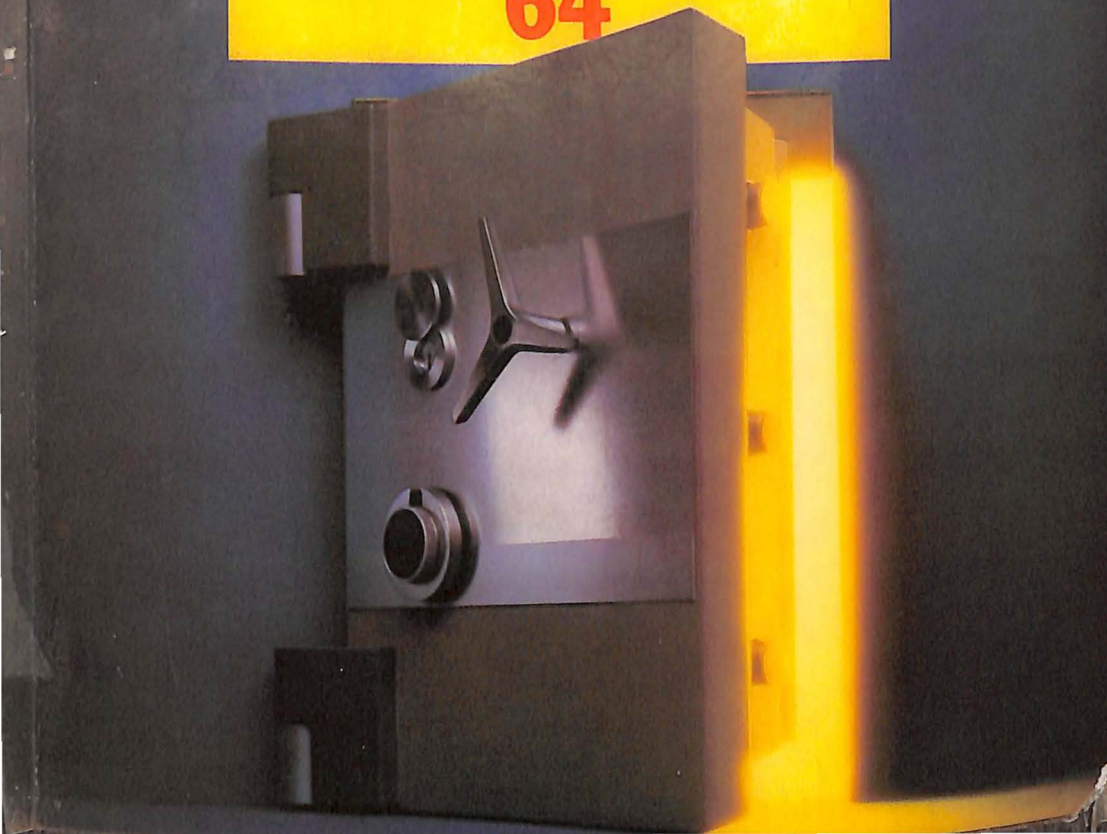
Pan

PERSONAL COMPUTER

COMPUTER NEWS LIBRARY

JOHN P. GIBBONS

CRACKING THE CODE on the COMMODORE 64



Contents

Introduction

The explosion of interest in microcomputing over the last few years has produced large numbers of BASIC programmers who find that this high-level language, once the learning stages have been passed, fails to deliver the flexibility, speed and power of the processor chip effectively. This is especially true of the 64, where the standard BASIC is, to put it politely, somewhat lacking in sophistication. Since the 64 is in fact an extremely sophisticated computer, programming in a language that cannot make effective use of the 64's capabilities produces frustration in the user as a result of the slow, ungainly mass of PEEKS and POKES that results from any attempt to manipulate the machine's potential. The answer lies in the use of machine code.

This is fast, since it operates the chips that do the work directly, and versatile, allowing the user to control every aspect of the computer. It should also be noted that machine code produces lengthy programs and is somewhat tedious to write, because of the simplicity of the individual instructions. However, the results are rewarding and this book is designed to make the process as painless as possible, as well as providing some major utilities and BASIC enhancements, plus a machine code monitor, as complete and ready-to-enter programs. Effectively, it provides both the tools you need to get started in machine code programming and a set of additional BASIC commands and functions (machine code programs that are activated from BASIC statements) while at the same time, these programs and their descriptions provide the examples of machine code programming in practice that help you grasp the process.

The book starts with an overview of the 64 system, introducing the 6510 processor and other chips, and continues with the elements of machine code and assembler programming and their use. The presentation of the instruction set, registers, addressing modes and the other bits and pieces of the machine code programmer's stock-in-trade is somewhat circular, in that, for example, it makes no sense to describe the registers without reference to the instructions which operate upon them, and the instructions cannot be described without assuming knowledge of the registers. The sequence

adopted first introduces all the concepts and then goes through them again in detail, but you would be well advised to read through this whole section and then work through it again to make sure you understand all the pieces of the jigsaw and how they fit together.

After a couple of simple examples, I've chosen to present a major program, 'Wedge-MON', both because it gives you a machine code monitor that provides the facilities needed to work effectively with machine code, and because it illustrates the way in which a large program is built up. It is no more difficult to understand the short sections of code into which the program is broken up in their context than it would be if they were presented as isolated operations, and they make more sense as part of a program. The other vital and often omitted element of serious machine code programming that can only be presented in this way is the structuring of a program, into loops, branches and subroutines, and the algorithms or recipes for the operations that must be defined before any coding is attempted. BASIC programmers are too often the victims of the 'hackers syndrome' – sitting down at the keyboard before the program is well thought out and hacking it into shape (or chaos) as they go. This is a recipe for disaster if you are working with machine code, since it lacks the easy visibility, helpful error messages and simple editing and modification facilities of BASIC. Careful thought and planning is needed, so there is an emphasis on program design.

The next three chapters are concerned with adding commands to BASIC. The core command-handling routine is presented first, and then two sets of command routines, the first a set of programmer's utilities, and the second a group of graphics, sound and sprite commands.

The appendices contain reference material, and two further programs which did not fit comfortably within the text. There is also an appendix on number systems, to which anyone not familiar with binary and hexadecimal should turn after this introduction, since such familiarity is assumed in what follows.

All programs are given in two forms, assembler code and BASIC loader programs. The assembler code may be entered directly by any reader possessing an assembler program, and is used to present the program code in a comprehensible form. The BASIC loader programs simply insert the correct values into memory, and allow the programs to be entered by keying in and RUNning the program. The DATA they contain, which is the machine code produced by the assembled listing of assembler code, is meaningless to the eye, whereas assembler code is totally explicit once the conventions are understood. The particular assembler program used for this book

uses standard conventions, and these are described before we start to use the listings. This is another example of the circularity problem mentioned earlier, and the description will not be clear until you have worked through what follows. However, it seemed better to place it all together for easy reference than scatter it around the text.

A great deal of information on various aspects of the 64 is incorporated in the programs and text. All essential details are given, but the *Commodore 64 Programmer's Reference Guide* is assumed to be available as backup information. This is an essential reference for any 64 user, and especially for the machine code programmer. The standard work on the 6502 chip itself (of which the 6510 is a twin apart from one minor detail) is *Programming the 6502* by R. Zaks, published by Sybex, which can be consulted for the more esoteric aspects of machine code programming on the chip itself, rather than, as is done in this book, treating the 64 system as a whole.

1 An overview of the CBM 64 system

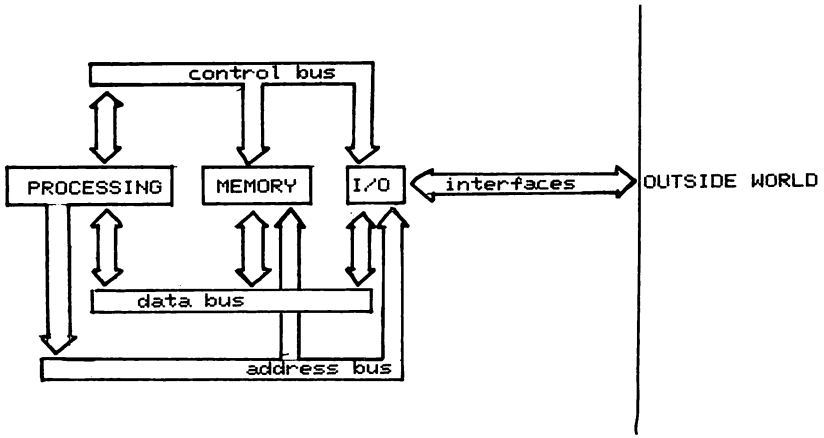
The first step in any study of machine code programming should be an overview of the machine that we wish our programs to control. This may seem obvious, but it is one respect in which machine code programming differs from programming in a high-level language such as BASIC where we may cheerfully take for granted much of the 'nuts and bolts' level of the operation of the computer. This said, 64 BASIC programmers will probably already have a better understanding of their machine than users of some other micros, since the standard 64 BASIC is relatively primitive in that direct recourse to the hardware via PEEKS and POKES is virtually essential if any useful or complex programming is to be done. The essential tool of the 64 programmer, the *Commodore 64 Programmer's Reference Guide* contains further details not covered here.

The System

According to the dictionary a system is a 'collection of parts united by some form of regulated interaction'. All computer systems essentially consist of three basic parts or subsystems for different functions:

1. Processing
2. Memory
3. Input/Output

The processing, memory and input/output (I/O) functions of the 64 are connected to, and communicate with, each other by means of common conductors called *busses* while the I/O connects with the outside world (including computer peripherals) by way of various interfaces. We can represent the relationships between the elements in diagrammatic form as shown overleaf:



Processing

A processor is a device capable of performing logical and arithmetic operations on data, including the functions needed for storage and retrieval of data from memory. In the 64 all processing except generation of the TV image and sound production is handled by the 6510 microprocessor (described in detail later in this chapter). This chip decodes and executes machine code instructions stored in ROM or RAM memory and also provides the timing signals which synchronise all other system events. Graphic displays, including sprite generation and manipulation, are produced by the 6567 Video Interface Chip (VIC).

The VIC Chip
Five distinct screen display modes are made available to the programmer by this specialised graphics chip. These are as follows:

1. In **Text** or Character Display mode one of the 256 screen character codes in the current character set is fetched from the screen memory area of RAM and processed to derive an address into the character base area of memory in which the binary pixel patterns for each character are stored. The default character base is the ROM character generator, but 'custom' user-defined characters may be created by redirecting the base address to a RAM location where the re-defined character set information is stored. In addition to the screen code one of the 16 possible character colours is fetched from the corresponding position in the colour table area of RAM. Only one foreground colour is allowed per character and all characters share the same background colour.

2. In **Multicolour Text** mode each character position can contain four

colours – two foreground and two background. This is achieved only at the cost of halving the horizontal resolution since two bits are now required to define each pixel of the character pattern, these two bits giving (binary) counting capability for the four possible colours.

3. The **Extended Colour** mode trades off a reduction in the number of characters available, allowing 64 instead of the full 256, against the ability to select one of four background colours for each character. Bits 7 and 6 of the character code value now contain the background colour information, and the 256 codes are used to represent four sets of the first 64 characters, each set having a different background colour.

4. In **Bit-Map** mode each pixel displayed on the screen corresponds to a bit in RAM memory. Thus a bit-mapped screen with a resolution of 320 by 200 pixels will require 320*200 bits or 8000 bytes of display base memory to support it. In this mode screen memory data is no longer interpreted as character codes but instead is taken as representing colour information. Each byte divides into two nybbles, allowing two of the 16 possible colours to be selected for use within each eight by eight block of pixels.

5. **Multicolour Bit-Map** mode requires two bits to define each pixel in the bit-mapped screen, thus halving the horizontal resolution, but allowing three foreground colours and one background colour to be displayed within each eight by eight block of pixels.

Sprites

There are eight separate sprites available on the 64, each of which may be defined as a 24 by 21 block of pixels and positioned on the screen at a given X and Y co-ordinate. Each sprite requires 24*21 bits (63 bytes) plus one marker byte to define it, with the position of the 64-byte block defining each sprite being indicated by the eight sprite pointers located at the end of the screen RAM. These sprite pointers are checked by the VIC chip at the end of every screen raster line. When the raster count equals the sprite's Y co-ordinate then sprite data is displayed on the screen. Standard sprites may be one of 16 colours except for a display in multicolour mode, when at the usual expense of halving the horizontal resolution up to three colours (four if we include transparent in the count) may be displayed.

Each sprite may be expanded to twice normal size in either the horizontal and vertical directions, or both. The display priority of each sprite can be set with respect to other displayed data (i.e. whether they are displayed in front of or behind characters), but their priority with regard to each other is fixed, with Sprite 0 having

the highest priority and Sprite 7 the lowest.Collisions between sprites and collisions with background data (excepting transparent areas) set bits in the Sprite-Sprite and Sprite-Data collision registers respectively.

Screen blanking
It is possible to disable the screen display entirely. Providing that all sprites are also disabled this will prevent the VIC chip 'stealing' time from the 6510 processor operations and hence speeds up 6510 processing.

Screen Scrolling
The whole screen may be moved 1 to 7 pixels in both horizontal and vertical directions. By using this feature in conjunction with a machine code character scroll routine it is possible to create smooth 'panning' of the display data.

Raster Register
The display screen is written by means of the raster scan as a series of horizontal lines starting at the top of the screen and working downwards. The raster register returns the current raster position in two bytes as a nine-bit number. You will notice that the number is larger than the 200 vertical pixels of the screen. This is because the high and low portions of the raster scan count occur below and above the visible screen boundary, and do not produce a display. The virtue of the ability to know where the raster is on the screen at any time is that it enables us to rid our graphics of 'glitch' – the game programmer's worst enemy. To understand how it works we must know why glitch occurs in the first place.

Glitch results from the movement of a character while it is in the process of being drawn by the raster. For instance, if the top four pixel lines of a character have already been drawn, and its position is then changed the bottom four pixels will be drawn at the new position. Since the top four will have to remain in the old position until the raster has finished drawing that screen and gets back up to the top of the screen again to redraw, what results is a nasty glitch.

The obvious answer is to wait until the raster is below the visible portion of the screen before updating any part of the graphics display. The graphics are then said to be 'raster synchronized', and should hopefully be glitch-free. Problems still occur, however, when either the writing process takes longer than one raster scan or the graphics cannot spare the necessary waiting time required to allow the raster to complete a cycle. The machine code scroll routine given later in this book presents a solution to these problems.

VIC Chip Interrupts

The capability to generate interrupts is one of the VIC chip's most powerful features. An interrupt occurs whenever the Interrupt Request (IRQ) line to the VIC chip is pulled low. The processor then suspends its current operation and jumps to a routine which 'services' the interrupt before returning control to the main processing program, which takes up where it left off. The VIC chip has four separate sources of interrupts, of which three are of particular interest to the programmer:

1. When the raster scan count reaches a pre-set value an interrupt is generated
2. A Sprite collision with the background display
3. A Sprite to Sprite collision

This facility can be used to free the main program from the need to continually check for sprite collisions. Whenever a collision occurs an interrupt is generated and this can be used to pass control to a routine which services the event. The raster interrupt function causes an interrupt to be generated each time the raster scan passes a certain point on the screen. This greatly increases the power of the VIC chip, since it allows us to control sections of the screen, and generate special effects not otherwise achievable. These include mixed graphics modes, where the graphics mode is changed at a given raster position, so that, say, half the screen can be in hi-res mode and half in text mode, and sprite multiplication, allowing more than eight sprites to be on screen at any one time.

An example program for this latter technique is given in Appendix 8. It allows up to 64 sprites to be displayed simultaneously, using a series of raster interrupts which change the position of each of the sprites at several points down the screen. The effect is that the raster scan draws one set of sprites, an interrupt then occurs which passes control to a routine to redefine the position of the sprites into the next section of the screen, and the raster draws another set when the interrupt terminates, and the process is repeated. By the time you get through the rest of this book, you will hopefully have no problem understanding the code.

The VIC chip interrupts are controlled by the Interrupt Control Registers, which consist of the Interrupt Status Register and the Interrupt Latch.

The SID Chip

All the sound functions of the 64 are processed and produced by the 6581 Sound Interface Device or SID chip. The output appears as three voices or channels, each consisting of a combined Tone Oscil-

lator and Waveform Generator, which may be programmed to control the pitch and waveform (Triangular, Sawtooth, Pulse or Noise). Each voice also has an Envelope Generator which may be programmed to produce variable rates of increasing or decreasing volume as the note is being sounded. These are defined as a set of ADSR values – Attack, Decay, Sustain and Release.

Additionally, there are three types of filtering available, allowing the harmonic content of each voice to be controlled. These are the High-pass Filter which passes unchanged all frequencies above a set value, and diminishes all lower frequencies, the Low-pass Filter which performs the opposite function, allowing through only those frequencies below a set value, and a Band-pass Filter which allows only a narrow band of frequencies around the specified value to be heard.

Dynamic effects may be achieved by changing the sound parameters while a tone is being sounded, either directly or else by making use of the fact that the SID chip allows the output of the Voice 3 oscillator to be taken as an input for the parameters of the other two voices. Other SID facilities are the synchronization and ring modulation effects in which the outputs from the oscillators are combined in various ways to produce composite waveforms.

Memory

In order to do useful work a microprocessor requires an associated memory store. The 6510 is what is known as an eight-bit processor, which refers to the organisation of its data bus (eight parallel lines). However, the addressing capacity (the amount of memory to which a processor has access) is limited by the width of its address bus. In common with other eight-bit microprocessors, such as the Z80 and 6809 chips, the 6510 has sixteen address lines, allowing a maximum of 2 ↑ 16 or 65536 (64K) memory locations to be directly addressed. By arranging to switch banks of memory in and out of the 64K address space as required the 64 manages to increase this figure to 88K, consisting of 64K RAM, 20K ROM and 4K of memory-mapped I/O. In the I/O area, the registers of peripheral devices such as the VIC II, SID, and CIA chips are memory-mapped to appear in the 6510 processor's address space between $D000 and $DFFF. That is, the address lines are decoded to point to the device chips instead of RAM memory locations, with the result that this can be addressed as RAM locations by machine code load and store instructions (as well as PEEK and POKE commands in BASIC).

Programming in machine code requires that you become familiar with the 64's somewhat complex memory map. A full exposition of

the memory organisation is not the concern of this book and readers are directed to the *Commodore 64 Programmer's Reference Guide*. However, Appendix 3 contains a guide to memory which highlights those areas which are of particular interest to the machine code programmer.

Input/Output

Communication with the outside world and peripheral devices occurs via a number of devices. Firstly, the 6510 chip itself contains an integral I/O port (at locations 0 and 1) controlling whether RAM, ROM or I/O is currently 'seen' in certain blocks of the 64's memory. Secondly, as we have seen, the VIC II and SID chips output the TV images and sound, via video and audio interfaces, to the TV and loudspeaker.

Thirdly, all other system I/O is handled by the two 6526 Complex Interface Adaptor (CIA) chips whose functions include the handling of input from the keyboard, joystick and light pen; control of the User Port, Serial and RS232 I/O, cassette data transfer, and timing for interrupts.

These CIA chips are complicated and perform a wide variety of functions, most of which fall outside the scope of this book. Each CIA consists of:

Two eight-bit Data Ports, each with their associated Data Direction register by means of which any bit of the port may be configured for either input or output. Commodore have wired up the CIA1 ports to handle inputs from the keyboard and joysticks, leaving CIA2 to handle the serial, user port and RS232 I/O. We will not be concerned with CIA2 operations in this book, except for bits 0 and 1 of Port B, which are set up as the bank select switches for the VIC II chip.

Two 16-bit Interval Timers, the first to control timing of I/O and the second to generate timed interupts. The latter of these is of most interest to us. Of the five possible sources of interrupt generation on the 6526 one is the 'running out' of a timer. If the final act of the subsequent interrupt servicing routine is then to reset and restart the timer an interrupt can be made to occur at regular time intervals. The 64's operating system uses a 1/60th of a second timed interrupt, generated by the CIA1 chip's A timer to scan the keyboard for keypresses without apparently halting BASIC processing, for if the time interval is sufficiently short then the interrupt-driven process and the process that has been interrupted appear simultaneous. We shall be dealing with interrupts in more detail later on in this book.

A 24-hour clock, which counts time in hours, minutes, seconds and 1/10ths of a second, and can be used to generate an 'alarm' interrupt for real-time applications. (Real-time means interacting with reference to events in the 'real' world outside the computer.)

An eight-bit Shift register allows serial I/O, whereby data is transferred bit by bit down a single data line, as opposed to parallel I/O where data is moved a byte at a time, each bit of the byte going down the eight parallel lines at the same time. The register functions in a similar manner to the 6510 shift instruction ASL. For example a data byte for output is first loaded in to the Serial Data register and then transferred to the shift register, from which it is shifted out one bit at a time. The signal rate is determined by the A timer of the CIA, each timer pulse triggering a data shift. When eight shifts have occurred an interrupt is generated to indicate that another byte may now be loaded onto the data register.

Timer and Interrupt Control registers. The timers can function in a number of different modes whose full description is beyond this book. Timer Control registers A and B select the current mode for their corresponding timers.

There are five separate sources of interrupts on the CIA. A single register, the Interrupt Control register, is written to in order to specify which of the possible interrupt conditions are enabled and read after an interrupt has occurred to determine the source of the interrupt.

The 6510 Microprocessor

The 6510 chip may justifiably be regarded as the 'brain' of the 64 since it alone is capable of executing machine code instructions, which are the only instructions that the 64 (or any computer) ultimately 'understands'. Since you can enter BASIC statements into the 64 and have them executed correctly, you may query this statement. Note, however, that I used 'ultimately' in the sentence.

In order to illustrate the essential difference between a high level programming language such as BASIC and machine code let's consider the problem of sharpening a pencil with a pencil sharpener. Imagine trying to explain the operational procedure to someone who had never used a pencil sharpener before. For instance:

1. Insert the pencil into the hole in the sharpener
2. Turn the pencil clockwise in the sharpener
3. Remove the pencil from the sharpener
4. Examine the point

5. Is the point sharp? If not, then go back to step 1.
6. Stop – pencil sharpened

The above steps constitute a simple 'menu' or *algorithm* for the process of sharpening pencils with a pencil sharpener. However, you will notice that we assume our subject understands 'sharpness', knows the meaning of 'clockwise', etc. If this is not the case then these terms will themselves have to be explained in yet simpler language our man can understand. Should we be dealing with an idiot then we can assume very little and even words like 'turn' and 'point' may have to be broken down further. Programming a computer in machine code is somewhat like writing instructions for an idiot! Although we may develop an algorithm in 'high level' terms for our own convenience it must ultimately be coded in the *instruction set* of the 6510 processor.

Machine code instructions are, individually, much simpler and far less powerful than BASIC commands. Typically, a single instruction does little more than alter a single location in memory or a register within the 6510. How then, you may wonder, is it possible to program the CBM 64 in a 'near English' language like BASIC if the 6510 microprocessor can only execute such simple machine code instructions? Well, the answer is that the apparent sophistication of BASIC is not an inbuilt feature of the hardware, but is the result of programming. A large machine code program known as the BASIC interpreter is held in ROM and set in motion when you switch on your 64. It accepts complex BASIC commands from the keyboard and at RUN-time executes lengthy machine code routines to implement those commands. BASIC is the best-known example of a problem-oriented language, such languages being designed to facilitate the specification of algorithms (problem-solving methods) in a particular problem area – in the case of BASIC with regard to algebraic formulae. BASIC commands bear no direct relation to machine code instructions and so can be machine-independent, with the same version of BASIC running on, say, computers using 6510, Z80 and 6809 chips for processing, but with the interpreter code entirely different in each case.

Chip Architecture

The 6510 microprocessor is a direct descendent of the 6502 chip used in the Vic 20, Apple, Atari and BBC microcomputers, to name but a few. What distinguishes the 6510 from its predecessor is the incorporation of a programmable, eight-bit, bi-directional I/O port on the chip. This port appears at memory address $0001 and is used

by the 64 system for memory management, enabling the BASIC and/or Kernal ROMs and/or the system I/O area to be switched out, making extra RAM memory available to the machine code programmer. Note that we can't make use of any of the BASIC or Kernal routines if we have these areas switched out! A microprocessor like the 6510 combines on one chip numerous elements that previously would have been housed on several separate chips. These elements include:

1. The Arithmetic Logic Unit: If the 6510 can be described as the brain of the 64 then the ALU is the heart of the 6510, to continue with the anatomic analogies. Together with an associated register, the A register or Accumulator, it is responsible for performing all arithmetic and logical operations in the 6510.

2. The Control Block: The control block consists of the decoding logic circuits and two important registers, the Instruction register and the only 16-bit (two-byte) register in the 6510, the Program Counter or PC register. Together they combine to perform the Fetch-Execute cycle whereby machine code instructions are fetched from memory and decoded to generate the internal control signals that will perform the specified operation.

3. The System Clock: This is an internal 1 MHz oscillator which provides the timing signals by which all system events are synchronised.

4. The Processor Registers: From the programmer's point of view the 6510 appears as five 8-bit registers which may be manipulated by machine code instructions. These are:

1. The Accumulator or A register
2. The X register
3. The Y register
4. The Processor Status or P register
5. The Stack Pointer or SP register

The next chapter considers the 6510 chip and its operation in greater detail.

2 Operating the 6510

Having reviewed the elements of the 64 system, an essential process since the machine code programmer needs to bear in mind that it is a Central Processing Unit that a program is being written for, and this sits in the middle of a web of connections to other chips, memory and devices, all of which it is necessary to manipulate, we can now look directly at the 6510 in greater detail. The first thing to look at is the way in which the chip gets its instructions, and then look at what these are. The registers need to be described, as these are the only bit of the chip that is visible to the programmer, and the methods by which memory is accessed. We can then look at how we go about writing a machine code program.

The complete description of the instruction set will be left to the next chapter, but the set of mnemonics for the instruction set, together with short descriptions of the operations, needs to be introduced here, and is given below. First let us consider what constitutes a machine code program.

Instruction Decoding

A machine code program consists of a sequence of 6510 instructions, stored in consecutive bytes of memory. Following each one-byte operation code (op code) specifying the operation to be performed may be one or two operand bytes. These contain either the data to be operated on or the address (location in memory) where the data is to be found (fetched from) or stored (placed into). Where the op code specifies the complete operation, it is not followed by any operand bytes.

The 6510 communicates with its memory store and peripheral devices by means of busses. These are not the big red things, but sets of shared signal lines along which address, data and control information is passed.

During one execution cycle, the contents of the Program Counter or PC register are placed on the 16-bit uni-directional (read-only) Address Bus, and the value is read by the memory control devices to fetch an instruction op code from the specified address. This op

code value enters the 6510 by means of the 8-bit bi-directional (read and write) Data Bus. Any operand bytes following the op code are fetched in a similar fashion during the decoding phase. The op code indicates to the 6510 how many (if any) operand bytes are to be fetched, and how it should interpret them. In the course of executing the instruction further fetches or stores may result. In each case the address is first placed on the address bus, and data transfer then occurs over the data bus. The organisation of these operations is achieved by the Control Bus, which comprises the Read/Write line controlling the direction in which transfer is to occur, and the system clock synchronizing all processing events. The time taken for an operation depends on the number of memory manipulations required and the operation itself, and is measured in clock cycles.

After each fetch from memory of an op code or operand the Program Counter is incremented, so that it points to the next memory location. This auto-increment feature keeps the fetch-execute cycle turning. Control is passed to different sections of the program by changing the PC register value so that it points to the new address.

For the 6510 this is all easy stuff, and we don't need to know how it does it. What we do need to know is how we put the instructions into memory for it to execute.

Writing Machine Code

In the early days of single-board microcomputers there was very little software available and everybody who wished to operate the little wonders had to program in machine code the hard way. This meant converting your written out program (op codes, operands and data) into binary numbers and laboriously entering the binary into consecutive memory locations in the computer by means of a bank of eight flipper switches on the front panel. No TV display either, for the dedicated computer freak of yesteryear – results were displayed in binary on a bank of eight lights! A simple program to add 1 to memory location $1000 would appear in binary as below, if we were still stuck in those primitive days:

```
00011000
10101101
00000000
00010000
01101001
00000001
10001101
00000000
```

00010000
01100000

You would have had a row of eight switches, connected to the data bus, and enter the sequence above a byte at a time, using up for a zero and down for a one. As you might expect the process of entering all those 1's and 0's was notoriously error-prone, not to mention extremely boring and time consuming. However, first came paper tape readers for input, and later the cassette interface which automated the process of entering binary. The next step was the development of software to allow programs to be entered using numbers expressed in hexadecimal (base 16) notation from a hex keypad. This is easier both practically and in conceptual terms, since conversion of binary numbers to hexadecimal and vice-versa is quite simple, and fewer digits need to be entered (fewer than we need for decimal notation, let alone binary!). One hex digit represents four binary bits, and the relationship holds good however large the binary number. (Refer to the appendices if you are not sure about the binary, hexadecimal and decimal number systems.) We can express binary 1111 0001 as hexadecimal F 1, and a 16-bit number, which in binary is extremely unwieldy, by four hex digits, so that decimal 45,914, which is 1011001101011010, is B35A in hex. Once you're practiced with the different number systems, all you need for conversion is some mental arithmetic and the table below. While you practice, you'll find conversion tables in the appendix.

HEX	BINARY	DECIMAL
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

This is a considerable improvement on binary and makes our program a lot more manageable:

```
18
AD
00
10
69
01
8D
00
10
60
```

The resulting code is, however, still a collection of fairly tedious and meaningless numbers unless you happen to have all the hexadecimal op codes memorized. (Don't laugh – a lot of old hackers have!) With the advent of more sophisticated input/output devices now taken for granted, such as alpha-numeric keyboards and monitor/TV displays it became possible to write simple *assembler* programs.

An assembler simplifies the process of coding immensely, since it allows the use of three letter mnemonics to represent operation codes, alleviating the need to remember (or look up) a vast number of op codes. When the assembler is run it simply converts (assembles) the mnemonics in sequence into the corresponding binary values for the computer. The advantage of an assembler is that the code is at last meaningful to the eye – providing that you are familiar with the mnemonics and the system used to specify the addressing mode. Depending on the complexity of the assembler, the data can be entered in hex, decimal or binary. Our program becomes, in assembler code:

```
CLC
LDA   $1000
ADC   #$01
STA   $1000
RTS
```

Assemblers vary in sophistication between the simplest line assemblers, where the mnemonic code is assembled into memory a line at a time, and complex multi-pass programs which allow the programmer to define variable values and branch destination addresses as labels. One great advantage of the latter facility is that the destination addresses of any branches do not have to be worked out by the programmer.

MOS Technology, the makers of the 6502 microprocessor (identical in this respect to the 6510) have published a standard table of the 6502 mnemonics for the op codes and addressing modes. There are other standards in use, but all the assembler programs given in this book use the standard notation for mnemonics and addressing modes.

It is possible to write quite large programs using simple development software, and indeed some very well-known programmers claim to work exclusively with line assemblers. This may however be part of the 'whiz-kid' hype that tends to pervade the games programming world. For all machine code programming a proper assembler is extremely useful, and for large programming jobs it is a vital tool. Readers are advised to buy a good editor/assembler package at the earliest opportunity. There are a number of good programs on the market, but my particular preference would be for Brad Templeton's PAL 64 assembler with the POWER editor and the Commodore 64 assembler development package from CBM. Whichever you choose there are certain features to look out for. In the assembler some facility should exist for 'chaining' a sequence of assembler code source files from disk or tape storage so that large programs can be assembled. If this facility is missing then the programmer is limited to a single source file and although 38K of formatted source (assembler) code may sound like a lot, when assembled into memory it only represents about 2.5K of 'object' machine code. The editor should ideally also provide some 'toolkit' functions such as DELETE and RENUMBER facilities. If it doesn't then you may be able to use those featured later in this book. As a final note, you should try to avoid line editors – screen editing is much quicker and seems more natural after using the 64's excellent BASIC editor.

If your budget cannot stretch to such things at the moment then fear not, for Chapter 4 is devoted to the first big program of the book – a machine-code monitor incorporating a line assembler/disassembler, a machine code SAVE routine and an assortment of other useful functions. The assembler can only be used to write small machine code programs, but even if you possess a full labelling assembler you will probably find 'Wedge-MON' invaluable for saving and loading machine code programs to disk or tape, as well as for debugging programs and general fiddling about. It will also help in exploring the mysteries of the CBM ROMS (of which more later).

Next we need to look at the supposedly easy-to-remember three letter mnemonics for the op codes. These are given here with short definitions of their function.

6510 Instruction Mnemonics

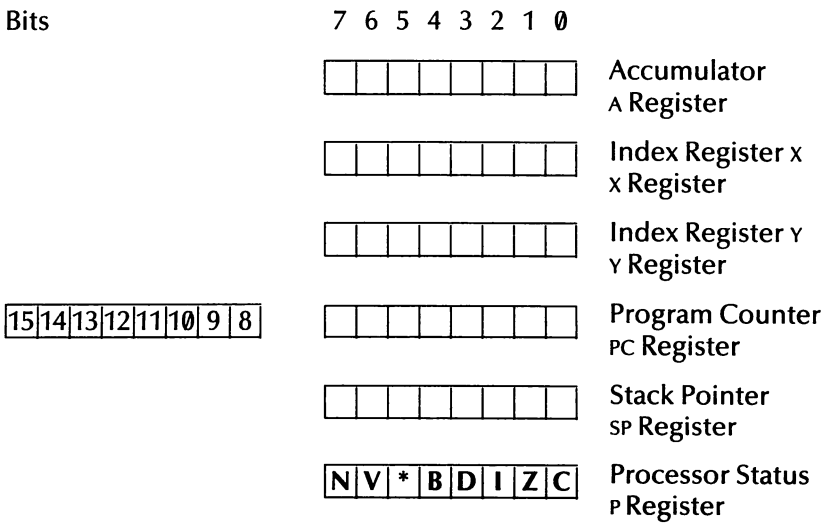
ADC	Add memory to Accumulator with Carry
AND	Logical AND memory with Accumulator
ASL	Arithmetic Shift Left one bit
BCC	Branch if Carry flag Clear
BCS	Branch if Carry flag Set
BEQ	Branch if result equal to zero (Z flag set)
BIT	Compare memory bits with Accumulator
BMI	Branch if result minus (N flag set)
BNE	Branch if result not equal to zero (Z flag clear)
BPL	Branch if result plus (N flag clear)
BRK	Break (Jump to Interrupt routine)
BVC	Branch if V (overflow flag) clear
BVS	Branch if V (overflow flag) set
CLC	Clear Carry flag
CLD	Clear Decimal mode flag
CLI	Clear Interrupt flag (enable Interrupts)
CLV	Clear V (overflow) flag
CMP	Compare memory with Accumulator
CPX	Compare memory with X register
CPY	Compare memory with Y register
DEC	Decrement memory location value by 1
DEX	Decrement X register by 1
DEY	Decrement Y register by 1
EOR	Exclusive OR Accumulator with memory value
INC	Increment memory value by 1
INX	Increment X register by 1
INY	Increment Y register by 1
JMP	Jump to new address
JSR	Jump to subroutine
LDA	Load Accumulator with memory value
LDX	Load X register with memory
LDY	Load Y register with memory
LSR	Logical Shift Right one bit
NOP	No Operation
ORA	Logical inclusive OR Accumulator with memory
PHA	Push Accumulator contents onto stack
PHP	Push P register contents onto stack
PLA	Pull value from stack into Accumulator
PLP	Pull value from stack into P register
ROL	Rotate Left one bit
ROR	Rotate Right one bit
RTI	Return from interrupt routine

RTS	Return from Subroutine
SBC	Subract memory from Accumulator with carry
SEC	Set Carry flag
SED	Set Decimal mode flag
SEI	Set Interrupt disable flag
STA	Store Accumulator in memory
STX	Store X register in memory
STY	Store Y register in memory
TAX	Transfer Accumulator to X register
TAY	Transfer Accumulator to Y register
TSX	Transfer Stack register to X register
TXA	Transfer X register to Accumulator
TXS	Transfer X register to Stack register
TYA	Transfer Y register to Accumulator

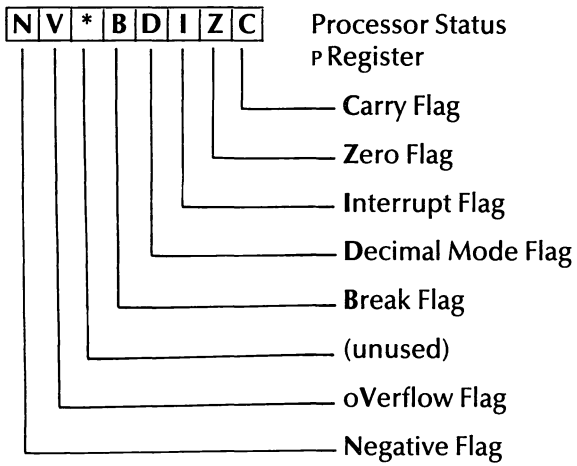
You will notice that there are mostly operations concerned with registers in the above list. We need to take a look at these, and then the ways in which memory is addressed.

The 6510 Registers

The arrangement of the 6510 from the point of view of the programmer is often represented by a diagram like this:



The flags of the processor status P register are represented as follows:



The 6510 registers A (Accumulator), X and Y may be thought of as memory locations internal to the chip. Using machine code the contents of a RAM or ROM memory location may be copied into these registers and modified. Subsequently, the modified contents or result may be sorted back in RAM. For example take the code sequence:

LDA $1000	load the Accumulator from memory location $1000
AND #$0F	perform logical AND of the accumulator and the number $0F
STA $10	store the Accumulator contents at memory location $10

The Accumulator is the most important of these registers, firstly because it is the only register with associated arithmetic and logical instructions (ADC,SBC,AND,ORA,EOR,BIT) and, secondly, because it is capable of load and store operations using indirect addressing unlike the other two.

The main role of the X and Y registers is in *indexing*. This is a powerful technique whereby the base operand address in an instruction is modified by addition of the contents of one or other of the index registers to yield a new effective address. This means that one can access large amounts of memory in a short fast loop simply by incrementing an index register.

Unlike the Accumulator, the index registers can be incremented and decremented in a single instruction, the result of which may be

tested via the N (negative) and Z (zero) flags. This brings us to the fourth of the programming registers, the P, Processor status or Flag register. This can be thought of as eight separate *flags* or *status-bits*:

Bit:	7	6	5	4	3	2	1	0
Flag:	N	V	–	B	D	I	Z	C

The N (Negative), V (oVerflow), Z (Zero) and C (Carry) flags are affected as a result of certain instructions and serve to inform later sections of the program that a certain condition identified with that flag has occurred. Each bit may be *set*, i.e. a binary 1 occupies that position, or *clear*, i.e. a 0, also known as *reset*.

The N flag is set (to 1) whenever bit 7 of a result holds a 1 (set), i.e. when the result is equal to or greater than $80 (128 decimal), and therefore Negative in 2's complement representation. It is affected by all load, store and transfer instructions and also by arithmetic logical and shift operations.

The V flag is affected by arithmetic instructions and indicates an overflow from bit 6 to bit 7. It is also used by the BIT instruction.

The Z flag is *set* when the result of an operation is equal to zero. It is affected by most loads and transfers and all arithmetic and logical instructions but not by stores or branches.

The C flag is used in arithmetic operations, being set when a carry is generated from an ADC operation and cleared when a borrow results from an SBC. It also acts as 'bit 8' in arithmetic shift operations.

The N,V,Z and C flag states can be tested using the conditional branch instructions BMI, BVS, BEQ and BCS, and their converse forms, BPL, BVC, BNE and BCC.

The remaining three flags of the status register are the B, D, and I flags. The B flag is set whenever a break interrupt is generated by the processor encountering a BRK instruction. When the D flag is set it indicates the processor is in *binary coded decimal* (BCD) mode. This can be set by the programmer with an SED instruction. The I flag is the Interrupt Disable bit and may be set by the instruction SEI. It has the effect of preventing IRQ interrupts.

The fifth and last eight-bit register in the 6510 is the SP register or Stack Pointer. The *stack* is an area of RAM memory 256 bytes (a *page* of memory) in length and located in Page One of memory, occupying the locations from $0100 to $01FF, and is used both by the processor itself, and by the programmer if required, for temporary storage. Whenever the 6510 encounters a JSR (jump to subroutine) instruction or an interrupt is generated the processor stores the next op code address, minus 1, on the stack, so that when it comes to an RTS (return from subroutine) or an RTI (return from interrupt) the pro-

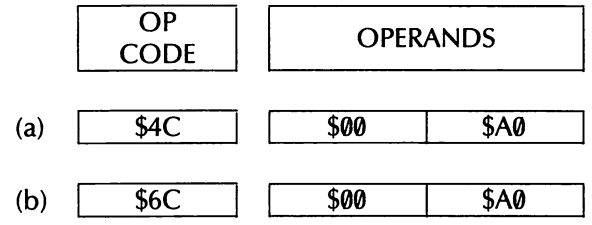
gram can return to where it left off. The stack operates with what is known as a LIFO structure (Last In, First Out). If fills up from the end and works forward so that when subroutines are nested – providing the stack does not overflow – the first address stored will be the last address retrieved. The Stack Pointer points to the next free position in the stack. On power-up the Stack Pointer is set to $FF and for every item 'pushed' on to the stack it is decremented by one. The programmer has access to the stack via the PHA, PLA, PHP and PLP instructions.

There is one further register in the 6510, the only 16-bit register, called the Program Counter or PC register. It contains the address of the instruction currently being executed by the processor. On powering up the 6510 the program counter is initialised with the address $FFFC, which is the entry point into the 64's operating system software. The machine code programmer gains control by redirecting the processor to his code via the BASIC SYS command or by causing a processor interrupt having first changed the corresponding vectors to point to his own code instead of the system code. This will be covered in later chapters, when we cover the practicalities of programming. Now let's look at how the 6510 accesses memory.

Instruction Addressing Modes

We have seen that the 64 decodes an instruction stored in memory by fetching the op code value from a memory location. The 56 op code mnemonics correspond to the possible *types* of operation that the 6510 can execute. However, there are more than 56 op codes. This is because the op code not only specifies the type of operation, but also specifies the *addressing mode* for the instruction, which enables the 6510 to decide how the operand bytes are to be interpreted, and how many of these bytes there are following the op code. The 6510 has the capacity to use a variety of different methods to compute the address (memory location) with which the instruction is concerned, i.e. the memory address from which data is to be taken, into which data is to be placed, or in which data is to be altered, according to the operation to be performed. Instructions which specify a complete operation, where no memory location is involved, such as NOP (No OPeration), RTS (Return from Subroutine) or TAX (Transfer Accumulator to X register) are said to use the Implied address mode. These only have one op code. Other instructions, where a memory location is involved, will need to specify the way in which the memory location is to be worked out, and this means that each operation, as specified by the mnemonic, will have several op codes, one for each of the different address modes that can be used

with the instruction. This idea is probably best understood by looking at an example.



(a) and (b) are two sets of three consecutive memory locations. Each byte stores a binary value, given here in hexadecimal, and each group of bytes represents a machine code instruction, consisting of an op code and two operands. Both of the op codes specify the JMP instruction but with different addressing modes. In example (a) $4C specifies the JMP instruction with the *absolute* addressing mode and the processor then knows that it is to regard the two bytes following the op code as the destination address (low byte/high byte) for the instruction. In assembler code, this is specified by entering:

(a) JMP $A000

Note that we enter the hex number in high byte/low byte format. The assembler automatically swaps the bytes into the low/high byte sequence the 6510 requires.

In example (b), however, the addressing mode is *indirect*, and the processor understand from the op code that the operand bytes represent *not* the literal value of the address but the address at which the address for the jump is stored. This is entered in assembler as:

(b) JMP ($A000)

The assembler takes note of the brackets, and places the correct op code in memory to JMP to the memory location pointed to by the *contents* of $A000 and $A001 (low byte first, i.e. in $A000).

There are in fact eleven different addressing modes used in 6510 machine code. No instruction employs every addressing mode, but some use as many as eight while others are restricted to one.

IMMEDIATE

In the immediate addressing mode a single byte follows the op code. This is taken to be a literal value, i.e. a number 0–255. In

assembler, this is signified by placing a hash sign, #, before the number. Thus:

```
LDA  #$10
```

reads as 'load the accumulator with $10'.

ABSOLUTE

The two bytes which follow the op code represent an address which is a source for reading a value from or a destination address into which to place a value, depending on the operation to be performed. For instance, the instruction:

```
LDA $1000
```

instructs the 6510 to load the accumulator with the contents of location $1000.

ZERO PAGE

This is a special form of absolute addressing where the address is in Page Zero of memory (locations 0 to 255 decimal, 00 to FF hex) and therefore only needs one byte. Zero-page addressing not only takes up less space for the instruction, but is also quicker to execute than absolute addressing, since it requires the 6510 to perform less processing. An example would be:

```
LDA $B2
```

meaning 'load the accumulator with the contents of location $B2'.

ABSOLUTE INDEXED

This mode uses the index registers (the X and Y registers) to offset the absolute address given. The actual memory location accessed is the specified address *plus* the current value of the index register used. The assembler statement:

```
LDA $1000,X
```

will load the accumulator with the value stored at address ($1000+X), where X is the X register value. Note that the machine code version still occupies only three bytes.

ZERO PAGE INDEXED

The value of the index register (X or Y) concerned is added to a Zero-page address to derive the address from which the data is taken. If the X register has a current value of $1B (27 decimal), then the instruction:

LDA $20,X

will load the accumulator with the value stored at location $20+$1B, i.e. $3B (59 decimal).

INDIRECT

This mode (often called Pure Indirect addressing) is *only* available to the JMP instruction. The two bytes following the op code represent a pointer to the address, as we have seen above. The assembler syntax is:

JMP ($1000)

INDIRECT (INDEXED)

With the exception of the JMP instruction, as we have seen, all indirect addressing on the 6510 is effected via Page Zero locations. There are two forms of indirect addressing which are totally different in effect.

In *pre-indexed* addressing the byte following the op code is added to the X register to give the address of the low byte of a Page Zero pointer. This low byte, and the high byte which follows, specify the address which is to be accessed.

LDA ($10,X)

If the X register held the value 5, then the accumulator would be loaded with the value held in the address specified by the contents of $15 and $16. If $15 held $1B, say, and $16 held $0A, then the accumulator would be loaded with the contents of $0A1B.

In *post-indexed* addressing the byte following the op code is the low byte of a Page Zero two-byte pointer, the contents of which are evaluated, and then added to the Y register value to give the address to be accessed. Assume $10 holds $E1, and $11 holds $01. The instruction:

LDA ($10),Y

Will, if Y=5, load the accumulator from the address held in locations $10, $11, plus 5, i.e. $01E1 plus 5, or $01E6.

It is important to appreciate the distinction between these two types of addressing.

IMPLIED

This refers to single byte instructions which act solely on either internal registers within the processor or the stack, and which therefore do not require any following operands. The instructions thus occupy a single byte. Examples are:

CLC, which clears (sets to 0) the Carry flag of the Status register.

PHA, which places the A register value on the stack.

RELATIVE

The relative addressing mode is used by all the 6510 conditional branch instructions, and by no other instructions. The branch instructions test various flags of the P (Processor Status) register, and program control branches a specified number of bytes backwards or forwards from the current Program Counter (PC) register position. The operand is a single byte, treated as a signed 8-bit value, which allows a branch forward (positive) of 127 bytes, or backward (negative) of up to 128 bytes. See the section on the branch instructions for an explanation of how the operand byte is evaluated to give the offset or displacement value for the branch. Since the process of counting the number of bytes required for a jump to the desired location, and then working out the byte value is tedious and inefficient, monitors and assemblers allow the branch instructions to be specified in terms of the address to be jumped to (make sure it's within the available offset range!), either directly as a hex value, or as a label:

```
BEQ   $B2A1
BCC   DESTN
```

The program then converts this to give a value for the offset and inserts the correct single byte value into memory following the op code. Disassemblers do the same thing in reverse, decoding the operand byte value and inserting the actual address to which the branch will occur into the display.

If you look at Appendix 1 you will find a table which gives the 6510 op code hex values for each instruction in each of the addressing modes that may be used with that particular instruction. Appendix 2

contains a table which gives the op codes, ASCII character codes and CBM BASIC control codes and tokens for the numbers 0–255.

Before we go on to look at the instruction set in detail, and how the different instructions are combined into programs, the way in which the assembler listings are formatted needs to be described.

Assembler Conventions

The particular assembler which was used to create the programs to be found in this book is Brad Templeton's PAL assembler. This assembler conforms closely to the MOS Technology conventions in the way the op code mnemonics and addressing modes are represented. A look at these conventions will help you to understand the assembler listings as given in this book, and allow you to convert them if you are using an assembler which requires different conventions to be used.

Numbers

Numbers are assumed to be decimal unless the digits are prefixed by either the $ *or* % symbols, indicating that the number is expressed in hexadecimal or binary representation, respectively. Thus the following examples all have the same result, but use different number systems.

LDA #10 means load accumulator with the value 10 *decimal*

LDA #$0A means load accumulator with 0A *hexadecimal*

LDA #%00001010 means load accumulator with 00001010 *binary*

This facility allows us to use the most convenient form for a number: Beginners may prefer decimal notation, binary values make masks or shift/rotate operations clearer, and so on. Mostly, hex is the preferred form (less typing!).

Addressing modes

The different addressing modes are indicated as shown below. EXPR refers to the operand expression which defines the number or address, expressed in one of the three possible number systems given above.

Format	Addressing Mode	Example
#EXPR	Immediate	LDA #$0A
EXPR	Absolute	LDA $1000
EXPR	Relative (Branch only)	BEQ $1000
A	Accumulator	ASL A

EXPR,X	Absolute, indexed by X	LDA $1000,X
EXPR,Y	Absolute, indexed by Y	LDA $1000,Y
(EXPR,X)	Preindexed indirect	LDA ($0A,X)
(EXPR),Y	Postindexed indirect	LDA ($0A),Y
(EXPR)	Indirect	JMP ($1000)

Labels

PAL allows character labels to be eight characters long. A label is essentially a variable name assigned a number value. Since some other assemblers allow only six characters in a label all the labels used in this book are a maximum of six characters long. For example the assembler instruction:

```
ADDRS=$1000
```

creates the label ADDRS and sets it to 1000 hex. We can then use the label name instead of the value, as a destination for jumps, etc. A sophisticated feature of PAL is its expression evaluator whereby expressions may be included in the operand field. Such expressions are evaluated during assembly to produce an effective operand. PAL offers many arithmetic and Boolean operators but for simplicity I have just used '+' and '−'.

```
ADDRS=$1000
LDA ADDRS        (equivalent to LDA $1000)
LDY ADDRS+1      (equivalent to LDY $1001)
```

Program Counter

The * label refers to the current value of the assembler's 'program counter'. At the beginning of each assembler program this is set equal to the start address of the program. The instruction:

```
*=$C000
```

sets the start point of the code assembly to C000 hex. The program counter may also be reset during assembly, and this feature is used to reserve a block of memory for data storage. For example, to reserve 20 bytes of memory for a data table, we can insert the following instruction within the assembler listing:

```
*=*+20
```

Modifiers

Since many 6510 instructions require single byte numbers as operands most assemblers allow the upper or lower byte of a two byte expression to be made available via the expression modifiers >and<:

```
ADDRS=$1000
LDA #<ADDRS     (equivalent to LDA #$00, the low byte)
LDY #>ADDRS     (equivalent to LDY #$10, the high byte)
```

Pseudo-operators
These are instructions to the assembler program which do not of themselves generate code. The flexibility of an assembler can be judged by the number of pseudo-operators offered, and like most quality assemblers PAL provides a wide range. However, for the sake of portability between assemblers only a few of the more common of these have been used in the assembly language listings given in this book, as follows.

.BYTE: This is used to place byte values for data into the code sequence. The values are inserted at the current program counter position, and in succeeding bytes of memory. The values are literal values, and are separated by commas:

```
.BYTE $2E,<ADDRS,$FF
```

This stores a sequence of three bytes, the second being the low byte of ADDRS.

.WORD: This places two-byte values into the code in the Low/High byte format:

```
.WORD ADDRS
```

This has the same effect as would:

```
.BYTE <ADDRS,>ADDRS
```

.ASC: This pseudo-op enters a string of the ASCII codes of the characters into the code sequence:

```
.ASC "HELLO MUM"
```

This enters the ASCII code for H, E, L, etc, into the sequence of bytes starting at the current assembler program counter value.

.END: .END is optional in PAL and merely serves to indicate the end of a program file.

Comments
Comments are included in PAL text after a semi-colon ';' character,

and are ignored by the assembler – they are there for our convenience only, as with REM statements in BASIC.

```
LDA ADDRS  ;A COMMENT
```

Note that the major assembler listings in this book do *not* make use of the facility to follow an instruction with a comment. This is a constraint of the book format, and instead the code is divided into short functional sequences and the comments corresponding to these sequences placed in a group above the actual instruction sequences.

3 The 6510 machine code instructions

You have seen the 6510 instruction set mnemonics, with the minimal definitions of their functions that were given earlier. Some of the operations are obvious, some not. The op codes themselves, i.e. the binary numbers representing each instruction in the various possible addressing modes, are given in Appendix 1. The function of this chapter is to provide further details of the instructions and their actions. The instruction set is divided into nine functional groups and each individual instruction defined, together with its effect, if any, on the Processor Status (P) register, after an outline presentation of the instructions within the group. Again, the examples given involve other instructions than the one under discussion, and illustrate their interactions, so more than one pass through this section will be necessary before you can proceed, referring back as necessary to the definitions given here.

Arithmetic Operations

ADC	SBC	
CMP	CPX	CPY
INC	DEC	
INX	DEX	
INY	DEY	

The contents of memory locations may be added to or subtracted from the Accumulator. There are no arithmetic instructions for the other registers. Multi-byte additions or subtractions may be performed, with any carries or borrows generated being brought forward to the next stage of the operation.

The simplest addition is that of a single-precision addition where two 8-bit numbers are added together to produce an 8-bit result in the accumulator:

```
CLC
LDA ADDRS
ADC #$01
```

It is important to remember that the ADC instruction is an AD+Carry operation, so that for a single byte addition it is necessary to CLear the Carry flag with CLC before using the ADC instruction, avoiding an incorrect result caused by a set Carry flag from a previous operation. The importance of this add+carry feature becomes apparent when we look at the adding together of two 16-bit numbers to produce a 16-bit result. Adding 16-bit numbers involves two stages of 8-bit addition, since we cannot directly manipulate 16-bit numbers. First the *low* bytes of the two numbers are added and then the two *high* bytes. Any carry generated in the first stage is then automatically added in to the result of the second stage.

In the case of a two-byte addition suppose the value $0510 is to be added to the contents of the addresses labelled HIBYT and LOBYT, and the result then stored back in HIBYT/LOBYT:

```
CLC         clear the Carry flag
LDA LOBYT   load Accumulator with contents of LOBYT
ADC #$10    add $10
STA LOBYT   store result back in LOBYT
LDA HIBYT   load Accumulator with contents of HIBYT
ADC #$05    add $05 plus Carry, if any, from the first addition
STA HIBYT   store result back in HIBYT
```

An indefinate number of such stages may be set up, with the carries generated from each ADC being passed on to the next, so that we can arrange to add any size of number. However, it is extremely unlikely that you will require more than the two-stage or double-precision addition operation shown above.

Subtraction is an analogous operation to addition, except that any carry required should instead be thought of as an 'inverse-borrow'. So the Carry flag must be *set* with an SEC instruction, rather than cleared, before we use an SBC instruction. Below is an example of the subtraction of two bytes to produce a single byte result in the accumulator. The accumulator is loaded with the value held in the location specified by the assembler label ADDRS, and $01 (1 decimal) is then subtracted:

```
SEC
LDA ADDRS
SBC #$01
```

If the result of the SBC operation 'underflows', i.e. if the result is less than zero, then a 'borrow' (Carry flag clear) is generated. In the same way as for multi-byte addition, multiple precision subtraction may be achieved by combining several stages of subtraction, passing any 'borrows' generated from stage to stage:

```
SEC
LDA ADDRS1
SBC ADDRS2
STA RSLT
LDA ADDRS1+1
SBC ADDRS2+1
STA RSLT+1
```

In the operation of the CMP, CPX and CPY instructions the contents of the Accumulator (A), X and Y registers respectively are compared to the contents of a memory location. These instructions are included in the 'arithmetic' group because they are essentially subtractions. They work by subtracting the contents of the memory location from the register concerned, although the contents of that register remain unchanged. The result sets flags in register P (Status register) as for a subtraction and these may be tested for by the conditional branch instructions BEQ, BCS and BCC.

For example, suppose we wished to compare the accumulator contents with the value $05, branching to the address specified by the label HIGHR if the accumulator value was greater than $05, branching to LOWER if it was less than $05 and to EQUAL if $05 was the accumulator value:

```
CMP #$05     condition Z and C flags on result of comparison
BEQ EQUAL    if Z flag set then branch to EQUAL
BCC LOWER    if C flag clear branch to LOWER
BCS HIGHR    if C flag set and Z flag clear branch to HIGHR
```

The INC and DEC instructions allow the value held in specified memory locations to be incremented (increased by 1) or decremented (decreased by 1) in a single step and the result examined by means of the Status register flags. Similarly INX, DEX, INY and DEY allow the X and Y register contents to be incremented and decremented. Note that when a register or memory location value is incremented past $FF (255 decimal) or decremented less than $00 the value 'wraps around'. Thus $FF increments to $00 and $00 decrements to $FF.

To increment the 64's screen colour, for example, we need to increment the VIC II register at location $D021:

```
INC $D021
```

To add 1 to the two-byte (low/high) pointer PNTR,PNTR+1, the INC instruction is implemented. If the value in PNTR was already $FF, it will become $00, which can be tested for by BNE:

```
INC  PNTR    increment low byte
BNE  FIN     if low byte not zero then go to FINish
INC  PNTR+1  (overflow has occurred) so increment high byte
FIN  NOP
```

The increment and decrement instructions are obviously useful in counting operations, such as loops. A loop where we want to use the value of what would be termed the 'loop variable' in BASIC can be set up quite simply:

```
       LDY  #$01    load Y with 1
AGAIN  TYA          transfer Y to Accumulator
       PHA          put it on stack
       ...
       ...
       ...          loop processing, using any
       ...          registers, and Y value (still there)
       ...
       PLA          get Y loop value from stack
       TAY          put it back in Y
       INY          add 1
       CPY  #$0A    compare with 10
       BNE  AGAIN   back if not equal to 10
```

This loop would execute nine times.

Certain addressing modes allow the specified address to be indexed by the values stored in the X or Y registers. The X or Y value is added to the address to define the memory location involved in the operation. This enables incrementing or decrementing loops to be constructed which can access large areas of memory. Take for example the following routine which clears the 64 screen:

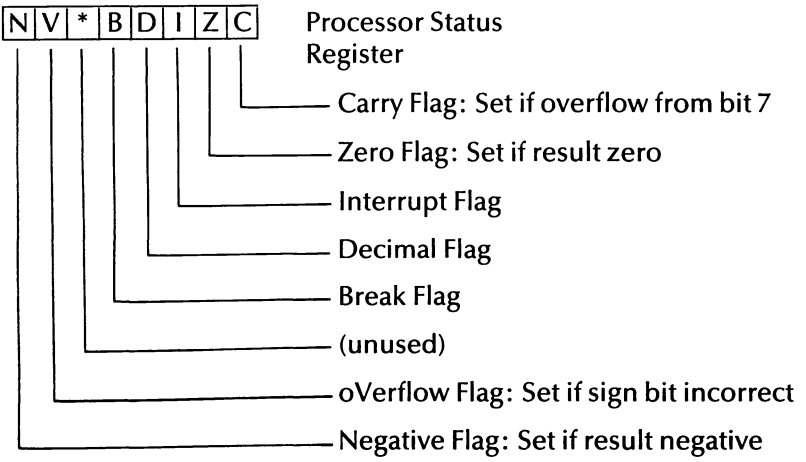
```
       LDX  #0      load the X register with zero
       TXA          transfer the X register value to the
                    Accumulator
LOOP   STA  $0400,X store zero at $0400+X register
       STA  $0500,X store zero at $0500+X register
       STA  $0600,X store zero at $0600+X register
       STA  $0700,X store zero at $0700+X register
       INX          increment X register
       BNE  LOOP    if X register not equal to zero then back to
                    LOOP
```

ADC: ADd with Carry to Accumulator

This instruction adds a number to the contents of the A register (Accumulator), plus 1 if the Carry flag (bit 0 of the Processor Status or P register) is set. The number may be specified directly, in immediate mode, or be the value stored at the memory location specified. The Carry flag will be set if the previous operation produced a result greater than 255 decimal ($FF). We can see the process if we look at an addition in binary:

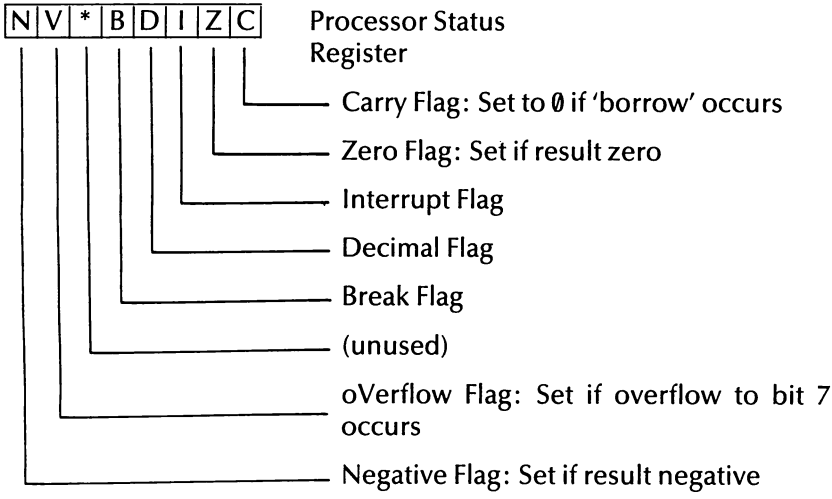
```
  01010101        101
+ 11011000      +216
----------      ----
 111111101      317 decimal
```

The Carry flag acts as an extra bit (bit 8), and enables the result of an operation which produces a number greater than 255 to be incorporated in further processing. In the case of ADC, this is extremely useful in adding multi-byte numbers. Since the process of adding the Carry bit is automatic, we need a way of clearing it before using ADC, which is done with the CLC instruction. The ADC operation will set (i.e. to 1) all the flags in the Status register that are affected by arithmetic results – Negative, oVerflow, Zero and Carry. The result of an ADC operation can thus be tested for all the conditions that can cause a conditional branch. The Zero flag is set if the result in the Accumulator is 0, the V flag if the result of a 2's complement addition lies outside the range −128 to +127, and N if the result is negative in signed 8-bit format (bit 7 set).



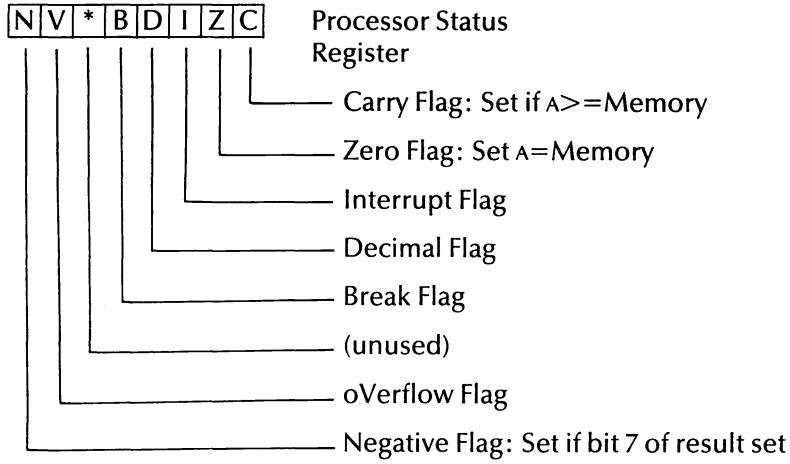
SBC: SuBtract from Accumulator with Carry

This is the opposite of the ADC instruction. It subtracts the data specified (direct numeric, or memory location contents) from the Accumulator value, but the Carry in this case must be thought of as a 'borrow'. The result of an SBC operation (held in the Accumulator) is the Accumulator value as was minus the specified value, minus the *inverse* of the Carry flag (0 if Carry=1, 1 if Carry=0). This is for use in multi-byte subtraction (borrow 1 from high byte if result would be less than zero), but for simple subtraction the Carry flag must be set before the operation is performed.



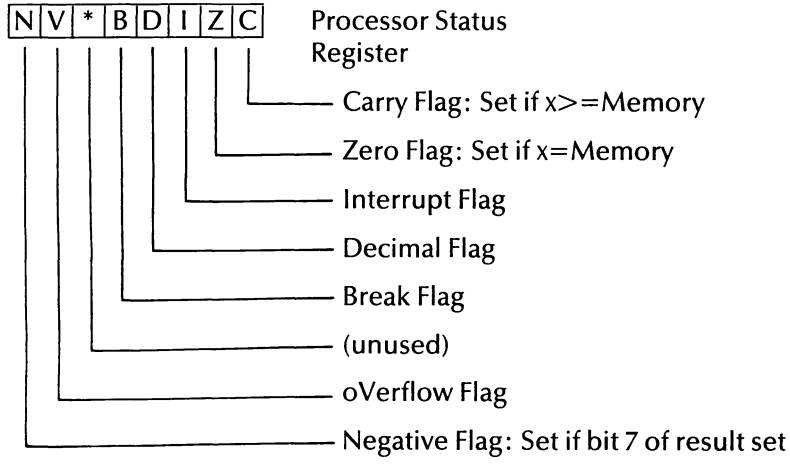
CMP: CoMPare memory with Accumulator

This instruction compares the value stored in the Accumulator with that stored in the specified memory location by subtracting the memory value from the Accumulator value. The result is not stored, but the appropriate flags in the P register are set according to the result. The Accumulator and memory contents are not affected by the operation. With immediate addressing, the value compared is the literal value following the op code.



CPX: ComPare memory with **X** register

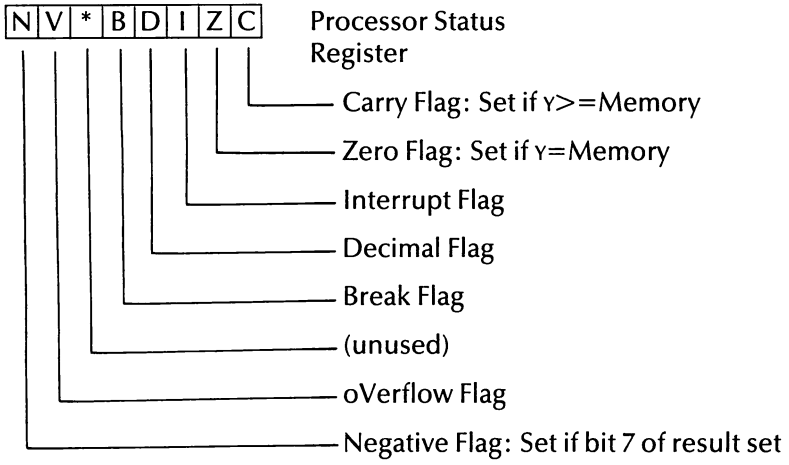
This instruction compares the value stored in the X register with the specified literal value, or that stored in the specified memory location, by subtracting the memory value from the X register value. The appropriate flags in the Processor Status (P) register are set according to the result of this subtraction, but the result is not stored. The contents of X and the memory location are not affected by the operation.



CPY: ComPare memory with **Y** register

The CPY instruction compares the value stored in the Y register with that specified directly or stored in the specified memory location by

subtracting the contents of memory from the Y register value. The appropriate flags in the Processor Status (P) register are set according to the result of this subtraction, but the result is not stored. The contents of the Y register and the memory location are not affected by the operation.

