



BEST OF PCW

ASSEMBLER ROUTINES

FOR THE
6502

David Barrow

Best of
Personal Computer World
ASSEMBLER ROUTINES
FOR THE 6502

David Barrow



Century Communications

London

ACKNOWLEDGEMENTS

I would particularly like to thank all the contributors to *Sub Set* over the years.

Without them
the series and this book
would not have been possible.

Copyright © David Barrow 1985

Based on material published in *Personal Computer World* magazine.

All rights reserved

First published in Great Britain in 1985 by

Century Communications Limited
12-13 Greek Street, London, W1V 5LE

ISBN 0 7126 0507 X

Edited and produced working directly from the author's word-processor by NWL Editorial Services, Rushley, High Ham, Somerset, TA10 9DG

Printed and bound in Great Britain
by Billings & Sons Limited, Worcester.

Contents

Introduction	
CHAPTER 1	1
Delays	
CHAPTER 2	9
Extra Instructions	
CHAPTER 3	19
Register Indirect	
CHAPTER 4	27
Parameter Access, Save and Restore	
CHAPTER 5	39
Program Relative Addressing	
CHAPTER 6	51
Search, Sort and Case	
CHAPTER 7	61
Data Compression	
CHAPTER 8	73
Data Moves	
CHAPTER 9	85
Reducing Errors	
CHAPTER 10	93
Legible Listings	
CHAPTER 11	103
Dot Graphics	
CHAPTER 12	111
Random Numbers	
CHAPTER 13	125
Extracting Roots	
CHAPTER 14	139
Converting Decimal to Binary	
CHAPTER 15	149
Converting 16-bit, Gray and Others	
CHAPTER 16	159
Arithmetic: 16-bit	
CHAPTER 17	165
Arithmetic: 32-bit	
APPENDIX A	175
Index of Routines	
Index	179

Introduction

Programming in machine code has never been more popular. Almost completely dismissed a few years ago as a tedious anachronism, machine code is recognised today as essential in all programs which need absolute speed and control.

For over four years, *PCW* has published a regular series, *PCW Sub Set*, devoted entirely to assembly language programming. Here for the first time the best 6502 routines and programming hints from the series are collected together in one volume, along with 6502 conversions of some of the Z-80 routines.

All the routines have been rechecked to find those elusive bugs that somehow managed to get into the original magazine versions. The documentation and listing format has been completely revised to new *Sub Set* standards, giving as much information as possible about the routines and the way that they work.

PCW SUB SET

The following letter is an edited version of that which appeared under the heading of 'Software datasheets' in the COMMUNICATION section of *PCW* in May, 1980. Although it was aimed primarily at the Z-80 programmer, other codes (including the 6502) were not forgotten. A 6502 version of the sample routine can be found in chapter one.

SOFTWARE DATASHEETS

Nothing makes me so wild as to hear computer hobbyists being told that any program that more or less works is a good one. Bad programming, whether perpetrated by professionals or amateurs, is an abomination that pollutes the mental processes of anyone, including those who actually wrote it, who might later want to use and modify it. Hobbyists do not have to use machine code programming of anything but the highest quality.

Any software takes time to develop and test into something the originator can use. Just a little extra time could turn it into something that could be shared and improved by others, so that everyone finishes up with a set of first class software products.

I would like, through *PCW*, to get a group of people writing Z-80 general purpose subroutines, to defined standards, for criticism and improvement by others. I

would edit and check that contributions worked and conformed to standard and even supply routines, if necessary, for the first few months, to get the project off the ground. The ideas of the Z-80 routines could, of course, be worked in 6502, 6800 and other machine codes as well.

To illustrate the standards I have in mind, I list Rules and Documentation standards for the routines, together with an example. I would try to improve on the presentation of these, perhaps PCW readers could help with this.

Rules and Documentation for Software Datasheets for Z-80 general purpose subroutines. (Developed from the paper 'Microcomputer Software Design' by Thomas P. Hughes, Dwight H. Sawin III and David R. Hadden Jr. of the U.S. Army Electronics Command.)

RULES

1. Registers not being used to convey data into or out of the routine will, if used by the routine, be saved on entry to and restored before exit from the routine.

; = DL1S - One second delay at 2MHz

;/ 'DL1S' - Level 0

;/ To use 2000000 time states, inclusive of call, without other effect.

;/ ACTION: (SP-2) := L

(SP-1) := H

H := (SP+1)

L := (SP)

repeated 42,551 times

;/ INPUT: None

;/ OUTPUT: None

;/ REGs USED: None

;/ STACK USE: 6

;/ LENGTH: 19

;/ SUBr DEPENDENCIES: None

;/ INTERFACES: None

;/ 8080 COMPATIBLE?: No

DL1S

PUSH AF	; save flags	F5
PUSH BC	; & registers	C5
LD BC,42551	; set loop counter	01 37 A6
PUSH HL	; main	E5
POP HL	; delay	E1
DEC BC	; decrement counter	0B
LD A,C	; set zero flag only if both	79
OR B	; bytes of BC are zero	B0
JR NZ,-7	; jump if not zero to loop	20 F9
PUSH HL	; make up	E5
POP HL	; delay to	E1
NOP	; 1,999,983	00
NOP	; time states	00
POP BC	; restore registers	C1
POP AF	; and flags	F1
RET	; return	C9

2. It is assumed that the general routines library will always be in memory (possibly ROM) so that routines may call other general routines.

3. RAM addresses, outside the general routines' library, will never be explicitly specified in routines. References to RAM may be made through the contents of registers, which the caller supplies as pointers or as address parameters immediately following the call in the main routine.

4. Registers HL, DE, IX and IY will be used as pointers to RAM.

5. Registers B and BC will be used to pass single and double byte counts.

6. To avoid having areas of RAM that need to be defined by the user, the stack may be used for local RAM.

7. Data may be supplied to a subroutine as parameters

immediately following the call in the main routine.

8. The alternate register set will not be used by routines, to leave it available for processing interrupts.

9. Routines that call no other routines are classed as level 0 and all others as level 1.

DOCUMENTATION

1. The first part of the documentation, marked ';;=', contains a brief textual description of the routine.

2. The second part, marked ';;/', contains a technical description in eleven sections developed after a format used by Nicaud.

3. The third part is a complete listing of the routine, with assembler mnemonics, comments and object code.

Alan Tootill,
Enfield,
Middlesex.

A very interesting letter. I agree with Alan's sentiments and would like to help him move this idea forward. The magazine welcomes all ideas, proposals and modifications from interested readers; some we'll publish and all will be passed to Alan. Please write stat-

ing your area of interest and likely involvement, marking your envelope 'standards'. If you do not wish your letter to be printed please write 'not to be published' on it. I look forward to hearing from you - Ed.

With the help and encouragement of David Tebbutt, then Editor of *PCW*, and a quick response from several Z-80 machine code enthusiasts, *PCW Sub Set* began as a regular series in September 1980. It was presented by Alan Tootill until July 1984. Unlike other sections of *PCW*, contributions to *Sub Set* went unpaid until April 1982. From the start everyone was prepared to share their routines and ideas freely, accepting the criticism and improvement of their work by other contributors as reward enough.

THE FIRST *SUB SET*

Most of the September article was given over to a summary of the criticism received by Alan of the proposed '*Rules and Documentation*'. Then, as now, contributors to *Sub Set* didn't want to be too restricted in the standard of code sent in. This was the beginning of *Sub Set*'s renowned classification system with routines which met the rigorous standards being awarded Class 1 status - others being published as Class 2 routines.

Another innovation in the September 1980 issue concerned the Datasheet method of indicating jumps and calls to absolute addresses. Reference to addresses in the general routine area would be by means of labels in the operand field and by the dummy symbols 'XX XX' in the machine code. Reference to any other area of memory - in particular to addresses within the routine - would also use labels in the assembler listing but in this case the symbols 'YY YY' would take the place of the address in the code.

A new section was added to the documentation saying if the routine was 'Time critical'. If so then the description had also to declare either the exact number or the maximum number of Time states (system clock cycles) used by the routine. The declaration would be optional for routines not time critical.

Documentation changes

Minor revisions were made to the documentation of *Sub Set* Datasheets at irregular intervals in the succeeding months. Generally, however, the initial format suited Z-80 routines quite well and for the first few months Z-80 was the only code being sent in by readers. It was not until September 1981 that codes other than Z-80 appeared in any quantity.

Sub Set standards were perfectly adequate for the 6809 which, like the Z-80, has enough registers to deal with parameter input. But the 6502, with only three 8-bit registers, has to use specific memory locations in Page Zero as pseudo-registers for all but the simplest routine. Even parameters embedded in the program after the subroutine call are inaccessible without using Page Zero. Unfortunately, because of system use, no specific Page Zero addresses can be guaranteed available. The solution adopted in *Sub Set* was to designate an anonymous block of sixteen Page Zero bytes as honorary general purpose 'registers'. The notation for these in the assembler listing would be 'M0' to 'MF' - symbolised in the machine code by 'ZZ'.

With minor exceptions, *Sub Set* steered clear of code written for specific computer systems during its first four years. But, since the hardware in different computers is becoming increasingly diverse and incompatible, machine specific routines have been included in the series since September 1984. This shift of emphasis prompted a rearrangement of the Datasheet format to (a) highlight the specific system each routine is written for and (b) identify the routine's more general features as an aid to its possible conversion for other computers.

DATASHEET DOCUMENTATION

Sub Set Datasheets have three parts: name, description and assembler listing.

The name of the routine consists of the assembler label (marked by a preceding '=') and verbal expansion. This part should also contain the labels of any other entry points to the routine, these being preceded by '>', '/' or similar symbol.

The descriptive part has fourteen headed sections giving four types of information:

A general definition of the routine:

- **JOB.** The task performed by the routine and any special considerations such as whether the job is time critical.
- **ACTION.** The method used to perform the job.

System implementation of the routine:

- **CPU.** The processor for which the routine is written. This may include details such as the clock frequency specification.
- **HARDWARE.** Any particular computer or hardware configuration that the routine is written for and/or specifying exactly what hardware is used or affected.
- **SOFTWARE.** The name and essential details of any system software or subroutines used by the routine.

Operation details of the routine:

- **INPUT.** The meaning of any flags, registers, memory locations, stack or other source of information or data passed to the routine.

- **OUTPUT.** All information passed back from the routine and the contents or state of changed input variables.
- **ERRORS.** Any possible errors which could result from inputting invalid data, the routine being interrupted or from any other cause.
- **REGISTER USE.** All registers disturbed by use of the routine either for passing information or corrupted by the routine.
- **STACK USE.** The maximum number of bytes that could be added to the stack during execution of the routine. This includes deeper subroutine calls from the routine but does not include the two-byte return address to the program calling the routine.
- **RAM USE.** Any read/write memory (excluding stack) used for passing information between the routine and the calling program or as workspace or storage by the routine. 6502 'pseudo-registers' M0 to MF are included in this section.
- **LENGTH.** The number of bytes occupied by the assembled routine.
- **CYCLES.** This is optional for routines which are not time critical but if included should ideally be an expression giving the exact number of clock cycles that the routine executes in. If the timing is variable then either the maximum and/or a close approximate value will suffice.

Classification of the routine:

- **CLASS.** A set of six criteria indicating situations where the routine may be used safely without any change. Those met by the routine are preceded by '*', those not met by '-'. Routines are Class 1 only if all six are satisfied, thus:
 - * **Discreet:** no input variable is changed except to pass information from the routine.
 - * **Interruptable:** any interrupt will have no effect on the routine's operation, or vice versa. The Z-80 alternate register set is reserved for interrupt use.
 - * **Promable:** the routine can be fixed in read only memory. It does not alter its own code.

- * **Re-entrant:** the routine can be entered again without error to either re-entered or re-entering use from a program interrupting its execution.
- * **Relocatable:** no change is necessary for the routine to operate correctly at any location.
- * **Robust:** the routine will not 'crash' or produce unflagged erroneous results for any reason when entered at the correct point(s).

The third part of the Datasheet is the assembler listing. This is a standardised form for *Sub Set* and does not represent the assembler conventions of any particular processor. It consists of five fields:

- **Label.** Not more than six characters, the first of which must be a letter.
- **Instruction mnemonic or assembler directive** specific to the particular CPU instruction set.
- **Operand** specific to the particular CPU instruction set.
- **Comment.** Preceded by a colon. Comments may also take up a complete line to show the structure of the routine.
- **Machine code in hexadecimal.** Undefined absolute addresses may appear as 'lo hi' (Z-80 and 6502 form - low order byte first) or 'hi lo' (6809 form - high order byte first). The 6502 Page Zero pseudo-registers are signified by 'M0' to 'MF'. Immediate data which can differ between systems may appear as any doubled lower-case letter.

Delays

1

What better routine to start off with than a 6502 version of the first *Sub Set* datasheet – Alan Tootill's one-second delay routine, DL1S. Here it is showing how the revised documentation distinguishes between those aspects which are particular to the system and processor and those which are not. The operating time of each instruction is given alongside in the comments and the total for each section is shown in a full comment line before that section.

```

=====
:= DL1S      Delay for one second.
=====
:JOB         Time critical routine to use one second's worth
:            of clock cycles, including the call to DL1S.
:ACTION      Set loop count = INT(Clock Hz / n).
:            For count: [ use n cycles ].
:            Fine tune for remainder of cycles.
-----
:CPU         6502 running at 1 MHz.
:HARDWARE    None.
:SOFTWARE    None.
-----
:INPUT       None.
:OUTPUT      None.
:ERRORS      Inaccurate if interrupt occurs, or if loop is
:            located across a page boundary.
:REG USE     None.
:STACK USE   4
:RAM USE     None.
:LENGTH     26
:CYCLES      999994 (as 55553 * 18 + 40)
-----
:CLASS 2     *discreet -interruptable *promable
:*-*-*-     -reentrant *relocatable -robust
=====
:
LPCNTH =     $D9      :hi and lo bytes of 16-bit
LPCNTL =     $01      :loop counter (value 55553).
:
:...Save and initialise. Uses 21 cycles.
DL1S  PHP           : 3.  Save flags                08
      PHA           : 3.  and registers              48
      TXA           : 2.  used                      8A
      PHA           : 3.  in the routine.            48
      LDA #LPCNTH   : 2.  Get 16-bit loop counter    A9 D9
      LDX #LPCNTL   : 2.  in AX and dec it for end    A2 01
      DEX           : 2.  test on gone below zero.    CA
      CLD           : 2.  Clear for binary subtract.  D8
      SEC           : 2.  Set for first subtract.     38

```

DELAYS

```
:
:...Main delay loop. Use up 18 * 55553 - 1 cycles.
:...Carry always set for subtraction if loop occurs.
DL00P PHA      : 3.  Save count hi-byte, get      48
      TXA      : 2.  count lo-byte first        8A
      SBC  #1   : 2.  and decrement it, take     E9 01
      TAX      : 2.  any carry through hi-byte   AA
      PLA      : 4.  repeating if C = 1 as       68
      SBC  #0   : 2.  C = 0 only when count     E9 00
      BCS DL00P : 3/2. goes below 0 at end.      B0 F6
:
:...Restore registers and return. Uses 20 cycles.
      PLA      : 4.  Restore                      68
      TAX      : 2.  registers                    AA
      PLA      : 4.  and                          68
      PLP      : 4.  flags used.                  28
      RTS      : 6.  Exit after 1 second.         60
:=====
```

ASSEMBLING DLIS FOR OTHER SPEEDS

Assembling routines in clearly defined sections makes it far easier to change and update them. When Alan Tootill wrote DLIS, most Z-80 computers were running at or around 2 MHz but now the Z-80A, running at approximately 4 MHz, is being used in many computers. Similarly, more modern computers have the 6502 running at about 2 MHz instead of the original 1 MHz. In fact, it is not very likely that your computer's clock speed will be exactly 1 MHz and you will probably find it necessary to rewrite DLIS for a delay of precisely one second.

The 21 cycles used in saving and initialising registers on entry to DLIS and the 20 cycles needed to restore register values and return are clearly separated out as unchangeable timing overheads. So too are the 18 cycles in the main delay section needed to decrement the count and loop if not zero. (Only 17 cycles are used when the loop terminates since no branch takes place.) However, there are three possible variables – the loop count in A and X and two sets of dummy instructions which may be inserted in the main delay section and in the restore/return section. Thus, DLIS timing has the formula,

$(\text{clockMHz} - 6) = 41 + (\text{LPCNT} * ('main' + 18)) - 1 + 'fine'$,
where:

$0 < \text{LPCNT} \leq 65536$, 'main' => 0, 'fine' => 0

and dummy instructions are used to exhaust 'main' and 'fine' times.

Of course, if your 6502 does run at exactly 2 MHz then a one second delay is a simple matter of calling DLIS twice in succession.

A UNIVERSAL DELAY

The formula used in DL1S can be used to produce delays other than one second by calculating the correct variables and, if necessary, finding dummy instructions with the right timings. This is the method I used for my delays until one day the thought struck me that I was wasting a lot of time. I would willingly put up with a longer routine if it could be adapted more quickly to the delay time I needed.

The obvious solution was to write a routine that acted dynamically on a number equal to the clock cycles in any given fraction of a second. It would have to subtract from that number a value exactly equal to the clock cycles taken to perform the subtraction, repeating this process until reaching zero when the routine would terminate. Of course, it couldn't be that simple: every routine has certain overheads such as the time taken to save and restore register values, test for loop terminating conditions, and so on, and these have to be taken into account.

UDELAY is a routine of mine which originally appeared in *Sub Set* as UDRS. It is longer and more complex than DL1S but very adaptable. For a 2 MHz clock speed, the minimum delay is just under $\frac{1}{8163}$ second and the maximum (with FRAC = 65535 and input X = 0) just under 8.4 seconds.

The problem with UDELAY is, of course, that the system clock specification in any computer is only an ideal – the quartz crystals used do vary within a given tolerance. The real clock speed of your computer is unlikely to be divisible into any exact fraction of a second without any remainder. Nevertheless, the very slight inexactitude shouldn't make any noticeable difference except on extremely long delays.

```

=====
:= UDELAY    Universal delay.
=====
:JOB         Time critical routine to produce an accurate
:            delay of an input value times a unit delay
:            (assembled fraction of the system clock hertz)
:            including the subroutine call to UDELAY.
:ACTION      On entry: [ Fraction - UDELAY time overheads ].
:            For count: [ Fraction - loop time overheads
:                        Use up fraction cycles ].
-----
:CPU         6502
:HARDWARE    None.
:SOFTWARE    None.
=====

```

DELAYS

```

:INPUT      X = Number of unit delays wanted (00H = 256).
:OUTPUT     X = 0.
:ERRORS     Inaccurate if interrupt occurs or if FRAC is
:           assembled as any value less than 245.
:REG USE    X
:STACK USE  4
:RAM USE    None.
:LENGTH    45
:CYCLES     FRAC * X (including 6 for JSR UDELAY)
:-----
:CLASS 2    -discreet -interruptable *promable
:--*--*--   -reentrant *relocatable -robust
:=====
:
:...Assemble unit delay (FRAC) as the number of clock cycles
:...in the required fraction of a second.
FRAC  =      mnnn      :245 <= nnnn < 65536.
FRHI  =      mm        :INT(FRAC / 256).
FRLO  =      nn        :FRAC - (FRHI * 256).
:
UDTOH =      -245      :UDELAY timing overheads, subtracted
                        :from FRAC on 1st entry into NUMLP.
NLTOH =      -205      :NUMLP timing overheads, subtracted
                        :from FRAC on each repeat of NUMLP.
DLTIME =      18       :DIVLP operating time, subtracted from
                        :FRAC repeatedly until time out.
RCOUNT =      -17      :REMLP iterations - one for each
                        :possible FRAC remainder after DIVLP.
:
:...Overheads (incl. JSR): 6 + 15 + 205 - 1 + 20 = 245.
UDELAY PHP      : 3.   Save flags                      08
      PHA       : 3.   and registers                    48
      TYA       : 2.   used in                          98
      PHA       : 3.   UDELAY.                          48
      CLD       : 2.   Ensure binary arithmetic.       D8
      LDA #UDTOH : 2.   Negative timing overheads.     A9 0B
:
:...Repeat X times. Overheads: 10 + 17 + 171 + 7 = 205.
NUMLP CLC       : 2.   Subtract timing overheads       18
      ADC #FRLO  : 2.   from unit delay cycles          69 nn
      TAY       : 2.   in A and Y. Use addition         A8
      LDA #-1    : 2.   of 2's complement for          A9 FF
      ADC #FRHI  : 2.   ease of repeats.                69 mm
:
:...Use up 17 + 18 * INT((FRAC - overheads) / 18) cycles.
DIVLP PHA       : 3.   Subtract DIVLP                  48
      TYA       : 2.   cycles from reduced             98
      SBC #DLTIME : 2.   FRAC until time out            E9 12
      TAY       : 2.   when result                     A8
      PLA       : 4.   gone below zero.                 68
      SBC #0     : 2.   Remainder is in                 E9 00
      BCS DIVLP  : 2/3. negative form.                 B0 F6
:

```

```

: ...Use up 171 + REMAINDER((FRAC - overheads) / 18) cycles.
      LDA RCOUNT : 2.   Loop for maximum possible A9 EF
REMLP INY          : 2.   remainder, using 10         C8
      BNE REMCNT  : 2/3. cycles if no remainder      D0 01
      DEY          : 2.   in current loop, else       88
REMCNT ADC #1      : 2.   use 11 cycles and reset     69 01
      BNE REMLP   : 2/3. remainder count in Y.       D0 F8
:
      LDA #NLTOH   : 2.   Negative timing overheads A9 33
      DEX          : 2.   for repeat of NUMLP.       CA
      BNE NUMLP    : 2/3. Repeat unit delay * X.     D0 DF
:
      PLA          : 4.   Restore registers          68
      TAY          : 2.   and flags                  A8
      PLA          : 4.   and exit                   68
      PLP          : 4.   after                      28
      RTS          : 6.   X * shortest delay.        60
:=====

```

DEAD KEY SCROLLS

Watching the results of all your computation evaporating off the top of the screen while your brain is still decoding the first word can be very frustrating. Inserting a delay in the scroll routine will slow it down, of course, but the perfectionist is still not satisfied. Why not have a delay routine that you can switch on and off? Then you can let the boring bits whizz past and slow the scroll down only when the part you want to scrutinize is coming up. And while you are at it, you might as well include a 'wait' function to temporarily stop scrolling.

All this is achieved at the touch of a SPACE BAR in this 6502 version of SLOWUP – originally appearing in *Sub Set* in Z-80 code for the TRS-80. SLOWUP doesn't actually switch the delay on and off, instead it alternates between short and long delays so the speed of both fast and slow scrolls can be adjusted. SLOWUP performs the switch by complementing the delay count when it notices the SPACE BAR being pressed. Thus, a long delay of \$FFFE becomes the shortest possible \$0001. Fine-tuning of the delays is done by ANDing a mask with the high order byte of the delay count. For example, the hi-byte of the delay count could be ANDed with \$7F and this would halve the long delay \$FFFE to \$7FFE but have no effect on the short complement \$0001.

Scrolling is not the only activity that could benefit from SLOWUP. Try patching it in to those fast action games and give yourself a sporting chance to save the Galaxy.

DELAYS

```

=====
:= SLOWUP    Switchable delay.
=====
:JOB        Switch between adjustable long and short delays
:           on SPACE-key press with WAIT function until the
:           SPACE-key is released.
:ACTION     Fine-tune delay count by bit masking.
:           For repeat count:
:           [ For delay count: [ use time ] ].
:           IF SPACE-key pressed THEN:
:           [ Complement delay count.
:           Wait for key release. ]
-----
:CPU        6502
:HARDWARE   Keyboard.
:SOFTWARE   Written to patch in to SCROLL routine.
:           "INKEY" - subroutine to return ASCII code of key
:           pressed in A without changing other registers.
-----
:INPUT      2-byte delay count in M0,1.
:           1-byte delay repeat count in M2.
:           1-byte delay count bit-mask in M3.
:OUTPUT     M0,1 complemented if SPACE BAR pressed.
:           Registers, M2 and M3 unchanged.
:ERRORS     Double-complement can occur if re-entered.
:REG USE    None.
:STACK USE  7 + INKEY stack use in excess of 5.
:RAM USE    M0 to M3
:LENGTH     64
:CYCLES     Minimum delay if SPACE bar not pressed:
:           REPT * (delaycount * 20 + 36) + 44 + INKEY.
:           { 0 < REPT <= 256. 0 < delaycount <= 65536. }
-----
:CLASS 2    *discreet *interruptable *promable
:***-*      -reentrant *relocatable -robust
=====
:
SLOCT = M0      :Stored 2-byte delay count.
REPT = M2       :Stored coarse-tune delay loop count.
DCMASK = M3     :Stored fine-tune mask for delay count.
SPACE = $20     :ASCII SPACE.
:
SLOWUP PHP      :Save flags                      08
      PHA      :and registers                    48
      TXA      :used in SLOWUP                   8A
      PHA      :and load X with delay repeat     48
      LDX REPCNT :count for coarse timing.       A6 M2
:
...Delay.
REPL LDA SLOCT  :Save delay count                  A5 M0
      PHA      :on stack so it can be            48
      LDA SLOCT+1 :directly used as count.         A5 M1
      PHA      :Fine tune by masking out         48
      AND DCMASK :selected bits of delay          25 M3
      STA SLOCT+1 :count hi-byte before use.      85 M1
:

```

DELAYS

```

: ...Main delay loop. Min. 20, max. 24 cycles each loop.
DELLP LDA SLOCT :Test if count lo-byte      A5 M0
      BNE LODEC  :is zero, if it is then     D0 O2
      DEC SLOCT+1 :dec hi-byte as well.      C6 M1
LODEC DEC SLOCT  :Dec count lo-byte.         C6 M0
      LDA SLOCT  :Test if 16-bit             A5 M0
      ORA SLOCT+1 :delay count is at zero,   O5 M1
      BNE DELLP  :repeating until time out.  D0 F2
:
      PLA        :Restore delay              68
      STA SLOCT+1 :count to                  85 M1
      PLA        :page zero                  68
      STA SLOCT  :for next delay.            85 M0
      DEX        :Coarse tune by any repeats CA
      BNE REPLP  :of main delay.            D0 DF
:
: ...Possible switch between long/short delay.
      JSR INKEY  :Get any key press and      20 lo hi
      CMP #SPACE :test for SPACE BAR,        C9 20
      BNE SUPEND :exit if not pressed.       D0 O0
:
      LDA SLOCT+1 :SPACE pressed so switch from A5 M1
      EOR #$FF    :long to short/short to long 49 FF
      STA SLOCT+1 :by bit changes.           85 M1
:
: ...Wait.
RELLP JSR INKEY  :Loop until call to INKEY    20 lo hi
      CMP #SPACE :returns SPACE BAR not      C9 20
      BNE RELLP  :pressed.                   D0 F9
:
SUPEND PLA       :Restore registers and       68
      TAX        :flags used in SLOWUP       AA
      PLA        :and exit after             68
      PLP        :long or short              28
      RTS        :delay.                     60
:=====

```


Extra Instructions

Two types of 'extra instructions' are dealt with in this chapter. The first sort are those which the 6502 will execute if fed unspecified codes. The second are desirable operations similar to those carried out by other processors but entirely lacking from the 6502 instruction set. This latter type has to be emulated by short routines.

UNSPECIFIED INSTRUCTIONS

The 6502 uses only 151 of the 256 instructions made possible by an 8-bit opcode. The actions caused by the other 105 are 'unspecified' and no wonder!

Andrew Johnson obviously had a tremendous amount of patience when he worked out the effects of the unspecified codes. Most of them combine two distinct normal actions but several produce changes to more than one register or memory location. His findings are listed here for you to use as you see fit.

Firstly, the largest group containing instructions which can be described as a sequence of 2 specified actions (though perhaps with non-specified addressing modes):

Code	Effect (1)	then	Effect (2)
07 zz	ASL zz		ORA zz
17 zz	ASL zz,X		ORA zz,X
27 zz	ROL zz		AND zz
37 zz	ROL zz,X		AND zz,X
47 zz	LSR zz		EOR zz
57 zz	LSR zz,X		EOR zz,X
67 zz	ROR zz		ADC zz
77 zz	ROR zz,X		ADC zz,X
A7 zz	LDX zz		LDA zz
B7 zz	LDX zz,Y		LDA zz,Y
C7 zz	DEC zz		CMP zz
D7 zz	DEC zz,X		CMP zz,X
E7 zz	INC zz		SBC zz
F7 zz	INC zz,X		SBC zz,X

EXTRA INSTRUCTIONS

Code	Effect (1)	then	Effect (2)
03 zz	ASL (zz,X)		ORA (zz,X)
13 zz	ASL (zz),Y		ORA (zz),Y
23 zz	ROL (zz,X)		AND (zz,X)
33 zz	ROL (zz),Y		AND (zz),Y
43 zz	LSR (zz,X)		EOR (zz,X)
53 zz	LSR (zz),Y		EOR (zz),Y
63 zz	ROR (zz,X)		ADC (zz,X)
73 zz	ROR (zz),Y		ADC (zz),Y
A3 zz	LDX (zz,X)		LDA (zz,X)
B3 zz	LDX (zz),Y		LDA (zz),Y
C3 zz	DEC (zz,X)		CMP (zz,X)
D3 zz	DEC (zz),Y		CMP (zz),Y
E3 zz	INC (zz,X)		SBC (zz,X)
F3 zz	INC (zz),Y		SBC (zz),Y
0F lo hi	ASL \$hilo		ORA \$hilo
1F lo hi	ASL \$hilo,X		ORA \$hilo,X
2F lo hi	ROL \$hilo		AND \$hilo
3F lo hi	ROL \$hilo,X		AND \$hilo,X
4F lo hi	LSR \$hilo		EOR \$hilo
5F lo hi	LSR \$hilo,X		EOR \$hilo,X
6F lo hi	ROR \$hilo		ADC \$hilo
7F lo hi	ROR \$hilo,X		ADC \$hilo,X
AF lo hi	LDX \$hilo		LDA \$hilo
BF lo hi	LDX \$hilo,Y		LDA \$hilo,Y
CF lo hi	DEC \$hilo		CMP \$hilo
DF lo hi	DEC \$hilo,X		CMP \$hilo,X
EF lo hi	INC \$hilo		SBC \$hilo
FF lo hi	INC \$hilo,X		SBC \$hilo,X
0B nn	AND #nn		--
1B lo hi	ASL \$hilo,Y		ORA \$hilo,Y
2B nn	AND #nn		--
3B lo hi	ROL \$hilo,Y		ORA \$hilo,Y
4B nn	AND #nn		LSR A
5B lo hi	LSR \$hilo,Y		EOR \$hilo,Y
6B nn	AND #nn		ROR A
7B lo hi	ROR \$hilo,Y		ADC \$hilo,Y
8B nn	TXA		AND #nn
AB nn	LDA #nn		TAX
DB lo hi	DEC \$hilo,Y		CMP \$hilo,Y
EB nn	SBC #nn		--
FB lo hi	INC \$hilo,Y		SBC \$hilo,Y
9C lo hi	STY \$hilo,X		--
9E lo hi	STX \$hilo,Y		--

Secondly, a group which cannot easily be expressed using standard mnemonics:

87 zz	:store [A AND X] to zz
97 zz	:store [A AND X] to zz,Y
83 zz	:store [A AND X] to (zz,X)
93 zz	:store [A AND X] to (zz),Y
8F lo hi	:store [A AND X] to \$hilo
9F lo hi	:store [A AND X] to \$hilo,X

EXTRA INSTRUCTIONS

```
9B lo hi      :store [A AND X] to S and $hilo,Y
BB lo hi      :store [$hilo,Y AND S] to A, X and S
CB nn         :store [A AND X] - #nn to X
```

Thirdly, some extra NOPs (xx is don't care):

```
1-byte:  1A  3A  5A  7A  DA  FA

2-byte:  04 xx  14 xx  34 xx  44 xx  54 xx
         64 xx  74 xx  80 xx  89 xx  F4 xx

3-byte:  0C xx xx  1C xx xx  3C xx xx  5C xx xx
         7C xx xx  DC xx xx  FC xx xx
```

And finally, a group of 14 single-byte instructions which cause the 6502 to HALT and wait for RESET or possibly an interrupt:

```
02  12  32  42  52  62  72  82  92  B2  C2  D2  E2  F2
```

EXTRA ADDRESSING MODES

The 6502 pre-indexed indirect, post-indexed indirect and absolute indexed by Y addressing modes – ‘(zz,X)’, ‘(zz),Y’ and ‘\$hilo,Y’ respectively – are not supported for the six instructions ‘ASL, ROL, LSR, ROR, DEC and INC’ which act on memory. The unspecified instructions do use these three addressing modes but since they also combine two actions their use is somewhat limited. However, with a little care, they can be made to perform as required.

‘ASL+ORA’ (codes \$1B, \$13 and \$03) will set the correct C, N and Z flags for the Arithmetic Shift Left and store the result both in memory and in the Accumulator if A is first cleared. With A initially zero, the ‘ORA’ part merely loads A with the result of the ‘ASL’. Similarly, ‘ROL+AND’ will give the correct ‘ROL’ flags if A is initially set to \$FF. ‘LSR+EOR’ requires A to be zero.

‘ROR+ADC’ cannot be relied upon to give the correct ‘ROR’ result unless bit 0 of the memory location addressed is known to be reset. since bit 0 is moved out to Carry by the rotation, it affects the later addition.

‘DEC+CMP’ could possibly be useful if A is holding the non-zero terminal value for a loop count at the time the decrement occurs, though this is not likely. ‘INC+SBC’ doesn’t seem to offer a great deal of scope for advanced programming.

EXTRA INSTRUCTIONS

THE USUAL WARNING

None of the codes are guaranteed to produce the effect that Andrew Johnson found them to have, even though he did test them on three different computers. Another *Sub Set* reader, Andrew Civil, has found that his particular 6502 treats the opcode '9F lo hi' as meaning 'Store [A AND X AND (\$hi + 1)] to \$hilo,Y' – not especially useful unless you want to store the result in page \$FE.

Assemblers will, of course, throw out non-specified mnemonics and operands as invalid so you will have to enter the codes in hexadecimal form. And if you manage to get programs using them to work in your computer, they may crash when transferred to another. You use them at your own risk.

TOOLKIT

Many 6502 toolkit routines are quite short and hardly worth the bother of writing as subroutines. It takes 3 bytes to call a subroutine and one byte to return from it, so a code sequence has to be 5 bytes long before any space is saved by writing it as a subroutine.

In terms of program length, you break even on the fourth call and show a profit of 2 bytes on every subsequent call. With 7 bytes of code the break-even point comes on the second call.

But program length is not the only factor that you may have to take into account. Timing overheads are considerable on short subroutines: the 'JSR' instruction takes 6 clock cycles and the 'RTS' uses up another 6, making 12 in all – possibly more than the code execution time. You must offset this against the ease of writing and the greater program readability gained from using subroutines.

REGISTER ROTATES, TRANSFERS AND EXCHANGES

The 6502 is sadly lacking in instructions to move data between registers. It has a group of six transfer instructions which allow values to be transferred between A and the index registers and between X and the stack pointer. However, movement between X and Y is not allowed and exchanges of data cannot take place.

ROTEX is a toolkit routine of mine which appeared in *Sub Set*. It provides several 'instructions' which I think really ought to be in the 6502 instruction set but does so only at a tremendous cost in execution time.

ROTEX uses a byte saving trick that is often found where space is minimal. Entry at EXAY falls through to the RRXY but must have the Carry flag set whereas RRXY needs Carry reset. So the 2 bytes which save P and reset C at the start of RRXY have to be skipped.

EXTRA INSTRUCTIONS

This could be done by 'BCS +2' but would take 2 bytes. Instead the skip is achieved in one byte by using the dummy opcode \$2C to swallow the 2 bytes in the instruction 'BIT \$1808'. This affects only the unused N, Z and V flags – not the important Carry flag. Entry at RRAXY misses the dummy opcode, of course, and executes 'PHP, CLC'. The same method is used to save another byte when EXAX falls through to RLAXY.

```

=====
:= ROTREX   Register rotate, transfer and exchange.
:> TXY      Transfer X to Y.
:> EXAY     Exchange A with Y.
:> RRAXY    Rotate right by one byte through A, X, Y.
:> TYX      Transfer Y to X.
:> EXAX     Exchange A with X.
:> RLAXY    Rotate left by one byte through A, X, Y.
:> EXXY     Exchange X with Y.
:> PLXYAP   Pull register set X, Y, A, P and PC (return).
:> PLYXAP   Pull register set Y, X, A, P and PC (return).
=====
:JOB        Toolkit to perform Class 1 rotations, transfers
:           and exchanges acting on 8-bit registers.
:           Also provides "jump to" register set pull from
:           stack and return.
:ACTION     Push registers.
:           Manipulate on stack.
:           Pull registers in required order.
=====
:CPU        6502
:HARDWARE   None.
:SOFTWARE   None.
=====
:INPUT      None.
:OUTPUT     (Input registers are denoted by single quotes):
:           TXY: Y = 'X'; N, Z give sign and zero status.
:           TYX: X = 'Y'; N, Z give sign and zero status.
:           EXAY: A = 'Y'; Y = 'A'; P is unchanged.
:           EXAX: A = 'X'; X = 'A'; P is unchanged.
:           EXXY: X = 'Y'; Y = 'X'; P is unchanged.
:           RRAXY: A = 'Y'; X = 'A'; Y = 'X'; P unchanged.
:           RLAXY: A = 'X'; X = 'Y'; Y = 'A'; P unchanged.
:           PLXYAP: X, Y, A, P, PC pulled (X first).
:           PLYXAP: Y, X, A, P, PC pulled (Y first).
:ERRORS     None.
:REG USE    Any of A, X, Y and P (see OUTPUT).
:STACK USE  4 at each entry point (0 for PLXYAP and PLYXAP).
:RAM USE    None.
:LENGTH    56
:CYCLES     TXY: 72.      RRAXY: 65.      PLXYAP: 26.
:           TYX: 68.      RLAXY: 59.      PLYXAP: 26.
:           EXAX: 64.     EXAY: 68.      EXXY: 42.
=====
:CLASS 1    *discreet *interruptable *promable
:*****    *reentrant *relocatable *robust
=====

```

EXTRA INSTRUCTIONS

:	ROTREX =	\$0100	:To index page 1 stack.	
:				
TXY	TAY		:Move A to Y, X to A	A8
	TXA		:then exchange A with Y.	8A
:				
EXAY	PHP		:Push P then set C for	08
	SEC		:restore in XYAP order.	38
	.BYTE \$2C		:Skip 2 bytes by "BIT \$1808".	2C
:				
RRAXY	PHP		:Push P then reset C for	08
	CLC		:restore in YXAP order.	18
	PHA		:Y will be stored over this A.	48
	PHA		:A will be pulled to X.	48
	TXA		:X will be	8A
	PHA		:pulled to Y.	48
	TYA		:Move Y to A for storing.	98
	CLV		:Clear V for Branch always	B8
	BVC	REPAST	:to X/Y store over stacked A.	50 0C
:				
TYX	TAX		:Move A to X, Y to A	AA
	TYA		:then exchange A with X.	98
:				
EXAX	PHP		:Push P then reset C for	08
	CLC		:restore in YXAP order.	18
	.BYTE \$2C		:Skip 2 bytes by "BIT \$3808".	2C
:				
RLAXY	PHP		:Push P then set C for	08
	SEC		:restore in XYAP order.	38
	PHA		:X will be stored over this A.	48
	PHA		:A will be pulled to Y.	48
	TYA		:Y will be	98
	PHA		:pulled to X.	48
	TXA		:Move X to A for storing.	8A
:				
REPAST	TSX		:Index stack and move stack	BA
	INX		:index to point to first A	E8
	INX		:pushed. "ROTREX+3,X" wouldn't	E8
	INX		:work if stack wrapped around.	E8
	STA	ROTREX,X	:Overwrite 1st A with X or Y	9D 00 01
	BCC	PLYXAP	:then pull in correct order.	90 0D
:				
PLXYAP	PLA		:Exit for	68
	TAX		:TXY, EXAY and RLAXY.	AA
	PLA		:Also as "jump to"	68
	TAY		:exit for any routine	A8
	PLA		:which has pushed registers	68
	PLP		:in P, A, Y, X order.	28
	RTS		:	60
:				
EXXY	PHP		:Push registers in	08
	PHA		:P, A, Y, X order then	48
	TYA		:pull as though pushed	98
	PHA		:in P, A, X, Y order	48
	TXA		:so X and Y are exchanged	8A
	PHA		:without affecting A and P.	48

EXTRA INSTRUCTIONS

:		
PLYXAP	PLA	:Exit for EXXY, 68
	TAY	:TYX, EXAX and RRAXY. A8
	PLA	:Also as "jump to" 68
	TAX	:exit for any routine AA
	PLA	:which has pushed registers 68
	PLP	:in P, A, X, Y order. 28
	RTS	: 60
:		
:	=====	

USING ROTREX

ROTREX does provide some elementary functions on which to build other missing instructions. Here are three short subroutines to perform simple operations that unhappily weren't programmed into the 6502. Apart from the Z flag in the 16-bit left shift, all return the correct status – something not done by the 'PHA, go-through-A, PLA' method.

:			
...	Transfer Stack pointer to index register Y.		
TSY	JSR	EXXY	:Save X in Y and move Stack 20 lo hi
	TSX		:pointer to X, bump it past BA
	INX		:return address then, by jump E8
	INX		:to EXXY, transfer it to Y, E8
	JMP	EXXY	:restore X and return. 4C lo hi
:			
...	Increment accumulator.		
INCA	JSR	EXAX	:Save X in A, get A in X and 20 lo hi
	INX		:add 1 affecting only N & Z. E8
	JMP	EXAX	:Restore A & X and return. 4C lo hi
:			
...	Arithmetic Shift Left XY.		
ASLXY	JSR	RRAXY	:Rotate Y into A and shift 20 lo hi
	ASL	A	:left, 0 into bit 0. 0A
	JSR	RRAXY	:Rotate X into A and shift 20 lo hi
	ROL	A	:left, C from Y into bit 0. 2A
	JMP	RRAXY	:Restore all regs and return. 4C lo hi

QUICKER TRANSFERS, EXCHANGES AND ROTATES

Suitably abashed at the slowness of the ROTREX package I have provided TERAXY which demonstrates some quicker methods of attaining the same ends. This time though it is at the cost of corrupting one page zero location and not being re-entrant. Unlike ROTREX, the routines in TERAXY all return possibly useful flag information – and operate without the benefit of confusing tricks!

EXTRA INSTRUCTIONS

```

=====
:= TERAXY   Register transfer, exchange and rotate.
:> TXY      Transfer X to Y.
:> TYX      Transfer Y to X.
:> EXAX     Exchange A with X.
:> EXAY     Exchange A with Y.
:> RRAXY    Rotate right by one byte through A, X, Y.
:> RLAXY    Rotate left by one byte through A, X, Y.
=====
:JOB        Toolkit to perform transfers, exchanges and
:           rotations acting on 8-bit registers, returning
:           Sign and Zero status.
:ACTION     Use temporary RAM store.
:           If necessary, test register to set status.
-----
:CPU        6502
:HARDWARE   None.
:SOFTWARE   None.
-----
:INPUT      None.
:OUTPUT     M0 changed in each case.
:           (Input registers are denoted by single quotes):
:           TXY: Y = 'X'; Y sign and zero status in N, Z.
:           TYX: X = 'Y'; X sign and zero status in N, Z.
:           EXAX: A = 'X'; X = 'A'; A status in N, Z.
:           EXAY: A = 'Y'; Y = 'A'; A status in N, Z.
:           RRAXY: A = 'Y'; X = 'A'; Y = 'X'
:                A status in N, Z.
:           RLAXY: A = 'X'; X = 'Y'; Y = 'A'
:                A status in N, Z.
:ERRORS     Re-entry would overwrite register values in
:           temporary 1-byte store, M0.
:REG USE    Any of A, X, Y and P (see OUTPUT).
:STACK USE  0
:RAM USE    M0
:LENGTH     40
:CYCLES     TXY: 12.      EXAX: 14.      RRAXY: 18.
:           TYX: 12.      EXAY: 14.      RLAXY: 18.
-----
:CLASS 2    -discreet *interruptable *promable
:***-      -reentrant *relocatable -robust
=====
:
:
TXY      STX  M0      :Move X to temporary storage then      86 M0
          LDY  M0      :into Y, getting N and Z status.      A4 M0
          RTS          :                                     60
:
TYX      STY  M0      :Move Y to temporary storage then      84 M0
          LDX  M0      :into X, getting N and Z status.      A6 M0
          RTS          :                                     60
:
EXAX     STX  M0      :Move X to temporary storage.          86 M0
          TAX          :Move A to X then get X from          AA
          LDA  M0      :temp store to A giving N and Z      A5 M0
          RTS          :status of value returned in A.       60

```

EXTRA INSTRUCTIONS

```

:
EXAY  STY  MO      :Move Y to temporary storage.      84 MO
      TAY          :Move A to Y then get Y from        A8
      LDA  MO      :temp store to A giving N and Z    A5 MO
      RTS          :status of value returned in A.     60

:
RRAXY STX  MO      :Store X temporarily while        86 MO
      TAX          :moving right A to X and           AA
      TYA          :Y to A. Then complete rotation    98
      LDY  MO      :by moving stored X to Y.          A4 MO
      ORA  #0       :Get N and Z status of value now  09 00
      RTS          :in A by logical test of A.        60

:
RLAXY  STY  MO      :Store Y temporarily while        84 MO
      TAY          :moving left A to Y and            A8
      TXA          :X to A. Then complete rotation    8A
      LDX  MO      :by moving stored Y to X.          A6 MO
      ORA  #0       :Get N and Z status of value now  09 00
      RTS          :in A by logical test of A.        60
:=====

```

STACK TOP EXCHANGES

The Z-80 has a set of 3 powerful instructions to exchange any of the 16-bit registers HL, IX or IY with the two bytes on top of stack. These execute in 19 or 23 clock cycles. As a rough guide to the worth of these instructions, 2 Z-80 clock cycles are usually assumed to be the equivalent of one 6502 clock cycle.

The 6502 lacks any instruction to exchange stack top values. This is a pity because the return address to a calling program can be used as a pointer to subroutine parameters embedded in the program immediately after the 'JSR' instruction.

