Coding Algorithms

As you can see from the preceding descriptions the instruction set of the 6502 is quite basic, having only simple 8 bit operations. Complex operations such as 16 or 32 bit arithmetic and memory transfers have to be performed by executing a sequence of simpler operations. This sections describes how to build these algorithms and is based on code taken from my macro library (available from the download section).

If you find any bugs in the code, have routines to donate to the library, or can suggest improvements then please mail me.

Standard Conventions

The 6502 processor expects addresses to be stored in 'little endian' order, with the least significant byte first and the most significant byte second. If the value stored was just a number (e.g. game score, etc.) then we could write code to store and manipulate it in 'big endian' order if we wished, however the algorithms presented here always use 'little endian' order so that they may be applied either to simple numeric values or addresses without modification.

> *The terms 'big endian' and 'little endian' come from Gulliver's Travels. The people of Lilliput and Blefuscu have been fighting a war over which end of an boiled egg one should crack to eat it. In computer terms it refers to whether the most or least significant portion of a binary number is stored in the lower memory address.*

To be safe the algorithms usually start by setting processor flags and registers to safe initial values. If you need to squeeze a few extra bytes or cycles out of the routine you might be able to remove some of these initializations depending on the preceding instructions.

Simple Memory Operations

Probably the most fundamental memory operation is clearing an area of memory to an initial value, such as zero. As the 6502 cannot directly move values to memory clearing even a small region of memory requires the use of a register. Any of A, X or Y could be used to hold the initial value, but in practice A is normally used because it can be quickly saved and restored (with PHA and PLA) leaving X and Y free for application use.

```
; Clearing 16 bits of memory
_CLR16  LDA #0          ;Load constant zero into A
        STA MEM+0       ;Then clear the least significant byte
        STA MEM+1       ;... followed by the most significant

; Clearing 32 bits of memory
_CLR32  LDA #0          ;Load constant zero into A
        STA MEM+0       ;Clear from the least significant byte
        STA MEM+1       ;... up
        STA MEM+2       ;... to
        STA MEM+3       ;... the most significant
```

Moving a small quantity of data requires a register to act as a temporary container during the transfer. Again any of A, X, or Y may be used, but as before using A as the temporary register is often the most practical.

```
; Moving 16 bits of memory
_XFR16  LDA SRC+0       ;Move the least significant byte
        STA DST+0
        LDA SRC+1       ;Then the most significant
        STA DST+1

; Moving 32 bits of memory
_XFR32  LDA SRC+0       ;Move from least significant byte
        STA DST+0
        LDA SRC+1       ;... up
        STA DST+1
        LDA SRC+2       ;... to
        STA DST+2
        LDA SRC+3       ;... the most significant
        STA DST+3
```

Provided the source and destination areas do not overlap then the order in which the bytes are moved is irrelevant, but it usually pays to be consistent in your approach to make mistakes easier to spot.

All of the preceding examples can be extended to apply to larger memory areas but will generate increasingly larger code as the number of bytes involved grows. Algorithms that iterate using a counter and use index addressing to access memory will result in smaller code but will be slightly slower to execute.

This trade off between speed and size is a common issue in assembly language programming and there are times when one approach is clearly better than the other (e.g. when trying to squeeze code into a fixed size ROM - SIZE, or manipulate data during a video blanking period - SPEED).

```
; Clear 32 bits of memory iteratively
_CLR32C LDX #3
        LDA #0
_LOOP   STA MEM,X
        DEX
        BPL _LOOP

; Move 32 bits of memory iteratively
_XFR32C LDX #3
_LOOP   LDA SRC,X
        STA DST,X
        DEX
        BPL _LOOP
```

Another basic operation is setting a 16 bit word to an initial constant value. The easiest way to do this is to load the low and high portions into A one at a time and store them.

```
; Setting a 16 bit constant
_SET16I LDA #LO NUM     ;Set the least significant byte of the constant
        STA MEM+0
        LDA #HI NUM     ;... then the most significant byte
        STA MEM+1
```

Logical Operations

The simplest forms of operation on binary values are the logical AND, logical OR and exclusive OR illustrated by the following truth tables.