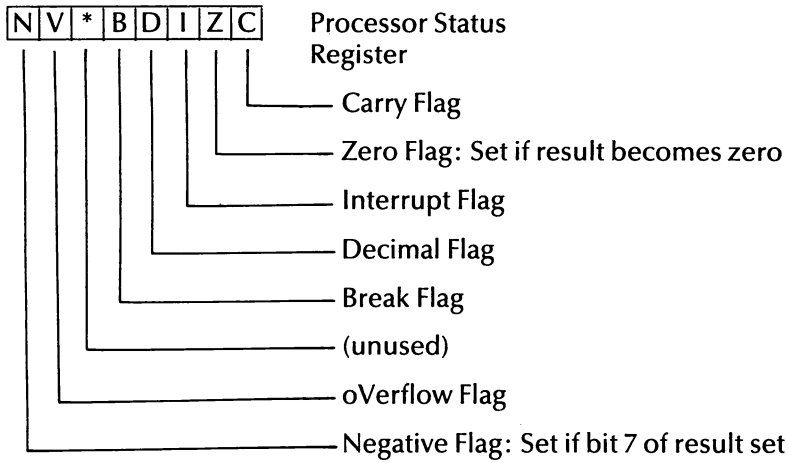


DEC: DECrement memory value

DEX: DEcrement **X** register

DEY: DEcrement **Y** register

These three instructions perform the operation of subtracting one from the value stored in the specified memory location or register. Note that the DEX and DEY instructions use implied addressing, and are single byte instructions needing no operand bytes. The effect of the operations on the P register are the same, dependent on the result of the operation in the byte concerned, as given below.

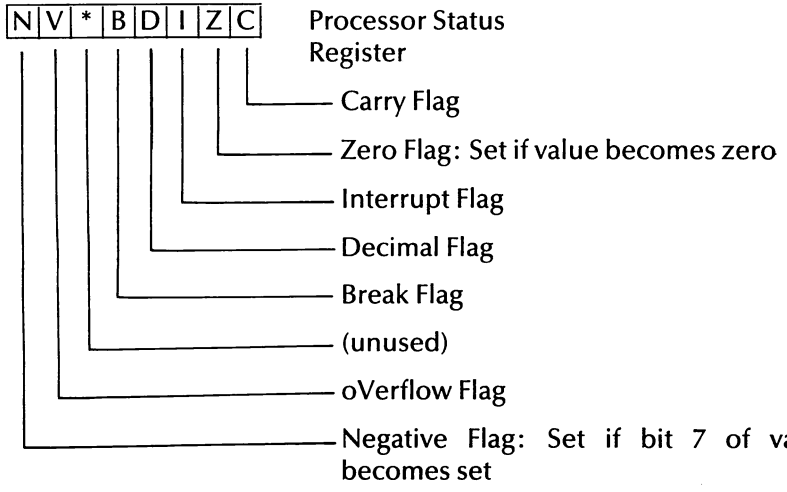


INC: INCrement memory value

INX: INcrement **X** register

INY: INcrement **Y** register

These instructions perform the operation of adding one to the value stored in the memory location or register specified. Note that the INX and INY instructions use implied addressing, and are single byte instructions needing no operand bytes. INC can use several addressing modes (see Appendix 1). The flags affected within the P register are the same, dependent on the result of the operation in the byte concerned.



Logical Operations

AND ORA EOR

BIT

The Logical or Boolean instructions allow 'bitwise' manipulation of the Accumulator contents. Specific bits may be switched on and off respectively by ORing and ANDing the Accumulator value with the appropriate *mask* or bit pattern. This is best demonstrated by representing the masks in binary format. If we wish to set (to 1) bit 0 of the Accumulator without affecting other bits, we use:

```
ORA #%00000001
```

and to reset bit 0 to zero:

```
AND #%11111110
```

and to 'flip' bit 0, i.e. if set (1) then reset (to 0) and if clear (0) then set it (to 1):

```
EOR #%00000001
```

The BIT instruction effectively ANDs the Accumulator value with the value held in a memory location, but the contents of the Accumulator are not disturbed, as they would be if the AND instruction was used. The result is notified by the Z, V and N flags, with Z being set if the result of the and operation is zero, and V and N being made equal to bits 6 and 7, respectively, of the memory location.

For example, to test bit 0 of the memory location labelled as LOC branching to location MATCH if set:

```
LDA #%00000001
BIT  LOC
BNE MATCH
```

To test bits 6 and 7 no mask is required, as we can test the N and V flags directly with the conditional branch instructions. E.g. to test bit 7 of LOC, branching to MATCH if set:

```
BIT  LOC
BMI MATCH
```

AND: Logically **AND** Accumulator with byte.

This instruction ANDs the accumulator value with the value of the operand, an immediate (literal) value, or that of a memory location. This is the same operation as the AND operator in BASIC performs. Each bit of the accumulator byte is compared with the corresponding bit of the operand. If both bits are set (i.e. equal to 1) then the resulting byte will have 1 in the same position. If either or both the bits was a zero, the result will be a zero. Assume the accumulator holds the value $1B (27 decimal), and we code the instruction:

```
AND #$3E
```

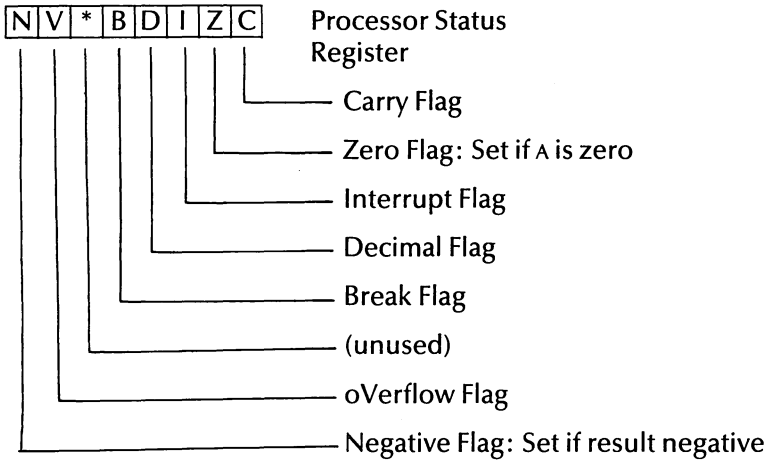
This tells the 6510 to take the accumulator value and AND it with the literal value $3E (62 decimal). Since we're using hexadecimal, it is easy to see the binary forms of the numbers:

```
$     1      B
    0001   1011
$     3      E
    0011   1110
```

The AND operation then produces a result from the bit patterns:

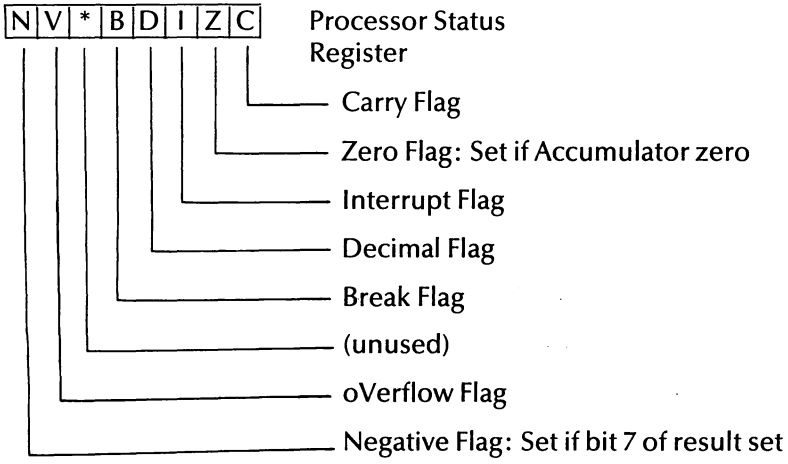
```
      00011011        27 decimal
AND   00111110   AND  62 decimal
      --------       ---
      00011010        26 decimal
```

The result is stored in the Accumulator. The use of this is primarily in 'masking' bytes to check specific bits in a byte, as mentioned above. Masking a byte by ANDing with $02, for instance, which is 00000010 in binary, will give a result of either $02, if bit 1 of the accumulator was set, or 0 if it was not.



ORA: Logical **OR** Accumulator with byte

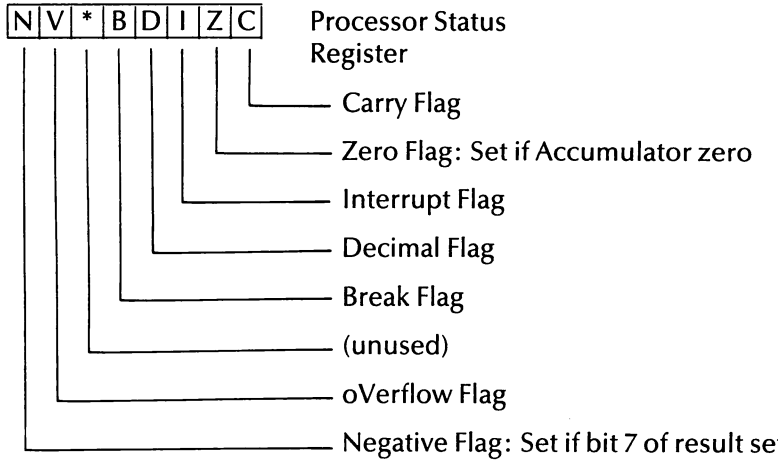
The Accumulator contents are ORed with the literal (immediate) value following the op code, or that held in the address in memory specified. Each bit in the two bytes is compared, and if either or both bits are set, the corresponding bit in the result will be set. If both bits are 0, the resulting bit will be a zero. The result is stored in the Accumulator.



EOR: Exclusive **OR** Accumulator with byte

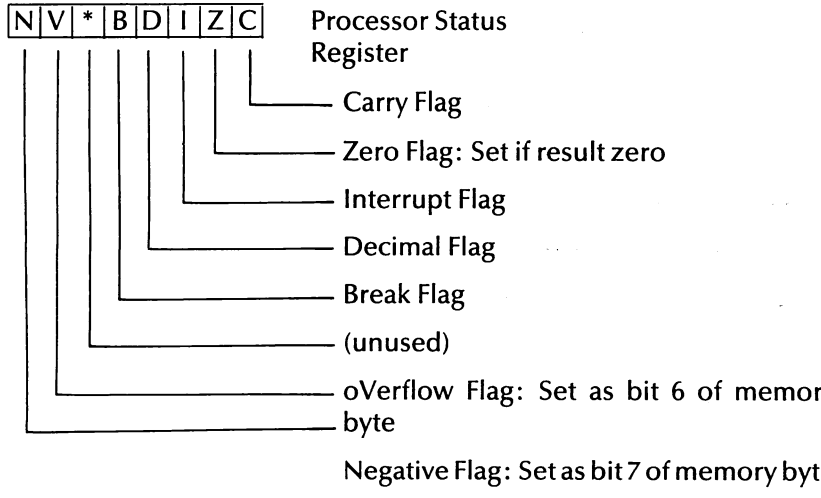
The EOR operation checks the bits in the Accumulator against the immediate number or that at the specified memory address in the same way as AND and ORA, but in this case a bit in the result is set only

if the Accumulator and memory bits were different. If the bits were the same, either 0 or 1, the result bit is a zero. The effect is to 'flip' any set bit in the Accumulator which matches a set bit in the value with which it is EORed.



BIT: Compare Accumulator **BIT**s with memory.

This is a logical version of the compare operations, in that the Accumulator contents are not disturbed, but the relevant flags are set in the P register according to the result obtained. The memory contents at the address specified are ANDed with the Accumulator. Bits 6 and 7 of the P register (the V and N flags) are set to the same value as the corresponding bits in memory.



Load and Store Instructions

LDA	STA
LDX	STX
LDY	STY
TAX	TXA
TAY	TYA
TSX	TXS

The LDA, LDX, LDY, STA, STX and STY instructions cause the register concerned to be LOADed with a value from memory or have the current register value STOred back to memory. Transfer of data bytes between registers is possible using the TAX, TXA, TAY, TYA, TSX and TXS instructions.

To load the Accumulator with the value stored in memory location ADDR1, transfer the Accumulator value to the X register, and store the X register contents at location ADDR2, we use:

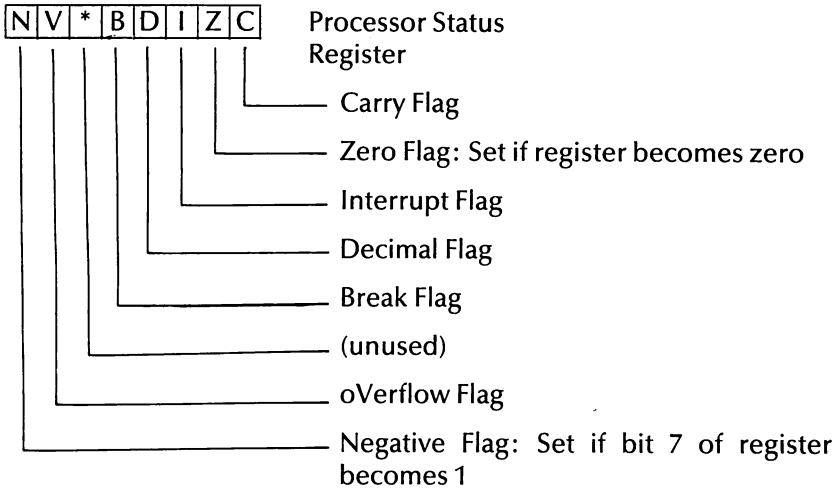
```
LDA ADDR1
TAX
STX ADDR2
```

LDA: LoaD Accumulator with memory value

LDX: LoaD X register with memory value

LDY: LoaD Y register with memory value

The three operations perform the same function for each of the A, X and Y registers. The value of the memory location specified, or the immediate mode value given, is placed in the register. The P register flags are affected as shown below by the new value placed in the register.



STA:STore Accumulator contents in memory

STX:STore **X** register contents in memory

STY:Store **Y** register contents in memory

These instructions simply place the current value of the register into the specified location in memory. No flags are affected in the P register.

TAX:Transfer Accumulator contents to **X** register

TAY:Transfer Accumulator contents to **Y** register

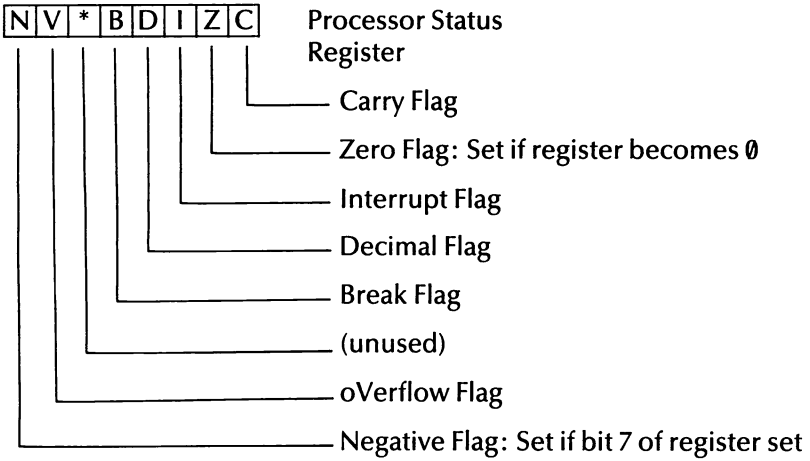
TXA:Transfer **X** register contents to Accumulator

TYA:Transfer **Y** register contents to Accumulator

TSX:Transfer **S** register contents to **X** register

TXS:Transfer **X** register contents to **S** register

These instructions all transfer a value from the first register given in the mnemonic into the second. The contents of the first register are preserved. The P register flags are set according to the new value placed in the register to which the value has been transferred. Note that there are no instructions for direct X to Y or Y to X transfers, and that the Stack Pointer value can only be transferred to or from X.



Shift and Rotate Instructions

ASL LSR

ROL ROR

The shift and rotate instructions both have the effect of moving all the bits in the accumulator or memory location specified by the operand one position to the left or right so that one bit enters the byte while at the other end another bit exits. The difference between the two operations is that while in both cases the bit exiting the byte goes into the Carry flag bit, in a shift the bit entering the byte is always a zero while in a rotate instruction it comes *from* the Carry flag bit. Thus performing eight successive shifts on the Accumulator or memory location will result in a zero result, while nine rotates will leave the contents unchanged.

There are no direct multiplication or division instructions in 6510 machine code, so we perform these operations by means of successive shifts. A simple instruction:

```
ASL BYTE
```

will multiply the value stored in location BYTE by two. Similarly, the sequence:

```
LSR BYTE
LSR BYTE
```

will divide BYTE by four. To multiply a two-byte number by means of shift instructions we must ROL the *high* byte so that the bit 'shifted out' of the low byte is rotated into the high byte. For instance, if we wish to multiply the value held in BYTE/BYTE+1 by four, we need to program a code sequence like this:

```
ASL  BYTE
ROL  BYTE+1
ASL  BYTE
ROL  BYTE+1
```

A similar sequence is necessary for division, except that in this case we must first LSR the high byte and then ROR the low byte. So, to divide BYTE/BYTE+1 by eight:

```
LSR  BYTE+1
ROR  BYTE
LSR  BYTE+1
ROR  BYTE
```

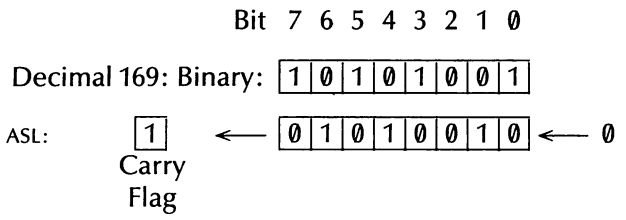
The obvious problem with the above operations is that they only allow multiplication and division by powers of two, i.e. *2, *4, *8, etc. Other multipliers can however be achieved quite simply, by combining shift operations with adds and subtracts. For instance, we can multiply a value, represented by BYTE, by three using first an ASL instruction, and then an ADC:

```
LDA  BYTE
ASL  A
CLC
ADC  BYTE
STA  BYTE
```

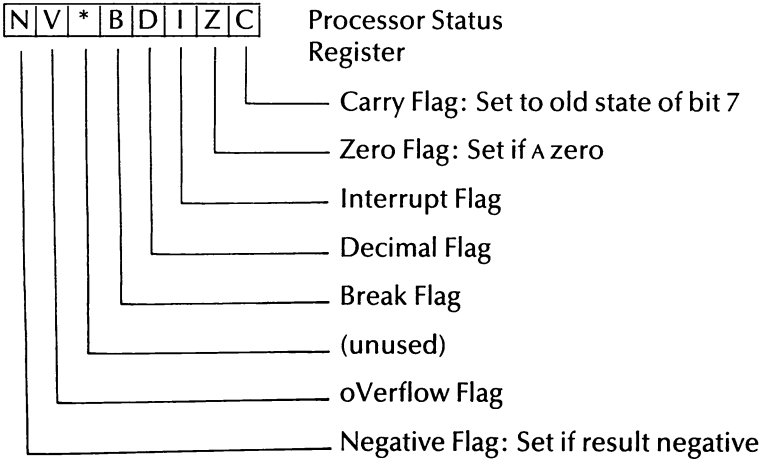
Algebraically, this might be expressed as 3B=2B+B.

ASL: **A**rithmetic **S**hift **L**eft

This instruction can apply either to a memory location value or to the current Accumulator value. Its effect is to shift all the bits in a byte one place to the left, so that bit 0 is moved to the bit 1 position, and so on. Bit 0 has a zero inserted, and bit 7, which drops off the end, is placed in the Carry flag of the status (P) register. The effect of the instruction is to multiply the byte value by two:

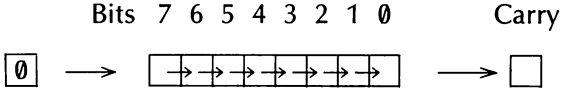


The resulting (9-bit) value, taking the Carry flag into account, is 256+82, giving 338, or 2*169.

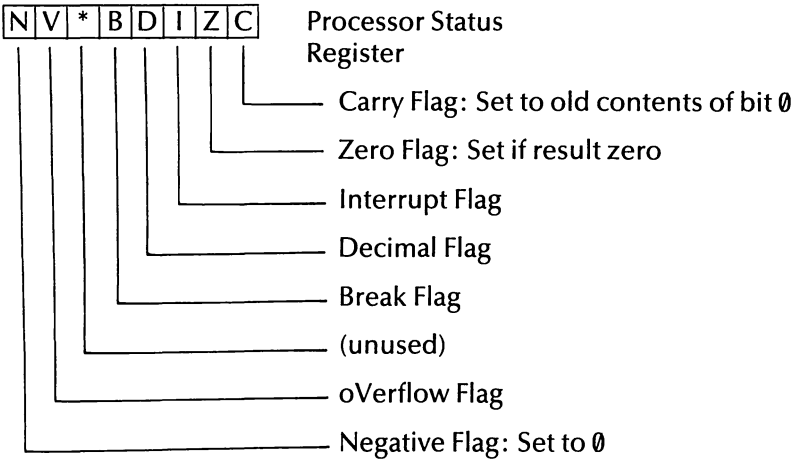


LSR: Logical **S**hift **R**ight

This instruction acts on the contents of the specified memory address or the accumulator, according to the addressing mode. All bits are shifted to the right one place, bit 7 becoming bit 6 and so on. Bit 7 becomes a zero, and bit 0 is shifted into the Carry flag:

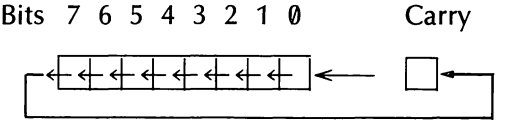


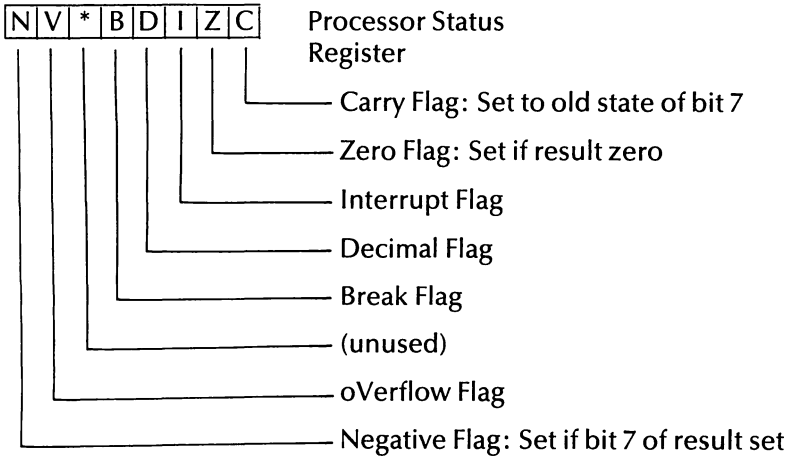
The instruction is effectively a division by two, the Carry flag being available to check if there was a non-integer result to the division.



ROL: ROtate Left

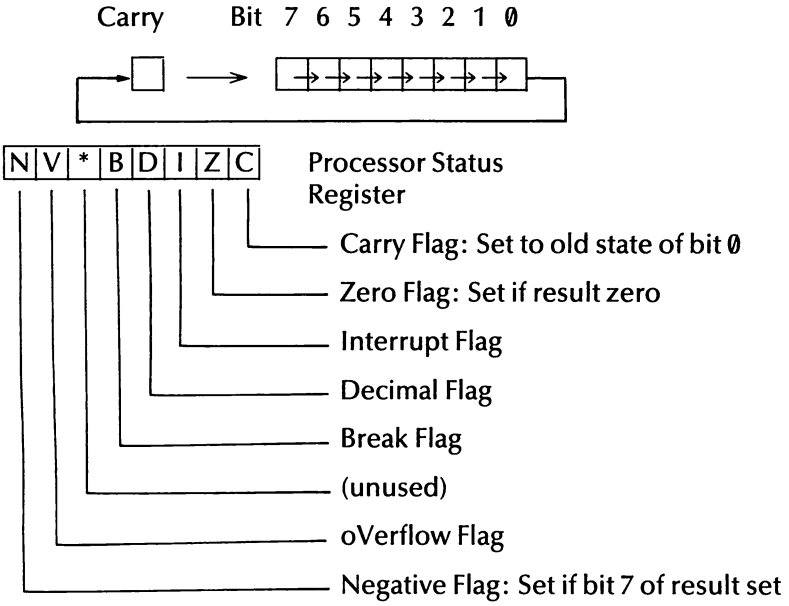
The Accumulator or specified memory location has all the bits shifted a place to the left. The current Carry flag value is placed in bit 0, and bit 7 then placed in the Carry flag. The effect is a loop shifting through nine bits:





ROR: ROtate Right

The anti-clockwise equivalent of ROL, all bits being shifted one place to the right:



Conditional Branch Instructions

BCC	BCS
BEQ	BNE
BMI	BPL
BVC	BVS

The eight conditional branch instructions allow program control to be transferred a number of bytes forward or back from the current location depending on the state of four flags in the status register P. Note that assemblers allow the destination to be signified by a *label*, or a direct address, so that a conditional branch to LOOP (an address label) if Carry set can be coded:

```
SEC
BCS  LOOP
```

and the correct offset for arriving at LOOP will be worked out automatically.

It is important to note that all the branch instructions use the relative addressing mode. The operand is a single byte, treated as an 8-bit signed integer. Negative numbers are represented by setting bit 7 to 1, to indicate that it is a negative number. The possible range of the values of the operand is −128 to +127, with 10000000 binary ($80) representing −1. The branch will be made this number of bytes backwards (negative) or forward (positive) from the current PC register value. Since it has first accepted and evaluated the branch instruction and the operand byte following it, you should remember that the branch, relative to the address of the Branch instruction, will be within the range +129 (+127+2) and −126 (−128+2). These are the maximum loop and jump sizes we can construct using the branch instructions. The use of labels, or direct address destinations, with assembler/monitor programs means no calculation is required, but it is still vital to ensure the destination is within range.

The branch instructions are extremely important, as they are the means of coding conditional jumps into a program, fulfilling the same role as the IF ... THEN GO ... construct in BASIC, that of changing program flow as a result of tests (the compare instructions). Since the size of a branch is limited, it is common to branch to a section of code which will then use the JSR or JMP instructions to transfer control to a subroutine or code section if extensive processing is required.

The example below may help to make the mode of use of branch instructions clear; the ADC instruction is one of the few that affects all four flags:

```
CLC             clear the C flag
LDA   #$40      load the Accumulator with $40
ADC   #$40      add $40 to the Accumulator
BPL   TEST      branch if N flag clear
BCS   TEST      branch if C flag set
BEQ   TEST      branch if Z flag set
BVC   TEST      branch if V flag clear
```

All the above tests fail. The result of adding $40 to $40 is $80. The bit pattern of $80 is 10000000 so bit 7 and therefore the 'sign bit' N is set. The addition does not generate a carry so C is clear. The result was not zero so the Z flag is clear and an overflow from bit 6 into bit 7 occurred, so the V flag is set. No branch (to the address signified by the TEST label) is taken, and processing will continue to the next operation.

The branch instructions do not affect any of the P register flags.

BCC: Branch if Carry Clear

This is a conditional branch instruction. Its single byte operand defines the number of bytes backward or forward which the program is to jump if the Carry flag of the Status (P) register is clear, i.e. 0. If the Carry flag is set (1), then the instruction will have no effect. If the branch condition is met, the Program Counter (PC) register is modified by adding or subtracting the operand value from the current address, and the next instruction taken from the resulting address. The BCC instruction is useful in arithmetic operations, since it checks whether a carry has been generated, and in conjunction with the comparison instructions CMP, CPX and CPY, which use the Carry flag to signifiy the result of the comparison. The branch will be taken if the register value was greater than the value with which it was compared.

BCS: Branch if Carry Set

This instruction causes the program to branch backwards or forwards the number of bytes indicated by the operand byte if the Carry flag is set, the opposite condition to BCC. The relative jump specified will be taken if the result of a CMP, CPY or CPX instruction shows that the register value is less than or equal to the data byte compared.

BEQ: Branch if EQual to zero

The branch jump will be taken if the Z (zero) flag of the P register is set when the instruction is executed. Setting the Z flag to 1 indicates that the result of an operation is zero. Note that comparisons set the Zero flag to indicate that the register and byte values are the same, i.e. the *difference* is zero. This is often used to check whether a register which is being decremented (see DEC, DEX, DEY) has reached zero, or if the result of a comparison is zero.

BMI: Branch on MInus

The branch will be taken if the N flag of the P register is set to 1, i.e. if the result of the last calculation or comparison produced a negative number as a result, or bit 7 was set. Note that 0 is still counted as a positive value, so a zero result does *not* cause a branch.

BNE: Branch if Not Equal to zero

The branch is taken if the result of a comparison is different from zero, signified by the Zero flag being reset to 0. This instruction is the opposite test to that performed by BEQ.

BPL: **B**ranch if **PL**us

This is the counterpart of the BMI instruction, and causes the branch to be taken if the N (negative) flag is set to 0, indicating a positive (*not* negative) result to the last comparison (register value greater than data value) or calculation. Seldom used for branching after a comparison, other instructions providing more precise conditional tests, but used in loops which decrement a memory location or register, when it will branch as long as the value is 0 or greater.

BVC: **B**ranch if o**V**erflow flag **C**lear

The branch is taken with this instruction if the V (oVerflow) flag is Clear, i.e. 0, indicating that a carry has occurred from bit 6 to bit 7. For a standard binary number this merely means that a number greater than $7F (127 decimal) has been generated, but in the case of signed binary values, where bit 7 is the sign bit, this occurrence changes the sign of the number from positive (bit 7=0) to negative (bit 7=1), a matter of some concern! It also has uses in BCD (binary coded decimal) arithmetic, and can be used with BIT to check the state of bit 6. CLV clears the V flag, but there is no instruction to set it directly available in the 6510 instruction set.

BVS: **B**ranch if o**V**erflow **S**et

The opposite to BVC above. The branch is taken if the V flag is set, and hence if an overflow from bit 6 to bit 7 has occurred. It is also used in conjunction with BIT, since this instruction affects the state of the V flag.

Unconditional Branch Instructions

JMP JSR

RTS RTI

The unconditional branch instructions transfer control directly and unconditionally to a stated memory location anywhere in the address space. In the case of RTS and RTI the destination for the jump is obtained from the stack.

E.g. an unconditional branch to the 64's initialization routine resets the operating system:

```
JMP $FCE2
```

JMP: **JuMP** to new address

This is the equivalent of the GOTO in BASIC, and the new address is placed in the Program Counter register, so that the next instruction will be taken from that address. The usefulness of JMP, which would otherwise simply tend to encourage 'spaghetti programming' with jumps all over the place whenever the programmer ran out of space for a routine, is that it is the only instruction that can use indirect addressing. In this addressing mode the specified location is that at which the final destination address is stored. We specify an address, and the destination is placed in this address and the following byte. This gives us the power to change the destination to which a JMP will be made. The stored address, or 'vector' can be changed, and since the 64 uses RAM vectors for the operating system, this gives us an entry point into the system.

JMP alters no P register flags.

JSR: Jump to **SubR**outine

This is the equivalent of the GOSUB instruction in BASIC. The address of the end of the JSR instruction is placed on the stack, and the jump is made to the subroutine. Execution continues until an RTS is encountered, when the stored address, plus 1, is taken off the stack and placed in the PC register, so that the instruction following the original JSR is the next one executed. Subroutines are vital in machine code programming, as they allow both multiple use of repetitive sections of code, and allow the use of a control program which calls sections of code, the placement of which is not crucial, as appropriate.

No P register flags are affected.

RTS: **ReT**urn from **S**ubroutine

As explained under JSR above, this indicates the end of a subroutine, and causes the return address to be taken off the stack. Note that since instructions exist to manipulate the stack, the return address may be changed, or an RTS without a corresponding JSR may be used to cause a jump to the address (plus 1) stored on the stack.

No P register flags are affected.

RTI: **ReT**urn from **I**nterrupt routine

This is used with Interrupt routines only, which we will meet with later. The effect is of a PLP instruction followed by an RTS, i.e. the top byte on the stack is taken off and placed in the P register, and the next two bytes of the stack are placed in the PC register as a return

address. Replacing the P register exactly as it was before the Interrupt occurred ensures that processing may continue as if nothing had happened.

Operations on Flags

```
CLC        SEC
CLV
```

These instructions allow the programmer to set or clear certain flags in the P or Processor Status register, e.g. to clear the Carry prior to an add operation:

```
CLC
LDA ADDRS
ADC #$01
STA ADDRS
```

CLC: **CL**ear the Carry flag

This places a zero in bit 0 of the P register, and has no other effect. It is important when performing ADC operations, and with the ROL and ROR instructions.

SEC: **SE**t Carry flag

The opposite of CLC, this places a 1 in bit 0 of the P register (the Carry flag). It is essential for use with SBC instruction, and in multiplication and division using the shift and rotate instructions. Since the Carry flag is the only flag that can be set or reset as required, it is also often used to 'flag' a particular condition, such as an error, since a branch can then be made as appropriate.

CLV: **C**lear o**V**erflow flag

Inserts a zero into bit 6 of the P register, and does nothing else. Not often used except in complex arithmetic. Note that there is not a corresponding instruction to set the overflow flag.

Stack Operations

```
PHA PLA
PHP PLP
```

The stack instructions permit the programmer temporary storage of the Accumulator and Processor status registers on to the stack. For

example, to store registers prior to entering a subroutine FUNCT, and retrieve them on returning, we PusH the P and A values directly, then transfer the X and Y values to A before pushing the values on to the stack in turn. Retrieval must be in *reverse* order of storage:

```
PHP             push P register on to stack
PHA             push A value on to stack
TXA             transfer X value to A
PHA             push A contents to stack
TYA             transfer Y value to A
PHA             push A contents to stack

JSR  FUNCT      jump to subroutine

PLA             pull Y value off stack into A
TAY             transfer Y value to Y register
PLA             pull X value off stack into A
TAX             transfer X value to X register
PLA             pull A off stack
PLP             pull status register P
```

PHA: **P**us**H** **A**ccumulator on to stack

PHP: **P**us**H** **P**rocessor Status register on to stack

These two instructions place the contents of the Accumulator and P register, respectively, on the stack at the current position of the Stack Pointer. S is then decremented to point to the next free location on the stack. No P register flags are affected.

PLA: **P**ul**L** byte into **A**ccumulator from stack

PLP: **P**ul**L** byte into **P**rocessor Status register from stack

These instructions take the last byte pushed on to the stack and place it in the register concerned. The Stack Pointer S is then incremented to point to the previous value placed on the stack. No P register flags are affected by PLA, but the entire P register is replaced with the bit pattern of the byte pulled from the stack in the case of PLP.

Instructions to the Processor

NOP	BRK
CLI	SET
CLD	SED

The NOP instruction stands for NO oPeration, and does just that. The processor fetches and decodes the instruction but there is no execution. NOPs are useful for leaving 'space' in a program so that changes may be made to the code without resorting to re-assembly.

The BRK instruction forces a BRK interrupt and causes the processor to suspend the current program and jump to memory location $FFFE, where the 64 operating system takes control and returns the system to BASIC.

The remaining four instructions affect the I and D bits of the status register. SEI stops IRQ interrupts from occurring while CLI re-enables them. (More about interrupts later).

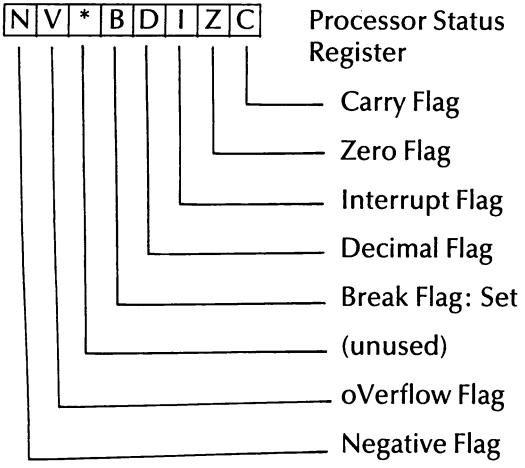
SED puts the processor in Binary Coded Decimal or BCD mode. In this mode each byte stores two *decimal* digits, one in each nybble (four bits), i.e. a number from 00 to 99 can be held in a byte. In all subsequent adds and subracts after a SED carries (and 'internal carries' from bit 3 (low nybble) to bit 4) are generated when a result is greater than 9 decimal, rather than $0F as in binary mode. CLD re-enables the binary mode.

NOP: No **OP**eration

Does nothing. Affects no flags. Used to fill memory, patch out code.

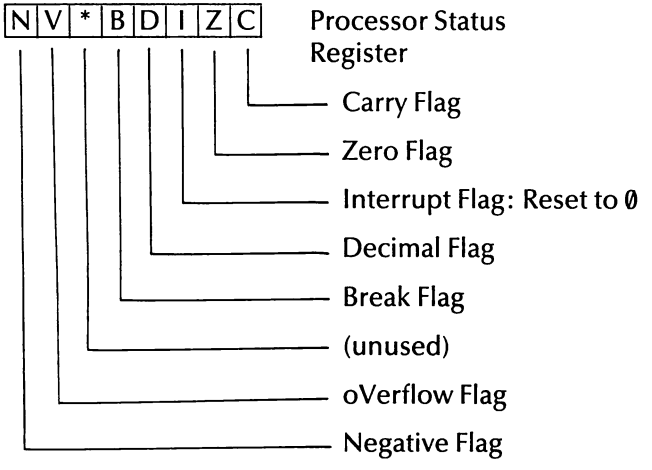
BRK: **BR**ea**K**

Effectively a STOP instruction, BRK forces an interrupt and stops any further 6510 processing. However, the interrupt routine may be altered by the user. The Break flag is set to indicate that the interrupt was forced, the Interrupt flag is set, and the PC register pushed on to the stack, followed by the P register. The jump to the interrupt handling routines is via an indirect JMP to the vector at locations $FFFE/$FFFF.



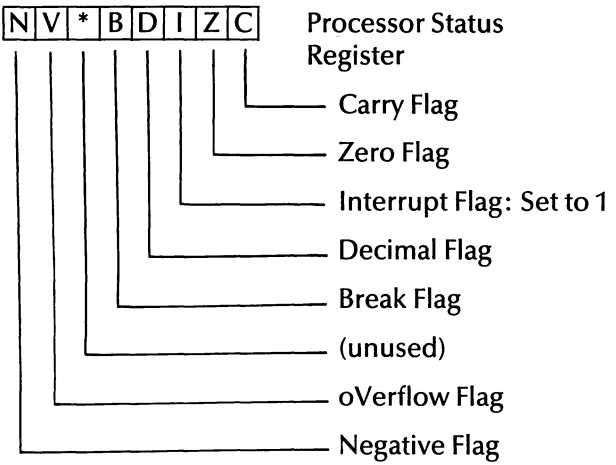
CLI: **CL**ear **I**nterrupt flag

Places a zero in bit 2 of the P register. This enables IRQ (**I**nterrupt **ReQ**uest) interrupts (i.e. not BRK interrupts).



SEI: **SE**t **I**nterrupt flag

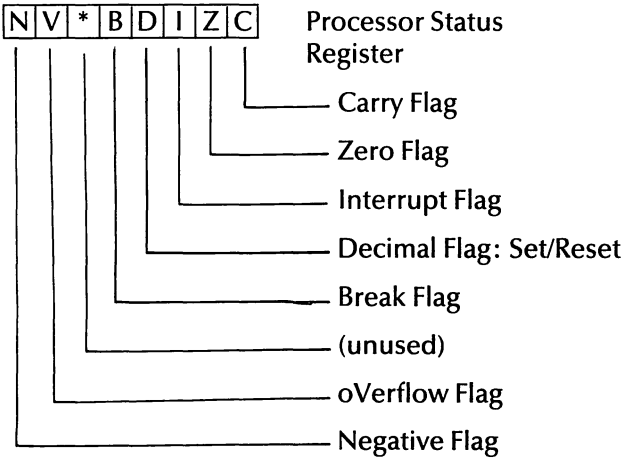
Places a 1 in bit 2 of the P register, preventing IRQ interrupts from occurring. Turning off interrupts speeds up processing, and also prevents the screen or keyboard from working!



CLD: **CL**ear **D**ecimal mode flag

SED: **SE**t **D**ecimal mode flag

These instructions switch in (SED) and out (CLD) the 6510's Binary Coded Decimal mode by respectively setting and clearing the Decimal flag of the P register. Whilst in BCD mode each nybble of a byte is treated as a decimal digit 0–9, and the high nybble is incremented whenever the low nybble exceeds 9. This is used for simple additive arithmetic, when display of the values is required.



We have looked at the architecture of the 6510 and we have seen what the various 6510 instructions can do. Next we need to see how these instructions may be put together to produce useful programs.

This process will be elucidated by way of two examples, one very simple and the other a little more difficult.

Screen Fill

This is a straightforward routine which simply fills the entire text screen with a character of your choice. You should remember that the text screen memory of the 64 is a block of 1000 bytes of RAM starting at address $0400, each byte of this block defining by its contents – in text mode – which character will be displayed. It is therefore necessary to set each byte of the screen RAM to the value of the code for the chosen character.

One method of doing this would be to load the accumulator with the character code (e.g. #$24) and then store this value in each of the bytes which correspond to the screen locations. This would be coded as:

```
LDA #$24      load the
              accumulator with
              the character
              code value
STA $0400     store the
              accumulator
              value in the
              first screen
              byte address
STA $0401     store the
              accumulator
              value at
              screen+1
STA $0402     store the
              accumulator
              value at
              screen+2
STA $0403     (etc..)
```

This method, although fast, is incredibly wasteful of program space (and programming time!), requiring a total of 3002 bytes of code to fill the whole screen. A much better solution is obviously to use a loop construction. Look at the following assembler listing:

```
        * = $C000       code is
                        assembled
                        starting at
                        $C000 (49152)
10      LDX #$00        load X register
                        with zero
15 LP   LDA #$24        load A
                        (accumulator)
                        with the
                        character code
20      STA $0400,X     store .A at the
                        screen address
                        + .X
65      DEX             decrement .X
70      BNE LP          if .X>0 then
                        relative branch
                        back to LP
75      RTS             return to BASIC
```

With the loop method, instead of having 256 STA instructions each executed once only there is a single STA instruction which is executed 256 times. Note that the first address to be filled is specified as $0400 plus the X register value. This is first of all set to zero (LDX#$00), and on each subsequent pass through the loop the X register value is decremented. This gives $0400+#$FF, $0400+#$FE, etc, until X is zero. When this happens the or Zero flag is set and the conditional branch in line 70 fails.

You will notice that the above loop will not fill the whole of the screen RAM. This is because the maximum amount of memory that can be accessed by one STA in such a loop is 256 bytes (the X register can only store values 0 to 255 decimal). To fill the whole screen we must add three more STA instructions with base operand values 256 bytes apart, so we need to add the following three lines to the above loop:

```
25      STA $0500,X
30      STA $0600,X
35      STA $0700,X
```

Finally, in order to see the characters we have placed on the screen we must also fill the colour memory section of RAM with a colour different from that of the background. The colour RAM block matches the screen RAM block byte for byte and starts at location $D800. So we need to add the following five lines of code within the loop, and our program will be complete:

```
40      LDA #$00
45      STA $D800,X
50      STA $D900,X
55      STA $DA00,X
60      STA $DB00,X
```

Below you will find the above code in the form of a BASIC loader program. If you do not yet possess an assembler then you can key in the BASIC loader, RUN it, correct any errors and then call SYS 49152. If all is well the screen will fill in a flash with $ characters. If you are not yet convinced that programming in machine code is a good idea, then the proof may be to compare the speed of the above code to that of the same program in BASIC:

```
10      FOR X=0 TO 999
20      POKE 1024+X,36
30      POKE 55296+X,0
40      NEXT
```

Screen Fill BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 49151+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA162,0,169,36,157,0,4,157
112 DATA0,5,157,0,6,157,0,7
113 DATA169,0,157,0,216,157,0,217
114 DATA157,0,218,157,0,219,202,208
115 DATA225,96
116 DATA-3415
117 DATA END
```

Decimal to Hex Converter

This is a short but quite elegant routine which is somewhat more complicated to understand than the screen fill but also rather more useful.

We will assemble the converter to $C000 so that to use it we SYS 49152,N where N is the decimal value (an integer number from 0 to 65535) which is to be converted into its hexadecimal equivalent.

As you can see, unlike the screen fill, the Dec to Hex Converter requires input data to work on but fortunately for us there are three operating system routines which we can use to make the input and output of data a much simpler operation than it would be if we had to write all the code ourselves. Indeed, we needn't concern ourselves with the mechanics at all! Since we are 'borrowing' the in-built routines of the 64 we only need to know the address of the routine, the values needed by the routine and the output result. For this program we will use:

$AEFD handles a comma after the SYS

$AD8A

$B7F7 get in the decimal number (0–65535) after the SYS as two-byte binary value in the temporary integer store (address $14,$15)

$FFD2 outputs an ASCII byte held in the Accumulator to the screen

There is no need to understand exactly how these routines work at this stage but we will be seeing much more of them later on in the book. Let us consider a rough description of the method or algorithm for the program:

1. Check for the comma after the SYS
2. Get in the decimal number and convert to a two-byte binary number
3. Output the *high* byte as two hex digits
4. Output the *low* byte as two hex digits
5. Return to BASIC

Steps 1. and 2. above are handled by JSR'ing to our first two ROM routines at $AEFD and $AD8A/B7F7, the result from the latter being returned as a two-byte binary number in $14 (*low* byte) and $15 (*high* byte).

```
        * = $C000   program is
                    assembled to $C000
10      JSR $AEFD   check for a comma
                    after the SYS. If
                    not present
                    generate a syntax
                    error message and
                    return to BASIC
```

```
12      JSR $AD8A
13      JSR $B7F7   get the decimal
                    number after the
                    comma as binary in
                    $14,$15 (low
                    byte,high byte)
14      LDA $15     load the
                    accumulator with
                    the HIGH byte
16      JSR HEX     JSR to HEX which
                    outputs .A to the
                    screen as 2 hex
                    digits
18      LDA $14     load .A with the
                    LOW byte for HEX
```

Steps 4. and 5. of our algorithm rely on a subroutine HEX to output a binary number in A to the screen as two hex digits. HEX works by dividing the accumulator contents into two 4-bit 'nybbles' each of which is converted to a single ASCII digit 0–F. This is output to the screen via the ROM routine at $FFD2 thus:

1. Temporarily store the A register value on the stack (does not affect contents of A)
2. Shift A *right* four times to obtain the upper nybble
3. Jump to subroutine at step 6
4. Retrieve original A value from the stack back into A
5. AND the A register with #$0E to obtain the lower nybble.
6. If nybble value less than 10 (#$0A) then go to 9
7. Make A=A+#$37
8. Return via routine at $FFD2
9. A=A+#$30
10. Return via routine at $FFD2

Here is the corresponding code:

```
20 HEX  PHA         store .A on the
                    stack
22      LSR A       shift .A to the
                    right 4 times to
                    obtain the upper
                    nybble in .A
24      LSR A
26      LSR A
```

```
28        LSR A
30        JSR PUT     process upper
                      nybble
32        PLA         retrieve .A from
                      the stack
34        AND #$0F    AND .A with
                      #%00001111 (#$0F)
                      to obtain the
                      lower nybble in
                      .A
36 PUT    CMP #$0A    compare nybble in
                      .A with 10 (#$0A)
38        BCC GUN     if less than 10
                      relative branch
                      to GUN
40        ADC #$06    add #$06 to .A +
                      carry (=1) making
                      effectively add
                      #$07 - carry is
                      cleared by this
                      addition
42 GUN    ADC #$30    add #$30 to .A +
                      carry (=0) making
                      effectively add
                      #$30
44        JMP $FFD2   return from
                      subroutine via
                      the ROM routine
                      at $FFD2 which
                      outputs the .A
                      value to the
                      screen as the
                      corresponding
                      ASCII character
```

Note that in lines 14–20 use is made of the subroutine labelled as HEX to successively process the high and low bytes of the input data. This avoids the clumsy construction:

```
LDA $15     process $15
JSR HEX
LDA $14     process $14
JSR HEX
RTS         return to BASIC
```

By placing the HEX subroutine at line 20 the final two instructions are no longer required as the program returns to BASIC when it meets the RTS instruction in HEX. The same applies to the PUT subroutine in lines 30–36. The whole program can be thought of as three subroutines:

1. The whole program
2. The HEX subroutine
3. The PUT subroutine

All of these return via a single RTS instruction within the ROM routine at address $FFD2. Upon entering the ADC in line 40 the Carry is *set* after the CMP operation in line 36. This means that this add becomes effectively (+6+the Carry), and the Carry is 1, i.e. (+7). The Carry will be cleared by this addition for the subsequent ADC in line 38.

As with the screen fill routine this program is also coded in the form of a BASIC loader for those of you who do not possess an assembler. Type in the loader program below and then RUN it. After correcting any errors, and RUNning it successfully the routine will be ready to use.

Decimal/Hex BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 49151+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA32,253,174,32,138,173,32,247
112 DATA183,165,21,32,16,192,165,20
113 DATA72,74,74,74,74,32,27,192
114 DATA104,41,15,201,10,144,2,105
115 DATA6,105,48,76,210,255
116 DATA-3816
117 DATA END
```

This same technique, of presenting assembler listings, with descriptions in the text, and BASIC loader programs, is followed for all the programs in this book. The next chapter presents the first of the major programs, a machine code monitor. Don't be put off by the size of the listing – the principles are no more complex than those dealt with above.

4 The Wedge-MON monitor

The 'Wedge-MON' program presented here is an assembler and machine code program development tool that enables the programmer to enter programs, and examine, alter and generally manipulate in useful ways the contents of the computer's memory. In its fully expanded form it offers fourteen functions. These are given below, with their definitions and the syntax of the commands. Use of the commands is described later in the chapter, as is the program itself. First the commands and the practicalities of entering the program.

The 'Wedge-MON' functions

DISASSEMBLE: disassemble a line of machine code. This reverses the assembly process, and takes the values held in the sequence of memory bytes specified, decodes it and turns it into assembler code. Using the <cursor down> key, which may be done repeatedly, disassembles the code a line at a time.

Syntax: @D (start) - (finish)
or: @D (start) <cursor down>

Example: @D C000-0010
or: @D C000 <cursor down>

ASSEMBLE: assemble a line of machine code. This takes the assembler code, assigns the correct op code by interpreting the addressing mode syntax and places the byte(s) for the op code and operand values into memory.

Syntax: @A (addr) (mnemonic) (operand)

Example:
```
@A C000 LDA #$10
@A C002 STA $1000
@A C005 RTS
```

MEMORY DUMP: display memory as hexadecimal values. The specified range of memory addresses are displayed, with their contents given in hexadecimal. The <cursor down> key may be used repeatedly, to display an eight-byte sequence of memory for each press.

Syntax: @M (start)-(finish)
or: @M (start) <cursor down>

Example: @M C000-C010
or: @M C000 <cursor down>

MEMORY WRITE: write hexadecimal bytes to memory. This provides the facility of entering or altering each byte of memory. Eight bytes following the specified address are entered at a time. This is a companion command to @M.

Syntax: @W (addr) (byte) (byte) . . .

Example: @W C000 FF DE 10 1E 55 AA FF CE

ASCII DUMP: display memory as ASCII bytes. This is the same as @M, but the display is given as the characters corresponding to the values stored in memory. Mainly of use in dealing with data stored in memory.

Syntax: @I (start)-(finish)
or: @I (start) <cursor down>

Example: @I C000-C00F
or: @I C000 <cursor down>

ASCII WRITE: write ASCII character codes to memory. The same as @W, except that the ASCII code values of the eight-byte sequence entered are placed in memory.

Syntax: @C (addr) (ASCII char)(ASCII char) . . .

Example: @C C000 RT..*34.
@C C008 BASIC*..

TRANSFER: transfer a block of memory. This command enables a specified block of memory to be shifted to a new position in memory, defined by the start address given.

Syntax: @T (start)-(finish),(new start)

Example: @T C000-C010,8000

FILL: fill a block of memory with a given value. The area of memory specified by the start and finish addresses is overwritten, so that each byte will contain the specified value.

Syntax: @F (start)-(finish),(byte value)

Example: @F 0200-02FF,00

HUNT: hunt (search) for a specified sequence of bytes in a section of memory. The sequence of bytes can be specified as hexadecimal values, with the * symbol functioning as a 'wild card' which may signify any value, or in ASCII characters, following an apostrophe.

Syntax: @H (start)-(finish),(byte) (byte)..
or: @H (start)-(finish),'(ASCII chars)

Example: @H A000-F000,23 60 21
@H A000-F000,'BASIC
@H A000-F000,23 * 21

SAVE: save a specified block of memory to tape or disk. The command is followed by the name under which the memory sequence is to be saved, the device number, and the start and end address of the portion of memory which is to be stored, separated by commas.

Syntax: @S "filename",(device no.),(start),(finish)

Example: @S "RAT",08,C000,C100 (disc)
@S "RAT",01,C000,C100 (tape)

LOAD: load a program from tape or disk. @S would not be much use without this command, which simply places a stored memory sequence back where it came from.

Syntax: @L "filename",(device no.)

Example: @L "RAT",08 (to load from disc)
@L "RAT",01 (to load from tape)

GO: transfer control to a user machine code routine. The address specifies the start location of the routine.

Syntax: @G (address)

Example: @G 2000

REGISTERS: initialize the registers prior to using the GO command (@G). The values of the A, X, Y and S (status) registers are specified, separated by spaces.

Syntax: @R (A) (X) (Y) (S)

Example: @R FF 00 01 30

EXIT: quit 'Wedge-MON'. This enables you to switch out the monitor if required.

Syntax: @X

Entering 'Wedge-MON'

'Wedge-MON' is a large program offering a large range of functions, and if you had to enter the whole lot in in one go there would be a great deal of typing to be done before you saw any results. For this reason the program is broken down into eight modules: a main program and seven command files. While the control program *must* be entered, the command files are 'optional' and you can enter them one by one whenever you feel like typing them in. Each file entered adds one or more new commands to the monitor, so that 'Wedge-MON' can be gradually built up to its complete version. In case you think of any additional commands which would be useful, I will show you how to incorporate your own command routines once you are competent in machine code. Here are the modules:

1. The Control Program: This module contains the code to accept data input from the keyboard, and direct control to the appropriate command routines. It also contains a 'kernal' of input-output routines which are used by all the 'Wedge-MON' commands. Because of the central role it plays this file *must* be entered before any of the command modules, and is also the only 'Wedge-MON' file that is indispensable.

2. @A (assemble) and @D (disassemble): This is the largest of the 'Wedge-MON' command files, but also perhaps the most useful. These two commands are placed together in one file since they use certain large data tables in common.

3. @M (memory dump), @W (memory write), @I (ASCII dump) and @C (ASCII write): This file adds four extremely useful commands to the monitor, and since it is comparatively short it is probably the best one to enter first if you can't wait to see some action.

4. @T (memory transfer): This is another short file that adds the capability to moving specified blocks of memory around within the 64. These blocks can be machine code programs, data tables, etc.

5. @F (fill memory): This is the shortest of the command files, providing a potentially very useful facility for the debugging of machine code programs.

6. @H (hunt memory): This search facility is useful for exploring the 64's ROMS as well as helping program development, and is best used in combination with files 2. and 3. of 'Wedge-MON'.

7. @G (go to machine code) and @R (set registers): These commands are mainly used with @A, but are also useful for accessing and investigating operating system code routines within the 64's ROMS.

8. @S (save machine code) and @L (load machine code): If you were only to enter this file combined with say, file 3, you would be getting a powerful utility. These @S and @L commands are obviously crucial if you are going to be writing your own machine code routines.

Entering the Control Program

All the above files are provided as both assembler code source files and in BASIC hex loader program format. If you possess an assembler then you can assemble the source file, otherwise key in the BASIC loader. The BASIC loader program is equipped with a checksum facility to help track down typing errors. The loader is divided into blocks of 64 bytes of DATA, each terminated by a negative checksum value which is the sum of all 64 bytes in the preceding block. When you RUN the loader program it POKES each sequence of 64 bytes into memory and then sums them, comparing this sum with the checksum value at the end. If the values do not match then an error will be flagged for that DATA block, and you must find and correct the error before trying again. This system is not quite 100 per cent reliable, but it should, unless you are unlucky enough to enter a set of errors which cancel out in the addition process, spot all typing errors.

Once the loader is entered and you have saved it to disc or tape (very important!), enter RUN (return) and wait. The amount of time taken to POKE the data into memory will depend on the size of the file, but for the control program this will be around half a minute. When the loader program has successfully finished type SYS 49152 <return> and you should be greeted with the message *** WEDGE-MON 1984 J.P. GIBBONS ***. If so, all is well, but at this stage, unless you

have also entered one of the command files you will find that only one of the 'Wedge-MON' commands actually works. This is @X, which dis-enables 'Wedge-MON' and the command routine for which is contained within the control program itself.

The first job now is to save 'Wedge-MON' as a machine code file, so that you don't have to repeat the preceding rigmarole every time you want to load your monitor. This is achieved by the following BASIC program which simply PEEKS memory locations and PRINT#'s the values to the cassette or disc file.

Cassette Version:

```
20 OPEN5,1,5,"WEDGE-MON,P,W"
30 PRINT#5,CHR$(0);CHR$(192);
40 FORX=49152TO52736
50 PRINT#5,CHR$(PEEK(X));
60 NEXTX
70 CLOSE5
```

Disc Version:

```
10 OPEN15,8,15,"I0"
20 OPEN5,8,5,"@0:WEDGE-MON,P,W"
30 PRINT#5,CHR$(0);CHR$(192);
40 FORX=49152TO52736
50 PRINT#5,CHR$(PEEK(X));
60 NEXTX
70 CLOSE5
80 CLOSE15
```

Note that the range of memory saved is that occupied by the control program plus *all* the command files. This is so that the same program may be used at later stages to save intermediate versions of 'Wedge-MON' consisting of the control program plus one or more commands. After this, whenever you require 'Wedge-MON' simply load it from tape or disc, using BASIC's LOAD command. be sure to remember, however, to add ',1' to the command to tell BASIC that it is a machine code file that is being LOADed and not a BASIC program. For disc, use:

LOAD "WEDGE-MON",8,1

and for tape:

LOAD "WEDGE-MON",1,1

Important note: When LOADing programs in this way, you *must* enter NEW *after* the LOAD is finished, to be sure of a system reset. The program is not affected, and is activated by SYS 49152.

Entering the Command Files

Each of the command files is provided, like the control program, in both assembler source files and BASIC loader program versions. These are entered exactly as described for the control module, except that the control program or the latest intermediate version of 'Wedge-MON' *must* be loaded first. As well as loading the command routine into memory it is also necessary to inform the control program that it has a new command or commands available. This is the function of the POKEs at the end of each BASIC loader program. So, if using the loader program, after typing RUN <return> to load the machine code into memory you must then enter RUN 1000 <return> to link the added command(s) into the monitor.

Note: Those of you entering the assembler source code directly must nonetheless execute these POKEs, otherwise the new commands will not work. There are two POKEs for each new command added. To enable 'Wedge-MON' enter SYS 49152 <return>, as before.

When a new command has been loaded and linked this latest intermediate stage of 'Wedge-MON' must be saved as a machine code file, using either the BASIC memory save routine given above or the @S monitor command if you have entered this module. In the latter case, you should use for discs:

@S "@0:WEDGE-MON",08,C000 – CE00

and, for tape:

@S "WEDGE-MON",01,C000 – CE00

Using the 'Wedge-MON' Commands

'Wedge-MON' has been designed to work in parallel to BASIC, and should have no effect on normal BASIC processing other than slowing it down slightly, which may be avoided (see below, on @X use). All the monitors commands are direct-mode statements, that is they will not work from within a BASIC program, and if you try to incorporate them thus the normal BASIC error messages will be generated. In this section some examples of the use of the monitor commands will be given.

Using the @D command
With the @D command in memory you may use 'Wedge-MON' to examine its own code, by typing @D C000 and pressing <return>. You should then see on the screen:

```
@D C000 LDX #$02
```

This is the first instruction of 'Wedge-MON' (which starts at $C000) disassembled (load the X register with the value 2). Press <cursor down> and the disassembly continues with the next line of assembler. To escape the command, press any other key.

Using the @A, @G and @R commands
Try typing the following:

```
@A CE00  LDA #$00
```

Hit the <return> key, and you will then see:

```
@A CE00  LDA #$00
@A  CE02
```

You are now using the Assemble command @A to enter assembler code starting at address $CE00. The instruction that you typed (load the accumulator with the value zero) has been accepted by 'Wedge-MON' as a valid line, which we know because it has printed the address of the next available memory location on the following line, slightly indented from the first. We may now enter the rest of the program, following each line with <return>:

```
STA $D020

RTS
```

You should now see:

```
@A CE00  LDA #$00
@A CE02  STA $D020
@A CE05  RTS
@A  CE06
```

Now move the cursor down a couple of lines and then type:

```
@G CE00
```

What should have happened is that the border colour changed to black. You have written and executed a small machine code routine to change the border colour. The border colour is controlled from the VICII chip register located at $D020, and by storing a zero in this register the border turns black. You will also notice that the values of the processor registers are displayed under the register headings in @R format on returning from the routine. These register values may be intialised prior to execution of a machine code routine by the @R command, and the purpose of the display being preceded by @R is to make the modification of register values simpler. Cursor up a line, and edit the AC value to 0E, followed by <return>, and then enter:

```
@G CE02
```

plus <return> again, and the border colour reverts to normal. This is because we have initialized the accumulator with the value ($0E) normally held in $D020 and then jumped into our routine at address $CE02, i.e. *after* the instruction at $CE00 where the accumulator is loaded with zero.

Using the @M and @W commands
If you enter the following, followed by <return>:

```
@M D020
```

You should see a display something like this:

```
@M D020  0E 06 FF FF FF FF FF FF
```

These values represent the contents of the VIC chip registers from $D020 to $D027. (Note that the values given above may be different from the ones you get, depending on how you have your machine set up.) To enter new values you must either move the cursor up to the @M and change it to @W, and then simply make your changes on the displayed line before typing <return>, or type:

```
@W D020
```

and then simply enter your new values. When you hit the <return> key the next available address will be displayed, as with the @A command.

The @I and @C commands work in similar fashion, but with the ASCII characters instead of hex numbers being displayed and entered.

Note: Do not @C over an area of code if you want it to retain its function. If you @I an area of machine code to the screen as ASCII characters it will appear as gibberish, but you may do this without harming the code. If however you change the command letter to @C and then try to re-enter the ASCII that you have dumped then the code will be terminally altered, probably leading to a crash the next time that you try to execute it. The reason for this will become clear when we take the 'Wedge-MON' program apart.

Using the @L and @S commands

Commodore BASIC lacks a command whereby machine code programs may be saved to a storage device, although ROM routines exist to perform this function in machine code. Using the @S command, an area of memory may be saved to tape or disc. The area of memory concerned does not of course necessarily have to be a machine code program, and the data for a single sprite, a block of sprites, a screen display or other types of data may be saved just as easily.

The @L command may seem redundant, since BASIC already has a machine code load facility, using LOAD "file",device,1, but this operation tends to corrupt BASIC pointers and requires a NEW statement after loading to be sure of a system reset. Using @L, however, you can load in machine code without disturbing any BASIC program you may already have in memory.

Using the @H, @F and @T commands

Your use of these commands will probably be largely for debugging and exploration purposes. To illustrate the use of @H, imagine that you were examining an area of the ROM and you wanted to investigate all the occurrences of the instruction PHP within this area. You would simply use the @H command to look for the op code for PHP. To give a slightly more tricky example you may wish to discover all the JSR subroutine calls to ROM Kernal routines, i.e. JSR $FF?? instruction sequences between $A000 and $FFFF. This can be accomplished thus:

```
@H A000-FFFE,20 * FF
```

$20 is the op code for JSR and $FF the high byte of the destination address. Since the low byte of the destination is indeterminate we must use the 'wildcard' character * to represent it. Hit the <return> key and prepare for the rush. As might be expected there are quite a few JSR's to Kernal routines in the ROM! In practice, for greater utility, we would have to narrow our search to a smaller area of memory. You should note that the upper limit for the search was set at $FFFE and not $FFFF. This is a limitation of the @H command, and if you

should set the upper limit for a search to $FFFF the program will carry on searching indefinitely. Fixing this would have made the code longer, so a compromise seemed in order. The contents of $FFFF are for most purposes invariant in any case.

The memory fill command @F can be used to fill the screen memory with a certain character code, the whole colour table with a specific colour code, an area of memory with NOP codes or zeroes, fill in a sprite data block, initialise the values in a data table, etc. The many uses of this command will become apparent as you experiment further.

The @T command enables the contents of memory blocks to be shuffled about. For instance, suppose you had defined a sprite in Sprite block 13. You could then copy the contents of Sprite block 13 to the block 14 locations, etc. Another important use would be to copy the machine code ROM routines down into the RAM memory so that they may be experimented with or altered. It is important to remember that when moving blocks of machine code from one area of RAM to another the routines will not run at the new location if the code contains any absolute JMP's or JSR's to routines within the transferred block. These instructions, and also any word tables, will have to be altered by hand, according to the offset of the transfer. However, all relative branches and absolute JMP's or JSR's to routines *outside* the transfer block should work as before.

Using @X

'Wedge-MON' when activated will have a slight slowing effect on BASIC processing, and so a quit facility is provided in @X to allow the monitor to be switched off if the full speed of BASIC is required. To reactivate you need only use SYS 49152.

Having reviewed what 'Wedge-MON' can do, here is the first batch of code. This is the control section, the main module which handles all the others.

The Control Program Code

First we give the assembler listing, to which all the following discussion of the program refers, and then the BASIC loader program.