The only way to exchange a 6502 register with the top of stack is to write a routine and EXSX by Robert Whisson shows one way of doing the job. In fact it doesn't exchange X with the current top of stack but with the value that was on top immediately before JSR. EXSX put the 2-byte return address above it. If you decide to use the code in sequence rather than as a subroutine you must change the stack indexing in EXSX from '\$0105,X' to '\$0103,X'

```

:=====
:= EXSX      Exchange (stack pointer), index register.
:=====
:JOB         To exchange the 8-bit value held in a register
:            with that on the top of stack.
:ACTION      Push register.
:            Index entry top of stack, get value and push it.
:            Index stacked register value, get value.
:            Index entry top of stack, store register value.
:            Pull top of stack value to register.
:            Pull and discard stacked register value.
:-----

```

EXTRA INSTRUCTIONS

```

:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
:-----
:INPUT        None.
:OUTPUT       X contains the value at top of stack on entry.
:             Top of stack contains the value in X on entry.
:             P is changed (N and Z give status of A).
:ERRORS       The routine could alter one or two values at the
:             start of page two, and not X and (S), if a
:             wraparound stack is used.
:REG USE      X P
:STACK USE    3
:RAM USE      None.
:LENGTH      19
:CYCLES       46
:-----
:CLASS 2      -discreet *interruptable *promable
:-*****-    *reentrant *relocatable -robust
:=====
:
EXSX  PHA          :Save A for use on EXSX.          48
      TXA          :Stack X and move stack          8A
      PHA          :pointer to X, so X indexes      48
      TSX          :5 below pre-JSR stack top.      BA
:
      LDA $0105,X  :Get value from input stack      BD 05 01
      PHA          :top and save on new top.        48
      LDA $0101,X  :Get pushed input X and store    BD 01 01
      STA $0105,X  :to input stack top.            9D 05 01
:
      PLA          :Restore input stack top         68
      TAX          :value to X (exchange done).     AA
      PLA          :Discard pushed X value.         68
      PLA          :Restore A                       68
      RTS          :and exit.                      60
:=====

```

EXSX can easily be extended to exchange X and Y with the top 2 bytes of stack (prior to the call) by inserting the following 3 instructions at the end of the middle section, after 'STA \$0105,X'. The extended form (EXSXY) can be called by a subroutine which needs to access the return address to the original calling program.

```

TYA          :Move input Y to A and get          98
LDY $0106,X  :2nd on stack on entry to Y.      BC 06 01
STA $0106,X  :Move input Y to 2nd on stack.    9D 06 01

```

Register Indirect

The Register Indirect addressing mode is not implemented on the 6502 – basically, one suspects, because of the lack of 16-bit registers. In the Z-80 and other 8-bit processors which do boast a few 16-bit registers it is one of the most powerful addressing modes. After all, incrementing and adjusting register contents is quick and easy.

XYMOD by David Heale is a routine which attempts to make good this omission by substituting the contents of the X and Y index registers (doubled up to hold a 16-bit address) for the address operand of any 3-byte instruction. The instruction must follow straight on after JSR XYMOD.

As an example, if X contains \$AB and Y contains \$CD, then

```
JSR  XYMOD    :Call followed by instruction    20 lo hi
ROL  $FFFF    :with dummy address field.      2E FF FF
```

will become,

```
JSR  XYMOD    :Call to XYMOD inserts XY        20 lo hi
ROL  $ABCD    :into ROL address field.         2E CD AB
```

during the execution of XYMOD without either X or Y being affected. If that piece of code is written inside a loop which alters the value of X and Y then a different memory location will be rotated in each iteration.

```
=====
:= XYMOD      Modify absolute address operand.
=====
:JOB          To replace the 2-byte address operand of an
:             instruction immediately following the call to
:             XYMOD with an address input in registers.
:ACTION       Use return address as pointer to instruction.
:             Index instruction second byte.
:             Move input low order byte to instruction.
:             Index instruction third byte.
:             Move input high order byte to instruction.
=====
```

REGISTER INDIRECT

```

:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
:-----
:INPUT        XY contains 16-bit address (high order in X).
:             A 3-byte instruction should follow immediately
:             after "JSR XYMOD" in the calling program.
:OUTPUT       The contents of XY are stored in program bytes 2
:             and 3 after "JSR XYMOD".
:             M1 to M4 and P are changed.
:ERRORS       No check is made on the calling program to test
:             that it is in read/write memory.
:REG USE      X Y P
:STACK USE    None.
:RAM USE      M1 to M4
:LENGTH      29
:CYCLES       61
:-----
:CLASS 2      -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
:=====
:
XYMOD STA M1      :Save registers A and Y in page      85 M1
      STY M4      :zero for use in XYMOD.              84 M4
:
      PLA         :Move stacked Program Counter        68
      STA M2      :(i.e. return address - 1)           85 M2
      PLA         :to page zero as pointer to          68
      STA M3      :program bytes to be accessed,       85 M3
      PHA         :and return it to stack              48
      LDA M2      :for RTS at routine end.             A5 M2
      PHA         :                                     48
:
      LDY #2       :Index lo-byte of instruction.      A0 02
      LDA M4       :Get input lo-byte (saved Y)        A5 M4
      STA (M2),Y   :and store in program address.      91 M2
      INY         :Index hi-byte of instruction.       C8
      TXA         :Get input hi-byte from X to A        8A
      STA (M2),Y   :and store in program address.      91 M2
:
      LDY M4       :Restore saved Y and A               A4 M4
      LDA M1       :from page zero and return to       A5 M1
      RTS         :execute modified instruction.       60
:=====

```

THE ZERO OPTION

XYMOD's appearance in *Sub Set* prompted an immediate response from readers aghast at the idea of a routine modifying the very code that called it.

Improved, 'respectable' versions, by John Kerr (RINXY) and Conor O'Neill (XYMODS) don't even hint at altering the calling program. Instead they both access the instruction opcode, using it and the address from XY to write a one-off subroutine. Each routine exits through its own creation.

REGISTER INDIRECT

The location chosen by both John and Conor for the single instruction subroutine is within the set of 16 page zero 'pseudo-registers', M0 to MF. These are assumed to be dealt with by the system as though they were registers – this being the simplest way of standardising page zero use to the sort of general purpose routines in *Sub Set*.

The method of RINXY obviates the need for the address operand in the instruction following 'JSR' and so the call is implemented in only 4 bytes (the opcode here is for 'ROL'):

```
JSR RINXY    :Call followed by opcode          20 lo hi
.BYTE $2E    :of instruction RINXY forms.      2E
```

XYMODS retains the dummy address operand:

```
JSR XYMODS   :Call followed by essential      20 lo hi
ROL $FFFF    :opcode and helpful operand.     2E FF FF
```

The operand is needed as little by XYMODS as by RINXY but Conor was aware that to leave it in makes program readability and assembly/disassembly much easier.

The most important difference to be found between the 2 routines is in the way that the constructed subroutines, RISUB and XYSUB, terminate. XYMODS pulls the return address from stack and uses it to form a 3-byte 'JMP' back to the calling program but RINXY, corrupting fewer page zero locations, ends the subroutine with a 1-byte 'RTS'. However, if the opcode is \$4C or \$6C – 'JMP \$hilo' or 'JMP (\$hilo)' – then the end of the page zero subroutine is never executed. This is unimportant in XYSUB but in RISUB the skipped 'RTS' will cause a stacking error.

```
=====
:= RINXY      Register Indirect addressing mode emulator.
=====
:JOB          To provide a Register Indirect addressing mode
:             by building up a single instruction subroutine
:             from an opcode following the call to RINXY and
:             a register input address.
:ACTION       Increment return address past opcode byte.
:             Use return address as pointer to opcode byte.
:             Copy opcode byte to byte 1 of single-instruction
:             subroutine.
:             Store 16-bit register contents to bytes 2 and 3
:             as subroutine address operand.
:             Store RETURN opcode to last byte of subroutine.
:             Jump to subroutine.
=====
```

REGISTER INDIRECT

```

:CPU          6502
:HARDWARE     None.
:SOFTWARE     "RISUB" - a 4-byte subroutine constructed in
:              page zero to effect the opcode action on the
:              address input to RINXY in XY.
:-----
:INPUT        XY contains 16-bit address (high order in X).
:              The 1st byte opcode of a 3-byte instruction must
:              follow "JSR RINXY" in the calling program.
:OUTPUT       M1 to M4 contains a subroutine built from the
:              opcode followed by the contents of XY, followed
:              by $60 (RTS). Control is passed to M1.
:ERRORS       A stacking error will occur if the opcode is $4C
:              or $6C - JMP $hilo or JMP ($hilo).
:REG USE      X Y
:STACK USE    2
:RAM USE      M1 to M4
:LENGTH      41
:CYCLES       Minimum 71.
:-----
:CLASS 2      -discreet *interruptable *promable
:-****-      *reentrant *relocatable -robust
:=====
:
RISUB = M1      :Start of 4-byte subroutine location.
OPRTS = $60     :Opcode for RTS instruction.
:
RINXY PHP       :Save flags 08
      PHA       :and accumulator. 48
      LDA #OPRTS :Store RTS at end of page A9 60
      STA RISUB+3 :zero subroutine, after 85 M4
      STX RISUB+2 :high order byte of address. 86 M3
:
PAGE TSX        :Index stack, bump stacked BA
      INX       :Program Counter (return E8
      INC $0102,X :address - 1) past 1-byte FE 02 01
      BEQ PAGE   :opcode following JSR RINXY. F0 FA
:
      TSX        :Index stack and BA
      LDA $0103,X :copy new return address BD 03 01
      STA RISUB   :lo-byte to page zero. 85 M1
      LDA $0104,X :then return address hi-byte BD 04 01
      STA RISUB+1 :to follow it. 85 M2
:
      LDX #0      :Index program and copy the A2 00
      LDA (RISUB,X) :opcode to start of page zero A1 M1
      STA RISUB   :subroutine then insert low 85 M1
      STY RISUB+1 :order byte of input address. 84 M2
:
      LDX RISUB+2 :Restore X from subroutine. A6 M3
      PLA        :Restore A and P from stack 68
      PLP        :and exit by jump to single 28
      JMP RISUB   :instruction subroutine. 4C M1 00
:=====

```

REGISTER INDIRECT

```

=====
:= XYMODS    Copy instruction with modified address operand.
=====
:JOB         To provide a Register Indirect addressing mode
:            by modifying the address operand of a single
:            instruction following the call to XYMODS, copied
:            to a subroutine, with a register input address.
:ACTION      Move return address from stack to subroutine
:            bytes 5 and 6.
:            Use return address as pointer to opcode byte.
:            Copy opcode byte to byte 1 of single-instruction
:            subroutine.
:            Store 16-bit register contents to bytes 2 and 3
:            as subroutine address operand.
:            Increment return address to act as jump address
:            back to program after the instruction.
:            Store JUMP opcode to byte 4 of subroutine.
:            Jump to subroutine.
-----
:CPU         6502
:HARDWARE    None.
:SOFTWARE    "XYSUB" - a 4-byte subroutine constructed in
:            page zero to effect the opcode action on the
:            address input to XYMODS in XY.
-----
:INPUT       XY contains 16-bit address (high order in X).
:            The 3-byte instruction (with dummy address
:            operand) must immediately follow "JSR XYMODS" in
:            the calling program.
:OUTPUT      M1 to M6 contains a subroutine built from the
:            opcode (instruction 1st byte) followed by the
:            contents of XY as address operand and terminated
:            by a JMP built from the return address. Control
:            is passed to M1.
:            M0 and P are changed.
:ERRORS      None.
:REG USE     X Y P
:STACK USE   None.
:RAM USE     M0 to M6
:LENGTH     40
:CYCLES      Minimum 60.
-----
:CLASS 2     -discreet *interruptable *promable
:*****      *reentrant *relocatable *robust
=====
:
XYSUB =      M1           :Start address of 6-byte subroutine.
OPJMP =      $4C          :Opcode for JMP instruction.
:
XYMODS STA   M0           :Save A not blocking stack. 85 M0
      PLA           :pull return address 68
      STA   XYSUB+4      :and store in page zero 85 M5
      PLA           :subroutine in bytes 68
      STA   XYSUB+5      :5 and 6. 85 M6
      STY   XYSUB+1      :Store address operand in 84 M2
      STX   XYSUB+2      :bytes 2 and 3. 86 M3
:

```

REGISTER INDIRECT

```

LDY #1           :Index program and copy      A0 01
LDA (XYSUB+4),Y  :opcode to byte 1 of         B1 M5
STA XYSUB        :page zero subroutine.      85 M1
:
LDA XYSUB+4      :Increment return address    A5 M5
CLC              :past 3-byte instruction     18
ADC #4           :adding 1 to convert RTS     69 04
BCC JAOK         :address to correct JMP     90 02
INC XYSUB+5      :address for return.        E6 M6
JAOK STA XYSUB+4  :Store JMP opcode before    85 M5
LDA #0PJMP       :JMP address so subroutine  A9 4C
STA XYSUB+3      :ends with a jump.         85 M4
:
LDA M0           :Restore A and Y from       A5 M0
LDY XYSUB+1      :page zero and exit by     A4 M2
JMP XYSUB        :jump to subroutine.       4C M1 00
:=====

```

INTELLIGENT REGISTER INDIRECT

INDXY by Cormac Duffin acts like RINXY to build a 4-byte subroutine in page zero but is intelligent enough to test for the 'JMP' opcodes, \$4C and \$6C. If the instruction is a jump then INDXY removes the return address from stack to prevent the stacking error that would occur.

```

:=====
:= INDXY      Register Indirect addressing mode emulator.
:=====
:JOB          To provide a Register Indirect addressing mode
:             by building up a single instruction subroutine
:             from an opcode following the call to INDXY and
:             a register input address.
:ACTION       Increment return address past opcode byte.
:             Store 16-bit register contents to bytes 2 and 3
:             as subroutine address operand.
:             Store RETURN opcode to last byte of subroutine.
:             Use return address as pointer to opcode byte.
:             Copy opcode byte to byte 1 of subroutine.
:             IF opcode is a JUMP THEN:
:             [ Remove return address from stack. ]
:             Jump to subroutine.
:-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     "INSUB" - a 4-byte subroutine constructed in
:             page zero to effect the opcode action on the
:             address input to INDXY in XY.
:-----
:INPUT        XY contains 16-bit address (high order in X).
:             The 1st byte opcode of a 3-byte instruction must
:             follow "JSR INDXY" in the calling program.
:OUTPUT       M1 to M4 contains a subroutine built from the
:             opcode followed by the contents of XY, followed
:             by $60 (RTS). Control is passed to M1.
:             M5 to M8 and Y are changed.

```

REGISTER INDIRECT

```
:ERRORS      None.
:REG USE      X Y
:STACK USE    1
:RAM USE      M1 to M8
:LENGTH      52
:CYCLES       Minimum 94.
```

```
:-----
:CLASS 2      -discreet *interruptable *promable
:-*****      *reentrant *relocatable *robust
:=====
```

```
:
INSUB  =      M1           :Start address of 4-byte subroutine.
TEMPZ  =      M5           :1st of 4-byte temporary storage.
OPRTS  =      $60          :Opcode for RTS instruction.
OPJMP  =      $4C          :Opcode for JMP instruction.
JMPMSK =      $DF          :Bitmask to reduce $6C to $4C.
```

```
:
INDXY STA TEMPZ           :Save A and P in                85 M5
    PHP                   :page zero temporary            08
    PLA                   :store so return address        68
    STA TEMPZ+1           :on stack is not blocked.      85 M6
```

```
:
    SEC                   :Prepare to add.                38
    PLA                   :Pull return address off        68
    ADC #0                 :stack and store it in         69 00
    STA TEMPZ+2           :page zero, after              85 M7
    PLA                   :incrementing it past          68
    ADC #0                 :opcode byte, as pointer       69 00
    STA TEMPZ+3           :to opcode byte. Then put      85 M8
    PHA                   :adjusted return address        48
    LDA TEMPZ+2           :back to stack for exit        A5 M7
    PHA                   :from INSUB if not JMP.         48
```

```
:
    STX INSUB+2           :Store address operand        86 M3
    STY INSUB+1           :to bytes 2 & 3 of INSUB        84 M2
    LDA #OPRTS            :and terminate it with         A9 60
    STA INSUB+3           :code $60 for RTS exit.        85 M4
```

```
:
    LDY #0                :Index program and copy        A0 00
    LDA (TEMPZ+2),Y       :opcode to byte 1 of        B1 M7
    STA INSUB             :page zero subroutine.        85 M1
```

```
:
    AND #JMPMSK           :Test opcode for absolute    29 DF
    CMP #OPJMP            :or indirect jump, skip        C9 4C
    BNE RESTOR            :if it isn't, else clear       D0 02
    PLA                   :return address off stack       68
    PLA                   :to prevent stack error.        68
```

```
:
RESTOR LDA TEMPZ+1        :Restore P                A5 M6
    PHA                   :via stack,                    48
    LDA TEMPZ             :restoring A first so         A5 M5
    PLP                   :flags not changed and         28
    JMP INSUB             :exit to subroutine.           4C M1 00
```

REGISTER INDIRECT

FINAL THOUGHTS

INDXY is 52 bytes and 94 cycles, RINXY is 41 bytes and 71 cycles and XYMODS is 40 bytes and 60 cycles. David Heale's XYMOD stands its ground at only 29 bytes and 61 cycles.

The concepts behind these routines are certainly thought provoking – subroutines that rewrite the program calling them or build new subroutines from a dissection of the program – but is the subject worthy?

Leaving out 'CPX, CPY, LDX, LDY, STX and STY' which are very improbable uses of an (XY) operand, there are only 17 instructions which may benefit from this extra addressing mode. With a little thought, 8 of them can be programmed in only 4 bytes each – the number of bytes taken by 'JSR' plus the single-byte opcode – using only 2 page zero locations.

The method is simple: initialise and keep M0 to a value of 0 then use the 2-instruction sequence 'STX M1; opc (M0),Y'. This will produce the effect of 'opc (XY)' for 'ADC, AND, CMP, EOR, LDA, ORA, SBC and STA' with a maximum execution time of 9 clock cycles.

The remaining instructions – 'BIT, ASL, ROL, LSR, ROR, DEC, INC, JSR and JMP' – do not support post-indexed indirect addressing. It may possibly be worthwhile to implement this register indirect form for the shift instructions alone but how often do you use X and Y to hold a 16-bit address? If you need to store the address in page zero in order to use it then it might not be a bad idea to have it there from the start, leaving X and Y free for their normal function as loop counters and index registers.

Parameter Access, Save and Restore

The design of the 6502 can present something of a problem when you attempt to write general purpose library routines to *Sub Set Class 1* standards. Class 1 routines must be 'discreet' – they should not change any parameters except to pass useful information back from the routine – and so the contents of registers and memory locations used within a subroutine have to be preserved in some way. Since the 6502 allows only P and A to be pushed on to the page 1 stack, any other values have first to be transferred or loaded into A before they can be pushed.

If, for example, you need to save all registers and 2 of the page zero pseudo-registers then the byte overhead on your subroutine is a massive 25 and your source program 21 lines long even before you include the actual processing,

SAVE	PHP	:Save flags,	08
	PHA	:Accumulator,	48
	TXA	:Index X via A,	8A
	PHA	:	48
	TYA	:Index Y via A,	98
	PHA	:	48
	LDA M0	:pseudo-register 0 via A	A5 M0
	PHA	:	48
	LDA M1	:and pseudo-register 1 via A.	A5 M1
	PHA	:	48
PROCES	.	:Do	.
	.	:actual	.
	.	:processing.	.
RESTOR	PLA	:Restore pseudo-register 1 via A,	68
	STA M1	:	85 M1
	PLA	:pseudo-register 0 via A,	68
	STA M0	:	85 M0
	PLA	:Index Y via A,	68
	TAY	:	A8
	PLA	:Index X via A,	68
	TAX	:	AA
	PLA	:Accumulator,	68
	PLP	:and flags.	28
	RTS	:Exit subroutine.	60

Each additional page zero location pushed adds 6 bytes to the object code and 4 lines to the source program.

PARAMETER ACCESS, SAVE AND RESTORE

There are other problems. The Accumulator is changed by passing the different values through it. If you have used A to send a value to the subroutine, that value is embedded deep in the stack and can only be obtained by using X to index the stack, adding another 4 bytes to the routine and changing X:

```
GETA   TSX           :Index stack and recover A      BA
       LDA $0105,X   :from above M1, M0, Y and X.    BD 05 01
```

Passing values to a subroutine by embedding them in the program immediately after the 'JSR' instruction is also awkward. Again X has to index the stack but in this case the return address, above P, has to be moved into page zero before it can be used as the base address of the parameter list.

```
GETPC  LDA  M2       :Save pseudo-register 2        A5 M2
       PHA           :on stack via A,                48
       LDA  M3       :then pseudo-register 3        A5 M3
       PHA           :also via A.                   48
       TSX           :Index stack, get ret. addr.    BA
       LDA $0109,X   :lo-byte from above M3, M2,    BD 09 01
       STA  M2       :M1, M0, Y, X, A and P into    85 M2
       LDA $010A,X   :M2 then the high order       BD 0A 01
       STA  M3       :byte into M3, both via A.     85 M3
```

After the parameters have been accessed – usually indexed by Y with post-indexed indirect addressing – the number of parameter bytes has to be added to the return address in M2 and M3, the new address moved back to stack and M2 and M3 restored.

In a large, structured program, subroutines will be nested to many levels. If a large number of those subroutines need to save registers and values from the pseudo-register block, as well as access embedded parameters, the fixed page stack of the 6502 at only 256 bytes is soon going to be used up.

The routines in this chapter attempt to solve these problems associated with conserving values and accessing the parameters passed to a routine. Probably they will not meet your exact needs but what they can do is reduce the opening and closing sequences from many bytes to just one or two 3-byte subroutine calls. One thing that must be said, however, is that the processes are comparatively slow; if you need fast code then the subroutines are unlikely to be of any help.

PARAMETER ACCESS, SAVE AND RESTORE

ACCESS

BOX and COX are a couple of my routines designed to provide complete and easy access to all register and flag values and addresses of the routine and its parameters. In the original *Sub Set* version they were combined as a single routine with 2 entry points. They operate slightly quicker when separated.

BOX first pushes P, A, X and Y to stack. If it has been called as the first instruction of a routine then the top 8 stack bytes now contain the registers followed by the return address to the routine (i.e. the address of the third byte of that routine) and the return address to the program which called the routine where parameters may be embedded.

The second action of BOX is to exchange the eight page zero pseudo-registers, M8 to MF, with the top 8 bytes of stack. The page zero values are thus saved and the stacked values available for pre-indexed and post-indexed indirect addressing.

COX is the inverse process and should be called at the end of a routine which opens with JSR BOX. It restores the eight page zero values, putting the registers and addresses back on stack – after changing the values in MC and MD (presumed to be the address of the third byte of JSR BOX) to the return address (third byte of JSR COX). Then all registers are restored and COX returns to the instruction after JSR COX with the return address from the routine on top of stack ready for 'RTS'.

```
=====
:= BOX      Stack registers, block exchange memory/stack.
=====
:JOB        To push registers to stack and then exchange the
:           block of stack containing them with a block of
:           readily accessible memory.
:ACTION     Push all registers.
:           Index stack and memory blocks.
:           FOR block length:
:           [ Exchange stack with memory. ]
:           Copy return address to top of stack.
:           Return.
-----
:CPU        6502
:HARDWARE   None.
:SOFTWARE   None.
-----
:INPUT      None..
:OUTPUT     M8 to MF on stack.
:           Input Y in M8. Input X in M9.
:           Input A in MA. Input P in MB.
:           Return address (3rd byte JSR BOX) in MC,D.
:           Top 2-bytes of stack on entry in ME,F.
:           X = 0. Y indexes MF on stack. A and P changed.
```

PARAMETER ACCESS, SAVE AND RESTORE

```

:ERRORS      None.
:REG USE     P A X Y
:STACK USE   6
:RAM USE     M8 to MF
:LENGTH     34
:CYCLES      289
:-----
:CLASS 2     -discreet *interruptable *promable
:-*****     *reentrant *relocatable *robust
:=====
:
STACK = $0100 :6502 page 1 stack base.
PZERO = zz    :Assemble as address above page zero
              :pseudo-register MF as block index.
:
BOX  PHP      :Push registers                08
     PHA      :P, A, X and Y                48
     TXA      :onto stack                    8A
     PHA      :in preparation                48
     TYA      :for exchange                  98
     PHA      :with page zero block.         48
:
     TSX      :Index stack by Y at one       BA
     TXA      :location lower in memory      8A
     TAY      :than stacked Y.              A8
     LDX #-8   :Index page zero from M8.     A2 F8
:
BOXL INY      :Index next stack byte.        C8
     LDA STACK,Y :Get next byte from stack,  B9 00 01
     PHA      :save it and move next        48
     LDA PZERO,X :byte from page zero block  B5 zz
     STA STACK,Y :to corresponding stack.    99 00 01
     PLA      :Recover stack byte, move to   68
     STA PZERO,X :corresponding pz byte.     95 zz
     INX      :Index next pz block byte     E8
     BNE BOXL   :and repeat for M8 to MF.    D0 F0
:
     LDA MD     :Copy exchanged return      A5 MD
     PHA      :address from pz back         48
     LDA MC     :to top of stack            A5 MC
     PHA      :to enable RTS exit          48
     RTS      :from BOX.                   60
:=====

:=====
:= COX      Block exchange memory/stack, unstack registers.
:=====
:JOB        To exchange a block of readily accessible memory
:            with a block of stack and then pull registers
:            from that block.
:ACTION     Move return address to correct memory location.
:            Index stack and memory blocks.
:            FOR block length:
:            [ Exchange stack with memory. ]
:            Pull all registers.
:            Return.

```

PARAMETER ACCESS, SAVE AND RESTORE

```

-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
-----
:INPUT        None.
:OUTPUT       Input top 8 bytes of stack in M8 to MF.
:             Input M8 in Y. Input M9 in X.
:             Input MA in A. Input MB in P.
:             Input contents of MC,D are lost.
:             Input ME,F on top 2-bytes of stack.
:ERRORS       None.
:REG USE      P A X Y
:STACK USE    None.
:RAM USE      M8 to MF
:LENGTH      34
:CYCLES       295
-----
:CLASS 2      -discreet *interruptable *promable
:-*****      *reentrant *relocatable *robust
:=====
:
STACK = $0100      :6502 page 1 stack base.
PZERO = zz        :Assemble as address above page zero
                  :pseudo-register MF as block index.
:
COX   PLA         :Move return address to          68
      STA MC      :page zero where exchange      85 MC
      PLA         :will put it on stack top      68
      STA MD      :after all registers pulled.   85 MD
:
      TSX         :Index stack by Y at one        BA
      TXA         :location lower in memory      8A
      TAY         :than start of block.          A8
      LDX #-8     :Index page zero from M8.      A2 F8
:
COXL  INY         :Index next stack byte.         C8
      LDA PZERO,X :Get next pz block byte,      B5 zz
      PHA         :save it and move next        48
      LDA STACK,Y :byte from stack to          B9 00 01
      STA PZERO,X :corresponding pz block.      95 zz
      PLA         :Recover pz byte, move to      68
      STA STACK,Y :corresponding stack byte.    99 00 01
      INX         :Index next pz block byte     E8
      BNE COXL    :and repeat for M8 to MF.     D0 F0
:
      PLA         :Restore previous             68
      TAY         :contents of                  A8
      PLA         :M8 to MB                     68
      TAX         :to registers                  AA
      PLA         :Y, X, A and P, and           68
      PLP         :exit to correct              28
      RTS        :return address.              60
=====

```

PARAMETER ACCESS, SAVE AND RESTORE

USER STACK

PSHU and PULU by Martin Ford provide you with the advanced stacking facilities of the 6809. With a single call you can save or restore any combination of the PC, P, A, X, Y and 16-bit pseudo-registers, M0,1, M2,3 and M4,5 to a stack anywhere in memory. They appeared in *Sub Set* as PSH16 and PLL16.

The parameter byte which determines the registers to be pushed or pulled is embedded after the 'JSR' instruction. Each bit stands for one of the eight registers and if set that register is pushed (PSHU) or pulled (PULU). If the bit is reset no push or pull of the corresponding register takes place.

The order in which the registers are dealt with cannot be changed. The Program Counter is always pushed first, followed by P, A, X, Y, (M5,M4), (M3,M2) and finally, (M1,M0). They are pulled in reverse order.

The User Stack Pointer occupies 2 bytes of page zero, preferably not within the 16-byte pseudo-register block. Any of the values on the User Stack can be easily accessed by post-indexed indirect addressing: set Y equal to the offset of the required byte and execute the instruction 'LDA (USPL),Y'. The top of Ustack has a zero offset since the User Stack Pointer addresses the last byte pushed (as in the 6809) and not the next available location as the 6502 hardware stack pointer does.

```
=====
:= PSHU      Push specified registers to User Stack.
=====
:JOB         To store all the register and pseudo-register
:            memory values, specified by a byte following the
:            call to PSHU, to an area of memory dedicated as
:            "User Stack", indexed by a "User Stack Pointer".
:ACTION      Save registers and pseudo-registers to system
:            stack in the correct order.
:            Bump return address past push parameter byte.
:            Use return address to get parameter byte.
:            Set register-size code byte.
:            REPEAT 8 times:
:            [ IF current parameter byte bit set THEN:
:              [ Copy byte from stack to User stack.
:                Index next free User stack location.
:              IF current register-size code bit set THEN:
:                [ Copy byte from stack to User stack.
:                  Index next free User stack location. ] ]
:            Index next stack byte.
:            IF current register-size code bit set THEN:
:              [ Index next stack byte. ] ]
:            Restore pseudo-registers and registers from
:            system stack in correct order.
=====
```

PARAMETER ACCESS, SAVE AND RESTORE

```
:CPU          6502
:HARDWARE     RAM dedicated as "User Stack", not limited by
:             any page boundary restrictions.
:SOFTWARE     None.
:-----
:INPUT        "USP" addresses current top-of-Ustack, one byte
:             above next free Ustack location.
:             Set bits in the parameter byte immediately after
:             "JSR PSHU" specify which registers and page zero
:             words are to be pushed:
:             Bit 0:  M0,1      Bit 4:  X
:             Bit 1:  M2,3      Bit 5:  A
:             Bit 2:  M4,5      Bit 6:  P
:             Bit 3:  Y         Bit 7:  PC
:OUTPUT        "USP" addresses last byte pushed to Ustack.
:             PSHU returns to byte after parameter byte.
:ERRORS       None.
:REG USE      None.
:STACK USE    10
:RAM USE      "USP" - 2-byte "User Stack Pointer" in page zero
:             (not within the 16-byte pseudo-register block).
:LENGTH      107
:CYCLES       Not given.
:-----
```

```
:CLASS 1      *discreet *interruptable *promable
:*****      *reentrant *relocatable *robust
:=====
```

```
:
USPL = qq      :Address of stored User Stack Pointer
USPH = pp      :high and low bytes in page zero.
ZIB = zz      :Assemble as M0-1 to act as index base
:             :for addressing pseudo-register block.
```

```
:
PSHU PHP       :Push registers          08
PHA           :P, A, X and Y           48
TXA           :onto 6502 page 1        8A
PHA           :system stack for easy   48
TYA           :access and free for     98
PHA           :use in PSHU.            48
TSX           :Index Stack below Y.    BA
```

```
:
LDY #6        :Index count 6 pz bytes.  A0 06
PSH1 LDA ZIB,Y :Push all page zero      B9 zz 00
PHA           :pseudo-registers M0 to M5 48
DEY           :with M0 on stack top.     88
BNE PSH1      :Leave Y = 0.             D0 F9
```

```
:
INC $0105,X   :Increment low order byte  FE 05 01
LDA $0105,X   :of return address (and   BD 05 01
STA M0        :also high order byte if  85 M0
BNE PSH2      :necessary) and copy     D0 03
INC $0106,X   :it to page zero so that  FE 06 01
PSH2 LDA $0106,X :it can be used to index BD 06 01
STA M1        :and get parameter byte.  85 M1
```

PARAMETER ACCESS, SAVE AND RESTORE

```

:
LDA #$87      :Register-size code, 1 byte      A9 87
STA M2        :(reset) or 2 bytes (set).       85 M2
LDA (M0),Y    :Copy parameter byte to         B1 M0
STA M3        :page zero for testing.         85 M3
LDA #8        :Set count for 8 bits           A9 08
STA M4        :in parameter byte.            85 M4
DEY          :Set Y = 255 and (USPL),Y to      88
DEC USPH      :index next free Ustack byte. C6 pp

:
:....Push loop, 8 iterations, 1 or 2 bytes pushed each loop.
PSH3 ROL M3    :Shift next parameter bit to    26 M3
      BCC PSH4  :carry, skip if no push.       90 10

:
      LDA $0106,X :Copy one byte from stack    BD 06 01
      STA (USPL),Y :to Ustack and index next   91 qq
      DEY         :byte on Ustack.            88
      BIT M2      :Test push size of current  24 M2
      BPL PSH4    :"register", skip if 1 byte. 10 06
      LDA $0105,X :Else copy next byte from    BD 05 01
      STA (USPL),Y :stack to Ustack and index  91 qq
      DEY         :next byte on Ustack.       88

:
PSH4 DEX       :Move stack index down one      CA
      ROL M2     :"register", test reg-size    26 M2
      BCC PSH5   :and move down by 2 bytes    90 01
      DEX        :if PC, M0,1, M2,3 or M4,5.   CA

:
PSH5 DEC M4    :Repeat for all 8 bits in        C6 M4
      BNE PSH3   :parameter byte.            D0 E2

:
      SEC       :Compute new top-of-Ustack     38
      TYA       :address,                    98
      ADC USPL   :USPH,L = USPH,L + Y + 1,     65 qq
      STA USPL   :(since Y + USPH,L indexes    85 qq
      BCC PSH6   :next free byte) leaving      90 02
      INC USPH   :USP addressing last push.    E6 pp

:
PSH6 LDY #0    :Index block of 6 pz bytes.     A0 00
PSH7 INY       :Index next byte and           C8
      PLA       :restore from 6502 page 1      68
      STA ZIB,Y  :stack to correct page 0     99 zz 00
      CPY #6     :location, repeating for      C0 06
      BNE PSH7   :M0 to M6.                  D0 F7

:
      PLA       :Restore                      68
      TAY       :all                          A8
      PLA       :registers                    68
      TAX       :and flags from               AA
      PLA       :6502 page 1 stack and        68
      PLP       :return to program location   28
      RTS      :after parameter byte.         60
:=====

```

PARAMETER ACCESS, SAVE AND RESTORE

```

=====
:= PULU      Pull specified registers from User Stack.
=====
:JOB         To restore all the register and pseudo-register
:            memory values, specified by a byte following the
:            call to PULU, from a memory area dedicated as
:            "User Stack", indexed by a "User Stack Pointer".
:ACTION      Save registers and pseudo-registers to system
:            stack in the correct order.
:            Bump return address past pull parameter byte.
:            Use return address to get parameter byte.
:            Set register-size code byte.
:            REPEAT 8 times:
:            [ IF current parameter byte bit set THEN:
:              [ Copy byte from User stack to stack.
:                Index next User stack location.
:                IF current register-size code bit set THEN:
:                  [ Copy byte from User stack to stack.
:                    Index next User stack location. ] ]
:              Index next stack byte.
:              IF current register-size code bit set THEN:
:                [ Index next stack byte. ] ]
:            Restore pseudo-registers and registers from
:            system stack in correct order.
-----
:CPU         6502
:HARDWARE    RAM dedicated as "User Stack", not limited by
:            any page boundary restrictions.
:SOFTWARE    None.
-----
:INPUT       "USP" addresses current top-of-Ustack, the next
:            byte to be pulled.
:            Set bits in the parameter byte immediately after
:            "JSR PULU" specify which registers and page zero
:            words are to be pulled:
:            Bit 0:  M0,1      Bit 4:  X
:            Bit 1:  M2,3      Bit 5:  A
:            Bit 2:  M4,5      Bit 6:  P
:            Bit 3:  Y         Bit 7:  PC
:OUTPUT      "USP" addresses current top-of-Ustack, one byte
:            higher in memory than last byte pulled.
:            PULU returns to byte after parameter byte if the
:            Program Counter has not been pulled, else to the
:            byte following the address pulled to the PC.
:ERRORS      None.
:REG USE     All registers that are restored from Ustack.
:STACK USE   10
:RAM USE     "USP" - 2-byte "User Stack Pointer" in page zero
:            (not within the 16-byte pseudo-register block).
:            Any of M0,1, M2,3 and M4,5 restored from Ustack.
:LENGTH     104
:CYCLES      Not given.
-----
:CLASS 1     *discreet *interruptable *promable
:*****      *reentrant *relocatable *robust
=====

```

PARAMETER ACCESS, SAVE AND RESTORE

:			
USPL	=	qq	:Address of stored User Stack Pointer
USPH	=	pp	:high and low bytes in page zero.
ZIB	=	zz	:Assemble as MO-1 to act as index base :for addressing pseudo-register block.
:			
PULU	PHP		:Push registers 08
	PHA		:P, A, X and Y 48
	TXA		:onto 6502 page 1 8A
	PHA		:system stack for easy 48
	TYA		:access and free for 98
	PHA		:use in PULU. 48
:			
	LDY	#6	:Index count 6 pz bytes. A0 06
PUL1	LDA	ZIB,Y	:Push all page zero B9 zz 00
	PHA		:pseudo-registers M0 to M5 48
	DEY		:with M0 on stack top. 88
	BNE	PUL1	:Leave Y = 0. D0 F9
:			
	TSX		:Index Stack below M0. BA
	INC	\$010B,X	:Increment low order byte FE 0B 01
	LDA	\$010B,X	:of return address (and BD 0B 01
	STA	M0	:also high order byte if 85 M0
	BNE	PUL2	:necessary) and copy D0 03
	INC	\$010C,X	:it to page zero so that FE 0C 01
PUL2	LDA	\$010C,X	:it can be used to index BD 0C 01
	STA	M1	:and get parameter byte. 85 M1
:			
	LDA	#\$E1	:Register-size code, 1 byte A9 E1
	STA	M2	:(reset) or 2 bytes (set). 85 M2
	LDA	(M0),Y	:Copy parameter byte to B1 M0
	STA	M3	:page zero for testing. 85 M3
	LDA	#8	:Set count for 8 bits A9 08
	STA	M4	:in parameter byte. 85 M4
:			
	:...Pull loop, 8 iterations, 1 or 2 bytes pulled each loop.		
PUL3	ROR	M3	:Shift next parameter bit to 66 M3
	BCC	PUL4	:carry, skip if no pull. 90 10
:			
	LDA	(USPL),Y	:Copy one byte from Ustack B1 qq
	STA	\$0101,X	:to stack and index next 9D 06 01
	INY		:byte on Ustack. C8
	BIT	M2	:Test pull size of current 24 M2
	BPL	PUL4	: "register", skip if 1 byte. 10 06
	LDA	(USPL),Y	:Else copy next byte from B1 qq
	STA	\$0102,X	:Ustack to stack and index 9D 02 01
	INY		:next byte on Ustack. C8
:			
PUL4	INX		:Move stack index up one E8
	ROL	M2	: "register", test reg-size 26 M2
	BCC	PUL5	:and move up by 2 bytes 90 01
	INX		:if PC, M0,1, M2,3 or M4,5. E8
:			
PUL5	DEC	M4	:Repeat for all 8 bits in C6 M4
	BNE	PUL3	:parameter byte. D0 E2

PARAMETER ACCESS, SAVE AND RESTORE

```

:
    CLC                :Compute new top-of-Ustack      18
    TYA                :address,                        98
    ADC  USPL          :USPH,L = USPH,L + Y,           65 qq
    STA  USPL          :(Y is number of bytes          85 qq
    BCC  PUL6          :pulled) leaving USP            90 02
    INC  USPH          :addressing last pull + 1.       E6 pp
:
PUL6  LDY  #0          :Index block of 6 pz bytes.     A0 00
PUL7  INY            :Index next byte and             C8
      PLA            :restore from 6502 page 1         68
      STA  ZIB,Y      :stack to correct page 0        99 zz 00
      CPY  #6          :location, repeating for       C0 06
      BNE  PUL7        :M0 to M6.                    D0 F7
:
      PLA            :Restore all                      68
      TAY            :registers and flags             A8
      PLA            :from 6502 page 1 stack and      68
      TAX            :return either to program        AA
      PLA            :location after parameter        68
      PLP            :byte (PC not pulled) or         28
      RTS            :to new PC + 1.                  60
:=====

```


5 Program Relative Addressing

The routines in this chapter all use the current contents of the Program Counter – the pointer register used by the CPU to keep track of program flow.

Some instruction sets, notably that of the 6809, support various program relative addressing modes. Jumps, subroutine calls, loads, stores, shifts and indeed all program control and memory operations can be performed relative to the program position of the instruction. Consequently, any 6809 program can be position-independent and the object code will operate correctly at any location.

The only form of program relative addressing available on the 6502 are the eight conditional Branch instructions. To operate on memory you need to specify the actual location, either as an absolute direct address or indirectly by reference to two consecutive page zero bytes where the address is stored. Direct and indirect addresses can also be indexed in various ways using the X and Y registers for many 6502 operations but not for jumps or subroutine calls.

DYNAMIC ADDRESSING

If you can gain access to the address held in the Program Counter then you can use it dynamically to address any part of the program, or its associated data, as an offset from the current instruction. However, the only way to access the contents of the 6502's Program Counter is to take it from the stack inside a subroutine – which must be addressed directly. The first 6502 *Sub Set* routine to do this was `FIND` by P. Nowasad.

`FIND`, rather strangely, pulls the return address into Y (high order byte) and X, increments YX and then returns, not to the routine which called it but to the routine or program which called that one. (The increment is needed to form the true return address; the 6502 stacks the address of the third byte of the 'JSR' instruction).

PROGRAM RELATIVE ADDRESSING

This sample use of `FIND` was given in *Sub Set* (`WRITE` is a routine to print the message pointed to by `YX`):

Address	Label	Instruction	Comment
0000		JSR BLOCK	:Call to get message address.
0003		JSR WRITE	:Call to print message at YX.
.		.	.
.		.	.
0060	BLOCK	JSR FIND	:Get message address to YX,
0063		.BYTE 7	:returning to "JSR WRITE" on
0064		.TEXT "Message"	:exit from FIND.

To the best of my knowledge, no *Sub Set* reader wrote in to ask why, since the absolute, non-dynamic, address of `BLOCK` must have been programmed into `JSR BLOCK`, a routine such as `FIND` was necessary. After all, the entire operation could be simplified to:

0000		LDY #BLOCKH	:Y = BLOCK address hi-byte.
0002		LDX #BLOCKL	:X = BLOCK address lo-byte.
0004		JSR WRITE	:Call to print message at YX.
.		.	.
.		.	.
0060	BLOCK	.BYTE 7	:Message length.
0061		.TEXT "Message"	:Message contents.

The only reason that I can think of for not doing it the second way is that an error would occur if the object program (machine code) were relocated by a routine intelligent enough to change all '`JMP`' and '`JSR`' address fields to the new position but too dumb to recognise that the immediate data in `Y` and `X` is an address.

```

=====
:= FIND      Find address of memory block.
=====
:JOB         To return control to a program one level higher
:            than that which called FIND, with the address of
:            the FIND-calling program in registers.
:ACTION      Pull return address to registers.
:            Return.
:-----
:CPU         6502
:HARDWARE    None.
:SOFTWARE    None.
:-----
:INPUT       The program needing the address of the memory
:            block must have called that block; the block
:            must then have immediately called FIND.
:OUTPUT      YX addresses the location following "JSR FIND".
:            P and A are changed.
:            Exit is to the program which called the block
:            that called FIND.

```

PROGRAM RELATIVE ADDRESSING

```

:ERRORS      None.
:REG USE     P A X Y
:STACK USE   None (-2).
:RAM USE     None.
:LENGTH     9
:CYCLES      23 (+1 if address high byte is incremented).
:-----
:CLASS 2     -discreet *interruptable *promable
:*****      *reentrant *relocatable *robust
:=====
:
FIND  PLA      :Pull return address - 1          68
      TAX      :low order byte to X via A.        AA
      PLA      :Pull return address - 1          68
      TAY      :high order byte to Y via A.       A8
:
      INX      :Increment to address byte        E8
      BNE FNDOUT :immediately after "JSR FIND",   DO 01
      INY      :exit to program which called     C8
FNDOUT RTS     :block with block address in YX.   60
:=====

```

FIND OUT WHERE I'M AT

FIND is actually a variation on the basic concept of a routine which outputs, in registers or memory, the address it returns to. Once the address has been found, it can be used dynamically to make all further addressing program relative. The original idea in *Sub Set* was a 3-byte Z-80 routine:

```

FOWIA POP HL   :Pull (pop in Z-80) return address to E1
      PUSH HL  :16-bit register HL. Save it again and E5
      RET      :return to instruction addressed by HL. C9

```

FIND can easily be adapted to perform the same operation as FOWIA by inserting three instructions at the end of the first section (after 'TAY'):

```

PHA      :Save return address back to stack      48
TXA      :so exit is to 1st byte of instruction 8A
PHA      :following "JSR FIND", addressed by YX. 48

```

A more useful approach is that of GETLOC by Vincent Fojut. 2 bytes longer than the adapted FIND but 6 or 7 cycles quicker, GETLOC stores the true return address in page zero where it can be used with various indirect addressing modes.

```

:=====
:= GETLOC    Get program location.
:=====
:JOB        To return the current Program Counter contents
:           of the calling program.
:ACTION     Pull return address to pseudo-register memory.
:           Jump to address in memory.

```

PROGRAM RELATIVE ADDRESSING

```
:-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
:-----
:INPUT        None.
:OUTPUT       M0,1 addresses the location immediately
:             following "JSR GETLOC".
:             P and A are changed.
:ERRORS       Arithmetic error if D = 1 (decimal mode).
:REG USE      P A
:STACK USE    None.
:RAM USE      M0 M1
:LENGTH      14
:CYCLES       25
:-----
:CLASS 2      , -discreet *interruptable *promable
:-----
:*****      *reentrant *relocatable -robust
:-----
:
GETLOC CLC          :Clear for no-carry addition.          18
      PLA           :Pull return address - 1 low           68
      ADC #1        :order byte, incrementing for          69 01
      STA M0        :true return address, into M0.         85 M0
      PLA           :Pull return address - 1 high           68
      ADC #0        :order byte, adding any carry          69 00
      STA M1        :from lo-byte increment, to M1.        85 M1
      JMP (M0)      :Jump to true return address.          6C M0 00
:-----
```

PROGRAM RELATIVE SUBROUTINE CALLS

A subroutine which pulls the return address from stack can, of course, use it to address the program which called it and pick up any data or parameters embedded in the program.