Logical AND (AND)

	0	1
0	0	0
1	0	1

Logical OR (ORA)

	0	1
0	0	1
1	1	1

Exclusive OR (EOR)

	0	1
0	0	1
1	1	0

These results can be summarized in English as:

- The result of a logical AND is true (1) if and only if both inputs are true, otherwise it is false (0).
- The result of a logical OR is true (1) if either of the inputs its true, otherwise it is false (0).
- The result of an exclusive OR is true (1) if and only if one input is true and the other is false, otherwise it is false (0).

The tables show result of applying these operations on two one-bit values but as the 6502 comprises of eight bit registers and memory each instruction will operate on two eight bit values simultaneously as shown below.

	Logical AND (AND)	***Logical OR (ORA)***	***Exclusive OR (EOR)***
Value 1	0 0 1 1 0 0 1 1	0 0 1 1 0 0 1 1	0 0 1 1 0 0 1 1
Value 2	0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1
Result	0 0 0 1 0 0 0 1	0 1 1 1 0 1 1 1	0 1 1 0 0 1 1 0

It is important to understand the properties and practical applications of each of these operations as they are extensively used in other algorithms.

- Logical AND operates as a filter and is often used to select a subset of bits from a value (e.g. the status flags from a peripheral control chip).
- Logical OR allows bits to be inserted into an existing value (e.g. to set control flags in a peripheral control chip).
- Exclusive OR allows selected bits to be set or inverted.

In the 6502 these operations are implemented by the AND, ORA and EOR instructions. One of the values to be operated on will be the current contents of the accumulator, the other is in memory either as an immediate value or at a specified location. The result of the operation is placed in the accumulator and the zero and negative flags are set accordingly.

```
; Example logical operations
        AND #$0F        ;Filter out all but the least 4 bits
        ORA BITS,X      ;Insert some bits from a table
        EOR (DATA),Y    ;EOR against some data
```

A very common use of the EOR instruction is to calculate the 'complement' (or logical NOT) of a value. This involves inverting every bit in the value and is most easily calculated by exclusively ORing against an all ones value.

```
; Calculate the complement
        EOR #$FF
```

The macro library contains reference code for 16 and 32 bit AND, ORA, EOR and NOT operations although there is very little use for them outside of interpreters.

Shifts & Rotates

The shift and rotate instructions allow the bits within either the accumulator or a memory location to be moved by one place either up (left) or down (right). When the bits are moved a new value will be needed to fill the vacant position created at one end of the value, and similarly the bit displaced at the opposite end will need to be caught and stored.

Both shifts and rotates catch the displaced bit in the carry flag but they differ in how they fill the vacant position; shifts will always fill the vacant bit with a zero whilst a rotate will fill it with the value of the carry flag as it was at the start of the instruction.

For example the following diagram shows the result of applying an 'Arithmetic Shift Left' (ASL) to the value $4D to give $9A.

```
               +---+---+---+---+---+---+---+---+
Initial:       | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
               +---+---+---+---+---+---+---+---+
                 |   |   |   |   |   |   |   |
                /   /   /   /   /   /   /   /
               /   /   /   /   /   /   /   /    0
              /   /   /   /   /   /   /   /   /
             /   |   |   |   |   |   |   |   |
            /    v   v   v   v   v   v   v   v
               +---+---+---+---+---+---+---+---+
Result:   C=0  | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
               +---+---+---+---+---+---+---+---+
```

Whist the following shows the result of applying a 'Rotate Left' (ROL) to the same value, but assuming that the carry contained the value one.

```
               +---+---+---+---+---+---+---+---+
Initial:       | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | C=1
               +---+---+---+---+---+---+---+---+
                 |   |   |   |   |   |   |   |   /
                /   /   /   /   /   /   /   /   /
               /   /   /   /   /   /   /   /   /
              /   /   /   /   /   /   /   /   /
             /   |   |   |   |   |   |   |   |
            /    v   v   v   v   v   v   v   v
               +---+---+---+---+---+---+---+---+
Result:   C=0  | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
               +---+---+---+---+---+---+---+---+
```

Shifting the bits within a value (and introducing a zero as the least significant bit) has the effect of multiplying its value by two. In order to apply this multiplication to a value larger than a single byte we use ASL to shift the first byte and then ROL all the subsequent bytes as necessary using the carry flag to temporarily hold the displaced bits as they are moved from one byte to the next.

```
; Shift a 16bit value by one place left (e.g. multiply by two)
_ASL16  ASL MEM+0       ;Shift the LSB
        ROL MEM+1       ;Rotate the MSB
```