Assembler Listing: The Control Program

```
1000 SYS 700
1010 .OPT OO
1020 *=$C000
1030 ;----------------
```

```
1040 TEMP1   =$14
1050 TEMP2   =$1D
1060 CURR    =$4D;*
1070 END     =$4F;*
1080 PFLAG   =$FE
1090 ;----------------
1100 FNCTN   =$02A7;**
1110 CHAR    =$02AA
1120 ;----------------
1130 DSASBL =$C300
1140 MEMORY =$C880
1150 INTERG =$C8A0
1160 ;----------------
1170 ;--ENTRY POINT--
1180 ;SET UP CHRGET WEDGE BY INSERTING
1190 ;'JMP WEDGE' INTO CHRGET ROUTINE/
1200 ;PRINT COPYRIGHT MESSAGE/RETURN TO
1210 ;BASIC
1220 ;----------------
1230 ENTRY   LDX #$02
1240 HA      LDA RAT,X
1250 ↑       STA $007C,X
1260 ↑       DEX
1270 ↑       BPL HA
1280 ↑       LDY #>CREDIT
1290 ↑       LDA #<CREDIT
1300 ↑       JMP $AB1E
1310 ;----------------
1320 RAT     JMP WEDGE
1330 ;----------------
1340 ;CHECK FOR '@'-IF PRESENT JUMP TO
1350 ;MAIN CODE ELSE CONTINUE WITH
1360 ;MORMAL CHRGET ROUTINE.
1370 ;----------
1380 WEDGE   CMP #$40       ;'@
1390 ↑       BEQ CHECK
1400 ;----------
1410 OUT     CMP #$3A       ;':
1420 ↑       BCS TACITE
1430 ↑       JMP $0080
1440 ;----------------
1450 ;CHECK SOURCE OF CHRGET CALL BY
1460 ;PULLING RETURN ADDRESS OFF STACK-
1470 ;IF NOT $A48C THEN IGNORE
```

```
1480 ;----------------
1490 CHECK  PLA
1500 ↑      CMP #$8C
1510 ↑      BNE JACKIT
1520 ↑      PLA
1530 ↑      CMP #$A4
1540 ↑      BEQ CODE
1550 ↑      PHA
1560 ↑      LDA #$8C
1570 JACKIT PHA
1580 ↑      LDA #$40       ;'@
1590 ↑      SEC
1600 TACITE RTS
1610 ;----------------
1620 ;--CONTROL SECTION--
1630 ;REPLACE RETURN ADDRESS ON STACK/
1640 ;CHECK INPUT BUFFER FOR CHAR AFTER
1650 ;'@'-IF NONE THEN EXIT/CHECK CHAR
1660 ;AGAINST COMTAB TABLE-IF NO MATCH
1670 ;THEN EXIT/CHAR STORED IN CHAR/
1680 ;USE INDEX TO FETCH CORRESPONDING
1690 ;ADDRESSES FROM COMAD TABLES/
1700 ;COMAD1=INDIRECT JUMP ADDRESS IN
1710 ;FNCTN+1,+2/COMAD2 IS PUSHED ON
1720 ;THE STACK/JMP TO LATTER VIA RTS
1730 ;----------------
1740 CODE   PHA
1750 ↑      LDA #$8C
1760 ↑      PHA
1770 ↑      LDX #$4C
1780 ↑      STX FNCTN
1790 ↑      LDX #$00
1800 ↑      STX PFLAG
1810 ;----------
1820 COMPS  JSR $0073
1830 ↑      BNE TRA
1840 ESCAP  JMP EXIT1
1850 ;----------
1860 TRA    STA CHAR
1870 ↑      LDX #$FF
1880 TRA1   INX
1890 ↑      LDA COMTAB,X
1900 ↑      BEQ ESCAP
1910 ↑      CMP CHAR
```

```
1920 ↑       BNE TRA1
1930 ↑       TXA
1940 ↑       ASL A
1950 ↑       TAX
1960 ↑       LDA COMAD1,X
1970 ↑       STA FNCTN+1
1980 ↑       LDA COMAD1+1,X
1990 ↑       STA FNCTN+2
2000 ;----------
2010 ↑       LDA COMAD2+1,X
2020 ↑       PHA
2030 ↑       LDA COMAD2,X
2040 ↑       PHA
2050 ↑       RTS
2060 ;----------------
2070 ;--MAIN LOOP FOR @D,@M AND @I--
2080 ;GET IN HEX PARAMETERS
2090 ;(IE. START-END OR JUST START)-
2100 ;EXIT IF ERRONEOUS-SET FLAG IF
2110 ;ONLY ONE PARAMETER GIVEN/JSR TO
2120 ;COMMAND ROUTINE THROUGH INDIRECT
2130 ;JSR AT FNCTN/CURSOR BACK TO
2140 ;START OF LINE/CHECK IF CURRENT
2150 ;ADDRESS=END PARAMETER-IF NO THEN
2160 ;ROUND LOOP AGAIN FOR NEXT LINE
2170 ;ELSE IF YES OR IF '1 PARAMETER'
2180 ;FLAG WAS SET OR IF [STOP] KEY
2190 ;IS PRESSED THEN SIT IN LOOP
2200 ;WAITING FOR A KEY FROM KEYBOARD-
2210 ;IF KEY=ANYTHING OTHER THAN
2220 ;[CURSOR DOWN] THEN EXIT ELSE LOOP
2230 ;BACK FOR ANOTHER LINE/BEFORE
2240 ;EXITING SET NO. CHARS IN KEYBOARD
2250 ;BUFFER TO ZERO AND MAKE SURE
2260 ;QUOTE MODE IS SWITCHED OFF
2270 ;----------------
2280 MAINLP JSR PARAMS
2290 ↑       BCS EXIT1
2300 ↑       BNE NXTLIN
2310 ↑       INC PFLAG
2320 ;----------
2330 NXTLIN JSR FNCTN
2340 ;----------
2350 ↑       JSR HOMCSR
```

```
2360 ;----------
2370 ↑       JSR COMPAR
2380 ↑       BCC ROUND
2390 ;----------
2400 KEY     LDA #0
2410 ↑       STA $C6
2420 KEY1    LDA $C6
2430 ↑       BEQ KEY1
2440 ↑       LDA $028D
2450 ↑       BNE EXIT0
2460 ↑       LDA $C5
2470 ↑       CMP #$07
2480 ↑       BNE EXIT0
2490 ;----------
2500 ROUND   LDA #$0D        ;'RET
2510 ↑       JSR $FFD2
2520 ↑       JSR NXTBYT
2530 ↑       JMP NXTLIN
2540 ;----------
2550 EXIT0   LDA #0
2560 ↑       STA $C6
2570 ↑       STA $D4
2580 ;----------
2590 EXIT1   LDA #0
2600 ↑       JMP $0080
2610 ;----------------
2620 ;COMMON EXIT FOR '@A' '@W' AND
2630 ;'@C' COMMANDS- FIRST STEP IS THE
2640 ;CORRESPONDING REVERSE PROCEDURE
2650 ;EFFECTED BY A JSR TO THE
2660 ;COMMAND ROUTINE VIA THE INDIRECT
2670 ;JSR AT FNCTN - EG. FOR '@A'
2680 ;THE INSTRUCTION AT THE CURRENT
2690 ;ADDRESS IS DISASSEMBLED OVER THE
2700 ;INPUT LINE/PRINT 'RET' TO MOVE
2710 ;DOWN ONE LINE/INCREMENT CURRENT
2720 ;ADDRESS POINTER/PRINT '@'/PRINT
2730 ;THE COMMAND CHARACTER (STORED IN
2740 ;CHAR)/PRINT THE CURRENT ADDRESS
2750 ;IN HEX/PUT [CURSOR UP],[CURSOR
2760 ;DOWN] IN KEYBOARD BUFFER/PRINT
2770 ;3 SPACES/EXIT
2780 ;----------------
2790 COMEXT JSR FNCTN
```

```
2800 ↑       LDA #$0D        ;'RET
2810 ↑       JSR $FFD2
2820 ↑       JSR NXTBYT
2830 ↑       LDA #$40
2840 ↑       JSR $FFD2
2850 ↑       LDA CHAR
2860 ↑       JSR $FFD2
2870 ↑       LDX #$02
2880 ↑       JSR SPACES
2890 ↑       LDA CURR+1
2900 ↑       JSR OUTPUT
2910 ↑       LDA CURR
2920 ↑       JSR OUTPUT
2930 ↑       LDA #$91        ;'CURS UP
2940 ↑       STA $0277
2950 ↑       LDA #$11        ;'CURS DOWN
2960 ↑       STA $0278
2970 ↑       LDA #$02
2980 ↑       STA $C6
2990 ↑       LDX #$02
3000 ↑       JSR SPACES
3010 ↑       JMP EXIT1
3020 ;----------------
3030 ;--GET TWO BYTE HEX PARAMETERS--
3040 ;GET IN HEX PARAMETERS FROM BUFFER
3050 ;(EITHER 1 PARAMETER OR START-END)
3060 ;-STORE IN 4D,4E AND 4F,50/IF
3070 ;PARAMETERS ERRONEOUS RETURN WITH
3080 ;CARRY SET-ELSE CARRY CLEAR/IF 1
3090 ;PARAMETER ONLY THEN BEQ SUCCEEDS
3100 ;ELSE BNE SUCCEEDS
3110 ;----------------
3120 PARAMS LDX #$FE
3130 ;----------
3140 PARENT INX
3150 ↑      INX
3160 ↑      JSR HEXGET
3170 ↑      BCS REJEK2
3180 ↑      STA CURR+1,X
3190 ↑      JSR HEXGET
3200 ↑      BCS REJEK2
3210 ↑      STA CURR,X
3220 ↑      TXA
3230 ↑      BNE REJEK2
```

```
3240 ;----------
3250 ↑       JSR $0073
3260 ↑       BEQ REJEK1
3270 ↑       CMP #$2D
3280 ↑       BEQ PARENT
3290 ;----------
3300 ↑       SEC
3310 ↑       RTS
3320 REJEK1 CLC
3330 REJEK2 RTS
3340 ;----------------
3350 ;--PRINT SPACES--
3360 ;IF ROUTINE IS ENTERED AT SPACE
3370 ;THEN PRINT 1 SPACE ELSE IF
3380 ;ENTERED AT SPACES THE PRINT X
3390 ;SPACES ELSE IF ENTERED AT ANYCHR
3400 ;PRINT X OF THE ASCII CODE IN A
3410 ;----------------
3420 SPACE  LDX #$01
3430 SPACES LDA #$20       ;'SPACE
3440 ANYCHR JSR $FFD2
3450 ↑       DEX
3460 ↑       BNE ANYCHR
3470 ↑       RTS
3480 ;----------------
3490 ;--OUTPUT CURRENT ADDRESS--
3500 ;PRINT '@'/ PRINT THE COMMAND
3510 ;CHARACTER/PRINT THE CURRENT
3520 ;ADDRESS IN HEX/PRINT 3 SPACES
3530 ;----------------
3540 ADROUT LDA #$40       ;'@
3550 ↑       JSR $FFD2
3560 ↑       LDA CHAR
3570 ↑       JSR $FFD2
3580 ↑       JSR SPACE
3590 ↑       LDA CURR+1
3600 ↑       JSR OUTPUT
3610 ↑       LDA CURR
3620 ↑       JSR OUTPUT
3630 ;----------
3640 ↑       LDX #$03
3650 ↑       JMP SPACES
3660 ;----------------
3670 ;--RETURN CURSOR TO COL 0--
```

```
3680 ;READ THE CURSOR POSITION Y,X/
3690 ;MAKE THE COLUMNS(Y)=0/RESET
3700 ;CURSOR POSITION-(IE.CURSOR BACK
3710 ;TO START OF LINE)
3720 ;----------------
3730 HOMCSR SEC
3740 ↑      JSR $FFF0
3750 ↑      LDY #0
3760 ↑      CLC
3770 ↑      JMP $FFF0
3780 ;----------------
3790 ;--OUTPUT HEX BYTE--
3800 ;TAKE HIGH NIBBLE OF A BINARY NO.
3810 ;AND PRINT IT AS HEX DIGIT/DO THE
3820 ;SAME FOR THE LOW NIBBLE
3830 ;----------------
3840 OUTPUT PHA
3850 ↑      LSR A
3860 ↑      LSR A
3870 ↑      LSR A
3880 ↑      LSR A
3890 ↑      JSR PUTIT
3900 ↑      PLA
3910 ↑      AND #$0F
3920 ;----------
3930 PUTIT  CMP #$0A
3940 ↑      BCC GUN
3950 ↑      ADC #$06
3960 GUN    ADC #$30
3970 ↑      JMP $FFD2
3980 ;----------------
3990 ;--GET IN A SINGLE HEX BYTE--
4000 ;GET HEX DIGIT IN .A FROM INPUT
4010 ;BUFFER/MULT TEMP1 BY 16/ADD .A
4020 ;/GET IN NEXT DIGIT/ADD TO TEMP1
4030 ;SEC AND EXIT IF NON-HEX CHAR
4040 ;ENCOUNTERED
4050 ;----------------
4060 HEXGET LDA #$01
4070 ↑      STA TEMP2
4080 ↑      LDA #0
4090 ↑      STA TEMP1
4100 ;----------
4110 GETIN  JSR $0073
```

```
4120 ↑       BCC NUMBR
4130 ↑       CMP #$47       ;'G
4140 ↑       BCS NOTOK
4150 ↑       CMP #$41       ;'A
4160 ↑       BCC NOTOK
4170 ;----------
4180 ALPHA   SBC #$08
4190 NUMBR   SBC #$2F
4200 ;----------
4210 ↑       ASL TEMP1
4220 ↑       ASL TEMP1
4230 ↑       ASL TEMP1
4240 ↑       ASL TEMP1
4250 ;----------
4260 ↑       CLC
4270 ↑       ADC TEMP1
4280 ↑       STA TEMP1
4290 ;----------
4300 ↑       DEC TEMP2
4310 ↑       BPL GETIN
4320 ↑       RTS
4330 ;----------
4340 NOTOK   SEC
4350 ↑       PHA
4360 ↑       PLA
4370 ↑       RTS
4380 ;---------------
4390 ;--GET NEXT BYTE--
4400 ;INCREMENT CURRENT ADDRESS POINTER
4410 ;/LOAD IN NEXT BYTE FROM (ADDRESS)
4420 ;---------------
4430 NXTBYT  INC CURR
4440 ↑       BNE RATS
4450 ↑       INC CURR+1
4460 RATS    LDY #0
4470 ↑       LDA (CURR),Y
4480 ↑       RTS
4490 ;---------------
4500 ;--COMPARE START TO END--
4510 ;IF [STOP] KEY PRESSED OR IF
4520 ;'1 PARAMETER' FLAG SET OR IF
4530 ;END PARAMETER(4F,50)=CURRENT ADDR
4540 ;(4D,4E) THEN EXIT CARRY SET
4550 ;ELSE EXIT CARRY CLEAR
```

```
4560 ;---------------
4570 COMPAR LDA $91
4580 ↑      BPL COPIT
4590 ;----------
4600 ↑      LDA PFLAG
4610 ↑      BNE COPIT
4620 ;----------
4630 ↑      LDA CURR+1
4640 ↑      CMP $50
4650 ↑      BCC HOPIT
4660 ↑      LDA CURR
4670 ↑      CMP END
4680 ↑      BCC HOPIT
4690 ;----------
4700 COPIT  SEC
4710 HOPIT  RTS
4720 ;---------------
4730 ;--GET ADDRESS--
4740 ;PRINT [CURSOR UP]/GET HEX
4750 ;ADDRS FROM INPUT BUFFER IN CURR,
4760 ;CURR+1/EXIT CARRY SET IF ILLEGAL
4770 ;---------------
4780 GETADR LDA #$91      ;'CURS UP
4790 ↑      JSR $FFD2
4800 ;----------
4810 ↑      JSR HEXGET
4820 ↑      BCS LUD
4830 ↑      STA CURR+1
4840 ↑      JSR HEXGET
4850 ↑      BCS LUD
4860 ↑      STA CURR
4870 LUD    RTS
4880 ;---------------
4890 ;--DISABLE WEDGE-MON--
4900 ;REPLACE THE 'JMP WEDGE' IN CHRGET
4910 ;WITH THE 3 NORMAL BYTES/PRINT OK
4920 ;EXIT
4930 ;----------
4940 KILLIT JSR $0073
4950 ↑      BNE KOUT
4960 ↑      LDX #$02
4970 KA     LDA CAT,X
4980 ↑      STA $007C,X
4990 ↑      DEX
5000 ↑      BPL KA
```

```
5010 ↑       BMI CNCLUD
5020 ;----------
5030 KOUT    JMP EXIT0
5040 ;---------------
5050 ;PRINT 'OK'/EXIT
5060 ;---------------
5070 CNCLUD LDA #$64
5080 ↑       LDY #$A3
5090 ↑       JSR $AB1E
5100 ↑       JMP EXIT0
5110 ;---------------
5120 ;REPLACEMENT BYTES FOR CHRGET
5130 ;ROUTINE - USED IN @X COMMAND
5140 ;---------------
5150 CAT     .BYTE $C9,$3A,$B0
5160 ;---------------
5170 ;STRINGS TO BE PRINTED BY THE ROM
5180 ;ROUTINE $AB1E
5190 ;---------------
5200 CREDIT .BYTE $0D
5210 ↑       .ASC "    ** WEDGE-MON "
5220 ↑       .ASC "1984 JP.GIBBONS *"
5230 ↑       .ASC "*"
5240 ↑       .BYTE $0D,$00
5250 ;---------------
5260 ;ASCII TABLE OF COMMAND CHARACTERS
5270 ;---------------
5280 COMTAB .ASC  "DMIAWCTFHSLGRX"
5290 ↑       .BYTE $00,$00,$00,$00,$00
5300 ↑       .BYTE $00,$00,$00,$00,$00
5310 ;---------------
5320 ;TABLES OF ACTION ADDRESSES FOR
5330 ;COMMAND DISPATCH
5340 ;---------------
5350 COMAD1 .WORD DSASBL,MEMORY,INTERG
5360 ↑       .WORD DSASBL,MEMORY,INTERG
5370 ;---------------
5380 COMAD2 .WORD EXIT0-1,EXIT0-1
5390 ↑       .WORD EXIT0-1,EXIT0-1
5400 ↑       .WORD EXIT0-1,EXIT0-1
5410 ↑       .WORD EXIT0-1,EXIT0-1
5420 ↑       .WORD EXIT0-1,EXIT0-1
5430 ↑       .WORD EXIT0-1,EXIT0-1
5440 ↑       .WORD EXIT0-1,KILLIT-1
5450 ;---------------
```

BASIC Loader: The Control Program

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 49151+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA162,2,189,17,192,149,124,202
112 DATA16,248,160,193,169,213,76,30
113 DATA171,76,20,192,201,64,240,7
114 DATA201,58,176,20,76,128,0,104
115 DATA201,140,208,8,104,201,164,240
116 DATA8,72,169,140,72,169,64,56
117 DATA96,72,169,140,72,162,76,142
118 DATA167,2,162,0,134,254,32,115
119 DATA-7687
120 REM --- BLOCK 2
121 DATA0,208,3,76,166,192,141,170
122 DATA2,162,255,232,189,251,193,240
123 DATA242,205,170,2,208,245,138,10
124 DATA170,189,19,194,141,168,2,189
125 DATA20,194,141,169,2,189,32,194
126 DATA72,189,31,194,72,96,32,230
127 DATA192,176,51,208,2,230,254,32
128 DATA167,2,32,48,193,32,138,193
129 DATA-8579
130 REM --- BLOCK 3
131 DATA144,19,169,0,133,198,165,198
132 DATA240,252,173,141,2,208,17,165
133 DATA197,201,7,208,11,169,13,32
134 DATA210,255,32,127,193,76,119,192
135 DATA169,0,133,198,133,212,169,0
136 DATA76,128,0,32,167,2,169,13
137 DATA32,210,255,32,127,193,169,64
138 DATA32,210,255,173,170,2,32,210
139 DATA-8033
140 REM --- BLOCK 4
141 DATA255,162,2,32,10,193,165,78
142 DATA32,58,193,165,77,32,58,193
```

```
143 DATA169,145,141,119,2,169,17,141
144 DATA120,2,169,2,133,198,162,2
145 DATA32,10,193,76,166,192,162,254
146 DATA232,232,32,80,193,176,24,149
147 DATA78,32,80,193,176,17,149,77
148 DATA138,208,12,32,115,0,240,6
149 DATA-7152
150 REM --- BLOCK 5
151 DATA201,45,240,228,56,96,24,96
152 DATA162,1,169,32,32,210,255,202
153 DATA208,250,96,169,64,32,210,255
154 DATA173,170,2,32,210,255,32,8
155 DATA193,165,78,32,58,193,165,77
156 DATA32,58,193,162,3,76,10,193
157 DATA56,32,240,255,160,0,24,76
158 DATA240,255,72,74,74,74,74,32
159 DATA-7641
160 REM --- BLOCK 6
161 DATA69,193,104,41,15,201,10,144
162 DATA2,105,6,105,48,76,210,255
163 DATA169,1,133,29,169,0,133,20
164 DATA32,115,0,144,10,201,71,176
165 DATA26,201,65,144,22,233,8,233
166 DATA47,6,20,6,20,6,20,6
167 DATA20,24,101,20,133,20,198,29
168 DATA16,222,96,56,72,104,96,230
169 DATA-5487
170 REM --- BLOCK 7
171 DATA77,208,2,230,78,160,0,177
172 DATA77,96,165,145,16,16,165,254
173 DATA208,12,165,78,197,80,144,7
174 DATA165,77,197,79,144,1,56,96
175 DATA169,145,32,210,255,32,80,193
176 DATA176,9,133,78,32,80,193,176
177 DATA2,133,77,96,32,115,0,208
178 DATA12,162,2,189,210,193,149,124
179 DATA-7269
180 REM --- BLOCK 8
181 DATA202,16,248,48,3,76,160,192
182 DATA169,100,160,163,32,30,171,76
183 DATA160,192,201,58,176,13,32,32
184 DATA32,32,42,42,32,87,69,68
185 DATA71,69,45,77,79,78,32,49
186 DATA57,56,52,32,74,80,46,71
```

```
187 DATA73,66,66,79,78,83,32,42
188 DATA42,13,0,68,77,73,65,87
189 DATA-5026
190 REM --- BLOCK 9
191 DATA67,84,70,72,83,76,71,82
192 DATA88,0,0,0,0,0,0,0
193 DATA0,0,0,0,195,128,200,160
194 DATA200,0,195,128,200,160,200,159
195 DATA192,159,192,159,192,159,192,159
196 DATA192,159,192,159,192,159,192,159
197 DATA192,159,192,159,192,159,192,159
198 DATA192,179,193
199 DATA-7394
200 DATA END
```

Wedge-MON Design and Coding

At 2.6K in length 'Wedge-MON' is by BASIC standards a pretty short program, but in assembler terms it is getting towards the sort of length you would expect of a professional utility. To give some idea of scale a similar, but more comprehensive, commercial monitor program occupies 4K, commercial 64 games are normally 4–16K, and the whole ROM is just about 20K in length if you include the character generator. 'Wedge-MON' is thus larger than anything you would normally expect to find listed in a book or magazine, but hardly exceptional in terms of a functional utility. Designing a large program requires a lot of thought if one is not to end up with repetitive and disordered code needing to be rewritten two dozen times. With careful planning and forethought you can usually manage to keep it down to an even dozen! By way of a demonstration of the process the section which follows goes through the stages of designing this program. First we need to state in general terms the basic requirements of 'Wedge-MON':

1. We need to be able to type commands at the keyboard which we can intercept before the BASIC interpreter gets them and either misinterprets them or declares them as syntax errors.

2. On the basis of these commands we need to be able to transfer control to the appropriate command-handling routines.

3. Command-handling routines will be needed to get in parameters from the keyboard and in response either output results to the screen or printer, alter the contents of memory or access storage devices. Upon exit from these routines control must be returned to BASIC and normal BASIC processing must not be disturbed.

The Control Program module must perform all the functions, other than the direct operations of the individual commands. We will look at the Control Program and see how it satisfies these requirements.

1. Intercept 'Wedge-MON' Commands

In order to do anything about altering the process we must first know something about how BASIC gets its input. To do this we look at the section of ROM code that controls the activity of the BASIC editor when lines are entered from the keyboard. This line input loop code, as you will be able to check using 'Wedge-MON' @D facility, looks like this:

```
A480   JMP   ($0302)   Jump to the address held at $0302,$0303
A483   JSR   $A560     Jump to LOAD BUFFER subroutine
A486   STX   $7A       }Initialize CHRGET pointer to $01FF
A486   STY   $7B       }
A48A   JSR   $0073     Jump to CHRGET subroutine
A48D   TAX             Transfer byte from CHRGET to X register
A48E   BEQ   $A480     If byte=zero then back to $A480
. . .
. . . Process Input Line
. . .
```

The jump to the address stored at $0302 and $0303 is a RAM *vector* which normally points back to address $A483. The reason for the indirect jump is to allow the possibility of transferring control to a *user patch* routine by changing the vector address. Without such RAM vectors at strategic points the 64's operating system would be inaccessible to the machine code programmer.

The detection of keypresses is the responsibility of the ROM Key-scan routine which is called by the BASIC interrupt every 1/60th of a second. The characters it detects are placed in a ten-byte character storage buffer (the keyboard buffer). The BASIC ROM subroutine at $A560, which I have called LOAD BUFFER, takes characters from the keyboard buffer and places them in an 80-character buffer, called the BASIC or system INPUT buffer, until it detects a carriage return character, whereupon it puts a zero in the buffer to mark the end-of-line and returns. Both of these buffers are found in Page Two of memory, the keyboard buffer occupying locations $0277 to $0280 and the INPUT buffer $0208 to $0280.

CHRGET is a subroutine which resides in Page Zero RAM after having been copied down from ROM location $E3A2 as part of the

operating system power-up procedure. Its function is to fetch the next character which is not a space character from the input buffer and return with it in the accumulator. The code in Zero-page looks like this:

```
0073  INC $7A
0075  BNE $0079
0077  INC $7B
0079  LDA $0200
007C  CMP #$3A
007E  BCS $008A
0080  CMP #$20
0082  BEQ $0073
0084  SEC
0085  SBC #$30
0087  SEC
0088  SBC #$D0
008A  RTS
```

CHRGET is a very interesting routine in many ways. For a start it is *self-modifying*. Instead of addressing the input buffer indirectly and altering the index value (i.e. using LDA ($7A),Y), CHRGET addresses the buffer absolutely and alters the instruction operand value itself (i.e. LDA $0200..LDA $0209..etc.). This is the reason why it must be housed in RAM. CHRGET ignores space characters ($20) and sets flags according to the value of the byte it loads from the buffer. On returning from CHRGET the Carry flag is clear (BCC succeeds) only if the Accumulator contains an ASCII numeral, thus allowing a direct test for a line number. The Zero flag is set (BEQ succeeds) if the Accumulator contains zero or a colon, indicating an end-of-line.

Looking again at the BASIC line input loop you should see that there are two obvious ways of examining the buffer for 'Wedge-MON' commands before the buffer is processed by the normal BASIC routines. The first option is to change the RAM vector at ($0302) to point to a routine which duplicates the ROM code, but also includes a JSR to a check subroutine to look for 'Wedge-MON' commands before passing the line back to the BASIC interpreter if it is not a monitor command. The second way is to alter the CHRGET routine in RAM by adding a jump to a routine which checks the character fetched from the buffer before returning it.

The generic term for such methods of intercepting the system code through RAM entry points is *wedging*. For 'Wedge-MON' the choice was to use the second option given above – the so-called CHRGET wedge – for the reason that this method is the one most

commonly encountered in commercial software. Much of this has come over to the 64 from the CBM PET machine which shares the CHRGET routine but lacks the vector at $0302. One commercial package using this CHRGET Wedge technique is CBM's DOS Wedge program which adds disc handling commands to the 64's BASIC. There are more than the two ways given above of wedging BASIC, and in the 'Expander' program given later in the book you will see another.

All the code for setting up the CHRGET wedge and the 'kernal' of input-output routines used by all the 'Wedge-MON' functions are contained in the Control Program. For this reason, the Control Program *must* be entered before any of the command routines can work.

Setting up the CHRGET Wedge (Lines 1230–1330)

This is the routine to which you jump when you first SYS 49152. It activates the wedge by putting three bytes coding for JMP WEDGE into the CHRGET routine at $007C. If you disassemble CHRGET with 'Wedge-MON' and compare it with the original given earlier you will see the result.

We return to BASIC via the ROM routine at $AB1E. This is one of several ROM routines exploited by 'Wedge-MON' and they are listed in Appendix 4. This is a print routine which outputs to the screen an ASCII string, terminated by a zero byte. The address of the string is passed to this routine in the Y (high byte) and A (low byte) registers. In this case the string is my credit message, located at CREDIT (see line 5200). Notice that the program uses a JMP to this subroutine and not a JSR, so that on conclusion, instead of returning to 'Wedge-MON', control passes straight back to BASIC.

Checking for Wedge-MON Commands (Lines 1380–1390)

In most programs that employ a CHRGET wedge new commands are prefixed with a special 'wedge character'. This is usually a little-used character, such as ! or @ which can be used to instantly distinguish wedge commands from other input. Our program is no exception and you will notice that all 'Wedge-MON' commands are prefixed by @. Our first job then must be to check CHRGET returns for this character. If this check fails then lines 1410–1430 substitute for the normal CHRGET code which was replaced by our wedge.

Assuming that we have detected an @ character our next problem is that although the CHRGET subroutine is called from many points in the ROM code of the 64 (and by 'Wedge-MON' itself) we are only interested in those calls which come from the BASIC line input loop. To save unnecessary processing lines 1490–1600 check the origin of

the subroutine call to CHRGET. This is achieved by pulling off the stack (PLA) the return address (two bytes) for the JSR. If this is not $A48C then we push the return address back on to the stack using PHA and ignore the @. This check also has the effect of ignoring 'Wedge-MON' commands encountered within a BASIC program.

We have now satisfied the first of our original requirements for 'Wedge-MON'.

2. Transfer Control to Command Routines

Control Section (Lines 1740–2050)

We now assume that the input buffer contains a 'Wedge-MON' command and our first action after replacing the CHRGET return address on the stack is to initialize a Zero-page flag PFLAG to zero and to place a JMP Absolute operand (#$4C) at FNCTN ($02A7). The reason for the latter operation will become clear a little later.

We must now get in the command character following @ from the input buffer. This we do by calling CHRGET with a JSR $0073 instruction. If the Zero flag is set on return from CHRGET then there was no character after @, and we exit via the EXIT1 routine at line 2590. If a character exists the next stage is to compare it with the list of the 14 valid command characters. These are stored in the table COMTAB at lines 5280–5300. The loop at lines 1860–1920 performs the comparisons. If no comparison exists then we exit, again via EXIT1, since the character is not valid. If however the character is a valid command character then it is stored at the address CHAR and the program can pass on to the final part of the control section.

Following the loop at lines 1860–1920 we now have in the X register the index value of the command character in the COMTAB table. We can use this index to reference tables COMAD1 and COMAD2 (at lines 5350 and 5380) to retrieve the addresses of the routines that will execute the command. However, since COMTAB is a BYTE table and COMAD1 and COMAD2 are WORD (i.e. two-byte) tables it is necessary to first multiply the index value by 2, which is done in lines 1930 to 1950, using ASL.

The reason why two addresses are required to effect one command should become clear in the discussion below on linking commands into the control program. The first address retrieved from the table COMAD1 is placed at locations FNCTN+1 and FNCTN+2 ($02A8 and $02A9) and the second, from table COMAD2, is pushed on the stack (high byte first). This done, control is transferred to the address on the stack by means of an RTS instruction. Some readers will find this mysterious. You may well be wondering 'why RTS?' and 'what JSR are we returning from?'. The answer is that we are not executing a normal return from a subroutine but instead are using the RTS

instruction in a rather unorthodox fashion. You will recall that a JSR instruction works by pushing the current contents of the PC (Program Counter) register on to the stack and then jumping to a subroutine. The RTS instruction simply returns control to the *byte following* the address stored at the top of the stack. When using RTS it matters not one whit *how* the address got on to the stack, so RTS may also be used as an absolute JMP to an address on the stack. In our case, the jump is to the command routine address from COMAD2 that was pushed on to the stack in lines 2010–2040. We will meet this technique again in later programs and there are two important points to remember when using it. Firstly, since RTS returns control to the byte following that specified by the address on the stack, the address we push must be the address to which we want to jump *minus* 1, and, secondly, the address is pushed on to the stack *high byte first*.

Before any commands are linked into the control program all the entries in the table COMAD2 simply point to line 2550, which exits straight back to BASIC. The POKEs at the end of each BASIC command loader serve the purpose of writing the command address into its correct position in COMAD2, and they *must also be used by anyone entering the assembler code in order to activate the commands*.

It should be clear from the last two paragraphs that to add new commands to 'Wedge-MON' all that is required once the command routine is coded is to add a new command character to the end of the table COMTAB and a corresponding address (less 1) to the command action table COMTAB2. With further user expansion in mind nine bytes have been left free after COMTAB, allowing up to nine extra user commands to be added.

We have now satisfied the second of our requirements.

3. Individual Command-handling Routines

The 'Wedge-MON' commands fall into three distinct groups, as follows.

Group (a): Memory display commands

The commands @D, @M and @I display the contents of memory in three different ways, but share a common algorithm:

1. Get in the command parameters (start address – end address or just start address)
2. If no end parameter is given then set a flag
3. Print a line from (current address)
4. Cursor back to column zero
5. Compare current address to end address
6. If the flag is not set, and current address not equal to end address and 'stop' key not pressed then go to 3

7. Wait for a key to be pressed
8. If key is <cursor down> then go to 3
9. Exit to BASIC

Since the only difference between @D, @M and @I in terms of our algorithm lies in the nature of the operations needed to execute step 3. it makes sense to have all three commands controlled by a common loop, with a variable element in step 3. to come with each individual print function.

Group (b): Memory write commands
The commands @A, @W and @C are related in function in a similar way to those of group (a). @A is the inverse of @D, @W the opposite of @M, and @C is the reverse operation to @I.

1. Print <cursor up>
2. Get in address
3. Get in n data bytes after address
4. Store data bytes at (address)
5. Perform the 'opposite' procedure over the line just inputted (e.g. if @W then @M over input line)
6. Print <return>
7. Print @
8. Print 'command character'
9. Print 2 spaces
10. Print address plus n
11. Print 3 spaces
12. Exit to BASIC

This group of commands, despite differences in steps 1., 2. and 3. all share steps 4. to 12. in the above algorithm.

Group (c): Non-related commands
The remaining eight 'Wedge-MON' commands have rather less in common with each other than those of the first two groups and therefore must be handled by entirely separate routines.

We will now look at the Control Program code for these command routines.

Control Loop for Group (a) commands @D, @M and @I
(lines 2070–2600)

It is to MAINLP at line 2280 that the above three commands are directed from the control section.

The first task is to 'get in' or retrieve the parameters for our command. These will be located in the input buffer following the

command character. 'Wedge-MON' uses four routines to retrieve parameters from the buffer:

1. CHRGET
2. HEXGET at lines 4060–4370
3. PARAMS at lines 3120–3330
4. GETADR at lines 4780–4870

The CHRGET routine, as we know, gets the next character from the buffer in A.

HEXGET uses CHRGET to get in a hex byte, i.e. two ASCII digits, from the buffer and converts it to a binary number in A.

PARAMS uses HEXGET to get either (a) a two-byte (four digits) hex number or (b) two two-byte numbers, separated by a hyphen, into the Zero-page locations CURR, CURR+1, END and END+1.

GETADR is a specialized form of PARAMS for the group (b) commands. It outputs a #$91 (cursor up) character, and then gets a single two-byte address into CURR, CURR+1.

Hex ASCII to binary conversion is quite simple. The algorithm for HEXGET demonstrates this:

1. Set COUNTER=1
2. Set RESULT=0
3. Get an ASCII character from the buffer in the A register
4. If the ASCII character is numeric then go to 8.
5. If ASCII A – F then go to 7.
6. Error – exit
7. Subtract 7 from Accumulator
8. Subtract 48 from Accumulator
9. Multiply RESULT by 16
10. Add Accumulator to RESULT
11. Subtract 1 from COUNTER
12. If COUNTER greater than 0 then back to 3. to get the low nybble
13. Both nybbles input – exit

Points to note:

(a) Lines 4210–4240 multiply the RESULT variable of our algorithm (TEMP1 in the code) by 16 using four ASL instructions.

(b) In line 4120 numeric ASCII bytes are easily spotted by checking the Carry flag. Remember that CHRGET returns Carry clear for ASCII number codes, and since the Carry is clear the SBC #$2F instruction at line 4190 becomes effectively a SEC/SBC #$30 instruction sequence.

(c) HEXGET errors are flagged by returning with the Carry flag set, and with the erroneous byte stored in the Accumulator.

(d) Finally, the PHA/PLA sequence at lines 4350–4360 is to condition the Zero flag on the basis of the Accumulator contents.

The PARAMS and GETADR routines, like HEXGET, also return with the Carry flag set if they encounter erroneous data in the buffer, with the rogue byte contained in the A register. PARAMS also conditions the Z flag depending on the nature of the input data, such that the Z flag is set if a single parameter was input and clear if the input was two parameters separated by a hyphen. Looking back to the main program you can see how PARAMS is used in lines 2280–2310. Notice that we exit if PARAMS returns an error and that a flag (PFLAG) is set if the input was 'one parameter', as opposed to two separated by a hyphen.

'Indirect JSR' at FNCTN (Line 2330)

Line 2330 contains the variable element of the group (a) commands control loop. The destination of this JSR will depend on which command is presently using the loop, @D, @M or @I. The destination routine outputs a line from memory in the format specified by the command being executed.

In 6510 assembler there is no indirect JSR instruction of a form similar to JMP (Indirect) but it is possible to simulate it by using a method which employs 'self-modifying code', a term you may remember from our discussion of the CHRGET routine. All that is required is to place an absolute JMP op code (#$4C) at a convenient place in memory. Next, we place the address to which we would JSR in the two bytes following the op code. Now simply, JSR to the JMP instruction and we have our 'indirect JSR'. You will recall that in lines 1770–1780 we placed a #$4C at FNCTN (address $02A7) and that in the Control section we used the COMTAB table index derived from our command character to get an address from COMAD1 which we placed at FNCTN+1 and FNCTN+2. These Page Two locations were chosen because they are not used by BASIC or the operating system and are therefore not liable to corruption.

A JSR to HOMCSR at lines 3730–3770 returns the cursor to column zero (the far left side of the screen). It uses the operating system Kernal routine PLOT which can be used to position the cursor anywhere on the screen.

The JSR to COMPAR at lines 4570–4710 decides whether, having outputted a line, we then immediately get the next line (return was with Carry clear) or just sit and wait for a key to be pressed (return Carry set). The conditions for passing directly to the next line are:

(a) that the <STOP> key is not pressed. This is ascertained by checking bit 7 of the operating system Zero-page flag $91.

(b) that a memory 'range' was requested and not just a single address. We test the 'one parameter' flag PFLAG.

(c) that the current address does not equal the end parameter.

Lines 2400–2480 wait for a key to be pressed. Operating system flags $C6 and $C5 indicate respectively the number of characters currently in the keyboard buffer and the code of the current key being pressed. We wait in a loop until $C6 is greater than zero, and then check which key is pressed. If it is anything other than <cursor down> then the program exits to BASIC.

Line 2500 outputs a 'carriage return' character to the screen. JSR $FFD2 accesses the Kernal character output routine CHROUT. This routine outputs to the screen the ASCII byte contained in the A register and returns without affecting any registers. This is the routine used by 'Wedge-MON' for all screen output.

A JSR to NXTBYT at lines 4430–4480 increments the current address pointer at CURR (the start address) and loads the next byte from the new current address. At line 2530 we loop back to fetch the next line from memory.

'Wedge-MON' Common Exit Points (Lines 2550–2600)

There are two common exit points for all 'Wedge-MON' commands. Return to BASIC is effected via EXIT1, which goes through CHRGET at $0080 having first loaded A with zero. This has the effect of making BASIC think it has reached the end of the input buffer so that it ignores any further debris that might otherwise generate syntax errors and so spoil our screen output. For the sake of tidy screen output 'Wedge-MON' group (a) and (b) commands must not be allowed to generate error messages. Exit via EXIT0 clears the keyboard buffer pointer and dis-enables 'quote mode' before returning to BASIC. This is required by some of the later 'Wedge-MON' commands.

Group (b) Commands Common Exit (Lines 2790–3010)

This code represents steps 4 to 14 in the general algorithm given for the group (b) memory write commands @A, @W and @C. The first step is to perform the corresponding 'reverse' procedure over the line that has just been input. For instance, if the command is @A then the line just assembled to memory is disassembled again over what you have just typed. This is why each of the group (b) commands starts by printing 'cursor up' (in GETADR). This might seem like a futile procedure, and indeed in most cases the new line will be exactly the same as the old. However, it serves as a check that the line has been written to memory correctly. You can see it in action by trying to over-write ROM memory. The corresponding 'reverse' procedure is accessed via the 'indirect' subroutine set up at FNCTN by the main

control section of the program. Note that @A, @W and @C have the same FNCTN addresses, respectively, as @D, @M and @I.

The next part of the COMEXT code takes us down a line, in preparation for the next input, by printing a 'carriage return'. The current address pointer in CURR is incremented to point to the next available address by NXTBYT and this, preceded by @ plus the current command character and indented by two character positions instead of the normal one. This indentation of the address is the sign that all has gone well with the previous instruction.

Lines 2930–2980 may seem mysterious. These came about when I found that upon exit to BASIC the editor stubbornly refused to accept the line left under the cursor unless I first cursored up and then back to it. This was solved by the quick fix of putting these cursor control key characters in the keyboard buffer before exiting. The effect of this is the same as having physically pressed the keys upon returning to BASIC. The keyboard buffer sits between $0277 and $0280, and the ASCII key codes are placed at the first two positions in the buffer. To inform the operating system that they are there the value 2 is stored in Zero-page location $C6, which records the number of keystrokes in the buffer. This technique is useful in a number of situations. One obvious application is programming the function keys to dump predefined BASIC commands into the buffer. The only drawback is that the buffer can hold a maximum of ten characters.

OK Exit (Lines 5070–5100)

This used by most of the group (C) commands on completion of their function. Before jumping back to BASIC via EXIT0 the message OK is printed by the ROM routine $AB1E. We have already seen this routine printing the CREDIT message in lines 1280–1300. You will recall it prints out a string of ASCII characters (terminated by a zero byte) of which the location in memory is held in A (*low* byte) and V (*high* byte). In this case we print the message OK from ROM location $A364.

Command Output Routines

We have already seen the input routines HEXGET, PARAMS and GETADR which are used by the command routines to get in data from the input buffer. Also required by the commands are a set of output routines to print data to screen and printer:

1. $FFD2
2. OUTPUT at lines 3840–3970
3. ADROUT at lines 3540–3650
4. SPACE at lines 3420–3470

We have already met with the ROM Kernal routine CHROUT located at

$FFD2 for printing the A register value as an ASCII character. OUTPUT outputs A as a hex number, i.e. two hex digits. Binary to hex conversion is carried out by the following algorithm:

1. Store A on the stack
2. Divide A by 16
3. Jump to subroutine 6.
4. Recover A from the stack
5. AND A with 15
6. If A value less than 10 then 8.
7. Add 7 to A
8. Add 48 to A
9. Output A as an ASCII byte
10. Return

Notice that, like the reverse routine HEXGET, it first outputs the high nybble and then the low nybble, and also that the ADC #$06 in line 2820 is effectively an ADC #$07, since the Carry was set before the addition instruction was executed.

The ADROUT routine uses OUTPUT to output the current address CURR in two bytes, high byte first, preceded by @ and the command character (stored in CHAR) plus a 'space' character. Finally it prints three more spaces and then returns via the SPACES routine (lines 3420–3470).

The space print routine has three entry points: SPACE prints just one space and returns; SPACES prints multiple spaces and ANYCHR prints multiple characters corresponding to whichever ASCII code is held in the A register. The number of spaces or characters printed is specified by the X register contents.

@X Command: Quit 'Wedge-MON' (Lines 4940–5030)

This is the only command routine that is actually held within the body of the control program, and its operation is very simple. Lines 4940–4950 check for data in the buffer after the command character. If there is any then the command aborts. This prevents 'Wedge-MON' being disabled as a result of a typing mistake. In lines 4960–5000 the three CHRGET bytes displaced by our wedge are replaced from the CAT table (line 5150), so that the CHRGET routine reverts to its normal form. Exit is via the CNCLUD routine, which simply prints OK as a prompt.

We have examined in detail the Control Program, and will now give the code for, and then look at, the seven individual Command Files.

Command File 1: The @D (Group (a)) and @A (Group (b)) Commands

Assembler Listing: Command File 1

```
1030 *=$C300
1040 ;----------
1050 CURR    =$4D;*
1060 WKSPC   =$4F;*
1070 MNMON   =$FD
1080 ADMOD   =$FE
1090 ;----------
1100 MBUFF   =$02B4
1110 ;----------
1120 EXIT0   =$C0A0
1130 COMEXT  =$C0AB
1140 SPACE   =$C108
1150 SPACES  =$C10A
1160 ADROUT  =$C113
1170 OUTPUT  =$C13A
1180 HEXGET  =$C150
1190 NXTBYT  =$C17F
1200 GETADR  =$C1A0
1210 ;----------
1220 ;--DISASSEMBLE MEMORY--
1230 ;PRINT CURRENT ADDRESS IN HEX/ GET
1240 ;BYTE AT (ADDRESS) IN Y AS INDEX
1250 ;INTO MNEMONIC CODE TABLE ARY2 TO
1260 ;OBTAIN INDEX INTO MNEMONIC TABLE
1270 ;& PRINT MNEMONIC/USE Y TO INDEX
1280 ;ADDRESSING MODE TABLE ARY3 TO
1290 ;OBTAIN INDEX INTO ADDRESS MODE
1300 ;ACTION TABLE TABL/JSR ADRMOD-
1310 ;PUSH SELECTED ADDRESS FROM TABL
1320 ;ON STACK AND RTS TO IT TO HANDLE
1330 ;ADDRESSING MODE/PRINT $0B SPACES
1340 ;----------
1350 DSASBL JSR ADROUT
1360 ;----------
1370        LDY #0
1380        LDA (CURR),Y
1390        TAY
1400        PHA
1410        LDX ARY2,Y
1420        LDY #$02
```

```
1430 LOOP   LDA ARY1,X
1440        JSR $FFD2
1450        INX
1460        DEY
1470        BPL LOOP
1480 ;----------
1490        JSR SPACE
1500 ;----------
1510        PLA
1520        TAY
1530        LDX ARY3,Y
1540        BMI TIDY
1550        JSR ADRMOD
1560 ;----------
1570 TIDY   LDX #$0B
1580        JMP SPACES
1590 ;----------
1600 ADRMOD TXA
1610        ASL A
1620        TAY
1630        LDA TABL+1,Y
1640        PHA
1650        LDA TABL,Y
1660        PHA
1670        RTS
1680 ;----------
1690 ;--OUTPUT OPERANDS--
1700 ;THESE ARE THE ROUTINES WHOSE
1710 ;ADDRESSES ARE HELD IN THE
1720 ;ADDRESSING MODE ACTION TABLE TABL
1730 ;EACH GETS IN BYTES AFTER AN
1740 ;OP-CODE AND PRESENTS THEM
1750 ;ACCORDING TO THE RELEVENT
1760 ;ADDRESSING MODE
1770 ;----------
1780 IMM    LDA #$23      ;'#
1790        JSR $FFD2
1800 ZPG    LDA #$24      ;'$
1810        JSR $FFD2
1820        JSR NXTBYT
1830        JMP OUTPUT
1840 ;----------
1850 ABS    LDA #$24      ;'$
1860        JSR $FFD2
```

```
1870          JSR NXTBYT
1880          PHA
1890          JSR NXTBYT
1900          JSR OUTPUT
1910          PLA
1920          JMP OUTPUT
1930 ;----------
1940 ABX      JSR ABS
1950 ZPGXIN   LDA #$2C      ;',
1960          JSR $FFD2
1970          LDA #$58      ;'X
1980          JMP $FFD2
1990 ;----------
2000 ABY      JSR ABS
2010 ZPGYIN   LDA #$2C      ;',
2020          JSR $FFD2
2030          LDA #$59      ;'Y
2040          JMP $FFD2
2050 ;----------
2060 ZPX      JSR ZPG
2070          JMP ZPGXIN
2080 ;----------
2090 ZPY      JSR ZPG
2100          JMP ZPGYIN
2110 ;----------
2120 IND      LDA #$28      ;'(
2130          JSR $FFD2
2140          JSR ABS
2150          LDA #$29      ;')
2160          JMP $FFD2
2170 ;----------
2180 ENY      LDA #$28      ;'(
2190          JSR $FFD2
2200          JSR ZPG
2210          LDA #$29      ;')
2220          JSR $FFD2
2230          JMP ZPGYIN
2240 ;----------
2250 ENX      LDA #$28      ;'(
2260          JSR $FFD2
2270          JSR ZPX
2280          LDA #$29      ;')
2290          JMP $FFD2
2300 ;----------
```

```
2310 ACC     LDA #$41      ;'A
2320         JMP $FFD2
2330 ;----------
2340 ;CALCULATE BRANCH ADDRESS FOR
2350 ;A RELATIVE INSTRUCTION AND
2360 ;PRINT IT
2370 ;----------
2380 REL     LDA #$24      ;'$
2390         JSR $FFD2
2400 ;----------
2410         JSR NXTBYT
2420         BMI NEGAT
2430 ;----------
2440         CLC
2450         ADC #$01
2460         ADC CURR
2470         PHA
2480         LDA #0
2490         ADC CURR+1
2500         JMP EXITE
2510 ;----------
2520 NEGAT   EOR #$FF
2530         STA $4B
2540         SEC
2550         LDA CURR
2560         SBC $4B
2570         PHA
2580         LDA CURR+1
2590         SBC #0
2600 ;----------
2610 EXITE   JSR OUTPUT
2620         PLA
2630         JMP OUTPUT
2640 ;----------
2650 ;--ASSEMBLE ROUTINE--
2660 ;PRINT [CURSOR UP]/GET IN HEX
2670 ;ADDRESS FROM BUFFER-EXIT IF
2680 ;ILLEGAL/GET 3 CHAR OP-CODE
2690 ;MNEMONIC-EXIT IF NOT ALPHA/STORE
2700 ;AT MBUFF/CHECK WITH MNEMONIC
2710 ;ASCII TABLE ARY1-EXIT IF NO MATCH
2720 ;- STORE INDEX AS MNEMONIC CODE
2730 ;/CHECK INDEX AGAINST RELTAB TABLE
2740 ;TO DETERMINE IF INSTRUCTION USES
```

```
2750 ;RELATIVE ADDRESSING-IF SO THEN
2760 ;GET IN BRANCH ADDRESS FROM BUFFER
2770 ;AND CALCULATE BRANCH OFFSET IN A
2780 ;- X HOLDS THE RELATIVE ADDRESSING
2790 ;CODE $0B/IF NOT RELATIVE
2800 ;ADDRESSING THEN GET IN CHARACTERS
2810 ;AFTER THE MNEMONIC USING THESE TO
2820 ;DEDUCE ADDRESSING MODE IN X-
2830 ;HEX OPERAND BYTES ARE EVALUATED
2840 ;AND PUSHED ON STACK TO BE LATER
2850 ;PULLED OFF AND STORED AT THE
2860 ;CURRENT ADDRESS+1 & 2/THE
2870 ;RETRIEVED MNEMONIC AND ADRESSING
2880 ;MODE CODE VALUES ARE SEARCHED FOR
2890 ;IN THE CORRESPONDING TABLES ARY2
2900 ;AND ARY3 THE INDEX TO WHICH
2910 ;REPRESENTS THE OP-CODE VALUE
2920 ;WHICH IS STORED AT THE CURRENT
2930 ;ADDRESS
2940 ;----------
2950 ASSEMB JSR GETADR
2960        BCS KLUD1
2970 ;----------
2980        LDX #0
2990 ROU    JSR $0073
3000        BCC KLUD1
3010        CMP #$5B     ;'Z
3020        BCS KLUD1
3030        CMP #$41
3040        BCC KLUD1    ;'A
3050        STA MBUFF,X
3060        INX
3070        CPX #$03
3080        BNE ROU
3090 ;----------
3100 ;GET IN MNEMONIC CODE
3110 ;----------
3120        LDY #0
3130 MLOOP1 STY MNMON
3140        LDX #0
3150 MLOOP2 LDA ARY1,Y
3160        CMP MBUFF,X
3170        BEQ EQUALS
3180        LDY MNMON
```

```
3190          INY
3200          INY
3210          INY
3220          CPY #$A8
3230          BCC MLOOP1
3240 KLUD1    JMP X3
3250 ;----------
3260 EQUALS   INY
3270          INX
3280          CPX #$03
3290          BCC MLOOP2
3300 ;----------
3310 ;CHECK IF RELATIVE ADDRESSING
3320 ;----------
3330 RELCHK   LDX #$07
3340          LDA MNMON
3350 FRI      CMP RELTAB,X
3360          BEQ ISREL
3370          DEX
3380          BPL FRI
3390          BMI NOTREL
3400 ;----------
3410 ISREL    JSR $0073
3420          CMP #$24        ;'$
3430          BNE KLUD2
3440 ;----------
3450          JSR HEXGET
3460          BCS KLUD2
3470          STA WKSPC+1
3480          JSR HEXGET
3490          BCS KLUD2
3500          SEC
3510          SBC #$02
3520          STA WKSPC
3530          BCS SKIP
3540          DEC WKSPC+1
3550 ;----------
3560          SEC
3570 SKIP     SBC CURR
3580          TAY
3590          LDA WKSPC+1
3600          SBC CURR+1
3610          BMI MINT
3620 ;----------
```

```
3630          BNE KLUD2
3640          CPY #$80
3650          BCS KLUD2
3660          BCC PAN
3670 ;----------
3680 MINT     CMP #$FF
3690          BNE KLUD2
3700          CPY #$80
3710          BCC KLUD2
3720 ;----------
3730 PAN      TYA
3740          LDY #$01
3750          LDX #$0B
3760          JMP COMSL
3770 ;----------
3780 NOADMO   LDX #$FF
3790          JMP REGRUP
3800 ;----------
3810 NOTREL   LDY #$01
3820          JSR $0073
3830          BEQ NOADMO
3840          CMP #$41    ;'A
3850          BEQ ACCUM
3860          CMP #$23    ;'#
3870          BEQ IMMED
3880          CMP #$24    ;'$
3890          BEQ ABSOLU
3900          CMP #$28    ;'(
3910          BEQ BRAC
3920 KLUD2    JMP X3
3930 ;----------
3940 ACCUM    LDX #$0A
3950          JMP REGRUP
3960 ;----------
3970 IMMED    JSR $0073
3980          CMP #$24    ;'$
3990          BNE KLUD2
4000          JSR HEXGET
4010          BCS KLUD2
4020          LDX #0
4030          JMP COMSL
4040 ;----------
4050 ABSOLU   JSR HEXGET
4060          BCS KLUD2
```

```
4070          PHA
4080          JSR HEXGET
4090          BCC TEBBIT
4100          BEQ ZPAGE
4110          CMP #$2C    ;',
4120          BEQ ZPICOM
4130          JMP X2
4140 ;----------
4150 TEBBIT   PHA
4160          JSR $0073
4170          BEQ ABSO
4180          CMP #$2C    ;',
4190          BEQ ABICOM
4200          JMP X1
4210 ;----------
4220 ABSO     LDX #$01
4230          JMP COMX
4240 ;----------
4250 ABICOM   JSR $0073
4260          CMP #$58    ;'X
4270          BEQ ABSOX
4280          CMP #$59    ;'Y
4290          BEQ ABSOY
4300          JMP X1
4310 ;----------
4320 ABSOX    LDX #$03
4330          JMP COMX
4340 ;----------
4350 ABSOY    LDX #$04
4360          JMP COMX
4370 ;----------
4380 ZPAGE    LDX #$02
4390          JMP COMS
4400 ;----------
4410 ZPICOM   JSR $0073
4420          CMP #$58    ;'X
4430          BEQ ZPAGX
4440          CMP #$59    ;'Y
4450          BEQ ZPAGY
4460 KLUD3    JMP X2
4470 ;----------
4480 ZPAGX    LDX #$05
4490          JMP COMS
4500 ;----------
```

```
4510 ZPAGY  LDX #$06
4520        JMP COMS
4530 ;----------
4540 BRAC   JSR $0073
4550        CMP #$24    ;'$
4560        BNE KLUD4
4570        JSR HEXGET
4580        BCS KLUD4
4590        PHA
4600        JSR HEXGET
4610        BCC INDRCT
4620        CMP #$29    ;')
4630        BEQ INDIRY
4640        CMP #$2C    ;','
4650        BNE KLUD3
4660 ;----------
4670 INDIRX LDX #$09
4680        JMP COMS
4690 ;----------
4700 INDIRY LDX #$08
4710        JMP COMS
4720 ;----------
4730 INDRCT LDX #$07
4740        JMP COMXL
4750 ;----------
4760 ;PULL OPERAND VALUES AND STORE
4770 ;AFTER THE CURRENT ADDRESS
4780 ;----------
4790 COMX   PLA
4800 COMXL  STA (CURR),Y
4810        INY
4820 COMS   PLA
4830 COMSL  STA (CURR),Y
4840 ;----------
4850 ;CHECK ARY2 AND ARY3 FOR RETRIEVED
4860 ;MNEMONIC AND ADDRESSING MODE CODE
4870 ;VALUES/INDEX=THE OP-CODE
4880 ;----------
4890 REGRUP STX ADMOD
4900        LDX #0
4910 TOR    LDA ARY2,X
4920        CMP MNMON
4930        BNE PYM
4940        LDA ARY3,X
```

```
4950          CMP ADMOD
4960          BEQ JAY
4970 PYM      INX
4980          BNE TOR
4990 KLUD4    JMP X3
5000 ;----------
5010 ;STORE OP-CODE AT CURRENT ADDRESS
5020 ;----------
5030 JAY      LDY #0
5040          TXA
5050          STA (CURR),Y
5060 ;----------
5070 ;EXIT VIA COMMON EXIT IN CONTROL
5080 ;SECTION/
5090 ;----------
5100          JMP COMEXT
5110 ;----------
5120 ;ERROR EXIT - PLA'S ACCOMODATE
5130 ;ERRORS ENCOUNTERED WHILE OPERANDS
5140 ;ON STACK/PRINT 'RET'/EXIT
5150 ;----------------
5160 X1       PLA
5170 X2       PLA
5180 X3       LDA #$0D        ;'RET
5190          JSR $FFD2
5200          JMP EXIT0
5210 ;----------------
5220 ;ADDRESSING MODE ACTION TABLE FOR
5230 ;DISASSEMBLER
5240 ;----------------
5250 TABL     .WORD IMM-1,ABS-1,ZPG-1
5260          .WORD ABX-1,ABY-1,ZPX-1
5270          .WORD ZPY-1,IND-1,ENY-1
5280          .WORD ENX-1,ACC-1,REL-1
5290 ;----------------
5300 ;TABLE OF ENTRIES IN MNEMONIC
5310 ;TABLE THAT USE RELATIVE
5320 ;ADDRESSING-USED BY ASSEMBLER
5330 ;----------------
5340 RELTAB .BYTE $09,$0C,$0F,$15
5350          .BYTE $18,$1B,$21,$24
5360 ;--------------------
5370 ;MNEMONIC TABLE- CONTAINS ASCII
5380 ;INSTRUCTION MNEMONICS-USED BY
```

```
5390 ;ASSEMBLER AND DISASSEMBLER
5400 ;--------------------
5410 ARY1    .ASC "ADCANDASLBCCBCSBEQ"
5420         .ASC "BITBMIBNEBPLBRKBVC"
5430         .ASC "BVSCLCCLDCLICLVCMP"
5440         .ASC "CPXCPYDECDEXDEYEOR"
5450         .ASC "INCINXINYJMPJSRLDA"
5460         .ASC "LDXLDYLSRNOPORAPHA"
5470         .ASC "PHPPLAPLPROLRORRTI"
5480         .ASC "RTSSBCSECSEDSEISTA"
5490         .ASC "STXSTYTAXTAYTSXTXA"
5500         .ASC "TXSTYA???"
5510 ;-------------------------------
5520 ;256 BYTE TABLE OF MNEMONIC CODES
5530 ;-------------------------------
5540 ARY2 .BYTE$1E,$66,$A8,$A8,$A8,$66
5550       .BYTE$06,$A8,$6C,$66,$06,$A8
5560       .BYTE$A8,$66,$06,$A8,$1B,$66
5570       .BYTE$A8,$A8,$A8,$66,$06,$A8
5580       .BYTE$27,$66,$A8,$A8,$A8,$66
5590       .BYTE$06,$A8,$54,$03,$A8,$A8
5600       .BYTE$12,$03,$75,$A8,$72,$03
5610       .BYTE$75,$A8,$12,$03,$75,$A8
5620       .BYTE$15,$03,$A8,$A8,$A8,$03
5630       .BYTE$75,$A8,$84,$03,$2D,$A8
5640       .BYTE$A8,$03,$75,$A8,$7B,$45
5650       .BYTE$A8,$A8,$A8,$45,$60,$A8
5660       .BYTE$69,$45,$60,$A8,$51,$45
5670       .BYTE$60,$A8,$21,$45,$A8,$A8
5680       .BYTE$A8,$45,$60,$A8,$2D,$45
5690       .BYTE$A8,$A8,$A8,$45,$60,$A8
5700       .BYTE$7E,$00,$A8,$A8,$A8,$00
5710       .BYTE$78,$A8,$6F,$00,$78,$A8
5720       .BYTE$51,$00,$78,$A8,$24,$00
5730       .BYTE$A8,$A8,$A8,$00,$78,$A8
5740       .BYTE$8A,$00,$A8,$A8,$A8,$00
5750       .BYTE$78,$A8,$A8,$8D,$A8,$A8
5760       .BYTE$93,$8D,$90,$A8,$42,$A8
5770       .BYTE$9F,$A8,$93,$8D,$90,$A8
5780       .BYTE$09,$8D,$A8,$A8,$93,$8D
5790       .BYTE$90,$A8,$A5,$8D,$A2,$A8
5800       .BYTE$A8,$8D,$A8,$A8,$5D,$57
5810       .BYTE$5A,$A8,$5D,$57,$5A,$A8
5820       .BYTE$99,$57,$96,$A8,$5D,$57
```

```
5830        .BYTE$5A,$A8,$0C,$57,$A8,$A8
5840        .BYTE$5D,$57,$5A,$A8,$30,$57
5850        .BYTE$9C,$A8,$5D,$57,$5A,$A8
5860        .BYTE$39,$33,$A8,$A8,$39,$33
5870        .BYTE$3C,$A8,$4E,$33,$3F,$A8
5880        .BYTE$39,$33,$3C,$A8,$18,$33
5890        .BYTE$A8,$A8,$A8,$33,$3C,$A8
5900        .BYTE$2A,$33,$A8,$A8,$A8,$33
5910        .BYTE$3C,$A8,$36,$81,$A8,$A8
5920        .BYTE$36,$81,$48,$A8,$4B,$81
5930        .BYTE$63,$A8,$36,$81,$48,$A8
5940        .BYTE$0F,$81,$A8,$A8,$A8,$81
5950        .BYTE$48,$A8,$87,$81,$A8,$A8
5960        .BYTE$A8,$81,$48,$A8
5970 ;-------------------------------
5980 ;256 BYTE TABLE OF ADDRESSING
5990 ;CODES
6000 ;-------------------------------
6010 ARY3   .BYTE$FF,$09,$FF,$FF,$FF,$02
6020        .BYTE$02,$FF,$FF,$00,$0A,$FF
6030        .BYTE$FF,$01,$01,$FF,$0B,$08
6040        .BYTE$FF,$FF,$FF,$05,$05,$FF
6050        .BYTE$FF,$04,$FF,$FF,$FF,$03
6060        .BYTE$03,$FF,$01,$09,$FF,$FF
6070        .BYTE$02,$02,$02,$FF,$FF,$00
6080        .BYTE$0A,$FF,$01,$01,$01,$FF
6090        .BYTE$0B,$08,$FF,$FF,$FF,$05
6100        .BYTE$05,$FF,$FF,$04,$FF,$FF
6110        .BYTE$FF,$03,$03,$FF,$FF,$09
6120        .BYTE$FF,$FF,$FF,$02,$02,$FF
6130        .BYTE$FF,$00,$0A,$FF,$01,$01
6140        .BYTE$01,$FF,$0B,$08,$FF,$FF
6150        .BYTE$FF,$05,$05,$FF,$FF,$04
6160        .BYTE$FF,$FF,$FF,$03,$03,$FF
6170        .BYTE$FF,$09,$FF,$FF,$FF,$02
6180        .BYTE$02,$FF,$FF,$00,$0A,$FF
6190        .BYTE$07,$01,$01,$FF,$0B,$08
6200        .BYTE$FF,$FF,$FF,$05,$05,$FF
6210        .BYTE$FF,$04,$FF,$FF,$FF,$03
6220        .BYTE$03,$FF,$FF,$09,$FF,$FF
6230        .BYTE$02,$02,$02,$FF,$FF,$FF
6240        .BYTE$FF,$FF,$01,$01,$01,$FF
6250        .BYTE$0B,$08,$FF,$FF,$05,$05
6260        .BYTE$06,$FF,$FF,$04,$FF,$FF
```

```
6270        .BYTE$FF,$03,$FF,$FF,$00,$09
6280        .BYTE$00,$FF,$02,$02,$02,$FF
6290        .BYTE$FF,$00,$FF,$FF,$01,$01
6300        .BYTE$01,$FF,$0B,$08,$FF,$FF
6310        .BYTE$05,$05,$06,$FF,$FF,$04
6320        .BYTE$FF,$FF,$03,$03,$04,$FF
6330        .BYTE$00,$09,$FF,$FF,$02,$02
6340        .BYTE$02,$FF,$FF,$00,$FF,$FF
6350        .BYTE$01,$01,$01,$FF,$0B,$08
6360        .BYTE$FF,$FF,$FF,$05,$05,$FF
6370        .BYTE$FF,$04,$FF,$FF,$FF,$03
6380        .BYTE$03,$FF,$00,$09,$FF,$FF
6390        .BYTE$02,$02,$02,$FF,$FF,$00
6400        .BYTE$FF,$FF,$01,$01,$01,$FF
6410        .BYTE$0B,$08,$FF,$FF,$FF,$05
6420        .BYTE$05,$FF,$FF,$04,$FF,$FF
6430        .BYTE$FF,$03,$03,$FF
6440 ;---------------------
```

BASIC Loader: Command File 1

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 49919+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA32,19,193,160,0,177,77,168
112 DATA72,190,23,198,160,2,189,108
113 DATA197,32,210,255,232,136,16,246
114 DATA32,8,193,104,168,190,23,199
115 DATA48,3,32,42,195,162,11,76
116 DATA10,193,138,10,168,185,77,197
117 DATA72,185,76,197,72,96,169,35
118 DATA32,210,255,169,36,32,210,255
119 DATA-7657
120 REM --- BLOCK 2
121 DATA32,127,193,76,58,193,169,36
122 DATA32,210,255,32,127,193,72,32
123 DATA127,193,32,58,193,104,76,58
```

```
124 DATA193,32,70,195,169,44,32,210
125 DATA255,169,88,76,210,255,32,70
126 DATA195,169,44,32,210,255,169,89
127 DATA76,210,255,32,59,195,76,92
128 DATA195,32,59,195,76,105,195,169
129 DATA-7962
130 REM --- BLOCK 3
131 DATA40,32,210,255,32,70,195,169
132 DATA41,76,210,255,169,40,32,210
133 DATA255,32,59,195,169,41,32,210
134 DATA255,76,105,195,169,40,32,210
135 DATA255,32,115,195,169,41,76,210
136 DATA255,169,65,76,210,255,169,36
137 DATA32,210,255,32,127,193,48,13
138 DATA24,105,1,101,77,72,169,0
139 DATA-7898
140 REM --- BLOCK 4
141 DATA101,78,76,211,195,73,255,133
142 DATA75,56,165,77,229,75,72,165
143 DATA78,233,0,32,58,193,104,76
144 DATA58,193,32,160,193,176,46,162
145 DATA0,32,115,0,144,39,201,91
146 DATA176,35,201,65,144,31,157,180
147 DATA2,232,224,3,208,235,160,0
148 DATA132,253,162,0,185,108,197,221
149 DATA-7763
150 REM --- BLOCK 5
151 DATA180,2,240,12,164,253,200,200
152 DATA200,192,168,144,235,76,68,197
153 DATA200,232,224,3,144,230,162,7
154 DATA165,253,221,100,197,240,5,202
155 DATA16,248,48,67,32,115,0,201
156 DATA36,208,83,32,80,193,176,78
157 DATA133,80,32,80,193,176,71,56
158 DATA233,2,133,79,176,3,198,80
159 DATA-8454
160 REM --- BLOCK 6
161 DATA56,229,77,168,165,80,229,78
162 DATA48,8,208,50,192,128,176,46
163 DATA144,8,201,255,208,40,192,128
164 DATA144,36,152,160,1,162,11,76
165 DATA32,197,162,255,76,34,197,160
166 DATA1,32,115,0,240,244,201,65
167 DATA240,15,201,35,240,16,201,36
```

```
168 DATA240,29,201,40,240,116,76,68
169 DATA-7861
170 REM --- BLOCK 7
171 DATA197,162,10,76,34,197,32,115
172 DATA0,201,36,208,241,32,80,193
173 DATA176,236,162,0,76,32,197,32
174 DATA80,193,176,226,72,32,80,193
175 DATA144,9,240,49,201,44,240,50
176 DATA76,67,197,72,32,115,0,240
177 DATA7,201,44,240,8,76,66,197
178 DATA162,1,76,27,197,32,115,0
179 DATA-7002
180 REM --- BLOCK 8
181 DATA201,88,240,7,201,89,240,8
182 DATA76,66,197,162,3,76,27,197
183 DATA162,4,76,27,197,162,2,76
184 DATA31,197,32,115,0,201,88,240
185 DATA7,201,89,240,8,76,67,197
186 DATA162,5,76,31,197,162,6,76
187 DATA31,197,32,115,0,201,36,208
188 DATA62,32,80,193,176,57,72,32
189 DATA-6612
190 REM --- BLOCK 9
191 DATA80,193,144,18,201,41,240,9
192 DATA201,44,208,217,162,9,76,31
193 DATA197,162,8,76,31,197,162,7
194 DATA76,28,197,104,145,77,200,104
195 DATA145,77,134,254,162,0,189,23
196 DATA198,197,253,208,7,189,23,199
197 DATA197,254,240,6,232,208,239,76
198 DATA68,197,160,0,138,145,77,76
199 DATA-8216
200 REM --- BLOCK 10
201 DATA171,192,104,104,169,13,32,210
202 DATA255,76,160,192,53,195,69,195
203 DATA58,195,88,195,101,195,114,195
204 DATA120,195,126,195,139,195,155,195
205 DATA168,195,173,195,9,12,15,21
206 DATA24,27,33,36,65,68,67,65
207 DATA78,68,65,83,76,66,67,67
208 DATA66,67,83,66,69,81,66,73
209 DATA-6965
210 REM --- BLOCK 11
211 DATA84,66,77,73,66,78,69,66
```

```
212 DATA80,76,66,82,75,66,86,67
213 DATA66,86,83,67,76,67,67,76
214 DATA68,67,76,73,67,76,86,67
215 DATA77,80,67,80,88,67,80,89
216 DATA68,69,67,68,69,88,68,69
217 DATA89,69,79,82,73,78,67,73
218 DATA78,88,73,78,89,74,77,80
219 DATA-4786
220 REM --- BLOCK 12
221 DATA74,83,82,76,68,65,76,68
222 DATA88,76,68,89,76,83,82,78
223 DATA79,80,79,82,65,80,72,65
224 DATA80,72,80,80,76,65,80,76
225 DATA80,82,79,76,82,79,82,82
226 DATA84,73,82,84,83,83,66,67
227 DATA83,69,67,83,69,68,83,69
228 DATA73,83,84,65,83,84,88,83
229 DATA-4941
230 REM --- BLOCK 13
231 DATA84,89,84,65,88,84,65,89
232 DATA84,83,88,84,88,65,84,88
233 DATA83,84,89,65,63,63,63,30
234 DATA102,168,168,168,102,6,168,108
235 DATA102,6,168,168,102,6,168,27
236 DATA102,168,168,168,102,6,168,39
237 DATA102,168,168,168,102,6,168,84
238 DATA3,168,168,18,3,117,168,114
239 DATA-6235
240 REM --- BLOCK 14
241 DATA3,117,168,18,3,117,168,21
242 DATA3,168,168,168,3,117,168,132
243 DATA3,45,168,168,3,117,168,123
244 DATA69,168,168,168,69,96,168,105
245 DATA69,96,168,81,69,96,168,33
246 DATA69,168,168,168,69,96,168,45
247 DATA69,168,168,168,69,96,168,126
248 DATA0,168,168,168,0,120,168,111
249 DATA-7014
250 REM --- BLOCK 15
251 DATA0,120,168,81,0,120,168,36
252 DATA0,168,168,168,0,120,168,138
253 DATA0,168,168,168,0,120,168,168
254 DATA141,168,168,147,141,144,168,66
255 DATA168,159,168,147,141,144,168,9
```

```
256 DATA141,168,168,147,141,144,168,165
257 DATA141,162,168,168,141,168,168,93
258 DATA87,90,168,93,87,90,168,153
259 DATA-8217
260 REM --- BLOCK 16
261 DATA87,150,168,93,87,90,168,12
262 DATA87,168,168,93,87,90,168,48
263 DATA87,156,168,93,87,90,168,57
264 DATA51,168,168,57,51,60,168,78
265 DATA51,63,168,57,51,60,168,24
266 DATA51,168,168,168,51,60,168,42
267 DATA51,168,168,168,51,60,168,54
268 DATA129,168,168,54,129,72,168,75
269 DATA-6840
270 REM --- BLOCK 17
271 DATA129,99,168,54,129,72,168,15
272 DATA129,168,168,168,129,72,168,135
273 DATA129,168,168,168,129,72,168,255
274 DATA9,255,255,255,2,2,255,255
275 DATA0,10,255,255,1,1,255,11
276 DATA8,255,255,255,5,5,255,255
277 DATA4,255,255,255,3,3,255,1
278 DATA9,255,255,2,2,2,255,255
279 DATA-8663
280 REM --- BLOCK 18
281 DATA0,10,255,1,1,1,255,11
282 DATA8,255,255,255,5,5,255,255
283 DATA4,255,255,255,3,3,255,255
284 DATA9,255,255,255,2,2,255,255
285 DATA0,10,255,1,1,1,255,11
286 DATA8,255,255,255,5,5,255,255
287 DATA4,255,255,255,3,3,255,255
288 DATA9,255,255,255,2,2,255,255
289 DATA-8800
290 REM --- BLOCK 19
291 DATA0,10,255,7,1,1,255,11
292 DATA8,255,255,255,5,5,255,255
293 DATA4,255,255,255,3,3,255,255
294 DATA9,255,255,2,2,2,255,255
295 DATA255,255,255,1,1,1,255,11
296 DATA8,255,255,5,5,6,255,255
297 DATA4,255,255,255,3,255,255,0
298 DATA9,0,255,2,2,2,255,255
299 DATA-8293
```

```
300 REM --- BLOCK 20
301 DATA0,255,255,1,1,1,255,11
302 DATA8,255,255,5,5,6,255,255
303 DATA4,255,255,3,3,4,255,0
304 DATA9,255,255,2,2,2,255,255
305 DATA0,255,255,1,1,1,255,11
306 DATA8,255,255,255,5,5,255,255
307 DATA4,255,255,255,3,3,255,0
308 DATA9,255,255,2,2,2,255,255
309 DATA-7774
310 REM --- BLOCK 21
311 DATA0,255,255,1,1,1,255,11
312 DATA8,255,255,255,5,5,255,255
313 DATA4,255,255,255,3,3,255
314 DATA-3102
315 DATA END
1000 REM --ENABLE @D
1001 POKE 49695,109:POKE 49696,192
1002 REM --ENABLE @A
1003 POKE 49701,217:POKE 49702,195
```

Disassemble an Instruction for @D (Lines 1350–1670)

Disassembly is the reverse of assembly. The contents of memory are translated into 6510 instructions consisting of an op code mnemonic plus the operand(s), if present, printed out in the correct format according to the addressing mode. Relative branch instructions (BEQ, BPL, etc.) are presented as the op code mnemonic plus the *destination address*.

