set it is necessary to change the daisy wheel cassette. On the other hand these printers give a high quality output on ordinary paper and can produce carbon copies.

The BROTHER HR35 is a typical example of a high quality daisy wheel printer, but it does have some special features. It has a 7K buffer for text reprinting. It also has color printing super/subscript, underlining, proportional spacing and underlining. It prints out at 36CPS with character spacing at 10,12 or 15 CPI, or indeed it can print out the text proportionally spaced. This means that the character "i" uses less space than the character "m", say.

BROTHER have configured one of their daisy wheel printers specifically for the C-64, this is the lower cost HR10.

DAISY WHEELS

There are over 20 types of daisy wheel cassette available. For example, there is PICA PITCH PRESTIGE ITALIC.

Get the jump on typing tasks with our quick and lively Brother E-Z electronic efficiency experts.
ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
1234567890

or the ELITE PITCH QUADRO

Get the jump on typing tasks with our quick and
lively Brother E-Z electronic efficiency experts.
ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
1234567890

COMMUNICATIONS WITH PRINTERS

Nowadays, printers are becomming more and more intelligent, and can exhibit many of the features of small computers. For example, a rather interesting printer in the BROTHER range is the EP-44 Personal Electronic Printer. This printer can be used as a portable typewriter, with an LCD display for previewing text before printing. Indeed, the EP 44 has a 4K memory for storing text.

The printer has a calculator function provides the user with the standard 4 functions and also a percentage function for arithmetic. It uses a 24*18 thermal dot matrix and thus has a high quality of text output, as can be seen from the following sample.

1 2 3 4 5 6 7 8 9 0 - ! ¢ p o i u y t r e w q a s d f g h j
k l ; = ½ . , m n b v c x z a q 1 2 w s x c d e 3 4 r f v b
g t 5 6 y h n m j u 7 8 i k , . l o 9 0 p ; ½ = ¢ - ! p o i

* " / @ £ _ & ' ( ) ? ½ $ P O I U Y T R E W Q A S D F G H J
K L : + % . , M N B V C X Z A Q * " W S X C D E / @ R F V B
G T £ _ Y H N M J U & ' I K , . L O ( ) P : % + $? ½ $ P O

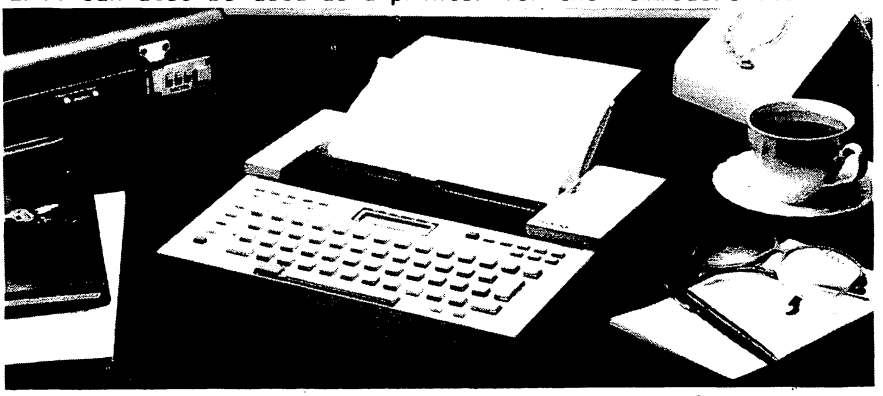
ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

The EP44 comes with an RS232 interface, with a printing speed

of 16 CPS, printing out 10 CPI.

The RS232 serial interface opens the EP44 to communications. It is possible to store 4K of text in the EP44 and transmit that text to the Commodore 64 for processing. Of course, the EP44 can also be used as a printer for the Commodore 64.



The EP44 Printer. Photograph courtesy of Jones & Brother®, a member of the Brother Industries Group.

THE VIC 1541 DISK DRIVE

The VIC 1541 SINGLE DRIVE FLOPPY DISK can be used as simply as a replacement for a COMMODORE Datassette, with almost the same commands. To LOAD a program, "PROGRAM1", say, from the disk drive, we simply type

```
LOAD "PROGRAM1",8
```

There are some commands over and above the cassette. First, we can use wild card characters in file names. The wild card characters are ? and *. If we want to LOAD the first file on the disk starting with the string ST, then we could type

```
LOAD"ST*",8
```

The Disk Operating System (DOS) will attempt to match ST* with a list of files (directory) that it maintains on the disk. If we are not sure of a single character then "ST?" will attempt to find a file whose name starts with "ST" with any other single letter following. The directory, itself, can be loaded into memory by the command:

```
LOAD"$",8
```

DOS SUPPORT

When we buy our disk drive, we should have some free programs on a demonstration disk. One of these programs is the DOS SUPPORT. When this has been LOADED and RUN, it creates a wedge into the BASIC ROM and changes some of the DOS commands.

LOAD can now be replaced by /. Thus

```
LOAD "PROGRAM",8
```

can be replaced by

```
/"PROGRAM"
```

@ and > can be used to send commands to disk. @$ will display the disk directory without upsetting the program in memory. As well as LOAD, SAVE and VERIFY we also have the usual I/O commands, for reading and writing information to disk files. These commands are discussed in chapter 7.

However, the 1541 contains a microprocessor in its own right and this can be used to manage the disk drive. To communicate with the disk drive we must OPEN a COMMAND channel to the disk. The format is:

```
OPEN 15,8,15
```

If we PRINT# to this channel, we effectively send commands to the DOS.

When using a disk for the first time, it must be formatted. This involves erasing the entire disc, and putting timing and block markers on to it. At this time the directory and a Block Allocation Map are set up. The format of the command is, (assuming that channel 15 is OPEN as file 15):

```
PRINT#15,"NEW0:name,id"
```

or

```
PRINT#15,"N0:name,id"
```

where "name" is the name of the disk and "id" is a 2 character for the disk. The "name" is inserted into the directory, while "id" is written on to every block on the disk.

If we wish to clear a disk for further use, the command is

```
PRINT#15,"N0,name"
```

We can make a backup copy of a file by using the COPY command, the format is:

```
PRINT#15,"COPY0:newname=0:oldname"
```

or

```
PRINT#15,"C0:newname=0:oldname"
```

If we wish to concatenate files, the copy command is:

```
PRINT#15,"c0:newname=0:n1,0:n2,..."
```

If we wish to RENAME a file on a disk, we use the RENAME command:

```
PRINT#15,"RENAME0:newname=oldname"
```

or

```
PRINT#15,"R0:new=old"
```

To delete a file from the directory, we use the scratch command

```
PRINT#15,"SCRATCH0:name"
```

or

```
PRINT#15,"S0:name"
```

Note that we can use wild card characters to delete more than one file at a time.

If we hit an error condition, we can reset the disk drive to its power up condition. The command is:

```
PRINT#15,"INITIALISE"
```

or

```
PRINT#15,"I"
```

When we have been using the disk drive for some time, information can be spread over the disk in a rather inefficient and disorganized fashion. The VALIDATE command will optimize the amount of free space on the disk. However, if we have a random file (see chapter 7), the VALIDATE command will destroy our data. The command is:

```
PRINT#15,"VALIDATE"
```

or PRINT#15,"V"

The command channel must be closed after all processing has been completed. The command is:

```
CLOSE 15
```

DISK ERRORS

The command channel is used by DOS to allow us to read error messages from the disk drive. This must be done from within a BASIC program. It can be useful to have an error subroutine which can be accessed after attempting a disk command.

The subroutine would be

```
INPUT#15,A$,B$,C$,D$:IF A$="0"THEN RETURN
PRINT A$,B$,C$,D$
RETURN
```

Where the error message is built up as follows:

```
A$ is the error number
B$ is the error name
C$ is the track number
D$ is the block number
```

PROGRAMMING THE DISK CONTROLLER

The VIC 1541 contains a 6502 microprocessor and 2 of 6522 I/O internal timers. There is also 16K of ROM and 2K of RAM. The ROM forms the intelligence of the system and the 2K RAM is normally used as buffer space. Since the drive is an intellegent device, it can process commands from the Commodore 64 itself, thus releasing the main processor for other tasks. Another consequence of the drive's intelligence, is its programmability. It is possible to write machine code programs and execute them on the disk drive.

The DOS commands which allow we to do this are discussed in the following paragraphs.

If we have a block of machine code on disc, the BLOCK-EXECUTE

command will load it into the buffer and start execution. Control will return to DOS via a RTS. The format is:

```
     PRINT#15,"BLOCK-EXECUTE:"channel,drive,track,block
or   PRINT#15,"B-E:"channel,drive,track,block.
```

The MEMORY-READ command sets up the error channel to point to the memory of the disk drive. We can then GET# from the error channel to read disk memory.

```
     PRINT#15,"M-R:"CHR$(lo byte of address)
                    CHR$(hi byte of address)
     GET#15,A$
     PRINT ASC(A$+CHR$(10))
```

With a single MEMORY-WRITE command we can write up to 34 bytes to the disk drive controller's memory. Once the code is written on to disk it can be run by a MEMORY-EXECUTE or USER command. The format of the MEMORY-WRITE is

```
     PRINT#15,"M-W:"CHR$(lo byte)CHR$(hi byte);
              number of bytes;byte data
```

The format of the command to execute our code is

```
     PRINT#15,"M-E:"CHR$(lo)CHR$(hi)
```

The USER commands take the form of a jump table within the memory of the disk drive. The table is as follows.

```
     U3 or UC      jump to $0500
     U4 or UD      jump to $0503
     U5 or UE      jump to $0506
     U6 or UF      jump to $0509
     U7 or UG      jump to $050C
     U8 or UH      jump to $050F
     U9 or UI      jump to $FFFA
```

If we send the command:

```
     PRINT#15,"U"+X$
```

then the 6502 control transfers to the appropriate vector. Notice that we have 3 bytes to work with, therefore we can set up another jump table at $0500, allowing us to write our own standard routines. Thus we can use the drive to extend the capabilities of the Commodore 64.

There are other user commands

U1 or UA	forces a block read without changing buffer pointer
U2 or UB	forces a block write without changing B-P
U; or UJ	is power up vector
UI+	sets disk speed to that required by C-64
VI-	sets disk speed to that required by VIC-20

The Commodore VIC 1541 disk drive is not only a fast data storage device, but also is a computer in its own right.

The philosophy of flexibility is present in the 1541 as much as in the Commodore 64.

QUIZ 9.1

1. Write a program which allows the disk to be examined, track by track.

2. Write a routine to disk drive which will write all blocks to zero if a password is not given. Use a USER command.

DISK COMMAND SUMMARY

In general, the format of a disk command is

```
PRINT#filenum,command
```

where filenumber is conveniently 15, and commands are given in the following table

COMMAND	FORMAT
NEW	"N
COPY	"CO:new=Oold
RENAME	"RO:new=O:old
SCRATCH	"SO:name
INITIALIZE	"I
VALIDATE	"V
BLOCK-READ	"B-R:"ch,dr,tr,bl
BLOCK-WRITE	"B-W:"ch,dr,tr,bl

```
BLOCK-ALLOCATE   "B-A:"dr,tr,bl
BLOCK-FREE       "B-F:"dr,tr,bl
BUFFER-POINTER   "B-P:"ch,position
POSITION         "P"CHR$8(ch)CHR$(lo)CHR$(pos)
BLOCK-EXECUTE    "B-E:"ch,dr,tr,bl
MEMORY-READ      "M-E:"CHR$(lo)CHR$(hi)
MEMORY-WRITE     "M-W:"CHR$(lo)CHR$(char)"data"
MEMORY-EXECUTE   "M-E:"CHR$(lo)CHR$(hi)
USER COMMANDS    "Vn:"
```

where ch is channel number
dr is drive number
tr is track number
bl is block number
lo is low byte of address
hi is high byte of address.

As well as DOS command we can execute BASIC commands OPEN,CLOSE,PRINT#,INPUT#,GET#

SUMMARY

The Commodore 64 has a very flexible input/output environment.

The RF modulated output drives a television.

The audio socket provides signals to drive a monitor plus an audio input.

Joysticks, paddles and light pens may be connected (but not simultaneously) to the games ports.

The serial port connects directly to IEEE serial devices such as the Commodore MPS 81 printer and 1541 disk drive. This port may also be used with suitable adaptors (for example Interpod) to control RS232, Centronics and IEEE parallel devices.

The user port provides digital input and output signals and handshaking lines. This port may also be used for RS232 signalling with appropriate voltage level conversion.

The cassette port controls the cassette recorder and the ROM switching lines.

The cartridge expansion port allows ROM based plug in software to be used. It also allows a second processor to take over control of the machine.

A wide range of printers may be connected to the Commodore 64, although most of the non-Commodore ones will require some kind of interface device.

The 1541 Disk Drive has a microprocessor on board and is thus capable of being driven by software. There are a range of inbuilt DOS commands to allow the user to manipulate the device.

APPENDIX A

6510 Assembly Language Instruction Set

ADC Add Memory to Accumulator with Carry
AND "AND" Memory with Accumulator
ASL Shift Left One Bit (Memory or Accumulator)

BCC Branch on Carry Clear
BCS Branch on Carry Set
BEQ Branch on Result Zero
BIT Test Bits in Memory with Accumulator
BMI Branch on Result Minus
BNE Branch on Result not Zero
BPL Branch on Result Plus
BRK Force Break
BVC Branch on Overflow Clear
BVS Branch on Overflow Set

CLC Clear Carry Flag
CLD Clear Decimal Mode
CLI Clear Interrupt Disable Bit
CLV Clear Overflow Flag
CMP Compare Memory and Accumulator
CPX Compare Memory and Index X
CPY Compare Memory and Index Y

DEC Decrement Memory by One
DEX Decrement Index X by One
DEY Decrement Index Y by One

EOR "Exclusive-Or" Memory with Accumulator

INC Increment Memory by One
INX Increment Index X by One
INY Increment Index Y by One

JMP Jump to New Location
JSR Jump to New Location Saving Return Address

LDA Load Accumulator with Memory
LDX Load Index X with Memory
LDY Load Index Y with Memory
LSR Shift Right One Bit (Memory or Accumulator)

NOP No Operation

ORA "OR" Memory with Accumulator

PHA Push Accumulator on Stack
PHP Push Processor Status on Stack
PLA Pull Accumulator from Stack
PLP Pull Processor Status from Stack

ROL Rotate One Bit Left (Memory or Accumulator)
ROR Rotate One Bit Right (Memory or Accumulator)
RTI Return from Interrupt
RTS Return from Subroutine

SBC Subtract Memory from Accumulator with Borrow
SEC Set Carry Flag
SED Set Decimal Mode
SEI Set Interrupt Disable Status
STA Store Accumulator in Memory
STX Store Index X in Memory
STY Store Index Y in Memory

TAX Transfer Accumulator to Index X
TAY Transfer Accumulator to Index Y
TSX Transfer Stack Pointer to Index X
TXA Transfer Index X to Accumulator
TXS Transfer Index X to Stack Pointer
TYA Transfer Index Y to Accumulator

ADC Add memory to accumulator with carry

Operation: A + M + C—A, C

N	Z	C	I	D	V
/	/	/	–	–	/

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	ADC	#Oper	69	2	2
Zero Page	ADC	Oper	65	2	3
Zero Page, X	ADC	Oper, X	75	2	4
Absolute	ADC	Oper	6D	3	4
Absolute, X	ADC	Oper, X	7D	3	4*
Absolute, Y	ADC	Oper, Y	79	3	4*
(Indirect, X)	ADC	(Oper, X)	61	2	6
(Indirect), Y	ADC	(Oper), Y	71	2	5*

*Add 1 if page boundary is crossed.