The behavior of the right shift as rotates follows the same pattern. For example we can apply a 'Logical Shift Right' (LSR) to the value $4D to give $26.

```
               +---+---+---+---+---+---+---+---+
Initial:       | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
               +---+---+---+---+---+---+---+---+
                 |   |   |   |   |   |   |   |
                  \   \   \   \   \   \   \   \
              0    \   \   \   \   \   \   \   \
                \   \   \   \   \   \   \   \   \
                 |   |   |   |   |   |   |   |   \
                 v   v   v   v   v   v   v   v    \
               +---+---+---+---+---+---+---+---+
Result:        | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |  C=1
               +---+---+---+---+---+---+---+---+
```

Or a 'Rotate Right' (ROR) of the same value, but assuming that the carry contained the value one to give $A6.

```
               +---+---+---+---+---+---+---+---+
Initial:   C=1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
               +---+---+---+---+---+---+---+---+
             \   |   |   |   |   |   |   |   |
              \   \   \   \   \   \   \   \   \
               \   \   \   \   \   \   \   \   \
                \   \   \   \   \   \   \   \   \
                 |   |   |   |   |   |   |   |   \
                 v   v   v   v   v   v   v   v    \
               +---+---+---+---+---+---+---+---+
Result:        | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |  C=1
               +---+---+---+---+---+---+---+---+
```

Not surprisingly if left shifts multiply a value by two then right shifts do an unsigned division by two. Again if we are applying the division to a multi-byte value we will typically use LSR on the first byte (the MSB this time) and ROR on all subsequent bytes.

```
; Shift a 16 bit value by one place right (e.g. divide by two)
_LSR16  LSR MEM+1       ;Shift the MSB
        ROR MEM+0       ;Rotate the LSB
```

There are a number of applications for shifts and rotates, not least the coding of generic multiply and divide algorithms which are discussed later.

As was pointed out earlier right shifting a value two divide it by two only works on unsigned values. This is because the LSR is will always place a zero in the most significant bit of the MSB. To make this algorithm work for all two complement coded values we need to ensure that value of this bit is copied back into itself to keep the value the same sign. We can use another shift to achieve this.

```
; Divide a signed 16 bit value by two
_DIV2   LDA MEM+1       ;Load the MSB
        ASL A           ;Copy the sign bit into C
        ROR MEM+1       ;And back into the MSB
        ROR MEM+0       ;Rotate the LSB as normal
```

Addition & Subtraction

The 6502 processor provides 8 bit addition and subtraction instructions and a carry/borrow flag that is used to propagate the carry bit between operations.

To implement a 16 bit addition the programmer must code two pairs of additions; one for the least significant bytes and one for the most significant bytes. The carry flag must be cleared before the first addition to ensure that an additional increment isn't performed.

```
; 16 bit Binary Addition
        CLC             ;Ensure carry is clear
        LDA VLA+0       ;Add the two least significant bytes
        ADC VLB+0
        STA RES+0       ;... and store the result
        LDA VLA+1       ;Add the two most significant bytes
        ADC VLB+1       ;... and any propagated carry bit
        STA RES+1       ;... and store the result
```

Subtraction follows the same pattern but the carry must be set before the first pair of bytes are subtracted to get the correct result.

```
; 16 bit Binary Subtraction
        SEC             ;Ensure carry is set
        LDA VLA+0       ;Subtract the two least significant bytes
        SBC VLB+0
        STA RES+0       ;... and store the result
        LDA VLA+1       ;Subtract the two most significant bytes
        SBC VLB+1       ;... and any propagated borrow bit
        STA RES+1       ;... and store the result
```

Both the addition and subtraction algorithm can be extended to 32 bits by repeating the LDA/ADC/STA or LDA/SBC/STA pattern for two further bytes worth of data.