This is how RLTVL by Gavin Every picks up the single-byte signed offset written as the next byte after JSR RLTVL. The routine could just as well pick up a 16-bit offset allowing program relative calls to anywhere in memory.

To convert RLTVL into a relative jump routine, omit the initial two pushes of P which clear a space for the call address and change the indexing of the stacked return address from '\$0109,X' and '\$010A,X' to '0107,X' and '0108,X'. The section which puts the the computed offset address on stack does not need to be changed, it will now overwrite the return address.

Note that 'RTI' is not simply the equivalent of 'PLP' with 'RTS'. The 'Return from Subroutine' instruction increments the address pulled to the Program Counter but the 'Return from Interrupt' does not.

PROGRAM RELATIVE ADDRESSING

```

=====
:= RLTVL    Program relative subroutine call.
=====
:JOB      To pass control to a subroutine at an address
:          formed by adding a signed offset byte (-128 to
:          +127) to the current Program Counter contents.
:ACTION   Reserve stack space for subroutine address.
:          Get return address from stack.
:          Using return address as index, get offset byte.
:          Add offset to return address.
:          Store computed address to reserved stack space.
:          "Return" to displaced subroutine.
-----
:CPU      6502
:HARDWARE  None.
:SOFTWARE  None.
-----
:INPUT    The signed offset byte must immediately follow
:          "JSR RLTVL" and must give the displacement of
:          the desired routine from the next location.
:OUTPUT   Exit is made to location:
:          Return address + 2 + signed offset
:ERRORS   None.
:REG USE   None.
:STACK USE 8
:RAM USE   None.
:LENGTH   67
:CYCLES    Minimum: 131. Maximum: 139.
-----
:CLASS 1   *discreet *interruptable *promable
:*****    *reentrant *relocatable *robust
=====
PAD      =      MO      :2-byte address storage in page zero.
:
RLTVL    PHP      :Reserve space for computed      08
          PHP      :displaced subroutine address.  08
:
          PHP      :Save                          08
          PHA      :flags                          48
          TXA      :and                            8A
          PHA      :registers                       48
          TYA      :on                             98
          PHA      :stack.                         48
          LDA PAD+1 :Save two bytes                 A5 M1
          PHA      :of page zero                    48
          LDA PAD   :on stack for use                A5 MO
          PHA      :indexing offset byte.           48
          CLD      :Ensure binary addition.         D8
:
          TSX      :Index stack from below MO      BA
          INC $0109,X :and increment return address FE 09 01
          BNE GETOA  :to allow for offset byte,     D0 03
          INC $010A,X :addressing offset byte.      FE 0A 01
:
GETOA    LDA $0109,X :Move incremented return      BD 09 01
          STA PAD    :address to page zero for     85 MO
          LDY $010A,X :offset byte indexing,       BC 0A 01
          STY PAD+1  :leaving hi-byte in Y.        84 M1

```

PROGRAM RELATIVE ADDRESSING

```

:
    LDX #0      :Zero index, since (PAD) is      A2 00
    LDA (PAD,X) :address of offset. Get offset   A1 M0
    BPL ADDOFF  :byte, if negative then adjust  10 01
    DEY        :current PC hi-byte.            88

:
ADDOFF SEC      :Add offset to adjusted current 38
    ADC PAD     :Program Counter (+ 1 to allow 65 M0
    BCC STSA    :for offset byte) giving       90 01
    INY        :address of displaced routine.   C8

:
STSA TSX        :Index stack from below M0     BA
    STA $0107,X :and store actual address of   9D 07 01
    TYA        :displaced routine in 2 stack  98
    STA $0108,X :bytes reserved for it.       9D 08 01

:
    PLA        :Restore                       68
    STA PAD    :two page zero                 85 M0
    PLA        :bytes used for                68
    STA PAD+1  :holding offset address.       85 M1
    PLA        :Restore registers             68
    TAY        :Y, X and A                    A8
    PLA        :by normal pulling, then       68
    TAX        :use RTI to restore P          AA
    PLA        :and effect jump to displaced  68
    RTI        :routine without PC increment. 40

:=====

```

BIRCH is a *Sub Set* routine that I wrote to cut down program length by providing shorter relative calls. The idea is similar to the use of the RESTART instructions of the Z-80 which can function as single-byte subroutine calls.

Both external interrupt and the software interrupt 'BRK' cause the current contents of the PC to be pushed to stack and the PC loaded from memory locations \$FFFE and \$FFFF. The two types of interrupt are distinguished by the state of the Break flag which is reset if an external interrupt has occurred but set for 'BRK'.

The 6502's single-byte 'BRK' can be followed by any number of parameters. In BIRCH, just one byte is appended and if this is a zero then a 'breakpoint' is indicated. A non-zero second byte is treated as an offset and added to the return address forming the relative call address.

```

:=====
:= BIRCH      Break, Interrupt and Relative Call Handler.
:=====
:JOB         To distinguish between hardware and software
:            interrupts, doing preparatory register storage,
:            and treat software interrupts with a following
:            non-zero byte as a program relative subroutine
:            call, the byte giving the signed offset to the
:            displaced subroutine.

```

PROGRAM RELATIVE ADDRESSING

```

:ACTION      IF hardware Interrupt:
:             THEN:
:             [ Jump to Interrupt service routine. ]
:             ELSE:
:             [ Push registers and exchange stack/memory.
:               IF offset byte = 0:
:               THEN:
:               [ Jump to Breakpoint service routine. ]
:             ELSE:
:             [ Subroutine Addr. = Return addr. + offset.
:               Exchange stack/memory and pull registers.
:               Jump to displaced subroutine. ] ]
:-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     "BOX" - Subroutine to push registers and
:             exchange stack block with page zero block.
:             "COX" - Subroutine to exchange page zero block
:             with stack block and pull registers.
:             Vectored Break and Interrupt service routines
:             are needed for the use of those functions. The
:             Break routine should end "JSR COX; RTS" and the
:             Interrupt routine should end "RTI".
:-----
:INPUT        If BRK entry then the byte following the BRK
:             instruction must be either 0 (for the Break
:             function) or relative subroutine call offset
:             (-128 to +127).
:OUTPUT       B = 0 on entry: Jump to vectored Interrupt
:             service routine with PC, P and A on stack.
:             B = 1 on entry:
:             BRK+1 = 0: Jump to vectored Break routine.
:             M8 to MF on stack.
:             ME, F = PC (offset addr. or ret. addr. - 1)
:             MD = Stack pointer immediately before BRK.
:             MC = Current Stack pointer.
:             MB, MA, M9, M8 = P, A, X, Y before BRK.
:             A = 0. X = S. Y = return address low byte.
:             Reset: D. Set: B, I. Unknown: S, V, Z, C.
:             BRK+1 > 0: Exit to displaced subroutine.
:             Register and page zero values as immediately
:             before BRK.
:             Return address - 1 on stack top for RTS.
:ERRORS       None.
:REG USE      Interrupt: None. Relative Call: None.
:             Break: Registers changed but saved in page zero.
:STACK USE    (including return address on stack at entry)
:             IRQ: 5. Break: 10 (including JSR BOX).
:             Relative Call: 10 (including JSR BOX, JMP COX).
:RAM USE      Interrupt: None. Relative Call: None.
:             Break: M8 to MF are in use but saved on stack.
:LENGTH      55
:CYCLES       Interrupt: 19. Break: 354 minimum.
:             Relative Call: 663 minimum.
:-----
:CLASS 2      -discreet *interruptable *promable
:-----
:*****      *reentrant *relocatable *robust
:=====

```

PROGRAM RELATIVE ADDRESSING

```

:
IRQVEC = $hilo :Address of location holding address
              :of interrupt service routine.
BRKVEC = $hilo :Address of location holding address
              :of Breakpoint service routine.

:
BIRCH PHA      :Save A and                      48
      PHP      :move status                      08
      PLA      :to A for test of                 68
      AND #$10  :Break flag.                     29 10
      BNE OSIER :Skip if set (BRK), else          00 03
      JMP (IRQVEC) :jump to interrupt service.    6C lo hi

:
OSIER JSR BOX+2 :Get regs in page zero.           20 lo hi
      CLD      :Clear for binary addition.        D8
      STY MD    :MD = Stack pointer at BRK.       84 MD

:
      LDA MB     :Get stored status                A5 MB
      AND #$EF   :register P and clear             29 EF
      STA MB     :Break flag, then re-store.       85 MB

:
BEECH LDY ME     :Decrement stored return          A4 ME
      BNE ALDER  :address for RTS return and       00 02
      DEC MF     :to address offset byte          C6 MF
ALDER DEC ME     :at BRK + 1.                     C6 ME

:
      LDA (ME,X) :Get offset byte and skip if      A1 ME
      BPL SAVIN  :forward call (with X = 0)        10 01
      DEX        :else X = $FF as sign extend.     CA
SAVIN BEQ BRIAR  :Skip if offset 0 (Break).        F0 0C

:
      ADC ME     :Compute offset address - 1:      65 ME
      TAY        :add offset in X and A to         A8
      TXA        :return address in MF and ME      8A
      ADC MF     :result in Y and A.               65 MF

:
CEDAR PHA      :Push offset address - 1           48
      TYA      :on to top of stack                98
      PHA      :for RTS to displaced              48
      JMP COX   :routine from COX.                4C lo hi

:
BRIAR TSX      :Move current Stack pointer        BA
      STX MC    :to MC to complete info in        86 MC
      JMP (BRKVEC) :page zero for Break routine.  6C lo hi
:=====

```

You might find that a single-byte offset offers too small a range. BIRCH can be changed to operate with a 16-byte offset following the 'BRK' instruction. Substitute the following 24 bytes of code for the 21 bytes in the three sections beginning at the label BEECH and ending just before label CEDAR.

PROGRAM RELATIVE ADDRESSING

LARCH	LDY	#\$FF	:Index offset lo-byte at one	A0 FF
	DEC	MF	:less than return address.	C6 MF
	LDA	(ME),Y	:Get offset lo-byte and	B1 ME
	INC	MF	:restore return address.	E6 MF
	CLC		:Add offset lo-byte to	18
	ADC	ME	:return address lo-byte	65 ME
	TAY		:with result to Y. Then get	A8
	LDA	(ME,X)	:offset hi-byte + return address	A1 ME
	ADC	MF	:hi-byte + C from lo-byte add.	65 MF
:				
	CMP	MF	:Compare result with return	C5 MF
	BNE	CEDAR	:address to test for zero offset	D0 04
	CPY	ME	:going to breakpoint if zero but	C4 ME
	BEQ	BRIAR	:relative call if not zero.	F0 06

BREAKPOINTS

BOX and COX from the last chapter are used to good effect in BIRCH which has stored all necessary register contents in page zero when it exits to your breakpoint routine. Being located in page zero makes them readily accessible for display and the stored Program Counter can be used to index program memory.

After your breakpoint routine has allowed you to review the current state of the machine and make any necessary adjustments to register contents (including the Program Counter for continuation at a different place), end it with JSR COX to re-stack then restore all registers and 'RTS' to get back into the program.

LONGER BRANCHES

LONGBR by Vincent Fojut gives to the 6502 almost the same branching freedom as the 6809 which can branch on simple (1 flag test) or complex (2 or 3 flags tested) conditions. The call to LONGBR must, of course, have a 16-bit offset following the 'JSR' instruction but it also needs a one-byte parameter or mask giving the jump conditions.

Bit 5 of the P register does not show any status and so the corresponding bit in the condition mask can be used to indicate whether the jump is to be made on set or reset flags. The other bits in the mask are used to determine which flags are to be tested.

The final test is made on the zero status of the complete set of selected flags. This does have the unfortunate result of resetting Z if ANY tested flag is set but setting Z only if ALL tested flags are reset. So, for example, a branch made on the combined state of the Zero and Carry flags would have these two asymmetric effects:

PROGRAM RELATIVE ADDRESSING

Z	C	BRANCH IF CLEAR (Z AND C = 0)	BRANCH IF SET (Z OR C = 1)
0	0	Branch	No Branch
0	1	No Branch	Branch
1	0	No Branch	Branch
1	1	No Branch	Branch

```

=====
:= LONGBR 16-bit offset, complex-conditional branch.
=====
:JOB      To test status against a condition mask and
:          branch to an offset address if conditions true.
:ACTION   Store status register.
:          Pull return address as mask and offset address.
:          Get mask. Isolate true-on-set/reset bit.
:          Mask status register, isolating test bits.
:          IF true-on-set/reset = masked-status THEN:
:          [ Get offset.
:            Branch address = return address + offset. ]
:          Bump return/branch address past mask and offset.
:          Jump to return/branch address.
-----
:CPU      6502
:HARDWARE None.
:SOFTWARE None.
-----
:INPUT    Condition mask byte following "JSR LONGBR".
:          16-bit offset (low order byte first) immediately
:          following condition mask byte.
:          The condition mask is constructed as follows:
:          Bit 5 = 0: condition true if ALL of the tested
:          flags are reset.
:          Bit 5 = 1: condition true if ANY of the tested
:          flags are set.
:          Bits 7,6,4,3,2,1,0: set to include corresponding
:          status bit in test, reset to exclude the
:          corresponding status bit:
:          Bit 7: N Negative (sign) flag.
:          Bit 6: V Overflow flag.
:          Bit 4: B Break flag.
:          Bit 3: D Decimal mode flag.
:          Bit 2: I Interrupt disable flag.
:          Bit 1: Z Zero flag.
:          Bit 0: C Carry flag.
:OUTPUT   Condition not met: M0,1 = address of location
:          following 16-bit offset.
:          Condition met: M0,1 = address of location at
:          offset + address of location after offset.
:          Exit to location (M0,1).
:          M2 = A. M3 is changed.
:ERRORS   None.
:REG USE   None.
:STACK USE 2
:RAM USE   M0 to M3
:LENGTH   81
:CYCLES    No branch: 115 to 119. Branch: 152 to 156.

```

PROGRAM RELATIVE ADDRESSING

```

:-----
:CLASS 2      -discreet *interruptable *promable
:*****      *reentrant *relocatable *robust
:=====
:
PCAD  =  M0      :Return/branch address 2-byte storage.
ATS   =  M2      :Temporary store for A.
STATE =  M3      :Store for flags and test results.
:
:
LONGBR STA  ATS      :Save A and P in page zero      85 M2
      PHP           :so not to block stack for      08
      PLA           :pulling of return address.      68
      STA  STATE     :Also save status for test.      85 M3
:
      PLA           :Pull stacked Program Counter    68
      STA  PCAD      :(return address - 1) and      85 M0
      PLA           :store to page zero as code,      68
      STA  PCAD+1     :offset and return address.    85 M1
:
      LDA  STATE     :Now save P, copied from        A5 M3
      PHA           :page zero.                      48
      TYA           :Save Y for use as index          98
      PHA           :and temporary storage.          48
      CLD           :Ensure binary arithmetic.       08
:
      LDY  #1         :Index condition mask byte      A0 01
      LDA  (PCAD),Y   :and get in A. Save in Y        B1 M0
      TAY           :and check bit 5 for reset        A8
      AND  #$20       :or set condition true.         29 20
      PHP           :Save test result (Z flag).       08
      TYA           :Recover mask to A.              98
:
      AND  #$DF       :Mask out reset/set bit 5.      29 DF
      AND  STATE     :Isolate status bits            25 M3
      PHP           :to test and move flag           08
      PLA           :results to A, masking out        68
      AND  #2         :all but Z result and save it   29 02
      STA  STATE     :for true/false check.          85 M3
:
      PLA           :Recover reset/set truth.         68
      AND  #2         :Isolate Z result and compare   29 02
      EOR  STATE     :with status, skipping if        45 M3
      BNE  INC4       :status not true (no match).     00 12
:
      LDY  #3         :Condition met so index offset  A0 03
      LDA  (PCAD),Y   :getting high order byte       B1 M0
      PHA           :saved on stack.                  48
      DEY           :Index offset low order byte      88
      LDA  (PCAD),Y   :and get in A.                  B1 M0
:
      CLC           :Prepare to add.                  18
      ADC  PCAD       :Add offset lo-byte to          65 M0
      STA  PCAD       :Program Counter lo-byte.       85 M0
      PLA           :Recover offset hi-byte           68
      ADC  PCAD+1     :and add to PC hi-byte.         65 M1
      STA  PCAD+1     :PCAD = branch address - 4.     85 M1

```

PROGRAM RELATIVE ADDRESSING

```

:
INC4  CLC          :Prepare to add.          18
      LDA  #4       :Add 4 to return/branch   A9 04
      ADC  PCAD      :address to skip condition 65 M0
      STA  PCAD      :mask and offset and convert 85 M0
      BCC  RESTOR    :return address - 1 to true 90 02
      INC  PCAD+1    :return address for jump.   E6 M1

:
RESTOR PLA         :Restore Y from            68
      TAY          :stack via A.             A8
      LDA  ATS      :Restore A from page zero. A5 M2
      PLP          :Restore P and exit by jump 28
      JMP  (PCAD)    :to return or branch address. 6C M0 00
:=====

```

Search, Sort and Case

The three routines in this chapter perform quite diverse tasks. Their one common aspect is that they all make fundamental use of the computer's ability to discriminate between data.

KEYWORD STRING SEARCH

Dennis May's routine `MATCH` is not quite the usual string search routine. You wouldn't use it to try and match a complete string of finite length against entries in a table but rather to test for the occurrence of any 'keyword' substrings within an object string which may be any length.

As an example of how useful such a routine is, imagine the need to compress a large text file. The table would contain 128 of the most common English words and phrases and `MATCH` used to find all occurrences of them within the text. The higher level routine, which calls `MATCH`, would replace each matched substring by a single byte token equal to 128 more than its table position. Routines to effect this compression, using `MATCH`, and to re-expand the text are in the next chapter.

There are, of course, many other uses for `MATCH`. A couple that spring to mind are searching for keyword commands in a high level language program (such as `PRINT`, `GOSUB`, `OPEN`, and so on) and as part of the parsing process to decode the meaning of a sentence in an Artificial Intelligence program.

```

=====
:= MATCH      Keyword substring match.
=====
:JOB          To test for a match between any string held in a
:              table of strings and the whole or first part of
:              an object string, returning table position of
:              string if match found.

```

SEARCH, SORT AND CASE

```

:ACTION      String position index = 0.
:            REPEAT:
:            [ REPEAT:
:              [ Index next character position.
:                Compare characters. ]
:            UNTIL characters not equal.
:            IF end of table string:
:            THEN: [ Address object string + 1. ]
:            ELSE: [ Address next table string.
:                  Increment string position index.
:                  Test for end of table. ] ]
:            UNTIL string match OR end of table.
:-----
:CPU          6502
:HARDWARE     Object string and string table in memory.
:SOFTWARE     None.
:-----
:INPUT        M0,1 addresses start of object string to match.
:            String length: >= longest string in table.
:            String terminator: none.
:            M4,5 addresses start of string table.
:            Table length: up to 254 strings.
:            Table terminator: byte $FF.
:            String length: up to 255 characters (bytes).
:            Character (byte) range: 0 to $7F.
:            String terminator: bit 7 set on last byte.
:OUTPUT       X = $FF: Match not found.
:            M2 = X. Y = 0. A = $FF. P is changed.
:            M4,5 addresses table terminator byte ($FF).
:            M0,1 and M3 are unchanged.
:            X < $FF: Match found.
:            M2 = X. A and P are changed.
:            M3 = Y = length - 1 of matched string.
:            M4,5 addresses start of matched table string.
:            M0,1 addresses end + 1 of matched section of
:            object string.
:ERRORS       Arithmetic error if D = 1 (decimal mode).
:            No overflow check on either string lengths or
:            number of strings in table, both will give
:            erroneous output information.
:REG USE      P A X Y
:STACK USE    None.
:RAM USE      M0 to M5
:LENGTH       61
:CYCLES       Not given.
:-----
:CLASS 2      -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
:=====
:
OBJ      =      M0          :Stored address of object string.
POS      =      M2          :For storing matched string table index.
LEN      =      M3          :For storing matched string length - 1.
TAB      =      M4          :Stored address of string table.
:
MATCH    LDY    #0          :Initialise string index and           AO 00
          STY    POS        :position index to zero.           84 M2
:

```

SEARCH, SORT AND CASE

```

STRLP DEY          :Initially index byte 0.          88
:
CHARLP INY         :Index next byte of both strings. C8
      LDA (OBJ),Y  :Get object byte and, ensuring   B1 M0
      SEC          :no borrow going in, subtract    38
      SBC (TAB),Y  :current table string byte, loop  F1 M4
      BEQ CHARLP   :until bytes don't match.        F0 F8
:
      CMP #$80     :If difference is only last-char C9 80
      BEQ FOUND    :bit 7, then string match found. F0 1E
:
      DEY         :Else, test from no-match byte.   88
NXTSTR INY         :Index next string byte and load  C8
      LDA (TAB),Y  :to test sign bit, looping until  B1 M4
      BPL NXTSTR   :last-char (bit 7 set) indexed.  10 FB
:
      TYA         :Move index to A and, adding in   98
      SEC         :set carry to convert index to    38
      ADC TAB      :string length, add to base      65 M4
      STA TAB      :address so address incremented  85 M4
      BCC ENDTST   :to start of next string in     90 02
      INC TAB+1    :table, or table terminator.     E6 M5
:
ENDTST INC POS     :Bump string index.               E6 M2
      LDY #0       :Index byte 0 of next string     A0 00
      LDA (TAB),Y  :and load to test for table      B1 M4
      CMP #$FF     :terminator, repeating if any    C9 FF
      BNE STRLP    :strings left to test.           D0 D9
:
      STA POS      :Else string index = -1 for      85 M2
      BEQ EXIT     :match not found, go exit.       F0 0B
:
FOUND STY LEN      :Store string length - 1.        84 M3
      TYA         :Move last byte index to A and,   98
      ADC OBJ      :adding set carry (from CMP) to  65 M0
      STA OBJ      :convert index to length, add to 85 M0
      BCC EXIT     :string address, addressing byte 90 02
      INC OBJ+1    :following matched substring.    E6 M1
:
EXIT LDX POS       :Output X is string position     A6 M2
      RTS         :index (match) or -1 (no match).  60
:=====

```

STRING SORT

ALSORT is another routine by Dennis May. This time the comparison is not to search for equality but to determine the correct sequence of 2 strings.

ALSORT utilises the bubble sort which has acquired the reputation of inefficiency. Bubble sorts are very slow compared to other sorts such as the Shell sort or Hoare's 'Quicksort', which has the distinction of being both the quickest and slowest (in its worst case) known method of sorting.

SEARCH, SORT AND CASE

Although slow, the bubble sort is very easy to implement and if you are dealing with elements of different length then it is one of only two really practical methods – the other being a variation known as the ‘ripple’ sort. All the exchanges are between two consecutive elements and consequently there are no problems of spacing as there would be in other sorts. The Quicksort, for example, could involve different sized elements from the start and end of the list being exchanged and this would require all intermediate bytes to be shifted by the difference between the 2 exchanged elements.

```
:======
:= ALSORT   ASCII sort of variable length strings.
:======
:JOB        Bubble sort. To sort a RAM held list of variable
:           length strings into ascending ASCII order.
:ACTION     IF number of strings > 1 THEN:
:           [ REPEAT:
:             [ Clear pass exchange flag.
:             FOR number of strings - 1:
:             [ IF current string > next string THEN:
:             [ Move current to temporary.
:             Move next to current.
:             Move temporary to current + 1.
:             Set pass exchange flag. ] ] ]
:           UNTIL pass exchange flag clear. ]
:-----
:CPU        6502
:HARDWARE    List of strings in RAM.
:           Temporary RAM storage to hold longest string.
:SOFTWARE    None.
:-----
:INPUT      M0,1 addresses start of string list.
:           M2,3 contains number of strings in list.
:           Max. number of strings: 65535.
:           Max. string length: 256 including terminator.
:           String terminator: $0D (carriage return).
:           Character range: 0 to $FF (excluding $0D).
:OUTPUT     List sorted in ascending numerical order.
:           M0 to M3 unchanged. P, A and X changed.
:           M4 to MA, Y and temporary store may be changed.
:ERRORS     Arithmetic error if D = 1 (decimal mode).
:REG USE    P A X Y
:STACK USE  1
:RAM USE    M0 to MA
:LENGTH     161
:CYCLES     Not given.
:-----
:CLASS 2    -discreet *interruptable *promable
:--*---     -reentrant -relocatable -robust
:======
```

SEARCH, SORT AND CASE

```

:
TEMP = $hilo :Temporary storage during exchange.
LSA = M0 :2-byte stored start address of list.
INS = M2 :2-byte stored input number of strings.
NSP = M4 :2-byte next string pointer store.
SC = M6 :2-byte string count store.
CSP = M8 :2-byte current string pointer store.
PXF = MA :1-byte pass exchange flag store.
CRST = $0D :Carriage-return string terminator.
:
ALSORT LDX INS :Get number of strings lo-byte A6 M2
      TXA :in X and A. 8A
      ORA INS+1 :Test by ORing hi-byte if 05 M3
      BNE ALSB :number is zero, continue if D0 01
ALSA RTS :not, else exit immediately. 60
:
ALSB DEX :Now test for only 1 string, CA
      TXA :lo-byte dec gives zero result 8A
      ORA INS+1 :on OR with hi-byte if only 1. 05 M3
      BEQ ALSA :Exit if only 1, else sort. F0 F9
:
ALSC LDX INS+1 :Pass initialisation. A6 M3
      LDY INS :Set string count to A4 M2
      STX SC+1 :number of strings in list. 86 M7
      STY SC : 84 M6
      LDX LSA+1 :Set next string pointer A6 M1
      LDY LSA :to list start address. A4 M0
      STX NSP+1 : 86 M5
      STY NSP : 84 M4
      LDY #0 :Clear pass exchange flag. A0 00
      STY PXF : 84 MA
:
ALSD LDX NSP+1 :Initially, or if no exchange A6 M5
      LDA NSP :last comparison, set current A5 M4
      STX CSP+1 :pointer to next pointer. 86 M9
      STA CSP : (Exchange moves pointer). 85 M8
:
ALSE LDY #-1 :Find start of next string. A0 FF
ALSF INY :Test from byte 0 C8
      LDA (NSP),Y :looping until index Y B1 M4
      CMP #CRST :points to terminator C9 0D
      BNE ALSF :of current string. D0 F9
:
      TYA :Add index + set Carry 98
      SEC :as current string length 38
      ADC NSP :to next string pointer, 65 M4
      STA NSP :moving 85 M4
      BCC ALSG :it from current 90 02
      INC NSP+1 :to next string. E6 M5
:
ALSG LDY #-1 :Compare strings. A0 FF
ALSH INY :Index from byte 0 comparing C8
      LDA (NSP),Y :next with current, skip out B1 M4
      CMP (CSP),Y :if next < current (C = 0) D1 M8
      BNE ALSI :or next > current (C = 1). D0 04
      CMP #CRST :If terminator then end loop C9 0D
      BNE ALSH :with current = next (C = 1). D0 F5
ALSI BCS ALSN :Skip if next >= current. B0 34

```

SEARCH, SORT AND CASE

:			
ALSJ	LDY #-1	:Exchange strings (1).	A0 FF
	INY	:Move current string	C8
	LDA (CSP),Y	:to temporary store,	B1 M8
	STA TEMP,Y	:from byte 0 to	99 lo hi
	CMP #CRST	:and including terminator.	C9 0D
	BNE ALSJ	:	D0 F6
:			
ALSK	LDY #-1	:Exchange strings (2).	A0 FF
	INY	:Move next string	C8
	LDA (NSP),Y	:down memory to	B1 M4
	STA (CSP),Y	:replace current string,	91 M8
	CMP #CRST	:from byte 0 to	C9 0D
	BNE ALSK	:and including terminator.	D0 F7
:			
	TYA	:Add index + set Carry as	98
	SEC	:length of moved string	38
	ADC CSP	:to current string pointer	65 M8
	STA CSP	:so it now addresses 1st byte	85 M8
	BCC ALSL	:of space to receive stored	90 02
	INC CSP+1	:current string.	E6 M9
:			
ALSL	LDY #-1	:Exchange strings (3).	A0 FF
ALSM	INY	:Move stored current string	C8
	LDA TEMP,Y	:to above shifted down next	B9 lo hi
	STA (CSP),Y	:string, from byte 0 to and	91 M8
	CMP #CRST	:including terminator.	C9 0D
	BNE ALSM	:(CMP leaves Carry set.)	D0 F6
:			
	ROR PXF	:Set bit 7 of pass exchange	66 MA
	LDA #\$FF	:flag. Set current exchange	A9 FF
	PHA	:flag on stack so current	48
	BMI ALSO	:pointer not moved again.	30 03
:			
ALSN	LDA #0	:Clear current exchange flag	A9 00
	PHA	:so current pointer is moved.	48
:			
ALSO	LDX SC	:Subtract string done	A6 M6
	BNE ALSP	:(compared, if not exchanged)	D0 02
	DEC SC+1	:from string count.	C6 M7
ALSP	DEC SC	:	C6 M6
:			
	LDX SC	:Test for end of pass.	A6 M6
	DEX	:Pass ends when only	CA
	TXA	:one string remains.	8A
	ORA SC+1	:Lo-byte - 1 OR hi-byte	05 M7
	BEQ ALSQ	:is zero when count = 1.	F0 05
:			
	PLA	:Get current exchange flag.	68
	BNE ALSE	:If set then current already	D0 94
	BEQ ALSD	:at next, else go move it.	F0 8A
:			
ALSQ	PLA	:End of pass. Tidy up stack.	68
	BIT PXF	:If pass exchange flag (bit 7)	24 MA
	BPL ALSR	:is clear then end, else	10 03
	JMP ALSC	:another pass to do.	4C lo hi
ALSR	RTS	:Exit ALSORT, list sorted.	60
:	=====		

QUICKER SORTING

ALSORT could be improved in efficiency, making it quicker to process long lists, but at the cost of a longer routine. The first improvement is quite easy to implement. After the first pass, the highest value string has been 'bubbled' along to the end of the list and so on the second pass the last string need not be tested. On each successive pass one more string can be disregarded. Change the routine so that at the end of each pass the number of strings to sort is decremented and a further pass occurs only if two or more strings remain.

The second improvement is more complex. Instead of beginning each pass at the start of the list, you could start at the string immediately preceding the first exchange on the last pass. All strings before it weren't exchanged last time and they are not going to be exchanged this time either! One extra variable is needed to save the address of the last string not exchanged and this becomes the start string for the next pass. Whenever the start string is exchanged, begin the next pass at the start of the list.

CASE STRUCTURE

CASEOF by Peter Villadsen is a routine to perform case structure logic on an input case-value. It is rather like the ON <variable> GOSUB <line number> command in BASIC except that the values tested need not be sequential.

Only one byte is compared in each iteration and the search is not for the table position of a match but for the address associated with the matched value and used to pass control to a case action. Each address is stored with its associated case key making all table entries 3 bytes long.

CASEOF can search through any number of case tables since the table base address is input in page zero and not programmed in. Consequently you can put it to many different uses such as calling subroutines by index rather than by address. Referring back to the subject of the last chapter, as all case actions are entered by an indirect jump to the address stored in page zero pseudo-registers M2 and M3, they can be written to use only program relative addressing.

Because indexed addressing is used by CASEOF, the maximum table size allowed is 256 bytes, or 84 cases. The 256 possible cases of a one-byte key can be tested for and acted on by using the no-match ELSE case action as a link setting CASEOF to work on 4 successive tables.

SEARCH, SORT AND CASE

```

:=====
:= CASEOF   Case structure.
:=====
:JOB        To compare an input case-value with a list of
:           case keys, branching to the associated action on
:           key match or to default action on no key match.
:ACTION     Initialise for first test on case-key 1.
:           REPEAT:
:           [ Index next case-key.
:             Compare input case-value with case-key. ]
:           UNTIL case-key match or end-of-table.
:           IF end-of-table THEN: [ Index "ELSE" case. ]
:           Jump to case address.
:-----
:CPU        6502
:HARDWARE    Memory containing case table.
:SOFTWARE    Subroutines associated with each case.
:-----
:INPUT      X contains case-value.
:           MO,1 addresses start of case table.
:           Case table format:
:           ELSE case (no key match):
:           (MO,1) + 0 : 3 * keys (i.e. table length - 3).
:           (MO,1) + 1 : ELSE case action address lo-byte.
:           (MO,1) + 2 : ELSE case action address hi-byte.
:           Key cases (n = 1 to 84):
:           (MO,1) + 3n+0 : Case n key.
:           (MO,1) + 3n+1 : Case n action address lo-byte.
:           (MO,1) + 3n+2 : Case n action address hi-byte.
:OUTPUT     M2,3 addresses action associated with either
:           matched case key or ELSE action if no key match.
:           Control passed to location addressed by M2,3.
:           All registers and MO,1 unchanged.
:ERRORS     Program control error if (MO,1) = 0 or is not a
:           multiple of 3.
:REG USE    X
:STACK USE  5 + case routine stack use in excess of 5.
:RAM USE    MO to M3
:LENGTH    49
:CYCLES     Matched case: 38 + 35 * key position.
:           ELSE case: 57 + 35 * number of keys.
:-----
:CLASS 2    *discreet *interruptable *promable
:*****-    *reentrant *relocatable -robust
:=====
:
CAS  =  MO      :2-byte stored address of case table.
ACT  =  M2      :2-byte store for action address.
:
CASEOF PHP      :Save flags                      08
      PHA      :and registers                    48
      TYA      :for use                          98
      PHA      :in CASEOF,                       48
      TXA      :leaving input case-value         8A
      PHA      :in A for comparisons.            48
      LDX #0    :Index table length byte.        A2 00
      LDY #3    :Initially index 1st key.        A0 03
:

```

SEARCH, SORT AND CASE

```

TSTNXT CMP (CAS),Y :Test for key match and      D1 M0
      BEQ FOUND :exit loop if found.          FO 10
:
      PHA :Else save case-value and          48
      TYA :compare key index with table      98
      CMP (CAS,X) :length, exiting loop to    C1 M0
      BEQ ELSE :ELSE case if end of table.    FO 07
:
      PLA :Else restore case-value          68
      INY :and increment                    C8
      INY :Y to index key                   C8
      INY :of next case.                    C8
      SEC :Then ensure branch occurs        38
      BCS TSTNXT :and go test next key.      BO EF
:
ELSE PLA :Tidy up stack.                    68
      LDY #0 :index no-match ELSE case.      AO 00
:
FOUND INY :Index lo-byte of case            C8
      LDA (CAS),Y :action address and copy    B1 M0
      STA ACT :it to action address store.    85 M2
      INY :Index hi-byte of case            C8
      LDA (CAS),Y :action address and copy    B1 M0
      STA ACT+1 :it to action address store.  85 M3
:
      PLA :Restore                          68
      TAX :all                              AA
      PLA :registers                        68
      TAY :and                              A8
      PLA :flags                            68
      PLP :then jump to correct             28
      JMP (ACT) :action for case.            6C M2 00
:=====

```


Data Compression⁷

A single-density 40-track floppy disk will hold about 100 KBytes. This might appear to be an adequate capacity for almost any purpose but it is soon used up. One disk will hold only the same amount of information as about 40 of the pages in this book.

Although memory is becoming cheaper, the RAM available in most personal computers in present use is still probably in the range 16K to 32K. With a program, or suite of programs, loaded in to deal with the processing, you will be lucky to have ten book pages worth of immediately accessible data.

Transmitting data by modem is not particularly fast, as you will know if you make extensive use of the telephone for inter-computer communication. Time is money – with a vengeance on long distance calls.

These are all good reasons for compacting data into as few bytes as possible. In this chapter you will find routines to deal with various ways of compressing and expanding both numerical and textual data.

12-BIT NUMBERS

SQSH and EXPD by David Heale can give you 25% space saving on numerical data – but only if the numbers are held to 12-bit accuracy and normally take up 2 bytes with bits 12 to 15 either zeros or ‘don’t care’. The numbers have to be stored in the normal 6502 form of low-order byte first.

SQSH processes two numbers at a time, treating them as a single block of 4 bytes (referred to as s1 to s4 in the routine documentation) with the high nibbles (top 4 bits) of the second and fourth bytes unused. The result of the processing is a block of 3 packed bytes (d1 to d3 in the routine documentation). The inverse process, EXPD, acts on a source block of 3 bytes, expanding them to two 12-bit values.

Since post-indexed addressing is used, the pointers are not moved past the block in either routine. If you repeatedly call SQSH or EXPD from inside a loop to deal with long strings of numbers, the loop will have to increment the pointers to the start of each new 4- or 3-byte block. For SQSH, you can set both source and destination pointers to the start of the string and the compressed string will replace the first 75% of the source string bytes.

DATA COMPRESSION

```

=====
:= SQSH      Squash Data.
=====
:JOB         To compress two 16-bit words of numerical data,
:            held to 12-bit accuracy, into 3 bytes.
:ACTION      Move source byte 1 to destination byte 1.
:            Discard s2-hi. Move s2-lo to d2-hi.
:            Move s3-hi to d2-lo. Move s3-lo to d3-hi.
:            Discard s4-hi. Move s4-lo to d3-lo.
:-----
:CPU         6502
:HARDWARE    RAM: 4 bytes source, 3 bytes for destination.
:SOFTWARE    None.
:-----
:INPUT       M0,1 addresses first byte of 4-byte source.
:            M2,3 addresses first byte of 3-byte destination.
:OUTPUT      M0,1 & M2,3 and source data are unchanged.
:            Y =3. A = last squashed byte.
:ERRORS      No check for destination overwrite of source.
:REG USE     A Y P
:STACK USE   None.
:RAM USE     M0 M1 M2 M3
:LENGTH     41
:CYCLES      98
:-----
:CLASS 2     -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
:=====
:
SRCES =      M0           :2-byte stored address of source.
DESTS =      M2           :2-byte stored address of destination.
:
SQSH  LDY #0           :Index first bytes.                      A0 00
      LDA (SRCES),Y :Move source 1                          B1 M0
      STA (DESTS),Y :to destination 1.                      91 M2
:
      INY           :Index second bytes.                      C8
      LDA (SRCES),Y :Get source 2,                          B1 M0
      ASL A         :lo-nibble                                0A
      ASL A         :into A                                  0A
      ASL A         :hi-nibble                                0A
      ASL A         :and store                                0A
      STA (DESTS),Y :in destination 2.                      91 M2
:
      INY           :Index third bytes.                      C8
      LDA (SRCES),Y :Get source 3,                          B1 M0
      LSR A         :hi-nibble                                4A
      LSR A         :into A                                  4A
      LSR A         :lo-nibble                                4A
      LSR A         :and merge                                4A
      DEY           :(indexing second byte)                  88
      ORA (DESTS),Y :with s2 lo-nibble                      11 M2
      STA (DESTS),Y :and store in destination 2.          91 M2
:

```

DATA COMPRESSION

```

      INY          :Index third bytes.          C8
      LDA (SRCE),Y :Get source 3,              B1 M0
      ASL A        :lo-nibble                  0A
      ASL A        :into A                     0A
      ASL A        :hi-nibble                  0A
      ASL A        :and merge                   0A
      INY          :(indexing fourth byte)      C8
      ORA (SRCE),Y :with s4 lo-nibble.         11 M0
      DEY          :Index third byte and store  88
      STA (DEST),Y :in destination 3.          91 M2
      RTS          :Exit, data compressed.      60
=====

```

```

=====
:= EXPD      Expand Data.
=====

```

```

:JOB        To expand 3 bytes of numerical data, compressed
:           by "SQSH", back to two 16-bit words with the
:           four most significant bits of each set to zero.
:ACTION     Move source byte 1 to destination byte 1.
:           Move s2-hi to d2-lo. Clear d2-hi.
:           Move s2-lo to d3-hi. Move s3-hi to d3-lo.
:           Move s3-lo to d4-lo. Clear d4-hi.
=====

```

```

:CPU        6502
:HARDWARE   RAM: 3 bytes source, 4 bytes for destination.
:SOFTWARE   None.
=====

```

```

:INPUT      M2,3 addresses first byte of 3-byte source.
:           M0,1 addresses first byte of 4-byte destination.
:OUTPUT     M0,1 & M2,3 and source data are unchanged.
:           Y =4. A = last expanded byte.
:ERRORS     No check for destination overwrite of source.
:REG USE    A Y P
:STACK USE  None.
:RAM USE    M0 M1 M2 M3
:LENGTH    42
:CYCLES     100
=====

```

```

:CLASS 2    -discreet *interruptable *promable
:*****     *reentrant *relocatable -robust
=====

```

```

DESTE = M0      :2-byte stored address of destination.
SRCEE = M2      :2-byte stored address of source.

```

```

EXPD  LDY #0     :Index first bytes.          A0 00
      LDA (SRCEE),Y :Move source 1 to        B1 M2
      STA (DESTE),Y :destination 1.          91 M0

```

```

      INY          :Index second bytes.        C8
      LDA (SRCEE),Y :Get source 2,            B1 M2
      LSR A        :hi-nibble into            4A
      LSR A        :lo-nibble A,              4A
      LSR A        :clearing hi-nibble        4A
      LSR A        :and store in              4A
      STA (DESTE),Y :destination 2.          91 M0

```

DATA COMPRESSION

```
:
LDA (SRCEE),Y :Get source 2,          B1 M2
ASL A          :lo-nibble             0A
ASL A          :into                   0A
ASL A          :hi-nibble A,           0A
ASL A          :clearing lo-nibble A,  0A
INY            :(index third bytes)    C8
STA (DESTE),Y :and store in destination 3. 91 M0

:
LDA (SRCEE),Y :Get source 3,          B1 M2
LSR A          :hi-nibble             4A
LSR A          :into                   4A
LSR A          :lo-nibble A           4A
LSR A          :and merge with         4A
ORA (DESTE),Y :s2-lo, then store      11 M0
STA (DESTE),Y :in destination 3.      91 M0

:
LDA (SRCEE),Y :Get source 3 lo-nibble, B1 M2
AND #$0F      :clearing hi-nibble,    29 0F
INY            :(index fourth byte)    C8
STA (DESTE),Y :and store in destination 4. 91 M0
RTS           :Exit, data expanded.    60
:=====
```

TOKENISED TEXT

TKNIN and TKNOUT are based on a *Sub Set Z-80* routine TOKN. TKNIN is written to make use of the keyword string comparison routine MATCH from the chapter on Comparisons. TKNOUT does not use MATCH but it does use the same token expansion table.

TKNIN converts a normal ASCII file to one which is a mixture of normal ASCII characters (\$01 to \$7F) and tokens (\$80 to \$FF). Each token is the table index to the substring it has replaced. The 128 substrings can, of course, be any of the most commonly used English phrases, words and letter groupings.

TKNOUT is the reverse process. It writes a destination file using the ASCII codes from the source but converts codes \$80 to \$FF to the full ASCII expansion found in the table. The file-out will obviously be longer than the file-in, although there is no way to tell just how much greater it will be. You must make sure that the destination area is large enough.

Both TKNIN and TKNOUT could be adapted to read from and write to disk handling routines. TKNOUT could write straight to a printer or to screen.

Here is a short example of how the system works, with the symbol _ used to clearly mark spaces. First, part of the expansion table (tokens \$80 to \$94). Bit 7 of the last byte in each string is set to show termination.

DATA COMPRESSION

TOKEN	EXPANSION (HEX)	EXPANSION (TEXT)
\$80	54 48 45 A0	THE_
\$81	49 CE	IN
\$82	57 41 53 A0	WAS_
\$83	57 4F 52 C4	WORD
\$84	41 4E 44 A0	AND_
\$85	48 49 CD	HIM
\$86	4E 4F D4	NOT
\$87	4C 49 47 48 D4	LIGHT
\$88	57 49 54 C8	WITH
\$89	47 4F C4	GOD
\$8A	4F 55 D4	OUT
\$8B	54 48 49 4E C7	THING
\$8C	41 4E 44 20 54 48 45 A0	AND_THE_
\$8D	49 4E 20 54 48 45 A0	IN_THE_
\$8E	4D 41 44 C5	MADE
\$8F	53 48 49 4E C5	SHINE
\$90	44 41 52 CB	DARK
\$91	45 4E C4	END
\$92	4E 45 53 D3	NESS
\$93	43 4F CD	COM
\$94	50 52 C5	PRE

Now the very repetitive beginning of John's Gospel using the tokens \$80 to \$94,

TOKENISED (HEX)	EXPANDED (TEXT)
8D 42 45 47 81 4E 81 47 20 82	IN_THE_BEGINNING_WAS_
83 2C 20 8C 83	THE_WORD,_AND_THE_WORD_
20 82 88 20 89 2C 20 8C	WAS_WITH_GOD,_AND_THE_
83 20 82 89 2E 20 80	WORD_WAS_GOD._THE_
53 41 4D 45 20 82 8D	SAME_WAS_IN_THE_
42 45 47 81 4E 81 47 20 88 20	BEGINNING_WITH_
89 2E 20 41 4C 4C 8B 53 20	GOD._ALL_THINGS_
57 45 52 45 20 8E 20 42 59 20	WERE_MADE_BY_
85 3B 20 84 88 8A 20 85 20	HIM;_AND_WITHOUT_HIM_
82 86 20 41 4E 59 20 8B 20	WAS_NOT_ANY_THING_
8E 20 54 48 41 54 20 82 8E 2E	MADE_THAT_WAS_MADE.
20 81 20 82 4C 49 46 45 3B 20	_IN_HIM_WAS_LIFE;_
8C 4C 49 46 45 20 82 80	AND_THE_LIFE_WAS_THE_
87 20 4F 46 20 4D 45 4F 2E 20	LIGHT_OF_MEN._
8C 87 20 8F 54 48 20	AND_THE_LIGHT_SHINETH_
81 20 90 92 3B 20 8C	IN_DARKNESS;_AND_THE_
90 92 20 93 94 48 91 45 44 20	DARKNESS_COMPREHENDED_
49 54 20 86 2E 00	IT_NOT.<terminator>

As you can see, the use of tokens has reduced a 328 character string (including null terminator) to only 152 bytes. Not quite the Bible on the back of a postage stamp but nevertheless a space saving of over 50%. The passage does contain more repetition than might be expected in normal English text but only 21 of the 128 available tokens have been used.