The 'Wedge-MON' disassembler routine can be described as a 'table-driven' program. To disassemble an instruction it first fetches the op code byte from memory. This is used as an index into two 256-byte tables: ARY2 at lines 5540–5960 and ARY3 at lines 6010–6430. ARY2 is a table of values which serve as an index into a third table, ARY1, at lines 5410–5500. This is an ASCII table containing all the three letter 6510 instruction mnemonics. ARY3 is a table of addressing mode codes. It holds the coded addressing modes in this form:

$00 = Immediate
$01 = Absolute
$02 = Zero-page absolute
$03 = Absolute, indexed by X
$04 = Absolute, indexed by Y
$05 = Zero-page, indexed by X

$06 = Zero-page, indexed by Y
$07 = Indirect
$08 = Indirect, indexed by Y
$09 = Indirect, indexed by X
$0A = Accumulator
$0B = Relative

The idea is that from ARY2 the program finds which mnemonic to print, and from ARY3 both how many operand bytes are to be fetched from memory and in what format they should be output to the screen. For example, if we were to disassemble the two bytes $A5, $10 we would proceed as follows:

1. Load the Y register with the op code ($A5)
2. Load the X register with mnemonic code (given by ARY2,Y)
3. Print the mnemonic given by ARY1,X,ARY1+1,X,ARY1+2,X
4. Load A with the addressing code (equals ARY3,Y)
5. Use the A register value to select correct handling routine
6. Get in the operand byte ($10) and output it in the correct format
7. Return

There is another possible approach when writing a disassembler that does not depend on 256-byte tables. Instead, the op codes can be 'decoded' to reveal the instruction and addressing mode. The problem is that there are many exceptions to the 'rules' for decoding instructions. Some op codes use binary bits 2, 3 and 4 to specify the addressing mode, while the remainder of the byte identifies the instruction. Other op codes are decoded in a different way. In order to keep 'Wedge-MON' as simple as possible opting for the 'table' approach seemed appropriate.

Going now to the source code we must first output the address of the instruction we are going to disassemble. This is handled by the control program routine ADROUT.

Having printed the address of the instruction we can now disassemble it. First we discover the mnemonic by indirectly loading the op code from (CURR). We transfer this to the Y register and get the index value into the mnemonic table from ARY2. Using ARY1 we then print the three-letter mnemonic. Similarly, we use the op code to get the addressing mode from ARY3. If this is negative ($FF), then the instruction takes no operands and so we print 11 spaces and exit at lines 1570–1580. Where there are operand value(s) then we have the problem of directing control to the correct routine to handle fetching the operands from memory and their display.

This is solved by using the addressing mode 'code' to index a table

of addresses for the handling routines. The address of the needed routine is pushed on to the stack and then jumped to by means of an RTS in the same way that we used command characters to select command-handling routines in the main control section.

The addressing-mode handling routines are located at lines 1780–2630. They use the control program routines NXTBYT and OUTPUT, and the $FFD2 routine in ROM, and are are all fairly straightforward, except for the REL routine at lines 2380–2630. As you should be aware by now relative branch instructions transfer control up to 128 bytes forward or back from their current position and are stored in memory as the op code followed by the offset. In common with most other disassemblers 'Wedge-MON' disassembles these in a more useful form as mnemonic plus branch destination address, and REL must calculate and output the latter. The algorithm for the process is as follows:

1. Get offset byte from (current address)
2. If negative offset then go to 5.
3. Add 1, then add current address to give destination address
4. Jump to 7.
5. Obtain the 1's complement of offset byte value
6. Subtract this from current address to give destination
7. Print destination

The 1's complement form is obtained by means of the EOR #$FF instruction in line 2520, the effect of which is to 'flip' all the bits in a byte, each 1 becoming a 0, and vice versa.

Assemble a Line of Code to Memory for @A (Lines 2950–5350)

The ASSEMB routine is the largest single routine in 'Wedge-MON'. It uses the same tables as the disassembler to perform the reverse procedure and convert an ASCII mnemonic plus hex ASCII operand bytes to binary numbers in memory. The algorithm for the operation looks like this:

1. Get in the address the instruction is to be assembled to
2. Get in the three character instruction mnemonic and store it
3. Compare the mnemonic against the ARY1 table to obtain the mnemonic code
4. If the mnemonic code indicates a relative branch instruction then go to 7.
5. Get in succeeding bytes and use them to deduce the addressing mode
6. Jump to 8.
7. Get in destination address and calculate offset byte value

8. Store the offset byte *or* the retrieved operands *after* the instruction address
9. Loop through the mnemonic code table ARY2 and the addressing table ARY3 looking for the calculated values
10. Store the op code (given by the index from step 9.) *at* the instruction address
11. Go to common Group (b) exit routine

The Control Program routine GETADR is used to output a 'cursor up' and get the instruction address from the input buffer into locations CURR,CURR+1.

Next the three mnemonic characters are input at lines 2980–3080. Non-alphabetic characters cause an error exit. The mnemonic is then stored in a buffer MBUFF. Lines 3120–3230 return the mnemonic *code* in MNMON by comparing the three stored mnemonic characters against the ASCII table ARY1. If no match is found then the characters were not a legal 6510 mnemonic, in which case an error exit results.

Relative Instructions

Having got the mnemonic code we can now tell if our instruction code is a relative branch by comparing it against RELTAB (at lines 5340–5350) which is a table of mnemonic codes for all the relative branch instructions. This is done by lines 3330–3390. If the comparison fails control passes to NOTREL at line 3810, and otherwise ISREL (lines 3410–3760) gets in the branch destination address and uses it to calculate the offset byte for the relative instruction. REL concludes by jumping to COMSL at line 4830, with the X register containing the code for *relative* addressing ($0B) and the Accumulator containing the offset byte. More of COMSL in a moment.

Non-relative Instructions

As we have seen the relative branch instructions are a special case. For all other instructions the addressing mode is deduced from the characters input. This is best understood by looking at lines 3810–4740. If the next character after the mnemonic is A then the addressing mode is *Accumulator*. If #, then it must be an *immediate* instruction, and so on. We need to input characters until the addressing mode is unequivocally established and we then load X with the code for that mode before jumping to one of COMX, COMXL, COMS or COMSL at lines 4790–4830 depending on what has been pushed on the stack in the process.

For instance, if the first character is $ there is a possibility of six different modes. We start by assuming that it is Absolute addressing

and branch to ABSOLU at line 4050. A hex byte is input in A and pushed on to the stack. We try to input another hex byte, but if on return from HEXGET the Carry flag and Zero flag are both set then the buffer is empty. In this case we can assume Zero-page addressing and jump to the ZPAGE routine at line 4380. Otherwise, we push the second hex byte on to the stack and search the buffer for a ,X or ,Y sequence to check for indexed modes, and so on.

Lines 4790–4830 perform any necessary pulling of stored operands off the stack and store these *after* the instruction address. At REGRUP in lines 4890–4980 the addressing code is stored in ADMOD and a search of the ARY2 and ARY3 tables is made for values of mnemonic code and addressing code corresponding to the values held in MNMON and ADMOD. The value of the index upon exit from this loop is the op code for our instruction and so it is finally (lines 5030–5050) placed *at* the instruction address and exit is made via the common exit routine COMEXT in the control program.

Error Exit (Lines 5160–5200)

Since errors may be encountered with various data pushed on the stack the error exit has three points of entry, X1, X2 and X3, which perform the PLA operations necessary to tidy things up. The carriage return character output by line 5180 is to counter the 'cursor up' performed at the start of the routine.

Command File 2: The @M, @I (Group (a)) and @W, @C (Group (b)) Commands

Assembler Listing: Command File 2

```
1020 *=$C880
1030 ;----------
1040 CURR   =$4D
1050 FLAG   =$FD
1060 ;----------
1070 EXIT0  =$C0A0
1080 COMEXT =$C0AB
1090 SPACE  =$C108
1100 SPACES =$C10A
1110 ADROUT =$C113
1120 OUTPUT =$C13A
1130 HEXGET =$C150
1140 NXTBYT =$C17F
1150 GETADR =$C1A0
```

```
1160 ;----------
1170 ;--OUTPUT MEMORY AS HEX--
1180 ;PRINT CURRENT ADDRESS IN HEX
1190 ;/GET NEXT 8 BYTES FROM (ADDRESS)
1200 ;AND OUTPUT IN HEX WITH A SPACE
1210 ;BETWEEN EACH/PRINT 5 SPACES
1220 ;----------
1230 MEMORY JSR ADROUT
1240 ;----------
1250         LDA #$07
1260         STA FLAG
1270         LDY #0
1280 RA5     LDA (CURR),Y
1290 RA6     JSR OUTPUT
1300         JSR SPACE
1310         DEC FLAG
1320         BMI RA7
1330 ;----------
1340         JSR NXTBYT
1350         JMP RA6
1360 ;----------
1370 RA7     LDX #$05
1380         JMP SPACES
1390 ;----------
1400 ;--OUTPUT MEMORY AS ASCII--
1410 ;PRINT CURRENT ADDRESS IN HEX/GET
1420 ;NEXT 8 BYTES FROM (ADDRESS) AND
1430 ;PRINT AS ASCII CHARACTERS/IF BYTE
1440 ;>$80 THEN SUBTRACT $80/IF BYTE IS
1450 ;>$60 OR <$20 OR='COLON','@',OR
1460 ;'*' THEN PRINT '.'/IF BYTE=
1470 ;'SPACE' THEN PRINT '*'
1480 ;----------
1490 INTERG JSR ADROUT
1500 ;----------
1510         LDX #$07
1520         LDY #0
1530 TA6     LDA (CURR),Y
1540 TA7     BPL TA8
1550         EOR #$80
1560 TA8     CMP #$60
1570         BCS OUTOF
1580         CMP #$40        ;'@
1590         BEQ OUTOF
```

```
1600          CMP #$3A        ;':
1610          BEQ OUTOF
1620          CMP #$2A        ;'*
1630          BEQ OUTOF
1640          CMP #$20        ;'SPACE
1650          BCC OUTOF
1660          BNE HITIT
1670          LDA #$2A        ;'*
1680          BNE HITIT
1690 ;----------
1700 OUTOF    LDA #$2E        ;'.
1710 HITIT    JSR $FFD2
1720          DEX
1730          BMI HITIT2
1740          JSR NXTBYT
1750          JMP TA7
1760 ;----------
1770 HITIT2   LDX #$05
1780          JMP SPACES
1790 ;----------
1800 ;--WRITE HEX BYTES TO MEMORY--
1810 ;PRINT [CURSOR UP]/GET IN HEX
1820 ;ADDRESS FROM BUFFER-IF ILLEGAL
1830 ;THEN EXIT/GET IN NEXT 8 HEX BYTES
1840 ;FROM BUFFER-IF MORE THAN 8 OR IF
1850 ;NON-HEX THEN EXIT/STORE BINARY
1860 ;VALUES AT (ADDRESS)/PRINT 'RET',
1870 ;'@','.',ADDRESS+8,'SPACE'/EXIT
1880 ;----------
1890 MWRITE   JSR GETADR
1900          BCS MOUT
1910 ;----------
1920          LDY #0
1930          JSR HEXGET
1940          BCC DUBIOS
1950          BCS MOUT
1960 ;----------
1970 PI       JSR HEXGET
1980          BCS NACCEP
1990 DUBIOS   STA $0380,Y
2000          INY
2010          CPY #$09
2020          BCC PI
2030          BCS MOUT
```

```
2040 ;----------
2050 NACCEP BNE MOUT
2060 ;----------
2070        DEY
2080 OK1    LDA $0380,Y
2090        STA (CURR),Y
2100        DEY
2110        BPL OK1
2120 ;----------
2130        JMP COMEXT
2140 ;----------
2150 MOUT   LDA #$0D
2160        JSR $FFD2
2170        JMP EXIT0
2180 ;----------
2190 ;--WRITE ASCII BYTES TO MEMORY--
2200 ;PRINT [CURSOR UP]/GET IN HEX
2210 ;ADDRESS FROM BUFFER-EXIT IF
2220 ;ILLLEGAL OR NON HEX/GET IN NEXT
2230 ;8 ASCII CHARS FROM BUFFER-EXIT
2240 ;IF MORE THAN 8/STORE ASCII CHARS
2250 ;AT (ADDRESS) UNLESS ='.' THEN
2260 ;STORE 'SPACE'/PRINT 'RET','@',
2270 ;'.',ADDRESS+8,'SPACE'/EXIT
2280 ;----------
2290 TXTINP JSR GETADR
2300        BCS TOUT2
2310 ;----------
2320        LDY #$FF
2330 REBOU  INY
2340        JSR $0073
2350        STA $0380,Y
2360        BNE REBOU
2370        TYA
2380        BEQ TOUT2
2390        CPY #$09
2400        BCS TOUT2
2410 ;----------
2420 TK0    DEY
2430        BMI TK3
2440 TK1    LDA $0380,Y
2450        CMP #$2E      ;'.
2460        BEQ TK0
2470        CMP #$2A      ;'*
```

```
2480          BNE TK2
2490          LDA #$20      ;'SPACE
2500 TK2      STA (CURR),Y
2510          BNE TK0
2520 ;----------
2530 TK3      JMP COMEXT
2540 ;----------
2550 TOUT2    LDA #$0D
2560          JSR $FFD2
2570          JMP EXIT0
2580 ;----------
```

BASIC Loader: Command File 2

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 51327+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA32,19,193,169,7,133,253,160
112 DATA0,177,77,32,58,193,32,8
113 DATA193,198,253,48,6,32,127,193
114 DATA76,139,200,162,5,76,10,193
115 DATA32,19,193,162,7,160,0,177
116 DATA77,16,2,73,128,201,96,176
117 DATA22,201,64,240,18,201,58,240
118 DATA14,201,42,240,10,201,32,144
119 DATA-6901
120 REM --- BLOCK 2
121 DATA6,208,6,169,42,208,2,169
122 DATA46,32,210,255,202,48,6,32
123 DATA127,193,76,169,200,162,5,76
124 DATA10,193,32,160,193,176,38,160
125 DATA0,32,80,193,144,7,176,29
126 DATA32,80,193,176,10,153,128,3
127 DATA200,192,9,144,243,176,14,208
128 DATA12,136,185,128,3,145,77,136
129 DATA-7055
130 REM --- BLOCK 3
```

```
131 DATA16,248,76,171,192,169,13,32
132 DATA210,255,76,160,192,32,160,193
133 DATA176,41,160,255,200,32,115,0
134 DATA153,128,3,208,247,152,240,27
135 DATA192,9,176,23,136,48,17,185
136 DATA128,3,201,46,240,246,201,42
137 DATA208,2,169,32,145,77,208,236
138 DATA76,171,192,169,13,32,210,255
139 DATA-8420
140 REM --- BLOCK 4
141 DATA76,160,192
142 DATA-428
143 DATA END
1000 REM --ENABLE @M
1001 POKE 49697,109:POKE 49698,192
1002 REM --ENABLE @W
1003 POKE 49703,217:POKE 49704,200
1004 REM --ENABLE @I
1005 POKE 49699,109:POKE 49700,192
1006 REM --ENABLE @C
1007 POKE 49705,12:POKE 49706,201
1008 END
```

Group (a) Commands @M and @I

Dump Eight Bytes of Memory as Hex for @M (Lines 1230–1380)

As with the DSASBL routine we first print the current address by means of ADROUT. The rest of the routine is very simple compared to the disassembler. We get in the next eight bytes from (CURR) and use OUTPUT to print them as hex bytes, with a space between each. The location FLAG is used as a counter for the loop, since both index registers are occupied with other things.

Dump Eight Bytes of Memory as ASCII for @I (Lines 1490–1780)

This routine is structured in exactly the same way as the hex dump routine above, but is complicated by a number of checks required for various reasons:

1. ASCII codes greater than $80 will produce graphics characters, which are not very useful, so in line 1550 $80 is subtracted (by 'flipping' bit 7) from all values over $80 to make them alphanumeric.
2. Many ASCII codes are not printable and others, when printed,

produce either graphics characters or characters which confuse the BASIC editor or 'Wedge-MON'. The program checks for such codes and replaces them with the code for a full stop character. The disallowed codes are those greater than $60 (graphics) or less than $20 (unprintable), @ (confuses Wedge-MON) and : (confuses BASIC).

3. Space characters are intercepted and replaced with *, because the BASIC editor has a nasty habit of eating leading spaces. To avoid confusion with 'space' the * character must then also be replaced by the full stop.

In view of the foregoing it should be obvious that what gets printed on the screen by @I may well not be an exact representation of the memory that was read. You should now finally see why, if you try try to re-enter ASCII dumps back into memory with @C, you may corrupt the region concerned!

Group (b) Commands @W and @C

These two commands are effectively the opposites of the Group (a) commands @M and @I. Unlike the Group (a) commands, which are all passed by the main module to a common address MAINLP, the Group (b) commands are initially handled by separate routines but come together at COMEXT when the processing becomes similar.

Write Eight Hex Bytes to Memory: @W (Lines 1890–2170)

The routine GETADR prints 'cursor up' and places the command address into (CURR). Lines 1920–2030 get eight hex bytes into a buffer located in Page Three. If there are more than eight bytes or any other errors in the line the whole line is rejected. Otherwise the eight bytes are stored in eight consecutive memory locations starting from the command address (CURR). Exit is by the common exit routine COMEXT.

Write Eight ASCII Bytes to Memory: @C (Lines 2290–2580)

This routine follows a similar procedure to the hex input command above. You should merely note that full stop characters (#$2E) are ignored and not stored in memory and that all asterisks are replaced by 'space' characters. Exit is via COMEXT.

Command File 3: The @T (Group (c)) Command

Assembler Listing: Command File 3

```
1020 *=$C980
1030 ;----------
1040 STRT   =$4D
1050 END    =$4F
1060 NEWST  =$51
1070 SIZE   =$59
1080 ;----------
1090 EXIT0  =$C0A0
1100 PARAMS =$C0E6
1110 PARENT =$C0E8
1120 CNCLUD =$C1C8
1130 ;----------
1140 ;--MEMORY TRANSFER ROUTINE--
1150 ;GET IN PARAMETERS(START-END OF
1160 ;TRANSFER BLOCK) IF ILLEGAL OR
1170 ;IF ONLY 1 PARAMETER THEN EXIT/
1180 ;GET IN COMMA/GET IN THIRD
1190 ;PARAMETER IN NEWST (NEW START)
1200 ;/SUBTRACT START PARAMETER
1210 ;FROM END TO GIVE SIZE OF
1220 ;TRANSFER BLOCK IN SIZE - IF
1230 ;NEGATIVE THEN EXIT/COMPARE
1240 ;START WITH NEW START TO DETERMINE
1250 ;WHETHER MEMORY MOVE IS FROM LOWER
1260 ;ADDRESS TO HIGHER ADDRESS OR VICE
1270 ;VERSA/PERFORM CORRESPONDING
1280 ;MEMORY TRANSFER/PRINT 'OK'/EXIT
1290 ;----------
1300 TRANSF JSR PARAMS
1310        BCS TROUT
1320        BEQ TROUT
1330        JSR $0073
1340        CMP #$2C       ;',
1350        BNE TROUT
1360        LDX #$02
1370        JSR PARENT
1380        BCS TROUT
1390 ;----------
1400        SEC
1410        LDA END
```

```
1420          SBC STRT
1430          STA SIZE
1440          LDA END+1
1450          SBC STRT+1
1460          STA SIZE+1
1470          BCC TROUT
1480 ;----------
1490          LDA NEWST
1500          CMP STRT
1510          LDA NEWST+1
1520          SBC STRT+1
1530          BCC SHRINK
1540          BCS EXPAND
1550 ;----------
1560 TROUT    JMP EXIT0
1570 ;----------
1580 ;MOVE MEMORY FROM HIGHER TO LOWER
1590 ;ADDRESS
1600 ;----------
1610 SHRINK   LDY #0
1620          LDX SIZE+1
1630          BEQ ENT
1640 ;----------
1650 N2       LDA (STRT),Y
1660          STA (NEWST),Y
1670          INY
1680          BNE N2
1690          INC STRT+1
1700          INC NEWST+1
1710          DEX
1720          BNE N2
1730 ;----------
1740 ENT      LDX SIZE
1750          BEQ CNCL
1760 ;----------
1770          INX
1780 N3       LDA (STRT),Y
1790          STA (NEWST),Y
1800          INY
1810          DEX
1820          BNE N3
1830          BEQ CNCL
1840 ;----------
```

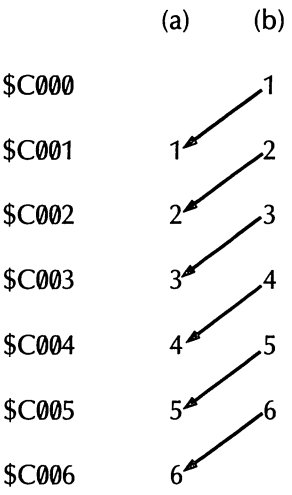
```
1850 ;MOVE MEMORY FROM LOWER TO HIGHER
1860 ;ADDRESS
1870 ;----------
1880 EXPAND CLC
1890        LDA STRT+1
1900        ADC SIZE+1
1910        STA STRT+1
1920        CLC
1930        LDA NEWST+1
1940        ADC SIZE+1
1950        STA NEWST+1
1960 ;----------
1970        LDX SIZE+1
1980        INX
1990        LDY SIZE
2000        BEQ NEXT2
2010 ;----------
2020 NEXT1  LDA (STRT),Y
2030        STA (NEWST),Y
2040        DEY
2050        BNE NEXT1
2060 ;----------
2070 NEXT2  LDA (STRT),Y
2080        STA (NEWST),Y
2090        DEY
2100        DEC STRT+1
2110        DEC NEWST+1
2120        DEX
2130        BNE NEXT1
2140 ;----------
2150 CNCL   JMP CNCLUD
2160 ;----------
```

BASIC Loader: Command File 3

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 51583+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
```

```
110 REM --- BLOCK 1
111 DATA32,230,192,176,43,240,41,32
112 DATA115,0,201,44,208,34,162,2
113 DATA32,232,192,176,27,56,165,79
114 DATA229,77,133,89,165,80,229,78
115 DATA133,90,144,12,165,81,197,77
116 DATA165,82,229,78,144,5,176,38
117 DATA76,160,192,160,0,166,90,240
118 DATA14,177,77,145,81,200,208,249
119 DATA-7842
120 REM --- BLOCK 2
121 DATA230,78,230,82,202,208,242,166
122 DATA89,240,51,232,177,77,145,81
123 DATA200,202,208,248,240,40,24,165
124 DATA78,101,90,133,78,24,165,82
125 DATA101,90,133,82,166,90,232,164
126 DATA89,240,7,177,77,145,81,136
127 DATA208,249,177,77,145,81,136,198
128 DATA78,198,82,202,208,237,76,200
129 DATA-9170
130 REM --- BLOCK 3
131 DATA193
132 DATA-193
133 DATA END
1000 REM --ENABLE @T
1001 POKE 49707,127:POKE 49708,201
1002 END
```

Strictly speaking, this routine is not so much a memory transfer process as a memory copy, since the area that is copied from is left unchanged unless overwritten. Writing a memory transfer routine is not quite as straightforward as it may seem at first. A problem arises when the 'destination' range for the memory block overlaps with the 'source' range. If you are not careful you can corrupt the data as you are moving it. Perhaps a diagram will make this clear:



Suppose we want to copy memory block (b) at $C000–$C005 to block (a) at $C001–$C006. There are two possible ways of executing the move.

1. Start from $C000 and work *down* in an *incrementing* loop:

```
      LDX #$00
LOOP  LDA $C000,X
      STA $C001,X
      INX
      CPX #$06
      BNE LOOP
```

2. Start from $C006 and work *up* in a *decrementing* loop:

```
      LDX #$06
LOOP  LDA $C000,X
      STA $C001,X
      DEX
      BPL LOOP
```

The first method is incorrect, since it will corrupt the second element in the table that it is about to move, by overwriting it with the first shift operation. If, however, we were moving block (a) to block (b), i.e. from a higher to a lower address, the opposite would be true and the second method would be abortive. We must therefore, after having got in the parameters for our transfer, check whether the source address is higher or lower in memory than the destination address, so that the appropriate 'move' routine can be executed.

Get Parameters (Lines 1300–1380)

The PARAMS routine should be familiar to you by now. Here it is used to get in the STRT and END parameters. Lines 1330–1350 get a 'comma'. In lines 1360–1380 we are using PARAMS in a different way. By entering at the location PARENT with #$02 held in X we get the third parameter (new start address) in $51,$52 (NEWST) instead of the usual $4D,$4E (STRT).

Subtract STRT from END (Lines 1400–1470)

This subtraction gives us the extent of the block of memory for transfer, which is stored in SIZE. The BCC in line 1470 checks for the error condition of the start parameter being greater than the end parameter and exits should this be the case.

Compare STRT to NEWST (Lines 1490–1540)

This comparison determines which move routine is to be used. If NEWST is greater than STRT then control jumps to EXPAND (lines 1880–2150), and otherwise the jump is to SHRINK (lines 1610–1830). This method of comparing two 16-bit numbers is quite neat, using the CMP instruction like a SBC while avoiding the need for an initial SEC operation.

Move Memory from Higher to Lower Address (Lines 1610–1830)

The high byte of SIZE (the extent of the transfer block) is used as the index for a loop (lines 1650–1720) transferring (SIZE+1) minus 1 pages (256 byte blocks) of memory.

The low byte of SIZE is used to index a loop (lines 1770–1830) transferring the remaining *bytes*, calculated as SIZE–1. Because the transfer block is of indefinite size it is necessary to use indirect addressing techniques.

Move Memory from Lower to Higher Address (Lines 1880–2130)

SIZE–1 is first added to STRT+1 and NEWST+1 and register Y is loaded with SIZE. The X register is loaded with SIZE+1. Therefore, (STRT),Y and (NEWST),Y respectively point to the bottom of the transfer block and the destination block. The loop at lines 1970–2050 transfers the remainder bytes while the loop at lines 2070–2130 transfers X *pages*.

The exits for this routine are TROUT at line 1560 which is the error exit and CNCL at line 2150 which prints OK.

Command File 4: The @F (Group (c)) Command

Assembler Listing: Command File 4

```
1020 *=$CA80
1030 ;----------
1040 STRT  =$4D;*
1050 END   =$4F;*
1060 ;----------
1070 EXIT0  =$C0A0
1080 PARAMS =$C0E6
1090 HEXGET =$C150
1100 CNCLUD =$C1C8
1110 ;----------
1120 ;--FILL MEMORY ROUTINE--
1130 ;GET IN PARAMETERS(START-END)/
1140 ;SUBTRACT START FROM END-IF
1150 ;NEGATIVE THEN EXIT/GET COMMA/GET
1160 ;THIRD PARAMETER/STORE AT
1170 ;START ADDRESS/COMPARE START TO
1180 ;END-IF EQUAL THEN FINISH ELSE
1190 ;INCREMENT START AND LOOP BACK/
1200 ;----------
1210 FILLUP JSR PARAMS
1220 ↑      BCS FOUT
1230 ↑      BEQ FOUT
1240 ;----------
1250 ↑      SEC
1260 ↑      LDA END
1270 ↑      SBC STRT
1280 ↑      LDA END+1
1290 ↑      SBC STRT+1
1300 ↑      BCC FOUT
1310 ;----------
1320 ↑      JSR $0073
1330 ↑      CMP #$2C       ;',
1340 ↑      BNE FOUT
1350 ↑      JSR HEXGET
1360 ↑      BCS FOUT
1370 ;----------
1380 ↑      LDY #0
1390 FILL1  STA (STRT),Y
1400 ↑      LDX STRT
1410 ↑      CPX END
```

```
1420 ↑        BNE INCR
1430 ↑        LDX STRT+1
1440 ↑        CPX END+1
1450 ↑        BNE INCR
1460 ;----------
1470 ↑        JMP CNCLUD
1480 ;----------
1490 INCR     INC STRT
1500 ↑        BNE FILL1
1510 ↑        INC STRT+1
1520 ↑        JMP FILL1
1530 ;----------
1540 FOUT     JMP EXIT0
1550 ;----------
```

BASIC Loader: Command File 4

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 51839+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA32,230,192,176,53,240,51,56
112 DATA165,79,229,77,165,80,229,78
113 DATA144,40,32,115,0,201,44,208
114 DATA33,32,80,193,176,28,160,0
115 DATA145,77,166,77,228,79,208,9
116 DATA166,78,228,80,208,3,76,200
117 DATA193,230,77,208,235,230,78,76
118 DATA160,202,76,160,192
119 DATA-7763
120 DATA END
1000 REM --ENABLE @F
1001 POKE 49709,127:POKE 49710,202
1002 END
```

The jump to the PARAMS routine gets in the range parameters STRT and END and the loop at lines 1250–1300 checks for a nonsense entry.

Lines 1320–1360 get in a comma and a single hex byte from HEXGET. At lines 1380–1520 the hex byte is then stored at consecutive locations from the lower value (STRT) of the memory range and the pointer incremented until it is equal to the higher value of the memory range (END). The error exit path is through FOUT at line 1540, otherwise exit is via CNCLUD at line 1470.

Command File 5: The @H (Group (c)) Command

Assembler Listing: Command File 5

```
1020 *=$CB00
1030 ;----------
1040 STRT   =$4D
1050 END    =$4F
1060 TEMPP  =$51
1070 COUNT1 =$FD
1080 COUNT2 =$FE
1090 ;----------
1100 BUFRR  =$0380
1110 ;----------
1120 EXIT0  =$C0A0
1130 PARAMS =$C0E6
1140 SPACE  =$C108
1150 OUTPUT =$C13A
1160 HEXGET =$C150
1170 CNCLUD =$C1C8
1180 ;----------
1190 ;--HUNT ROUTINE--
1200 ;GET IN PARAMETERS (START-END)
1210 ;SUBTRACT START FROM END-IF
1220 ;NEGATIVE THEN EXIT/GET COMMA/IF
1230 ;NEXT CHARCTER IS HEX OR '  THEN
1240 ;GET IN SUBSEQUENT BYTES AS HEX/IF
1250 ;ASCII THEN GET ASCII BYTES AND
1260 ;STORE THEM IN BUFFER/STORE INDEX
1270 ;IN COUNT1/CHECK FOR STOP KEY-IF
1280 ;PRESSED EXIT/COMPARE CONTENTS OF
1290 ;BUFFER WITH CONSECUTIVE MEMORY
1300 ;LOCATIONS STARTING AT START/IF
1310 ;SEQUENCE FOUND THEN PRINT ADDRESS
1320 ;WHERE SEQUENCE STARTS/IF SIX
1330 ;ADDRESSES HAVE BEEN PRINTED THEN
1340 ;PRINT 'RET'/IF CURRENT ADDRESS=
1350 ;END THEN FINISH AND PRINT OK
```

```
1360 ;----------
1370 HUNTER JSR PARAMS
1380        BCS HOUT
1390        BEQ HOUT
1400 ;----------
1410        LDA END
1420        CMP STRT
1430        LDA END+1
1440        SBC STRT+1
1450        BCC HOUT
1460 ;----------
1470        JSR $0073
1480        CMP #$2C        ;',
1490        BNE HOUT
1500 ;----------
1510        JSR HEXGET
1520        BCC BINGET
1530        BEQ HOUT
1540        CMP #$2A        ;'*
1550        BEQ BINGET
1560        CMP #$27        ;''
1570        BNE HOUT
1580 ;----------
1590 ;GET ASCII BYTES
1600 ;----------
1610        DEC $81
1620        LDY #$FF
1630 ASCGT2 INY
1640        JSR $0073
1650        STA BUFRR,Y
1660        BNE ASCGT2
1670        INC $81
1680        TYA
1690        BNE COLLEC
1700        BEQ HOUT
1710 ;----------
1720 ;GET HEX BYTES OR '*'
1730 ;----------
1740 BINGET LDY #0
1750 BINGT2 STA BUFRR,Y
1760        INY
1770        JSR HEXGET
1780        BCC BINGT2
1790        BEQ COLLEC
```

```
1800          CMP #$2A        ;'*
1810          BEQ BINGT2
1820 HOUT     JMP EXIT0
1830 ;----------
1840 COLLEC   STY COUNT1
1850          LDA #$07
1860          STA COUNT2
1870          LDY #0
1880          LDX #0
1890 ;----------
1900 ;SEARCH MEMORY FOR SEQUENCE
1910 ;----------
1920 SERCH    BIT $91
1930          BPL FINSH
1940 ;----------
1950          LDA BUFRR,X
1960          CMP (STRT),Y
1970          BEQ MATCH
1980          CMP #$2A        ;'*
1990          BEQ MATCH
2000 ;----------
2010          TXA
2020          BEQ IMEM
2030          LDA TEMPP
2040          STA STRT
2050          LDA TEMPP+1
2060          STA STRT+1
2070          JMP ABORT
2080 ;----------
2090 MATCH    TXA
2100          BNE NNOVO
2110          LDA STRT
2120          STA TEMPP
2130          LDA STRT+1
2140          STA TEMPP+1
2150 ;----------
2160 NNOVO    INY
2170          INX
2180          CPX COUNT1
2190          BNE SERCH
2200 ;----------
2210          LDA TEMPP+1
2220          JSR OUTPUT
2230          LDA TEMPP
```

```
2240          JSR OUTPUT
2250          JSR SPACE
2260          DEC COUNT2
2270          BNE NORET
2280          LDA #$0D        ;'RET
2290          JSR $FFD2
2300          LDA #$07
2310          STA COUNT2
2320 ;----------
2330 NORET    DEY
2340          CLC
2350          TYA
2360          ADC STRT
2370          STA STRT
2380          BCC ABORT
2390          INC STRT+1
2400 ;----------
2410 ABORT    LDY #0
2420          LDX #0
2430 ;----------
2440 IMEM     INC STRT
2450          BNE CMPAR
2460          INC STRT+1
2470 ;----------
2480 CMPAR    LDA END
2490          CMP STRT
2500          LDA END+1
2510          SBC STRT+1
2520          BCS SERCH
2530 ;----------
2540 FINSH    JMP CNCLUD
2550 ;----------
```

BASIC Loader: Command File 5

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 51967+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
```

```
110 REM --- BLOCK 1
111 DATA32,230,192,176,71,240,69,165
112 DATA79,197,77,165,80,229,78,144
113 DATA59,32,115,0,201,44,208,52
114 DATA32,80,193,144,30,240,45,201
115 DATA42,240,24,201,39,208,37,198
116 DATA129,160,255,200,32,115,0,153
117 DATA128,3,208,247,230,129,152,208
118 DATA22,240,17,160,0,153,128,3
119 DATA-7961
120 REM --- BLOCK 2
121 DATA200,32,80,193,144,247,240,7
122 DATA201,42,240,241,76,160,192,132
123 DATA253,169,7,133,254,160,0,162
124 DATA0,36,145,16,99,189,128,3
125 DATA209,77,240,18,201,42,240,14
126 DATA138,240,69,165,81,133,77,165
127 DATA82,133,78,76,172,203,138,208
128 DATA8,165,77,133,81,165,78,133
129 DATA-8220
130 REM --- BLOCK 3
131 DATA82,200,232,228,253,208,210,165
132 DATA82,32,58,193,165,81,32,58
133 DATA193,32,8,193,198,254,208,9
134 DATA169,13,32,210,255,169,7,133
135 DATA254,136,24,152,101,77,133,77
136 DATA144,2,230,78,160,0,162,0
137 DATA230,77,208,2,230,78,165,79
138 DATA197,77,165,80,229,78,176,153
139 DATA-8316
140 REM --- BLOCK 4
141 DATA76,200,193
142 DATA-469
143 DATA END
1000 REM --ENABLE @H
1001 POKE 49711,255:POKE 49712,202
1002 END
```

Lines 1370–1490 get in the STRT and END parameters as usual. At line 1530 the next entry in the input buffer is analysed. If it is a hex byte or a * wildcard character then we go to BINGET at line 1740. If it is a ' (apostrophe) character we go to line 1610. Any other input causes an error exit. BINGET and the routine at line 1610 get in succeeding hex or ASCII bytes from the buffer and store them as a 'search string' at

BUFRR (located in Page Three memory at $0380 onwards). When the end of the input buffer is reached we pass on to the SERCH loop at line 1920, which searches consecutive memory locations for the search string. The search loop works like this:

1. If 'stop' key pressed then 17.
2. If next byte in the search string equals contents of (current address + Y index) then 6.
3. If search string byte is * character then 6.
4. If search string byte is the *first* value in search string then 14.
5. Load current address from TEMPP and go to 14.
6. If search string byte is *not* the first value in search string then 8.
7. Store current address in TEMPP
8. If search string byte is *not the last* value in search string then increment Y index and go to 1.
9. Print TEMPP
10. Print 'space'
11. If count + 1 <> zero then 13.
12. Print 'return'. Count = 7
13. Current address = address plus Y index
14. Set search string byte to STRT. Set Y index to zero
15. Increment current address
16. If current address not equal to end of search range then 1.
17. End

Command File 6: The @G and @R (Group (C)) Commands

Assembler Listing: Command File 6

```
1020 *=$CC00
1030 ;----------
1040 ASTOR  =$02AB
1050 XSTOR  =$02AC
1060 YSTOR  =$02AD
1070 PSTOR  =$02AE
1080 ;----------
1090 EXIT0  =$C0A0
1100 PARAMS =$C0E6
1110 SPACE  =$C108
1120 OUTPUT =$C13A
1130 HEXGET =$C150
1140 ;----------
1150 ;--GO TO DESTINATION ROUTINE--
1160 ;GET IN PARAMETER (DESTINATION)/
1170 ;PUSH ADDRESS OF REPL ON STACK/
```

```
1180 ;INITIALIZE REGSTRS FROM MEMORY
1190 ;,D,E/ AND PROCESSOR STATUS WITH
1200 ;$F3 TO MASK OUT BCD AND INTERRUPT
1210 ;FLAGS/JUMP TO DESTINATION ADDRESS
1220 ;----------
1230 GOTOIT JSR PARAMS
1240        BCS GOUT
1250        BNE GOUT
1260        LDA #>REPL-1
1270        PHA
1280        LDA #<REPL-1
1290        PHA
1300        LDA PSTOR
1310        AND #$F3
1320        PHA
1330        LDX XSTOR
1340        LDY YSTOR
1350        LDA ASTOR
1360        PLP
1370        JMP ($004D)
1380 ;----------
1390 ;ON RETURNING FROM @G STORE
1400 ;REGISTERS IN MEMORY/PRINT
1410 ;'RET'/JUMP TO @R ROUTINE
1420 ;----------
1430 REPL   STA ASTOR
1440        STX XSTOR
1450        STY YSTOR
1460        PHP
1470        PLA
1480        STA PSTOR
1490        LDA #$0D       ;'RET
1500        JSR $FFD2
1510        JMP RETURN
1520 ;----------
1530 GOUT   JMP EXIT0
1540 ;----------
1550 ;--SET REGISTERS FOR @G--
1560 ;GETIN UP TO 4 HEX BYTES AND STORE
1570 ;AT ASTOR ETC../PRINT [CURSOR UP]
1580 ;TWICE/PRINT REGISTER DESCRIPTIONS
1590 ;/PRINT CONTENTS OF ASTOR,C,D,E
1600 ;UNDER REGISTER DESCRIPTIONS/EXIT
1610 ;----------
```

```
1620 RGSTRS LDX #0
1630 GTREG1 JSR HEXGET
1640        BCC NEXTE
1650        BEQ PRIN
1660        BCS RGOUT
1670 NEXTE  STA ASTOR,X
1680        INX
1690        CPX #$04
1700        BCC GTREG1
1710 ;----------
1720 PRIN   LDA #$91        ;'CURS UP
1730        JSR $FFD2
1740        JSR $FFD2
1750 ;----------
1760 ;ENTERS HERE AFTER @G
1770 ;----------
1780 RETURN LDX #0
1790 GT     LDA DSCRIP,X
1800        JSR $FFD2
1810        INX
1820        CPX #$12
1830        BNE GT
1840 ;----------
1850        LDY #0
1860 GTREG2 LDA ASTOR,Y
1870        JSR OUTPUT
1880        JSR SPACE
1890        INY
1900        CPY #$04
1910        BCC GTREG2
1920 ;----------
1930        LDA #$0D
1940        JSR $FFD2
1950 ;----------
1960 RGOUT  JMP EXIT0
1970 ;----------------
1980 DSCRIP .ASC "   AC XR YR SR"
1990        .BYTE $0D
2000        .ASC "@R "
2010 ;----------------
```

BASIC Loader: Command File 6

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 52223+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA32,230,192,176,49,208,47,169
112 DATA204,72,169,31,72,173,174,2
113 DATA41,243,72,174,172,2,172,173
114 DATA2,173,171,2,40,108,77,0
115 DATA141,171,2,142,172,2,140,173
116 DATA2,8,104,141,174,2,169,13
117 DATA32,210,255,76,84,204,76,160
118 DATA192,162,0,32,80,193,144,4
119 DATA-7082
120 REM --- BLOCK 2
121 DATA240,10,176,50,157,171,2,232
122 DATA224,4,144,239,169,145,32,210
123 DATA255,32,210,255,162,0,189,121
124 DATA204,32,210,255,232,224,18,208
125 DATA245,160,0,185,171,2,32,58
126 DATA193,32,8,193,200,192,4,144
127 DATA242,169,13,32,210,255,76,160
128 DATA192,32,32,32,65,67,32,88
129 DATA-8328
130 REM --- BLOCK 3
131 DATA82,32,89,82,32,83,82,13
132 DATA64,82,32
133 DATA-673
134 DATA END
1000 REM --ENABLE @G
1001 POKE 49717,255:POKE 49718,203
1002 REM --ENABLE @R
1003 POKE 49719,56:POKE 49720,204
1004 END
```

Go to Address with Registers Set (Lines 1230–1530)

PARAMS gets in the destination address in $4D,$4E. Lines 1260–1320 push the two bytes of the REPL 'return address' on to the stack, so that an RTS will return control to REPL (line 1430). The contents of ASTOR, XSTOR, YSTOR and PSTOR are loaded into the A, X, Y and P (status) registers respectively. Bits 2 and 3 of the P register are 'masked out' using AND #$F3 to protect the Decimal and Interrupt flags. Finally, control is transferred to the user routine by an indirect JMP to the parameter address ($004D).

When control returns from the user routine to REPL the register values are stored back in ASTOR, XSTOR, YSTOR and PSTOR and a carriage return character is printed. A jump is then made to the @R command code, entering at RETURN (line 1780) to print out the register contents.

Set Registers for an @G Command (Lines 1620–1740)

Get in as many bytes as there are in the buffer (up to four) and store these in ASTOR, XSTOR, YSTOR and PSTOR. Print two 'cursor up' characters.

Output Current Register Contents (Lines 1780–1960)

Lines 1780–1830 output the register 'descriptions' stored in the table DSCRIP. Lines 1850–1910 output the current values of locations ASTOR, etc.

Command File 7: The @S and @L (Group (c)) Commands

Assembler Listing: Command File 7

```
1020 *=$CD00
1030 ;----------
1040 STRT   =$4D
1050 END    =$4F
1060 ;----------
1070 EXIT0  =$C0A0
1080 PARAMS =$C0E6
1090 HEXGET =$C150
1100 CNCLUD =$C1C8
1110 ;----------
1120 ;--MEMORY SAVE ROUTINE--
1130 ;EXIT IF NO PARAMETERS/USE ROM
1140 ;ROUTINE TO HANDLE FILENAME/GET
1150 ;COMMA/GET DEVICE IN HEX-USE TO
1160 ;SET UP LOGICAL FILE/GET COMMA/
```

```
1170 ;GET IN HEX PARAMETERS (START-END)
1180 ;USE AS VALUES FOR ROM SAVE MEMORY
1190 ;ROUTINE/IF ON RETURN CARRY SET
1200 ;THEN ERROREXIT/GET IN STATUS
1210 ;WORD-IF BIT 6 SET THEN ERROR EXIT
1220 ;/PRINT 'OK'/EXIT
1230 ;----------
1240 SAVEIT JSR $0073
1250        BEQ SOUT
1260        JSR $E257
1270        JSR $0079
1280        CMP #$2C       ;',
1290        BNE SOUT
1300        JSR HEXGET
1310        BCS SOUT
1320        TAX
1330        LDY #0
1340        JSR $FFBA
1350        JSR $0073
1360        CMP #$2C       ;',
1370        BNE SOUT
1380        JSR PARAMS
1390        BCS SOUT
1400        BEQ SOUT
1410        LDX END
1420        LDY END+1
1430        INX
1440        BNE SAVE
1450        INY
1460 SAVE   LDA #STRT
1470        JSR $FFD8
1480        BCC TESERR
1490 ;----------
1500 ERRR   LDA #<ERRMES
1510        LDY #>ERRMES
1520        JSR $AB1E
1530 SOUT   JMP EXIT0
1540 ;----------
1550 ;--LOAD ROUTINE--
1560 ;IF NO PARAMETERS THEN EXIT/
1570 ;HANDLE FILENAME VIA ROM ROUTINE/
1580 ;GET COMMA/GET DEVICE NUMBER-
1590 ;ASSUME MACHINE CODE LOAD AND
1600 ;SET UP LOGICAL FILE/IF CARRY SET
```

```
1610 ;THEN ERROR EXIT/GET STATUS WORD-
1620 ;IF BIT 6 SET THEN ERROR EXIT/
1630 ;PRINT 'OK'/EXIT
1640 ;----------
1650 LOADIT JSR $0073
1660        BEQ LOUT
1670        JSR $E257
1680        JSR $0079
1690        CMP #$2C        ;',
1700        BNE LOUT
1710        JSR HEXGET
1720        BCS LOUT
1730        TAX
1740        LDY #$01
1750        JSR $FFBA
1760        LDA #0
1770        JSR $FFD5
1780        BCS ERRR
1790 TESERR JSR $FFB7
1800        AND #$BF
1810        BNE ERRR
1820        JMP CNCLUD
1830 ;----------
1840 LOUT   JMP EXIT0
1850 ;----------------
1860 ERRMES .BYTE $0D,$0D
1870        .ASC "ERROR"
1880        .BYTE $0D,$00
1890 ;----------------
```

BASIC Loader: Command File 7

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 52479+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA32,115,0,240,57,32,87,226
112 DATA32,121,0,201,44,208,47,32
```

```
113 DATA80,193,176,42,170,160,0,32
114 DATA186,255,32,115,0,201,44,208
115 DATA29,32,230,192,176,24,240,22
116 DATA166,79,164,80,232,208,1,200
117 DATA169,77,32,216,255,144,43,169
118 DATA111,160,205,32,30,171,76,160
119 DATA-7493
120 REM --- BLOCK 2
121 DATA192,32,115,0,240,38,32,87
122 DATA226,32,121,0,201,44,208,28
123 DATA32,80,193,176,23,170,160,1
124 DATA32,186,255,169,0,32,213,255
125 DATA176,213,32,183,255,41,191,208
126 DATA206,76,200,193,76,160,192,13
127 DATA13,69,82,82,79,82,13,0
128 DATA-6408
129 DATA END
1000 REM --ENABLE @S
1001 POKE 49713,255:POKE 49714,204
1002 REM --ENABLE @L
1003 POKE 49715,64:POKE 49716,205
1004 END
```

Save RAM Between Limits to a Device (Lines 1240–1530)

The ROM routine at $E257 is part of the BASIC SAVE command. It gets an ASCII filename in quotes from the input buffer and calls the Kernal routine SETNAM ($FFBD) to set up a filename for our save. In lines 1270–1320 we recover the device number from the buffer for the Kernal SETLFS routine ($FFBA), which sets up a logical file. PARAMS gets in the limits for the command STRT - END which are set up as parameters for the Kernal SAVE routine ($FFD8). The X and Y registers contain respectively the low and high bytes of END, while A contains the low byte of the Zero-page *pointer* to STRT (Note: *not* the low byte of STRT). If the Carry is set upon returning then an error message is printed and the routine exits at ERRR, lines 1500–1530. Otherwise the branch to line 1790 is made, where the Kernal routine READST ($FFB7) is used to check the I/O Status byte for errors. If an error is found then branch to ERRR, else jump to CNCLUD at line 6130 to print OK and exit.

Load RAM from a Device (Lines 7595–7690)

This routine is very similar to that for @S, except that the Kernal routine LOAD ($FFD5) is used.

5 Extending BASIC: The command handler

Many commercial programs, such as Simons BASIC, are now available which add new commands to the 64's BASIC, and indeed seldom was a BASIC more in need of extension. To be fair to the designers, BASIC 2 was conceived by Microsoft with the old 8K PET in mind, but for reasons best known to CBM it has been carried over unchanged from these early machines to first the Vic 20 and later the 64. The BASIC does not offer such 'toolkit' commands as RENUMBER or DELETE, nor does it support any of the 64's graphics or sound features, and compared with the versions of BASIC implemented on other micros such as the BBC and Spectrum it seems a very basic BASIC indeed. Commercial packages like Simons' BASIC and Adamsoft's 'Ultrabasic 64' go a long way towards filling the gap, but do not come cheap. However, with a little general knowledge of the 64's operating system and some programming work there is no reason why you can not put together your own extended BASIC. Read on!

As with 'Wedge-MON' the BASIC language expander program presented here consists of a collection of utilities controlled by commands entered from the keyboard. There is a control section consisting of a 'Command handler' module which directs control to a number of independent command routines. As with 'Wedge-MON', once the command handler has been entered into memory then you can gradually build up the program by adding the command routines one by one whenever you feel like typing them in, and the BASIC expander has again been designed so as to be easily extended by the user. Beyond this, the similarity between the two programs fades, for whereas 'Wedge-MON' operates in parallel with BASIC routines, the 'Expander' program can be regarded as an extension of BASIC itself. 'Expander' commands are processed by the BASIC interpreter exactly as are standard BASIC commands like PRINT, LIST, RUN etc., and, unlike 'Wedge-MON' commands, can be used from within BASIC programs as well as in direct mode.

Entering the Command Handler

As with 'Wedge-MON' full source listings are given of all the 'Expander' code and if you have an assembler you may wish to assemble

this source code directly. In writing 'Expander' Brad Templeton's PAL assembler was used, which allows the assembly of large programs by 'chaining' together smaller source files. 'Expander' was assembled as ten such files. It is not crucial that your assembler possess this facility as each of the command routines will 'stand alone', i.e. has its own set of labels and does not utilise any sections of the other command routines. The 'Command handler' however, requires the start addresses of all the command handling routines for its action address tables and will therefore need to be treated differently. The idea will be to assemble these tables with dummy values and then POKE in the correct addresses as you assemble each individual command routine. We will come back to this later.

For those without assemblers the situation is simpler. The 'Command handler' has been provided as a BASIC loader program. Type this in and (having first saved it to tape or disc!) RUN it until no errors are flagged. You may now use 'Wedge-MON', if you have entered it, to save the machine code directly to tape or disc. For tape use:

```
@S"EXPANDER",01,8088-9700
```

For disc, use:

```
@S "@0:EXPANDER",08,8088-9700
```

If you have not yet entered the @L and @S command routines into 'Wedge-MON' then fear not, for here is a BASIC program that will do the same job.

Tape Version:

```
20 OPEN5,1,5,"EXPANDER,P,W"
30 PRINT#5,CHR$(136);CHR$(128);
40 FORX=32904TO38656
50 PRINT#5,CHR$(PEEK(X));
60 NEXTX
70 CLOSE5
```

Disc Version:

```
10 OPEN15,8,15,"I0
20 OPEN5,8,5,"@0:EXPANDER,P,W"
30 PRINT#5,CHR$(136);CHR$(128);
40 FORX=32904TO38656
50 PRINT#5,CHR$(PEEK(X));
```

```
60 NEXTX
70 CLOSE5
80 CLOSE15
```

Enabling the Command Handler
Once you have entered the 'Command handler' into memory then try entering SYS 33132 <return>. This should cause the following message to be printed if all is well:

**** EXPANDER 1984 JP GIBBONS ****

*** 21247 BASIC BYTES FREE ***

Next, try entering some of the 'Expander' commands, e.g. PLOT <return>, DRAW <return>, RENUMBER <return>. These commands should be accepted by the 64, i.e. they should not generate syntax errors. If they do then re-check your BASIC loader program for errors. However, you will find that they achieve precisely nothing, because although the 'Command handler' now recognizes these commands the command handling routines that execute them have not yet been entered. You will start to enter these in succeeding sections of the book. First let us examine the process of adding commands to the 64's BASIC, and see how the 'Command handler' works.

Modifying BASIC

We saw in the discussion on 'Wedge-MON' how we can redirect control to our own machine code routines from BASIC by examining keyboard input before it is processed by the BASIC editor. As you will recall, this was achieved by 'wedging' into the CHRGET routine in RAM. Although fairly simple to program there are certain disadvantages to this method, especially if the new commands are to be used from within BASIC programs. One important consideration is that since all characters fetched from the BASIC program text area have to be checked for new commands in addition to all the normal processing operations BASIC execution slows down noticeably.

An alternative approach to the problem is to extend the normal processing of BASIC to take in our new commands. This is trickier to program than a CHRGET wedge because of the way that BASIC keywords are stored in memory and it requires the alteration of four separate interpreter routines. These are the 'Tokenise BASIC Text' 'Token List', 'Command Dispatch' and 'Function Evaluator' routines.

Thanks to CBM these four important routines are all easily accessible to the machine code programmer through RAM vectors *in*

Page Three memory at $0304, $0306, $0308 and $030A respectively. You will recall from earlier sections that at strategic points in the ROM the ROM code JMP's indirectly back to itself through *vectors* located in RAM, and that by changing these vectors to point to our own code we can insert ('wedge') code into the ROM routines.

Before we can design and code wedges into these routines we must examine how the routines work.

The 'Tokenise' Routine

When a line of BASIC is entered from the keyboard one of the first processes it undergoes is scanning by the ROM 'Tokenise' routine. This searches the input buffer for BASIC keywords and operators such as PRINT, PEEK, +, *, etc., and replaces each with a *token* value between 128 and 255 decimal which corresponds to that particular keyword. The reason for tokenisation is that tokens both take up less space in memory and are executed more quickly by the interpreter than would ASCII coded keywords. When adding commands to BASIC we exploit the fact that not all the available tokens are used by the existing BASIC. We alter the 'Tokenise' routine so that it includes our new commands in its search for keywords, and we assign to the new commands some of the previously unused token values.

The 'Tokenise' routine is located as $A57C – $A612. It is a complicated routine, and if you disassemble it with 'Wedge-MON' you will probably find it difficult to understand at first. The basic idea is that the 'old' (untokenised) line is scanned and a 'new' (tokenised) line created. This requires two buffer pointers to be created, one into the old line (at $7A) and the other into the new line (at $71). The process operates as set out below:

Stage 1. Get Bytes from 'Old' Line

1. Load a byte from the old line
2. If byte value less than #$80 then 5.
3. If byte=pi character then 19.
4. Increment index and go to 1.

Stage 2. Check for Special Cases

5. If byte=space character then 19.
6. Store byte in $08
7. If byte=quote character then 24.
8. If value in $0F > #$40 (i.e. if a DATA statement is being processed) then 19.
9. If byte=? character then load byte with PRINT token (#$99) and go to 19.

10. If byte=decimal number character then 19.
11. Store 'new line' buffer index in $71
12. Initialise Token Counter $0B to zero
13. Store 'old line' buffer index in $7A
14. Initialise Keyword Table index to zero.

Stage 3. Compare Buffer Contents with Keyword and Derive Token

15. Byte=byte minus current keyword character. If byte not zero then 17.
16. Increment old line buffer index and keyword table index and go to 15.
17. If byte not equal to #$80 then 25.
18. OR byte with token counter $0B to give the token

Stage 4. Store Character or token in New Line

19. Load new line index from $71 and store byte at the end of the new line
20. If byte=zero (indicates end-of-line) then 27.
21. If byte=: (colon) (indicates end-of-statement) then load $0F with zero
22. If byte=DATA token then load $0F with #$49
23. If byte=REM token then load $08 with zero
24. Dump succeeding bytes from old line into new line until byte=value in $08, then go to 19.

Stage 5. Get Next Keyword From Keyword Table

25. Increment token counter $0B and increment keyword table index to point to the next keyword. Go to 15. if not end of table

Stage 6. Exit

26. If byte=a non-token then 19.
27. Store byte and end of new line plus 2
28. Reset CHRGET pointer to $01FF
29. Return

In order to follow the sequence of the 'Tokenise' routine given above, a few more facts need to be noted.

Entries in the ROM keyword table are made up of ASCII characters, with separate keywords demarked by the device of making the last character in each keyword equal to its ASCII value *plus* #$80, i.e. with bit 7 set. Hence, in step 17. above, the result of subtracting the final

character of a keyword from its buffer equivalent is #$80, indicating the end of the keyword. It is because the keyword table is arranged in this way that BASIC commands may be entered in shorthand form, e.g. L followed by <shift>I for LIST since <shift>I is equal to ASCII I plus #$80.

The most confusing part of the above code is probably that which deals with 'special cases'. These include PRINT strings in quotes, REM lines and DATA statements, each of which must be excluded from the tokenisation operation. For quote and REM lines this is handled by step 24. where data is transferred directly from the old line to the new line, bypassing the main routine until a certain value (stored in $08) is encountered. In the case of a quote line this value is the 'closing' quote, and for a REM statement a zero, indicating end-of-line. DATA statements are handled differently, with a flag ($0F) being set to #$49 to indicate to the main routine that a DATA line is being processed. A zero byte or colon resets this flag to indicate the end of the data statement.

The 'Token List' Routine

The LIST command ROM code exploits a 'Token List' routine which effectively performs the opposite function to that of the 'Tokenise', routine, since it retrieves tokens from memory and uses them to index into a table of keywords. The keyword is thereby LISTed to the screen. If we wish to add commands to BASIC we must obviously alter this routine so that our new tokens will also be processed. The 'Token List' routine is at $A71A – $A741, and is part of the larger BASIC 'Text LIST' routine starting at $A69C. Bytes are brought in from the BASIC text area and passed to 'Token List' to check for tokens. A step by step explanation of the operation is as follows:

Stage 1. Token List

1. If byte=non-token (less than #$80) then 7.
2. If byte=pi character then 7.
3. If quote flag is set (i.e. if bit 7 of $0F is set) then 7.
4. Token=token minus #$80
5. Work through keyword table decrementing token by one for each keyword encountered
6. When token=zero then print current keyword except for last byte (using $AB47 which is a ROM print routine)

Stage 2. List Non-token bytes

7. Print byte
8. If byte='quote' then invert (i.e. set if unset and vice versa) the 'quote' flag ($0F) by EORing with #$FF
9. Get next byte. Go to 1.

Note that since token bytes are all greater than #$80 (i.e. have bit 7 set) they may be discerned from ASCII characters and tested for by using the BPL and BMI instructions, which test bit 7.

The 'quote' flag ($0F) is used to indicate whether the current byte is part of, for instance, a PRINT or LOAD string and is therefore not a token. This flag is used for a similar function in the 'Tokenise' routine.

The 'Command Dispatch' Routine

The execution of individual BASIC commands is directed by the 'Command Dispatch' routine. It uses a command token as the index value to access a table of the action addresses of BASIC command handling routines. Control is then transferred to the address derived from this table. When adding commands we need to arrange that our tokens are similarily used to index into a table of our command routines. The 'Command Dispatch' routine is located between $A7E4 and $A803 and is part of the BASIC command execution loop starting at $A7AE. It functions as follows:

Stage 1. BASIC Execution Loop

1. Update CHRGET pointer

Stage 2. Command Dispatch Section

2. Get in a byte of BASIC text
3. Jump subroutine 5.
4. Go to 1.
5. If byte=zero then 12.
6. Byte=byte minus #$80
7. If Carry clear then byte was non-token (i.e. value less than #$80). Go to 12.
8. If byte greater than #$23 then it is not a command token (i.e. it is a function or operator token). Go to 12.
9. Multiply byte by two
10. Use byte as index into the command action address table and push action address on to stack
11. JMP to CHRGET
12. . . . other processing . . .

Note that not all tokens are commands. Some are *operators* such as + and –, OR and AND; others are *functions* like INT, ASC, and FRE. These are dealt with in a different fashion than are commands, and by other sections of the BASIC interpreter. In step 11. above, the command dispatcher exits by means of a JMP to CHRGET. This means that

when control passes to the command routine A will contain the next byte of BASIC text after the command token, which will probably be a parameter for the command. Notice that the RTS at the end of CHRGET is being used as a JMP to the command routine, as we saw in 'Wedge-MON'.

There are generally two possible exits from a command routine, the first being an error exit via the RAM vector ($0300), in which case the stack is cleared and BASIC execution terminates with an error message, and the second by means of an RTS which returns from the subroutine at step 5., control passing back to the top of the BASIC execution loop at step 1. of the sequence.