Negation

The traditional approach to negating a twos complement number is to reverse all the bits (by EORing with $FF) and add one as shown below.

```
; 8 bit Binary Negation
        CLC             ;Ensure carry is clear
        EOR #$FF        ;Invert all the bits
        ADC #1          ;... and add one
```

This technique works well with a single byte already held in the accumulator but not with bigger numbers. With these it is easier just to subtract them from zero.

```
; 16 bit Binary Negation
        SEC             ;Ensure carry is set
        LDA #0          ;Load constant zero
        SBC SRC+0       ;... subtract the least significant byte
        STA DST+0       ;... and store the result
        LDA #0          ;Load constant zero again
        SBC SRC+1       ;... subtract the most significant byte
        STA DST+1       ;... and store the result
```

Decimal Arithmetic

The behavior of the ADC and SBC instructions can be modified by setting or clearing the decimal mode flag in the processor status register. Normally decimal mode is disabled and ADC/SBC perform simple binary arithmetic (e.g. $99 + $01 => $9A Carry = 0), but if the flag is set with a SED instruction the processor will perform binary coded decimal arithmetic instead (e.g. $99 + $01 => $00 Carry = 1).

To make the 16 bit addition/subtraction code work in decimal mode simply include an SED at the start and a CLD at the end (to restore the processor to normal).

```
; 16 bit Binary Code Decimal Addition
        SED             ;Set decimal mode flag
        CLC             ;Ensure carry is clear
        LDA VLA+0       ;Add the two least significant bytes
        ADC VLB+0
        STA RES+0       ;... and store the result
        LDA VLA+1       ;Add the two most significant bytes
        ADC VLB+1       ;... and any propagated carry bit
        STA RES+1       ;... and store the result
        CLD             ;Clear decimal mode
```

Binary coded values are more easily converted to displayable digits and are useful for holding numbers such as high scores.

```
; Print the BCD value in A as two ASCII digits
        PHA             ;Save the BCD value
        LSR A           ;Shift the four most significant bits
        LSR A           ;... into the four least significant
        LSR A
        LSR A
        ORA #'0'        ;Make an ASCII digit
        JSR PRINT       ;... and print it
        PLA             ;Recover the BCD value
        AND #$0F        ;Mask out all but the bottom 4 bits
        ORA #'0'        ;Make an ASCII digit
        JSR PRINT       ;... and print it
```

Another use for BCD is in the conversion of binary values to decimal ones. Some algorithms perform this conversion by counting the number of times that 10000's, 1000's, 100's, 10's and 1's can be subtracted from the binary value before it underflows, but I normally use a simple fixed loop that shifts the bits out of the binary value one at a time and adds it to an intermediate result that is being doubled (in BCD) on each iteration.

```
; Convert an 16 bit binary value into a 24bit BCD value
BIN2BCD LDA #0          ;Clear the result area
        STA RES+0
        STA RES+1
        STA RES+2
        LDX #16         ;Setup the bit counter
        SED             ;Enter decimal mode
_LOOP   ASL VAL+0       ;Shift a bit out of the binary
        ROL VAL+1       ;... value
        LDA RES+0       ;And add it into the result, doubling
        ADC RES+0       ;... it at the same time
        STA RES+0
        LDA RES+1
        ADC RES+1
        STA RES+1
        LDA RES+2
        ADC RES+2
        STA RES+2
        DEX             ;More bits to process?
        BNE _LOOP
        CLD             ;Leave decimal mode
```

One final odd use of decimal arithmetic is the conversion of hexadecimal digits to printable ASCII characters. The usual way to perform this conversion is to add $30 to the digit ($00 - $0F) to make an intermediate result which is then examined to see if it is greater than or equal to $3A. If it is then an additional $06 is added to make the result fall in the range $41 - $46 (e.g. 'A' - 'F').

```
; Convert a hex digit ($00-$0F) to ASCII ('0'-'9' or 'A'-'F')
HEX2ASC ORA #$30        ;Form the basic character code
        CMP #$3A        ;Does the result need adjustment?
        BCC .+4
        ADC #$05        ;Add 6 (5 and the carry) if needed
```

It turns out that in decimal mode the processor does basically the same correction after an addition and with the right arguments we can convert the digit to its ASCII character without performing any comparisons as shown in the following code.

```
; Convert a hex digit ($00-$0F) to ASCII ('0'-'9' or 'A'-'F')
HEX2ASC SED             ;Enter BCD mode
        CLC             ;Ensure the carry is clear
        ADC #$90        ;Produce $90-$99 (C=0) or $00-$05 (C=1)
        ADC #$40        ;Produce $30-$39 or $41-$46
        CLD             ;Leave BCD mode
```

Increments & Decrements

Assembly programs frequently use memory based counters that occasionally need incrementing or decrementing by one. One way to achieve this would be to load the LSB and MSB in turn and add or subtract one with the ADC/SBC instructions, but the 6502 has a more efficient way to do this using INC and DEC.

