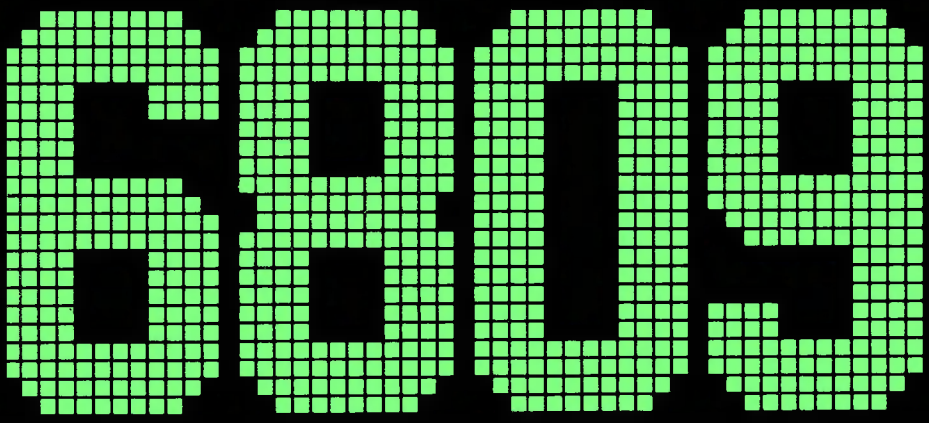
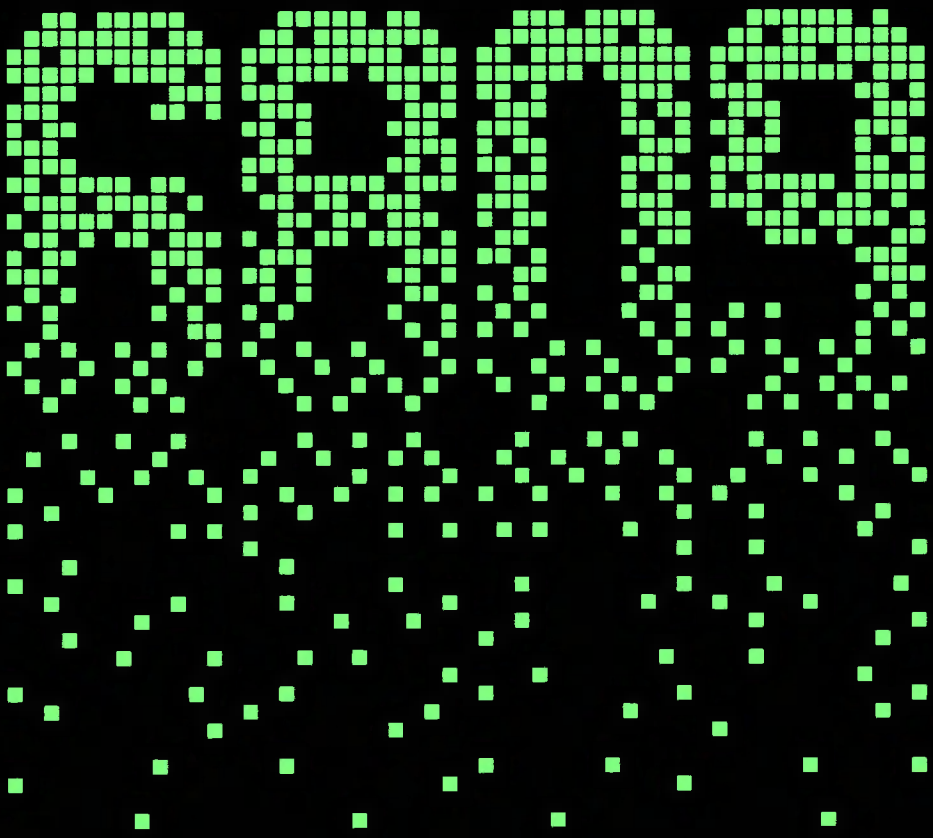
OSBORNE/McGRAW-HILL



6809 ASSEMBLY LANGUAGE SUBROUTINES

BY LANCE A. LEVENTHAL



Assembly language subroutines for the 6809

L. A. LEVENTHAL
and
S. CORDES

McGRAW-HILL BOOK COMPANY

London · New York · St Louis · San Francisco · Auckland
Bogotá · Guatemala · Hamburg · Lisbon · Madrid · Mexico
Montreal · New Delhi · Panama · Paris · San Juan · São Paulo
Singapore · Sydney · Tokyo · Toronto

Published by
McGRAW-HILL Book Company (UK) Limited
MAIDENHEAD • BERKSHIRE • ENGLAND

British Library Cataloguing in Publication Data
Leventhal, Lance A, *1945–*
Assembly language subroutines for the
6809.
1. Motorola 6809 microprocessor systems.
Assembly languages
I. Title II. Cordes, S
005.2'65

ISBN 0-07-707152-2

Library of Congress Cataloguing-in-Publication Data
Leventhal, Lance A, 1945–
Assembly language subroutines for the 6809 / L. A. Leventhal and S. Cordes
p. cm.
Includes index.
ISBN 0-07-707152-2
1. Motorola 6809 (Computer) -- Programming. 2. Assembler language
(Computer program language) 3. Subroutines (Computer programs)
I. Cordes, S. II. Title.
QA76.8.M689L49 1989
005.265--dc 19 88-39561

First published in Japanese

12348909
Typeset by Ponting-Green Publishing Services, London,
and printed and bound in Great Britain at the University Press, Cambridge

Contents

Preface

This book is intended as both a source and a reference for the 6809 assembly language programmer. It contains a collection of useful subroutines described in a standard format and accompanied by an extensive documentation package. All subroutines employ standard parameter passing techniques and follow the rules from the most popular assembler. The documentation covers the procedure, parameters, results, execution time, and memory usage; it also includes at least one example.

The collection emphasizes common tasks that occur in many applications. These tasks include code conversion, array manipulation, arithmetic, bit manipulation, shifting functions, string manipulation, sorting, and searching. We have also provided examples of input/output (I/O) routines, interrupt service routines, and initialization routines for common family chips such as parallel interfaces, serial interfaces, and timers. You should be able to use these programs as subroutines in actual applications and as starting points for more complex programs.

This book is intended for the person who wants to use assembly language immediately, rather than just learn about it. The reader could be

- An engineer, technician, or programmer who must write assembly language programs for a design project.

- A microcomputer user who wants to write an I/O driver, a diagnostic program, a utility, or a systems program in assembly language.

- An experienced assembly language programmer who needs a quick review of techniques for a particular microprocessor.
- A system designer who needs a specific routine or technique for immediate use.
- A high-level language programmer who must debug or optimize programs at the assembly level or must link a program written in a high-level language to one written in assembly language.
- A maintenance programmer who must understand quickly how specific assembly language programs work.
- A microcomputer owner who wants to understand the operating system of a particular computer, or who wants to modify standard I/O routines or systems programs.
- A student, hobbyist, or teacher who wants to see examples of working assembly language programs.

This book can also serve as a supplement for students of the Assembly Language Programming series.

This book should save the reader time and effort. The reader should not have to write, debug, test, or optimize standard routines, or search through a textbook for particular examples. The reader should instead be able to obtain easily the specific information, technique, or routine he or she needs.

Obviously, a book with such an aim demands feedback from its readers. We have, of course, tested all programs thoroughly and documented them carefully. If you find any errors, please inform the publisher. If you have suggestions for better methods or for additional topics, routines, or programming hints, please tell us about them. We have used our programming experience to develop this book, but we need your help to improve it. We would greatly appreciate your comments, criticisms, and suggestions.

Nomenclature

We have used the following nomenclature in this book to describe the architecture of the 6809 processor, to specify operands, and to represent general values of numbers and addresses.

6809 architecture

Figure N-1 shows the register structure of the 6809 microprocessor. Its byte-length registers are:

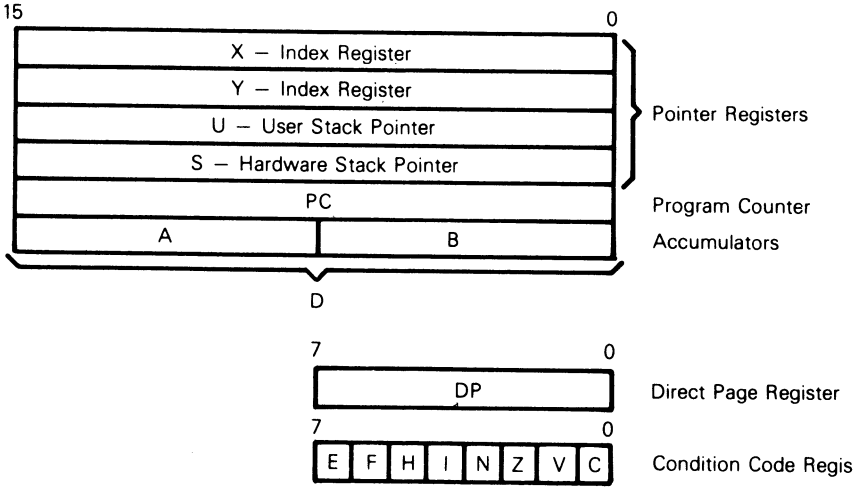


Figure N-1 6809 register structure.

A (accumulator A)
B (accumulator B)
CC (condition code register)
DP (direct page register)

The CC register consists of bits with independent functions and meanings, arranged as shown in Figure N-2.

The 6809's word-length registers are:

D (double accumulator, same as A and B together with A being the more significant byte)

PC (program counter)

S or SP (hardware stack pointer)

U (user stack pointer)

X (index register X)

Y (index register Y)

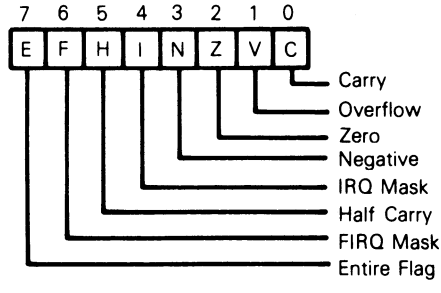


Figure N-2 6809 condition code (CC) register.

The 6809's flags (see Figure N-2) are as follows:

C (carry)
E (entire, used to differentiate between regular interrupts that save all registers and fast interrupts that do not)
F (fast interrupt mask bit)
H (half-carry, i.e. carry from bit 3 of a byte)
I (regular interrupt mask bit)
N (negative or sign)
V (overflow)

6809 assembler

Delimiters include

space	After a label or operation code and before a comment on the same line as an instruction
, (comma)	Between operands in the address field and ahead of the designations for zero offset indexing, autoincrementing, and autodecrementing
[ ]	Around indirect addresses
*	Before an entire line of comments
:	Optional after a label except not allowed in EQU statements
/	Around strings in FCC pseudo-operations

Pseudo-operations include

END	End of program
EQU	Equate; define the attached label
FCB	Form constant byte; enter byte-length data
FCC	Form constant character string; enter character data
FDB	Form double byte constant; enter word-length data
ORG	Set (location counter to) origin; place subsequent object code starting at the specified address
RMB	Reserve memory bytes; allocate a specified number of bytes for data storage
SETDP	Specify memory page to be treated as the direct page in subsequent assembly

Designations include

Number systems

% (prefix) or B (suffix)	Binary
& (prefix) or D (suffix)	Decimal
$ (prefix) or H (suffix)	Hexadecimal
@ (prefix) or Q (suffix)	Octal

The default mode is decimal; hexadecimal numbers using the H suffix must start with a digit (i.e. you must add a leading zero if the number starts with a letter).

Others

'	ASCII character
–	Autodecrementing by 1 (before a register name)
– –	Autodecrementing by 2 (before a register name)
+	Autoincrementing by 1 (after a register name)

++	Autoincrementing by 2 (after a register name)
$	Current value of location (program) counter
<	Force the assembler to use direct (page) addressing
>	Force the assembler to use extended (direct) addressing
#	Immediate addressing (in front of an operand)
PCR	Relative to the current value of the location counter (as in DEST,PCR)

Defaults include:

Direct page is page 0 unless a SETDP pseudo-operation specifies otherwise.

Unmarked addresses are either direct (if they are on the page specified as the direct page) or extended (direct).

Unmarked numbers are decimal.

Introduction

Each description of an assembly language subroutine contains the following information:

- Purpose of the routine
- Procedure
- Entry conditions
- Exit conditions
- Examples
- Registers used
- Execution time
- Program size
- Data memory required
- Special cases

The program listing also includes much of this information as well as comments describing each section.

We have made each routine as general as possible. This is difficult for the input/output (I/O) and interrupt service routines described in Chapters 8 and 9 since in practice these routines are always computer-dependent. In such cases, we have limited the dependence to generalized input and output handlers and interrupt managers. We have

drawn specific examples from the popular Radio Shack TRS-80 Color Computer (with BASIC in ROM), but the general principles are applicable to other 6809-based computers as well.

All routines use the following parameter passing techniques:

1. A single 8-bit parameter is passed in accumulator A. A second 8-bit parameter is passed in accumulator B.

2. A single 16-bit parameter is passed in accumulators A and B (more significant byte in A) if it is data and in index register X if it is an address.

3. Larger number of parameters are passed in the hardware stack, either directly or indirectly. We assume that the subroutine entry is via a JSR instruction that places the return address at the top of the stack, and hence on top of the parameters.

Where there is a trade-off between execution time and memory usage, we have chosen the approach that minimizes execution time. We have also chosen the approach that minimizes the number of repetitive calculations. For example, consider the case of array indexing. The number of bytes between the starting addresses of elements differing only by 1 in a particular subscript (known as the *size* of that subscript) depends only on the number of bytes per element and the bounds of the array. This allows us to calculate the sizes of the various subscripts as soon as we know the bounds. We therefore use the sizes as parameters for the indexing routines, so that they need not be calculated each time a particular array is indexed.

We have specified the execution time for most short routines. For longer routines, we provide an approximate execution time. The execution time of programs with many branches will obviously depend on which path the computer follows in a particular case. A complicating factor is that a conditional branch requires different numbers of clock cycles depending on whether the processor actually branches. Thus, a precise execution time is often impossible to define. The documentation always contains at least one typical example showing an approximate or maximum execution time.

Our philosophy on error indicators and special cases has been the following:

1. Routines should provide an easily tested indicator (such as the Carry flag) of whether any errors or exceptions have occurred.

2. Trivial cases, such as no elements in an array or strings of zero length, should result in immediate exits with minimal effect on the underlying data.

3. Misspecified data (such as a maximum string length of zero or an index beyond the end of an array) should result in immediate exits with minimal effects on the underlying data.

4. The documentation should include a summary of errors and exceptions (under the heading of 'Special cases').

5. Exceptions that may actually be convenient for the user (such as deleting more characters than could possibly be left in a string rather than counting the precise number) should be handled in a reasonable way, but should still be indicated as errors.

Obviously, no method of handling errors or exceptions can ever be completely consistent or well-suited to all applications. Our approach is that a reasonable set of subroutines must deal with this issue, rather than ignoring it or assuming that the user will always provide data in the proper form.

1 Code conversion

1A Binary to BCD conversion (BN2BCD)

Converts one byte of binary data to two bytes of BCD data.

Procedure The program subtracts 100 repeatedly from the original data to determine the hundreds digit, then subtracts 10 repeatedly from the remainder to determine the tens digit, and finally shifts the tens digit left four positions and combines it with the ones digit.

Entry conditions

Binary data in A

Exit conditions

BCD data in D

Examples

1. Data: (A) = $6D_{16}$ (109 decimal)
   Result: (D) = 0109_{16}

2. Data: (A) = $B7_{16}$ (183 decimal)
   Result: (D) = 0183_{16}

Registers used A, B, CC

Execution time 140 cycles maximum, depends on the number of subtractions required to determine the tens and hundreds digits

Program size 30 bytes

Data memory required 2 stack bytes

```
*
*
*
*
*    Title:              Binary to BCD Conversion
*    Name:               BN2BCD
*
*
*
*    Purpose:            Converts one byte of binary data to two
*                        bytes of BCD data
*
*    Entry:              Register A = Binary data
*
*    Exit:               Register D = BCD data
*
*    Registers Used:     A,B,CC
*
*    Time:               140 cycles maximum
*
*    Size:               Program 30 bytes
*                        Data     2 bytes on stack
*
*

BN2BCD:
          *
          *CALCULATE 100'S DIGIT
          *DIVIDE DATA BY 100 USING SUBTRACTIONS
          * B = QUOTIENT
          * A = REMAINDER
          *
          LDB       #$FF          START QUOTIENT AT -1
D100LP:   INCB                    ADD 1 TO QUOTIENT
          SUBA      #100          SUBTRACT 100 FROM DIVIDEND
```

```
          BCC       D100LP          JUMP IF DIFFERENCE STILL POSITIVE
          ADDA      #100            IF NOT, ADD THE LAST 100 BACK
          STB       ,-S             SAVE 100'S DIGIT ON STACK
          *
          *CALCULATE 10'S AND 1'S DIGITS
          *DIVIDE THE REMAINDER FROM CALCULATING THE 100'S DIGIT BY 10
          * B = 10'S DIGIT
          * A = 1'S DIGIT
          *
          LDB       #$FF            START QUOTIENT AT -1
D10LP:    INCB                      ADD 1 TO QUOTIENT
          SUBA      #10             SUBTRACT 10 FROM DIVIDEND
          BCC       D10LP           JUMP IF DIFFERENCE STILL POSITIVE
          ADDA      #10             IF NOT, ADD THE LAST 10 BACK
          *
          *COMBINE 1'S AND 10'S DIGITS
          *
          LSLB                      MOVE 10'S DIGIT TO HIGH NIBBLE
          LSLB
          LSLB
          LSLB
          STA       ,-S             SAVE 1'S DIGIT ON STACK
          ADDB      ,S+             COMBINE 1'S AND 10'S DIGITS IN B
          *
          *RETURN WITH D = BCD DATA
          *
          LDA       ,S+             RETURN 100'S DIGIT IN A
          RTS

*
*
*         SAMPLE EXECUTION
*
*

SC1A:
          *CONVERT 0A HEXADECIMAL TO 10 BCD
          LDA       #$0A
          JSR       BN2BCD          D = 0010H (A = 00, B = 10H)

          *CONVERT FF HEXADECIMAL TO 255 BCD
          LDA       #$FF
          JSR       BN2BCD          D = 0255H (A = 02, B = 55H)

          *CONVERT 0 HEXADECIMAL TO 0 BCD
          LDA       #0
          JSR       BN2BCD          D = 0000 (A = 00, B = 00)

          END
```

1B BCD to binary conversion (BCD2BN)

Converts one byte of BCD data to one byte of binary data.

Procedure The program masks off the more significant digit and multiplies it by 10 using shifts. Note that 10 = 8 + 2, and multiplying by 8 or by 2 is equivalent to one or three right shifts, respectively, of the more significant digit. The program then adds the product to the less significant digit.

Entry conditions

BCD data in A

Exit conditions

Binary data in A

Examples

1. Data: (A) = 99_{16}
   Result: (A) = $63_{16} = 99_{10}$
2. Data: (A) = 23_{16}
   Result: (A) = $17_{16} = 23_{10}$

Registers used A, B, CC

Execution time 46 cycles

Program size 18 bytes

Data memory required 1 stack byte

```
*
*
*
*
*     Title:              BCD to Binary Conversion
*     Name:               BCD2BN
*
*
*
*     Purpose:            Converts one byte of BCD data to two
*                         bytes of binary data
*
*     Entry:              Register A = BCD data
*
*     Exit:               Register A = Binary data
*
*     Registers Used:     A,B,CC
*
*     Time:               46 cycles
*
*     Size:               Program 18 bytes
*                         Data     1 byte on stack
*
*

BCD2BN:
          *
          *SHIFT UPPER DIGIT RIGHT TO MULTIPLY IT BY 8
          *
          TFR       A,B         SAVE ORIGINAL BCD VALUE IN B
          ANDA      #$F0        MASK OFF UPPER DIGIT
          LSRA                  SHIFT RIGHT 1 BIT
          STA       ,-S         SAVE UPPER DIGIT TIMES 8 ON STACK
          *
          *ADD UPPER DIGIT TIMES 8 TO LOWER DIGIT
          *
          ANDB      #$0F        MASK OFF LOWER DIGIT
          ADDB      ,S+         ADD LOWER DIGIT TO STACK VALUE
          STB       ,-S         SAVE SUM ON STACK
          *
          *SHIFT UPPER DIGIT TIMES 8 RIGHT TWICE
          *THE RESULT IS UPPER DIGIT TIMES 2
          *
          LSRA                  MULTIPLY HIGH DIGIT BY 2
          LSRA
          *
          *UPPER DIGIT * 10 = UPPER DIGIT * 8 + UPPER DIGIT * 2
          *
          ADDA      ,S+         ADD STACK VALUE TO TWICE HIGH DIGIT
          RTS

*
*
*         SAMPLE EXECUTION
```

```
*
*

SC1B:
          *CONVERT 0 BCD TO 0 HEXADECIMAL
          LDA       #0
          JSR       BCD2BN          A = 00

          *CONVERT 99 BCD TO 63 HEXADECIMAL
          LDA       #$99
          JSR       BCD2BN          A = 63H

          *CONVERT 23 BCD TO 17 HEXADECIMAL
          LDA       #$23
          JSR       BCD2BN          A = 17H

          END
```

1C Binary to hexadecimal ASCII conversion (BN2HEX)

Converts one byte of binary data to two ASCII characters corresponding to the two hexadecimal digits.

Procedure The program masks off each hexadecimal digit separately and converts it to its ASCII equivalent. This involves a simple addition of 30_{16} if the digit is decimal. If the digit is non-decimal, we must add an extra 7 to bridge the gap between ASCII 9 (39_{16}) and ASCII A (41_{16}).

Entry conditions

Binary data in A

Exit conditions

ASCII version of more significant hexadecimal digit in A
ASCII version of less significant hexadecimal digit in B

Examples

1. Data: (A) = FB_{16}
   Result: (A) = 46_{16} (ASCII F)
   (B) = 42_{16} (ASCII B)
2. Data: (A) = 59_{16}
   Result: (A) = 35_{16} (ASCII 5)
   (B) = 39_{16} (ASCII 9)

Registers used A, B, CC

Execution time 37 cycles plus 2 extra cycles for each non-decimal digit

Program size 27 bytes

Data memory required None

```
*
*
*
*
*    Title:              Binary to Hex ASCII
*    Name:               BN2HEX
*
*
*
*    Purpose:            Converts one byte of binary data to two
*                        ASCII characters
*
*    Entry:              Register A = Binary data
*
*    Exit:               Register A = ASCII more significant digit
*                        Register B = ASCII less significant digit
*
*    Registers Used:     A,B,CC
*
*    Time:               Approximately 37 cycles
*
*    Size:               Program 27 bytes
*                        Data    None
*
*

BN2HEX:
          *
          *CONVERT MORE SIGNIFICANT DIGIT TO ASCII
          *
          TFR       A,B         SAVE ORIGINAL BINARY VALUE
          LSRA                  MOVE HIGH DIGIT TO LOW DIGIT
          LSRA
          LSRA
          LSRA
          CMPA      #9
          BLS       AD30        BRANCH IF HIGH DIGIT IS DECIMAL
          ADDA      #7          ELSE ADD 7 SO AFTER ADDING '0' THE
                                * CHARACTER WILL BE IN 'A'..'F'
AD30:     ADDA      #'0         ADD ASCII 0 TO MAKE A CHARACTER
          *
          *CONVERT LESS SIGNIFICANT DIGIT TO ASCII
          *
          ANDB      #$0F        MASK OFF LOW DIGIT
          CMPB      #9
          BLS       AD30LD      BRANCH IF LOW DIGIT IS DECIMAL
          ADDB      #7          ELSE ADD 7 SO AFTER ADDING '0' THE
                                * CHARACTER WILL BE IN 'A'..'F'
AD30LD:   ADDB      #'0         ADD ASCII 0 TO MAKE A CHARACTER
          RTS
```

```
*
*
*        SAMPLE EXECUTION
*
*

SC1C:
         *CONVERT 0 TO ASCII '00'
         LDA       #0
         JSR       BN2HEX          *A='0'=30H, B='0'=30H

         *CONVERT FF HEXADECIMAL TO ASCII 'FF'
         LDA       #$FF
         JSR       BN2HEX          *A='F'=46H, B='F'=46H

         *CONVERT 23 HEXADECIMAL TO ASCII '23'
         LDA       #$23
         JSR       BN2HEX          *A='2'=32H, B='3'=33H

         END
```

1D Hexadecimal ASCII to binary conversion (HEX2BN)

Converts two ASCII characters (representing two hexadecimal digits) to one byte of binary data.

Procedure The program converts each ASCII character separately to a hexadecimal digit. This involves a simple subtraction of 30_{16} (ASCII 0) if the digit is decimal. If the digit is non-decimal, the program must subtract another 7 to account for the gap between ASCII 9 (39_{16}) and ASCII A (41_{16}). The program then shifts the more significant digit left four bit positions and combines it with the less significant digit. The program does not check the validity of the ASCII characters (i.e. whether they are indeed the ASCII representations of hexadecimal digits).

Entry conditions

More significant ASCII digit in A, less significant ASCII digit in B

Exit conditions

Binary data in A

Examples

1. Data: (A) = 44_{16} (ASCII D)
   (B) = 37_{16} (ASCII 7)
   Result: (A) = $D7_{16}$
2. Data: (A) = 31_{16} (ASCII 1)
   (B) = 42_{16} (ASCII B)
   Result: (A) = $1B_{16}$

Registers used A, B, CC

Execution time 39 cycles plus 2 extra cycles for each non-decimal digit

Program size 25 bytes

Data memory required 1 stack byte

```
*
*
*
*
*    Title:              Hex ASCII to Binary
*    Name:               HEX2BN
*
*
*
*    Purpose:            Converts two ASCII characters to one
*                        byte of binary data
*
*    Entry:              Register A = ASCII more significant digit
*                        Register B = ASCII less significant digit
*
*    Exit:               Register A = Binary data
*
*    Registers Used:     A,B,CC
*
*    Time:               Approximately 39 cycles
*
*    Size:               Program 25 bytes
*                        Data     1 byte on stack
*
*

*
*         CONVERT MORE SIGNIFICANT DIGIT TO BINARY
*
HEX2BN:
          SUBA      #'0         SUBTRACT ASCII OFFSET (ASCII 0)
          CMPA      #9          CHECK IF DIGIT DECIMAL
          BLS       SHFTMS      BRANCH IF DECIMAL
          SUBA      #7          ELSE SUBTRACT OFFSET FOR LETTERS
SHFTMS:   LSLA                  SHIFT DIGIT TO MORE SIGNIFICANT BITS
          LSLA
          LSLA
          LSLA
*
*         CONVERT LESS SIGNIFICANT DIGIT TO BINARY
*
          SUBB      #'0         SUBTRACT ASCII OFFSET (ASCII 0)
          CMPB      #9          CHECK IF DIGIT DECIMAL
          BLS       CMBDIG      BRANCH IF DECIMAL
          SUBB      #7          ELSE SUBTRACT OFFSET FOR LETTERS
*
*         COMBINE LESS SIGNIFICANT, MORE SIGNIFICANT DIGITS
*
CMBDIG:
          STB       ,-S         SAVE LESS SIGNIFICANT DIGIT IN STACK
```

```
          ADDA      ,S+         ADD DIGITS
          RTS
*
*
*        SAMPLE EXECUTION
*
*

SC1D:
          *CONVERT ASCII 'C7' TO C7 HEXADECIMAL
          LDA       #'C
          LDB       #'7
          JSR       HEX2BN      A=C7H

          *CONVERT ASCII '2F' TO 2F HEXADECIMAL
          LDA       #'2
          LDB       #'F
          JSR       HEX2BN      A=2FH

          *CONVERT ASCII '2A' TO 2A HEXADECIMAL
          LDA       #'2
          LDB       #'A
          JSR       HEX2BN      A=2AH

          END
```

1E Conversion of a binary number to decimal ASCII (BN2DEC)

Converts a 16-bit signed binary number into an ASCII string. The string consists of the length of the number in bytes, an ASCII minus sign (if needed), and the ASCII digit. Note that the length is a binary number, not an ASCII number.

Procedure The program takes the absolute value of the number if it is negative. The program then keeps dividing the absolute value by 10 until the quotient becomes 0. It converts each digit of the quotient to ASCII by adding ASCII 0 and concatenates the digits along with an ASCII minus sign (in front) if the original number was negative.

Entry conditions

Base address of output buffer in X
Value to convert in D (between −32 767 and +32 767)

Exit conditions

Order in buffer:

Length of the string in bytes (a binary number)
ASCII − (if original number was negative)
ASCII digits (most significant digit first)

Examples

1. Data: Value to convert = $3EB7_{16}$
   Result (in output buffer):
   05 (number of bytes in buffer)
   31 (ASCII 1)
   36 (ASCII 6)
   30 (ASCII 0)
   35 (ASCII 5)
   35 (ASCII 5)
   i.e. $3EB7_{16} = 16055_{10}$
2. Data: Value to convert = $FFC8_{16}$

Result (in output buffer):
03 (number of bytes in buffer)
2D (ASCII −)
35 (ASCII 5)
36 (ASCII 6)
i.e. $FFC8_{16} = -56_{10}$, when considered as a signed two's complement number

Registers used All

Execution time Approximately 1000 cycles

Program size 99 bytes

Data memory required 1 stack byte for each digit in the string. This does not include the output buffer, which should be 7 bytes long.

```
*
*
*
*
*    Title:              Binary to Decimal ASCII
*    Name:               BN2DEC
*
*
*
*    Purpose:            Converts a 16-bit signed binary number
*                        to ASCII data
*
*    Entry:              Register D = Value to convert
*                        Register X = Output buffer address
*
*    Exit:               The first byte of the buffer is the
*                        length, followed by the characters
*
*    Registers Used:     CC, D, X, Y
*
*    Time:               Approximately 1000 cycles
*
*    Size:               Program  99 bytes
*                        Data     up to 5 bytes on stack
*
*
*
*        SAVE ORIGINAL DATA IN BUFFER
*        TAKE ABSOLUTE VALUE IF DATA NEGATIVE
*
```

```
BN2DEC:
          STD       1,X         SAVE DATA IN BUFFER
          BPL       CNVERT      BRANCH IF DATA POSITIVE
          LDD       #0          ELSE TAKE ABSOLUTE VALUE
          SUBD      1,X
*
*         INITIALIZE STRING LENGTH TO ZERO
*
CNVERT:
          CLR       ,X          STRING LENGTH = ZERO
*
*         DIVIDE BINARY DATA BY 10 BY SUBTRACTING POWERS
*          OF TEN
*
DIV10:
          LDY       #-1000      START QUOTIENT AT -1000
*
*         FIND NUMBER OF THOUSANDS IN QUOTIENT
*
THOUSD:
          LEAY      1000,Y      ADD 1000 TO QUOTIENT
          SUBD      #10000      SUBTRACT 10000 FROM DIVIDEND
          BCC       THOUSD      BRANCH IF DIFFERENCE STILL POSITIVE
          ADDD      #10000      ELSE ADD BACK LAST 10000
*
*         FIND NUMBER OF HUNDREDS IN QUOTIENT
*
          LEAY      -100,Y      START NUMBER OF HUNDREDS AT -1
HUNDD:
          LEAY      100,Y       ADD 100 TO QUOTIENT
          SUBD      #1000       SUBTRACT 1000 FROM DIVIDEND
          BCC       HUNDD       BRANCH IF DIFFERENCE STILL POSITIVE
          ADDD      #1000       ELSE ADD BACK LAST 1000
*
*         FIND NUMBER OF TENS IN QUOTIENT
*
          LEAY      -10,Y       START NUMBER OF TENS AT -1
TENSD:
          LEAY      10,Y        ADD 10 TO QUOTIENT
          SUBD      #100        SUBTRACT 100 FROM DIVIDEND
          BCC       TENSD       BRANCH IF DIFFERENCE STILL POSITIVE
          ADDD      #100        ELSE ADD BACK LAST 100
*
*         FIND NUMBER OF ONES IN QUOTIENT
*
          LEAY      -1,Y        START NUMBER OF ONES AT -1
ONESD:
          LEAY      1,Y         ADD 1 TO QUOTIENT
          SUBD      #10         SUBTRACT 10 FROM DIVIDEND
          BCC       ONESD       BRANCH IF DIFFERENCE STILL POSITIVE
          ADDD      #10         ELSE ADD BACK LAST 10
          STB       ,-S         SAVE REMAINDER IN STACK
                                *THIS IS NEXT DIGIT, MOVING LEFT
                                *LEAST SIGNIFICANT DIGIT GOES INTO STACK
                                *  FIRST
          INC       ,X          ADD 1 TO LENGTH BYTE
```

```
          TFR       Y,D         MAKE QUOTIENT INTO NEW DIVIDEND
          CMPD      #0          CHECK IF DIVIDEND ZERO
          BNE       DIV10       BRANCH IF NOT - DIVIDE BY 10 AGAIN
*
*         CHECK IF ORIGINAL BINARY DATA WAS NEGATIVE
*         IF SO, PUT ASCII - AT FRONT OF BUFFER
*
          LDA       ,X+         GET LENGTH BYTE (NOT INCLUDING SIGN)
          LDB       ,X          GET HIGH BYTE OF DATA
          BPL       BUFLOAD     BRANCH IF DATA POSITIVE
          LDB       #'-         OTHERWISE, GET ASCII MINUS SIGN
          STB       ,X+         STORE MINUS SIGN IN BUFFER
          INC       -2,X        ADD 1 TO LENGTH BYTE FOR SIGN
*
*         MOVE STRING OF DIGITS FROM STACK TO BUFFER
*         MOST SIGNIFICANT DIGIT IS AT TOP OF STACK
*         CONVERT DIGITS TO ASCII BY ADDING ASCII 0
*
BUFLOAD:
          LDB       ,S+         GET NEXT DIGIT FROM STACK, MOVING RIGHT
          ADDB      #'0         CONVERT DIGIT TO ASCII
          STB       ,X+         SAVE DIGIT IN BUFFER
          DECA                  DECREMENT BYTE COUNTER
          BNE       BUFLOAD     LOOP IF MORE BYTES LEFT
          RTS
*
*
*        SAMPLE EXECUTION
*
*

SC1E:
          *CONVERT 0 TO ASCII '0'
          LDD       #0              D=0
          LDX       #BUFFER         X=BASE ADDRESS OF BUFFER
          JSR       BN2DEC          CONVERT
                                    * BUFFER SHOULD CONTAIN
                                    *    BINARY 1 (LENGTH)
                                    *    ASCII 0 (STRING)
          *CONVERT 32767 TO ASCII '32767'
          LDD       #32767          D=32767
          LDX       #BUFFER         X=BASE ADDRESS OF BUFFER
          JSR       BN2DEC          CONVERT
                                    * BUFFER SHOULD CONTAIN
                                    *    BINARY 5 (LENGTH)
                                    *    ASCII 32767 (STRING)
          *CONVERT -32767 TO ASCII '-32767'
          LDD       #-32767         D=-32767
          LDX       #BUFFER         X=BASE ADDRESS OF BUFFER
          JSR       BN2DEC          CONVERT
                                    * BUFFER SHOULD CONTAIN
                                    *    BINARY 6 (LENGTH)
                                    *    ASCII - (SIGN)
                                    *    ASCII 32767 (STRING)
BUFFER:   RMB       7               7-BYTE BUFFER
          END
```

1F Conversion of ASCII decimal to binary (DEC2BN)

Converts an ASCII string consisting of the length of the number (in bytes), a possible ASCII + or − sign, and a series of ASCII digits to two bytes of binary data. Note that the length is an ordinary binary number, not an ASCII number.

Procedure The program checks if the first byte is a sign and skips over it if it is. The program then uses the length of the string to determine the leftmost digit position. Moving left to right, it converts each digit to decimal (by subtracting ASCII 0), validates it, multiplies it by the corresponding power of 10, and adds the product to the running total. Finally, the program subtracts the binary value from zero if the string started with a minus sign. The program exits immediately, setting the Carry flag, if it finds something other than a leading sign or a decimal digit in the string.

Entry conditions

Base address of string in X

Exit conditions

Binary value in D
The Carry flag is 0 if the string was valid; the Carry flag is 1 if the string contained an invalid character.
Note that the result is a signed two's complement 16-bit number.

Examples

1. Data: String consists of
   04 (number of bytes in string)
   31 (ASCII 1)
   32 (ASCII 2)
   33 (ASCII 3)
   34 (ASCII 4)
   i.e. the number is $+1234_{10}$
   Result: (D) = $04D2_{16}$ (binary data)

i.e. $+1234_{10} = 04D2_{16}$

2. Data: String consists of
   06 (number of bytes in string)
   2D (ASCII −)
   33 (ASCII 3)
   32 (ASCII 2)
   37 (ASCII 7)
   35 (ASCII 5)
   30 (ASCII 0)
   i.e. the number is $-32\,750_{10}$
   Result: (D) = 8016_{16} (binary data)
   i.e. $-32\,750_{10} = 8012_{16}$

Registers used A, B, CC, X, Y

Execution time Approximately 60 cycles per ASCII digit plus a maximum of 125 cycles overhead

Program size 154 bytes

Data memory required 2 stack bytes

Special cases

1. If the string contains something other than a leading sign or a decimal digit, the program returns with the Carry flag set to 1. The result in D is invalid.

2. If the string contains only a leading sign (ASCII + or ASCII −), the program returns with the Carry flag set to 1 and a result of 0.

```
*
*
*
*
*   Title:          Decimal ASCII to Binary
*   Name:           DEC2BN
*
*
*
```

```
*     Purpose:            Converts ASCII characters to two bytes
*                         of binary data
*
*     Entry:              Register X = Input buffer address
*
*     Exit:               Register D = Binary data
*                         If no errors then
*                           Carry = 0
*                         else
*                           Carry = 1
*
*     Registers Used:     All
*
*     Time:               Approximately 60 cycles per ASCII digit
*                         plus a maximum of 125 cycles overhead
*
*     Size:               Program 154 bytes
*                         Data      2 bytes on stack
*
*
*
*          SAVE BUFFER POINTER, INITIALIZE BINARY VALUE TO ZERO
*
DEC2BN:
          TFR       X,Y         SAVE BUFFER POINTER TO EXAMINE SIGN LATER
          LDD       #0          INITIALIZE BINARY VALUE TO ZERO
          PSHS      D           SAVE BINARY VALUE ON STACK
          LDA       ,X+         GET BYTE COUNT
*
*          CHECK IF FIRST BYTE OF ACTUAL STRING IS SIGN
*
          LDB       ,X+         GET FIRST BYTE OF ACTUAL STRING
          CMPB      #'-         CHECK IF IT IS ASCII -
          BEQ       STMSD       BRANCH IF IT IS
          CMPB      #'+         CHECK IF IT IS ASCII +
          BEQ       STMSD       BRANCH IF IT IS
*
*          FIRST BYTE IS NOT A SIGN
*          SET A FLAG, MOVE POINTER BACK TO START AT FIRST DIGIT
*          INCREASE BYTE COUNT BY 1 SINCE NO SIGN INCLUDED
*
          CLR       -2,X        INDICATE NO SIGN IN BUFFER
          LEAX      -1,X        MOVE POINTER BACK TO FIRST DIGIT
          INCA                  ADD 1 TO BYTE COUNT
*
*          START CONVERSION AT MOST SIGNIFICANT DIGIT IN BUFFER
*          COULD BE UP TO SIX BYTES INCLUDING SIGN
*
STMSD:
          CMPA      #6          LOOK FOR 10000'S DIGIT
          BEQ       TENKD       BRANCH IF FOUND
          CMPA      #5          LOOK FOR 1000'S DIGIT
          BEQ       ONEKD       BRANCH IF FOUND
          CMPA      #4          LOOK FOR 100'S DIGIT
          BEQ       HUNDD       BRANCH IF FOUND
          CMPA      #3          LOOK FOR TENS DIGIT
```

```
          BEQ       TENSD     BRANCH IF FOUND
          CMPA      #2        LOOK FOR ONES DIGIT
          BEQ       ONESD     BRANCH IF FOUND
          BRA       ERREXIT   NO DIGITS, INDICATE ERROR
*
*         CONVERT 10000'S DIGIT TO BINARY
*         10000 = 40*250
*         NOTE: MUL CANNOT MULTIPLY BY MORE THAN 255
*
TENKD:
          LDB       ,X+       GET 10000'S ASCII DIGIT
          JSR       CHVALD    CONVERT TO BINARY, CHECK VALIDITY
          CMPB      #3        CHECK IF DIGIT TOO LARGE
          BHI       ERREXIT   TAKE ERROR EXIT IF IT IS
          LDA       #40       MULTIPLY BY 10000 IN TWO STEPS
          MUL                 FIRST MULTIPLY BY 40
          LDA       #250      THEN MULTIPLY BY 250
          MUL
          ADDD      ,S        ADD PRODUCT TO BINARY VALUE
          STD       ,S        SAVE SUM ON STACK
*
*         CONVERT 1000'S DIGIT TO BINARY
*         1000 = 4*250
*         NOTE: MUL CANNOT MULTIPLY BY MORE THAN 255
*
ONEKD:
          LDB       ,X+       GET 1000'S ASCII DIGIT
          JSR       CHVALD    CONVERT TO BINARY, CHECK VALIDITY
          LDA       #4        MULTIPLY BY 1000 IN TWO STEPS
          MUL                 FIRST MULTIPLY BY 4
          LDA       #250      THEN MULTIPLY BY 250
          MUL
          ADDD      ,S        ADD PRODUCT TO BINARY VALUE
          STD       ,S        SAVE SUM ON STACK
*
*         CONVERT 100'S DIGIT TO BINARY
*
HUNDD:
          LDB       ,X+       GET 100'S ASCII DIGIT
          JSR       CHVALD    CONVERT TO BINARY, CHECK VALIDITY
          LDA       #100      MULTIPLY BY 100
          MUL
          ADDD      ,S        ADD PRODUCT TO BINARY VALUE
          STD       ,S        SAVE SUM ON STACK
*
*         CONVERT TENS DIGIT TO BINARY
*
TENSD:
          LDB       ,X+       GET 10'S ASCII DIGIT
          JSR       CHVALD    CONVERT TO BINARY, CHECK VALIDITY
          LDA       #10       MULTIPLY BY 10
          MUL
          ADDD      ,S        ADD PRODUCT TO BINARY VALUE
          STD       ,S        SAVE SUM ON STACK
*
*         CONVERT ONES DIGIT TO BINARY
```

```
*
ONESD:
          LDB       ,X+         GET ONES ASCII DIGIT
          JSR       CHVALD      CONVERT TO BINARY, CHECK VALIDITY
          CLRA                  EXTEND TO 16 BITS
          ADDD      ,S          ADD DIGIT TO BINARY VALUE
          STD       ,S          SAVE SUM ON STACK
*
*         CHECK FOR MINUS SIGN
*
          LDB       ,Y          CHECK IF THERE WAS A SIGN BYTE
          BEQ       VALEXIT     BRANCH IF NO SIGN
          LDB       1,Y         GET SIGN BYTE
          CMPB      #'-         CHECK IF IT IS ASCII -
          BNE       VALEXIT     BRANCH IF IT ISN'T
*
*         NEGATIVE NUMBER, SO SUBTRACT VALUE FROM ZERO
*
          LDD       #0          SUBTRACT VALUE FROM ZERO
          SUBD      ,S
          STD       ,S          SAVE NEGATIVE AS VALUE
*
*         EXIT WITH BINARY VALUE IN D
*
VALEXIT:
          PULS      D           RETURN TOTAL IN D
          CLC                   CLEAR CARRY, INDICATING NO ERRORS
          RTS
*
*         ERROR EXIT - SET CARRY FLAG TO RETURN ERROR CONDITION
*
ERREXIT:
          PULS      D           RETURN TOTAL IN D
          SEC                   SET CARRY TO INDICATE ERROR
          RTS

*************************************************************
*ROUTINE: CHVALD
*PURPOSE: CONVERTS ASCII TO DECIMAL, CHECKS VALIDITY OF DIGITS
*ENTRY: ASCII DIGIT IN B
*EXIT: DECIMAL DIGIT IN B, EXITS TO ERREXIT IF DIGIT INVALID
*REGISTERS USED: B,CC
*************************************************************
CHVALD:   SUBB      #'0         CONVERT TO DECIMAL BY SUBTRACTING ASCII 0
          BCS       EREXIT      BRANCH IF ERROR (VALUE TOO SMALL)
          CMPB      #9          CHECK IF RESULT IS DECIMAL DIGIT
          BHI       EREXIT      BRANCH IF ERROR (VALUE TOO LARGE)
          RTS                   RETURN DECIMAL DIGIT IN B
EREXIT:   LEAS      2,S         REMOVE RETURN ADDRESS FROM STACK
          BRA       ERREXIT     LEAVE VIA ERROR EXIT
*
*
*     SAMPLE EXECUTION
*
*
```

```
SC1F:
          *CONVERT ASCII '1234' TO 04D2 HEX
          LDX       #S1       X=BASE ADDRESS OF S1
          JSR       DEC2BN    D=04D2 HEX

          *CONVERT ASCII '+32767' TO 7FFF HEX
          LDX       #S2       X=BASE ADDRESS OF S2
          JSR       DEC2BN    D=7FFF HEX

          *CONVERT ASCII '-32768' TO 8000 HEX
          LDX       #S3       X=BASE ADDRESS OF S3
          JSR       DEC2BN    D=8000 HEX

S1:       FCB       4
          FCC       /1234/
S2:       FCB       6
          FCC       /+32767/
S3:       FCB       6
          FCC       /-32768/
          END
```

2 Array manipulation and indexing

2A Memory fill (MFILL)

Places a specified value in each byte of a memory area of known size, starting at a given address.

Procedure The program simply fills the memory area with the value one byte at a time.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Value to be placed in memory

More significant byte of area size (in bytes)
Less significant byte of area size (in bytes)

More significant byte of base address
Less significant byte of base address

Exit conditions

The area from the base address through the number of bytes given by the area size is filled with the specified value. The area thus filled starts at BASE and continues through BASE+SIZE−1 (BASE is the base address and SIZE is the area size in bytes).

Examples

1. Data: Value = FF_{16}
   Area size (in bytes) = 0380_{16}
   Base address = $1AE0_{16}$
   Result: FF_{16} placed in addresses $1AE0_{16} - 1E5F_{16}$
2. Data: Value = 12_{16} (6809 operation code for NOP)
   Area size (in bytes) = $1C65_{16}$
   Base address = $E34C_{16}$
   Result 12_{16} placed in addresses $E34C_{16} - FFB0_{16}$

Registers used A, CC, X, Y

Execution time 14 cycles per byte plus 38 cycles overhead

Program size 18 bytes

Data memory required None

Special cases

1. A size of 0000_{16} is interpreted as 10000_{16}. It therefore causes the program to fill 65 536 bytes with the specified value.

2. Filling areas occupied or used by the program itself will cause unpredictable results. Obviously, filling the stack area requires special caution, since the return address is saved there.

```
*
*
*
*
*    Title:              Memory Fill
*    Name:               MFILL
*
*
*
*    Purpose:            Fills an area of memory with a value
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Value to be placed in memory
*                          High byte of area size in bytes
*                          Low byte of area size in bytes
*                          High byte of base address
*                          Low byte of base address
*
*    Exit:               Area filled with value
*
*    Registers Used:     A,CC,U,X
*
*    Time:               14 cycles per byte plus 38 cycles overhead
*
*    Size:               Program 18 bytes
*
*
*
*
*         OBTAIN PARAMETERS FROM STACK
*
MFILL:
          PULS      Y          SAVE RETURN ADDRESS IN Y
          PULS      A          GET BYTE TO FILL WITH
          LDX       2,S        GET BASE ADDRESS
          STY       2,S        PUT RETURN ADDRESS BACK IN STACK
          PULS      Y          GET AREA SIZE
*
*         FILL MEMORY ONE BYTE AT A TIME
*
FILLB:
          STA       ,X+        FILL ONE BYTE WITH VALUE
          LEAY      -1,Y       DECREMENT BYTE COUNTER
          BNE       FILLB      CONTINUE UNTIL COUNTER = 0
          RTS
*
*
*         SAMPLE EXECUTION
*
*

SC2A:
          *
          *FILL BF1 THROUGH BF1+15 WITH 00
```

```
          *
          LDY       #BF1       BASE ADDRESS
          LDX       #SIZE1     NUMBER OF BYTES
          LDA       #0         VALUE TO FILL WITH
          PSHS      A,X,Y      PUSH PARAMETERS
          JSR       MFILL      FILL MEMORY
          *
          *FILL BF2 THROUGH BF2+1999 WITH 12 HEX (NOP'S OPCODE)
          *
          LDY       #BF2       BASE ADDRESS
          LDX       #SIZE2     NUMBER OF BYTES
          LDA       #$12       VALUE TO FILL WITH
          PSHS      A,X,Y      PUSH PARAMETERS
          JSR       MFILL      FILL MEMORY

SIZE1     EQU       16         SIZE OF BUFFER 1 (10 HEX)
SIZE2     EQU       2000       SIZE OF BUFFER 2 (07D0 HEX)
BF1:      RMB       SIZE1      BUFFER 1
BF2:      RMB       SIZE2      BUFFER 2
          END
```

2B Block move (BLKMOV)

Moves a block of data from a source area to a destination area.

Procedure The program determines if the base address of the destination area is within the source area. If it is, then working up from the base address would overwrite some source data. To avoid this, the program works down from the highest address (sometimes called a *move right*). Otherwise, the program simply moves the data starting from the lowest address (sometimes called a *move left*). An area size (number of bytes to move) of 0000_{16} causes an exit with no memory changed. The program provides automatic address wraparound mod 64K.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of number of bytes to move
Less significant byte of number of bytes to move

More significant byte of base address of destination area
Less significant byte of base address of destination area

More significant byte of base address of source area
Less significant byte of base address of source area

Exit conditions

The block of memory is moved from the source area to the destination area. If the number of bytes to be moved is NBYTES, the base address of the destination area is DEST, and the base address of the source area is SOURCE, then the data in addresses SOURCE through SOURCE + NBYTES − 1 is moved to addresses DEST through DEST + NBYTES − 1.

Examples

1. Data: Number of bytes to move = 0200_{16}
   Base address of destination area = $05D1_{16}$
   Base address of source area = $035E_{16}$
   Result: The contents of locations $035E_{16}$–$055D_{16}$ are moved to $05D1_{16} - 07D0_{16}$.

2. Data: Number of bytes to move = $1B7A_{16}$
   Base address of destination area = $C946_{16}$
   Base address of source area = $C300_{16}$
   Result: The contents of locations $C300_{16} - DE79_{16}$ are moved to $C946_{16}$–$E4BF_{16}$.