The 'Function Evaluator' Routine

As we noted, not all BASIC keywords are commands, and this routine is concerned with *functions*, such as SIN, COS, PEEK and the like. These are processed in a similar way to commands, but differ in that, rather than simply performing an operation, they must evaluate some parameter of the system or alter data in such a way as to give rise to a result, i.e. they return a value. This result must then be made available to the system through a ROM routine known as the Expression Evaluator so that it may be used in arithmetic expressions. Adding new functions requires us to wedge into the appropriate section of this ROM routine.

The 'Function Evaluator' is part of the Expression Evaluator, which is a complex, lengthy (approximately 0.5K) routine residing at $AD9E. When a string or numeric expression is expected by the interpreter (after an = or a PRINT statement, say) the expression evaluator gets in the following sequence of bytes from the input buffer, checking for syntax errors and accumulating the result of the expression (if numeric) in the Floating-Point Accumulator bytes in Zero-page $61–$70. Token values equal to or greater than $B4 are identified as functions and are sent to the 'Function Evaluator' routine at $AFA7 where they are subdivided into string and numerical functions, the latter being of particular interest to us. A call is made to a subroutine at $AEF1 which evaluates the parameter between the brackets of the function as a two-byte integer in $14,$15. Finally, the token is used to index a table of addresses of function-handling routines, the address being placed in Zero-page at $55,$56. Zero-page $54 contains an absolute JMP, so that a JSR $0054 passes control to the function code. Upon returning from this function subroutine control passes back to the expression evaluator. Wedging code into the 'Function Evaluator' routine requires that we distinguish our allocated function tokens from those of the 64's functions and use these to index a table of our function handling routine addresses.

Note that we JSR to the function handling routines through a Zero-page *indirect* subroutine similar to to that which was used in 'Wedge-MON'. This contrasts with the 'Command Dispatch' routine, where control is transferred to the command routines using an RTS acting on an address previously pushed on to the stack. This means that the action table for function routines will consist of precise addresses, unlike that for the command routines which consists of the address *minus one*.

The Command Handler Code

Assembler Listing: 'Expander' Command Handler

```
1030 ;---------------
1040 TOKENZ =$8000
1050 ;---------------
1060 *=TOKENZ+$88
1070 ;---------------
1080 ;--TAIL-END OF 'TOKENISE'--
1090 ;CHECK 'WORD' TABLE FOR NEW
1100 ;KEYWORDS/CONVERT TO NEW TOKENS
1110 ;--(SEE TEXT)--
1120 ;---------------
1130         BEQ WEDTOK
1140         NOP
1150         NOP
1160         NOP
1170 ;-----------
1180 EXXIT   JMP $A609
1190 ;-----------
1200 WEDTOK  LDY #$FF
1210         DEX
1220 STERT1  INY
1230         INX
1240 STERT2  LDA $0200,X
1250         SEC
1260         SBC WORD,Y
1270         BEQ STERT1
1280         CMP #$80
1290         BEQ TOKENZ+$49
1300         LDX $7A
1310         INC $0B
1320 LOOP    INY
1330         LDA WORD-1,Y
```

```
1340          BPL LOOP
1350          LDA WORD,Y
1360          BNE STERT2
1370          LDA $0200,X
1380          BPL TOKENZ+$4B
1390          BMI EXXIT
1400 ;---------------
1410 ;--'TEXT LIST' WEDGE--
1420 ;TURNS NEW TOKENS BACK INTO NEW
1430 ;KEYWORDS FOR THE LIST COMMAND
1440 ;--(SEE TEXT)--
1450 ;---------------
1460 LIST     BPL OUT2
1470          CMP #$FF
1480          BEQ OUT2
1490          BIT $0F
1500          BMI OUT2
1510 ;----------
1520          CMP #$CC
1530          BCC BACK
1540          SBC #$CB
1550          TAX
1560          STY $49
1570 ;----------
1580          LDY #$FF
1590 LOOPX    DEX
1600          BEQ POTIT
1610 LOOPY    INY
1620          LDA WORD,Y
1630          BPL LOOPY
1640          BMI LOOPX
1650 ;----------
1660 POTIT    INY
1670          LDA WORD,Y
1680          BMI OUT1
1690          JSR $AB47
1700          BNE POTIT
1710 ;----------
1720 BACK     JMP $A724
1730 ;----------
1740 OUT1     JMP $A6EF
1750 ;----------
1760 OUT2     JMP $A6F3
1770 ;---------------
```

```
1780 ;--'CHARACTER DISPATCH' WEDGE--
1790 ;JUMP TO HANDLING-ROUTINES FOR
1800 ;NEW COMMANDS--(SEE TEXT)--
1810 ;---------------
1820 DSPTCH JSR $0073
1830        CMP #$DA
1840        BCS GO
1850 ;----------
1860        JSR $0079
1870        JMP $A7E7
1880 ;----------
1890 GO     JSR MYCODE
1900        JMP $A7AE
1910 ;----------
1920 MYCODE SBC #$DA
1930        ASL A
1940        TAY
1950        LDA ADDRS+1,Y
1960        PHA
1970        LDA ADDRS,Y
1980        PHA
1990        JMP $0073
2000 ;---------------
2010 ;--TOKEN EVALUATION WEDGE--
2020 ;REDIRECTS CONTROL TO NEW FUNCTION
2030 ;ROUTINES--(SEE TEXT)
2040 ;---------------
2050 FUNCT  LDA #$00
2060        STA $0D
2070 STARRT JSR $0073
2080        BCS ACCEP1
2090 ;----------
2100 REJEC1 JMP $BCF3
2110 ;----------
2120 ACCEP1 JSR $B113
2130        BCC ACCEP2
2140 ;----------
2150        JMP $AF28
2160 ;----------
2170 ACCEP2 CMP #$FF      ;'π
2180        BNE ACCEP3
2190 ;----------
2200        JMP $AE9E
2210 ;----------
```

```
2220 ACCEP3 CMP #$2E      ;'.
2230        BEQ REJEC1
2240        CMP #$AB      ;(-)
2250        BNE ACCEP4
2260 ;----------
2270        JMP $AF0D
2280 ;----------
2290 ACCEP4 CMP #$AA      ;(+)
2300        BEQ STARRT
2310        CMP #$B4      ;(SGN)
2320        BCS EVAL
2330 ;----------
2340        JMP $AEB9
2350 ;---------------
2360 EVAL   ASL A
2370        PHA
2380        TAX
2390        JSR $0073
2400        CPX #$8F
2410        BCS EVAL2
2420 ;----------
2430        JMP $AFD1
2440 ;----------
2450 EVAL2  CPX #$98
2460        BCC NOTFNC
2470        CPX #$B4
2480        BCS NOTFNC
2490 ;----------
2500        JSR $AEF1
2510        PLA
2520        TAY
2530 ;----------
2540        LDA FUNCS-$98,Y
2550        STA $55
2560        LDA FUNCS-$97,Y
2570        STA $56
2580        JSR $0054
2590        JMP $AD8D
2600 ;----------
2610 NOTFNC JMP $AFB1
2620 ;---------------
2630 ;--ENTRY POINT--
2640 ;COPY DOWN ROM TOKENISE
2650 ;ROUTINE INTO RAM
```

```
2660 ;---------------
2670 ENTRY  LDX #$88
2680 L1     LDA $A57B,X
2690        STA TOKENZ-1,X
2700        DEX
2710        BNE L1
2720 ;---------------
2730 ;COPY BASIC ROM/AMEND LIST ROUTINE
2740 ;---------------
2750        LDY #$00
2760        LDA #$A0
2770        STY $14
2780        STA $15
2790        LDX #$1F
2800 RAJJ   LDA ($14),Y
2810        STA ($14),Y
2820        DEY
2830        BNE RAJJ
2840        INC $15
2850        DEX
2860        BPL RAJJ
2870 ;---------------
2880        LDA #$60
2890        STA $A714
2900 ;---------------
2910 ;CHANGE BASIC RAM VECTORS TO POINT
2920 ;TO MY WEDGE ROUTINES
2930 ;---------------
2940 L2     LDX #$07
2950 L3     LDA VCTRS,X
2960        STA $0304,X
2970        DEX
2980        BPL L3
2990 ;---------------
3000 ;PRINT CREDIT MESSAGE
3010 ;---------------
3020        LDA #<CREDIT
3030        LDY #>CREDIT
3040        JSR $AB1E
3050 ;---------------
3060 ;SET NEW TOP OF BASIC TO PROTECT
3070 ;MACHINE CODE
3080 ;---------------
3090        LDA #$00
```

```
3100          LDY #$5B
3110          STA $33
3120          STA $37
3130          STY $34
3140          STY $38
3150 ;---------------
3160 ;CALCULATE NUMBER OF BYTES FREE
3170 ;(=END OF MEMORY-END OF BASIC)/
3180 ;PRINT IT VIA $BDCD
3190 ;---------------
3200          SEC
3210          SBC $2D
3220          PHA
3230          TYA
3240          SBC $2E
3250          TAY
3260          CLC
3270          PLA
3280          ADC #$02
3290          TAX
3300          TYA
3310          ADC #$00
3320          JSR $BDCD
3330 ;---------------
3340 ;--PRINT 'BASIC BYTES FREE'--
3350 ;THE STRING SITS AT $E460
3360 ;---------------
3370          LDA #$60
3380          LDY #$E4
3390          JSR $AB1E
3400 ;---------------
3410 ;EXIT TO BASIC-PRINT 'READY
3420 ;---------------
3430          JMP $A474
3440 ;-----------------------------
3450 ;--DUMMY COMMAND/FUNCTION--
3460 ;---------------
3470 DUMMY    RTS
3480 ;-----------------------------
3490 ;--PROGRAM DATA--
3500 ;REPLACEMENT VALUES FOR BASIC
3510 ;RAM VECTORS
3520 ;---------------
3530 VCTRS    .WORD TOKENZ,LIST,DSPTCH
```

```
3540          .WORD FUNCT
3550 ;---------------
3560 ;CREDIT MESSAGE FOR PRINTING BY
3570 ;ROM ROUTINE $AB1E
3580 ;---------------
3590 CREDIT .BYTE $0D
3600          .ASC "    *** TOOLKIT 19"
3610          .ASC "84 JP.GIBBONS ***"
3620          .BYTE $0D,$0D
3630          .ASC "          "
3640          .BYTE $00
3650 ;---------------
3660 ;ACTION TABLE OF VECTORS
3670 ;FOR NEW FUNCTIONS
3680 ;---------------
3690 FUNCS    .WORD DUMMY,DUMMY,DUMMY
3700          .WORD DUMMY,DUMMY,DUMMY
3710          .WORD DUMMY,DUMMY,DUMMY
3720          .WORD DUMMY,DUMMY,DUMMY
3730          .WORD DUMMY,DUMMY
3740 ;---------------
3750 ;ACTION TABLE OF VECTORS FOR
3760 ;NEW COMMANDS (ALL -1)
3770 ;---------------
3780 ADDRS    .WORD DUMMY-1,DUMMY-1
3790          .WORD DUMMY-1,DUMMY-1
3800          .WORD DUMMY-1,DUMMY-1
3810          .WORD DUMMY-1,DUMMY-1
3820          .WORD DUMMY-1,DUMMY-1
3830          .WORD DUMMY-1,DUMMY-1
3840          .WORD DUMMY-1,DUMMY-1
3850          .WORD DUMMY-1,DUMMY-1
3860          .WORD DUMMY-1,DUMMY-1
3870          .WORD DUMMY-1,DUMMY-1
3880          .WORD DUMMY-1,DUMMY-1
3890          .WORD DUMMY-1,DUMMY-1
3900          .WORD DUMMY-1,DUMMY-1
3910          .WORD DUMMY-1,DUMMY-1
3920          .WORD DUMMY-1,DUMMY-1
3930 ;---------------
3940 ;ASCII TABLE OF NEW TOKEN
3950 ;KEYWORDS/BIT 7 IS SET IN THE LAST
3960 ;LETTER OF EACH KEYWORD/LIST
3970 ;TERMINATED BY ZERO
```

```
3980 ;---------------
3990 ;FUNCTION TOKENS
4000 ;---------------
4010 WORD    .ASC "JO |"
4020         .ASC "FIR¬"
4030         .ASC "XSP_"
4040         .ASC "YSP_"
4050         .ASC "CDUM↖"
4060         .ASC "DDUM↖"
4070         .ASC "EDUM↖"
4080         .ASC "FDUM↖"
4090         .ASC "GDUM↖"
4100         .ASC "HDUM↖"
4110         .ASC "IDUM↖"
4120         .ASC "JDUM↖"
4130         .ASC "KDUM↖"
4140         .ASC "LDUM↖"
4150 ;---------------
4160 ;COMMAND TOKENS
4170 ;---------------
4180         .ASC "RENUMBE_"
4190         .ASC "DELET¬"
4200         .ASC "SEARC |"
4210         .ASC "REPLAC¬"
4220         .ASC "OL¬"
4230         .ASC "AUT┌"
4240         .ASC "IN↗"
4250         .ASC "PUTX |"
4260         .ASC "HIME↖"
4270         .ASC "SPRIT¬"
4280         .ASC "KSPRIT¬"
4290         .ASC "JSPRIT¬"
4300         .ASC "SOUN¬"
4310         .ASC "BITMAP¬"
4320         .ASC "BITMA┐"
4330         .ASC "MOV¬"
4340         .ASC "PLO| "
4350         .ASC "UNPLO| "
4360         .ASC "DRAo"
4370         .ASC "UNDRAo"
4380         .ASC "TEX| "
4390         .ASC "FILL"
4400         .ASC "SCROLL"
4410 ;-----------
```

```
4420          .ASC "DUMMY─"
4430          .ASC "DUMMY┬"
4440          .ASC "DUMMY┴"
4450          .ASC "DUMMY┤"
4460          .ASC "DUMMY├"
4470          .ASC "DUMMY┼"
4480          .ASC "DUMMY╲"
4490 ;-------------
4500          .BYTE $00
4510 ;-------------
```

BASIC Loader: 'Expander' Command Handler

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 32903+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA240,6,234,234,234,76,9,166
112 DATA160,255,202,200,232,189,0,2
113 DATA56,249,93,130,240,245,201,128
114 DATA240,167,166,122,230,11,200,185
115 DATA92,130,16,250,185,93,130,208
116 DATA228,189,0,2,16,149,48,213
117 DATA16,47,201,255,240,43,36,15
118 DATA48,39,201,204,144,29,233,203
119 DATA-9005
120 REM --- BLOCK 2
121 DATA170,132,73,160,255,202,240,8
122 DATA200,185,93,130,16,250,48,245
123 DATA200,185,93,130,48,8,32,71
124 DATA171,208,245,76,36,167,76,239
125 DATA166,76,243,166,32,115,0,201
126 DATA218,176,6,32,121,0,76,231
127 DATA167,32,255,128,76,174,167,233
128 DATA218,10,168,185,34,130,72,185
129 DATA-8485
130 REM --- BLOCK 3
131 DATA33,130,72,76,115,0,169,0
```

```
132 DATA133,13,32,115,0,176,3,76
133 DATA243,188,32,19,177,144,3,76
134 DATA40,175,201,255,208,3,76,158
135 DATA174,201,46,240,234,201,171,208
136 DATA3,76,13,175,201,170,240,218
137 DATA201,180,176,3,76,185,174,10
138 DATA72,170,32,115,0,224,143,176
139 DATA-7649
140 REM --- BLOCK 4
141 DATA3,76,209,175,224,152,144,25
142 DATA224,180,176,21,32,241,174,104
143 DATA168,185,109,129,133,85,185,110
144 DATA129,133,86,32,84,0,76,141
145 DATA173,76,177,175,162,136,189,123
146 DATA165,157,255,127,202,208,247,160
147 DATA0,169,160,132,20,133,21,162
148 DATA31,177,20,145,20,136,208,249
149 DATA-8460
150 REM --- BLOCK 5
151 DATA230,21,202,16,244,169,96,141
152 DATA20,167,162,7,189,206,129,157
153 DATA4,3,202,16,247,169,214,160
154 DATA129,32,30,171,169,0,160,91
155 DATA133,51,133,55,132,52,132,56
156 DATA56,229,45,72,152,229,46,168
157 DATA24,104,105,2,170,152,105,0
158 DATA32,205,189,169,96,160,228,32
159 DATA-7467
160 REM --- BLOCK 6
161 DATA30,171,76,116,164,96,0,128
162 DATA184,128,236,128,14,129,13,32
163 DATA32,32,32,42,42,42,32,84
164 DATA79,79,76,75,73,84,32,49
165 DATA57,56,52,32,74,80,46,71
166 DATA73,66,66,79,78,83,32,42
167 DATA42,42,13,13,32,32,32,32
168 DATA32,32,32,32,0,205,129,205
169 DATA-4422
170 REM --- BLOCK 7
171 DATA129,205,129,205,129,205,129,205
172 DATA129,205,129,205,129,205,129,205
173 DATA129,205,129,205,129,205,129,205
174 DATA129,204,129,204,129,204,129,204
175 DATA129,204,129,204,129,204,129,204
```

```
176 DATA129,204,129,204,129,204,129,204
177 DATA129,204,129,204,129,204,129,204
178 DATA129,204,129,204,129,204,129,204
179 DATA-10668
180 REM --- BLOCK 8
181 DATA129,204,129,204,129,204,129,204
182 DATA129,204,129,204,129,204,129,204
183 DATA129,204,129,204,129,74,79,217
184 DATA70,73,82,197,88,83,80,210
185 DATA89,83,80,210,83,67,79,76
186 DATA204,66,67,79,76,204,69,68
187 DATA85,77,205,70,68,85,77,205
188 DATA71,68,85,77,205,72,68,85
189 DATA-7915
190 REM --- BLOCK 9
191 DATA77,205,73,68,85,77,205,74
192 DATA68,85,77,205,75,68,85,77
193 DATA205,76,68,85,77,205,82,69
194 DATA78,85,77,66,69,210,68,69
195 DATA76,69,84,197,83,69,65,82
196 DATA67,200,82,69,80,76,65,67
197 DATA197,79,76,196,65,85,84,207
198 DATA73,78,203,80,85,84,88,217
199 DATA-6521
200 REM --- BLOCK 10
201 DATA72,73,77,69,205,83,80,82
202 DATA73,84,197,75,83,80,82,73
203 DATA84,197,74,83,80,82,73,84
204 DATA197,83,79,85,78,196,66,73
205 DATA84,77,65,80,195,66,73,84
206 DATA77,65,208,77,79,86,197,80
207 DATA76,79,212,85,78,80,76,79
208 DATA212,68,82,65,215,85,78,68
209 DATA-6333
210 REM --- BLOCK 11
211 DATA82,65,215,84,69,88,212,70
212 DATA73,76,204,83,67,82,79,76
213 DATA204,68,85,77,77,89,196,68
214 DATA85,77,77,89,197,68,85,77
215 DATA77,89,198,68,85,77,77,89
216 DATA199,68,85,77,77,89,200,68
217 DATA85,77,77,89,201,68,85,77
218 DATA77,89,202,0
219 DATA-5994
220 DATA END
```

The Command Handler Routines

Initialisation (Lines 2670–3430)
We deal with this section first as it is the entry point to the routine. SYS 33132 initiates 'Expander' and causes several things to happen. Lines 2670–2890 copy some ROM code down to RAM so that we can alter and use it. The first part of the ROM 'Tokenise' routine is copied down to just below 'Expander' in memory, as this part will be required in our 'Tokenise' wedge. Lines 2750–2890 copy the whole of BASIC into the RAM memory lying 'underneath' the BASIC ROM, creating a RAM copy of BASIC which we can access by switching out the ROM version. Lines 2880–2890 alter the LIST command in this RAM version of BASIC by adding an RTS instruction ($60) at $A714. This is for the benefit of the 'Toolkit' SEARCH and REPLACE commands, and will be covered later on.

'Expander' is actually enabled in lines 2940–2980, by changing the Page Three 'Tokenise', 'Token List', Command Dispatch' and 'Function Evaluator' vectors to the respective 'Expander' values (stored in the VCTRS table at lines 3530–3540). Since 'Expander' is sitting in the BASIC program area it is necessary to protect its code from being overwritten by BASIC. This is done in lines 3090–3140, which lower the top of memory. The top of memory is lowered to $5B00, further than is necessary to protect 'Expander', in order to accomodate a bit-mapped screen for the new PLOT and DRAW commands and to allow space to store sprites. This leaves only 21247 bytes free for BASIC programs. If the hi-res screen is not required, however, this memory may be recovered using the HIMEM command, giving 30719 bytes for BASIC. This will be explained later.

Now all that is required is to print the number of bytes free and exit. This figure is calculated by subtracting the End-of-BASIC pointer from the Top-of-Memory pointer in lines 3200–3320. The result is output to the screen in line 3320 by a very useful ROM routine, located at $BDCD, which outputs as ASCII decimal a two-byte binary number passed to it in X (*low* byte) and A (*high* byte). Finally, we exit via the ROM routine at $A474 which prints READY and returns us to BASIC.

Wedge Routines
These are the first routines given in the listing.

'Tokenise' Wedge (Lines 1130–1390)

Upon initialisation we copied the first #$88 bytes of 'Tokenise' down to RAM location $8000, which is the 'Tokenise' routine as far as step 25. in the description given earlier of the operation of this routine. At step 25. we have got to the end of the ROM keyword table and failed

to find a match with the contents of the input buffer. But now, rather than going on the step 26. which is what would normally happen, control instead passes to our wedge code. This can be thought of as an extension to 'Tokenise' which is tagged on to the end. The extension code compares the contents of the buffer with entries in the 'Expander' keyword table WORD. The wedge code follows a similar methodology to the ROM tokenise routine and should therefore not to be too difficult to follow. Lines 1290 and 1380 are branches back to earlier parts of the routine. Steps 27. onward of the original 'Tokenise' have been replaced with a JMP at line 1180 to the corresponding code in the ROM.

'Token List' Wedge (Lines 1460–1760)

Lines 1460–1580 check for non-tokens, the pi character and the quote condition, and return to the ROM routine if any of these are found. 'Expander' tokens are identified by lines 1520 – 1530 on the simple condition that these will all have the value #$CC or greater. If a standard 64 BASIC token is found then control is passed back to the ROM routine. Lines 1540–1550 subtract #$CB from the token to produce a counter for the loop at lines 1580–1640 which obtains the correct keyword from our keyword table WORD. In lines 1660–1700 the keyword is output to the screen by the ROM routine residing at $AB47, after which we exit back to the ROM code at line 1740.

'Command Dispatch' Wedge (Lines 1820–1990)

This is similar to the corresponding ROM routine, except where (in lines 1830–1870) we identity normal BASIC tokens and pass these back to the ROM code. We use 'Expander' tokens to obtain an index into the two-byte action address table ADDRS. This address is then pushed on to the stack and we exit via CHRGET. Upon returning from the command handler control is passed back to the interpreter loop in line 1900.

'Function Evaluator' Wedge (Lines 2050–2610)

The first step is to initialize the system variable $0D which indicates the type of data being evaluated. This location holds $00 if data is numeric and $FF if the data is string. Next we input a byte from CHRGET at line 2070 and reject non-tokens. If the Carry flag is *clear* on returning from CHRGET it indicates an ASCII *numeral*, which is not a function token, so exit back to the ROM is made at line 2100. The JSR $B113 instruction at line 2120 eliminates ASCII *alphabetic* bytes; lines 2170–2200 reject pi, line 2220 gets rid of decimal fractions and

lines 2240–2270 do the same for negative numbers. A *leading* plus sign (e.g.+1) is ignored (lines 2290–2300) and causes the next byte in the buffer to be fetched. Finally (phew!) we check for function tokens in line 2310. The token value is multiplied by two, and in lines 2450–2480 command tokens and 64 BASIC function tokens are rejected, leaving only 'Expander' function tokens. These are finally used to select the function address in lines 2540–2590 from the action address table FUNCS (set up in lines 3690–3730).

'Expander' Action Tables (Lines 3690–3920)

The Action tables FUNCS (lines 3690–3730) for *functions* and ADDRS (lines 3780–3920) for *commands* are assembled in the BASIC loader version of the 'Command Handler' so that all their entries point to the dummy routine at line 3470 which consists of just an RTS. If these tables were assembled to contain the actual command addresses then an inadvertent call to one of the commands whose handler had not yet been entered would probably have resulted in a system 'crash'. As it is actually arranged it is necessary, for each new command or function added to 'Expander', to perform two POKES to enter the address of the appropriate handling routine into the respective table.

You will notice that there exist, at the end of each table, some extra dummy addresses (eight in the case of functions, and seven in the commands table). These correspond to dummy entries in the 'Expander' keyword table, and allow the user the option of adding further commands and functions to the program. After taking a look at the 'Expander' commands and functions I will demonstrate how you may add your own.

'Expander' Keyword Table (Lines 4011–4500)

This is the ASCII table where the 'expander' keywords are stored. Note that the last letter of each keyword has bit 7 set (i.e. is plus #$80). On the PAL assembler you can represent this by using a shifted character from the keyboard in the .ASC pseudo-op, as in the listing given. Other assemblers may require a .BYTE command for entry of this last character. For example, in the case of JOY the code would need to be:

```
.ASC "JO"
.BYTE $D9
```

The dummy entries facilitate the adding of extra commands by the user. This will be covered later.

6 Extending BASIC: The toolkit utilities

This chapter presents the first set of command routines for use with the 'Expander' program, of which the core 'Command handler' routine was presented in the last chapter. The first command routine listed gives you the INK, PUTXY and HIMEM commands. This is the shortest of the command routines and will provide a quick entry into using the 'Expander' program. The routine which follows for the 'Toolkit' set of commands is rather more extensive and will take some time to enter, so you will be able to see the command handler in action sooner if you take them in the order presented. Of course, you may enter any of the command routines which make up the rest of this book in any order, as they all contribute to the fully fledged 'Expander'.

Entering 'Expander' Command Routines

Assuming you have entered the 'Command handler' section of 'Expander' we need to add some command routines. Each of these is again provided in both assembler source code and BASIC loader program format. The BASIC loader programs all have the same format: Lines 100–108 are the actual loader program which reads the DATA statements, loads their contents into memory and checks for errors; lines 110–999 are DATA statements containing the machine code arranged in blocks of 64 bytes with a *checksum* value at the end of each block. Finally, lines 1000 onwards contain POKEs, two for each new command added to 'Expander'. These POKEs enter the address of the new command (or function) into the correct 'Command handler' action address table.

In every case the procedure needed to incorporate a new command or function into 'Expander' is the same:

1. LOAD the 'Command handler' program into memory
2. LOAD the BASIC loader for the new command(s)
3. RUN the BASIC loader and correct any errors
4. RUN 1000 to link the new command(s) into the 'Command handler'

5. NEW the loader (having first saved it to memory)
6. Use 'Wedge-Mon' or the BASIC machine code SAVE program (provided in the previous chapter) to save Handler, plus the new command routine(s) memory as one program ('Expander')
7. Enter NEW and then SYS 33132 to enable 'Expander'.

If you go through the above procedure and try the new command(s) but get no result then one of several things has happened:

1. (Most likely.) You have set the command parameters incorrectly, e.g. so as to produce an 'invisible' sprite or a sound with zero volume. For each command specific pitfalls of this sort are considered, so read the section corresponding to the command concerned again.

2. (Less likely.) If the new commands cause a BASIC 'warm start', i.e. a result similar to hitting <run/stop> <restore>, or else just simply crash the 64 then the POKEs at lines 1000 onwards in the loader may be entered incorrectly. These are not 'checked' like the machine code in the DATA statements and so it is up to you to get them right!
3. (Most unlikely.) If the POKEs are right then RUN the loader again and check for errors in the DATA statements. If none are flagged then a typing error has managed to bamboozle my checksum error system, e.g. there is an extra spurious zero byte or two errors that cancel each other out, and you will have to check all the DATA the hard way, by hand. Good luck!

Note that in use 'Expander' commands should behave in every way like standard CBM commands *except* in one trivial respect (and I must admit that I don't know why it should be so). The IF ... THEN statement in BASIC normally allows any legal BASIC command to be placed after THEN:

```
10 IF X=0 THEN PRINT "RAT"
```

But for some mysterious reason it does *not* allow any 'Expander' commands in this position. This, however, is completely fixed by simply putting a colon after THEN, before the 'Expander' command:

```
10 IF X=0 THEN: SCROLL
```

The INK, PUTXY and HIMEM Commands

These are three simple commands that require a minimum of typing

to enter but which demonstrate the three stages of execution common to all BASIC commands:

1. Get in the Command parameters
2. Perform an operation on the basis of the parameters
3. Return to the interpreter via an RTS

INK

The INK command changes the text foreground colour to the INK parameter value. This is like typing <control> followed by 1–8 or <commodore> 1–8, except that in the case of INK *all* characters on the screen are changed to the new colour along with the cursor.

Syntax: INK (colour number)

Example: INK 1 changes all the characters on the screen to white and all subsequent printed characters are white.

PUTXY

The PUTXY command allows characters to be PRINTed at any position on the screen given by the (column), (row) command parameters. PUTXY thus eliminates the need for PRINT statements to be cluttered with the innumerable illegible cursor control hieroglyphics often seen in listings.

Syntax: PUTXY (column),(row),(any legal print item)

Example: PUTXY,10,12,"TEST" prints TEST at column 10 and row 12 of the screen. The limits for the parameters are (columns) 0–39 and (row) 0–24, but note that printing on row 24 will cause the screen to scroll up a line.

HIMEM

HIMEM entered without parameters causes the number of bytes left free for BASIC to be printed out. If there is a parameter then the Top-of-Memory pointer is first set to the parameter value, with all necessary BASIC pointers being reset. HIMEM, with or without a parameter, always has the effect of a BASIC CLR command, causing all variables, arrays, user-defined functions and files, etc., to be erased from memory. However, providing that the top of memory is not set to below the end of BASIC any BASIC program in memory will be unaffected.

Syntax: HIMEM (address)

Example: HIMEM 32768 sets the top of memory to 32768 ($8000)

The INK, PUTXY and HIMEM Code

Assembler Listing

```
1000 *=$8A00
1010 ;--------------------
1020 ;--GET IN 'INK' COLOUR--
1030 ;GET COMMA/GET COLOUR IN .X/STORE
1040 ;IN $0286 (SYSTEM VARIABLE -
1050 ;CURRENT CHARACTER COLOUR)
1060 ;--------------------
1070 INKY    JSR $B79E
1080         TXA
1090         AND #$0F
1100         STA $0286
1110 ;--------------------
1120 ;--FILL COLOUR TABLE--
1130 ;COLOUR TABLE IS FILLED WITH 'INK'
1140 ;COLOUR/RETURN TO BASIC
1150 ;--------------------
1160         LDX #$00
1170 TAJ     STA $D800,X
1180         STA $D900,X
1190         STA $DA00,X
1200         STA $DB00,X
1210         DEX
1220         BNE TAJ
1230         RTS
1240 ;--------------------
1250 ;--'PUTXY'COMMAND--
1260 ;--------------------
1270 XPOS    =$0334
1280 YPOS    =$0335
1290 ;--------------------
1300 ;--GET X-POSITION--
1310 ;GET COMMA/GET X PARAMETER IN .X/
1320 ;REJECT IF > #$28
1330 ;--------------------
1340 PRIN    JSR $B79E
1350         CPX #$28
1360         BCS ERREX
1370         STX XPOS
1380 ;--------------------
1390 ;--GET Y-POSITION--
```

```
1400 ;GET COMMA/GET Y PARAMETER/REJECT
1410 ;IF >##$18/GET FINAL COMMA
1420 ;----------------------
1430         JSR $AEFD
1440         JSR $B79E
1450         CPX #$19
1460         BCS ERREX
1470         STX YPOS
1480         JSR $AEFD
1490 ;----------------------
1500 ;--MOVE CURSOR AND JUMP TO ROM--
1510 ;LOAD .Y WITH X-POSITION AND .X
1520 ;WITH Y-POSITION/JSR TO KERNAL
1530 ;'PLOT' ROUTINE TO SET CURSOR/
1540 ;EXIT VIA ROM 'PRINT' ROUTINE
1550 ;----------------------
1560         CLC
1570         LDY XPOS
1580         LDX YPOS
1590         JSR $FFF0
1600         JMP $AA9D
1610 ;------------------------
1620 ;--ERROR EXIT--
1630 ;LOAD .X WITH 'ILLEGAL QUANTITY'
1640 ;CODE AND JUMP TO ROM ERROR
1650 ;ROUTINE
1660 ;------------------------
1670 ERREX   LDX #$0E
1680         JMP ($0300)
1690 ;------------------------
1700 ;--'HIMEM' COMMAND--
1710 ;-SET UPPER LIMIT OF BASIC-
1720 ;GET IN LIMIT/STORE IN $37,$38
1730 ;(HIGHEST ADDR USED BY BASIC)/
1740 ;SUBTRACT FROM THIS THE END-OF
1750 ;BASIC POINTER AND OUTPUT AS
1760 ;BASIC BYTES FREE/IF NO LIMIT
1770 ;PARAMETER THEN JUST OUTPUT NO.
1780 ;OF BYTES FREE/
1790 ;------------------------
1800 HYMEM   BNE CALCIT
1810 ;-----------
1820         LDA $37
1830         LDY $38
```

```
1840          BNE INFORM
1850 ;----------
1860 CALCIT JSR $AD8A
1870          JSR $B7F7
1880          LDA $14
1890          LDY $15
1900          STA $37
1910          STY $38
1920 ;-----------------------
1930 INFORM SEC
1940          SBC $2D
1950          PHA
1960          TYA
1970          SBC $2E
1980          TAY
1990          CLC
2000          PLA
2010          ADC #$02
2020          TAX
2030          TYA
2040          ADC #$00
2050 ;-----------------------
2060 ;$BDCD OUTPUTS A 2 BYTE NUMBER IN
2070 ;.X (HIGH) AND .A (LOW)
2080 ;-----------------------
2090          JSR $BDCD
2100 ;-----------------------
2110 ;$E460 CONTAINS THE 'BASIC BYTES
2120 ;FREE' MESSAGE
2130 ;-----------------------
2140          LDA #$60
2150          LDY #$E4
2160          JSR $AB1E
2170 ;-----------------------
2180 ;EXIT VIA THE BASIC 'CLR' ROUTINE
2190 ;WHICH RESETS THE REST OF THE
2200 ;BASIC POINTERS
2210 ;-----------------------
2220          JMP $A660
2230 ;-----------------------
```

BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 35327+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA32,158,183,138,41,15,141,134
112 DATA2,162,0,157,0,216,157,0
113 DATA217,157,0,218,157,0,219,202
114 DATA208,241,96,32,158,183,224,40
115 DATA176,32,142,52,3,32,253,174
116 DATA32,158,183,224,25,176,19,142
117 DATA53,3,32,253,174,24,172,52
118 DATA3,174,53,3,32,240,255,76
119 DATA-7310
120 REM --- BLOCK 2
121 DATA157,170,162,14,108,0,3,208
122 DATA6,165,55,164,56,208,14,32
123 DATA138,173,32,247,183,165,20,164
124 DATA21,133,55,132,56,56,229,45
125 DATA72,152,229,46,168,24,104,105
126 DATA2,170,152,105,0,32,205,189
127 DATA169,96,160,228,32,30,171,76
128 DATA96,166
129 DATA-6350
130 DATA END
1000 REM -- ENABLE NEW COMMANDS --
1001 AD=33325
1002 REM - INK
1003 POKE AD,255:POKE AD+1,137
1004 REM - PUTXY
1005 POKE AD+2,26:POKE AD+3,138
1006 REM - HIMEM
1007 POKE AD+4,70:POKE AD+5,138
1008 END
```

The next section looks at the code for these three commands and how they are put into effect.

The INK Routine (Lines 1070–1230)

Lines 1070–1100 use the ROM input routine at $B79E to input a byte parameter (literal, variable or expression) in the X register. This is our 'ink' value and in order to change the current text colour we store the input byte value in the system variable at $0286 which holds the current character colour. This will not change the colour of any text already printed, however, and to achieve this we must store the ink colour in the screen colour table, between $D800 and $DBFF. Although this requires that more than 256 bytes of memory be filled it is possible to do it in one loop without resorting to indirect addressing. In lines 1160–1230 the table is filled in four 256-byte blocks. This is a faster method of accessing tables than is utilising indirection, although where the table is larger than about 2K the number of operations required becomes unwieldy, and it is probably then better to write a loop using an indirect pointer or self-modifying code, as we saw earlier in the case of the CHRGET routine.

The PUTXY Routine (Lines 1340–1680)

PUTXY is a specialized form of BASIC's PRINT routine. Lines 1340–1480 are concerned with getting in the X co-ordinate and Y co-ordinate parameters using the ROM routines $B79E and $AEFD. The parameters are rejected if they fall outside the screen limits of #$27 for the X co-ordinate and #$18 for the Y co-ordinate. In lines 1560–1590 these parameters are used to position the cursor on the screen using the Kernal PLOT routine at $FFF0. The PLOT routine is used by the 64 for formatting the screen and can operate in two modes: a call with the Carry *set* returns in the Y and X registers, respectively, the X,Y co-ordinate position of the cursor, while a call with the Carry *clear*, as is the case of our routine, positions the cursor at the X,Y co-ordinates specified by register values Y, X. Finally, we jump into the normal PRINT routine at $AA9D.

The error exit at lines 1670–1680 prints a syntax error message if the parameters are outside the permitted range.

The HIMEM Routine (Lines 1800–2220)

If there is a parameter after the HIMEM command it is brought in by the lines 1860–1910. The ROM routine at $AD8A evaluates an expression or variable and that at $B7F7 converts it to an integer value stored in locations $14,$15. This is in turn stored in the Top-of-BASIC pointer, $37,38. If the command has no parameters then then the Top-of-BASIC pointer is loaded in A and Y, and control passes straight to the

second part of the routine at lines 1930–2040. Here the number of 'bytes free' is calculated from the Start-of-Variables pointer value, less the Top-of-BASIC pointer, minus 2. The result, held in A (low byte) and X (high byte) is passed to the ROM routine at $BDCD, which outputs it to the screen. This routine is used, by BASIC, mainly to output line numbers for the LIST command. Finally, in lines 2140–2220 we use the routine at $AB1E to output the ROM BASIC BYTES FREE message held at $E460 and then exit to BASIC via the CLR routine which tidies up all other system pointers.

'The 'Toolkit' Commands

The 'Toolkit' routines add to BASIC six major commands which could have been included in the standard ROM. They come under the category of programmer's utilities in that they aid the writing of BASIC programs. The commands and their syntax are given below.

RENUMBER

RENUMBER, as you might guess, renumbers BASIC lines and all line references in GOTOs, GOSUBs, etc. It is a two parameter renumber routine allowing both the starting line number and the increment or step to be defined.

Syntax: RENUMBER (step), (start)
where (step)=the renumber increment
and (start)=the starting line number

Example: RENUMBER 5,1000 renumbers in steps of 5 starting at line 1000. Typing RENUMBER with no parameters defaults to step 10 and start 1000.

DELETE

The DELETE command deletes a line or a range of lines from the BASIC text.

Syntax: DELETE (start)-(end)
where (start) is the first line for deletion and (end) the last.

Example: DELETE 1000-2000 deletes all lines between 1000 and 2000, inclusive.

AUTO

The AUTO command enables automatic line-numbering, i.e. each numbered line that is entered will be followed by a new line number equal to the last line number plus the step value.

Syntax: AUTO (step)

Example: AUTO 10. Every numbered line input is followed by a line number equal to that of the input line plus 10. Typing AUTO 0 or AUTO with no parameter disables auto line-numbering.

SEARCH

The SEARCH command searches for a character or string of characters within a BASIC program and LISTs all lines containing the search string.

Syntax: SEARCH /(string)/
where (string)=the search string

Example: SEARCH /PRINT X/ will LIST all program lines containing PRINT X. A line will be printed only once regardless of how many times the search string occurs within the line. Leading and following spaces may be included in the search string.

REPLACE

The REPLACE command replaces a search string with a replacement string wherever it occurs within a BASIC program and LISTs the line(s) concerned.

Syntax: REPLACE (search string)/(replace string)/

Example: REPLACE X/Y/ replaces all occurrences of X in the BASIC text area with Y.

OLD

The OLD command recovers a program NEWed in error.

Syntax: OLD

OLD takes no parameters. It may be used in conjunction with a reset switch or SYS 64738 to 'cold-start' the 64 and yet recover the BASIC program currently in memory.

After incorporation of the routines is accomplished by the procedure laid out in the introductory section of the preceding chapter, you can use any of the 'Toolkit' commands.

Potential Problems
Of the 'Toolkit' commands RENUMBER, DELETE, OLD and AUTO are fairly straightforward and should give no particular problems. The SEARCH and REPLACE commands however may sometimes appear to produce strange results.

This is a consequence of the fact that BASIC keywords are stored as tokens and that the parameter strings for SEARCH and REPLACE are also tokenized. For instance take the following BASIC program:

```
10 PRINT "REM"
20 REM PRINT
30 REM PRINTER PAPER
```

The command SEARCH/PRINT/ would only find the PRINT *token* in line 10 and not the PRINT's in lines 20 and 30 because, since these are within REM statements, they are not in tokenized form. Similarly, SEARCH/REM/ would only find the REMs in lines 20 and 30 and not that in line 10. SEARCH/PRINTER PAPER/ would fail altogether because the SEARCH string parameter would be partly tokenised to (PRINT token)+ER PAPER in ASCII codes and this string does not occur within the program. SEARCH/REM PRINTER PAPER/ would however succeed. It is important to be aware of such possible ambiguities in these commands.

We will now give the 'Toolkit' code listings, and hope that by this stage in the book the size of the listing will not prevent you either keying it in, or working through the code systematically in conjunction with the text that follows. More than half the code is the RENUMBER routine, so not much would be gained by splitting the routine. especially since DELETE uses routines in common. Anyway, here goes!

The 'Toolkit' Code

Assembler Listing

```
1000 *=$8500
1005 ;---------------
1010 WKSPC1 =$19
1015 WKSPC2 =$1A
1020 WKSPC3 =$1B
1025 WKSPC4 =$1C
1030 FLAG   =$1D
1035 ERROR  =$1F
1040 SPACE  =$20
1045 PFLAG  =$21
1050 INCRO  =$4B;*
1055 SRC    =$4E;*
1060 DST    =$50;*
1065 NBUFF  =$52
1070 DIFRNC =$53
```

```
1075 STRT    =$57;*
1080 SIZE    =$59;*
1085 YOFSET =$5B
1090 NOFSET =$5C
1095 CURR    =$FD;*
1100 ;----------
1105 AUTSTP =$03FC;*
1110 ;----------
1115 BUFFER =$0100
1120 BUFF1   =$033C
1125 BUFF2   =$036C
1130 ;---------------
1135 ;--RENUMBER ROUTINE--
1140 ;GET IN START AND INCREMENT
1145 ;PARAMETERS/CHECK LEGALITY
1150 ;-IF ILLEGAL ERROR EXIT
1155 ;---------------
1160 RENUMB BEQ DEFLT1
1165        BCS EXREN
1170        JSR $A96B
1175        LDA $14
1180        LDX $15
1185        STA INCRO
1190        STX INCRO+1
1195        JSR $0079
1200        BEQ DEFLT2
1205        JSR $AEFD
1210        JSR $A96B
1215        LDA $14
1220        LDX $15
1225        STA STRT
1230        STX STRT+1
1235        JMP START
1240 ;----------
1245 EXREN  LDX #$0B
1250        JMP ($0300)
1255 ;----------
1260 DEFLT1 LDA #$0A
1265        LDX #0
1270        STA INCRO
1275        STX INCRO+1
1280 DEFLT2 LDA #$E8
1285        LDX #$03
1290        STA STRT
```

```
1295          STX STRT+1
1300 ;----------
1305 START    LDA INCRO
1310          ORA INCRO+1
1315          BEQ ERR
1320 ;----------
1325          JSR TESTP
1330          BCC OK
1335 ERR      LDA #<ERROR1
1340          LDY #>ERROR1
1345          JMP $AB1E
1350 ;---------------
1355 ;SAVE CHRGET POINTER ON STACK/
1360 ;INITIALIZE ERROR FLAG
1365 ;---------------
1370 OK       LDA $7A
1375          PHA
1380          LDA $7B
1385          PHA
1390 ;----------
1395          LDA #0
1400          STA ERROR
1405 ;---------------
1410 ;--MAIN RENUMBER CONTROL LOOP--
1415 ;SEARCH BASIC TEXT FOR KEYWORD
1420 ;TOKENS TAKING A LINE REFERENCE/
1425 ;RENUMBER LINE REFERENCE/WHEN
1430 ;ALL REFERENCES ARE DONE RENUMBER
1435 ;PROGRAM LINES/CHECK ERROR FLAG-
1440 ;IF SET PRINT ERROR AND EXIT/ELSE
1445 ;RECOVER CHRGET POINTER FROM RAM
1450 ;AND EXIT VIA THE BASIC CLR
1455 ;ROUTINE/(CURR) POINTS TO LINK
1460 ;OF CURRENT TEXT LINE
1465 ;---------------
1470          LDA $2B
1475          LDX $2C
1480 ;----------
1485 R1       STA CURR
1490          STX CURR+1
1495 ;----------
1500 R2       LDY #$01
1505          LDA (CURR),Y
1510          BEQ ROUT
```

```
1515          LDY #$03
1520 ;----------
1525 R3       INY
1530 R4       LDA (CURR),Y
1535          BEQ T5
1540 ;---------------
1545 ;--(TOKEN) COMPARISONS-- ALL THESE
1550 ;KEYWORDS CAN HAVE ASSOCIATED
1555 ;LINE REFERENCES
1560 ;---------------
1565 R5       CMP #$89        ;(GOTO)
1570          BEQ Y1
1575          CMP #$8D        ;(GOSUB)
1580          BEQ Y1
1585          CMP #$9B        ;(LIST)
1590          BEQ Y1
1595          CMP #$8A        ;(RUN)
1600          BEQ Y1
1605          CMP #$A7        ;(THEN)
1610          BEQ Y1
1615 ;----------
1620          CMP #$CB        ;(GO)
1625          BNE R3
1630 ;---------------
1635 ;'GO' TOKEN SPECIAL CASE
1640 ;---------------
1645 JJ       INY
1650          LDA (CURR),Y
1655          BEQ T5
1660          CMP #$20        ;'SPACE
1665          BEQ JJ
1670          CMP #$A4        ;(TO)
1675          BNE R5
1680 ;---------------
1685 ;NOFSET HOLDS POSITION OF NUMBER
1690 ;IN CURRENT TEXT LINE/ALL SPACES
1695 ;BETWEEN KEY WORD AND NUMBER ARE
1700 ;PRESERVED
1705 ;---------------
1710 Y1       LDX #$00
1715          STX FLAG
1720 YS       STY NOFSET
1725          INY
1730          LDA (CURR),Y
```

```
1735          CMP #$20
1740          BEQ YS
1745          DEY
1750 ;---------------
1755 ;ASCII LINE NUMBER BYTES ARE GOT
1760 ;IN AND STORED IN 'BUFFER'/NBUFF
1765 ;HOLDS NO. CHARS IN BUFFER-1/FLAG
1770 ;SET IF ON GOTO OR ON GOSUB
1775 ;---------------
1780 Y2       INY
1785          LDA (CURR),Y
1790          CMP #$20        ;'SPACE
1795          BEQ Y2
1800          CMP #$2C        ;',
1805          BEQ OOT
1810          CMP #$AB        ;(-)
1815          BEQ OOT
1820          CMP #$30        ;'0
1825          BCC OUT
1830          CMP #$3A        ;':
1835          BCS OUT
1840          STA BUFFER,X
1845          INX
1850          BNE Y2
1855 ;----------
1860 OOT      INC FLAG
1865 OUT      TXA
1870          BEQ HUH
1875          LDA #0
1880          STA BUFFER,X
1885          DEX
1890          STX SPACE
1895          STY YOFSET
1900 ;---------------
1905 ;'REPLAC' REPLACES OLD LINE
1910 ;REFERENCE WITH NEW
1915 ;---------------
1920          JSR REPLAC
1925 ;---------------
1930 ;IF 'FLAG' SET THEN GO BACK AND
1935 ;GET REMAINING LINES ELSE GET NEXT
1940 ;KEYWORD TOKEN
1945 ;---------------
1950          LDY YOFSET
```

```
1955 HUH     LDA FLAG
1960         BNE Y1
1965         BEQ R4
1970 ;----------
1975 T5      LDY #$01
1980         LDA (CURR),Y
1985         TAX
1990         DEY
1995         LDA (CURR),Y
2000         JMP R1
2005 ;---------------
2010 ;--ALL REFERENCES RENUMBERED--
2015 ;RENUMBER LINES/IF NO ERROR THEN
2020 ;EXIT VIA 'CLR' ELSE PRINT ERROR
2025 ;AND EXIT
2030 ;---------------
2035 ROUT    JSR RENUM
2040         BIT ERROR
2045         BPL CLEAN
2050         LDA #<ERROR2
2055         LDY #>ERROR2
2060         JSR $AB1E
2065 CLEAN   PLA
2070         STA $7B
2075         PLA
2080         STA $7A
2085         JMP $A663
2090 ;---------------
2095 ;--RENUMBER A LINE REFERENCE--
2100 ;CONVERT ASCII NUMBER IN 'BUFFER'
2105 ;TO BINARY/FIND CORRESPONDING
2110 ;LINE IN TEXT INCREMENTING A
2115 ;COUNTER/CONVERT COUNTER TO ASCII
2120 ;DECIMAL IN BUFFER/EXPAND OR
2125 ;CONTRACT BASIC TEXT AS REQUIRED/
2130 ;REPLACE BUFFER CONTENTS IN TEXT
2135 ;AS NEW LINE REFERENCE/TIDY UP
2140 ;BASIC POINTERS/RETURN
2145 ;---------------
2150 REPLAC JSR ASCBIN
2155         JSR LINCNT
2160         JSR BINASC
2165         JSR EXPO
2170         BCC SKPP
```

```
2175          JSR PUTBAK
2180          JMP TIDY
2185 SKPP     JMP PUTBAK
2190 ;---------------
2195 ;--CONVERT ASCII TO BINARY--
2200 ;ASCII DECIMAL NUMBER IN 'BUFFER'
2205 ;IS CONVERTED TO A 2 BYTE BINARY
2210 ;NO. IN $14,$15 USING CHRGET AND
2215 ;ROM CONVERSION ROUTINE
2220 ;---------------
2225 ASCBIN LDA #<BUFFER
2230          LDX #>BUFFER
2235          STA $7A
2240          STX $7B
2245          JSR $0079
2250          JMP $A96B
2255 ;---------------
2260 ;--CONVERT BINARY TO ASCII--
2265 ;A 2 BYTE BINARY NUMBER IN WKSPC3,
2270 ;WKSPC4 IS  CONVERTED TO ASCII
2275 ;DECIMAL DIGITS IN 'BUFFER' BY
2280 ;SUBTRACTING FROM IT SUCCESSIVELY
2285 ;DECREASING POWERS OF 10/LEADING
2290 ;ZEROES ARE SUPPRESSED/THE NO. OF
2295 ;DIGITS IN THE BUFFER-1 IS STORED
2300 ;IN NBUFF
2305 ;---------------
2310 BINASC LDA #0
2315          STA SRC
2320          LDY #$08
2325 PPW      LDX #0
2330 PJ5      SEC
2335          LDA WKSPC3
2340          SBC TENS,Y
2345          STA SRC+1
2350          LDA WKSPC4
2355          SBC TENS+1,Y
2360          BCC TVAM
2365          STA WKSPC4
2370          LDA SRC+1
2375          STA WKSPC3
2380          INX
2385          BNE PJ5
2390 ;----------
```

```
2395 TVAM    TXA
2400         ORA SRC
2405         BEQ TD4
2410 ;----------
2415         TXA
2420 TD3     LDX SRC
2425         ORA #$30
2430         STA BUFFER,X
2435         INC SRC
2440 ;----------
2445 TD4     DEY
2450         DEY
2455         BMI QIT
2460         CPY #$02
2465         BCS PPW
2470         LDA WKSPC3
2475         BCC TD3
2480 ;----------
2485 QIT     STX NBUFF
2490         RTS
2495 ;---------------
2500 ;--SEARCH BASIC FOR A LINE NO.--
2505 ;THE BINARY LINE REFERENCE IS
2510 ;SEARCHED FOR IN THE BASIC TEXT/
2515 ;WKSPC3,4 IS INITIALIZED TO STRT
2520 ;AND INCREMENTED BY INCRO FOR
2525 ;EVERY TEXT LINE ENCOUNTERED SO
2530 ;THAT ON EXITING WKSPC3,4 IS EQUAL
2535 ;TO THE RENUMBERED REFERENCE/IF
2540 ;WKSPC4 OVERFLOWS THEN THE ERROR
2545 ;FLAG IS SET/THE ENTRY POINT AT
2550 ;'TESTP' IS USED FOR VALIDATION OF
2555 ;INPUT PARAMETERS/IF THE LINE NO.
2560 ;SOUGHT IS NOT FOUND IN THE BASIC
2565 ;TEXT THEN WKSPC3,4 IS SET TO
2570 ;$FFFF
2575 ;---------------
2580 TESTP   LDA #$FF
2585         STA $14
2590         STA $15
2595 ;----------
2600 LINCNT  LDA STRT
2605         LDX STRT+1
2610         STA WKSPC3
```

```
2615          STX WKSPC4
2620 ;----------
2625          LDA $2B
2630          LDX $2C
2635 ;----------
2640 JPG      STA SRC
2645          STX SRC+1
2650          LDY #$01
2655          LDA (SRC),Y
2660          BEQ RUM1
2665          INY
2670          INY
2675          LDA (SRC),Y
2680          CMP $15
2685          BNE AWOL
2690          DEY
2695          LDA (SRC),Y
2700          CMP $14
2705          BEQ RUM2
2710 ;----------
2715 AWOL     CLC
2720          LDA WKSPC3
2725          ADC INCR0
2730          STA WKSPC3
2735          LDA WKSPC4
2740          ADC INCR0+1
2745          STA WKSPC4
2750          BCS RUM3
2755 ;----------
2760          LDY #$01
2765          LDA (SRC),Y
2770          TAX
2775          DEY
2780          LDA (SRC),Y
2785          JMP JPG
2790 ;----------
2795 RUM1     LDA #$FF
2800          STA WKSPC3
2805          STA WKSPC4
2810          STA ERROR
2815 RUM2     CLC
2820 RUM3     RTS
2825 ;---------------
2830 ;--RENUMBER PROGRAM LINES--
```

```
2835 ;ALL LINES ARE RENUMBERED STARTING
2840 ;AT STRT AND IN STEPS OF INCRO
2845 ;---------------
2850 RENUM   LDA STRT
2855         LDX STRT+1
2860         STA WKSPC1
2865         STX WKSPC2
2870 ;----------
2875         LDA $2B
2880         LDX $2C
2885 ;----------
2890 TV      STA CURR
2895         STX CURR+1
2900 ;----------
2905         LDY #$01
2910         LDA (CURR),Y
2915         BEQ OFF
2920 ;----------
2925         INY
2930         INY
2935         LDA WKSPC2
2940         STA (CURR),Y
2945         DEY
2950         LDA WKSPC1
2955         STA (CURR),Y
2960 ;----------
2965         CLC
2970         LDA WKSPC1
2975         ADC INCRO
2980         STA WKSPC1
2985         LDA WKSPC2
2990         ADC INCRO+1
2995         STA WKSPC2
3000 ;----------
3005         DEY
3010         LDA (CURR),Y
3015         TAX
3020         DEY
3025         LDA (CURR),Y
3030         JMP TV
3035 ;----------
3040 OFF     RTS
3045 ;---------------
3050 ;--REPLACE LINE REFERENCE--
```

```
3055 ;THE ADJUSTED LINE REFERENCE IS
3060 ;REPLACED AFTER THE KEYWORD TOKEN
3065 ;---------------
3070 PUTBAK LDX #$FF
3075        LDY NOFSET
3080 PB1    INY
3085        INX
3090        LDA BUFFER,X
3095        STA (CURR),Y
3100        DEC NBUFF
3105        BPL PB1
3110        RTS
3115 ;---------------
3120 ;--TIDY UP BASIC--
3125 ;ROM ROUTINE $A533 REBUILDS ALL
3130 ;LINKS IN TEXT/NEW END-OF-BASIC
3135 ;POINTER SET
3140 ;---------------
3145 TIDY   JSR $A533
3150        CLC
3155        LDA $22
3160        ADC #$02
3165        STA $2D
3170        LDA $23
3175        ADC #0
3180        STA $2E
3185        RTS
3190 ;---------------
3195 ;--EXPAND OR CONTRACT BASIC--
3200 ;SUBTRACT THE LENGTH OF NEW LINE
3205 ;REFERENCE FROM SPACE AVAILABLE
3210 ;IN TEXT - IF ZERO THEN RETURN
3215 ;CARRY CLEAR/ THE
3220 ;DIFFERENCE IS STORED IN DIFRNC/
3225 ;CALCULATE START AND SIZE OF BLOCK
3230 ;TO BE MOVED/IF 'DIFFERENCE' IS
3235 ;POSITIVE THEN JUMP TO SHRINK
3240 ;ROUTINE ELSE EXPAND
3245 ;--------------------
3250 EXPO   SEC
3255        LDA SPACE
3260        SBC NBUFF
3265        BNE X1
3270        CLC
```

```
3275          RTS
3280 ;----------
3285 X1       STA DIFRNC
3290 ;----------
3295          CLC
3300          LDA CURR
3305          ADC YOFSET
3310          STA SRC
3315          LDA CURR+1
3320          ADC #0
3325          STA SRC+1
3330 ;----------
3335 ;ENTRY POINT FOR MEMORY MOVE USED
3340 ;BY DELETE COMMAND
3345 ;----------
3350 R33      SEC
3355          LDA $2D
3360          SBC SRC
3365          STA SIZE
3370          LDA $2E
3375          SBC SRC+1
3380          STA SIZE+1
3385 ;----------
3390          LDA SIZE
3395          BNE R44
3400          DEC SIZE+1
3405 R44      DEC SIZE
3410 ;----------
3415          SEC
3420          LDA YOFSET
3425          SBC DIFRNC
3430          STA YOFSET
3435 ;----------
3440          LDA DIFRNC
3445          BEQ SHRNK2
3450          BPL SHRINK
3455 ;---------------
3460 ;--EXPAND BASIC TEXT--
3465 ;CALCULATE SOURCE AND DESTINATION
3470 ;ADDRESSES IN SRC AND DST/
3475 ;EXPAND BASIC TEXT TO ACCOMODATE
3480 ;LARGER LINE REFERENCE/RETURN
3485 ;CARRY SET
3490 ;---------------
```

```
3495 EXPAND CLC
3500        LDA SIZE+1
3505        ADC SRC+1
3510        STA SRC+1
3515 ;----------
3520        CLC
3525        LDA DIFRNC
3530        EOR #$FF
3535        ADC #$01
3540        STA DIFRNC
3545 ;----------
3550        CLC
3555        ADC SRC
3560        STA DST
3565        LDA SRC+1
3570        ADC #$00
3575        STA DST+1
3580 ;----------
3585        INC SIZE+1
3590        LDX SIZE+1
3595        LDY SIZE
3600        BEQ NEXXT2
3605 ;----------
3610 NEXXT1 LDA (SRC),Y
3615        STA (DST),Y
3620        DEY
3625        BNE NEXXT1
3630 NEXXT2 LDA (SRC),Y
3635        STA (DST),Y
3640        DEY
3645        DEC SRC+1
3650        DEC DST+1
3655        DEX
3660        BNE NEXXT1
3665 ;----------
3670 CONCLD SEC
3675        RTS
3680 ;---------------
3685 ;--SHRINK BASIC TEXT--
3690 ;CALCULATE SOURCE AND DESTINATION
3695 ;ADDRESSES IN SRC AND DST/
3700 ;SHRINK BASIC TEXT WHERE NEW LINE
3705 ;REFERENCE IS SMALLER THAN OLD/
3710 ;RETURN CARRY SET
```

```
3715 ;---------------
3720 SHRINK SEC
3725        LDA SRC
3730        SBC DIFRNC
3735        STA DST
3740        LDA SRC+1
3745        SBC #$00
3750        STA DST+1
3755 ;----------
3760 SHRNK2 LDY #0
3765        LDX SIZE+1
3770        BEQ ENT
3775 ;----------
3780 N2     LDA (SRC),Y
3785        STA (DST),Y
3790        INY
3795        BNE N2
3800        INC SRC+1
3805        INC DST+1
3810        DEX
3815        BNE N2
3820 ;----------
3825 ENT    LDX SIZE
3830        BEQ CNCLUD
3835 ;----------
3840        INX
3845 N3     LDA (SRC),Y
3850        STA (DST),Y
3855        INY
3860        DEX
3865        BNE N3
3870 ;----------
3875 CNCLUD SEC
3880        RTS
3885 ;---------------
3890 ;--DELETE COMMAND--
3895 ;GET IN PARAMETERS/USE ROM ROUTINE
3900 ;$A613 TO FIND ADDRESSES OF LINES
3905 ;IN TEXT/DEFINE AS SOURCE AND
3910 ;DESTINATION FOR RENUMBER SHRINK
3915 ;ROUTINE/USE LATTER TO 'CLOSE UP'
3920 ;TEXT AROUND DELETE RANGE/TIDY
3925 ;BASIC POINTERS/EXIT TO BASIC VIA
3930 ;ROM 'CLR' ROUTINE
```

```
3935 ;---------------
3940 DELET  BEQ EXDEL
3945        BCC EGG
3950        CMP ##$AB          ;(-)
3955        BNE EXDEL
3960 ;----------
3965 EGG    JSR $A96B
3970        JSR $A613
3975        LDA $5F
3980        LDY $60
3985        STA DST
3990        STY DST+1
3995 ;----------
4000        JSR $0079
4005        BEQ ONEL
4010 ;----------
4015        CMP ##$AB          ;(-)
4020        BNE EXDEL
4025        JSR $0073
4030        JSR $A96B
4035        BNE EXDEL
4040        LDA $14
4045        ORA $15
4050        BNE SPEC
4055 ;----------
4060        SEC
4065        LDA $2D
4070        SBC ##$02
4075        STA SRC
4080        LDA $2E
4085        SBC #0
4090        STA SRC+1
4095        BNE DOIT
4100 ;----------
4105 SPEC   JSR $A613
4110        BCC NLN
4115 ;----------
4120 ONEL   LDY #0
4125        LDA ($5F),Y
4130        STA SRC
4135        INY
4140        LDA ($5F),Y
4145        STA SRC+1
4150        BEQ DOIT3
```

```
4155          BCS DOIT
4160 ;----------
4165 NLN      LDA $5F
4170          LDY $60
4175          STA SRC
4180          STY SRC+1
4185 ;----------
4190 DOIT     LDA SRC+1
4195          CMP DST+1
4200          BCC EXDEL
4205          BNE DOIT2
4210          LDA SRC
4215          CMP DST
4220          BCC EXDEL
4225 ;----------
4230 DOIT2    LDA #0
4235          STA DIFRNC
4240          JSR R33
4245          JSR TIDY
4250 DOIT3    JMP $A633
4255 ;----------
4260 EXDEL    LDX #$0B
4265          JMP ($0300)
4270 ;---------------
4275 ;--REPLACE AND SEARCH COMMANDS--
4280 ;BIT 7 OF FLAG IS SET FOR REPLACE
4285 ;AND CLEAR FOR SEARCH/GET SEARCH
4290 ;AND REPLACE STRINGS INTO BUFF1
4295 ;AND (FOR REPLACE) BUFF2 USING
4300 ;CHRGET WHICH IS FIRST ALTERED TO
4305 ;ACCEPT SPACES/RESTORE CHRGET/
4310 ;SEARCH BASIC TEXT FOR SEARCH
4315 ;STRING/IF REPLACE THEN
4320 ;SUBSTITUTE REPLACE STRING/FLAG
4325 ;LINE CONTAINING STRING BY USE
4330 ;OF LIST ROUTINE COPIED TO RAM
4335 ;AT SETUP/RETURN TO BASIC VIA
4340 ;CHRGET
4345 ;---------------
4350 REPLC    PHP
4355          LDY #$FF
4360          BMI COMM1
4365 ;---------------
4370 SERCH    PHP
```

```
4375          LDY #0
4380 ;---------------
4385 COMM1    STY FLAG
4390 ;---------------
4395 ;ALTER CHRGET AND GET SEARCH
4400 ;STRING IN BUFF1
4405 ;---------------
4410          PLP
4415          BEQ EXSER1
4420          BCC EXSER1
4425          CMP #$AD
4430          BNE EXSER1
4435 ;----------
4440          LDA #$01
4445          STA $81
4450 ;----------
4455          LDX #$FF
4460 PET      JSR $0073
4465          BEQ EXSER1
4470          CMP #$AD
4475          BEQ STG2
4480          CMP #$2F
4485          BEQ STG2
4490          INX
4495          STA BUFF1,X
4500          JMP PET
4505 ;----------
4510 STG2     INX
4515          BEQ EXSER1
4520          STX SPACE
4525          BIT FLAG
4530          BPL NEXXT
4535 ;---------------
4540 ;IF REPLACE COMMAND GET REPLACE
4545 ;STRING
4550 ;---------------
4555          LDX #$FF
4560 TET      JSR $0073
4565          BEQ EXSER1
4570          CMP #$AD
4575          BEQ STG3
4580          CMP #$2F
4585          BEQ STG3
4590          INX
```

```
4595          STA BUFF2,X
4600          JMP TET
4605 ;----------
4610 STG3     INX
4615          BEQ EXSER1
4620          STX NBUFF
4625          BNE NEXXT
4630 ;---------------
4635 ;--EXIT ROUTINES--
4640 ;EXSER1 IS THE ERROR EXIT/CHRGET
4645 ;IS REPAIRED BEFORE EXIT
4650 ;---------------
4655 EXSER1 LDA #$20
4660          STA $81
4665          LDX #$0B
4670          JMP ($0300)
4675 ;----------
4680 EXSER2 LDA PFLAG
4685          BEQ EXSER3
4690          JSR $A613
4695          JSR LISTIT
4700 EXSER3 JMP $0073
4705 ;----------
4710 NEXXT    LDA #$20
4715          STA $81
4720 ;----------
4725          LDA #0
4730          STA PFLAG
4735 ;----------
4740          LDX $2B
4745          LDA $2C
4750 ;----------
4755 INTRA    STX CURR
4760          STA CURR+1
4765 ;----------
4770          LDY #$01
4775          LDA (CURR),Y
4780          BEQ EXSER2
4785          INY
4790          LDA (CURR),Y
4795          STA STRT
4800          INY
4805          LDA (CURR),Y
4810          STA STRT+1
```

```
4815 ;---------------
4820 ;--SEARCH STRING COMPARISON LOOP--
4825 ;---------------
4830 LOOPY1 LDX #0
4835 LOOPY2 INY
4840        LDA (CURR),Y
4845        BEQ TA
4850        CMP BUFF1,X
4855        BEQ LOOPY3
4860        CPX #0
4865        BEQ LOOPY2
4870        LDY WKSPC1
4875        LDX #0
4880        BEQ LOOPY2
4885 LOOPY3 CPX #0
4890        BNE LOOPY4
4895        STY WKSPC1
4900 LOOPY4 INX
4905        CPX SPACE
4910        BNE LOOPY2
4915 ;----------
4920        STY YOFSET
4925        BIT FLAG
4930        BPL COMMON
4935 ;---------------
4940 ;SWITCH SWITCHES REPLACE FOR
4945 ;SEARCH STRING PERFORMING TEXT
4950 ;MOVES AS REQUIRED
4955 ;---------------
4960        JSR SWITCH
4965 ;----------
4970 COMMON LDA PFLAG
4975        BNE COMM3
4980        INC PFLAG
4985        BNE COMM5
4990 ;---------------
4995 ;IF CURRENT LINE CONTAINING SEARCH
5000 ;STRING IS NOT EQUAL TO 'OLD' LINE
5005 ;THEN 'OLD' LINE IS LISTED
5010 ;---------------
5015 COMM3  LDA STRT
5020        CMP $14
5025        BNE COMM4
5030        LDA STRT+1
```

```
5035          CMP $15
5040          BEQ COMM6
5045 ;---------------
5050 ;LIST A SINGLE LINE USING ROM LIST
5055 ;ROUTINE RESIDING IN RAM/'CURRENT'
5060 ;LINE BECOMES 'OLD' LINE
5065 ;---------------
5070 COMM4    JSR $A613
5075          JSR LISTIT
5080          LDA #$91
5085          JSR $FFD2
5090 COMM5    LDA STRT
5095          LDX STRT+1
5100          STA $14
5105          STX $15
5110 COMM6    LDY YOFSET
5115          BNE LOOPY1
5120 ;----------
5125 TA       LDY #0
5130          LDA (CURR),Y
5135          TAX
5140          INY
5145          LDA (CURR),Y
5150          JMP INTRA
5155 ;---------------
5160 ;--SWITCH REPLACE FOR SEARCH--
5165 ;PERFORM ANY REQ. TEXT MOVE/TIDY
5170 ;BASIC/INSERT CONTENTS OF BUFF2
5175 ;---------------
5180 SWITCH   INC YOFSET
5185          JSR EXPO
5190          DEC YOFSET
5195          BCC PUB1
5200          JSR PUB1
5205          JMP TIDY
5210 PUB1     LDY WKSPC1
5215          LDX #0
5220 PUB4     LDA BUFF2,X
5225          STA (CURR),Y
5230          INY
5235          INX
5240          CPX NBUFF
5245          BNE PUB4
5250          RTS
```

```
5255 ;---------------
5260 ;--MODIFIED BASIC LIST--
5265 ;SWITCH OUT BASIC ROM/CALL AMENDED
5270 ;LIST ROUTINE TO LIST A LINE
5275 ;---------------
5280 LISTIT LDA #$36
5285        STA $01
5290        JSR $A6BD
5295        LDA #$37
5300        STA $01
5305        RTS
5310 ;---------------
5315 ;--OLD COMMAND--
5320 ;RESET START-OF-BASIC POINTER/
5325 ;MAKE NON-ZERO HIGH BYTE OF FIRST
5330 ;LINK/JSR TO RENUMBER TIDY ROUTINE
5335 ; TO REBUILD LINKS AND RESET END-
5340 ;OF-BASIC POINTER/EXIT VIA ROM
5345 ;'CLR' ROUTINE
5350 ;---------------
5355 OLDE   BNE EXOLD
5360        LDY #$01
5365        LDX #$08
5370        STY $2B
5375        STX $2C
5380        TYA
5385        STA ($2B),Y
5390        JSR TIDY
5395        JMP $A660
5400 ;----------
5405 EXOLD  RTS
5410 ;---------------
5415 ;--AUTO COMMAND--
5420 ;GET IN 'STEP' PARAMETER/STORE IN
5425 ;AUTSTP/CHANGE BASIC WARM-START
5430 ;RAM VECTOR TO POINT TO WARMST/
5435 ;EXIT/IF NO PARAMETER OR STEP = 0
5440 ;THEN CANCEL AUTO BY RESETTING
5445 ;VECTOR TO DEFAULT VALUE/
5450 ;---------------
5455 AUT    BEQ CANCL
5460        JSR $A96B
5465        LDA $14
5470        LDX $15
```

```
5475          STA AUTSTP
5480          STX AUTSTP+1
5485          ORA $15
5490          BEQ CANCL
5495 ;----------
5500          LDA #<WRMST
5505          LDX #>WRMST
5510          STA $0302
5515          STX $0303
5520          RTS
5525 ;----------
5530 CANCL    LDA #$83
5535          LDX #$A4
5540          STA $0302
5545          STX $0303
5550          RTS
5555 ;---------------
5560 ;--CHECK FOR LINE NOS. IN BUFFER--
5565 ;WARMST REPLACES ROM LINE INPUT
5570 ;LOOP BUT INCLUDES A CHECK FOR
5575 ;LINE NUMBERS IN INPUT BUFFER/IF
5580 ;LINE NUMBER PRESENT RECOVER IT IN
5585 ;$14,15 AND ADD STEP/CONVERT
5590 ;RESULT TO ASCII DECIMAL USING
5595 ;RENUMBER BINASC ROUTINE/STORE IN
5600 ;KEYBOARD BUFFER PRECEDED BY A
5605 ;SPACE/RETURN TO ROM LINE INPUT
5610 ;LOOP TO RESUME PROCESSING LINE
5615 ;---------------
5620 WRMST    JSR $A560
5625          STX $7A
5630          STY $7B
5635          JSR $0073
5640          BCC EXAM
5645          JMP $A48D
5650 ;----------
5655 EXAM     PHP
5660          STX YOFSET
5665          STY NOFSET
5670          JSR $A96B
5675          BEQ VAMOO
5680          CLC
5685          LDA $14
5690          ADC AUTSTP
```

```
5695          STA WKSPC3
5700          LDA $15
5705          ADC AUTSTP+1
5710          STA WKSPC4
5715          JSR BINASC
5720          INX
5725          INX
5730          STX $C6
5735          LDA #$20
5740          STA $0277,X
5745          DEX
5750          STA $0277,X
5755 DX1      DEX
5760          BMI VAMOO
5765          LDA BUFFER,X
5770          STA $0277,X
5775          BNE DX1
5780 ;----------
5785 VAMOO    LDX YOFSET
5790          STX $7A
5795          LDY NOFSET
5800          STY $7B
5805          PLP
5810          JMP $A48A
5815 ;---------------
5820 ;POWERS OF TEN FOR RENUMBER
5825 ;BINASC ROUTINE
5830 ;---------------
5835 TENS     .BYTE $01,$00,$0A,$00
5840          .BYTE $64,$00,$E8,$03
5845          .BYTE $10,$27
5850 ;---------------
5855 ;MESSAGES FOR PRINTING BY
5860 ;ROM ROUTINE $AB1E
5865 ;---------------
5870 ERROR1   .BYTE $0D
5875          .ASC "ILLEGAL PARAMETERS"
5880          .BYTE $0D,$00
5885 ;---------------
5890 ERROR2   .BYTE $0D
5895          .ASC "CAUTION-BAD LINES!"
5900          .BYTE $0D,$00
5905 ;---------------
```