Incrementing is straight forward, we just increment the least significant byte until the result becomes zero. This indicates that the calculation has wrapped round (e.g. $FF + $01 => $00) and an increment to the most significant byte is needed.

```
; Increment a 16 bit value by one
_INC16  INC MEM+0       ;Increment the LSB
        BNE _DONE       ;If the result was not zero we're done
        INC MEM+1       ;Increment the MSB if LSB wrapped round
_DONE   EQU *
```

Decrementing is a little trickier because we need to know when the least significant byte is about to underflow from $00 to $FF. The answer is to test it first by loading it into the accumulator to set the processor flags.

```
; Decrement a 16 bit value by one
_DEC16  LDA MEM+0       ;Test if the LSB is zero
        BNE _SKIP       ;If it isn't we can skip the next instruction
        DEC MEM+1       ;Decrement the MSB when the LSB will underflow
_SKIP   DEC MEM+0       ;Decrement the LSB
```

Complex Memory Transfers

Moving data from one place to another is a common operation. If the amount of data to moved is 256 bytes or less and the source and target locations of the data are fixed then a simple loop around an indexed LDA followed by an indexed STA is the most efficient. Note that whilst both the X and Y registers can be used in indexed addressing modes an asymmetry in the 6502's instruction means that X is the better register to use if one or both of the memory areas resides on zero page.

```
; Move 256 bytes or less in a forward direction
        LDX #0          ;Start with the first byte
_LOOP   LDA SRC,X       ;Move it
        STA DST,X
        INX             ;Then bump the index ...
        CPX #LEN        ;... until we reach the limit
        BNE _LOOP
```

The corresponding code moving the last byte first is as follows:

```
; Move 256 bytes or less in a reverse direction
        LDX #LEN        ;Start with the last byte
_LOOP   DEX             ;Bump the index
        LDA SRC,X       ;Move a byte
        STA DST,X
        CPX #0          ;... until all bytes have moved
        BNE _LOOP
```

If the amount is even smaller (128 bytes or less) then we can eliminate the comparison against the limit and use the settings of the flags after a DEX to determine if the loop has finished.

```
; Move 128 bytes or less in a reverse direction
        LDX #LEN-1      ;Start with the last byte
_LOOP   LDA SRC,X       ;Move it
        STA DST,X
        DEX             ;Then bump the index ...
        BPL _LOOP       ;... until all bytes have moved
```

To create a completely generic memory transfer we must change to using indirect indexed addressing to access memory and use all the registers. The following code shows a forward transferring algorithm which first moves complete pages of 256 bytes followed by any remaining fragments of smaller size.

```
_MOVFWD LDY #0          ;Initialise the index
        LDX LEN+1       ;Load the page count
        BEQ _FRAG       ;... Do we only have a fragment?
_PAGE   LDA (SRC),Y     ;Move a byte in a page transfer
        STA (DST),Y
        INY             ;And repeat for the rest of the
        BNE _PAGE       ;... page
        INC SRC+1       ;Then bump the src and dst addresses
        INC DST+1       ;... by a page
        DEX             ;And repeat while there are more
        BNE _PAGE       ;... pages to move
_FRAG   CPY LEN+0       ;Then while the index has not reached
        BEQ _DONE       ;... the limit
        LDA (SRC),Y     ;Move a fragment byte
        STA (DST),Y
        INY             ;Bump the index and repeat
        BNE _FRAG\?
_DONE   EQU *           ;All done
```

Character Classification

The standard C library provides a set of functions for classifying (e.g. is letter, is digit, is ASCII, is upper case, etc.) and modifying (e.g. to upper case and to lower case) characters defined in a header called <ctype.h>. This section describes how a similar set of functions can be coded in 6502 assembler. There are two techniques that can be applied to solve this problem, namely, comparisons or look up tables.

Note: These functions will be restricted to just the normal ASCII character range $00-$7F.