DATA COMPRESSION

Such words as 'IN', 'OUT', 'WITH' and 'END' have been entered in the table without spaces since this enables words like 'WITH-OUT', 'WITHIN' or 'OUTSHINE' to be formed by concatenation. They could be repeated in the full expansion table with spaces after, or even before, to give even more saving.

```

=====
:= TKNIN      "Tokenise" ASCII text.
=====
:JOB          To convert a source file of full ASCII text to a
:             destination file of ASCII characters and
:             control codes and "tokens" ($80 to $FF).
:ACTION       REPEAT:
:             [ Call token string match routine.
:             IF match found
:             THEN: [ Use match string index as token. ]
:             ELSE: [ Get source character.
:                   Point to next source byte. ]
:             Write character/token to destination.
:             Point to next destination byte. ]
:             UNTIL terminator addressed.
:             Write terminator to destination.
-----
:CPU          6502
:HARDWARE     Memory containing source text.
:             Destination RAM to contain tokenised text.
:SOFTWARE     "MATCH" - a routine to test for a match between
:             source file substrings and "token" strings,
:             returning token (table position) in X and
:             incrementing source pointer past matched string.
-----
:INPUT        M0,1 addresses start of ASCII source text.
:             The text must terminate with a null (0) byte.
:             M6,7 addresses start of destination area.
:OUTPUT       M0,1 addresses source null terminator.
:             Source text is unchanged.
:             M6,7 addresses destination null terminator.
:             Destination contains tokenised text.
:             All registers changed.
:ERRORS       Destination may overwrite source.
:REG USE      P A X Y
:STACK USE    2
:RAM USE      M0 M1 M6 M7
:LENGTH      37
:CYCLES       Not given.
-----
:CLASS 2      -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
=====
:
SRCI  =  M0      :2-byte stored source address.
DSTI  =  M6      :2-byte stored destination address.
:
TKNIN JSR MATCH  :Check for token string match. 20 Lo hi
      LDY #0      :For index of current bytes.   A0 00
      INX         :Test if token matched and     E8
      BNE TMATCH  :skip if so.                   D0 0A

```

DATA COMPRESSION

```

:
    LDA (SRCI),Y :Else, get source character      B1 M0
    INC SRCI      :byte and increment source      E6 M0
    BNE WRITEB    :pointer ready for next        D0 08
    INC SRCI+1    :terminator/token match test.  E6 M1
    BNE WRITEB    :Go write byte to destination. D0 04
:
TMATCH DEX          :Restore correct token table  CA
      TXA          :offset, into A and set bit 7  8A
      ORA #$80      :as non-ASCII token.         09 80
:
WRITEB STA (DSTI),Y :Write ASCII character/token  91 M6
      INC DSTI      :to destination and increment E6 M6
      BNE TERMCH    :pointer to next character,   D0 02
      INC DSTI+1    :with any carry to hi-byte.   E6 M7
:
TERMCH LDA (SRCI),Y :Check for null terminator   B1 M0
      BNE TKNIN     :and continue if not.        D0 DE
:
      STA (DSTI),Y :Else write destination       91 M6
      RTS          :terminator and exit.         60
:=====

```

```

=====
:= TKNOUT   Output expanded "Tokenised" text.
=====
:JOB        To convert a source file consisting of ASCII
:           characters and control codes and "tokens" ($80
:           to $FF) to full ASCII text by reference to a
:           token expansion table.
:ACTION     REPEAT:
:           [ Get character.
:             IF character NOT terminator THEN:
:             [ IF character is a token THEN:
:             [ Convert token to expansion index.
:               Index correct expansion.
:               Write expansion bytes - 1 to destination.
:               Add expansion length - 1
:               to destination pointer.
:               Get last expansion character. ]
:             Write character to destination.
:             Point to next destination byte.
:             Point to next source byte. ] ]
:           UNTIL terminator.
:           Write terminator to destination.
:-----
:CPU        6502
:HARDWARE    Memory containing source text.
:           Memory containing token expansion table.
:           Destination RAM to contain expanded text.
:SOFTWARE    None.
:-----

```

DATA COMPRESSION

```

:INPUT      M0,1 addresses start of destination area.
:           M6,7 addresses start of tokenised source text.
:           The text must terminate with a null (0) byte.
:           M4,5 addresses start of token expansion table.
:           Table length: up to 128 strings.
:           String length: up to 255 characters (bytes).
:           Character (byte) range: 0 to $7F.
:           String terminator: bit 7 set on last byte.
:OUTPUT     M0,1 addresses destination null terminator.
:           Destination contains fully expanded text.
:           M6,7 addresses source null terminator.
:           Source text is unchanged.
:           M4,5 not changed. M2,3 may be changed.
:           All registers changed.
:ERRORS     Arithmetic error if D = 1 (decimal mode).
:           Destination may overwrite source or table.
:REG USE     P A X Y
:STACK USE   None.
:RAM USE     M0 to M7
:LENGTH     88
:CYCLES      Not given.
:-----
:CLASS 2     -discreet *interruptable *promable
:-*****-    *reentrant *relocatable -robust
:=====
:
DSTO  =      M0           :2-byte stored destination address.
EXPO  =      M2           :2-byte for storing expansion address.
TAB   =      M4           :2-byte stored expansion table address.
SRCO  =      M6           :2-byte stored source address.
:
TKNOUT LDY #0           :Index and get currently           A0 00
      LDA (SRCO),Y       :addressed character,           B1 M6
      BNE NONULL         :process if not null            D0 01
      STA (DSTO),Y       :else write destination         91 M0
      RTS               :terminator and exit.           60
NONULL BPL WRITEA       :Skip to write if ASCII.         10 3C
:
      LDX TAB           :Else token, so move             A6 M4
      STX EXPO          :token table address             86 M2
      LDX TAB+1         :to expansion address            A6 M5
      STX EXPO+1        :variable.                      86 M3
      AND #$7F         :Clear token flag, bit 7,        29 7F
      TAX              :and move to X as expansion       AA
      BEQ TKNX         :index, skip if expansion 0.     F0 14
:
TXLP   LDY #-1         :Index from byte 0.              A0 FF
TSLP   INY            :Index next character,            C8
      LDA (EXPO),Y     :load to test it and repeat      B1 M2
      BPL TSLP         :until end of expansion.         10 FB
:
      TYA              :Add expansion end index +       98
      SEC              :set Carry, as expansion         38
      ADC EXPO         :byte length, to expansion      65 M2
      STA EXPO         :address, giving address        85 M2
      BCC TXLPT        :of 1st byte of next           90 02
      INC EXPO+1       :expansion in EXPO.             E6 M3
:

```

DATA COMPRESSION

```

TXLPT DEX          :Repeat until EXP0 addresses      CA
      BNE TXLP      :required expansion.             DO EC
:
TKNX  LDY #0        :Index current bytes.            AO 00
WRITEX LDA (EXP0),Y :Loop, moving characters          B1 M2
      BMI DFREE      :from expansion (if not last    30 05
      STA (DST0),Y   :byte) to destination.         91 M0
      INY            :As Y never reaches 256 (0)      C8
      BNE WRITEX     :always loop for next byte.     DO F5
:
DFREE TYA           :Y indexes next free byte        98
      CLC            :so, add without carry,         18
      ADC DST0       :add next free byte index       65 M0
      STA DST0       :to destination pointer         85 M0
      BCC LASTX      :giving address of next        90 02
      INC DST0+1     :free byte in DST0.             E6 M1
:
LASTX LDA (EXP0),Y :Get last expansion byte and      B1 M2
      AND #$7F       :clear end-of-string flag.     29 7F
      LDY #0         :Restore index to 0.            AO 00
:
WRITEA STA (DST0),Y :Write ASCII/last expansion      91 M0
      INC DST0       :byte and increment            E6 M0
      BNE SRCINC     :destination pointer ready     DO 02
      INC DST0+1     :for next character.           E6 M1
:
SRCINC INC SRC0      :Increment source pointer       E6 M6
      BNE CONTIN     :to next character,            DO 02
      INC SRC0+1     :with any carry to hi-byte.    E6 M7
CONTIN SEC          :Ensure branch and continue      38
      BCS TKNOUT     :process until terminator.     BO A8
:=====

```

ESCAPE MESSAGES

Similar in concept to TKNOUT but possibly more versatile since it is not limited to only 128 substrings is MAKMSG, a 6502 version of a *Sub Set* routine originally written in Z-80 code. Where a non-ASCII byte sent TKNOUT off on a search through its expansion table, the same phenomenon causes MAKMSG to read the next 2 bytes as the address of the substring to be inserted.

MAKMSG acts recursively so the substrings can be nested to any depth. The 'escapes' don't necessarily have to be to the start of a substring and, if required, may be made to the ending of the last word in a long string. However, since each escape consists of the escape code itself and the substring address (3 bytes in all), the law of diminishing returns soon comes into effect. It simply isn't worth using escapes for words like 'THE' and 'IS' or word endings like 'ING'.

DATA COMPRESSION

```

=====
:= MAKMSG    Make Message.
=====
:JOB        To output ASCII characters and control codes
:           ($01 to $7F) as normal but to regard non-ASCII
:           values ($80 to $FF) as "escape" codes followed
:           by the address of a sub-message, to be output
:           before continuation of the main message.
:ACTION      REPEAT:
:           [ Get character.
:             IF character NOT terminator THEN:
:             [ IF character is ASCII
:               THEN: [ Output character. ]
:               ELSE: [ Save pointer.
:                     Get escape address.
:                     Call MAKMSG.
:                     Restore pointer. ]
:             Point to next character. ] ]
:           UNTIL terminator.
-----
:CPU         6502
:HARDWARE    Memory containing message and sub-messages.
:SOFTWARE    Subroutine "OUTCH" to output an ASCII character
:           in A to screen, printer, etc., without changing
:           register and flag values.
-----
:INPUT       MO,1 addresses 1st byte of top level message.
:           Each message and sub-message must terminate with
:           a null (0) byte.
:           The absolute "escape" addresses must be written
:           with low order byte first.
:           Sub-message nesting can be to any depth.
:OUTPUT      MO,1 addresses top level message terminator.
:           A = 0. Y = 0. X and P are unknown
:ERRORS      Arithmetic error if D = 1 (decimal mode).
:           An "infinite loop" can occur if a low level
:           message "escapes" to one higher up the chain.
:REG USE     P A X Y
:STACK USE   4 * message nesting + 2 + OUTCH stack use.
:RAM USE     MO M1
:LENGTH     55
:CYCLES      Not given.
-----
:CLASS 2     -discreet *interruptable *promable
:***--      *reentrant -relocatable -robust
=====
:
MSG          =    MO           :Stored address of current message.
:
MAKMSG LDY    #0              :Index currently addressed           AO 00
      LDA     (MSG),Y         :byte, get in A and skip to       B1 MO
      BNE     NOTERM          :process unless null byte,        DO 01
      RTS                      :exit at end of current level.    60
:
NOTERM BMI    ESCP            :Skip if escape code. Else         30 07
      JSR     OUTCH           :go output ASCII character.       20 lo hi
      LDA     #1              :A = 1 for single character        A9 01
      BNE     MSGINC          :increment to message pointer.     DO 1C

```

DATA COMPRESSION

```

:
ESCP  LDA  MSG+1  :Save this level message      A5 M1
      PHA          :pointer to stack through A,    48
      LDA  MSG     :so escape address can go      A5 M0
      PHA          :in MSG for recursive call.    48
:
      INY          :Index and get escape address  C8
      LDA  (MSG),Y :low order byte saved in X      B1 M0
      TAX          :so MSG not yet changed.       AA
      INY          :Index and get escape address  C8
      LDA  (MSG),Y :high order byte in A.         B1 M0
      STA  MSG+1    :Write escape address to MSG   85 M1
      STX  MSG      :and call MAKMSG to process   86 M0
      JSR  MAKMSG   :escape message at new level. 20 lo hi
:
      PLA          :Escape message finished, so   68
      STA  MSG      :restore pointer to this level 85 M0
      PLA          :back to MSG, addressing       68
      STA  MSG+1    :escape byte, so A = 3 to     85 M1
      LDA  #3       :move it past escape address. A9 03
:
MSGINC CLC          :No carry in to addition.     18
      ADC  MSG      :Add 1 or 3 to message        65 M0
      STA  MSG      :pointer, moving it to       85 M0
      BCC  MAKMSG   :address next character,      90 CD
      INC  MSG+1    :repeat always, after taking E6 M1
      BCS  MAKMSG   :care of any carry to hi-byte. B0 C9
:=====

```

If you are really short of text storage space then try combining TKNOUT and MAKMSG: use \$81 to \$FE for 'tokens' and \$FF for 'escapes'. Meanwhile, here's an example of how MAKMSG can be used to produce new jargon phrases almost as fast as they develop naturally. The lefthand column shows the addresses where the text code is located, the middle column gives the text in hexadecimal and the righthand column is what MAKMSG makes of it.

1234:	20 49 4E 46 4F 52 4D 00	INFORM
123C:	FF 34 12 41 54 49 4F 4E 00	INFORMATION
1245:	20 54 45 43 48 4E 00	TECHN
124C:	FF 45 12 49 43 41 4C 00	TECHNICAL
1254:	FF 45 12 4F 4C 4F 47 00	TECHNOLOG
125C:	FF 54 12 FF 4F 12 00	TECHNOLOGICAL
1263:	FF 54 12 49 53 54 00	TECHNOLOGIST
126A:	20 44 45 54 41 49 4C 00	DETAIL
1272:	FF 3C 12 FF 54 12 59 00	INFORMATION TECHNOLOGY
127A:	FF 3C 12 FF 63 12 00	INFORMATION TECHNOLOGIST
1281:	FF 4C 12 FF 6A 12 00	TECHNICAL DETAIL
1288:	FF 6A 12 45 44 FF 3C 12 00	DETAILED INFORMATION
1291:	FF 5C 12 4C 59 2D	TECHNOLOGICALLY-
	FF 34 12 45 44 00	INFORMED

Data Moves

8

The first routine in this chapter is not strictly a data move routine. TEXT by Andrew Johnson deals with outputting a string of text embedded in the program. It is included here because it illustrates how to extract data buried inside a program. TEXT can be adapted to put the extracted data straight to display RAM, into a variable storage area, or anywhere else you can think of.

TEXT is ideal for printing out short messages to the user, such as input prompts or error reports. And having the messages actually in the section which generates them can make an assembly language program more readable. On the negative side, disassembling machine code that contains embedded data is not easy. You could make use of the idea as a simple form of 'program protection' but don't lose your source program or you are in for a very frustrating time.

```
=====
:= TEXT      Output program embedded text.
=====
:JOB         To output a program embedded text string.
:ACTION      Pull return address as pointer to text.
:            Ensure addressing from text byte 0.
:            REPEAT:
:            [ Increment text pointer.
:            Get text character.
:            IF character NOT terminator THEN:
:            [ Output character. ] ]
:            UNTIL terminator.
:            Push text pointer as return address.
-----
:CPU         6502
:HARDWARE    None.
:SOFTWARE    "OUTPUT" - A routine to output character in A,
:            should not change registers, must not change Y.
-----
:INPUT       Text to be output must follow "JSR TEXT".
:            Text length: indefinite.
:            Information byte/s: None.
:            Text character range: $01 to $FF.
:            Terminator: null (0) byte following text.
:OUTPUT      Return to location following terminator.
:            M0,1 contains address of text terminator.
:            Y = 0. P and A are changed.
```

DATA MOVES

```

:ERRORS      None.
:REG USE     P A Y
:STACK USE   "OUTPUT" stack use.
:RAM USE     M0 M1
:LENGTH     31
:CYCLES      45 + text characters * (26 + "OUTPUT" cycles).
:-----
:CLASS 2     ~discreet *interruptable *promable
:--*****   *reentrant *relocatable *robust
:=====
:
PTR      =      M0      :2-byte store for return address.
:
TEXT     PLA          :Pull return address from      68
        STA PTR      :stack top and store in      85 M0
        PLA          :page zero for use as pointer  68
        STA PTR+1     :to embedded text.           85 M1
        LDY #0        :Index zeroth byte throughout. A0 00
:
TEXTLP   INC PTR      :Move pointer to address next  E6 M0
        BNE READCH    :text character, with any    D0 02
        INC PTR+1     :carry to pointer hi-byte.    E6 M1
:
READCH   LDA (PTR),Y   :Get current character but    B1 M0
        BEQ FINISH    :end if null terminator.     F0 06
:
        JSR OUTPUT    :Else output character, then  20 lo hi
        TYA          :ensure branch occurs (Y = 0)  98
        BEQ TEXTLP    :and go get next character.  F0 F0
:
FINISH   LDA PTR+1     :At end, pointer addresses    A5 M1
        PHA          :terminator, i.e. return      48
        LDA PTR      :address - 1, so will be      A5 M0
        PHA          :correct return address on    48
        RTS          :stack top for RTS exit.      60
:=====

```

INTELLIGENT TRANSFER

In a non-intelligent transfer routine which always starts at the lowest address, source data will be overwritten before it is moved if the destination start address is within the source block.

IBT by Alex Selby (an improvement to an original 6502 *Sub Set* routine, BLKMV) ensures against this possibility by transferring data from the highest address downwards when the destination is higher than the source. If the source is the higher of the two then the move starts at the lowest address. The only time this arrangement doesn't work is when 16-bit 'wraparound' addressing is used and \$0000 is assumed to follow on from \$FFFF.

```

=====
:= IBT      Intelligent Block Transfer.
=====
:JOB        To transfer a block of data so that the moved
:           data does not overwrite the yet uncopied source.
:ACTION     IF destination < source
:           THEN:
:           [ Address first byte of source and destination.
:             REPEAT:
:             [ Move byte from source to destination.
:               Increment source and destination pointers. ]
:             UNTIL transfer completed. ]
:           ELSE:
:           [ Address last byte of source and destination.
:             REPEAT:
:             [ Decrement source and destination pointers.
:               Move byte from source to destination. ]
:             UNTIL transfer completed. ]
=====
:CPU        6502
:HARDWARE    Source and destination RAM.
:SOFTWARE    None.
=====
:INPUT       M0,1 addresses first byte of source.
:           M2,3 addresses first byte of destination.
:           M4,5 = byte length of data block.
:OUTPUT      Block at destination.
:           Pointer hi-bytes, M1 and M3, are changed.
:           Source block may be overwritten.
:           No registers are changed.
:ERRORS      Source data could be overwritten prior to the
:           transfer if 16-bit "wraparound" addressing is
:           used (i.e. $0000 follows $FFFF).
:REG USE     None.
:STACK USE   4
:RAM USE     M0 to M5
:LENGTH      83
:CYCLES       Not given.
=====
:CLASS 2     -discreet *interruptable *promable
:-----     *reentrant -relocatable -robust
=====
:
SRCE  =  M0      :Stored source start address.
DEST  =  M2      :Stored destination start address.
LEN   =  M4      :Stored byte-length of source block.
:
IBT   PHP        :Save                      08
      PHA        :all                      48
      TXA        :registers                 8A
      PHA        :on                      48
      TYA        :stack                    98
      PHA        :and ensure               48
      CLD        :binary arithmetic.       D8
      LDX  LEN+1  :Get 256-byte block count. A6 M5
:

```

DATA MOVES

:....Test relative positions of source and destination.

SEC	:	No borrow going into	38
LDA SRCE	:	source - destination	A5 M0
SBC DEST	:	comparison.	E5 M2
LDA SRCE+1	:		A5 M1
SBC DEST+1	:	If source below dest	E5 M3
BCC RVRS	:	then move from end down.	90 18

:

:....Initialise index and count for 1st to last byte move.

LDY #0	:	Index from lowest byte.	A0 00
INX	:	Correct for DEC end check.	E8

:

:....1st to last byte move in 256-byte blocks.

FWDLP CPY LEN	:	Test possible end of	C4 M4
BNE FWDTR	:	block, skip if not there.	D0 03
DEX	:	Else dec length hi-byte	CA
BEQ EXIT	:	and end if block end.	F0 2E

:

FWDTR LDA (SRCE),Y	:	Get source byte into	B1 M0
STA (DEST),Y	:	destination and move index	91 M2
INY	:	to next byte, looping till	C8
BNE FWDLP	:	end of 256-byte index	D0 F2
INC SRCE+1	:	then inc base hi-bytes	E6 M1
INC DEST+1	:	for next 256-byte block	E6 M3
JMP FWDLP	:	and repeat.	4C lo hi

:

:....Note that carry flag is clear on branch to RVRS.

:....Move base addresses to last page of block and

:....initialise index and count for last to 1st byte move.

RVRS LDY LEN	:	Index from highest byte.	A4 M4
TXA	:	Add hi-byte of byte length	8A
ADC SRCE+1	:	to source and destination	65 M1
STA SRCE+1	:	base addresses so they	85 M1
TXA	:	index the highest page of	8A
CLC	:	the data blocks.	18
ADC DEST+1	:		65 M3
STA DEST+1	:		85 M3
INX	:	Correct for DEC end check.	E8

:

:....Last to 1st byte move in 256-byte blocks.

RVRS LP TYA	:	Test for end of index	98
BEQ HIDE C	:	256-byte block, skip if so.	F0 08

:

RVTR DEY	:	Pre-dec index and move one	88
LDA (SRCE),Y	:	byte from source to	B1 M0
STA (DEST),Y	:	destination and	91 M2
JMP RVRS LP	:	repeat till 1st byte done.	4C lo hi

:

HIDE C DEC SRCE+1	:	End of 256-byte block so	C6 M1
DEC DEST+1	:	dec base address hi-bytes to	C6 M3
DEX	:	index next lower block and	CA
BNE RVTR	:	repeat if count not done.	D0 F1

:

EXIT	PLA	:Restore	68
	TAY	:all	A8
	PLA	:registers	68
	TAX	:from	AA
	PLA	:stack	68
	PLP	:and	28
	RTS	:exit, transfer done.	60
:=====			

MEMORY BLOCK ROTATION

RRL by R. G. Cath performs a long rotation on a data block. You must input the address of the lowest byte in the block, the address of the byte immediately following the block and the byte distance each byte in the block has to be rotated. No memory outside the block is affected by the routine since any hop beyond the end of the block is trapped and wrapped around to become a hop into the lower end of the block.

RRL could be a useful subroutine for many programs – an insertion sort of multi-byte elements and a word processor are just two that spring to mind.

```

:=====
:= RRL      Rotate Right Long.
:=====
:JOB        To rotate a block of memory, each byte moving a
:           given distance upwards in memory, with all
:           distances past the end of the block recalculated
:           from the start of the block.
:ACTION     ON ERROR: [ Exit, overflow flag set. ]
:           Pointer = block-start. Wraparounds = 0.
:           WHILE Wraparounds NOT EQUAL TO movelength:
:           [ Cycle-start = pointer.
:             Move indexed byte to temp-store.
:             REPEAT:
:             [ REPEAT:
:               [ Exchange indexed byte with temp-store.
:                 Pointer = pointer + movelength. ]
:               UNTIL Pointer past block-end.
:               Wraparounds = wraparounds + 1.
:               Pointer = pointer - (block-end + 1).
:               Pointer = pointer + block-start. ]
:             UNTIL Pointer = Cycle-start.
:             Move temp-store to indexed byte.
:             Pointer = pointer + 1. ]
:           Set rotation completed flag.
:-----
:CPU        6502
:HARDWARE   RAM containing block to rotate.
:SOFTWARE   None.
:-----

```

DATA MOVES

```
:INPUT      M0,1 addresses 1st byte in block.
:           M2,3 addresses byte following block.
:           M4,5 is the byte distance to rotate each byte.
:           (M2,3) + (M4,5) - 1 should not exceed $FFFF.
:OUTPUT     Registers and M6 to MB changed.
:           M0 to M5 not changed.
:           C = 0: Invalid input, arithmetic overflow.
:           Block may be partly rotated.
:           C = 1: Rotation completed.
:ERRORS     Arithmetic error if D = 1 (decimal mode).
:REG USE    P A X Y
:STACK USE  2
:RAM USE    M0 to MB
:LENGTH    121
:CYCLES     Not given.
```

```
:-----
:CLASS 2    -discreet *interruptable *promable
:-----
:*****-    *reentrant *relocatable -robust
:=====
```

```
:
BST      =   M0      :2-byte stored block start address.
BND      =   M2      :2-byte stored block end address + 1.
BSD      =   M4      :2-byte stored byte distance to move.
WRP      =   M6      :2-byte store for wraparound counter.
PTR      =   M8      :2-byte store for current pointer.
CST      =   MA      :2-byte store for cycle start address.
:
RRL      LDY #0      :Zero index throughout.                A0 00
          STY WRP     :Set wraparound count to                84 M6
          STY WRP+1   :zero initially.                        84 M7
          LDA BST     :Set pointer to                          A5 M0
          STA PTR     :start of block                          85 M8
          LDA BST+1   :initially.                             A5 M1
          STA PTR+1   :                                       85 M9
:
BCYCLE   LDA BSD     :Cycle start.                            A5 M4
          CMP WRP     :Compare number of                      C5 M6
          BNE BSTART  :wraparounds with distance             D0 08
          LDA BSD+1   :to move each byte.                     A5 M5
          BMI EXITNV  :Exit, error, if move negative.        30 5D
          CMP WRP+1   :Rotation done if wraparounds           C5 M7
          BEQ EXITRD  := distance, else continue.             F0 5B
:
BSTART   LDA PTR     :Save pointer address at start           A5 M8
          STA CST     :of cycle for test when cycle           85 MA
          LDA PTR+1   :ends (i.e. when wrapped around         A5 M9
          STA CST+1   :pointer is back at start).             85 MB
          LDA (PTR),Y :Initially move 1st byte to             B1 M8
          PHA         :stack, ready for exchanges.            48
:
NXTHOP   LDA (PTR),Y :Exchange indexed                        B1 M8
          TAX         :byte with that                          AA
          PLA         :from last position                      68
          STA (PTR),Y :stored                                  91 M8
          TXA         :temporarily                              8A
          PHA         :on stack top.                           48
:
```

DATA MOVES

	CLC	:Prepare to add, no carry in.	18
	LDA PTR	:Add shift distance to	A5 M8
	ADC BSD	:current pointer, moving it	65 M4
	STA PTR	:to address byte at	85 M8
	LDA PTR+1	:destination for byte now	A5 M9
	ADC BSD+1	:stored on stack.	65 M5
	STA PTR+1	:But exit in error if "hopping"	85 M9
	BCS EXITPL	:gone past 64K memory.	B0 36
:			
	NXTPAS SEC	:Prepare to subtract, no borrow.	38
	LDA PTR	:Test relative position of	A5 M8
	SBC BND	:current pointer and block end,	E5 M2
	TAX	:setting Carry if pointer lower,	AA
	LDA PTR+1	:and temporarily retaining	A5 M9
	SBC BND+1	:difference in AX. If pointer	E5 M3
	BCC NXTHOP	:lower then continue "hopping".	90 DD
:			
	INC WRP	:Else gone past block end, and	E6 M6
	BNE PRESET	:AX is overshoot. So increment	D0 02
	INC WRP+1	:wraparound counter; then reset.	E6 M7
:			
	PRESET PHA	:Save overshoot hi-byte	48
	TXA	:while processing lo-byte.	8A
	CLC	:Prepare to add, no carry in.	18
	ADC BST	:Add block start to overshoot	65 M0
	STA PTR	:resetting pointer within block.	85 M8
	PLA	:Restore overshoot hi-byte and	68
	ADC BST+1	:add block start hi-byte. Okay	65 M1
	STA PTR+1	:if pointer in block but exit as	85 M9
	BCS EXITPL	:error if past 64K memory.	B0 16
:			
	CMP CST	:At end of pass, when wraparound	C5 MB
	BNE NXTPAS	:has occurred, test if at start	D0 DC
	LDA PTR	:location. If so then cycle has	A5 M8
	CMP CST	:ended but if not then another	C5 MA
	BNE NXTPAS	:pass needed.	D0 D6
:			
	PLA	:At end of cycle, when pointer	68
	STA (PTR),Y	:addresses start location, move	91 M8
	INC PTR	:byte from stack to replace 1st	E6 M8
	BNE BCYCLE	:byte moved this cycle.	D0 9F
	INC PTR+1	:Move pointer to next location	E6 M9
	CLV	:and go test if another cycle	B8
	BVC BCYCLE	:is needed.	50 9A
:			
	EXITPL PLA	:Tidy up stack before exit.	68
	EXITNV CLC	:Reset Carry to show rotation	18
	RTS	:invalid/incomplete, exit.	60
:			
	EXITRD SEC	:Set Carry to show rotation	38
	RTS	:completed successfully, exit.	60
:			
	=====		

MATRIX TRANSPOSITION

There are two ways that you can store a two dimensional array, or matrix, of single-byte elements in linearly addressed memory. Either the bytes in each row are stored contiguously or else each successive location holds the next byte down the column. For example, the matrix:

A	B	C	D
E	F	G	H
I	J	K	L

can be stored as 'A B C D E F G H I J K L' (sequential row storage) or as 'A E I B F J C G K D H L' (sequential column storage).

It doesn't usually matter which way matrices are stored since any byte can be accessed fairly easily by using a simple multiplication for one dimension and an addition for the other. Sometimes, however, it does matter very much because multiplication on the 6502 is not a particularly fast operation. Occasionally, especially in graphics applications, a matrix has to be ordered in both ways for quicker access.

TRANS by Vernon Webb will turn, or transpose, a matrix from one form of storage to the other. The most interesting feature is that the matrix is transposed in its own space, without recourse to any other storage area except a few bytes of page zero used for variables. This eliminates any problems that can result from working in a small amount of RAM but the method of TRANS is very slow – for large arrays at least. For a matrix composed of R rows by C columns, the number of elements that have to be moved is,

$$(RC + 4) * (R - 1) * (C - 1) / 4.$$

This means that only 5 bytes have to be moved in a 2 by 3 matrix but the number increases rapidly. 105 elements have to be moved in a 4 by 6 matrix (shown below) and the massive number of 14,625 bytes have to be shuffled around to transpose a 16 by 16 matrix – an average of just over 57 moves for each byte. Perhaps it is just as well that TRANS doesn't attempt to transpose matrices of more than 256 elements!

The method of TRANS is really quite simple. It rotates a block of elements extending from the position where the next column-sequential byte is to go to that byte's current, row-sequential position. The byte is put in its rightful place, the bytes below it being shuffled up to make room – rather like the method used in insertion sorts.

This is demonstrated below on a small 4 by 6 matrix. The top line shows the input row-sequential storage and the bottom line is the column-sequential transposed result. The intervening 15 lines show all intermediate states with the block from 'ED' to 'ES' (about to be rotated) highlighted as upper-case letters.

AS ROWS: < 1 > < 2 > < 3 > < 4 >
a b c d e f g h i j k l m n o p q r s t u v w x

```

CI RI ED ES
0 1 1 6  a B C D E F G h i j k l m n o p q r s t u v w x
0 2 2 12 a g B C D E F H I J K L M n o p q r s t u v w x
0 3 3 18 a g m B C D E F H I J K L N O P Q R S t u v w x
1 1 5 9  a g m s b C D E F H i j k l n o p q r t u v w x
1 2 6 14 a g m s b h C D E F I J K L N o p q r t u v w x
1 3 7 19 a g m s b h n C D E F I J K L O P Q R T u v w x
2 1 9 12 a g m s b h n t c D E F I j k l o p q r u v w x
2 2 10 16 a g m s b h n t c i D E F J K L O p q r u v w x
2 3 11 20 a g m s b h n t c i o D E F J K L P Q R U v w x
3 1 13 15 a g m s b h n t c i o u d E F J k l p q r v w x
3 2 14 18 a g m s b h n t c i o u d j E F K L P q r v w x
3 3 15 21 a g m s b h n t c i o u d j p E F K L Q R V w x
4 1 17 18 a g m s b h n t c i o u d j p v e F K l q r w x
4 2 18 20 a g m s b h n t c i o u d j p v e k F L Q r w x
4 3 19 22 a g m s b h n t c i o u d j p v e k q F L R W x

```

AS COLUMNS: < 1 > < 2 > < 3 > < 4 > < 5 > < 6 >
a g m s b h n t c i o u d j p v e k q w f l r x

```

=====
:= TRANS      Own-space matrix transposition.
=====
:JOB          To transpose a 2-dimensional array, or matrix,
:             in its own RAM space, storage row-after-row
:             becoming column-after-column.
:ACTION       FOR col-index = 0 to cols - 2:
:             [ FOR row-index = 1 to rows - 1:
:             [ Source = Matrix-start +
:             (col-index * rows) +
:             (row-index * cols) -
:             (row-index * col-index).
:             Dest = Matrix-start + row-index +
:             (col-index * rows).
:             (Temp) = (source).
:             (Dest + 1, source) = (dest, source - 1).
:             (Dest) = (temp). ] ]
:-----
:CPU          6502
:HARDWARE     RAM containing matrix.
:SOFTWARE     None.
:-----
:INPUT        M0,1 addresses first byte of matrix.
:             M2 contains number of rows in matrix.
:             M3 contains number of columns in matrix.
:             Maximum matrix elements: 256.

```

DATA MOVES

```
:OUTPUT      M0 to M3 are unchanged. M4 to M8 changed.
:            All registers unchanged.
:ERRORS      Arithmetic error if D = 1 (decimal mode).
:            Incorrect transposition if M2 * M3 > 256.
:REG USE     None.
:STACK USE   4
:RAM USE     M0 to M8.
:LENGTH     102
:CYCLES      Not given.
```

```
-----
:CLASS 2     -discreet *interruptable *promable
:****--     *reentrant -relocatable -robust
=====
```

```
:
MTX  =  M0      :2-byte stored Matrix start address.
ROWS =  M2      :1-byte stored number of rows.
COLS =  M3      :1-byte stored number of columns.
CI   =  M4      :1-byte store for column index.
CR   =  M5      :1-byte store for CI * ROWS by adding.
RI   =  M6      :1-byte store for row index.
ES   =  M7      :1-byte store for element source index.
ED   =  M8      :1-byte store for element dest. index.
:
TRANS PHP      :Save                      08
      PHA      :flags                    48
      TYA      :and                      98
      PHA      :registers                48
      TXA      :for use                  8A
      PHA      :in TRANS.               48
:
      LDA #-1   :Ensure column index is 0   A9 FF
      STA CI    :when first INC'd in NEWCOL. 85 M4
      SEC      :So no borrow in to subtract. 38
      LDA #0    :Ensure CI*ROWS variable is 0 A9 00
      SBC ROWS  :when first incremented by  E5 M2
      STA CR    :ROWS in NEWCOL.           85 M5
:
NEWCOL INC CI   :Index next column, first row. E6 M4
      CLC      :Ensure no carry in to add.  18
      LDA CR    :Compute offset of first row A5 M5
      ADC ROWS  :of next column from MTX,   65 M2
      STA CR    :(i.e. column index * ROWS). 85 M5
      STA ED    :Also destination offset.   85 M8
      LDA #0    :Ensure row index is 1 when  A9 00
      STA RI    :first INC'd in NEWROW.     85 M6
:
NEWROW INC RI   :Index next row, this column. E6 M6
      LDA CR    :Do: ES=CR-(RI*CI)+(RI*COLS), A5 M5
      SEC      :Prepare for A-(RI*CI) by    38
      LDX CI    :repeated subtraction.     A6 M4
:
SUBLP BEQ SUBDON :Skip out when CI loops done. FD 06
      SBC RI    :Subtract RI once for      E5 M6
      DEX      :every CI.                  CA
      JMP SUBLP :Loop to exit on CI done test. 4C lo hi
:
SUBDON CLC      :Prepare for A-(RI*COLS) by  18
      LDX RI    :repeated addition.        A6 M6
```

DATA MOVES

```

:
ADDLP BEQ ADDDON :Skip out when RI loops done.      F0 06
      ADC COLS   :Add COLS once for                  65 M3
      DEX        :every RI.                          CA
      JMP ADDLP  :Loop to exit on RI done test.      4C lo hi
:
ADDON STA ES     :New source index. Next dest       85 M7
      INC ED     :1 up on last positioned byte.     E6 M8
      TAY        :Index and get source element      A8
      LDA (MTX),Y :save in X for destination        B1 M0
      TAX        :store after rest shifted up.      AA
:
SHFTLP DEY       :Shift up block below              88
      LDA (MTX),Y :current source by one byte,      B1 M0
      INY        :ED to ES-1 going into            C8
      STA (MTX),Y :ED+1 to ES, leaving ED free      91 M0
      DEY        :to receive current element.      88
      CPY ED     :Repeat until destination          C4 M8
      BNE SHFTLP :byte is indexed by Y.            D0 F5
:
      TXA        :Position current element          8A
      STA (MTX),Y :at destination.                  91 M0
:
      LDX ROWS   :Test for complete positioning     A6 M2
      DEX        :of currently indexed column      CA
      CPX RI     :when rows indexed 1 to last      E4 M6
      BNE NEWROW :have been processed.             D0 C9
:
      LDX COLS   :Test for end of transposition     A6 M3
      DEX        :when columns indexed 0 to        CA
      DEX        :last - 1 have been processed     CA
      CPX CI     :(last column automatically      E4 M4
      BNE NEWCOL :okay when others okay).          D0 B2
:
      PLA        :Restore                          68
      TAX        :registers                        AA
      PLA        :and                             68
      TAY        :flags                            A8
      PLA        :used in                          68
      PLP        :TRANS.                           28
      RTS        :Exit, matrix transposed.         60
=====

```


Reducing Errors

There is an ancient (pre-decimalisation) story of how the urgent message, 'Send reinforcements, we are going to advance.' was passed by word of mouth along the trenches. When the message reached Battalion HQ, the top brass were somewhat taken aback by the request, 'Lend three and fourpence, we are going to a dance.'

I don't know if the advance (of three shillings and four pennies) was made but the story does illustrate vividly the corruption that can occur to 'soft' data.

Errors in data may have several causes. The most common ones are a slip of the finger during input and corruption during transmission or magnetic storage. Such errors may be no more than a time consuming frustration to the computer hobbyist but in the business world they can be very costly.

Many methods have been developed to ensure the validity and integrity of important data. Some merely indicate that an error has occurred whilst others can identify the errors and correct them. None can guarantee 100% accuracy.

CHECKSUMS

The checksum technique is mainly used to detect mistakes made when typing in numbers. An extra digit (or even a letter) called the check digit is calculated from the sum of all the digits in the number and appended to it. Thereafter, whenever the number is typed in, the checksum digit is recalculated and compared to the appended check digit. If the two match then the actual number is assumed to be valid. Any disparity means that an error has been made and the number must be re-entered.

Simply adding the digits in the number string is not the best method of calculating the check digit as it allows too many errors to pass unnoticed. For example, 402173Q with 4 and 2 transposed on input would be erroneously accepted as 204173Q. The digits of both numbers sum to 17:

To guard against transposition errors, most checksums assign a unique weighting to each digit position. Lance A. Leventhal in his book *'6502 Assembly Language Programming'* describes a method

REDUCING ERRORS

known as 'Aligned 1, 3, 7 Mod 10' and sets it as an exercise. Calculating the checksums of 402173 and 204173 gives the following results.

Checksum :	$1*4 + 3*0 + 7*2 + 1*1 + 3*7 + 7*3 = 61.$
Checksum digit :	$(61 \text{ Mod } 10) = 1.$
Checksum :	$1*2 + 3*0 + 7*4 + 1*1 + 3*7 + 7*3 = 73.$
Checksum digit :	$(73 \text{ Mod } 10) = 3.$

Transposition of 4 and 2 when inputting 4021731 (the number 402173 with appended check digit 1) would be detected but the method is not foolproof. Other errors, such as transposition of 0 and 7 in the same number, would not be found. There are checksum weighting sequences that are more successful in trapping errors.

CHKSUM is my solution to the 'Aligned 1, 3, 7 Mod 10' problem set by Leventhal. It appeared in *Sub Set* as CADIDS in a double-datasheet containing both Z-80 and 6502 coding of the same method.

```
=====
:= CHKSUM   BCD Checksum.
=====
:JOB        To calculate the checksum digit of a BCD string
:           using the 'Aligned 1, 3, 7 Mod 10' method.
:ACTION     Clear checksum, (CHK). Clear weighting, (WGT).
:           FOR each byte:
:           [ FOR each digit:
:           [ IF WGT = 7 THEN: [ Clear WGT. ]
:           WGT = WGT * 2 + 1
:           FOR WGT: [ CHK = CHK + digit ] ] ]
:           CHK = CHK Mod 10.
:-----
:CPU        6502
:HARDWARE   Memory containing BCD string.
:SOFTWARE   None.
:-----
:INPUT      M0,1 addresses BCD string first (hi-order) byte.
:           M2 = Number of bytes in string.
:           (A leading or trailing zero must be used for a
:           string with an odd number of digits.)
:OUTPUT     M0,1 Addresses string + 1. M2 = Checksum digit.
:           Y = 0. X = 4. A and P are unchanged.
:ERRORS     No check made for valid BCD digits in number.
:REG USE    X Y
:STACK USE  5
:RAM USE    M0 M1 M2
:LENGTH     77
:CYCLES     134 + average of 83 per digit.
:-----
:CLASS 2    -discreet *interruptable *promable
:-----
:*****    *reentrant *relocatable -robust
:=====
```

REDUCING ERRORS

```

:
NUM      =      M0      :2-byte stored address of number.
CHK      =      M2      :Number byte-length (input),
                        :check digit (output).
CNT      =      M3      :For number byte count.
WGT      =      M4      :For computed digit weighting.
DGT      =      M5      :For storing each successive digit.
:
CHKSUM PHP                :Save flags and set index for      08
      LDX #4              :loop which saves 4 bytes...      A2 04
PUSHLP  PHA                :Save A first push, then          48
      LDA CHK-1,X         :M5, M4 and M3.                    B5 M1
      DEX                 :Leave M2 (byte length) in A        CA
      BNE PUSHLP          :and X = 0.                         D0 FA
:
      STA CNT              :Set byte count = byte length.     85 M3
      TXA                 :Initially clear checkdigit,        8A
      TAY                 :zeroise NUM index and              A8
      STA WGT              :initialise weighting to 0.         85 M4
      SED                 :Ensure decimal arithmetic.         F8
:
NXTBYT LDX #2              :Count 2 digits per byte.          A2 02
NXTDIG STA CHK             :Store partial check digit.         85 M2
:
      LDA (NUM),Y          :Get currently addressed byte      B1 M0
      CPX #2               :and test if currently on low      E0 02
      BNE NEWDIG           :order digit (skip it's okay)      D0 04
      LSR A                :else on high order digit          4A
      LSR A                :which has to be moved              4A
      LSR A                :down to low order                  4A
      LSR A                :before it can be added.            4A
NEWDIG STA DGT             :Store digit to page 0 for adds.    85 M5
:
      LDA WGT              :Get last weighting and            A5 M4
      CMP #7               :test if at top limit,              C9 07
      BNE NEWWGT           :skipping if not,                  D0 01
      TYA                  :else reset to 0 (Y = 0).            98
NEWWGT SEC                 :Rotate left with Carry set so      38
      ROL A                :weighting = weighting * 2 + 1.     2A
      STA WGT              :Store new weighting.              85 M4
:
      TAY                  :Weighting is add loop count.       A8
      LDA CHK              :Get partial check digit and,       A5 M2
ADDWD  CLC                 :ensure no carry to mess it up,      18
      ADC DGT              :add digit * weighting by           65 M5
      DEY                  :repeated decimal addition.         88
      BNE ADDWD            :Leave Y = 0 on exit from loop.      D0 FA
:
      DEX                  :Repeat for                          CA
      BNE NXTDIG           :two digits per byte.              D0 DB
:
      INC NUM              :Address next byte of               E6 M0
      BNE NBTEST           :number string, take care of        D0 02
      INC NUM+1            :any carry to hi-byte.              E6 M1
NBTEST DEC CNT             :Repeat for all bytes in             C6 M3
      BNE NXTBYT          :number string.                      D0 CF
:

```

REDUCING ERRORS

AND	#\$0F	:Mod 10. A = check digit.	29 0F	
PULLLP	STA	CHK,X	:Loop, storing check digit first	95 M2
PLA			:then, pulling and restoring	68
INX			:M3, M4 and M5, then	E8
CPX	#4		:finally restoring A from stack	E0 04
BNE	PULLLP		:on last iteration.	D0 F8
PLP			:Restore flags	28
RTS			:and exit, check digit found.	60

:-----

PARITY CODING

ECAL and EFIX by John Kerr use a form of parity coding based on the position of each bit in a block of data. Hence, they can be used not only to detect a bit inversion that may have occurred in the block but also to locate and correct it.

Before storage or transmission, an error correction byte (ecb) is calculated for every 31 bytes of data and appended to make a neat 32-byte block. On retrieval or receipt of the data, a new ecb is calculated and exclusive-ORed with the appended ecb. If all bits in the resultant 'correction code' are zero then the data can be assumed valid. If not then any value above 7 in the correction code gives the position of a single bit in the data block. EFIX assumes this bit has been inverted and corrects the data by re-inverting the bit.