'Toolkit' BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 34047+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA240,40,176,33,32,107,169,165
112 DATA20,166,21,133,75,134,76,32
113 DATA121,0,240,30,32,253,174,32
114 DATA107,169,165,20,166,21,133,87
115 DATA134,88,76,58,133,162,11,108
116 DATA0,3,169,10,162,0,133,75
117 DATA134,76,169,232,162,3,133,87
118 DATA134,88,165,75,5,76,240,5
119 DATA-6475
120 REM --- BLOCK 2
121 DATA32,90,134,144,7,169,199,160
122 DATA137,76,30,171,165,122,72,165
123 DATA123,72,169,0,133,31,165,43
124 DATA166,44,133,253,134,254,160,1
125 DATA177,253,240,124,160,3,200,177
126 DATA253,240,106,201,137,240,33,201
127 DATA141,240,29,201,155,240,25,201
128 DATA138,240,21,201,167,240,17,201
129 DATA-8956
130 REM --- BLOCK 3
131 DATA203,208,227,200,177,253,240,77
132 DATA201,32,240,247,201,164,208,219
133 DATA162,0,134,29,132,92,200,177
134 DATA253,201,32,240,247,136,200,177
135 DATA253,201,32,240,249,201,44,240
136 DATA18,201,171,240,14,201,48,144
137 DATA12,201,58,176,8,157,0,1
138 DATA232,208,227,230,29,138,240,15
139 DATA-9938
140 REM --- BLOCK 4
```

```
141 DATA169,0,157,0,1,202,134,32
142 DATA132,91,32,247,133,164,91,165
143 DATA29,208,189,240,146,160,1,177
144 DATA253,170,136,177,253,76,90,133
145 DATA32,169,134,36,31,16,7,169
146 DATA220,160,137,32,30,171,104,133
147 DATA123,104,133,122,76,99,166,32
148 DATA14,134,32,96,134,32,28,134
149 DATA-7228
150 REM --- BLOCK 5
151 DATA32,3,135,144,6,32,226,134
152 DATA76,242,134,76,226,134,169,0
153 DATA162,1,133,122,134,123,32,121
154 DATA0,76,107,169,169,0,133,78
155 DATA160,8,162,0,56,165,27,249
156 DATA189,137,133,79,165,28,249,190
157 DATA137,144,9,133,28,165,79,133
158 DATA27,232,208,232,138,5,78,240
159 DATA-7314
160 REM --- BLOCK 6
161 DATA10,138,166,78,9,48,157,0
162 DATA1,230,78,136,136,48,8,192
163 DATA2,176,207,165,27,144,235,134
164 DATA82,96,169,255,133,20,133,21
165 DATA165,87,166,88,133,27,134,28
166 DATA165,43,166,44,133,78,134,79
167 DATA160,1,177,78,240,41,200,200
168 DATA177,78,197,21,208,7,136,177
169 DATA-7202
170 REM --- BLOCK 7
171 DATA78,197,20,240,34,24,165,27
172 DATA101,75,133,27,165,28,101,76
173 DATA133,28,176,20,160,1,177,78
174 DATA170,136,177,78,76,108,134,169
175 DATA255,133,27,133,28,133,31,24
176 DATA96,165,87,166,88,133,25,134
177 DATA26,165,43,166,44,133,253,134
178 DATA254,160,1,177,253,240,34,200
179 DATA-7253
180 REM --- BLOCK 8
181 DATA200,165,26,145,253,136,165,25
182 DATA145,253,24,165,25,101,75,133
183 DATA25,165,26,101,76,133,26,136
184 DATA177,253,170,136,177,253,76,181
```

```
185 DATA134,96,162,255,164,92,200,232
186 DATA189,0,1,145,253,198,82,16
187 DATA245,96,32,51,165,24,165,34
188 DATA105,2,133,45,165,35,105,0
189 DATA-7768
190 REM --- BLOCK 9
191 DATA133,46,96,56,165,32,229,82
192 DATA208,2,24,96,133,83,24,165
193 DATA253,101,91,133,78,165,254,105
194 DATA0,133,79,56,165,45,229,78
195 DATA133,89,165,46,229,79,133,90
196 DATA165,89,208,2,198,90,198,89
197 DATA56,165,91,229,83,133,91,165
198 DATA83,240,71,16,56,24,165,90
199 DATA-7300
200 REM --- BLOCK 10
201 DATA101,79,133,79,24,165,83,73
202 DATA255,105,1,133,83,24,101,78
203 DATA133,80,165,79,105,0,133,81
204 DATA230,90,166,90,164,89,240,7
205 DATA177,78,145,80,136,208,249,177
206 DATA78,145,80,136,198,79,198,81
207 DATA202,208,237,56,96,56,165,78
208 DATA229,83,133,80,165,79,233,0
209 DATA-7714
210 REM --- BLOCK 11
211 DATA133,81,160,0,166,90,240,14
212 DATA177,78,145,80,200,208,249,230
213 DATA79,230,81,202,208,242,166,89
214 DATA240,9,232,177,78,145,80,200
215 DATA202,208,248,56,96,240,113,144
216 DATA4,201,171,208,107,32,107,169
217 DATA32,19,166,165,95,164,96,133
218 DATA80,132,81,32,121,0,240,38
219 DATA-8609
220 REM --- BLOCK 12
221 DATA201,171,208,84,32,115,0,32
222 DATA107,169,208,76,165,20,5,21
223 DATA208,15,56,165,45,233,2,133
224 DATA78,165,46,233,0,133,79,208
225 DATA28,32,19,166,144,15,160,0
226 DATA177,95,133,78,200,177,95,133
227 DATA79,240,34,176,8,165,95,164
228 DATA96,133,78,132,79,165,79,197
```

```
229 DATA-6985
230 REM --- BLOCK 13
231 DATA81,144,21,208,6,165,78,197
232 DATA80,144,13,169,0,133,83,32
233 DATA27,135,32,242,134,76,51,166
234 DATA162,11,108,0,3,8,160,255
235 DATA48,3,8,160,0,132,29,40
236 DATA240,70,144,68,201,173,208,64
237 DATA169,1,133,129,162,255,32,115
238 DATA0,240,53,201,173,240,11,201
239 DATA-6827
240 REM --- BLOCK 14
241 DATA47,240,7,232,157,60,3,76
242 DATA54,136,232,240,35,134,32,36
243 DATA29,16,51,162,255,32,115,0
244 DATA240,22,201,173,240,11,201,47
245 DATA240,7,232,157,108,3,76,85
246 DATA136,232,240,4,134,82,208,22
247 DATA169,32,133,129,162,11,108,0
248 DATA3,165,33,240,6,32,19,166
249 DATA-6890
250 REM --- BLOCK 15
251 DATA32,37,137,76,115,0,169,32
252 DATA133,129,169,0,133,33,166,43
253 DATA165,44,134,253,133,254,160,1
254 DATA177,253,240,221,200,177,253,133
255 DATA87,200,177,253,133,88,162,0
256 DATA200,177,253,240,78,221,60,3
257 DATA240,10,224,0,240,242,164,25
258 DATA162,0,240,236,224,0,208,2
259 DATA-8751
260 REM --- BLOCK 16
261 DATA132,25,232,228,32,208,225,132
262 DATA91,36,29,16,3,32,6,137
263 DATA165,33,208,4,230,33,208,23
264 DATA165,87,197,20,208,6,165,88
265 DATA197,21,240,19,32,19,166,32
266 DATA37,137,169,145,32,210,255,165
267 DATA87,166,88,133,20,134,21,164
268 DATA91,208,171,160,0,177,253,170
269 DATA-7323
270 REM --- BLOCK 17
271 DATA200,177,253,76,146,136,230,91
272 DATA32,3,135,198,91,144,6,32
```

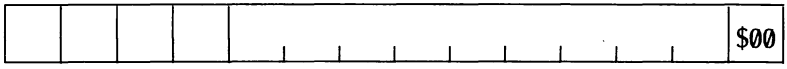
```
273 DATA21,137,76,242,134,164,25,162
274 DATA0,189,108,3,145,253,200,232
275 DATA228,82,208,245,96,169,54,133
276 DATA1,32,189,166,169,55,133,1
277 DATA96,208,17,160,1,162,8,132
278 DATA43,134,44,152,145,43,32,242
279 DATA-7621
280 REM --- BLOCK 18
281 DATA134,76,96,166,96,240,28,32
282 DATA107,169,165,20,166,21,141,252
283 DATA3,142,253,3,5,21,240,11
284 DATA169,110,162,137,141,2,3,142
285 DATA3,3,96,169,131,162,164,141
286 DATA2,3,142,3,3,96,32,96
287 DATA165,134,122,132,123,32,115,0
288 DATA144,3,76,141,164,8,134,91
289 DATA-6283
290 REM --- BLOCK 19
291 DATA132,92,32,107,169,240,42,24
292 DATA165,20,109,252,3,133,27,165
293 DATA21,109,253,3,133,28,32,28
294 DATA134,232,232,134,198,169,32,157
295 DATA119,2,202,157,119,2,202,48
296 DATA8,189,0,1,157,119,2,208
297 DATA245,166,91,134,122,164,92,132
298 DATA123,40,76,138,164,1,0,10
299 DATA-6840
300 REM --- BLOCK 20
301 DATA0,100,0,232,3,16,39,13
302 DATA73,76,76,69,71,65,76,32
303 DATA80,65,82,65,77,69,84,69
304 DATA82,83,13,0,13,67,65,85
305 DATA84,73,79,78,45,66,65,68
306 DATA32,76,73,78,69,83,33,13
307 DATA0
308 DATA-2955
309 DATA END
1000 REM -- ENABLE NEW COMMANDS --
1001 AD=33313
1002 REM - RENUMBER
1003 POKE AD,255:POKE AD+1,132
1004 REM - DELETE
1005 POKE AD+2,164:POKE AD+3,135
1006 REM - SEARCH
```

```
1007 POKE AD+4,33:POKE AD+5,136
1008 REM - REPLACE
1009 POKE AD+6,28:POKE AD+7,136
1010 REM - OLD
1011 POKE AD+8,48:POKE AD+9,137
1012 REM - AUTO
1013 POKE AD+10,68:POKE AD+11,137
1014 END
```

The RENUMBER Routine

The RENUMBER routine is the largest section of the 'Toolkit' commands code, and is also the most complex. In order to appreciate how the renumber routine works it is obviously necessary to first know how BASIC programs are stored in memory.

A BASIC program is composed of lines, each consisting of binary line numbers, tokenised keywords and ASCII characters arranged in what is known as a *linked list* structure. That is, each BASIC line is preceded by a two-byte binary *link address* which points to the address in memory of the next line in the list. The end of a line is indicated by a zero byte and the end of the program by a zero link address. Each line has a structure like this:



LINK ADDRESS | LINE NUMBER | TOKENS/ASCII BYTES OF LINE | END OF LINE

Apart from the storage of text lines BASIC also requires RAM space for the storage of program variables. Numeric variables are stored immediately after the BASIC text, working upwards, with array variables stored above simple variables. String variables are placed at the top of memory and work downwards. Several Zero-page pointers are required to keep track of all this stored data:

Pointer	Stores
$2B,2C	Start of BASIC Text
$2D,$2E	Start of BASIC Variables
$2F,$30	Start of BASIC Arrays
$31,$32	End of BASIC Arrays
$33,$34	Bottom of String Storage
$37,$38	Top of BASIC Memory

By altering the values of the pointers it is possible to protect areas of memory from being overwritten by BASIC. We have already seen an example of this in lines 3090–3140 of the 'Command Handler'. These ideas may become clear if we take a look at a concrete example, such as this BASIC program:

```
10 GET A$:IF A$="" THEN 10
20 PRINT "RAT";
30 PRINT "TRAP"
40 END
```

This is stored in memory as:

```
0801  17  08                          - Link to line 20 ($0817)
0803  0A  00                          - Line number for line 10
0805  A1                             - Get token
0806  20  41  24                      - ASCII ' A$'
0809  3A                             - ASCII ':'
080A  8B                             - IF token
080B  20  41  24                      - ASCII ' A$'
080E  B2                             - = token
080F  22  22  20                      - ASCII '"" '
0812  A7                             - THEN token
0813  20  31  30                      - ASCII ' 10'
0816  00                             - End of line 10
0817  25  08                          - Link to line 30 ($0825)
0819  14  00                          - Line number for line 20
081B  99                             - PRINT token
081C  20  22  52  41  54  20  22  3B  - ASCII ' "RAT ";'
0824  00                             - End of line 20
0825  30  08                          - Link to line 40
0826  1E  00                          - Line number for line 30
0827  99                             - PRINT token
0828  20  22  54  52  41  50  22      - ASCII ' "TRAP" '
082F  00                             - End of line 30
0830  36  08                          - Link to end of program
                                       ($0836)
0832  28  00                          - Line number for line 40
0834  80                             - END token
0835  00                             - End of line 40
0836  00  00                          - End of program
```

The values for the BASIC text pointers will thus be:

Start of BASIC	– $0801
Start of Variables	– $0838

Designing the Renumber Routine

You may have noticed above that, while BASIC line numbers are stored as two binary bytes and are thus of constant length, line references following THEN, GOTO and GOSUB statements are ASCII decimal digits and are therefore of variable length, depending on the size of the line number. For instance, the line reference in GOTO 10 occupies two bytes whereas that in GOTO 1000 requires four.

This fact complicates considerably the writing of a full renumber routine for the 64 because renumbering line references may require that BASIC text be expanded or contracted depending on whether the new (renumbered) line reference is longer or shorter than that which it replaces. The actual renumbering of line *numbers* themselves is in comparison quite a simple procedure and we can safely leave that job until last.

The RENUMBER Algorithm

The sequence of operations for the RENUMBER routine is as follows.

1. Get in the (increment) and (start) parameters for the command and check the legality of these.
2. Next search through the BASIC text for those tokens which have an associated line reference that must be renumbered e.g. GOTO, GOSUB, etc.
3. Having found such a token we get in the ASCII line reference which follows and store this in an area of memory called BUFFER.
4. The ASCII line reference is converted to a *binary* line number (two bytes).
5. Search the BASIC text looking for this line number. At the start of the search a counter is initialised to the value of (start) and for every new line encountered in the search it is incremented by (increment). The value of this counter on exiting the search is then equal to the *new* (renumbered) line number.
6. Convert the value of the counter to an ASCII line reference in BUFFER.
7. Expand or contract BASIC following the token depending on whether the new line reference is longer or shorter than that which it replaces.

8. Put the contents of BUFFER in after the token.
9. Rebuild the 'chaining' of lines via link addresses.
10. Go to step 2. unless the end of code has been reached.
11. Renumber program lines starting at (start) and in steps of (increment)
12. Finished – return to BASIC via the CLR routine.

We will now describe the code which executes this algorithm in the program.

1. Get in and test parameters (Lines 1160–1345)

Lines 1170–1230 are concerned with getting in the increment and start parameters into INCRO and STRT. Two ROM routines are used which you will not have met before: the routine at $AEFD checks to see if the next byte in the buffer is a comma. If not, then it causes a syntax error. It returns via CHRGET with the character after the comma in A. $A96B gets an ASCII integer number (up to 65535 decimal) into $14,$15. If there are no parameters after the command, or only one, then defaults of STRT=1000 ($03E8) and INCRO=10 ($0A) are set up. At line 1325 the command parameters are tested to make sure they are legal. This is done using the routine responsible for step 5. of our general algorithm. In this case, if the counter overflows out of two bytes then we reject the parameters and print error message 1 BAD PARAMETERS, returning to BASIC via the ROM print routine at $AB1E. As we have seen before, this routine prints a string of ASCII bytes, which is terminated by a zero and the address of which is pointed to by the A (low byte) and Y (high byte) registers.

2. Search BASIC for tokens with line references (Lines 1370–1675)

In Lines 1370–1400, having checked that our parameters are legal we save the CHRGET pointer on the stack, since we will be altering it later on, and initialise the error flag to zero.

We are now ready to start searching the BASIC text for tokens. In lines 1470–1490 the text pointer CURR is initialised to the start of BASIC. Lines 1500–1515 check for the end of BASIC, while lines 1525–1535 check for the end of a text line. In lines 1565–1625 succeeding bytes are checked for the tokens for GOTO, GOSUB, LIST, RUN, THEN and GO. (Lines 1645–1675). The GO token is a special case since we also have to look in succeeding bytes for a TO token, ignoring any spaces in between. (The 'Toolkit' tokens RENUMBER and DELETE are not included in the search since these are obviously not meant to be used from within programs, although it is possible to do so.)

3. Get in ASCII line reference (Lines 1710–1895)

In lines 1710–1745 spaces between token and line reference are accounted for and NOFSET holds the index Y offset from (CURR) of the last byte *before* the line reference. Lines 1780–1850 get in successive bytes of the line reference into BUFFER until a non-ASCII numeric byte is reached. If this non-ASCII numeric byte was a comma or a hyphen (minus) character then the flag FLAG is set (line 1860) to indicate an ON GOTO, ON GOSUB, or LIST (number)–(number) type statement, i.e. a statement where there is more than one number associated with the token. In lines 1875–1895 a zero is placed at the end of BUFFER and the number of characters (less 1) in the buffer is stored in SPACE (which indicates how much 'space' is available for the new line reference), while the Y register offset from (CURR) of the next byte *after* the line reference is stored in YOFSET.

We have now placed the line reference in BUFFER, and at line 1920 we JSR to REPLAC, which is the routine which actually renumbers the reference.

4. Convert ASCII line reference to Binary (Line 2150)

The subroutine ASCBIN (lines 2225–2250) performs this conversion using the ROM routine at $A96B. You will recall from step 1. that we used this routine to get in ASCII parameters from the input buffer. By making the CHRGET pointer point instead to the location of BUFFER we can use it to convert our ASCII line reference into a binary line number held in $14,$15. This is the reason why we placed the contents of the CHRGET pointer on the stack in step 2.

5. Search BASIC for line number in $14,$15 (Line 2155)

The JSR to LINCNT (lines 2600–2820) performs the search. It is easy to locate the line numbers in BASIC text, since they are offset two bytes from the link address in each line. Our 'counter' is WKSPC3 (low byte) and WKSPC4 (high byte), and it is initialised in lines 2600–2615 to the (start) parameter STRT. The text pointer SRC is initialised from $2B,$2C (the BASIC system pointer to the first BASIC link) in lines 2625–2630. In lines 2655–2660 a check is made for the end of BASIC (zero link address). Assuming the end of BASIC is not reached we INY until the Y register holds 3, and compare the current line number with $14,$15 (lines 2665–2705). If a match is found then exit at line 2815 with Carry flag clear. If no match is found then we add the (increment) parameter INCRO to our counter (lines 2715–2750), get the address of the next line in A and X, go back to the top of the loop at line 2640 and start again.

There are two possible error exits for the two types error conditions. The end of BASIC being reached with the line number not found, for example, if there was a GOTO to a non-existent text line produces the first type of error. This is dealt with in line 2795 where the counter is set to $FFFF (65535) and the error flag is set. The second type is produced if the counter overflows. This is checked for by the BCS in line 2750 and it causes a return with the Carry *set*. This later is used in step 2. to check the validity of the command parameters. In this case entry is at TESTP, line 2580, instead of at LINCNT, where $14,$15 is set to $FFFF to ensure that we search the whole of BASIC. Returning with Carry *set* indicates that the parameters were *invalid* in this case.

6. Convert WKSPC3,4 to an ASCII line reference in BUFFER (Line 2160)

The BINASC routine (lines 2310–2490) carries out this operation. After step 6. the counter WKSPC3,4 should contain the binary value of our (renumbered) line reference. By successively subtracting from this value decreasing powers of 10 (i.e. 10,000 . . .100, 10 etc.) we may obtain the ten thousands . . . hundreds, tens, units digit sequence for a decimal number which is easily converted to ASCII and stored in BUFFER. There is a ROM routine which can perform this task, used to output decimal numbers, but since it is very slow it is not used here. The powers of ten required for our routine are stored in a 'look-up' table TENS (lines 5835–5845). In lines 2325–2385 these powers of ten are subtracted in a loop from WKSPC3,4 until it 'underflows', whereupon the X register value of the index for the loop is ORed with #$30 to convert it to ASCII and then stored in BUFFER as a decimal digit (lines 2415–2435). Lines 2395–2405 prevent 'leading zeroes' being stored in BUFFER by setting the flag SRC when the first non-zero digit has been generated, so that if the digit is zero and SRC is also zero then the digit is not stored. Lines 2455–2475 continue when the tens have been subtracted to set the remainder to the units digit. Finally, in line 2485, the *length* of the new line reference (the number of characters in BUFFER minus 1) is stored in NBUFF.

7. Expand or Contract BASIC as required (Line 2165)

The routine EXPO at lines 3250–3880 carries out this procedure. Lines 3250–3275 determine whether any memory move is required by subtracting the length of the new line number (from step 6.), held in NBUFF from the space already available, which is stored from step 3. in SPACE. If the result is zero then no memory move is required and we exit with Carry clear. If the result was not zero then the 'difference' between the two values is stored in DIFRNC. The memory block that requires moving is from the first byte following the *old*

line reference to the end of BASIC. The start value of this block, SRC, is calculated in lines 3295–3325 by adding the value of YOFSET (from step 3.) to the text pointer CURR. The SIZE of the block is calculated in lines 3350–3405 by subtracting SRC from the End-of-BASIC (Start-of-Variables) pointer $2D,$2E and subtracting 1. In lines 3415–3420 YOFSET is updated for the memory move by subtracting DIFRNC from it.

Finally, in lines 3440–3450, depending on whether DIFRNC is positive or negative we either branch to a memory SHRINK routine or carry on to the memory EXPAND routine to effect the actual move. You may ignore line 3445 for now, as this is only used specifically by the DELETE command. DIFRNC can never be zero for RENUMBER. If you look back to the discussion of the 'Wedge-MON' memory move command @T you will recall that we had to make a distinction between moving memory from a *higher* to a *lower* address and moving from a *lower* to a *higher* address, since different routines were required in each case. This is exactly the same situation as here. In order to protect data from being corrupted during the memory move we must employ different routines for expansion or opening up a space in the BASIC text and shrinking or closing up the program text.

Lines 3495-3675 carry out expansion. The source, SRC, for the memory move is calculated in lines 3495–3510 as equal to the start location of the memory block plus the high byte of its size. The destination DST is calculated in lines 3520–3575 as equal to the SRC plus the 'absolute' (2's complement) value of DIFRNC. The transfer loop at lines 3610–3660 transfers SIZE bytes plus SIZE+1 pages of memory from (SRC) to (DST) working *downwards* through memory. The routine exits with Carry *set*.

Lines 3720–3880 carry out the shrinking of BASIC. The source for the move is the start of the memory block SRC while the destination DST is calculated in lines 3720–3750. The transfer loop at lines 3780–3865 transfers SIZE+1 pages plus SIZE remainder bytes from (SRC) to (DST) moving *upwards* through memory. The routine exits Carry *set*.

8. Replace the contents of BUFFER after the token (Lines 2170-2185)

If no memory move was required in step 7. then the Carry flag will be clear and we can skip step 9. and jump straight to the PUTBAK subroutine (lines 3070–3110) which replaces the contents of BUFFER, i.e. the *new* (renumbered) line reference, after the token in place of the old line reference. Note that the offset from (CURR) of the first free byte after the token is derived from NOFSET.

If however the Carry was set after returning from the step 7.

subroutine then a memory move occurred, and so we must must pass on to step 9. after a JSR to PUTBAK so that the BASIC link addresses can be rebuilt and the pointers adjusted.

9. Rebuild chaining of BASIC and update BASIC pointers (Line 2180)

If a memory move occurred in step 7. then the BASIC link addresses around the point of the move will be disturbed and will need to be rebuilt. Also the Start-of-Variables pointer (End-of-BASIC) will need to be amended. The routine TIDY (lines 3145–3185) performs these tasks using the ROM routine for rechaining BASIC found at $A533 and deriving the new Start-of-Variables pointer by adding two to the final value of that routine's 'text pointer' held in $22,$23.

10. Go back to Step 2. unless end of BASIC reached

Upon returning from REPLAC at line 1950 the position in the BASIC text is recovered from YOFSET so that (CURR),Y will equal the next byte after the last renumbered line reference. In lines 1955–1965 we examine FLAG. If *set* this indicates that more than one line reference is associated with the last token (e.g. for an ON...GOTO) so we go back step 3. to look for subsequent line references. If *clear* then loop back to step 2. and look for the next token.

11. If end of BASIC reached then renumber lines (Line 2035)

We should by now have renumbered all ASCII line references in the BASIC program and it is only left to renumber the program's binary line numbers. This comparatively simple job is undertaken by the RENUM routine (lines 2850–3040). This works in the same way as the LINCNT routine of step 5. We initialise a counter WKSPC2,3 with the (start) parameter STRT. For every line number encountered we replace the line number with the value of the counter and increment the counter by the value INCRO. We exit when the end of BASIC is reached.

12. Check ERROR flag and exit via CLR routine (Lines 2040–2085)

If the BASIC text contained some non-existent line references then bit 7. of the error flag ERROR will be set. We test for this in lines 2040–2045. If there are no errors then we exit via line 2065, first recovering the CHRGET pointer and then jumping into the CLR routine which normalises all BASIC pointers and returns us to BASIC. If however there are errors then before exiting we print error message 2, BAD LINES via the ROM routine $AB1E. The spurious line references will have been renumbered to 65535. It would seem better to finish

renumbering the program and then flag an error than to halt the renumber process midway.

Designing the DELETE Routine

The DELETE command routine is shorter than that for RENUMBER, but it utilises many of the RENUMBER subroutines. It is however complicated by its range of possible command parameters. The basic idea of DELETE is to find the addresses in memory of the BASIC text lines specified in the DELETE parameters. This done, the SHRINK routine from RENUMBER may move memory from the higher address down to the lower address, thus excluding the range of lines in between.

Lines 3940-4050 are concerned with getting in parameters for the command. The parameters for DELETE are similar to those of the BASIC LIST command. The possibilities are:

a)	LINE		deletes a *single* line
(b)	LINE1	– LINES2	deletes a *range* of lines
(c)		– LINES2	deletes from *start* up to LINES2
(d)	LINE1	–	deletes from LINES1 to the *end*
(e)		–	deletes from *start* to *end*

Our code hence has to be able to cope with these five possibilities. It is worth looking at this in some detail. Line 3940 causes a syntax error if there are no parameters at all, and we then proceed as follows.

First parameter

In line 3965 the ROM routine at $A96B is used to get in the first parameter as a binary line number in $14,$15. The $A96B routine cannot cope with non-numbers so if a 'minus' token (a hyphen is tokenized to the *minus* arithmetic operator) is in the buffer then the routine will return $14,$15 equal to zero, effectively making the first parameter equal the start of the program and coping with possibilities (c) and (e) above. At lines 3970–3990 the address of this line within the BASIC text is placed into $5F, $60 by the ROM routine at $A613. This routine searches through BASIC until it finds a line number equal to or greater than the value held in $14,$15. This address is then stored in DST, since it will be the destination for our memory move.

Second Parameter

At lines 4000–4005 a check is made for possibility (a) above, i.e. no further parameters. If this is the case then we branch to lines 4120–4155 where the source SRC for our memory move is defined as the address of the next line *after* that found in line 3970 (given by $5F). If there are further parameters they are got in by lines

4015–4050. If there is just a hyphen (minus token) then in lines 4060–4090 the source for the memory move is set to the End-of-BASIC, taking in possibilities (d) and (e) above. If, however, there is a second line number (possibility (b)) its address is pulled in at line 4105. Note that the routine at $A613 conditions the Carry flag depending on whether the sought-after line is found or not. If the line is found then the routine returns the line's address with the Carry *set*, otherwise it returns with the address of the *next* line with the Carry *clear*. The branch instruction at line 4110 copes with possibility of being asked to DELETE a line that does not exist. Thus, if the Carry is *set* then SRC will equal ($5F), and otherwise SRC=$5F,$60. Lines 4190–4220 reject silly entries such as DELETE 100–10.

The actual DELETE operation is effected by lines 4230–4250. We use the routine from RENUMBER called EXPO (lines 3145–3880) and enter it at line 3350. EXPO calculates the size of the transfer block, SIZE, and by first initializing DIFRNC to zero control is directed straight to the SHRINK routine (lines 3720–3880), entered at line 3760, which performs the memory contraction.

At line 4245, after returning from the memory move, a JSR is taken to the TIDY routine from RENUMBER (lines 3145–3185) to rebuild 'chaining' and reset the Start-of-Variables pointer. Exit is, as with RENUMBER, via the CLR routine. Lines 4260–4265 are the error exit, with #$0B being the code for 'syntax error'.

Designing the SEARCH and REPLACE Commands

The SEARCH and REPLACE commands are fairly complex to program but the SEARCH code is similar to the (simpler) @H (Hunt) command of 'Wedge-MON'. REPLACE has much in common with RENUMBER and also uses some of its subroutines.

The SEARCH/REPLACE Algorithm

Get Parameters

1. Place SEARCH string into BUFF1. Store number of characters in SPACE
2. If REPLACE specified then get REPLACE string into BUFF2. Store number of characters in NBUFF

Search for SEARCH String

3. Set (CURR) to start of BASIC. Initialise PFLAG to zero
4. Initialise X to zero and Y to 1
5. Store line number in STRT
6. If End-of-BASIC then go to step 28.

7. Increment Y. If end of line then go to 27.
8. If (CURR),Y=BUFF1,X then go to step 11.
9. If X=zero then 7.
10. Load Y from WKSPC1. Set X to zero. Go to 7.
11. If X greater than zero then 13.
12. Store Y in WKSPC1
13. Increment X. If X<>SPACE then 7.
14. Store Y in YOFSET
15. If only SEARCH specified then 21.

Replace SEARCH String with REPLACE String

16. Increment YOFSET
17. JSR to RENUMBER EXPO routine
18. Decrement YOFSET
19. Load Y from WKSPC1
20. Place contents of BUFF2 at (CURR),Y

Print Old Line ($14,$15) if Current Line Number (STRT) Different

21. If PFLAG>0 then 23.
22. Increment PFLAG. Go to 25.
23. If STRT=$14,$15 then 26.
24. LIST the line with its line number in $14,$15
25. $14,$15=STRT
26. Load Y with YOFSET. Load X with zero. Go to 7.

Access Next Text Line

27. Set (CURR) to point to next line. Go to 4.

Exit at End of Search

28. If PFLAG=zero then 30.
29. LIST the line with the line number in $14,$15
30. Exit to BASIC via CHRGET

Some of the above might at first seem unnecessarily complicated, but there are good reasons for this, however. For example, in the search loop (steps 2. to 15.) when a match is found in the text with the first character of BUFF1 the Y register offset in the text line is stored in WKSPC1 (step 12.) The reasons for this are twofold. Firstly, if the command is a REPLACE then we know that the replacement string is to be placed at (CURR)+WKSPC1. Secondly, if there is only a partial match between the text and BUFF1 we want to re-start the search at (CURR)+WKSPC1+1, and not at any later point. As an example, say our

search string was /RAT/ and occurring in the BASIC text was the line:

10 REM RRAT

There would be a match between the two R's of RAT and RRAT but the next R of the text and the A of the search string would not match. Now it is important that we re-start the search at the second text R and not at A, or we will fail to find the match.

Another area of complexity is that in steps 21. to 25. we LIST not the line with the *current* line number but that with the *previous* line number. The reason for this is simple. Take the useless BASIC program below as an example:

10 REM RAT RAT RAT RAT RAT
20 REM RAT RAT RAT RAT RAT

If we were to SEARCH for RAT with the command SEARCH/RAT/ it would be most inconvenient if each line were to be LISTed five times, once for every RAT found. We only ever need a line to be LISTed once, regardless of the number of times the search string occurs within it. However, this device of only LISTing the *old* line numbers could leave us with an *old* line unLISTed at the end. This is checked for in steps 28. to 29.

Note also that in step 16. we increment YOFSET before the JSR to EXPO. This is because the RENUMBER routine EXPO expects YOFSET to be one greater than the SEARCH and REPLACE strings, just as it does the old and new (renumbered) ASCII line references in RENUMBER. The next section describes how this algorithm has been coded.

The SEARCH and REPLACE Routine

Get Parameters (Lines 4410–4625)

Although both the commands use the same code they are distinguished by the value of FLAG, which is set to $FF for REPLACE and $00 for SEARCH (lines 4350–4385).

You may find lines 4440–4445 confusing. By storing the value #$01 in location $81 the CHRGET routine is altered so that it no longer ignores 'space' characters. This enables leading spaces to be incorporated in the search and replace strings. Before finally leaving the routine at lines 4655 and 4710 CHRGET is restored to normal. The string delimiter for SEARCH and REPLACE is the oblique stroke, '/' which is one of the characters that is tokenised into a BASIC operator, in this case as the division sign. Now, while the opening oblique in a SEARCH

or REPLACE command will always be in *token* form it is necessary for the succeeding characters to check also for the *untokenized* (ASCII) oblique character. This is because a REM or quotes may be included in the command string, in which case subsequent characters would not be tokenised. For example in SEARCH /REM RAT/ the first backslash would be the token (#$AD), but the second would be the ASCII character (#$2F).

Search Loop (Lines 4830–4930)

This should be fairly easy to follow from the algorithm description. Note in lines 4920–4930 how the differentiation between SEARCH and REPLACE is achieved by BITing FLAG. This operation will result in bit 7 of the P register (the Negative flag) being set if FLAG is #$FF, and the branch will not occur.

Replace SEARCH String with REPLACE (line 4960)

The replace is carried out by the SWITCH routine at lines 5180–5250. This is rather like a combination of the RENUMBER routines REPLAC and PUTBAK.

LIST a BASIC Line (Lines 5070–5115)

This routine requires us to 'bank out' the BASIC ROM. You may remember from the 'Command Handler' program earlier on in the book that one of the initialisation activities of 'Expander' was to copy the whole BASIC interpreter from ROM down to the RAM area underlying the ROM. We also altered the RAM version of the ROM LIST routine by placing an RTS at $A714. Now you will see the reason for this. Altering the list routine in this way enables us to use it as a subroutine from our own programs to LIST individual BASIC lines. Line 5070 prepares data for the LIST routine by using the ROM routine at $A613 that we met in DELETE to derive the address of the line contained in $14,$15. The actual LIST is performed by LISTIT at lines 5280–5305. In order to gain access to our RAM version of BASIC we must first 'bank out' the ROM version of BASIC by using the 6510 I/O port located at address $01. Lines 5280–5285 switch out the ROM by writing #$36 to address $01. A JSR to $A6BD performs the LIST and at lines 5295–5305 the ROM BASIC is switched back in. After LISTing a line it is necessary to cursor up one row to prevent blank lines being left between LISTed lines. This is achieved by outputting a <cursor up> character at lines 5080–5085.

Exit Routine (Lines 4655–4700)

Lines 4655–4670 are the error exit. Exit is back to BASIC at line 4700 through CHRGET, since the CHRGET pointer is still pointing to an oblique character, which would otherwise cause a syntax error.

The OLD Command Routine

The OLD command is a simple but very useful routine that will get you out of a lot of programming scrapes. To understand how OLD can magically resurrect accidentally or inadvertently NEWed BASIC programs it is first necessary to know precisely what happens when you type NEW:

1. Two zero bytes are placed in the first link address of BASIC. This address will usually be $0801, $0802 if you have moved the start of BASIC. You will recall from the RENUMBER routine that a zero link address indicates the end of BASIC text. The 64 thinks no program exists.
2. The Start-of-Variables pointer at $2D,$2E is set to the start of BASIC plus 2.
3. The CHRGET pointer is set to the start of BASIC.
4. At this point the NEW command joins the CLR command.

To recover a NEWed BASIC program it is necessary to reverse the first two steps by rebuilding the first link address of BASIC and resetting the Start-of-Variables pointer to the end of the BASIC text. We have already seen how to do this in the TIDY routine of RENUMBER.

Before we JSR to TIDY, however, we must change the high byte of the first link from zero or TIDY will not work. This is achieved in lines 5380–5385. After returning from TIDY we exit to BASIC via the CLR routine.

The AUTO Command Routine

The idea of an AUTO command is that after any numbered line is entered via the keyboard the next line number, equal to the previously entered line number plus whatever AUTO increment value has been set should automatically be printed to the screen. You should remember that in connection with 'Wedge-MON' we discussed the technique of wedging BASIC via the *warm start* vector at $0302. This allowed us to examine keyboard input before it was passed to the BASIC editor. You may also recall how by writing to the Keyboard Buffer at $0277 we were able to 'instantaneously' print characters to the screen as though they had been typed in. Combining these two techniques gives us an AUTO facility.

Setting Up the Wedge (Lines 5455–5550)

This section of the code gets in the (step) parameter and sets up the wedge. The step value is stored in AUTSTP which is located at $03FC, $03FD, a relatively safe area of memory. It is important to note that since AUTSTP must be preserved any subsequent programs you write must not corrupt these two memory locations if you want AUTO to work. If AUTSTP is greater than zero then the warm start vector is changed to point to line 5620. If, on the other hand, AUTSTP equals zero then the vector is reset to the system default value and AUTO is disabled.

The Wedge Code (Lines 5620 – 5810)

Lines 5620–6545 duplicate the ROM code. Line 5620 gets characters into the input buffer until a <return> is entered. In line 5635 the first character in the buffer is retrieved by CHRGET. If the Carry is clear, i.e. if it is an ASCII line number, then control passes to the rest of the wedge at line 5655, otherwise we exit back to the normal ROM routine at line 5645. In lines 5655–5670 the CHRGET pointer is temporarily stored and the ROM routine at $A96B is used to input the ASCII line number from the buffer as a binary number in $14,$15. If this is zero we immediately exit to line 5785. If non-zero this is added to AUTSTP to produce the AUTO line number in WKSPC3,4. At line 5715 the RENUMBER routine BINASC is used to convert WKSPC3,4 to ASCII digits in BUFFER. The X register value (from BINASC) is incremented twice and then stored in the system variable $C6 as the number of characters currently held in the keyboard buffer. In lines 5735–5775 first a 'space' and then the contents of BUFFER are placed in the keyboard buffer. Finally, in lines 5785–5810 the CHRGET pointer is recovered and we exit back to the ROM code.

7 Extending BASIC: Graphics and sound

This final chapter presents a further set of command routines for 'Expander', concerned with graphics and sound programming. SPRITE and KSPRITE simplify enormously the programming of sprites, and JSPRITE enables the movement of sprites to be controlled by a joystick, with the speed of movement specified. SOUND similarily eradicates the multiple POKES normally required to handle sound on the 64, enabling a single BASIC instruction to produce a note or noise. SCROLL gives a left and right character scroll facility, and the group of Hi-res commands allows simple drawing and plotting on the bit-mapped screen. Finally, four functions for use with the SPRITE and JSPRITE commands are given. The 'Expander' program is then complete, but as mentioned before, there is room for you to add your own routines, as soon as you've finished the typing for this lot! As usual, both assembler code and BASIC loader versions of the routines are given, and incorporation of the new commands into 'Expander' follows the same procedure as that given at the start of Chapter 5.

The SPRITE and KSPRITE Commands

Anyone who has tried programming sprites in BASIC knows what a pain it can be. To put a sprite on the screen requires at least five POKES, and possibly as many as eight or nine, depending on what parameters you want to set up. Such a mess of POKES leads to a large, slow and generally unreadable program. But help is on its way!

The 'Expander' sprite command SPRITE allows multiple sprites to be controlled easily and naturally by allowing you to enter all the sprite parameters, or as many as you wish to enter, using only the one command. In order to enable and position an expanded, multi-coloured, sprite at given screen co-ordinates and deriving its data from a given memory block you require only one BASIC command!

SPRITE

Syntax: SPRITE (number), (x-position), (y-position), (block), (colour), (expand), priority

Where: (number) defines the sprite number
(x-position) defines the sprite's horizontal co-ordinate
(y-position) defines the sprite's vertical co-ordinate
(block) defines the memory block from which the sprite derives its pattern information
(colour) defines the sprite colour (values > 15 enable multicolour)
(expand) defines how the sprite is expanded: 0=unexpanded, 1=x–expanded, 2=y–expanded, 3=both x and y expansion
(priority) defines the sprite's display priority with regard to background text. 1 specifies sprite has *higher* priority, 0 specifies *lower* priority.

This list of seven parameters may seem a bit daunting but luckily (or by design!) not all of them have to be entered every time. The SPRITE command will accept as few or as many parameters, up to the full set of seven, that you choose to set. Typically, the user would set up all the parameters to initialize the sprite and then just enter as many as are required to identify and position the sprite.

Example: SPRITE 0 simply enables sprite zero without affecting its other characteristics, while SPRITE with no parameters enables *all* sprites. SPRITE 0,100,100 moves sprite zero to position 100,100 without changing its colour, expansion, etc.

KSPRITE

KSPRITE kills or disables sprites. It takes only one parameter, which identifies the particular sprite to be disabled. KSPRITE with no parameters kills all sprites – they may be all re-enabled with SPRITE.

Syntax: KSPRITE (number)

Example: KSPRITE 0 turns off sprite zero. Its position, etc., remains unchanged and SPRITE 0 switches it back on. A most important use of KSPRITE if you possess a disk drive is to switch off all sprites prior to disk access. It seems to be a quirk of the VIC chip's Phase 2 DMA (Direct Memory Access) that having sprites on the screen upsets the timing for disk activities. The problem arises when the VIC II chip stops the processor momentarily so that it can access sprite data in memory. I personally do not know what the situation is as regards tape, but the best advice would seem to be to do a KSPRITE before any SAVEing or LOADing, and use SPRITE when you have finished, if you want to ensure the integrity of your data.

Potential Problems with SPRITE and KSPRITE

If the machine crashes, 'warm starts' or performs the wrong command then see the discussion of problems in the introduction to Chapter 6.

If you go through the incorporation procedure and try the SPRITE command, with the result that the machine does not crash but you get no sprites then you may have set the command parameters such as to produce an 'invisible' sprite. This may be due to setting the sprite colour equal to the background colour, the sprite data coming from a memory block full of zeroes, or the sprite being positioned off the screen. To check, try the following statement:

```
SPRITE 0,120,150,0,7,3
```

This should produce a large yellow undefined sprite in the middle of the screen under all circumstances. If this does not work then we must assume some typing error. Go back and read the introduction to the previous chapter. Now comes the code for the commands, and then we will look at how the SPRITE and KSPRITE commands are coded.

The SPRITE and KSPRITE Code

Assembler Listing

```
1000 *=$8B00
1010 ;-------------------
1020 PTR     =$14;*
1030 SPRIT   =$0334
1040 XCOOR   =$0335
1050 YCOOR   =$0337
1060 BLOCK   =$0338
1070 COLOU   =$0339
1080 EXPAN   =$033A
1090 ;-------------------
1100 ;--KILL SPRITE(S)--
1110 ;IF NO PARAMETERS THEN CLEAR
1120 ;SPRITE-ENABLE REGISTER ELSE MASK
1130 ;OUT BIT CORRESPONDING TO SPRITE
1140 ;-------------------
1150 KSPRI   BNE KSEL
1160         LDA #$00
1170         BEQ KALL
1180 ;---------
```

```
1190 KSEL   JSR $B79E
1200        TXA
1210        AND #$07
1220        TAX
1230        LDA MSK,X
1240        EOR #$FF
1250        AND $D015
1260 KALL   STA $D015
1270        RTS
1280 ;-----------------------
1290 ;--SETUP A SPRITE--
1300 ;GET IN SPRITE NUMBER AND SET
1310 ;CORRESPONDING BIT IN SPRITE-
1320 ;ENABLE REGISTER
1330 ;-----------------------
1340 SPRI   BNE SPRI2
1350        LDA #$FF
1360        BNE SPRI3
1370 ;----------
1380 SPRI2  JSR $B79E
1390        TXA
1400        AND #$07
1410        STA SPRIT
1420 ;----------
1430        TAX
1440        LDA MSK,X
1450        ORA $D015
1460 SPRI3  STA $D015
1470 ;-----------------------
1480 ;--GET IN SPRITE X POSITION--
1490 ;-----------------------
1500        JSR CHKIT
1510        JSR $AD8A
1520        JSR $B7F7
1530        LDA $14
1540        LDX $15
1550        STA XCOOR
1560        STX XCOOR+1
1570 ;-----------------------
1580 ;--GET IN SPRITE Y-POSITION--
1590 ;-----------------------
1600        JSR CHKIT
1610        JSR $B79E
1620        STX YCOOR
```

```
1630 ;-----------------------
1640 ;--FEED PARAMS TO VIC II CHIP --
1650 ;THE X AND Y POSITIONS ARE STORED
1660 ;IN THE CORRECT REGISTERS
1670 ;-----------------------
1680         LDA SPRIT
1690         ASL A
1700         TAX
1710         LDA XCOOR
1720         STA $D000,X
1730         LDA YCOOR
1740         STA $D001,X
1750         LDX SPRIT
1760         LDA MSK,X
1770         LDY XCOOR+1
1780         BEQ NOTH
1790         ORA $D010
1800         BNE CONS
1810 NOTH    EOR #$FF
1820         AND $D010
1830 CONS    STA $D010
1840 ;-----------------------
1850 ;--GET IN MEMORY BLOCK--
1860 ;-----------------------
1870         JSR CHKIT
1880         JSR $B79E
1890         STX BLOCK
1900 ;-----------------------
1910 ;--STORE BLOCK AFTER SCREEN--
1920 ;THE SCREEN ADDRESS IS DERIVED
1930 ;FROM THE VIC II AND CIA REGISTERS
1940 ;THAT DETERMINE ITS POSITION/
1950 ;BLOCK IS STORED AFTER THE SCREEN
1960 ;AS THE SPRITE POINTER
1970 ;-----------------------
1980         LDA $D018
1990         AND #$F0
2000         LSR A
2010         LSR A
2020         ORA #$03
2030         LDY #$F8
2040         STY PTR
2050         STA PTR+1
2060         LDA $DD00
```

```
2070          AND #$03
2080          EOR #$03
2090          CLC
2100          ROR A
2110          ROR A
2120          ROR A
2130          ORA PTR+1
2140          STA PTR+1
2150          LDY SPRIT
2160          LDA BLOCK
2170          STA (PTR),Y
2180 ;-----------------------
2190 ;--GET SPRITE COLOUR--
2200 ;COLOUR IS STORED IN CORRESPONDING
2210 ;SPRITE COLOUR REGISTER/IF COLOUR
2220 ;IS > #$0F THEN CORRESPONDING
2230 ;BIT IN MULTICOLOUR REGISTER SET
2240 ;ELSE IT IS RESET
2250 ;-----------------------
2260          JSR CHKIT
2270          JSR $B79E
2280          TXA
2290          LDX SPRIT
2300          STA $D027,X
2310          CMP #$10
2320          BCC NMCOL
2330          LDA MSK,X
2340          ORA $D01C
2350          STA $D01C
2360          BNE EXPA
2370 ;----------
2380 NMCOL    LDA MSK,X
2390          EOR #$FF
2400          AND $D01C
2410          STA $D01C
2420 ;-----------------------
2430 ;--GET IN 'SPRITE EXPAND' PARAMS--
2440 ;GET IN PARAMETER/IF BIT 0 IS SET
2450 ;THEN SET CORRESPONDING BIT IN
2460 ;HORIZONTAL EXPAND REGISTER ELSE
2470 ;RESET IT
2480 ;-----------------------
2490 EXPA     JSR CHKIT
2500          JSR $B79E
```

```
2510          STX EXPAN
2520          LDX SPRIT
2530          LDA #$01
2540          BIT EXPAN
2550          BEQ VER2
2560          LDA MSK,X
2570          ORA $D017
2580          STA $D017
2590          BNE HORZ
2600 ;----------
2610 VER2     LDA MSK,X
2620          EOR #$FF
2630          AND $D017
2640          STA $D017
2650 ;-----------------------
2660 ;IF BIT 1 OF EXPAND PARAMETER SET
2670 ;THEN SET BIT IN VERTICAL EXPAND
2680 ;REGISTER ELSE RESET IT
2690 ;-----------------------
2700 HORZ     LDA #$02
2710          BIT EXPAN
2720          BEQ HORZ2
2730          LDA MSK,X
2740          ORA $D01D
2750          STA $D01D
2760          BNE PRIOR
2770 ;----------
2780 HORZ2    LDA MSK,X
2790          EOR #$FF
2800          AND $D01D
2810          STA $D01D
2820 ;--------------------
2830 ;--GET IN DISPLAY PRIORITY--
2832 ;IF=1 THEN SPRITE HAS PRIORITY
2834 ;ELSE IF=0 TEXT HAS PRIORITY
2840 ;--------------------
2850 PRIOR    JSR CHKIT
2860          JSR $B79E
2870          LDY SPRIT
2880          TXA
2890          AND #$01
2891          BNE DOMIN
2892          LDA MSK,Y
2893          ORA $D01B
```

```
2894          BNE COALES
2895 ;----------
2896 DOMIN    LDA MSK,Y
2897          EOR #$FF
2900          AND $D01B
2910 COALES   STA $D01B
2911 ;--------------------
2912 ;--EXIT TO BASIC--
2913 ;--------------------
2914          RTS
2950 ;--------------------
2960 ;--CHECK FOR END OF COMMAND--
2970 ;IF NO MORE PARAMETERS THEN POP
2980 ;STACK AND RETURN TO BASIC ELSE
2990 ;JUMP TO COMMA HANDLER
3000 ;--------------------
3010 CHKIT    JSR $0079
3020          BEQ BYPAAS
3030          JMP $AEFD
3040 ;----------
3050 BYPAAS   PLA
3060          PLA
3070          RTS
3080 ;--------------------
3090 ;--BITMASKS--
3100 ;VALUES CORRESPOND TO 2 X WHERE X
3110 ;= SPRITE (0-7)
3120 ;--------------------
3130 MSK      .BYTE $01,$02,$04,$08
3140          .BYTE $10,$20,$40,$80
3150 ;--------------------
```

BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 35583+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
```

```
110 REM --- BLOCK 1
111 DATA208,4,169,0,240,15,32,158
112 DATA183,138,41,7,170,189,64,140
113 DATA73,255,45,21,208,141,21,208
114 DATA96,208,4,169,255,208,16,32
115 DATA158,183,138,41,7,141,52,3
116 DATA170,189,64,140,13,21,208,141
117 DATA21,208,32,53,140,32,138,173
118 DATA32,247,183,165,20,166,21,141
119 DATA-7159
120 REM --- BLOCK 2
121 DATA53,3,142,54,3,32,53,140
122 DATA32,158,183,142,55,3,173,52
123 DATA3,10,170,173,53,3,157,0
124 DATA208,173,55,3,157,1,208,174
125 DATA52,3,189,64,140,172,54,3
126 DATA240,5,13,16,208,208,5,73
127 DATA255,45,16,208,141,16,208,32
128 DATA53,140,32,158,183,142,56,3
129 DATA-5959
130 REM --- BLOCK 3
131 DATA173,24,208,41,240,74,74,9
132 DATA3,160,248,132,20,133,21,173
133 DATA0,221,41,3,73,3,24,106
134 DATA106,106,5,21,133,21,172,52
135 DATA3,173,56,3,145,20,32,53
136 DATA140,32,158,183,138,174,52,3
137 DATA157,39,208,201,16,144,11,189
138 DATA64,140,13,28,208,141,28,208
139 DATA-5980
140 REM --- BLOCK 4
141 DATA208,11,189,64,140,73,255,45
142 DATA28,208,141,28,208,32,53,140
143 DATA32,158,183,142,58,3,174,52
144 DATA3,169,1,44,58,3,240,11
145 DATA189,64,140,13,23,208,141,23
146 DATA208,208,11,189,64,140,73,255
147 DATA45,23,208,141,23,208,169,2
148 DATA44,58,3,240,11,189,64,140
149 DATA-6671
150 REM --- BLOCK 5
151 DATA13,29,208,141,29,208,208,11
152 DATA189,64,140,73,255,45,29,208
153 DATA141,29,208,32,53,140,32,158
```

```
154 DATA183,172,52,3,138,41,1,208
155 DATA8,185,64,140,13,27,208,208
156 DATA8,185,64,140,73,255,45,27
157 DATA208,141,27,208,96,32,121,0
158 DATA240,3,76,253,174,104,104,96
159 DATA-6974
160 REM --- BLOCK 6
161 DATA1,2,4,8,16,32,64,128
162 DATA-255
163 DATA END
1000 REM -- ENABLE NEW COMMANDS --
1001 AD=33331
1002 REM - SPRITE
1003 POKE AD,24:POKE AD+1,139
1004 REM - KSPRITE
1005 POKE AD+2,255:POKE AD+3,138
1006 END
```

The KSPRITE Routine (Lines 1150–1270)

A parameter is stored in X by the routine at $B79E. Lines 1200–1220 transfer it to A and then AND it with #$07, thus masking our values greater than seven (since only the bottom three bits of the parameter value are now used to identify the sprite). This value is used to index the bit mask table MSK (Lines 3130–3140) and find the mask corresponding to that sprite. The bit mask is equal to 2, raised to the power of the sprite number. In lines 1230–1270 we 'flip' the mask by EORing the mask value with #$FF. This has the effect of turning all the 1's to 0's and vice versa. ANDing this binary pattern with the Sprite Enable register, location $D015, causes the bit corresponding to the parameter sprite to be turned off and consequently that sprite is selectively disabled. If there are no parameters for KSPRITE then the Sprite Enable register is simply zeroised, disabling all sprites.

The SPRITE Routine

Enable Sprite(s) (Lines 1340–1460)

The sprite parameter is pulled in and used to select a mask from the MSK table which is ORed into the Sprite Enable register at $D015. The sprite number is stored in SPRIT. If there are no parameters then the register is loaded with #$FF and *all* sprites are enabled.

Get in X and Y Positions (Lines 1500–1830)

In lines 1500–1620 the sprite's position parameters are input and

stored in XCOOR, XCOOR+1 and YCOOR. From now on, before any parameters are input the end of the input stream is checked for by CHKIT (lines 3010–3070). This checks first to see if the buffer is empty, in which case it pops the stack and returns to BASIC. If there is further data in the buffer it returns via the comma handler routine at $AEFD. In lines 1680–1830 the position parameters are written to the VIC II registers which correspond to the selected sprite. These are indexed by the sprite number from SPRIT. If the high byte of the sprite X co-ordinate position is set then the corresponding MSK value is ORed in to Position register $D010, else its 2's complement form is ANDed out (lines 1810–1820).

Get In Sprite Memory Block (Lines 1870–2170)

In this section the sprite memory block is input and stored in its position after the screen table. Since the screen can be moved around within a memory bank and different memory banks may also be selected (see below, under the BITMAP command) it is not sufficient to assume that the screen will be at $0800. In lines 1980–2170 a pointer to the end of the screen is calculated in (PTR) from the Screen Address and Bank Address registers $D018 and $DD00. The sprite number from SPRIT is used to Y index PTR and select the correct sprite pointer for the sprite concerned.

Set Sprite Colour (Lines 2260–2410)

The sprite colour is stored in the Colour Register $D027, indexed by the sprite number from SPRIT. Colour values greater than 16 cause the bit corresponding to that sprite to be set in the Multicolour register $D01C, else that bit is cleared.

Set Sprite Expansion (Lines 2490–2810)

Lines 2530–2540 and 2700–2710 respectively test bits 0 and 1 of the sprite expansion parameter input in lines 2490–2510. On this basis the corresponding bits in the Horizontal and Vertical Expansion registers $D017 and $D01D are either set or cleared.

Set Display Priority (Lines 2850–2910)

Lines 2850–2910 clear or set the respective bit corresponding to that sprite in the Sprite–Text Priority register $D01B, depending on whether the priority parameter was 1 or zero.

Exit back to BASIC at line 2914.

The JSPRITE Command

The JSPRITE routine, which illustrates use of the BASIC Interrupt and the reading of Joystick data is an exciting command for the reason that it makes available to the BASIC programmer a facility that would normally require a total machine code implementation. JSPRITE is meant to be used in conjunction with the SPRITE and KSPRITE commands of the preceding section and its function is to link the movement of designated sprites to the movement of Joystick 2. That is, once a sprite is so linked it will move right when the joystick is pulled right, and so on. Up to eight sprites simultaneously may be so linked and command parameters include 'wall' limits for each sprite and the overall speed of movement. By using JSPRITE it is possible to move sprites smoothly around the screen far faster than would be possible from BASIC, allowing the development, for instance, of BASIC games virtually indistinguishable from their machine code equivalents.

JSPRITE

Syntax: JSPRITE (number), (low x–limit), (high x–limit), (low y–limit), (high y–limit), (speed)

Where: (number) defines the sprite number
(low x–limit) defines the leftmost boundary of the screen window for movement
(high x–limit) defines the rightmost boundary of the screen window for movement
(low y–limit) defines the topmost boundary of the screen window for movement
(high y-limit) defines the bottommost boundary of the screen window for movement
(speed) defines the speed of sprite movement

JSPRITE with no parameters 'un-links' all sprites, while JSPRITE X unlinks sprite number X. The speed parameter is optional and defaults to a value of 16666. It is really an 'inverse speed' since lower values speed up movement while higher values reduce it. It is only possible to set one overall speed and that set for one sprite applies to all. When changing the speed parameter you will notice the cursor flash rate and the speed of cursor movement will vary accordingly. The reason for this will be made clear below. While speed is optional it is necessary in order to link a sprite to define all four limit parameters.

Example:
```
SPRITE 0,100,100,0,7,0
SPRITE 1,100,100,0,7,0
JSPRITE 0,11,331,39,239
JSPRITE 1,11,331,39,239
```

These BASIC statements will enable sprites 0 and 1 and link them to the joystick and they will observe the normal screen limits. See the section on Functions later in this chapter for the XSPR(X) and YSPR(X) functions, which further enhance the usefulness of JSPRITE.

Potential Problems with JSPRITE

What was said in this regard for the SPRITE and KSPRITE commands applies equally for JSPRITE. Additionally, as we shall see, JSPRITE works by 'piggy-backing' on the system Interrupt Request (IRQ) interrupt, but makes no allowance for other user IRQ wedges. So if you are also using the IRQ for another program it will have to allow for JSPRITE or strange things could happen. We shall look in detail at how JSPRITE works after giving the code for the routine.