AND AND memory with accumulator

Operation: A ∧ M—A

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	AND	#Oper	29	2	2
Zero Page	AND	Oper	25	2	3
Zero Page, X	AND	Oper, X	35	2	4
Absolute	AND	Oper	2D	3	4
Absolute, X	AND	Oper, X	3D	3	4*
Absolute, Y	AND	Oper, Y	39	3	4*
(Indirect, X)	AND	(Oper, X)	21	2	6
(Indirect), Y	AND	(Oper), Y	31	2	5*

*Add 1 if page boundary is crossed.

ASL — Shift Left One Bit (Memory or Accumulator)

Operation: C—76543210—0

N Z C I D V
/ / / – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Accumulator	ASL	A	0A	1	2
Zero Page	ASL	Oper	06	2	5
Zero Page, X	ASL	Oper, X	16	2	6
Absolute	ASL	Oper	0E	3	6
Absolute, X	ASL	Oper, X	1E	3	7

BCC — Branch on Carry Clear

Operation: Branch on C = 0

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BCC	Oper	90	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

BCS — Branch on carry set

Operation: Branch on C = 1

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BCS	Oper	B0	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to next page.

BEQ — Branch on result zero

Operation: Branch on Z = 1

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BEQ	Oper	F0	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to next page.

BIT

Test bits in memory with accumulator

Operation: a ∧ M, M_7 → N, M_6 → V

N	Z	C	I	D	V
M_7	/	–	–	–	M_6

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Zero Page	BIT	Oper	24	2	3
Absolute	BIT	Oper	2C	3	4

BMI

Branch on result minus

Operation: Branch on N = 1

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BMI	Oper	30	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

BNE

Branch on result not zero

Operation: Branch on Z = 0

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BNE	Oper	D0	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

BPL

Branch on result plus

Operation: Branch on N = 0

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BPL	Oper	10	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

BRK — Force Break

Operation: Forced Interrupt PC + 2 P

N	Z	C	I	D	V
–	–	–	*	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	BRK	00	1	7

*A BRK command cannot be masked by setting I.

BVC — Branch on overflow clear

Operation: Branch on V = O

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BVC	Oper	50	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

BVS — Branch on overflow set

Operation: Branch on V = 1

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BVS	Oper	70	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

CLC — Clear carry flag

Operation: 0 – C

N	Z	C	I	D	V
–	–	0	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	CLC	18	1	2*

CLD

Clear decimal mode

Operation: 0 – D

N	Z	C	I	D	V
–	–	–	–	0	–

Addressing mode	Assembly language form	OP Code	No. Bytes	No. Cycles
Implied	CLD	D8	1	2

CLI

Clear interrupt disable bit

Operation: 0—I

N	Z	C	I	D	V
–	–	–	0	–	–

Addressing mode	Assembly language form	OP Code	No. Bytes	No. Cycles
Implied	CLI	58	1	2

CLV

Clear overflow flag

Operation: 0—V

N	Z	C	I	D	V
–	–	–	–	–	0

Addressing mode	Assembly language form	OP Code	No. Bytes	No. Cycles
Implied	CLV	B8	1	2

CMP

Compare memory and accumulator

Operation: A—M

N	Z	C	I	D	V
/	/	/	–	–	–

Addressing mode	Assembly language form		OP Code	No. Bytes	No. Cycles
Immediate	CMP	#Oper	C9	2	2
Zero Page	CMP	Oper	C5	2	3
Zero Page, X	CMP	Oper, X	D5	2	4
Absolute	CMP	Oper	CD	3	4
Absolute, X	CMP	Oper, X	DD	3	4*
Absolute, Y	CMP	Oper, Y	D9	3	4*
(Indirect, X)	CMP	(Oper, X)	C1	2	6
(Indirect), Y	CMP	(Oper), Y	D1	2	5*

*Add 1 if page boundary is crossed.

CPX

Compare memory and index X

Operation: X—M

N	Z	C	I	D	V
/	/	/	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	CPX	#Oper	E0	2	2
Zero Page	CPX	Oper	E4	2	3
Absolute	CPX	Oper	EC	3	4

CPY

Compare memory and index Y

Operation: Y—M

N	Z	C	I	D	V
/	/	/	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	CPY	#Oper	C0	2	2
Zero Page	CPY	Oper	C4	2	3
Absolute	CPY	Oper	CC	3	4

DEC

Decrement memory by one

Operation: M-1—M

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Zero Page	DEC	Oper	C6	2	5
Zero Page, X	DEC	Oper, X	D6	2	6
Absolute	DEC	Oper	CE	3	6
Absolute, X	DEC	Oper, X	DE	3	7

DEX

Decrement index X by one

Operation: X-1X

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	DEX	CA	1	2

DEY

Decrement index Y by one

Operation: Y-1—Y

N Z C I D V
/ / – – – –

Addressing mode	Assembly language form	OP Code	No. Bytes	No. Cycles
Implied	DEY	88	1	2

EOR

Exclusive—Or memory with accumulator

Operation: A-M—A

N Z C I D V
/ / – – – –

Addressing mode	Assembly language form		OP Code	No. Bytes	No. Cycles
Immediate	EOR	#Oper	49	2	2
Zero Page	EOR	Oper	45	2	3
Zero Page, X	EOR	Oper, X	55	2	4
Absolute	EOR	Oper	4D	3	4
Absolute, X	EOR	Oper, X	5D	3	4*
Absolute, Y	EOR	Oper, Y	59	3	4*
(Indirect, X)	EOR	(Oper, X)	41	2	6
(Indirect), Y	EOR	(Oper), Y	51	2	5*

*Add 1 if page boundary is crossed.

INC

Increment memory by one

Operation: M + 1—M

N Z C I D V
/ / – – – –

Addressing mode	Assembly language form		OP Code	No. Bytes	No. Cycles
Zero Page	INC	Oper	E6	2	5
Zero Page, X	INC	Oper, X	F6	2	6
Absolute	INC	Oper	EE	3	6
Absolute, X	INC	Oper, X	FE	3	7

INX

Increment index X by one

Operation: X + 1—X

N Z C I D V
/ / – – – –

Addressing mode	Assembly language form	OP Code	No. Bytes	No. Cycles
Implied	INX	E8	1	2

INY

Increment index Y by one

Operation: Y+1—Y

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Implied	INY		C8	1	2

JMP

Jump to new location

Operation: (PC+1)—PCL
(PC+2)—PCH

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Absolute	JMP	Oper	4C	3	3
Indirect	JMP	(Oper)	6C	3	5

JSR

Jump to new location saving return address

Operation: PC+2, (PC+1)—PCL
(PC+2)—PCH

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Absolute	JSR	Oper	20	3	6

LDA

Load accumulator with memory

Operation: M—A

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	LDA	#Oper	A9	2	2
Zero Page	LDA	Oper	A5	2	3
Zero Page, X	LDA	Oper, X	B5	2	4
Absolute	LDA	Oper	AD	3	4
Absolute, X	LDA	Oper, X	BD	3	4*
Absolute, Y	LDA	Oper, Y	B9	3	4*
(Indirect, X)	LDA	(Oper, X)	A1	2	6
(Indirect), Y	LDA	(Oper), Y	B1	2	5*

*Add 1 if page boundary is crossed.

LDX

Load index X with memory

Operation: M—X

N Z C I D V
/ / – – – –

Addressing mode	Assembly language form		OP Code	No. Bytes	No. Cycles
Immediate	LDX	#Oper	A2	2	2
Zero Page	LDX	Oper	A6	2	3
Zero Page, Y	LDX	Oper, Y	B6	2	4
Absolute	LDX	Oper	AE	3	4
Absolute, Y	LDX	Oper, Y	BE	3	4*

*Add 1 when page boundary is crossed.

LDY

Load index Y with memory

Operation: M—Y

N Z C I D V
/ / – – – –

Addressing mode	Assembly language form		OP Code	No. Bytes	No. Cycles
Immediate	LDY	#Oper	A0	2	2
Zero Page	LDY	Oper	A4	2	3
Zero Page, X	LDY	Oper, X	B4	2	4
Absolute	LDY	Oper	AC	3	4
Absolute, X	LDY	Oper, X	BC	3	4*

*Add 1 when page boundary is crossed.

LSR

Shift right one bit (memory or accumulator)

Operation: 0—7 6 5 4 3 2 1 0—C

N Z C I D V
0 / / – – –

Addressing mode	Assembly language form		OP Code	No. Bytes	No. Cycles
Accumulator	LSR	A	4A	1	2
Zero Page	LSR	Oper	46	2	5
Zero Page, X	LSR	Oper, X	56	2	6
Absolute	LSR	Oper	4E	3	6
Absolute, X	LSR	Oper, X	5E	3	7

NOP

No operation

Operation: No operation (2 cycles)

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	NOP	EA	1	2

ORA

OR memory with accumulator

Operation: AVM—A

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	ORA	#Oper	09	2	2
Zero Page	ORA	Oper	05	2	3
Zero Page, X	ORA	Oper, X	15	2	4
Absolute	ORA	Oper	0D	3	4
Absolute, X	ORA	Oper, X	1D	3	4*
Absolute, Y	ORA	Oper, Y	19	3	4*
(Indirect, X)	ORA	(Oper, X)	01	2	6
(Indirect), Y	ORA	(Oper), Y	11	2	5*

*Add 1 on page crossing.

PHA

Push accumulator on stack

Operation: A|

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	PHA	48	1	3

PHP

Push processor status on stack

Operation: P|

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	PHP	08	1	3

PLA

Pull accumulator from stack

Operation: A↑

N Z C I D V
/ / – – – –

Addressing mode	Assembly language form	OP Code	No. Bytes	No. Cycles
Implied	PLA	68	1	4

PLP

Pull processor status from stack

Operation: P↑

N Z C I D V
From Stack

Addressing mode	Assembly language form	OP Code	No. Bytes	No. Cycles
Implied	PLP	28	1	4

ROL

Rotate one bit left (memory or accumulator)

Operation: 7 6 5 4 3 2 1 0—C (M or A)

N Z C I D V
/ / / – – –

Addressing mode	Assembly language form		OP Code	No. Bytes	No. Cycles
Accumulator	ROL	A	2A	1	2
Zero Page	ROL	Oper	26	2	5
Zero Page, X	ROL	Oper, X	36	2	6
Absolute	ROL	Oper	2E	3	6
Absolute, X	ROL	Oper, X	3E	3	7

ROR

Rotate one bit right (memory or accumulator)

Operation: C—7 6 5 4 3 2 1 0 (M or A)

N Z C I D V
/ / / – – –

Addressing mode	Assembly language form		OP Code	No. Bytes	No. Cycles
Accumulator	ROR	A	6A	1	2
Zero Page	ROR	Oper	66	2	5
Zero Page, X	ROR	Oper, X	76	2	6
Absolute	ROR	Oper	6E	3	6
Absolute, X	ROR	Oper, X	7E	3	7

RTI

Return from interrupt

Operation: P|PC|

N Z C I D V
From Stack

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	RTI	40	1	6

RTS

Return from subroutine

Operation: PC|, PC + 1—PC

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	RTS	60	1	6

SBC

Subtract memory from accumulator with borrow

Operation: A—M—C—A
Note: C = Borrow

N Z C I D V
/ / / – – /

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	SBC	# Oper	E9	2	2
Zero Page	SBC	Oper	E5	2	3
Zero Page, X	SBC	Oper, X	F5	2	4
Absolute	SBC	Oper	ED	3	4
Absolute, X	SBC	Oper, X	FD	3	4*
Absolute, Y	SBC	Oper, Y	F9	3	4*
(Indirect, X)	SBC	(Oper, X)	E1	2	6
(Indirect), Y	SBC	(Oper), Y	F1	2	5*

*Add 1 when page boundary is crossed.

SEC

Set carry flag

Operation: 1—C

N Z C I D V
– – 1 – – –

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	SEC	38	1	2

SED

Set decimal mode

Operation: 1—D

N	Z	C	I	D	V
–	–	–	–	1	–

Addressing mode	Assembly language form	OP Code	No. Bytes	No. Cycles
Implied	SED	F8	1	2

SEI

Set interrupt disable status

Operation: 1—I

N	Z	C	I	D	V
–	–	–	1	–	–

Addressing mode	Assembly language form	OP Code	No. Bytes	No. Cycles
Implied	SEI	78	1	2

STA

Store accumulator in memory

Operation: A—M

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	Assembly language form		OP Code	No. Bytes	No. Cycles
Zero Page	STA	Oper	85	2	3
Zero Page, X	STA	Oper, X	95	2	4
Absolute	STA	Oper	8D	3	4
Absolute, X	STA	Oper, X	9D	3	5
Absolute, Y	STA	Oper, Y	99	3	5
(Indirect, X)	STA	(Oper, X)	81	2	6
(Indirect), Y	STA	(Oper), Y	91	2	6

STX

Store index X in memory

Operation: X—M

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	Assembly language form		OP Code	No. Bytes	No. Cycles
Zero Page	STX	Oper	86	2	3
Zero Page, Y	STX	Oper, Y	96	2	4
Absolute	STX	Oper	8E	3	4

STY

Store index Y in memory

Operation: Y—M

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Zero Page	STY	Oper	84	2	3
Zero Page, X	STY	Oper, X	94	2	4
Absolute	STY	Oper	8C	3	4