The look up table required to implement character classification needs a byte per character. Bits within the look up table indicate how the character is to be classified (e.g. control character, printable character, white space, decimal digit, hexadecimal digit, punctuation, upper case latter or lower case letter). To test a character for a specific classification you load its description byte from the table and test for the presence of certain bits (e.g. with AND).

```
; Constants describing the role of each classification bit
_CTL    EQU $80
_PRN    EQU $40
_WSP    EQU $20
_PCT    EQU $10
_UPR    EQU $08
_LWR    EQU $04
_DGT    EQU $02
_HEX    EQU $01

; Test if the character in A is a control character
ISCNTRL TAX
        LDA #_CTL
        BNE TEST

; Test if the character in A is printable
ISPRINT TAX
        LDA #_PRN
        BNE TEST

; Test if the character in A is punctation
ISPUNCT TAX
        LDA #_PCT
        BNE TEST

; Test if the character in A is upper case
ISUPPER TAX
        LDA #_UPR
        BNE TEST

; Test if the character in A is lower case
ISLOWER TAX
        LDA #_LWR
        BNE TEST

; Test if the character in A is a letter
ISALPHA TAX
        LDA #_UPR|_LWR
        BNE TEST

; Test if the character in A is a decimal digit
ISDIGIT TAX
        LDA #_DGT
        BNE TEST

; Test if the character in A is a hexadecimal digit
ISXDIGIT TAX
        LDA #_HEX
        BNE TEST

; Test if the character in A is letter or a digit
ISALNUM TAX
        LDA #_DGT|_UPR|_LWR

; Tests for the required bits in the look up table value
TEST    AND CTYPE,X
        BEQ FAIL

; Set the carry flag if any target bits were found
PASS    TXA
```

```
        SEC
        RTS

; Test if the character in A is in the ASCII range $00-$7F
ISASCII TAX
        BPL PASS

; Clear the carry flag if no target bits were found
FAIL    TXA
        CLC
        RTS

; If A contains a lower case letter convert it to upper case
TOUPPER JSR ISLOWER
        BCC *+4
        AND #$DF
        RTS

; If A contains an upper case letter convert it to lower case
TOLOWER JSR ISUPPER
        BCC *+4
        ORA #$20
        RTS

; The lookup table of character descriptions
CTYPE   DB  _CTL                ; NUL
        DB  _CTL                ; SOH
        DB  _CTL                ; STX
        DB  _CTL                ; ETX
        DB  _CTL                ; EOT
        DB  _CTL                ; ENQ
        DB  _CTL                ; ACK
        DB  _CTL                ; BEL
        DB  _CTL                ; BS
        DB  _CTL|_WSP           ; TAB
        DB  _CTL|_WSP           ; LF
        DB  _CTL|_WSP           ; VT
        DB  _CTL|_WSP           ; FF
        DB  _CTL|_WSP           ; CR
        DB  _CTL                ; SO
        DB  _CTL                ; SI
        DB  _CTL                ; DLE
        DB  _CTL                ; DC1
        DB  _CTL                ; DC2
        DB  _CTL                ; DC3
        DB  _CTL                ; DC4
        DB  _CTL                ; NAK
        DB  _CTL                ; SYN
        DB  _CTL                ; ETB
        DB  _CTL                ; CAN
        DB  _CTL                ; EM
        DB  _CTL                ; SUB
        DB  _CTL                ; ESC
        DB  _CTL                ; FS
        DB  _CTL                ; GS
        DB  _CTL                ; RS
        DB  _CTL                ; US
        DB  _PRN|_WSP           ; SPACE
        DB  _PRN|_PCT           ; !