The method can only cope with one corrupt bit in each data block. More than one inversion could together have no effect on the parity coding of the ecb or even fool EFIX into actually changing the state of a valid bit. And valid data could be 'corrected' by EFIX were a bit inversion to occur in the ecb itself. However, according to John, ECAL and EFIX can correct about 95% of errors in a system where the probability of bit error is less than 0.4%.

```
:-----
:= ECAL      Calculate error correction byte.
:-----
:JOB         To calculate a single byte parity code, capable
:            of being used to detect and correct a single bit
:            error in a 1 to 31 byte data block.
:ACTION      IF block byte length > 0 AND < 32 THEN:
:            [ Parity mask bit-count = bytes * 8 + 7.
:              Clear error correction byte (ecb).
:              Index first data byte.
:              WHILE bit-count > 7:
:              [ Get current data byte.
:                FOR bits 7 to 0 of data byte:
:                [ IF current bit = 1 THEN:
:                  [ ecb = ecb EOR bit-count. ]
:                  Bit-count = bit-count - 1. ]
:                Index next data byte. ] ]
:-----
```

REDUCING ERRORS

```

:CPU          6502
:HARDWARE     Memory containing data block.
:SOFTWARE     None.
-----
:INPUT        MO,1 addresses first byte of the data block.
:             Y = no. of bytes in data block (max. 31).
:OUTPUT       A, Y and MO,1 are unchanged.
:             C = 1: aborted (Y = 0 or Y > 31). X unchanged.
:             C = 0: X contains error correction byte (ecb).
:ERRORS       None.
:REG USE      P X Y
:STACK USE    3
:RAM USE      MO M1
:LENGTH      60
:CYCLES       63 + average 175 per byte.
-----
:CLASS 1      *discreet *interruptable *promable
:*****       *reentrant *relocatable *robust
=====
:
DATA  =  MO      :2-byte stored address of data block.
BCNT  =  M2      :For storing parity mask bit-count.
BYTE  =  M3      :For storing current data byte.
:
ECAL  CPY #32     :Abort if data block length          C0 20
      BCS ENDRTS  :is greater than 31.                 B0 37
:
      PHA         :Else, save A                        48
      LDA BCNT    :and M2                              A5 M2
      PHA         :for use in ECAL.                    48
      TYA         :Get block length in A, testing      98
      SEC         :if zero, set abort flag and         38
      BEQ ENDPLL  :abort if length is zero.            F0 2B
:
      ASL A       :Else okay, so multiply by 8          0A
      ASL A       :to convert byte length to           0A
      ASL A       :bit length and add 7 to form         0A
      ORA #7      :parity mask bit-count in M2          09 07
      STA BCNT    :(:0 means bit 0 of block + 1).      85 M2
:
      LDA BYTE    :Save M3 for use as store for        A5 M3
      PHA         :process of current byte.            48
      LDX #0      :Clear error correction byte.        A2 00
      LDY #0      :Index data from first byte.        A0 00
:
BYTELP LDA (DATA),Y :Get currently indexed byte to      B1 M0
      STA BYTE    :page zero for processing.           85 M3
      TXA         :Move ecb to A and set up count      8A
      LDX #8      :of 8 in X for bits in this byte.   A2 08
:
BITLP  ASL BYTE    :Move next data bit to Carry.       06 M3
      BCC NXTBIT  :"ecb EOR 0" if bit reset, else     90 02
      EOR BCNT    :"ecb EOR bit-count".               45 M2
NXTBIT DEC BCNT    :Prepare bit-count for next bit.    C6 M2
      DEX         :Mark off one bit processed         CA
      BNE BITLP   :and repeat for all 8 this byte.    D0 F5
:

```

REDUCING ERRORS

```

      INY           :Index next data byte.           C8
      LDX BCNT      :Test for block end, resetting   A6 M2
      CPX #8        :Carry when bit-count = 7.      E0 08
      TAX           :Move ecb to X and repeat       AA
      BCS BYTELP    :for all data block.           B0 E6
:
      PLA           :Restore M3                     68
      STA BYTE      :from stack.                   85 M3
:
ENDPLL PLA          :Restore M2 and A either         68
      STA BCNT      :after ecb found (C = 0) or     85 M2
      PLA           :on zero length (C = 1).        68
ENDRTS RTS          :Exit ECAL.                     60
:=====

```

```

:=====
:= EFIX      Validate data with error correction byte.
:=====
:JOB         To examine a 1 to 31 byte data block with
:            appended error correction byte, correcting any
:            single bit inversion indicated.
:ACTION      IF block byte length > 0 AND < 32 THEN:
:            [ Calculate new error correction byte (ecb).
:              Correction code = new ecb EOR appended ecb.
:              IF set bit(s) in correction code THEN:
:                [ IF correction code addresses data bit THEN:
:                  [ Correct bit error by re-inversion. ] ] ]
:-----
:CPU         6502
:HARDWARE    Memory containing data block.
:SOFTWARE    "ECAL" - routine to calculate ecb.
:-----
:INPUT       M0,1 addresses first byte of the data block.
:            Y = no. of data bytes in block (max. 31).
:            (Y indexes appended error correction byte).
:OUTPUT      M0,1 is unchanged.
:            C = 1: aborted (Y = 0 or Y > 31).
:            C = 0: Data assumed valid.
:            Error not found: Y is unchanged.
:            Error found: Single bit corrected.
:            Y indexes corrected byte.
:ERRORS      More than one real bit-error, or a bit error in
:            the error correction byte can result in an
:            uncorrupted bit being inverted. Several errors
:            in the data can cancel each other out.
:REG USE     P Y
:STACK USE   7 (including JSR ECAL)
:RAM USE     M0 M1
:LENGTH     53
:CYCLES      Average 192 + 175 per data byte.
:-----
:CLASS 2     *discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
:=====

```

REDUCING ERRORS

```

:
DATA      =      M0      :2-byte stored data block address.
ERRI      =      M2      :Store for error correction code.
:
EFIX      PHA          :Save A and X                      48
          TXA          :for use in EFIX.                  8A
          PHA          :                                  48
          JSR ECAL      :Get new ecb for block but        20 lo hi
          BCS ENDPLR    :abort (C = 1) if ECAL aborts.    B0 29
:
          LDA ERRI      :Save page zero storage          A5 M2
          PHA          :for error correction.             48
          TXA          :Move new ecb to A and test       8A
          EOR (DATA),Y :against appended ecb, if         51 M0
          CMP #8        :difference < 8 then no          C9 08
          BCC ENDPLZ    :error found in data block.       90 1C
:
          TAX          :Any set bits in EOR'd ecb        AA
          LSR A         :bits 7 to 3 are index to         4A
          LSR A         :position of corrupt byte in     4A
          LSR A         :data block, so convert to       4A
          STA ERRI      :byte index, store in page 0     85 M2
          CPY ERRI      :and test if position is in      C4 M2
          BCC ENDPLZ    :data block, exit if outside.    90 12
:
          TYA          :In block, so convert from        98
          SBC ERRI      :index-from-block-end to         E5 M2
          TAY          :index-from-start, in Y.          A8
:
          TXA          :Bits 2,1,0 of EOR'd ecb          8A
          AND #7        :index corrupt bit in byte       29 07
          TAX          :so isolate them, clear A and     AA
          LDA #0        :using set C from "SBC ERRI",    A9 00
SETBLP    ROL A         :Shift one set bit into A        2A
          DEX          :in same position as the          CA
          BPL SETBLP    :inverted bit in corrupt byte.   10 FC
:
          EOR (DATA),Y :Use set bit to re-invert         51 M0
          STA (DATA),Y :corrupt bit, return to block.    91 M0
:
ENDPLZ    PLA          :Restore page zero byte           68
          STA ERRI      :used for error code.            85 M2
ENDPLR    PLA          :Restore X and A                  68
          TAX          :then exit EFIX with              AA
          PLA          :C = 1 if aborted or              68
          RTS          :C = 0 if block okay.             60
:=====

```


Legible Listings

Most computers come supplied with efficient software that includes various output formats for the different types of data and programs that you are likely to use. Sometimes, however, the printout or screen display doesn't carry all the information you might require. Or the information is there but difficult to read.

This chapter contains two routines which 'patch in' to the computer's output routine and edit the information being sent to it. The first of these routines is written to be generally applicable and requires only that your computer uses ASCII codes. The second, though, is written specifically for the *BBC* computer as an assembler program embedded in and set up by *BBC BASIC*. Don't skip it if you don't have a *BBC* computer, the method it uses can easily be adapted to customise all formatted outputs on other computers.

IN CONTROL

CTRPRT by Tim Herklots intercepts ASCII data being sent to an output routine and converts all control codes \$00 to \$1F into their 2- or 3-letter standard abbreviations. The abbreviations are bracketed by LESS-THAN and GREATER-THAN symbols < and >.

One major use for CTRPRT is to extend the capabilities of code dump routines. Some of these print out the ASCII characters alongside the hexadecimal digits in case text data and not machine code is coming through. ASCII control codes (and values above \$7E, for that matter) are usually printed either as spaces or periods – not very useful if your code is doing a lot of screen movements!

Here is an example of the sort of effect CTRPRT has on a section of code. It is a data string which clears the screen and then prints out two messages at carefully positioned screen locations.

NORMAL HEX DUMP

0C 0A 0A 0A 11 11 4D 45	ME
53 53 41 47 45 20 31 0D	SSAGE 1.
0A 0A 0A 6D 65 73 73 61	...messa
67 65 20 32	ge 2

LEGIBLE LISTINGS

HEX DUMP THROUGH CTRPRT

```
0C 0A 0A 0A 11 11 4D 45    <FF><LF><LF><LF><DC1><DC1>ME
53 53 41 47 45 20 31 0D    SSAGE 1<CR>
0A 0A 0A 6D 65 73 73 61    <LF><LF><LF>messa
67 65 20 32                ge 2
```

```
=====
:= CTRPRT    Control character name print.
=====
```

```
:JOB        To intercept an ASCII control code destined for
:            a print routine, converting it to its three
:            letter abbreviation enclosed in brackets.
:ACTION     IF character is a control code, THEN:
:            [ Use code * 3 as index to abbreviation table.
:              Copy abbreviation to stack. Put "<" on stack.
:              FOR count of 4:
:                [ Pull byte. IF NOT "*" THEN: [ Print. ]. ]
:              Set character = ">". ]
:            Exit to Print routine.
```

```
-----
:CPU        6502
:HARDWARE   None.
:SOFTWARE   "WRCHAR" - subroutine to print ASCII char. in A.
```

```
-----
:INPUT      A contains ASCII character or control code.
:OUTPUT     If input is a character ($20+), A is unchanged.
:            If input is a control code, output A = ">".
:ERRORS     None.
:REG USE    A P
:STACK USE  4
:RAM USE    M0
:LENGTH     146 (Code: 50. Data: 96).
:CYCLES     Not given.
```

```
-----
:CLASS 2    -discreet *interruptable *promable
:***-*      *reentrant -relocatable *robust
=====
```

```
:
CTRPRTP CMP #32      :Pass straight through if      C9 20
          BCS CPEND   :not a control code.          B0 2B
:
          STA M0      :Else code table offset      85 M0
          ASL A        :is 3 * code number.        0A
          ADC M0       :                          65 M0
          STX M0       :Save X and move offset      86 M0
          TAX          :to X as code index.        AA
          LDA CTAB+2,X :Get last letter            BD lo hi
          PHA          :to stack.                  48
          LDA CTAB+1,X :Get middle letter          BD lo hi
          PHA          :to stack.                  48
          LDA CTAB,X   :Get first letter           BD lo hi
          PHA          :to stack.                  48
          LDA #$3C     :Get LESS-THAN symbol      A9 3C
          PHA          :to stack.                  48
          LDX M0       :Restore X.                A6 M0
          LDA #4       :Set count of stacked      A9 04
          STA M0       :letters in M0.             85 M0
```

LEGIBLE LISTINGS

```

:
CPLP   PLA           :Get character off stack      68
      CMP   #42       :but if "*" space filler    C9 2A
      BEQ   CPLTST     :then miss printing it.      F0 03
      JSR   WRCHAR     :Go print character.        20 lo hi
CPLTST DEC   MO       :Repeat for four stacked      C6 MO
      BNE   CPLP       :characters.          DO F4
      LDA   #$3E       :End with GREATER-THAN.    A9 3E
:
CPEND  JMP   WRCHAR   :Exit to print last char.    4C lo hi
:
CTAB   .BYTE 78,85,76 :NUL   Null                    4E 55 4C
      .BYTE 83,79,72 :SOH   Start of Heading      53 4F 48
      .BYTE 83,84,88 :STX   Start Text          53 54 58
      .BYTE 69,84,88 :ETX   End Text            45 54 58
      .BYTE 69,79,84 :EOT   End of Transmission  45 4F 54
      .BYTE 69,78,81 :ENQ   Enquiry             45 4E 51
      .BYTE 65,67,75 :ACK   Acknowledge          41 43 4B
      .BYTE 66,69,76 :BEL   Bell                42 45 4C
      .BYTE 66,83,42 :BS*   Backspace           42 53 2A
      .BYTE 72,84,42 :HT*   Horizontal Tab       48 54 2A
      .BYTE 76,70,42 :LF*   Line Feed            4C 46 2A
      .BYTE 86,84,42 :VT*   Vertical Tab         56 54 2A
      .BYTE 70,70,42 :FF*   Form Feed           46 46 2A
      .BYTE 67,82,42 :CR*   Carriage Return      43 52 2A
      .BYTE 83,79,42 :SO*   Shift Out            53 4F 2A
      .BYTE 83,73,42 :SI*   Shift In             53 49 2A
      .BYTE 68,76,69 :DLE   Data Link Escape     44 4C 45
      .BYTE 68,67,49 :DC1   Direct Control 1     44 43 31
      .BYTE 68,67,50 :DC2   Direct Control 2     44 43 32
      .BYTE 68,67,51 :DC3   Direct Control 3     44 43 33
      .BYTE 68,67,52 :DC4   Direct Control 4     44 43 34
      .BYTE 78,65,75 :NAK   Negative Acknowledge 4E 41 4B
      .BYTE 83,89,78 :SYN   Synchronous Idle    53 59 4E
      .BYTE 69,84,66 :ETB   End Transmission Block 45 54 42
      .BYTE 67,65,78 :CAN   Cancel              43 41 4E
      .BYTE 69,77,42 :EM*   End of Medium        45 4D 2A
      .BYTE 83,85,66 :SUB   Substitute          53 55 42
      .BYTE 69,83,67 :ESC   Escape              45 53 43
      .BYTE 70,83,42 :FS*   Form Separator       46 53 2A
      .BYTE 71,83,42 :GS*   Group Separator      47 53 2A
      .BYTE 82,83,42 :RS*   Record Separator     52 53 2A
      .BYTE 85,83,42 :US*   Unit Separator       55 53 2A
:=====

```

BBC ASSEMBLER LISTING

BBC BASIC is a pretty powerful tool, allowing you to embed assembly language programs which can use BASIC variables. The assembler also supports multi-statement lines as in BASIC.

The only problem with this wealth of facilities is that of readability. The *Beeb* doesn't go in for standard assembler formatting and, rather than helping you to find your way through a routine, comments tend to obscure – as in Listing 1 of LSTFMT.

LEGIBLE LISTINGS

```

10REM - "LSTFMT" - FORMATTED ASSEMBLER LISTING
20HX=PAGE/256-1:LX=&15
30*FX 6,10
40*KEY 10 IN OLDIM
50*KEY0"?&20E=&A4:?"&20F=&E0IM"
60*KEY1"?&20E=LX:?"&20F=HX:!"&70=0IM"
70REM - VECTOR PRINT ROUTINE WRCH
80OSWRCH=?&20E+?"&20F*256
90REM - SYSTEM READ OF CURSOR POSITION
100OSBYTE=&FFF4:csrpos=&86
110REM - SET UP PAGE ZERO USE
120asmfl=&70:cmtfl=&71:lblfl=&72:litfl=&73:tabst=&74:tempA
=&75
130REM - NAME FORMATTING CHARACTERS
140asmin=&5B:asmout=&5D:colon=&3A:quotes=&22:carret=&0D:ln
feed=&0A
150space=&20:label=&2E:commnt=&5C
160REM - FIELD TAB SETTINGS
170coltab=4:lbltab=6:mnmtab=14:cmntab=28:lwidth=59
180FOR I=0 TO 2 STEP 2
190P%=HX*256+LX
200[OPT I
210STA tempA \Save value in zero page
220PHP:TXA:PHA:TYA:PHA \Save registers and
230LDX tempA \get input value in X for tests.
240TXA:SEC:SBC #carret:BNE qtest \Test for line end.
250STA cmtfl:STA lblfl:STA litfl \Clear comment, label & l
iteral flags.
260BEQ bexit \Hop to exit.
270.qtest CPX #quotes:BNE lftest \Test for literal start/e
nd.
280LDA litfl:EOR #1:STA litfl \Toggle literal flag on/off.
290DEX:BNE width \Hop to line-end test.
300.lftest LDA litfl:BNE width \No formatting if flag on.
310CPX #colon:BNE asstst \Test statement as new line.
320STA cmtfl:STA lblfl \Clear flags
330JSR lfeed \Go to new line.
340LDY #coltab:JSR tabout \Tab to colon position.
350TXA:JSR OSWRCH \Print colon,
360LDA #space:STA tempA:BNE exit \then a space on exit.
370.asstst CPX #asmin:BNE asetst \Test for assembler start
.
380INC asmfl:BNE width \Flag on and go to line-end test.
390.asetst CPX #asmout:BNE aftst \Test for assembler end.
400DEC asmfl:BEQ width \Flag off and go to line-end test.
410.aftst LDA asmfl:BEQ width \BASIC if flag off - no form
at.
420LDA cmtfl:BNE width \Inside a comment - carry on.
430CPX #commnt:BNE lbftst \Test for comment start.
440INC cmtfl:LDA #cmntab:JSR pos \Comment flag on and test
print position.
450BCC exit:BEQ exit:TAY:JSR tabout \Tab to comment field
if needed.
460.bexit BEQ exit \Also "stepping stone to 'exit'.
470.lbftst LDA lblfl:BEQ lbtest \Test if inside a label.
480CPX #space:BNE exit \Print if not end of label,
490DEC lblfl:BEQ exit \else flag off and print space.
500.lbtest CPX #label:BNE lintst \Test for label start.

```

LEGIBLE LISTINGS

```

510INC lblfl:BNE exit \Flag on and exit to print "."
520.lintst CPX #space:BEQ exit \Print space on exit.
530CPX #&30:BCC mnmtst:CPX #&3A:BCC exit \Digits ok - prob
ably line number.
540.mnmtst LDA #mnmtab:JSR pos:BCC exit:BEQ exit \Exit if
in mnemonic,
550TAY:JSR tabout:BEQ exit \else tab to mnemonic field.
560.width LDA #lwidth:JSR pos:BCS exit \Okay if not at lin
e-end
570JSR lfeed \else next line and
580LDA cmtfl:BEQ lblpos \skip if not in a comment
590LDY #cmntab:JSR tabout \else tab up to comment position
and
600LDA #commnt:JSR OSWRCH:INY:BNE exit \write new comment
symbol.
610.lblpos LDY #lbltab:JSR tabout \Else tab to label field
.
620.exit PLA:TAY:PLA:TAX:PLP \Restore registers and
630LDA tempA:JMP OSWRCH \exit through character print rout
ine.
640.tabout LDA #space:JSR OSWRCH:DEY:BNE tabout:RTS
650.lfeed LDA #carret:JSR OSWRCH \Using OSNEWL would send
CHR$ 13 and 10
660LDA #lnfeed:JSR OSWRCH \through LSTFMT and overwrite
670RTS \the character stored in tempA.
680.pos PHA:LDA #csrpos:JSR OSBYTE \Read text cursor posit
ion.
690STX tabst:LDX tempA:PLA \store it and recover registers
700SEC:SBC tabst:RTS \get position difference and return.
710J
720NEXT

```

LSTFMT is a patch by Sverrir Karlsson which intercepts characters on their way to the *BBC* print routine WRCH. It formats the listing of both normal BASIC and assembler lines to one statement or instruction per printed line. Assembly language is also formatted to normal line number, label, mnemonic, operand and comment fields. Listing 2 is the result of LSTFMT acting on itself.

```

10 REM - "LSTFMT" - FORMATTED ASSEMBLER LISTING
20 H%=PAGE/256-1
: L%=&15
30 *FX 6,10
40 *KEY 10 IN OLDIM
50 *KEY0"?&20E=&A4:?&20F=&E0IM"
60 *KEY1"?&20E=L%:?&20F=H%:!!&70=0IM"
70 REM - VECTOR PRINT ROUTINE WRCH
80 OSWRCH=?&20E+?&20F*256
90 REM - SYSTEM READ OF CURSOR POSITION
100 OSBYTE=&FFF4
: csrpos=&86
110 REM - SET UP PAGE ZERO USE
120 asmfl=&70
: cmtfl=&71
: lblfl=&72
: litfl=&73

```

LEGIBLE LISTINGS

```

: tabst=&74
: tempA=&75
130 REM - NAME FORMATTING CHARACTERS
140 asmin=&5B
: asmout=&5D
: colon=&3A
: quotes=&22
: carret=&0D
: lnfeed=&0A
150 space=&20
: label=&2E
: commnt=&5C
160 REM - FIELD TAB SETTINGS
170 coltab=4
: lbltab=6
: mnmtab=14
: cmntab=28
: lwidth=59
180 FOR I=0 TO 2 STEP 2
190 P%=H%*256+L%
200 [ OPT I
210 STA tempA \Save value in zero page
220 PHP
: TXA
: PHA
: TYA
: PHA \Save registers and
230 LDX tempA \get input value in X for tests.
\
240 TXA
: SEC
: SBC,#carret
: BNE qtest \Test for line end.
250 STA cmtfl
: STA lblfl
: STA litfl \Clear comment, label & literal
\flags.
260 BEQ bexit \Hop to exit.
270 .qtest CPX #quotes
: BNE lftest \Test for literal start/end.
280 LDA litfl
: EOR #1
: STA litfl \Toggle literal flag on/off.
290 DEX
: BNE width \Hop to line-end test.
300 .lftest LDA litfl
: BNE width \No formatting if flag on.
310 CPX #colon
: BNE asstst \Test statement as new line.
320 STA cmtfl
: STA lblfl \Clear flags
330 JSR lfeed \Go to new line.
340 LDY #coltab
: JSR tabout \Tab to colon position.
350 TXA
: JSR OSWRCH \Print colon,
360 LDA #space
: STA tempA

```

LEGIBLE LISTINGS

```

:      BNE exit      \then a space on exit.
370 .asstst CPX #asmin
:      BNE asetst    \Test for assembler start.
380      INC asmfl
:      BNE width     \Flag on and go to line-end test
:                      \.
390 .asetst CPX #asmout
:      BNE aftst      \Test for assembler end.
400      DEC asmfl
:      BEQ width      \Flag off and go to line-end tes
:                      \t.
410 .aftst  LDA asmfl
:      BEQ width      \BASIC if flag off - no format.
420      LDA cmtfl
:      BNE width      \Inside a comment - carry on.
430      CPX #commnt
:      BNE lbftst     \Test for comment start.
440      INC cmtfl
:      LDA #cmntab
:      JSR pos        \Comment flag on and test print
:                      \position.
450      BCC exit
:      BEQ exit
:      TAY
:      JSR tabout     \Tab to comment field if needed.
:                      \
460 .bexit  BEQ exit   \Also "stepping stone to 'exit'.
:                      \
470 .lbftst LDA lblfl
:      BEQ lbtest     \Test if inside a label.
480      CPX #space
:      BNE exit       \Print if not end of label,
490      DEC lblfl
:      BEQ exit       \else flag off and print space.
500 .lbtest CPX #label
:      BNE lintst     \Test for label start.
510      INC lblfl
:      BNE exit       \Flag on and exit to print "."
520 .lintst CPX #space
:      BEQ exit       \Print space on exit.
530      CPX #830
:      BCC mnmtst
:      CPX #83A
:      BCC exit       \Digits ok - probably line numbe
:                      \r.
540 .mnmtst LDA #mnmtab
:      JSR pos
:      BCC exit
:      BEQ exit       \Exit if in mnemonic,
550      TAY
:      JSR tabout
:      BEQ exit       \else tab to mnemonic field.
560 .width  LDA #lwidth
:      JSR pos
:      BCS exit       \Okay if not at line-end
570      JSR lfeed     \else next line and
580      LDA cmtfl
:      BEQ lblpos     \skip if not in a comment

```

LEGIBLE LISTINGS

```

590      LDY #cmntab
:      JSR tabout      \else tab up to comment position
                        \ and
600      LDA #commnt
:      JSR OSWRCH
:      INY
:      BNE exit        \write new comment symbol.
610 .lblpos LDY #lbltab
:      JSR tabout      \Else tab to label field.
620 .exit  PLA
:      TAY
:      PLA
:      TAX
:      PLP             \Restore registers and
630      LDA tempA
:      JMP OSWRCH      \exit through character print ro
                        \utine.
640 .tabout LDA #space
:      JSR OSWRCH
:      DEY
:      BNE tabout
:      RTS
650 .lfeed LDA #carret
:      JSR OSWRCH      \Using OSNEWL would send CHR$ 13
                        \ and 10
660      LDA #lnfeed
:      JSR OSWRCH      \through LSTFMT and overwrite
670      RTS           \the character stored in tempA.
680 .pos   PHA
:      LDA #csrpos
:      JSR OSBYTE      \Read text cursor position.
690      STX tabst
:      LDX tempA
:      PLA             \store it and recover registers
700      SEC
:      SBC tabst
:      RTS            \get position difference and ret
                        \urn.
710 J
720 NEXT

```

LSTFMT should be typed in as it appears in Listing 1 (without spaces after line numbers – the *BBC* LIST option command will insert them later) and RUN. After running, type LIST01, press FUNCTION KEY 1 and type LIST. The program should appear as in Listing 2 both on screen and, if you have enabled your printer (CTRL B), on paper.

Pressing FUNCTION KEY 0 will cause listing in the normal List Option 1 and the BREAK key (FUNCTION KEY 10) will cause a reversion to unspaced listing.

The string assigned in *KEY1 loads the address of LSTFMT into the OSWRCH vector causing print subroutine calls to be directed to LSTFMT instead of WRCH. The string in *KEY0 replaces the address of WRCH in the vector, disabling LSTFMT. *FX 6,10 stops line feed

(\$0A, 'LF') being sent to the printer – remove it if your printer doesn't perform line feed automatically when it receives a carriage return.

LSTFMT assembles the machine code at the page below the start of BASIC program space. In a cassette system this is &E00 (to use the *BBC BASIC* hex symbol) but &1900 in a DFS disk system. To be certain that the machine code will not overwrite anything – or be overwritten – you could clear a 256-byte page for it below the BASIC program space by typing PAGE = &F00 (or PAGE = &1A00 for a disk system).

One final word of warning: don't attempt to edit your programs in the formatted mode; LSTFMT is strictly a one way process.

Dot Graphics 11

The routines in this chapter are all different methods of manipulating graphics information of a very particular type and were sent to *Sub Set* in response to the following challenge.

ELEGANT SOLUTION INVITED

In this problem, the target byte holds information which determines how 4 graphics dots are to be displayed as either background (not showing) or as one of three colours. This dot information is held in the least significant 2 bits of a byte, which represent the four states 00, 01, 10 and 11 with the 6 most significant bits of the byte reset. The information for the dots, which are numbered 0 to 3, is arranged in the target byte thus:

bit	7	6	5	4	3	2	1	0
dot	0	1	2	3	0	1	2	3

Dot 0 information is in bits 7 and 3 of the byte, dot 1 in bits 6 and 2, dot 2 in bits 5 and 1 and dot 3 in bits 4 and 0.

Given a 2-byte address of the target byte, a 1-byte dot number (binary 0 to 3) and 1 byte of dot information (wherever you like it in either registers, memory, or as parameters embedded in the code following the subroutine call), we want the most elegant routine to place the dot information, according to dot number, in the target byte, without disturbing any of the information relating to the other 3 dots.

Those of you with the right hardware will recognise the format of the target byte as that used in display modes 1 and 5 of the *BBC Micro*. So solutions in 6502 code will be particularly relevant, though solutions in other code will be interesting.

6502 ELEGANT SOLUTIONS

There was a very large response to this challenge and only a selection of the shortest, quickest and most unusual contributions could be published. Some readers favoured fully processed solutions, which tend to be short but slow, whilst others went for the fast but byte-heavy table look-up methods.

DOT1 is by D. A. Stanford. At 32 bytes, it is the shortest solution to operate in a reasonable time.

DOT GRAPHICS

```

=====
:= DOT1      Put graphics dot information
=====
:JOB         To store graphic information to 2 specified bits
:            of a graphic byte.
:ACTION      Reformat dot information byte to match graphic
:            byte format.
:            Construct bit mask to graphic byte format.
:            Mask out old dot information in graphic byte.
:            Mask in new dot information to graphic byte.
-----
:CPU         6502
:HARDWARE    Graphics byte in (video) RAM.
:SOFTWARE    None.
-----
:INPUT       M0,1 addresses graphic byte where information is
:            to be stored.
:            M2 contains the dot number relating to pairs of
:            bits in the graphic byte:
:            dot number = 0: graphic byte bit 7 & bit 3.
:            dot number = 1: graphic byte bit 6 & bit 2.
:            dot number = 2: graphic byte bit 5 & bit 1.
:            dot number = 3: graphic byte bit 4 & bit 0.
:            M3 contains information to be entered in the
:            addressed graphic byte bit pair:
:            bit 1,M3 going to high nibble dot.
:            bit 0,M3 going to low nibble dot.
:OUTPUT      Information entered. M0, M1 and M2 not changed..
:            P, A, Y and M3 changed.
:ERRORS      No check that dot number and dot information are
:            both in the range 0 to 3.
:REG USE     P A Y
:STACK USE   1
:RAM USE     M0 to M3
:LENGTH     32
:CYCLES      Minimum: 54. Maximum: 95.
-----
:CLASS 2     -discreet *interruptable *promable
:-*****-    *reentrant *relocatable -robust
=====
ADDR  =      M0           :2-byte stored graphic byte address.
DOTNO  =      M2           :Stored dot number.
DOTINF  =      M3          :Stored dot information.
:
DOT1   LDA  DOTINF         :Get inf as %0000 00ab  C=?   A5 M3
      LSR  A              :Split bits: %0000 000a  C=b   4A
      ROR  A              :           %b000 0000  C=a   6A
      PHP                     :Save bit "a" in stacked C.   08
      LSR  A              :           %0b00 0000  C=0   4A
      LSR  A              :           %00b0 0000  C=0   4A
      LSR  A              :           %000b 0000  C=0   4A
      PLP                     :Restore bit "a" to C.       28
      ROR  A              :           %a000 b000  C=0   6A
      STA  DOTINF         :Store split information.        A5 M3
:
      LDA  #$77           :Mask to delete old information. A9 77
      LDY  DOTNO          :Get dot no. as count, skip if  A4 M2
      BEQ  INSRT          :mask & info in right place,    F0 07

```

```

:
SHIFT  LSR  DOTINF  :Else shift info byte by 1 bit, 46 M3
        SEC          :prepare to shift in set bit, 38
        ROR  A        :shift delete-mask by 1 bit 6A
        DEY          :until info and mask are both 88
        BNE  SHIFT    :at right dot number.      80 F9
:
INSRT  AND  (ADDR),Y :Mask out old information and 31 M0
        ORA  DOTINF   :replace by new information 05 M3
        STA  (ADDR),Y :then re-store graphic byte. 91 M0
        RTS          :Exit, information placed. 60
:=====

```

D. A. Stanford also contributed a variation to DOT1 which replaces the 8-byte rotation sequence (instructions 2 to 9) by a single 3-byte instruction which reads the reformed dot information from a 4-byte table. DOT2 beats DOT1 by one byte and 15 clock cycles.

```

DATA  .BYTE %00000000 :Split of information 00. 00
        .BYTE %00001000 : 01. 08
        .BYTE %10000000 : 10. 80
        .BYTE %10001000 : 11. 88
:
DOT2  LDY  DOTINF      :Convert from index as A4 M3
        LDA  DATA,Y    :%0000000ab to split form B9 lo hi
        STA  DOTINF      :%a000b000 in DOTINF. 85 M3
:
:...Continue as for DOT1.

```

But, faster and shorter though it is, D. A. Stanford didn't think DOT2 as 'elegant' as DOT1.

The notion of 'elegance' in a routine, code or merely the method used seemed to puzzle many contributors. Elegant solutions are those which are simple and don't require a lot of forced manipulation. For example, the following code splits the 2 bits of information to separate nibbles (albeit to dot number 3 instead of dot number 0) without the mixture of rotates, shifts, pushes and pulls used in DOT1. It is my 6502 implementation of a method contributed in 8085 code.

```

SPLIT3 LDA  DOTINF :Get information as %000000ab, A5 M3
        CLC          :Prepare for addition. 18
        ADC  #$0E     :Propagate bit 1 to bit 4 and 69 0E
        AND  #$11     :mask out all unwanted bits. 29 11
        STA  DOTINF   :Store information as %000a000b. 85 M3

```

SPLIT3 is 3 bytes shorter and 13 clock cycles faster than the method used in DOT1. Unfortunately, it takes another 4 bytes and 4 clock cycles to extend the method so that it converts the information to the form needed by DOT1.

DOT GRAPHICS

SPLIT	LDA	DOTINF	:Get information as %000000ab,	A5 M3
	CLC		:Prepare for addition.	18
	ADC	#\$0E	:Propagate bit 1 to bit 4 and	69 0E
	AND	#\$11	:mask out all unwanted bits.	29 11
	ADC	#\$77	:Propagate: 4 to 7 and 0 to 3 and	69 77
	AND	#\$88	:mask out all unwanted bits.	29 88
	STA	DOTINF	:Store information as %a000b000.	85 M3

TABLE LOOK-UP

DOT3 by Oscar Burke uses look-up tables for both the mask to delete the existing information in the dot bits and the split information to be entered. With 19 bytes for the routine and another 20 bytes for the two tables, it is nearly 25% longer than DOT1 but quite a bit faster. Operating speed is probably more important than the length of a routine which deals only with one dot on a high resolution screen. Oscar has squeezed an extra ounce of speed out of DOT3 by having the index registers X and Y already loaded on entry, which could well be the case in a complete application.

```

=====
:= DOT3      Put graphics dot information
=====
:JOB         To store graphic information to 2 specified bits
:            of a graphic byte.
:ACTION      Use dot number to index and get mask-out byte.
:            Use dot number and dot information to index
:            mask-in byte.
:            Mask out old dot information in graphic byte.
:            Mask in new dot information to graphic byte.
:-----
:CPU         6502
:HARDWARE    Graphics byte in (video) RAM.
:SOFTWARE    None.
:-----
:INPUT       M0,1 contains the base address of a block or
:            file of graphic bytes.
:            Y indexes the particular graphic byte where
:            information is to be stored.
:            X contains the dot number relating to pairs of
:            bits in the graphic byte:
:            dot number = 0: graphic byte bit 7 & bit 3.
:            dot number = 1: graphic byte bit 6 & bit 2.
:            dot number = 2: graphic byte bit 5 & bit 1.
:            dot number = 3: graphic byte bit 4 & bit 0.
:            M3 contains information to be entered in the
:            addressed graphic byte bit pair:
:            bit 1,M3 going to high nibble dot.
:            bit 0,M3 going to low nibble dot.
:OUTPUT      Information entered. Y, M0, M1, M3 not changed.
:            P, A, X changed.
:ERRORS      No check that dot number and dot information are
:            both in the range 0 to 3.

```

```

:REG USE      P A X Y
:STACK USE    1
:RAM USE      M0 M1 M3
:LENGTH      19 (plus 20 bytes for tables MASK and VAL).
:CYCLES       43
:-----
:CLASS 2      -discreet *interruptable *promable
:***--        *reentrant -relocatable -robust
:=====
:
BASE = M0      :2-byte stored base address of block
               :of graphic bytes (indexed by Y).
DOTINF = M3     :Stored dot information.
:
DOT3 LDA MASK,X :Get mask for later mask out BD lo hi
PHA           :of old info, saved on stack. 48
:
TXA           :Convert dot number to index 8A
ASL A         :table VAL, dot number indexes 0A
ASL A         :block of 4 bytes, dot info 0A
ORA DOTINF    :indexes particular byte in 05 M3
TAX           :block, index to X. AA
:
PLA           :Restore mask to A. 68
AND (BASE),Y  :Mask out old information and 31 M0
ORA VAL,X     :replace by new information 1D lo hi
STA (BASE),Y  :then re-store graphic byte. 91 M0
RTS           :Exit, information placed. 60
:
MASK .BYTE %01110111 :Mask out for dot 0. 77
      .BYTE %10111011 :Mask out for dot 1. BB
      .BYTE %11011101 :Mask out for dot 2. DD
      .BYTE %11101110 :Mask out for dot 3. EE
:
VAL .BYTE %00000000 :Mask in, dot 0 info 00. 00
     .BYTE %00001000 : 0 01. 08
     .BYTE %10000000 : 0 10. 80
     .BYTE %10001000 : 0 11. 88
     .BYTE %00000000 : 1 00. 00
     .BYTE %00000100 : 1 01. 04
     .BYTE %01000000 : 1 10. 40
     .BYTE %01000100 : 1 11. 44
     .BYTE %00000000 : 2 00. 00
     .BYTE %00000010 : 2 01. 02
     .BYTE %00100000 : 2 10. 20
     .BYTE %00100010 : 2 11. 22
     .BYTE %00000000 : 3 00. 00
     .BYTE %00000001 : 3 01. 01
     .BYTE %00010000 : 3 10. 10
     .BYTE %00010001 : 3 11. 11
:=====

```

PROCESSED LOOK-UP

Not quite the shortest and not quite the fastest is this compromise between the use of look-up tables and computation by Glen Slade. DOT4 actually manipulates the mask to be both a mask-out byte to

DOT GRAPHICS

delete existing information and a mask-in byte to pick up the new information only in the required dot positions.

```

=====
:= DOT4      Put graphics dot information
=====
:JOB         To store graphic information to 2 specified bits
:            of a graphic byte.
:ACTION      Use dot number to index and get position mask.
:            Use inverted mask to mask out old information.
:            Use dot information to index dot value table.
:            Use mask to get dot specific value.
:            Merge in new dot value to graphic byte.
-----
:CPU         6502
:HARDWARE    Graphics byte in (video) RAM.
:SOFTWARE    None.
-----
:INPUT       M0,1 addresses graphic byte where information is
:            to be stored.
:            M2 contains the dot number relating to pairs of
:            bits in the graphic byte:
:            dot number = 0: graphic byte bit 7 & bit 3.
:            dot number = 1: graphic byte bit 6 & bit 2.
:            dot number = 2: graphic byte bit 5 & bit 1.
:            dot number = 3: graphic byte bit 4 & bit 0.
:            M3 contains information to be entered in the
:            addressed graphic byte bit pair:
:            bit 1,M3 going to high nibble dot.
:            bit 0,M3 going to low nibble dot.
:OUTPUT      Information entered. M0, M1, M2, M3 not changed.
:            P, A, X, Y and M4 changed.
:ERRORS      No check that dot number and dot information are
:            both in the range 0 to 3.
:REG USE     P A X Y
:STACK USE   1
:RAM USE     M0 M1 M3 M4
:LENGTH     25 (plus 8 bytes for tables MASK and VAL).
:CYCLES      48
-----
:CLASS 2     -discreet *interruptable *promable
:-****-      *reentrant -relocatable -robust
=====
:
ADDR = M0      :2-byte stored graphic byte address.
DOTNO = M2     :Stored dot number.
DOTINF = M3    :Stored dot information.
TEMP = M4      :For temporary storage of graphic byte.
:
DOT4 LDY DOTNO :Use dot number as index to      A4 M2
     LDA MASK,Y :getting dot position mask.      B9 lo hi
     PHA        :Saved for masking inf later.  48
:
     EOR #$FF   :Invert to mask-out byte.      49 FF
     LDY #0     :Zero index addressed byte.    A0 00
     AND (ADDR),Y :Mask out old information and 31 M0
     STA TEMP   :store for later merge in.     85 M4

```

```

:
    LDX DOTINF :Use info number as index.      A6 M3
    PLA       :Recover mask-in byte and get    68
    AND VAL,X  :value at dot bits only then    3D lo hi
    ORA TEMP   :merge with unaffected dots     05 M4
    STA (ADDR),Y :and re-store graphic byte.    91 M0
    RTS       :Exit, information placed.       60

:
MASK .BYTE %10001000 :Mask for dot 0.          88
     .BYTE %01000100 :Mask for dot 1.          44
     .BYTE %00100010 :Mask for dot 2.          22
     .BYTE %00010001 :Mask for dot 3.          11

:
VAL  .BYTE %00000000 :Info 00 for all dots.     00
     .BYTE %00001111 : " 01 "                 0F
     .BYTE %11110000 : " 10 "                 F0
     .BYTE %11111111 : " 11 "                 FF
:=====

```

THE ROTATIONAL METHOD

DOT5 by W. Anderton is by far the slowest method at 164 clock cycles but is as short as DOT1 and very elegant in principle. The target byte is rotated nine times through the Carry flag and, as the bits relating to the addressed dot move into Carry, they are replaced by the new information bits.

The target byte in DOT5 is addressed absolutely since rotate instructions cannot use the indexed indirect addressing modes. However, the routine can easily be altered to use indirect addressing. Indirectly pick up the target byte in A just before the first call to ROLL (using the address in M0,1 indexed by Y, as in the other DOT routines) and store the adjusted byte back after the last ROLL-CALL (irresistible!). The first instruction in ROLL should be changed from 'ROR GBYTE' to 'ROR A'.

```

:=====
:= DOT5      Put graphics dot information
:=====
:JOB        To store graphic information to 2 specified bits
:           of a graphic byte.
:ACTION     Rotate graphic byte lo-nibble bit to Carry.
:           Move new lo-nibble information bit to Carry.
:           Rotate graphic byte hi-nibble bit to Carry (new
:           information bit being rotated in).
:           Move new hi-nibble information bit to Carry.
:           Rotate new information bit to correct place in
:           graphic byte.
:-----
:CPU        6502
:SOFTWARE   Graphics byte in (video) RAM.
:SOFTWARE   "ROLL" - Local subroutine to rotate the graphic
:           byte right through Carry by X bits.
:-----

```

DOT GRAPHICS

```
:INPUT      M2 contains the dot number relating to pairs of
:           bits in the graphic byte:
:           dot number = 0: graphic byte bit 7 & bit 3.
:           dot number = 1: graphic byte bit 6 & bit 2.
:           dot number = 2: graphic byte bit 5 & bit 1.
:           dot number = 3: graphic byte bit 4 & bit 0.
:           M3 contains information to be entered in the
:           graphic byte bit pair:
:           bit 1,M3 going to high nibble dot.
:           bit 0,M3 going to low nibble dot.
:OUTPUT      Information entered. M2 not changed.
:           P, A, X and M3 changed.
:ERRORS      No check that dot number and dot information are
:           both in the range 0 to 3.
:           Arithmetic error if D = 1 (decimal mode).
:REG USE      P A X
:STACK USE    2
:RAM USE      M2 M3
:LENGTH      32
:CYCLES       164
```

```
:-----
:CLASS 2      -discreet *interruptable *promable
:~***~      *reentrant -relocatable -robust
:=====
```

```
:
GBYTE = $hilo :Absolute address of graphic byte.
DOTNO = M2    :Stored dot number.
DOTINF = M3   :Stored dot information.
:
DOT5 LDA #4    :Calculate number of rotations A9 04
SEC      :needed to shift addressed 38
SBC DOTNO :lo-nibble bit out to Carry, E5 M2
TAX      :move to X as rotation count AA
JSR ROLL  :and get bit out to Carry. 20 lo hi
:
LSR DOTINF :Replace C by new lo-nib bit, 46 M3
LDX #4     :and go rotate it in, getting A2 04
JSR ROLL  :old hi-nib bit out to Carry. 20 lo hi
:
LSR DOTINF :Replace C by new hi-nib bit 46 M3
LDX DOTNO  :and compute no. of rotates A6 M2
INX        :needed to restore correct E8
JSR ROLL  :positions, go do it, then 20 lo hi
RTS       :exit, information placed. 60
:
ROLL ROR GBYTE :Rotate graphic byte, through 6E lo hi
DEX      :Carry, for count of X. CA
BNE ROLL  :(Total of 9 rotations done in D0 FA
RTS       :DOT5, leaving GBYTE correct.) 60
:=====
```

Random Numbers

Random numbers are readily obtained in the real world. Rolling dice will provide you with a set of values 1 to 6 which are fairly random although any slight irregularity in a die will predispose it to favour one face more than another. Random binary values can be built up from tossing a single coin several times. Heads for a 1, tails for a 0. The way a coin lands depends on many variables outside your control.

Since computers operate with very precise regularity, they cannot produce a true random sequence. However, it is easy to produce a repeating *pseudo-random* sequence which has many characteristics similar to those of a true random sequence. How soon the sequence repeats depends on the precision to which the values are calculated. A good pseudo-random number generator working to 16-bit precision should produce 65536 different numbers before repeating.

Pseudo-random sequences are useful because every value occurs just once and apparently has no relation to the previously generated value. Programs can be tested with all possible data, in a way which doesn't rely on regularity, by going through an entire sequence of 'random' numbers.

The method of calculating pseudo-random numbers usually involves multiplying the previous number, or an initial value known as a *seed*, by a carefully chosen factor and then adding a constant – again chosen very carefully to eliminate undesirable effects. The result is then divided by another value and the remainder taken as the new random number.

Other generating methods can be used. For example, a sequence of binary numbers which exhibits random spread can be formed by manipulating individual bits from the previous number. Values read from independently changing hardware devices are sometimes used where speed or program length is more important than the degree of randomness.

Many pseudo-random number routines have appeared in *Sub Set* but, strangely, most of them have been for the Z-80. The routines in this chapter therefore consist of two routines contributed for the

RANDOM NUMBERS

6502, RND16B and RANDI, and three direct conversions from the Z-80 originals, RND16, RND32 and RND31. The discussion on random method owes nothing to the actual language in which the routines discussed were written.

RANDOM BITS VIA THE 6522

RND16B by T. A. Browning makes use of the two 16-bit counters of the 6522 Versatile Interface Adapter (VIA) to compute 16 random bits.

I cannot say just how random this method is. I suspect that if RND16B is used from inside a loop which operates in a regular, fixed number of clock cycles, then it might not be at all random. The loop iteration time could well coincide with the counter decrement time.

For one-off random number requests, RND16B does provide a very fast method of obtaining a 'non-calculated' value.

```
=====
:= RND16B    Compute 16 random bits.
=====
:JOB         To compute 16 random bits using a hardware timer
:            device with two independent 16-bit counters.
:ACTION      Set counters to decrement continuously without
:            interrupt.
:            Read contents of timer 2 into registers.
:            Exclusive-or hi-byte with lo-byte of timer 1.
:            Exclusive-or lo-byte with hi-byte of timer 1.
:            Write contents of registers to timer 2.
:-----
:CPU         6502
:HARDWARE    6522 Versatile Interface Adapter (VIA).
:SOFTWARE    None.
:-----
:INPUT       None.
:OUTPUT      A,Y contains an unknown (random) number.
:            Sign and Zero flags show status of byte in A.
:            VIA counters are free-running.
:            VIA port input latches are disabled.
:ERRORS      The randomness of the result depends entirely on
:            the regularity with which RND16B is called.
:REG USE     A Y P
:STACK USE   None.
:RAM USE     None.
:LENGTH     25
:CYCLES      38
:-----
:CLASS 2     -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
:=====
:
RND16B LDA #0      :Clear VIA ACR putting timers      A9 00
          STA ACR   :in one-shot mode.                8D lo hi
:
```

RANDOM NUMBERS

```

LDA T2C-L :Exclusive-or lo-bytes with      AD lo hi
EOR T1C-L :hi-bytes of other counter      4D lo hi
TAY       :with Y as low byte result      A8
LDA T2C-H :and A as high byte result.     AD lo hi
EOR T1C-L :                               4D lo hi
:
STY T2C-L :Write result into counter of    8C lo hi
STA T2C-H :timer 2, letting it decrement. 8D lo hi
RTS       :Exit, random bits in AY.       60
:=====

```

DOUBLY RANDOM

RANDI is a routine of mine which I originally wrote for the 6502 processor, converted to Z-80 and submitted to *Sub Set* as a 'double datasheet'.