TAX

Transfer accumulator to index X

Operation: A—X

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TAX	AA	1	2

TAY

Transfer accumulator to index Y

Operation: A—Y

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TAY	A8	1	2

TSX

Transfer stack pointer to index X

Operation: S—X

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TSX	BA	1	2

TXA Transfer index X to accumulator

Operation: X—A

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TXA	8A	1	2

TXS Transfer index X to stack pointer

Operation: X—S

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TXS	9A	1	2

TYA Transfer index Y to accumulator

Operation: Y—A

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TYA	98	1	2

00 – BRK
01 – ORA—(Indirect, X)
02 – Future Expansion
03 – Future Expansion
04 – Future Expansion
05 – ORA—Zero Page
06 – ASL—Zero Page
07 – Future Expansion
08 – PHP
09 – ORA—Immediate
0A – ASL—Accumulator
0B – Future Expansion
0C – Future Expansion
0D – ORA—Absolute
0E – ASL—Absolute
0F – Future Expansion
10 – BPL
11 – ORA—(Indirect), Y
12 – Future Expansion
13 – Future Expansion
14 – Future Expansion
15 – ORA—Zero Page, X
16 – ASL—Zero Page, X
17 – Future Expansion
18 – CLC
19 – ORA—Absolute, Y
1A – Future Expansion
1B – Future Expansion
1C – Future Expansion
1D – ORA—Absolute, X
1E – ASL—Absolute, X
1F – Future Expansion
20 – JSR
21 – AND—(Indirect, X)
22 – Future Expansion
23 – Future Expansion
24 – BIT—Zero Page
25 – AND—Zero Page
26 – ROL—Zero Page
27 – Future Expansion
28 – PLP
29 – AND—Immediate
2A – ROL—Accumulator
2B – Future Expansion
2C – BIT—Absolute
2D – AND—Absolute
2E – ROL—Absolute
2F – Future Expansion
30 – BMI
31 – AND—(Indirect), Y
32 – Future Expansion
33 – Future Expansion
34 – Future Expansion
35 – AND—Zero Page, X
36 – ROL—Zero Page, X
37 – Future Expansion
38 – SEC
39 – AND—Absolute, Y
3A – Future Expansion
3B – Future Expansion
3C – Future Expansion
3D – AND—Absolute, X
3E – ROL—Absolute, X
3F – Future Expansion
40 – RTI
41 – EOR—(Indirect, X)
42 – Future Expansion
43 – Future Expansion
44 – Future Expansion
45 – EOR—Zero Page
46 – LSR—Zero Page
47 – Future Expansion
48 – PHA
49 – EOR—Immediate
4A – LSR—Accumulator
4B – Future Expansion
4C – JMP—Absolute
4D – EOR—Absolute
4E – LSR—Absolute
4F – Future Expansion
50 – BVC
51 – EOR—(Indirect), Y
52 – Future Expansion
53 – Future Expansion
54 – Future Expansion
55 – EOR—Zero Page, X
56 – LSR—Zero Page, X
57 – Future Expansion
58 – CLI
59 – EOR—Absolute, Y
5A – Future Expansion
5B – Future Expansion
5C – Future Expansion
5D – EOR—Absolute, X
5E – LSR—Absolute, X
5F – Future Expansion
60 – RTS
61 – ADC—(Indirect, X)
62 – Future Expansion
63 – Future Expansion
64 – Future Expansion
65 – ADC—Zero Page
66 – ROR—Zero Page
67 – Future Expansion
68 – PLA
69 – ADC—Immediate
6A – ROR—Accumulator
6B – Future Expansion
6C – JMP—Indirect
6D – ADC—Absolute
6E – ROR—Absolute
6F – Future Expansion
70 – BVS
71 – ADC—(Indirect), Y
72 – Future Expansion
73 – Future Expansion
74 – Future Expansion
75 – ADC—Zero Page, X
76 – ROR—Zero Page, X
77 – Future Expansion
78 – SEI
79 – ADC—Absolute, Y
7A – Future Expansion
7B – Future Expansion
7C – Future Expansion
7D – ADC—Absolute, X
7E – ROR—Absolute, X
7F – Future Expansion
89 – Future Expansion
81 – STA—(Indirect, X)
82 – Future Expansion
83 – Future Expansion
84 – STY—Zero Page
85 – STA—Zero Page
86 – STX—Zero Page
87 – Future Expansion
88 – DEY
89 – Future Expansion
8A – TXA
8B – Future Expansion
8C – STY—Absolute

8D – STA—Absolute
8E – STX—Absolute
8F – Future Expansion
90 – BCC
91 – STA—(Indirect), Y
92 – Future Expansion
93 – Future Expansion
94 – STY—Zero Page, X
95 – STA—Zero Page, X
96 – STX—Zero Page, Y
97 – Future Expansion
98 – TYA
99 – STA—Absolute, Y
9A – TXS
9B – Future Expansion
9C – Future Expansion
9D – STA—Absolute, X
9E – Future Expansion
9F – Future Expansion
A0 – LDY—Immediate
A1 – LDA—(Indirect, X)
A2 – LDX—Immediate
A3 – Future Expansion
A4 – LDY—Zero Page
A5 – LDA—Zero Page
A6 – LDX—Zero Page
A7 – Future Expansion
A8 – TAY
A9 – LDA—Immediate
AA – TAX
AB – Future Expansion
AC – LDY—Absolute
AD – LDA—Absolute
AE – LDX—Absolute
AF – Future Expansion
B0 – BCS
B1 – LDA—(Indirect), Y
B2 – Future Expansion
B3 – Future Expansion
B4 – LDY—Zero Page, X
B5 – LDA—Zero Page, X
B6 – LDX—Zero Page, Y
B7 – Future Expansion
B8 – CLV
B9 – LDA—Absolute, Y
BA – TSX
BB – Future Expansion
BC – LDY—Absolute, X
BD – LDA—Absolute, X
BE – LDX—Absolute, Y
BF – Future Expansion
C0 – CPY—Immediate
C1 – CMP—(Indirect, X)
C2 – Future Expansion
C3 – Future Expansion
C4 – CPY—Zero Page
C5 – CMP—Zero Page
C6 – DEC—Zero Page
C7 – Future Expansion
C8 – INY
C9 – CMP—Immediate
CA – DEX
CB – Future Expansion
CC – CPY—Absolute
CD – CMP—Absolute
CE – DEC—Absolute
CF – Future Expansion
D0 – BNE
D1 – CMP—(Indirect), Y
D2 – Future Expansion
D3 – Future Expansion
D4 – Future Expansion
D5 – CMP—Zero Page, X
D6 – DEC—Zero Page, X
D7 – Future Expansion
D8 – CLD
D9 – CMP—Absolute, Y
DA – Future Expansion
DB – Future Expansion
DC – Future Expansion
DD – CMP—Absolute, X
DE – DEC—Absolute, X
DF – Future Expansion
E0 – CPX—Immediate
E1 – SBC—(Indirect, X)
E2 – Future Expansion
E3 – Future Expansion
E4 – CPX—Zero Page
E5 – SBC—Zero Page
E6 – INC—Zero Page
E7 – Future Expansion
E8 – INX
E9 – SBC—Immediate
EA – NOP
EB – Future Expansion
EC – CPX—Absolute
ED – SBC—Absolute
EE – INC—Absolute
EF – Future Expansion
F0 – BEQ
F1 – SBC—(Indirect), Y
F2 – Future Expansion
F3 – Future Expansion
F4 – Future Expansion
F5 – SBC—Zero Page, X
F6 – INC—Zero Page, X
F7 – Future Expansion
F8 – SED
F9 – SBC—Absolute, Y
FA – Future Expansion
FB – Future Expansion
FC – Future Expansion
FD – SBC—Absolute, X
FE – INC—Absolute, X
FF – Future Expansion

APPENDIX B

Commodore 64 Memory Map

LABEL	ADDRESS	DESCRIPTION
D6510	0000	6510 On-Chip Data-Direction Register
R6510	0001	6510 On-Chip 8-Bit Input/Output Register
	0002	Unused
ADRAY1	0003–0004	Jump Vector: Convert Floating—Integer
ADRAY2	0005–0006	Jump Vector: Convert Integer—Floating
CHARAC	0007	Search Character
ENDCHR	0008	Flag: Scan for Quote at End of String
TRMPOS	0009	Screen Column From Last TAB
VERCK	000A	Flag: 0 = Load, 1 = Verify
COUNT	000B	Input Buffer Pointer / No. of Subscripts
DIMFLG	000C	Flag: Default Array DIM-ension
VALTYP	000D	Data Type: $FF = String, $00 = Numeric
INTFLG	000E	Data Type: $80 = Integer, $00 = Floating
GARBFL	000F	Flag: DATA scan/LIST quote/Garbage Call
SUBFLG	0010	Flag: Subscript Ref / User Function Call
INPFLG	0011	Flag: $00 = INPUT, $40 = GET, $98 = READ
TANSGN	0012	Flag: TAN sign / Comparison Result
	0013	Flag: INPUT Prompt
LINNUM	0014–0015	Temp: Integer Value
TEMPPT	0016	Pointer: Temporary String Stack
LASTPT	0017–0018	Last Temp String Address
TEMPST	0019–0021	Stack for Temporary Strings
INDEX	0022–0025	Utility Pointer Area
RESHO	0026–002A	Floating-Point Product of Multiply
TXTTAB	002B–002C	Pointer: Start of BASIC Text
VARTAB	002D–002E	Pointer: Start of BASIC Variables
ARYTAB	002F–0030	Pointer: Start of BASIC Arrays

LABEL	ADDRESS	DESCRIPTION
STREND	0031–0032	Pointer: End of BASIC Arrays (+1)
FRETOP	0033–0034	Pointer: Bottom of String Storage
FRESPC	0035–0036	Utility String Pointer
MEMSIZ	0037–0038	Pointer: Highest Address Used by BASIC
CURLIN	0039–003A	Current BASIC Line Number
OLDLIN	003B–003C	Previous BASIC Line Number
OLDTXT	003D–003E	Pointer: BASIC Statement for CONT
DATLIN	003F–0040	Current DATA Line Number
DATPTR	0041–0042	Pointer:Current DATA Item Address
INPPTR	0043–0044	Vector: INPUT Routine
VARNAM	0045–0046	Current BASIC Variable Name
VARPNT	0047–0048	Pointer: Current BASIC Variable Data
FORPNT	0049–004A	Pointer: Index Variable for FOR/NEXT
	004B–0060	Temp Pointer / Data Area
FACEXP	0061	Floating-Point Accumulator #1: Exponent
FACHO	0062–0065	Floating Accum. #1: Mantissa
FACSGN	0066	Floating Accum. #1: Sign
SGNFLG	0067	Pointer: Series Evaluation Constant
BITS	0068	Floating Accum. #1: Overflow Digit
ARGEXP	0069	Floating-Point Accumulator #2: Exponent
ARGHO	006A–006D	Floating Accum. #2: Mantissa
ARGSGN	006E	Floating Accum. #2: Sign
ARISGN	006F	Sign Comparison Result: Accum. #1 vs #2
FACOV	0070	Floating Accum. #1. Low-Order (Rounding)
FBUFPT	0071–0072	Pointer: Cassette Buffer
CHRGET	0073–008A	Subroutine: Get Next Byte of BASIC Text
CHRGOT	0079	Entry to Get Same Byte of Text Again
TXTPTR	007A–007B	Pointer: Current Byte of BASIC Text
RNDX	008B–008F	Floating RND Function Seed Value
STATUS	0090	Kernal I/O Status Word: ST
STKEY	0091	Flag: STOP key / RVS key
SVXT	0092	Timing Constant for Tape
VERCK	0093	Flag: 0 = Load, 1 = Verify
C3PO	0094	Flag: Serial Bus—Output Char.

LABEL	ADDRESS	DESCRIPTION
		Buffered
BSOUR	0095	Buffered Character for Serial Bus
SYNO	0096	Cassette Sync No.
	0097	Temp Data Area
LDTND	0098	No. of Open Files / Index to File Table
DFLTN	0099	Default Input Device (0)
DFLTO	009A	Default Output (CMD) Device (3)
PRTY	009B	Tape Character Parity
DPSW	009C	Flag: Tape Byte-Received
MSGFLG	009D	Flag: $80 = Direct Mode, $00 = Program
PTR1	009E	Tape Pass 1 Error Log
PTR2	009F	Tape Pass 2 Error Log
TIME	00A0–00A2	Real-Time Jiffy Clock (approx) 1/60 Sec
	00A3–00A4	Temp Data Area
CNTDN	00A5	Cassette Sync Countdown
BUFPNT	00A6	Pointer: Tape I/O Buffer
INBIT	00A7	RS-232 Input Bits / Cassette Temp
BITCI	00A8	RS-232 Input Bit Count / Cassette Temp
RINONE	00A9	RS-232 Flag: Check for Start Bit
RIDATA	00AA	RS-232 Input Byte Buffer/ Cassette Temp
RIPRTY	00AB	RS-232 Input Parity / Cassette Short Cnt
SAL	00AC–00AD	Pointer: Tape Buffer/Screen Scrolling
EAL	00AE–00AF	Tape End Addresses/End of Program
CMP0	00B0–00B1	Tape Timing Constants
TAPE1	00B2–00B3	Pointer: Start of Tape Buffer
BITTS	00B4	RS-232 Out Bit Count / Cassette Temp
NXTBIT	00B5	RS-232 Next Bit to Send/Tape EOT Flag
RODATA	00B6	RS-232 Out Byte Buffer
FNLEN	00B7	Length of Current File Name
LA	00B8	Current Logical File Number
SA	00B9	Current Secondary Address
FA	00BA	Current Device Number
FNADR	00BB–00BC	Pointer: Current File Name
ROPRTY	00BD	RS-232 Out Parity / Cassette Temp
FSBLK	00BE	Cassette Read/Write Block Count
MYCH	00BF	Serial Word Buffer

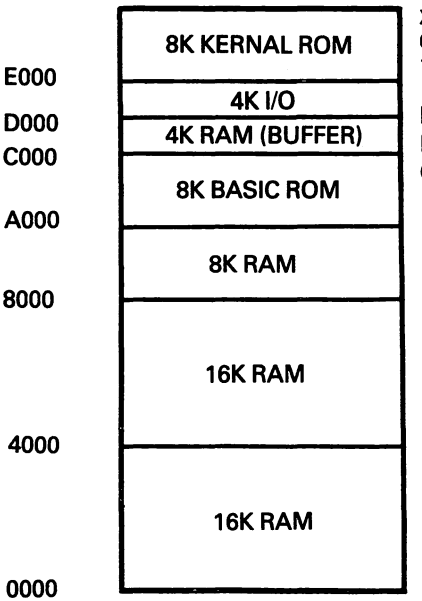
LABEL	ADDRESS	DESCRIPTION
CAS1	00C0	Tape Motor Interlock
STAL	00C1–00C2	I/O Start Address
MEMUSS	00C3–00C4	Tape Load Temps
LSTX	00C5	Current Key Pressed: CHR$(n) 0 = No Key
NDX	00C6	No. of Chars. in Keyboard Buffer (Queue)
RVS	00C7	Flag: Print Reverse Chars.— 1=Yes, 0=No Used
INDX	00C8	Pointer: End of Logical Line for INPUT
LXSP	00C9–00CA	Cursor X-Y Pos. at Start of INPUT
SFDX	00CB	Flag: Print Shifted Chars.
BLNSW	00CC	Cursor Blink enable: 0 = Flash Cursor
BLNCT	00CD	Timer: Countdown to Toggle Cursor
GDBLN	00CE	Character Under Cursor
BLNON	00CF	Flag: Last Cursor Blink On/Off
CRSW	00D0	Flag: INPUT or GET from Keyboard
PNT	00D1–00D2	Pointer: Current Screen Line Address
PNTR	00D3	Cursor Column on Current Line
QTSW	00D4	Flag: Editor in Quote Mode, $00 = NO
LNMX	00D5	Physical Screen Line Length
TBLX	00D6	Current Cursor Physical Line Number
	00D7	Temp Data Area
INSRT	00D8	Flag: Insert Mode, >0 = # INSTs
LDTB1	00D9–00F2	Screen Line Link Table / Editor Temps
USER	00F3–00F4	Pointer: Current Screen Color RAM loc.
KEYTAB	00F5–00F6	Vector: Keyboard Decode Table
RIBUF	00F7–00F8	RS-232 Input Buffer Pointer
ROBUF	00F9–00FA	RS-232 Output Buffer Pointer
FREKZP	00FB–00FE	Free 0-Page Space for User Programs
BASZPT	00FF	BASIC Temp Data Area
	0100–01FF	Micro-Processor System Stack Area
	0100–010A	Floating to String Work Area
BAD	0100–013E	Tape Input Error Log
BUF	0200–0258	System INPUT Buffer
LAT	0259–0262	KERNAL Table: Active Logical File No's.