        DB  _PRN|_PCT           ; "
        DB  _PRN|_PCT           ; #
        DB  _PRN|_PCT           ; $
        DB  _PRN|_PCT           ; %
        DB  _PRN|_PCT           ; &
        DB  _PRN|_PCT           ; '
        DB  _PRN|_PCT           ; (
        DB  _PRN|_PCT           ; )
        DB  _PRN|_PCT           ; *
        DB  _PRN|_PCT           ; +
        DB  _PRN|_PCT           ; ,
        DB  _PRN|_PCT           ; -
        DB  _PRN|_PCT           ; .
        DB  _PRN|_PCT           ; /
        DB  _PRN|_DGT|_HEX      ; 0
        DB  _PRN|_DGT|_HEX      ; 1
        DB  _PRN|_DGT|_HEX      ; 2
        DB  _PRN|_DGT|_HEX      ; 3
        DB  _PRN|_DGT|_HEX      ; 4
        DB  _PRN|_DGT|_HEX      ; 5
        DB  _PRN|_DGT|_HEX      ; 6
        DB  _PRN|_DGT|_HEX      ; 7
        DB  _PRN|_DGT|_HEX      ; 8
        DB  _PRN|_DGT|_HEX      ; 9
        DB  _PRN|_PCT           ; :
        DB  _PRN|_PCT           ; ;
        DB  _PRN|_PCT           ; <
        DB  _PRN|_PCT           ; =
        DB  _PRN|_PCT           ; >
        DB  _PRN|_PCT           ; ?
        DB  _PRN|_PCT           ; @
        DB  _PRN|_UPR|_HEX      ; A
        DB  _PRN|_UPR|_HEX      ; B
        DB  _PRN|_UPR|_HEX      ; C
        DB  _PRN|_UPR|_HEX      ; D
        DB  _PRN|_UPR|_HEX      ; E
        DB  _PRN|_UPR|_HEX      ; F
        DB  _PRN|_UPR           ; G
        DB  _PRN|_UPR           ; H
        DB  _PRN|_UPR           ; I
        DB  _PRN|_UPR           ; J
        DB  _PRN|_UPR           ; K
        DB  _PRN|_UPR           ; L
        DB  _PRN|_UPR           ; M
        DB  _PRN|_UPR           ; N
        DB  _PRN|_UPR           ; O
        DB  _PRN|_UPR           ; P
        DB  _PRN|_UPR           ; Q
        DB  _PRN|_UPR           ; R
        DB  _PRN|_UPR           ; S
        DB  _PRN|_UPR           ; T
        DB  _PRN|_UPR           ; U
        DB  _PRN|_UPR           ; V
        DB  _PRN|_UPR           ; W
        DB  _PRN|_UPR           ; X
        DB  _PRN|_UPR           ; Y
        DB  _PRN|_UPR           ; Z
        DB  _PRN|_PCT           ; [
        DB  _PRN|_PCT           ; \
        DB  _PRN|_PCT           ; ]
        DB  _PRN|_PCT           ; ^
        DB  _PRN|_PCT           ; _
        DB  _PRN|_PCT           ; `
        DB  _PRN|_LWR|_HEX      ; a
        DB  _PRN|_LWR|_HEX      ; b
        DB  _PRN|_LWR|_HEX      ; c
        DB  _PRN|_LWR|_HEX      ; d
        DB  _PRN|_LWR|_HEX      ; e
        DB  _PRN|_LWR|_HEX      ; f
        DB  _PRN|_LWR           ; g
        DB  _PRN|_LWR           ; h
        DB  _PRN|_LWR           ; i
        DB  _PRN|_LWR           ; j
        DB  _PRN|_LWR           ; k
        DB  _PRN|_LWR           ; l
```

```
          DB  _PRN|_LWR           ; m
          DB  _PRN|_LWR           ; n
          DB  _PRN|_LWR           ; o
          DB  _PRN|_LWR           ; p
          DB  _PRN|_LWR           ; q
          DB  _PRN|_LWR           ; r
          DB  _PRN|_LWR           ; s
          DB  _PRN|_LWR           ; t
          DB  _PRN|_LWR           ; u
          DB  _PRN|_LWR           ; v
          DB  _PRN|_LWR           ; w
          DB  _PRN|_LWR           ; x
          DB  _PRN|_LWR           ; y
          DB  _PRN|_LWR           ; z
          DB  _PRN|_PCT           ; {
          DB  _PRN|_PCT           ; |
          DB  _PRN|_PCT           ; }
          DB  _PRN|_PCT           ; ~
          DB  _CTL                ; DEL
```