The 6502 and Z-80 differ greatly in the way that BCD adjustment of arithmetic results takes place. Add and subtract operations on the Z-80 need a following 'DAA' instruction to adjust the value in the accumulator to BCD. Consequently, programming for 2-mode operation is neither easy nor quick since the routine has to test which mode is in force (I used the Carry flag to indicate this) and branch to the appropriate section.

The method of BCD adjustment on the 6502 – by having one bit in the status register determining whether binary or BCD arithmetic is currently being performed – is perfect for 2-mode routines. Although it normally carries the overheads of having to 'CLD' to ensure binary (or 'SED' to ensure decimal) operations are performed correctly, it does allow routines to be written which operate in either mode depending on the input state of this flag.

```

:=====
:= RANDI      Random Integer in binary or BCD.
:=====
:JOB          To compute a pseudo-random integer as a value in
:             the series:  $R(i+1) = (R(i) * a + c) \bmod m$ .
:             For binary: a = 257, c = 41 and m =  $2^{**}32$ .
:             For BCD: a = 101, c = 29 and m =  $10^{**}8$ .
:ACTION       Temp = constant.
:             FOR variable low order byte to high order byte:
:             [ Accumulator = temp.
:               Temp = byte.
:               Byte = byte + accumulator. ]
:-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
:-----
:INPUT        M0,1,2,3 contains seed or previous random value.
:             (Low order byte in M0, high order in M3.)
:             Decimal mode flag (D) set or reset accordingly.

```

RANDOM NUMBERS

```

:OUTPUT      MO,1,2,3 contains new random number.
:            X = 0. Y = high order byte of previous number.
:            P is unchanged.
:ERRORS      Arithmetic error if the random variable does not
:            contain valid BCD digits on entry with D = 1.
:REG USE     P X Y
:STACK USE   2
:RAM USE     MO M1 M2 M3
:LENGTH     20
:CYCLES      101
:-----
:CLASS 2     -discreet *interruptable *promable
:--****-     *reentrant *relocatable -robust
:=====
:
RVAR = MO    :4-byte stored random number variable.
CNST = $29   :Constant to be added to RVAR * 257.
              :if D = 0 (binary mode) CNST has the
              :value 41 (decimal), if D = 1 (decimal
              :mode) it has the value 29 (decimal).
:
RANDI PHP    :Save flags                                08
      PHA    :and A for use in RANDI.                    48
      CLC    :Ensure no carry to first add.              18
      LDY #CNST :Constant as lo-byte of shifted          A0 29
              :variable added to variable.
      LDX #-4  :Index from low order byte.                A2 FC
:
RNDLP TYA    :Get RVAR last byte, shifted.               98
      LDY RVAR+4,X :Pick up next RVAR byte to shift.    B4 M4
      ADC RVAR+4,X :Add in shifted byte to current      75 M4
      STA RVAR+4,X :byte, i.e. part RVAR * 257          95 M4
      INX      :Repeat for all four bytes of            E8
      BNE RNDLP :random variable.                      D0 F6
:
      PLA      :Restore A and P, leaving                68
      PLP      :X = 0 and Y = previous hi-byte.        28
      RTS      :Exit, new RVAR formed.                 60
:=====

```

RANDOM THEORY

The subject of random numbers provoked a lengthy discussion in *Sub Set* about the usefulness of certain constants.

The first formula based random number generator to appear in *Sub Set* was published in July 1981. It was based on the formula:

$$X_{i+1} = aX_i + C \bmod M$$

with

$$M = 2^{16} \text{ (i.e. 65536)}$$

$$C = \text{any odd number}$$

$$a = 1 \bmod 4 \text{ (1, 5, 9, 13, etc.)}$$

An 'a' of 257 was used in the routine, since it is an easy multiplier, but 4 other possible values of 'a' were given: 765, 889, 989 and 2009.

In October 1981, Brian Steel, referring to Knuth, *The Art of Computer Programming*, Volume 2, Chapter 3, wrote of two important points concerning the randomness of the method and values,

Firstly, in a mixed congruential sequence:

$$X_{i+1} = aX_i + C \bmod M$$

it is important that 'potency' is high. Potency is the power to which (a-1) must be raised before it is directly divisible by M (which it will always be eventually, if your other criteria are met). 257 is a bad 'a', since $a-1 = 256$ and $256^2 = 65536$ or 2^{16} : thus potency is only 2. If you choose 'a' as $5 \bmod 8$ you will ensure higher potency (e.g. 5, 13, 21, etc.).

Secondly, 'a' should be greater than $0.01M$ but less than $0.99M$ for good results. 257 is less than $0.01 * 2^{16}$ and therefore fails here too.

By observing these guidelines you have a better chance of getting a good PRANG [pseudo-random number generator], but sadly it can't be guaranteed on these (plus your) criteria alone. However, no good PRANG has a potency of less than 3, and 5 or more is desirable for this measurement.

In February 1982, *Sub Set* published another Z-80 datasheet which used an 'a' of 1021 and thus satisfied Brian Steel's potency requirement.

The contributor, Andrew Bain, also gave a simple method of calculating 'potency' when $M = 65536$. Continue to divide (a-1) by 2 until you obtain an odd number result. Divide the number of divisions made into 16 and the integer result is the potency.

In June 1982, Dr. Brian Ripley joined in the randomness argument,

You ask if there are better multipliers 'a'. There are! The best reference is indeed Knuth, but the second (1981) edition is needed. He stresses the importance of the 'spectral test' which identifies the problems with all generators known to be bad for other reasons. None of $a = 257, 765, 889, 989, 1021$ or 2009 passes acceptably. High potency is a necessary but not sufficient condition for a good generator. The choice of $a = 257$ is dreadful, and $a = 765$ and 1021 are bad. A search through multipliers from 1 to 2001 with $a = 1 \bmod 4$ (so the period is $M = 2^{16}$) suggests the following, in roughly decreasing order of merit; 293, 389, 1509, 249, 1785, 685.

RANDOM NUMBERS

Little seems to be known about the choice of C except that it should be odd. All the more powerful tests look at the whole sequence, which doesn't depend on C . The choice $C = 41$ seems as good as any.

A more serious point is whether 16 bits are good enough. It seems that for serious work they are not. The spectral test looks at the lattice of successive k -tuples $(X_i/M, X_{i+1}/M, \dots, X_{i+k}/M)$. In four dimensions the points of the lattice are at least $1/16$ apart, which is very different from the uniform scatter they should have. It would be worth the effort of producing a 32-bit generator. Try $a = 69069, 71365$ or 100485 .

Unfortunately this is a rather technical subject area in which many mistakes have been made. (See, e.g., P. J. Brown's comments in 'Writing Interactive Compilers and Interpreters'.) All I can say is that it is one of my research interests and the algorithm used is an improvement on Knuth's.

In the same issue doubts were cast by Ettrick Thomson on the randomness of the sequences produced by the algorithm when using a modulus which is a power of 2.

For 16-bit numbers and $M = 2^{16}$ ($M = 65536$), bit 15 has a period of 2^{16} , bit 14 a period of 2^{15} , and so on down to bit 0 which has a period of 2^1 . The least significant byte repeats every 256 numbers and all numbers in the sequence are alternatively odd and even.

16-BIT RANDOM

RND16 (a 6502 conversion of a Z-80 *Sub Set* datasheet) uses an 'a' of 1509, one of the numbers thought good by Dr. Brian Ripley.

The normal method used for 16-bit multiplication is quite slow but when the multiplier is always the same, it can be factorised and the multiplication carried out by binary shifting and addition which is relatively quick. 1509 factorises nicely to various nested powers of 2 and the routine executes in less than half the time it would take using 16-bit long multiplication.

```
=====
:= RND16      16-bit pseudo-random number generator.
=====
:JOB          To generate a 16-bit pseudo-random number using
:             the series,
:              $R(i+1) = (R(i) * 1509 + 41) \bmod 2^{*}16$ .
:ACTION       The multiplier, 1509, is factorised to powers of
:             2 to allow multiplication by shifting and binary
:             addition. 1509R is factorised,
:              $((32R + (16R - R)) * 8 + R) * 4 + R$ .
:             Taking only the low 2 bytes ensures  $\bmod 2^{*}16$ .
```

RANDOM NUMBERS

```

:-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
:-----
:INPUT        M0,1 contains the previous random value or seed.
:OUTPUT       M0,1 contains the new pseudo-random number.
:ERRORS       None.
:REG USE      None
:STACK USE    5
:RAM USE      M0 M1
:LENGTH      131
:CYCLES       259 or 263
:-----
:CLASS 1      *discreet *interruptable *promable
:*****      *reentrant *relocatable *robust
:=====
:
RNDV  =  M0      :2-byte stored random number variable.
COPY  =  M2      :2-bytes for storing last random number.
CNST  =  41      :Constant to add after multiplication.
:
RND16 PHP                :Save flags and                      08
      PHA                :accumulator for use in RND16.        48
      LDA COPY+1          :Save two bytes of page zero        A5 M3
      PHA                :to hold previous                    48
      LDA COPY            :random number for                  A5 M2
      PHA                :adding/subtracting.                48
      CLD                :Ensure binary arithmetic.           D8
:
      LDA RNDV            :Save previous                      A5 M0
      STA COPY            :random number (R)                  85 M2
      LDA RNDV+1          :in COPY for adding/subtracting    A5 M1
      STA COPY+1          :in multiplication by shifts.      85 M3
:
      ASL RNDV            :Shift left by 4 bits                06 M0
      ROL RNDV+1          :to multiply by 16                   26 M1
      ASL RNDV            :giving,                             06 M0
      ROL RNDV+1          :RNDV = R * 16.                     26 M1
      ASL RNDV            :                                   06 M0
      ROL RNDV+1          :                                   26 M1
      ASL RNDV            :                                   06 M0
      ROL RNDV+1          :                                   26 M1
:
      LDA RNDV+1          :Save 16R on stack (hi-byte)        A5 M1
      PHA                :and in A (lo-byte).                 48
      LDA RNDV            :                                   A5 M0
:
      ASL RNDV            :One bit left shift,                06 M0
      ROL RNDV+1          :RNDV = R * 32.                     26 M1
:
      CLC                :Prepare to add.                     18
      ADC RNDV            :Add R * 16 back                    65 M0
      STA RNDV            :in to R * 32                      85 M0
      PLA                :(getting hi-byte off stack)         68
      ADC RNDV+1          :giving,                             65 M1
      STA RNDV+1          :RNDV = R * 48.                     85 M1
:

```

RANDOM NUMBERS

SEC	:Prepare to subtract.	38
LDA RNDV	:Subtract saved R from	A5 M0
SBC COPY	:current RNDV	E5 M2
STA RNDV	:giving,	85 M0
LDA RNDV+1	:RNDV = R * 47.	A5 M1
SBC COPY+1	:	E5 M3
STA RNDV+1	:	85 M1
:		
ASL RNDV	:Shift left by 3 bits	06 M0
ROL RNDV+1	:to multiply by 8	26 M1
ASL RNDV	:giving,	06 M0
ROL RNDV+1	:RNDV = 47R * 8 or,	26 M1
ASL RNDV	:RNDV = R * 376.	06 M0
ROL RNDV+1	:	26 M1
:		
CLC	:Prepare to add.	18
LDA RNDV	:Add saved R to	A5 M0
ADC COPY	:current RNDV	65 M2
STA RNDV	:giving,	85 M0
LDA RNDV+1	:RNDV = R * 377.	A5 M1
ADC COPY+1	:	65 M3
STA RNDV+1	:	85 M1
:		
ASL RNDV	:Shift left by 2 bits	06 M0
ROL RNDV+1	:to multiply by 4 giving,	26 M1
ASL RNDV	:RNDV = 377R * 4 or,	06 M0
ROL RNDV+1	:RNDV = R * 1508.	26 M1
:		
CLC	:Prepare to add.	18
LDA RNDV	:Add saved R to	A5 M0
ADC COPY	:current RNDV	65 M2
STA RNDV	:giving,	85 M0
LDA RNDV+1	:RNDV = R * 1509.	A5 M1
ADC COPY+1	:	65 M3
STA RNDV+1	:	85 M1
:		
CLC	:Prepare to add.	18
LDA #CNST	:Add constant to	A9 29
ADC RNDV	:current RNDV	65 M0
STA RNDV	:giving,	85 M0
BNE ENDR16	:RNDV = R * 1509 + 41	D0 02
INC RNDV+1	:mod 2**16 because 2-bytes only.	E6 M1
:		
ENDR16 PLA	:Restore saved	68
STA COPY	:page zero	85 M2
PLA	:two bytes.	68
STA COPY+1	:	85 M3
PLA	:Restore accumulator	68
PLP	:and flags, then exit with	28
RTS	:new random number in RNDV.	60
:	=====	

32-BIT RANDOM

RND32 (again a 6502 conversion from the Z-80) uses another of Dr. Brian Ripley's good multipliers, 69069. This too factorises quite well and could have been dealt with as a sequence of shifts and additions

RANDOM NUMBERS

in the same way as RND16. However, as well as having a regular decimal pattern it has a repeating bit pattern making the use of a loop possible.

69069 in hexadecimal is \$10DCD and this splits to \$10101, which can be effected by byte shifting, and \$CCC which is achieved in RND32 by an iterative doubling process – this adds in an extra 1 in two consecutive iterations out of every four.

The loop count used is negative and counts upwards to zero because this produces a reset bit 1 in the count in the first 2 of every 4 iterations. A positive count-down would have reset bits in the first and last of every four iterations.

```

=====
:= RND32      32-bit pseudo-random number generator.
=====
:JOB          To generate a 32-bit pseudo-random number using
:              the series,
:               $R(i+1) = (R(i) * 69069 + 41) \bmod 2^{**}32$ .
:ACTION       Uses the factorisation of 69069 =
:               $3 * (2^{**}10 + 2^{**}6 + 2^{**}2) + 2^{**}16 + 2^{**}8 + 2^{**}0$ ,
:              to multiply by shift and addition.
:              Restricting the partial and final result to
:              4 bytes ensures "mod  $2^{**}32$ ".
-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
-----
:INPUT        M0 to M3 contains previous random value or seed.
:              (Low order byte in M0, high order in M3.)
:OUTPUT       M0 to M3 contains the new pseudo-random number.
:ERRORS       None.
:REG USE      None.
:STACK USE    7
:RAM USE      M0 M1 M2 M3
:LENGTH       138
:CYCLES       808
-----
:CLASS 1      *discreet *interruptable *promable
:*****      *reentrant *relocatable *robust
=====
:
RNUM  =  M0      :4-byte stored random variable.
CORN  =  M4      :4-byte store for random number copy.
CNSTNT = 41      :Constant to be added in.
:
RND32  PHP      :Save flags,                      08
      PHA      :accumulator                      68
      TXA      :and index X                      8A
      PHA      :for use in RND32.                68
      CLD      :Ensure binary arithmetic.        D8
:

```

RANDOM NUMBERS

	LDX #0	:Index from low order bytes.	A2 00
RPSHLP	LDA CORN,X	:Save 4 bytes of page zero	B5 M4
	PHA	:on stack and	68
	LDA RNUM,X	:copy previous random number	B5 M0
	STA CORN,X	:(R) to it for adding in.	95 M4
	INX	:Repeat for low order	E8
	CPX #4	:to high order bytes	E0 04
	BNE RPSHLP	:of RNUM and CORN.	D0 F4
:			
	LDX #-11	:Set loop counter, 11 passes.	A2 F5
SHFTLP	ASL RNUM	:RNUM = RNUM * 2.	06 M0
	ROL RNUM+1	:Loop will produce: RNUM =	26 M1
	ROL RNUM+2	:3R * (2**10 + 2**6 + 2**2).	26 M2
	ROL RNUM+3	:	26 M3
	TXA	:If bit 1 of loop count	8A
	AND #2	:is set (1) then	29 02
	BNE SLINX	:skip to loop end test.	F0 19
:			
	CLC	:Else, prepare to add,	18
	LDA RNUM	:RNUM = RNUM + R.	A5 M0
	ADC CORN	:	65 M4
	STA RNUM	:(This occurs on passes	85 M0
	LDA RNUM+1	:1, 4, 5, 8 and 9.	A5 M1
	ADC CORN+1	:Were X to count down	65 M5
	STA RNUM+1	:from +11 then it	85 M1
	LDA RNUM+2	:would occur on passes	A5 M2
	ADC CORN+2	:3, 4, 7, 8 and 11	65 M6
	STA RNUM+2	:giving an incorrect	85 M2
	LDA RNUM+3	:factor of 2866	A5 M3
	ADC CORN+3	:instead of the	65 M7
	STA RNUM+3	:correct 3276.)	85 M3
:			
SLINX	INX	:Repeat 11 times, ending	E8
	BNE SHFTLP	:with RNUM = 3276R.	D0 D7
:			
	CLC	:Prepare to add.	18
	LDA RNUM	:Add R giving,	A5 M0
	ADC CORN	:RNUM = RNUM + R * 2**0.	65 M4
	STA RNUM	:	85 M0
	LDA RNUM+1	:This and the next two	A5 M1
	ADC CORN+1	:additions will produce	65 M5
	STA RNUM+1	:RNUM = RNUM +	85 M1
	LDA RNUM+2	:R * (2**0 + 2**16 + 2**8) +	A5 M2
	ADC CORN+2	:constant 41.	65 M6
	STA RNUM+2	:	85 M2
	LDA RNUM+3	:	A5 M3
	ADC CORN+3	:	65 M7
	STA RNUM+3	:	85 M3
:			
	CLC	:Prepare to add.	18
	LDA RNUM+2	:Add low 2 bytes of R	A5 M2
	ADC CORN	:into high 2 bytes of RNUM	65 M4
	STA RNUM+2	:giving,	85 M2
	LDA RNUM+3	:RNUM = RNUM + R * 2**16.	A5 M3
	ADC CORN+1	:	65 M5
	STA RNUM+3	:	85 M3
:			

RANDOM NUMBERS

```

CLC                :Prepare to add.                18
LDA RNUM           :Add constant to lo-byte RNUM,   A5 M0
ADC #CNSTNT        :byte 0,R to byte 1,RNUM,       69 29
STA RNUM           :byte 1,R to byte 2,RNUM        85 M0
LDA RNUM+1         :and                             A5 M1
ADC CORN           :byte 2,R to byte 3,RNUM        65 M4
STA RNUM+1         :giving,                        85 M1
LDA RNUM+2         :RNUM = RNUM + R * 2**8 + 41.     A5 M2
ADC CORN+1         :                               65 M5
STA RNUM+2         :                               85 M2
LDA RNUM+3         :                               A5 M3
ADC CORN+2         :                               65 M6
STA RNUM+3         :                               85 M3
:
LDX #4            :Index from high order byte.     A2 04
RPULLP PLA        :Restore saved page zero         68
STA CORN-1,X      :bytes used for R copy.          95 M3
DEX              :Repeat for high order           CA
BNE RPULLP        :byte to low order.             D0 FA
:
PLA              :Restore                         68
TAX              :registers                      AA
PLA              :and                           68
PLP              :flags used in RND32 then exit    28
RTS              :with new random number in RNUM.  60
:=====

```

31-BIT RANDOM

The last of these conversions from the Z-80, RND31, solves the periodicity problem Ettrick Thomson found to occur when using a modulus that is a power of 2.

The algorithm of RND31, which rotates the top 9 bits with the bottom 22 bits, is slightly difficult to relate to the formula:

$$R(i+1) = R(i) * (2^9 + 1) \bmod (2^{31} - 1)$$

but it does indeed work and is very fast. The sequence repeats after $(2^{31} - 2)$ numbers, giving all possible values except 0 and \$7FFFFFFF.

The primitive root $(2^9 + 1)$ may be too small to produce good random numbers. A better choice would be $(2^{17} + 1)$. You can use the algorithm of RND31 for this larger element, simply rotate 2 bytes + 1 bit from high order to low order instead of the 1 byte + 1 bit that is rotated in RND31.

```

:=====
:= RND31          31-bit pseudo-random number generator.
:=====
:JOB            To generate a 31-bit pseudo-random number using
:              the series,
:               $R(i+1) = (R(i) * a) \bmod m$ ,
:               $m = 2^{**}31 - 1$  (a Mersenne prime)
:               $a = 2^{**}9 + 1$  (a primitive root of m).
:

```

RANDOM NUMBERS

```
:ACTION      Move RN bits 21 - 0 into TEMP bits 30 - 9.
:            Move RN bits 30 - 22 into TEMP bits 8 - 0.
:            Clear TEMP bit 31.
:            RN = RN + TEMP.
:            IF bit 31,RN = 1 THEN:
:            [ Bit 31,RN = 0. RN = RN + 1. ]
```

```
-----
:CPU         6502
:HARDWARE    None.
:SOFTWARE    None.
-----
```

```
:INPUT       RN contains previous random number or seed.
:OUTPUT      RN contains new random number.
:            P, A, X and M0, M1, M2, M3 are changed.
:ERRORS      Arithmetic error if D = 1 (decimal mode).
:            A re-entering routine could overwrite a
:            partially completed new random number resulting
:            in an out-of-sequence number being generated by
:            the re-entered routine.
:REG USE     P A X
:STACK USE   None.
:RAM USE     M0 M1 M2 M3
:            "RN" - 4 bytes containing 31-bit pseudo-random
:            number, not necessarily appended to RND31.
:LENGTH     71 (not including 4-byte RN).
:CYCLES      Minimum: 143. Maximum: 207.
```

```
-----
:CLASS 2     -discreet *interruptable -promable
:--*-----  -reentrant -relocatable -robust
=====
```

```
:
TEMP = MO :4-byte temporary store for rotated RN
:         before adding back to RN.
RN .BYTE dd :31-bit seed for pseudo-random number.
:         .BYTE cc :Stored anywhere in memory.
:         .BYTE bb :High order byte at
:         .BYTE aa :RN+3, must be less than 128 ($80).
```

```
:
:...1st step - get RN bits 21 - 0 into TEMP bits 30 - 9,
:...clearing TEMP bits 31 and 8.
```

```
RND31 LDA RN :Move RN bits 7 to 0 into AD Lo hi
      ASL A :Carry and TEMP bits 15 to 9, 0A
      STA TEMP+1 :clearing TEMP bit 8 for 85 M1
      LDA RN+1 :later receiving RN bit 30. AD Lo hi
      ROL A :Move RN 15 to 8 and C 2A
      STA TEMP+2 :into C and TEMP 23 to 16. 85 M2
      LDA RN+3 :Move RN bits 21 to 16 and C AD Lo hi
      ROL A :into TEMP bits 30 to 24, 2A
      AND #$7F :clearing TEMP 31. 29 7F
      STA TEMP+3 :22 bits moved to TEMP. 85 M3
```

RANDOM NUMBERS

```

: ...Next step - get RN bits 30. - 22 into TEMP bits 8 - 0,
: ...giving completed rotation in TEMP.
    LDA RN+4      :Move RN bits 31 to 24      AD lo hi
    STA TEMP      :into TEMP bits 7 to 0.      85 M0
    LDA RN+3 2    :Get RN byte containing bits AD lo hi
    ROL A         :23 and 22. Shift left twice 2A
    ROL TEMP      :discarding unused hi-bit     26 M0
    ROL A         :so TEMP bits 7 to 0 get      2A
    ROL TEMP      :RN bits 29 to 22 and         26 M0
    BCC RNADD     :RN bit 30 goes to TEMP       90 02
    INC TEMP+1    :bit 8.                      E6 M1
:
: ...Add rotated value to original, exit if < (2**31 - 1).
RNADD CLC        :Prepare for addition.        18
    LDX #-4       :Index from low order bytes. A2 FC
RNADLP LDA RN-252,X :Do 32-bit addition of      BD lo hi
    ADC TEMP+4,X  :rotated number in TEMP      75 M4
    STA RN-252,X  :added to original in RN     9D lo hi
    INX          :with result to RN.           E8
    BNE RNADLP    :                          D0 F5
    TAX          :Test high order byte for     AA
    BPL RNEXIT    :bit 31 set, exit if not.    10 0F
:
: ...Correct for RN > (2**31 - 1). RN = (2**31 - 1) is not
: ...possible so correction will not set bit 31.
    AND #$7F      :Clear bit 31 of new random 29 7F
    STA RN+3      :number in RN.              8D lo hi
    LDX #0        :Index from low order byte. A2 00
RNINC INC RN,X    :Increment RN byte,         FE lo hi
    BNE RNEXIT    :exit if no carry,          D0 03
    INX          :else index next higher byte E8
    BNE RNINC     :and repeat until no carry. D0 F8
:
RNEXIT RTS       :Exit, RN = new random value. 60
:=====

```


Extracting Roots

13

The four routines in this chapter will find only the integer square or cube roots of 16-bit or 32-bit values. The root returned from them is that of the largest square or cube less than or equal to the input value. The remainder (input value minus largest square/cube) is also returned.

The methods used can be extended to calculate roots to any precision of binary fraction. To do this, continue the iterative process (making sure that your page zero variables are lengthened to take the longer results and prevent overflow) for as many root bits as you need. The low order bits in excess of the numbers quoted in each routine will be the binary fraction.

SQUARE ROOTS

SQR15 and SQR16 are different entry points to a routine by John Kerr which calculates the 8-bit integer square root and 9-bit remainder (square – root²) of an input 16-bit square. When entered at SQR15, an initial test is made on the sign of the input value and if found to be negative then the routine aborts. SQR16 acts on a full 16-bit unsigned, or absolute, value.

SQR31 and SQR32 also by John Kerr do the same job as SQR15 and SQR16 but on 32-bit signed (positive) and unsigned values. The integer root of a 32-bit square is 16 bits and the remainder 17 bits.

An analysis by John Kerr of the root extraction method is given later in this chapter.

```
=====
:= SQR15    Square root of 16-bit signed (positive) value.
:> SQR16    Square root of 16-bit unsigned (absolute) value.
=====
:JOB        To extract the integer square root of a 16-bit
:           value, either unsigned or signed (negative
:           values returned unprocessed).
```

EXTRACTING ROOTS

```

:ACTION      IF positive value:
:            THEN:
:            [ Initialise part-root to 0.
:              FOR root bits count, 8 TO 1:
:                [ Subtrahend = part-root * 2**(count * 2)
:                  + 1 * 2**((count - 1) * 2).
:                IF subtrahend =< square:
:                  THEN: [ Square = square - subtrahend.
:                        Part-root = part-root * 2 + 1. ]
:                  ELSE: [ Part-root = part-root * 2. ] ]
:                Set root-done flag. ]
:            ELSE: [ Set root-not-done flag. ]
:-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
:-----
:INPUT        M1,2 contains the 16-bit square.
:OUTPUT       SQR15: N = 1: Negative input value - no root.
:              A = M2. P X Y M1 M2 M3 M4 unchanged.
:              N = 0: As for SQR16.
:              SQR16: N = 0. C = 0. Y = 0. A = X = M3.
:              8-bit integer square root in M1,2 (M2 = 0).
:              9-bit remainder in M3,4 (bits 7-1,M4 = 0).
:ERRORS       Arithmetic error if D = 1 (decimal mode).
:REG USE      P A X Y
:STACK USE    None.
:RAM USE      M1 M2 M3 M4
:LENGTH      64
:CYCLES       Maximum 615.
:-----
:CLASS 2      -discreet *interruptable *promable
:--****-      *reentrant *relocatable -robust
:=====
:
:...Assign I/O page zero variables.
SQR   =   M1       :2-byte stored square (input) or 2-byte
                  :root with hi-byte = 0 (output).
REM   =   M3       :2-byte store for output 9-bit remainder.
:
:...Assign working pseudo-registers in page zero to coincide
:...with I/O and thus reduce initial & terminal data moves.
PARTRT = M1       :For working high byte of subtrahend, the
                  :same as partial root at each stage.
RSQLO  = M3       :For lowest byte of 3-byte accumulator
                  :containing square for 2-bit shifting.
COUNT = M4       :For root bits count.
:
SQR15  LDA  SQR+1  :Test sign of input square, exit      A5 M2
        BMI  EXIT  :immediately if negative.             30 38
:
SQR16  TAY        :Transfer input to low two bytes      A8
        LDA  SQR   :of reducing square accumulator     A5 M1
        STA  RSQLO :in X, Y, M3, initialising high     85 M3
        LDX  #0    :byte X to 0.                       A2 00
        STX  PARTRT:initialise part-root to 0.          86 M1
        LDA  #8    :Set up count of 8 in count         A9 08
        STA  COUNT :register for 8-bit root.            85 M4

```

EXTRACTING ROOTS

```

PAIRLP CPX PARTRT :Test if subtrahend will subtract      E4 M1
      BCC NEXTPR :from reducing square... first          90 0E
      BNE PRTSUB :test on hi-byte (part-root), then      D0 04
      CPY #$40   :on constant set bit at 2-bits         C0 40
      BCC NEXTPR :below current part-root.              90 08
:
PRTSUB TYA      :Subtrahend is less than square          98
      SBC #$40   :so subtract it from current            E9 40
      TAY      :part of reducing square. Carry          A8
      TXA      :was set on entry to subtraction         8A
      SBC PARTRT :by test. Carry will be set on          E5 M1
      TAX      :exit by subtraction.                    AA
:
NEXTPR ROL PARTRT :Do part-root * 2 + result of sub.     26 M1
      ASL RSQLO  :Shift left reducing square            06 M3
      TYA      :in X, Y, M3 to bring next              98
      ROL A     :pair of bits from square               2A
      TAY      :into high byte. After 8                 A8
      TXA      :iterations the low two bytes           8A
      ROL A     :will be zero and the remainder         2A
      TAX      :of the root extraction will be          AA
      ASL RSQLO  :in Carry and the high byte X.         06 M3
      TYA      :                                         98
      ROL A     : (Count in M4 will have                 2A
      TAY      :reduced to zero, leaving it              A8
      TXA      :ready to be remainder high byte         8A
      ROL A     :by rotating remainder high bit          2A
      TAX      :in from Carry flag.)                    AA
      DEC COUNT :Repeat until 8-bit                      C6 M4
      BNE PAIRLP :root found.                            D0 D8
:
      STX REM    :Move 9-bit remainder to 2-byte         86 M3
      ROL REM+1  :output variable and clear hi-byte     26 M4
      STY SQR+1  :of root, 8-bit result in lo-byte.     84 M2
EXIT   RTS      :Exit, root done or N = 1.              60
:=====

```

```

:=====
:= SQR31   Square root of 32-bit signed (positive) value.
:> SQR32   Square root of 32-bit unsigned (absolute) value.
:=====
:JOB       To extract the integer square root of a 32-bit
:          value, either unsigned or signed (negative
:          values returned unprocessed).
:ACTION    IF positive value:
:          THEN:
:          [ Initialise part-root to 0.
:            FOR root bits count, 16 TO 1:
:            [ Subtrahend = part-root * 2** (count * 2)
:              + 1 * 2** ((count - 1) * 2).
:            IF subtrahend <= square:
:            THEN: [ Square = square - subtrahend.
:                  Part-root = part-root * 2 + 1. ]
:            ELSE: [ Part-root = part-root * 2. ] ]
:            Set root-done flag. ]
:          ELSE: [ Set root-not-done flag. ]

```

EXTRACTING ROOTS

```

:-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
:-----
:INPUT        M1,2,3,4 contains the 16-bit square.
:OUTPUT       SQR31: N = 1: Negative input value - no root.
:             A, X, Y and M1 to MA are unchanged.
:             N = 0: As SQR32.
:             SQR32: N = 0. C = 0. A = M1. Y = X = 0.
:             M9 = M1. MA = M2.
:             16-bit integer square root in M1,2,3,4.
:             17-bit remainder in M5,6,7,8.
:ERRORS       Arithmetic error if D = 1 (decimal mode).
:REG USE      P A X Y
:STACK USE    None.
:RAM USE      M1 to MA
:LENGTH      75
:CYCLES       Maximum 4350.
:-----
:CLASS 2      -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
:=====
:...Assign I/O page zero variables.
SQR1  =  M1      :4-byte stored square (input) or 4-byte
                :root with hi-word = 0 (output).
REML  =  M5      :4-byte store for 17-bit remainder out.
:
:...Assign working pseudo-registers in page zero to coincide
:...with I/O and thus reduce initial & terminal data moves.
RSQ   =  M1      :For 6-byte accumulator containing
                :square reduced by shifts and subtracts.
MND   =  M4      :For working 3-byte minuend, highest 3
                :bytes of RSQ.
SND   =  M8      :For working 3-byte subtrahend, highest
                :2-bytes contain the partial root at
                :each stage. Lowest is constant $40.
:
SQR31 BIT SQR1+3 :Test sign of input square and      24 M4
      BMI EXITL  :exit immediately if negative.      30 46
:
SQR32 LDA #0     :Else prepare to clear working      A9 00
      LDX #6     :registers.                        A2 06
CLEAR STA MND,X  :Clear MA,9 for partial root,      95 M4
      DEX        :M7 for output remainder,          CA
      BNE CLEAR  :M6,5 for hi-word of RSQ.           D0 FB
:
      LDA #$40   :Initialise constant set bit        A9 40
      STA SND    :in SND lowest byte.                85 M8
      LDY #16    :Set count for 16-bit root.          A0 10
:
PRLOOP LDX #3     :Index from highest bytes.          A2 03
SUBTST LDA MND-1,X :test if subtrahend will           B5 M3
      CMP SND-1,X :subtract from minuend, skipping    D5 M7
      BCC RSLT    :out appropriately, but            90 10
      BNE SUBPRT  :looping for as many of the         D0 03
      DEX        :3 bytes as necessary.              CA
      BNE SUBTST  :                                  D0 F5

```

EXTRACTING ROOTS

```

:
SUBPRT LDX #-3      :Index from lowest bytes.          A2 FD
SUBLP  LDA MND+3,X  :Subtraction will go, so           B5 M7
      SBC SND+3,X   :subtract part-root + constant     F5 MB
      STA MND+3,X   :from shifted part-square.        95 M7
      INX           :C = 1 on entry by test, C = 1      E8
      BNE SUBLP     :on exit by subtraction.           D0 F7
:
RSLT   ROL SND+1    :Do part-root * 2 + result of      26 M9
      ROL SND+2     :subtraction. Leave C = 0.         26 MA
:
      TYA           :Ensure bit 7,A = 0.              98
NPAIR  LDX #-6      :Index from low order byte.       A2 FA
NBIT   ROL RSQ+6,X  :Shift reducing square left       36 M7
      INX           :by one bit into minuend.         E8
      BNE NBIT      :Repeat for all 6 bytes of RSQ.    D0 FB
      EOR #$80      :Flip sign bit in A, so shift     49 80
      BMI NPAIR     :done twice for bit-pair.         30 F5
:
      DEY           :Repeat until 16-bit root found,   88
      BNE PRLLOOP   :Leave C = remainder bit 16.       D0 D5
:
      LDA SND+2     :Transfer found root from          A5 MA
      STA SQRL+1    :subtrahend hi-word to output     85 M2
      LDA SND+1     :variable lo-word. Output hi-word A5 M9
      STA SQRL      :cleared by RSQ shifting.         85 M1
      STY REML+3    :REML,REML+1 = MND+1,MND+2, clear 84 M8
      ROL REML+2    :hi-byte, store remainder bit 16. 26 M7
EXITL  RTS         :Exit, root done or N = 1.         60
:=====

```

CUBE ROOTS

CURT16 and CURT32 are 6502 versions of original Z-80 *Sub Set* routines. They use the cube root method developed by John Kerr to calculate the integer cube roots of 16-bit and 32-bit unsigned values respectively. You can easily adapt both for signed number work by adding an initial test on the sign of the input value (bit 7 of the high order byte) and aborting if the bit is set, as in SQR31.

John Kerr's cube root extraction method is similar to that used in the square root programs but with two major differences. 3 bits instead of 2 are processed in each iteration, except in the first iteration when only 1 bit (CURT16) or 2 bits (CURT32) are shifted in to the accumulator, neither 16 nor 32 being a multiple of 3. The partially formed cube root cannot be used directly as the subtrahend.

Here is John's account of how he adapted the square root method to work for cube roots. As well as explaining why the cube root subtrahend requires such an awkward adjustment, it offers a possible way of constructing algorithms for other mathematical functions.

EXTRACTING ROOTS

In the algorithm for square root extraction, the input number is shifted left, 2 bits at a time, into a working accumulator. At some arbitrary stage in the process, the 'virtual input' is that part of the input which has been shifted in and operated upon; call it 'x'. The results of processing, so far, are a partial result 's' for the square root, and a partial remainder 'r' in the working accumulator:

$$\begin{aligned}\text{virtual input} &= x \\ \text{partial result} &= s \\ \text{partial remainder} &= r\end{aligned}$$

these quantities being related by the equation:

$$s^2 + r = x$$

After the next round of processing, the virtual input has been augmented by the 2-bit number 'y' (the next 2 bits of the real input). The next least significant bit of the result has been found; call it 'd', so the new partial result is $2s + d$. the remainder has also changed:

$$\begin{aligned}\text{virtual input} &= 4x + y \\ \text{partial result} &= 2s + d \\ \text{partial remainder} &= q\end{aligned}$$

So the previous equation now becomes:

$$(2s + d)^2 + q = 4x + y$$

Combine the two equations to eliminate 'x', remembering that as 'd' is a single bit (zero or one), d^2 reduces to d. The result is:

$$q = 4r + y$$

when $d=0$, indicating that nothing has been subtracted from the working accumulator; or else,

$$q = 4r + y - (4s+1)$$

when $d=1$, i.e. the result bit is set after a subtraction.

This shows that the required subtrahend is $4s+1$; that is, the previous partial result shifted 2 places left, then incremented.

The same analysis can be applied to a hypothetical algorithm for extracting cube roots. It yields the subtrahend value which should be used, if the method is to be sound. Starting again at some arbitrary point during execution, we have:

virtual input = x
 partial result = c
 partial remainder = r

this time related by the equation:

$$c^3 + r = x$$

Now the next 3 bits of the input number are shifted into the working accumulator, giving a new virtual input of $8x+y$. After processing, a new result bit 'd' and a new partial remainder have been calculated:

virtual input = $8x + y$
 partial result = $2c + d$
 partial remainder = q

and the relevant equation is:

$$(2c+d)^3 + q = 8x + y$$

As before, the two equations are combined to eliminate the unknown 'x', and the result examined assuming $d=0$, then $d=1$. The first case (subtraction failed) gives:

$$q = 8r + y$$

The second gives:

$$q = 8r + y - (12c^2 + 6c + 1)$$

This does not augur well for the direct method of cubic root extraction; the required subtrahend is an ugly quadratic function of the 'result-so-far', which looks like it will require recalculation, squaring and all, after each round of processing. Luckily, it doesn't: because the subtrahend can be built up progressively, from an initial non-zero 'c' of 1 (so $c^2 = 1$), some multiple of ' c^2 ' always exists in the previous subtrahend. After a little more algebra, the cube root algorithm, devoid of full-length multiplication, is obtained.

The algorithm is shown in the documentation ACTION section of CURT16 and CURT32.