Note that Example 2 presents a more difficult problem than Example 1 because the source and destination areas overlap. If, for instance, the program simply moved data to the destination area starting from the lowest address, it would initially move the contents of $C300_{16}$ to $C946_{16}$. This would destroy the old contents of $C946_{16}$, which are needed later in the move. The solution to this problem is to move the data starting from the highest address if the destination area is above the source area but overlaps it.

Registers used All

Execution time 20 cycles per byte plus 59 cycles overhead if data can be moved starting from the lowest address (i.e. left); 95 cycles overhead if data must be moved starting from the highest address (i.e. right) because of overlap.

Program size 55 bytes

Data memory required None

Special cases

1. A size (number of bytes to move) of 0 causes an immediate exit with no memory changed.

2. Moving data to areas occupied or used by the program itself or by the stack will have unpredictable results.

```
*
*
*
*
*    Title:              Block Move
*    Name:               BLKMOV
*
*
*
*    Purpose:            Move data from source to destination
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          High byte of number of bytes to move
*                          Low byte of number of bytes to move
*                          High byte of base address of
*                            destination area
*                          Low byte of base address of
*                            destination area
*                          High byte of base address of source
*                            area
*                          Low byte of base address of source
*                            area
*
*    Exit:               Data moved from source to destination
*
*    Registers Used:     All
*
*    Time:               20 cycles per byte
*                        Overhead is: 59 cycles if no problem with
*                          overlap, 95 cycles if overlap
*
*    Size:               Program 55 bytes
*
*
*
*
*         EXIT IMMEDIATELY IF AREA SIZE IS 0
*
BLKMOV:
          LDD       2,S       GET AREA SIZE
          BEQ       BLKEXIT   RETURN IMMEDIATELY IF SIZE IS ZERO
*
*         DETERMINE IF DESTINATION AREA IS ABOVE SOURCE AREA AND
*         OVERLAPS IT (OVERLAP CAN BE MOD 64K).  OVERLAP OCCURS
*         IF BASE ADDRESS OF DESTINATION AREA MINUS BASE ADDRESS
*         OF SOURCE AREA (MOD 64K) IS LESS THAN NUMBER OF BYTES
*         TO MOVE
*
          LDD       4,S       GET BASE ADDRESS OF DESTINATION
          SUBD      6,S       SUBTRACT BASE ADDRESS OF SOURCE
```

```
            CMPD        2,S         COMPARE DIFFERENCE TO AREA SIZE
            BLO         MVRIGHT     BRANCH IF OVERLAP PROBLEM
*
*           NO OVERLAP SO MOVE BLOCK STARTING FROM LOWEST ADDRESS
*
MVLEFT:
            PULS        D,X,Y       GET RETURN ADDRESS, SIZE, DESTINATION
            LDU         ,S          GET SOURCE ADDRESS
            STD         ,S          PUT RETURN ADDRESS BACK IN STACK
BYTEL:      LDA         ,U+         GET NEXT BYTE FROM SOURCE
            STA         ,Y+         MOVE IT TO DESTINATION
            LEAX        -1,X        DECREMENT BYTE COUNTER
            BNE         BYTEL       CONTINUE UNTIL COUNTER = 0
            RTS
*
*           OVERLAP SO MOVE BLOCK STARTING FROM HIGHEST ADDRESS
*           TO AVOID DESTROYING DATA
*
MVRIGHT:
            LDD         4,S         GET BASE ADDRESS OF DESTINATION
            ADDD        2,S         ADD LENGTH TO OBTAIN TOP ADDRESS
            TFR         D,Y         SAVE TOP ADDRESS OF DESTINATION
            LDD         6,S         GET BASE ADDRESS OF SOURCE
            ADDD        2,S         ADD LENGTH TO OBTAIN TOP ADDRESS
            TFR         D,U         SAVE TOP ADDRESS OF SOURCE
            PULS        D,X         GET RETURN ADDRESS, SIZE
            LEAS        2,S         ADJUST STACK POINTER TO REMOVE EXTRA BYTES
            STD         ,S          PUT RETURN ADDRESS BACK IN STACK
BYTER:
            LDA         ,-U         GET NEXT BYTE FROM SOURCE
            STA         ,-Y         MOVE IT TO DESTINATION
            LEAX        -1,X        DECREMENT BYTE COUNTER
            BNE         BYTEL       CONTINUE UNTIL COUNTER = 0
BLKEXIT:
            RTS

*
*
*           SAMPLE EXECUTION
*
*

SRC1        EQU         $1000       BASE ADDRESS OF FIRST SOURCE AREA
SRC2        EQU         $2008       BASE ADDRESS OF SECOND SOURCE AREA
DEST        EQU         $2010       BASE ADDRESS OF DESTINATION AREA
LEN         EQU         $11         NUMBER OF BYTES TO MOVE

SC2B:
*
*           MOVE 11 HEX BYTES FROM 1000-1010 HEX TO 2010-2020 HEX
*           DEMONSTRATES MOVE LEFT (LOWEST ADDRESS UP)
*
            LDU         #SRC1       BASE ADDRESS OF SOURCE AREA
            LDY         #DEST       BASE ADDRESS OF DESTINATION AREA
            LDX         #LEN        NUMBER OF BYTES TO MOVE
            PSHS        U,X,Y       SAVE PARAMETERS IN STACK
```

```
        JSR     BLKMOV    MOVE DATA FROM SOURCE TO DESTINATION
*
*       MOVE 11 HEX BYTES FROM 2008-2018 HEX TO 2010-2020 HEX
*       DEMONSTRATES MOVE RIGHT (HIGHEST ADDRESS DOWN) SINCE
*       SOURCE AND DESTINATION AREAS OVERLAP AND DESTINATION
*       IS ABOVE SOURCE
*
        LDU     #SRC2     BASE ADDRESS OF SOURCE AREA
        LDY     #DEST     BASE ADDRESS OF DESTINATION AREA
        LDX     #LEN      NUMBER OF BYTES TO MOVE
        PSHS    U,X,Y     SAVE PARAMETERS IN STACK
        JSR     BLKMOV    MOVE DATA FROM SOURCE TO DESTINATION
        END
```

2C Two-dimensional byte array indexing (D2BYTE)

Calculates the address of an element of a two-dimensional byte-length array, given the array's base address, the element's two subscripts, and the size of a row (i.e. the number of columns). The array is assumed to be stored in row major order (i.e. by rows), and both subscripts are assumed to begin at 0.

Procedure The program multiplies the row size (number of columns in a row) times the row subscript (since the elements are stored by rows) and adds the product to the column subscript. It then adds the sum to the base address.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of column subscript
Less significant byte of column subscript

More significant byte of the size of a row (in bytes)
Less significant byte of the size of a row (in bytes)

More significant byte of row subscript
Less significant byte of row subscript

More significant byte of base address of array
Less significant byte of base address of array

Exit conditions

Address of element in X

Examples

1. Data: Base address = $3C00_{16}$
   Column subscript = 0004_{16}

Size of row (number of columns) = 0018_{16}
Row subscript = 0003_{16}
Result: Element address = $3C00_{16} + 0003_{16} \times 0018_{16} + 0004_{16} = 3C00_{16} + 0048_{16} + 0004_{16} = 3C4C_{16}$
i.e. the address of ARRAY(3,4) is $3C4C_{16}$.

2. Data: Base address = $6A4A_{16}$
Column subscript = 0037_{16}
Size of row (number of columns) = 0050_{16}
Row subscript = 0002_{16}
Result Element address = $6A4A_{16} + 0002_{16} \times 0050_{16} + 0037_{16} = 6A4A_{16} + 00A0_{16} + 0037_{16} = 6B21_{16}$
i.e. the address of ARRAY(2,35) is $6B21_{16}$.

Note that all subscripts are hexadecimal (e.g. $37_{16} = 55_{10}$).
The general formula is

ELEMENT ADDRESS = ARRAY BASE ADDRESS + ROW SUBSCRIPT×ROW SIZE + COLUMN SUBSCRIPT

Note that we refer to the *size* of the row subscript; this is the number of consecutive memory addresses for which the subscript has the same value. It is also the distance in bytes from the address of an element to the address of the element with the same column subscript but a row subscript 1 larger.

Registers used CC, D, X, Y

Execution time Approximately 785 cycles

Program size 36 bytes

Data memory required None

```
*
*
*
*    Title:          Two-Dimensional Byte Array Indexing
*    Name:           D2BYTE
*
*
*
```

```
*    Purpose:            Given the base address of a byte array,
*                        two subscripts 'I' and 'J', and the size
*                        of the first subscript in bytes, calculate
*                        the address of A[I,J].  The array is assumed
*                        to be stored in row major order (A[0,0],
*                        A[0,1],...,A[K,L]), and both dimensions
*                        are assumed to begin at zero as in the
*                        following Pascal declaration:
*                          A:ARRAY[0..2,0..7] OF BYTE;
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          High byte of second subscript (column element)
*                          Low byte of second subscript (column element)
*                          High byte of first subscript size, in bytes
*                          Low byte of first subscript size, in bytes
*                          High byte of first subscript (row element)
*                          Low byte of first subscript (row element)
*                          High byte of array base address
*                          Low byte of array base address
*                        NOTE:
*                          The first subscript size is the length of
*                          a row in bytes.
*
*    Exit:               Register X = Element address
*
*    Registers Used:     CC,D,X,Y
*
*    Time:               Approximately 785 cycles
*
*    Size:               Program 36 bytes
*
*
*
*
D2BYTE:
          *
          *ELEMENT ADDRESS = ROW SIZE*ROW SUBSCRIPT + COLUMN
          *  SUBSCRIPT + BASE ADDRESS
          *
          LDD       #0          START ELEMENT ADDRESS AT 0
          LDY       #16         SHIFT COUNTER = 16
          *
          *MULTIPLY ROW SUBSCRIPT * ROW SIZE USING SHIFT AND
          *  ADD ALGORITHM
          *
MUL16:
          LSR       4,S         SHIFT HIGH BYTE OF ROW SIZE
          ROR       5,S         SHIFT LOW BYTE OF ROW SIZE
          BCC       LEFTSH      JUMP IF NEXT BIT OF ROW SIZE IS 0
          ADDD      6,S         OTHERWISE, ADD SHIFTED ROW SUBSCRIPT
                                * TO ELEMENT ADDRESS
LEFTSH:
          LSL       7,S         SHIFT LOW BYTE OF ROW SUBSCRIPT
          ROL       6,S         SHIFT HIGH BYTE PLUS CARRY
```

```
          LEAY      -1,Y       DECREMENT SHIFT COUNTER
          BNE       MUL16      LOOP 16 TIMES
          *
          *ADD COLUMN SUBSCRIPT TO ROW SUBSCRIPT * ROW SIZE
          *
          ADDD      2,S        ADD COLUMN SUBSCRIPT
          ADDD      8,S        ADD BASE ADDRESS OF ARRAY
          TFR       D,X        EXIT WITH ELEMENT ADDRESS IN X
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
          PULS      D          GET RETURN ADDRESS
          LEAS      6,S        REMOVE PARAMETERS FROM STACK
          STD       ,S         PUT RETURN ADDRESS BACK IN STACK
          RTS

*
*
*         SAMPLE EXECUTION
*
*

SC2C:
          LDU       #ARY       BASE ADDRESS OF ARRAY
          LDY       SUBS1      FIRST SUBSCRIPT
          LDX       SSUBS1     SIZE OF FIRST SUBSCRIPT
          LDD       SUBS2      SECOND SUBSCRIPT
          PSHS      U,X,Y,D    PUSH PARAMETERS
          JSR       D2BYTE     CALCULATE ADDRESS
                               *FOR THE INITIAL TEST DATA
                               *X = ADDRESS OF ARY(2,4)
                               *  = ARY + (2*8) + 4
                               *  = ARY + 20 (CONTENTS ARE 21)
                               *NOTE BOTH SUBSCRIPTS START AT 0
*
*DATA
*
SUBS1:    FDB        2         SUBSCRIPT 1
SSUBS1:   FDB        8         SIZE OF SUBSCRIPT 1 (NUMBER OF BYTES
                               * PER ROW)
SUBS2:    FDB        4         SUBSCRIPT 2
*THE ARRAY (3 ROWS OF 8 COLUMNS)
ARY:      FCB        1,2,3,4,5,6,7,8
          FCB        9,10,11,12,13,14,15,16
          FCB        17,18,19,20,21,22,23,24

          END
```

2D Two-dimensional word array indexing (D2WORD)

Calculates the address of an element of a two-dimensional word-length (16-bit) array, given the array's base address, the element's two subscripts, and the size of a row (i.e. the number of columns). The array is assumed to be stored in row major order (i.e. by rows), and both subscripts are assumed to begin at 0.

Procedure The program multiplies the row size (number of bytes in a row) times the row subscript (since the elements are stored by rows), adds the product to the doubled column subscript (doubled because each element occupies 2 bytes), and adds the sum to the base address.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of column subscript
Less significant byte of column subscript

More significant byte of the size of a row (in bytes)
Less significant byte of the size of a row (in bytes)

More significant byte of row subscript
Less significant byte of row subscript

More significant byte of base address of array
Less significant byte of base address of array

Exit conditions

Base address of element in X
The element occupies the address in X and the next higher address

Examples

1. Data: Base address = $5E14_{16}$
   Column subscript = 0008_{16}

Size of row (in bytes) = $001C_{16}$ (i.e. each row has 0014_{10} or $000E_{16}$ word-length elements)
Row subscript = 0005_{16}

Result: Element base address = $5E14_{16} + 0005_{16} \times 001C_{16} + 0008_{16} \times 2 = 5E14_{16} + 008C_{16} + 0010_{16} = 5EB0_{16}$
i.e. the base address of ARRAY(5,8) is $5EB0_{16}$ and the element occupies addresses $5EB0_{16}$ and $5EB1_{16}$.

2. Data: Base address = $B100_{16}$
Column subscript = 0002_{16}
Size of row (in bytes) = 0008_{16} (i.e. each row has four word-length elements)
Row subscript = 0006_{16}

Result: Element's base address = $B100_{16} + 0006_{16} \times 0008_{16} + 0002_{16} \times 2 = B100_{16} + 0030_{16} + 0004_{16} = B134_{16}$
i.e. the base address of ARRAY(6,2) is $B134_{16}$ and the element occupies addresses $B134_{16}$ and $B135_{16}$.

The general formula is

ELEMENT'S BASE ADDRESS = ARRAY BASE ADDRESS + ROW SUBSCRIPT × ROW SIZE + COLUMN SUBSCRIPT × 2

Note that one parameter of this routine is the size of a row in bytes. The size for word-length elements is the number of columns per row times 2 (the size of an element in bytes). The reason for choosing this parameter rather than the number of columns or the maximum column index is that it can be calculated once (when the array bounds are determined) and used whenever the array is accessed. The alternative parameters (number of columns or maximum column index) would require extra calculations during each indexing operation.

Registers used CC, D, X, Y

Execution time Approximately 790 cycles

Program size 38 bytes

Data memory required None

```
*
*
*
*    Title:              Two-Dimensional Word Array Indexing
*    Name:               D2WORD
*
*
*
*    Purpose:            Given the base address of a word array,
*                        two subscripts 'I' and 'J', and the size
*                        of the first subscript in bytes, calculate
*                        the address of A[I,J].  The array is assumed
*                        to be stored in row major order (A[0,0],
*                        A[0,1],...,A[K,L]), and both dimensions
*                        are assumed to begin at zero as in the
*                        following Pascal declaration:
*                          A:ARRAY[0..2,0..7] OF WORD;
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          High byte of second subscript (column element)
*                          Low byte of second subscript (column element)
*                          High byte of first subscript size, in bytes
*                          Low byte of first subscript size, in bytes
*                          High byte of first subscript (row element)
*                          Low byte of first subscript (row element)
*                          High byte of array base address
*                          Low byte of array base address
*                        NOTE:
*                          The first subscript size is the length of
*                          a row in words * 2.
*
*    Exit:               Register X = Element's base address
*
*    Registers Used:     CC,D,X,Y
*
*    Time:               Approximately 790 cycles
*
*    Size:               Program 38 bytes
*
*

D2WORD:
          *
          *ELEMENT ADDRESS = ROW SIZE*ROW SUBSCRIPT + 2*COLUMN
          *  SUBSCRIPT + BASE ADDRESS
          *
          LDD       #0          START ELEMENT ADDRESS AT 0
          LDY       #16         SHIFT COUNTER = 16
          *
          *MULTIPLY ROW SUBSCRIPT * ROW SIZE USING SHIFT AND
          *  ADD ALGORITHM
          *
```

```
MUL16:
          LSR       4,S         SHIFT HIGH BYTE OF ROW SIZE
          ROR       5,S         SHIFT LOW BYTE OF ROW SIZE
          BCC       LEFTSH      JUMP IF NEXT BIT OF ROW SIZE IS 0
          ADDD      6,S         OTHERWISE, ADD SHIFTED ROW SUBSCRIPT
                                * TO ELEMENT ADDRESS
LEFTSH:
          LSL       7,S         SHIFT LOW BYTE OF ROW SUBSCRIPT
          ROL       6,S         SHIFT HIGH BYTE PLUS CARRY
          LEAY      -1,Y        DECREMENT SHIFT COUNTER
          BNE       MUL16       LOOP 16 TIMES
          *
          *ADD COLUMN SUBSCRIPT TWICE TO ROW SUBSCRIPT * ROW SIZE
          *
          ADDD      2,S         ADD COLUMN SUBSCRIPT
          ADDD      2,S         ADD COLUMN SUBSCRIPT AGAIN
          ADDD      8,S         ADD BASE ADDRESS OF ARRAY
          TFR       D,X         EXIT WITH ELEMENT ADDRESS IN X
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
          PULS      D           GET RETURN ADDRESS
          LEAS      6,S         REMOVE PARAMETERS FROM STACK
          STD       ,S          PUT RETURN ADDRESS BACK ON STACK
          RTS

*
*
*         SAMPLE EXECUTION
*
*
SC2D:
          LDU       #ARY        BASE ADDRESS OF ARRAY
          LDY       SUBS1       FIRST SUBSCRIPT
          LDX       SSUBS1      SIZE OF FIRST SUBSCRIPT
          LDD       SUBS2       SECOND SUBSCRIPT
          PSHS      U,X,Y,D     PUSH PARAMETERS
          JSR       D2WORD      CALCULATE ADDRESS
                                *FOR THE INITIAL TEST DATA
                                *X = ADDRESS OF ARY(2,4)
                                *  = ARY + (2*16) + 4 * 2
                                *  = ARY + 40 (CONTENTS ARE 2100H)
                                *NOTE BOTH SUBSCRIPTS START AT 0
*
*DATA
*
SUBS1:    FDB       2           SUBSCRIPT 1
SSUBS1:   FDB       16          SIZE OF SUBSCRIPT 1 (NUMBER OF BYTES
                                * PER ROW)
SUBS2:    FDB       4           SUBSCRIPT 2

*THE ARRAY (3 ROWS OF 8 COLUMNS)
ARY:      FDB       0100H,0200H,0300H,0400H,0500H,0600H,0700H,0800H
          FDB       0900H,1000H,1100H,1200H,1300H,1400H,1500H,1600H
          FDB       1700H,1800H,1900H,2000H,2100H,2200H,2300H,2400H
          END
```

2E N-dimensional array indexing (NDIM)

Calculates the base address of an element of an *N*-dimensional array given the array's base address and *N* pairs of sizes and subscripts. The size of a dimension is the number of bytes from the base address of an element to the base address of the element with an index 1 larger in the dimension but the same in all other dimensions. The array is assumed to be stored in row major order (i.e. by rows), and both subscripts are assumed to begin at 0.

Note that the size of the rightmost subscript is simply the size of an element in bytes; the size of the next subscript is the size of an element times the maximum value of the rightmost subscript plus 1, and so on. All subscripts are assumed to begin at 0. Otherwise, the user must normalize them (see the second example at the end of the listing).

Procedure The program loops on each dimension, calculating the offset in it as the subscript times the size. After calculating the overall offset, the program adds it to the array's base address to obtain the element's base address.

Entry Conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of number of dimensions
Less significant byte of number of dimensions

More significant byte of size of rightmost dimension
Less significant byte of size of rightmost dimension

More significant byte of rightmost subscript
Less significant byte of rightmost subscript

.
.
.

More significant byte of size of leftmost dimension
Less significant byte of size of leftmost dimension

More significant byte of leftmost subscript

Less significant byte of leftmost subscript

More significant byte of base address of array
Less significant byte of base address of array

Exit conditions

Base address of element in X
The element occupies memory addresses START through START + SIZE − 1, where START is the calculated address and SIZE is the size of an element in bytes.

Example

Data: Base address = $3C00_{16}$
Number of dimensions = 0003_{16}
Rightmost subscript = 0005_{16}
Rightmost size = 0003_{16} (3-byte entries)
Middle subscript = 0003_{16}
Middle size = 0012_{16} (six 3-byte entries)
Leftmost subscript = 0004_{16}
Leftmost size = $007E_{16}$ (seven sets of six 3-byte entries)

Result: Element base address = $3C00_{16} + 0005_{16} \times 0003_{16} + 0003_{16} \times 0012_{16} + 0004_{16} \times 007E_{16} = 3C00_{16} + 000F_{16} + 0036_{16} + 01F8_{16} = 3ECD_{16}$,
i.e. the element is ARRAY(4,3,5); it occupies addresses $3E3D_{16}$ – $3E3F_{16}$. (The maximum values of the various subscripts are 6 (leftmost) and 5 (middle), with each element occupying 3 bytes.)

The general formula is

$$\text{ELEMENT BASE ADDRESS} = \text{ARRAY BASE ADDRESS} + \sum_{i=0}^{N-1} \text{SUBSCRIPT}_i \times \text{SIZE}_i$$

where:

N is the number of dimensions
SUBSCRIPT_i is the ith subscript
SIZE_i is the size of the ith dimension

Note that we use the size of each dimension as a parameter to reduce the number of repetitive multiplications and to generalize the procedure.

The sizes can be calculated and saved as soon as the bounds of the array are known. Those sizes can then be used whenever indexing is performed on the array. Obviously, the sizes do not change if the bounds are fixed, and they should not be recalculated as part of each indexing operation. The sizes are also general, since the elements can themselves consist of any number of bytes.

Registers used All

Execution time Approximately 720 cycles per dimension plus 67 cycles overhead

Program size 49 bytes

Data memory required None

Special case If the number of dimensions is 0, the program returns with the base address in X.

```
*
*
*
*
*    Title:          N-Dimensional Array Indexing
*    Name:           NDIM
*
*
*
*    Purpose:        Calculate the address of an element in an
*                    N-dimensional array given the base address,
*                    N pairs of size in bytes and subscripts, and
*                    the number of dimensions of the array.  The
*                    array is assumed to be stored in row major
*                    order (e.g., A[0,0,0],A[0,0,1],...,A[0,1,0],
*                    A[0,1,1],...).  Also, it is assumed that all
*                    dimensions begin at 0 as in the following
*                    Pascal declaration:
*                      A:ARRAY[0..10,0..3,0..5] OF SOMETHING
*
*    Entry:          TOP OF STACK
*                      High byte of return address
*                      Low byte of return address
*                      High byte of number of dimensions
```

```
*                          Low byte of number of dimensions
*                          High byte of dim N-1 size
*                          Low byte of dim N-1 size
*                          High byte of dim N-1 subscript
*                          Low byte of dim N-1 subscript
*                          High byte of dim N-2 size
*                          Low byte of dim N-2 size
*                          High byte of dim N-2 subscript
*                          Low byte of dim N-2 subscript
*                             .
*                             .
*                             .
*                          High byte of array base address
*                          Low byte of array base address
*                        NOTE:
*                          All sizes are in bytes.
*
*    Exit:               Register X = Element's base address
*
*    Registers Used:     All
*
*    Time:               Approximately 720 cycles per dimension plus
*                        67 cycles overhead
*
*    Size:               Program  49 bytes
*
*
*

*
*         EXIT IMMEDIATELY IF NUMBER OF DIMENSIONS IS ZERO
*
NDIM:
          PULS      U           SAVE RETURN ADDRESS
          LDX       2,S         GET BASE ADDRESS IF ZERO DIMENSIONS
          LDY       ,S++        GET NUMBER OF DIMENSIONS
          BEQ       EXITNDIM    BRANCH IF NUMBER OF DIMENSIONS IS ZERO
          *
          *ELEMENT ADDRESS = BASE ADDRESS + SIZE(I)*SUBSCRIPT(I) FOR
          *  I = 0 TO N-1
          *
          LDD       #0          START ELEMENT ADDRESS AT ZERO
          *
          *MULTIPLY ROW SUBSCRIPT * ROW SIZE USING SHIFT AND
          *  ADD ALGORITHM
          *
NEXTDIM:
          LDX       #16         SHIFT COUNTER = 16
MUL16:
          LSR       ,S          SHIFT HIGH BYTE OF ROW SIZE
          ROR       1,S         SHIFT LOW BYTE OF ROW SIZE
          BCC       LEFTSH      JUMP IF NEXT BIT OF ROW SIZE IS 0
          ADDD      2,S         OTHERWISE, ADD SHIFTED ROW SUBSCRIPT
                                * TO ELEMENT ADDRESS
LEFTSH:
          LSL       3,S         SHIFT LOW BYTE OF ROW SUBSCRIPT
```

```
          ROL       2,S         SHIFT HIGH BYTE PLUS CARRY
          LEAX      -1,X        DECREMENT SHIFT COUNTER
          BNE       MUL16       LOOP 16 TIMES
          *
          *MOVE STACK POINTER PAST FINISHED DIMENSION
          *
          LEAS      4,S         REMOVE SIZE, SUBSCRIPT FROM STACK
          *
          *CONTINUE IF MORE DIMENSIONS LEFT
          *
          LEAY      -1,Y        DECREMENT NUMBER OF DIMENSIONS
          BNE       NEXTDIM     BRANCH IF ANY DIMENSIONS LEFT
          *
          *ADD TOTAL OFFSET TO BASE ADDRESS OF ARRAY
          *
          ADDD      ,S          ADD BASE ADDRESS OF ARRAY
          TFR       D,X         MOVE ELEMENT ADDRESS TO X
EXITNDIM:
          STU       ,S          PUT RETURN ADDRESS BACK IN STACK
          RTS


*
*         SAMPLE EXECUTION
*
*

SC2E:
          *
          *CALCULATE ADDRESS OF AY1[1,3,0]
          *SINCE LOWER BOUNDS OF ARRAY 1 ARE ALL ZERO, IT IS
          *  NOT NECESSARY TO NORMALIZE THEM
          *
          LDU       #AY1        BASE ADDRESS OF ARRAY
          LDY       #1          FIRST SUBSCRIPT
          LDX       A1SZ1       SIZE OF FIRST SUBSCRIPT
          LDD       #3          SECOND SUBSCRIPT
          PSHS      U,X,Y,D     PUSH PARAMETERS
          LDU       #A1SZ2      SIZE OF SECOND SUBSCRIPT
          LDY       #0          THIRD SUBSCRIPT
          LDX       #A1SZ3      SIZE OF THIRD SUBSCRIPT
          LDD       #A1DIM      NUMBER OF DIMENSIONS
          PSHS      U,X,Y,D     PUSH PARAMETERS
          JSR       NDIM        CALCULATE ADDRESS
                                *AY = STARTING ADDRESS OF ARY1(1,3,0)
                                *   = ARY + (1*126) + (3*21) + (0*3)
                                *   = ARY+189
          *
          *CALCULATE ADDRESS OF AY2[-1,6]
          * SINCE LOWER BOUNDS OF ARRAY 2 DO NOT START AT 0, SUBSCRIPTS
          * MUST BE NORMALIZED
          *
          LDX       #AY2        BASE ADDRESS OF ARRAY
          LDD       #-1         GET UNNORMALIZED FIRST SUBSCRIPT
          SUBD      #A2D1L      NORMALIZE FIRST SUBSCRIPT (SUBTRACT
                                * LOWER BOUND
```

```
          PSHS      D,X         PUSH PARAMETERS
          LDX       #A2SZ1      SIZE OF FIRST SUBSCRIPT
          LDD       #6          GET UNNORMALIZED SECOND SUBSCRIPT
          SUBD      #A2D2L      NORMALIZE SECOND SUBSCRIPT (SUBTRACT
                                * LOWER BOUND
          PSHS      D,X         PUSH PARAMETERS
          LDX       #A2SZ2      SIZE OF SECOND SUBSCRIPT
          LDD       #A2DIM      NUMBER OF DIMENSIONS
          PSHS      D,X         PUSH PARAMETERS
          JSR       NDIM        CALCULATE ADDRESS
                                *AY = STARTING ADDRESS OF AY2(-1,6)
                                *   = AY2+(((-1)-(-5))*18)+((6-2)*2)
                                *   = AY2+80

*DATA
*AY1 : ARRAY[A1D1L..A1D1H,A1D2L..A1D2H,A1D3L..A1D3H] 3-BYTE ELEMENTS
*          [ 0  . .  3  ,  0  .  . 5  ,  0  . . 6  ]
A1DIM     EQU       3           NUMBER OF DIMENSIONS
A1D1L     EQU       0           LOW BOUND OF DIMENSION 1
A1D1H     EQU       3           HIGH BOUND OF DIMENSION 1
A1D2L     EQU       0           LOW BOUND OF DIMENSION 2
A1D2H     EQU       5           HIGH BOUND OF DIMENSION 2
A1D3L     EQU       0           LOW BOUND OF DIMENSION 3
A1D3H     EQU       6           HIGH BOUND OF DIMENSION 3
A1SZ3     EQU       3           SIZE OF ELEMENT IN DIMENSION 3
A1SZ2     EQU       ((A1D3H-A1D3L)+1)*A1SZ3  SIZE OF ELEMENT IN D2
A1SZ1     EQU       ((A1D2H-A1D2L)+1)*A1SZ2  SIZE OF ELEMENT IN D1
AY1:      RMB       ((A1D1H-A1D1L)+1)*A1SZ1  ARRAY

*AY2 : ARRAY [A2D1L..A2D1H,A2D2L..A2D2H] OF WORD
*            [  -5 ..  -1  ,  2  .. 10  ]
A2DIM     EQU       2           NUMBER OF DIMENSIONS
A2D1L     EQU       -5          LOW BOUND OF DIMENSION 1
A2D1H     EQU       -1          HIGH BOUND OF DIMENSION 1
A2D2L     EQU       2           LOW BOUND OF DIMENSION 2
A2D2H     EQU       10          HIGH BOUND OF DIMENSION 2
A2SZ2     EQU       2                          SIZE OF ELEMENT IN D2
A2SZ1     EQU       ((A2D2H-A2D2L)+1)*A2SZ2  SIZE OF ELEMENT IN D1
AY2:      RMB       ((A2D1H-A2D1L)+1)*A2SZ1  ARRAY

          END
```

3 Arithmetic

3A 16-bit multiplication (MUL16)

Multiplies two 16-bit operands obtained from the stack and returns the 32-bit product in the stack. All numbers are stored in the usual 6809 style with their more significant bytes on top of the less significant bytes.

Procedure The program multiplies each byte of the multiplier by each byte of the multiplicand. It then adds the 16-bit partial products to form a full 32-bit product.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of multiplier
Less significant byte of multiplier

More significant byte of multiplicand
Less significant byte of multiplicand

Exit conditions

Order in stack (starting from the top)

More significant byte of more significant word of product
Less significant byte of more significant word of product

More significant byte of less significant word of product
Less significant byte of less significant word of product

Examples

1. Data: Multiplier = 0012_{16} (18_{10})
   Multiplicand = $03D1_{16}$ (977_{10})
   Result: Product = $000044B2_{16}$ ($17\,586_{10}$)

2. Data: Multiplier = $37D1_{16}$ ($14\,289_{10}$)
   Multiplicand = $A045_{16}$ ($41\,029_{10}$)
   Result: Product = $22F1AB55_{16}$ ($586\,264\,381_{10}$)

The more significant word of the product is incorrect if either operand is a signed negative number. To handle this case, determine the product's sign and replace all negative operands with their absolute values (two's complements) before calling MUL16.

To reduce the product to a 16-bit value for compatibility with other 16-bit arithmetic operations, follow the subroutine call with

```
LEAS  2,S   DROP MORE SIGNIFICANT WORD
```

Of course, this makes sense only in cases (such as Example 1) in which the more significant word is 0.

Registers used CC, D, U, X

Execution time Approximately 200 cycles

Program size 64 bytes

Data memory required 2 stack bytes

```
*
*
*
*
*    Title:              16 Bit Multiplication
*    Name:               MUL16
*
*
*
*    Purpose:            Multiply two unsigned 16-bit words and
*                        return a 32-bit unsigned product.
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          High byte of multiplier
*                          Low byte of multiplier
*                          High byte of multiplicand
*                          Low byte of multiplicand
*
*    Exit:               Product = multiplicand * multiplier
*                        TOP OF STACK
*                          High byte of high word of product
*                          Low byte of high word of product
*                          High byte of low word of product
*                          Low byte of low word of product
*
*    Registers Used:     CC,D,U,X
*
*    Time:               Approximately 200 cycles
*
*    Size:               Program  64 bytes
*                        Data     2 stack bytes
*
*

MUL16:
*
*         CLEAR PARTIAL PRODUCT IN FOUR STACK BYTES
*
          LDU       ,S          SAVE RETURN ADDRESS
          CLRA                  CLEAR 4-BYTE PARTIAL PRODUCT ON STACK
          CLRB
          STD       ,S          USE BYTES OCCUPIED BY RETURN ADDRESS
          PSHS      D           PLUS 2 EXTRA BYTES ON TOP OF STACK
*
*         MULTIPLY LOW BYTE OF MULTIPLIER TIMES LOW BYTE
*           OF MULTIPLICAND
*
          LDA       5,S         GET LOW BYTE OF MULTIPLIER
          LDB       7,S         GET LOW BYTE OF MULTIPLICAND
          MUL                   MULTIPLY BYTES
          STB       3,S         STORE LOW BYTE OF PRODUCT
          STA       2,S         STORE HIGH BYTE OF PRODUCT
*
*         MULTIPLY LOW BYTE OF MULTIPLIER TIMES HIGH BYTE
```

```
*           OF MULTIPLICAND
*
            LDA         5,S         GET LOW BYTE OF MULTIPLIER
            LDB         6,S         GET HIGH BYTE OF MULTIPLICAND
            MUL                     MULTIPLY BYTES
            ADDB        2,S         ADD LOW BYTE OF PRODUCT TO
                                    * PARTIAL PRODUCT
            STB         2,S
            ADCA        #0          ADD HIGH BYTE OF PRODUCT PLUS CARRY
                                    * TO PARTIAL PRODUCT

            STA         1,S         STORE HIGH BYTE OF PRODUCT
*
*           MULTIPLY HIGH BYTE OF MULTIPLIER TIMES LOW BYTE
*             OF MULTIPLICAND
*
            LDA         4,S         GET HIGH BYTE OF MULTIPLIER
            LDB         7,S         GET LOW BYTE OF MULTIPLICAND
            MUL                     MULTIPLY BYTES
            ADDB        2,S         ADD LOW BYTE OF PRODUCT TO
                                    * PARTIAL PRODUCT
            STB         2,S
            ADCA        1,S         ADD HIGH BYTE OF PRODUCT PLUS CARRY
                                    * TO PARTIAL PRODUCT

            STA         1,S
            BCC         MULHH       BRANCH IF NO CARRY
            INC         ,S          ELSE INCREMENT MOST SIGNIFICANT
                                    * BYTE OF PARTIAL PRODUCT
*
*           MULTIPLY HIGH BYTE OF MULTIPLIER TIMES HIGH BYTE
*             OF MULTIPLICAND
*
MULHH:
            LDA         4,S         GET HIGH BYTE OF MULTIPLIER
            LDB         6,S         GET HIGH BYTE OF MULTIPLICAND
            MUL                     MULTIPLY BYTES
            ADDB        1,S         ADD LOW BYTE OF PRODUCT TO PARTIAL
                                    * PRODUCT
            ADCA        ,S          ADD HIGH BYTE OF PRODUCT PLUS CARRY
                                    * TO PARTIAL PRODUCT
                                    * HIGH BYTES OF PRODUCT END UP IN D
*
*           RETURN WITH 32-BIT PRODUCT AT TOP OF STACK
*
            LDX         2,S         GET LOWER 16 BITS OF PRODUCT FROM STACK
            LEAS        6,S         REMOVE PARAMETERS FROM STACK
            PSHS        D,X         PUT 32-BIT PRODUCT AT TOP OF STACK
            JMP         ,U          EXIT TO RETURN ADDRESS
*
*
*           SAMPLE EXECUTION
*
*

SC3A:
```

```
        LDY     #1023     GET MULTIPLICAND
        LDX     #255      GET MULTIPLIER
        PSHS    X,Y       SAVE PARAMETERS IN STACK
        JSR     MUL16     16-BIT MULTIPLY
                          *RESULT OF 1023 * 255 = 260865
                          *  = 0003FB01 HEX
        PULS    X,Y       GET PRODUCT
        STX     PRODMS    IN MEMORY PRODMS   = 00H
                          *         PRODMS+1 = 03H
        STY     PRODLS    *         PRODLS   = FBH
                          *         PRODLS+1 = 01H

PRODMS: RMB     2         MORE SIGNIFICANT WORD OF PRODUCT
PRODLS: RMB     2         LESS SIGNIFICANT WORD OF PRODUCT

        END
```

3B 16-bit division (SDIV16, UDIV16, SREM16, UREM16)

Divides two 16-bit operands obtained from the stack and returns either the quotient or the remainder at the top of the stack. There are four entry points: SDIV16 and SREM16 return a 16-bit signed quotient or remainder, respectively, from dividing two 16-bit signed operands. UDIV16 and UREM16 return a 16-bit unsigned quotient or remainder, respectively, from dividing two 16-bit unsigned operands. All 16-bit numbers are stored in the usual 6809 style with the more significant byte on top of the less significant byte. The divisor is stored on top of the dividend. If the divisor is 0, the Carry flag is set to 1 and the result is 0; otherwise, the Carry flag is cleared.

Procedure If the operands are signed, the program determines the signs of the quotient and remainder and takes the absolute values of all negative operands. The program then performs an unsigned division using a shift-and-subtract algorithm. It shifts the quotient and dividend left, placing a 1 bit in the quotient each time a trial subtraction succeeds. Finally, it negates (i.e. subtracts from zero) all negative results. The Carry flag is cleared if the division is proper and set if the divisor is 0. A 0 divisor also causes a return with a result (quotient or remainder) of 0.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of divisor
Less significant byte of divisor

More significant byte of dividend
Less significant byte of dividend

Exit conditions

Order in stack starting from the top

More significant byte of result (quotient or remainder)
Less significant byte of result (quotient or remainder)

If the divisor is non-zero, Carry = 0 and the result is normal

If the divisor is zero, Carry = 1 and the result is 0000_{16}.

Examples

1. Data: Dividend = $03E0_{16} = 992_{10}$
   Divisor = $00B6_{16} = 182_{10}$
   Result: Quotient (from UDIV16) = 0005_{16}
   Remainder (from UREM16) = $0052_{16} = 0082_{10}$
   Carry = 0 (no divide-by-0 error)

2. Data: Dividend = $D73A_{16} = -10\,438_{10}$
   Divisor = $02F1_{16} = 753_{10}$
   Result: Quotient (from SDIV16) = $FFF3_{16} = -13_{10}$
   Remainder (from SREM16) = $FD77_{16} = -649_{10}$
   Carry = 0 (no divide-by-zero error)

Note that this routine produces a signed remainder. Its sign is the same as that of the dividend. To convert a negative remainder into an unsigned one, simply subtract 1 from the quotient and add the divisor to the remainder. The result of Example 2 is then

Quotient = $FFF2_{16} = -14_{10}$
Remainder (always positive) = $0068_{16} = 104_{10}$

Registers used A, B, CC, X, Y

Execution time: A maximum of 955 cycles plus an overhead of 10 (UREM16), 2 (UDIV16), 119 (SREM16), or 103 (SDIV16) cycles. Execution time depends on how many trial subtractions are successful and thus require the replacement of the previous dividend by the remainder. Each successful trial subtraction takes 9 extra cycles.

Program size 145 bytes

Data memory required 3 stack bytes

Special case If the divisor is 0, the program returns with the Carry flag set to 1 and a result (quotient or remainder) of 0.

```
*
*
*
*
*    Title:              16-Bit Division
*    Name:               SDIV16, UDIV16, SREM16, UREM16
*
*
*
*    Purpose:            SDIV16
*                          Divide 2 signed 16-bit words and
*                          return a 16-bit signed quotient.
*
*                        UDIV16
*                          Divide 2 unsigned 16-bit words and
*                          return a 16-bit unsigned quotient
*
*                        SREM16
*                         Divide 2 signed 16-bit words and
*                          return a 16-bit signed remainder
*
*                        UREM16
*                          Divide 2 unsigned 16-bit words and
*                          return a 16-bit unsigned remainder
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          High byte of divisor
*                          Low byte of divisor
*                          High byte of dividend
*                          Low byte of dividend
*
*    Exit:               TOP OF STACK
*                          High byte of result
*                          Low byte of result
*
*                        If no errors then
*                          Carry := 0
*                        else
*                          divide by zero error
*                          Carry := 1
*                          quotient : = 0
*                          remainder : = 0
*
*    Registers Used:     A,B,CC,X,Y
*
*    Time:               Approximately 955 cycles
*
*    Size:               Program  145 bytes
*                        Data       3 stack bytes
*
*

*
*SIGNED DIVISION, RETURNS REMAINDER
```

```
*
SREM16:
          LDA       #$FF            INDICATE REMAINDER TO BE RETURNED
          STA       ,-S             SAVE INDICATOR ON STACK
          BRA       CHKSGN          GO CHECK SIGNS
*
*SIGNED DIVISION, RETURNS QUOTIENT
*
SDIV16:
          CLR       ,-S             INDICATE QUOTIENT TO BE RETURNED
*
*IF DIVISOR IS NEGATIVE, TAKE ITS ABSOLUTE VALUE AND INDICATE
*  THAT QUOTIENT IS NEGATIVE
*
CHKSGN:
          LDD       #0              INDICATE QUOTIENT, REMAINDER POSITIVE
          PSHS      D               SAVE INDICATOR ON STACK
          LEAX      5,S             POINT TO DIVISOR
          TST       ,X              CHECK IF DIVISOR IS POSITIVE
          BPL       CHKDVD          BRANCH IF DIVISOR IS POSITIVE
          SUBD      ,X              ELSE TAKE ABSOLUTE VALUE OF DIVISOR
          STD       ,X
          COM       1,S             INDICATE QUOTIENT IS NEGATIVE
          BRA       CHKZRO
*
*IF DIVIDEND IS NEGATIVE, TAKE ITS ABSOLUTE VALUE, INDICATE THAT
*  REMAINDER IS NEGATIVE, AND INVERT SIGN OF QUOTIENT
*
CHKDVD:
          LEAX      2,X             POINT TO HIGH BYTE OF DIVIDEND
          TST       ,X              CHECK IF DIVIDEND IS POSITIVE
          BPL       CHKZRO          BRANCH IF DIVIDEND IS POSITIVE
          LDD       #0              ELSE TAKE ABSOLUTE VALUE OF DIVIDEND
          SUBD      ,X
          STD       ,X
          COM       ,S              INDICATE REMAINDER IS NEGATIVE
          COM       1,S             INVERT SIGN OF QUOTIENT
*
*UNSIGNED 16-BIT DIVISION, RETURNS QUOTIENT
*
UDIV16:
          CLR       ,-S             INDICATE QUOTIENT TO BE RETURNED
          BRA       CLRSGN
*
*UNSIGNED 16-BIT DIVISION, RETURNS REMAINDER
*
UREM16:
          LDA       #$FF            INDICATE REMAINDER TO BE RETURNED
          STA       ,-S
*
*UNSIGNED DIVISION, INDICATE QUOTIENT, REMAINDER BOTH POSITIVE
*
CLRSGN:
          LDD       #0              INDICATE QUOTIENT, REMAINDER POSITIVE
          PSHS      D
*
```

```
*CHECK FOR ZERO DIVISOR
*EXIT, INDICATING ERROR, IF FOUND
*
CHKZRO:
          LEAX      5,S             POINT TO DIVISOR
          LDD       ,X              TEST DIVISOR
          BNE       STRTDV          BRANCH IF DIVISOR NOT ZERO
          STD       2,X             DIVISOR IS ZERO, SO MAKE RESULT ZERO
          SEC                       INDICATE DIVIDE BY ZERO ERROR
          BRA       EXITDV          EXIT INDICATING ERROR
*
*DIVIDE UNSIGNED 32-BIT DIVIDEND BY UNSIGNED 16-BIT DIVISOR
*MEMORY ADDRESSES HOLD BOTH DIVIDEND AND QUOTIENT.  EACH TIME WE
*  SHIFT THE DIVIDEND ONE BIT LEFT, WE ALSO SHIFT A BIT OF THE
*  QUOTIENT IN FROM THE CARRY AT THE FAR RIGHT
*AT THE END, THE QUOTIENT HAS REPLACED THE DIVIDEND IN MEMORY
*  AND THE REMAINDER IS LEFT IN REGISTER D
*
STRTDV:
          LDD       #0              EXTEND DIVIDEND TO 32 BITS WITH 0
          LDY       #16             BIT COUNT = 16
          CLC                       START CARRY AT ZERO
*
*SHIFT 32-BIT DIVIDEND LEFT WITH QUOTIENT ENTERING AT FAR RIGHT
*
DIV16:
          ROL       3,X             SHIFT LOW BYTE OF DIVIDEND
                                    * QUOTIENT BIT ENTERS FROM CARRY
          ROL       2,X             SHIFT NEXT BYTE OF DIVIDEND
          ROLB                      SHIFT NEXT BYTE OF DIVIDEND
          ROLA                      SHIFT HIGH BYTE OF DIVIDEND
*
*DO A TRIAL SUBTRACTION OF DIVISOR FROM DIVIDEND
*IF DIFFERENCE IS NON-NEGATIVE, SET NEXT BIT OF QUOTIENT.
*  PERFORM ACTUAL SUBTRACTION, REPLACING QUOTIENT WITH DIFFERENCE.
*IF DIFFERENCE IS NEGATIVE, CLEAR NEXT BIT OF QUOTIENT
*
          CMPD      ,X              TRIAL SUBTRACTION OF DIVISOR
          BCS       CLRCRY          BRANCH IF SUBTRACTION FAILS
          SUBD      ,X              TRIAL SUBTRACTION SUCCEEDED,
                                    * SO SUBTRACT DIVISOR FROM
                                    * DIVIDEND
          SEC                       SET NEXT BIT OF QUOTIENT TO 1
          BRA       DECCNT
CLRCRY:
          CLC                       TRIAL SUBTRACTION FAILED, SO
                                    * SET NEXT BIT OF QUOTIENT TO 0
*
*UPDATE BIT COUNTER
*CONTINUE THROUGH 16 BITS
*
DECCNT:
          LEAY      -1,Y            CONTINUE UNTIL ALL BITS DONE
          BNE       DIV16
*
*SHIFT LAST CARRY INTO QUOTIENT
```

```
*
          ROL       3,X             SHIFT LAST CARRY INTO QUOTIENT
          ROL       2,X             INCLUDING MORE SIGNIFICANT BYTE
*
*SAVE REMAINDER IN STACK
*NEGATE REMAINDER IF INDICATOR SHOWS IT IS NEGATIVE
*
          STD       ,X              SAVE REMAINDER IN STACK
          TST       ,S              CHECK IF REMAINDER IS POSITIVE
          BEQ       TSTQSN          BRANCH IF REMAINDER IS POSITIVE
          LDD       #0              ELSE NEGATE IT
          SUBD      ,X
          STD       ,X              SAVE NEGATIVE REMAINDER
*
*NEGATE QUOTIENT IF INDICATOR SHOWS IT IS NEGATIVE
*
TSTQSN:
          TST       1,S             CHECK IF QUOTIENT IS POSITIVE
          BEQ       TSTRTN          BRANCH IF QUOTIENT IS POSITIVE
          LDD       #0              ELSE NEGATE IT
          SUBD      7,S
          STD       7,S             SAVE NEGATIVE QUOTIENT
*
*SAVE QUOTIENT OR REMAINDER, DEPENDING ON FLAG IN STACK
*
TSTRTN:
          CLC                       INDICATE NO DIVIDE-BY-ZERO ERROR
          TST       2,S             TEST QUOTIENT/REMAINDER FLAG
          BEQ       EXITDV          BRANCH TO RETURN QUOTIENT
          LDD       ,X              REPLACE QUOTIENT WITH REMAINDER
          STD       7,S
*
*REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITDV:
          LDX       3,S             SAVE RETURN ADDRESS
          LEAS      7,S             REMOVE PARAMETERS FROM STACK
          JMP       ,X              EXIT TO RETURN ADDRESS

*
*
*         SAMPLE EXECUTION
*
*

SC3B:
*
*SIGNED DIVIDE, OPRND1 / OPRND2, STORE QUOTIENT AT QUOTNT
*
          LDY       OPRND1          GET DIVIDEND
          LDX       OPRND2          GET DIVISOR
          PSHS      X,Y             SAVE PARAMETERS IN STACK
          JSR       SDIV16          SIGNED DIVIDE, RETURN QUOTIENT
          PULS      X               GET QUOTIENT
          STX       QUOTNT          RESULT OF -1023 / 123 = -8
                                    * IN MEMORY QUOTNT     = FF HEX
```

```
                                    *           QUOTNT + 1 = F8 HEX
*
*UNSIGNED DIVIDE, OPRND1 / OPRND2, STORE QUOTIENT AT QUOTNT
*
          LDY       OPRND1          GET DIVIDEND
          LDX       OPRND2          GET DIVISOR
          PSHS      X,Y             SAVE PARAMETERS IN STACK
          JSR       UDIV16          UNSIGNED DIVIDE, RETURN QUOTIENT
          PULS      X               GET QUOTIENT
          STX       QUOTNT          RESULT OF 64513 / 123 = 524
                                    * IN MEMORY QUOTNT     = 02 HEX
                                    *           QUOTNT + 1 = 0C HEX
*
*SIGNED DIVIDE, OPRND1 / OPRDN2, STORE REMAINDER AT REMNDR
*
          LDY       OPRND1          GET DIVIDEND
          LDX       OPRND2          GET DIVISOR
          PSHS      X,Y             SAVE PARAMETERS IN STACK
          JSR       SREM16          SIGNED DIVIDE, RETURN REMAINDER
          PULS      X               GET REMAINDER
          STX       REMNDR          REMAINDER OF -1023 / 123 = -39
                                    * IN MEMORY REMNDR     = FF HEX
                                    *           REMNDR + 1 = C7 HEX
*
*UNSIGNED DIVIDE, OPRND1 / OPRND2, STORE REMAINDER AT REMNDR
*
          LDY       OPRND1          GET DIVIDEND
          LDX       OPRND2          GET DIVISOR
          PSHS      X,Y             SAVE PARAMETERS IN STACK
          JSR       UREM16          UNSIGNED DIVIDE, RETURN REMAINDER
          PULS      X               GET QUOTIENT
          STX       REMNDR          RESULT OF 64513 / 123 = 61
                                    * IN MEMORY REMNDR     = 00 HEX
                                    *           REMNDR + 1 = 3D HEX
*
*DATA
*
OPRND1    FDB       -1023           DIVIDEND (64513 UNSIGNED)
OPRND2    FDB       123             DIVISOR
QUOTNT    RMB       2               QUOTIENT
REMNDR    RMB       2               REMAINDER
          END
```

3C Multiple-precision binary addition (MPBADD)

Adds two multi-byte unsigned binary numbers. Both are stored with their least significant bytes at the lowest address. The sum replaces the number with the base address lower in the stack. The length of the numbers (in bytes) is 255 or less.