LABEL	ADDRESS	DESCRIPTION
FAT	0263–026C	KERNAL Table: Device No. for Each File
SAT	026D–0276	KERNAL Table: Second Address Each File
KEYD	0277–0280	Keyboard Buffer Queue (FIFO)
MEMSTR	0281–0282	Pointer: Bottom of Memory for O.S.
MEMSIZ	0283–0284	Pointer: Top of Memory for O.S.
TIMOUT	0285	Flag: Kernal Variable for IEEE Timeout
COLOR	0286	Current Character Color Code
GDCOL	0287	Background Color Under Cursor
HIBASE	0288	Top of Screen Memory (Page)
XMAX	0289	Size of Keyboard Buffer
RPTFLG	028A	Flag: REPEAT Key Used, $80 = Repeat
KOUNT	028B	Repeat Speed Counter
DELAY	028C	Repeat Delay Counter
SHFLAG	028D	Flag: Keyb'rd SHIFT Key/CTRL Key/C = Key
LSTSHF	028E	Last Keyboard Shift Pattern
KEYLOG	028F–0290	Vector: Keyboard Table Setup
MODE	0291	Flag: $00 = Disable SHIFT Keys, $80 = Enable SHIFT Keys
AUTODN	0292	Flag: Auto Scroll Down, 0 = ON
M51CTR	0293	RS-232: 6551 Control Register Image
M51CDR	0294	RS-232: 6551 Command Register Image
M51AJB	0295–0296	RS-232 Non-Standard BPS (Time/2-100) USA
RSSTAT	0297	RS-232: 6551 Status Register Image
BITNUM	0298	RS-232 Number of Bits Left to Send
BAUDOF	0299–029A	RS-232 Baud Rate: Full Bit Time (μs)
RIDBE	029B	RS-232 Index to End of Input Buffer
RIDBS	029C	RS-232 Start of Input Buffer (Page)
RODBS	029D	RS-232 Start of Output Buffer (Page)
RODBE	029E	RS-232 Index to End of Output Buffer
IRQTME	029F–02A0	Holds IRQ Vector During Tape I/O
ENABL	02A1	RS-232 Enables
	02A2	TOD Sense During Cassette I/O

LABEL	ADDRESS	DESCRIPTION
	02A3	Temp Storage for Cassette Read
	02A4	Temp D1IRQ Indicator For Cassette Read
	02A5	Temp For Line Index
	02A6	PAL/NTSC Flag, 0 = NTSC, 1 = PAL
	02A7–02FF	Unused
IERROR	0300–0301	Vector: Print BASIC Error Message
IMAIN	0302–0303	Vector: BASIC Warm Start
ICRNCH	0304–0305	Vector: Tokenize BASIC Text
IQPLOP	0306–0307	Vector: BASIC Text LIST
IGONE	0308–0309	Vector: BASIC Char. Dispatch
IEVAL	030A–030B	Vector: BASIC Token Evaluation
SAREG	030C	Storage for 6510 .A Register
SXREG	030D	Storage for 6510 .X Register
SYREG	030E	Storage for 6510 .Y Register
SPREG	030F	Storage for 6510 .SP Register
USRPOK	0310	USR Function Jump Instr (4C)
USRADD	0311–0312	USR Address Low Byte/High Byte
	0313	Unused
CINV	0314–0315	Vector: Hardware IRQ Interrupt
CBINV	0316–0317	Vector: BRK Instr. Interrupt
NMINV	0318–0319	Vector: Non-Maskable Interrupt
IOPEN	031A–031B	KERNAL OPEN Routine Vector
ICLOSE	031C–031D	KERNAL CLOSE Routine Vector
ICHKIN	031E–031F	KERNAL CHKIN Routine Vector
ICKOUT	0320–0321	KERNAL CHKOUT Routine Vector
ICLRCH	0322–0323	KERNAL CLRCHN Routine Vector
IBASIN	0324–0325	KERNAL CHRIN Routine Vector
IBSOUT	0326–0327	KERNAL CHROUT Routine Vector
ISTOP	0328–0329	KERNAL STOP Routine Vector
IGETIN	032A–032B	KERNAL GETIN Routine Vector
ICLALL	032C–032D	KERNAL CLALL Routine Vector
USRCMD	032E–032F	User-Defined Vector
ILOAD	0330–0331	KERNAL LOAD Routine Vector
ISAVE	0332–0333	KERNAL SAVE Routine Vector
	0334–033B	Unused
TBUFFR	033C–03FB	Tape I/O Buffer
	03FC–03FF	Unused
VICSCN	0400–07FF	1024 Byte Screen Memory Area
	0400–07E7	Video Matrix: 25 Lines × 40 Columns
	07F8–07FF	Sprite Data Pointers
	0800–9FFF	Normal BASIC Program Space
	8000–9FFF	VSP Cartridge ROM—8192 Bytes

LABEL	ADDRESS	DESCRIPTION
	A000–BFFF	BASIC ROM—8192 Bytes (or 8K RAM)
	C000–CFFF	RAM—4096 Bytes
	D000–DFFF	Input/Output Devices and Color RAM or Character Generator ROM or RAM—4096 Bytes
	E000–FFFF	KERNAL ROM—8192 Bytes (or 8K RAM)

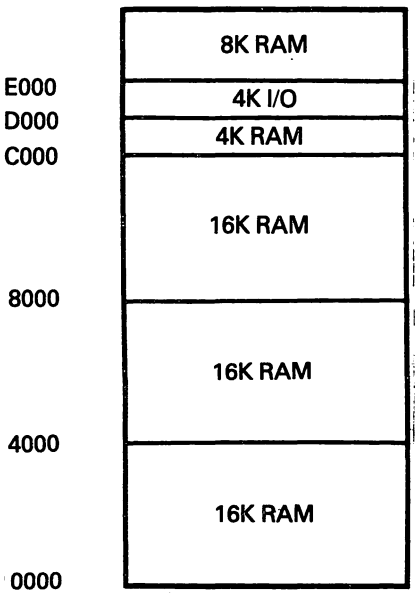
Commodore 64 Memory Layouts



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM = 1
HIRAM = 1
GAME = 1
EXROM = 1

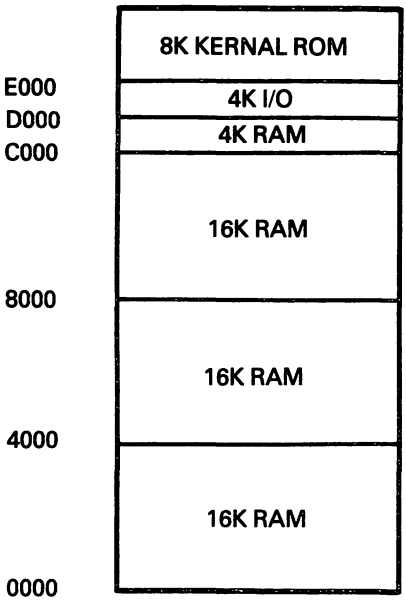
This is the default BASIC memory map which provides BASIC 2.0 and 38K contiguous bytes of user RAM.



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM = 1
HIRAM = 0
GAME = 1
EXROM = X
OR
LORAM = 1
HIRAM = 0
GAME = 0
(THE CHARACTER ROM IS NOT ACCESSIBLE BY THE CPU IN THIS MAP)
EXROM = 0

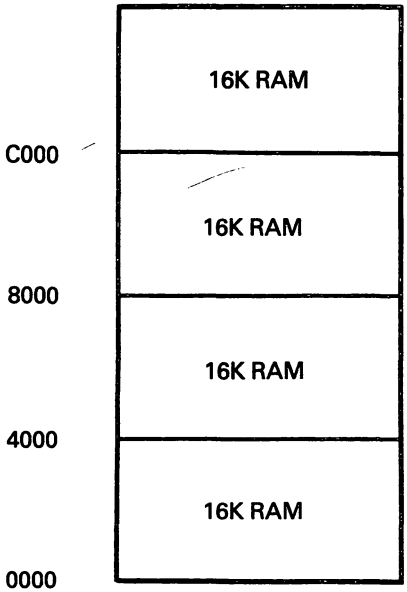
This map provides 60K bytes of RAM and I/O devices. The user must write his/her own I/O driver routines.



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM = 0
HIRAM = 1
GAME = 1
EXROM = X

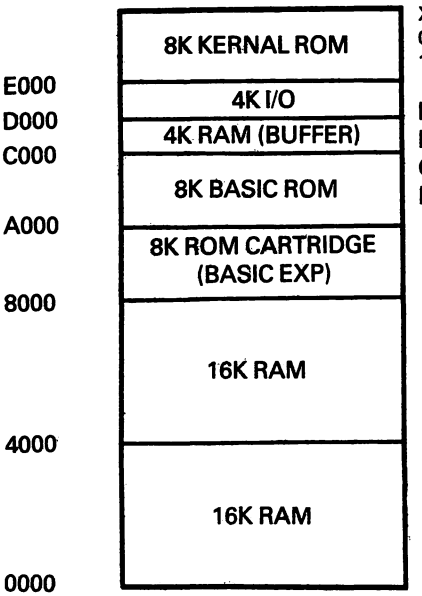
This map is intended for use with softload languages (including CP/M), providing 52K contiguous bytes of user RAM, I/O devices, and I/O driver routines.



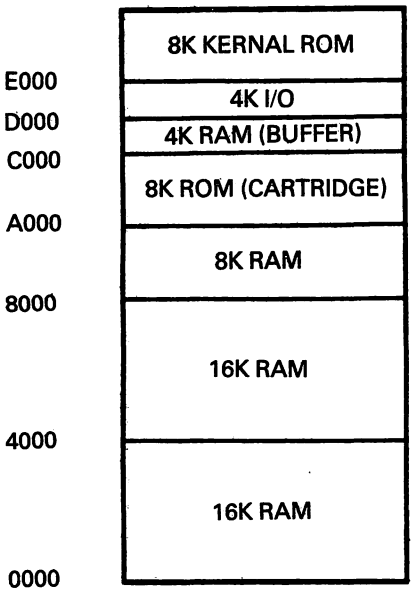
X = DON'T CARE
0 = LOW
1 = HIGH

LORAM = 0
HIRAM = 0
GAME = 1
EXROM = X
OR
LORAM = 0
HIRAM = 0
GAME = X
EXROM = 0

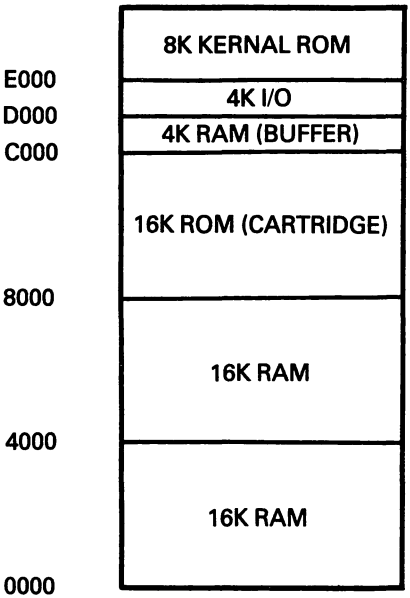
This map gives access to all 64K bytes of RAM. The I/O devices must be banked back into the processor's address space for any I/O operation.



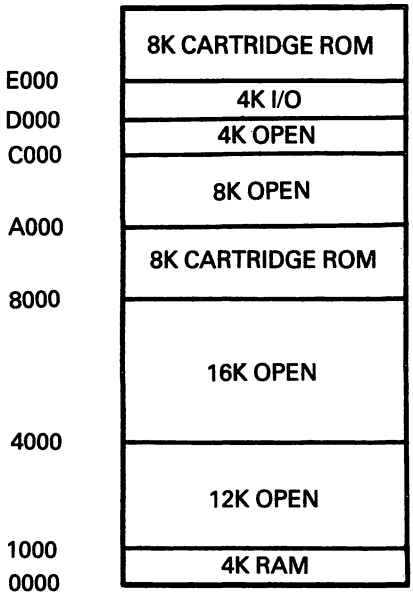
This is the standard configuration for a BASIC system with a BASIC expansion ROM. This map provides 32K contiguous bytes of user RAM and up to 8K bytes of BASIC "enhancement."



This map provides 40K contiguous bytes of user RAM and up to 8K bytes of plug-in ROM for special ROM-based applications which do not require BASIC.



This map provides 32K contiguous bytes of user RAM and up to 16K bytes of plug-in ROM for special ROM-based applications which do not require BASIC (word processors, other languages, etc.).



This is the ULTIMAX video game memory map. Note that the 2K byte "expansion RAM" for the ULTIMAX, if required, is accessed out of the COMMODORE 64 and any RAM in the cartridge is ignored.

APPENDIX D

ASCII and CHR$ Codes

This appendix shows you what characters will appear if you PRINT CHR$(X), for all possible values of X. It will also show the values obtained by typing PRINT ASC("x"), where x is any character you can type. This is useful in evaluating the character received in a GET statement, converting upper/lower case, and printing character based commands (like switch to upper/lower case) that could not be enclosed in quotes.