EXTRACTING ROOTS

```

=====
:= CURT16    Cube Root of a 16-bit unsigned integer value.
=====
:JOB          To extract the integer cube root and remainder
:             of a 16-bit unsigned integer.
:ACTION       Clear part-root, remainder and subtrahend.
:             FOR root bits count 6 TO 1:
:             [ IF count = 6
:             THEN: [ Shift 1 cube bit into remainder. ]
:             ELSE: [ Shift 3 cube bits into remainder. ]
:             IF remainder > subtrahend:
:             THEN:
:             [ Remainder = remainder - (subtrahend + 1).
:             Part-root = part-root * 2 + 1.
:             Subtrahend = subtrahend * 4
:             + part-root * 18. ]
:             ELSE:
:             [ Part-root = part-root * 2.
:             Subtrahend = subtrahend * 4
:             - part-root * 6. ] ]
:
-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
-----
:INPUT        M0,1 contains the 16-bit cube.
:OUTPUT       M0,1 is unchanged.
:             M4,5 contains the 6-bit integer cube root.
:             M2,3 contains the 13-bit remainder.
:ERRORS       None.
:REG USE      None.
:STACK USE    7
:RAM USE      M0 to M5
:LENGTH       157
:CYCLES       Minimum: 1403. Maximum: 1431.
-----
:CLASS 1      *discreet *interruptable *promable
:*****       *reentrant *relocatable *robust
=====
CUBE   =   M0           :2-byte stored cube, input and output.
REM    =   M2           :2-byte store for working minuend and
                        :output 13-bit remainder.
ROOT   =   M4           :2-byte store for working part-root and
                        :output 6-bit cube root (M5 = 0).
COUNT =   M5           :1-byte for loop count.
SND    =   M6           :2-bytes for working subtrahend.
:
CURT16 PHP           :Save all                      08
      PHA           :flags                          48
      TXA           :and                            8A
      PHA           :registers                       48
      TYA           :to be                          98
      PHA           :used in routine.                48
      LDA SND+1      :Save two bytes                  A5 M7
      PHA           :of page zero                     48
      LDA SND        :for use as                      A5 M6
      PHA           :subtrahend.                      48
      CLD           :Ensure binary arithmetic.        D8

```

EXTRACTING ROOTS

```

:
LDA #0 :Prepare to clear 6 bytes for A9 00
LDX #6 :workspace and variables. A2 06
CLRLP STA REM-1,X :M7 & M6 for subtrahend, 95 M1
DEX :M5 & M4 for root (M5 also count), CA
BNE CLRLP :M3 & M2 for remainder. D0 FB

:
LDY #6 :Initialise count for A0 06
STY COUNT :6-bit root. 84 M5
LDX #1 :Do only 1 shift, 1st iteration. A2 01

:
CLOOP ASL CUBE :Shift next cube bit 06 M0
ROL CUBE+1 :out into Carry flag and 26 M1
BCC REMIN :rotate back in to cube so it 90 02
INC CUBE :is unchanged on exit. E6 M0

:
REMIN ROL REM :Shift cube bit 26 M2
ROL REM+1 :up into remainder. 26 M3
DEX :Do 1 bit first iteration, CA
BNE CLOOP :3 bits in other five. D0 F1

:
ASL ROOT :New part-root if no subtraction. 06 M4
LDA REM :Try to subtract subtrahend A5 M2
SBC SND :from remainder (in each E5 M6
STA REM :iteration, remainder is 85 M2
LDA REM+1 :last remainder * 8 + next set A5 M3
SBC SND+1 :of three bits from cube). E5 M7
STA REM+1 :But if subtraction result is 85 M3
BCC ADBACK :negative then go add it back. 90 09

:
INC ROOT :Else show result in part-root. E6 M4
LDA ROOT :Get part-root * 3 in A, A5 M4
ASL A :ready for later multiplying 0A
ADC ROOT :by 6 to give part-root * 18). 65 M4
BCC NEWSND :Branch always to form new SND. 90 0F

:
ADBACK SEC :Add current subtrahend 38
LDA REM :back to remainder A5 M2
ADC SND :and restore it to 65 M6
STA REM :the value held 85 M2
LDA REM+1 :before subtraction. A5 M3
ADC SND+1 :Add takes it past $FFFF so 65 M7
STA REM+1 :Carry will be set at NEWSND. 85 M3
LDA ROOT :Get part root for multiply by 6. A5 M4

:
NEWSND PHP :Save add/subtract flag (C). 08
ASL SND :Shift SND by one bit to 06 M6
ROL SND+1 :multiply by 2. 26 M7
LDY SND :Save it in XY for later A4 M6
LDX SND+1 :add/subtract of root * n. A6 M7

:
STA SND :SND = part-root * 1 (or 3). 85 M6
ASL A :A = p-r * 2 (or 6) with overflow 0A
ROL SND+1 :bit saved in SND+1. Clears Carry. 26 M7
ADC SND :SND = lo-byte result of 65 M6
STA SND :p-r * 3 (or 9). 85 M6
LDA SND+1 :Get "* 2 (or 6)" overflow bit A5 M7
AND #1 :without changing Carry and add 29 01

```

EXTRACTING ROOTS

```

ADC #0      :carry from lo-byte result,      69 00
STA SND+1   :M6,7 = part-root * 3 (or 9).    85 M7
PLP         :Restore add/subtract flag      28
BCS SNDSUB  :and do appropriate action.     80 0C
:
TYA         :Add part-root * 9              98
ADC SND     :to subtrahend * 2              65 M6
STA SND     :putting result                 85 M6
TXA         :back to                       8A
ADC SND+1   :subtrahend variable, then      65 M7
STA SND+1   :branch always to shift effecting 85 M7
BCC SNDDBL  : "SND = SND * 4 * ROOT * 18".  90 0A
:
SNDSUB TYA   :Subtract part-root * 3        98
SBC SND     :from subtrahend * 2            E5 M6
STA SND     :putting result back           85 M6
TXA         :to subtrahend variable.       8A
SBC SND+1   :Following shift will effect   E5 M7
STA SND+1   : "SND = SND * 4 - ROOT * 6".   85 M7
:
SNDDBL ASL  SND :Shift to multiply by 2, and 06 M6
ROL  SND+1   :complete new subtrahend.     26 M7
:
LDX #3      :Shift 3 bits into REM next time. A2 03
DEC COUNT   :Repeat for 6-bit cube root,   C6 M5
BNE CLOOP   :leaving M5 = 0 as root hi-byte. D0 8C
:
PLA         :Restore two                   68
STA SND     :page zero bytes               85 M6
PLA         :used for                     68
STA SND+1   :subtrahend.                  85 M7
PLA         :Restore                      68
TAY         :all                          A8
PLA         :registers                    68
TAX         :and                          AA
PLA         :flags,                       68
PLP         :then exit with integer cube root, 28
RTS         :remainder and intact cube.    60
:=====
:=====
:= CURT32   Cube Root of a 32-bit unsigned integer value.
:=====
:JOB        To extract the integer cube root and remainder
:           of a 32-bit unsigned integer.
:ACTION     Clear part-root, remainder and subtrahend.
:           FOR root bits count 11 TO 1:
:           [ IF count = 11
:           THEN: [ Shift 2 cube bits into remainder. ]
:           ELSE: [ Shift 3 cube bits into remainder. ]
:           IF remainder > subtrahend:
:           THEN:
:           [ Remainder = remainder - (subtrahend + 1).
:           Part-root = part-root * 2 + 1.
:           Subtrahend = subtrahend * 4
:           + part-root * 18. ]

```

EXTRACTING ROOTS

```

:      ELSE:
:      [ Part-root = part-root * 2.
:        Subtrahend = subtrahend * 4
:          - part-root * 6. ] ]
-----
:CPU      6502
:HARDWARE  None.
:SOFTWARE  Local subroutine use, acting on page zero
:          at X to X+3 (one argument),
:          "ASL4", "ROL4", "LSR4", "ROR4",
:          and X+4 to X+7 (2nd argument),
:          "ADD4", "ADC4", "SUB4", "SBC4".
-----
:INPUT     M0,1,2,3 contains the 32-bit cube.
:OUTPUT    M0,1,2,3 is unchanged.
:          M8,9,A,B contains the 11-bit integer cube root.
:          M4,5,6,7 contains the 23-bit remainder.
:ERRORS    None.
:REG USE   None.
:STACK USE  11
:RAM USE    M0 to MB
:LENGTH    187 (CURT32 = 139, Subroutines = 48).
:CYCLES     Minimum: 8599. Maximum: 9420.
-----
:CLASS 2   *discreet *interruptable *promable
:****-*    *reentrant -relocatable *robust
=====
CUBEL = M0      :4-byte stored cube, input and output.
REML  = M4      :4-byte store for working minuend and
                :output 23-bit remainder.
ROOTL = M8      :4-byte store 11-bit root output.
WSND  = M8      :4-bytes for working subtrahend.
WPRT  = MC      :4-bytes for working part-root.
:
CURT32 PHP      :Save all                      08
      PHA      :flags                        48
      TXA      :and                          8A
      PHA      :registers                     48
      TYA      :to be                        98
      PHA      :used in routine.              48
      CLD      :Ensure binary arithmetic.     D8
:
      LDX #4    :Index from high byte.         A2 04
PSHWS LDA WPRT-1,X :Save 4 bytes of page       B5 MB
      PHA      :zero workspace MF to MC        48
      DEX      :for use as working             CA
      BNE PSHWS :part-root.                    D0 FA
:
      LDA #0    :Clear 12 bytes for            A9 00
      LDX #12   :workspace and variables.     A2 0C
CLRVLV STA REML-1,X :MF,E,D,C for part root,   95 M3
      DEX      :MB,A,9,8 for subtrahend,       CA
      BNE CLRVLV :M7,6,5,4 for remainder.     D0 FB
:
      LDA #11   :Count for 11-bit root.        A9 0B
      LDY #2    :Get 2 cube bits, first time.  A0 02

```

EXTRACTING ROOTS

:			
CUBELP	PHA	:Save count.	48
NXTBIT	LDX #CUBEL	:Shift next cube bit	A2 M0
	JSR ASL4	:out into Carry flag and	20 lo hi
	BCC ROTREM	:rotate back in to cube so it	90 02
	INC CUBEL	:is unchanged on exit.	E6 M0
:			
ROTREM	LDX #REML	:Shift cube bit	A2 M4
	JSR ROL4	:up into remainder.	20 lo hi
	DEY	:Do 2 bits first iteration,	88
	BNE NXTBIT	:3 bits in other ten.	D0 EF
:			
	LDX #REML	:Index remainder and go	A2 M4
	JSR SBC4	:subtract subtrahend + 1.	20 lo hi
	LDX #WPRT	:Part-root = part-root * 2	A2 MC
	JSR ROL4	:+ subtraction result.	20 lo hi
	LDA WPRT	:Get subtraction result out	A5 MC
	ROR A	:into Carry and go add	6A
	BCC ADDBCK	:back if negative result.	90 1B
:			
	JSR ASL4	:Multiply WPRT by 4	20 lo hi
	JSR ASL4	:multiply WPRT by 4, then	20 lo hi
	LDX #WSND	:index subtrahend to get	A2 M8
	JSR ADD4	:old WSND + WPRT * 4, leaving	20 lo hi
	JSR LSR4	:X indexing WPRT, so restore	20 lo hi
	JSR LSR4	:to correct part-root.	20 lo hi
	LDX #WSND	:Index subtrahend, then	A2 M8
	JSR ASL4	:old WSND * 2 + WPRT * 8,	20 lo hi
	JSR ADD4	:old WSND * 2 + WPRT * 9,	20 lo hi
	BCC LSTDBL	:then go complete new WSND.	90 11
:			
ADDBCK	SEC	:Add subtrahend + 1 back	38
	LDX #REML	:to remainder, restoring it	A2 M4
	JSR ADC4	:to value held before sub.	20 lo hi
	JSR SUB4	:Old WSND - WPRT.	20 lo hi
	LDX #WSND	:Restore index, then	A2 M8
	JSR ASL4	:Old WSND * 2 - WPRT * 2.	20 lo hi
	JSR SUB4	:Old WSND * 2 - WPRT * 3.	20 lo hi
:			
LSTDBL	LDX #WSND	:Old WSND * 4 - WPRT * 6, or	A2 M8
	JSR ASL4	:old WSND * 4 + WPRT * 18.	20 lo hi
:			
	LDY #3	:Get 3 cube bits next time.	A0 03
	PLA	:Restore count and	68
	SEC	:decrement it.	38
	SBC #1	:repeat for all 11 bits	E9 01
	BNE CUBELP	:of cube root.	D0 A6
:			
PLLWS	LDX #0	:Index from low order bytes	A2 04
	LDA WPRT,X	:Move working part root	B5 MC
	STA ROOTL,X	:to output root.	95 M8
	PLA	:restore 4 page zero	68
	STA WPRT,X	:bytes used for working	95 MC
	INX	:part root.	E8
	CPX #4	:	E0 04
	BNE PLLWS	:	D0 F4

EXTRACTING ROOTS

:			
	PLA	:Restore	68
	TAY	:all	A8
	PLA	:registers	68
	TAX	:and	AA
	PLA	:flags, then exit	68
	PLP	:with integer cube root,	28
	RTS	:remainder and intact cube.	60
:			
:	...Subroutines.		
:			
ASL4	CLC	:= ROL4 with clear carry.	18
ROL4	ROL 0,X	:Rotate 4 bytes	36 00
	ROL 1,X	:from low order	36 01
	ROL 2,X	:to high order,	36 02
	ROL 3,X	:leaving X unchanged and	36 03
	RTS	:Carry = carry out.	60
:			
LSR4	CLC	:= ROR4 with clear carry.	18
ROR4	ROR 3,X	:Rotate 4 bytes	76 03
	ROR 2,X	:from high order	76 02
	ROR 1,X	:to low order	76 01
	ROR 0,X	:leaving X unchanged and	76 00
	RTS	:Carry = carry out.	60
:			
ADD4	CLC	:= ADC4 with no carry in.	18
ADC4	LDY #4	:Count 4 bytes.	A0 04
AD4LP	LDA 0,X	:Add 4 bytes, from low	B5 00
	ADC 4,X	:memory to high memory	75 04
	STA 0,X	:(X) = (X) + (X+4), leave	95 00
	INX	:X = input X + 4, Y = 0,	E8
	DEY	:A = result high order byte,	88
	BNE AD4LP	:Z = 1 and C = carry out.	D0 F6
	RTS	:	60
:			
SUB4	SEC	:= SBC4 with no borrow in.	38
SBC4	LDY #4	:Count 4 bytes.	A0 04
SB4LP	LDA 0,X	:Subtract 4 bytes, from low	B5 00
	SBC 4,X	:memory to high memory	F5 04
	STA 0,X	:(X) = (X) - (X+4), leave	95 00
	INX	:X = input X + 4, Y = 0,	E8
	DEY	:A = result high order byte,	88
	BNE SB4LP	:Z = 1 and C = borrow out.	D0 F6
	RTS	:	60
:	=====		

Converting Decimal to Binary

The usual method of converting from one base into another, say from base-A to base-B, is to initially clear a base-B accumulator and then repeat the following sequence until all base-A digits have been processed.

- 1 Multiply base-B accumulator by base-A using base-B arithmetic.
- 2 Multiply base-A number by base-A getting next digit out as overflow.
- 3 Add base-A overflow digit to base-B accumulator using base-B arithmetic.

When converting from binary to any other base, operation (1) can be done easily by a left shift through the entire binary number. The highest binary digit (bit) overflows into the Carry flag. Operation (3) can be performed at the same time as operation (2) by adding in the Carry as the partial result is added to itself for the multiplication by 2.

Converting to binary from another base is not so easy. The binary partial result has to be multiplied by the base of the source number. Luckily, most conversions are from decimal (base 10) and there is a simple method of multiplying a binary number by 10 using shifts (binary multiplication by 2) and addition.

- 1 Shift number left by one bit (number * 2).
- 2 Save number.
- 3 Shift number left by one bit (number * 4).
- 4 Shift number left by one bit (number * 8).
- 5 Add in saved number (number * 10).

32-BIT SIGNED NUMBERS

SADB4 and SBAD4 are the conversion section of a 32-bit signed number suite that also includes routines to negate, multiply, divide and get the absolute value. The arithmetic routines, including SNEG4 which is called by both SADB4 and SBAD4, appear later in the book.

CONVERTING DECIMAL TO BINARY

The 32-bit suite made its first appearance in *Sub Set* as a set of 5 Z-80 routines (modesty prevents me from crediting its authorship). It was soon followed by a marvellous translation into 6502 code by Dennis May. I must say that Dennis's code used the available registers and page zero pseudo-registers to far better effect than the use made of the Z-80's many registers by the original routines. Then, as though to demonstrate the maxim that there's always a better way if you bother to look for it, Vincent Fojut went to work on the 6502 suite and produced faster, shorter code.

Another maxim states that you can have the fastest code or the shortest but not both. You can have SADB4 at 124 bytes and a maximum of 1,650 clock cycles (SADB4Q) or at 99 bytes but much slower with a maximum execution time of 2,878 clock cycles (SADB4S). There is only one version of SBAD4.

Both forms of SADB4 use the multiplicative method of conversion described above. This is a reasonably quick conversion method but testing for errors, such as overflow of the result accumulator, does tend to add a lot of timing overheads to any routine. If the absolute value of the input number string is validated as less than 2^{31} then both SADB4Q and SADB4S can be written to operate much quicker by leaving out the overflow checks.

```
=====
:= SADB4Q   ASCII decimal to 4-byte signed integer.
=====
:JOB        To convert an integer decimal value held as a
:           string of ASCII digits in memory and preceded by
:           a "+" or "-" sign (or assumed positive if no
:           sign) to a two's complement signed 32-bit number
:           (+/-  $2^{31} - 1$ ) held in registers.
:           Priority: speed.
:ACTION     ON overflow: [ Set overflow flag. Exit. ]
:           Clear 32-bit result accumulator.
:           Read and save first string byte.
:           IF byte = "+" or "-" sign THEN:
:           [ Read next string byte. ]
:           IF byte is ASCII digit THEN:
:           [ Convert digit to binary.
:           REPEAT:
:           [ Result = result * 10 + digit.
:           Read next string byte.
:           IF byte is ASCII digit THEN:
:           [ Convert digit to binary. ] ]
:           UNTIL non-digit terminator. ]
:           IF first byte = "-" THEN: [ Negate result. ]
:-----
:CPU        6502
:HARDWARE    Memory containing ASCII decimal string.
:SOFTWARE    "SNEG4" - Routine to negate a 32-bit two's
:           complement signed integer held in M0 to M3.
:-----
```

CONVERTING DECIMAL TO BINARY

```

:INPUT      MC,D addresses the first byte of the string.
:           The string must be stored high order low memory
:           and terminate with any non-digit character.
:OUTPUT     MC,D is unchanged. X contains 1st string byte.
:           Y indexes string terminator.
:           M4, M5, M6, M7 and A are changed.
:           C = 0: conversion successful.
:           MO,1,2,3 contains signed binary value.
:           C = 1: overflow has occurred.
:ERRORS     Arithmetic error if D = 1 (decimal mode).
:REG USE     P A X Y
:STACK USE   2
:RAM USE     MO to M7, MC and MD
:LENGTH     124
:CYCLES      Minimum: 203
:           Maximum: 1650 (excluding any leading zeros).
:-----
:CLASS 2     -discreet *interruptable *promable
:-*****     *reentrant *relocatable -robust
:=====
:
STR      =    MC           :2-byte stored address of first byte of
                        :ASCII number string.
BIN      =    MO           :4-byte store for binary result.
TMP      =    M4           :4-byte temporary storage.
:
SADB4Q LDY #0             :index STR from zeroth byte.      AO 00
        STY BIN           :Use Y = 0 to initially          84 MO
        STY BIN+1         :clear result                    84 M1
        STY BIN+2         :accumulator.                   84 M2
        STY BIN+3         :                               84 M3
:
        LDA (STR),Y       :Get first string byte and       B1 MC
        TAX               :save for later negate test.     AA
        CMP "-"           :Test if negative sign and       C9 2D
        BEQ SDBQ2         :go get next byte if so.         F0 56
        CMP "+"           :Test for positive sign and      C9 2B
        BNE SDBQ3         :go test digit validity if      D0 55
        BEQ SDBQ2         :not, else get next byte.        F0 50
:
SDBQ1 ASL BIN             :Shift partial                   06 MO
        ROL BIN+1         :result up by one bit:          26 M1
        ROL BIN+2         :BIN = last result * 2.         26 M2
        ROL BIN+3         :                               26 M3
        BMI OVFWQ         :Exit if too big.               30 59
:
        ADC BIN           :Add new digit                   65 MO
        STA TMP           :in to result so far            85 M4
        LDA BIN+1         :while transferring             A5 M1
        ADC #0             :it to temporary               69 00
        STA TMP+1         :storage accumulator.          85 M5
        LDA BIN+2         :TMP = last result * 2         A5 M2
        ADC #0             :+ new digit.                  69 00
        STA TMP+2         :                               85 M6
        LDA BIN+3         :                               A5 M3
        ADC #0             :                               69 00
        STA TMP+3         :                               85 M7
        BMI OVFWQ         :Exit if too big.               30 41

```

CONVERTING DECIMAL TO BINARY

```

:
ASL BIN      :Shift partial                06 M0
ROL BIN+1    :result up by                 26 M1
ROL BIN+2    :one bit.                     26 M2
ROL BIN+3    :BIN = last result * 4.       26 M3
BMI OVFWQ    :Exit if too big.             30 37

:
ASL BIN      :Shift partial                06 M0
ROL BIN+1    :result up by                 26 M1
ROL BIN+2    :one bit.                     26 M2
ROL BIN+3    :BIN = last result * 8.       26 M3
BMI OVFWQ    :Exit if too big.             30 2D

:
LDA BIN      :Add "last result * 2        A5 M0
ADC TMP      :+ new digit" back           65 M4
STA BIN      :into BIN accumulator        85 M0
LDA BIN+1    :giving                      A5 M1
ADC TMP+1    :BIN = last result * 10      65 M5
STA BIN+1    :+ new digit.                85 M1
LDA BIN+2    :                           A5 M2
ADC TMP+2    :                           65 M6
STA BIN+2    :                           85 M2
LDA BIN+3    :                           A5 M3
ADC TMP+3    :                           65 M7
STA BIN+3    :                           85 M3
BVS OVFWQ    :Exit if too big.            70 13

:
SDBQ2 INY    :Index next string byte      C8
LDA (STR),Y  :and get in A.               B1 MC

:
SDBQ3 SEC    :Prepare to subtract.         38
SBC #$30     :Strip ASCII digits hi-nib,  E9 30
CMP #$0A     :test if valid, go add in if C9 0A
BCC SDBQ1    :so, else end of string.     90 A6

:
CPX "-"      :Test for negative start     E0 2D
BNE SDBQ4    :sign, skipping if not,      D0 03
JSR SNEG4    :else negate binary result.  20 lo hi
SDBQ4 CLC    :Set valid result flag, C = 0 18
RTS          :and exit, conversion done.  60

:
OVFWQ SEC    :Set string too big flag,     38
RTS          :C = 1 and exit on overflow.  60
=====

```

```

=====
:= SADB4S    ASCII decimal to 4-byte signed integer.
=====
:JOB        To convert an integer decimal value held as a
:           string of ASCII digits in memory and preceded by
:           a "+" or "-" sign (or assumed positive if no
:           sign) to a two's complement signed 32-bit number
:           (+/- 2**31 - 1) held in registers.
:           Priority: length.

```

CONVERTING DECIMAL TO BINARY

```

:ACTION      ON overflow: [ Set overflow flag. Exit. ]
:            Clear 32-bit result accumulator.
:            Read first string byte and save sign test flag.
:            IF byte = "+" or "-" sign THEN:
:            [ Read next string byte. ]
:            IF byte is ASCII digit THEN:
:            [ Convert digit to binary.
:            REPEAT:
:            [ Result = result * 10 + digit.
:            Read next string byte.
:            IF byte is ASCII digit THEN:
:            [ Convert digit to binary. ] ]
:            UNTIL non-digit terminator. ]
:            Restore sign test flag.
:            IF sign negative THEN: [ Negate result. ]
:-----
:CPU         6502
:HARDWARE    Memory containing ASCII decimal string.
:SOFTWARE    "SNEG4" - Routine to negate a 32-bit two's
:            complement signed integer held in M0 to M3.
:            "DBLBIN" - Local subroutine to multiply result
:            accumulator by 2.
:-----
:INPUT       MC,D addresses the first byte of the string.
:            The string must be stored high order low memory
:            and terminate with any non-digit character.
:OUTPUT      MC,D is unchanged. X contains 1st string byte.
:            Y indexes string terminator.
:            M4, M5, M6, M7 and A are changed.
:            C = 0: conversion successful.
:            M0,1,2,3 contains signed binary value.
:            C = 1: overflow has occurred.
:ERRORS      Arithmetic error if D = 1 (decimal mode).
:REG USE     P A X Y
:STACK USE   2
:RAM USE     M0 to M7, MC and MD
:LENGTH     99
:CYCLES      Minimum: 351
:            Maximum: 2878 (excluding any leading zeros).
:-----
:CLASS 2     -discreet *interruptable *promable
:***--      *reentrant -relocatable -robust
:=====
:
STR   =      MC           :2-byte stored address of first byte of
                        :ASCII number string.
BIN   =      M0           :4-byte store for binary result.
TMP   =      M4           :4-byte temporary storage.
:
SADB4S LDY #0           :index STR from zeroth byte.      A0 00
      LDX #-4           :Count 4 bytes.                  A2 FC
SDBS1 STY BIN+4,X       :Use Y = 0 to initially          94 M4
      INX               :clear result accumulator        E8
      BNE SDBS1         :indexed by X.                   D0 FB
:

```

CONVERTING DECIMAL TO BINARY

	LDA (STR),Y	:Get first string byte and	B1 MC
	CMP "-"	:Test for negative sign,	C9 2D
	PHP	:save test flag (Z), and	08
	BEQ SDBS5	:go get next byte if so.	F0 31
	CMP "+"	:Test for positive sign and	C9 2B
	BNE SDBS6	:go test digit validity if	D0 30
	BEQ SDBS5	:not, else get next byte.	F0 2B
:			
SDBS2	STA TMP	:Store new digit to temporary	85 M4
	STX TMP+1	:accumulator and clear rest	86 M5
	STX TMP+2	:of accumulator bytes by	86 M6
	STX TMP+3	:X (= 0 after loops).	86 M7
:			
	JSR DBLBIN	:BIN = last result * 2.	20 lo hi
:			
	LDX #-4	:Index from low order bytes.	A2 FC
SDBS3	LDA TMP+4,X	:Add "last result * 2"	B5 M8
	ADC BIN+4,X	:to new digit	75 M4
	STA TMP+4,X	:giving,	95 M8
	INX	:TMP = last result * 2	E8
	BNE SDBS3	:+ new digit.	D0 F7
	BVS OVFS2	:Exit if too big.	70 27
:			
	JSR DBLBIN	:BIN = last result * 4.	20 lo hi
	JSR DBLBIN	:BIN = last result * 8.	20 lo hi
:			
	LDX #-4	:Index from low order bytes.	A2 FC
SDBS4	LDA BIN+4,X	:Add "last result * 2 + new	B5 M4
	ADC TMP+4,X	:digit" back into BIN	75 M8
	STA BIN+4,X	:accumulator giving,	95 M4
	INX	:BIN = last result * 10	E8
	BNE SDBS4	:+ new digit.	D0 F7
	BVS OVFS2	:Exit if too big.	70 14
:			
SDBS5	INY	:Index next string byte	C8
	LDA (STR),Y	:and get in A.	B1 MC
:			
SDBS6	SEC	:Prepare to subtract.	38
	SBC #\$30	:Strip ASCII digits hi-nib,	E9 30
	CMP #\$0A	:test if valid, go add in if	C9 0A
	BCC SDBS2	:so, else end of string.	90 CB
:			
	PLP	:Restore "-" test flags and	28
	BNE SDBS7	:skip if not negative number,	D0 03
	JSR SNEG4	:else negate binary result.	20 lo hi
SDBS7	CLC	:Set valid result flag, C = 0	18
	RTS	:and exit, conversion done.	60
:			
OVFS1	PLA	:DBLBIN overflow exit, clear	68
	PLA	:return address from stack.	68
OVFS2	PLP	:Clear stacked "-" test flags.	28
	SEC	:Set string too big flag,	38
	RTS	:C = 1 and exit on overflow.	60
:			

CONVERTING DECIMAL TO BINARY

```

:...Subroutine to multiply result accumulator by 2.
DBLBIN ASL  BIN      :Shift partial          06 M0
      ROL  BIN+1     :result up by one bit:   26 M1
      ROL  BIN+2     :BIN = last BIN * 2.     26 M2
      ROL  BIN+3     :Exit if too big through 26 M3
      BMI  OVFWQ     :routine overflow exit,  30 F1
      RTS           :else return normally.    60
=====

```

SBAD4, converting from binary to ASCII decimal, uses the inverse process of division. This is a far slower method in most cases but may occasionally be the better way to do the job. In general terms, converting from base-A to base-B, the method is to initially clear a base-B accumulator and then repeat the following sequence for the number of digits in the base-B accumulator.

- 1 Shift base-B accumulator right by one base-B digit (i.e. divide base-B accumulator by base-B leaving a digit space at the top).
- 2 Divide base-A number by base-B using base-A arithmetic to give an integer quotient and remainder.
- 3 Store the remainder in the highest digit space at the top of the base-B accumulator.

Operation (1), which generally is quite slow, is dealt with in SBAD4 by pushing each newly found digit onto stack. The digits are pulled off stack and stored to memory in the correct order when the conversion is complete.

Strictly speaking, the range of a 32-bit 2's complement signed number should run from \$80 00 00 00 (-2,147,483,648 decimal, or -2^{31}) to \$7F FF FF FF (+2,147,483,647 decimal, or $2^{31} - 1$) with bit 31 giving the sign. However, 32 bits cannot hold the absolute value of the lowest figure ($0 - (-2^{31}) = -2^{31}$ to 32-bit precision) so it should be treated as either erroneous input or as overflow.

```

=====
:= SBAD4      4-byte signed integer to ASCII decimal.
=====
:JOB          To convert a two's complement signed 32-bit
:              number (+/- 2**31 - 1) held in registers to an
:              integer decimal value held as a string of ASCII
:              digits in memory, preceded by "-" sign if
:              negative, and terminated by an ASCII carriage
:              return code ($0D).
:ACTION       Address result string first byte.
:              IF number is negative THEN:
:              [ Write "-" to string first byte.
:                Address next string byte.
:                Get absolute value of number. ]
:              Initialise stack count to 0.

```

CONVERTING DECIMAL TO BINARY

```

:      REPEAT:
:      [ Digit = remainder (number / 10).
:        Number = integer (number / 10).
:        Push digit.
:        Add 1 to stack count. ]
:      UNTIL number = 0.
:      FOR stack count:
:      [ Pull digit.
:        Digit = digit + ASCII "0".
:        Write digit to string.
:        Address next string byte. ]
:      Write ASCII carriage return string terminator.
:-----
:CPU      6502
:HARDWARE  Memory to contain ASCII decimal string.
:SOFTWARE  "SNEG4" - Routine to negate a 32-bit two's
:           complement signed integer held in M0 to M3.
:-----
:INPUT     M0,1,2,3 contains the signed 32-bit value.
:           (M3 is the high order byte).
:           MC,D addresses the first byte of the destination
:           area for result string.
:OUTPUT    MC,D is unchanged.
:           Y indexes string terminator.
:           M0,1,2,3 = 0. MF, P, A and X are changed.
:ERRORS    Arithmetic error if D = 1 (decimal mode).
:REG USE   P A X Y
:STACK USE  Maximum 10.
:RAM USE   M0 M1 M2 M3 MC MD MF
:LENGTH   71
:CYCLES    Minimum: 1096. Maximum: 11202.
:-----
:CLASS 2   -discreet *interruptable *promable
:--****-   *reentrant *relocatable -robust
:=====
:
STR      =   MC           :2-byte stored address of first byte of
                        :destination for ASCII number string.
BIN      =   M0           :4-byte stored binary value.
PTR      =   MF           :for storing string pointer.
:
SBAD4    LDY #0           :Index result string byte 0.      A0 00
          BIT BIN+3        :Test binary sign bit 31        24 M3
          BPL SBD1         :and skip if positive (= 0).     10 08
:
          LDA "-"          :Else store an ASCII negative    A9 2D
          STA (STR),Y       :sign as string 1st byte        91 MC
          INY               :then index next byte, get       C8
          JSR SNEG4         :absolute value of number.       20 lo hi
:
SBD1     STY PTR           :Save string index to free Y.     84 MF
          LDX #0           :Set digits count to 0.          A2 00
:
SBD2     LDY #32          :Set bit count for division.      A0 20
          LDA #0           :clear division accumulator.     A9 00
:

```

CONVERTING DECIMAL TO BINARY

```

SBD3  ASL  BIN      :Shift number up by one bit      06 M0
      ROL  BIN+1    :moving quotient in to          26 M1
      ROL  BIN+2    :replace dividend, next         26 M2
      ROL  BIN+3    :dividend bit shifted          26 M3
      ROL  A        :into division accumulator.     2A
:
      CMP  #10      :Test if 10 will subtract,      C9 0A
      BCC  SBD4     :skipping if not, else          90 04
      SBC  #10      :subtract 10 (decimal base)     E9 0A
      INC  BIN      :and set quotient bit.          E6 M0
:
SBD4  DEY          :Repeat for all dividend bits,    88
      BNE  SBD3     :remainder is next digit.       D0 EC
:
      PHA          :Push newly found digit and      48
      INX          :count it for later pull.        E8
      LDA  BIN+3    :Test all four bytes            A5 M3
      ORA  BIN+2    :of binary number              05 M2
      ORA  BIN+1    :for zero. If zero then         05 M1
      ORA  BIN      :all digits on stack, else      05 M0
      BNE  SBD2     :go find next digit.           D0 DC
:
      LDY  PTR      :Restore string index to Y.     A4 MF
SBD5  PLA          :Get next digit from stack,      68
      CLC          :prepare to add.                18
      ADC  "0"      :Add ASCII "0" as base value   69 30
      STA  (STR),Y  :and store digit, then          91 MC
      INY          :index next string byte.        C8
      DEX          :Repeat for all digits          CA
      BNE  SBD5     :pushed in division loop.      D0 F6
:
      LDA  #13      :Finally, terminate string     A9 0D
      STA  (STR),Y  :with ASCII carriage return    91 MC
      RTS          :and exit, conversion done.     60
=====

```


15 Converting Gray, 16-bit and Others

Here are a few of the more unusual and challenging ideas that have appeared in the columns of *Sub Set*. Gray Code is a lazy way of incrementing to the next number (you only have to change one bit!) and the other base conversions should come in useful when we evolve to a life form with 36 fingers. The 16-bit conversion is quite normal except for a handy trick with the overflow flag.

16-BIT TO ASCII DECIMAL

PRAY by Andrew Watson converts the 16-bit unsigned number held in registers A and Y to a string of ASCII decimal digits. As in SBAD4, the 32-bit routine in the last chapter, PRAY stores the intermediate results on stack. The final result is not written to memory but sent to an output routine. Andrew has labelled the output subroutine CHROUT but this is merely a terminological convenience; any routine which outputs the ASCII character in A can be used by PRAY.

PRAY's conversion method is fundamentally the same as that used in SBAD4 but the implementation is interesting. Andrew has used 2 bytes of stack, indexed by X, as workspace rather than the page zero pseudo-registers so use of PRAY is independent of the system software's base page arrangements. The cost of such freedom is an extra 2 clock cycles for each rotation of each byte – possibly 3400 cycles in total.

Because A carries the last digit out of the conversion section of PRAY into the output section, it cannot be used to test if the number has reduced to zero by the normal method of LDA BYTE1: ORA BYTE2: BNE NXTDG. The method used by Andrew relies on the fact that 'SBC #10' is the only instruction in the ROLL loop to affect the overflow flag. As the subtraction is never performed unless A is 10 or more, overflow cannot occur and V is always cleared. So, by setting V prior to the loop, BVC NXTDG will branch only when a subtraction has occurred in the loop, a quotient bit has been set and the number is not zero.

CONVERTING GRAY, 16-BIT AND OTHERS

```

=====
:= PRAY      Print 16-bit value as ASCII decimal.
=====
:JOB         To convert a 16-bit unsigned number held in
:            registers to ASCII decimal, with leading zeros
:            suppressed, and send the result to an output
:            routine.
:ACTION      Initialise digit as end-marker.
:            REPEAT:
:            [ Push digit.
:              Digit = remainder (number / 10).
:              Number = integer (number / 10). ]
:            UNTIL number = 0.
:            REPEAT:
:            [ Digit = digit + ASCII "0".
:              Call output routine.
:              Pull digit. ]
:            UNTIL digit = end-marker.
-----
:CPU         6502
:HARDWARE    None.
:SOFTWARE    "CHROUT" - a subroutine to output the ASCII
:            character in A (to printer, screen memory,
:            remote terminal, etc.) without changing other
:            register values.
-----
:INPUT       A (high byte) and Y contain number to be output.
:OUTPUT      A = 0. Y = -1 (unless changed by CHROUT).
:ERRORS      Arithmetic error if D = 1 (decimal mode).
:REG USE     A Y
:STACK USE   (Minimum 8, maximum 11) + CHROUT stack use.
:RAM USE     None.
:LENGTH     49
:CYCLES      Approximately 44 + digits * (488 + CHROUT time).
-----
:CLASS 2     -discreet *interruptable *promable
:-****-      *reentrant *relocatable -robust
=====
:
PRAY  PHP           :Save flags.                08
      PHA           :Push A and Y to store number  48
      TYA           :on stack, high order         98
      PHA           :byte in higher memory.        48
      TXA           :Save X, for use               8A
      PHA           :indexing stack 2 bytes        48
      TSX           :below stacked number.         BA
      LDA  #$80      :Push end-marker 1st time.    A9 80
:
NXTDG PHA           :Push last digit (or marker).  48
      LDY  #16       :Count for 16 bits.           A0 10
      SEC           :Subtract to set overflow flag, 38
      LDA  #0        :cleared only on non-zero     A9 00
      SBC  #$80      :quotient in division.       E9 80
:
ROLL  ROL  A         :(1st time clears A). Rotate  2A
      CMP  #10       :next number bit into remainder C9 0A
      BCC  NBIT      :in A, if >= 10 then subtract, 90 02
      SBC  #10       :set result bit C, clear V.   E9 0A

```

CONVERTING GRAY, 16-BIT AND OTHERS

```

:
NBIT  ROL  $0102,X :Shift result bit into number 3E 02 01
      ROL  $0103,X :and next number bit to Carry. 3E 03 01
      DEY      :Repeat 16 times for 16 bits      88
      BPL  ROLL  :after 1st time, 17 in all.      10 F0
:
      BVC  NXTDG  :Another digit if number > 0.    50 E6
:
PRNTDG ORA  #$30   :Convert digit to ASCII         09 30
      JSR  CHROUT  :decimal and go output it.      20 lo hi
      PLA      :Pull next byte and repeat if      68
      BPL  PRNTDG  :digit, else end-marker $80.    10 F8
:
      PLA      :Restore X.                        68
      TAX      :                                AA
      PLA      :Remove number from stack,         68
      PLA      :leaving A = 0.                   68
      PLP      :Restore flags and exit,           28
      RTS      :number output.                   60
:=====

```

OTHER BASES

XBIN, by Dennis May, and BINX, an amalgamation of routines by Dennis May and Vincent Fojut, convert between 32-bit binary and numbers in other bases, from 2 to 36, stored as ASCII digits. Hexadecimal uses the upper-case letters A to F to stand for the digits following on from 9 and the idea is extended to use all the upper-case letters.

You can use the method to convert between binary and numbers up to base 62 by bringing the lower-case letters into play. However, XBIN and BINX check only for the discontinuity between ASCII '9' and ASCII 'A' which is used to codify the symbols, :, ;, <, =, >, ? and @. There is another break between Z and a for the six symbols, [, \,], ↑, _ and '. To jump it you will need to insert code for the appropriate test and corresponding addition or subtraction.

```

:=====
:= XBIN      Unsigned ASCII base 2-36 to 32-bit conversion.
:=====
:JOB        To convert an unsigned number, of any base 2 to
:           36, held in memory as ASCII digits and upper
:           case letters to a 32-bit binary number held in
:           registers or base-page "pseudo-registers".
:ACTION     Clear result.
:           Get first character.
:           ON overflow: [ Set overflow flag & exit. ]
:           WHILE character NOT terminator:
:           [ Convert character to binary coded digit.
:             Result = result * base + digit.
:             Index and get next character. ]
:           Set conversion completed flag.
:-----

```

CONVERTING GRAY, 16-BIT AND OTHERS

```

:CPU      6502
:HARDWARE Memory containing ASCII number string.
:SOFTWARE None.
-----
:INPUT    M4,5 addresses 1st byte of ASCII string which
:         must terminate with $0D (carriage return).
:         M6 contains the number base.
:OUTPUT   Registers changed. M7 to MC changed.
:         M4 to M6 not changed.
:         C = 0: conversion completed.
:         32-bit result in M0 to M3 (M3 is high order).
:         C = 1: overflow during process.
:         M0 to M3 is indeterminate.
:ERRORS   No test is made for non-upper-case alphanumeric
:         characters in ASCII string.
:         No test is made for digits greater than base.
:         Arithmetic error if D = 1 (decimal mode).
:REG USE  P A X Y
:STACK USE 2
:RAM USE   M0 to MC
:LENGTH   106
:CYCLES    Not given.
-----
:CLASS 2   -discreet *interruptable *promable
:--***--   *reentrant -relocatable -robust
=====
ASCN  =  M4      :Stored address of ASCII string.
BASE  =  M6      :Stored ASCII number base (2 to 36).
BTMP  =  MB      :Storage for working BASE.
RSLT  =  M0      :4-byte result location (low byte).
RTMP  =  M7      :Storage for working RSLT (low byte).
INDX  =  MC      :Storage for ASCII string pointer.
:
XBIN  LDY #0      :Clear for RSLT clear.           A0 00
      LDX #4      :Index for RSLT 4 bytes.         A2 04
:
XBIN1 DEX         :Index RSLT next byte             CA
      STY RSLT,X  :and clear it, repeat             94 M0
      BNE XBIN1   :until RSLT clear. X = 0.          D0 FB
      STX INDX    :Initialise ASCII index to 0.      86 MC
:
XBIN2 LDY INDX    :Index current ASCII byte          A4 MC
      LDA (ASCN),Y :and pick it up.                 B1 M4
      CMP #$0D     :If ASCII "carriage return"       C9 0D
      BEQ END      :terminator then completed.       F0 52
:
      SEC         :Strip ASCII digits high          38
      SBC #$30     :nibble and test for if           E9 30
      CMP #$0A     :greater than digit 9,            C9 0A
      BCC ASCY     :adjusting for gap between        90 02
      SBC #7       : "9" and "A" if it is.           E9 07
ASCY  PHA         :Save new digit.                   48
      LDA #0       :Clear for RTMP clear.            A9 00
      LDX #4       :Index for RTMP 4 bytes.          A2 04
:
XBIN3 DEX         :Index RTMP next byte             CA
      STA RTMP,X   :and clear it, repeat             95 M7
      BNE XBIN3    :until RTMP clear.                D0 FB

```

CONVERTING GRAY, 16-BIT AND OTHERS

```

LDA BASE      :Move base to temp byte for      A5 M6
STA BTMP      :use as multiplier.              85 MB
LDY #8        :Count for 8-bit multiplier.     AO 08
:
XBIN4 ASL BTMP :Shift next multiplier bit      06 MB
      PHP      :into C and save it.            08
      ASL RTMP  :Shift left partial product   06 M7
      ROL RTMP+1 :for possible addition at    26 M8
      ROL RTMP+2 :next bit place.             26 M9
      ROL RTMP+3 :                           26 MA
      BCS OVFW1  :Skip out if product too big. B0 2E
      PLP      :Get multiplier bit to C and    28
      BCC XBIN6  :skip if 0, no add this place. 90 0E
      LDX #-4    :Else index from low bytes.  A2 FC
      CLC      :Clear for low bytes add.      18
:
XBIN5 LDA RSLT+4,X :Add multiplicand byte to    B5 M4
      ADC RTMP+4,X :partial product.          75 MB
      STA RTMP+4,X :                           95 MB
      INX        :Index next and repeat for    E8
      BNE XBIN5  :all four bytes.             D0 F7
      BCS OVFW2  :Out if product too big.      B0 1E
:
XBIN6 DEY        :Repeat for all 8 bits of      88
      BNE XBIN4  :multiplier (base).          D0 DF
:
      CLC      :Clear for add.                18
      PLA      :Get new digit and add to      68
      ADC RTMP  :product low byte, result to  65 M7
      STA RSLT  :partial conversion result.   85 M0
      LDX #-3   :Index from byte 2.          A2 FD
:
XBIN7 LDA RTMP+4,X :Move other three product   B5 MB
      ADC #0      :bytes to conversion result  69 00
      STA RSLT+4,X :adding in any carry from   95 M4
      INX        :digit add in to low byte and E8
      BNE XBIN7  :subsequent carries.         D0 F7
      BCS OVFW3  :Out if result too big.      B0 09
:
      INC INDX   :Index next ASCII byte and   E6 MC
      JMP XBIN2  :continue conversion.        4C lo hi
:
END   CLC      :Flag conversion complete      18
      RTS      :(C = 0) and exit.            60
:
OVFW1 PLA      :Lose multiplier bit.          68
OVFW2 PLA      :Lose digit.                  68
OVFW3 RTS      :Exit (C = 1) on overflow.     60
=====

```

```

=====
:= BINX      32-bit to unsigned ASCII base 2-36 conversion.
=====
:JOB        To convert a 32-bit binary number held in
:            registers or base-page "pseudo-registers" to an
:            unsigned number, of any base 2 to 36, held in
:            memory as ASCII digits and upper-case letters.
:

```

CONVERTING GRAY, 16-BIT AND OTHERS

```

:ACTION      Push terminator byte.
:            REPEAT:
:            [ Digit = remainder (number / base).
:            Number = integer (number / base).
:            IF digit < 10
:            THEN: [ Digit = digit + ASCII "0". ]
:            ELSE: [ Digit = digit + ASCII "A" - 10. ]
:            Push digit. ]
:            UNTIL number = 0.
:            Address destination.
:            REPEAT:
:            [ Pull byte and store to destination.
:            Address next destination. ]
:            UNTIL terminator byte stored.
:-----
:CPU         6502
:HARDWARE    Destination RAM for conversion result.
:SOFTWARE    None.
:-----
:INPUT       M0,1,2,3 contains unsigned 32-bit number.
:            (High order byte in M3).
:            M4,5 contains address of destination for result.
:            MD contains conversion base (2 to 36).
:OUTPUT      Conversion result at address in M4,5 (high order
:            byte first) terminated by ASCII carriage return.
:            M4 M5 & MD are unchanged.
:            M0 = M1 = M2 = M3 = 0. A = $0D. Carry = 0.
:            Y indexes terminator (Y= result string length).
:ERRORS      Arithmetic error if D = 1 (decimal mode).
:REG USE     P A Y
:STACK USE   Minimum: 2. Maximum: 33.
:RAM USE     M0 M1 M2 M3 M4 M5 MD
:LENGTH     55
:CYCLES      Minimum: 1127. Maximum: 38538.
:-----
:CLASS 2     -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
:=====
:
BNO  =  M0      :4-byte stored binary number.
ANOA  =  M4      :2-byte stored address for first byte
                :of ASCII base 2-36 result.
BAS  =  MD      :Stored base, $02 to $24 (2 to 36).
ABAS  =  $30     :ASCII "0" - lowest converted value.
:
BINX   LDA  #13   :Push ASCII carriage return code   A9 0D
        PHA                :on stack as stack base.      48
:
REMLP  LDY  #32   :Count for 32 bits in DIVLP.        A0 20
        LDA  #0     :Clear division accumulator.      A9 00
:
DIVLP  ASL  BNO    :Shift dividend/quotient           06 M0
        ROL  BNO+1  :up by one bit. Quotient shifts    26 M1
        ROL  BNO+2  :in to replace dividend.          26 M2
        ROL  BNO+3  :Next dividend bit is rotated     26 M3
        ROL  A      :into remainder-accumulator      2A
:

```

CONVERTING GRAY, 16-BIT AND OTHERS

```

      CMP BAS      :Test if base can be subtracted      C5 M0
      BCC DIVLPT   :skipping if not, else                90 04
      SBC BAS      :subtract base from remainder        E5 M0
      INC BNO      :and set quotient result bit.        E6 M0
:
DIVLPT DEY         :Repeat for all dividend bits.        88
      BNE DIVLP    :Remainder = next digit. Y = 0.      D0 EC
:
      CMP #10      :If remainder-digit is in range      C9 0A
      BCC ASCBAS   :0 to 9 then okay so skip, else      90 02
      ADC #6        :add 7 (C = 1) for range A to Z.    69 06
ASCBAS ADC #ABAS   :ASCII starts at $30 = "0".          69 30
      PHA          :Save digit in order on stack.       48
:
      LDA BNO      :Test quotient from last             A5 M0
      ORA BNO+1    :division (= dividend for next      05 M1
      ORA BNO+2    :division) to see if reduced        05 M2
      ORA BNO+3    :to zero (= conversion done)        05 M3
      BNE REPLP    :and repeat until so.               D0 D5
:
STORE PLA         :Move one stacked digit to           68
      STA (ANO),Y  :destination. (Y = 0 at start.)     91 M4
      INY          :Index next destination place.      C8
      CMP #ABAS   :If last byte >= lowest digit      C9 30
      BCS STORE   :then repeat, else carriage        B0 F8
      RTS         :return so end, string stored.      60
:=====

```

You can use the two routines in sequence to convert from one obscure, ASCII encoded base to another. With the address of the ASCII number string in M4 and M5, the current number base in M6 and the required number base in MD, a call to BCNV will write the result string to the same location as the current string.

```

BCNV JSR XBIN      :Convert to 32-bit binary            20 lo hi
      BCS OVFWX    :but exit, C = 1, on overflow.      B0 03
      JMP BINX     :Go convert to new base, BINX      4C lo hi
                        :exits with C = 0 = no overflow.
OVFWX RTS         :Exit on XBIN overflow.              60

```

BINARY TO GRAY CODE

The process of incrementing or decrementing a binary number may involve just a single-bit change, such as in the step from 6 to 7 (%0110 to %0111). Usually, however, more than one bit changes state and in 4-bit wraparound arithmetic as many as all 4 bits can change when %1111 (decimal 15) increments to %0000 (decimal 0, or 16).

In some applications the individual bits may not be packed together in a single byte. Accessing all of them could be a process that simply takes too long. In other cases an increment might have to be carried out by a masking operation – difficult if some bits have to be cleared

CONVERTING GRAY, 16-BIT AND OTHERS

and others set. Reading rapidly changing binary numbers one bit at a time could produce an out of sequence value if several bits have changed state during the access.

Binary code is obviously not ideal for any of those situations but there is a type of code in which consecutive numbers differ by only one bit. Gray Code is used where speed and ease of change is important and where misreading the value can prove disastrous. It is often used in programming revolving mechanisms – which can be anything from the stepper motor on a plotter to a battleship gun turret.

Several sequences meet the criterion of a single-bit change between consecutive numbers. The following table gives just one of the possible 4-bit sequences.

Decimal	Hex	Binary	Gray Code
0	0	0000	0000
1	1	0001	0001
2	2	0010	0011
3	3	0011	0010
4	4	0100	0110
5	5	0101	0111
6	6	0110	0101
7	7	0111	0100
8	8	1000	1100
9	9	1001	1101
10	A	1010	1111
11	B	1011	1110
12	C	1100	1010
13	D	1101	1011
14	E	1110	1001
15	F	1111	1000

GYCON by Vincent Fojut converts a 4-bit binary value to 4-bit Gray Code by means of a look-up table. Vincent supplied it to *Sub Set* at the same time as his GETLOC (see chapter 5) which is called by GYCON to get the address of a known point in the program into M0,1. The Gray Code value is then picked up by program relative addressing (masquerading as post indexed indirect addressing).

```
:=====
:= GYCON      Binary to Gray Code conversion.
:=====
:JOB          To convert a 4-bit binary value to 4-bit Gray
:             Code using a look-up table.
:ACTION       Set pointer to start of look-up table.
:             Add binary value to pointer.
:             Read code addressed by pointer.
:-----
```

CONVERTING GRAY, 16-BIT AND OTHERS

```

:CPU      6502
:HARDWARE  None.
:SOFTWARE  "GETLOC" - A subroutine to return in MO,1 the
:           address of the instruction immediately following
:           "JSR GETLOC".
:-----
:INPUT     Y = 4-bit binary value.
:OUTPUT    A = 4-bit Gray Code.
:           MO M1 Y and P are changed.
:ERRORS    The table will be exceeded if Y > 15.
:           Arithmetic error if D = 1 (decimal mode).
:REG USE   P A Y
:STACK USE 2
:RAM USE   MO M1
:LENGTH   26
:CYCLES    48 (including GETLOC cycles).
:-----
:CLASS 2   -discreet *interruptable *promable
:-----
:*****    *reentrant *relocatable -robust
:=====
:
GYCON JSR GETLOC :Get address of 4th byte of 20 lo hi
      TYA       :GYCON in MO,1. Add binary 98
      ADC #7     :value to routine bytes - 3 69 07
      TAY       :as index to table entry. A8
      LDA (MO),Y :Get Gray Code equivalent B1 MO
      RTS       :and exit. 60
:
      .BYTE $00,$01 :Look-up table 00 01
      .BYTE $03,$02 :of 4-bit 03 02
      .BYTE $06,$07 :Gray Code 06 07
      .BYTE $05,$04 :equivalents 05 04
      .BYTE $0C,$0D :of binary 0C 0D
      .BYTE $0F,$0E :values 0F 0E
      .BYTE $0A,$0B :0 to 15 0A 0B
      .BYTE $09,$08 :in ascending order. 09 08
:=====

```

COMPUTED GRAY CODE CONVERSION

Using a look-up table to convert from binary to Gray Code is quick but unnecessarily lengthy. A computed conversion can be quicker for small values and is definitely shorter for 8-bit values where a 256-byte look-up table would be needed. In fact the conversion need not be computed at all but can be done by specialised hardware circuits. BGCB is a *Sub Set* routine of mine which emulates the hardware logic.