Procedure The program clears the Carry flag initially and adds the operands one byte at a time, starting with the least significant bytes. The final Carry flag indicates whether the overall addition produced a carry. A length of 0 causes an immediate exit with no addition.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Length of the operands in bytes

More significant byte of base address of second operand (address containing the least significant byte of array 2)
Less significant byte of base address of second operand (address containing the least significant byte of array 2)

More significant byte of base address of first operand and sum (address containing the least significant byte of array 1)
Less significant byte of base address of first operand and sum (address containing the least significant byte of array 1)

Exit conditions

First operand (array 1) replaced by first operand (array 1) plus second operand (array 2)

Example

Data: Length of operands (in bytes) = 6
Top operand (array 2) = $19D028A193EA_{16}$

Bottom operand (array 1) = $293EABF059C7_{16}$
Result: Bottom operand (array 1) = Bottom operand (array 1) + Top operand (array 2) = $430ED491EDB1_{16}$
Carry = 0

Registers used A, B, CC, U, X

Execution time 21 cycles per byte plus 36 cycles overhead. For example, adding two 6-byte operands takes

$21 \times 6 + 36 = 162$ cycles

Program size 25 bytes

Data memory required None

Special case A length of 0 causes an immediate exit with the sum equal to the bottom operand (i.e. array 1 is unchanged). The Carry flag is cleared.

```
*
*
*
*
*    Title:              Multiple-Precision Binary Addition
*    Name:               MPBADD
*
*
*
*    Purpose:            Add 2 arrays of binary bytes
*                        Array1 := Array 1 + Array 2
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Length of the arrays in bytes
*                          High byte of array 2 address
*                          Low byte of array 2 address
*                          High byte of array 1 address
*                          Low byte of array 1 address
*
*                          The arrays are unsigned binary numbers
```

```
*                         with a maximum length of 255 bytes,
*                         ARRAY[0] is the least significant
*                         byte, and ARRAY[LENGTH-1] is the
*                         most significant byte.
*
*    Exit:                Array1 := Array1 + Array2
*
*    Registers Used:      A,B,CC,U,X
*
*    Time:                21 cycles per byte plus 36 cycles overhead
*
*    Size:                Program 25 bytes
*
*
*

MPBADD:
          *
          *CHECK IF LENGTH OF ARRAYS IS ZERO
          *EXIT WITH CARRY CLEARED IF IT IS
          *
          CLC                  CLEAR CARRY TO START
          LDB       2,S        CHECK LENGTH OF ARRAYS
          BEQ       ADEXIT     BRANCH (EXIT) IF LENGTH IS ZERO
          *
          *ADD ARRAYS ONE BYTE AT A TIME
          *
          LDX       5,S        GET BASE ADDRESS OF ARRAY 1
          LDU       3,S        GET BASE ADDRESS OF ARRAY 2
ADDBYT:
          LDA       ,U+        GET BYTE FROM ARRAY 2
          ADCA      ,X         ADD WITH CARRY TO BYTE FROM ARRAY 1
          STA       ,X+        SAVE SUM IN ARRAY 1
          DECB
          BNE       ADDBYT     CONTINUE UNTIL ALL BYTES SUMMED
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
ADEXIT:
          LDX       ,S         SAVE RETURN ADDRESS
          LEAS      7,S        REMOVE PARAMETERS FROM STACK
          JMP       ,X         EXIT TO RETURN ADDRESS

*
*
*         SAMPLE EXECUTION
*
*

SC3C:
          LDY       AY1ADR     GET FIRST OPERAND
          LDX       AY2ADR     GET SECOND OPERAND
          LDA       #SZAYS     LENGTH OF ARRAYS IN BYTES
          PSHS      A,X,Y      SAVE PARAMETERS IN STACK
          JSR       MPBADD     MULTIPLE-PRECISION BINARY ADDITION
                               *RESULT OF 12345678H + 9ABCDEF0H
```

```
                                *  = ACF13568H
                                * IN MEMORY AY1       = 68H
                                *           AY1+1     = 35H
                                *           AY1+2     = F1H
                                *           AY1+3     = ACH
                                *           AY1+4     = 00H
                                *           AY1+5     = 00H
                                *           AY1+6     = 00H
*
*  DATA
*
SZAYS     EQU       7           LENGTH OF ARRAYS IN BYTES

AY1ADR    FDB       AY1         BASE ADDRESS OF ARRAY 1
AY2ADR    FDB       AY2         BASE ADDRESS OF ARRAY 2

AY1:      FCB       $78,$56,$34,$12,0,0,0
AY2:      FCB       $F0,$DE,$BC,$9A,0,0,0

          END
```

3D Multiple-precision binary subtraction (MPBSUB)

Subtracts two multi-byte unsigned binary numbers. Both are stored with their least significant bytes at the lowest address. The subtrahend (number to be subtracted) is stored on top of the minuend (number from which it is subtracted). The difference replaces the minuend. The length of the numbers (in bytes) is 255 or less.

Procedure The program clears the Carry flag initially and subtracts the subtrahend from the minuend one byte at a time, starting with the least significant bytes. The final Carry flag indicates whether the overall subtraction required a borrow. A length of 0 causes an immediate exit with no subtraction.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Length of the operands in bytes

More significant byte of base address of subtrahend
Less significant byte of base address of subtrahend

More significant byte of base address of minuend
Less significant byte of base address of minuend

Exit conditions

Minuend replaced by minuend minus subtrahend

Example

Data:	Length of operands (in bytes) = 4
	Minuend = $2F5BA7C3_{16}$
	Subtrahend = $14DF35B8_{16}$
Result:	Minuend = $1A7C720B_{16}$
	Carry = 0, since no borrow is necessary

Registers used A, B, CC, U, X

Execution time 21 cycles per byte plus 36 cycles overhead. For example, subtracting two 6-byte operands takes

$21 \times 6 + 36 = 162$ cycles

Program size 25 bytes

Data memory required None

Special case A length of 0 causes an immediate exit with the minuend unchanged (i.e. the difference is equal to the minuend). The Carry flag is cleared.

```
*
*
*
*    Title:              Multiple-Precision Binary Subtraction
*    Name:               MPBSUB
*
*
*
*    Purpose:            Subtract 2 arrays of binary bytes
*                        Minuend := Minuend - Subtrahend
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Length of the arrays in bytes
*                          High byte of subtrahend address
*                          Low byte of subtrahend address
*                          High byte of minuend address
*                          Low byte of minuend address
*
*                          The arrays are unsigned binary numbers
*                          with a maximum length of 255 bytes,
*                          ARRAY[0] is the least significant
*                          byte, and ARRAY[LENGTH-1] is the
*                          most significant byte.
*
*    Exit:               Minuend := Minuend - Subtrahend
*
*    Registers Used:     A,B,CC,U,X
*
*    Time:               21 cycles per byte plus 36 cycles overhead
*
```

```
*    Size:              Program 25 bytes
*
*
*

MPBSUB:
          *
          *CHECK IF LENGTH OF ARRAYS IS ZERO
          *EXIT WITH CARRY CLEARED IF IT IS
          *
          CLC                 CLEAR CARRY TO START
          LDB       2,S       CHECK LENGTH OF ARRAYS
          BEQ       SBEXIT    BRANCH (EXIT) IF LENGTH IS ZERO
          *
          *SUBTRACT ARRAYS ONE BYTE AT A TIME
          *
          LDX       3,S       GET BASE ADDRESS OF SUBTRAHEND
          LDU       5,S       GET BASE ADDRESS OF MINUEND
SUBBYT:
          LDA       ,U        GET BYTE OF MINUEND
          SBCA      ,X+       SUBTRACT BYTE OF SUBTRAHEND WITH BORROW
          STA       ,U+       SAVE DIFFERENCE IN MINUEND
          DECB
          BNE       SUBBYT    CONTINUE UNTIL ALL BYTES SUBTRACTED
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
SBEXIT:
          LDX       ,S        SAVE RETURN ADDRESS
          LEAS      7,S       REMOVE PARAMETERS FROM STACK
          JMP       ,X        EXIT TO RETURN ADDRESS

*
*
*         SAMPLE EXECUTION
*
*

SC3D:
          LDY       AY1ADR    GET BASE ADDRESS OF MINUEND
          LDX       AY2ADR    GET BASE ADDRESS OF SUBTRAHEND
          LDA       #SZAYS    GET LENGTH OF ARRAYS IN BYTES
          PSHS      A,X,Y     SAVE PARAMETERS IN STACK
          JSR       MPBSUB    MULTIPLE-PRECISION BINARY SUBTRACTION
                              *RESULT OF 2F3E4D5CH-175E809FH
                              *  = 17DFCCBDH
                              * IN MEMORY AY1       = BDH
                              *           AY1+1     = CCH
                              *           AY1+2     = DFH
                              *           AY1+3     = 17H
                              *           AY1+4     = 00H
                              *           AY1+5     = 00H
                              *           AY1+6     = 00H
*
*  DATA
*
```

```
SZAYS     EQU       7          LENGTH OF ARRAYS IN BYTES

AY1ADR    FDB       AY1        BASE ADDRESS OF ARRAY 1
AY2ADR    FDB       AY2        BASE ADDRESS OF ARRAY 2

AY1:      FCB       $5C,$4D,$3E,$2F,0,0,0
AY2:      FCB       $9F,$80,$5E,$17,0,0,0

          END
```

3E Multiple-precision binary multiplication (MPBMUL)

Multiplies two multi-byte unsigned binary numbers. Both are stored with their least significant byte at the lowest address. The product replaces the multiplicand. The length of the numbers (in bytes) is 255 or less. Only the less significant bytes of the product are returned to provide compatibility with other multiple-precision binary operations.

Procedure The program multiplies the numbers one byte at a time, starting with the least significant bytes. It keeps a full double-length unsigned partial product in memory locations starting at PPROD (more significant bytes) and in the multiplicand (less significant bytes). The less significant bytes of the product replace the multiplicand as it is shifted. A length of 0 causes an exit with no multiplication.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Length of the operands in bytes

More significant byte of base address of multiplicand
Less significant byte of base address of multiplicand

More significant byte of base address of multiplier
Less significant byte of base address of multiplier

Exit conditions

Multiplicand replaced by multiplicand times multiplier

Example

Data: Length of operands (in bytes) = 4
Multiplicand = = $0005D1F7_{16} = 381431_{10}$
Multiplier = $00000AB1_{16} = 2737_{10}$
Result: Multiplicand = $3E39D1C7_{16} = 1043976647_{10}$

Note that MPBMUL returns only the less significant bytes (i.e. the number of bytes in the multiplicand and multiplier) of the product to maintain compatibility with other multiple-precision binary arithmetic operations. The more significant bytes of the product are available starting with their least significant byte at address PPROD. The user may need to check those bytes for a possible overflow.

Registers used All

Execution time Depends on the length of the operands and on the number of non-zero bytes in the multiplicand. If all bytes in the multiplicand are non-zero, the execution time is approximately

$90 \times \text{LENGTH}^2 + 90 \times \text{LENGTH} + 39$

If, for example, the operands are 4 bytes (32 bits) long, the execution time is approximately

$90 \times 16 + 90 \times 4 + 39 = 1440 + 360 + 39 = 1839$ cycles

There is a saving of $90 \times \text{LENGTH}$ cycles for each multiplicand byte that is 0.

Program size 96 bytes

Data memory required 256 bytes anywhere in RAM for the more significant bytes of the partial product (starting at address PPROD). This includes an overflow byte. Also 2 stack bytes.

Special case A length of 0 causes an immediate exit with the product equal to the multiplicand. The Carry flag is cleared.

```
*
*
*
*
*    Title:              Multiple-Precision Binary Multiplication
*    Name:               MPBMUL
*
*
*
*    Purpose:            Multiply 2 arrays of binary bytes
```

```
*                       Multiplicand := Multiplicand * multiplier
*
*    Entry:             TOP OF STACK
*                         High byte of return address
*                         Low byte of return address
*                         Length of the arrays in bytes
*                         High byte of multiplicand address
*                         Low byte of multiplicand address
*                         High byte of multiplier address
*                         Low byte of multiplier address

*                         The arrays are unsigned binary numbers
*                         with a maximum length of 255 bytes,
*                         ARRAY[0] is the least significant
*                         byte, and ARRAY[LENGTH-1] is the
*                         most significant byte.
*
*    Exit:              Multiplicand := Multiplicand * multiplier
*
*    Registers Used:    All
*
*    Time:              Assuming all multiplicand bytes are non-zero,
*                       then the time is approximately:
*                        (90 * length^2) + (90 * length) + 39 cycles
*
*    Size:              Program  96 bytes
*                       Data     256 bytes plus 2 stack bytes
*
*

MPBMUL:
*
*         CHECK LENGTH OF OPERANDS
*         EXIT IF LENGTH IS ZERO
*         SAVE LENGTH FOR USE AS LOOP COUNTER
*
          LDB       2,S         GET ARRAY LENGTH
          BEQ       EXITML      EXIT (RETURN) IF LENGTH IS ZERO
          PSHS      B           SAVE LENGTH AS MULTIPLICAND BYTE COUNTER
          LEAS      -1,S        RESERVE SPACE FOR MULTIPLICAND BYTE
*
*         CLEAR PARTIAL PRODUCT AREA (OPERAND LENGTH PLUS 1 BYTE FOR
*           OVERFLOW)
*
          LDX       #PPROD      POINT TO PARTIAL PRODUCT AREA
          CLRA                  GET ZERO FOR CLEARING
CLRPRD:
          STA       ,X+         CLEAR BYTE OF PARTIAL PRODUCT
          DECB
          BNE       CLRPRD      CONTINUE UNTIL ALL BYTES CLEARED
*
*         LOOP OVER ALL MULTIPLICAND BYTES
*         MULTIPLYING EACH ONE BY ALL MULTIPLIER BYTES
*
PROCBT:
          LDU       5,S         POINT TO MULTIPLICAND
```

```
          LDA       ,U          GET NEXT BYTE OF MULTIPLICAND
          STA       ,S          SAVE NEXT BYTE OF MULTIPLICAND
          BEQ       MOVBYT      SKIP MULTIPLICATION IF BYTE IS ZERO
*
*         MULTIPLY BYTE OF MULTIPLICAND TIMES EACH BYTE OF
*          MULTIPLIER
*
MULSTP:
          LDB       4,S         GET LENGTH OF OPERANDS IN BYTES
          CLRA                  SAVE AS 16-BIT LOOP COUNTER IN
          TFR       D,U           REGISTER U
          LDY       #PPROD      POINT TO PARTIAL PRODUCT
          LDX       7,S         POINT TO MULTIPLIER
MULLUP:
          LDA       ,X+         GET NEXT BYTE OF MULTIPLIER
          LDB       ,S          GET CURRENT BYTE OF MULTIPLICAND
          MUL                   MULTIPLY
          ADDB      ,Y          ADD RESULT TO PREVIOUS PRODUCT
          STB       ,Y+
          ADCA      ,Y
          STA       ,Y
          BCC       DECCTR      BRANCH IF ADDITION DOES NOT PRODUCE CARRY
          CLRA                  OTHERWISE, RIPPLE CARRY
OVRFL:
          INCA                  MOVE ON TO NEXT BYTE
          INC       A,Y         INCREMENT NEXT BYTE
          BEQ       OVRFL       BRANCH IF CARRY KEEPS RIPPLING
DECCTR:
          LEAU      -1,U        DECREMENT BYTE COUNT
          BNE       MULLUP      LOOP UNTIL MULTIPLICATION DONE
*
*         MOVE LOW BYTE OF PARTIAL PRODUCT INTO RESULT AREA
*         THIS OVERWRITES THE MULTIPLICAND BYTE USED IN THE
*           LATEST MULTIPLICATION LOOP
*
MOVBYT:
          LDX       5,S         POINT TO MULTIPLICAND AND RESULT
          LDY       #PPROD      POINT TO PARTIAL PRODUCT AREA
          LDB       ,Y          GET BYTE OF PARTIAL PRODUCT
          STB       ,X+         STORE IN ORIGINAL MULTIPLICAND
          STX       5,S         SAVE UPDATED MULTIPLICAND POINTER
*
*         SHIFT PARTIAL PRODUCT RIGHT ONE BYTE
*
          LDB       4,S         GET LENGTH OF OPERANDS IN BYTES
SHFTRT:
          LDA       1,Y         GET NEXT BYTE OF PARTIAL PRODUCT
          STA       ,Y+         MOVE BYTE RIGHT
          DECB                  DECREMENT BYTE COUNT
          BNE       SHFTRT      CONTINUE UNTIL ALL BYTES SHIFTED
          CLR       ,Y          CLEAR OVERFLOW
*
*         COUNT MULTIPLICAND DIGITS
*
          DEC       1,S         DECREMENT DIGIT COUNTER
          BNE       PROCBT      CONTINUE THROUGH ALL MULTIPLICAND DIGITS
```

```
          LEAS      2,S         REMOVE TEMPORARIES FROM STACK
*
*         REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITML:
          LDU       ,S          SAVE RETURN ADDRESS
          LEAS      7,S         REMOVE PARAMETERS FROM STACK
          JMP       ,U          EXIT TO RETURN ADDRESS

*
*         DATA
*
PPROD:    RMB       256         PARTIAL PRODUCT BUFFER WITH OVERFLOW BYTE
*
*
*         SAMPLE EXECUTION
*
*

SC3E:
          LDX       AY1ADR      GET MULTIPLICAND
          LDY       AY2ADR      GET MULTIPLIER
          LDA       #SZAYS      LENGTH OF OPERANDS IN BYTESS
          PSHS      A,X,Y       SAVE PARAMETERS IN STACK
          JSR       MPBMUL      MULTIPLE-PRECISION BINARY MULTIPLICATION
                                *RESULT OF 12345H * 1234H = 14B60404H
                                * IN MEMORY AY1       = 04H
                                *           AY1+1     = 04H
                                *           AY1+2     = B6H
                                *           AY1+3     = 14H
                                *           AY1+4     = 00H
                                *           AY1+5     = 00H
                                *           AY1+6     = 00H
          BRA       SC3E        CONTINUE
*
*  DATA
*
SZAYS     EQU       7           LENGTH OF OPERANDS IN BYTES

AY1ADR    FDB       AY1         BASE ADDRESS OF ARRAY 1
AY2ADR    FDB       AY2         BASE ADDRESS OF ARRAY 2

AY1:      FCB       $45,$23,$01,0,0,0,0
AY2:      FCB       $34,$12,0,0,0,0,0

          END
```

3F Multiple-precision binary division (MPBDIV)

Divides two multi-byte unsigned binary numbers. Both are stored with their least significant byte at the lowest address. The quotient replaces the dividend, and the address of the least significant byte of the remainder ends up in register X. The length of the numbers (in bytes) is 255 or less. The Carry flag is cleared if no errors occur; if a divide by 0 is attempted, the Carry flag is set to 1, the dividend is left unchanged, and the remainder is set to 0.

Procedure The program divides using the standard shift-and-subtract algorithm, shifting quotient and dividend and placing a 1 bit in the quotient each time a trial subtraction succeeds. An extra buffer holds the result of the trial subtraction; that buffer is simply switched with the buffer holding the dividend if the subtraction succeeds. The program sets the Carry flag if the divisor is 0 and clears Carry otherwise.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Length of the operands in bytes

More significant byte of base address of divisor
Less significant byte of base address of divisor

More significant byte of base address of dividend
Less significant byte of base address of dividend

Exit conditions

Dividend replaced by quotient (dividend divided by divisor).
If the divisor is non-zero, Carry = 0 and the result is normal.
If the divisor is 0, Carry = 1, the dividend is unchanged, and the remainder is 0.
The remainder is stored starting with its least significant byte at the address in X.

Example

Data: Length of operands (in bytes) = 3
Top operand (array 2 or divisor) = $000F45_{16} = 3909_{10}$
Bottom operand (array 1 or dividend) = $35A2F7_{16} = 3515127_{10}$

Result: Bottom operand (array 1) = Bottom operand (array 1) / Top operand (array 2) = $000383_{16} = 899_{10}$
Remainder (starting at address in X) = $0003A8_{16} = 936_{10}$
Carry flag = 0 to indicate no divide by zero error.

Registers used All

Execution time Depends on the length of the operands and on the number of 1 bits in the quotient (requiring a replacement of the dividend by the remainder). If the average number of 1 bits in the quotient is four per byte, the execution time is approximately

$400 \times \text{LENGTH}^2 + 580 \times \text{LENGTH} + 115$ cycles

where LENGTH is the length of the operands in bytes. If, for example, LENGTH = 4 (32-bit division), the approximate execution time is

$400 \times 4^2 + 580 \times 4 + 115 = 8835$ cycles

Program size 137 bytes

Data memory required 514 bytes anywhere in RAM for the buffers holding either the high dividend or the result of the trial subtraction (255 bytes starting at addresses HIDE1 and HIDE2, respectively), and for the pointers that assign the buffers to specific purposes (2 bytes starting at addresses HDEPTR and DIFPTR, respectively). Also 2 stack bytes.

Special cases

1. A length of 0 causes an immediate exit with the Carry flag cleared, the quotient equal to the original dividend, and the remainder undefined.

2. A divisor of 0 causes an exit with the Carry flag set to 1, the quotient equal to the original dividend, and the remainder equal to 0.

```
*
*
*
*
*    Title:              Multiple-Precision Binary Division
*    Name:               MPBDIV
*
*
*
*    Purpose:            Divide 2 arrays of binary bytes
*                        Array1 := Array 1 / Array 2
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Length of arrays in bytes
*                          High byte of divisor address
*                          Low byte of divisor address
*                          High byte of dividend address
*                          Low byte of dividend address
*
*                          The arrays are unsigned binary numbers
*                          with a maximum length of 255 bytes,
*                          ARRAY[0] is the least significant
*                          byte, and ARRAY[LENGTH-1] is the
*                          most significant byte.
*
*    Exit:               Array1 := Array1 / Array2
*                        Register X = Base address of remainder
*                        If no errors then
*                          Carry := 0
*                        else
*                          divide-by-zero error
*                          Carry := 1
*                          quotient := array 1 unchanged
*                          remainder := 0
*
*    Registers Used:     All
*
*    Time:               Assuming there are length/2 1 bits in the
*                        quotient then the time is approximately
*                          (400 * length^2) + (580 * length) +
*                           115 cycles
*
*    Size:               Program  137 bytes
*                        Data     514 bytes plus 2 stack bytes
*
*

MPBDIV:
*
*         EXIT INDICATING NO ERROR IF LENGTH OF OPERANDS IS ZERO
```

```
*
          LDB       2,S         TEST LENGTH OF OPERANDS
          BEQ       GOODRT      BRANCH (GOOD EXIT) IF LENGTH IS ZERO
*
*         SET UP HIGH DIVIDEND AND DIFFERENCE POINTERS
*         CLEAR HIGH DIVIDEND AND DIFFERENCE ARRAYS
*         ARRAYS 1 AND 2 ARE USED INTERCHANGEABLY FOR THESE TWO
*           PURPOSES.  THE POINTERS ARE SWITCHED WHENEVER A
*           TRIAL SUBTRACTION SUCCEEDS
*
          LDX       #HIDE1      GET BASE ADDRESS OF ARRAY 1
          STX       HDEPTR      DIVIDEND POINTER = ARRAY 1
          LDU       #HIDE2      GET BASE ADDRESS OF ARRAY 2
          STU       DIFPTR      DIFFERENCE POINTER = ARRAY 2
          CLRA                  GET ZERO FOR CLEARING ARRAYS
CLRHI:
          STA       ,X+         CLEAR BYTE OF ARRAY 1
          STA       ,U+         CLEAR BYTE OF ARRAY 2
          DECB
          BNE       CLRHI       CONTINUE THROUGH ALL BYTES
*
*         CHECK WHETHER DIVISOR IS ZERO
*         IF IT IS, EXIT INDICATING DIVIDE-BY-ZERO ERROR
*
          LDB       2,S         GET LENGTH OF OPERANDS
          LDX       3,S         GET BASE ADDRESS OF DIVISOR
CHKZRO:
          LDA       ,X+         EXAMINE BYTE OF DIVISOR
          BNE       INITDV      BRANCH IF BYTE IS NOT ZERO
          DECB                  CONTINUE THROUGH ALL BYTES
          BNE       CHKZRO
          SEC                   ALL BYTES ARE ZERO - INDICATE
                                * DIVIDE-BY-ZERO ERROR
          BRA       DVEXIT      EXIT
*
*         SET COUNT TO NUMBER OF BITS IN THE OPERANDS
*         COUNT := (LENGTH * 8)
*
INITDV:
          LDB       2,S         GET LENGTH OF OPERANDS IN BYTES
          LDA       #8          MULTIPLY LENGTH TIMES 8
          MUL
          PSHS      D           SAVE BIT COUNT AT TOP OF STACK
*
*         DIVIDE USING TRIAL SUBTRACTIONS
*
          CLC                   START QUOTIENT WITH 0 BIT
SHFTST:
          LDX       7,S         POINT TO BASE ADDRESS OF DIVIDEND
          LDB       4,S         GET LENGTH OF OPERANDS IN BYTES
*
*         SHIFT QUOTIENT AND LOWER DIVIDEND LEFT ONE BIT
*
SHFTQU:
          ROL       ,X+         SHIFT BYTE OF QUOTIENT/DIVIDEND LEFT
          DECB                  CONTINUE THROUGH ALL BYTES
```

```
          BNE       SHFTQU
*
*         SHIFT UPPER DIVIDEND LEFT WITH CARRY FROM LOWER DIVIDEND
*
          LDX       HDEPTR    POINT TO BASE ADDRESS OF UPPER DIVIDEND
          LDB       4,S       GET LENGTH OF OPERANDS IN BYTES
SHFTRM:
          ROL       ,X+       SHIFT BYTE OF UPPER DIVIDEND LEFT
          DECB                CONTINUE THROUGH ALL BYTES
          BNE       SHFTRM
*
*         TRIAL SUBTRACTION OF DIVISOR FROM DIVIDEND
*         SAVE DIFFERENCE IN CASE IT IS NEEDED LATER
*
          LDU       DIFPTR    POINT TO DIFFERENCE
          LDX       HDEPTR    POINT TO UPPER DIVIDEND
          LDY       5,S       POINT TO DIVISOR
          LDB       4,S       GET LENGTH OF OPERANDS IN BYTES
          CLC                 CLEAR BORROW INITIALLY
SUBDVS:
          LDA       ,X+       GET BYTE OF UPPER DIVIDEND
          SBCA      ,Y+       SUBTRACT BYTE OF DIVISOR WITH BORROW
          STA       ,U+       SAVE DIFFERENCE
          DECB                CONTINUE THROUGH ALL BYTES
          BNE       SUBDVS
*
*         NEXT BIT OF QUOTIENT IS 1 IF SUBTRACTION WAS SUCCESSFUL,
*          0 IF IT WAS NOT
*         THIS IS COMPLEMENT OF FINAL BORROW FROM SUBTRACTION
*
          BCC       RPLCDV    BRANCH IF SUBTRACTION WAS SUCCESSFUL,
                              * I.E., IT PRODUCED NO BORROW
          CLC                 OTHERWISE, TRIAL SUBTRACTION FAILED SO
                              * MAKE NEXT BIT OF QUOTIENT ZERO
          BRA       SETUP
*
*         TRIAL SUBTRACTION SUCCEEDED, SO REPLACE UPPER DIVIDEND
*           WITH DIFFERENCE BY SWITCHING POINTERS
*         SET NEXT BIT OF QUOTIENT TO 1
*
RPLCDV:
          LDX       HDEPTR    GET HIGH DIVIDEND POINTER
          LDU       DIFPTR    GET DIFFERENCE POINTER
          STU       HDEPTR    NEW HIGH DIVIDEND = DIFFERENCE
          STX       DIFPTR    USE OLD HIGH DIVIDEND FOR NEXT DIFFERENCE
          SEC                 SET NEXT BIT OF QUOTIENT TO 1
*
*         DECREMENT 16-BIT BIT COUNT BY 1
*
SETUP:
          LDX       ,S        GET SHIFT COUNT
          LEAX      -1,S      DECREMENT SHIFT COUNT BY 1
          STX       ,S
          BNE       SHFTST    CONTINUE UNLESS SHIFT COUNT EXHAUSTED
*
*         SHIFT LAST CARRY INTO QUOTIENT IF NECESSARY
```

```
*
          LEAS      2,S         REMOVE SHIFT COUNTER FROM STACK
          BCC       GOODRT      BRANCH IF NO CARRY
          LDX       5,S         POINT TO LOWER DIVIDEND/QUOTIENT
          LDB       2,S         GET LENGTH OF OPERANDS IN BYTES
LASTSH:
          ROL       ,X+         SHIFT BYTE OF QUOTIENT
          DECB                  CONTINUE THROUGH ALL BYTES
          BNE       LASTSH
*
*         CLEAR CARRY TO INDICATE NO ERRORS
*
GOODRT:
          CLC                   CLEAR CARRY - NO DIVIDE-BY-ZERO ERROR
*
*         REMOVE PARAMETERS FROM STACK AND EXIT
*
DVEXIT:
          LDX       HDEPTR      GET BASE ADDRESS OF REMAINDER
          LDU       ,S          SAVE RETURN ADDRESS
          LEAS      7,S         REMOVE PARAMETERS FROM STACK
          JMP       ,U          EXIT TO RETURN ADDRESS

*
*         DATA
*
HDEPTR:   RMB       2           POINTER TO HIGH DIVIDEND
DIFPTR:   RMB       2           POINTER TO DIFFERENCE BETWEEN HIGH
                                * DIVIDEND AND DIVISOR
HIDE1:    RMB       255         HIGH DIVIDEND BUFFER 1
HIDE2:    RMB       255         HIGH DIVIDEND BUFFER 2

*
*         SAMPLE EXECUTION
*
*

SC3F:
          LDX       AY1ADR      GET DIVIDEND
          LDY       AY2ADR      GET DIVISOR
          LDA       #SZAYS      LENGTH OF ARRAYS IN BYTES
          PSHS      A,X,Y       SAVE PARAMETERS IN STACK
          JSR       MPBDIV      MULTIPLE-PRECISION BINARY DIVISION
                                *RESULT OF 14B60404H / 1234H = 12345H
                                * IN MEMORY AY1      = 45H
                                *           AY1+1    = 23H
                                *           AY1+2    = 01H
                                *           AY1+3    = 00H
                                *           AY1+4    = 00H
                                *           AY1+5    = 00H
                                *           AY1+6    = 00H
          BRA       SC3F

*
*  DATA
*
```

```
SZAYS     EQU       7          LENGTH OF ARRAYS IN BYTES

AY1ADR    FDB       AY1        BASE ADDRESS OF ARRAY 1 (DIVIDEND)
AY2ADR    FDB       AY2        BASE ADDRESS OF ARRAY 2 (DIVISOR)

AY1:      FCB       $04,$04,$B6,$14,0,0,0,0
AY2:      FCB       $34,$12,0,0,0,0,0,0

          END
```

3G Multiple-precision binary comparison (MPBCMP)

Compares two multi-byte unsigned binary numbers and sets the Carry and Zero flags. Sets the Zero flag to 1 if the operands are equal and to 0 otherwise. Sets the Carry flag to 1 if the subtrahend is larger than the minuend and to 0 otherwise. Thus, it sets the flags as if it had subtracted the subtrahend from the minuend.

Procedure The program compares the operands one byte at a time, starting with the most significant bytes and continuing until it finds corresponding bytes that are not equal. If all the bytes are equal, it exits with the Zero flag set to 1. Note that the comparison starts with the operands' most significant bytes, whereas the subtraction (Subroutine 3D) starts with the least significant bytes.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Length of the operands in bytes

More significant byte of base address of subtrahend
Less significant byte of base address of subtrahend

More significant byte of base address of minuend
Less significant byte of base address of minuend

Exit conditions

Flags set as if subtrahend had been subtracted from minuend

Zero flag = 1 if subtrahend and minuend are equal, 0 if they are not equal

Carry flag = 1 if subtrahend is larger than minuend in the unsigned sense, 0 if it less than or equal to the minuend

Examples

1. Data: Length of operands (in bytes) = 6
   Top operand (subtrahend) = $19D028A193EA_{16}$
   Bottom operand (minuend) = $4E67BC15A266_{16}$
   Result: Zero flag = 0 (operands are not equal)
   Carry flag = 0 (subtrahend is not larger than minuend)
2. Data: Length of operands (in bytes) = 6
   Top operand (subtrahend) = $19D028A193EA_{16}$
   Bottom operand (minuend) = $19D028A193EA_{16}$
   Result: Zero flag = 1 (operands are equal)
   Carry flag = 0 (subtrahend is not larger than minuend)
3. Data: Length of operands (in bytes) = 6
   Top operand (subtrahend) = $19D028A193EA_{16}$
   Bottom operand (minuend) = $0F37E5991D7C_{16}$
   Result: Zero flag = 0 (operands are not equal)
   Carry flag = 1 (subtrahend is larger than minuend)

Registers used All

Execution time 20 cycles per byte that must be examined plus approximately 47 cycles overhead. That is, the program continues until it finds corresponding bytes that are not the same; each pair of bytes it must examine requires 20 cycles. There is a savings of 5 cycles if it finds unequal bytes.

Examples:

1. Comparing two 6-byte numbers that are equal takes

   $20 \times 6 + 47 = 167$ cycles

2. Comparing two 8-byte numbers that differ in the next to most significant bytes takes

   $20 \times 2 + 47 - 5 = 82$ cycles

Program Size: 30 bytes

Data memory required None

Special case A length of 0 causes an immediate exit with both the Carry flag and the Zero flag set to 1.

```
*
*
*
*    Title:              Multiple-Precision Binary Comparison
*    Name:               MPBCMP
*
*
*
*    Purpose:            Compare 2 arrays of binary bytes and
*                        return the Carry and Zero flags set or
*                        cleared
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Length of operands in bytes
*                          High byte of subtrahend address
*                          Low byte of subtrahend address
*                          High byte of minuend address
*                          Low byte of minuend address
*
*                          The arrays are unsigned binary numbers
*                          with a maximum length of 255 bytes,
*                          ARRAY[0] is the least significant
*                          byte, and ARRAY[LENGTH-1] is the
*                          most significant byte.
*
*    Exit:               IF minuend = subtrahend THEN
*                          C=0,Z=1
*                        IF minuend > subtrahend THEN
*                          C=0,Z=0
*                        IF minuend < subtrahend THEN
*                          C=1,Z=0
*                        IF array length = 0 THEN
*                          C=1,Z=1
*
*    Registers Used:     All
*
*    Time:               20 cycles per byte that must be examined plus
*                          47 cycles overhead
*
*    Size:               Program  30 bytes
*

*
*         CHECK IF LENGTH OF ARRAYS IS ZERO
*         EXIT WITH SPECIAL FLAG SETTING (C=1, Z=1) IF IT IS
*
```

```
MPBCMP:
          LDU       ,S          SAVE RETURN ADDRESS
          SEC                   SET CARRY IN CASE LENGTH IS 0
          LDB       2,S         GET LENGTH OF ARRAYS IN BYTES
          BEQ       EXITCP      BRANCH (EXIT) IF LENGTH IS ZERO
                                * C=1,Z=1 IN THIS CASE
*
*         COMPARE ARRAYS BYTE AT A TIME UNTIL UNEQUAL BYTES ARE FOUND OR ALL
*         BYTES COMPARED
*
          LDX       5,S         GET BASE ADDRESS OF MINUEND
          LDY       3,S         GET BASE ADDRESS OF SUBTRAHEND
          LEAX      B,X         DETERMINE ENDING ADDRESS OF MINUEND
          LEAY      B,Y         DETERMINE ENDING ADDRESS OF SUBTRAHEND
CMPBYT:
          LDA       ,-X         GET BYTE FROM MINUEND
          CMPA      ,-Y         COMPARE TO BYTE FROM SUBTRAHEND
          BNE       EXITCP      BRANCH (EXIT) IF BYTES ARE NOT EQUAL
          DECB
          BNE       CMPBYT      CONTINUE UNTIL ALL BYTES COMPARED
                                * IF PROGRAM FALLS THROUGH, THEN THE
                                * ARRAYS ARE IDENTICAL AND THE FLAGS ARE
                                * SET PROPERLY (C=0,Z=1)
*
*         REMOVE PARAMETERS FROM STACK AND EXIT
*         BE CAREFUL NOT TO AFFECT FLAGS (PARTICULARLY ZERO FLAG)
*
EXITCP:
          LEAS      7,S         REMOVE PARAMETERS FROM STACK
          JMP       ,U          EXIT TO RETURN ADDRESS

*
*
*         SAMPLE EXECUTION
*
*

SC3G:
          LDX       AY1ADR      GET BASE ADDRESS OF MINUEND
          LDY       AY2ADR      GET BASE ADDRESS OF SUBTRAHEND
          LDA       #SZAYS      GET LENGTH OF OPERANDS IN BYTES
          PSHS      A,X,Y       SAVE PARAMETERS IN STACK
          JSR       MPBCMP      MULTIPLE-PRECISION BINARY COMPARISON
                                *RESULT OF COMPARE (2F3E4D5CH,175E809FH)
                                *  IS C=0,Z=0
*
*  DATA
*
SZAYS     EQU       7           LENGTH OF OPERANDS IN BYTES

AY1ADR    FDB       AY1         BASE ADDRESS OF ARRAY 1
AY2ADR    FDB       AY2         BASE ADDRESS OF ARRAY 2

AY1:      FCB       $5C,$4D,$3E,$2F,0,0,0
AY2:      FCB       $9F,$80,$5E,$17,0,0,0

          END
```

3H Multiple-precision decimal addition (MPDADD)

Adds two multi-byte unsigned decimal (BCD) numbers. Both numbers are stored with their least significant digits at the lowest address. The sum replaces the number with the base address lower in the stack. The length of the numbers (in bytes) is 255 or less.

Procedure The program clears the Carry flag initially and then adds the operands one byte (two digits) at a time, starting with the least significant digits. The final Carry flag indicates whether the overall addition produced a carry. The sum replaces the operand with the base address lower in the stack (array 1 in the listing). A length of 0 causes an immediate exit with no addition.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Length of the operands in bytes

More significant byte of base address of second operand (address containing the least significant byte of array 2)
Less significant byte of base address of second operand (address containing the least significant byte of array 2)

More significant byte of base address of first operand and sum (address containing the least significant byte of array 1)
Less significant byte of base address of first operand and sum (address containing the least significant byte of array 1)

Exit conditions

First operand (array 1) replaced by first operand (array 1) plus second operand (array 2)

Example

Data: Length of operands (in bytes) = 6
Top operand (array 2) = 196028819315_{16}
Bottom operand (array 1) = 293471605987_{16}

Result: Bottom operand (array 1) = Bottom operand (array 1) + Top operand (array 2) = 489500425302_{16}
Carry = 0

Registers used A, B, CC, U, X

Execution time 23 cycles per byte plus 36 cycles overhead. For example, adding two 6-byte operands takes

$23 \times 6 + 36 = 174$ cycles

Program size 26 bytes

Data memory required None

Special case A length of 0 causes an immediate exit with the sum equal to the bottom operand (i.e. array 1 is unchanged). The Carry flag is cleared.

```
*
*    Title:              Multiple-Precision Decimal Addition
*    Name:               MPDADD
*
*
*
*    Purpose:            Add 2 arrays of BCD bytes
*                        Array1 := Array 1 + Array 2
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Length of the arrays in bytes
*                          High byte of array 2 address
*                          Low byte of array 2 address
*                          High byte of array 1 address
*                          Low byte of array 1 address
*
*                          The arrays are unsigned BCD numbers
*                          with a maximum length of 255 bytes,
*                          ARRAY[0] is the least significant
*                          byte, and ARRAY[LENGTH-1] is the
*                          most significant byte
*
*    Exit:               Array1 := Array1 + Array2
*
*    Registers Used:     A,B,CC,U,X
*
*    Time:               23 cycles per byte plus 36 cycles overhead
*
*    Size:               Program  26 bytes
*

MPDADD:
          *
          *CHECK IF LENGTH OF ARRAYS IS ZERO
          *EXIT WITH CARRY CLEARED IF IT IS
```

```
            *
            CLC                 CLEAR CARRY TO START
            LDB       2,S       CHECK LENGTH OF ARRAYS
            BEQ       ADEXIT    BRANCH (EXIT) IF LENGTH IS ZERO
            *
            *ADD OPERANDS 2 DIGITS AT A TIME
            *
            LDX       5,S       GET BASE ADDRESS OF ARRAY 1
            LDU       3,S       GET BASE ADDRESS OF ARRAY 2
ADDBYT:
            LDA       ,U+       GET 2 DIGITS FROM ARRAY 2
            ADCA      ,X        ADD 2 DIGITS FROM ARRAY 1 WITH CARRY
            DAA                 MAKE ADDITION DECIMAL
            STA       ,X+       SAVE SUM IN ARRAY 1
            DECB
            BNE       ADDBYT    CONTINUE UNTIL ALL DIGITS SUMMED
            *
            *REMOVE PARAMETERS FROM STACK AND EXIT
            *
ADEXIT:
            LDX       ,S        SAVE RETURN ADDRESS
            LEAS      7,S       REMOVE PARAMETERS FROM STACK
            JMP       ,X        EXIT TO RETURN ADDRESS

*
*           SAMPLE EXECUTION
*

SC3H:
            LDY       AY1ADR    GET FIRST OPERAND
            LDX       AY2ADR    GET SECOND OPERAND
            LDA       #SZAYS    LENGTH OF OPERANDS IN BYTES
            PSHS      A,X,Y     SAVE PARAMETERS IN STACK
            JSR       MPDADD    MULTIPLE-PRECISION BCD ADDITION
                                *RESULT OF 12345678H + 35914028H
                                *  = 48259706H
                                * IN MEMORY AY1      = 06H
                                *           AY1+1    = 97H
                                *           AY1+2    = 25H
                                *           AY1+3    = 48H
                                *           AY1+4    = 00H
                                *           AY1+5    = 00H
                                *           AY1+6    = 00H
            BRA       SC3H      REPEAT TEST
*
*  DATA
*
SZAYS       EQU       7         LENGTH OF OPERANDS IN BYTES

AY1ADR      FDB       AY1       BASE ADDRESS OF ARRAY 1
AY2ADR      FDB       AY2       BASE ADDRESS OF ARRAY 2

AY1:        FCB       $78,$56,$34,$12,0,0,0
AY2:        FCB       $28,$40,$91,$35,0,0,0

            END
```

3I Multiple-precision decimal subtraction (MPDSUB)

Subtracts two multi-byte unsigned decimal (BCD) numbers. Both are stored with their least significant digits at the lowest address. The subtrahend (number to be subtracted) is stored on top of the minuend (number from which it is subtracted). The difference replaces the minuend. The length of the numbers (in bytes) is 255 or less.

Procedure The program first clears the Carry flag and then subtracts the subtrahend from the minuend one byte (two digits) at a time, starting with the least significant digits. It does the decimal subtraction by forming the ten's complement of the subtrahend and adding it to the minuend. The final Carry flag indicates (in an inverted sense) whether the overall subtraction required a borrow. A length of 0 causes an immediate exit with no subtraction.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Length of the operands in bytes

More significant byte of base address of subtrahend
Less significant byte of base address of subtrahend

More significant byte of base address of minuend
Less significant byte of base address of minuend

Exit conditions

Minuend replaced by minuend minus subtrahend

Example

Data: Length of operands (in bytes) = 6
Minuend = 293471605987_{16}
Subtrahend = 196028819315_{16}

Result: Minuend = 097442786672_{16}
Carry = 1, since no borrow is necessary

Registers used A, B, CC, U, X

Execution time 27 cycles per byte plus 36 cycles overhead. For example, subtracting two 6-byte operands takes

$27 \times 6 + 36 = 198$ cycles

Program size 30 bytes

Data memory required None

Special case A length of 0 causes an immediate exit with the minuend unchanged (i.e., the difference is equal to the minuend). The Carry flag is set (1).

```
*
*
*
*
*    Title:              Multiple-Precision Decimal Subtraction
*    Name:               MPDSUB
*
*
*
*    Purpose:            Subtract 2 arrays of BCD bytes
*                        Minuend := Minuend - Subtrahend
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Length of the operands in bytes
*                          High byte of subtrahend address
*                          Low byte of subtrahend address
*                          High byte of minuend address
*                          Low byte of minuend address
*
*                          The arrays are unsigned BCD numbers with a
*                          maximum length of 255 bytes, ARRAY[0] is the
*                          least significant byte, and ARRAY[LENGTH-1]
*                          the most significant byte.
*
```

```
*     Exit:               Minuend : = Minuend - Subtrahend
*
*     Registers Used:     A,B,CC,U,X
*
*     Time:               27 cycles per byte plus 36 cycles overhead
*
*     Size:               Program 30 bytes
*
*
*

MPDSUB:
          *
          *CHECK IF LENGTH OF ARRAYS IS ZERO
          *EXIT WITH CARRY SET IF IT IS
          *
          SEC                 SET CARRY TO START
          LDB       2,S       CHECK LENGTH OF ARRAYS
          BEQ       SBEXIT    BRANCH (EXIT) IF LENGTH IS ZERO
          *
          *SUBTRACT OPERANDS 2 DIGITS AT A TIME BY ADDING TEN'S
          *  COMPLEMENT OF SUBTRAHEND TO MINUEND
          *CARRY IS INVERTED BORROW IN TEN'S COMPLEMENT ARITHMETIC
          *NOTE THAT DAA WORKS ONLY AFTER ADDITION INSTRUCTIONS
          *BYTE OF TEN'S COMPLEMENT = 99 HEX + INVERTED BORROW
          *  - BYTE OF SUBTRAHEND.  RESULT IS ALWAYS NON-NEGATIVE
          *  AND CARRY AND HALF CARRY ARE ALWAYS 0, SO NO PROBLEM
          *  WITH SUBTRACTING BCD OPERANDS
          *
          LDX       5,S       GET BASE ADDRESS OF MINUEND
          LDU       3,S       GET BASE ADDRESS OF SUBTRAHEND
SUBBYT:
          LDA       #$99      FORM 2 DIGITS OF 10'S COMPLEMENT
          ADCA      #0          OF SUBTRAHEND
          SUBA      ,U+
          ADDA      ,X        ADD 2 DIGITS OF MINUEND
          DAA                 MAKE RESULT DECIMAL
          STA       ,X+       SAVE DIFFERENCE OVER MINUEND
          DECB
          BEQ       SUBBYT    CONTINUE UNTIL ALL DIGITS SUBTRACTED
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
SBEXIT:
          LDX       ,S        SAVE RETURN ADDRESS
          LEAS      7,S       REMOVE PARAMETERS FROM STACK
          JMP       ,X        EXIT TO RETURN ADDRESS

*
*
*         SAMPLE EXECUTION
*
*

SC3I:
          LDY       AY1ADR    GET BASE ADDRESS OF MINUEND
```

```
            LDX         AY2ADR      GET BASE ADDRESS OF SUBTRAHEND
            LDA         #SZAYS      GET LENGTH OF OPERANDS IN BYTES
            PSHS        A,X,Y       SAVE PARAMETERS IN STACK
            JSR         MPDSUB      MULTIPLE-PRECISION DECIMAL SUBTRACTION
                                    *RESULT OF 28364150H-17598093H
                                    *  = 10766057H
                                    * IN MEMORY AY1       = 57H
                                    *           AY1+1     = 60H
                                    *           AY1+2     = 76H
                                    *           AY1+3     = 10H
                                    *           AY1+4     = 00H
                                    *           AY1+5     = 00H
                                    *           AY1+6     = 00H
            BRA         SC3I        REPEAT TEST
*
*  DATA
*
SZAYS       EQU         7           LENGTH OF OPERANDS IN BYTES

AY1ADR      FDB         AY1         BASE ADDRESS OF ARRAY 1
AY2ADR      FDB         AY2         BASE ADDRESS OF ARRAY 2

AY1:        FCB         $50,$41,$36,$28,0,0,0
AY2:        FCB         $93,$80,$59,$17,0,0,0

            END
```

3J Multiple-precision decimal multiplication (MPDMUL)

Multiplies two multi-byte unsigned decimal (BCD) numbers. Both numbers are stored with their least significant digits at the lowest address. The product replaces the multiplicand. The length of the numbers (in bytes) is 255 or less. Only the less significant bytes of the product are returned to provide compatibility with other multiple-precision decimal operations.

Procedure The program handles each digit of the multiplicand separately. It masks the digit off, shifts it (if it is the upper digit of a byte), and then uses it as a counter to determine how many times to add the multiplier to the partial product. The least significant digit of the partial product is saved as the next digit of the full product, and the partial product is shifted right 4 bits. The program uses a flag to determine whether it is currently working with the upper or lower digit of a byte. A length of 0 causes an exit with no multiplication.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Length of the operands in bytes

More significant byte of base address of multiplicand
Less significant byte of base address of multiplicand

More significant byte of base address of multiplier
Less significant byte of base address of multiplier

Exit conditions

Multiplicand replaced by multiplicand times multiplier

Example

Data: Length of operands (in bytes) = 4

Multiplicand = 0003518_{16}
Multiplier = 00006294_{16}
Result: Multiplicand = 221422826_{16}

Note that MPDMUL returns only the less significant bytes (i.e. the number of bytes in the multiplicand and multiplier) of the product to maintain compatibility with other multiple-precision decimal arithmetic operations. The more significant bytes of the product are available starting with their least significant byte at address PROD. The user may have to check those bytes for a possible overflow or extend the operands with additional zeros.