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	0	)	41	R	82		123		164
	1	*	42	S	83		124		165
	2	+	43	T	84		125		166
	3	,	44	U	85		126		167
	4	–	45	V	86		127		168
	5	.	46	W	87		128		169
	6	/	47	X	88		129		170
	7	0	48	Y	89		130		171
DISABLES	8	1	49	Z	90		131		172
ENABLES	9	2	50	[	91		132		173
	10	3	51	£	92		133		174
	11	4	52	]	93	f1	134		175
	12	5	53	↑	94	f3	135		176
	13	6	54	←	95	f5	136		177
	14	7	55		96	f7	137		178
	15	8	56		97	f2	138		179
	16	9	57		98	f4	139		180
	17	:	58		99	f6	140		181
	18	;	59		100	f8	141		182
	19	<	60		101		142		183
	20	=	61		102		143		184
	21	>	62		103		144		185
	22	?	63		104		145		186
	23	@	64		105		146		187
	24	A	65		106		147		188
	25	B	66		107		148		189
	26	C	67		108		149		190
	27	D	68		109		150		191
	28	E	69		110		151		
	29	F	70		111		152		
	30	G	71		112		153		
	31	H	72		113		154		
SPACE	32	I	73		114		155		
!	33	J	74		115		156		
"	34	K	75		116		157		
#	35	L	76		117		158		
$	36	M	77		118		159		
%	37	N	78		119	SPACE	160		
&	38	O	79		120		161		
'	39	P	80		121		162		
(	40	Q	81		122		163		

CODES	**192-223**	**SAME AS**	**96-127**
CODES	**224-254**	**SAME AS**	**160-190**
CODE	**255**	**SAME AS**	**126**

APPENDIX E

BASIC Keywords

NAME	FORMAT	FUNCTION
ABS	ABS (<expression>)	Returns absolute value
AND	<expression> AND <expression>	Boolean Operator
ASC	ASC (<string$>)	Returns ASCII value of first character of <string$>
ATN	ATN (<number>)	arc tangent
CHR$	CHR$ (<number>)	Converts ASCII to character
CLOSE	CLOSE <file number>	Closes data file or channel
CLR	CLR	Clears RAM
CMD	CMD <file number>, string	Switches primary output device.
CONT	CONT	Restarts program
COS	COS (<number>)	Cosine
DATA	DATA <list of constants>	Stores data
DEF FN	DEF FN<names>(<variable>) = <expression>	User defined function
DIM	DIM <variable>(<subscripts>)	Defines array
END	END	Finishes Program Execution
EXP	EXP(<number>)	Exponential
FN	FN<names>(<number>)	Executes user defined function
FOR..TO..[STEP..]	FOR <variable>=<start> TO <limit>[STEP<increment>]	Sets up FOR..NEXT loop
FRE	FRE(<variable>)	Returns available RAM
GET	GET<variable list>	Reads keyboard buffer
GET#	GET#<file number>,<variable list>	Gets character from device
GOSUB	GOSUB<line number>	Calls BASIC subroutine
GOTO	GOTO<line number> or GOTO<line number>	Transfers control to line number
IF...THEN	IF<expression>THEN<line number> IF<expression>GOTO<line number> IF<expression>THEN<statements>	Decision statement

NAME	FORMAT	FUNCTION
INPUT	INPUT["<prompt>";]<variable list>	Allows input of data from keyboard
INPUT#	INPUT#<file number>,<variable list>	Allows input of data from file
INT	INT(<numeric>)	Integer function
LEFT$	LEFT$(<string>,<integer>)	String slice function
LEN	LEN(<string>)	Length of string
LET	[LET]<variable>=<expression>	Assigns values to variables
LIST	LIST[[<first line>]-[<last line>]]	Lists program lines on screen
LOAD	LOAD["<file name>"][,<device>] [,<address>]	Reads program from tape or disk into memory
LOG	LOG(<numeric>)	Natural logarithm
MID$	MID$(<string>,<startpos>[,<length>])	String slice function
NEW	NEW	Deletes program from memory
NEXT	NEXT[<variable>][,<variable>]...	End of FOR..NEXT loop
NOT	NOT<expression>	Boolean Operator
ON	ON<variable>GOTO/GOSUB <line number>[,<linenumber>]..	Multiway switch
OPEN	OPEN<file num>,[<device>] [,<address>][,"file name[,<type>] [,<mode>]"]	Opens channel for I/O
OR	<operand> OR <operand>	Boolean operator
PEEK	PEEK(<numeric>)	Returns contents of memory
POKE	POKE<location>,<value>	Sets contents of memory
POS	POS(<dummy>)	Returns current cursar position
PRINT	PRINT[<variable>][<,/;><variable>]	Writes data to screen
PRINT#	PRINT#<file number>[<variable>] [<,/;><variable>]....	Writes data to file
READ	READ <variable>[,<variable>]...	Reads from DATA statement
REM	REM[<remark>]	Sets up remark in listing
RESTORE	RESTORE	Restores data pointer to start of program
RETURN	RETURN	Returns from subroutine
RIGHT$	RIGHT$(<string>,<numeric>)	String slice function
RND	RND(<numeric>)	Random number function

NAME	FORMAT	FUNCTION
RUN	RUN[<line number>]	Starts program execution
SAVE	SAVE["<file name>"][,<device>] [,<address>]	Saves program to device
SGN	SGN(<numeric>)	Sign function
SIN	SIN(<numeric>)	Sine function
SPC	SPC(<numeric>)	Within PRINT statement, prints <numeric> spaces
SQR	SQR(<numeric>)	Square Root
STATUS	STATUS	Returns completion STATUS for last I/O operation
STEP	[STEP<expression>]	Defines increment on FOR structure
STOP	STOP	Halts execution of program
STR$	STR$(<numeric>)	Returns string representation of <numeric>
SYS	SYS<memory location>	Calls machine code subroutine
TAB	TAB(<numeric>)	Within PRINT statement moves print position to <numeric>
TAN	TAN(<numeric>)	Tangent
TIME	TI	Reads internal TImer in "jiffies"
TIME$	TI$	TI$ can be set and is updated by TI and returns 6 digits of hours, minutes and seconds.
USR	USR(<numeric>)	Jumps to User defined m/c Subroutine whose address is located at 785-786. <numeric> is loaded into accumulator and after return, USR returns value in accumulator
VAL	VAL[<string>)	Returns numeric VALue of string
VERIFY	VERIFY["<file name>"][,<device>]	Compares BASIC program in RAM with SAVEd program on device
WAIT	WAIT<location>,<mask-1> [,<mask-2>]	WAIT causes suspension until <location> has specified bit pattern

APPENDIX F

Screen Display Codes

The following chart lists all of the characters built in to the Commodore 64 character sets. It shows which numbers should be POKEd into screen memory (locations 1024–2023) to get a desired character. Also shown is which character corresponds to a number PEEKed from the screen.

Two character sets are available, but only one set at a time. This means that you cannot have characters from one set on the screen at the same time you have characters from the other set displayed. The sets are switched by holding down the SHIFT and COMMODORE keys simultaneously.

From BASIC, POKE 53272,29 will switch to upper case mode and POKE 53272,31 switches to lower case.

Any number on the chart may also be displayed in REVERSE. The reverse character code may be obtained by adding 128 to the values shown.

If you want to display a solid circle at location 1504, POKE the code for the circle (81) into location 1504: POKE 1504,81.

There is a corresponding memory location to control the color of each character displayed on the screen (locations 55296–56295). To change the color of the circle to yellow (color code 7) you would POKE the corresponding memory location (55776) with the character color: POKE 55776,7.

SCREEN CODES

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	U	u	21	*		42	?		63
A	a	1	V	v	22	+		43	⊟		64
B	b	2	W	w	23	,		44	♠	A	65
C	c	3	X	x	24	–		45	◫	B	66
D	d	4	Y	y	25	.		46	⊟	C	67
E	e	5	Z	z	26	/		47	▭	D	68
F	f	6	[		27	0		48	▭	E	69
G	g	7	£		28	1		49	▭	F	70
H	h	8	]		29	2		50	◧	G	71
I	i	9	↑		30	3		51	◨	H	72
J	j	10	←		31	4		52	◱	I	73
K	k	11	SPACE		32	5		53	◹	J	74
L	l	12	!		33	6		54	◰	K	75
M	m	13	"		34	7		55	□	L	76
N	n	14	#		35	8		56	◲	M	77
O	o	15	$		36	9		57	◳	N	78
P	p	16	%		37	:		58	□	O	79
Q	q	17	&		38	;		59	❑	P	80
R	r	18	'		39	<		60	■	Q	81
S	s	19	(		40	=		61	▭	R	82
T	t	20	)		41	>		62	▽	S	83

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
	T	84			95			106			117
	U	85	SPACE		96			107			118
	V	86			97			108			119
	W	87			98			109			120
	X	88			99			110			121
	Y	89			100			111		☑	122
	Z	90			101			112			123
		91			102			113			124
		92			103			114			125
		93			104			115			126
		94			105			116			127

Codes from 128-225 are reversed images of codes 0-127.

APPENDIX G

Planning Sheets

In this appendix we present some of the simple planning sheets which we have found useful in our exploration of the Commodore 64. These sheets are