Each Gray Code bit is the complement of the corresponding binary value bit if the next higher binary bit is set. So the conversion method only needs to shift the partial result one-bit lower and exclusive-OR it with the original value. Try it out with several values from the table above.

CONVERTING GRAY, 16-BIT AND OTHERS

Conversion of a 4-bit or even an 8-bit binary value to Gray Code by a computed method is hardly worth the bother of writing as a subroutine. It is only a 5-byte, 8 clock cycle sequence, easily written straight into any program:

```

STA MO      :Save 4-bit or 8-bit binary.          85 MO
LSR A       :Shift binary down 1 bit and          4A
EOR MO      :complement if next higher bit set.  45 MO

```

What does make it worth writing as a subroutine is the fact that if the process is repeated for a count one less than the value bit length, it will convert back from Gray Code to binary. BGCB will convert both 4-bit and 8-bit codes from binary to Gray Code on an input count of 1 and from Gray Code to binary on counts of 3 or 7. I don't know what it produces on other counts.

```

=====
:= BGCB      Convert between Gray Code and binary.
=====
:JOB         To convert from a binary value to Gray Code or
:            from Gray Code to binary, using either 4-bit or
:            8-bit values in either case.
:ACTION      FOR input count:
:            [ Result = (result / 2) EOR (input value). ]
:-----
:CPU         6502
:HARDWARE    None.
:SOFTWARE    None.
:-----
:INPUT       Y = 1. 4-bit or 8-bit binary value in A.
:            Y = 3. 4-bit Gray Code in A.
:            Y = 7. 8-bit Gray Code in A.
:OUTPUT      Converted value in A.
:            P, X and Y are changed.
:ERRORS      No check made for a valid count input.
:REG USE     P A X Y
:STACK USE   1
:RAM USE     None.
:LENGTH     11
:CYCLES      Binary to Gray Code: 23.
:            Gray Code to 4-bit Binary: 45.
:            Gray Code to 8-bit Binary: 89.
:-----
:CLASS 2     -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
=====
:
BGCB TSX          :Index stack workspace where      BA
      PHA          :input value is stored.          48
BGCB LSR A        :Exclusive-or half previous       4A
      EOR $0100,X  :result with stacked input       5D 00 01
      DEY          :code 1, 3 or 7 times,           88
      BNE BGCB LSR :depending on input count.       D0 F9
      TXS          :Tidy stack. Exit with Gray      9A
      RTS          :from binary or vice versa.      60
=====

```

Arithmetic:

16-bit

16

The 6502 is the poor relation of the 8-bit processor family. It has no 16-bit registers on which to perform higher precision arithmetic. The routines in this chapter show how to use the few 8-bit registers that are available to it for binary multiplication and division and decimal halving on 2-byte values.

A BCD EXERCISE

Binary Coded Decimal digits are normally stored 2 to a byte. They utilise only the first 10 of the 16 values that can be coded in 4 bits – that is, %0000 to %1010, leaving %1011 to %1111 unused. The 6502 will perform decimal addition or subtraction on BCD values if the decimal mode flag D is set.

Dividing BCD numbers by 2 isn't achieved easily on the 6502. Hexadecimal halving is just a matter of right shifting by one bit with any carry between digits having the correct value of 8 (half of 16). Right shifting a BCD value still gives a value of 8 to any inter-digit carry. This has to be tested for and adjusted to the correct decimal value of 5 by subtracting 3.

When *Sub Set* issued a challenge for the shortest Z-80 code to halve a 4-digit BCD value in the 16-bit HL register-pair there was quite a large response ranging in length from an anonymous contribution at 81 bytes (!) to the shortest, my HLFHL, at 21 bytes. Before settling for HLFHL, I had spent a not insignificant number of hours investigating all the possible methods of dividing a 4-digit BCD number by 2 (and some impossible ones) so I must admit to being rather annoyed with myself for not spotting an improvement that reduced the length of HLFHL to 20 bytes. The person responsible for my ego deflation was Gavin Every. Gavin is also responsible for this neat 6502 version.

HLFXY, as its name suggests, halves the BCD value held in the index registers X and Y. With the need to transfer the bytes to and from the accumulator for the shifts and tests, it is very compact at 27 bytes. The only imperfection in the routine is that it doesn't test for valid BCD input – presumably because the calling routine is

ARITHMETIC: 16-BIT

expected to be working only on BCD values – and uses binary arithmetic without first clearing the decimal mode flag, which surely must be set on entry if BCD work is being done.

```

=====
:= HLFXY      Halve 4-digit BCD string in registers.
=====
:JOB          To divide a register held string of 4 BCD digits
:              by two, setting carry for remainder.
:ACTION       Halve by right shifting one bit.
:              Adjust any carry from value 8 to value 5.
=====
:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
=====
:INPUT        X and Y together contain a 4-digit BCD number.
:              (High order digit in high-nibble X.)
:OUTPUT       X and Y together contain the BCD result of
:              input XY divided by 2. Carry contains remainder.
:ERRORS       Arithmetic error if D = 1 (decimal mode)!!
:              No check for valid BCD input.
:REG USE      P X Y
:STACK USE    1
:RAM USE      None.
:LENGTH      27
:CYCLES       Minimum: 38. Maximum: 49.
=====
:CLASS 2      *discreet *interruptable *promable
:*****~     *reentrant *relocatable -robust
=====
:
HLFX Y PHA          :Save A.                                48
      TXA           :Move high pair of digits to            8A
      LSR A         :A and divide by 2, then move           4A
      TAX           :halved pair back to X.                 AA
      AND #$08      :Without affecting Carry flag,          29 08
      BEQ LOGET     :skip if no half carry, else            FO 03
      DEX           :subtract 3 to convert carry            CA
      DEX           :from thousands worth 8 hundreds       CA
      DEX           :into carry worth 5 hundreds.          CA
:
LOGET TYA           :Get low pair in A and skip if no       98
      BCC LOHALF    :Carry from hundreds, else add 10      90 02
      ADC #$9F      :to tens (worth 5 when halved).         69 9F
:
LOHALF ROR A        :Divide by 2, getting tens right       6A
      TAY           :and move pair back to Y.               A8
      AND #$08      :Without affecting Carry flag,          29 08
      BEQ EXIT      :skip if no half carry, else            FO 03
      DEY           :subtract 3 to convert carry            88
      DEY           :from tens worth 8 units                88
      DEY           :into carry worth 5 units.              88
:
EXIT PLA            :Restore A and exit with BCD             68
      RTS           :value halved and remainder in C.        60
=====

```

BINARY DIVISION

DUBDIV by Tim Groves was the very first 6502 datasheet to use the pseudo-register set, M0 to MF, and did so in the issue that marked the start of *Sub Set's* second year.

The normal method of long division is used in DUBDIV and superbly demonstrates the superiority of working in binary values. In decimal division the divisor may have to be subtracted from the dividend up to nine times at each digit place. In binary there is a maximum of one subtraction at each bit place.

```

=====
:= DUBDIV    16-bit unsigned division.
=====
:JOB         To divide one 16-bit unsigned integer by another
:            and return the 16-bit unsigned integer quotient
:            and 16-bit remainder.
:ACTION      Clear remainder and quotient accumulators.
:            FOR count of dividend bits:
:            [ Dividend = dividend * 2.
:            Carry = dividend overflow bit.
:            Remainder = remainder * 2 + Carry.
:            IF remainder >= divisor:
:            THEN: [ Remainder = remainder - divisor.
:                   Carry = 1. ]
:            ELSE: [ Carry = 0. ]
:            Quotient = quotient * 2 + Carry. ]
-----
:CPU         6502
:HARDWARE    None.
:SOFTWARE    None.
-----
:INPUT       M2,3 contains the dividend.
:            M4,5 contains the divisor.
:OUTPUT      M0,1 contains the integer quotient.
:            M2,3 and A,Y contains the remainder.
:            P is changed.
:            If M4,5 = 0 then the quotient in M0,1 = $FFFF
:            and the remainder = input dividend.
:ERRORS      No check for division by zero.
:REG USE     P A Y
:STACK USE   2
:RAM USE     M0 M1 M2 M3 M4 M5
:LENGTH     58
:CYCLES      Minimum: 887. Maximum: 1191.
-----
:CLASS 2     -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
=====
QUOT  =      M0      :2-bytes for quotient output.
REM   =      M2      :2-bytes for remainder output.
DVND  =      M2      :2-byte stored dividend input.
DVSR  =      M4      :2-byte stored divisor input.
:

```

ARITHMETIC: 16-BIT

DUBDIV	TXA	:Save X for use as bit count.	8A
	PHA	:	48
	CLD	:Ensure binary subtractions.	D8
	LDA #0	:Clear remainder-accumulator	A9 00
	TAY	:hi-byte and lo-byte.	A8
	STA QUOT+1	:Clear quotient hi-byte	85 M1
	STA QUOT	:and lo-byte.	85 M0
	LDX #16	:Set count for 16 dividend bits.	A2 10
:			
NXTBIT	ASL DVND	:Shift dividend up one bit,	06 M2
	ROL DVND+1	:moving next bit out to Carry.	26 M3
	PHA	:Save accumulator hi-byte	48
	TYA	:and enable accumulator lo-byte.	98
	ROL A	:Shift remainder up, getting new	2A
	TAY	:dividend bit. Save lo-byte	A8
	PLA	:and re-enable hi-byte.	68
	ROL A	:Complete accumulator shift up.	2A
:			
	CMP DVSR+1	:Will divisor subtract?	C5 M5
	BEQ CHEKLO	:Test lo-bytes if hi-bytes equal.	F0 0C
	BCC QBIT	:No divide in at this bit place.	90 0E
:			
SUBDIV	PHA	:Result will be positive. Save	48
	TYA	:accumulator hi-byte, enable	98
	SBC DVSR	:lo-byte and subtract divisor	E5 M4
	TAY	:lo-byte (C = 1 from CMP or CPY).	A8
	PLA	:Re-enable hi-byte and subtract	68
	SBC DVSR+1	:divisor. Non-negative result	E5 M5
	BCS QBIT	:so C = 1 always.	B0 04
:			
CHEKLO	CPY DVSR	:Will subtraction give negative	C4 M4
	BCS SUBDIV	:result? Subtract only if not.	B0 F2
:			
QBIT	ROL QUOT	:Rotate C in, writing quotient	26 M0
	ROL QUOT+1	:result bit at correct bit place.	26 M1
	DEX	:Repeat for all 16 bits	CA
	BNE NXTBIT	:of dividend.	D0 DB
:			
	STA REM+1	:Store remainder to page	85 M3
	STY REM	:zero output variable.	84 M2
	PLA	:Restore X from stack	68
	TAX	:via A.	AA
	LDA REM+1	:Recover remainder to AY.	A5 M3
	RTS	:Exit, division completed.	60
:=====			

BINARY MULTIPLICATION

No corresponding multiplication routine accompanied DUBDIV in *Sub Set* but since the process is no less essential than division for 16-bit work, I include one here. DUBMUL is compatible with DUBDIV, using the same 6-byte block of page zero pseudo-registers and returning part of the result in registers A and Y.

ARITHMETIC: 16-BIT

```

=====
:= DUBMUL    16-bit unsigned multiplication.
=====
:JOB        To multiply two 16-bit unsigned integers and
:           return the 32-bit unsigned integer product.
:ACTION     Clear product accumulators.
:           FOR count of multiplier bits:
:           [ Product = product * 2.
:           Multiplier = multiplier * 2.
:           Carry = multiplier overflow bit.
:           IF Carry = 1 THEN:
:           [ Product = product + multiplicand. ] ]
-----
:CPU        6502
:HARDWARE   None.
:SOFTWARE   None.
-----
:INPUT      M2,3 contains the multiplier.
:           M4,5 contains the multiplicand.
:OUTPUT     M0,1,2,3 contains the product.
:           A,Y contains the product highest two bytes.
:           P is changed.
:ERRORS     None.
:REG USE    P A Y
:STACK USE  2
:RAM USE    M0 M1 M2 M3 M4 M5
:LENGTH     62
:CYCLES     Minimum: 727. Maximum: 1271.
-----
:CLASS 2    -discreet *interruptable *promable
:-*****    *reentrant *relocatable *robust
=====
PROD  =  M0      :4-bytes for product output.
MPLR  =  M2      :2-byte stored multiplier input.
MCND  =  M4      :2-byte stored multiplicand input.
:
DUBMUL TXA      :Save X for use as bit count.      8A
      PHA      :                               48
      CLD      :Ensure binary additions.          D8
      LDA #0    :Clear product-accumulator        A9 00
      TAY      :highest two bytes A and Y, and     A8
      STA PROD+1 :lowest two bytes, M1             85 M1
      STA PROD  :and M0.                          85 M0
      LDX #16   :Set count for 16 multiplier bits. A2 10
:
NEXTBT ASL PROD  :Shift product up one bit,        06 M0
      ROL PROD+1 :moving next bit out to Carry.    26 M1
      PHA      :Save product hi-byte and         48
      TYA      :enable product next highest byte. 98
      ROL A     :Shift product byte up one bit    2A
      TAY      :Save next highest byte and       A8
      PLA      :re-enable hi-byte. Full product  68
      ROL A     :shift up to next bit place.     2A
:
      ASL MPLR   :Shift multiplier up one bit,    06 M2
      ROL MPLR+1 :moving next bit out to Carry.  26 M3
      BCC ENDTST :Skip if no add this bit place. 90 16
:

```

ARITHMETIC: 16-BIT

```

      CLC          :Prepare to add, no carry in.      18
      PHA          :Save product hi-byte.            48
      LDA PROD     :Add multiplicand lowest          A5 M0
      ADC MCND     :byte to product lowest           65 M4
      STA PROD     :byte.                            85 M0
      LDA PROD+1   :Add multiplicand next lowest     A5 M1
      ADC MCND+1   :byte to product next lowest      65 M5
      STA PROD+1   :byte.                            85 M1
      PLA          :Re-enable product hi-byte.       68
      BCC ENDTST   :Skip if no carry to hi-bytes.    90 05
:
      INY          :Add in carry by increment which  C8
                  :doesn't affect Carry flag.
      BNE ENDTST   :Skip if no carry to hi-byte,     D0 02
      ADC #0       :else use C = 1 from lo-bytes.    69 00
:
      ENDTST DEX   :Repeat for all 16 bits           CA
      BNE NEXTBT   :of multiplier.                  D0 D7
:
      STA PROD+3   :Store product highest two bytes  85 M3
      STY PROD+2   :to page zero output variable.   84 M2
      PLA          :Restore X from stack             68
      TAX          :via A.                           AA
      LDA PROD+3   :Recover product hi-byte to A.    A5 M3
      RTS          :Exit, multiplication completed.  60
:=====

```

Arithmetic:

32-bit

17

A 32-bit (4-byte) number can have any value between 0 and $2^{32} - 1$ (4,294,967,295 or \$FF FF FF FF), if unsigned or absolute arithmetic is used. In 2's complement notation, the usual method of dealing with sign in integer arithmetic, 32 bits have a range of -2^{31} (-2,147,483,648 or \$80 00 00 00) to $2^{31} - 1$ (2,147,483,647 or \$7F FF FF FF). Bit 31, the highest bit, shows the sign of the number: 0 is positive and 1 is negative. A 2's complement number can be sign-changed (i.e. switched from positive to negative, or vice versa) by subtracting it from zero (negation). An alternative method is to change the state of each individual bit (complement) and then add 1.

A SIGNED ARITHMETIC SUITE

The 5 routines included here form a complete suite for performing simple integer arithmetic on 32-bit signed numbers. Along with the 32-bit conversion routines in chapter 14, which convert 32-bit values to and from signed ASCII decimal, and routines for keyboard input and screen display (both of which should be in your system software), you have the basis of a calculator working to 9 decimal digits precision.

The initial 6502 work on this suite was by Dennis May who provided *Sub Set* with the six routines, SBAD4 and SADB4 (see chapter 14), SNEG4, ABS4, SMUL4 and SDIV4. These were improved by Vincent Fojut to the 8 routines, SBAD4, SADB4Q and SADB4S (given in chapter 14), SNEG4, ABS4, ABSNEG, SMUL4 and SDIV4.

INTERFACING

The suite has a standard Input/Output interfacing protocol. It uses the full block of 16 page zero pseudo-registers, named M0 to MF for *Sub Set* Datasheet purposes (see Introduction) with the following variable assignments:

M0 M1 M2 M3

Primary 32-bit accumulator (AZA). Used for the binary representation in the conversion routines, the value to be negated in the negate routine and for output of product or quotient.

ARITHMETIC: 32-BIT

M4 M5 M6 M7

Secondary 32-bit accumulator (AZB). Used for input of multiplier or dividend.

M8 M9 MA MB

Tertiary 32-bit accumulator (AZC). Used for input of multiplier and or divisor and for output of remainder.

MC MD

String Pointer. Used for the address of the source or destination ASCII decimal digit string in the conversion routines.

ME and MF

Temporary storage locations. Used for storing index register contents and product, quotient and remainder signs.

In all cases the low-order byte of an address or value is stored in the lowest memory location of the variable.

Errors are flagged by setting Carry – clearing Carry for a successful operation – in any routine where invalid input is possible.

ABSOLUTE AND NEGATIVE VALUES

ABSNEG doesn't properly belong in the suite: it returns the absolute value of a 32-bit number at 4 page zero locations addressed by X; the other members of the suite all act on variables within the 16-byte block. ABS4 calls ABSNEG twice to convert the secondary and tertiary accumulators to their absolute values after first calculating the sign of their product or quotient by exclusive-ORing the 2 sign bits. Within the suite, ABSNEG is merely a local subroutine of ABS4; outside it could be very useful in many contexts.

SNeg4 performs much the same operation of subtracting a 32-bit value from 0 as does ABSNEG but does it whatever the sign of input value. It also moves the result sign into the Carry flag but in doing so it unfortunately destroys the information about the zero status of the negated variable; the Carry out of the negation is set only if the variable is zero.

```
=====
:= ABSNEG   Absolute value of a 4-byte signed integer.
=====
:JOB       To return the absolute value of a 32-bit signed
:          integer using two's complement notation.
:ACTION    IF number is negative THEN:
:          [ Index from low order byte.
:            Ensure no borrow in.
:            FOR count of 4:
:            [ Accumulator = 0 - indexed byte.
:              Indexed byte = accumulator.
:              Index next byte. ] ]
```

```

-----
:CPU          6502
:HARDWARE     None.
:SOFTWARE     None.
-----
:INPUT        4 page zero locations at 0,X to 3,X contain the
:              number.
:OUTPUT       X = input X + 4.
:              4 page zero locations at -4,X to -1,X contain
:              the absolute value of the input number.
:              A = number hi-byte. Y = 0. P is changed.
:ERRORS       The absolute value of $80000000 cannot be
:              represented in 32-bit 2's complement notation.
:              Arithmetic error if D = 1 (decimal mode).
:REG USE      P A X Y
:STACK USE    None.
:RAM USE      None.
:LENGTH      18
:CYCLES       83 (or 13 if positive value input).
-----
:CLASS 2      -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
:=====
:
ABSNEG LDA 3,X      :Test sign of input number          B5 03
      BPL EXIT      :and skip if already positive.      10 0D
:
      LDY #4        :Set count for 4 bytes.              A0 04
      SEC          :No borrow in to subtraction.        38
:
ABNGLP LDA #0       :Loop, subtracting all four          A9 00
      SBC 0,X      :bytes in turn from 0, with any      F5 00
      STA 0,X      :borrows from previous bytes, and   95 00
      INX          :restore negated result to same      E8
      DEY          :locations. X indexes bytes and      88
      BNE ABNGLP   :Y counts 'em off.                  D0 F6
:
EXIT RTS           :Exit, number absolute.              60
=====

```

```

=====
:= ABS4      Absolute value of two 4-byte signed integers.
=====
:JOB         To return the absolute value of two 32-bit
:             signed integers using two's complement notation
:             and the sign difference of the input values.
:ACTION      Sign difference = sign 1st no. EOR sign 2nd no.
:             Index 1st number and call absolute routine.
:             Index 2nd number and call absolute routine.
=====
:CPU          6502
:HARDWARE     None.
:SOFTWARE     "ABSNEG" - a subroutine to test the 4-byte page
:             zero value indexed by X and negate if negative.
=====

```

ARITHMETIC: 32-BIT

```
:INPUT      M4,5,6,7 contains 1st 32-bit number.
:           M8,9,A,B contains 2nd 32-bit number.
:OUTPUT     Both M4,5,6,7 and M8,9,A,B contain the absolute
:           values of the numbers input. X = 0.
:           ME contains the sign difference of the two input
:           numbers (ME = 1 if the signs were different).
:           P and A are changed.
:ERRORS     The absolute value of $80000000 cannot be
:           represented in 32-bit 2's complement notation.
:REG USE    P A X
:STACK USE  2
:RAM USE    M4 to MB and ME
:LENGTH    19
:CYCLES     33 + 2 * ABSNEG cycles.
```

```
:-----
:CLASS 2    -discreet *interruptable *promable
:-*****-   *reentrant *relocatable -robust
:=====
```

```
:
:AZB      =   M4      :4-byte accumulator holding 1st number.
:AZC      =   M8      :4-byte accumulator holding 2nd number.
:SGN      =   ME      :1 byte to output sign difference of
:                   :the two input numbers.
:
:ABS4     LDA  AZB+3   :Exclusive-or 1st no. hi-byte      A5 M7
:           EOR  AZC+3   :with 2nd no. hi-byte, storing    45 MB
:           STA  SGN     :sign difference in bit 7,ME.      85 ME
:
:           LDX  #AZB    :Index 1st number from $00 and    A2 M4
:           JSR  ABSNEG  :convert it to absolute value.     20 lo hi
:
:           LDX  #AZC    :Index 2nd number from $00 and    A2 M8
:           JSR  ABSNEG  :convert it to absolute value.     20 lo hi
:
:           LDX  #0      :Exit with X cleared to a          A2 00
:           RTS          :useful value of zero.             60
```

```
:=====
```

```
:=====
:= SNEG4    Negate 4-byte signed integer.
:=====
```

```
:JOB        To negate a 32-bit signed integer using two's
:           complement notation, returning sign status of
:           the result.
:ACTION     Index from low order byte.
:           Ensure no borrow in.
:           FOR count of 4:
:           [ Accumulator = 0 - indexed byte.
:             Indexed byte = accumulator.
:             Index next byte. ]
:           Move accumulator highest bit (sign) to Carry.
```

```
:-----
:CPU        6502
:HARDWARE   None.
:SOFTWARE   None.
:-----
```

ARITHMETIC: 32-BIT

```
:INPUT      MO,1,2,3 contains 32-bit number to be negated.
:OUTPUT     MO,1,2,3 contains negated 32-bit number.
:           C = number sign bit (positive if C = 0).
:           X = 0. A is changed.
:ERRORS     The negative value, $80000000, cannot be negated
:           since it has no positive equivalent in 32-bit
:           2's complement notation.
:           Arithmetic error if D = 1 (decimal mode).
:REG USE     P A X
:STACK USE   None.
:RAM USE     MO M1 M2 M3
:LENGTH     14
:CYCLES      71
:-----
:CLASS 2     -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
:=====
:
AZA    =      MO           :4-byte accumulator holding number.
:
SNEG4   LDX  #-4           :Index from lowest byte.             A2 FC
        SEC              :To subtract with no borrow in.       38
:
SNEGLP  LDA  #0            :Loop, subtracting MO, M1, M2         A9 00
        SBC  AZA+4,X       :and M3 in turn from 0, with any     F5 M4
        STA  AZA+4,X       :borrows from previous bytes, and   95 M4
        INX              :restore negated result to same        E8
        BMI  SNEGLP        :locations. Leave A = hi-byte.       30 F7
:
        ASL  A             :Move negated value sign bit to      0A
        RTS              :Carry and exit.                       60
:=====
```

MULTIPLICATION AND DIVISION

Both multiplication and division are straightforward, using the same 'long' operations as the 16-bit operations of the last chapter for the main process. The difference lies in the need to convert possibly negative values to a positive form (i.e. find their absolute values) before the operation can be performed and to negate the appropriate result variable if any of the product, quotient or remainder signs are negative.

```
:=====
:= SMUL4     4-byte signed integer multiplication.
:=====
:JOB         To multiply two 32-bit signed integers using
:           two's complement notation returning the 32-bit
:           signed product or overflow information.
:ACTION      ON overflow: [ Set overflow flag. Exit. ]
:           Product sign = m'plier sign EOR m'cand sign.
:           Multiplier = absolute multiplier.
:           Multiplicand = absolute multiplicand.
:           Product = 0.
```

ARITHMETIC: 32-BIT

```

:      FOR multiplier bits:
:      [ Product = product * 2.
:        Multiplier = multiplier * 2.
:        Carry = multiplier overflow bit.
:        IF Carry = 1 THEN:
:          [ Product = product + multiplicand. ] ]
:      IF product sign negative THEN:
:      [ Product = 0 - product. ]
:      Set valid result flag.
:-----
:CPU      6502
:HARDWARE  None.
:SOFTWARE  "ABS4" - a subroutine to return the sign
:          difference and absolute values of two 4-byte
:          signed integers.
:          "SNEG4" - a subroutine to negate a 4-byte
:          signed integer.
:-----
:INPUT     M4,5,6,7 holds the 32-bit signed multiplier.
:          M8,9,A,B holds the 32-bit signed multiplicand.
:OUTPUT    All registers changed. ME changed.
:          C = 0: multiplication completed.
:          M0,1,2,3 holds the 32-bit signed product.
:          M4,5,6,7 = 0.
:          M8,9,A,B holds the absolute multiplicand.
:          C = 1: product overflow.
:          State of M0 to MB uncertain.
:ERRORS    Arithmetic error if D = 1 (decimal mode).
:REG USE   P A X Y
:STACK USE  4
:RAM USE    M0 to MB and ME
:LENGTH    63
:CYCLES     Minimum: 1756. Maximum: 4134.
:-----
:CLASS 2   -discreet *interruptable *promable
:-****-    *reentrant *relocatable -robust
:=====
:
AZA  =  M0      :4-byte product accumulator.
AZB  =  M4      :4-byte multiplier accumulator.
AZC  =  M8      :4-byte multiplicand accumulator.
SGN  =  ME      :1 byte containing product sign after
                :call to ABS4.
:
SMUL4 JSR ABS4   :Get sign difference (i.e.      20 lo hi
      STX AZA    :product sign) of m'plier and  86 M0
      STX AZA+1  :m'cand, their absolute values 86 M1
      STX AZA+2  :and clear X. Use X to clear   86 M2
      STX AZA+3  :product accumulator.         86 M3
      LDY #32    :Set count for 32 bits.        A0 20
:
SMLP  ASL AZA    :Shift partial product left   06 M0
      ROL AZA+1  :by one bit (product * 2)      26 M1
      ROL AZA+2  :to next bit place, ready for  26 M2
      ROL AZA+3  :add in of multiplicand.       26 M3
      BMI SMOVF  :Overflow if value goes to     30 26
      BCS SMOVF  :32 or 33 bits.                B0 24
:

```

```

ASL  AZB      :Shift multiplier left by one      06 M4
ROL  AZB+1    :bit, moving next bit out to      26 M5
ROL  AZB+1    :Carry to determine if m'cand     26 M6
ROL  AZB+1    :added to product at this bit     26 M7
BCC  SMLPT    :place, skipping if not.          90 0E
:
LDX  #-4      :Index from lowest bytes.          A2 FC
CLC                                     18
SMADD LDA AZA+4,X :Add multiplicand to product   B5 M4
ADC  AZC+4,X  :at correct place value for       75 MC
STA  AZA+4,X  :bit place of multiplier bit.     95 M4
INX                                     E8
BMI  SMADD    :repeat for all 4 bytes.          30 F7
BVS  SMOVFW   :Exit if overflow.                70 0C
:
SMLPT DEY      :Repeat for all 32 bits           88
BNE  SMLP     :of multiplier.                   D0 D9
:
BIT  SGN      :Test stored sign difference       24 ME
BPL  SMEXIT   :(i.e. product sign), skip if    10 03
JSR  SNEG4    :positive, else negate product.  20 lo hi
SMEXIT CLC     :Set valid result flag, C = 0,    18
RTS                                     :and exit, multiplication done. 60
:
SMOVFW SEC    :Set product overflow flag,       38
RTS           :C = 1, and exit on error.        60
=====

```

```

=====
:= SDIV4      4-byte signed integer division.
=====
:JOB          To divide one 32-bit signed integer by another,
:              using two's complement notation, and returning
:              the 32-bit signed quotient and 32-bit remainder
:              or division by zero information.
:ACTION      IF divisor = 0:
:              THEN:
:              [ Set division by zero flag. ]
:              ELSE:
:              [ Remainder sign = dividend sign.
:                Quotient sign = sign (divisor EOR dividend).
:                Acc-lo = absolute dividend. Acc-hi = 0.
:                Divisor = absolute divisor.
:                FOR acc-lo bits:
:                [ Acc = acc * 2.
:                  IF acc-hi >= divisor THEN:
:                  THEN: [ Acc-hi = acc-hi - divisor.
:                        Quotient = quotient * 2 + 1. ]
:                  ELSE: [ Quotient = quotient * 2. ] ]
:                Remainder = acc-hi.
:                IF remainder sign negative THEN:
:                [ Remainder = 0 - remainder. ]
:                IF quotient sign negative THEN:
:                [ Quotient = 0 - quotient. ]
:                Set valid result flag. ]
=====

```

ARITHMETIC: 32-BIT

```

:CPU          6502
:HARDWARE     None.
:SOFTWARE     "ABS4" - a subroutine to return the sign
:              difference and absolute values of two 4-byte
:              signed integers.
:              "SNEG4" - a subroutine to negate a 4-byte
:              signed integer.
:-----
:INPUT        M4,5,6,7 holds the 32-bit signed dividend.
:              M8,9,A,B holds the 32-bit signed divisor.
:OUTPUT       C = 0: division completed.
:              M0,1,2,3 holds the 32-bit signed quotient.
:              M4,5,6,7 holds absolute value of quotient.
:              M8,9,A,B holds the 32-bit signed remainder.
:              All registers changed. ME and MF changed.
:              C = 1: division by zero error.
:              Input divisor (M8,9,A,B) = 0.
:              A = 0. No other variable changed.
:ERRORS       Arithmetic error if D = 1 (decimal mode).
:REG USE      P A X Y
:STACK USE    4
:RAM USE      M0 to MB, ME and MF
:LENGTH      106
:CYCLES       Minimum: 4237. Maximum: 6505.
:-----
:CLASS 2      -discreet *interruptable *promable
:*****      *reentrant *relocatable -robust
:=====
:
:...Working variables.
SGN  =  ME      :1 byte to hold quotient sign.
SGNR =  MF      :1 byte to hold remainder sign.
AZA  =  M0      :4-byte dividend high half accumulator.
AZB  =  M4      :4-byte dividend low half accumulator,
                :quotient shifted in to low half as
                :dividend shifts up to high half.
AZC  =  M8      :4-byte divisor accumulator.
:
SDIV4 LDA AZC    :Check if divisor is zero          A5 M8
      ORA AZC+1  :by ORing all divisor bytes        05 M9
      ORA AZC+2  :into A, leaving A = 0 only         05 MA
      ORA AZC+3  :if all divisor bytes = 0.          05 MB
      BEQ SDZERO :Exit immediately if zero.          F0 5E
:
      LDA AZB+3  :Store dividend sign as sign        A5 M7
      STA SGNR   :of remainder.                      85 MF
      JSR ABS4   :Get sign difference of divisor     20 lo hi
      STX AZA    :and dividend as quotient sign,    86 M0
      STX AZA+1  :their absolute values and         86 M1
      STX AZA+2  :clear X. Use X to clear high      86 M2
      STX AZA+3  :half of dividend accumulator.    86 M3
      LDY #32    :Set count for 32 bits.             A0 20
:
SDLP  ASL AZB    :Shift low half dividend left      06 M4
      ROL AZB+1  :by one bit moving next             26 M5
      ROL AZB+2  :dividend bit out into Carry       26 M6
      ROL AZB+3  :for move into high half,          26 M7
:

```

ARITHMETIC: 32-BIT

	ROL	AZA	:Rotate dividend high half	26 M0
	ROL	AZA+1	:left by one bit getting next	26 M1
	ROL	AZA+2	:dividend bit into right bit	26 M2
	ROL	AZA+3	:place for divisor subtraction.	26 M3
:				
	LDX	#-4	:Index from lowest bytes.	A2 FC
	SEC		:No borrow in to subtraction.	38
SDSUB	LDA	AZA+4,X	:Subtract divisor from high	B5 M4
	SBC	AZC+4,X	:half of dividend at current	F5 MC
	STA	AZA+4,X	:dividend bit place.	95 M4
	INX		:Index next higher bytes and	E8
	BMI	SDSUB	:repeat for all 4 bytes.	30 F7
	BCS	SDQBIT	:Skip if subtracts okay.	B0 OD
:				
	LDX	#-4	:Index from lowest bytes.	A2 FC
SDADD	LDA	AZA+4,X	:Add divisor back to dividend	B5 M4
	ADC	AZC+4,X	:high half to restore it to	75 MC
	STA	AZA+4,X	:value held before subtraction.	95 M4
	INX		:Index next higher bytes and	E8
	BMI	SDADD	:repeat for all 4 bytes.	30 F7
	BEQ	SDLPT	:Leave quotient bit = zero.	F0 O2
:				
SDQBIT	INC	AZB	:Sub. gone: set quotient bit.	E6 M4
:				
SDLPT	DEY		:Repeat for all 32 bits	88
	BNE	SDLP	:of multiplier.	D0 D0
:				
	BIT	SGNR	:Test stored remainder sign,	24 MF
	BPL	SDTFR	:skip if positive,	10 O3
	JSR	SNEG4	:else negate remainder.	20 lo hi
:				
SDTFR	LDX	#-4	:Index from lowest bytes.	A2 FC
SDTFRL	LDA	AZA+4,X	:Move remainder from dividend	B5 M4
	STA	AZC+4,X	:high half to remainder I/O.	95 MC
	LDA	AZB+4,X	:Move quotient from dividend	B5 M8
	STA	AZA+4,X	:low half to quotient I/O.	95 M4
	INX		:Index next higher bytes and	E8
	BNE	SDTFRL	:repeat for 4 bytes of each.	D0 F5
:				
	BIT	SGN	:Test stored quotient sign,	24 ME
	BPL	SDEXIT	:skip if positive,	10 O3
	JSR	SNEG4	:else negate quotient.	20 lo hi
SDEXIT	CLC		:Set valid result flag, C = 0,	18
	RTS		:and exit, division done.	60
:				
SDZERO	SEC		:Set division by zero flag,	38
	RTS		:C = 1, and exit on error.	60
:				
	:=====			

Index of Routines

This index lists the names of the datasheets in alphabetic order, the labels of all entry points and local subroutines in the datasheets, and the names of subroutines given in the text.

Routine	Page	Description
ABSNEG	166	Absolute value of a 4-byte signed integer.
ABS4	167	Absolute value of two 4-byte signed integers.
ADC4	135	4-byte addition with carry.
ADD4	135	4-byte addition without carry.
ALSORT	54	ASCII sort of variable length strings.
ASLXY	15	Arithmetic Shift Left XY.
ASL4	135	4-byte Arithmetic Shift Left.
BCNV	155	Base conversion: unsigned ASCII (base 2 to 36) to unsigned ASCII (base 2 to 36).
BGCB	158	Binary to or from Gray Code conversion.
BINX	153	32-bit to unsigned ASCII (base 2 to 36) conversion.
BIRCH	44	Break, Interrupt and Relative Call Handler.
BOX	29	Stack registers, block exchange memory/stack.
CASEOF	58	Single byte key CASE routine.
CHKSUM	86	BCD Checksum.
COX	30	Block exchange memory/stack, unstack registers.
CTRPRT	94	Control character name print.
CURT16	132	Integer cube root of a 16-bit unsigned value.
CURT32	134	Integer cube root of a 32-bit unsigned value.
DBLBIN		Double 4-byte value, <i>see</i> SADB4S
DL1S	1	Delay for one second.
DOT1	104	Put graphics dot information.
DOT2	105	Put graphics dot information.
DOT3	106	Put graphics dot information.
DOT4	108	Put graphics dot information.
DOT5	109	Put graphics dot information.
DUBDIV	161	16-bit unsigned division.
DUBMUL	163	16-bit unsigned multiplication.

Routine	Page	Description
ECAL	88	Calculate error correction (parity) byte.
EFIX	90	Validate data with error correction byte.
EXAX	13	Exchange A with X.
EXAY	13	Exchange A with Y.
EXPD	63	Expand 12-bit data.
EXSX	17	Exchange (SP),index register X.
EXSXY	18	(Extended form of EXSX).
EXXY	13	Exchange X with Y.
FIND	40	Find address of memory block.
FOWIA	41	Find Out Where I'm At (Z-80 code).
GETA	28	Get stacked A.
GETLOC	41	Get program location.
GETPC	28	Get Return Address from stack.
GYCON	156	Binary nibble to Gray Code conversion.
HLFX	160	Halve 4-digit BCD string in registers.
IBT	75	Intelligent Block Transfer.
INCA	15	Increment Accumulator A.
INDXY	24	Register Indirect addressing mode emulator.
INSUB		(Constructed by INDXY), <i>see</i> INDXY
LARCH		(Long Address BIRCH extension), <i>see</i> BIRCH
LFEED		(LSTFMT line feed), <i>see</i> LSTFMT
LSR4	135	4-byte Logical Shift Right.
LSTFMT	96	Formatted assembler listing.
LONGBR	48	16-bit offset, complex-conditional branch.
MAKMSG	70	Make message from sub-messages.
MATCH	51	Keyword substring match.
PLXYAP	13	Pull register set X, Y, A, P and PC.
PLYXAP	13	Pull register set Y, X, A, P and PC.
POS		(LSTFMT cursor position read), <i>see</i> LSTFMT
PRAY	150	Print 16-bit value as ASCII decimal.
PSHU	32	Push specified registers to User Stack.
PULU	35	Pull specified registers from User Stack.
RANDI	113	Pseudo-random integer in binary or BCD.
RESTOR	27	Restore registers and 2 page zero bytes.
RINXY	21	Register Indirect addressing mode emulator.
RISUB		(Constructed by RINXY), <i>see</i> RINXY
RLAXY	13	Rotate left by one byte through A, X, Y.
RLTVL	43	Program relative subroutine call.
RND16B	112	Compute 16 random bits.
RND16	116	16-bit pseudo-random number generator.
RND31	121	31-bit pseudo-random number generator.
RND32	119	32-bit pseudo-random number generator.
ROLL		(Graphics byte rotation through Carry), <i>see</i> DOT5

APPENDIX A – INDEX OF ROUTINES

Routine	Page	Description
ROL4	135	4-byte Rotate Left.
ROR4	135	4-byte Rotate Right.
ROTEX	13	Register rotate, transfer and exchange: toolkit.
RRXY	13	Rotate right by one byte through A, X, Y.
RRL	77	Rotate memory right long.
SADB4Q	140	ASCII decimal to 4-byte signed integer (quick).
SADB4S	142	ASCII decimal to 4-byte signed integer (short).
SAVE	27	Save registers and 2 page zero bytes.
SBAD4	145	4-byte signed integer to ASCII decimal.
SBC4	135	4-byte subtraction with carry (borrow).
SDIV4	171	4-byte signed integer division.
SLOWUP	6	Switchable delay.
SMUL4	169	4-byte signed integer multiplication.
SNeg4	168	Negate 4-byte signed integer.
SPLIT0	106	Split graphics dot information.
SPLIT3	105	Split graphics dot information.
SQR15	125	Integer square root of a 16-bit signed (positive) value.
SQR16	125	Integer square root of a 16-bit unsigned (absolute) value.
SQR31	127	Integer square root of a 32-bit signed (positive) value.
SQR32	127	Integer square root of a 32-bit unsigned (absolute) value.
SQSH	62	Squash 12-bit data.
SUB4	135	4-byte subtraction without carry (borrow).
TABOUT		(LSTFMT tabulation), <i>see</i> LSTFMT
TERAXY	16	Register transfer, exchange and rotate: toolkit.
TEXT	73	Output program embedded text.
TKNIN	66	'Tokenise' ASCII text.
TKNOUT	67	Output expanded "Tokenised" text.
TRANS	81	Own-space matrix transposition.
TSY	15	Transfer Stack Pointer to Y.
TXY	13	Transfer X to Y.
TYX	13	Transfer Y to X.
UDELAY	3	Universal delay.
XBIN	151	Unsigned ASCII (base 2 to 36) to 32-bit conversion.
XYMOD	19	Modify absolute address operand.
XYMODS	23	Copy instruction with modified address operand.
XYSUB		(Constructed by SYMODS), <i>see</i> XYMODS

Index

Absolute addresses viii, xi
ACTION (datasheet section) ix
Algorithm construction 129
Array storage 80
Art of Computer Programming, The 115
ASCII 93, 139, 149
Assembler standards xi , 95
Associated key 57
Bases (2-36) 151, 155
BASIC 57, 95
BBC assembler 95
BBC microcomputer 93, 103
BCD (Binary coded decimal) 113, 159
Bit error 88
Bit inversion 88
Block addressing 40
Break flag 44
Breakpoint 44
Brown, P.J. 116
Bubble sorts 53
Checksums 85
CLASS (datasheet section) x
Clock cycles viii
Clock speed 2
Condition tests 47
Control codes 93
Conversion method 139, 145
CPU (datasheet section) ix
Cube root method 129
CYCLES (datasheet section) x
Datasheets v , ix
Decimal mode flag 159
Delay formula 2
Discreet x
Documentation vi , ix
Dummy instructions 2
Dummy opcode 13
Dynamic addressing 39

INDEX

- Elegance 105
- Elegant solution 103
- Embedded parameters 28
- Embedded text 73
- ERRORS (datasheet section) x
- Escapes 69
- Expansion table 64
- Extra addressing modes 11
- Extra NOPs 11
- Formatted listing 95
- Function key 100
- Gray Code 155
- Hadden, David R. (Jr.) vi
- HALT 11
- HARDWARE (datasheet section) ix
- Hex dump 93
- Hex symbol 101
- Hughes, Thomas P. vi
- INPUT (datasheet section) ix
- Intelligent transfer 74
- Interfacing protocol 165
- Interruptable x
- Jargon phrases 71
- JOB (datasheet section) ix
- Keyword search 51
- Knuth, Donald E. 115
- LENGTH (datasheet section) x
- Leventhal, Lance A. 85
- Line feed 100
- Look-up tables 106
- Matrix transposition 80
- Memory block rotation 77
- Microcomputer Software Design* vi
- Modifying code 20
- Multiplication by factorisation 116
- Multiplication by shifting 116, 139
- Negative sign 165
- ON .. GOSUB .. 57
- OUTPUT (datasheet section) x
- Overflow flag 149
- Page zero viii, xi, 27, 29, 165
- Parity coding 88
- PCW *SubSet* v

- Potency 115
- Positive sign 165
- Product sign 166
- Program relative addressing 41, 156
- Promable x
- Pseudo-random sequence 111
- Pseudo-registers x, 29, 165
- Quicker sorting 57
- Quotient sign 166
- RAM USE (datasheet section) x
- Random binary values 111
- Random number formulae 113, 114, 121
- Random number theory 114
- Re-entrant xi
- Register use (datasheet section) x
- Relocatable xi
- Reset 11
- RESTART (Z-80 1-byte subroutine call) 44
- Robust xi
- Sawin, Dwight H. (III) vi
- Seeding 111, 122
- SOFTWARE (datasheet section) ix
- Software interrupt 44
- Space bar 5
- Spectral test 115
- Square root method 129
- Stack exchanges 17
- Stacking error 21
- Stack page 27
- Stack use (datasheet section) x
- Stack workspace 149
- String termination 64
- Subroutines 12
- SubSet*, see *PCW SubSet*
- Time states, see *clock cycles*
- Timing overheads 2, 12, 140
- Tokens 51, 64
- Toolkits 12
- Two's complement 145, 165
- Unspecified instructions 9
- Upper-case ASCII 151
- U.S. Army Electronics Command vi
- User stack 32
- VIA (Versatile Interface Adapter) 112

INDEX

Wait function 5
Weighting 85
Wraparound addressing 74
Wraparound arithmetic 155
Writing Interactive Compilers and Interpreters 166
Zero page, *see* page zero
6502 Assembly Language Programming 85

Every serious assembler programmer needs well-documented sub-routines. This is a collection of the best routines for the 6502 compiled from the famous SUBSET series that has been running in Personal Computer World magazine for several years.

For the first time these invaluable routines have been collected together in a logical way and are introduced and explained to those new to the series by the editor.

Each 'Datasheet' has been documented and thoroughly tested to the standards set by the column.

This book presents the distilled experience of the best machine-code programmers and will be an essential part of any serious assembler programmer's library.

£7.95

ISBN 0 7126 0507 X
CENTURY COMMUNICATIONS LTD

ISBN 0-7126-0507-X



9 780712 605076

KQ-452-941