Registers used All

Execution time Depends on the length of the operands and on the size of the digits in the multiplicand (since those digits determine how many times the multiplier must be added to the partial product). If the average digit in the multiplicand has a value of 5, then the execution time is approximately

$170 \times \text{LENGTH}^2 + 370 \times \text{LENGTH} + 80$ cycles

where LENGTH is the number of bytes in the operands. If, for example, LENGTH = 6 (12 digits), the approximate execution time is

$$\begin{aligned} 170 \times 6^2 + 370 \times 6 + 80 &= 170 \times 36 + 2220 + 80 \\ &= 6120 + 2300 \\ &= 8420 \text{ cycles} \end{aligned}$$

Program size 164 bytes

Data memory required 511 bytes anywhere in RAM. This is temporary storage for the high bytes of the partial product (256 bytes starting at address PROD) and for the multiplicand (255 bytes starting at address MCAND). Also 3 stack bytes.

Special case A length of 0 causes an immediate exit with the multiplicand unchanged. The more significant bytes of the product (starting at address PROD) are undefined.

```
*
*
*
*
*    Title:              Multiple-Precision Decimal Multiplication
*    Name:               MPDMUL
*
*
*
*    Purpose:            Multiply 2 arrays of BCD bytes
*                        Multiplicand : = Multiplicand * multiplier
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Length of the arrays in bytes
*                          High byte of multiplicand address
*                          Low byte of multiplicand address
*                          High byte of multiplier address
*                          Low byte of multiplier address
*
*                          The arrays are unsigned BCD numbers
*                          with a maximum length of 255 bytes,
*                          ARRAY[0] is the least significant
*                          byte, and ARRAY[LENGTH-1] is the
*                          most significant byte.
*
*    Exit:               Multiplicand := Multiplicand * multiplier
*
*    Registers Used:     All
*
*    Time:               Assuming average digit value of multiplicand
*                        is 5, then the time is approximately
*                         (170 * length^2)+ (370 * length) + 80 cycles
*
*    Size:               Program  164 bytes
*                        Data     511 bytes plus 3 stack bytes
*
*
*
*         TEST LENGTH OF OPERANDS
*         EXIT IF LENGTH IS ZERO
*
MPDMUL:
          LDB       2,S         GET LENGTH OF OPERANDS IN BYTES
          LBEQ      EXITML      BRANCH (EXIT) IF LENGTH IS ZERO
*
*         SAVE DIGIT COUNTER AND UPPER/LOWER DIGIT FLAG ON STACK,
*           MAKE ROOM FOR NEXT DIGIT OF MULTIPLICAND ON STACK
*
          CLRA                  CLEAR DIGIT FLAG INITIALLY (LOWER DIGIT)
          PSHS      A,B         SAVE LENGTH, DIGIT FLAG ON STACK
          LEAS      -1,S        RESERVE SPACE ON STACK FOR NEXT DIGIT
                                * OF MULTIPLICAND
*
```

```
*         SAVE MULTIPLICAND IN TEMPORARY BUFFER (MCAND)
*         CLEAR PARTIAL PRODUCT CONSISTING OF UPPER BYTES
*           STARTING AT PROD AND LOWER BYTES REPLACING
*           MULTIPLICAND
*
          LDX       6,S         GET BASE ADDRESS OF MULTIPLICAND
          LDY       #MCAND      GET BASE ADDRESS OF TEMPORARY BUFFER
          LDU       #PROD       GET BASE ADDRESS OF UPPER PRODUCT
INITLP:
          LDA       ,X          MOVE BYTE OF MULTIPLICAND TO TEMPORARY
          STA       ,Y+          BUFFER
          CLRA
          STA       ,X+         CLEAR BYTE OF LOWER PRODUCT
          STA       ,U+         CLEAR BYTE OF UPPER PRODUCT
          DECB
          BNE       INITLP      CONTINUE THROUGH ALL BYTES
          STA       ,U          CLEAR OVERFLOW BYTE ALSO
*
*         LOOP THROUGH ALL BYTES OF MULTIPLICAND
*         USE EACH DIGIT TO DETERMINE HOW MANY TIMES TO ADD
*          MULTIPLIER TO PARTIAL PRODUCT
*
          LDU       #MCAND      POINT TO FIRST BYTE OF MULTIPLICAND
*
*         LOOP THROUGH 2 DIGITS PER BYTE
*         DURING LOWER DIGIT, DIGIT FLAG = 0
*         DURING UPPER DIGIT, DIGIT FLAG = FF HEX
*
PROCDG:
          LDA       ,U          GET BYTE OF MULTIPLICAND
          LDB       1,S         GET DIGIT FLAG
          BEQ       MASKDG      BRANCH IF ON LOWER DIGIT
          LSRA                  SHIFT UPPER DIGIT TO LOWER DIGIT
          LSRA
          LSRA
          LSRA
MASKDG:
          ANDA      #$0F        MASK OFF CURRENT DIGIT
          BEQ       MOVDIG      BRANCH (SKIP ADDITION) IF DIGIT IS ZERO
          STA       ,S          SAVE DIGIT ON STACK
*
*         ADD MULTIPLIER TO PRODUCT NUMBER OF TIMES GIVEN BY
*          DIGIT OF MULTIPLICAND
*
ADMULT:
          LDB       5,S         GET LENGTH OF OPERANDS
          LDY       #PROD       GET BASE ADDRESS OF PRODUCT
          LDX       8,S         GET BASE ADDRESS OF MULTIPLIER
          CLC                   CLEAR CARRY INITIALLY
ADBYTE:
          LDA       ,X+         GET NEXT BYTE OF MULTIPLIER
          ADCA      ,Y          ADD TO BYTE OF UPPER PRODUCT
          DAA                   MAKE SUM DECIMAL
          STA       ,Y+         STORE AS NEW PRODUCT
          DECB                  DECREMENT LOOP COUNTER
          BNE       ADBYTE      CONTINUE UNTIL LOOP COUNTER = 0
```

```
          LDA       ,Y          ADD CARRY TO OVERFLOW BYTE
          ADCA      #0
          DAA                   MAKE SUM DECIMAL
          STA       ,Y          SAVE NEW OVERFLOW BYTE
          DEC       ,S          DECREMENT NUMBER OF ADDITIONS
          BNE       ADMULT      CONTINUE UNITL ALL ADDITIONS DONE
*
*         STORE THE LEAST SIGNIFICANT DIGIT OF UPPER PRODUCT AS
*           THE NEXT DIGIT OF MULTIPLICAND
*
MOVDIG:
          LDX       6,S         GET BASE ADDRESS OF MULTIPLICAND
          LDY       #PROD       GET BASE ADDRESS OF UPPER PRODUCT
          LDB       ,Y          GET LEAST SIGNIFICANT BYTE OF PRODUCT
          ANDB      #$0F        MASK OFF LOWER DIGIT
          LDA       1,S         GET DIGIT FLAG
          BEQ       LOWDGT      BRANCH IF ON LOWER DIGIT
          ASLB                  ELSE SHIFT PRODUCT DIGIT TO UPPER DIGIT
          ASLB
          ASLB
          ASLB
          ADDB      ,X          ADD TO UPPER DIGIT OF MULTIPLICAND BYTE
          STB       ,X+
          BRA       SHFPRD      BRANCH TO SHIFT PRODUCT
LOWDGT:
          STB       ,X          STORE DIGIT IN MULTIPLICAND
*
*         SHIFT PARTIAL PRODUCT RIGHT 1 DIGIT (4 BITS)
*
SHFPRD:
          LDA       #4          SHIFT ONE DIGIT (4 BITS)
SETSHF:
          LDB       5,S         GET LENGTH
          INCB                  SHIFT LENGTH+1 BYTES TO INCLUDE OVERFLOW
          LDY       #PROD       POINT TO PARTIAL PRODUCT
          LEAY      B,Y         POINT PAST OVERFLOW BYTE
          CLC                   CLEAR CARRY INTO OVERFLOW
SHFARY:
          ROR       ,-Y         SHIFT BYTE OF PRODUCT RIGHT
          DECB                  CONTINUE THROUGH ALL BYTES
          BNE       SHFARY
          DECA                  DECREMENT SHIFT COUNT
          BNE       SETSHF      CONTINUE THROUGH 4 1-BIT SHIFTS
*
*         CHANGE OVER TO NEXT DIGIT IF ON LOWER DIGIT
*
          LDA       #$FF        GET UPPER DIGIT MARKER
          CMPA      1,S         COMPARE TO DIGIT FLAG
          BEQ       HIDIG       BRANCH IF ON UPPER DIGIT
          STA       1,S         ELSE SET DIGIT FLAG TO UPPER DIGIT
          BRA       PROCDG      PROCESS NEXT DIGIT
*
*         MOVE ON TO NEXT BYTE IF ON UPPER DIGIT
*
HIDIG:
          CLR       1,S         CLEAR DIGIT FLAG TO INDICATE LOW DIGIT
```

```
            LEAU    1,U         PROCEED TO NEXT BYTE OF MULTIPLICAND
            LDD     6,S         GET MULTIPLICAND POINTER
            ADDD    #1          POINT TO NEXT BYTE
            STD     6,S         SAVE MULTIPLICAND POINTER
            DEC     2,S         DECREMENT DIGIT COUNTER
            BNE     PROCDG      PROCESS NEXT DIGIT
            LEAS    3,S         REMOVE TEMPORARY STORAGE FROM STACK
*
*           REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITML:
            LDU     ,S          GET RETURN ADDRESS
            LEAS    7,S         REMOVE PARAMETERS FROM STACK
            JMP     ,U          EXIT TO RETURN ADDRESS

*
*           DATA
*
PROD:       RMB     256         PRODUCT BUFFER WITH OVERFLOW BYTE
MCAND:      RMB     255         MULTIPLICAND BUFFER

*
*
*           SAMPLE EXECUTION
*
*

SC3J:
            LDX     AY1ADR      GET MULTIPLICAND
            LDY     AY2ADR      GET MULTIPLIER
            LDA     #SZAYS      GET LENGTH OF ARRAYS IN BYTES
            PSHS    A,X,Y       SAVE PARAMETERS IN STACK
            JSR     MPDMUL      MULTIPLE-PRECISION DECIMAL MULTIPLICATION
                                *RESULT OF 1234H * 5718H = 7056012H
                                * IN MEMORY AY1      = 12H
                                *           AY1+1    = 60H
                                *           AY1+2    = 05H
                                *           AY1+3    = 07H
                                *           AY1+4    = 00H
                                *           AY1+5    = 00H
                                *           AY1+6    = 00H
            BRA     SC3J        REPEAT TEST

SZAYS       EQU     7           LENGTH OF ARRAYS IN BYTES

AY1ADR      FDB     AY1         BASE ADDRESS OF ARRAY 1
AY2ADR      FDB     AY2         BASE ADDRESS OF ARRAY 2

AY1:        FCB     $34,$12,0,0,0,0,0
AY2:        FCB     $18,$57,0,0,0,0,0

            END
```

3K Multiple-precision decimal division (MPDDIV)

Divides two multi-byte unsigned decimal (BCD) numbers. Both numbers are stored with their least significant digits at the lowest address. The quotient replaces the dividend; the base address of the remainder is also returned. The length of the numbers (in bytes) is 255 or less. The Carry flag is cleared if no errors occur; if a divide by 0 is attempted, the Carry flag is set to 1, the dividend is unchanged, and the remainder is set to 0.

Procedure The program divides by determining how many times the divisor can be subtracted from the dividend. It saves that number in the quotient, makes the remainder into the new dividend, and rotates the dividend and the quotient left one digit. The program subtracts using ten's complement arithmetic; the divisor is therefore replaced by its nine's complement to increase speed.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Length of the operands in bytes

More significant byte of base address of divisor
Less significant byte of base address of divisor

More significant byte of base address of dividend
Less significant byte of base address of dividend

Exit Conditions

Dividend replaced by dividend divided by divisor
If the divisor is non-zero, Carry = 0 and the result is normal
If the divisor is zero, Carry = 1, the dividend is unchanged, and the remainder is zero
The base address of the remainder (i.e. the address of its least significant digits) is in register X. The divisor is replaced by its nine's complement

Example

Data:	Length of operands (in bytes) = 4
	Dividend = 22142298_{16}
	Divisor = 00006294_{16}
Result:	Dividend = 00003518_{16}
	Remainder (base address in X) = 00000006_{16}
	Carry = 0, indicating no divide-by-0 error

Registers used All

Execution time Depends on the length of the operands and on the size of the digits in the quotient (determining how many times the divisor must be subtracted from the dividend). If the average digit in the quotient has a value of 5, the execution time is approximately

$410 \times \text{LENGTH}^2 + 750 \times \text{LENGTH} + 150$ cycles

where LENGTH is the length of the operands in bytes. If, for example, LENGTH = 6 (12 digits), the approximate execution time is

$$\begin{aligned} 410 \times 6^2 + 750 \times 6 + 150 &= 410 \times 36 + 4500 + 150 \\ &= 14\,760 + 4650 \\ &= 19\,410 \text{ cycles} \end{aligned}$$

Program size 169 bytes

Data memory required 514 bytes anywhere in RAM. This includes the buffers holding either the high dividend or the result of the trial subtraction (255 bytes each starting at addresses HIDE1 and HIDE2, respectively), and the pointers that assign the buffers to specific purposes (2 bytes each starting at addresses HDEPTR and DIFPTR, respectively). Also 3 stack bytes.

Special cases

1. A length of 0 causes an immediate exit with the Carry flag cleared, the quotient equal to the original dividend, and the remainder undefined.

2. A divisor of 0 causes an exit with the Carry flag set to 1, the quotient equal to the original dividend, and the remainder equal to 0.

```
*
*
*
*
*    Title:              Multiple-Precision Decimal Division
*    Name:               MPDDIV
*
*
*
*    Purpose:            Divide 2 arrays of BCD bytes
*                        Quotient := Dividend / divisor
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Length of operands in bytes
*                          High byte of divisor address
*                          Low byte of divisor address
*                          High byte of dividend address
*                          Low byte of dividend address
*
*                          The arrays are unsigned BCD numbers
*                          with a maximum length of 255 bytes,
*                          ARRAY[0] is the least significant
*                          byte, and ARRAY[LENGTH-1] is the
*                          most significant byte.
*
*    Exit:               Dividend := dividend / divisor
*                        If no errors then
*                          Carry := 0
*                          Dividend unchanged
*                          remainder := 0
*
*    Registers Used:     All
*
*    Time:               Assuming the average digit value in the
*                        quotient is 5, then the time is approximately
*                          (410 * length^2) + (750 * length) + 150
*                           cycles
*
*    Size:               Program  169 bytes
*                        Data     510 bytes plus 3 stack bytes
*
*
*
*         CHECK LENGTH OF OPERANDS
*         EXIT WITH CARRY CLEARED IF LENGTH IS ZERO
*
MPDDIV:
          CLC                   CLEAR CARRY IN CASE OF ZERO LENGTH
          LDB       2,S         GET LENGTH OF OPERANDS
          LBEQ      EXITDV      BRANCH (EXIT) IF LENGTH IS ZERO
*
*         SET UP HIGH DIVIDEND AND DIFFERENCE POINTERS
*         CLEAR HIGH DIVIDEND AND DIFFERENCE ARRAYS
```

```
*         ARRAYS 1 AND 2 ARE USED INTERCHANGEABLY FOR THESE TWO
*           PURPOSES.  THE POINTERS ARE SWITCHED WHENEVER A
*           TRIAL SUBTRACTION SUCCEEDS
*
          LDX       #HIDE1     GET BASE ADDRESS OF ARRAY 1
          STX       HDEPTR     DIVIDEND POINTER = ARRAY 1
          LDU       #HIDE2     GET BASE ADDRESS OF ARRAY 2
          STU       DIFPTR     DIFFERENCE POINTER = ARRAY 2
          CLRA                 GET ZERO FOR CLEARING
CLRHI:
          STA       ,X+        CLEAR BYTE OF ARRAY 1
          STA       ,U+        CLEAR BYTE OF ARRAY 2
          DECB                 CONTINUE THROUGH ALL BYTES
          BNE       CLRHI
*
*         CHECK WHETHER DIVISOR IS ZERO - EXIT WITH CARRY SET IF IT IS
*
          LDB       2,S        GET LENGTH OF OPERANDS
          LDX       3,S        POINT TO DIVISOR
CHKZRO:
          LDA       ,X+        GET BYTE OF DIVISOR
          BNE       NINESC     BRANCH (EXIT) IF BYTE IS NOT ZERO
          DECB                 CONTINUE THROUGH ALL BYTES
          BNE       CHKZRO
          SEC                  ALL BYTES ARE ZERO - SET CARRY AND EXIT
          LBRA      EXITDV     INDICATING DIVIDE-BY-ZERO ERROR
*
*         TAKE NINES COMPLEMENT OF DIVISOR TO SIMPLIFY SUBTRACTION
*
NINESC:
          LDB       2,S        GET LENGTH OF OPERANDS
          LDX       3,S        POINT TO DIVISOR
NINESB:
          LDA       #$99       TAKE NINES COMPLEMENT OF EACH BYTE
          SUBA      ,X
          STA       ,X+
          DECB                 CONTINUE THROUGH ALL BYTES
          BNE       NINESB
*
*         SET COUNT TO NUMBER OF DIGITS PLUS 1
*          COUNT := LENGTH * 2 + 1
*
          LDB       2,S        GET LENGTH OF OPERANDS
          CLRA                 EXTEND LENGTH TO 16 BITS
          ASLB                 MULTIPLY LENGTH TIMES 2
          ROLA
          ADDD      #1         2 * LENGTH + 1
          PSHS      D          SAVE DIGIT COUNT ON STACK
          CLR       ,-S        SAVE TENS COUNT ON STACK
*
*         SET UP FOR DIGIT SHIFT
*
DIGSET:
          LDY       1,S        GET DIGIT COUNT
          LEAY      -1,Y       DECREMENT DIGIT COUNT
          STY       1,S        SAVE DECREMENTED DIGIT COUNT
```

```
          BEQ       CHKTNS    BRANCH IF ALL DIGITS DONE
          LDA       #4        FOUR BITS PER DIGIT
*
*         DIGIT SHIFT
*
DIGSHF:
          LDX       8,S       POINT TO DIVIDEND
          LSL       ,S        SHIFT HIGH BIT INTO CARRY
          LDB       5,S       GET LENGTH OF OPERANDS
*
*         SHIFT QUOTIENT AND LOWER DIVIDEND LEFT ONE BIT
*
SHFTQU:
          ROL       ,X+       SHIFT BYTE OF QUOTIENT/DIVIDEND LEFT
          DECB                CONTINUE THROUGH ALL BYTES
          BNE       SHFTQU
*
*         SHIFT UPPER DIVIDEND LEFT WITH CARRY FROM LOWER DIVIDEND
*
          LDX       HDEPTR    POINT TO BASE ADDRESS OF UPPER DIVIDEND
          LDB       5,S       GET LENGTH OF OPERANDS
SHFTUP:
          ROL       ,X+       SHIFT BYTE OF UPPER DIVIDEND LEFT
          DECB                CONTINUE THROUGH ALL BYTES
          BNE       SHFTUP
          DECA                DECREMENT DIGIT BIT COUNT
          BNE       DIGSHF    LOOP UNTIL DONE
*
*         PERFORM DIVISION BY TRIAL SUBTRACTIONS
*         KEEP REMAINDER IN CASE IT IS NEEDED LATER
*         FINAL CARRY IS AN INVERTED BORROW
*
          CLR       ,S        TENS COUNTER = 0
SETSUB:
          LDU       DIFPTR    POINT TO DIFFERENCE
          LDX       HDEPTR    POINT TO UPPER DIVIDEND
          LDY       6,S       POINT TO DIVISOR
          LDB       5,S       GET LENGTH OF OPERANDS IN BYTES
          SEC                 SET INVERTED BORROW INITIALLY
                              * TO FORM 10'S COMPLEMENT
SUBDVS:
          LDA       ,X+       GET BYTE OF HIGH DIVIDEND
          ADCA      ,Y+       SUBTRACT BYTE OF DIVISOR BY ADDING
                              * BYTE OF NINE'S COMPLEMENT
          DAA                 MAKE DIFFERENCE DECIMAL
          STA       ,U+       SAVE DIFFERENCE
          DECB                CONTINUE THROUGH ALL BYTES
          BNE       SUBDVS
*
*         IF DIFFERENCE IS POSITIVE (CARRY SET), REPLACE HIGH
*           DIVIDEND WITH DIFFERENCE AND ADD 10 TO 10'S COUNT
*
          BCC       DIGSET    BRANCH IF DIFFERENCE IS NEGATIVE
          LDX       HDEPTR    GET HIGH DIVIDEND POINTER
          LDU       DIFPTR    GET DIFFERENCE POINTER
          STU       HDEPTR    NEW HIGH DIVIDEND = DIFFERENCE
```

```
            STX         DIFPTR      USE OLD HIGH DIVIDEND FOR NEXT DIFFERENCE
            LDA         #$10        ADD 10 TO 10'S COUNT
            ADDA        ,S
            STA         ,S          SAVE SUM ON STACK
            BRA         SETSUB      CONTINUE WITH TRIAL SUBTRACTIONS
*
*           DO LAST SHIFT IF TENS COUNT IS NOT ZERO
*
CHKTNS:
            LDA         ,S          GET TENS COUNT
            LEAS        3,S         REMOVE TEMPORARIES FROM STACK
            BEQ         GOODRT      BRANCH IF TENS COUNT IS ZERO
            PSHS        A           SAVE TENS COUNT
            LDA         #4          4 BIT SHIFT TO MOVE DIGIT
CSHIFT:
            LDX         6,S         POINT TO QUOTIENT
            LDB         3,S         GET LENGTH OF OPERANDS
            LSL         ,S          SHIFT TENS COUNT INTO CARRY
LSTSHF:
            ROL         ,X+         SHIFT QUOTIENT LEFT 1 BIT
            DECB                    CONTINUE THROUGH ALL BYTES
            BNE         LSTSHF
            DECA                    CONTINUE THROUGH 4 BIT SHIFT
            BNE         CSHIFT
            LEAS        1,S         REMOVE TEMPORARY STORAGE FROM STACK
GOODRT:
            CLC                     CLEAR CARRY FOR GOOD RETURN
*
*           REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITDV:
            LDX         HDEPTR      GET BASE ADDRESS OF REMAINDER
            LDU         ,S          SAVE RETURN ADDRESS
            LEAS        7,S         REMOVE PARAMETERS FROM STACK
            JMP         ,U          EXIT TO RETURN ADDRESS

*
*           DATA
*
HDEPTR:     RMB         2           POINTER TO HIGH DIVIDEND
DIFPTR:     RMB         2           POINTER TO DIFFERENCE BETWEEN HIGH
                                    * DIVIDEND AND DIVISOR
HIDE1:      RMB         255         HIGH DIVIDEND BUFFER 1
HIDE2:      RMB         255         HIGH DIVIDEND BUFFER 2

*
*
*           SAMPLE EXECUTION
*
*

SC3K:
            LDX         AY1ADR      GET DIVIDEND
            LDY         AY2ADR      GET DIVISOR
            LDA         #SZAYS      LENGTH OF ARRAYS IN BYTES
            PSHS        A,X,Y       SAVE PARAMETERS IN STACK
```

```
          JSR       MPDDIV    MULTIPLE-PRECISION DECIMAL DIVISION
                              *RESULT OF 3822756 / 1234 = 3097
                              * IN MEMORY AY1      = 97H
                              *           AY1+1    = 30H
                              *           AY1+2    = 00H
                              *           AY1+3    = 00H
                              *           AY1+4    = 00H
                              *           AY1+5    = 00H
                              *           AY1+6    = 00H
          BRA       SC3K      REPEAT TEST

SZAYS     EQU       7         LENGTH OF ARRAYS IN BYTES

AY1ADR    FDB       AY1       BASE ADDRESS OF ARRAY 1 (DIVIDEND)
AY2ADR    FDB       AY2       BASE ADDRESS OF ARRAY 2 (DIVISOR)

AY1:      FCB       $56,$27,$82,$03,0,0,0
AY2:      FCB       $34,$12,0,0,0,0,0,0

          END
```

3L Multiple-precision decimal comparison

Compares two multi-byte unsigned decimal (BCD) numbers, setting the Carry and Zero flags. Sets the Zero flag to 1 if the operands are equal and to 0 otherwise. Sets the Carry flag to 1 if the subtrahend is larger than the minuend and to 0 otherwise. It thus sets the flags as if it had subtracted the subtrahend from the minuend.

Note This program is exactly the same as Subroutine 3G, the multiple-precision binary comparison, since the form of the operands does not matter if they are only being compared. See Subroutine 3G for a listing and other details.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Length of the operands in bytes

More significant byte of base address of subtrahend
Less significant byte of base address of subtrahend

More significant byte of base address of minuend
Less significant byte of base address of minuend

Exit conditions

Flags set as if subtrahend had been subtracted from minuend

Zero flag = 1 if subtrahend and minuend are equal, 0 if they are not equal

Carry flag = 1 if subtrahend is larger than minuend in the unsigned sense, 0 if it less than or equal to the minuend

Examples

1. Data: Length of operands (in bytes) = 6
   Top operand (subtrahend) = 196528719340_{16}

		Bottom operand (minuend) = 456780153266_{16}
	Result:	Zero flag = 0 (operands are not equal)
		Carry flag = 0 (subtrahend is not larger than minuend)
2.	Data:	Length of operands (in bytes) = 6
		Top operand (subtrahend) = 196528719340_{16}
		Bottom operand (minuend) = 196528719340_{16}
	Result:	Zero flag = 1 (operands are equal)
		Carry flag = 0 (subtrahend is not larger than minuend)
3.	Data:	Length of operands (in bytes) = 6
		Top operand (subtrahend) = 196528719340_{16}
		Bottom operand (minuend) = 073785991074_{16}
	Result:	Zero flag = 0 (operands are not equal)
		Carry flag = 1 (subtrahend is larger than minuend)

4 Bit manipulation and shifts

4A Bit field extraction (BFE)

Extracts a field of bits from a word and returns it in the least significant bit positions. The width of the field and its lowest bit position are specified.

Procedure The program obtains a mask consisting of right-justified 1 bits covering the width of the field. It shifts the mask left to align it with the specified lowest bit position and obtains the field by logically ANDing the mask with the data. It then normalizes the bit field by shifting it right to make it start in bit 0.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Starting (lowest) bit position in the field (0–15)

Width of the field in bits (0–15)

More significant byte of data
Less significant byte of data

Exit conditions

Bit field in register D (normalized to bit 0)

Examples

1. Data: Value = $F67C_{16} = 1111011001111100_2$
   Lowest bit position = 4
   Width of field in bits = 8
   Result: Bit field = $0067_{16} = 0000000001100111_2$
   We have extracted 8 bits from the original data, starting with bit 4 (i.e. bits 4–11).

2. Data: Value = $A2D4_{16} = 1010001011010100_2$
   Lowest bit position = 6
   Width of field in bits = 5
   Result: Bit field = $000B_{16} = 0000000000001011_2$
   We have extracted 5 bits from the original data, starting with bit 6 (i.e. bits 6–10).

Registers used A, B, CC, U, X

Execution time 27 × LOWEST BIT POSITION plus 85 cycles overhead. The lowest bit position determines how many times the program must shift the mask left and the bit field right. For example, if the field starts in bit 6, the execution time is

$27 \times 6 + 85 = 162 + 85 = 247$ cycles

Program size 67 bytes (including the table of masks)

Data memory required None

Special cases

1. Requesting a field that would extend beyond the end of the word causes the program to return with only the bits through bit 15. That is,

no wraparound is provided. If, for example, the user asks for a 10-bit field starting at bit 8, the program will return only 8 bits (bits 8 – 15).

2. Both the lowest bit position and the number of bits in the field are interpreted mod 16. That is, for example, bit position 17 is equivalent to bit position 1 and a field of 20 bits is equivalent to a field of 4 bits.

3. Requesting a field of zero width causes a return with a result of 0.

```
*
*
*
*
*    Title:              Bit Field Extraction
*    Name:               BFE
*
*
*
*    Purpose:            Extract a field of bits from a 16-bit
*                        word and return the field normalized
*                        to bit 0.
*                        NOTE: IF THE REQUESTED FIELD IS TOO
*                              LONG, THEN ONLY THE BITS THROUGH
*                              BIT 15 WILL BE RETURNED.  FOR
*                              EXAMPLE, IF A 4 BIT FIELD IS
*                              REQUESTED STARTING AT BIT 15, THEN
*                              ONLY 1 BIT (BIT 15) WILL BE
*                              RETURNED.
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Lowest (starting) bit position in
*                            the field (0..15)
*                          Width of field in bits (1..16)
*                          High byte of data
*                          Low byte of data
*
*    Exit:               Register D = Field (normalized to bit 0)
*
*    Registers Used:     A,B,CC,U,X
*
*    Time:               85 cycles overhead plus
*                          (27 * lowest bit position) cycles
*
*    Size:               Program  67 bytes
*
*
*

BFE:
         LDU       ,S        SAVE RETURN ADDRESS
*
*        EXIT WITH ZERO RESULT IF WIDTH OF FIELD IS ZERO
*
```

```
          CLRB                MAKE LOW BYTE OF FIELD ZERO INITIALLY
          LDA       3,S       GET FIELD WIDTH
          BEQ       EXITBF    BRANCH (EXIT) IF FIELD WIDTH IS ZERO
                              * NOTE: RESULT IN D IS ZERO
*
*         USE FIELD WIDTH TO OBTAIN EXTRACTION MASK FROM ARRAY
*         MASK CONSISTS OF A RIGHT-JUSTIFIED SEQUENCE OF 1 BITS
*           WITH LENGTH GIVEN BY THE FIELD WIDTH
*
          DECA                SUBTRACT 1 FROM FIELD WIDTH TO FORM INDEX
          ANDA      #$0F      BE SURE INDEX IS 0 TO 15
          ASLA                MULTIPLY BY 2 SINCE MASKS ARE WORD-LENGTH
          LEAX      MSKARY,PCR  GET BASE ADDRESS OF MASK ARRAY
          LDX       A,X       GET MASK FROM ARRAY
*
*         SHIFT MASK LEFT LOGICALLY TO ALIGN IT WITH LOWEST BIT
*           POSITION IN FIELD
*
          LDA       2,S       GET LOWEST BIT POSITION
          ANDA      #$0F      MAKE SURE VALUE IS BETWEEN 0 AND 15
          BEQ       GETFLD    BRANCH WITHOUT SHIFTING IF LOWEST
                              * BIT POSITION IS 0
          STA       ,S        SAVE LOWEST BIT POSITION IN STACK TWICE
          STA       1,S         TO COUNT SHIFTS OF MASK, RESULT
          TFR       X,D       MOVE MASK TO REGISTER D FOR SHIFTING
SHFTMS:
          ASLB                SHIFT LOW BYTE OF MASK LEFT LOGICALLY
          ROLA                SHIFT HIGH BYTE OF MASK LEFT
          DEC       ,S        CONTINUE UNTIL 1 BITS ALIGNED TO
          BNE       SHFTMS      FIELD'S LOWEST BIT POSITION
*
*         OBTAIN FIELD BY LOGICALLY ANDING SHIFTED MASK WITH VALUE
*
GETFLD:
          ANDB      5,S       AND LOW BYTE OF VALUE WITH MASK
          ANDA      4,S       AND HIGH BYTE OF VALUE WITH MASK
*
*         NORMALIZE FIELD TO BIT 0 BY SHIFTING RIGHT LOGICALLY FROM
*           LOWEST BIT POSITION
*
          TST       1,S       TEST LOWEST BIT POSITION
          BEQ       EXITBF    BRANCH (EXIT) IF LOWEST POSITION IS 0
SHFTFL:
          LSRA                SHIFT HIGH BYTE OF FIELD RIGHT LOGICALLY
          RORB                SHIFT LOW BYTE OF FIELD RIGHT
          DEC       1,S       CONTINUE UNTIL LOWEST BIT OF FIELD IS
          BNE       SHFTFL      IN BIT POSITION 0
*
*         REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITBF:
          LEAS      6,S       REMOVE PARAMETERS FROM STACK
          JMP       ,U        EXIT TO RETURN ADDRESS
*
*         ARRAY OF MASKS WITH 1 TO 15 ONE BITS RIGHT-JUSTIFIED
*
```

```
MSKARY:
          FDB       %0000000000000001
          FDB       %0000000000000011
          FDB       %0000000000000111
          FFB       %0000000000001111
          FDB       %0000000000011111
          FDB       %0000000000111111
          FDB       %0000000001111111
          FDB       %0000000011111111
          FDB       %0000000111111111
          FDB       %0000001111111111
          FDB       %0000011111111111
          FDB       %0000111111111111
          FDB       %0001111111111111
          FDB       %0011111111111111
          FDB       %0111111111111111

*
*         SAMPLE EXECUTION
*
SC4A:
          LDA       POS             GET LOWEST BIT POSITION
          LDB       NBITS           GET FIELD WIDTH IN BITS
          LDX       VAL             GET DATA
          PSHS      A,B,X           SAVE PARAMETERS IN STACK
          JSR       BFE             EXTRACT BIT FIELD
                                    *RESULT FOR VAL=1234H, NBITS=4
                                    *  POS=4 IS D = 0003H

          BRA       SC4A
*
*DATA
*
VAL       FDB       $1234           DATA
NBITS     FCB       4               FIELD WIDTH IN BITS
POS       FCB       4               LOWEST BIT POSITION
```

4B Bit field insertion (BFI)

Inserts a field of bits into a word. The width of the field and its lowest (starting) bit position are the parameters.

Procedure The program obtains a mask consisting of right-justified 0 bits covering the width of the field. It then shifts the mask and the bit field left to align them with the specified lowest bit position. It logically ANDs the mask and the original data word, thus clearing the required bit positions, and then logically ORs the result with the shifted bit field.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Starting (lowest) bit position in the field (0–15)

Width of the field in bits (0–15)

More significant byte of bit field (value to insert)
Less significant byte of bit field (value to insert)

More significant byte of data
Less significant byte of data

Exit conditions

Result in register D

The result is the original data value with the bit field inserted, starting at the specified lowest bit position

Examples

1. Data: Value = $F67C_{16}$ = 1111011001111100_2
   Lowest bit position = 4
   Number of bits in the field = 8
   Bit field = $008B_{16}$ = 0000000010001011_2

Result: Value with bit field inserted = $F8BC_{16}$ = 1111100010111100_2
The 8-bit field has been inserted into the original value starting at bit 4 (i.e. into bits 4 – 11)

2. Data: Value = $A2D4_{16}$ = 1010001011010100_2
Lowest bit position = 6
Number of bits in the field = 5
Bit field = 0015_{16} = 0000000000010101_2
Result: Value with bit field inserted = $A554_{16}$ = 1010010101010100_2
The 5-bit field has been inserted into the original value starting at bit 6 (i.e. into bits 6 – 10). Those five bits were 01011_2 ($0B_{16}$) and are now 10101_2 (15_{16}).

Registers used A, B, CC, U, X

Execution time 30 × LOWEST BIT POSITION plus 91 cycles overhead. The lowest bit position of the field determines how many times the program must shift the mask and the field left. For example, if the starting position is bit 10, the execution time is

30 × 10 + 91 = 300 + 91 = 391 cycles

Program size 67 bytes (including the table of masks)

Data memory required None

Special cases

1. Attempting to insert a field that would extend beyond the end of the word causes the program to insert only the bits through bit 15. That is, no wraparound is provided. If, for example, the user attempts to insert a 6-bit field starting at bit 14, only 2 bits (bits 14 and 15) are actually replaced.

2. Both the lowest bit position and the length of the bit field are interpreted mod 16. That is, for example, bit position 17 is the same as

bit position 1 and a 20-bit field is the same as a 4-bit field.

3. Attempting to insert a field of zero width causes a return with a result equal to the initial data.

```
*
*
*
*
*    Title:              Bit Field Insertion
*    Name:               BFI
*
*
*
*    Purpose:            Inserts a field of bits which is
*                        normalized to bit 0 into a 16-bit word.
*                        NOTE: IF THE REQUESTED FIELD IS TOO LONG, THEN
*                              ONLY THE BITS THROUGH BIT 15 WILL BE
*                              INSERTED.  FOR EXAMPLE, IF A 4-BIT FIELD
*                              IS TO BE INSERTED STARTING AT BIT 15,
*                              THEN ONLY THE FIRST BIT WILL BE INSERTED
*                              AT BIT 15.
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Bit position at which inserted field will
*                            start (0..15)
*                          Number of bits in the field (1..16)
*                          High byte of value to insert
*                          Low byte of value to insert
*                          High byte of value
*                          Low byte of value
*
*    Exit:               Register D = Value with field inserted
*
*    Registers Used:     A,B,CC,U,X
*
*    Time:               91 cycles overhead plus
*                          (30 * lowest bit position) cycles
*
*    Size:               Program  67 bytes
*
*
*

BFI:
         LDU       ,S        SAVE RETURN ADDRESS
*
*        EXIT WITH DATA AS RESULT IF FIELD WIDTH IS ZERO
*
         LDD       6,S       GET DATA
         TST       3,S       CHECK FIELD WIDTH
         BEQ       EXITBF    BRANCH (EXIT) IF FIELD WIDTH IS ZERO
                             * RESULT IN D IS ORIGINAL DATA
```

```
*
*         USE FIELD WIDTH TO OBTAIN MASK FROM ARRAY
*         MASK HAS A NUMBER OF RIGHT-JUSTIFIED 0 BITS GIVEN
*           BY FIELD WIDTH
*
          LDA       3,S         GET FIELD WIDTH
          DECA                  CONVERT FIELD WIDTH TO ARRAY INDEX
          ANDA      #$0F        MAKE SURE INDEX IS 0 TO 15
          ASLA                  MULTIPLY BY 2 SINCE MASKS ARE WORD-LENGTH
          LEAX      MSKARY,PCR  GET BASE ADDRESS OF MASK ARRAY
          LDX       A,X         GET MASK FROM ARRAY
*
*         SHIFT MASK AND FIELD TO BE INSERTED LEFT TO ALIGN THEM WITH
*           THE FIELD'S LOWEST BIT POSITION
*
          LDA       2,S         GET LOWEST BIT POSITION
          ANDA      #$0F        BE SURE POSITION IS 0 TO 15
          BEQ       INSERT      BRANCH IF POSITION IS 0 AND NO SHIFTING
                                * IS NECESSARY
          STA       ,S          SAVE LOWEST POSITION IN STACK FOR USE
                                * AS COUNTER
          TFR       X,D         MOVE MASK TO REGISTER D FOR SHIFTING
SHFTLP:
          SEC                   FILL MASK WITH ONES
          ROLB                  SHIFT LOW BYTE OF MASK LEFT, PUTTING A
                                * 1 IN BIT 0
          ROLA                  SHIFT HIGH BYTE OF MASK LEFT
          ASL       5,S         SHIFT LOW BYTE OF INSERT VALUE LEFT
          ROL       4,S         SHIFT HIGH BYTE OF INSERT VALUE LEFT
          DEC       ,S
          BNE       SHFTLP      CONTINUE UNTIL INSERT VALUE'S LEAST
                                * SIGNIFICANT BIT IS IN LOWEST BIT
                                * POSITION
*
*         USE MASK TO CLEAR FIELD, THEN OR IN INSERT VALUE
*
INSERT:
          ANDA      6,S         AND HIGH BYTE OF VALUE WITH MASK
          ANDB      7,S         AND LOW BYTE OF VALUE WITH MASK
          ORA       4,S         OR IN HIGH BYTE OF INSERT VALUE
          ORB       5,S         OR IN LOW BYTE OF INSERT VALUE
*
*         REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITBF:
          LEAS      8,S         REMOVE PARAMETERS FROM STACK
          JMP       ,U          EXIT TO RETURN ADDRESS

*
*         MASK ARRAY USED TO CLEAR THE BIT FIELD INITIALLY
*         HAS 0 BITS RIGHT-JUSTIFIED IN 1 TO 15 BIT POSITIONS
*
MSKARY:
          FDB       %1111111111111110
          FDB       %1111111111111100
          FDB       %1111111111111000
```

```
          FDB       %1111111111110000
          FDB       %1111111111100000
          FDB       %1111111111000000
          FDB       %1111111110000000
          FDB       %1111111100000000
          FDB       %1111111000000000
          FDB       %1111110000000000
          FDB       %1111100000000000
          FDB       %1111000000000000
          FDB       %1110000000000000
          FDB       %1100000000000000
          FDB       %1000000000000000

*
*
*         SAMPLE EXECUTION
*
*

SC4B:
          LDA       POS             GET LOWEST BIT POSITION OF FIELD
          LDB       NBITS           GET FIELD WIDTH IN BITS
          LDX       VALINS          GET VALUE TO INSERT
          LDY       VAL             GET VALUE
          PSHS      A,B,X,Y         SAVE PARAMETERS IN STACK
          JSR       BFI             INSERT BIT FIELD
                                    *RESULT FOR VAL=1234H, VALINS=0EH,
                                    *  NBITS = 4, POS = 0CH IS
                                    *  REGISTER D = E234H
          BRA       SC4B

*
*DATA
*
VAL       FDB       $1234           DATA VALUE
VALINS    FDB       $000E           VALUE TO INSERT
NBITS     FCB       4               FIELD WIDTH IN BITS
POS       FCB       $0C             LOWEST BIT POSITION IN FIELD

          END
```

4C Multiple-precision arithmetic shift right (MPASR)

Shifts a multi-byte operand right arithmetically by a specified number of bit positions. The length of the operand (in bytes) is 255 or less. Sets the Carry flag from the last bit shifted out of the rightmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure The program obtains the sign bit from the most significant byte, saves that bit in the Carry, and then rotates the entire operand right 1 bit, starting with the most significant byte. It repeats the operation for the specified number of shifts.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Number of shifts (bit positions)

Length of the operand in bytes

More significant byte of base address of operand (address of its least significant byte)
Less significant byte of base address of operand (address of its least significant byte)

Exit conditions

Operand shifted right arithmetically by the specified number of bit positions. The original sign bit is extended to the right.

The Carry flag is set from the last bit shifted out of the rightmost bit position. It is cleared if either the number of shifts or the length of the operand is 0.

Examples

1. Data: Length of operand (in bytes) = 8

Operand = $85A4C719FE06741E_{16}$
Number of shifts = 4

Result: Shifted operand = $F85A4C719FE06741_{16}$.
This is the original operand shifted right 4 bits arithmetically. The four most significant bits thus all take on the value of the original sign bit (1).
Carry = 1, since the last bit shifted from the rightmost bit position was 1.

2. Data: Length of operand (in bytes) = 4
Operand = $3F6A42D3_{16}$
Number of shifts = 3

Result: Shifted operand = $07ED485A_{16}$.
This is the original operand shifted right 3 bits arithmetically. The three most significant bits thus all take on the value of the original sign bit (0).
Carry = 0, since the last bit shifted from the rightmost bit position was 0.

Registers used A, B, CC, U, X

Execution time NUMBER OF SHIFTS × (28 + 13 × LENGTH OF OPERAND IN BYTES) + 50 cycles.

If, for example, NUMBER OF SHIFTS = 6 and LENGTH OF OPERAND IN BYTES = 8, the execution time is

$6 \times (28 + 13 \times 8) + 50 = 6 \times 132 + 50 = 842$ cycles

Program size 39 bytes

Data memory required None

Special cases

1. If the length of the operand is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

2. If the number of shifts is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

```
*
*
*
*
*    Title:              Multiple-Precision Arithmetic Shift Right
*    Name:               MPASR
*
*
*
*    Purpose:            Arithmetic shift right a multi-byte operand
*                        N bits.
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Number of bits to shift
*                          Length of the operand in bytes
*                          High byte of operand base address
*                          Low byte of operand base address
*
*                          The operand is stored with ARRAY[0] as its
*                          least significant byte and ARRAY[LENGTH-1]
*                          as its most significant byte
*
*    Exit:               Operand shifted right with the most
*                        significant bit propagated.
*                        Carry := Last bit shifted from least
*                                 significant position.
*
*    Registers Used:     A,B,CC,U,X
*
*    Time:               50 cycles overhead plus
*                          (13 * length) + 28 cycles per shift
*
*    Size:               Program  39 bytes
*
*
*

MPASR:
          LDU       ,S        SAVE RETURN ADDRESS
*
*         EXIT IF LENGTH OF OPERAND OR NUMBER OF BITS TO SHIFT
*           IS ZERO.  CARRY IS CLEARED IN EITHER CASE
*
          CLC                 CLEAR CARRY INITIALLY
          LDA       2,S       GET NUMBER OF BITS TO SHIFT
          BEQ       EXITAS    EXIT IF NUMBER OF BITS TO SHIFT IS ZERO
          LDA       3,S       GET LENGTH OF OPERAND
          BEQ       EXITAS    EXIT IF LENGTH OF OPERAND IS ZERO
*
*         SAVE POINTER TO MOST SIGNIFICANT BYTE OF OPERAND
*
          DECA                OFFSET OF MOST SIGNIFICANT BYTE =
                              * LENGTH OF OPERAND - 1
          LDX       4,S       GET BASE ADDRESS OF OPERAND
```

```
          LEAX      A,X         POINT TO MOST SIGNIFICANT BYTE
          STX       ,S          SAVE POINTER TO MOST SIGNIFICANT BYTE
*
*         SHIFT ENTIRE OPERAND RIGHT ONE BIT ARITHMETICALLY
*         USE SIGN OF MOST SIGNIFICANT BYTE AS INITIAL CARRY INPUT
*           TO PRODUCE ARITHMETIC SHIFT
*
ASRLP:
          LDX       ,S          POINT TO MOST SIGNIFICANT BYTE
          LDA       ,X+         GET MOST SIGNIFICANT BYTE
          ASLA                  SHIFT BIT 7 TO CARRY FOR SIGN EXTENSION
          LDB       3,S         GET LENGTH OF OPERAND IN BYTES
*
*         SHIFT EACH BYTE OF OPERAND RIGHT ONE BIT
*         START WITH MOST SIGNIFICANT BYTE
*
ASRLP1:
          ROR       ,-X         ROTATE NEXT BYTE RIGHT
          DECB
          BNE       ASRLP1      CONTINUE THROUGH ALL BYTES
*
*         COUNT NUMBER OF SHIFTS
*
          DEC       2,S         DECREMENT NUMBER OF SHIFTS
          BNE       ASRLP       CONTINUE UNTIL DONE
*
*         REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITAS:
          LEAS      6,S         REMOVE PARAMETERS FROM STACK
          JMP       ,U          EXIT TO RETURN ADDRESS

*
*         SAMPLE EXECUTION
*
*

SC4C:
          LDA       SHIFTS      GET NUMBER OF SHIFTS
          LDB       #SZAY       GET LENGTH OF OPERAND IN BYTES
          LDX       AYADR       GET BASE ADDRESS OF OPERAND
          PSHS      A,B,X       SAVE PARAMETERS IN STACK
          JSR       MPASR       ARITHMETIC SHIFT RIGHT
                                *RESULT OF SHIFTING AY=EDCBA087654321H
                                *4 BITS IS AY=FEDCBA98765432H, C=0
                                *  IN MEMORY AY   = 032H
                                *            AY+1 = 054H
                                *            AY+2 = 076H
                                *            AY+3 = 098H
                                *            AY+4 = 0BAH
                                *            AY+5 = 0DCH
                                *            AY+6 = 0FEH
          BRA       SC4C

*
*DATA SECTION
```

```
*
SZAY      EQU       7          LENGTH OF OPERAND IN BYTES
SHIFTS:   FCB       4          NUMBER OF SHIFTS
AYADR:    FDB       AY         BASE ADDRESS OF OPERAND
AY:       FCB       $21,$43,$65,$87,$A9,$CB,$ED

          END
```

4D Multiple-precision logical shift left (MPLSL)

Shifts a multi-byte operand left logically by a specified number of bit positions. The length of the operand (in bytes) is 255 or less. Sets the Carry flag from the last bit shifted out of the leftmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure The program clears the Carry initially (to fill with a 0 bit) and then shifts the entire operand left 1 bit, starting with the least significant byte. It repeats the operation for the specified number of shifts.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Number of shifts (bit positions)

Length of the operand in bytes

More significant byte of base address of operand (address of its least significant byte)

Less significant byte of base address of operand (address of its least significant byte)

Exit conditions

Operand shifted left logically by the specified number of bit positions. The least significant bit positions are filled with 0s.

The Carry flag is set from the last bit shifted out of the leftmost bit position. It is cleared if either the number of shifts or the length of the operand is 0.

Examples

1. Data: Length of operand (in bytes) = 8
   Operand = $85A4C719FE06741E_{16}$
   Number of shifts = 4

Result: Shifted operand = $5A4C719FE06741E0_{16}$.
This is the original operand shifted left 4 bits logically. The four least significant bits are all cleared.
Carry = 0, since the last bit shifted from the leftmost bit position was 0.

2. Data: Length of operand (in bytes) = 4
Operand = $3F6A42D3_{16}$
Number of shifts = 3
Result: Shifted operand = $FB521698_{16}$.
This is the original operand shifted left 3 bits logically. The three least significant bits are all cleared.
Carry = 1, since the last bit shifted from the leftmost bit position was 1.

Registers used A, B, CC, U, X

Execution time NUMBER OF SHIFTS × (24 + 13 × LENGTH OF OPERAND IN BYTES) + 32 cycles.