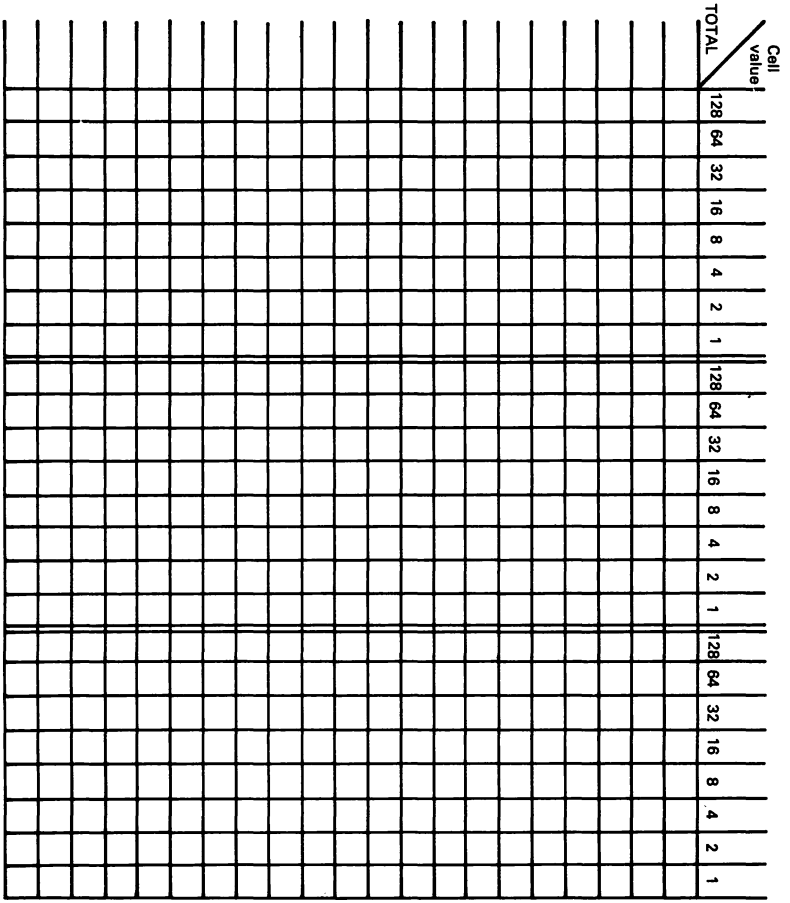
G1 SPRITE PLANNER

G2 USER DEFINED CHARACTER PLANNER

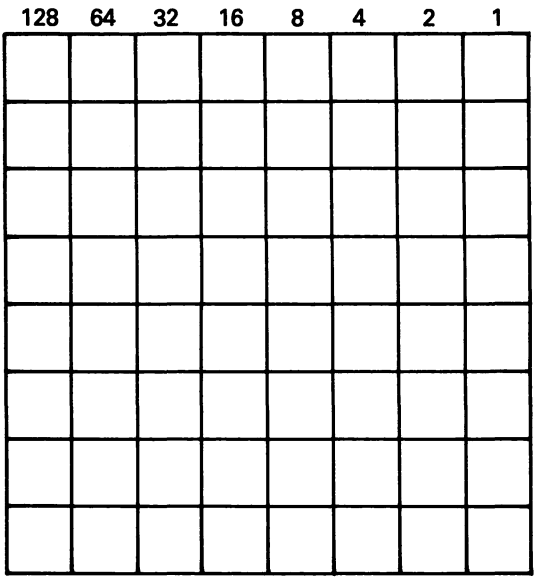
G3 CALC RESULT LAYOUT FORM

G4 SCREEN LAYOUT SHEET

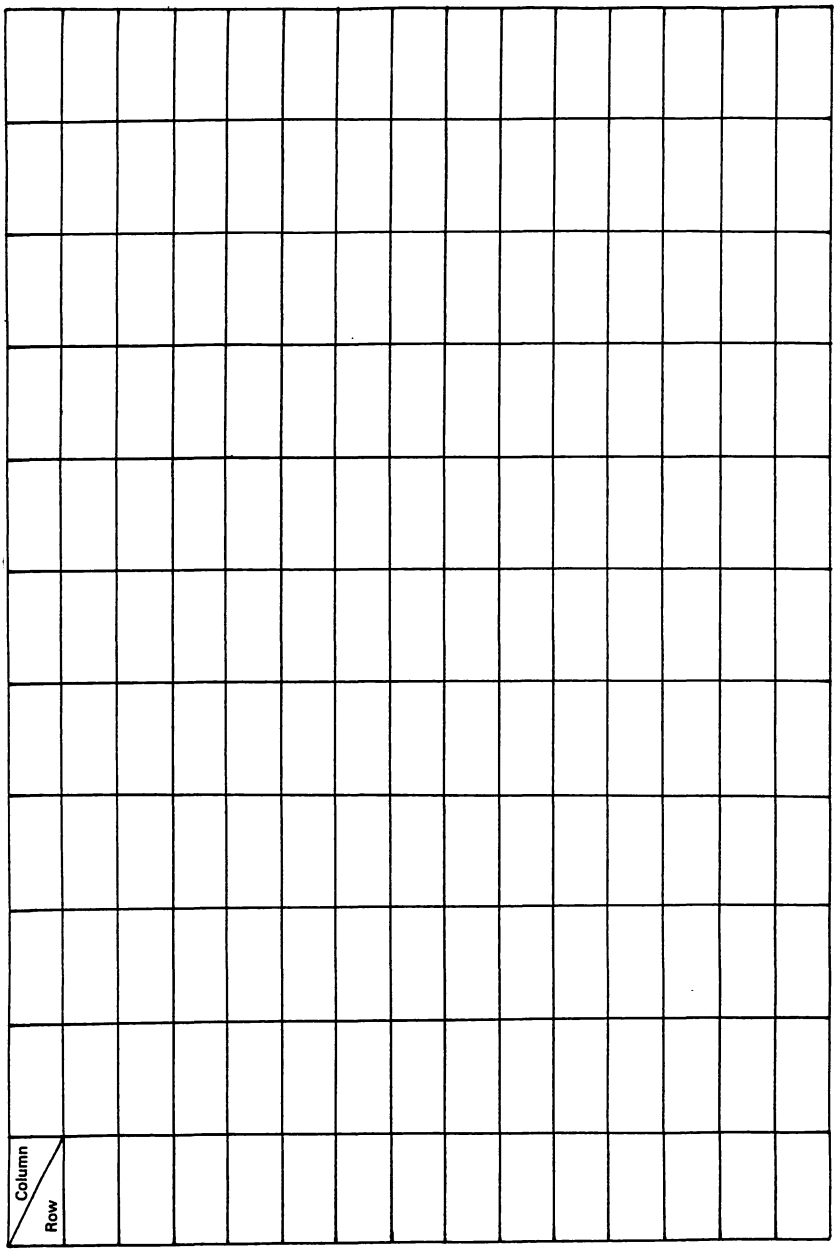
G1. SPRITE PLANNER



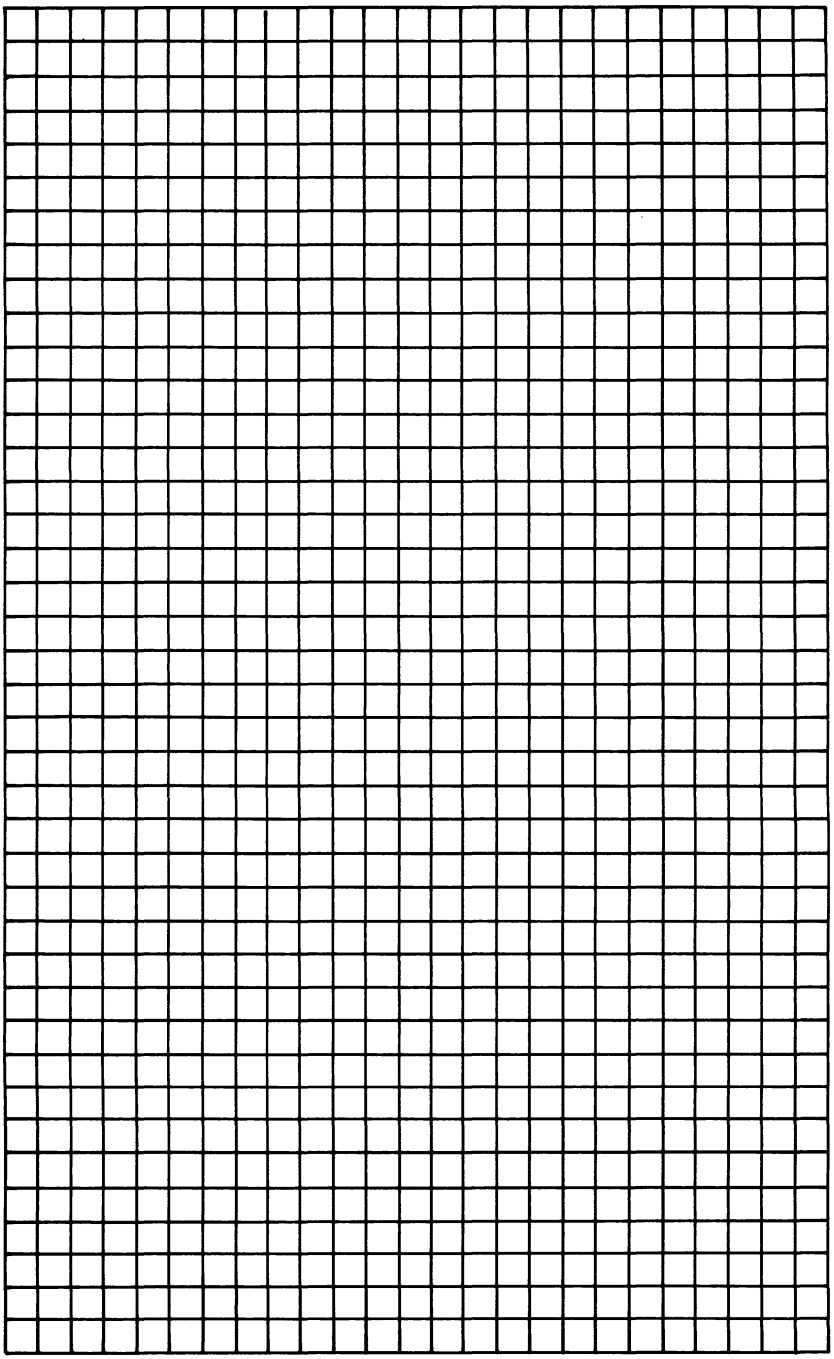
G2. CHARACTER PLANNER



G3. CALC RESULT LAYOUT FORM



G4. SCREEN LAYOUT SHEET



APPENDIX H

Useful ROM Addresses

The following BASIC ROM addresses are included in this book for the readers' use at their own risk. These addresses are not fully documented and some experimentation should be carried out before using them.

Much of the information in this appendix was obtained from 'The Complete COMMODORE 64 ROM Disassembly' by Peter Gerrard and Kevin Bergin (Duckworth Home Computing, 1984), which we found very useful when investigating the micro.

Address	Routine
A000	ROM control vectors
A00C	Keyword action vectors
A052	Function vectors
A080	Operator vectors
A09E	Keywords
A19E	Error messages
A328	Error message vectors
A35B	Misc. messages
A389	Scan stack for FOR/GOSUB
A3B8	Move memory
A3FB	Check stack depth
A408	Check memory space
A435	'out of memory'
A437	Error routine
A469	BREAK entry
A474	'ready'
A480	Ready for BASIC
A49C	Handle new line
A533	Re-chain lines
A560	Receive input line
A579	Crunch tokens
A613	Find Basic Line
A642	[NEW]
A65E	[CLR]
A68E	Back up text pointer
A69C	[LIST]
A742	[FOR]
A7ED	Execute statement
A81D	[RESTORE]
A82C	Break
A82F	[STOP]
A831	[END]
A857	[CONT]
A871	[RUN]
A883	[GOSUB]
A8A0	[GOTO]
A8D2	[RETURN]
A8F8	[DATA]
A906	Scan for next statement
A928	[IF]
A93B	[REM]
A94B	[ON]
A96B	Get fixed point number
A9A5	[LET]
AA80	[PRINT#]
AA86	[CMD]
AAA0	[PRINT]
AB1E	Print string
AB3B	Print format character
AB4D	Bad input routine
AB7B	[GET]
ABA5	[INPUT#]
ABBF	[INPUT]
ABF9	Prompt and input
AC06	[READ]
ACFC	Input error messages
AD1E	[NEXT]
AD78	Type match check
AD9E	Evaluate expression
AEA8	Constant - pi
AEF1	Evaluate within brackets
AEF7	')'
AEFF	Comma..
AF08	Syntax error
AF14	Check range
AF28	Search for variable

Address	Routine
AFA7	Setup FN reference
AFE9	[OR]
AFF0	[AND]
B016	Compare
B081	[DIM]
B08B	Locate variable
B113	Check alphabetic
B11D	Create variable
B194	Array pointer subroutine
B1A5	Value 32768
B1B2	Float - fixed
B1D1	Set up array
B248	'bad subscript'
B24D	'illegal quantitiy'
B34C	Compute array size
B37D	[FRE]
B391	Fix-float
B39E	[POS]
B3A6	Check direct
B3B3	[DEF]
B3E1	Check fn syntax
B374	[FN]
B465	[STR$]
B475	Calculate string vector
B487	Set up string
B4F4	Make room for string
B526	Garbage collection
B5BD	Check salvageability
B606	Collect string
B63D	Concatenate
B67A	Build string to memory
B6A3	Discard unwanted string
B6DB	Clean descriptor stack
B6EC	[CHR$]
B700	[LEFT$]
B72C	[RIGHT$]
B737	[MID$]
B761	Pull string parameters
B77C	[LEN]
B782	Exit string-mode
B78B	[ASC]
B79B	Input byte parameter
B7AD	Perform [VAL]
B7EB	Parameters for POKE/WAIT
B7F7	Float-fixed
B80D	[PEEK]
B824	[POKE]
B82D	[WAIT]
B849	Add 0.5
B850	Subtract-from
B853	[subtract]
B86A	[add]
B947	Complement FAC#1
B97E	'overflow'
B983	multiply by zero byte
B9EA	[LOG]
BA2B	[multiply]
BA59	Multiply-a-bit
BA8C	Memory to FAC#2
BAB7	Adjust FAC#1/#2
BAD4	Underflow/overflow
BAE2	Multiply by 10
BAF9	+ 10 in floating pt.
BAFE	Divide by 10
BB12	[divide]
BBA2	Memory to FAC#1
BBC7	FAC#1 to memory

Address	Routine
BBFC	FAC#2 to FAC#1
BC0C	FAC#1 to FAC#2
BC1B	Round FAC#1
BC2B	Get sign
BC39	[SGN]
BC58	[ABS]
BC5B	Compare FAC#1 to mem.
BC9B	Float-fixed
BCCC	[INT]
BCF3	String to FAC
BD7E	Get ASCII digit
BDC2	Print 'IN...'
BDCD	Print line number
BDDD	Float to ASCII
BF16	Decimal constants
BF3A	TI constants
BF71	[SQR]
BF7B	[power]
BFB4	[negative]
BFED	[EXP]
E043	Series eval. 1
E059	Series eval. 2
E097	[RND]
E12A	[SYS]
E156	[SAVE]
E165	[LOAD]
E1BE	[OPEN]
E1C7	[CLOSE]
E1D4	Parameters for LOAD/SAVE
E206	Check default parameters
E20E	Check for comma
E219	Parameters for OPEN/CLOSE
E264	[COS]
E26B	[SIN]
E2B4	[TAN]
E30E	[ATN]
E37B	Warm restart
E394	Initialize
E3A2	CHRGET for zero page
E3BF	Initialize Basic
E447	Vectors for $300
E453	Initialize vectors
E45F	Power-up message
E500	Get I/O address
E505	Get screen size
E50A	Put/get row/column
E518	Initialize I/O
E544	Clear screen
E566	Home cursor
E56C	Set screen pointers
E5A0	Set I/O defaults
E5B4	input from keyboard
E632	Input from screen
E684	Quote test
E691	Setup screen print
E6B6	Advance cursor
E6ED	Retreat cursor
E701	Back into previous line
E716	Output to screen
E87C	Got to next line
E891	<return>
E8A1	check line decrement
E8B3	Check line increment
E8CB	Set color mode
E8DA	Colour code table
E8EA	Scroll screen
E965	Open space on screen

Address	Routine
E9C8	Move a screen line
E9E0	Synchronize color transfer
E9F0	Interrupt - clock etc.
EA87	Read keyboard
EB79	Keyboard select vectors
EB81	Keyboard 1 - unshifted
EBC2	Keyboard 2 - shifted
EC03	Keyboard 3 - 'comm'
EC44	Graphics/text control
EC4F	Set graphics/text mode
EC78	Keyboard 4
ECB9	Video chip setup
ECE7	Shift/run equivalent
ECF9	Screen in address low
ED0B	Send 'talk'
ED11	Send 'listen'
ED40	Send to serial bus
EDB2	Serial timeout
EDB9	Send listen SA
EDBE	Clear ATN
EDC7	Send talk SA
EDCC	Wait for clock
EDDD	Send serial deferred
EDEF	Send 'untalk'
EE03	Send 'unlisten'
EE13	Receive from serial bus
EE85	Serial clock on
EE8E	Serial clock off
EE97	Serial output '1'
EEA0	Serial output '0'
EEA9	Get serial in and clock
EEB3	Delay 1 ms.
EEBB	RS-232 send
EF06	Send new RS-232 byte
EF2E	No - DSR error
EF33	No - CTS error
EF3B	Disable timer
EF4A	Compute bit count
EF59	RS232 recieve
EF7E	Setup to recieve
EFC4	Receive parity error
EFCC	Receive overflow
EFCF	Receive break
EFD2	Framing error
EFE1	Submit to RS232
F00D	No - DSR error
F017	Send to RS232 buffer
F04D	Input from RS232
F086	Get from RS232
F0A4	Check serial bus idle
F0BD	Messages
F12B	Print if direct
F13E	Get..
F14E	..from RS232
F157	Input
F199	Get..tape/serial/RS232
F1CA	Output..
F1DD	..to tape
F20E	Set input device
F250	Set output device
F291	Close file
F30F	Find file
F31F	Set file values
F32F	Abort all files
F333	Restore default I/O
F34A	Do open file
F3D5	Send SA

Address	Routine
F409	Open RS232
F49E	Load program
F5A4	'searching'
F5C1	Print filename
F5D2	'loading/verifying'
F5DD	Save program
F68F	Print 'saving'
F69B	Bump clock
F6BC	Log PIA key reading
F6DD	Get time
F6E4	Set time
F6ED	Check stop key
F6FB	Output error messages
F72C	Find any tape header
F76A	Write tape header
F7D0	Get buffer address
F7D7	Set buffer start/end pointer
F7EA	Find specific header
F80D	Bump tape pointer
F817	'press play'
F82E	Check tape status
F838	'press record'
F841	Initiate tape read
F864	Initiate tape write
F875	Common tape code
F8D0	Check tape stop
F8E2	Set read timing
F92C	Read tape bits
FA60	Store tape chars.
FB8E	Reset pointer
FB97	New character setup
FBA6	Send transition to tape
FBC8	Write data to tape
FBCD	IRQ entry point
FC57	Write tape leader
FC93	Restore normal IRQ
FCB8	Set IRQ vector
FCCA	Kill tape monitor
FCD1	Check r/w pointer
FCDB	Bump r/w pointer
FCE2	Power reset entry
FD02	Check 8-rom
FD10	8-rom mask
FD15	Kernal reset
FD1A	Kernal move
FD30	Vectors
FD50	Initialize system consts.
FD9B	IRQ vectors
FDA3	Initialize I/O
FDDD	Enable timer
FDF9	Save file name data
FE00	Save file details
FE07	Get status
FE18	Flag status
FE1C	Set status
FE21	Set timeout
FE25	Read/set top of memory
FE27	Read top of memory
FE34	read/set bottom of memory
FE43	NMI entry
FE66	Warm start
FEB6	Reset IRQ and exit
FEBC	Interrupt exit
FEC2	RS-232 timing table
FED6	NMI RS-232 in
FF07	NMI RS-232 out
FF43	Fake IRQ

Address	Routine
FF48	IRQ entry
FF81	Jumbo jump table
FFF6	Hardware vectors

APPENDIX I

The Minigraph Utility

In order to develop high resolution graphics ideas we have used the "MINIGRAPH" program. This program was written by PAUL SCHATZ and can be found in TIM ONOSKO'S book "Commodore 64: getting the most from it", which is published by Prentice/Hall International.

The program loads a set of machine code subroutine into memory locations 36883 to 37681. Using these routine we can reduce our own programs dramatically.

Once the routines have been installed, we can do the following.

* The bit map is enabled by issuing the command SYS 50195.
* The bit map is disabled with SYS 50198.
* The bit map is cleared with SYS 50207.
* Colors displayed on the bit map are set with SYS 50210,F,B. F and B are the Foreground and Background numbers, and have values between 0 and 15.
* A point is plotted on the bit map with the command SYS 50201,X,Y,M. Where X and Y are the horizontal and vertical positions of the point. The horizontal position can be from 0 to 319 and the vertical position can be from 0 to 199. The upper left hand corner is 0, 0. The last variable M, sets the plotting mode. If M=1 the foreground color is used. If M=2 then background color is used. If M=3 then the color of the point is flipped.
* A line is drawn with the command SYS 50204,X,Y,M. Where X and Y are the coordinates of the end point of the line, the color of the line is determined by M, as before.