The JSPRITE Code

Assembler Listing

```
1000 *=$8D00
1010 ;--------------------
1020 SPEED  =$0336;*
1030 ;--------------------
1040 ;--GET IN SPRITE PARAMETER--
1050 ;IF NO PARAMETERS THEN CLEAR
1060 ;'ENABEL' BYTE AND LIMITS TABLE/
1070 ;RESTORE DEFAULT IRQ VECTOR AND
1080 ;IRQ RATE/RETURN
1090 ;--------------------
1100 JSPRI  BNE OK1
1110 ;----------
1120        LDX #$28
1130        LDA #$00
1140 STC    STA ENABEL,X
1150        DEX
1160        BPL STC
1170 ;----------
1180        SEI
1190        LDA #$31
1200        LDY #$EA
1210        STA $0314
1220        STY $0315
1230        LDA #$1A
1240        LDY #$41
1250        STA $DC04
```

```
1260          STY $DC05
1270          CLI
1280          RTS
1290 ;--------------------
1300 ;MULTIPLY SPRITE NUMBER BY 2 AND
1310 ;STORE IN NSPR/MULTIPLY BY 6 AND
1320 ;STORE IN NSPRB6/
1330 ;--------------------
1340 OK1      JSR $B79E
1350          TXA
1360          AND #$07
1370          ASL A
1380          STA NSPR
1390          ASL A
1400          CLC
1410          ADC NSPR
1420          STA NSPRB6
1430 ;-----------------------
1440 ;--UNLINK SPRITE--
1450 ;IF NO FURTHER PARAMETERS THEN
1460 ;UNLINK SPRITE
1470 ;-----------------------
1480          JSR $0079
1490          BNE OK2
1500 ;----------
1510          LDX NSPR
1520          LDA MAASK,X
1530          EOR #$FF
1540          AND ENABEL
1550          STA ENABEL
1560          RTS
1570 ;-----------------------
1580 ;--GET IN X-LIMITS--
1590 ;GET IN LOWER X LIMIT/STORE IN
1600 ;LIMITS TABLE AT LIMITS,LIMITS+1
1610 ;(+SPRITE NUMBER*6)
1620 ;-----------------------
1630 OK2      JSR $AEFD
1640          JSR $AD8A
1650          JSR $B7F7
1660          LDY NSPRB6
1670          LDA $14
1680          STA LIMITS,Y
1690          LDA $15
```

```
1700          BEQ STLX
1710          LDX NSPR
1720          LDA MAASK,X
1730 STLX     STA LIMITS+1,Y
1740 ;-----------------------
1750 ;STORE UPPER X-LIMIT AT LIMITS+2,
1760 ;LIMITS+3 (+SPRITE NUMBER*6)
1770 ;-----------------------
1780          JSR $AEFD
1790          JSR $AD8A
1800          JSR $B7F7
1810          LDY NSPRB6
1820          LDA $14
1830          STA LIMITS+2,Y
1840          LDA $15
1850          BEQ STHX
1860          LDX NSPR
1870          LDA MAASK,X
1880 STHX     STA LIMITS+3,Y
1890 ;-----------------------
1900 ;--GET IN Y LIMITS--
1910 ;STORE LOWER Y-LIMIT AT LIMITS+4
1920 ;(+SPRITE NUMBER*6)
1930 ;-----------------------
1940          JSR $AEFD
1950          JSR $B79E
1960          LDY NSPRB6
1970          TXA
1980          STA LIMITS+4,Y
1990 ;-----------------------
2000 ;STORE UPPER Y LIMIT AT LIMITS+5
2010 ;(+SPRITE NUMBER*6)
2020 ;-----------------------
2030          JSR $AEFD
2040          JSR $B79E
2050          LDY NSPRB6
2060          TXA
2070          STA LIMITS+5,Y
2080 ;-----------------------
2090 ;--GET IN IRQ RATE--
2100 ;IF NO PARAMETER THEN DEFAULT RATE
2110 ;$411A (16666)/STORE IN SPEED,+1
2120 ;-----------------------
2130          LDX #$1A
```

```
2140          LDY #$41
2150          JSR $0079
2160          BEQ DFL
2170          JSR $AEFD
2180          JSR $AD8A
2190          JSR $B7F7
2200          LDX $14
2210          LDY $15
2220 DFL      STX SPEED
2230          STY SPEED+1
2240 ;-----------------------
2250 ;--ENABLE JSPRITE--
2260 ;-----------------------
2270 ;IF ALL PARAMETERS ENTERED THEN
2280 ;LINK SPRITE TO JOYSTICK BY
2290 ;SETTING BIT IN ENABEL BYTE
2300 ;-----------------------
2310          LDX NSPR
2320          LDA ENABEL
2330          ORA MAASK,X
2340          STA ENABEL
2350 ;-----------------------
2360 ;--RE-VECTOR IRQ INTERRUPT--
2370 ;CHANGE IRQ VECTOR TO POINT TO
2380 ;OUR CODE/INITIALIZE TIMER WITH
2390 ;SET SPEED
2400 ;-----------------------
2410          SEI
2420          LDA #<IRQ
2430          LDY #>IRQ
2440          STA $0314
2450          STY $0315
2460          LDA SPEED
2470          LDY SPEED+1
2480          STA $DC04
2490          STY $DC05
2500          CLI
2510          RTS
2520 ;-----------------------
2530 ;--READ JOYSTICK--
2540 ;MASK OUT UPPER 4 BITS AND FLIP
2550 ;LOWER 4
2560 ;-----------------------
2570 IRQ      LDA $DC00
```

```
2580          AND #$0F
2590          EOR #$0F
2600          BEQ EXSH2
2610 ;-----------------------
2620 ;--SETUP INDIRECT SUBROUTINE--
2630 ;MULTIPLY JOYSTICK VALUE BY 2 AND
2640 ;USE IT TO INDEX ACTION TABLE OF
2650 ;'DIRECTION-HANDLING' ROUTINE
2660 ;ADDRESSES/STORE ADDRESS IN
2670 ;RUTEEN+2,+3
2680 ;-----------------------
2690 SH2      ASL A
2700          TAX
2710          LDA ACTION,X
2720          LDY ACTION+1,X
2730          STA RUTEEN+1
2740          STY RUTEEN+2
2750 ;-----------------------
2760 ;--SPRITE MOVE SELECTION LOOP--
2770 ;SELECT WHICH SPRITES TO MOVE BY
2780 ;ROTATING ENABEL BYTE - CARRY SET
2790 ;INDICATES MOVE CORRESPONDING
2800 ;SPRITE. CARRY CLEAR MEANS DO NOT
2810 ;MOVE./EXIT AFTER 8 ROTATES/
2820 ;-----------------------
2830          CLC
2840          LDX #$08
2850 SHLOOP   DEX
2860          BMI EXSH
2870          ROL ENABEL
2880          BCC SHLOOP
2890 ;-----------------------
2900 ;MULTIPLY LOOP COUNTER BY 2 (IN
2910 ;X) AND BY 6 (IN .Y)
2920 ;-----------------------
2930          TXA
2940          PHA
2950          ASL A
2960          TAX
2970          STA NSPRB6
2980          ASL A
2990          CLC
3000          ADC NSPRB6
3010          TAY
```

```
3020 ;-----------------------
3030 ;--MOVE SPRITE IN GIVEN DIRECTION-
3040 ;JSR TO SELECTED ROUTINE TO MOVE
3050 ;SPRITE X IN THE JOYSTICK DIREC.
3060 ;-----------------------
3070         JSR RUTEEN
3080 ;-----------------------
3090 ;RECOVER LOOP COUNTER FROM STACK
3100 ;/SET CARRY/ JUMP BACK TO TOP OF
3110 ;LOOP/
3120 ;-----------------------
3130         PLA
3140         TAX
3150         SEC
3160         BCS SHLOOP
3170 ;-----------------------
3180 ;--EXIT LOOP--
3190 ;ROTATE ENABEL ONCE MORE TO
3200 ;RESTORE VALUE/
3210 ;-----------------------
3220 EXSH    ROL ENABEL
3230 ;-----------------------
3240 ;EXIT TO BASIC INTERRUPT HANDLER
3250 ;-----------------------
3260 EXSH2   JMP $EA31
3270 ;--------------------
3280 ;--JOYSTICK DIRECTION ROUTINES--
3290 ;IF LIMIT IS REACHED THEN DO NOT
3300 ;MOVE
3310 ;--------------------
3320 ;--MOVE NORTH-EAST--
3330 ;--------------------
3340 NES     JSR EAS
3350 ;--------------------
3360 ;--MOVE NORTH--
3370 ;--------------------
3380 NOR     LDA $D001,X
3390         CMP LIMITS+4,Y
3400         BEQ NRX
3410         DEC $D001,X
3420 NRX     RTS
3430 ;--------------------
3440 ;--MOVE SOUTH-EAST--
3450 ;--------------------
```

```
3460 SES     JSR SOU
3470 ;---------------------
3480 ;--MOVE EAST--
3490 ;---------------------
3500 EAS     LDA $D000,X
3510         CMP LIMITS+2,Y
3520         BNE ECON
3530         LDA $D010
3540         AND MAASK,X
3550         CMP LIMITS+3,Y
3560         BEQ EAX
3570 ;----------
3580 ECON    INC $D000,X
3590         BNE EAX
3600         LDA MAASK,X
3610         ORA $D010
3620         STA $D010
3630 EAX     RTS
3640 ;---------------------
3650 ;--MOVE SOUTH-WEST--
3660 ;---------------------
3670 SWE     JSR WES
3680 ;---------------------
3690 ;--MOVE SOUTH--
3700 ;---------------------
3710 SOU     LDA $D001,X
3720         CMP LIMITS+5,Y
3730         BEQ SRX
3740         INC $D001,X
3750 SRX     RTS
3760 ;---------------------
3770 ;--MOVE NORTH-WEST--
3780 ;---------------------
3790 NWE     JSR NOR
3800 ;---------------------
3810 ;--MOVE WEST--
3820 ;---------------------
3830 WES     LDA $D000,X
3840         CMP LIMITS,Y
3850         BNE WCON
3860         LDA $D010
3870         AND MAASK,X
3880         CMP LIMITS+1,Y
3890         BEQ WEX
```

```
3900 ;----------
3910 WCON    LDA $D000,X
3920         BNE WCON2
3930         LDA MAASK,X
3940         EOR #$FF
3950         AND $D010
3960         STA $D010
3970 WCON2   DEC $D000,X
3980 WEX     RTS
3990 ;--------------------
4000 ;--2 BYTE BIT MASK TABLE--
4010 ;--------------------
4020 MAASK   .BYTE $01,$00,$02,$00
4030         .BYTE $04,$00,$08,$00
4040         .BYTE $10,$00,$20,$00
4050         .BYTE $40,$00,$80,$00
4060 ;--------------------
4070 ;--TABLE OF DIRECTION ADRRESSES--
4080 ;--------------------
4090 ACTION .WORD $0000,NOR,SOU
4100         .WORD $0000,WES,NWE
4110         .WORD SWE,$0000,EAS
4120         .WORD NES,SES
4130 ;--------------------
4140 ;--STORAGE SPACE--
4150 ;--------------------
4160 RUTEEN JMP $0000
4170 NSPR    NOP
4180 NSPRB6 NOP
4190 ENABEL BRK
4200 LIMITS *=*+$30
4210 ;--------------------
```

BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 36095+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
```

```
110 REM --- BLOCK 1
111 DATA208,33,162,40,169,0,157,181
112 DATA142,202,16,250,120,169,49,160
113 DATA234,141,20,3,140,21,3,169
114 DATA26,160,65,141,4,220,140,5
115 DATA220,88,96,32,158,183,138,41
116 DATA7,10,141,179,142,10,24,109
117 DATA179,142,141,180,142,32,121,0
118 DATA208,15,174,179,142,189,138,142
119 DATA-7252
120 REM --- BLOCK 2
121 DATA73,255,45,181,142,141,181,142
122 DATA96,32,253,174,32,138,173,32
123 DATA247,183,172,180,142,165,20,153
124 DATA182,142,165,21,240,6,174,179
125 DATA142,189,138,142,153,183,142,32
126 DATA253,174,32,138,173,32,247,183
127 DATA172,180,142,165,20,153,184,142
128 DATA165,21,240,6,174,179,142,189
129 DATA-9088
130 REM --- BLOCK 3
131 DATA138,142,153,185,142,32,253,174
132 DATA32,158,183,172,180,142,138,153
133 DATA186,142,32,253,174,32,158,183
134 DATA172,180,142,138,153,187,142,162
135 DATA26,160,65,32,121,0,240,13
136 DATA32,253,174,32,138,173,32,247
137 DATA183,166,20,164,21,142,54,3
138 DATA140,55,3,174,179,142,173,181
139 DATA-8351
140 REM --- BLOCK 4
141 DATA142,29,138,142,141,181,142,120
142 DATA169,224,160,141,141,20,3,140
143 DATA21,3,173,54,3,172,55,3
144 DATA141,4,220,140,5,220,88,96
145 DATA173,0,220,41,15,73,15,240
146 DATA49,10,170,189,154,142,188,155
147 DATA142,141,177,142,140,178,142,24
148 DATA162,8,202,48,26,46,181,142
149 DATA-7166
150 REM --- BLOCK 5
151 DATA144,248,138,72,10,170,141,180
152 DATA142,10,24,109,180,142,168,32
153 DATA176,142,104,170,56,176,227,46
```

```
154 DATA181,142,76,49,234,32,47,142
155 DATA189,1,208,217,186,142,240,3
156 DATA222,1,208,96,32,84,142,189
157 DATA0,208,217,184,142,208,11,173
158 DATA16,208,61,138,142,217,185,142
159 DATA-8322
160 REM --- BLOCK 6
161 DATA240,14,254,0,208,208,9,189
162 DATA138,142,13,16,208,141,16,208
163 DATA96,32,99,142,189,1,208,217
164 DATA187,142,240,3,254,1,208,96
165 DATA32,32,142,189,0,208,217,182
166 DATA142,208,11,173,16,208,61,138
167 DATA142,217,183,142,240,19,189,0
168 DATA208,208,11,189,138,142,73,255
169 DATA-8434
170 REM --- BLOCK 7
171 DATA45,16,208,141,16,208,222,0
172 DATA208,96,1,0,2,0,4,0
173 DATA8,0,16,0,32,0,64,0
174 DATA128,0,0,0,32,142,84,142
175 DATA0,0,99,142,96,142,81,142
176 DATA0,0,47,142,29,142,44,142
177 DATA76,0,0,234,234,0
178 DATA-3607
179 DATA END
1000 REM -- ENABLE NEW COMMAND --
1001 REM - JSPRITE
1002 POKE 33335,255:POKE 33336,140
1003 END
```

The JSPRITE Routine

In the introductory chapters of the book it was mentioned that BASIC used something call a 'Timed 1/60th second IRQ Interrupt' to handle such jobs as scanning the keyboard for keypresses and updating the jiffy clock. However, no detail was provided as to how this may be used by the machine code programmer. JSPRITE is a perfect example of its use. You may recall that when a processor interrupt occurs the 6510 suspends its current operation and jumps to ($FFFE)) This vector is in ROM and is therefore immutable in the standard configuration 64, although it is possible to change this vector by re-configuring the 64 and switching out the Kernal ROM. However, the 64 operating system makes the system IRQ available to the programmer via a second vector at $0314 that is in RAM and may hence be altered. IRQ

wedge programs change the vector at $0314 to point to their own code and having completed their operations then jump to the default system vector. Unless the wedge program is doing a very large amount of work its presence will be transparent to the user and it should not affect normal BASIC processing. The JSPRITE routine is just such an IRQ wedge. It uses the IRQ to sample the input from joystick 2, and on the basis of that input it alters the positions of any sprites that have been previously linked to it.

Get in Sprite Number Parameter (Lines 1100–1560)

The sprite number is input in lines 1340–1400. It is subsequently stored, multiplied by 2 in NSPR and multiplied by 6 in NSPRB6. Note that the multiplication by 6 is achieved by adding the number*2 result to the number*4 result. In lines 1480–1490 a check is made for the end of the input buffer. If there are no more parameters then the corresponding sprite bit is ANDed out of the JSPRITE Enable register ENABEL and we return to BASIC. If there are no parameters at all to JSPRITE the ENABEL and LIMITS tables are cleared, the IRQ wedge is disabled and the IRQ rate is set back to the default value of 16666 (1/60th second).

Get In Sprite Boundary Limits (Lines 1630–2070)

If there are LIMITS parameters then they are placed into the area of memory corresponding to that particular sprite's boundary limits. Each sprite requires six bytes to define its horizontal and vertical boundary limits, i.e. two x co-ordinates (each of two bytes) and two y co-ordinates The *high* x co-ordinates are stored in separate bytes as their respective sprite bit masks to simplify later comparison.

Get In IRQ Rate (Lines 2130–2230)

The Interrupt rate parameter is optional and if it is absent then a default rate of 16666 or $411A is assumed. The rate parameter is stored in SPEED.

Link Sprite to Joystick (Lines 2310–2340)

If all the limits parameters were entered successfully then the sprite is linked into JSPRITE control by setting the appropriate bit in the ENABEL register. Note that the table of bitmasks MAASK is a two-byte table. This is so merely for the convenience of not having to require a third sprite pointer. Having 'enabled' the sprite the IRQ wedge is set up in lines 2410–2510 by first setting (SEI) the IRQ disable bit of the P register in line 2410 and then changing the IRQ vector to line 2570

and setting the IRQ rate to equal SPEED. Before exiting to BASIC the IRQ disable bit is cleared.

The IRQ Wedge (Lines 2570–3980)

Every 1/60th of a second (or whatever rate is set) the processor is directed to line 2570, where the joystick input is read from Data Port A of the CIA#1 chip, at $DC00. If there is no input then the program exits directly to the system IRQ vector in line 3260. If there is an input then the value is multiplied by two and used to index an action table of direction handling routines. This address is stored at RUTEEN+1 and RUTEEN+2 for the 'indirect' subroutine RUTEEN. The joystick data refers to bits 0 to 4 of the data byte, representing 'up', 'down', 'left' and 'right' switches respectively. Each bit is normally set (1) and is reset (0) if a switch is closed. ANDing with #$0F sets the high nybble of the byte to zero. EORing with #$0F swaps 0's and 1's in the byte, giving a value for the switch or combination of switches activated.

Sprite Selection (Lines 2830–3160)

Lines 2830–2880 determine which sprites are to be moved by rotating the ENABEL register in a loop indexed by X. If the result of the ROL operation is a *set* Carry flag this indicates that the bit corresponding to sprite X was set in the ENABEL register and it is hence to be moved in the direction indicated by the indirect subroutine RUTEEN. If the carry is clear then sprite X is not to be moved. In lines 2930–3010 X values for those sprites which are to be moved are multiplied by two (in X) and by six (in Y) to index the Sprite Position and LIMITS tables respectively and the sprite is moved in the chosen joystick direction by a JSR RUTEEN in line 3070. Upon returning the loop is resumed until ENABEL has been rotated eight times when it exits to line 3220 where ENABEL is rotated once more to restore its starting value. At line 3260 the IRQ wedge exits to the system interrupt routine.

Direction Handling Routines (Lines 3340–3980)

The names of these routines follow the compass directions. There are four fundamental directions: NOR at line 3380, SOU at line 3710, EAS at line 3500 and WES at line 3830. Intermediate directions are obtained by combining two of the four fundamentals. What each routine does is compare each sprite position with its LIMIT value. If it has reached its limit then it is *not* moved. On the other hand, if it has not reached its limit then it *is* moved.

Variable Storage and Interrupts

When writing IRQ routines you have to be very careful about variable

storage. You must either use memory locations that you know the main program will not be using or, if you are forced to use variables in common with the main program (such as certain Zero-page locations) then they must be stacked upon entering the interrupt and restored upon exiting. If this is not done you are almost certain to get conflict between the two. JSPRITE stores all its data for use during the interrupt just after the JSPRITE code (in lines 4160–4200), where it will not be touched by any other program.

The SOUND Command

Just as is the case with sprites the 64's sound facilities are normally only available to the BASIC programmer through a mountain of POKE commands. The problems are worse with sound in that the SID chip is finicky about the *order* in which POKEs are made. For instance, POKEing the volume to zero immediately *after* a note can result in ugly unmusical clicks, while 'gateing' the waveform *before* setting up the envelope produces no sound at all! it is better of course to have a single SOUND command to take care of everything simply and efficiently.

SOUND

Syntax: SOUND (voice), (pitch), (volume), (A), (D), (S), (R), (waveform), (pulse-rate)

Where: (voice) defines one of the three SID chip voices, numbered 0–2
(pitch) defines the frequency register value from 0–65535 of the selected voice oscillator
(volume) defines the volume of sound from 0 (silent) to 15 (loudest)
(A) defines the Attack parameter of the sound envelope 0–15
(D) defines the Decay parameter of the sound envelope 0–15
(S) defines the Sustain parameter of the sound envelope 0–15
(R) defines the Release parameter of the sound envelope 0–15
(waveform) defines the type of sound produced: 0=Triangle, 1=Sawtooth, 2=Pulse and 3=Noise
(pulse-rate) defines the form of the pulse. This only applies if the Pulse waveform (2) is selected. *Effective* range is 0–4096

A table of frequency register values for musical notes is provided in Appendix 6 with the corresponding audible frequencies. Like the SPRITE command considered earlier SOUND does not require all of the parameters listed above to be entered each time. Thus it is possible to obtain a note with just SOUND 0,4000, for example. Unlike SPRITE, however, where parameters not entered are simply absent, in SOUND a series of default values are established, since the SID chip generally requires that all these parameters be initialized for each sounding of a note. The defaults for the minimum command of SOUND (voice), (pitch) are for Triangle waveform, A,D,S,R and values of 0,9,0,0 and maximum volume. Note that if any one of the envelope parameters (A, D, S, and R) are entered then they must *all* be entered, and that if the waveform value of 2 (Pulse waveform) is selected then a pulse rate *must* also be entered. After sounding a note the sound is not switched off but allowed to decay naturally, which can result in a constant buzz for certain envelopes. If this is the case then the sound will have to be switched off after an appropriate delay by a SOUND voice command e.g. to switch off voice 0 use SOUND 0. SOUND with *no* parameters has the effect of clearing all SID registers and putting SID to sleep with a click.

Multiple voices are no problem. Simply have two or three SOUND commands, one after the other. For instance:

```
10 SOUND 0,4461
20 SOUND 1,5619
30 SOUND 2,6675
```

will play a chord in the key of C.

The SOUND Code

Assembler Listing

```
1000 *=$8F00
1010 ;--------------------
1020 WAVEF   =$0336
1030 VOLUM   =$0337
1040 VOIC    =$0338
1050 ;--------------------
1060 ;--SWITCH OFF ALL VOICES--
1070 ;IF NO PARAMETERS THEN CLEAR ALL
1080 ;SID REGISTERS
1090 ;--------------------
```

```
1100 SOUN   BNE SOUN2
1110 ;--------------------
1120        LDX #$18
1130        LDA #$00
1140 SOULP  STA $D400,X
1150        DEX
1160        BPL SOULP
1170        RTS
1180 ;--------------------
1190 ;--GET IN 'VOICE' PARAMETER
1200 ;REJECT VALUES GREATER THAN 2/
1210 ;CLEAR WAVEFORM REGISTER FOR
1220 ;CHOSEN VOICE/
1230 ;--------------------
1240 SOUN2  JSR $B79E
1250        CPX #$03
1260        BCS ERSOU
1270        STX VOIC
1280        LDA #$00
1290        LDY VOICB7,X
1300        STA $D404,Y
1310 ;--------------------
1320 ;IF NO FURTHER PARAMETERS THEN
1330 ;EXIT
1340 ;--------------------
1350        JSR $0079
1360        BNE SOUN3
1370        RTS
1380 ;--------------------
1390 ;--'ILLEGAL QUANTITY' ERROR EXIT-
1400 ;--------------------
1410 ERSOU  LDX #$0E
1420        JMP ($0300)
1430 ;--------------------
1440 ;--GET IN 'PITCH' PARAMETER
1450 ;OSCILLATOR FREQUENCY IS STORED
1460 ;IN REGISTERS OF CHOSEN VOICE
1470 ;--------------------
1480 SOUN3  JSR $AEFD
1490        JSR $AD8A
1500        JSR $B7F7
1510        LDX VOIC
1520        LDY VOICB7,X
1530        LDA $14
```

```
1540          STA $D400,Y
1550          LDA $15
1560          STA $D401,Y
1570 ;--------------------
1580 ;--GET VOLUME PARAMETER--
1590 ;DEFAULTS TO MAXIMUM IF NO PARAM
1600 ;--------------------
1610 DFOLT    LDA #$0F
1620          ORA $D418
1630          STA $D418
1640          JSR $0079
1650          BEQ DFOLT0
1660 ;----------
1670          JSR $AEFD
1680          JSR $B79E
1690          STX VOLUM
1700          LDA $D418
1710          AND #$F0
1720          ORA VOLUM
1730          STA $D418
1740 ;--------------------
1750 ;--GET ENVELOPE--
1760 ;IF NO PARAMETER DEFAULTS TO
1770 ;ADSR 0,9,0,0
1780 ;--------------------
1790 DFOLT0 LDX VOIC
1800          LDY VOICB7,X
1810          LDA #$09
1820          STA $D405,Y
1830          LDA #$00
1840          STA $D406,Y
1850          JSR $0079
1860          BEQ DFOLT1
1870 ;----------
1880          JSR ATKDCY
1890          LDX VOIC
1900          LDY VOICB7,X
1910          STA $D405,Y
1920          JSR ATKDCY
1930          LDX VOIC
1940          LDY VOICB7,X
1950          STA $D406,Y
1960 ;--------------------
1970 ;--GET WAVEFORM--
```

```
1980 ;DEFAULT WAVEFORM IS 0=TRIANGLE
1990 ;1=SAWTOOTH 2=PULSE 3=NOISE
2000 ;--------------------
2010 DFOLT1 LDA #$11
2020        STA WAVEF
2030        JSR $0079
2040        BEQ DFOLT2
2050 ;----------
2060        JSR $AEFD
2070        JSR $B79E
2080        TXA
2090        AND #$03
2100        TAX
2110        LDA WAVE,X
2120        STA WAVEF
2130        CMP #$41
2140        BNE DFOLT2
2150 ;--------------------
2160 ;IF WAVE WAS PULSE THEN GET IN
2170 ;PULSE RATE
2180 ;--------------------
2190        JSR $AEFD
2200        JSR $AD8A
2210        JSR $B7F7
2220        LDX VOIC
2230        LDY VOICB7,X
2240        LDA $14
2250        STA $D402,Y
2260        LDA $15
2270        AND #$0F
2280        STA $D403,Y
2290 ;--------------------
2300 ;--SOUND THE NOTE--
2310 ;SOUNDING IS ENABLED BY GATING
2320 ;THE WAVEFORM VALUE
2330 ;--------------------
2340 DFOLT2 LDX VOIC
2350        LDY VOICB7,X
2360        LDA WAVEF
2370        STA $D404,Y
2380        RTS
2390 ;--------------------
2400 ;--GET IN A,D OR S,R VALUES--
2410 ;GETS IN 2 NYBBLES AND COMBINES
2420 ;THEM INTO 1 BYTE
```

```
2430 ;--------------------
2440 ATKDCY JSR $AEFD
2450        JSR $B79E
2460        TXA
2470        ASL A
2480        ASL A
2490        ASL A
2500        ASL A
2510        STA WAVEF
2520        JSR $AEFD
2530        JSR $B79E
2540        TXA
2550        AND #$0F
2560        ORA WAVEF
2570        RTS
2580 ;--------------------
2590 ;--TABLE OF WAVFORM VALUES--
2600 ;--------------------
2610 WAVE   .BYTE $11,$21,$41,$81
2620 ;--------------------
2630 ;--VOICE*3 TABLE--
2640 ;--------------------
2650 VOICB7 .BYTE $00,$07,$0E
2660 ;--------------------
```

BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 36607+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA208,11,162,24,169,0,157,0
112 DATA212,202,16,250,96,32,158,183
113 DATA224,3,176,17,142,56,3,169
114 DATA0,188,246,143,153,4,212,32
115 DATA121,0,208,6,96,162,14,108
116 DATA0,3,32,253,174,32,138,173
```

```
117 DATA32,247,183,174,56,3,188,246
118 DATA143,165,20,153,0,212,165,21
119 DATA-7176
120 REM --- BLOCK 2
121 DATA153,1,212,169,15,13,24,212
122 DATA141,24,212,32,121,0,240,20
123 DATA32,253,174,32,158,183,142,55
124 DATA3,173,24,212,41,240,13,55
125 DATA3,141,24,212,174,56,3,188
126 DATA246,143,169,9,153,5,212,169
127 DATA0,153,6,212,32,121,0,240
128 DATA24,32,215,143,174,56,3,188
129 DATA-6885
130 REM --- BLOCK 3
131 DATA246,143,153,5,212,32,215,143
132 DATA174,56,3,188,246,143,153,6
133 DATA212,169,17,141,54,3,32,121
134 DATA0,240,47,32,253,174,32,158
135 DATA183,138,41,3,170,189,242,143
136 DATA141,54,3,201,65,208,27,32
137 DATA253,174,32,138,173,32,247,183
138 DATA174,56,3,188,246,143,165,20
139 DATA-7870
140 REM --- BLOCK 4
141 DATA153,2,212,165,21,41,15,153
142 DATA3,212,174,56,3,188,246,143
143 DATA173,54,3,153,4,212,96,32
144 DATA253,174,32,158,183,138,10,10
145 DATA10,10,141,54,3,32,253,174
146 DATA32,158,183,138,41,15,13,54
147 DATA3,96,17,33,65,129,0,7
148 DATA14
149 DATA-5147
150 DATA END
1000 REM -- ENABLE NEW COMMAND --
1001 REM - SOUND
1002 POKE 33337,255:POKE 33338,142
1003 END
```

The SOUND Routine

Get Voice Parameter (Lines 1100–1370)

If there are *no* parameters for the SOUND command it causes all the

SID chip registers to be cleared in a loop at lines 1120–1170 before exit to BASIC. Parameter values greater than two are rejected and lead to an error exit at lines 1410–1420. If the (voice) parameter is legal then it is first stored in VOIC, and then used to index the VOICE*7 table VOICB7 and the waveform register for the chosen voice is cleared in lines 1240–1300. Note, the VOICB7 table is used because the corresponding SID registers for the three separate voices are stored seven bytes apart from each other. So, for example, the waveform register for voice 0 is at $D404 + 0, that for voice 1 at $D404 + 7 and for voice 2, $D404 + 14. If there are no further parameters after (voice) then exit to BASIC.

Get Pitch Parameter (Lines 1480–1560)

The (pitch) parameter is stored in the corresponding voice frequency registers.

Get Volume Parameter (Lines 1610–1730)

There is only one volume register for all three voices, and so we simply input our value and store it in the appropriate register. However, the volume register is a 'shared' register, with Filter Mode selection bits in the high nybble, and so we must be careful not to disturb these other values when we change the volume. This parameter is optional and the default setting is maximum (15).

Get Envelope Parameters (Lines 1790–1950)

The Attack/Decay and Sustain/Release values are pulled in by a common routine, ATKDCY, at line 2440. This routine gets in two bytes from 0–15 separated by commas and combines then into a single byte value. As with (volume), these parameters are also optional, the default values being 0,9,0,0 for (A), (D), (S), (R). However, if you enter one of them then you must enter all four.

Get Waveform Parameter (Lines 2010–2140)

The default value is 0, Triangle. 1 gives Sawtooth, 2, Pulse and 3, Noise. If (waveform) was 2, Pulse, then you *must* subsequently enter the pulse rate or a ?SYNTAX ERROR is generated. The value may be up to 65535, but the correct range within which the pulse waveform varies is 0–4096, since the register is a 12-bit location.

Sounding the Note (Lines 2340–2380)

Once all the parameters are entered the note is sounded by 'gating'

the chosen waveform value in its corresponding waveform register and then exiting to BASIC.

The SCROLL Command

SCROLL is a command that facilitates some interesting screen effects for games, etc. It allows the screen contents to be physically scrolled by one character in the upwards, left or right direction, depending on the parameter. In the case of left and right scrolling both the character that is scrolled onto the screen and its colour may be defined as parameters, the default character being a space.

SCROLL

Syntax: SCROLL (direction), (char code), (colour)

Where: (direction) defines scroll direction: 0, Left or 1, Right
(char code) defines the screen code of the character that will be scrolled onto the screen from left or right
(colour) defines the colour of the scroll character

SCROLL without any parameters performs an upwards scroll, and it is not possible to change the 'space' scroll character, which is also the default character for left and right scrolling if no parameters follow the direction. The default colour is the current text colour.

Potential Problems with SCROLL

Excepting those already discussed in relation to loading for other commands the only real problem you might find in using SCROLL is that of 'glitch'. If you try to use these scroll routines in concert with the VIC II scroll registers to get a smooth scroll as described in the *Programmer's Reference Guide* you will find it very disappointing. You will remember from the introductory chapter on the hardware that I discussed using the raster register of the VIC II chip to eliminate glitching in graphics programs. Unfortunately, in the case of the scroll this is not really good enough, since the time it takes to scroll the screen is greater than the available interval of one raster scan period. Several solutions to the problem exist. One way is to write much faster scroll routines by avoiding the use of programming loops and indirect addressing and instead generating thousands of LDA BYTE/STA BYTE sequences. However, even using the fastest possible routines there is not time within one raster scan to scroll both screen and colour table and furthermore you will find that about 4K of RAM will be taken up with each scroll routine.

The obvious way around this is either (a) to scroll only a portion of the screen or (b) to avoid scrolling the colour table by having a

monochrome screen or working entirely in multicolour. Such routines would make use of the raster interrupt techniques demonstrated in the multiple sprite routine of Appendix 8 to ensure that all scrolling occurred either behind or in front of the raster line.

If such compromises do not suit you and you are prepared to lose some scrolling speed then another effective method is to allow the scroll to occur over more than one raster scan by scrolling between two separate screens of data and raster-synchronising a switch between two screens, i.e. do the scrolling on the screen that is not being displayed and then when the scroll is complete flip screens. This requires that RAM be made available to house the second screen, usually done by moving the bottom of BASIC up by 1024 bytes.

It is really a rotten shame that Commodore did not see fit to make the 64 a 2MHz rather than 1MHz machine in which case scrolling would not be such a problem. On the whole, considering the inconvenience involved in these methods, unless you happen to be writing the latest improved version of 'Defender 64' I would learn to live with the glitch.

The SCROLL Code

Assembler Listing

```
1000 *=$9200
1010 ;---------
1020 SSRC    =$4E;*
1030 CSRC    =$50;*
1040 SCRRL   =$02A7;**
1050 INSER   =$0336
1060 INCOL   =$0337
1070 ;--------------
1080 ;--GET DIRECTION PARAMETER--
1090 ;IF=ZERO JUMP TO UPS/
1100 ;IF =1 OR =2 MULTIPLY BY 2 AND USE
1110 ;TO INDEX THE DIREC TABLE FOR THE
1120 ;ADDRESS OF THE SCROLL ROUTINE/
1130 ;STORE ADDRESS IN SCRRL+1 AND +2
1140 ;AND PUT A JMP (#$4C) IN SCRRL/
1150 ;JUMP TO COMSCR/
1160 ;--------------
1170 SCRO    JSR $0079
1180         BEQ UPS
```

```
1190 ;----------
1200         JSR $B79E
1210         TXA
1220         AND #$01
1230         ASL A
1240         TAX
1250         LDA DIREC,X
1260         STA SCRRL+1
1270         LDA DIREC+1,X
1280         STA SCRRL+2
1290         LDA #$4C
1300         STA SCRRL
1310         JMP COMSCR
1320 ;--------------
1330 ;--SCROLL THE SCREEN UP--
1340 :ROM ROUTINE TO SCROLL UP
1350 ;--------------
1360 UPS     JMP $E8EA
1370 ;--------------
1380 ;--GET SCROLL PARAMETER--
1390 ;GET IN ASCII CHARACTER TO BE
1400 ;SCROLLED ONTO THE SCREEN/IF NO
1410 ;PARAMETER THEN ASSUME 'SPACE
1420 ;--------------
1430 COMSCR LDX #$20
1440         JSR $0079
1450         BEQ COMSC2
1460         JSR $AEFD
1470         JSR $B79E
1480 COMSC2 STX INSER
1540 ;--------------
1550 ;--GET COLOUR FOR SCROLL PARAM--
1560 ;ASSUME COLOUR UNDER CURSOR IF
1570 ;NONE
1580 ;--------------
1590         LDX $0286
1600         JSR $0079
1610         BEQ COMSC3
1620         JSR $AEFD
1630         JSR $B79E
1640 COMSC3 STX INCOL
1650 ;--------------
```

```
1660 ;--COMMON LOOP FOR L AND R SCROLL-
1670 ;INITIALIZE 2 POINTERS - SSRC INTO
1680 ;THE SCREEN TABLE AND CSRC INTO
1690 ;THE COLOUR TABLE/JUMP 25 TIMES TO
1700 ;INDIRECT LINE SCROLL ROUTINE
1710 ;(EITHER LEFT OF RIGHT) ADDING 40
1720 ;TO THE POINTERS FOR EVERY LINE/
1730 ;RETURN TO BASIC
1740 ;--------------
1750         LDA #$04
1760         LDX #$D8
1770         LDY #$00
1780         STA SSRC+1
1790         STX CSRC+1
1800         STY SSRC
1810         STY CSRC
1820 ;----------
1830         LDX #$18
1840 SCLOOP  JSR SCRRL
1850         CLC
1860         LDA SSRC
1870         ADC #$28
1880         STA SSRC
1890         STA CSRC
1900         BCC W22
1910         INC SSRC+1
1920         INC CSRC+1
1930 W22     DEX
1940         BPL SCLOOP
1950         RTS
1960 ;---------------
1970 ;--SCROLL A LINE RIGHT--
1980 ;SCROLL 39 CHARACTERS 1 POSITION
1990 ;TO THE RIGHT AND PUT THE INSERT
2000 ;CHARACTER IN THE LEFTMOST
2010 ;POSITION
2020 ;---------------
2030 SCRRIT  LDY #$26
2040 LOOR    LDA (CSRC),Y
2050         PHA
2060         LDA (SSRC),Y
2070         INY
2080         STA (SSRC),Y
2090         PLA
```

```
2100          STA (CSRC),Y
2110          DEY
2120          DEY
2130          BPL LOOR
2140          INY
2150          LDA INSER
2160          STA (SSRC),Y
2170          LDA INCOL
2180          STA (CSRC),Y
2190          RTS
2200 ;---------------
2210 ;--SCROLL A LINE LEFT--
2220 ;MOVE 40 CHARACTERS 1 POSITION TO
2230 ;THE LEFT AND PUT THE INSERT
2240 ;CHARACTER AT THE RIGHTMOST
2250 ;POSITION/
2260 ;---------------
2270 SCRLFT LDY #$01
2280 LOOL   LDA (CSRC),Y
2290          PHA
2300          LDA (SSRC),Y
2310          DEY
2320          STA (SSRC),Y
2330          PLA
2340          STA (CSRC),Y
2350          INY
2360          INY
2370          CPY #$28
2380          BNE LOOL
2390          DEY
2400          LDA INSER
2410          STA (SSRC),Y
2420          LDA INCOL
2430          STA (CSRC),Y
2440          RTS
2450 ;---------------
2460 ;LEFT AND RIGHT SCROLL ROUTINES
2470 ;FOR THE INDIRECT SUBROUTINE SCRRL
2480 ;---------------
2490 DIREC   .WORD SCRLFT,SCRRIT
2500 ;---------------
```

BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 37375+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA32,121,0,240,28,32,158,183
112 DATA138,41,1,10,170,189,167,146
113 DATA141,168,2,189,168,146,141,169
114 DATA2,169,76,141,167,2,76,36
115 DATA146,76,234,232,162,32,32,121
116 DATA0,240,6,32,253,174,32,158
117 DATA183,142,54,3,174,134,2,32
118 DATA121,0,240,6,32,253,174,32
119 DATA-6961
120 REM --- BLOCK 2
121 DATA158,183,142,55,3,169,4,162
122 DATA216,160,0,133,79,134,81,132
123 DATA78,132,80,162,24,32,167,2
124 DATA24,165,78,105,40,133,78,133
125 DATA80,144,4,230,79,230,81,202
126 DATA16,235,96,160,38,177,80,72
127 DATA177,78,200,145,78,104,145,80
128 DATA136,136,16,241,200,173,54,3
129 DATA-7134
130 REM --- BLOCK 3
131 DATA145,78,173,55,3,145,80,96
132 DATA160,1,177,80,72,177,78,136
133 DATA145,78,104,145,80,200,200,192
134 DATA40,208,239,136,173,54,3,145
135 DATA78,173,55,3,145,80,96,136
136 DATA146,107,146
137 DATA-4963
138 DATA END
1000 REM -- ENABLE NEW COMMAND --
1001 REM - SCROLL
1002 POKE 33357,255:POKE 33358,145
1003 END
```

The SCROLL Routines

Get Parameters (Lines 1170–1640)

If SCROLL has no parameters then we simply jump to line 1360, where we exit to the ROM SCROLL (*up*) routine. There is also a ROM routine to scroll *down* located at $E981 but since this upsets the screen editor pointers if used repeatedly it is not included in the routine here. If the SCROLL command has a parameter this is ANDed with #$01 and multiplied by two to provide an index into the word table DIREC, which holds two addresses for the 'indirect' subroutine at SCRRL. The chosen address is stored at SCRR+1 and SCRRL+2 and a JMP absolute (#$4C) is placed at SCRRL. At lines 1430–1480 the SCROLL character is pulled in. This is the character that will be scrolled onto the screen. A second parameter, for the colour of the SCROLL character, is entered at lines 1590–1640. Both of these parameters are optional, the defaults being a space character of colour equal to the current text colour.

Common SCROLL Loop (Lines 1750–1950)

In lines 1750–1810 two Zero-page pointers are set up, one into the screen memory, SSRC and the other into the colour table, CSRC. In lines 1830–1950 a loop is set up which JSR's 25 times to the 'indirect' line scroll routine SCRRL. For every line scrolled 40 is added to the value of the pointers, and upon terminating this loop the whole screen has been scrolled so we exit to BASIC.

Individual Line SCROLL Routines (Lines 2030–2440)

In lines 2030–2190 a line is scrolled from *left to right* and the scroll character enters from the *left*, while at lines 2270–2440 a screen line is scrolled from *right to left* and the SCROLL character enters the screen from the *right*.

The Hi-res Commands

This group of command routines, consisting of the BITMAPC, BITMAP, MOVE, PLOT, UNPLOT, DRAW, UNDRAW and TEXT commands is lumped together into one of the larger programs in the book because they are all concerned with Hi-res graphics. Hi-res is one area where BASIC simply is not capable of producing acceptable results and hence some recourse to machine code is nearly always required. The problems arise from the large amount of data that has to be shifted around in setting up a bit-mapped screen and the amount of arith-

metic required to calculate addresses in the bit-map from x and y co-ordinate values. The BITMAP and BITMAPC commands are designed to cope with the first problem and allow you to simply select a 320 by 200 pixel bit-map 'mode'. The PLOT and UNPLOT commands allow points to be very quickly plotted onto this bit-map at given co-ordinates at a rate limited only by the speed of BASIC itself, while the DRAW and UNDRAW commands utilize the fast Bresenham line draw algorithm to draw lines between specified points. TEXT is included for completeness, but is not really vital since the BITMAP mode is designed to 'collapse' back to standard text mode upon ENDing a program, or in response to any error condition. All the commands use the value of the last point plotted (stored by the routine) as an implicit parameter where appropriate.

BITMAPC

Syntax: BITMAPC (foregnd), (backgnd), (border)

Where: (foregnd) defines the foreground colour, i.e. the colour in which points and lines are plotted
(backgnd) defines the background colour
(border) defines the screen border colour

All parameters are optional, the default being white foreground on a black background with black border. The Last-Point-Plotted is initialised at 0, 0 (top left corner of screen).

Example: BITMAPC 2,6,8 clears the bit-map area, setting up the mode with blue background and orange border colours, with a red foreground plotting colour.

BITMAPC 2 clears the bit-map, sets a red plotting colour and default (black) background and border.

BITMAP

The BITMAP command is exactly the same as the BITMAPC command with the exception that the bit-map mode is enabled *without* clearing the old bit-mapped screen.

PLOT

Syntax: Plot (x-coord), (y-coord)

Where: (x-coord) defines the horizontal co-ordinate in the bit-map from 0–319
(y–coord) defines the vertical co–ordinate in the bit–map from 0–199

Example: PLOT 160,100 writes a point in the middle of the screen. This point is stored as the Last-Point-Plotted.

UNPLOT

This command has the same syntax as the PLOT command and UNPLOT (x-coord), (y-coord) acts as a PLOT so long as the specified point is clear, i.e. in background colour. If, however, the point is set then UNPLOT unsets it or erases the point.

DRAW

Syntax: DRAW (x-coord), (y-coord)

Example: DRAW 319,199 draws a line from the Last-Point-Plotted to 319,199 in the current foreground colour. 319,199 is stored as the new Last-Point-Plotted co-ordinates.

UNDRAW

UNDRAW has the same syntax as DRAW and bears to it the same relationship as does UNPLOT to PLOT. Thus if you UNDRAW over a line already present on screen then it will be erased, but otherwise it performs as a DRAW command.

TEXT

TEXT takes no parameters. It simply re-establishes Text mode, as will the ending of a program.

MOVE

With syntax similar to that of PLOT, MOVE allows you to reset the Last-Point-Plotted without actually writing any data to the bit-map.

Potential Problems
As is pointed out above the bit-map mode automatically reverts back to text mode when the program either ends or encounters an error. Halting a program by pushing the <run stop> key, however, will leave you in the graphics mode. To get back to text mode either type TEXT <return>, LIST <return> or, probably easiest, generate a syntax error by pressing any key followed by <return>.

The Hi-res Commands Code

Assembler Listing

```
1000 *=$9300
1010 ;----------
1020 CHTAB   =$14;*
1030 D       =$1B;*
1040 ENDX    =$1D;*
1050 INCR1   =$1F;*
1060 INCR2   =$21;*
1070 POS     =$23;*
1080 POS1    =$4B;*
1090 COUNT   =$4D;*
1100 YSIZE   =$4F;*
1110 XCOORH  =$57
1120 YCOORH  =$58
1130 TEMP1   =$59
1140 TEMP2   =$5A
1150 TEMP3   =$5B
1160 ;----------
1170 WHICHX  =$02AA;**
1180 WHPLOT  =$02AD;**
1190 ;----------
1200 XC1     =$02C0;*
1210 XC2     =$02C2;*
1220 DX      =$02C4;*
1230 DXNEG   =$02C6
1240 ;----------
1250 YC1     =$02C7;*
1260 YC2     =$02C9;*
1270 DY      =$02CB;*
1280 DYNEG   =$02CD
1290 ;----------
1300 WRITE   =$02F0;*
1310 COLSTR  =$0313
1320 XSTOR   =$0334;*
1330 ;----------
1340 LASTX   =$03F9;*
1350 LASTY   =$03FB;
1360 ;----------
1370 YL      =$9A00
1380 YH      =$9B00
1390 LOXL    =$9C00
1400 HIXL    =$9D00
```

```
1410 LOXH   =$9E00
1420 HIXH   =$9F00
1430 ;----------------------
1440 ;--CLEAR BIT-MAP--
1450 ;FILL BIT MAP WITH ZEROES/
1460 ;ENTRY POINT FOR 'BITMAPC' COMMAND
1470 ;----------------------
1480 MAPC   LDA #$60
1490        STA CHTAB+1
1500        LDA #$00
1510        STA CHTAB
1520        LDX #$1F
1530        TAY
1540 RAJ    STA (CHTAB),Y
1550        DEY
1560        BNE RAJ
1570        INC CHTAB+1
1580        DEX
1590        BPL RAJ
1600 ;----------------------
1610 ;--'BITMAP' COLOUR PARAMETERS--
1620 ;GET IN ANY PARAMETERS/TEMP1=
1630 ;FOREGND COLOUR/TEMP2=BACKGND COL/
1640 ;TEMP3=BORDER COLOUR/DEFAULT IS
1650 ;WHITE ON BLACK/
1660 ;ENTRY POINT FOR 'BITMAP' COMMAND
1670 ;----------------------
1680 MAP    LDA #$01
1690        LDX #$00
1700        LDY #$00
1710        STA TEMP1
1720        STX TEMP2
1730        STY TEMP3
1740 ;----------
1750        JSR $0079
1760        BEQ CLS
1770        JSR $B79E
1780        STX TEMP1
1790        JSR $0079
1800        BEQ CLS
1810        JSR $AEFD
1820        JSR $B79E
1830        STX TEMP2
1840        JSR $0079
```

```
1850          BEQ CLS
1860          JSR $AEFD
1870          JSR $B79E
1880          STX TEMP3
1890 ;----------
1900 CLS      LDA TEMP1
1910          ASL A
1920          ASL A
1930          ASL A
1940          ASL A
1950          ORA TEMP2
1960 ;-----------------------
1970 ;--CLEAR SCREEN--
1980 ;FILL SCREEN WITH 'BITMAP'
1990 ;PARAMETER VALUES/
2000 ;-----------------------
2010          LDX #$00
2020 GET      STA $5C00,X
2030          STA $5D00,X
2040          STA $5E00,X
2050          STA $5F00,X
2060          DEX
2070          BNE GET
2080 ;-----------------------
2090 ;--SETUP VIC II CHIP--
2100 ;SCREEN ($5C00) BIT-MAP ($6000)
2110 ;-----------------------
2120          LDA #$3B
2130          STA $D011
2140          LDA #$78
2150          STA $D018
2160          LDA TEMP3
2170          STA $D020
2180 ;-----------------------
2190 ;--SELECT MEMORY BANK 1--
2200 ;-----------------------
2210          LDA $DD02
2220          ORA #$03
2230          STA $DD02
2240          LDA $DD00
2250          AND #$FC
2260          ORA #$02
2270          STA $DD00
2280 ;-----------------------
```

```
2290 ;--SETUP RAM PLOT TABLE--
2300 ;--Y-COMPONENT
2310 ;-----------------------
2320         LDA #<YL
2330         LDX #>YL
2340         STA POS
2350         STX POS+1
2360         LDA #<YH
2370         LDX #>YH
2380         STA POS1
2390         STX POS1+1
2400 ;----------
2410         LDX #0
2420 START1  LDA #0
2430         STA TEMP1
2440         TXA
2450 ;-----------------------
2460 ; Y/8
2470 ;-----------------------
2480         LSR A
2490         LSR A
2500         LSR A
2510 ;-----------------------
2520 ; TEMP2,3=(Y/8)*64
2530 ;-----------------------
2540         ASL A
2550         ROL TEMP1
2560         ASL A
2570         ROL TEMP1
2580         ASL A
2590         ROL TEMP1
2600         ASL A
2610         ROL TEMP1
2620         ASL A
2630         ROL TEMP1
2640         ASL A
2650         ROL TEMP1
2660 ;----------
2670         STA TEMP2
2680         LDY TEMP1
2690         STY TEMP3
2700 ;-----------------------
2710 ; .A,TEMP1=(Y/8)*256
2720 ;-----------------------
```

```
2730          ASL A
2740          ROL TEMP1
2750          ASL A
2760          ROL TEMP1
2770 ;-----------------------
2780 ; (YAND7)+((Y/8)*256)+((Y/8)*64)
2790 ;THIS VALUE IS STORED IN MEMORY
2800 ;-----------------------
2810          PHA
2820          CLC
2830          TXA
2840          AND #$07
2850          ADC TEMP2
2860          STA TEMP2
2870          BCC ROT
2880          INC TEMP3
2890          CLC
2900 ROT      PLA
2910          ADC TEMP2
2920 ;----------
2930          LDY #0
2940          STA (POS),Y
2950          LDA TEMP1
2960          ADC TEMP3
2970          STA (POS1),Y
2980 ;----------
2990          INC POS
3000          INC POS1
3010          BNE SKIP1
3020          INC POS+1
3030          INC POS1+1
3040 SKIP1    INX
3050          BNE START1
3060 ;-----------------------
3070 ;--SETUP RAM PLOT TABLE--
3080 ;--X-COMPONENT
3090 ;-----------------------
3100          LDA #<LOXL
3110          LDX #>LOXL
3120          STA POS
3130          STX POS+1
3140          LDA #<LOXH
3150          LDX #>LOXH
3160          STA POS1
```

```
3170          STX POS1+1
3180 ;----------
3190          LDY #0
3200          STY COUNT
3210          STY COUNT+1
3220 ;-----------------------
3230 ; X/8
3240 ;-----------------------
3250 START2 LDA COUNT+1
3260          STA TEMP1
3270          LDA COUNT
3280          LSR TEMP1
3290          ROR A
3300          LSR TEMP1
3310          ROR A
3320          LSR TEMP1
3330          ROR A
3340 ;-----------------------
3350 ; (X/8)*8  THIS VALUE IS STORED
3360 ;-----------------------
3370          ASL A
3380          ROL TEMP1
3390          ASL A
3400          ROL TEMP1
3410          ASL A
3420          ROL TEMP1
3430 ;----------
3440          STA (POS),Y
3450          LDA TEMP1
3460          STA (POS1),Y
3470 ;----------
3480          INC POS
3490          INC POS1
3500          BNE SKIP2
3510          INC POS+1
3520          INC POS1+1
3530 SKIP2    INC COUNT
3540          BNE START2
3550          INC COUNT+1
3560          LDA COUNT+1
3570          CMP #$02
3580          BNE START2
3590 ;-----------------------
3600 ;--RESET SYSTEM ERROR VECTOR--
```

```
3610 ;-----------------------
3620         LDA #<EXIT2
3630         LDY #>EXIT2
3640         STA $0300
3650         STY $0301
3660 ;-----------------------
3670 ;--INITIALIZE PLOTTER AND RETURN--
3680 ;-----------------------
3690         LDA #$00
3700         STA LASTX
3710         STA LASTX+1
3720         STA LASTY
3730         RTS
3740 ;-----------------------
3750 ;--'MOVE' COMMAND--
3760 ;GETS X- AND Y- PARAMS FROM BUFFER
3770 ;USE THEM TO SET PLOTTER POSITION
3780 ;-----------------------
3790 MOV     JSR PRAMS
3800 MOV2    STX LASTX
3810         STY LASTY
3820         STA LASTX+1
3830         RTS
3840 ;-----------------------
3850 ;--'PLOT' COMMAND--
3860 ;PLOTS A POINT AT X,Y
3870 ;-----------------------
3880 PLO     LDA #<ORIT
3890         LDY #>ORIT
3900         BNE CONSO
3910 ;-----------------------
3920 ;--'UNPLOT' COMMAND--
3930 ;UNPLOTS A POINT AT X,Y
3940 ;-----------------------
3950 UNPLO   LDA #<EORIT
3960         LDY #>EORIT
3970 ;----------
3980 CONSO   STA WRITE
3990         STY WRITE+1
4000         JSR PRAMS
4010         JSR MOV2
4020         STA XCOORH
4030         JMP PLOT
4040 ;-----------------------
```

```
4050 ;--X,Y PARAMETER INPUT--
4060 ;GET IN PARAMETERS PRECEDED BY
4070 ;COMMAS/IF OUTSIDE SCREEN LIMITS
4080 ;THEN EXIT/.X = X COORD (LOW)/
4090 ;.A = X COORD (HIGH)/.Y = Y COORD
4100 ;/RETURN
4110 ;-----------------------
4120 PRAMS   JSR $0079
4130         BEQ EXIT1
4140         JSR $AD8A
4150         JSR $B7F7
4160         LDA $14
4170         LDX $15
4180         STA XSTOR
4190         STX XSTOR+1
4200         CMP #$40
4210         TXA
4220         SBC #$01
4230         BCS EXIT1
4240 ;----------
4250         JSR $AEFD
4260         BEQ EXIT1
4270         JSR $B79E
4280         CPX #$C8
4290         BCS EXIT1
4300 ;----------
4310         TXA
4320         TAY
4330         LDX XSTOR
4340         LDA XSTOR+1
4350         RTS
4360 ;-----------------------
4370 ;--ERROR EXIT--
4380 ;CLEAN UP STACK/LOAD .X WITH THE
4390 ;'ILLEGAL ARGUMENTS' VALUE/RESET
4400 ;SYSTEM ERROR VECTOR/JUMP TO
4410 ;SYSTEM ERROR ROUTINE
4420 ;-----------------------
4430 EXIT1   PLA
4440         PLA
4450         LDX #$0E
4460 EXIT2   JSR TEX
4470         LDA #$8B
4480         LDY #$E3
```

```
4490          STA $0300
4500          STY $0301
4510          JMP ($0300)
4520 ;-----------------------
4530 ;--RESTABLISH TEXT MODE--
4540 ;RESET MEMORY BANK 0/RESET VIC II
4550 ;CHIP DEFAULTS/RETURN
4560 ;-----------------------
4570 TEX      LDA $DD02
4580          ORA #$03
4590          STA $DD02
4600          LDA $DD00
4610          AND #$FC
4620          ORA #$03
4630          STA $DD00
4640 ;----------
4650          LDA #$1B
4660          STA $D011
4670          LDA #$15
4680          STA $D018
4690          LDA #$0E
4700          STA $D020
4710          RTS
4720 ;-----------------------
4730 ;--'DRAW' COMMAND--
4740 ;-----------------------
4750 DROR     LDA #<ORIT
4760          LDY #>ORIT
4770          BNE DOODL1
4780 ;-----------------------
4790 ;--'UNDRAW' COMMAND--
4800 ;-----------------------
4810 UNDROR LDA #<EORIT
4820          LDY #>EORIT
4830 ;-----------------------
4840 ;SET THE INDIRECT JMP ADDRESS FOR
4850 ;THE PLOT AND PLOTQ ROUTINES
4860 ;-----------------------
4870 DOODL1 STA WRITE
4880          STY WRITE+1
4890 ;-----------------------
4900 ;--INITIALIZE 'DRAW' AND 'UNDRAW'-
4910 ;SET (X1,Y1) AND (X2,Y2)/
4920 ;INITIALIZE VARIABLES
```

```
4930 ;-----------------------
4940 DOODL2 LDX LASTX
4950        LDY LASTY
4960        LDA LASTX+1
4970        STX XC1
4980        STY YC1
4990        STA XC1+1
5000        LDA #$00
5010        STA YC1+1
5020        JSR PRAMS
5030        JSR MOV2
5040        STX XC2
5050        STY YC2
5060        STA XC2+1
5070        LDA #$00
5080        STA YC2+1
5090 ;----------
5100        STA DYNEG
5110        STA DXNEG
5120 ;-----------------------
5130 ;--SET UP 'JMP'S FOR INDIRECT
5140 ;SUBROUTINES--
5150 ;-----------------------
5160        LDA #$4C
5170        STA WHICHX
5180        STA WHPLOT
5190 ;-----------------------
5200 ; DX=XC2-XC1
5210 ;-----------------------
5220        SEC
5230        LDA XC2
5240        SBC XC1
5250        STA DX
5260        LDA XC2+1
5270        SBC XC1+1
5280        STA DX+1
5290        BCS NEXT
5300 ;-----------------------
5310 ; ABS(DX) IF DX IS NEGATIVE
5320 ;-----------------------
5330        INC DXNEG
5340        LDA #$FF
5350        EOR DX
5360        STA DX
```

```
5370          INC DX
5380          LDA #$FF
5390          EOR DX+1
5400          STA DX+1
5410 ;----------------------
5420 ; DY=YC2-YC1
5430 ;----------------------
5440 NEXT     SEC
5450          LDA YC2
5460          SBC YC1
5470          STA DY
5480          LDA YC2+1
5490          SBC YC1+1
5500          STA DY+1
5510          BCS Q
5520 ;----------------------
5530 ; ABS(DY) IF DY IS NEGATIVE
5540 ;----------------------
5550          INC DYNEG
5560          LDA #$FF
5570          EOR DY
5580          STA DY
5590          INC DY
5600          LDA #$FF
5610          EOR DY+1
5620          STA DY+1
5630 ;----------------------
5640 ;-IF DY=>DX THEN SWAP X AND Y
5650 ;AND SET INDIRECT PLOT ROUTINE TO
5660 ;PLOTQ (PLOTS Y,X) ELSE PLOT
5670 ;(PLOTS X,Y)
5680 ;----------------------
5690 Q        LDA #<PLOT
5700          LDY #>PLOT
5710          STA WHPLOT+1
5720          STY WHPLOT+2
5730 ;----------
5740          LDA DY
5750          CMP DX
5760          LDA DY+1
5770          SBC DX+1
5780          BCC SLDLEF
5790 ;----------
5800          LDA #<PLOTQ
```

```
5810          LDY #>PLOTQ
5820          STA WHPLOT+1
5830          STY WHPLOT+2
5840 ;----------
5850          LDX #$06
5860 LP99     LDA XC1,X
5870          LDY YC1,X
5880          STA YC1,X
5890          TYA
5900          STA XC1,X
5910          DEX
5920          BPL LP99
5930 ;----------------------
5940 ;-CALCULATE INCR1, INCR2 AND D-
5950 ;----------------------
5960 ;INCR1=DY*2
5970 ;----------------------
5980 SLDLEF LDA DY+1
5990          STA INCR1+1
6000          LDA DY
6010          ASL A
6020          STA INCR1
6030          ROL INCR1+1
6040 ;----------------------
6050 ; D=INCR1-DX
6060 ;----------------------
6070          SEC
6080          SBC DX
6090          STA D
6100          LDA INCR1+1
6110          SBC DX+1
6120          STA D+1
6130 ;----------------------
6140 ; INCR2=INCR1-DX*2
6150 ;----------------------
6160          ASL DX
6170          ROL DX+1
6180          SEC
6190          LDA INCR1
6200          SBC DX
6210          STA INCR2
6220          LDA INCR1+1
6230          SBC DX+1
6240          STA INCR2+1
```

```
6250 ;----------------------
6260 ;--SET START AND END OF LINE--
6270 ;PLOT FROM LOWEST Y TO HIGHEST Y
6280 ;----------------------
6290 ;-COMPARE YC1 TO YC2-
6300 ;----------------------
6310         LDA DYNEG
6320         BEQ TAP
6330         LDA #$02
6340 TAP     TAX
6350 ;----------
6360         LDA YC1,X
6370         PHA
6380         LDA XC1,X
6390         PHA
6400         LDA XC1+1,X
6410         LDY YC1+1,X
6420         STA XCOORH
6430         STY YCOORH
6440 ;----------
6450         TXA
6460         EOR #$02
6470         TAX
6480         LDA XC1,X
6490         LDY XC1+1,X
6500         STA ENDX
6510         STY ENDX+1
6520 ;----------------------
6530 ;--SET PLUSX OR MINUSX FOR X
6540 ;ADJUST) INDIRECT SUBROUTINE
6550 ;WHICHX
6560 ;----------------------
6570         LDX #<PLUSX
6580         LDY #>PLUSX
6590 ;----------------------
6600 ;-COMPARE DX TO DY-
6610 ;----------------------
6620         LDA DXNEG
6630         EOR DYNEG
6640         BEQ EG1
6650 ;----------
6660         LDX #<MINUSX
6670         LDY #>MINUSX
6680 ;----------------------
```

```
6690 EG1     STX WHICHX+1
6700         STY WHICHX+2
6710         PLA
6720         TAX
6730         PLA
6740         TAY
6750         JSR WHPLOT
6760 ;-----------------------
6770 ;BRESENHAM DRAWING LOOP
6780 ;-----------------------
6790 EG2     CPX ENDX
6800         BNE PAT1
6810         LDA XCOORH
6820         CMP ENDX+1
6830         BEQ EXIT
6840 ;----------
6850 PAT1    JSR WHICHX
6860         LDA D+1
6870         BPL STAR
6880         JSR DNEG
6890         JMP BARG
6900 STAR    INY
6910         BNE TWEE
6920         INC YCOORH
6930 TWEE    JSR DPOS
6940 BARG    JSR WHPLOT
6950         JMP EG2
6960 ;-----------------------
6970 ;--EXIT TO BASIC--
6980 ;-----------------------
6990 EXIT    RTS
7000 ;-----------------------
7010 ;--PLOT/UNPLOT A POINT AT X,Y--
7020 ;.X AND .Y CONTAIN X AND Y COORDS/
7030 ;X AND Y COMPONENTS OF RAM PLOT
7040 ;TABLE ARE ADDED TO DERIVE PLOT
7050 ;ADDRESS
7060 ;-----------------------
7070 PLOT    CLC
7080         STY TEMP2
7090 ;----------
7100         LDA XCOORH
7110         BNE BIGXC
7120 ;----------
```

```
7130          LDA YL,Y
7140          ADC LOXL,X
7150          STA CHTAB
7160          LDA YH,Y
7170          ADC LOXH,X
7180          ORA #$60
7190          STA CHTAB+1
7200          BNE PUTIT
7210 ;----------
7220 BIGXC    LDA YL,Y
7230          ADC HIXL,X
7240          STA CHTAB
7250          LDA YH,Y
7260          ADC HIXH,X
7270          ORA #$60
7280          STA CHTAB+1
7290 ;-----------------------
7300 ;(XAND7) SELECTS CORRECT BIT MASK
7310 ;-----------------------
7320 PUTIT    TXA
7330          AND #$07
7340          TAY
7350          LDA MASKS,Y
7360          LDY #0
7370 ;-----------------------
7380 ;-JMP TO THE SELECTED OUTPUT
7390 ;ROUTINE THROUGH (WRITE)
7400 ;-----------------------
7410          JMP (WRITE)
7420 ;-----------------------
7430 ;--PLOT/UNPLOT A POINT AT Y,X--
7440 ;-----------------------
7450 ;-----------------------
7460 PLOTQ    CLC
7470          STY TEMP2
7480 ;----------
7490          LDA YCOORH
7500          BNE BIGXCQ
7510 ;----------
7520          LDA YL,X
7530          ADC LOXL,Y
7540          STA CHTAB
7550          LDA YH,X
7560          ADC LOXH,Y
```

```
7570          ORA #$60
7580          STA CHTAB+1
7590          BNE PUTITQ
7600 ;----------
7610 BIGXCQ LDA YL,X
7620          ADC HIXL,Y
7630          STA CHTAB
7640          LDA YH,X
7650          ADC HIXH,Y
7660          ORA #$60
7670          STA CHTAB+1
7680 ;----------
7690 PUTITQ TYA
7700          AND #$07
7710          TAY
7720          LDA MASKS,Y
7730          LDY #0
7740          JMP (WRITE)
7750 ;-----------------------
7760 ;--PLOT/UNPLOT OUTPUT ROUTINES--
7770 ;ORIT 'ORS' THE MASK IN WHILE
7780 ;EORIT 'EORS' IT OUT
7790 ;-----------------------
7800 ORIT     ORA (CHTAB),Y
7810          STA (CHTAB),Y
7820          LDY TEMP2
7830          RTS
7840 ;----------
7850 EORIT    EOR (CHTAB),Y
7860          STA (CHTAB),Y
7870          LDY TEMP2
7880          RTS
7890 ;-----------------------
7900 ;--ADD INCR1 TO D--
7910 ;-----------------------
7920 DNEG     CLC
7930          LDA D
7940          ADC INCR1
7950          STA D
7960          LDA D+1
7970          ADC INCR1+1
7980          STA D+1
7990          RTS
8000 ;-----------------------
```

```
8010 ;--ADD INCR2 TO D--
8020 ;-----------------------
8030 DPOS   CLC
8040        LDA D
8050        ADC INCR2
8060        STA D
8070        LDA D+1
8080        ADC INCR2+1
8090        STA D+1
8100        RTS
8110 ;-----------------------
8120 ;--ADD 1 TO XCOORD--
8130 ;-----------------------
8140 PLUSX  INX
8150        BNE Q2
8160        INC XCOORH
8170 Q2     RTS
8180 ;-----------------------
8190 ;--SUBTRACT 1 FROM XCOORD--
8200 ;-----------------------
8210 MINUSX TXA
8220        BNE Q5
8230        DEC XCOORH
8240 Q5     DEX
8250        RTS
8260 ;-----------------------
8270 ;--BIT MASKS FOR PLOT ROUTINES--
8280 ;-----------------------
8290 MASKS  .BYTE $80,$40,$20,$10
8300        .BYTE $08,$04,$02,$01
8310 ;-------------
```

BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 37631+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
```

```
111 DATA169,96,133,21,169,0,133,20
112 DATA162,31,168,145,20,136,208,251
113 DATA230,21,202,16,246,169,1,162
114 DATA0,160,0,133,89,134,90,132
115 DATA91,32,121,0,240,31,32,158
116 DATA183,134,89,32,121,0,240,21
117 DATA32,253,174,32,158,183,134,90
118 DATA32,121,0,240,8,32,253,174
119 DATA-7088
120 REM --- BLOCK 2
121 DATA32,158,183,134,91,165,89,10
122 DATA10,10,10,5,90,162,0,157
123 DATA0,92,157,0,93,157,0,94
124 DATA157,0,95,202,208,241,169,59
125 DATA141,17,208,169,120,141,24,208
126 DATA165,91,141,32,208,173,2,221
127 DATA9,3,141,2,221,173,0,221
128 DATA41,252,9,2,141,0,221,169
129 DATA-6696
130 REM --- BLOCK 3
131 DATA0,162,154,133,35,134,36,169
132 DATA0,162,155,133,75,134,76,162
133 DATA0,169,0,133,89,138,74,74
134 DATA74,10,38,89,10,38,89,10
135 DATA38,89,10,38,89,10,38,89
136 DATA10,38,89,133,90,164,89,132
137 DATA91,10,38,89,10,38,89,72
138 DATA24,138,41,7,101,90,133,90
139 DATA-4962
140 REM --- BLOCK 4
141 DATA144,3,230,91,24,104,101,90
142 DATA160,0,145,35,165,89,101,91
143 DATA145,75,230,35,230,75,208,4
144 DATA230,36,230,76,232,208,178,169
145 DATA0,162,156,133,35,134,36,169
146 DATA0,162,158,133,75,134,76,160
147 DATA0,132,77,132,78,165,78,133
148 DATA89,165,77,70,89,106,70,89
149 DATA-7207
150 REM --- BLOCK 5
151 DATA106,70,89,106,10,38,89,10
152 DATA38,89,10,38,89,145,35,165
153 DATA89,145,75,230,35,230,75,208
154 DATA4,230,36,230,76,230,77,208
```

```
155 DATA212,230,78,165,78,201,2,208
156 DATA204,169,156,160,148,141,0,3
157 DATA140,1,3,169,0,141,249,3
158 DATA141,250,3,141,251,3,96,32
159 DATA-7083
160 REM --- BLOCK 6
161 DATA103,148,142,249,3,140,251,3
162 DATA141,250,3,96,169,173,160,150
163 DATA208,4,169,180,160,150,141,240
164 DATA2,140,241,2,32,103,148,32
165 DATA66,148,133,87,76,59,150,32
166 DATA121,0,240,44,32,138,173,32
167 DATA247,183,165,20,166,21,141,52
168 DATA3,142,53,3,201,64,138,233
169 DATA-7496
170 REM --- BLOCK 7
171 DATA1,176,21,32,253,174,240,16
172 DATA32,158,183,224,200,176,9,138
173 DATA168,174,52,3,173,53,3,96
174 DATA104,104,162,14,32,172,148,169
175 DATA139,160,227,141,0,3,140,1
176 DATA3,108,0,3,173,2,221,9
177 DATA3,141,2,221,173,0,221,41
178 DATA252,9,3,141,0,221,169,27
179 DATA-6614
180 REM --- BLOCK 8
181 DATA141,17,208,169,21,141,24,208
182 DATA169,14,141,32,208,96,169,173
183 DATA160,150,208,4,169,180,160,150
184 DATA141,240,2,140,241,2,174,249
185 DATA3,172,251,3,173,250,3,142
186 DATA192,2,140,199,2,141,193,2
187 DATA169,0,141,200,2,32,103,148
188 DATA32,66,148,142,194,2,140,201
189 DATA-7889
190 REM --- BLOCK 9
191 DATA2,141,195,2,169,0,141,202
192 DATA2,141,205,2,141,198,2,169
193 DATA76,141,170,2,141,173,2,56
194 DATA173,194,2,237,192,2,141,196
195 DATA2,173,195,2,237,193,2,141
196 DATA197,2,176,22,238,198,2,169
197 DATA255,77,196,2,141,196,2,238
198 DATA196,2,169,255,77,197,2,141
```

```
199 DATA-7705
200 REM --- BLOCK 10
201 DATA197,2,56,173,201,2,237,199
202 DATA2,141,203,2,173,202,2,237
203 DATA200,2,141,204,2,176,22,238
204 DATA205,2,169,255,77,203,2,141
205 DATA203,2,238,203,2,169,255,77
206 DATA204,2,141,204,2,169,59,160
207 DATA150,141,174,2,140,175,2,173
208 DATA203,2,205,196,2,173,204,2
209 DATA-8102
210 REM --- BLOCK 11
211 DATA237,197,2,144,28,169,116,160
212 DATA150,141,174,2,140,175,2,162
213 DATA6,189,192,2,188,199,2,157
214 DATA199,2,152,157,192,2,202,16
215 DATA240,173,204,2,133,32,173,203
216 DATA2,10,133,31,38,32,56,237
217 DATA196,2,133,27,165,32,237,197
218 DATA2,133,28,14,196,2,46,197
219 DATA-7162
220 REM --- BLOCK 12
221 DATA2,56,165,31,237,196,2,133
222 DATA33,165,32,237,197,2,133,34
223 DATA173,205,2,240,2,169,2,170
224 DATA189,199,2,72,189,192,2,72
225 DATA189,193,2,188,200,2,133,87
226 DATA132,88,138,73,2,170,189,192
227 DATA2,188,193,2,133,29,132,30
228 DATA162,215,160,150,173,198,2,77
229 DATA-7359
230 REM --- BLOCK 13
231 DATA205,2,240,4,162,221,160,150
232 DATA142,171,2,140,172,2,104,170
233 DATA104,168,32,173,2,228,29,208
234 DATA6,165,87,197,30,240,27,32
235 DATA170,2,165,28,16,6,32,187
236 DATA150,76,52,150,200,208,2,230
237 DATA88,32,201,150,32,173,2,76
238 DATA21,150,96,24,132,90,165,87
239 DATA-6968
240 REM --- BLOCK 14
241 DATA208,20,185,0,154,125,0,156
242 DATA133,20,185,0,155,125,0,158
```

```
243 DATA9,96,133,21,208,18,185,0
244 DATA154,125,0,157,133,20,185,0
245 DATA155,125,0,159,9,96,133,21
246 DATA138,41,7,168,185,228,150,160
247 DATA0,108,240,2,24,132,90,165
248 DATA88,208,20,189,0,154,121,0
249 DATA-6384
250 REM --- BLOCK 15
251 DATA156,133,20,189,0,155,121,0
252 DATA158,9,96,133,21,208,18,189
253 DATA0,154,121,0,157,133,20,189
254 DATA0,155,121,0,159,9,96,133
255 DATA21,152,41,7,168,185,228,150
256 DATA160,0,108,240,2,17,20,145
257 DATA20,164,90,96,81,20,145,20
258 DATA164,90,96,24,165,27,101,31
259 DATA-6031
260 REM --- BLOCK 16
261 DATA133,27,165,28,101,32,133,28
262 DATA96,24,165,27,101,33,133,27
263 DATA165,28,101,34,133,28,96,232
264 DATA208,2,230,87,96,138,208,2
265 DATA198,87,202,96,128,64,32,16
266 DATA8,4,2,1
267 DATA-3879
268 DATA END
1000 REM -- ENABLE NEW COMMANDS --
1001 AD=33339
1002 REM - BITMAPC
1003 POKE AD,255:POKE AD+1,146
1004 REM - BITMAP
1005 POKE AD+2,20:POKE AD+3,147
1006 REM - MOVE
1007 POKE AD+4,62:POKE AD+5,148
1008 REM - PLOT
1009 POKE AD+6,75:POKE AD+7,148
1010 REM - UNPLOT
1011 POKE AD+8,81:POKE AD+9,148
1012 REM - DRAW
1013 POKE AD+10,205:POKE AD+11,148
1014 REM - UNDRAW
1015 POKE AD+12,211:POKE AD+13,148
1016 REM - TEXT
1017 POKE AD+14,171:POKE AD+15,148
1018 END
```

The Hi-res Routines

Screen Memory

The memory area for the bit-mapped screen is held at $6000 onwards in *memory bank 2*, below the 'Expander' code which starts at $8000, and the colour table for the bit-map is stored below the bit-map itself, starting at $5C00.