If for example, NUMBER OF SHIFTS = 6 and LENGTH OF OPERAND IN BYTES = 8, the execution time is

$6 \times (24 + 13 \times 8) + 32 = 6 \times 128 + 32 = 800$ cycles

Program size 31 bytes

Data memory required None

Special cases

1. If the length of the operand is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.
2. If the number of shifts is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

```
*
*
*
*
*    Title:              Multiple-Precision Logical Shift Left
*    Name:               MPLSL
*
*
*
*    Purpose:            Logical shift left a multi-byte operand
*                        N bits.
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Number of bits to shift
*                          Length of the operand in bytes
*                          High byte of operand base address
*                          Low byte of operand base address
*
*                          The operand is stored with ARRAY[0] as its
*                          least significant byte and ARRAY[LENGTH-1]
*                          as its most significant byte
*
*    Exit:               Operand shifted left filling the least
*                        significant bits with zeros.
*                        CARRY := Last bit shifted from most
*                                 significant position
*
*    Registers Used:     A,B,CC,U,X
*
*    Time:               32 cycles overhead plus
*                          ((13 * length) + 24) cycles per shift

*
*    Size:               Program  31 bytes
*
*
*

MPLSL:
          LDU       ,S          SAVE RETURN ADDRESS
*
*         EXIT IF LENGTH OF OPERAND OR NUMBER OF BITS TO SHIFT
*           IS ZERO.  CARRY IS CLEARED IN EITHER CASE
*
          CLC                   CLEAR CARRY
          LDA       2,S         GET NUMBER OF BITS TO SHIFT
          BEQ       EXITLS      EXIT IF NUMBER OF BITS TO SHIFT IS ZERO
          LDA       3,S         GET LENGTH OF OPERAND
          BEQ       EXITLS      EXIT IF LENGTH OF OPERAND IS ZERO
*
*         SHIFT ENTIRE OPERAND LEFT ONE BIT LOGICALLY
*         USE ZERO AS INITIAL CARRY INPUT TO PRODUCE LOGICAL SHIFT
*
LSLLP:
```

```
          LDX       4,S       POINT TO LEAST SIGNIFICANT BYTE
          LDB       3,S       GET LENGTH OF OPERAND IN BYTES
          CLC                 CLEAR CARRY TO FILL WITH ZEROS
*
*         SHIFT EACH BYTE OF OPERAND LEFT ONE BIT
*         START WITH LEAST SIGNIFICANT BYTE
*
LSLLP1:
          ROL       ,X+       SHIFT NEXT BYTE LEFT
          DECB
          BNE       LSLLP1    CONTINUE THROUGH ALL BYTES
*
*         COUNT NUMBER OF SHIFTS
*
          DEC       2,S       DECREMENT NUMBER OF SHIFTS
          BNE       LSLLP     CONTINUE UNTIL DONE
*
*         REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITLSL:
          LEAS      6,S       REMOVE PARAMETERS FROM STACK
          JMP       ,U        EXIT TO RETURN ADDRESS

*
*
*         SAMPLE EXECUTION
*
*

SC4D:
          LDA       SHIFTS    GET NUMBER OF SHIFTS
          LDB       #SZAY     GET LENGTH OF OPERAND IN BYTES
          LDX       AYADR     GET BASE ADDRESS OF OPERAND
          PSHS      A,B,X     SAVE PARAMETERS IN STACK
          JSR       MPLSL     LOGICAL SHIFT LEFT
                              *RESULT OF SHIFTING AY=EDCBA087654321H
                              *4 BITS IS AY=DCBA9876543210H, C=0
                              *  IN MEMORY AY   = 010H
                              *            AY+1 = 032H
                              *            AY+2 = 054H
                              *            AY+3 = 076H
                              *            AY+4 = 098H
                              *            AY+5 = 0BAH
                              *            AY+6 = 0DCH
          BRA       SC4D

*
*DATA SECTION
*
SZAY      EQU       7         LENGTH OF OPERAND IN BYTES
SHIFTS:   FCB       4         NUMBER OF SHIFTS
AYADR:    FDB       AY        BASE ADDRESS OF OPERAND
AY:       FCB       $21,$43,$65,$87,$A9,$CB,$ED

          END
```

4E Multiple-precision logical shift right (MPLSR)

Shifts a multi-byte operand right logically by a specified number of bit positions. The length of the operand (in bytes) is 255 or less. Sets the Carry flag from the last bit shifted out of the rightmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure The program clears the Carry initially (to fill with a 0 bit) and then shifts the entire operand right 1 bit, starting with the most significant byte. It repeats the operation for the specified number of shifts.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Number of shifts (bit positions)

Length of the operand in bytes

More significant byte of base address of operand (address of its least significant byte)
Less significant byte of base address of operand (address of its least significant byte)

Exit conditions

Operand shifted right logically by the specified number of bit positions. The most significant bit positions are filled with 0s.

The Carry flag is set from the last bit shifted out of the rightmost bit position. It is cleared if either the number of shifts or the length of the operand is 0.

Examples

1. Data: Length of operand (in bytes) = 8

Operand = $85A4C719FE06741E_{16}$
Number of shifts = 4
Result: Shifted operand = $085A4C719FE06741_{16}$.
This is the original operand shifted right 4 bits logically. The four most significant bits are all cleared.
Carry = 1, since the last bit shifted from the rightmost bit position was 1.

2. Data: Length of operand (in bytes) = 4
Operand = $3F6A42D3_{16}$
Number of shifts = 3
Result: Shifted operand = $07ED485A_{16}$.
This is the original operand shifted right 3 bits logically. The three most significant bits are all cleared.
Carry = 0, since the last bit shifted from the rightmost bit position was 0.

Registers used A, B, CC, X, U

Execution time NUMBER OF SHIFTS × (23 + 13 × LENGTH OF OPERAND IN BYTES) + 48 cycles.

If, for example, NUMBER OF SHIFTS = 6 and LENGTH OF OPERAND IN BYTES = 8, the execution time is

$6 \times (23 + 13 \times 8) + 48 = 6 \times 127 + 48 = 810$ cycles

Program size 37 bytes

Data memory required None

Special cases

1. If the length of the operand is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.
2. If the number of shifts is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

```
* *
*
*
*    Title:              Multiple-Precision Logical Shift Right
*    Name:               MPLSR
*
*
*
*    Purpose:            Logical shift right a multi-byte operand
*                        N bits.
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Number of bits to shift
*                          Length of the operand in bytes
*                          High byte of operand base address
*                          Low byte of operand base address
*
*                          The operand is stored with ARRAY[0] as its
*                          least significant byte and ARRAY[LENGTH-1]
*                          as its most significant byte
*
*    Exit:               Operand shifted right filling the most
*                        significant bits with zeros.
*                        Carry := Last bit shifted from least
*                                 significant position.
*
*    Registers Used:     A,B,CC,U,X
*
*    Time:               48 cycles overhead plus
*                          ((13 * length) + 23) cycles per shift
*
*    Size:               Program  37 bytes
*
*
*

MPLSR:
          LDU       ,S          SAVE RETURN ADDRESS
*
*         EXIT IF LENGTH OF OPERAND OR NUMBER OF BITS TO SHIFT
*           IS ZERO.  CARRY IS CLEARED IN EITHER CASE
*
          CLC                   CLEAR CARRY INITIALLY
          LDA       2,S         GET NUMBER OF BITS TO SHIFT
          BEQ       EXITLS      EXIT IF NUMBER OF BITS TO SHIFT IS ZERO
          LDA       3,S         GET LENGTH OF OPERAND
          BEQ       EXITLS      EXIT IF LENGTH OF OPERAND IS ZERO
*
*         SAVE POINTER TO END OF OPERAND
*
          LDX       4,S         GET BASE ADDRESS OF OPERAND
          LEAX      A,X         CALCULATE ENDING ADDRESS OF OPERAND
          STX       ,S          SAVE ENDING ADDRESS OF OPERAND
*
```

```
*         SHIFT ENTIRE OPERAND RIGHT ONE BIT LOGICALLY
*         USE ZERO AS INITIAL CARRY INPUT TO PRODUCE LOGICAL SHIFT
*
LSRLP:
          LDX       ,S          POINT TO END OF OPERAND
          LDB       3,S         GET LENGTH OF OPERAND IN BYTES
          CLC                   CLEAR CARRY TO FILL WITH ZEROS
*
*         SHIFT EACH BYTE OF OPERAND RIGHT ONE BIT
*         START WITH MOST SIGNIFICANT BYTE
*
LSRLP1:
          ROR       ,-X         SHIFT NEXT BYTE RIGHT
          DECB
          BNE       LSRLP1      CONTINUE THROUGH ALL BYTES
*
*         COUNT NUMBER OF SHIFTS
*
          DEC       2,S         DECREMENT NUMBER OF SHIFTS
          BNE       LSRLP       CONTINUE UNTIL DONE
*
*         REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITLS:
          LEAS      6,S         REMOVE PARAMETERS FROM STACK
          JMP       ,U          EXIT TO RETURN ADDRESS
*
*
*         SAMPLE EXECUTION
*
SC4E:
          LDA       SHIFTS      GET NUMBER OF SHIFTS
          LDB       #SZAY       GET LENGTH OF OPERAND IN BYTES
          LDX       AYADR       GET BASE ADDRESS OF OPERAND
          PSHS      A,B,X       SAVE PARAMETERS IN STACK
          JSR       MPLSR       LOGICAL SHIFT RIGHT
                                *RESULT OF SHIFTING AY=EDCBA087654321H
                                *4 BITS IS AY=0EDCBA98765432H, C=0
                                *  IN MEMORY AY    = 032H
                                *             AY+1 = 054H
                                *             AY+2 = 076H
                                *             AY+3 = 098H
                                *             AY+4 = 0BAH
                                *             AY+5 = 0DCH
                                *             AY+6 = 00EH
          BRA       SC4E

*
*DATA SECTION
*
SZAY      EQU       7           LENGTH OF OPERAND IN BYTES
SHIFTS:   FCB       4           NUMBER OF SHIFTS
AYADR:    FDB       AY          BASE ADDRESS OF OPERAND
AY:       FCB       $21,$43,$65,$87,$A9,$CB,$ED

          END
```

4F Multiple-precision rotate right (MPRR)

Rotates a multi-byte operand right by a specified number of bit positions as if the most significant bit and least significant bit were connected. The length of the operand (in bytes) is 255 or less. Sets the Carry flag from the last bit shifted out of the rightmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure The program shifts bit 0 of the least significant byte of the operand to the Carry flag and then shifts the entire operand right 1 bit, starting with the most significant byte. It repeats the operation for the specified number of rotates.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Number of rotates (bit positions)

Length of the operand in bytes

More significant byte of base address of operand (address of its least significant byte)
Less significant byte of base address of operand (address of its least significant byte)

Exit conditions

Operand rotated right by the specified number of bit positions. The most significant bit positions are filled from the least significant bit positions.

The Carry flag is set from the last bit shifted out of the rightmost bit position. It is cleared if either the number of shifts or the length of the operand is 0.

Examples

1. Data: Length of operand (in bytes) = 8
Operand = $85A4C719FE06741E_{16}$
Number of rotates = 4
Result: Shifted operand = $E85A4C719FE06741_{16}$.
This is the original operand rotated right 4 bits. The four most significant bits are equivalent to the original four least significant bits.
Carry = 1, since the last bit shifted from the rightmost bit position was 1.

2. Data: Length of operand (in bytes) = 4
Operand = $3F6A42D3_{16}$
Number of rotates = 3
Result: Shifted operand = $67ED485A_{16}$.
This is the original operand rotated right 3 bits. The three most significant bits (011) are equivalent to the original three least significant bits.
Carry = 0, since the last bit shifted from the rightmost bit position was 0.

Registers used A, B, CC, U, X

Execution time NUMBER OF ROTATES × (32 + 13 × LENGTH OF OPERAND IN BYTES) + 48 cycles.
If, for example, NUMBER OF ROTATES = 6 and LENGTH OF OPERAND IN BYTES = 8, the execution time is

$6 \times (32 + 13 \times 8) + 48 = 6 \times 136 + 48 = 864$ cycles

Program size 40 bytes

Data memory required None

Special cases

1. If the length of the operand is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

2. If the number of rotates is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

```
*
*
*
*
*    Title:              Multiple-Precision Rotate Right
*    Name:               MPRR
*
*
*
*    Purpose:            Rotate right a multi-byte operand
*                        N bits.
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Number of bits to rotate
*                          Length of the operand in bytes
*                          High byte of operand base address
*                          Low byte of operand base address
*
*                          The operand is stored with ARRAY[0] as its
*                          least significant byte and ARRAY[LENGTH-1]
*                          as its most significant byte
*
*                        Operand rotated right
*                        Carry := Last bit shifted from least
*                                 significant position.
*
*    Registers Used:     A,B,CC,U,X
*
*    Time:               48 cycles overhead plus
*                          ((13 * length) + 32) cycles per shift
*
*    Size:               Program  40 bytes
*
*
*

MPRR:
          LDU       ,S          SAVE RETURN ADDRESS
*
*         EXIT IF LENGTH OF OPERAND OR NUMBER OF BITS TO ROTATE
*           IS ZERO.  CARRY IS CLEARED IN EITHER CASE
*
          CLC                   CLEAR CARRY INITIALLY
          LDA       2,S         GET NUMBER OF BITS TO ROTATE
          BEQ       EXITRR      EXIT IF NUMBER OF BITS TO ROTATE IS ZERO
          LDA       3,S         GET LENGTH OF OPERAND
          BEQ       EXITRR      EXIT IF LENGTH OF OPERAND IS ZERO
*
*         SAVE POINTER TO END OF OPERAND
*
```

```
            LDX         4,S         GET BASE ADDRESS OF OPERAND
            LEAX        A,X         POINT TO END OF OPERAND
            STX         ,S          SAVE POINTER TO END OF OPERAND
*
*           ROTATE ENTIRE OPERAND RIGHT ONE BIT
*           USE PREVIOUS LEAST SIGNIFICANT BIT AS INITIAL CARRY INPUT
*             TO PRODUCE ROTATION
*
RRLP:
            LDX         4,S         POINT TO LEAST SIGNIFICANT BYTE
            LDA         ,X          GET LEAST SIGNIFICANT BYTE
            LSRA                    SHIFT BIT 0 TO CARRY FOR USE IN ROTATION
            LDB         3,S         GET LENGTH OF OPERAND IN BYTES
            LDX         ,S          POINT TO END OF OPERAND
*
*           SHIFT EACH BYTE OF OPERAND RIGHT ONE BIT
*           START WITH MOST SIGNIFICANT BYTE
*
RRLP1:
            ROR         ,-X         SHIFT NEXT BYTE RIGHT
            DECB
            BNE         RRLP1       CONTINUE THROUGH ALL BYTES
*
*           COUNT NUMBER OF ROTATES
*
            DEC         2,S         DECREMENT NUMBER OF ROTATES
            BNE         RRLP        CONTINUE UNTIL DONE
*
*           REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITRR:
            LEAS        6,S         REMOVE PARAMETERS FROM STACK
            JMP         ,U          EXIT TO RETURN ADDRESS
            RTS

*
*
*           SAMPLE EXECUTION
*
*

SC4F:
            LDA         ROTATS      GET NUMBER OF ROTATES
            LDB         #SZAY       GET LENGTH OF OPERAND IN BYTES
            LDX         AYADR       GET BASE ADDRESS OF OPERAND
            PSHS        A,B,X       SAVE PARAMETERS IN STACK
            JSR         MPRR        ROTATE RIGHT
                                    *RESULT OF ROTATING AY=EDCBA087654321H
                                    *4 BITS IS AY=1EDCBA98765432H, C=0
                                    *  IN MEMORY AY   = 032H
                                    *            AY+1 = 054H
                                    *            AY+2 = 076H
                                    *            AY+3 = 098H
                                    *            AY+4 = 0BAH
                                    *            AY+5 = 0DCH
                                    *            AY+6 = 01EH
```

```
          BRA       SC4F

*
*DATA SECTION
*
SZAY      EQU       7          LENGTH OF OPERAND IN BYTES
ROTATS:   FCB       4          NUMBER OF ROTATES
AYADR:    FDB       AY         BASE ADDRESS OF OPERAND
AY:       FCB       $21,$43,$65,$87,$A9,$CB,$ED
          END
```

4G Multiple-precision rotate left (MPRL)

Rotates a multi-byte operand left by a specified number of bit positions as if the most significant bit and least significant bit were connected. The length of the number (in bytes) is 255 or less. Sets the Carry flag from the last bit shifted out of the leftmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure The program shifts bit 7 of the most significant byte of the operand to the Carry flag. It then shifts the entire operand left 1 bit, starting with the least significant byte. It repeats the operation for the specified number of rotates.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Number of rotates (bit positions)

Length of the operand in bytes

More significant byte of base address of operand (address of its least significant byte)
Less significant byte of base address of operand (address of its least significant byte)

Exit conditions

Operand rotated left by the specified number of bit positions (the least significant bit positions are filled from the most significant bit positions).

The Carry flag is set from the last bit shifted out of the leftmost bit position. It is cleared if either the number of shifts or the length of the operand is 0.

Examples

1. Data: Length of operand (in bytes) = 8

Operand = $85A4C719FE06741E_{16}$
Number of rotates = 4

Result: Shifted operand = $5A4C719FE06741E8_{16}$.
This is the original operand rotated left 4 bits. The four least significant bits are equivalent to the original four most significant bits.
Carry = 0, since the last bit shifted from the leftmost bit position was 0.

2. Data: Length of operand (in bytes) = 4
Operand = $3F6A42D3_{16}$
Number of rotates = 3

Result: Shifted operand = $FB521699_{16}$.
This is the original operand rotated left 3 bits. The three least significant bits (001) are equivalent to the original three most significant bits.
Carry = 1, since the last bit shifted from the leftmost bit position was 0.

Registers used A, B, CC, U, X

Execution time NUMBER OF ROTATES × (34 + 13 × LENGTH OF OPERAND IN BYTES) + 50 cycles.

If, for example, NUMBER OF ROTATES = 6 and LENGTH OF OPERAND IN BYTES = 8, the execution time is

$6 \times (34 + 13 \times 8) + 50 = 6 \times 138 + 50 = 878$ cycles

Program size 41 bytes

Data memory required None

Special cases

1. If the length of the operand is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

2. If the number of rotates is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

```
*
*
*
*
*    Title:              Multiple-Precision Rotate Left
*    Name:               MPRL
*
*
*
*    Purpose:            Rotate left a multi-byte operand
*                        N bits.
*
*    Entry:              TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          Number of bits to rotate
*                          Length of the operand in bytes
*                          High byte of operand base address
*                          Low byte of operand base address
*
*                          The operand is stored with ARRAY[0] as its
*                          least significant byte and ARRAY[LENGTH-1]
*                          as its most significant byte
*
*    Exit:               Number rotated left
*                        Carry := Last bit shifted from the most
*                                 significant position.
*
*    Registers Used:     A,B,CC,U,X
*
*    Time:               50 cycles overhead plus
*                          ((13 * length) + 34) cycles per shift
*
*    Size:               Program  41 bytes
*
*
*

MPRL:
          LDU       ,S          SAVE RETURN ADDRESS
*
*         EXIT IF LENGTH OF OPERAND OR NUMBER OF BITS TO ROTATE
*           IS ZERO.  CARRY IS CLEARED IN EITHER CASE
*
          CLC                   CLEAR CARRY
          LDA       2,S         GET NUMBER OF BITS TO ROTATE
          BEQ       EXITRL      EXIT IF NUMBER OF BITS TO ROTATE IS ZERO
          LDA       3,S         GET LENGTH OF OPERAND
          BEQ       EXITRL      EXIT IF LENGTH OF OPERAND IS ZERO
*
*         SAVE POINTER TO MOST SIGNIFICANT BYTE OF OPERAND
*
          DECA                  OFFSET OF MOST SIGNIFICANT BYTE =
                                * LENGTH OF OPERAND - 1
          LDX       4,S         GET BASE ADDRESS OF OPERAND
          LEAX      A,X         POINT TO MOST SIGNIFICANT BYTE
```

```
          STX       ,S          SAVE POINTER TO MOST SIGNIFICANT BYTE
*
*         ROTATE ENTIRE OPERAND LEFT ONE BIT
*         USE PREVIOUS MOST SIGNIFICANT BIT AS INITIAL CARRY INPUT
*           TO PRODUCE ROTATION
*
RLLP:
          LDX       ,S          POINT TO MOST SIGNIFICANT BYTE
          LDA       ,X+         GET MOST SIGNIFICANT BYTE
          ASLA                  SHIFT BIT 7 TO CARRY FOR USE IN ROTATION
          LDB       3,S         GET LENGTH OF OPERAND IN BYTES
          LDX       4,S         GET BASE ADDRESS OF OPERAND
*
*         SHIFT EACH BYTE OF OPERAND RIGHT ONE BIT
*         START WITH LEAST SIGNIFICANT BYTE
*
RLLP1:
          ROL       ,-X         SHIFT NEXT BYTE LEFT
          DECB
          BNE       RLLP1       CONTINUE THROUGH ALL BYTES
*
*         COUNT NUMBER OF ROTATES
*
          DEC       2,S         DECREMENT NUMBER OF ROTATES
          BNE       RRLP        CONTINUE UNTIL DONE
*
*         REMOVE PARAMETERS FROM STACK AND EXIT
*
EXITRL:
          LEAS      6,S         REMOVE PARAMETERS FROM STACK
          JMP       ,U          EXIT TO RETURN ADDRESS

*
*
*         SAMPLE EXECUTION
*
*

SC4G:
          LDA       ROTATS      GET NUMBER OF ROTATES
          LDB       #SZAY       GET LENGTH OF OPERAND IN BYTES
          LDX       AYADR       GET BASE ADDRESS OF OPERAND
          PSHS      A,B,X       SAVE PARAMETERS IN STACK
          JSR       MPRL        ROTATE LEFT
                                *RESULT OF ROTATING AY=EDCBA087654321H
                                *4 BITS IS AY=DCBA987654321EH, C=0
                                *  IN MEMORY AY   = 01EH
                                *            AY+1 = 032H
                                *            AY+2 = 054H
                                *            AY+3 = 076H
                                *            AY+4 = 098H
                                *            AY+5 = 0BAH
                                *            AY+6 = 0DCH
          BRA       SC4G

*
```

```
*DATA SECTION
*
SZAY      EQU       7          LENGTH OF OPERAND IN BYTES
ROTATS:   FCB       4          NUMBER OF ROTATES
AYADR:    FDB       AY         BASE ADDRESS OF OPERAND
AY:       FCB       $21,$43,$65,$87,$A9,$CB,$ED

          END
```

5 String manipulation

5A String compare (STRCMP)

Compares two strings and sets the Carry and Zero flags accordingly. Sets the Zero flag to 1 if the strings are identical and to 0 otherwise. Sets the Carry flag to 1 if the string with the base address higher in the stack (string 2) is larger than the other string (string 1), and to 0 otherwise. Each string consists of at most 256 bytes, including an initial byte containing the length. If the two strings are identical through the length of the shorter, the longer string is considered to be larger.

Procedure The program first determines which string is shorter. It then compares the strings one byte at a time through the length of the shorter. It exits with the flags set if it finds corresponding bytes that differ. If the strings are the same through the length of the shorter, the program sets the flags by comparing the lengths.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of base address of string 2
Less significant byte of base address of string 2

More significant byte of base address of string 1
Less significant byte of base address of string 1

Exit conditions

Flags set as if string 2 had been subtracted from string 1. If the strings are the same through the length of the shorter, the flags are set as if the length of string 2 had been subtracted from the length of string 1.

Zero flag = 1 if the strings are identical, 0 if they are not identical.

Carry flag = 1 if string 2 is larger than string 1, 0 if they are identical or string 1 is larger. If the strings are the same through the length of the shorter, the longer one is considered to be larger.

Examples

1. Data: String 1 = 05'PRINT' (05 is the length of the string)
String 2 = 03'END' (03 is the length of the string)
Result: Zero flag = 0 (strings are not identical)
Carry flag = 0 (string 2 is not larger than string 1)

2. Data: String 1 = 05'PRINT' (05 is the length of the string)
String 2 = 02'PR' (02 is the length of the string)
Result: Zero flag = 0 (strings are not identical)
Carry flag = 0 (string 2 is not larger than string 1)

The longer string (string 1) is considered to be larger. To determine whether string 2 is an abbreviation of string 1, use Subroutine 5C (Find the position of a substring). String 2 is an abbreviation if it is part of string 1 and starts at the first character.

3. Data: String 1 = 05'PRINT' (05 is the length of the string)
String 2 = 06'SYSTEM' (06 is the length of the string)
Result: Zero flag = 0 (strings are not identical)
Carry flag = 1 (string 2 is larger than string 1)

We are assuming here that the strings consist of ASCII characters. Note that the initial length byte is a hexadecimal number, not a character. We have represented this byte as two hexadecimal digits in front of the string; the string itself is surrounded by single quotation marks.

This routine treats spaces like other characters. Assuming ASCII strings, the routine will, for example, find that SPRINGMAID is larger

than SPRING MAID, since an ASCII M ($4D_{16}$) is larger than an ASCII space (20_{16}).

Note that this routine will not order strings alphabetically as defined in common uses such as indexes and telephone directories. Instead, it uses the ASCII character order shown in Appendix 3. Note, in particular, that:

1. Spaces precede all other printing characters.
2. Periods, commas, and dashes precede numbers.
3. Numbers precede letters.
4. Capital letters precede lower-case letters.

This ordering produces such non-standard results as the following:

1. 9TH AVENUE SCHOOL would precede CAPITAL CITY SCHOOL (or, in fact, any string starting with a letter). 9TH AVENUE will not be treated as if it started with the letter N.
2. EZ8 MOTEL would precede East Street Motel since a capital Z precedes a lower-case a.
3. NEW YORK would precede NEWARK or NEWCASTLE since a space precedes any letter.

Registers used A, B, CC, U, X

Execution time

1. If the strings are not identical through the length of the shorter, the execution time is approximately

$45 + 20 \times$ NUMBER OF CHARACTERS COMPARED

If, for example, the routine compares five characters before finding a disparity, the execution time is approximately

$45 + 20 \times 5 = 45 + 100 = 145$ cycles

2. If the strings are identical through the length of the shorter, the execution time is approximately

$66 + 20 \times$ LENGTH OF SHORTER STRING

If, for example, the shorter string is 8 bytes long, the execution time is

$66 + 20 \times 8 = 66 + 160 = 226$ cycles

Program size 36 bytes

Data memory required None

Special case If either string (but not both) has a 0 length, the program returns with the flags set as though the other string were larger. If both strings have 0 length, they are considered to be equal.

```
*
*       Title           String Compare
*       Name:           STRCMP
*

*       Purpose:        Compare 2 strings and return C and Z flags set
*                       or cleared.
*
*       Entry:          TOP OF STACK
*                         High byte of return address
*                         Low byte of return address
*                         High byte of string 2 address
*                         Low byte of string 2 address
*                         High byte of string 1 address
*                         Low byte of string 1 address
*
*                         Each string consists of a length byte
*                         followed by a maximum of 255 characters.
*
*       Exit:           IF string 1 = string 2 THEN
*                         Z=1,C=0
*                       IF string 1 > string 2 THEN
*                         Z=0,C=0
*                       IF string 1 < string 2 THEN
*                         Z=0,C=1
*
*       Registers Used: A,B,CC,U,X
*
*       Time:           45 cycles overhead plus 20 cycles per byte plus
*                       21 cycles if the strings are identical through
*                       the length of the shorter one.
*
*       Size:           Program 36 bytes
*
*
STRCMP:
          *
          *DETERMINE WHICH STRING IS SHORTER
          *LENGTH OF SHORTER = NUMBER OF BYTES TO COMPARE
```

```
          *
          LDX       4,S         GET BASE ADDRESS OF STRING 1
          LDU       2,S         GET BASE ADDRESS OF STRING 2
          LDB       ,X+         GET LENGTH OF STRING 1
          CMPB      ,U+         COMPARE TO LENGTH OF STRING 2
          BCS       BEGCMP      BRANCH IF STRING 1 IS SHORTER
                                * ITS LENGTH IS NUMBER OF BYTES TO COMPARE
          LDB       -1,U        OTHERWISE, STRING 2 IS SHORTER
                                * ITS LENGTH IS NUMBER OF BYTES TO COMPARE
          *
          *COMPARE STRINGS THROUGH LENGTH OF SHORTER
          *EXIT AS SOON AS CORRESPONDING CHARACTERS DIFFER
          *
BEGCMP:
          TSTB                  CHECK IF SHORTER STRING HAS ZERO LENGTH
          BEQ       EXITSC      BRANCH (EXIT) IF IT DOES
CMPLP:
          LDA       ,X+         GET CHARACTER FROM STRING 1
          CMPA      ,U+         COMPARE TO CHARACTER FROM STRING 2
          BNE       EXITSC      BRANCH IF CHARACTERS ARE NOT EQUAL
                                * Z,C WILL BE PROPERLY SET OR CLEARED
          DECB                  COUNT CHARACTERS
          BNE       CMPLP       CONTINUE UNTIL ALL BYTES COMPARED
          *
          *STRINGS SAME THROUGH LENGTH OF SHORTER
          *SO USE LENGTHS TO SET FLAGS
          *
          LDA       [4,S]       GET LENGTH OF STRING 1
          CMPA      [2,S]       COMPARE LENGTH OF STRING 2
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
EXITSC:
          LDU       ,S          SAVE RETURN ADDRESS
          LEAS      6,S         REMOVE PARAMETERS FROM STACK
          JMP       ,U          EXIT TO RETURN ADDRESS

*
*       SAMPLE EXECUTION:
*


SC5A:
          LDY       #S1         BASE ADDRESS OF STRING 1
          LDX       #S2         BASE ADDRESS OF STRING 2
          PSHS      X,Y         SAVE PARAMETERS IN STACK
          JSR       STRCMP      COMPARE STRINGS
                                *COMPARING "STRING 1" AND "STRING 2"
                                *  RESULTS IN STRING 1 LESS THAN
                                *  STRING 2, SO Z=0,C=1
          BRA       SC5A        LOOP THROUGH TEST
*
*         TEST DATA
*
S1        FCB       $20
          FCC       /STRING 1
```

```
S2      FCB     $20
        FCC     /STRING 2                       /

        END
```

5B String concatenation (CONCAT)

Combines (concatenates) two strings, placing the second immediately after the first in memory. If the concatenation would produce a string longer than a specified maximum, the program concatenates only enough of string 2 to give the combined string its maximum length. The Carry flag is cleared if all of string 2 can be concatenated. It is set to 1 if part of string 2 must be dropped. Each string consists of at most 256 bytes, including an initial byte containing the length.

Procedure The program uses the length of string 1 to determine where to start adding characters, and the length of string 2 to determine how many characters to add. If the sum of the lengths exceeds the maximum, the program indicates an overflow. It then reduces the number of characters it must add to the maximum length minus the length of string 1. Finally, it moves the characters from string 2 to the end of string 1, updates the length of string 1, and sets the Carry flag to indicate whether characters were discarded.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Maximum length of string 1

More significant byte of base address of string 2
Less significant byte of base address of string 2

More significant byte of base address of string 1
Less significant byte of base address of string 1

Exit conditions

String 2 concatenated at the end of string 1 and the length of string 1 increased accordingly. If the combined string would exceed the maximum length, only the part of string 2 that would give string 1 its

maximum length is concatenated. If any part of string 2 must be dropped, the Carry flag is set to 1. Otherwise, the Carry flag is cleared.

Examples

1. Data: Maximum length of string 1 = $0E_{16} = 14_{10}$
   String 1 = 07'JOHNSON' (07 is the length of the string)
   String 2 = 05', DON' (05 is the length of the string)
   Result: String 1 = 0C'JOHNSON, DON' ($0C_{16} = 12_{10}$ is the length of the combined string with string 2 placed after string 1)
   Carry = 0, since no characters were dropped

2. Data: String 1 = 07'JOHNSON' (07 is the length of the string)
   String 2 = 09', RICHARD' (09 is the length of the string)
   Result: String 1 = 0E'JOHNSON, RICHA' ($0E_{16} = 14_{10}$ is the maximum length allowed, so the last two characters of string 2 have been dropped)
   Carry = 1, since characters had to be dropped

Note that we are representing the initial byte (containing the string's length) as two hexadecimal digits in both examples.

Registers used All

Execution time Approximately

17 × NUMBER OF CHARACTERS CONCATENATED plus 95 cycles overhead

NUMBER OF CHARACTERS CONCATENATED is usually the length of string 2, but will be the maximum length of string 1 minus its current length if the combined string would be too long. If, for example, NUMBER OF CHARACTERS CONCATENATED is 14_{16} (20_{10}), the execution time is

17 × 20 + 95 = 340 + 95 = 435 cycles

The overhead is an extra 28 cycles if the string must be truncated.

Program size 59 bytes

Data memory required None

Special cases

1. If the concatenation would make the string exceed its specified maximum length, the program concatenates only enough of string 2 to reach the maximum. If any of string 2 must be truncated, the Carry flag is set to 1.

2. If string 2 has a length of 0, the program exits with the Carry flag cleared (no errors) and string 1 unchanged. That is, a length of 0 for either string is interpreted as 0, not as 256.

3. If the original length of string 1 exceeds the specified maximum, the program exits with the Carry flag set to 1 (indicating an error) and string 1 unchanged.

```
*
*       Title           String Concatenation
*       Name:           CONCAT
*
*
*       Purpose:        Concatenate 2 strings into one string.
*
*       Entry:          TOP OF STACK
*                         High byte of return address
*                         Low byte of return address
*                         Maximum length of string 1
*                         High byte of string 2 address
*                         Low byte of string 2 address
*                         High byte of string 1 address
*                         Low byte of string 1 address
*
*                         Each string consists of a length byte
*                         followed by a maximum of 255 characters.
*
*       Exit:           String 1 := string 1 concatenated with string 2
*                       If no errors then
*                           Carry := 0
*                       else
*                         begin
*                           Carry := 1
*                           if the concatenation makes string 1 too
*                           long, concatenate only the part of string 2
*                           that results in string 1 having its maximum
*                           length
*                           if length(string1) > maximum length then
```

```
*                              no concatenation is done
*                          end
*
*       Registers Used: All
*
*       Time:           Approximately 17 * (length of string 2) cycles
*                       plus 95 cycles overhead
*
*       Size:           Program 59 bytes
*

CONCAT:
          LDU       ,S        SAVE RETURN ADDRESS
          *
          *DETERMINE WHERE TO START ADDING CHARACTERS
          *CONCATENATION STARTS AT THE END OF STRING 1
          *END OF STRING 1 = BASE 1 + LENGTH1 + 1, WHERE
          *  THE EXTRA 1 IS FOR THE LENGTH BYTE
          *
          CLR       1,S       INDICATE NO TRUNCATION NECESSARY
          LDX       5,S       GET BASE ADDRESS OF STRING 1
          LDA       ,X        GET LENGTH OF STRING 1
          LEAX      A,X       POINT TO LAST BYTE IN STRING 1
          LEAX      1,X       POINT JUST BEYOND END OF STRING 1
          *
          *NEW CHARACTERS COME FROM STRING 2, STARTING AT
          *  BASE2+1 (SKIPPING OVER LENGTH BYTE)
          *
          LDY       3,S       GET BASE ADDRESS OF STRING 2
          LDB       ,Y+       GET LENGTH OF STRING 2 AND POINT TO
                              * FIRST DATA BYTE
          BEQ       SETTRN    BRANCH (EXIT) IF STRING 2 HAS ZERO LENGTH
                              * NO ERROR IN THIS CASE
          *
          *DETERMINE HOW MANY CHARACTERS TO CONCATENATE
          *THIS IS LENGTH OF STRING 2 IF COMBINED STRING WOULD
          *  NOT EXCEED MAXIMUM LENGTH
          *OTHERWISE, IT IS THE NUMBER THAT WOULD BRING COMBINED
          *  STRING TO ITS MAXIMUM LENGTH - THAT IS, MAXIMUM LENGTH
          *  MINUS LENGTH OF STRING 1
          *
          STB       ,S        SAVE LENGTH OF STRING 2 IN STACK
          ADDA      ,S        ADD STRING LENGTHS TO DETERMINE LENGTH
                              * OF COMBINED STRING
          BCS       TOOLNG    BRANCH IF LENGTH WILL EXCEED 255 BYTES
          CMPA      2,S       COMPARE TO MAXIMUM LENGTH
          BLS       DOCAT     BRANCH IF LENGTH DOES NOT EXCEED MAXIMUM
          *
          *COMBINED STRING IS TOO LONG
          * INDICATE STRING OVERFLOW WITH FF MARKER IN STACK
          * SET NUMBER OF CHARACTERS TO CONCATENATE =  MAXLEN - S1LEN
          * SET NEW LENGTH OF STRING 1 TO MAXIMUM LENGTH
          *
TOOLNG:
          COM       1,S       INDICATE STRING TRUNCATION (MARKER = FF)
          LDB       2,S       NUMBER OF CHARACTERS TO CONCATENATE =
```

```
          SUBB      [5,S]       MAXIMUM LENGTH - STRING 1 LENGTH
          BLS       SETTRN     BRANCH (EXIT) IF ORIGINAL STRING WAS
                               * TOO LONG
          LDA       2,S        NEW LENGTH = MAXIMUM LENGTH
          *
          *CONCATENATE STRINGS BY MOVING CHARACTERS FROM STRING 2
          * TO THE AREA FOLLOWING STRING 1
          *
DOCAT:
          STA       [5,S]      SAVE NEW LENGTH IN STRING 1'S LENGTH BYTE
          TSTB                 CHECK NUMBER OF BYTES TO CONCATENATE
          BEQ       SETTRN     BRANCH (EXIT) IF NO BYTES TO CONCATENATE
CATLP:
          LDA       ,Y+        GET BYTE FROM STRING 2
          STA       ,X+        MOVE BYTE TO AREA FOLLOWING STRING 1
          DECB                 CONTINUE UNTIL ALL BYTES MOVED
          BNE       CATLP
          *
          *SET CARRY FROM TRUNCATION INDICATOR IN STACK
          *CARRY = 1 IF CHARACTERS HAD TO BE TRUNCATED, 0 OTHERWISE
          *
SETTRN:
          ROR       1,S        SET CARRY FROM TRUNCATION INDICATOR
                               * CARRY = 1 IF TRUNCATION, 0 IF NOT
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
          LEAS      7,S        REMOVE PARAMETERS FROM STACK
          JMP       ,U         EXIT TO RETURN ADDRESS


*
*       SAMPLE EXECUTION:
*


SC5B:
          LDY       #S1        GET BASE ADDRESS OF STRING 1
          LDX       #S2        GET BASE ADDRESS OF STRING 2
          LDA       #$20       GET MAXIMUM LENGTH OF STRING 1
          PSHS      A,X,Y      SAVE PARAMETERS IN STACK
          JSR       CONCAT     CONCATENATE STRINGS
                               *RESULT OF CONCATENATING
                               * "LASTNAME" AND ", FIRSTNAME"
                               * IS S1 = 13H,"LASTNAME, FIRSTNAME"
          BRA       SC5B       LOOP THROUGH TEST
*
*TEST DATA
*
S1:       FCB       8          LENGTH OF S1 IN BYTES
          FCC       /LASTNAME                  / 32 BYTE MAX LENGTH
S2:       FCB       $0B        LENGTH OF S2 IN BYTES
          FCC       /, FIRSTNAME               / 32 BYTE MAX LENGTH

          END
```

5C Find the position of a substring (POS)

Searches for the first occurrence of a substring within a string. Returns the index at which the substring starts if it is found and 0 otherwise. The string and the substring each consist of at most 256 bytes, including an initial byte containing the length. Thus, if the substring is found, its starting index cannot be less than 1 or more than 255.

Procedure The program moves through the string searching for the substring. It continues until it finds a match or until the remaining part of the string is shorter than the substring and hence cannot possibly contain it. If the substring does not appear in the string, the program clears register A; otherwise, the program places the substring's starting index in register A.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of base address of substring
Less significant byte of base address of substring

More significant byte of base address of string
Less significant byte of base address of string

Exit conditions

Register A contains index at which first occurrence of substring starts if it is found; register A contains 0 if substring is not found

Examples

1. Data: String = 1D'ENTER SPEED IN MILES PER HOUR' ($1D_{16} = 29_{10}$ is the length of the string)
   Substring = 05'MILES' (05 is the length of the substring)
   Result: Register A contains 10_{16} (16_{10}), the index at which the substring 'MILES' starts

2. Data: String = 1B'SALES FIGURES FOR JUNE 1981' ($1B_{16}$ = 27_{10} is the length of the string)
   Substring = 04'JUNE' (04 is the length of the substring)
   Result: Register A contains 13_{16} (19_{10}), the index at which the substring 'JUNE' starts

3. Data: String = 10'LET Y1 = X1 + R7' (10_{16} = 16_{10} is the length of the string)
   Substring = 02'R4' (02 is the length of the substring)
   Result: Register A contains 0, since the substring 'R4' does not appear in the string LET Y1 = X1 + R7

4. Data: String = 07'RESTORE' (07 is the length of the string)
   Substring = 03'RES' (03 is the length of the substring)
   Result: Register A contains 1, the index at which the substring 'RES' starts. An index of 1 indicates that the substring could be an abbreviation of the string. Interactive programs, such as BASIC interpreters and word processors, often use abbreviations to save on typing and storage.

Registers used All

Execution time Data-dependent, but the overhead is approximately 100 cycles, each successful match of one character takes 20 cycles, and each unsuccessful match of one character takes 58 cycles. The worst case is when the string and substring always match except for the last character in the substring, such as

String = 'AAAAAAAAB'
Substring = 'AAB'

The execution time in that case is

(STRING LENGTH − SUBSTRING LENGTH + 1) × (20 × (SUBSTRING LENGTH −1) + 58) + 100

If, for example, STRING LENGTH = 9 and SUBSTRING LENGTH = 3 (as in the example above), the execution time is

(9 − 3 + 1) × (20 × (3 − 1) + 58) + 100 = 7 × 98 + 100
= 686 + 100
= 786 cycles.

Program size 71 bytes

Data memory required 2 stack bytes

Special case

1. If either the string or the substring has a length of 0, the program exits with 0 in register A, indicating that it did not find the substring.

2. If the substring is longer than the string, the program exits with 0 in register A, indicating that it did not find the substring.

3. If the program returns an index of 1, the substring may be regarded as an abbreviation of the string. That is, the substring occurs in the string, starting at the first character. A typical example would be a string PRINT and a substring PR.

4. If the substring occurs more than once in the string, the program will return only the index to the first occurrence (the one with the smallest starting index).

```
*       Title           Find the Position of a Substring
*       Name:           POS
*
*
*       Purpose:        Search for the first occurrence of a substring
*                       in a string and return its starting index.
*                       If the substring is not found, a 0 is returned.
*
*       Entry:          TOP OF STACK
*                         High byte of return address
*                         Low byte of return address
*                         High byte of substring address
*                         Low byte of substring address
*                         High byte of string address
*                         Low byte of string address
*
*                         Each string consists of a length byte
*                         followed by a maximum of 255 characters.
*
*       Exit:           If the substring is found then
*                         Register A = its starting index
*                       else
*                         Register A = 0
*
*       Registers Used: All
*
*
*       Time:           Since the algorithm is so data dependent
```

```
*                       a simple formula is impossible but the
*                       following statements are true and a
*                       worst case is given below:
*
*                       100 cycles overhead.
*                       Each match of 1 character takes 20 cycles
*                       A mismatch takes 58 cycles
*
*                       Worst case timing occurs when the
*                       string and substring always match
*                       except for the last character of the
*                       substring, such as:
*                           string = 'AAAAAAAAAB'
*                           substring = 'AAB'
*
*       Size:           Program 71 bytes
*                       Data     2 stack bytes
*


POS:
          LDU       ,S          SAVE RETURN ADDRESS
          *
          *EXIT, INDICATING SUBSTRING NOT FOUND, IF STRING OR SUBSTRING
          *  HAS ZERO LENGTH OR IF SUBSTRING IS LONGER THAN STRING
          *
          CLRA                  INDICATE SUBSTRING NOT FOUND
          LDX       2,S         GET BASE ADDRESS OF SUBSTRING
          LDY       4,S         GET BASE ADDRESS OF STRING
          LDB       ,Y+         GET STRING LENGTH
          BEQ       EXITPO      BRANCH (EXIT) IF STRING LENGTH IS ZERO
          TST       ,X          CHECK SUBSTRING LENGTH
          BEQ       EXITPO      BRANCH (EXIT) IF SUBSTRING LENGTH IS ZERO
          SUBB      ,X          COMPARE STRING LENGTH, SUBSTRING LENGTH
          BCS       EXITPO      BRANCH (EXIT) IF SUBSTRING IS LONGER THAN
                                * STRING
          *
          *SAVE INITIAL LOOP VARIABLES IN STACK
          *THESE ARE (BOTTOM TO TOP):
          *  ADDRESS OF FIRST CHARACTER IN SUBSTRING
          *  LENGTH OF PART OF STRING THAT MUST BE EXAMINED
          *  LENGTH OF SUBSTRING
          *  ADDRESS OF FIRST CHARACTER IN STRING (POINTER TO
          *    INITIAL SECTION TO BE EXAMINED)
          *
          INCB                  LENGTH OF PART THAT MUST BE EXAMINED IS
                                * STRING LENGTH - SUBSTRING LENGTH + 1
                                * REMAINDER IS TOO SHORT TO CONTAIN
                                * SUBSTRING
          LDA       ,X+         GET SUBSTRING LENGTH, MOVE POINTER TO
                                * FIRST CHARACTER IN SUBSTRING
          STX       2,S         SAVE ADDRESS OF FIRST CHARACTER IN
                                * SUBSTRING
          STD       ,S          SAVE LENGTHS IN STACK AS INITIAL VALUES
                                * FOR COUNTERS
          PSHS      Y           SAVE ADDRESS OF FIRST STRING BYTE
```

```
            *
            *SEARCH FOR SUBSTRING IN STRING
            *START SEARCH AT BASE OF STRING
            *CONTINUE UNTIL REMAINING STRING SHORTER THAN SUBSTRING
            *
CMPPOS:
            LDY         ,S          GET CURRENT STARTING POSITION IN STRING
            LDX         4,S         GET BASE ADDRESS OF SUBSTRING
            LDB         2,S         GET SUBSTRING LENGTH
            *
            *COMPARE BYTES OF SUBSTRING WITH BYTES OF STRING,
            * STARTING AT CURRENT POSITION IN STRING
            *
CHBYTE:
            LDA         ,Y+         GET BYTE OF STRING
            CMPA        ,X+         COMPARE TO BYTE OF SUBSTRING
            BNE         NOTFND      BRANCH IF NOT SAME, SUBSTRING NOT FOUND
            DECB                    CONTINUE THROUGH SUBSTRING
            BNE         CHBYTE
            *
            *SUBSTRING FOUND - CALCULATE INDEX AT WHICH IT STARTS IN
            * STRING
            *
            LDD         ,S          GET STARTING ADDRESS OF SECTION CONTAINING
                                    * SUBSTRING
            SUBD        6,S         SUBTRACT ADDRESS OF STRING'S LENGTH
                                    * BYTE. DIFFERENCE ENDS UP IN B
            TFR         B,A         SAVE INDEX IN A
            BRA         REMTMP      EXIT, REMOVING TEMPORARIES FROM STACK
            *
            *ARRIVE HERE IF SUBSTRING NOT FOUND
            *MOVE STRING POINTER UP 1 FOR NEXT COMPARISON
            *COUNT NUMBER OF COMPARISONS
            *
NOTFND:
            LDD         ,S          MOVE CURRENT (STARTING) POSITION IN
            ADDD        #1           STRING UP 1 CHARACTER
            STD         ,S
            DEC         3,S         SEARCH THROUGH SECTION OF STRING
            BNE         CMPPOS        THAT COULD CONTAIN SUBSTRING
            CLRA                    SUBSTRING NOT FOUND AT ALL - MAKE
                                    * STARTING INDEX ZERO
            *
            *REMOVE TEMPORARY STORAGE, PARAMETERS FROM STACK AND EXIT
            *
REMTMP:
            LEAS        2,S         REMOVE TEMPORARIES FROM STACK
EXITPO:
            LEAS        6,S         REMOVE PARAMETERS FROM STACK
            JMP         ,U          EXIT TO RETURN ADDRESS


*
*       SAMPLE EXECUTION:
*
```

```
SC5C:
          LDY       #STG       GET BASE ADDRESS OF STRING
          LDX       #SSTG      GET BASE ADDRESS OF SUBSTRING
          PSHS      X,Y        SAVE PARAMETERS IN STACK
          JSR       POS                FIND POSITION OF SUBSTRING
                               * SEARCHING "AAAAAAAAAB" FOR "AAB"
                               * RESULTS IN REGISTER A=8
          BRA       SC5C       LOOP THROUGH TEST
*
*         TEST DATA
*
STG:      FCB       $0A        LENGTH OF STRING
          FCC       /AAAAAAAAAB                        / 32 BYTE MAX
SSTG:     FCB       3          LENGTH OF SUBSTRING
          FCC       /AAB                               / 32 BYTE MAX

          END
```

5D Copy a substring from a string (COPY)

Copies a substring from a string, given a starting index and the number of bytes to copy. Each string consists of at most 256 bytes, including an initial byte containing the length. If the starting index of the substring is 0 (i.e. the substring would start in the length byte) or is beyond the end of the string, the substring is given a length of 0 and the Carry flag is set to 1. If the substring would exceed its maximum length or would extend beyond the end of the string, then only the maximum number or the available number of characters (up to the end of the string) are placed in the substring, and the Carry flag is set to 1. If the substring can be formed as specified, the Carry flag is cleared.

Procedure The program exits immediately if the number of bytes to copy, the maximum length of the substring, or the starting index is 0. It also exits immediately if the starting index exceeds the length of the string. If none of these conditions holds, the program checks whether the number of bytes to copy exceeds either the maximum length of the substring or the number of characters available in the string. If either is exceeded, the program reduces the number of bytes to copy accordingly. It then copies the bytes from the string to the substring. The program clears the Carry flag if the substring can be formed as specified and sets the Carry flag if it cannot.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Number of bytes to copy

Starting index to copy from

More significant byte of base address of substring
Less significant byte of base address of substring

More significant byte of base address of string
Less significant byte of base address of string

Maximum length of substring

Exit conditions

Substring contains characters copied from string. If the starting index is 0, the maximum length of the substring is 0, or the starting index is beyond the length of the string, the substring will have a length of 0 and the Carry flag will be set to 1. If the substring would extend beyond the end of the string or would exceed its specified maximum length, only the available characters from the string (up to the maximum length of the substring) are copied into the substring; the Carry flag is set in this case also. If no problems occur in forming the substring, the Carry flag is cleared.