The program has some error correcting features to help the user to enter the data correctly. The main error correcting feature is a checksum at the end of each DATA line. If an 'OUT OF DATA ERROR' occurs, then the probable cause is a miskeying when inputting the long data lines. Line 50 can then be changed to use different values for K. K is the number of DATA statements in the program, and if K is reduced then the error should be easily identified.

```
10 REM PROGRAM - GRAFICS
20 REM BASED ON MINIGRAPH BY PAUL SCHATZ
30 REM PROGRAM LOADS SOME GRAPHICS PRIMITIVES
40 REM INTO RAM FOR USE BY GRAPHICS PROGRAMS
50 I=50194:L=190:K=17:PRINT
60 PRINT
70 PRINT "          LOADING GRAPHICS"
80 A=0:L=L+10
90 PRINT"*";
100 FOR J=1 TO K
110 READ BY
120 I=I+1:POKE I, BY:A=A+BY
130 NEXT J
140 READ SM
150 IF SM<>A THEN PRINT "DATA ERROR IN LINE"L:STOP
160 IF I=50959 THEN K=16:GOTO 80
170 IF I<>50975 THENGOTO 80
180 PRINT"   GRAPHICS LOADED"
190 NEW
200 DATA 76, 244, 198, 76, 7, 199, 76, 37, 196, 76, 114,
         197, 76, 30, 197, 76, 66, 1941
210 DATA 197, 169, 0, 141, 4, 196, 32, 253, 174, 32, 235,
         183, 224, 200, 144, 3, 76, 2263
220 DATA 26, 199, 142, 3, 196, 166, 20, 165, 21, 240, 8,
         201, 1, 208, 240, 224, 64, 2124
230 DATA 176, 236, 141, 2, 196, 142, 1, 196, 32, 253, 174,
         32, 158, 183, 224, 3, 176, 2325
240 DATA 220, 142, 0, 196, 173, 0, 196, 240, 27, 74, 144,
         12, 32, 174, 196, 32, 135, 1993
250 DATA 196, 29, 158, 196, 145, 251, 96, 32, 174, 196, 32,
         135, 196, 61, 166, 196, 145, 2404
260 DATA 251, 96, 32, 174, 196, 32, 135, 196, 93, 158, 196,
         145, 251, 96, 173, 1, 196, 2421
270 DATA 41, 7, 170, 120, 160, 52, 132, 1, 160, 0, 177, 251,
         160, 55, 132, 1, 88, 1707
280 DATA 160, 0, 96, 128, 64, 32, 16, 8, 4, 2, 1, 127, 191,
         223, 239, 247, 251, 1789
290 DATA 253, 254, 169, 0, 133, 251, 169, 224, 133, 252,
         173, 1, 196, 41, 248, 24, 101, 2622
300 DATA 251, 133, 251, 173, 2, 196, 101, 252, 133, 252,
         173, 3, 196, 72, 41, 7, 24, 2260
310 DATA 101, 251, 133, 251, 144, 2, 230, 252, 104, 74, 74,
         74, 10, 170, 189, 236, 196, 2491
320 DATA 24, 101, 251, 133, 251, 189, 237, 196, 101, 252,
         133, 252, 96, 0, 0, 64, 1, 2281
```

```
330 DATA 128, 2, 192, 3, 0, 5, 64, 6, 128, 7, 192, 8, 0, 10,
        64, 11, 128, 948
340 DATA 12, 192, 13, 0, 15, 64, 16, 128, 17, 192, 18, 0, 20,
        64, 21, 128, 22, 922
350 DATA 192, 23, 0, 25, 64, 26, 128, 27, 192, 28, 0, 30,
        162, 32, 169, 224, 133, 1455
360 DATA 252, 169, 0, 133, 251, 168, 145, 251, 200, 208,
        251, 230, 252, 202, 208, 246, 96, 3262
370 DATA 32, 253, 174, 32, 158, 183, 224, 16, 144, 3, 76,
        26, 199, 96, 169, 192, 133, 2110
380 DATA 252, 169, 0, 133, 251, 32, 52, 197, 138, 10, 10,
        10, 10, 141, 9, 196, 32, 1642
390 DATA 52, 197, 138, 13, 9, 196, 162, 2, 160, 0, 145, 251,
        200, 208, 251, 230, 252, 2466
400 DATA 202, 16, 246, 145, 251, 200, 192, 232, 144, 249,
        96, 169, 0, 141, 8, 196, 141, 2628
410 DATA 10, 196, 32, 253, 174, 32, 235, 183, 224, 200, 144,
        3, 76, 26, 199, 142, 7, 2136
420 DATA 196, 166, 20, 165, 21, 240, 8, 201, 1, 208, 240,
        224, 64, 176, 236, 141, 6, 2313
430 DATA 196, 142, 5, 196, 32, 253, 174, 32, 158, 183, 201,
        3, 176, 220, 142, 0, 196, 2309
440 DATA 173, 5, 196, 56, 237, 1, 196, 141, 12, 196, 173, 6,
        196, 237, 2, 196, 141, 2164
450 DATA 13, 196, 16, 20, 206, 10, 196, 56, 169, 0, 237, 12,
        196, 141, 12, 196, 169, 1845
460 DATA 0, 237, 13, 196, 141, 13, 196, 169, 0, 141, 11, 196,
        173, 7, 196, 56, 237, 1982
470 DATA 3, 196, 141, 14, 196, 173, 8, 196, 237, 4, 196, 141,
        15, 196, 16, 20, 206, 1958
480 DATA 11, 196, 56, 169, 0, 237, 14, 196, 141, 14, 196,
        169, 0, 237, 15, 196, 141, 1988
490 DATA 15, 196, 169, 0, 141, 18, 196, 173, 14, 196, 56,
        237, 12, 196, 173, 15, 196, 2003
500 DATA 237, 13, 196, 144, 27, 174, 14, 196, 173, 12, 196,
        141, 14, 196, 142, 12, 196, 2083
510 DATA 174, 15, 196, 173, 13, 196, 141, 15, 196, 142, 13,
        196, 206, 18, 196, 173, 12, 2075
520 DATA 196, 141, 16, 196, 173, 13, 196, 141, 17, 196, 32,
        91, 196, 173, 18, 196, 208, 2199
530 DATA 18, 173, 1, 196, 205, 5, 196, 208, 27, 173, 2, 196,
        205, 6, 196, 208, 19, 2034
540 DATA 240, 16, 173, 3, 196, 205, 7, 196, 208, 9, 173, 4,
        196, 205, 8, 196, 208, 2243
550 DATA 1, 96, 173, 18, 196, 208, 6, 32, 192, 198, 76, 118,
        198, 32, 218, 198, 32, 1992
```

```
560 DATA 152, 198, 32, 152, 198, 16, 20, 173, 18, 196, 208,
         6, 32, 218, 198, 76, 140, 2033
570 DATA 198, 32, 192, 198, 32, 172, 198, 32, 172, 198, 32,
         91, 196, 76, 64, 198, 173, 2254
580 DATA 16, 196, 56, 237, 14, 196, 141, 16, 196, 173, 17,
         196, 237, 15, 196, 141, 17, 2060
590 DATA 196, 96, 173, 16, 196, 24, 109, 12, 196, 141, 16,
         196, 173, 17, 196, 109, 13, 1879
600 DATA 196, 141, 17, 196, 96, 173, 10, 196, 208, 9, 238, 1,
         196, 208, 3, 238, 2, 2128
610 DATA 196, 96, 173, 1, 196, 208, 3, 206, 2, 196, 206, 1,
         196, 96, 173, 11, 196, 2156
620 DATA 208, 9, 238, 3, 196, 208, 3, 238, 4, 196, 96, 173,
         3, 196, 208, 3, 206, 2188
630 DATA 4, 196, 206, 3, 196, 96, 173, 0, 221, 41, 252, 141,
         0, 221, 169, 59, 141, 2119
640 DATA 17, 208, 169, 8, 141, 24, 208, 96, 173, 0, 221, 9,
         3, 141, 0, 221, 169, 1808
650 DATA 27, 141, 17, 208, 169, 21, 141, 24, 208, 96, 32, 7,
         199, 76, 72, 178, 1616
```

This program must be entered before attempting to develop any of the high resolution programs noted in the book. Some of the Quizzes in the high resolution chapter have not been supplied with suggested solutions. These are given in our companion book "100 Programs For The Commodore 64", published by Prentice/Hall International. To duplicate such material here would make this book too long, and sufficient information is given in the relevant chapters to enable readers to develop their own solutions.

ROUTINE NAME	DESCRIPTION	DATA REQUIRED	REGISTERS USED	STACK	CALL ADDRESS
ACPTR	Gets data from serial bus. Can be used to access disk. Data is returned in the accumulator. The TALK routine must be used to command a device to send data. Any secondary address must be sent with TSKA. Errors are returned via the status register.	None	A,X	13 bytes	$FFA5; 65445
CHKIN	Defines a channel as input. Once a file is opened using OPEN, it can be set to input with CHKIN. Errors are returned via the status register.	Logical file number must be in X register.	A,X	None	$FFC6; 65478
CHKOUT	Defines an opened channel as output. Not needed when data is sent to screen. Errors are returned via the status register.	Logical file number must be in X register.	A,X	4 bytes minimum	$FFC9; 65481
CHRIN	Gets a character from a previously opened input channel. Data is retuned in the accumulator.	None	A,X	7 bytes minimum	$FFCF; 65487

ROUTINE NAME	DESCRIPTION	DATA REQUIRED	REGISTERS USED	STACK	CALL ADDRESS
CHROUT	Outputs a character to a previously opened output channel. Errors are returned via the status register. Use OPEN and CHKOUT first unless output is to the screen. All open output channels to which data is not to be sent must be closed using CLRCHN.	Data to be sent must be in accumulator	A	8 bytes minimum	$FFD2; 65490
CINT	Initializes screen editor and video chip. This routine is called by program cartridges.	None	A,X,Y	4 bytes	$FF81; 65409
CIOUT	Transmits a byte over the serial bus. LISTEN must first be used to command a serial device to prepare to receive data. SECOND may also be required.	Data to be sent is in the accumulator.	A	5 bytes	$FFA8; 65448
CLALL	Closes all open files. CLRCHN is called automatically to close I/O channels.	None	A,X	11 bytes	$FFE7; 65511
CLOSE	Closes a logical file. Error Errors are returned via the staus register.	File number is loaded in the accumulator.	A,X,Y	2 bytes minimum	$FFC3; 65475

ROUTINE NAME	DESCRIPTION	DATA REQUIRED	REGISTERS USED	STACK	CALL ADDRESS
CLRCHN	Closes all I/O channels and restores I/O to default values.	None	A,X	9 bytes	$FFCC; 65484
GETIN	Gets a character from an input channel. The channel must first be opened and CHKIN used. If the channel is the keyboard, one character is removed from the queue. The character is returned in the accumulator. Errors are returned via the status register.	None	A,X,Y	7 bytes	$FFE4; 65508
IOBASE	Defines I/O memory base. X and Y registers are loaded with low and high bytes of base address.	None	X,Y	2 bytes	$FFF3; 65523
IOINIT	Initializes all I/O devices and routines. Used in program cartridges.	None	A,X,Y	None	$FF84; 65412
LISTEN	Commands device on serial bus to listen. Errors are returned via the status register.	Device number must be placed in accumulator	A	None	$FFB1; 65457

ROUTINE NAME	DESCRIPTION	DATA REQUIRED	REGISTERS USED	STACK	CALL ADDRESS
LOAD	Loads RAM from, or verifies RAM contents with, data from an input device. SETLFS and SETNAM must be used and the device channel opened first. Errors are returned via the status register.	Accumulator is loaded with 0 for load and 1 for verify. If relocated load is required, X and Y must be set to start address.	A,X,Y	None	$FFD5; 65493
MEMBOT	Sets or reads address of bottom of memory from or into X(low) and Y(high) registers.	Carry flag set for read and clear for set.	X,Y	None	$FF9C; 65436
MEMTOP	Sets or reads address of top of memory from or into X(low) and Y(high) registers.	Carry flag set for read and clear for set.	X,Y	None	$FF99; 65433
OPEN	Opens a logical file. SETLFS and SETNAM must be used first. Errors are returned via the status register.	None	A,X,Y	None	$FFC0; 65472
PLOT	Sets cursor to location given by X and Y registers, or puts current cursor position into X and Y depending on the status of the carry flag. X holds the row and Y the column number.	If carry flag is set, cursor position is put in X and Y otherwise the cursor is repositioned.	A,X,Y	2 bytes	$FFF0; 65520

ROUTINE NAME	DESCRIPTION	DATA REQUIRED	REGISTERS USED	STACK	CALL ADDRESS
RAMTAS	Tests RAM and sets top and bottom memory pointers. May form part of initialization of program cartridge.	None	A,X,Y	2 bytes	$FF87; 65414
RDTIM	Reads the system clock value into A, X and Y. A holds the MSB, Y holds the LSB. The system clock has a resolution of 1/60th of a second.	None	A,X,Y	2 bytes	$FFDE; 65502
READST	Reads the status word. Used in error handling. See table at the end of this appendix.	None	A	2 bytes	$FFB7; 65463
RESTOR	Restores default values of all system vectors.	None	A,X,Y	2 bytes	$FF8A; 65418
SAVE	Saves RAM to an I/O device. SETLFS is a prerequisite, as is SETNAM, except, in the latter case, where RAM is saved to cassette recorder.	A holds a page zero offset to point to the RAM start. X and Y hold the final address.	A,X,Y	None	$FFD8; 65496

ROUTINE NAME	DESCRIPTION	DATA REQUIRED	REGISTERS USED	STACK	CALL ADDRESS
SCNKEY	Scans the keyboard and stores the ASCII value of any key found in the keyboard queue. Normally called by interrupt handler; called directly only if IRQ interrupt bypassed. IOINIT is normal preparatory routine in this case.	None	A,X,Y	5 bytes	$FF9F; 65439
SCREEN	Returns screen format – 40 in X and 25 in Y. Implemented to provide compatibility with 80 column machines.	None	A,X,Y	2 bytes	$FFED; 65517
SECOND	Sends secondary address after after LISTEN. Cannot be used after TALK. Errors in status.	A contains the address.	A	8 bytes	$FF93; 65427
SETLFS	Sets logical file number, device number and secondary address for other routines (for example OPEN).	A contains the device number, X the file number and Y the secondary address (255 if none).	A,X,Y	2 bytes	$FF93; 65427

ROUTINE NAME	DESCRIPTION	DATA REQUIRED	REGISTERS USED	STACK	CALL ADDRESS
SETMSG	Controls the printing of error and control messages by KERNAL routines.	If bit 7 of A is 1 an error message is selected. If bit 6 is set a control message is printed.	A	2 bytes	$FF90; 65424
SETNAM	Sets up the file name for the OPEN, SAVE and LOAD routines.	Length of name in A. X and Y hold address of file name.	A,X,Y	2 bytes	$FFBD; 65469
SETTIM	Sets system clock. Clock is measured in jiffies (1/60th of a second) and held as a three byte number.	A (MSB), X and Y (LSB) hold the setting.	A,X,Y	2 bytes	$FFDB; 65499
SETTMO	Sets or resets IEEE bus card timeout flag. If flag is set computer waits for a device for 64 milliseconds. If there is no response computer exits handshake sequence.	If bit 7 of A is 0 flag is set. If bit 7 is 1 flag is reset.	A	2 bytes	$FFA2; 65442

ROUTINE NAME	DESCRIPTION	DATA REQUIRED	REGISTERS USED	STACK	CALL ADDRESS
STOP	Checks whether the STOP key was pressed during UDTIM. IF key was pressed zero flag is set. Otherwise accumulator holds last row of keyboard scan.	None	A,X	None	$FFE1; 65505
TALK	Commands a device on the serial bus to talk.	A contains the device number.	A	8 bytes	$FFB4; 65460
TKSA	Sends secondary address to device commanded to talk. Cannot be used after LISTEN. Errors returned via status register.	A contains the address.	A	8 bytes	$FF96; 65430
UDTIM	Updates the system clock. Normally called by interrupt routine. If IRQ is disabled this routine must be called by user programs every 1/60th second to maintain clock accuracy.	None	A,X,Y	2 bytes	$FFEA; 65514
UNLSN	Commands all devices on the serial bus to stop receiving data from the computer.	None	A	8 bytes	$FFAE; 65454

ROUTINE NAME	DESCRIPTION	DATA REQUIRED	REGISTERS USED	STACK	CALL ADDRESS
UNTLK	Commands all devices on the serial bus, previously set to talk, to stop sending data.	None	A	8 bytes	$FFAB; 65451
VECTOR	If carry bit is set this stores RAM vectors in list pointed to by X and Y. If flag is reset user list is transferred to RAM vectors.	X and Y must hold address of user list.	A,X,Y	2 bytes	$FF8D; 65421

STATUS WORD ERROR TABLE (SEE READST)

STATUS BIT	VALUE	CASSETTE READ	SERIAL READ/WRITE	TAPE LOAD/VERIFY
0	1		Time out write.	
1	2		Time out read.	
2	4	Short block.		Short block.
3	8	Long block.		Long block.
4	16	Unrecoverable read error.		Any mismatch.
5	32	Checksum error.		Checksum error.
6	64	End of file.	EOI line.	
7	-128	End of tape.	Device not present.	End of tape.