Clearing The Bit-map Memory Area (Lines 1480–1590)

This loop provides a good example of how to access a large area of memory (the 32 pages required for the bit-map) using post-indexed indirect addressing. The number of pages to be cleared, minus 1, is contained in the X register. This is the point at which the BITMAPC command enters.

Setting Up the Colour Table (Lines 1680–2070)

At this point the BITMAP command enters, avoiding the routine to clear the bit-map. Lines 1750–1880 get in any parameters after the command into TEMP1, TEMP2 and TEMP3. If there are no parameters then the defaults established in lines 1680–1730 are used. The values in TEMP1 and TEMP2 are combined into a single byte in lines 1900–1950, and this determines the value that is written into the colour table at lines 2010–2070. Note that in a colour table for a bit-map the upper and lower nybbles (four bits) in each byte represent the foreground and background colours respectively for the corresponding 64 pixel (8 by 8 block) region in the bit-map.

Switch to Bit-map Mode and Position Bit-map (Lines 2120–2270)

The VIC II control register $D011 controls the graphics 'mode', while the position of the bit-map and colour table within a memory bank is determined by $D018. Since we are changing memory banks we also need to change the bank select bits (0 and 1) of Port A of the CIA#2 chip, located at $DD00. Before we can change these it is necessary to set bits 0 and 1 in the Port A Data Direction register, $DD02.

Setting Up a RAM Plot Table (Lines 2320–3580)

In order to plot a point in a bit-map it is necessary to be able to calculate the *address* in the bit-map of the byte containing the point on the basis of its x and y co-ordinates. This is a time consuming task involving several multiplications, and if we were to perform the calculation each time a point were to be plotted it would con-

siderably slow down the process. An alternative strategy, where, as here, enough RAM memory is available, is to perform the calculations beforehand for every value of x and y and store the result in a table that can be quickly accessed. The calculation that must be performed involves the summation of two components, one depending on the y co-ordinate and the other on the x co-ordinate:

```
y component=INT(Y/8)*320+(YAND7)
x component=INT(X/8)*8
Address=(x component)+(y component)
```

The position of the specific bit within the byte is given by 2↑x, two to the power of x, but this is easily taken from a lookup table and will be left to the individual command routine.

What has been done is to construct 2-byte tables of the results of the above calculations for x values to 319 and y values to 199, YL, YH, LOXL, HIXL, LOXH, HIXH. When the time comes to plot points deriving the plot address will simply be a question of adding the y-table value, indexed by y to x-table,x. Thus, at the cost of making the BITMAP command fairly slow (though not so you'd notice), the PLOT and DRAW commands become much quicker.

Note that the multiplication Y*320 is achieved by adding Y*256 to Y*64 in lines 2540–2910.

Reset System Error Vector (Lines 3620–3650)

Whenever an error is detected in a BASIC program it causes an indirect JMP to the routine which prints the corresponding error message through a RAM vector at ($0300). By changing this vector to point to our code we can make errors switch off bit-map mode and re-establish the text screen before they are printed. The reason for doing this is that otherwise the error messages would not be visible and debugging BASIC would become a guessing game (somewhat like debugging machine code).

Initialise PLOTter (Lines 3690–3730)

The 'Expander' DRAW command is a two-parameter draw routine, i.e. it DRAWS *to* the parameter x,y position *from* the last position PLOTted or MOVEd to. BITMAP commands initialise the Last-Point-Plotted to 0,0, the top left hand corner of the screen. At line 3730 we return to BASIC.

MOVE Command (Lines 3790–3830)

PRAMS gets the x and y co-ordinate parameters into Y (y–coord), X (x–coord low) and A (x–coord high). These values are then used to reset the plotter position.

PLOT and UNPLOT Commands (Lines 3880–4030)

The only difference between these two commands lies in the fact that in the case of PLOT data is ORed into the bit-map while for UNPLOT it is EORed. The PLOT routine at lines 7070–7410 which performs the plotting has a variable element in the form of an indirect jump to (WRITE) to cope with the two sorts of PLOTting. This indirect address is set up at the entry points for the two commands, so that for PLOT the WRITE vector points to the ORIT routine at line 7800 and for UNPLOT the WRITE vector points to EORIT at line 7850. Line 4000 gets in the command parameters from PRAMS and JSR MOV2 updates the plotter for the new position. Finally a JMP is made in line 4030 to the PLOT routine at line 7070 onwards.

PLOT accepts the x and y co-ordinates in registers X and Y, with the *high* byte of the x co-ordinate stored in XCOORH. In line 7100 XCOORH is checked to see if the (x-coord) parameter is greater than 255, in which case we must access the x-component RAM table 256 bytes up from the start, i.e. in HIXL, HIXH, at x-coord low plus 256. The two components from the RAM table are then added to give an address in CHTAB. In lines 7320–7350 the correct bit mask for the plot is selected as MASK,Y where Y equals X AND 7. Finally, a JMP to (WRITE) either ORs or EORs the mask into the bit-map and returns to BASIC.

X and Y Co-ordinate Parameter Input (Lines 4120–4350).

The PRAMS routine gets in command parameters and checks them against the size of the screen. If they are outside the screen limits of x co-ordinate 0–319 and y 0–199 then the stack is popped and an ILLEGAL QUANTITY error is generated at lines 4430–4510.

Error Handler (Lines (Lines 4430–4510)

EXIT1 is for errors generated within PRAMS. EXIT2 is the new system error vector set in BITMAP. Any system error causes a JSR TEX which resets text mode and then the default system error vector is restored in lines 4470–4500, followed by a JMP to the latter.

DRAW and UNDRAW Commands (Lines 4750–6990)

In similar fashion to the PLOT and UNPLOT commands this pair is

handled by the same code except for the indirect (WRITE) vector which points to ORIT for the DRAW command and EORIT for UNDRAW. In lines 4950–5080 the two points for the line are set up. XC1 and YC1 are derived from the Last-Point-Plotted, while XC2 and YC2 come from the command parameters.

Drawing Lines

There are several possible methods for drawing lines on a micro. For example, one possibility is the so-called *incremental* method works thus:

1. Increment x
2. Calculate y as SLOPE*x
3. Plot a pixel at x, (y rounded)

The problems with this method are that the calculation of SLOPE*x and the rounding of y are time consuming, since floating point numbers must be used to obtain sufficient accuracy.

An alternative is the *Bresenham* algorithm, which uses only integer numbers. This works by accumulating, for each x increment, a 'decision variable' D whose value determines whether y is to be incremented, i.e. if D>0 then increment y. (Maths freaks are directed to *Fundamentals of Interactive Computer Graphics* by Foley and Van Dam, published by Addison Wesley.)

The above method, however, only gives lines for 0>=SLOPE<1 or 1/8th of a circle. To obtain other slopes you must:

1. Decrement x instead of incrementing
2. Swap x with y and plot y,x

This gives a range of half the circle. The other half is obtained by:

3. Always plotting from the *lowest* y to the *highest* y

Below is given an example of the same algorithm coded in BASIC, and you should find the BASIC and the machine code very similar. There are however several points worth noting, as follows.

Lines 190–240 of the BASIC program are performed by lines 5690–5920 of the machine code. Notice that in the machine code version DX and DY are not recalculated from the 'swapped' x and y co-ordinates, but are included in the swap in lines 5850–5920. Note also that instead of setting a flag (F in the BASIC) to determine which PLOT routine is used (x,y or y,x) the machine code uses an 'indirect subroutine' at WHPLOT.

Line 350 of the BASIC program is also handled rather differently in machine code. When DX and DY are calculated in lines 5220–5620 two flags, DXNEG and DYNEG, are set if the original values of DX and DY were *negative*. These flags are then used later on, in lines 6310–6340, to compare YC1 to YC2 (see line 300 of the BASIC version) and in lines 6620–6640 (corresponding to line 350 of the BASIC).

In place of the variable U in lines 340–410 of the BASIC another indirect subroutine is set up in the code to determine if x is *incremented* or *decremented* in lines 6470–6700. Also, note that in the machine code two PLOT routines are required, PLOT which plots x,y and PLOTQ which plots y,x.

```
100 BITMAPC
110 X1=100:Y1=100
120 X2=010:Y2=040
130 F=0
140 DX=X2-X1:AX=ABS(DX)
150 DY=Y2-Y1:AY=ABS(DY)
160 REM----------------
170 REM IFAX<AY SWAP X AND Y
180 REM----------------
190 IF AX>=AY THEN 280
200 F=1
210 T=Y2:Y2=X2:X2=T
220 T=Y1:Y1=X1:X1=T
230 DX=X2-X1:AX=ABS(DX)
240 DY=Y2-Y1:AY=ABS(DY)
250 REM----------------
260 REM INITIALIZE D/I1/I2
270 REM----------------
280 I1=AY*2:D=I1-AX
290 I2=2*(AY-AX)
300 IFY1>Y2THEN320
310 X=X1:Y=Y1:XE=X2:GOTO330
320 X=X2:Y=Y2:XE=X1
330 GOSUB500
340 U=1
350 IFDX<0ANDDY<0ORDX=>0ANDDY=>0THEN400
360 U=-1
370 REM----------------
380 REM BRESENHAM LOOP
390 REM----------------
400 IFX=XETHEN530
410 X=X+U
```

```
420 IFD=>0THEN440
430 D=D+I1:GOTO450
440 Y=Y+1:D=D+I2
450 GOSUB 500
460 GOTO 400
470 REM----------------
480 REM PLOT ROUTINES
490 REM----------------
500 IF F>0 THEN 520
510 PLOT X,Y:RETURN
520 PLOT Y,X:RETURN
530 WAIT198,1:POKE198,0
540 END
```

Functions

This section presents four *functions* for incorporation in 'Expander' which, although quite short and straightforward, considerably simplify the handling of sprites and joysticks. Tasks which would otherwise require a lot of tedious PEEKing of registers and ANDing of data to derive the needed result are handled by the function routines, which return the result after processing.

JOY(N)

The JOY(N(function reads the joystick and returns a value corresponding to the state of the joystick (0 or 1) specified in the brackets. JOY(0) returns the value of Joystick 1 and JOY(1) the value of Joystick 2. The values returned for the various joystick directions are as follows:

JOYSTICK DIRECTION	JOY(N)
Centred (no direction)	0
North	2
North-East	1
East	4
South-East	3
South	6
South-West	5
West	8
North-West	7

These particular values were chosen because they facilitate the writing of a joystick handling routine based on the ON . . . GOSUB construction of BASIC. An example of such a routine would be:

```
10 ON JOY(0) GOSUB 20,25,30,35,40,45,50,55
15 GOTO 10
20 GOSUB 35
25 Y=Y-1:RETURN
30 GOSUB 45
35 X=X+1:RETURN
40 GOSUB 55
45 Y=Y+1:RETURN
50 GOSUB 25
55 X=X-1:RETURN
```

FIRE(N)

FIRE(N) returns a value indicating whether or not the fire button of joystick of which the parameter is N (0 or 1, as for JOY(N)). This value is 1 if the fire button is being pressed, and a 0 if not.

XSPR(N)

XSPR(N) returns the x-position on the screen of the sprite whose number is specified in the brackets. Thus, PRINT XSPR(0) will print the current x-position of sprite 0.

YSPR(N)

YSPR(N) returns the y-position of the sprite whose number, specified in the brackets, is N.

These two sprite position functions enable you to keep track of the positions on screen of sprites whose movement is linked to the joystick by the JSPRITE command. Together, JSPRITE, XSPR(N) and YSPR(N) make a powerful tool for the writing of BASIC games.

The Functions Code

Assembler Listing

```
1000 *=$9100
1010 ;---------------
1020 ;--'XSPR' FUNCTION--
1030 ;GET SPRITE PARAMETER INTO $14/
1040 ;SEE IF BIT COORESPONDING TO PARAM
1050 ;SPRITE SET IN REGISTER $D010/IF
1060 ;SO THEN LOAD .A WITH 1 ELSE ZERO/
1070 ;LOAD .Y WITH SPRITE Y-POS/EXIT
```

```
1080 ;VIA $B391 (CONVERTS INTEGER TO
1090 ;FLOATING POINT.)
1100 ;---------------
1110 SPRIX  JSR $B7F7
1120        LDA $14
1130        AND #$07
1140        ASL A
1150        TAX
1160        LDA $D010
1170        AND MMASK,X
1180        BEQ HISKP
1190        LDA #$01
1200 HISKP  PHA
1210        LDY $D000,X
1220        PLA
1230        JMP $B391
1240 ;---------------
1250 ;--'YSPR' FUNCTION--
1260 ;GET SPRITE PARAMETER IN $14/USE
1270 ;TO INDEX SPRITE Y-POSITION REGSTR
1280 ;LOAD .A WITH ZERO/EXIT VIA $B391
1290 ;---------------
1300 SPRIY  JSR $B7F7
1310        LDA $14
1320        AND #$07
1330        ASL A
1340        TAX
1350        LDY $D001,X
1360        LDA #$00
1370        JMP $B391
1750 ;---------------
1760 ;--'JOY' FUNCTION--
1770 ;GET PARAMETER INTO $14,$15/FLIP
1780 ;BIT ZERO AND USE TO INDEX CIA 1
1790 ;DATA PORTS/FLIP LOWER 4 BITS OF
1800 ;PORT AND USE AS INDEX INTO JOYBIT
1810 ;TABLE/RETURN VIA $B391 (CONVERTS
1820 ;INTEGER TO FP.)/
1830 ;---------------
1840 JOYYY  JSR $B7F7
1850        LDA $14
1860        AND #$01
1870        EOR #$01
1880        TAX
```

```
1890          LDA $DC00,X
1900          AND #$0F
1910          EOR #$0F
1920          TAX
1930          LDY JOYBIT,X
1940          LDA #$00
1950          JMP $B391
1960 ;---------------
1970 ;--'FIRE' FUNCTION--
1980 ;GET PARAMETER IN $14,15/FLIP BIT
1990 ;ZERO AND USE AS INDEX TO CIA DATA
2000 ;PORTS/'AND' PORT VALUE WITH #$10
2010 ;IF ZERO RETURN VALUE 1 ELSE
2020 ;RETURN 0/
2030 ;---------------
2040 FIRRE    JSR $B7F7
2050          LDA $14
2060          AND #$01
2070          EOR #$01
2080          TAX
2090          LDY #$00
2100          LDA #$10
2110          AND $DC00,X
2120          BNE ZERJOY
2130          LDY #$01
2140 ZERJOY   LDA #$00
2150          JMP $B391
2160 ;---------------
2170 MMASK    .BYTE $01,$02,$04,$08
2180          .BYTE $10,$20,$40,$80
2190 ;---------------
2200 JOYBIT   .BYTE $00,$02,$06,$00
2210          .BYTE $08,$07,$05,$00
2220          .BYTE $04,$01,$03
2230 ;---------------
```

BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 37119+X,D
105 GOTO 101
```

```
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA32,247,183,165,20,41,7,10
112 DATA170,173,16,208,61,96,145,240
113 DATA2,169,1,72,188,0,208,104
114 DATA76,145,179,32,247,183,165,20
115 DATA41,7,10,170,188,1,208,169
116 DATA0,76,145,179,32,247,183,165
117 DATA20,41,1,73,1,170,189,0
118 DATA220,41,15,73,15,170,188,104
119 DATA-6747
120 REM --- BLOCK 2
121 DATA145,169,0,76,145,179,32,247
122 DATA183,165,20,41,1,73,1,170
123 DATA160,0,169,16,61,0,220,208
124 DATA2,160,1,169,0,76,145,179
125 DATA1,2,4,8,16,32,64,128
126 DATA0,2,6,0,8,7,5,0
127 DATA4,1,3
128 DATA-3504
129 DATA END
1000 REM -- ENABLE NEW FUNCTIONS --
1001 AD=33285
1002 REM - JOY(X)
1003 POKE AD,44:POKE AD+1,145
1004 REM - FIRE(X)
1005 POKE AD+2,70:POKE AD+3,145
1006 REM - XSPR(X)
1007 POKE AD+4,0:POKE AD+5,145
1008 REM - YSPR(X)
1009 POKE AD+6,27:POKE AD+7,145
1010 END
```

The Function Routines

Now let us look at how these functions are coded.

XSPR Function (Lines 1110–1230)

The JSR to $B7F7 in line 110 accesses a routine to convert the argument between the brackets from floating point format into an integer held in $14, $15. This is multiplied by two, giving an index into the 2-byte mask table MMASK and the sprite x-position registers. The latter value

is stored in Y. If the bit matching the parameter sprite in the x co-ordinate *high* byte register is *set* then A is loaded with 1, else it is loaded with zero. We exit the routine via the $B391 routine, which converts the integer result in Y, A into floating point form in the Floating Point Accumulator #1.

YSPR Function (Lines 1300–1370)
The integer result of the argument in brackets from the $B7F7 routine is multiplied by two and used to index the sprite y-position registers. The value from the registers is stored in register Y. A is loaded with zero. This result is passed back via the $B391 routine.

JOY Function (Lines 1840–1950)

The parameter from the brackets is ANDed with #$01, and then 'flipped' (EORed with #$01) and used to index the CIA#1 Data Ports, so that if the value in the brackets was 1 then we access $DC00+0 and if the value was 0 we access $DC00+1. The value from the data port is transferred to the X register and used to index the JOYBIT table of joystick values. The table value is stored in Y and A holds zero. The result is returned via the routine at $B391.

FIRE Function (Lines 2040–2150)

The parameter in brackets is used to access the Data Ports, as with JOY, but in this case, if bit 4 of the data port holds a 0, then we return A, Y equal to 1 else A, Y returns 0.

Adding Your Own Commands to 'Expander'

The 'Expander' program given in the text added to BASIC the commands which I personally thought were the most useful. Different programmers may well disagree with the selection. This need not be a problem, however, as the scope of 'Expander' is not limited to the command routines provided in this book.

If you feel confident enough to write your own command routines then it is very easy to incorporate them into the 'Expander' command handler.

If you refer back to lines 3690–4500 of the 'Expander' Command Handler routine you will see that in the 'Expander' Keyword table WORD there are several 'dummy' entries. You may also consider FILL (line 4390) to be a dummy entry also, as that was one command impossible to include in the time available for writing. To add a new function or command you must follow this procedure:

1. Load 'Expander'.
2. Load your new function or command routine.
3. POKE either the *Start Address*, for a new *function* routine, or the *Start Address minus 1* in the case of a command, into its respective command handler action table, i.e. FUNCS for functions or ADDRS for commands. The position of the address in the action table must of course correspond to the position of its keyword in the keyword table. FUNCS and ADDRS reside in memory at locations 33285 and 33313 respectively, the first free (dummy) address being located at 33293 for FUNCS and 33359 for ADDRS.
4. Change the dummy keyword in the keyword table to your new keyword. Note that while new command keywords can be as long as you like function keywords are limited to five letters, since ten 5-letter dummy function entries are placed *between* the last of the book function routines, the YSPR function, and the first of the command routines, RENUMBER, and it is important to keep ten keyword 'entries' between YSPR and RENUMBER, or the commands will not work. The first dummy function keyword starts at 33388 and the first dummy command at 33561 (not including FILL at 33551). Remember that the *last* character in each keyword is held as ASCII code *plus* 128.
5. Save 'Expander' plus your new handler routine as one program.

Where to Put Your New Routines

Space has been left after each of the 'Expander' command routines so that users can make amendments to them if required, and short commands or functions could also be placed in these areas. For example, after the functions code given above there are more than 256 bytes left free. A memory map of the 'Expander' command routines is provided in Appendix 5. Alternatively, there is RAM available between $96EC and $99FF (the RAM plot tables for the DRAW and PLOT commands occupy $9A00–$9FFF). A third possibility is to lower the Top-of-BASIC from $5B00 and put your routines under the bit mapped screen.

Writing the Routines

The whole purpose of this book has been to show, by example, how functional machine code is written and if it has done its job, then by following the general methodology of the 'Expander' command routines you should now be able go have a go at writing commands and functions of your own. You might start off with something simple, such as SCREEN and BORDER commands to change screen and border colours, or DEEK and DOKE (2-byte PEEK and POKE) routines. You may then move on to multicolour hi-res graphics or the FILL command that fell foul of this book's deadline. Remember that planning produces better code, and good luck!

APPENDICES

Appendix 1
6510 Op codes

The tables presented in this appendix give the 6510 Op Codes, with the hex codes for each mnemonic categorised by address mode.

The notation is as follows:

R1	Register Changed
R2	Sending Register
A	Accumulator
X	Register X
Y	Register Y
S	Stack Register
PC	Program Counter
P	Processor Register (Status Flags)
M	Memory Location

The P Register flags are:

Bit 0	**C**	Carry
1	**Z**	Zero
2	**I**	Interrupt
3	**D**	Decimal
4	**B**	Break
5	*	(not used)
6	**V**	oVerflow
7	**N**	Negative

MNEMONIC	DESCRIPTION	R1	R2	IMPLIED	IMMED	ZERO PAGE	Z-PAGE X
ADC	ADD WITH CARRY	A			69	65	75
AND	LOGICAL AND	A			29	25	35
ASL	ARITHMETIC SHIFT LEFT	A/M		0A(A)		06	16
BCC	BRANCH ON CARRY CLEAR						
BCS	BRANCH ON CARRY SET						
BEQ	BRANCH IF EQUAL TO ZERO						
BIT	COMPARE BITS WITH ACCUMULATOR					24	
BMI	BRANCH ON MINUS						
BNE	BRANCH ON NOT EQUAL TO ZERO						
BPL	BRANCH ON PLUS						
BRK	BREAK	S		00			
BVC	BRANCH ON OVERFLOW CLEAR						
BVS	BRANCH ON OVERFLOW SET						
CLC	CLEAR CARRY			18			
CLD	CLEAR DECIMAL			D8			
CLI	CLEAR INTERRUPT MASK			58			
CLV	CLEAR OVERFLOW FLAG			B8			
CMP	COMPARE TO ACCUMULATOR				C9	C5	D5
CPX	COMPARE TO REG-X				E0	E4	
CPY	COMPARE TO REG-Y				C0	C4	
DEC	DECREMENT MEMORY					C6	D6
DEX	DECREMENT REG-X	X		CA			
DEY	DECREMENT REG-Y	Y		88			
EOR	EXCLUSIVE OR ACCUMULATOR	A			49	45	55
INC	INCREMENT MEMORY					E6	F6
INX	INCREMENT REG-X	X		E8			
INY	INCREMENT REG-Y	Y		C8			
JMP	JUMP TO ADDRESS	PC					

Z-PAGE Y	INDIRECT X	INDIRECT Y	RELATIVE ADDRESS	ABSOLUTE ADDRESS	ABSOLUTE X	ABSOLUTE Y	INDIRECT	P-REGISTER N	V	*	B	D	I	Z	C
	61	71		6D	7D	79		×	×					×	×
	21	31		2D	3D	39		×						×	
				0E	1E			×						×	×
			90												
			B0												
			F0												
				2C				M7	M6					×	
			30												
			D0												
			10												
											×				
			50												
			70												
												0			0
									0				0		
	C1	D1		CD	DD	D9		×						×	×
				EC				×						×	×
				CC				×						×	×
				CE	DE			×						×	
								×						×	
								×						×	
	41	51		4D	5D	59		×						×	
				EE	FE			×						×	
								×						×	
								×						×	
				4C			6C								

MNEMONIC	DESCRIPTION	R1	R2	IMPLIED	IMMED	ZERO PAGE	Z-PAGE X
JSR	JUMP TO SUBROUTINE	PC/S					
LDA	LOAD ACCUMULATOR	A			A9	A5	B5
LDX	LOAD REG-X	X			A2	A6	
LDY	LOAD REG-Y	Y			A0	A4	B4
LSR	LOGICAL SHIFT RIGHT	A/M		4A(A)		46	56
NOP	NO OPERATION			EA			
ORA	INCLUSIVE OR ACCUMULATOR	A			09	05	15
PHA	PUSH A ONTO STACK	S	A	48			
PHP	PUSH P-REGISTER ONTO STACK	S	P	08			
PLA	PULL ACCUMULATOR FROM STACK	A		68			
PLP	PULL P-REGISTER FROM STACK	P		28			
ROL	ROTATE LEFT ONE BIT	A/M		2A(A)		26	36
ROR	ROTATE RIGHT ONE BIT	A/M		6A(A)		66	76
RTI	RETURN FROM INTERRUPT	P/ PC/S		40			
RTS	RETURN FROM SUBROUTINE	PC/S		60			
SBC	SUBTRACT WITH CARRY	A			E9	E5	F5
SEC	SET CARRY	P		38			
SED	SET DECIMAL	P		F8			
SEI	SET INTERRUPT MASK (DISABLE)	P		78			
STA	STORE ACCUMULATOR	M	A			85	95
STX	STORE REG-X	M	X			86	
STY	STORE REG-Y	M	Y			84	94
TAX	TRANSFER A TO X	X	A	AA			
TAY	TRANSFER A TO Y	Y	A	A8			
TSX	TRANSFER S TO X	X	S	BA			
TXA	TRANSFER X TO A	A	X	8A			
TXS	TRANSFER X TO S	S	X	9A			
TYA	TRANSFER Y TO A	A	Y	98			

Z-PAGE Y	INDIRECT X	INDIRECT Y	RELATIVE ADDRESS	ABSOLUTE ADDRESS	ABSOLUTE X	ABSOLUTE Y	INDIRECT	P-REGISTER N	V	*	B	D	I	Z	C
				20											
	A1	B1		AD	BD	B9		X						X	
B6				AE		BE		X						X	
				AC	BC			X						X	
				4E	5E			0						X	X
	01	11		0D	1D	19		X						X	
								X						X	
								X	X	X	X	X	X	X	X
				2E	3E			X						X	X
				6E	7E			X						X	X
								X	X	X	X	X	X	X	X
	E1	F1		ED	FD	F9		X	X					X	X
												1			1
													1		
	81	91		8D	9D	99									
96				8E											
				8C											
								X						X	
								X						X	
								X						X	
								X						X	
								X						X	

Appendix 2
Op codes, tokens and ASCII characters

DEC	HEX	BASIC TEXT	6502 OP-CODE	ASCII CHAR CODE
0	00	END-OF-LINE	BRK	
1	01		ORA ($00,X)	
2	02			
3	03			
4	04			
5	05		ORA $00	WHITE
6	06		ASL $00	
7	07			
8	08		PHP	SHFT/COM +
9	09		ORA #$00	SHFT/COM -
10	0A		ASL A	
11	0B			
12	0C			
13	0D		ORA $0000	RETURN
14	0E		ASL $0000	LOW CASE +
15	0F			
16	10		BPL	
17	11		ORA ($00),Y	CURS DOWN
18	12			RVS ON
19	13			CLR/HOME
20	14			INST/DEL
21	15		ORA $00,X	
22	16		ASL $00,X	
23	17			
24	18		CLC	
25	19		ORA $0000,Y	
26	1A			
27	1B			
28	1C			RED
29	1D		ORA $0000,X	CURS RIGHT
30	1E		ASL $0000,X	GREEN
31	1F			BLUE
32	20	SPACE	JSR $0000	SPACE
33	21	!	AND ($00,X)	!
34	22	"		"
35	23	#		#
36	24	$	BIT $00	$
37	25	%	AND $00	%
38	26	&	ROL $00	&

39	27	'		'
40	28	(	PLP	(
41	29	)	AND #$00	)
42	2A	*	ROL A	*
43	2B	+		+
44	2C	,	BIT $0000	,
45	2D	-	AND $0000	-
46	2E	.	ROL $0000	.
47	2F	/		/
48	30	0	BMI	0
49	31	1	AND ($00),Y	1
50	32	2		2
51	33	3		3
52	34	4		4
53	35	5	AND $00,X	5
54	36	6	ROL $00,X	6
55	37	7		7
56	38	8	SEC	8
57	39	9	AND $0000,Y	9
58	3A	:		:
59	3B	;		;
60	3C	<		<
61	3D	=	AND $0000,X	=
62	3E	>	ROL $0000,X	>
63	3F	?		?
64	40	@	RTI	@
65	41	A	EOR ($00,X)	A
66	42	B		B
67	43	C		C
68	44	D		D
69	45	E	EOR $00	E
70	46	F	LSR $00	F
71	47	G		G
72	48	H	PHA	H
73	49	I	EOR #$00	I
74	4A	J	LSR A	J
75	4B	K		K
76	4C	L	JMP $0000	L
77	4D	M	EOR $0000	M
78	4E	N	LSR $0000	N
79	4F	O		O
80	50	P	BVC	P
81	51	Q	EOR ($00),Y	Q
82	52	R		R
83	53	S		S
84	54	T		T
85	55	U	EOR $00,X	U
86	56	V	LSR $00,X	V
87	57	W		W
88	58	X	CLI	X
89	59	Y	EOR $0000,Y	Y
90	5A	Z		Z
91	5B			
92	5C			
93	5D		EOR $0000,X	
94	5E		LSR $0000,X	
95	5F			

96	60		RTS	
97	61		ADC (*00,X)	
98	62			
99	63			
100	64			
101	65		ADC *00	
102	66		ROR *00	
103	67			
104	68		PLA	
105	69		ADC #*00	
106	6A		ROR A	
107	6B			
108	6C		JMP (*0000)	
109	6D		ADC *0000	
110	6E		ROR *0000	
111	6F			
112	70		BVS	
113	71		ADC (*00),Y	
114	72			
115	73			
116	74			
117	75		ADC *00,X	
118	76		ROR *00,X	
119	77			
120	78		SEI	
121	79		ADC *0000,Y	
122	7A			
123	7B			
124	7C			
125	7D		ADC *0000,X	
126	7E		ROR *0000,X	
127	7F			
128	80	END		
129	81	FOR	STA (*00,X)	ORANGE
130	82	NEXT		
131	83	DATA		
132	84	INPUT#	STY *00	
133	85	INPUT	STA *00	F1
134	86	DIM	STX *00	F3
135	87	READ		F5
136	88	LET	DEY	F7
137	89	GOTO		F2
138	8A	RUN	TXA	F4
139	8B	IF		F6
140	8C	RESTORE	STY *0000	F8
141	8D	GOSUB	STA *0000	SHFT/RET
142	8E	RETURN	STX *0000	UP CASE +
143	8F	REM		
144	90	STOP	BCC	BLACK
145	91	ON	STA (*00),Y	CURS UP
146	92	WAIT		RVS OFF
147	93	LOAD		CLR/HOME
148	94	SAVE	STY *00,X	INST/DEL
149	95	VERIFY	STA *00,X	BROWN
150	96	DEF	STX *00,Y	LT. RED
151	97	POKE		GREY 1
152	98	PRINT#	TYA	GREY 2
153	99	PRINT	STA *0000,Y	LT. GREEN

154	9A	CONT	TXS	LT. BLUE
155	9B	LIST		GREY 3
156	9C	CLR		PURPLE
157	9D	CMD	STA $0000,X	CURS LEFT
158	9E	SYS		YELLOW
159	9F	OPEN		CYAN
160	A0	CLOSE	LDY #$00	SPACE
161	A1	GET	LDA ($00,X)	
162	A2	NEW	LDX #$00	
163	A3	TAB(		
164	A4	TO	LDY $00	
165	A5	FN	LDA $00	
166	A6	SPC(	LDX $00	
167	A7	THEN		
168	A8	NOT	TAY	
169	A9	STEP	LDA #$00	
170	AA	+	TAX	
171	AB	-		
172	AC	*	LDY $0000	
173	AD	/	LDA $0000	
174	AE		LDX $0000	
175	AF	AND		
176	B0	OR	BCS	
177	B1	>	LDA ($00),Y	
178	B2	=		
179	B3	<		
180	B4	SGN	LDY $00,X	
181	B5	INT	LDA $00,X	
182	B6	ABS	LDX $00,Y	
183	B7	USR		
184	B8	FRE	CLV	
185	B9	POS	LDA $0000,Y	
186	BA	SQR	TSX	
187	BB	RND		
188	BC	LOG	LDY $0000,X	
189	BD	EXP	LDA $0000,X	
190	BE	COS	LDX $0000,Y	
191	BF	SIN		
192	C0	TAN	CPY #$00	
193	C1	ATN	CMP ($00,X)	
194	C2	PEEK		
195	C3	LEN		
196	C4	STR$	CPY $00	
197	C5	VAL	CMP $00	
198	C6	ASC	DEC $00	
199	C7	CHR$		
200	C8	LEFT$	INY	
201	C9	RIGHT$	CMP #$00	
202	CA	MID$	DEX	
203	CB	GO		
204	CC	JOY*	CPY $0000	
205	CD	FIRE*	CMP $0000	
206	CE	XSPR*	DEC $0000	
207	CF	YSPR*		
208	D0	DUMMY*	BNE	
209	D1	DUMMY*	CMP ($00),Y	
210	D2	DUMMY*		

```
211  D3    DUMMY*
212  D4    DUMMY*
213  D5    DUMMY*          CMP $00,X
214  D6    DUMMY*          DEC $00,X
215  D7    DUMMY*
216  D8    DUMMY*          CLD
217  D9    DUMMY*          CMP $0000,Y
218  DA    RENUMBER*
219  DB    DELETE*
220  DC    REPLACE*
221  DD    OLD*            CMP $0000,X
222  DE    AUTO*           DEC $0000,X
223  DF    PUTXY*
224  E0    HIMEM*          CPX #$00
225  E1    SPRITE*         SBC ($00,X)
226  E2    SOUND*
227  E3    BITMAPC*
228  E4    BITMAP*         CPX $00
229  E5    MOVE*           SBC $00
230  E6    PLOT*           INC $00
231  E7    UNPLOT*
232  E8    DRAW*           INX
233  E9    UNDRAW*         SBC #$00
234  EA    TEXT*           NOP
235  EB    DUMMY*
236  EC    SCROLL*         CPX $0000
237  ED                    SBC $0000
238  EE                    INC $0000
239  EF
240  F0                    BEQ
241  F1                    SBC ($00),Y
242  F2
243  F3
244  F4
245  F5                    SBC $00,X
246  F6                    INC $00,X
247  F7
248  F8                    SED
249  F9                    SBC $0000,Y
250  FA
251  FB
252  FC
253  FD                    SBC $0000,X
254  FE                    INC $0000,X
255  FF
```

Appendix 3
Useful memory locations

All addresses in hex.

00	Data direction register for 6510 on-chip I/O Port
01	6510 I/O Port
14,15	Safe locations
19–21	Safe locations
2B/2C	Start-of-BASIC pointer
2D/2E	Start-of-Variables pointer
2F/30	Start-of-Arrays pointer
31/32	End-of-Arrays (+1) pointer
33/34	Bottom-of-strings pointer
37/38	Top-of-BASIC pointer
4B–53	Safe locations
54	JMP instruction for Functions
55–60	Safe locations
61–70	Floating point accumulators #1 and #2
73–8A	CHRGET routine
91	<STOP> and <RVS> keys flag
A0–A2	Jiffy clock
A3/A4	DO NOT USE! Temporary store for Kernal I/O status
C5	Current key pressed
C6	Number of characters in keyboard buffer
C7	1=Print REVERSE characters, 0=don't
CC	0=Flash cursor, 1=don't
CE	Character under cursor
D3	Cursor column on current line
D4	Quote mode flag: 0 when *not* in mode
FB–FE	Safe locations

0100–01FF	Stack
0200–0258	System Input Buffer
0277–0280	Keyboard Buffer
0286	Current character colour
0287	Background colour under cursor
0288	Editor screen page,=Screen Address divided by 256

0289	Size of keyboard buffer
028A	If #$80 *all* keys repeat
028B	Repeat speed
028C	Repeat delay
028D	<SHIFT> and <CONTROL> keys flag
0291	0=Disable Shift, #$80=enable
02A7–02FF	Safe locations
0300/0301	Error vector
0302/0303	Warm Start vector
0304/0305	Tokenise vector
0306/0307	Token List vector
0308/0309	Token Dispatch vector
030A/030B	Token Evaluation vector
030C	Storage of A for SYS
030D	Storage of X for SYS
030E	Storage of Y for SYS
030F	Storage of SP for SYS
0310	JMP Instruction for BASIC USR command
0311/0312	USR address
0313	Safe location
0314/0315	IRQ vector
0316/0317	BRK vector
0318/0319	NMI vector
031A/031B	Kernal OPEN vector
031C/031D	Kernal CLOSE vector
031E/031F	Kernal CHKIN vector
0320/0321	Kernal CHKOUT vector
0322/0323	Kernal CLRCHN vector
0324/0325	Kernal CHRIN vector
0326/0327	Kernal CHROUT vector
0328/0329	Kernal STOP vector
032A/032B	Kernal GETIN vector
032C/032D	Kernal CLALL vector
032E,032F	Safe locations
0330/0331	Kernal LOAD vector
0332/0333	Kernal SAVE vector
0334–03FF	Safe locations
0400–07F7	Screen
07F8–07FF	Sprite pointers
0800–9FFF	BASIC Text area
(8000–9FFF)	(Cartridge ROM)
A000–BFFF	BASIC ROM

C000–CFFF Safe RAM

D000–DFFF I/O *or* Character Generator ROM

D000–D02E VIC II Chip
D400–D7FF SID Chip
D500–D7FF SID Images

D800–DBFF Colour Table

DC00–DCFF CIA #1
DD00–DDFF CIA #2

E000–FFFF Kernal ROM

FFFA/FFFB 6510 NMI vector
FFFC/FFFD 6510 Cold Start vector
FFFE/FFFF 6510 IRQ vector

Appendix 4
ROM routines

This appendix provides the vector or absolute addresses of the BASIC ROM routines used in the programs given in this book. Note that some operations require two routines to be called in succession, and one given below a specific sequence of instructions. Refer to the program code and descriptions for examples of the use of these routines.

Get in Data

$0073	Get next byte from input buffer in A
$0079	Get current byte from buffer in A
$A96B	Get an ASCII number <64000 (not expresssions) in $14, $15
$AD8A $B7F7	Get an expression or variable in $14, $15
$AD9E $B6A6	Get a string (in quotes) and store at an address in $22, $23. Length of string is in X
$AEF1	Evaluate an expression or variable in brackets in $14, $15
$AEFD	Get in a data byte preceded by a comma. Return with byte in A
$B79E	Evaluate a byte variable or expression as an integer in X
$B7EB	Get two numbers separated by a comma. The first is a number <65535 in $14, $15 and the second an expression or variable <256 in X

Output Data

$AB1E	Output to the screen a string of ASCII bytes terminated by zero. The address of the string is stored in A (low byte) and Y (high byte)

$B391 Convert an integer in Y (low byte) and A (high byte) to floating point form in floating point accumulator#1 ($61–$66)

$BDCD Output a number in X (low byte) and A (high byte) to screen as decimal

$FFD2 Output ASCII character in A to screen

Basic Routines

$A533 Rebuild 'chaining' of link addresses in BASIC text

$A613 Search for a line number stored in $14, $15. Returns address of line in $5F, $60

$A660 CLR routine

Miscellaneous Routines

$0079

$A96B Convert ASCII decimal number to binary in $14, $15. ASCII number is stored in a buffer whose address is in $7A, $7B

LDX #$90
SEC
JSR $BC49
JSR $BDDF Convert binary number in $62, $63 into ASCII decimal at $0100 onwards

$E8EA Scroll the screen up

Appendix 5
Book program memory map

This appendix provides the overall memory areas used by the major programs given in this book, 'Wedge-MON' and 'Expander', as well as the start and end addresses for each separate routine or memory area within them. The smaller routines given in the text, as well as the multiple sprite routine and the interrupt timer routine given in the appendices will conflict with 'Wedge-MON', as they also use the free RAM area from $C000 onwards, as they are listed. If you wish to use these routines, or any of your own, they will require to be relocated to a safe area of memory.

$C000–CD77	'Wedge–MON'
$C000–C23A	Control program
$C300–C816	Command file 1: @D, @A
$C880–C942	Command file 2: @M, @I, @W, @C
$C980–CA00	Command file 3: @T
$CA80–CABC	Command file 4: @F
$CB00–CBC2	Command file 5: @H
$CC00–CC8A	Command file 6: @G, @R
$CD00–CD77	Command file 7: @S, @L
$5C00–$9FFF	'Expander'
$5C00–5FFF	Colour table for bit-map
$6000–7FFF	Bit-map of screen
$8000–8343	'Command Handler'
$8500–89F9	'Toolkit': RENUMBER/DELETE/OLD AUTO/SEARCH/REPLACE
$8A00–8A79	INK/PUTXY/HIMEM
$8B00–8C47	SPRITE/KSPRITE
$8D00–8EB5	JSPRITE
$8F00–8FF8	SOUND
$9100–9172	JOY/FIRE/XSPR/YSPR
$9200–92AA	SCROLL
$9300–96EB	BITMAPC/BITMAP/MOVE/PLOT/UNPLOT/TEXT/DRAW/UNDRAW
$9A00–9FFF	(RAM PLOT table)

Note that locations not noted within 'Expander' and 'Wedge–MON' memory are free for user routines, e.g. $8EB6 (end of JSPRITE) to $8EFF (start of SOUND)

Appendix 6
SOUND frequency table

The table given below provides the information required for programming musical note values using the SOUND command of 'Expander', or for that matter by use of standard BASIC techniques. The note values for each semitone in six octaves are given, with the corresponding audible frequency (Hertz) value produced, and the frequency register values required by the SID chip. The values given have been adjusted for an even-tempered scale based on 440 cps for concert A, and are correct for the system clock of the 64 running in the UK, as opposed to the CBM documentation, which is based on the (higher) clock value used in the USA.

Hertz	Note	Octave	Frequency
62	B	1	1055
65	C	2	1106
69	C#	2	1174
73	D	2	1243
78	D#	2	1328
82	E	2	1396
87	F	2	1481
92	F#	2	1566
98	G	2	1668
104	G#	2	1770
110	A	2	1873
116	A#	2	1975
123	B	2	2094
131	C	3	2230
139	C#	3	2366
147	D	3	2503
156	D#	3	2656
165	E	3	2809
175	F	3	2979
185	F#	3	3150
196	G	3	3337
208	G#	3	3541
220	A	3	3746

Hertz	Note	Octave	Frequency
233	A#	3	3967
247	B	3	4206
262	C	4	4461
277	C#	4	4716
294	D	4	5006
311	D#	4	5295
330	E	4	5619
349	F	4	5942
370	F#	4	6300
392	G	4	6675
415	G#	4	7066
440	A	4	7492
466	A#	4	7935
494	B	4	8412
523	C	5	8905
554	C#	5	9433
587	D	5	9995
622	D#	5	10591
659	E	5	11221
698	F	5	11885
740	F#	5	12601
784	G	5	13350
831	G#	5	14150
880	A	5	14984
932	A#	5	15870
988	B	5	16824
1046	C	6	17811
1109	C#	6	18884
1175	D	6	20008
1244	D#	6	21183
1318	E	6	22443
1397	F	6	23788
1480	F#	6	25202
1568	G	6	26700
1661	G#	6	28284
1760	A	6	29969
1865	A#	6	31757
1976	B	6	33648
2093	C	7	35640
2217	C#	7	37751
2349	D	7	39999
2489	D#	7	42383

Hertz	Note	Octave	Frequency
2637	E	7	44903
2794	F	7	47577
2960	F#	7	50404
3136	G	7	53401
3322	G#	7	56568
3520	A	7	59939
3729	A#	7	63498

Appendix 7
Number systems

Since we tend to regard numbers as the symbolic representation of objective universal values, it is somewhat sobering to realise that like the languages we use to communicate with one another, systems of numbers are just another rather clumsy attempt by humans to impose their own order on an anarchic universe. The extent to which we have programmed ourselves to regard numbers as the symbols of truth is supported by the fact that any given symbol or collection of symbols is perceived as an expression in its own right, without reference to the system of representation of which it is part.

For most of us in the West, our perception of the 'reality' signified by a number is actually determined by its relationship to the decimal system that constitutes the prevalent method of expressing quantity. One of the first hurdles confronting most newcomers to computing is the need to become acquainted with two alternative sets of value representation (one of which boasts a brace of subsets critical to the comprehension and application of machine code).

Binary representation

The decimal system with which we are all familiar makes use of ten symbols (0–9) which can be used in combination to express an infinite range of values. This method is referred to as a 'base 10' system of representation, since the significance of a digit is determined by its group position and each position expresses the power of the base value (10). Thus we assign meaning to the symbols 203 by the automatic recognition that:

```
          10^2    10^1    10^0

203=2*10^2+0*10^1+3*10^0
```

The fundamental logic circuits (AND, OR gates, adder circuits, memory bits, etc) of modern digital computers operate on two different voltage levels – high and low. The initiation of any digital computing operation is ultimately determined by the recognition of

specific sequences of high/low voltage settings. For processing to be established by the electrical expression of decimal values, a computer's processor would require the design of a circuit that recognised an impractical ten voltage variations. By making use of the binary system we can circumvent the need for cumbersome circuitry and express an adequate value range using combinations of only two symbols – 0 and 1.

As its name implies, binary is a base 2 system, that is, a symbols position within a group expresses a power of 2 (as opposed to decimal's power of 10). Obviously, the lower the base value utilised by a system, the more digits required for the expression of a quantity. Thus while decimal can represent two hundred and three with only three symbols, the same value in binary requires eight:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	0	1	0	1	1

=1*2^7+1*2^6+0*2^5+0*2^4+1*2^3+0*2^2+1*2^1+1*2^0

Each machine code instruction is signalled by a specific sequence of eight high (1)/low (0) voltages which collectively express a binary value. Each group of digits is referred to as a byte, and each element of a byte is called a bit.

Within any given byte, individual bits are usually identified by a number (0–7) which expresses their position. These numbers are used as a means of establishing the weight of a given bit. Bit numbers increase from left to right, thus:

7	6	5	4	3	2	1	0
128	64	32	16	8	4	2	1

The line of values below the bit numbers signify the weights of individual bits (calculated by raising each bit-value to the power of the base value – i.e. ^2).

Hexadecimal representation

Despite the fact that binary notation provides us with a system of value representation that clearly expresses the manner in which numbers are stored and are identified by digital computers, its endless strings of digits are somewhat cumbersome. Although at first glance it may be difficult to believe, the hexadecimal number

system, which is organised to base 16, can make a programmer's life considerably easier.

In addition to the standard symbols used by the binary system, hexadecimal notion also makes use of the first six letters of the alphabet. Let's take a look at the various representations of decimal 0–15 for the number systems we have discussed thus far, as shown below.

DEC	HEX	BIN
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Thus if we want to express our example value of two hundred and three in hexadecimal notion we would write it thus:

```
16^1   16^0

C      B       =12*16^1+11*16^0
```

As we have seen, one byte comprises eight bits, and each byte is used to hold the binary representation of a value (with each bit storing a binary digit). Thus any byte can hold values from 0 to 11111111 binary (i.e. 0 to 255 decimal).

Perhaps now the value of hexadecimal notion is a little clearer. Since each hexadecimal can be used to represent four 'binary' bits, a complete byte can be expressed by a two digit hexadecimal number (0 to FF).

In the ever creative world of computing terminology a four bit sequence is referred to as a nybble (which can be represented by a single hexadecimal digit), and thus one byte equals two nybbles

(two digits in hex). To differentiate between the two nybbles in a byte, bits 0–3 are known as the lower-order nybble and bits 4–7 as the higher-order nybble, or low and high nybbles.

Addition and subtraction in binary

Before we can outline the notational subset of the binary system, we must take a quick look at how simple addition and subtraction are performed. The rules for addition are simple:

```
1+1=(1)0
1+0=1
0+1=1
0+0=0
```

You'll notice that in the first example line both a zero and a one are produced. The one is known as a Carry bit which indicates an 'overflow' from one column to another. Let's see how this works in practice with an example of four bit binary addition:

```
  1001
+ 0101
------
  1110
```

As you can see, column one offers an example of a Carry bit, in which the relevant solution column takes a 0 and 1 is carried over to the next column. The same process can be applied to binary values which are stored in eight bits:

```
01010011
01110100
--------
10100111
```

Right, now we get to the tricky bit!. While the idea may take a little getting used to, you're going to have to face up to the grim truth that in binary arithmetic the process of subtraction is actually an act of addition! Well, at least it is the addition of a positive value to a negative value. Let's see if we can make things a little clearer.

We'll start by thinking in decimal. Imagine that you're standing at point zero. In one direction stretches an infinity of positive values, each engraved on a numeric paving stone; in the opposite direction an endless sequential footpath of negative values – and you're in the middle of it all. Now say we wish to calculate a simple subtraction,

for example, 8–3. We tackle the positive value first. Without hesitation we turn and face the path of positive increment, and boldly traverse eight full steps. With hardly a pause for breath, we about turn to face zero and add to our journey by taking three equally positive steps in the negative direction. Squinting down at our feet we see that we've arrived on a stone storing a positive value of 5, which is something of a relief having adopted such a tortuous path to discover the result of 8–3!

The purpose of this rather tedious numeric travelogue is to subtly modify your concept of subtraction. Instead of considering it as a process which takes away a specified quantity from a self-contained value, you'll hopefully be able to regard it as a continuous numeric progress whose destination is determined by directions indicated by one of two signs (+ or −).

Before we move on to see how this interminable analogy applies to the inner world of your micro, we need to clarify the manner in which negative numbers are represented in binary. We've explained how one byte is composed of eight bits, each of which stores a single binary digit. We also learned that each bit has a unique 'weight' and is normally described by one of eight referential values (0–7) allocated in descending order from left to right. To tackle the intricacies of signed binary arithmetic, we must focus our attention on bit 7 (the one on the far left of the byte in question).

Bit 7 is known as the 'most significant bit' (msb), and establishes the sign of a number. Under most circumstances, when this bit is 'unset' (0), a positive value is indicated and 'set' (1) when a negative value is represented. So, when using signed binary notion we establish values and their signs like this:

Let's take decimal 8. The value itself is expressed in bits 0–6, whilst bit 7 determines the sign. Thus:

```
+8=0 0001000
−8=1 0001000
```

All well and good. However, the manipulation of the msb is only part of the story of signed binary. Negative value representation can only be accurately implemented with the introduction of the two notational sub-systems we mentioned at the beginning of this section – 1's and 2's complement

1's and 2's Complement

To convert a positive binary value into its negative equivalent, we must obtain its '2's complement'. This may sound a little daunting, but actually it simply involves an undemanding inversion.

Let's return to our original subtraction and concentrate on the −3. The first step in the conversion process is to take a look at the positive binary form of this value:

00000011

The next stage is where '1's complement' comes in. This is achieved by simply inverting the value stored in each bit, thus:

11111100

Now we're very clost to a binary negative, all we have to do is add 1 to our 1's complement conversion:

```
  11111100
       +1
  --------
  11111101
```

The 2's complement value returned is the binary representation of −3 which we can now add to the positive B in our original sum:

```
    00001000
    11111101
  --------
(1) 00000101
```

Since you've almost certainly had your fill of numbers systems and their consequences, we'll draw a veil over the overflow from bit 7 (1). Suffice it to say that the result of our efforts has generated a binary value – 00000101 – which you'll be relieved to learn is the representation of decimal 5. Which only goes to show that even binary notion can convert decimal 8 – 3 into something close to the correct result!

Appendix 8
Multiple sprite routine

Assembler Listing

```
1020 ;----------
1030 ;--DISPLAY 64 SPRITES
1040 ;SIMULTANEOUSLY ON SCREEN USING
1050 ;RASTER INTERRUPTS--
1060 ;----------
1070 ;THE IDEA OF THIS PROGRAM IS TO
1080 ;CREATE RASTER INTERRUPTS AT 8
1090 ;POINTS DOWN THE SCREEN. EACH
1100 ;INTERRUPT RESETS THE Y POSITION
1110 ;OF A LINE OF 8 SPRITES ENABLING
1120 ;EACH SPRITE TO APPEAR AT 8
1130 ;POSITIONS ON THE SCREEN
1140 ;----------
1150 ;THIS ROUTINE CAN BE ADAPTED TO
1160 ;PRODUCE OTHER RASTER EFFECTS SUCH
1170 ;AS MIXED GRAPHICS MODES AND
1172 ;COLOUR BANDING
1180 ;----------
1190 *=$C000
1200 ;----------
1210 ;--DEFINE A FILLED-IN SPRITE--
1220 ;SPRITE BLOCK 14 ($0E) IS FILLED
1230 ;IN
1240 ;----------
1250         LDX #$3F
1260         LDA #$FF
1270 L1      STA $0380,X
1280         DEX
1290         BPL L1
1300 ;----------
1310 ;--SET UP 8 SPRITES--
1320 ;8 GREEN SPRITES ARE POSITIONED
1330 ;ACROSS THE SCREEN/THEY ARE SET
```

```
1340 ;TO SPRITE BLOCK 14 ($0E) AND
1350 ;ENABLED
1360 ;----------
1370         LDY #$07
1380         LDX #$0E
1390 LOOP1   LDA #$05
1400         STA $D027,Y
1410         LDA SPX,Y
1420         STA $D000,X
1430         LDA #$0E
1440         STA $07F8,Y
1450         DEX
1460         DEX
1470         DEY
1480         BPL LOOP1
1490 ;----------
1500         LDA #$C0
1510         STA $D010
1520         LDA #$00
1530         STA $D017
1540         STA $D01D
1550         LDA #$FF
1560         STA $D015
1570 ;----------
1580 ;--SET UP INTERRUPT--
1590 ;----------
1600         SEI
1610 ;----------
1620 ;KILL ALL CIA INTERRUPTS
1630 ;----------
1640         LDA #$7F
1650         STA $DC0D
1660 ;----------
1670 ;ENABLE RASTER INTERRUPT/RESET IRQ
1680 ;VECTOR/SET INITIAL POSITION OF
1690 ;RASTER IRQ AND INITIAL Y POS OF
1700 ;SPRITES (RESET)
1710 ;----------
1720         LDA #$01
1730         STA $D01A
1740         STA $D019
1750         LDA #<IRQ
1760         LDY #>IRQ
1770         STA $0314
```

```
1780          STY $0315
1790          JSR RESET
1800          CLI
1810 ;----------
1820 ;--MAIN PROGRAM WAITS IN AN
1830 ;ENDLESS LOOP--
1840 ;----------
1850 WAIT     JMP WAIT
1860 ;----------
1870 ;--LOW BYTES OF X-POSITIONS FOR 8
1880 ;SPRITES--
1890 ;----------
1900 SPX      .BYTE $40,$60,$80,$A0
1910          .BYTE $C0,$E0,$00,$20
1920 ;----------
1930 ;--IRQ HANDLER--
1940 ;----------
1950 ;RESTART RASTER INTERRUPT
1960 ;----------
1970 IRQ      LDA #$01
1980          STA $D019
1990 ;----------
2000 ;RESET Y-POSITIONS FOR 8 SPRITES/
2010 ;SETUP NEXT RASTER INTERRUPT
2020 ;POSITION/
2030 ;----------
2040          JSR SET
2050 ;----------
2060 ;EXIT IRQ VIA A ROM ROUTINE TO
2070 ;PULL REGISTERS AND RTI/
2080 ;----------
2090          JMP $EA81
2100 ;----------
2110 ;SET UPDATES SPRITE Y POSITIONS
2120 ;AND SETS THE NEW RASTER INTERRUPT
2130 ;POSITION/
2140 ;RESET RESETS RASTER INTERRUPT TO
2150 ;THE TOP OF THE SCREEN
2160 ;----------
2170 SET      LDA $D012
2180          CMP #$E5
2190          BCC ONGO
2200 ;----------
2210 RESET    CLC
```

```
2220          LDA #$30
2230 ONGO     LDX #$0E
2240          ADC #$03
2250 OG1      STA $D001,X
2260          DEX
2270          DEX
2280          BPL OG1
2290          ADC #$16
2300          STA $D012
2310          LDA $D011
2320          AND #$7F
2330          STA $D011
2340          RTS
2350 ;----------
2360 .END
```

BASIC Loader

```
100 C=0
101 READ D$
102 IF D$="END" THEN PRINT D$:END
103 D=VAL(D$):IFD<0 THEN 106
104 X=X+1:C=C+D:POKE 49151+X,D
105 GOTO 101
106 IFC=ABS(D) THEN 100
107 PRINT "ERROR IN BLOCK ";INT(X/64)
108 GOTO 100
110 REM --- BLOCK 1
111 DATA162,63,169,255,157,128,3,202
112 DATA16,250,160,7,162,14,169,5
113 DATA153,39,208,185,84,192,157,0
114 DATA208,169,14,153,248,7,202,202
115 DATA136,16,235,169,192,141,16,208
116 DATA169,0,141,23,208,141,29,208
117 DATA169,255,141,21,208,120,169,127
118 DATA141,13,220,169,1,141,26,208
119 DATA-8304
120 REM --- BLOCK 2
121 DATA141,25,208,169,92,160,192,141
122 DATA20,3,140,21,3,32,110,192
123 DATA88,76,81,192,64,96,128,160
124 DATA192,224,0,32,169,1,141,25
125 DATA208,32,103,192,76,129,234,173
126 DATA18,208,201,229,144,3,24,169
```

```
127 DATA48,162,14,105,3,157,1,208
128 DATA202,202,16,249,105,22,141,18
129 DATA-7114
130 REM --- BLOCK 3
131 DATA208,173,17,208,41,127,141,17
132 DATA208,96
133 DATA-1236
134 DATA END
```

Appendix 9
Timer interrupt routine

Assembler Listing

```
1030 ;-----------
1040 ;--SETTING UP A TIMER INTERRUPT
1050 ;OFF CIA TIMER A--
1060 ;-----------
1070 ;THIS WILL BE REQUIRED WHERE AN
1080 ;INTERRUPT IS NEEDED IN A MACHINE
1090 ;CODE PROGRAM WHERE IT IS NOT
1100 ;POSSIBLE TO 'BORROW' BASIC'S.
1110 ;-----------
1120 *=$C000
1130 ;-----------
1140 ;--SET IRQ VECTOR--
1150 ;-----------
1160         LDA #<IRQ
1170         LDY #>IRQ
1180         STA $0314
1190         STY $0315
1200 ;----------
1210 ;--SET TIMER SPEED--
1220 ;($411A = 60 IRQ'S PER SECOND)
1230 ;----------
1240         LDA #$1A
1250         LDY #$41
1260         STA $DC04
1270         STY $DC05
1280 ;----------
1290 ;--ENABLE TIMER A INTERRUPTS--
1300 ;----------
1310         LDA #$81
1320         STA $DC0D
1330 ;----------
1340 ;--SET 'ONE-SHOT' MODE IN TIMER--
1350 ;----------
```

```
1360          LDA ##$91
1370          STA $DC0E
1380 ;----------
1390 ;--ENABLE IRQS AND BACK TO BASIC--
1400 ;----------
1410          CLI
1420          RTS
1430 ;----------
1440 ;--IRQ HANDLER--
1450 ;INCREMENT SCREEN COLOUR SIXTY
1460 ;TIMES PER SECOND/
1470 ;----------
1480 IRQ      INC $D021
1490 ;----------
1500 ;--IRQ EXIT--
1510 ;IT IS NECESSARY TO READ $DC0D
1520 ;IN ORDER TO RESTART TIMER/FINALLY
1530 ;PULL REGISTERS AND EXIT VIA RTI
1540 ;----------
1550          LDA $DC0D
1560          PLA
1570          TAY
1580          PLA
1590          TAX
1600          PLA
1610          RTI
1620 ;----------
```

INDEX

Index

Y

Z

Robert Erskine & Humphrey Walwyn with Paul Stanley and Michael Bews
Sixty Programs for the Sinclair ZX Spectrum £5.95

Sixty Programs for the BBC Micro £5.95

Sixty Programs for the Dragon 32 £5.95

Sixty Programs for the Oric 1 £5.95

Sixty Programs for the Atari £5.95

Sixty Programs for the Commodore 64 £5.95

Sixty Programs for the Vic 20 £5.95

Sixty Programs for the Electron £5.95

Ian Adamson
The Companion to the Oric 1 £5.95

Geoff Wheelwright
The Companion to the BBC Micro £4.95

Keith Bowden
The Companion to the Commodore 64 £5.95

Jeremiah Jones and Geoff Wheelwright
The Companion to the Electron £5.95

Jean Frost
Instant Arcade Games for the Sinclair ZX Spectrum £3.95

Instant Arcade Games for the BBC Micro £3.95

Instant Arcade Games for the Dragon 32 £3.95

Instant Arcade Games for the Electron £3.95

J. J. Clessa
Micropuzzles £2.95

Ian Scales
Spectrum Peripherals Guide £4.95

Tony Takoushi
Best Software Guide: Vic 20/Commodore 64 £3.95

Best Software Guide: Spectrum Games £3.95

Jeff Aughton
Invaluable Utilities for your BBC Micro £5.95

All these books are available at your local bookshop or newsagent, or can be ordered direct from the publisher. Indicate the number of copies required and fill in the form below.

Name_____
(Block letters please)

Address_____

_____

Send to Pan Books (CS Department), PO Box 40, Basingstoke, Hants. Please enclose remittance to the value of the cover price plus: 35p for the first book plus 15p per copy for each additional book ordered to a maximum charge of £1.25 to cover postage and package. Applicable only in the UK.

While every effort is made to keep prices low, it is sometimes necessary to increase prices at short notice. Pan Books reserve the right to show on covers and charge new retail prices which may differ from those advertised in the text or elsewhere.

MASTER YOUR MICRO'S MOTHER TONGUE!

Harness the full power of the 64's hardware and escape the confines of BASIC with this practical guide to machine code programming. The path to professional programming expertise is made clear as you are introduced to the hardware and the 6510 instruction set and learn to combine the separate elements of machine code into the fast and efficient code found in commercial programs.

A full machine code monitor with 14 commands gives you the essential tools to interface with the 64's architecture, assemble the annotated routines and access the speed and sophistication of machine code. Learn good programming practice and the tricks of the trade as you use the sprite, sound and hi-res graphics routines, and get to grips with interrupt handling for multiple sprites and smooth screen scrolls in your own programs.

Enhance your 64's abilities with the toolkit of extra BASIC commands (including auto line number, full renumber, and search/replace facilities) and learn how to incorporate machine code routines as additional BASIC statements, both the ones provided in this book, and the ones you'll go on to write once you've mastered the process of 'Cracking the Code'.

Cover photography by Peter Williams
with grateful acknowledgement for the kind assistance of
Chubb & Son's Lock & Safe Co. Ltd.

ISBN 0-330-28664-1

U.K. £6.95

90000

9 780330 286640