Examples

1. Data: String = 10'LET Y1 = R7 + X4' ($10_{16} = 16_{10}$ is the length of the string)
Maximum length of substring = 2
Number of bytes to copy = 2
Starting index = 5

Result: Substring = 02'Y1' (2 is the length of the substring).
We have copied 2 bytes from the string starting at character #5 (i.e. characters 5 and 6)
Carry = 0, since no problems occur in forming the substring

2. Data: String = 0E'8657 POWELL ST' ($0E_{16} = 14_{10}$ is the length of the string)
Maximum length of substring = $10_{16} = 16_{10}$
Number of bytes to copy = $0D_{16} = 13_{10}$
Starting index = 06

Result: Substring = 09'POWELL ST' (09 is the length of the substring)
Carry = 1, since there were not enough characters available in the string to provide the specified number of bytes to copy

3. Data: String = 16'9414 HEGENBERGER DRIVE' ($16_{16} = 22_{10}$ is the length of the string)
Maximum length of substring = $10_{16} = 16_{10}$
Number of bytes to copy = $11_{16} = 17_{10}$
Starting index = 06

Result: Substring = 10'HEGENBERGER DRIV' ($10_{16} = 16_{10}$ is the length of the substring)

Carry = 1, since the number of bytes to copy exceeded the maximum length of the substring

Registers used All

Execution time Approximately

17 × NUMBER OF BYTES COPIED plus 150 cycles overhead

NUMBER OF BYTES COPIED is the number specified if no problems occur, or the number available or the maximum length of the substring if copying would extend beyond either the string or the substring. If, for example, NUMBER OF BYTES COPIED = 12_{10} ($0C_{16}$), the execution time is

17 × 12 + 150 = 204 + 150 = 354 cycles

Program size 85 bytes

Data memory required None

Special cases

1. If the number of bytes to copy is 0, the program assigns the substring a length of 0 and clears the Carry flag, indicating no error.

2. If the maximum length of the substring is 0, the program assigns the substring a length of 0 and sets the Carry flag to 1, indicating an error.

3. If the starting index of the substring is 0, the program assigns the substring a length of 0 and sets the Carry flag to 1, indicating an error.

4. If the source string does not even reach the specified starting index, the program assigns the substring a length of 0 and sets the Carry flag to 1, indicating an error.

5. If the substring would extend beyond the end of the source string, the program places all the available characters in the substring and sets the Carry flag to 1, indicating an error. The available characters are the ones from the starting index to the end of the string.

6. If the substring would exceed its specified maximum length, the program places only the specified maximum number of characters in the substring. It sets the Carry flag to 1, indicating an error.

```
*       Title           Copy a Substring from a String
*       Name:           COPY
*
*
*       Purpose:        Copy a substring from a string given a starting
*                       index and the number of bytes.
*
*       Entry:          TOP OF STACK
*                         High byte of return address
*                         Low byte of return address
*                         Number of bytes to copy
*                         Starting index to copy from
*                         High byte of destination string address
*                         Low byte of destination string address
*                         High byte of source string address
*                         Low byte of source string address
*                         Maximum length of destination string
*
*                         Each string consists of a length byte
*                         followed by a maximum of 255 characters.
*
*       Exit:           Destination string := The substring from the
*                       string.
*                       If no errors then
*                         Carry := 0
*                       else
*                         begin
*                           the following conditions cause an
*                           error and the Carry flag = 1.
*                           if (index = 0) or (maxlen = 0) or
*                              (index > length(source) then
*                             the destination string will have a zero
*                             length.
*                           if (index + count - 1) > length(source))
*                           then
*                             the destination string becomes everything
*                             from index to the end of source string.
*                         end
*
*       Registers Used: All
*
*
*       Time:           Approximately (17 * count) cycles plus
*                       150 cycles overhead
*
*       Size:           Program 85 bytes
*
*

COPY:
```

```
        LDU     ,S      SAVE RETURN ADDRESS
        *
        *EXIT IF ZERO BYTES TO COPY, ZERO MAXIMUM SUBSTRING
        *  LENGTH, OR ZERO STARTING INDEX
        *LENGTH OF SUBSTRING IS ZERO IN ALL CASES
        *
        CLR     ,S      LENGTH OF SUBSTRING = 0
        LDA     2,S     CHECK NUMBER OF BYTES TO COPY
        BEQ     OKEXIT  BRANCH IF ZERO BYTES TO COPY, NO ERROR
                        * SUBSTRING WILL JUST HAVE ZERO LENGTH
        LDA     8,S     CHECK MAXIMUM LENGTH OF SUBSTRING
        BEQ     EREXIT  TAKE ERROR EXIT IF SUBSTRING HAS ZERO
                        MAXIMUM LENGTH
        LDA     3,S     CHECK STARTING INDEX
        BEQ     EREXIT  TAKE ERROR EXIT IF STARTING INDEX IS
                        * ZERO (LENGTH BYTE)
        *
        *CHECK IF SOURCE STRING REACHES STARTING INDEX
        *TAKE ERROR EXIT IF IT DOESN'T
        *
        LDX     6,S     GET ADDRESS OF SOURCE STRING
        CMPA    ,X      COMPARE STARTING INDEX TO LENGTH OF
                        * SOURCE STRING
        BHI     EREXIT  TAKE ERROR EXIT IF STARTING INDEX IS
                        * TOO LARGE

        *
        *CHECK IF THERE ARE ENOUGH CHARACTERS IN SOURCE STRING
        * TO SATISFY THE NEED
        *THERE ARE IF STARTING INDEX + NUMBER OF BYTES TO COPY - 1
        * IS LESS THAN OR EQUAL TO THE LENGTH OF THE SOURCE
        * STRING
        *
        CLR     1,S     INDICATE NO TRUNCATION NEEDED
        LDB     2,S     COUNT = NUMBER OF BYTES TO COPY
        ADDA    2,S     ADD COUNT TO STARTING INDEX
        BCS     REDLEN  BRANCH IF SUM IS GREATER THAN 255
        DECA            CALCULATE INDEX OF LAST BYTE IN AREA
                        * SPECIFIED FOR COPYING
        CMPA    ,X      COMPARE TO LENGTH OF SOURCE STRING
        BLS     CHKMAX  BRANCH IF SOURCE STRING IS LONGER
        *
        *CALLER ASKED FOR TOO MANY CHARACTERS
        *JUST RETURN EVERYTHING BETWEEN STARTING INDEX AND THE END OF
        *  THE SOURCE STRING
        *COUNT := LENGTH(SSTRG) - STARTING INDEX + 1
        *INDICATE TRUNCATION OF COUNT
        *
REDLEN:
        LDB     ,X      GET LENGTH OF SOURCE STRING
        SUBB    3,S     COUNT = LENGTH - STARTING INDEX + 1
        INCB
        COM     1,S     INDICATE TRUNCATION OF COUNT BY
                        * SETTING MARKER TO FF
        *
        *DETERMINE IF THERE IS ENOUGH ROOM IN THE SUBSTRING
```

```
          *CHECK IF COUNT IS LESS THAN OR EQUAL TO MAXIMUM LENGTH
          *  OF DESTINATION STRING.  IF NOT, SET COUNT TO
          *  MAXIMUM LENGTH
          *IF COUNT > MAXLEN THEN COUNT := MAXLEN
          *
CHKMAX:
          CMPB      8,S         COMPARE COUNT TO MAXIMUM SUBSTRING LENGTH
          BLS       MOVSTR      BRANCH (NO PROBLEM) IF COUNT IS LESS
                                * THAN OR EQUAL TO MAXIMUM
          LDB       8,S         OTHERWISE, REPLACE COUNT WITH MAXIMUM
          *
          *MOVE SUBSTRING TO DESTINATION STRING
          *
MOVSTR:
          STB       ,S          SAVE COUNT (LENGTH OF SUBSTRING)
          LDA       3,S         GET STARTING INDEX
          LEAX      A,X         POINT TO FIRST CHARACTER IN SOURCE STRING
          LDY       4,S         POINT TO BASE OF DESTINATION STRING
          LEAY      1,Y         POINT TO FIRST CHARACTER IN SUBSTRING
MVLP:
          LDA       ,X+         GET BYTE FROM SOURCE STRING
          STA       ,Y+         MOVE BYTE TO DESTINATION STRING
          DECB                  CONTINUE THROUGH SPECIFIED NUMBER OF
          BNE       MVLP          BYTES (COUNT)
          ROL       1,S         MAKE CARRY INDICATE WHETHER REQUEST WAS
                                * FULLY SATISFIED (1 IF IT WAS, 0 IF NOT)
          BCS       EREXIT
          *
          *MAKE CARRY INDICATE WHETHER ERRORS OCCURRED
          *0 IF NOT, 1 IF THEY DID
          *
OKEXIT:   CLC                   CLEAR CARRY, GOOD EXIT
          BRA       EXITCP
EREXIT:   SEC                   SET CARRY, ERROR EXIT
          *
          *SET LENGTH OF SUBSTRING (COUNT)
          *
EXITCP:
          LDA       ,S          GET SUBSTRING LENGTH
          STA       [4,S]       SAVE LENGTH IN SUBSTRING'S LENGTH BYTE
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
          LEAS      9,S         REMOVE PARAMETERS FROM STACK
          JMP       ,U          EXIT TO RETURN ADDRESS

*
*       SAMPLE EXECUTION:
*


SC5D:
          LDA       MXLEN       MAXIMUM LENGTH OF SUBSTRING
          PSHS      A           SAVE MAXIMUM LENGTH IN STACK
          LDY       #SSTG       BASE ADDRESS OF SOURCE STRING
          LDX       #DSTG       BASE ADDRESS OF DESTINATION STRING
```

```
          LDB       IDX         STARTING INDEX TO COPY FROM
          LDA       CNT         NUMBER OF BYTES TO COPY
          PSHS      A,B,X,Y     SAVE PARAMETERS IN STACK
          JSR       COPY        COPY SUBSTRING
                                *COPYING 3 CHARACTERS STARTING AT INDEX 4
                                * FROM '12.345E+10' GIVES '345'
          BRA       SC5D        LOOP THROUGH TEST

*
*DATA SECTION
*
IDX       FCB       4           STARTING INDEX FOR COPYING
CNT       FCB       3           NUMBER OF CHARACTERS TO COPY
MXLEN     FCB       $20         MAXIMUM LENGTH OF DESTINATION STRING
SSTG      FCB       $0A         LENGTH OF STRING
          FCC       /12.345E+10                       / 32 BYTE MAX
DSTG      FCB       0           LENGTH OF SUBSTRING
          FCC       /                                 / 32 BYTE MAX

          END
```

5E Delete a substring from a string (DELETE)

Deletes a substring from a string, given a starting index and a length. The string consists of at most 256 bytes, including an initial byte containing the length. The Carry flag is cleared if the deletion can be performed as specified. The Carry flag is set if the starting index is 0 or beyond the length of the string; the string is left unchanged in either case. If the deletion extends beyond the end of the string, the Carry flag is set to 1 and only the characters from the starting index to the end of the string are deleted.

Procedure The program exits immediately if either the starting index or the number of bytes to delete is 0. It also exits if the starting index is beyond the length of the string. If none of these conditions holds, the program checks whether the string extends beyond the area to be deleted. If it does not, the program simply truncates the string by setting the new length to the starting index minus 1. If it does, the program compacts the string by moving the bytes above the deleted area down. The program then determines the new string's length and exits with the Carry cleared if the specified number of bytes were deleted, and with the Carry set to 1 if any errors occurred.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Number of bytes to delete

Starting index to delete from

More significant byte of base address of string
Less significant byte of base address of string

Exit conditions

Substring deleted from string. If no errors occur, the Carry flag is cleared. If the starting index is 0 or beyond the length of the string, the Carry flag is set and the string is unchanged. If the number of bytes to

delete would go beyond the end of the string, the Carry flag is set and the characters from the starting index to the end of the string are deleted.

Examples

1. Data: String = 26'SALES FOR MARCH AND APRIL OF THIS YEAR' ($26_{16} = 38_{10}$ is the length of the string)
   Number of bytes to delete = $0A_{16} = 10_{10}$
   Starting index to delete from = $10_{16} = 16_{10}$
   Result: String = 1C'SALES FOR MARCH OF THIS YEAR' ($1C_{16} = 28_{10}$ is the length of the string with 10 bytes deleted starting with the 16th character – the deleted material is 'AND APRIL ')
   Carry = 0, since no problems occurred in the deletion
2. Data: String = 28'THE PRICE IS $3.00 ($2.00 BEFORE JUNE 1)' ($28_{16} = 40_{10}$ is the length of the string)
   Number of bytes to delete = $30_{16} = 48_{10}$
   Starting index to delete from = $13_{16} = 19_{10}$
   Result: String = 12'THE PRICE IS $3.00' ($12_{16} = 18_{10}$ is the length of the string with all remaining bytes deleted)
   Carry = 1, since there were not as many bytes left in the string as were supposed to be deleted

---

Registers used All

Execution time Approximately

17 × NUMBER OF BYTES MOVED DOWN + 120 cycles overhead

NUMBER OF BYTES MOVED DOWN is 0 if the string can be truncated and is STRING LENGTH – STARTING INDEX – NUMBER OF BYTES TO DELETE + 1 if the string must be compacted. That is, it takes extra time if the deletion creates a 'hole' in the string that must be filled.

Examples

1. STRING LENGTH = 20_{16} (32_{10})
   STARTING INDEX = 19_{16} (25_{10})

NUMBER OF BYTES TO DELETE = 08

Since there are exactly 8 bytes left in the string starting at index 19_{16}, all the routine must do is truncate it (i.e. cut off the end of the string). This takes

$17 \times 0 + 120 = 120$ cycles

2. STRING LENGTH = 40_{16} (64_{10})
   STARTING INDEX = 19_{16} (25_{10})
   NUMBER OF BYTES TO DELETE = 08

Since there are 20_{16} (32_{10}) bytes above the truncated area, the routine must move them down eight positions to fill the 'hole'. Thus NUMBER OF BYTES MOVED DOWN = 32_{10} and the execution time is

$17 \times 32 + 120 = 544 + 120 = 664$ cycles

Program size 80 bytes

Data memory required None

Special cases

1. If the number of bytes to delete is 0, the program exits with Carry flag cleared (no errors) and the string unchanged.

2. If the string does not even extend to the specified starting index, the program exits with the Carry flag set to 1 (indicating an error) and the string unchanged.

3. If the number of bytes to delete exceeds the number available, the program deletes all bytes from the starting index to the end of the string and exits with the Carry flag set to 1 (indicating an error).

```
*       Title           Delete a Substring from a String
*       Name:           DELETE
*
*
*       Purpose:        Delete a substring from a string given a
*                       starting index and a length.
*
*       Entry:          TOP OF STACK
*                         High byte of return address
*                         Low byte of return address
*                         Number of bytes to delete (count)
*                         Starting index to delete from (index)
```

```
*                         High byte of string address
*                         Low byte of string address
*
*                         The string consists of a length byte
*                         followed by a maximum of 255 characters.
*
*       Exit:             Substring deleted.
*                         If no errors then
*                           Carry := 0
*                         else
*                           begin
*                             the following conditions cause an
*                             error with Carry flag = 1.
*                             if (index = 0) or (index > length(string))
*                               then do not change string
*                             if count is too large then
*                               delete only the characters from
*                               index to end of string
*                           end
*
*       Registers used: All
*
*       Time:             Approximately 17 * (LENGTH(STRG)-INDEX-COUNT+1)
*                         plus 120 cycles overhead
*
*       Size:             Program 80 bytes
*


DELETE:
          LDU       ,S          SAVE RETURN ADDRESS
          *
          *INITIALIZE ERROR INDICATOR (DELERR) TO 0
          *
          CLR       ,S          INDICATE NO ERRORS
          *
          *EXIT IF COUNT IS ZERO, STARTING INDEX IS ZERO, OR
          *  STARTING INDEX IS BEYOND THE END OF THE STRING
          *
          LDB       2,S         CHECK NUMBER OF BYTES TO DELETE
          BEQ       OKEXIT      BRANCH (GOOD EXIT) IF NOTHING TO DELETE
          LDA       3,S         CHECK STARTING INDEX
          BEQ       EREXIT      BRANCH (ERROR EXIT) IF STARTING INDEX IS
                                * ZERO - THAT IS, IN LENGTH BYTE
          LDX       4,S         GET BASE ADDRESS OF STRING
          CMPA      ,X          CHECK IF STARTING INDEX IS WITHIN STRING
          BHI       EREXIT      BRANCH (ERROR EXIT) IF STARTING INDEX
                                * IS BEYOND END OF STRING
          *
          *CHECK WHETHER NUMBER OF CHARACTERS REQUESTED TO BE
          *  DELETED ARE PRESENT
          *THEY ARE IF STARTING INDEX + NUMBER OF BYTES TO DELETE - 1
          *  IS LESS THAN OR EQUAL TO STRING LENGTH
          *IF NOT, THEN DELETE ONLY TO END OF STRING
          *
          ADDA      2,S         COMPUTE STARTING INDEX + COUNT
```

```
        BCS     TRUNC     TRUNCATE IF INDEX + COUNT > 255
        DECA              END OF DELETED AREA IS AT INDEX GIVEN BY
                          * STARTING INDEX + COUNT - 1
        CMPA    ,X        COMPARE TO LENGTH OF SUBSTRING
        BCS     CNTOK     BRANCH IF MORE THAN ENOUGH CHARACTERS
        BEQ     TRUNC     TRUNCATE BUT NO ERROR (EXACTLY ENOUGH
                          * CHARACTERS)
        COM     ,S        INDICATE ERROR - NOT ENOUGH CHARACTERS
                          * TO DELETE
        *
        *TRUNCATE THE STRING - NO COMPACTING NECESSARY
        *SIMPLY REDUCE ITS LENGTH TO STARTING INDEX - 1
        *
TRUNC:
        LDA     3,S       STRING LENGTH = STARTING INDEX - 1
        DECA
        STA     ,X
        *
        *TEST ERROR INDICATOR AND EXIT ACCORDINGLY
        *
        LDA     ,S        TEST ERROR INDICATOR
        BEQ     OKEXIT    NO ERROR, TAKE GOOD EXIT
        BNE     EREXIT    OTHERWISE, TAKE ERROR EXIT
        *
        *DELETE SUBSTRING BY COMPACTING THE STRING
        *MOVE ALL CHARACTERS ABOVE THE DELETED AREA DOWN
        *
CNTOK:
        STA     1,S       SAVE INDEX TO END OF AREA TO BE DELETED
        LDB     ,X        NUMBER OF CHARACTERS TO MOVE = STRING
        SUBB    1,S        LENGTH - INDEX AT END OF AREA
        INCA                 ADD 1 TO INDEX AT END OF DELETED AREA
                          * THUS GIVING FIRST BYTE TO MOVE DOWN
        LEAY    A,X       POINT TO FIRST CHARACTER TO BE
                          * MOVED DOWN
        LDA     3,S       GET STARTING INDEX
        LEAX    A,X       POINT TO FIRST BYTE IN AREA TO BE DELETED
MVLP:
        LDA     ,Y+       GET CHARACTER FROM ABOVE DELETED AREA
        STA     ,X+       MOVE IT DOWN TO COMPACT STRING
        DECB              CONTINUE THROUGH END OF STRING
        BNE     MVLP
        *
        *COMPUTE AND SAVE LENGTH OF STRING AFTER DELETION
        *
        LDX     4,S       POINT TO STRING LENGTH
        LDA     ,X        GET ORIGINAL LENGTH
        SUBA    2,S       SUBTRACT NUMBER OF BYTES TO DELETE
        STA     ,X        DIFFERENCE IS NEW LENGTH
        *
        *CLEAR CARRY, INDICATING NO ERRORS
        *
OKEXIT:
        CLC               CLEAR CARRY, NO ERRORS
        BRA     EXITDE
        *
```

```
          *SET CARRY, INDICATING AN ERROR
          *
EREXIT:
          SEC                 SET CARRY, INDICATING ERROR
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
EXITDE:
          LEAS      6,S       REMOVE PARAMETERS FROM STACK
          JMP       ,U        EXIT TO RETURN ADDRESS

*
*       SAMPLE EXECUTION:
*

SC5E:
          LDX       #SSTG     GET BASE ADDRESS OF STRING
          LDB       IDX       GET STARTING INDEX FOR DELETION
          LDA       CNT       GET NUMBER OF CHARACTERS TO DELETE
          PSHS      A,B,X     SAVE PARAMETERS IN STACK
          JSR       DELETE    DELETE CHARACTERS
                        *DELETING 4 CHARACTERS STARTING AT INDEX 1
                        * FROM "JOE HANDOVER" LEAVES "HANDOVER"
          BRA       SC5E      LOOP THROUGH TEST

*
*DATA SECTION
*
IDX:      FCB       1         STARTING INDEX FOR DELETION
CNT:      FCB       4         NUMBER OF CHARACTERS TO DELETE
SSTG:     FCB       12        LENGTH OF STRING IN BYTES
          FCC       /JOE HANDOVER/

          END
```

5F Insert a substring into a string (INSERT)

Inserts a substring into a string, given a starting index. The string and substring each consist of at most 256 bytes, including an initial byte containing the length. The Carry flag is cleared if the insertion can be accomplished with no problems. The Carry flag is set if the starting index is 0 or beyond the length of the string. In the second case, the substring is concatenated to the end of the string. The Carry flag is also set if the insertion would make the string exceed a specified maximum length; in that case, the program inserts only enough of the substring to reach the maximum length.

Procedure The program exits immediately if the starting index or the length of the substring is 0. If neither is 0, the program checks whether the insertion would make the string longer than the specified maximum. If it would, the program truncates the substring. The program then checks whether the starting index is within the string. If not, the program simply concatenates the substring at the end of the string. If the starting index is within the string, the program must make room for the insertion by moving the remaining characters up in memory. This move must start at the highest address to avoid writing over any data. Finally, the program can move the substring into the open area. The program then determines the new string length. It exits with the Carry flag set to 0 if no problems occurred and to 1 if the starting index was 0, the substring had to be truncated, or the starting index was beyond the length of the string.

Entry conditions

Order in stack (starting from the top)

More significant byte of base address
Less significant byte of return address

Maximum length of string

Starting index at which to insert the substring

More significant byte of base address of substring
Less significant byte of base address of substring

More significant byte of base address of string

Less significant byte of base address of string

Exit conditions

Substring inserted into string. If no errors occur, the Carry flag is cleared. If the starting index or the length of the substring is 0, the Carry flag is set and the string is not changed. If the starting index is beyond the length of the string, the Carry flag is set and the substring is concatenated to the end of the string. If the insertion would make the string exceed its specified maximum length, the Carry flag is set and only enough of the substring is inserted to reach maximum length.

Examples

1. Data: String = 0A'JOHN SMITH' ($0A_{16} = 10_{10}$ is the length of the string)
Substring = 08'WILLIAM ' (08 is the length of the substring)
Maximum length of string = $14_{16} = 20_{10}$
Starting index = 06

Result: String = 12'JOHN WILLIAM SMITH' ($12_{16} = 18_{10}$ is the length of the string with the substring inserted)
Carry = 0, since no problems occurred in the insertion

2. Data: String = 0A'JOHN SMITH' ($0A_{16} = 10_{10}$ is the length of the string)
Substring = 0C'ROCKEFELLER' ($0C_{16} = 12_{10}$ is the length of the substring)
Maximum length of string = $14_{16} = 20_{10}$
Starting index = 06

Result: String = 14'JOHN ROCKEFELLESMITH' ($14_{16} = 20_{10}$ is the length of the string with as much of the substring inserted as the maximum length would allow)
Carry = 1, since some of the substring could not be inserted without exceeding the maximum length of the string

---

Registers used All

Execution time Approximately

17 × NUMBER OF BYTES MOVED + 17 × NUMBER OF BYTES INSERTED + 180 cycles

NUMBER OF BYTES MOVED is the number of bytes that must be moved to make room for the insertion. If the starting index is beyond the end of the string, this is 0 since the substring is simply placed at the end. Otherwise, this is STRING LENGTH − STARTING INDEX + 1, since the bytes at or above the starting index must be moved.

NUMBER OF BYTES INSERTED is the length of the substring if no truncation occurs. It is the maximum length of the string minus its current length if inserting the substring would produce a string longer than the maximum.

Examples

1. STRING LENGTH = 20_{16} (32_{10})
   STARTING INDEX = 19_{16} (25_{10})
   MAXIMUM LENGTH = 30_{16} (48_{10})
   SUBSTRING LENGTH = 06

That is, we want to insert a substring 6 bytes long, starting at the 25th character. Since 8 bytes must be moved up (NUMBER OF BYTES MOVED = 32 − 25 + 1) and 6 bytes must be inserted, the execution time is approximately

17 × 8 + 17 × 6 + 180 = 136 + 102 + 180 = 418 cycles

2. STRING LENGTH = 20_{16} (32_{10})
   STARTING INDEX = 19_{16} (25_{10})
   MAXIMUM LENGTH = 24_{16} (36_{10})
   SUBSTRING LENGTH = 06

As opposed to Example 1, here we can insert only 4 bytes the substring without exceeding the string's maximum length. Thus NUMBER OF BYTES MOVED = 8 and NUMBER OF BYTES INSERTED = 4. The execution time is approximately

17 × 8 + 17 × 4 + 180 = 136 + 68 + 180 = 384 cycles

Program size 115 bytes

Data memory required None

Special cases

1. If the length of the substring (the insertion) is 0, the program exits with the Carry flag cleared (no errors) and the string unchanged.

2. If the starting index for the insertion is 0 (i.e. the insertion would start in the length byte), the program exits with the Carry flag set to 1 (indicating an error) and the string unchanged.

3. If the insertion makes the string exceed the specified maximum length, the program inserts only enough characters to reach the maximum length. The Carry flag is set to 1 to indicate that the insertion has been truncated.

4. If the starting index of the insertion is beyond the end of the string, the program concatenates the insertion at the end of the string and indicates an error by setting the Carry flag to 1.

5. If the original length of the string exceeds its specified maximum length, the program exits with the Carry flag set to 1 (indicating an error) and the string unchanged.

```
*       Title           Insert a Substring into a String
*       Name:           INSERT
*
*
*       Purpose:        Insert a substring into a string given a
*                       starting index.
*
*       Entry:          TOP OF STACK
*                         High byte of return address
*                         Low byte of return address
*                         Maximum length of (source) string
*                         Starting index to insert the substring
*                         High byte of substring address
*                         Low byte of substring address
*                         High byte of (source) string address
*                         Low byte of (source) string address
*
*                         Each string consists of a length byte
*                         followed by a maximum of 255 characters.
*
*       Exit:           Substring inserted into string.
*                       If no errors then
*                         Carry = 0
*                       else
*                         begin
*                           the following conditions cause the
*                           Carry flag to be set.
*                           if index = 0 then
*                             do not insert the substring
*                           if length(string) > maximum length then
```

```
*                              do not insert the substring
*                            if index > length(string) then
*                              concatenate substring onto the end of the
*                              source string
*                            if length(string)+length(substring) > maxlen
*                              then insert only enough of the substring
*                              to reach maximum length
*                          end
*
*       Registers Used: All
*
*       Time:           Approximately
*                        17 * (LENGTH(STRING) - INDEX + 1) +
*                        17 * (LENGTH(SUBSTRING)) +
*                        180 cycles overhead
*
*       Size:           Program 115 bytes


INSERT:
*
*         START WITH ERROR INDICATOR CLEARED
*         POINTERS INITIALIZED TO BASE ADDRESSES OF STRING, SUBSTRING
*
          LDU       ,S          SAVE RETURN ADDRESS
          CLR       ,S          CLEAR ERROR INDICATOR (NO ERRORS)
          LDX       6,S         GET BASE ADDRESS OF STRING
          LDY       4,S         GET BASE ADDRESS OF SUBSTRING
          *
          *EXIT IF SUBSTRING LENGTH IS ZERO OR STARTING INDEX IS
          * ZERO
          *
          LDA       3,S         GET STARTING INDEX
          BEQ       EREXIT      EXIT, INDICATING ERROR, IF STARTING
                                * INDEX IS ZERO (LENGTH BYTE)
          LDB       ,Y          GET LENGTH OF SUBSTRING (NUMBER OF
                                * CHARACTERS TO INSERT
          BEQ       OKEXIT      EXIT IF NOTHING TO INSERT (NO ERROR)
          *
          *CHECK WHETHER THE STRING WITH THE INSERTION FITS IN THE
          * SOURCE STRING (I.E., IF ITS LENGTH IS LESS THAN OR EQUAL
          * TO THE MAXIMUM).
          *IF NOT, TRUNCATE THE SUBSTRING AND SET THE ERROR FLAG
          *
          LDA       ,Y          GET SUBSTRING LENGTH
          ADDA      ,X          SUBSTRING LENGTH + STRING LENGTH
          BCS       TRUNC       TRUNCATE SUBSTRING IF NEW LENGTH > 255
          CMPA      2,S         COMPARE TO MAXIMUM STRING LENGTH
          BLS       IDXLEN      BRANCH IF NEW LENGTH <= MAX LENGTH
          *
          *SUBSTRING DOES NOT FIT, SO TRUNCATE IT
          *
TRUNC:
          LDB       2,S         NUMBER OF CHARACTERS TO INSERT =
          SUBB      [6,S]        MAXIMUM LENGTH - STRING LENGTH
          BLS       EREXIT      TAKE ERROR EXIT IF MAXIMUM LENGTH < =
```

```
                        * STRING LENGTH
        COM     ,S      INDICATE SUBSTRING WAS TRUNCATED
        *
        *CHECK WHETHER STARTING INDEX IS WITHIN THE STRING.  IF NOT,
        *  CONCATENATE SUBSTRING ONTO THE END OF THE STRING
        *
IDXLEN:
        STB     1,S     SAVE NUMBER OF CHARACTERS TO INSERT
        LDA     ,X      GET STRING LENGTH
        CMPA    3,S     COMPARE TO STARTING INDEX
        BCC     LENOK   BRANCH IF STARTING INDEX IS WITHIN STRING
        INCA            ELSE SET STARTING INDEX TO END OF STRING
        STA     3,S
        LDA     #$FF    INDICATE ERROR IN INSERT
        STA     ,S
        BRA     MVESUB  JUST PERFORM MOVE, NOTHING TO OPEN UP
        *
        *OPEN UP A SPACE IN SOURCE STRING FOR THE SUBSTRING BY MOVING
        *  THE CHARACTERS FROM THE END OF THE SOURCE STRING DOWN TO
        *  INDEX, UP BY THE SIZE OF THE SUBSTRING
        *
LENOK:
        *
        *CALCULATE NUMBER OF CHARACTERS TO MOVE
        *  COUNT := STRING LENGTH - STARTING INDEX + 1
        *
        LDB     ,X      GET STRING LENGTH
        SUBB    2,S     SUBTRACT STARTING INDEX
        INCB            ADD 1
        *
        *SET SOURCE AND DESTINATION POINTERS
        *
        LEAX    A,X     POINT TO END OF STRING
        LEAX    1,X     POINT JUST PAST END OF STRING
        LDA     1,S     ADD NUMBER OF CHARACTERS TO INSERT
        LEAY    A,X     POINT JUST PAST END OF DESTINATION AREA
        *
        *MOVE CHARACTERS UP IN MEMORY TO MAKE ROOM FOR SUBSTRING
        *
OPNLP:
        LDA     ,-X     GET NEXT CHARACTER
        STA     ,-Y     MOVE IT UP IN MEMORY
        DECB            DECREMENT COUNTER
        BNE     OPNLP   CONTINUE THROUGH NUMBER OF CHARACTERS
                        * TO MOVE
        *
        *MOVE SUBSTRING INTO THE OPEN AREA
        *
MVESUB:
        LDX     6,S     GET STRING ADDRESS
        LDA     3,S     GET STARTING INDEX
        LEAX    A,X     POINT TO START OF OPEN AREA
        LDY     4,S     GET SUBSTRING ADDRESS
        LDB     1,S     GET NUMBER OF CHARACTERS TO INSERT
        LEAY    1,Y     POINT TO START OF SUBSTRING
        *
```

```
          *MOVE SUBSTRING BYTE AT A TIME
          *
MVELP:
          LDA       ,Y+         GET CHARACTER FROM SUBSTRING
          STA       ,X+         MOVE IT INTO OPEN AREA
          DECB                  DECREMENT COUNTER
          BNE       MVELP       CONTINUE UNTIL COUNTER = 0
          *
          *CALCULATE NEW STRING LENGTH
          *NEW LENGTH = OLD LENGTH PLUS NUMBER OF CHARACTERS
          * TO INSERT
          *
          LDX       6,S         POINT TO STRING LENGTH
          LDA       ,X          GET STRING LENGTH
          ADDA      1,S         ADD NUMBER OF CHARACTERS TO INSERT
          STA       ,X          SAVE SUM AS NEW STRING LENGTH
          *
          *CHECK ERROR FLAG
          *
          LDA       ,S          CHECK ERROR FLAG
          BNE       EREXIT      BRANCH IF ERROR OCCURRED
          *
          *SET CARRY FROM ERROR FLAG OR TEST
          *CARRY = 0 IF NO ERRORS, 1 IF ERRORS
          *
OKEXIT:
          CLC                   NO ERRORS
          BRA       EXITIN
EREXIT:
          SEC                   ERROR EXIT
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
EXITIN:
          LEAS      8,S         REMOVE PARAMETERS FROM STACK
          JMP       ,U          EXIT TO RETURN ADDRESS

*
*       SAMPLE EXECUTION:
*


SC5F:
          LDY       #STG        BASE ADDRESS OF STRING
          LDX       #SSTG       BASE ADDRESS OF SUBSTRING
          LDB       IDX         STARTING INDEX
          LDA       MXLEN       MAXIMUM LENGTH OF STRING
          PSHS      D,X,Y       SAVE PARAMETERS IN STACK
          JSR       INSERT      INSERT SUBSTRING
                        *RESULT OF INSERTING '-' INTO '123456' AT
                        * INDEX 1 IS '-123456'
          JMP       SC5F        LOOP THROUGH TEST

*
*DATA SECTION
IDX:      FCB       1           STARTING INDEX FOR INSERTION
```

```
MXLEN:  FCB     $20     MAXIMUM LENGTH OF DESTINATION
STG:    FCB     6       LENGTH OF STRING
        FCC     /123456                         / 32 BYTE MAX
SSTG    FCB     1       LENGTH OF SUBSTRING
        FCC     /-                              / 32 BYTE MAX

        END
```

5G Remove excess spaces from a string (SPACES)

Removes excess spaces from a string, including leading spaces, trailing spaces, and extra spaces within the string itself. The string consists of at most 256 bytes, including an initial byte containing the length.

Procedure The program exits immediately if the length of the string is 0. Otherwise, it first removes all leading spaces. It then sets a flag whenever it finds a space and deletes all subsequent spaces. If it reaches the end of the string with that flag set, it deletes the final trailing space as well. Finally, it adjusts the string's length.

Entry conditions

Base address of string in register X

Exit conditions

Excess spaces removed from string. The string is left with no leading or trailing spaces and no groups of consecutive spaces inside it.

Examples

1. Data: String = 0F' JOHN SMITH ' ($0F_{16} = 15_{10}$ is the length of the string)
   Result: String = 0A'JOHN SMITH' ($0A_{16} = 10_{10}$ is the length of the string with the extra spaces removed)
2. Data: String = 1B' PORTLAND, OREGON '($1B_{16} = 27_{10}$ is the length of the string)
   Result: String = 10'PORTLAND, OREGON' ($10_{16} = 16_{10}$ is the length of the string with the extra spaces removed)

Registers used All

Execution time Approximately

35 × LENGTH OF STRING IN BYTES + 65

If, for example, the string is 1C hex (28 decimal) bytes long, this is

35 × 28 + 65 = 980 + 65 = 1045 cycles

Program size 61 bytes

Data memory required 2 stack bytes

```
*
*       Title           Remove Extra Spaces from a String
*       Name:           SPACES
*
*
*       Purpose:        Remove leading, trailing, and extra
*                       internal spaces from a string
*
*       Entry:          Register X = Base address of string
*
*                         The string consists of a length byte
*                         followed by a maximum of 255 characters.
*
*       Exit:           Leading, trailing, and excess internal
*                       spaces removed
*
*       Registers Used: All
*
*       Time:           Approximately
*                        35 * (LENGTH(STRG) + 65 cycles overhead
*
*       Size:           Program 61 bytes
*                       Data     2 stack bytes
*
*

SPACES:
          *
          *SAVE BASE ADDRESS OF STRING
          *START COMPACTED STRING'S LENGTH AT ZERO
          *INDICATE INITIALLY LAST CHARACTER WAS NOT A SPACE
          *
          TFR       X,U         SAVE BASE ADDRESS OF STRING
          CLRA                  INDICATE LAST CHARACTER WAS NOT A SPACE
          CLRB                  COMPACTED STRING'S LENGTH = ZERO
          PSHS      A,B         SAVE INDICATOR, LENGTH IN STACK
          *
          *EXIT IF STRING LENGTH IS ZERO
          *
          LDB       ,X+         GET STRING LENGTH
```

```
        BEQ       EXITRE    BRANCH (EXIT) IF STRING LENGTH IS ZERO
        *
        *REMOVE ALL LEADING SPACES
        *
        TFR     X,Y         START POINTERS TO BOTH ORIGINAL, COMPACTED
                            * STRINGS AT FIRST CHARACTER IN STRING
LEADSP:
        LDA       ,X+       GET NEXT CHARACTER
        CMPA      #SPACE    IS IT A SPACE?
        BNE       MARKCH    BRANCH IF CHARACTER IS NOT A SPACE
        DECB                DECREMENT CHARACTER COUNT
        BNE       LEADSP    BRANCH IF NOT DONE WITH STRING
        CLR       ,U        STRING CONSISTED ENTIRELY OF SPACES
                            * MAKE ITS LENGTH ZERO
        BRA       EXITRE    EXIT
        *
        *WORK THROUGH MAIN PART OF STRING, OMITTING SPACES
        * THAT OCCUR IMMEDIATELY AFTER OTHER SPACES
        *
        *CHECK IF CURRENT CHARACTER IS A SPACE
        *IF SO, CHECK IF PREVIOUS CHARACTER WAS A SPACE
        *IF SO, OMIT CHARACTER FROM COMPACTED STRING
        *IF NOT, MARK CHARACTER AS A SPACE
        *
MVCHAR:
        LDA       ,X+       GET NEXT CHARACTER
        CMPA      #SPACE    IS IT A SPACE?
        BNE       MARKCH    BRANCH IF CHARACTER IS NOT A SPACE
        TST       ,S        CHECK IF LAST CHARACTER WAS A SPACE
        BEQ       CNTCHR    BRANCH IF IT WAS
        COM       ,S        INDICATE CURRENT CHARACTER IS A SPACE
        BRA       SVCHR
        *
        *INDICATE CURRENT CHARACTER IS NOT A SPACE
        *
MARKCH:
        CLR       ,S        INDICATE CURRENT CHARACTER NOT A SPACE
        *
        *SAVE CURRENT CHARACTER IN COMPACTED STRING
        *
SVCHR:
        STA       ,Y+       SAVE CHARACTER IN COMPACTED STRING
        INC       1,S       ADD 1 TO LENGTH OF COMPACTED STRING
        *
        *COUNT CHARACTERS
        *
CNTCHR:
        DECB                COUNT CHARACTERS
        BNE       MVCHAR    BRANCH IF ANY CHARACTERS LEFT
        *
        *OMIT LAST CHARACTER IF IT WAS A SPACE
        *
        TST       ,S        CHECK IF FINAL CHARACTER WAS A SPACE
        BEQ       SETLEN    BRANCH IF IT WAS NOT
        DEC       1,S       OMIT FINAL CHARACTER IF IT WAS A SPACE
        *
```

```
          *SET LENGTH OF COMPACTED STRING
          *
SETLEN:
          LDA       1,S       GET LENGTH OF COMPACTED STRING
          STA       ,U        SAVE AS LENGTH BYTE IN STRING
          *
          *REMOVE TEMPORARIES FROM STACK AND EXIT
          *
EXITRE:
          LEAS      2,S       REMOVE TEMPORARY DATA FROM STACK
          RTS

*
*CHARACTER DEFINITION
*
SPACE     EQU       $20       ASCII SPACE CHARACTER


*
*         SAMPLE EXECUTION:
*


SC5G:
          LDX       #STG      GET BASE ADDRESS OF STRING
          JSR       SPACES    REMOVE SPACES
                              *RESULT OF REMOVING SPACES FROM
                              * ' JOHN  SMITH  ' IS 'JOHN SMITH'
*
*DATA SECTION
*
STG:      FCB       $0E       LENGTH OF STRING IN BYTES
          FCC       / JOHN  SMITH  /   STRING

          END
```

6 Array operations

6A 8-bit array summation (ASUM8)

Adds the elements of an array, producing a 16-bit sum. The array consists of up to 255 byte-length elements.

Procedure The program starts the sum at 0. It then adds elements one at a time to the sum's less significant byte. It also adds the carries to the sum's more significant byte.

Entry conditions

Base address of array in register X
Size of array in bytes in register A

Exit conditions

Sum in register D

Example

Data: Size of array in bytes = (A) = 08

Array elements
$F7_{16} = 247_{10}$
$23_{16} = 35_{10}$
$31_{16} = 49_{10}$
$70_{16} = 112_{10}$
$5A_{16} = 90_{10}$
$16_{16} = 22_{10}$
$CB_{16} = 203_{10}$
$E1_{16} = 225_{10}$
Result: Sum = (D) = $03D7_{16} = 983_{10}$

Registers used A, B, CC, X, Y

Execution time Approximately 16 cycles per byte-length element plus 26 cycles overhead. If, for example, the array consists of $1C_{16}$ (28_{10}) elements, the execution time is approximately

$16 \times 28 + 26 = 448 + 26 = 474$ cycles

Program size 18 bytes

Data memory required None

Special case An array size of 0 causes an immediate exit with a sum of 0

```
*
*
*       Title           8-Bit Array Summation
*       Name:           ASUM8
*
*
*       Purpose:        Sum the elements of an array, yielding a 16 bit
*                       result. Maximum size is 255 byte-length
*                       elements.
*
*       Entry:          Register X = Base address of array
*                       Register A = Size of array in bytes
*
*       Exit:           Register D = Sum
*
```

```
*       Registers Used: A,B,CC,X,Y
*
*       Time:           Approximately 16 cycles per element plus
*                       26 cycles overhead
*
*       Size:           Program 18 bytes
*

          *
          *TEST ARRAY LENGTH
          *EXIT WITH SUM = 0 IF ARRAY HAS ZERO ELEMENTS
          *
ASUM8:
          TFR       A,B             SAVE ARRAY LENGTH IN B
          CLRA                      EXTEND ARRAY LENGTH TO 16 BITS
          TSTB                      CHECK IF ARRAY LENGTH IS ZERO
          BEQ       EXITAS          BRANCH (EXIT) IF ARRAY LENGTH IS
                                    * ZERO - SUM IS ZERO IN THIS CASE
          *
          *ADD BYTE-LENGTH ELEMENTS TO LOW BYTE OF SUM ONE AT A TIME
          *ADD CARRIES TO HIGH BYTE OF SUM
          *
          TFR       D,Y             SAVE 16-BIT ARRAY LENGTH IN Y
          CLRB                      START SUM AT ZERO (REMEMBER A IS
                                    * ALREADY ZERO)
SUMLP:
          ADDB      ,X+             ADD NEXT ELEMENT TO LOW BYTE OF
                                    * SUM
          ADCA      #0              ADD CARRY TO HIGH BYTE OF SUM
          LEAY      -1,Y            CONTINUE THROUGH ALL ELEMENTS
          BNE       SUMLP
EXITAS:
          RTS


*
*       SAMPLE EXECUTION
*
*
SC6A:
          LDX       #BUF            GET BASE ADDRESS OF BUFFER
          LDA       BUFSZ           GET BUFFER SIZE IN BYTES
          JSR       ASUM8           SUM ELEMENTS IN BUFFER
                                    SUM OF TEST DATA IS 07F8 HEX,
                                    * REGISTER D = 07F8H
          BRA       SC6A            LOOP FOR ANOTHER TEST

*TEST DATA, CHANGE FOR OTHER VALUES
SIZE      EQU       $10             SIZE OF BUFFER IN BYTES
BUFSZ:    FCB       SIZE            SIZE OF BUFFER IN BYTES

BUF:      FCB       0               BUFFER
          FCB       $11             DECIMAL ELEMENTS ARE 0,17,34,51,68
          FCB       $22             85,102,119,135,153,170,187,204
          FCB       $33             221,238,255
          FCB       $44
```

```
FCB     $55
FCB     $66
FCB     $77
FCB     $88
FCB     $99
FCB     $AA
FCB     $BB
FCB     $CC
FCB     $DD
FCB     $EE
FCB     $FF                     SUM = 07F8 (2040 DECIMAL)

END
```

6B 16-bit array summation (ASUM16)

Adds the elements of an array, producing a 24-bit sum. The array consists of up to 255 word-length (16-bit) elements arranged in the usual 6809 format with the more significant byte first.

Procedure The program starts the sum at 0. It then adds elements to the sum's less significant bytes one at a time, beginning at the base address. Whenever an addition produces a carry, the program adds 1 to the sum's most significant byte.

Entry conditions

Base address of array in X
Size of array in 16-bit words in A

Exit conditions

Most significant byte of sum in A
Middle and least significant bytes of sum in X

Example

Data:	Size of array (in 16-bit words) = (A) = 08
	Array elements
	$F7A1_{16} = 63\,393_{10}$
	$239B_{16} = 9\,115_{10}$
	$31D5_{16} = 12\,757_{10}$
	$70F2_{16} = 28\,914_{10}$
	$5A36_{16} = 23\,094_{10}$
	$166C_{16} = 5\,740_{10}$
	$CBF5_{16} = 52\,213_{10}$
	$E107_{16} = 57\,607_{10}$
Result:	Sum = $03DBA1_{16} = 252\,833_{10}$
	(A) = most significant byte of sum = 03_{16}
	(X) = middle and least significant bytes of sum = $DBA1_{16}$

Registers used A, B, CC, X, Y

Execution time Approximately 20 cycles per 16-bit element plus 44 cycles overhead. If, for example, the array consists of 12_{16} (18_{10}) elements, the execution time is approximately

$20 \times 18 + 44 = 360 + 44 = 404$ cycles

This approximation assumes no carries to the most significant byte of the sum; each carry increases execution time by 6 cycles.

Program size 27 bytes

Data memory required 1 stack byte

Special case An array size of 0 causes an immediate exit with a sum of 0

```
*
*
*       Title           16-Bit Array Summation
*       Name:           ASUM16
*
*       Purpose:        Sum the elements of an array, yielding a 24 bit
*                       result. Maximum size is 255 16-bit elements.
*
*       Entry:          Register X = Base address of array
*                       Register A = Size of array (in 16-bit words)
*
*       Exit:           Register A = High byte of sum
*                       Register X = Middle and low bytes of sum
*
*       Registers Used: A,B,CC,X,Y
*
*       Time:           Approximately 20 cycles per element plus
*                       44 cycles overhead
*
*       Size:           Program 27 bytes
*                       Data     1 stack byte
*
*

ASUM16:
          *
          *TEST ARRAY LENGTH
          *EXIT WITH SUM = 0 IF ARRAY HAS NO ELEMENTS
          *
          TFR       A,B             MOVE ARRAY LENGTH TO B
```

```
          CLRA                        EXTEND ARRAY LENGTH TO 16 BITS
          STA       ,-S               MAKE MSB OF SUM ZERO
          TSTB                        CHECK ARRAY LENGTH
          BEQ       EXITS1            BRANCH (EXIT) IF ARRAY LENGTH IS ZERO
                                      * SUM IS ZERO IN THIS CASE
          *
          *ADD WORD-LENGTH ELEMENTS TO LOW BYTES OF SUM ONE AT A TIME
          *ADD 1 TO HIGH BYTE OF SUM WHENEVER A CARRY OCCURS
          *
          TFR       D,Y               MOVE 16-BIT ARRAY LENGTH TO Y
          CLRB                        START SUM AT ZERO (REMEMBER A IS
                                      * ALREADY ZERO)
SUMLP:    ADDD      ,X++              ADD ELEMENT TO LOW BYTES OF SUM
          BCC       DECCNT            BRANCH IF NO CARRY
          INC       ,S                ELSE ADD 1 TO HIGH BYTE OF SUM
DECCNT:
          LEAY      -1,Y              CONTINUE THROUGH ALL ELEMENTS
          BNE       SUMLP
          *
          *MOVE SUM TO A (MOST SIGNIFICANT BYTE) AND X (LESS SIGNIFICANT
          *  BYTES)
          *
EXITS1:
          TFR       D,X               SAVE LOW BYTES OF SUM IN X
          LDA       ,S+               MOVE HIGH BYTE OF SUM TO A
          RTS

*
*         SAMPLE EXECUTION
*

*
SC6B:
          LDX       #BUF              GET BASE ADDRESS OF BUFFER
          LDA       BUFSZ             GET SIZE OF BUFFER IN WORDS
          JSR       ASUM16            SUM WORD-LENGTH ELEMENTS IN BUFFER
                                      * SUM OF TEST DATA IS 31FF8 HEX,
                                      * REGISTER X = 1FF8H
                                      * REGISTER A = 3
          BRA       SC6B              LOOP FOR ANOTHER TEST

*TEST DATA, CHANGE FOR OTHER VALUES
SIZE      EQU       $10               SIZE OF BUFFER IN WORDS
BUFSZ:    FCB       SIZE              SIZE OF BUFFER IN WORDS

BUF:      FDB       0                 BUFFER
          FDB       $111              DECIMAL ELEMENTS ARE 0,273,546,819,1092
          FDB       $222              1365,1638,1911,2184,2457,2730,3003,3276
          FDB       $333              56797,61166,65535
          FDB       $444
          FDB       $555
          FDB       $666
          FDB       $777
          FDB       $888
          FDB       $999
          FDB       $AAA
```

```
FDB         $BBB
FDB         $CCC
FDB         $DDDD
FDB         $EEEE
FDB         $FFFF           SUM = 31FF8 (204792 DECIMAL)

END
```

6C Find maximum byte-length element (MAXELM)

Finds the maximum element in an array. The array consists of up to 255 unsigned byte-length elements.

Procedure The program exits immediately (setting Carry to 1) if the array has no elements. Otherwise, the program assumes that the element at the base address is the maximum. It then works through the array, comparing the supposed maximum with each element and retaining the larger value and its address. Finally, the program clears Carry to indicate a valid result.