APPENDIX K

Minimum Facility Monitor

This program allows us to poke about in the micro's memory. We can examine portions of memory, change portions of memory. You can also save and load portions of memory to tape.

This is where micros can become fun. We can use this program to change our machine, but bear in mind that it is possible to damage our equipment if we poke about at random.

```
100 REM PROGRAM - MONITOR
110 PRINT
120 PRINT"MONITOR"
130 PRINT"THIS PROGRAM IS USED TO ALLOW YOU TO:"
140 PRINT:PRINT:PRINT"1. INPUT A MACHINE CODE PROGRAM"
150 PRINT"2. EXAMINE A PORTION OF MEMORY"
160 PRINT"3. SAVE A BLOCK OF RAM MEMORY"
170 PRINT"4. LOAD A BLOCK OF RAM MEMORY"
180 PRINT"5. END THIS SESSION"
190 PRINT"PRESS THE APPROPRIATE NUMBER"
200 GET R$:IF R$="" THEN 200
210 IF R$="5" THEN END
220 ON VAL(R$) GOSUB 1000,2000,3000,4000
230 IF VAL(R$)<1 OR VAL(R$)>5 THEN GOTO 100
240 GOTO 100
999 REM ROUTINE TO INPUT A SERIES OF BYTES
1000 PRINT
1010 PRINT"THIS ROUTINE ALLOWS YOU TO INPUT A"
1020 PRINT"SERIES OF BYTES IN EITHER HEX OR DECIMAL"
1030 PRINT"INPUT 'D' FOR DECIMAL"
1040 PRINT"INPUT 'H' FOR HEXADECIMAL"
1050 GET R$:IF R$="" OR (R$<>"D" AND R$<>"H") THEN 1050
1060 IF R$="D" THEN GOSUB 1300
1070 IF R$="H" THEN GOSUB 1600
1080 RETURN
1299 REM DECIMAL INPUT SUBROUTINE
1300 PRINT
1310 PRINT"WHAT IS THE STARTING ADDRESS OF BYTES"
1320 INPUT S
1330 PRINT "ENTER BYTES USE . TO FINISH"
1340 PRINT S;" ";PEEK(S);" ";
1350 INPUT B$
1360 IF B$="." THEN RETURN
1370 POKE S,VAL(B$)
1380 S=S+1
1390 GOTO 1340
```

```
1599 REM HEX INPUT SUBROUTINE
1600 PRINT
1610 PRINT"WHAT IS THE STARTING ADDRESS OF BYTES"
1620 INPUT S$
1630 PRINT "ENTER BYTES USE . TO FINISH"
1640 H$=S$
1650 GOSUB 5000:S=DC
1660 PRINT H$;" ";PEEK(S);" ";
1670 INPUT B$
1680 IF B$="." THEN RETURN
1690 H$=B$
1700 GOSUB 5000
1710 POKE S,DC
1720 S=S+1
1730 D$=MID$(STR$(S),2):GOSUB 6000
1740 GOTO 1650
1999 REM ROUTINE TO EXAMINE A SERIES OF BYTES
2000 PRINT
2010 PRINT"THIS ROUTINE ALLOWS YOU TO EXAMINE"
2020 PRINT"A SERIES OF BYTES IN BOTH HEX AND"
2030 PRINT"DECIMAL"
2040 PRINT"WHAT IS THE STARTING ADDRESS OF BYTES"
2050 INPUT "IN DECIMAL ";S
2060 PRINT "PRESS SPACE FOR MORE RETURN TO FINISH"
2065 PRINT"ADDRESS","DEC","HEX","ASC"
2070 B=PEEK(S)
2080 D$=MID$(STR$(B),2):GOSUB 6000
2090 PRINT S,B,H$,CHR$(B)
2100 GET A$:IF A$="" THEN 2100
2110 IF ASC(A$)=13 THEN 2140
2120 S=S+1
2130 GOTO 2070
2140 PRINT"PRESS ANY KEY TO RETURN TO MAIN MENU"
2150 GET A$:IF A$="" THEN 2150
2160 RETURN
2999 REM ROUTINE TO SAVE A BLOCK OF RAM MEMORY
3000 PRINT
3010 INPUT "ENTER START ADDRESS OF BLOCK";S
3020 INPUT "ENTER FINISH ADDRESS OF BLOCK";F
3030 L=F-S:IF L<0 THEN 3000
3040 INPUT "ENTER FILE NAME";F$
3050 OPEN 1,1,2,F$
3060 PRINT#1,S
3070 PRINT#1,F
3080 FOR I=1 TO L+1
3090 B=PEEK(S+I-1)
3100 PRINT#1,B
```

```
3110 NEXT I
3120 CLOSE 1
3130 PRINT "BLOCK WRITTEN TO TAPE"
3140 PRINT"PRESS ANY KEY TO RETURN TO MAIN MENU"
3150 GET A$:IF A$="" THEN 3150
3160 RETURN
3999 REM ROUTINE TO LOAD A BLOCK OF RAM MEMORY
4000 PRINT
4010 INPUT "ENTER FILE NAME";F$
4020 OPEN 1,1,0,F$
4030 INPUT#1,S
4040 INPUT#1,F
4050 L=F-S
4060 FOR I=1 TO L+1
4070 INPUT#1,B
4080 POKE S+I-1,B
4090 NEXT I
4100 PRINT "BLOCK WRITTEN TO RAM"
4110 PRINT"PRESS ANY KEY TO RETURN TO MAIN MENU"
4120 GET A$:IF A$="" THEN 4120
4130 RETURN
4990 REM SUBROUTINE TO CONVERT HEX TO DECIMAL
4991 REM HEX NUMBER ENTERED AS H$
4992 REM DECIMAL NUMBER RETURNED AS DC
4993 REM LOCAL VARIABLES: N,ER,HX,D
5000 DC=0
5010 FOR N=1 TO LEN(H$)
5020 ER=1
5030 HX=ASC(MID$(H$,N,1))
5040 IF HX>47 AND HX<58 THEN HX=HX-48:ERR=0
5050 IF HX>64 AND HX<71 THEN HX=HX-55:ERR=0
5060 IF ER=1 THEN PRINT"INVALID HEX":RETURN
5070 D=HX*16^(LEN(H$)-N)
5080 DC=DC+D
5090 NEXT N
5100 RETURN
5990 REM SUBROUTINE TO CONVERT DECIMAL TO HEX
5991 REM DECIMAL ENTERED AS D$
5992 REM HEX RETURNED AS H$
5993 REM LOCAL VARIABLES: N,D,DC,C
6000 FOR N=1 TO LEN(D$)
6010 C=ASC(MID$(D$,N,1))
6020 IF C<48 OR C>57 THEN PRINT "INVALID DECIMAL":RETURN
6030 NEXT N
6040 DC=VAL(D$)
6050 H$=""
6060 D=16*(DC/16-INT(DC/16))
```

```
6070 IF DC<16 THEN D=DC
6080 DC=INT(DC/16)
6090 D=D+48
6100 IF D>57 THEN D=D+7
6110 H$=CHR$(D)+H$
6120 IF DC>0 THEN GOTO 6060
6130 RETURN
```

APPENDIX L

Delete and Renumber Utilities

The two routines presented in this appendix allow the programmer to delete sections of code and also to renumber programs. Both routines are written in BASIC and could be considered as designs for the equivalent machine code routines. Since they are written in BASIC the last thing that they both do is to remove themselves from the computer. This is done by setting pointers and markers. These routines could be used to provide some insight in the the operation of the BASIC interpreter.

DELETE ROUTINE

This program can be used in program development to allow us to delete portions of a program when we no longer require them. Typically we would use extra lines of code when debugging a program. After the program has been debugged we would delete our debug lines.

The routine is appended to our program when we are at the development stage. When we execute the routine it deletes our section of code and then removes itself. Thus in the present state of the routine we can use it only once.

The program makes use of some of the special memory locations in the C-64 to carry out the delete. To delete a section of program type GOTO 60000.

```
10 REM SAMPLE LINE
20 REM SAMPLE LINE
30 REM SAMPLE LINE
40 REM SAMPLE LINE
50 REM SAMPLE LINE
60 REM SAMPLE LINE
60000 REM DELETE ROUTINE
60010 REM APPEND THIS ROUTINE TO YOUR
60020 REM PROGRAM WHILE DEVELOPING CODE
60030 REM THE ROUTINE DELETES ITSELF
60040 REM AFTER IT HAS BEEN USED
60050 PRINT "ENTER RANGE OF DELETE"
60060 INPUT"STARTING LINE NUMBER";S
60070 INPUT "FINISHING LINE NUMBER";F
60080 PT=2049
```

```
60090 AD=PEEK(PT)+PEEK(PT+1)*256
60100 NO=PEEK(PT+2)+PEEK(PT+3)*256
60110 IF NO<S THEN PT=AD:GOTO 60090
60120 FRS=PT:REM FRS IS STARTING ADDRESS OF LINE TO BE
DELETED
60130 PT=AD
60140 AD=PEEK(PT)+PEEK(PT+1)*256
60150 NO=PEEK(PT+2)+PEEK(PT+3)*256
60160 IF NO<F AND NO>0 THEN PT=AD:GOTO 60140
60170 LAS=PT:REM LAS=STARTING ADDRESS OF LAST LINE TO BE
DELETED
60180 REM AD = STARTING ADDRESS OF REST OF PROGRAM
60190 BL=AD-FRS:REM BL=LENGTH OF BLOCK TO BE DELETED
60200 PT=AD
60210 AD=PEEK(PT)+PEEK(PT+1)*256
60220 NO=PEEK(PT+2)+PEEK(PT+3)*256
60230 IF NO=60000 THEN GOTO 60360
60240 NAD=AD-BL
60250 POKE(PT-BL),(NAD-INT(NAD/256)*256)
60260 POKE(PT-BL+1),INT(NAD/256)
60270 POKE(PT-BL+2),PEEK(PT+2)
60280 POKE(PT-BL+3),PEEK(PT+3)
60290 REM NOW THE REST OF THE LINE
60300 I=4
60310 BT=PEEK(PT+I)
60320 IF BT=0 THEN POKE(PT-BL+I),0:GOTO 60200
60330 POKE(PT-BL+I),BT
60340 I=I+1
60350 GOTO 60310
60360 REM POKE TO END PROGRAM
60370 POKE (PT-BL),0:POKE(PT-BL+1),0
60380 REM TIDY UP BASIC POINTERS
60390 BL=BL+1482:REM BLOCK LENGTH INCREASED BY ROUTINE
LENGTH
60410 VL=PEEK(47)+PEEK(48)*256-BL
60420 POKE(47),(VL-INT(VL/256)*256)
60430 POKE(48),INT(VL/256)
60440 VL=PEEK(49)+PEEK(50)*256-BL
60450 POKE(49),(VL-INT(VL/256)*256)
60460 POKE(50),INT(VL/256)
60470 VL=PEEK(45)+PEEK(46)*256-BL+84
60480 POKE(49152),(VL-INT(VL/256)*256)
60490 POKE(49153),INT(VL/256)
60500 POKE45,PEEK(49152)
60510 POKE46,PEEK(49153)
60520 END
```

RENUMBER ROUTINE

This is a very useful routine to have available when developing programs. The program can be renumbered by simply calling this routine.

The program will renumber both GOTOs and GOSUBs, but will not renumber an IF...THEN linenumber unless the GOTO is present.

The routine deletes itself after the renumber has been completed.

A short sample program is shown in the code.

```
100 GOTO300
200 GOSUB400
300 GOTO600
400 REMSUB
500 RETURN
600 END
9999 REM RENUMBER PROGRAM
10000 REM PLACE THIS ROUTINE AT THE END
10001 REM OF YOUR PROGRAM THEN GOTO 10000
10002 BE=10:IN=10:REM STARTING LINE NUMBER AND INCREMENT
10003 DIM TB(200,2):I=1:REM TABLE OF LINE NOS OLD AND
NEW
10004 LI=BE:TB(I,1)=LI:REM LINE NUMBER
10005 PT=2049:REM POINTER TO MEMORY
10006 LO=PEEK(PT):HI=PEEK(PT+1)
10007 AD=LO+HI*256:REM STARTING ADDRESS OF NEXT LINE
10008 LO=PEEK(PT+2):HI=PEEK(PT+3)
10009 NU=LO+HI*256:TB(I,2)=NU:REM VALUE OF CURRENT LINE
NUMBER
10010 IF NU=9999 THEN 10016
10011 HI=INT(LI/256):LO=LI-HI*256:REM NEW LINE NUMBER
10012 POKE (PT+2),LO:POKE (PT+3),HI
10013 PT=AD:LI=LI+IN
10014 I=I+1:TB(I,1)=LI
10015 GOTO 10006
10016 REM NOW WE RENUMBER THE GOTOS AND GOSUBS
10017 PT=2049
10018 LO=PEEK(PT):HI=PEEK(PT+1)
10019 AD=LO+HI*256
10020 LO=PEEK(PT+2):HI=PEEK(PT+3)
```

```
10021 NU=LO+HI*256
10022 IF NU=9999 THEN 20000
10023 PT=PT+4
10024 IF PEEK(PT)=137 OR PEEK(PT)=141 GOTO 10027
10025 IF PEEK(PT)=0 THEN PT=PT+1:GOTO 10018
10026 PT=PT+1:GOTO 10024
10027 D$="":K=0
10028 PT=PT+1
10029 IF PEEK(PT)<48 OR PEEK(PT)>57 GOTO 10028
10030 K=K+1:D$=D$+CHR$(PEEK(PT)):PT=PT+1
10031 IF PEEK(PT)>47 AND PEEK(PT)<58 GOTO 10030
10032 D=VAL(D$)
10033 FOR J=1 TO I
10034 IF TB(J,2)=D THEN D=TB(J,1):J=I
10035 NEXT J
10036 D$=STR$(D)
10037 IF LEN(D$)<K THEN D$=" "+D$:GOTO 10037
10038 FOR J=1 TO K
10039 POKE (PT-K-1+J),(ASC(MID$(D$,J,1)))
10040 NEXT J
10041 IF PEEK(PT)=ASC(",") THEN GOTO10027
10042 GOTO 10024
20000 POKE PT,0:POKE (PT+1),0:REM PUT IN END OF PROGRAM
MARKER
```

Index

W

Z

PRENTICE
HALL
INTERNATIONAL
PERSONAL
COMPUTER
BOOK

COMMODORE 64 ADVANCED USER GUIDE is for all those who want to make greater use of the many features and facilities of the Commodore 64. If you find programming in BASIC too limited, then this book is for you.

The first chapters take you deeper into the machine. They explain what machine code and assembly language are, and how to use them.

You then learn how to combine BASIC and machine-code routines for faster, more efficient programs.

Two chapters on sound and graphics show you how to program SID and VIC, and how to develop high-resolution graphics, including 3-D, solid drawing and perspective.

The final chapters examine word processing, spreadsheet and database packages and the peripheral devices available for the Commodore 64.

ISBN 0-13-152026-1

cover design: The Pinpoint Design Company