If we use comparisons then each function will consist of a number of comparison stages to determine if a provided character has an appropriate value. In most cases these functions are quite small but one or two of them may involve many stages (e.g. is punctuation). The execution time will vary according to the number of the tests a character is subjected to.

```
ISUPPER CMP #'A'
        BCC FAIL
        CMP #'Z'+1
        BCS FAIL
        ; Drop thru here on success

ISLOWER CMP #'a'
        BCC FAIL
        CMP #'z'+1
        BCS FAIL
        ; Drop thru here on success

ISALPHA CMP #'A'
        BCC FAIL
        CMP #'Z'+1
        BCC PASS
        CMP #'a'
        BCC FAIL
        CMP #'z'+1
        BCS FAIL
PASS    EQU *
        ; Drop thru here on success
```

Which solution is best? As in so many cases it depends on your program. If you only need one or two tests and memory size is an issue then the comparison approach will generate less code but may be slightly slower (for the complex tests), otherwise the look up table is simple and fast.

Some notes on my macro library

As I said in the introduction to this section all of the algorithms presented here are taken from my macro library. Coding simple algorithms like these as macros has several advantages over subroutine libraries on the 6502 processor, namely:

- The assembler adjusts them automatically to zero page or absolute addressing depending on the parameters.
- They can be used either inline (for speed) or expanded into subroutines (to save space) as needed.

- The same macro can be used several times but customized in each case to suit its use at that time.
- The macros can optimize the code they generate under some circumstances (e.g. _XFR16 detects when the source and target addresses are the same and does nothing).

Another feature of the macros is that they will generate code for the 65SC02 processor using the additional instructions on that processor if the assembler defines the correct symbol. (This processor was used in the BBC Microcomputers 6502 second processor that's why I decided to support it).

The routines in the currently library are:

Macro Name	Description
_CLR16	Clears 16 bits of memory to zero
_CLR32	Clears 32 bits of memory to zero
_CLR32C	Clears 32 bits of memory to zero iteratively
_XFR16	Moves 16 bits of memory
_XFR32	Moves 32 bits of memory
_XFR32C	Moves 32 bits of memory iteratively
_SET16I	Load a 16 bit constant into memory
_NOT16	Compute the NOT of a 16 bit value
_NOT32	Compute the NOT of a 32 bit value
_NOT32C	Compute the NOT of a 32 bit value iteratively
_ORA16	Compute the OR of two 16 bit values
_ORA32	Compute the OR of two 32 bit values
_ORA32C	Compute the OR of two 32 bit values iteratively
_AND16	Compute the AND of two 16 bit values
_AND32	Compute the AND of two 32 bit values
_AND32C	Compute the AND of two 32 bit values iteratively
_EOR16	Compute the EOR of two 16 bit values
_EOR32	Compute the EOR of two 32 bit values
_EOR32C	Compute the EOR of two 32 bit values iteratively
_ASL16	Compute the arithmetic left shift of a 16 bit value
_ASL32	Compute the arithmetic left shift of a 32 bit value
_ROL16	Compute the left rotation of a 16 bit value
_ROL32	Compute the left rotation of a 32 bit value
_LSR16	Compute the logical right shift of a 16 bit value
_LSR32	Compute the logical right shift of a 32 bit value
_ROR16	Compute the right rotation of a 16 bit value
_ROR32	Compute the right rotation of a 32 bit value
_INC16	Increment a 16 bit value
_INC32	Increment a 32 bit value
_DEC16	Decrement a 16 bit value
_DEC32	Decrement a 32 bit value
_ADD16	Add two 16 bit values
_ADD32	Add two 32 bit values
_SUB16	Subtract two 16 bit values
_SUB32	Subtract two 32 bit values
_NEG16	Negate a 16 bit value
_NEG32	Negate a 32 bit value
_ABS16	Compute the absolute value of a 16 bit value

_ABS32	Compute the absolute value of a 32 bit value
_MUL16	Calculate the 16 bit product of two 16 bit values
_MUL16X	Calculate the 32 bit product of two 16 bit values
_MUL32	Calculate the 32 bit product of two 32 bit values
_MUL16I	Generate the code for a 16 bit constant multiply
_DIV16	Calculate the 16 bit quotient & remainder of a 16 bit value and 16 bit dividend
_DIV16X	Calculate the 16 bit quotient & remainder of a 32 bit value and 16 bit dividend
_DIV32	Calculate the 32 bit quotient & remainder of a 32 bit value and 32 bit dividend
_CMP16	Compare two 16 bit values
_CMP32	Compare two 32 bit values
_MEMFWD	Move a block for memory a forward direction
_MEMREV	Not Implemented
_MEMCPY	Not Implemented
_STRLEN	Compute the length of a 'C' style string
_STRCPY	Copy a 'C' style string
_STRCMP	Compare two 'C' style strings
_STRNCMP	Not implemented

Examine the code for more details on the macro parameters and usage.

<< Back Home Contents Next >>

This page was last updated on 19th March 2004