Entry conditions

Base address of array in register X
Size of array in bytes in register A

Exit conditions

Largest unsigned element in register A
Address of largest unsigned element in register X

Carry = 0 if result is valid, 1 if size of array is 0 and result is meaningless

Example

Data:	Size of array (in bytes) = (A) = 08
	Array elements
	$35_{16} = 53_{10}$ $44_{16} = 68_{10}$
	$A6_{16} = 166_{10}$ $59_{16} = 89_{10}$
	$D2_{16} = 210_{10}$ $7A_{16} = 122_{10}$
	$1B_{16} = 27_{10}$ $CF_{16} = 207_{10}$
Result:	The largest unsigned element is element #2
	($D2_{16} = 210_{10}$)
	(B) = largest element ($D2_{16}$)
	(X) = BASE + 2 (lowest address containing $D2_{16}$)
	Carry = 0, indicating that array size is non-zero and the result is valid

Registers used A, B, CC, X, Y

Execution time Approximately 14 to 26 cycles per element plus 27 cycles overhead. The larger number applies when the program must replace the previous maximum and its address with the current element and its address. If, on the average, that replacement is necessary in half of the iterations, the time is approximately

(14+26)/2 × ARRAY SIZE/2 + 27 cycles

If, for example, ARRAY SIZE = 18_{16} = 24_{10} bytes, the approximate execution time is

40/2 × 12 + 27 = 240 + 27 = 267 cycles

Program size 25 bytes

Data memory required None

Special cases

1. An array size of 0 causes an immediate exit with the Carry flag set to 1 to indicate an invalid result.

2. If the largest unsigned value occurs more than once, the program returns with the lowest possible address. That is, it returns with the address closest to the base address that contains the maximum value.

```
*
*
*       Title           Find Maximum Byte-Length Element
*       Name:           MAXELM
*
*
*       Purpose:        Given the base address and size of an array,
*                       find the largest element.
*
*       Entry:          Register X = Base address of array
*                       Register A = Size of array in bytes
*
*       Exit:           If size of array not zero then
*                         Carry flag = 0
*                         Register A = Largest element
*                         Register X = Address of that element
*                          If there are duplicate values of the largest
```

```
*                          element, register X contains the address
*                          nearest to the base address.
*                        else
*                          Carry flag = 1
*
*       Registers Used: A,B,CC,X,Y
*
*       Time:           Approximately 14 to 26 cycles per byte
*                       plus 27 cycles overhead
*
*       Size:           Program 25 bytes
*
*

MAXELM:
          *
          *EXIT WITH CARRY SET IF NO ELEMENTS IN ARRAY
          *
          SEC                   SET CARRY IN CASE ARRAY HAS NO ELEMENTS
          TSTA                  CHECK NUMBER OF ELEMENTS
          BEQ       EXITMX      BRANCH (EXIT) WITH CARRY SET IF NO
                                * ELEMENTS - INDICATES INVALID RESULT
          *
          *EXAMINE ELEMENTS ONE AT A TIME, COMPARING EACH ONE'S VALUE
          * WITH CURRENT MAXIMUM AND ALWAYS KEEPING LARGER VALUE AND
          * ITS ADDRESS.  IN THE FIRST ITERATION, TAKE THE FIRST
          * ELEMENT AS THE CURRENT MAXIMUM.
          *
          TFR       A,B         SAVE NUMBER OF ELEMENTS IN B
          LEAY      1,X         SET POINTER AS IF PROGRAM HAD JUST
                                * EXAMINED THE FIRST ELEMENT AND FOUND
                                * IT TO BE LARGER THAN PREVIOUS MAXIMUM
MAXLP:
          LEAX      -1,Y        SAVE ADDRESS OF ELEMENT JUST EXAMINED
                                * AS ADDRESS OF MAXIMUM
          LDA       ,X          SAVE ELEMENT JUST EXAMINED AS MAXIMUM
          *
          *COMPARE CURRENT ELEMENT TO MAXIMUM
          *KEEP LOOKING UNLESS CURRENT ELEMENT IS LARGER
          *
MAXLP1:
          DECB                  COUNT ELEMENTS
          BEQ       EXITLP      BRANCH (EXIT) IF ALL ELEMENTS EXAMINED
          CMPA      ,Y+         COMPARE CURRENT ELEMENT TO MAXIMUM
                                * ALSO MOVE POINTER TO NEXT ELEMENT
          BCC       MAXLP1      CONTINUE UNLESS CURRENT ELEMENT LARGER
          BCS       MAXLP       ELSE CHANGE MAXIMUM
          *
          *CLEAR CARRY TO INDICATE VALID RESULT - MAXIMUM FOUND
          *
EXITLP:
          CLC                   CLEAR CARRY TO INDICATE VALID RESULT
EXITMX:
          RTS

*
```

```
*         SAMPLE EXECUTION:
*
SC6C:
          LDX       #ARY        GET BASE ADDRESS OF ARRAY
          LDA       #SZARY      GET SIZE OF ARRAY IN BYTES
          JSR       MAXELM      FIND LARGEST UNSIGNED ELEMENT
                                *RESULT FOR TEST DATA IS
                                * A = FF HEX (MAXIMUM), X = ADDRESS OF
                                * FF IN ARY.
          BRA       SC6C        LOOP FOR MORE TESTING

SZARY     EQU       $10         SIZE OF ARRAY IN BYTES
ARY:      FCB       8
          FCB       7
          FCB       6
          FCB       5
          FCB       4
          FCB       3
          FCB       2
          FCB       1
          FCB       $FF
          FCB       $FE
          FCB       $FD
          FCB       $FC
          FCB       $FB
          FCB       $FA
          FCB       $F9
          FCB       $F8

          END
```

6D Find minimum byte-length element (MINELM)

Finds the minimum element in an array. The array consists of up to 255 unsigned byte-length elements.

Procedure The program exits immediately (setting Carry to 1) if the array has no elements. Otherwise, the program assumes that the element at the base address is the minimum. It then works through the array, comparing the current minimum to each element and retaining the smaller value and its address. Finally, the program clears Carry to indicate a valid result.

Entry conditions

Base address of array in register X
Size of array in bytes in register A

Exit conditions

Smallest unsigned element in register A
Address of smallest unsigned element in register X

Carry = 0 if result is valid, 1 if size of array is 0 and result is meaningless

Example

Data:	Size of array (in bytes) = (A) = 08
	Array elements
	$35_{16} = 53_{10}$ $44_{16} = 68_{10}$
	$A6_{16} = 166_{10}$ $59_{16} = 89_{10}$
	$D2_{16} = 210_{10}$ $7A_{16} = 122_{10}$
	$1B_{16} = 27_{10}$ $CF_{16} = 207_{10}$
Result:	The smallest unsigned element is element #3
	($1B_{16} = 27_{10}$)
	(A) = smallest element ($1B_{16}$)
	(X) = BASE + 3 (lowest address containing $1B_{16}$)
	Carry flag = 0, indicating that array size is non-zero and the result is valid

Registers used A, B, CC, X, Y

Execution time Approximately 14 to 26 cycles per element plus 27 cycles overhead. The larger number of cycles applies when the program must replace the previous minimum and its address with the current element and its address. If, on the average, that replacement is necessary in half of the iterations, the execution time is approximately

$(14 + 26)/2 \times \text{ARRAY SIZE}/2 + 27$ cycles

If, for example, ARRAY SIZE = $14_{16} = 20_{10}$, the approximate execution time is

$40/2 \times 10 + 27 = 200 + 27 = 227$ cycles

Program size 25 bytes

Data memory required None

Special cases

1. An array size of 0 causes an immediate exit with the Carry flag set to 1 to indicate an invalid result.

2. If the smallest unsigned value occurs more than once, the program returns with the lowest possible address. That is, it returns with the address closest to the base address that contains the minimum value.

```
*
*
*       Title           Find Minimum Byte-Length Element
*       Name:           MINELM
*
*
*       Purpose:        Given the base address and size of an array,
*                       find the smallest element
*
*       Entry:          Register X = Base address of array
*                       Register A = Size of array in bytes
*
*       Exit:           If size of array not zero then
*                         Carry flag = 0
*                         Register A = Smallest element
*                         Register X = Address of that element
*                          If there are duplicate values of the smallest
```

```
*                          element, register X contains the address
*                           nearest to the base address.
*                        else
*                          Carry flag = 1
*
*       Registers Used: A,B,CC,X,Y
*
*       Time:           Approximately 14 to 26 cycles per byte
*                       plus 27 cycles overhead
*
*       Size:           Program 25 bytes
*
MINELM:
          *
          *EXIT WITH CARRY SET IF ARRAY CONTAINS NO ELEMENTS
          *
          SEC                   SET CARRY IN CASE ARRAY HAS NO ELEMENTS
          TSTA                  CHECK NUMBER OF ELEMENTS
          BEQ       EXITMN      BRANCH (EXIT) WITH CARRY SET IF NO
                                * ELEMENTS - INDICATES INVALID RESULT
          *
          *EXAMINE ELEMENTS ONE AT A TIME, COMPARING EACH VALUE WITH
          * THE CURRENT MINIMUM AND ALWAYS KEEPING THE SMALLER VALUE
          * AND ITS ADDRESS.  IN THE FIRST ITERATION, TAKE THE FIRST
          * ELEMENT AS THE CURRENT MINIMUM.
          *
          TFR       A,B         SAVE NUMBER OF ELEMENTS IN B
          LEAY      1,X         SET POINTER AS IF PROGRAM HAD JUST
                                * EXAMINED THE FIRST ELEMENT
MINLP:
          LEAX      -1,Y        SAVE ADDRESS OF ELEMENT JUST EXAMINED
                                * AS ADDRESS OF MINIMUM
          LDA       ,X          SAVE ELEMENT JUST EXAMINED AS MINIMUM
          *
          *COMPARE CURRENT ELEMENT TO SMALLEST
          *KEEP LOOKING UNLESS CURRENT ELEMENT IS SMALLER
          *
MINLP1:
          DECB                  COUNT ELEMENTS
          BEQ       EXITLP      BRANCH (EXIT) IF ALL ELEMENTS EXAMINED
          CMPA      ,Y+         COMPARE CURRENT ELEMENT TO MINIMUM
          BLS       MINLP1      CONTINUE UNLESS CURRENT ELEMENT SMALLER
          BHI       MINLP       ELSE CHANGE MINIMUM
          *
          *CLEAR CARRY TO INDICATE VALID RESULT - MINIMUM FOUND
          *
EXITLP:
          CLC                   CLEAR CARRY TO INDICATE VALID RESULT
EXITMN:
          RTS

*
*         SAMPLE EXECUTION:
*
```

```
SC6D:
          LDX       #ARY        GET BASE ADDRESS OF ARRAY
          LDA       #SZARY      GET SIZE OF ARRAY IN BYTES
          JSR       MINELM      FIND MINIMUM VALUE IN ARRAY
                                *RESULT FOR TEST DATA IS
                                * A = 1 HEX (MINIMUM), X = ADDRESS OF
                                * 1 IN ARY.
          BRA       SC6D        LOOP FOR ANOTHER TEST

SZARY     EQU       $10         SIZE OF ARRAY IN BYTES
ARY:      FCB       8
          FCB       7
          FCB       6
          FCB       5
          FCB       4
          FCB       3
          FCB       2
          FCB       1
          FCB       $FF
          FCB       $FE
          FCB       $FD
          FCB       $FC
          FCB       $FB
          FCB       $FA
          FCB       $F9
          FCB       $F8

          END
```

6E Binary search (BINSCH)

Searches an array of unsigned byte-length elements for a particular value. The elements are assumed to be arranged in increasing order. Clears Carry if it finds the value and sets Carry to 1 if it does not. Returns the address of the value if found. The size of the array is specified and is a maximum of 255 bytes.

Procedure The program performs a binary search, repeatedly comparing the value with the middle remaining element. After each comparison, the program discards the part of the array that cannot contain the value (because of the ordering). The program retains upper and lower bounds for the part still being searched. If the value is larger than the middle element, the program discards that element and everything below it. The new lower bound is the address of the middle element plus 1. If the value is smaller than the middle element, the program discards that element and everything above it. The new upper bound is the address of the middle element minus 1. The program exits if it finds a match or if there is nothing left to search.

For example, assume that the array is

01_{16}, 02_{16}, 05_{16}, 07_{16}, 09_{16}, 09_{16}, $0D_{16}$, 10_{16}, $2E_{16}$, 37_{16}, $5D_{16}$, $7E_{16}$, $A1_{16}$, $B4_{16}$, $D7_{16}$, $E0_{16}$

and the value being sought is $0D_{16}$. The procedure works as follows.

In the first iteration, the lower bound is the base address and the upper bound is the address of the last element. So we have

LOWER BOUND = BASE
UPPER BOUND = BASE + LENGTH − 1 = BASE + $0F_{16}$
GUESS = (UPPER BOUND + LOWER BOUND)/2
= BASE + 7 (the result is truncated)
(GUESS) = ARRAY(7) = 10_{16} = 16_{10}

Since the value ($0D_{16}$) is less than ARRAY(7), we can discard the elements beyond #6. So we have

LOWER BOUND = BASE
UPPER BOUND = GUESS − 1 = BASE + 6
GUESS = (UPPER BOUND + LOWER BOUND)/2 = BASE + 3
(GUESS) = ARRAY(3) = 07

Since the value ($0D_{16}$) is greater than ARRAY(3), we can discard the

elements below #4. So we have

LOWER BOUND = GUESS + 1 = BASE + 4
UPPER BOUND = BASE + 6
GUESS = (UPPER BOUND + LOWER BOUND)/2 = BASE + 5
(GUESS) = ARRAY(5) = 09

Since the value ($0D_{16}$) is greater than ARRAY(5), we can discard the elements below #6. So we have
LOWER BOUND = GUESS + 1 = BASE + 6
UPPER BOUND = BASE + 6
GUESS = (UPPER BOUND + LOWER BOUND)/2 = BASE + 6
(GUESS) = ARRAY(6) = $0D_{16}$

Since the value ($0D_{16}$) is equal to ARRAY(6), we have found the element. If, on the other hand, the value were $0E_{16}$, the new lower bound would be BASE + 7 and there would be nothing left to search.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

Value to find

Size of the array in bytes

More significant byte of base address of array (address of smallest unsigned element)
Less significant byte of base address of array (address of smallest unsigned element)

Exit conditions

Carry = 0 if the value is found, 1 if it is not found. If the value is found, (X) = its address.

Examples

Length of array = $10_{16} = 16_{10}$
Elements of array are 01_{16}, 02_{16}, 05_{16}, 07_{16}, 09_{16}, 09_{16}, $0D_{16}$, 10_{16}, $2E_{16}$,

37_{16}, $5D_{16}$, $7E_{16}$, $A1_{16}$, $B4_{16}$, $D7_{16}$, $E0_{16}$

1. Data: Value to find = $0D_{16}$
   Result: Carry = 0, indicating value found
   (X) = BASE + 6 (address containing $0D_{16}$)
2. Data: Value to find = $9B_{16}$
   Result: Carry = 1, indicating value not found

Registers used All

Execution time Approximately 50 cycles per iteration plus 50 cycles overhead. A binary search will require on the order of $\log_2 N$ iterations, where N is the number of elements in the array.

If, for example, $N = 32$, the binary search will require approximately $\log_2 32 = 5$ iterations. The execution time will then be approximately

$50 \times 5 + 50 = 250 + 50 = 300$ cycles

Program size 64 bytes

Data memory required None

Special case A size of 0 causes an immediate exit with the Carry flag set to 1. That is, the array contains no elements and the value surely cannot be found.

```
*
*
*       Title           Binary Search
*       Name:           BINSCH
*
*
*       Purpose:        Search an ordered array of unsigned bytes,
*                       with a maximum size of 255 elements.
*
*       Entry:          TOP OF STACK
*                         High byte of return address
*                         Low byte of return address
*                         Value to find
*                         Length (size) of array
*                         High byte of base address of array
```

```
*                          Low byte of base address of array
*
*       Exit:              If the value is found then
*                            Carry flag = 0
*                            Register X = Address of value
*                          Else
*                            Carry flag = 1
*
*       Registers Used:    All
*
*       Time:              Approximately 50 cycles for each iteration of
*                          the search loop plus 50 cycles overhead
*
*                          A binary search takes on the order of log
*                          base 2 of N searches, where N is the number of
*                          elements in the array.
*
*       Size:              Program 64 bytes
*

*
BINSCH:
          *
          *EXIT WITH CARRY SET IF ARRAY CONTAINS NO ELEMENTS
          *
          LDU       ,S          SAVE RETURN ADDRESS
          SEC                   SET CARRY IN CASE ARRAY HAS NO ELEMENTS
          LDB       3,S         CHECK NUMBER OF ELEMENTS
          BEQ       EXITBS      BRANCH (EXIT) WITH CARRY SET IF NO
                                * ELEMENTS - VALUE SURELY CANNOT BE FOUND
          *
          *INITIALIZE INDEXES OF UPPER BOUND, LOWER BOUND
          *LOWER BOUND = BASE ADDRESS
          *UPPER BOUND = ADDRESS OF LAST ELEMENT =
          * BASE ADDRESS + SIZE - 1
          *
          DECB                  INDEX OF UPPER BOUND = NUMBER OF
          STB       1,S           ELEMENTS - 1
          CLR       ,S          INDEX OF LOWER BOUND = 0 INITIALLY
          LDX       4,S         GET BASE ADDRESS OF ARRAY
          *
          *ITERATION OF BINARY SEARCH
          *1) COMPARE VALUE TO MIDDLE ELEMENT
          *2) IF THEY ARE NOT EQUAL, DISCARD HALF THAT
          *   CANNOT POSSIBLY CONTAIN VALUE (BECAUSE OF ORDERING)
          *3) CONTINUE IF THERE IS ANYTHING LEFT TO SEARCH
          *
SRLOOP:
          LDA       ,S          ADD LOWER AND UPPER BOUND INDEXES
          ADDA      1,S
          RORA                  DIVIDE BY 2, TRUNCATING FRACTION
          *
          *IF INDEX OF MIDDLE ELEMENT IS GREATER THAN UPPER BOUND,
          * THEN ELEMENT IS NOT IN ARRAY
          *
          CMPA      1,S         COMPARE INDEX OF MIDDLE ELEMENT TO
```

```
                        * UPPER BOUND
        BHI     NOTFND  BRANCH (NOT FOUND) IF INDEX GREATER
                        * THAN UPPER BOUND
        *
        *IF INDEX OF MIDDLE ELEMENT IS LESS THAN LOWER BOUND, THEN
        * ELEMENT IS NOT IN ARRAY
        *
        CMPA    ,S      COMPARE INDEX OF MIDDLE ELEMENT TO
                        * LOWER BOUND
        BLO     NOTFND  BRANCH (NOT FOUND) IF INDEX LESS
                        * THAN LOWER BOUND
        *
        *CHECK IF MIDDLE ELEMENT IS THE VALUE BEING SOUGHT
        *
        LDB     A,X     GET ELEMENT WITH MIDDLE INDEX
        CMPB    2,S     COMPARE ELEMENT WITH VALUE SOUGHT
        BLO     RPLCLW  BRANCH IF VALUE LARGER THAN ELEMENT
        BEQ     FOUND   BRANCH IF VALUE FOUND
        *
        *VALUE IS SMALLER THAN ELEMENT WITH MIDDLE INDEX
        *MAKE MIDDLE INDEX - 1 INTO NEW UPPER BOUND
        *
        DECA            SUBTRACT 1 SINCE VALUE CAN ONLY BE
                        * FURTHER DOWN
        STA     1,S     SAVE DIFFERENCE AS NEW UPPER BOUND
        CMPA    #$FF    CONTINUE SEARCHING IF UPPER BOUND DOES
        BNE     SRLOOP   NOT UNDERFLOW
        BEQ     NOTFND  EXIT IF UPPER BOUND UNDERFLOWED
        *
        *VALUE IS LARGER THAN ELEMENT WITH MIDDLE INDEX
        *MAKE MIDDLE INDEX + 1 INTO NEW LOWER BOUND
        *
RPLCLW:
        INCA            ADD 1 SINCE VALUE CAN ONLY BE FURTHER UP
        STA     ,S      SAVE SUM AS NEW LOWER BOUND
        BNE     SRLOOP  CONTINUE SEARCHING IF LOWER BOUND DOES
                        * NOT OVERFLOW
        BEQ     NOTFND  EXIT IF LOWER BOUND OVERFLOWED
        *
        *FOUND THE VALUE - GET ITS ADDRESS AND CLEAR CARRY
        *
FOUND:
        LEAX    A,X     GET ADDRESS OF VALUE
        CLC             CLEAR CARRY, INDICATING VALUE FOUND
        BRA     EXITBS
        *
        *DID NOT FIND THE VALUE - SET CARRY TO INDICATE FAILURE
        *
NOTFND:
        SEC             SET CARRY, INDICATING VALUE NOT FOUND
        *
        *REMOVE PARAMETERS FROM STACK AND EXIT
        *
EXITBS:
        LEAS    6,S     REMOVE PARAMETERS FROM STACK
        JMP     ,U      EXIT TO RETURN ADDRESS
```

```
*
*          SAMPLE EXECUTION
*

*
SC6E:
           *SEARCH FOR A VALUE THAT IS IN THE ARRAY
           LDX       #BF             GET BASE ADDRESS OF BUFFER
           LDB       BFSZ       GET ARRAY SIZE IN BYTES
           LDA       #7         GET VALUE TO FIND
           PSHS      D,X        SAVE PARAMETERS IN STACK
           JSR       BINSCH     BINARY SEARCH
                                *CARRY FLAG = 0 (VALUE FOUND)
                                *X = ADDRESS OF 7 IN ARRAY

           *SEARCH FOR A VALUE THAT IS NOT IN THE ARRAY
           LDX       #BF             GET BASE ADDRESS OF BUFFER
           LDB       BFSZ       GET ARRAY SIZE IN BYTES
           LDA       #0         GET VALUE TO FIND
           PSHS      D,X        SAVE PARAMETERS IN STACK
           JSR       BINSCH     BINARY SEARCH
                                *CARRY FLAG = 1 (VALUE NOT FOUND)
           BRA       SC6E       LOOP FOR MORE TESTS
*
*DATA
*
SIZE       EQU       $10        SIZE OF BUFFER IN BYTES
BFSZ:      FCB       SIZE       SIZE OF BUFFER IN BYTES
BF:                             *BUFFER
           FCB       1
           FCB       2
           FCB       4
           FCB       5
           FCB       7
           FCB       9
           FCB       10
           FCB       11
           FCB       23
           FCB       50
           FCB       81
           FCB       123
           FCB       191
           FCB       199
           FCB       250
           FCB       255

           END
```

6F Quicksort (QSORT)

Arranges an array of unsigned word-length elements into ascending order using a quicksort algorithm. Each iteration selects an element and divides the array into two parts, one containing all elements larger than the selected element and the other containing all elements smaller than the selected element. Elements equal to the selected element may end up in either part. The parts are then sorted recursively in the same way. The algorithm continues until all parts contain either no elements or only one element. An alternative is to stop recursion when a part contains few enough elements (say, less than 20) to make a bubble sort practical.

The parameters are the array's base address, the address of its last element, and the lowest available stack address. The array can thus occupy all available memory, as long as there is room for the stack. Since the procedures that obtain the selected element, compare elements, move forward and backward in the array, and swap elements are all subroutines, they could be changed readily to handle other types of elements.

Ideally, quicksort should divide the array in half during each iteration. How closely the procedure approaches this ideal depends on how well the selected element is chosen. Since this element serves as a midpoint or pivot, the best choice would be the central value (or median). Of course, the true median is unknown. A simple but reasonable approximation is to select the median of the first, middle, and last elements.

Procedure The program first deals with the entire array. It selects the median of the current first, middle, and last elements as a central element. It moves that element to the first position and divides the array into two parts or partitions. It then operates recursively on the parts, dividing them into parts and stopping when a part contains no elements or only one element. Since each recursion places 6 bytes on the stack, the program must guard against overflow by checking whether the stack has reached to within a small buffer of its lowest available position.

Note that the selected element always ends up in the correct position after an iteration. Therefore, it need not be included in either partition.

Our rule for choosing the middle element is as follows, assuming that the first element is #1:

1. If the array has an odd number of elements, take the centre one.

For example, if the array has 11 elements, take #6.

2. If the array has an even number of elements and its base address is even, take the element on the lower (base address) side of the centre. for example, if the array starts in 0300_{16} and has 12 elements, take #6.

3. If the array has an even number of elements and its base address is odd, take the element on the upper side of the centre. For example, if the array starts in 0301_{16} and has 12 elements, take #7.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of base address of array
Less significant byte of base address of array

More significant byte of address of last word in array
Less significant byte of address of last word in array

More significant byte of lowest possible stack address
Less significant byte of lowest possible stack address

Exit conditions

Array sorted into ascending order, considering the elements as unsigned words. Thus, the smallest unsigned word ends up stored starting at the base address. Carry = 0 if the stack did not overflow and the result is proper. Carry = 1 if the stack overflowed and the final array is not sorted.

Example

Data: Length (size) of array = $0C_{16} = 12_{10}$
Elements = $2B_{16}$, 57_{16}, $1D_{16}$, 26_{16},
22_{16}, $2E_{16}$, $0C_{16}$, 44_{16},
17_{16}, $4B_{16}$, 37_{16}, 27_{16}.

Result: In the first iteration, we have:
Selected element = median of the first (#1 = $2B_{16}$), middle (#6 = $2E_{16}$), and last (#12 = 27_{16}) elements. The

selected element is therefore #1 ($2B_{16}$), and no swapping is necessary since it is already in the first position.

At the end of the iteration, the array is

27_{16}, 17_{16}, $1D_{16}$, 26_{16},
22_{16}, $0C_{16}$, $2B_{16}$, 44_{16},
$2E_{16}$, $4B_{16}$, 37_{16}, 57_{16}.

The first partition, consisting of elements less than $2B_{16}$, is 27_{16}, 17_{16}, $1D_{16}$, 26_{16}, 22_{16}, and $0C_{16}$.
The second partition, consisting of elements greater than $2B_{16}$, is 44_{16}, $2E_{16}$, $4B_{16}$, 37_{16}, and 57_{16}.
Note that the selected element ($2B_{16}$) is now in the correct position and need not be included in either partition.
We may now sort the first partition recursively in the same way:
Selected element = median of the first (#1 = 27_{16}), middle (#3 = $1D_{16}$), and last (#6 = $0C_{16}$) elements. Here, #3 is the median and must be exchanged initially with #1.
The final order of the elements in the first partition is:

$0C_{16}$, 17_{16}, $1D_{16}$, 26_{16}, 22_{16}, 27_{16}.

The first partition of the first partition (consisting of elements less than $1D_{16}$) is $0C_{16}$, 17_{16}. We will call this the (1,1) partition for short.
The second partition of the first partition (consisting of elements greater than $1D_{16}$) is 26_{16}, 22_{16}, and 27_{16}.
As in the first iteration, the selected element ($1D_{16}$) is in the correct position and need not be considered further.
We may now sort the (1,1) partition recursively as follows:
Selected element = median of the first (#1 = $0C_{16}$), middle (#1 = $0C_{16}$), and last (#2 = 17_{16}) elements. Thus the selected element is the first element (#1 = $0C_{16}$), and no initial swap is necessary.
The final order is obviously the same as the initial order, and the two resulting partitions contain 0 and 1 element, respectively. Thus the next iteration concludes the recursion, and we then sort the other partitions by the same method. Obviously, quicksort's overhead becomes a major factor when an array contains only a few elements. This is why one might use a bubble sort once quicksort has created small enough partitions.

Note that the example array does not contain any identical elements. During an iteration, elements that are the same as the selected element are never moved. Thus they may end up in either partition. Strictly speaking, then, the two partitions consist of elements 'less than or possibly equal to the selected element' and elements 'greater than or possibly equal to the selected element.'

References

M. J. Augenstein and A. M. Tenenbaum, *Data Structures and PL/I Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1979, pp. 460–471. There is also a Pascal version of this book entitled *Data Structures Using Pascal* (Prentice-Hall, Englewood Cliffs, NJ, 1982) and a BASIC version entitled *Data Structures for Personal Computers* (Y. Langsam, co-author, Prentice-Hall, Englewood Cliffs, NJ, 1985).

N. Dale and S. C. Lilly, *Pascal Plus Data Structures*, D. C. Heath, Lexington, MA, 1985, pp. 300–307.

D. E. Knuth, *The Art of Computer Programming. Vol. 3: Searching and Sorting*, Addison-Wesley, Reading, MA, 1973, pp. 114–123.

Registers used All

Execution time Approximately $N \times \log_2 N$ loops through PARTLP plus $2 \times N + 1$ overhead calls to SORT. Each iteration of PARTLP takes approximately 60 or 120 cycles (depending on whether an exchange is necessary), and each overhead call to SORT takes approximately 200 cycles. Thus the total execution time is of the order of

$$90 \times N \times \log_2 N + 200 \times (2 \times N + 1) \text{ cycles}$$

If, for example, $N = 16\,384$ (2^{14}), the total execution time should be around

$$90 \times 16\,384 \times 14 + 200 \times 32\,769 = 20\,600\,000 + 6\,600\,000$$
$$= \text{about } 27\,200\,000 \text{ cycles}$$

This is about 27 s at a typical 6809 clock rate of 1 MHz.

Program size 179 bytes

Data memory required 8 bytes anywhere in RAM for pointers to the first and last element of a partition (2 bytes starting at addresses FIRST and LAST, respectively), a pointer to the bottom of the stack (2 bytes starting at address STKBTM), and the original value of the stack pointer (2 bytes starting at address OLDSP). Each recursion level requires 6 bytes of stack space, and the routines themselves require another 4 bytes.

Special case If the stack overflows (i.e. comes too close to its boundary), the program exits with the Carry flag set to 1.

```
*       Title           Quicksort
*       Name:           QSORT
*
*       Purpose:        Arrange an array of unsigned words into
*                       ascending order using a quicksort, with a
*                       maximum size of 32767 words.
*
*       Entry:          TOP OF STACK
*                         High byte of return address
*                         Low byte of return address
*                         High byte of address of first word in array
*                         Low byte of address of first word in array
*                         High byte of address of last word in array
*                         Low byte of address of last word in array
*                         High byte of lowest available stack address
*                         Low byte of lowest available stack address
*
*       Exit:           If the stack did not overflow then
*                         The array is sorted into ascending order.
*                         Carry = 0
*                       Else
*                         Carry = 1
*
*       Registers Used: All

*       Time:           The timing is highly data-dependent but the
*                       quicksort algorithm takes approximately
*                       N * log (N) loops through PARTLP. There will be
*                             2
*                       2 * N+1 calls to Sort. The number of recursions
*                       will probably be a fraction of N but if all
*                       data is the same, the recursion could be up to
*                       N. Therefore, the amount of stack space should
*                       be maximized. NOTE: Each recursion level takes
*                       6 bytes of stack space.
```

```
*
*                       In the above discussion, N is the number of
*                       array elements.
*
*       Size:           Program 179 bytes
*                       Data      8 bytes plus 4 stack bytes
*

*
*
QSORT:
          PULS      D,X,Y,U         REMOVE PARAMETERS FROM STACK
          PSHS      D               PUT RETURN ADDRESS BACK IN STACK
          *
          *WATCH FOR STACK OVERFLOW
          *CALCULATE A THRESHOLD TO WARN OF OVERFLOW
          * (10 BYTES FROM THE END OF THE STACK)
          *SAVE THIS THRESHOLD FOR LATER COMPARISONS
          *ALSO SAVE THE POSITION OF THIS ROUTINE'S RETURN ADDRESS
          * IN THE EVENT WE MUST ABORT BECAUSE OF STACK OVERFLOW
          *
          STS       OLDSP           SAVE POINTER TO RETURN ADDRESS IN
                                    * CASE WE MUST ABORT
          LEAU      10,U            ADD SMALL BUFFER (10 BYTES) TO
                                    * LOWEST STACK ADDRESS
          STU       STKBTM          SAVE SUM AS BOTTOM OF STACK FOR
                                    * FIGURING WHEN TO ABORT
          *
          *WORK RECURSIVELY THROUGH THE QUICKSORT ALGORITHM AS
          * FOLLOWS:
          *  1. CHECK IF THE PARTITION CONTAINS 0 OR 1 ELEMENT.
          *     MOVE UP A RECURSION LEVEL IF IT DOES.
          *  2. USE MEDIAN TO OBTAIN A REASONABLE CENTRAL VALUE
          *     FOR DIVIDING THE CURRENT PARTITION INTO TWO
          *     PARTS.
          *  3. MOVE THROUGH THE ARRAY SWAPPING ELEMENTS THAT
          *     ARE OUT OF ORDER UNTIL ALL ELEMENTS BELOW THE
          *     CENTRAL VALUE ARE AHEAD OF ALL ELEMENTS ABOVE
          *     THE CENTRAL VALUE.  SUBROUTINE COMPARE
          *     COMPARES ELEMENTS, SWAP EXCHANGES ELEMENTS,
          *     PREV MOVES UPPER BOUNDARY DOWN ONE ELEMENT,
          *     AND NEXT MOVES LOWER BOUNDARY UP ONE ELEMENT.
          *  4. CHECK IF THE STACK IS ABOUT TO OVERFLOW.  IF IT
          *     IS, ABORT AND EXIT.
          *  5. ESTABLISH THE BOUNDARIES FOR THE FIRST PARTITION
          *     (CONSISTING OF ELEMENTS LESS THAN THE CENTRAL VALUE)
          *     AND SORT IT RECURSIVELY.
          *  6. ESTABLISH THE BOUNDARIES FOR THE SECOND PARTITION
          *     (CONSISTING OF ELEMENTS GREATER THAN THE CENTRAL
          *     VALUE) AND SORT IT RECURSIVELY.
          *
SORT:
          *
          *SAVE BASE ADDRESS AND ADDRESS OF LAST ELEMENT
          * IN CURRENT PARTITION
          *
```

```
        STX     FIRST           SAVE BASE ADDRESS
        STY     LAST            SAVE ADDRESS OF LAST ELEMENT
        *
        *CHECK IF PARTITION CONTAINS 0 OR 1 ELEMENTS
        * IT DOES IF FIRST IS EITHER LARGER THAN (0)
        * OR EQUAL TO (1) LAST.
        *
        *STOP WHEN FIRST >= LAST
        *
        CMPX    LAST            CALCULATE FIRST - LAST
        BCC     EXITPR          BRANCH (RETURN) IF DIFFERENCE IS
                                * POSITIVE - THIS PART IS SORTED
        *
        *START ANOTHER ITERATION ON THIS PARTITION
        *USE MEDIAN TO FIND A REASONABLE CENTRAL ELEMENT
        *MOVE CENTRAL ELEMENT TO FIRST POSITION
        *
        BSR     MEDIAN          SELECT CENTRAL ELEMENT, MOVE IT
                                * TO FIRST POSITION
        LDU     #0              BIT 0 OF REGISTER U = DIRECTION
                                * IF IT'S 0 THEN DIRECTION IS UP
                                * ELSE DIRECTION IS DOWN
        *
        *REORDER ARRAY BY COMPARING OTHER ELEMENTS WITH THE
        * CENTRAL ELEMENT.  START BY COMPARING THAT ELEMENT WITH
        * LAST ELEMENT.  EACH TIME WE FIND AN ELEMENT THAT
        * BELONGS IN THE FIRST PART (THAT IS, IT IS LESS THAN
        * THE CENTRAL ELEMENT), SWAP IT INTO THE FIRST PART IF IT
        * IS NOT ALREADY THERE AND MOVE THE BOUNDARY OF THE
        * FIRST PART DOWN ONE ELEMENT.  SIMILARLY, EACH TIME WE
        * FIND  AN ELEMENT THAT BELONGS IN THE SECOND PART (THAT
        * IS, IT IS GREATER THAN THE CENTRAL ELEMENT), SWAP IT
        * INTO THE SECOND PART IF IT IS NOT ALREADY THERE AND MOVE
        * THE BOUNDARY OF THE SECOND PART UP ONE ELEMENT.
        *ULTIMATELY, THE BOUNDARIES COME TOGETHER
        * AND THE DIVISION OF THE ARRAY IS THEN COMPLETE
        *NOTE THAT ELEMENTS EQUAL TO THE CENTRAL ELEMENT ARE NEVER
        * SWAPPED AND SO MAY END UP IN EITHER PART
        *
PARTLP:
        *
        *LOOP SORTING UNEXAMINED PART OF PARTITION
        * UNTIL THERE IS NOTHING LEFT IN IT
        *
        TFR     X,D             LOWER BOUNDARY
        PSHS    Y
        CMPD    ,S++            LOWER BOUNDARY-UPPER BOUNDARY
        BCC     DONE            EXIT WHEN EVERYTHING EXAMINED
        *
        *COMPARE NEXT 2 ELEMENTS.  IF OUT OF ORDER,  SWAP THEM
        *AND CHANGE DIRECTION OF SEARCH
        * IF FIRST > LAST THEN SWAP
        *
        LDD     ,X              COMPARE ELEMENTS
        CMPD    ,Y
        BLS     REDPRT          BRANCH IF ALREADY IN ORDER
```

```
            *
            *ELEMENTS OUT OF ORDER, SWAP THEM AND CHANGE DIRECTION
            *
            TFR         U,D             GET DIRECTION
            COMB                        CHANGE DIRECTION
            TFR         D,U             SAVE NEW DIRECTION
            JSR         SWAP            SWAP ELEMENTS
            *
            *REDUCE SIZE OF UNEXAMINED AREA
            *IF NEW ELEMENT LESS THAN CENTRAL ELEMENT, MOVE
            * TOP BOUNDARY DOWN
            *IF NEW ELEMENT GREATER THAN CENTRAL ELEMENT, MOVE
            * BOTTOM BOUNDARY UP
            *IF ELEMENTS EQUAL, CONTINUE IN LATEST DIRECTION
            *
REDPRT:
            CMPU        #0              CHECK DIRECTION
            BEQ         UP              BRANCH IF MOVING UP
            LEAX        2,X             ELSE MOVE TOP BOUNDARY DOWN BY
                                        * ONE ELEMENT
            BRA         PARTLP
UP:
            LEAY        -2,Y            MOVE BOTTOM BOUNDARY UP BY ONE
            JMP         PARTLP           ONE ELEMENT
            *
            *THIS PARTITION HAS NOW BEEN SUBDIVIDED INTO TWO
            * PARTITIONS. ONE STARTS AT THE TOP AND ENDS JUST
            * ABOVE THE CENTRAL ELEMENT.  THE OTHER STARTS
            * JUST BELOW THE CENTRAL ELEMENT AND CONTINUES
            * TO THE BOTTOM.  THE CENTRAL ELEMENT IS NOW IN
            * ITS PROPER SORTED POSITION AND NEED NOT BE
            * INCLUDED IN EITHER PARTITION
            *
DONE:
            *
            *FIRST CHECK WHETHER STACK MIGHT OVERFLOW
            *IF IT IS GETTING TOO CLOSE TO THE BOTTOM, ABORT
            * THE PROGRAM AND EXIT
            *
            TFR         S,D             CALCULATE SP - STKBTM
            SUBD        STKBTM
            BLS         ABORT           BRANCH (ABORT) IF STACK TOO LARGE
            *
            *ESTABLISH BOUNDARIES FOR FIRST (LOWER) PARTITION
            *LOWER BOUNDARY IS SAME AS BEFORE
            *UPPER BOUNDARY IS ELEMENT JUST BELOW CENTRAL ELEMENT
            *THEN RECURSIVELY QUICKSORT FIRST PARTITION
            *
            LDY         LAST            GET ADDRESS OF LAST ELEMENT
            PSHS        X,Y             SAVE CENTRAL, LAST ADDRESSES
            LEAY        -2,X            CALCULATE LAST FOR FIRST PART
            LDX         FIRST           FIRST IS SAME AS BEFORE
            BSR         SORT            QUICKSORT FIRST PART
            *
            *ESTABLISH BOUNDARIES FOR SECOND (UPPER) PARTITION
            *UPPER BOUNDARY IS SAME AS BEFORE
```

```
          *LOWER BOUNDARY IS ELEMENT JUST ABOVE CENTRAL ELEMENT
          *THEN RECURSIVELY QUICKSORT SECOND PARTITION
          *
          PULS      X,Y             GET FIRST, LAST FOR SECOND PART
          LEAX      2,X             CALCULATE FIRST FOR SECOND PART
          BSR       SORT            QUICKSORT SECOND PART
          CLC                       CLEAR CARRY, INDICATING NO ERRORS
EXITPR:
          RTS                       GOOD EXIT
          *
          *ERROR EXIT, SET CARRY TO 1
          *
ABORT:
          LDS       OLDSP           GET ORIGINAL STACK POINTER
          SEC                       INDICATE ERROR
          RTS                       RETURN WITH ERROR INDICATOR TO
                                    * ORIGINAL CALLER


*******************************
*ROUTINE: MEDIAN
*PURPOSE: DETERMINE WHICH ELEMENT IN A PARTITION
*       SHOULD BE USED AS THE CENTRAL ELEMENT OR PIVOT
*ENTRY: ADDRESS OF FIRST ELEMENT IN REGISTER X
*       ADDRESS OF LAST ELEMENT IN REGISTER Y
*EXIT:  CENTRAL ELEMENT IN FIRST POSITION
*       X,Y UNCHANGED
*REGISTERS USED: D,U
*******************************

MEDIAN:
          *
          *DETERMINE ADDRESS OF MIDDLE ELEMENT
          * MIDDLE := ALIGNED(FIRST + LAST) DIV 2
          *
          PSHS      Y               SAVE ADDRESS OF LAST IN STACK
          TFR       X,D             ADD ADDRESSES OF FIRST, LAST
          ADDD      ,S
          LSRA                      DIVIDE SUM BY 2
          RORB
          ANDB      #%11111110      ALIGN CENTRAL ADDRESS
          PSHS      D               SAVE CENTRAL ADDRESS ON STACK
          TFR       X,D             ALIGN MIDDLE TO BOUNDARY OF FIRST
          CLRA                      MAKE BIT 0 OF MIDDLE SAME AS BIT
          ANDB      #%00000001       0 OF FIRST
          ADDD      ,S++
          TFR       D,U             SAVE MIDDLE ADDRESS IN U
          *
          *DETERMINE MEDIAN OF FIRST, MIDDLE, LAST ELEMENTS
          *COMPARE FIRST AND MIDDLE
          *
          LDD       ,U              GET MIDDLE ELEMENT
          CMPD      ,X              MIDDLE - FIRST
          BLS       MIDD1           BRANCH IF FIRST >= MIDDLE
          *
```

```
            *WE KNOW (MIDDLE > FIRST)
            * SO COMPARE MIDDLE AND LAST
            *
            LDD         ,Y              GET LAST ELEMENT
            CMPD        ,U              LAST - MIDDLE
            BCC         SWAPMF          BRANCH IF LAST >= MIDDLE
                                        * MIDDLE IS MEDIAN
            *
            *WE KNOW (MIDDLE > FIRST) AND (MIDDLE > LAST)
            * SO COMPARE FIRST AND LAST (MEDIAN IS LARGER ONE)
            *
            CMPD        ,X              LAST - FIRST
            BHI         SWAPLF          BRANCH IF LAST > FIRST
                                        * LAST IS MEDIAN
            BRA         MEXIT           EXIT IF FIRST >= LAST
                                        * FIRST IS MEDIAN
            *
            *WE KNOW FIRST >= MIDDLE
            *SO COMPARE FIRST AND LAST
            *
MIDD1:
            LDD         ,Y              GET LAST
            CMPD        ,X              LAST - FIRST
            BCC         MEXIT           EXIT IF LAST > = FIRST
                                        * FIRST IS MEDIAN
            *
            *WE KNOW (FIRST >= MIDDLE) AND (FIRST > LAST)
            * SO COMPARE MIDDLE AND LAST (MEDIAN IS LARGER ONE)
            *
            CMPD        ,U              LAST - MIDDLE
            BHI         SWAPLF          BRANCH IF LAST > MIDDLE
                                        * LAST IS MEDIAN
            *
            *MIDDLE IS MEDIAN, MOVE ITS POINTER TO LAST
            *
SWAPMF:
            TFR         U,Y             MOVE MIDDLE'S POINTER TO LAST
            *
            *LAST IS MEDIAN, SWAP IT WITH FIRST
            *
SWAPLF:
            BSR         SWAP            SWAP LAST, FIRST
            *
            *RESTORE LAST AND EXIT
            *
MEXIT:
            PULS        Y               RESTORE ADDRESS OF LAST ELEMENT
            RTS


*********************************
*ROUTINE: SWAP
*PURPOSE: SWAP ELEMENTS POINTED TO BY X,Y
*ENTRY: X = ADDRESS OF ELEMENT 1
*       Y = ADDRESS OF ELEMENT 2
*EXIT:  ELEMENTS SWAPPED
```

```
*REGISTERS USED: D
*********************************

SWAP:
          LDD       ,X              GET FIRST ELEMENT
          PSHS      D               SAVE FIRST ELEMENT
          LDD       ,Y              GET SECOND ELEMENT
          STD       ,X              STORE SECOND IN FIRST
          PULS      D               GET SAVED FIRST ELEMENT
          STD       ,Y              STORE FIRST IN SECOND ADDRESS
          RTS


*
*DATA SECTION
*
FIRST:    RMB       2               POINTER TO FIRST ELEMENT OF PART
LAST:     RMB       2               POINTER TO LAST ELEMENT OF PART
STKBTM:   RMB       2               THRESHOLD FOR STACK OVERFLOW
OLDSP:    RMB       2               POINTER TO ORIGINAL RETURN ADDRESS

*
*       SAMPLE EXECUTION
*

*
*PROGRAM SECTION
SC6F:
          *
          *SORT AN ARRAY BETWEEN BEGBUF (FIRST ELEMENT)
          * AND ENDBUF (LAST ELEMENT)
          *LET STACK EXPAND 100 HEX BYTES
          *
          LEAU      -$100,S         BOUNDARY FOR STACK OVERFLOW
          LDX       #BEGBUF         ADDRESS OF FIRST ELEMENT
          LDY       #ENDBUF         ADDRESS OF LAST ELEMENT
          PSHS      U,X,Y           SAVE PARAMETERS IN STACK
          JSR       QSORT           SORT USING QUICKSORT
                                    *RESULT FOR TEST DATA IS
                                    * 0,1,2,3, ... ,14,15
          BRA       SC6F            LOOP TO REPEAT TEST

*
*DATA SECTION
*
BEGBUF:   FDB       15
          FDB       14
          FDB       13
          FDB       12
          FDB       11
          FDB       10
          FDB       9
          FDB       8
          FDB       7
          FDB       6
          FDB       5
```

```
            FDB         4
            FDB         3
            FDB         2
            FDB         1
ENDBUF:
            FDB         0

            END
```

6G RAM test (RAMTST)

Tests a RAM area specified by a base address and a length in bytes. Writes the values 0, FF_{16}, 10101010_2 (AA_{16}), and 01010101_2 (55_{16}) into each byte and checks whether they can be read back correctly. Places 1 in each bit position of each byte and checks whether it can be read back correctly with all other bits cleared. Clears the Carry flag if all tests run correctly; if it finds an error, it exits immediately, setting the Carry flag and returning the test value and the address at which the error occurred.

Procedure The program performs the single value tests (with 0, FF_{16}, AA_{16}, and 55_{16}) by first filling the memory area and then comparing each byte with the specified value. Filling the entire area first should provide enough delay between writing and reading to detect a failure to retain data (perhaps caused by improperly designed refresh circuitry). The program then performs the walking bit test, starting with bit 7; here it writes the data into memory and reads it back immediately for a comparison.

Entry conditions

Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of size (length) of test area in bytes
Less significant byte of size (length) of test area in bytes

More significant byte of base address of test area
Less significant byte of base address of test area

Exit conditions

1. If an error is found:

   Carry = 1
   Address containing error in register X
   Test value in A

2. If no error is found:

Carry = 0
All bytes in test area contain 0

Example

Data: Base address = 0380_{16}
Length (size) of area = 0200_{16}

Result: Area tested is the 0200_{16} bytes starting at address 0380_{16}, i.e. 0380_{16}–$057F_{16}$. The order of the tests is:

1. Write and read 0
2. Write and read FF_{16}
3. Write and read AA_{16} (10101010_2)
4. Write and read 55_{16} (01010101_2)
5. Walking bit test, starting with 1 in bit 7. That is, start with 10000000_2 (80_{16}) and move the 1 one position right for each subsequent test of a byte.

Registers used All

Execution time Approximately 268 cycles per byte tested plus 231 cycles overhead. Thus, for example, to test an area of size 0400_{16} = 1024_{10} would take

$268 \times 1024 + 231 = 274\,432 + 231 = 274\,663$ cycles

This is about 275 ms at a standard 6809 clock rate of 1 MHz.

Program size 97 bytes

Data memory required None

Special cases

1. An area size of 0000_{16} causes an immediate exit with no memory tested. The Carry flag is cleared to indicate no errors.
2. Since the routine changes all bytes in the tested area, using it to test

an area that includes itself will have unpredictable results.

Note that Case 1 means you cannot ask this routine to test the entire memory, but such a request would be meaningless anyway since it would require the routine to test itself.

3. Testing a ROM causes a return with an error indication after the first occasion on which the test value differs from the memory's contents.

```
*
*
*       Title           RAM Test
*       Name:           RAMTST
*
*       Purpose:        Test a RAM (read/write memory) area as follows:
*                         1) Write all 0 and test
*                         2) Write all 11111111 binary and test
*                         3) Write all 10101010 binary and test
*                         4) Write all 01010101 binary and test
*                         5) Shift a single 1 through each bit,
*                            while clearing all other bits
*
*                         If the program finds an error, it exits
*                         immediately with the Carry flag set and
*                         indicates the test value and where the
*                         error occurred.
*
*       Entry:          TOP OF STACK
*                         High byte of return address
*                         Low byte of return address
*                         High byte of area size in bytes
*                         Low byte of area size in bytes
*                         High byte of base address of area
*                         Low byte of base address of area
*
*       Exit:           If there are no errors then
*                         Carry flag equals 0
*                         test area contains 0 in all bytes
*                       else
*                         Carry flag equals 1
*                         Register X = Address of error
*                         Register A = Test value
*
*       Registers Used: All
*
*       Time:           Approximately 268 cycles per byte plus
*                       231 cycles overhead
*
*       Size:           Program 97 bytes

*
RAMTST:
          *
```

```
*EXIT INDICATING NO ERRORS IF AREA SIZE IS ZERO
*
PULS    U       SAVE RETURN ADDRESS
CLC             INDICATE NO ERRORS
LDX     ,S      GET AREA SIZE
BEQ     EXITRT  BRANCH (EXIT) IF AREA SIZE IS ZERO
                * CARRY = 0 IN THIS CASE
*
*FILL MEMORY WITH 0 AND TEST
*
CLRA            GET ZERO VALUE
BSR     FILCMP  FILL AND TEST MEMORY
BCS     EXITRT  BRANCH (EXIT) IF ERROR FOUND
*
*FILL MEMORY WITH FF HEX (ALL 1'S) AND TEST
*
LDA     #$FF    GET ALL 1'S VALUE
BSR     FILCMP  FILL AND TEST MEMORY
BCS     EXITRT  BRANCH (EXIT) IF ERROR FOUND
*
*FILL MEMORY WITH ALTERNATING 1'S AND 0'S AND TEST
*
LDA     #%10101010  GET ALTERNATING 1'S AND 0'S PATTERN
BSR     FILCMP  FILL AND TEST MEMORY
BCS     EXITRT  BRANCH (EXIT) IF ERROR FOUND
*
*FILL MEMORY WITH ALTERNATING 0'S AND 1'S AND TEST
*
LDA     #%01010101 GET ALTERNATING 0'S AND 1'S PATTERN
BSR     FILCMP  FILL AND TEST MEMORY
BCS     EXITRT  BRANCH (EXIT) IF ERROR FOUND
*
*PERFORM WALKING BIT TEST. PLACE A 1 IN BIT 7 AND
* SEE IF IT CAN BE READ BACK.  THEN MOVE THE 1 TO
* BITS 6, 5, 4, 3, 2, 1, AND 0 AND SEE IF IT CAN
* BE READ BACK
*
LDX     2,S     GET BASE ADDRESS OF AREA TO TEST
LDY     ,S      GET AREA SIZE IN BYTES
CLRB            GET ZERO TO USE IN CLEARING AREA
WLKLP:
LDA     #%10000000     MAKE BIT 7 1, ALL OTHER BITS 0
WLKLP1:
STA     ,X      STORE TEST PATTERN IN MEMORY
CMPA    ,X      TRY TO READ IT BACK
BNE     EXITCS  BRANCH (EXIT) IF ERROR FOUND
LSRA            SHIFT PATTERN TO MOVE 1 BIT RIGHT
BNE     WLKLP1  CONTINUE UNTIL PATTERN BECOMES ZERO
                * THAT IS, UNTIL 1 BIT MOVES ALL THE
                * WAY ACROSS THE BYTE
STB     ,X+     CLEAR BYTE JUST CHECKED
LEAY    -1,Y    DECREMENT 16-BIT COUNTER
BNE     WLKLP   CONTINUE UNTIL AREA CHECKED
CLC             NO ERRORS - CLEAR CARRY
BRA     EXITRT
*
```

```
          *FOUND AN ERROR - SET CARRY TO INDICATE IT
          *
EXITCS:
          SEC                 ERROR FOUND - SET CARRY
          *
          *REMOVE PARAMETERS FROM STACK AND EXIT
          *
EXITRT:
          LEAS      4,S       REMOVE PARAMETERS FROM STACK
          JMP       ,U        EXIT TO RETURN ADDRESS

***********************************
*ROUTINE: FILCMP
*PURPOSE: FILL MEMORY WITH A VALUE AND TEST
*         THAT IT CAN BE READ BACK
*ENTRY: A = TEST VALUE
*       STACK CONTAINS (IN ORDER STARTING AT TOP):
*         RETURN ADDRESS
*         AREA SIZE IN BYTES
*         BASE ADDRESS OF AREA
*EXIT:  IF NO ERRORS THEN
*         CARRY FLAG EQUALS 0
*       ELSE
*         CARRY FLAG EQUALS 1
*         X = ADDRESS OF ERROR
*         A = TEST VALUE
*         PARAMETERS LEFT ON STACK
*REGISTERS USED: CC,X,Y
***********************************

FILCMP:
          LDY       2,S           GET SIZE OF AREA IN BYTES
          LDX       4,S           GET BASE ADDRESS OF AREA
          *
          *FILL MEMORY WITH TEST VALUE
          *
FILLP:
          STA       ,X+           FILL A BYTE WITH TEST VALUE
          LEAY      -1,Y          CONTINUE UNTIL AREA FILLED
          BNE       FILLP
          *
          *COMPARE MEMORY AND TEST VALUE
          *
          LDY       2,S           GET SIZE OF AREA IN BYTES
          LDX       4,S           GET BASE ADDRESS OF AREA
CMPLP:
          CMPA      ,X+           COMPARE MEMORY AND TEST VALUE
          BNE       EREXIT        BRANCH (ERROR EXIT) IF NOT EQUAL
          LEAY      -1,Y          CONTINUE UNTIL AREA CHECKED
          BNE       CMPLP
          *
          *NO ERRORS FOUND, CLEAR CARRY AND EXIT
          *
          CLC                     INDICATE NO ERRORS
          RTS
          *
```

```
          *ERROR FOUND, SET CARRY, MOVE POINTER BACK, AND EXIT
          *
EREXIT:
          SEC                       INDICATE AN ERROR
          LEAX      -1,X            POINT TO BYTE CONTAINING ERROR
          RTS

*
*         SAMPLE EXECUTION
*

*
SC6G:
          *
          *TEST RAM FROM 2000 HEX THROUGH 300F HEX
          *  SIZE OF AREA = 1010 HEX BYTES
          *
          LDY       #$2000          GET BASE ADDRESS OF TEST AREA
          LDX       #$1010          GET SIZE OF AREA IN BYTES
          PSHS      X,Y             SAVE PARAMETERS IN STACK
          JSR       RAMTST          TEST MEMORY
                                    *CARRY FLAG SHOULD BE 0

          END
```

6H Jump table (JTAB)

Transfers control to an address selected from a table according to an index. The addresses are stored in the usual 6809 format (more significant byte first), starting at address JMPTBL. The size of the table (number of addresses) is a constant LENSUB, which must be less than or equal to 128. If the index is greater than or equal to LENSUB, the program returns control immediately with the Carry flag set to 1.

Procedure The program first checks if the index is greater than or equal to the size of the table (LENSUB). If it is, the program returns control with the Carry flag set. If it is not, the program obtains the starting address of the appropriate subroutine from the table and jumps to it. The result is like an indexed JSR instruction with range checking and automatic accounting for the 16-bit length of addresses.

Entry conditions

Index in A

Exit conditions

If (A) is greater than LENSUB, an immediate return with Carry = 1. Otherwise, control is transferred to appropriate subroutine as if an indexed call had been performed. The return address remains at the top of the stack.

Example

Data:	LENSUB (size of subroutine table) = 03
	Table consists of addresses SUB0, SUB1, and SUB2
	Index = (A) = 02
Result:	Control transferred to address SUB2 (PC = SUB2)

Registers used A, CC, X

Execution time 17 cycles besides the time required to execute the actual subroutine.

Program size 13 bytes plus 2 × LENSUB bytes for the table of starting addresses, where LENSUB is the number of subroutines.

Data memory required None

Special case Entry with an index greater than or equal to LENSUB causes an immediate exit with the Carry flag set to 1

```
*       Title           Jump Table
*       Name:           JTAB
*
*       Purpose:        Given an index, jump to the subroutine with
*                       that index in a table
*       Entry:          Register A is the subroutine number (0 to
*                                  LENSUB-1, the number of subroutines)
*                                  LENSUB must be less than or equal to
*                                  128.
*       Exit:           If the routine number is valid then
*                         execute the routine
*                       else
*                         Carry flag equals 1
*
*       Registers Used: A,CC,X
*
*       Time:           17 cycles plus execution time of subroutine
*
*       Size:           Program 13 bytes plus size of table (2*LENSUB)
*
*
*         EXIT WITH CARRY SET IF ROUTINE NUMBER IS INVALID
*         THAT IS, IF IT IS TOO LARGE FOR TABLE (>LENSUB - 1)
*
JTAB:
          CMPA      #LENSUB         COMPARE ROUTINE NUMBER, TABLE LENGTH
          BCC       EREXIT          BRANCH (EXIT) IF ROUTINE NUMBER TOO
                                    * LARGE
*
*         INDEX INTO TABLE OF WORD-LENGTH ADDRESSES
*         OBTAIN ROUTINE ADDRESS FROM TABLE AND TRANSFER CONTROL
*           TO IT
*
```

```
          ASLA                        DOUBLE INDEX FOR WORD-LENGTH ENTRIES
          LDX       #JMPTBL           GET BASE ADDRESS OF JUMP TABLE
          JMP       [A,X]             JUMP INDIRECTLY TO SUBROUTINE
*
*         ERROR EXIT - EXIT WITH CARRY SET
*
EREXIT:
          SEC                         INDICATE BAD ROUTINE NUMBER
          RTS

LENSUB    EQU       3                 NUMBER OF SUBROUTINES IN TABLE

*
*JUMP TABLE
*
JMPTBL:
          FDB       SUB0              ROUTINE 0
          FDB       SUB1              ROUTINE 1
          FDB       SUB2              ROUTINE 2


*
*THREE TEST SUBROUTINES FOR JUMP TABLE
*
SUB0:
          LDA       #1                TEST ROUTINE 0 SETS (A) = 1
          RTS

SUB1:
          LDA       #2                TEST ROUTINE 1 SETS (A) = 2
          RTS

SUB2:
          LDA       #3                TEST ROUTINE 2 SETS (A) = 3
          RTS

*
*         SAMPLE EXECUTION
*

*
*PROGRAM SECTION
SC6H:
          CLRA                        EXECUTE ROUTINE 0
          JSR       JTAB              AFTER EXECUTION, (A) = 1
          LDA       #1                EXECUTE ROUTINE 1
          JSR       JTAB              AFTER EXECUTION, (A) = 2
          LDA       #2                EXECUTE ROUTINE 2
          JSR       JTAB              AFTER EXECUTION, (A) = 3
          LDA       #3                EXECUTE ROUTINE 3
          JSR       JTAB              AFTER EXECUTION, CARRY = 1
                                      *INDICATING BAD ROUTINE NUMBER
          BRA       SC6H              LOOP FOR MORE TESTS

          END
```

7 Data structure manipulation

7A Queue manager (INITQ, INSRTQ, REMOVQ)

Manages a queue of 16-bit words on a first-in, first-out basis. The queue may contain up to 255 word-length elements plus an 8-byte header. Consists of the following routines:

1. INITQ starts the queue's head and tail pointers at the base address of its data area, sets the queue's length to 0, and sets its end pointer to just beyond the end of the data area.

2. INSRTQ inserts an element at the tail of the queue if there is room for it.

3. REMOVQ removes an element from the head of the queue if one is available.

These routines assume a data area of fixed length. The actual queue may occupy any part of it. If either the head or the tail reaches the physical end of the area, the routine simply sets it back to the base address, thus providing wraparound.

The queue header contains the following information:

1. Length of data area in words. This is a single byte specifying the maximum number of elements the queue can hold.

2. Queue length (number of elements currently in the queue)

3. Head pointer (address of oldest element in queue)

4. Tail pointer (address at which next entry will be placed)
5. End pointer (address just beyond the end of the data area).

Note that the first two items are byte-length and the last three are word-length.

Procedures

1. INITQ sets the head and tail pointers to the base address of the data area, establishes the length of the data area, sets the queue's length (a single byte) to 0, and sets the end pointer to the address just beyond the end of the data area.

2. INSRTQ checks whether the queue already occupies the entire data area. If so, it sets the Carry flag to indicate an overflow. If not, it inserts the element at the tail and increases the tail pointer. If the tail pointer has gone beyond the end of the data area, it sets it back to the base address.

3. REMOVQ checks whether the queue is empty. If so, it sets the Carry flag to indicate an underflow. If not, it removes the element from the head and increases the head pointer. If the head pointer has gone beyond the end of the data area, it sets it back to the base address.

The net result of a sequence of INSRTQs and REMOVQs is that the head 'chases' the tail across the data area. The occupied part of the data area starts at the head and ends just before the tail.

Entry conditions

1. INITQ
Base address of queue in register X
Length of data area in words in register A

2. INSRTQ
Base address of queue in register X
Element to be inserted in register U

3. REMOVQ
Base address of queue in register X

Exit conditions

1. INITQ
Head pointer and tail pointer both set to base address of data area, length of data area set to specified value, queue length set to 0, and end pointer set to address just beyond the end of the data area.

2. INSRTQ
Element inserted into queue, queue length increased by 1, and tail pointer adjusted if the data area is not full; otherwise, Carry = 1.

3. REMOVQ
Element removed from queue in register X, queue length decreased by 1, and head pointer adjusted if queue had an element; otherwise, Carry = 1.

Example

A typical sequence of queue operations would proceed as follows:

1. Initialize the queue. Call INITQ to set the head and tail pointers to the data area's base address, the queue length to 0, and the end pointer to the address just beyond the end of the data area.

2. Insert an element into the queue. Call INSRTQ to insert the element, increase the tail pointer by 2, and increase the queue length by 1.

3. Insert another element into the queue. Call INSRTQ again to insert the element, increase the tail pointer by 2, and increase the queue length by 1.

4. Remove an element from the queue. Call REMOVQ to remove an element, increase the head pointer by 2, and decrease the queue length by 1. Since the queue is organized on a first-in, first-out basis, the element removed is the first one inserted.

Registers used

1. INITQ: A, CC, U, X
2. INSRTQ: A, CC, X, Y
3. REMOVQ: A, CC, U, X, Y

Execution time

1. INITQ: 65 cycles
2. INSRTQ: 65 or 70 cycles, depending on whether wraparound is necessary
3. REMOVQ: 66 or 71 cycles, depending on whether wraparound is necessary

Program size 79 bytes

Data memory required None

```
*       Title           Queue Manager
*       Name:           INITQ, INSRTQ, REMOVQ
*
*       Purpose:        This program consists of three
*                       subroutines that manage a queue.
*
*                       INITQ initializes the empty queue.
*                       INSRTQ inserts a 16-bit element into
*                         the queue.
*                       REMOVQ removes a 16-bit element from
*                         the queue.
*
*       Entry:          INITQ
*                         Base address of queue in X
*                         Size of data area in words in A
*
*                       INSRTQ
*                         Base address of queue in X
*                         Element to be inserted in U
*
*                       REMOVQ
*                         Base address of queue in X
*
*       Exit:           INITQ
*                         Head pointer = Base address of data area
*                         Tail pointer = Base address of data area
*                         Queue length = 0
*                         End pointer = Base address of data area +
*                           2 * Size of data area in words
*
*                       INSRTQ
*                         If queue length is not buffer size,
*                              Element added to queue
*                              Tail pointer = Tail pointer + 2
*                              Queue length = Queue length + 1
*                              Carry = 0
```

```
*                         else Carry = 1
*
*                       REMOVQ
*                         If queue length is not zero,
*                             Element removed from queue in X
*                             Head pointer = Head pointer + 2
*                             Queue length = Queue length - 1
*                             Carry = 0
*                         else Carry = 1
*
*       Registers Used: INITQ
*                         A,B,CC,U,X
*                       INSRTQ
*                         A,CC,X,Y
*                       REMOVQ
*                         A,CC,U,X,Y
*
*       Time:           INITQ
*                         65 cycles
*                       INSRTQ
*                         65 or 70 cycles, depending on whether
*                         wraparound is necessary
*                       REMOVQ
*                         66 or 71 cycles, depending on whether
*                         wraparound is necessary
*
*       Size:           Program 79 bytes
*

*
*INITIALIZE AN EMPTY QUEUE
*HEADER CONTAINS:
*    1) SIZE OF DATA AREA IN WORDS (1 BYTE)
*    2) QUEUE LENGTH (1 BYTE)
*    3) HEAD POINTER (2 BYTES)
*    4) TAIL POINTER (2 BYTES)
*    5) END POINTER (2 BYTES)
*
INITQ:
          *
          *SET SIZE OF DATA AREA TO SPECIFIED VALUE
          *SET QUEUE LENGTH TO ZERO
          *
          LEAU      8,X             POINT TO START OF DATA AREA
          STA       ,X+             SET SIZE OF DATA AREA IN WORDS
          CLR       ,X+             QUEUE LENGTH = ZERO
          *
          *INITIALIZE HEAD AND TAIL POINTERS TO START OF DATA AREA
          *
          STU       ,X++            HEAD POINTER = START OF DATA AREA
          STU       ,X++            TAIL POINTER = START OF DATA AREA
          *
          *INITIALIZE END POINTER TO ADDRESS JUST BEYOND DATA AREA
          *
          TFR       A,B             EXTEND SIZE OF DATA AREA TO 16 BITS
          CLRA
```

```
          ASLB                        MULTIPLY SIZE OF DATA AREA TIMES 2
          ROLA                          SINCE SIZE IS IN WORDS
          LEAU      D,U               POINT JUST BEYOND END OF DATA AREA
          STU       ,X                END POINTER = ADDRESS JUST BEYOND
                                      * END OF DATA AREA
          RTS

*
*INSERT AN ELEMENT INTO A QUEUE
*
INSRTQ:
          *
          *EXIT WITH CARRY SET IF DATA AREA IS FULL
          *
          LDA       1,X               GET QUEUE LENGTH
          CMPA      ,X                COMPARE TO SIZE OF DATA AREA
          SEC                         INDICATE DATA AREA FULL
          BEQ       EXITIS            BRANCH (EXIT) IF DATA AREA IS FULL
          *
          *DATA AREA NOT FULL, SO STORE ELEMENT AT TAIL
          *ADD 1 TO QUEUE LENGTH
          *
          LDY       4,X               GET TAIL POINTER
          STU       ,Y                INSERT ELEMENT AT TAIL
          INC       1,X               ADD 1 TO QUEUE LENGTH
          *
          *INCREASE TAIL POINTER BY ONE 16-BIT ELEMENT (2 BYTES)
          *IF TAIL POINTER HAS REACHED END OF DATA AREA, SET IT
          *  BACK TO BASE ADDRESS
          *
          LEAY      2,Y               MOVE TAIL POINTER UP ONE ELEMENT
          CMPY      6,X               COMPARE TO END OF DATA AREA
          BNE       STORTP            BRANCH IF TAIL NOT AT END OF DATA
                                      * AREA
          LEAY      8,X               OTHERWISE, MOVE TAIL POINTER BACK TO
                                      * BASE ADDRESS OF DATA AREA
STORTP:
          STY       4,X               SAVE UPDATED TAIL POINTER
          CLC                         CLEAR CARRY (GOOD EXIT)
EXITIS:
          RTS

*
*REMOVE AN ELEMENT FROM A QUEUE
*
REMOVQ:
          *
          *EXIT WITH CARRY SET IF QUEUE IS EMPTY
          *
          LDA       1,X               GET QUEUE LENGTH
          SEC                         INDICATE QUEUE EMPTY
          BEQ       EXITRQ            BRANCH (EXIT) IF QUEUE IS EMPTY
          *
          *QUEUE NOT EMPTY, SO SUBTRACT 1 FROM QUEUE LENGTH
          *REMOVE ELEMENT FROM HEAD OF QUEUE
          *
```

```
            DEC       1,X             SUBTRACT 1 FROM QUEUE LENGTH
            LDU       2,X             GET HEAD POINTER
            LDY       ,U              GET ELEMENT FROM HEAD OF QUEUE
            *
            *MOVE HEAD POINTER UP ONE 16-BIT ELEMENT (2 BYTES)
            *IF HEAD POINTER HAS REACHED END OF DATA AREA, SET IT BACK
            *  TO BASE ADDRESS OF DATA AREA
            *
            LEAU      2,U             MOVE HEAD POINTER UP ONE ELEMENT
            CMPU      6,X             COMPARE TO END OF DATA AREA
            BNE       STORHP          BRANCH IF NOT AT END OF DATA AREA
            LEAU      8,X             OTHERWISE, MOVE HEAD POINTER BACK
                                      * TO BASE ADDRESS OF DATA AREA
STORHP:
            STU       2,X             SAVE NEW HEAD POINTER
            TFR       Y,X             MOVE ELEMENT TO X
            CLC                       INDICATE QUEUE NON-EMPTY,
                                      * ELEMENT FOUND
EXITRQ:
            RTS                       EXIT, CARRY INDICATES WHETHER
                                      * ELEMENT WAS FOUND (0 IF SO,
                                      * 1 IF NOT)

*
*           SAMPLE EXECUTION
*
*
SC7A:
            *
            *INITIALIZE EMPTY QUEUE
            *
            LDA       #5              DATA AREA HAS ROOM FOR 5 WORD-LENGTH
                                      * ELEMENTS
            LDX       #QUEUE          GET BASE ADDRESS OF QUEUE BUFFER
            JSR       INITQ           INITIALIZE QUEUE
            *
            *INSERT ELEMENTS INTO QUEUE
            *
            LDU       #$AAAA          ELEMENT TO BE INSERTED IS AAAA
            LDX       #QUEUE          GET BASE ADDRESS OF QUEUE
            JSR       INSRTQ          INSERT ELEMENT INTO QUEUE
            LDU       #$BBBB          ELEMENT TO BE INSERTED IS BBBB
            LDX       #QUEUE          GET BASE ADDRESS OF QUEUE
            JSR       INSRTQ          INSERT ELEMENT INTO QUEUE
            *
            *REMOVE ELEMENT FROM QUEUE
            *
            LDX       #QUEUE          GET BASE ADDRESS OF QUEUE
            JSR       REMOVQ          REMOVE ELEMENT FROM QUEUE
                                      * (X) = $AAAA (FIRST ELEMENT
                                      * INSERTED)
            BRA       SC7A            REPEAT TEST

*
*DATA
*
```

```
QUEUE   RMB     18          QUEUE BUFFER CONSISTS OF AN 8 BYTE
                            * HEADER FOLLOWED BY 10 BYTES FOR
                            * DATA (FIVE WORD-LENGTH ELEMENTS)
        END
```

7B Stack manager (INITST, PUSH, POP)

Manages a stack of 16-bit words on a first-in, last-out basis. The stack can contain up to 32 767 elements. Consists of the following routines:

1. INITST initializes the stack header, consisting of the pointer and its upper and lower bounds.

2. PUSH inserts an element into the stack if there is room for it.

3. POP removes an element from the stack if one is available.

Procedures

1. INITST sets the stack pointer and its lower bound to the base address of the stack's data area. It sets the upper bound to the address just beyond the end of the data area.

2. PUSH checks whether increasing the stack pointer by 2 will make it exceed its upper bound. If so, it sets the Carry flag. If not, it inserts the element at the stack pointer, increases the stack pointer by 2, and clears the Carry flag.

3. POP checks whether decreasing the stack pointer by 2 will make it less than its lower bound. If so, it sets the Carry flag. If not, it decreases the stack pointer by 2, removes the element, and clears the Carry flag.

Note that the stack grows toward higher addresses, unlike the 6809's hardware and user stacks, which grow toward lower addresses. Like the 6809's own stack pointers, this pointer always contains the next available memory address, *not* the last occupied address.

Entry conditions

1. INITST
Base address of stack in register X
Size of stack data area in words in register D

2. PUSH
Base address of stack in register X
Element in register D

3. POP
Base address of stack in register X

Exit conditions

1. INITST
Stack header set up with:

Stack pointer = Base address of stack's data area
Lower bound = Base address of stack's data area
Upper bound = Address just beyond end of stack's data area

2. PUSH
Element inserted into stack and stack pointer increased if there is room in the data area; otherwise, Carry = 1, indicating an overflow.

3. POP
Element removed from stack in register X and stack pointer decreased if stack was not empty; otherwise, Carry = 1, indicating an underflow.

Example

A typical sequence of stack operations proceeds as follows:

1. Initialize the empty stack with INITST. This involves setting the stack pointer and the lower bound to the base address of the stack's data area, and the upper bound to the address immediately beyond the end of the data area.

2. Insert an element into the stack. Call PUSH to put an element at the top of the stack and increase the stack pointer by 2.

3. Insert another element into the stack. Call PUSH to put an element at the top of the stack and increase the stack pointer by 2.

4. Remove an element from the stack. Call POP to decrease the stack pointer by 2 and remove an element from the top of the stack. Since the stack is organized on a last-in, first-out basis, the element removed is the latest one inserted.

Registers used

1. INITST: A, B, CC, U, X
2. PUSH: CC, U (D and X are unchanged)
3. POP: CC, U, X

Execution time:

1. INITST: 43 cycles
2. PUSH: 41 cycles
3. POP: 36 cycles

Program size

1. INITST: 13 bytes
2. PUSH: 19 bytes
3. POP: 14 bytes

Data memory required None

```
*       Title           Stack Manager
*       Name:           INITST, PUSH, POP
*
*
*       Purpose:        This program consists of three
*                       subroutines that manage a stack.
*
*                       INITST sets up the stack pointer and
*                         its upper and lower bounds
*                       PUSH inserts a 16-bit element into
*                         the stack.
*                       POP removes a 16-bit element from
*                         the stack.
*
*       Entry:          INITST
*                         Base address of stack in X
*                         Size of stack data area in words in D
*                       PUSH
*                         Base address of stack in X
*                         Element in D
*                       POP
*                         Base address of stack in X
*
*       Exit:           INITST
*                         Stack header set up with:
*                           Stack pointer = base address of stack
*                             data area
*                           Lower bound = base address of stack
*                             data area
*                           Upper bound = address just beyond end
*                             of stack data area
*
*                        PUSH
*                         If stack pointer is below upper bound,
```

```
*                            Element added to stack
*                            Stack pointer = Stack pointer + 2
*                            Carry = 0
*                          else Carry = 1
*
*                       POP
*                          If stack pointer is at or above lower bound,
*                            Element removed from stack in X
*                            Stack pointer = Stack pointer - 2
*                            Carry = 0
*                          else Carry = 1
*
*       Registers Used: INITST
*                         A,B,CC,U,X
*                       PUSH
*                         CC,U
*                       POP
*                         CC,U,X
*
*       Time:           INITST
*                         43 cycles
*                       PUSH
*                         41 cycles
*                       POP
*                         36 cycles
*
*       Size:           Program 46 bytes
*

*
*INITIALIZE AN EMPTY STACK
*HEADER CONTAINS:
*    1) STACK POINTER (2 BYTES)
*    2) LOWER BOUND (2 BYTES)
*    3) UPPER BOUND (2 BYTES)
*
          *
          *STACK POINTER = BASE ADDRESS OF STACK DATA AREA
          *LOWER BOUND = BASE ADDRESS OF STACK DATA AREA
          *
INITST:
          LEAU      6,X             GET BASE ADDRESS OF STACK DATA AREA
          STU       ,X++            STORE IT AS INITIAL STACK POINTER
          STU       ,X++            STORE IT AS LOWER BOUND ALSO
          *
          *UPPER BOUND = ADDRESS JUST BEYOND END OF STACK DATA AREA
          *
          ASLB                      MULTIPLY SIZE OF DATA AREA BY 2
          RORA                        SINCE SIZE IS IN WORDS
          LEAU      D,U             FIND ADDRESS JUST BEYOND END OF
                                    * STACK DATA AREA
          STU       ,X              STORE IT AS UPPER BOUND
          RTS

*
*INSERT A 16-BIT ELEMENT INTO A STACK
```

```
*
PUSH:
          *
          *EXIT INDICATING OVERFLOW (CARRY SET) IF STACK IS FULL
          *
          LDU       ,X              GET STACK POINTER
          LEAU      2,U             INCREMENT STACK POINTER BY 2
          CMPU      4,X             COMPARE TO UPPER BOUND
          BCC       OVRFLW          BRANCH IF STACK POINTER AT OR
                                    * ABOVE UPPER BOUND
                                    * NOTE: THIS COMPARISON HANDLES
                                    * SITUATIONS IN WHICH THE STACK
                                    * POINTER HAS BECOME MISALIGNED OR
                                    * GONE OUTSIDE ITS NORMAL RANGE.
          *
          *NO OVERFLOW - INSERT ELEMENT INTO STACK
          *UPDATE STACK POINTER
          *
          STD       -2,U            INSERT ELEMENT INTO STACK
          STU       ,X              SAVE INCREMENTED STACK POINTER
          CLC                       CLEAR CARRY TO INDICATE INSERTION
                                    * WORKED
          RTS
          *
          *OVERFLOW - SET CARRY AND EXIT
          *
OVRFLW:
          SEC                       SET CARRY TO INDICATE OVERFLOW
          RTS

*
*REMOVE A 16-BIT ELEMENT FROM A STACK
*
POP:
          *
          *EXIT INDICATING UNDERFLOW (CARRY SET) IF STACK IS EMPTY
          *
          LDU       ,X              GET STACK POINTER
          LEAU      -2,U            DECREASE STACK POINTER BY 2
          CMPU      2,X             COMPARE TO LOWER BOUND
          BCS       EXITSP          BRANCH (EXIT) IF BELOW LOWER BOUND
                                    * NOTE: THIS COMPARISON HANDLES
                                    * SITUATIONS IN WHICH THE STACK
                                    * POINTER HAS BECOME MISALIGNED OR
                                    * GONE OUTSIDE ITS NORMAL RANGE.
          *
          *NO UNDERFLOW - REMOVE ELEMENT AND DECREASE STACK POINTER
          *
          STU       ,X              SAVE UPDATED STACK POINTER
          LDX       ,U              REMOVE ELEMENT
EXITSP:
          RTS                       EXIT

*
*       SAMPLE EXECUTION
*
```

```
*
SC7B:
          *
          *INITIALIZE EMPTY STACK
          *
          LDX     #STACK          GET BASE ADDRESS OF STACK
          LDD     #STKSZ          GET SIZE OF STACK DATA AREA IN WORDS
          JSR     INITST          INITIALIZE STACK HEADER
          *
          *PUT ELEMENT 1 IN STACK
          *
          LDD     ELEM1           GET ELEMENT 1
          LDX     #STACK          GET BASE ADDRESS OF STACK AREA
          JSR     PUSH            PUT ELEMENT 1 IN STACK
          *
          *PUT ELEMENT 2 IN STACK
          *
          LDD     ELEM2           GET ELEMENT 2
          LDX     #STACK          GET BASE ADDRESS OF STACK AREA
          JSR     PUSH            PUT ELEMENT 2 IN STACK
          *
          *REMOVE ELEMENT FROM STACK
          *
          LDX     #STACK          GET BASE ADDRESS OF STACK
          JSR     POP             REMOVE ELEMENT FROM STACK TO X
                                  * X NOW CONTAINS ELEMENT 2
                                  * SINCE STACK IS ORGANIZED ON A
                                  * LAST-IN, FIRST-OUT BASIS
          BRA     SC7B            LOOP FOR MORE TESTS

*
*DATA
*
STACK     RMB     16              STACK HAS ROOM FOR 6-BYTE HEADER
                                  * AND 10 BYTES OF DATA (5 WORD-
                                  * LENGTH ELEMENTS)
ELEM1     RMB     2               2 BYTE ELEMENT
ELEM2     RMB     2               2 BYTE ELEMENT
STKSZ     EQU     5               SIZE OF STACK DATA AREA IN WORDS

          END
```

7C Singly linked list manager (INLST, RMLST)

Manages a linked list of elements, each of which has the address of the next element (or 0 if there is no next element) in its first two bytes. Consists of the following routines:

1. INLST inserts an element into the list, given the element it follows.

2. RMLST removes an element from the list (if one exists), given the element it follows.

Note that you can add or remove elements anywhere in the linked list. All you need is the address of the preceding element to provide the linkage.

Procedures

1. INLST obtains the link from the preceding element, sets that element's link to the new element, and sets the new element's link to the one from the preceding element.

2. RMLST first determines if there is a following element. If not, it sets the Carry flag. If so, it obtains that element's link and puts it in the current element. This unlinks the element and removes it from the list.

Entry conditions

1. INLST
Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of base address of preceding element
Less significant byte of base address of preceding element

More significant byte of base address of new element
Less significant byte of base address of new element

2. RMLST
Base address of preceding element in X

Exit conditions

1. INLST
Element inserted into list with preceding element linked to it. It is linked to the element that had been linked to the preceding element.

2. RMLST
If there is a following element, it is removed from the list, its base address is placed in register X, and the Carry flag is cleared.

Otherwise, register X = 0 and Carry flag = 1.

Example

A typical sequence of operations on a linked list is:

1. Initialize the empty list by setting the link in the header to zero.

2. Insert an element into the list by using the base address of the header as the previous element.

3. Insert another element into the list by using the base address of the element just inserted as the previous element.

4. Remove the first element from the linked list by using the base address of the header as the previous element. Note that we can remove either element from the list by supplying the proper previous element.

Registers used:

1. INLST: All
2. RMLST: CC, D, U, X

Execution time:

1. INLST: 29 cycles
2. RMLST: 35 cycles

Program size

1. INLST: 10 bytes
2. RMLST: 15 bytes

Data memory required None

```
*       Title           Singly Linked List Manager
*       Name:           INLST, RMLST
*
*       Purpose:        This program consists of two subroutines
*                       that manage a singly linked list.
*
*                       INLST inserts an element into the linked
*                         list.
*                       RMLST removes an element from the linked
*                         list.
*
*       Entry:          INLST
*                        TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          High byte of previous element's address
*                          Low byte of previous element's address
*                          High byte of entry address
*                          Low byte of entry address
*                       RMLST
*                          Base address of preceding element in
*                          register X
*
*       Exit:           INLST
*                         Element added to list
*                       RMLST
*                         If following element exists,
*                           its base address is in register X
*                           Carry = 0
*                         else
*                           register X = 0
*                           Carry = 1
*
*       Registers Used: INLST
*                         All
*                       RMLST
*                         CC,D,U,X
*
*       Time:           INLST
*                         29 cycles
*                       RMLST
*                         35 cycles
*
*       Size:           Program 25 bytes
*
```

```
*
*         INSERT AN ELEMENT INTO A SINGLY LINKED LIST
*
INLST:
          *
          *UPDATE LINKS TO INCLUDE NEW ELEMENT
          *LINK PREVIOUS ELEMENT TO NEW ELEMENT
          *LINK NEW ELEMENT TO ELEMENT FORMERLY LINKED TO
          *  PREVIOUS ELEMENT
          *
          PULS      X,Y,U           GET ELEMENTS, RETURN ADDRESS
          LDD       ,Y              GET LINK FROM PREVIOUS ELEMENT
          STD       ,U              STORE LINK IN NEW ELEMENT
          STU       ,Y              STORE NEW ELEMENT AS LINK IN
                                    * PREVIOUS ELEMENT
          *
          *NOTE: IF LINKS ARE NOT IN FIRST TWO BYTES OF ELEMENTS, PUT
          *  LINK OFFSET IN LAST 3 INSTRUCTIONS
          *
          *
          *EXIT
          *
          JMP       ,X              EXIT TO RETURN ADDRESS

*
*         REMOVE AN ELEMENT FROM A SINGLY LINKED LIST
*
RMLST:
          *
          *EXIT INDICATING FAILURE (CARRY SET) IF NO FOLLOWING ELEMENT
          *
          LDU       ,X              GET LINK TO FOLLOWING ELEMENT
          SEC                       INDICATE NO ELEMENT FOUND
          BEQ       RMEXIT          BRANCH IF NO ELEMENT FOUND
          *
          *UNLINK REMOVED ELEMENT BY TRANSFERRING ITS LINK TO
          *  PREVIOUS ELEMENT
          *NOTE: IF LINKS NOT IN FIRST TWO BYTES OF ELEMENTS, PUT
          *  LINK OFFSET IN STATEMENTS
          *
          LDD       ,U              GET LINK FROM REMOVED ELEMENT
          STD       ,X              MOVE IT TO PREVIOUS ELEMENT
          CLC                       INDICATE ELEMENT FOUND
          *
          *EXIT
          *
RMEXIT:
          TFR       U,X             EXIT WITH BASE ADDRESS OF REMOVED
                                    * ELEMENT OR 0 IN X
          RTS                       CARRY = 0 IF ELEMENT FOUND, 1
                                    * IF NOT

*
*         SAMPLE EXECUTION
*
*
```

```
SC7C:
          *
          *INITIALIZE EMPTY LINKED LIST
          *
          LDD       #0              CLEAR LINKED LIST HEADER
          STD       LLHDR           0 INDICATES NO NEXT ELEMENT
          *
          *INSERT AN ELEMENT INTO LINKED LIST
          *
          LDY       #ELEM1          GET BASE ADDRESS OF ELEMENT 1
          LDX       #LLHDR          GET PREVIOUS ELEMENT (HEADER)
          PSHS      X,Y             SAVE PARAMETERS IN STACK
          JSR       INLST           INSERT ELEMENT INTO LIST
          *
          *INSERT ANOTHER ELEMENT INTO LINKED LIST
          *
          LDY       #ELEM2          GET BASE ADDRESS OF ELEMENT 2
          LDX       #ELEM1          GET PREVIOUS ELEMENT
          PSHS      X,Y             SAVE PARAMETERS IN STACK
          JSR       INLST           INSERT ELEMENT INTO LIST
          *
          *REMOVE FIRST ELEMENT FROM LINKED LIST
          *
          LDX       #LLHDR          GET PREVIOUS ELEMENT
          JSR       RMLST           REMOVE ELEMENT FROM LIST
          *                         END UP WITH HEADER LINKED TO
          *                           SECOND ELEMENT
          *                         X CONTAINS BASE ADDRESS OF
          *                           FIRST ELEMENT
          BRA       SC7C            REPEAT TEST

*
*DATA
*
LLHDR     RMB       2               LINKED LIST HEADER
ELEM1     RMB       2               ELEMENT 1 - HEADER (LINK) ONLY
ELEM2     RMB       2               ELEMENT 2 - HEADER (LINK) ONLY
          END
```

7D Doubly linked list manager (INDLST, RMDLST)

Manages a doubly linked list of elements. Each element contains the address of the next element (or 0 if there is no next element) in its first two bytes. It contains the address of the preceding element (or 0 if there is no preceding element) in its next two bytes. Consists of the following routines:

1. INDLST inserts an element into the list, linking it to the preceding and following elements.

2. RMDLST first determines if there is a following element. If so, it obtains its address and removes its links from the preceding and following elements.

As with a singly linked list, you can add or remove elements from anywhere in the list. All you need is the address of the preceding element to provide the proper linkage.

Procedures:

1. INDLST first obtains the forward link from the preceding element (i.e. the address of the following element). It then changes the links as follows:

(a) The new element becomes the forward link of the preceding element.
(b) The preceding element becomes the backward link of the new element.
(c) The old forward link from the preceding element becomes the forward link of the new element.
(d) The new element becomes the backward link of the following element.

2. RMDLST first determines if there is a following element. If not, it sets the Carry flag. If so, it obtains that element's forward link (the next element) and makes it the forward link of the preceding element. It also makes the preceding element into the backward link of the next element. This unlinks the element, removing it from the list.

Entry conditions

1. INDLST
Order in stack (starting from the top)

More significant byte of return address
Less significant byte of return address

More significant byte of base address of preceding element
Less significant byte of base address of preceding element

More significant byte of base address of new element
Less significant byte of base address of new element

2. RMDLST
Base address of preceding element in register X

Exit conditions

1. INDLST
Element added to list with preceding and succeeding elements linked to it.

2. RMDLST
If there is a following element, it is removed from the list, its base address is placed in register X, and the Carry flag is cleared.

Otherwise, register X = 0 and Carry flag = 1.

Example

A typical sequence of operations on a doubly linked list is:

1. Initialize the empty list by setting both links in the header to zero.

2. Insert an element into the list by using the base address of the header as the previous element.

3. Insert another element into the list by using the base address of the element just added as the previous element.

4. Remove the first element from the list by using the base address of the header as the previous element. Note that we can remove either element from the list by supplying the proper previous element.

Registers used

1. INDLST: All
2. RMDLST: CC, U, X, Y

Execution time

1. INDLST: 53 cycles
2. RMDLST: 44 cycles

Program size

1. INDLST: 17 bytes
2. RMDLST: 18 bytes

Data memory required None

```
*       Title           Doubly Linked List Manager
*       Name:           INDLST, RMDLST
*
*       Purpose:        This program consists of two subroutines
*                       that manage a doubly linked list.
*
*                       INDLST inserts an element into the doubly
*                         linked list.
*                       RMDLST removes an element from the
*                         doubly linked list.
*
*       Entry:          INDLST
*                        TOP OF STACK
*                          High byte of return address
*                          Low byte of return address
*                          High byte of previous element's address
*                          Low byte of previous element's address
*                          High byte of entry address
*                          Low byte of entry address
*                       RMDLST
*                          Base address of preceding element in
*                          register X
*
*       Exit:           INDLST
*                         Element inserted into list
*                       RMDLST
*                         If following element exists,
*                           its base address is in register X
*                           Carry = 0
```

```
*                         else
*                           register X = 0
*                           Carry = 1
*
*       Registers Used: INDLST
*                         All
*                       RMDLST
*                         CC,U,X,Y
*
*       Time:           INDLST
*                         53 cycles
*                       RMDLST
*                         44 cycles
*
*       Size:           Program 35 bytes
*

*
*         INSERT AN ELEMENT INTO A DOUBLY LINKED LIST
*
INDLST:
          *
          *UPDATE LINKS TO INCLUDE NEW ELEMENT
          *LINK PREVIOUS ELEMENT TO NEW ELEMENT
          *LINK NEW ELEMENT TO PREVIOUS AND FOLLOWING ELEMENTS
          *LINK FOLLOWING ELEMENT TO NEW ELEMENT
          *
          PULS    D,X,Y             GET RETURN ADDRESS, ELEMENTS
          LDU     2,X               GET FOLLOWING ELEMENT
          STY     2,X               MAKE NEW ELEMENT INTO PREVIOUS
                                    * ELEMENT'S FORWARD LINK
          STX     ,Y                MAKE PREVIOUS ELEMENT INTO NEW
                                    * ELEMENT'S BACKWARD LINK
          STU     2,Y               MAKE FOLLOWING ELEMENT INTO NEW
                                    * ELEMENT'S FORWARD LINK
          STY     ,U                MAKE NEW ELEMENT INTO FOLLOWING
                                    * ELEMENT'S BACKWARD LINK
          *
          *NOTE: IF LINKS ARE NOT IN FIRST FOUR BYTES OF ELEMENTS,
          *  PUT LINK OFFSETS IN LAST 5 INSTRUCTIONS
          *
          *
          *EXIT
          *
          PSHS    D                 PUT RETURN ADDRESS BACK IN STACK
          RTS                       EXIT

*
*         REMOVE AN ELEMENT FROM A DOUBLY LINKED LIST
*
RMDLST:
          *
          *EXIT INDICATING FAILURE (CARRY SET) IF NO FOLLOWING ELEMENT
          *
          LDY     2,X               GET LINK TO FOLLOWING ELEMENT
          SEC                       INDICATE NO ELEMENT FOUND
```

```
          BEQ     RMDXIT              BRANCH IF NO ELEMENT FOUND
          *
          *ELEMENT EXISTS SO UNLINK IT BY TRANSFERRING ITS
          *  FORWARD LINK TO PREVIOUS ELEMENT AND ITS BACKWARD
          *  LINK TO FOLLOWING ELEMENT
          *NOTE: IF LINKS ARE NOT IN THE FIRST FOUR BYTES OF THE
          *  ELEMENTS, PUT CORRECT LINK OFFSETS IN STATEMENTS
          *
          LDU     2,Y                 GET FOLLOWING ELEMENT
          STU     2,X                 MAKE FOLLOWING ELEMENT INTO FORWARD
                                      * LINK OF PRECEDING ELEMENT
          STX     ,U                  MAKE PRECEDING ELEMENT INTO BACKWARD
                                      * LINK OF FOLLOWING ELEMENT
          CLC                         INDICATE ELEMENT FOUND
          *
          *EXIT
          *
RMDXIT:
          TFR     Y,X                 EXIT WITH BASE ADDRESS OF REMOVED
                                      * ELEMENT OR 0 IN X
          RTS                         CARRY = 0 IF ELEMENT FOUND, 1 IF NOT


*
*         SAMPLE EXECUTION
*
*
SC7D:
          *
          *INITIALIZE EMPTY DOUBLY LINKED LIST
          *
          LDD     #0                  CLEAR LINKED LIST HEADER
          STD     HDRFWD              FORWARD LINK
          STD     HDRBCK              BACKWARD LINK
                                      * 0 INDICATES NO LINK IN THAT
                                      * DIRECTION
          *
          *INSERT ELEMENT INTO DOUBLY LINKED LIST
          *
          LDY     #ELEM1              GET BASE ADDRESS OF ELEMENT 1
          LDX     #HDRFWD             GET PREVIOUS ELEMENT (HEADER)
          PSHS    X,Y                 SAVE PARAMETERS IN STACK
          JSR     INDLST              INSERT ELEMENT INTO LIST
          *
          *INSERT ANOTHER ELEMENT INTO DOUBLY LINKED LIST
          *
          LDY     #ELEM2              GET BASE ADDRESS OF ELEMENT 2
          LDX     #ELEM1              GET PREVIOUS ELEMENT
          PSHS    X,Y                 SAVE PARAMETERS IN STACK
          JSR     INDLST              INSERT ELEMENT INTO LIST
          *
          *REMOVE FIRST ELEMENT FROM DOUBLY LINKED LIST
          *
          LDX     #HDRFWD             GET PREVIOUS ELEMENT
          JSR     RMDLST              REMOVE ELEMENT FROM LIST
          *                           END UP WITH HEADER LINKED TO
```

```
*                          SECOND ELEMENT
*                        X CONTAINS BASE ADDRESS
*                          OF FIRST ELEMENT
*
        BRA     SC7D       REPEAT TEST

*
*DATA
*
HDRFWD  RMB     2          HEADER - FORWARD LINK
HDRBCK  RMB     2          HEADER - BACKWARD LINK
ELEM1   RMB     2          ELEMENT 1 - HEADER (LINKS) ONLY
ELEM2   RMB     2          ELEMENT 2 - HEADER (LINKS) ONLY
        END
```

8 Input/output

8A Read a line from a terminal (RDLINE)

Reads a line of ASCII characters ending with a carriage return and saves it in a buffer. Defines the control characters Control H (08 hex), which deletes the latest character, and Control X (18 hex), which deletes the entire line. Sends a bell character (07 hex) to the terminal if the buffer overflows. Echoes each character placed in the buffer. Echoes non-printable characters as an up-arrow or caret (^) followed by the printable equivalent (see Table 8-1). Sends a new line sequence (typically carriage return, line feed) to the terminal before exiting.

RDLINE assumes the following system-dependent subroutines:

1. RDCHAR reads a character from the terminal and puts it in register A.

2. WRCHAR sends the character in register A to the terminal.

3. WRNEWL sends a new line sequence to the terminal.

These subroutines are assumed to change all user registers.

RDLINE is an example of a terminal input handler. The control characters and I/O subroutines in a real system will, of course, be computer-dependent. A specific example in the listing is for a Radio Shack Color Computer with the following pointers to BASIC routines in ROM:

1. A000 and A001 contain a pointer to the routine that polls the keyboard and returns with either 0 (no key pressed) a character in register A.

2. A002 and A003 contain a pointer to the routine that sends the character in register A to an output device. The unit number (00 = screen, FE = printer) is in memory location 006F.

Procedure The program starts the loop by reading a character. If it is a carriage return, the program sends a new line sequence to the terminal and exits. Otherwise, it checks for the special characters Control H and Control X. If the buffer is not empty, Control H makes the program decrement the buffer pointer and character count by 1 and send a backspace string (cursor left on the Color Computer) to the terminal. Control X makes the program delete characters until the buffer is empty.

If the character is not special, the program determines whether the buffer is full. If it is, the program sends a bell character to the terminal. If not, the program stores the character in the buffer, echoes it to the terminal, and increments the character count and buffer pointer.

Table 8-1 ASCII control characters and printable equivalents

Name	Hex value	Printable equivalent
NUL	00	Control @
SOH	01	Control A
STX	02	Control B
ETX	03	Control C
EOT	04	Control D
ENQ	05	Control E
ACK	06	Control F
BEL	07	Control G
BS	08	Control H
HT	09	Control I
LF	0A	Control J
VT	0B	Control K
FF	0C	Control L
CR	0D	Control M
SO	0E	Control N
SI	0F	Control O

DLE	10	Control P
DC1	11	Control Q
DC2	12	Control R
DC3	13	Control S
DC4	14	Control T
NAK	15	Control U
SYN	16	Control V
ETB	17	Control W
CAN	18	Control X
EM	19	Control Y
SUB	1A	Control Z
ESC	1B	Control [
FS	1C	Control \
GS	1D	Control]
RS	1E	Control ^
VS	1F	Control _

Before echoing a character or deleting one from the display, the program must determine whether it is printable. If not (i.e. it is a non-printable ASCII control code), the program must display or delete two characters, the control indicator (up-arrow or caret) and the printable equivalent (see Table 8-1). Note, however, that the character is stored in its non-printable form. On the Radio Shack Color Computer, control characters are generated by pressing the down-arrow key, followed by another key. For example, to enter Control X, you must press down-arrow, then X.

Entry conditions

Base address of buffer in register X
Length (size) of buffer in bytes in register A

Exit conditions

Number of characters in the buffer in register A

Examples

1. Data: Line from keyboard is 'ENTERcr'

Result: Character count = 5 (line length)
Buffer contains 'ENTER'
'ENTER' echoed to terminal, followed by the new line sequence (typically either carriage return, line feed or just carriage return)
Note that the 'cr' (carriage return) character does not appear in the buffer.

2. Data: Line from keyboard is 'DMcontrolHNcontrolXENTET-controlHRcr'

Result: Character count = 5 (length of final line after deletions)
Buffer contains 'ENTER'
'DMBackspaceStringNBackspaceStringBackspaceString ENTETBackspaceStringR' sent to terminal, followed by a new line sequence. The backspace string deletes a character from the screen and moves the cursor left one space.
The sequence of operations is as follows:

Character typed	Initial buffer	Final buffer	Sent to terminal
D	empty	'D'	D
M	'D'	'DM'	M
control H	'DM'	'D'	backspace string
N	'D'	'DN'	N
control X	'DN'	empty	2 backspace strings
E	empty	'E'	E
N	'E'	'EN'	N
T	'EN'	'ENT'	T
E	'ENT'	'ENTE'	E
T	'ENTE'	'ENTET'	T
control H	'ENTET'	'ENTE'	backspace string
R	'ENTE'	'ENTER'	R
cr	'ENTER'	'ENTER'	New line string

What happened is the following:

(a) The operator types 'D', 'M'.

(b) The operator sees that 'M' is wrong (it should be 'N'), presses Control H to delete it, and types 'N'.

(c) The operator then sees that the initial 'D' is also wrong (it should

be 'E'). Since the error is not in the latest character, the operator presses Control X to delete the entire line, and then types 'ENTET'.

(d) The operator notes that the second 'T' is wrong (it should be 'R'), presses Control H to delete it, and types 'R'.

(e) The operator types a carriage return to end the line.

Registers used All

Execution time Approximately 76 cycles to put an ordinary character in the buffer, not considering the execution time of either RDCHAR or WRCHAR

Program size 139 bytes

Data memory required None

Special cases

1. Typing Control H (delete one character) or Control X (delete the entire line) when the buffer is empty has no effect.

2. The program discards an ordinary character received when the buffer is full, and sends a bell character to the terminal (ringing the bell).

```
*       Title           Read Line
*       Name:           RDLINE
*
*
*       Purpose:        Read characters from an input device until
*                       a carriage return is found. Defines the
*                       control characters
*                         Control H -- Delete latest character.
*                         Control X -- Delete all characters.
*                       Any other control character is placed in
*                       the buffer and displayed as the equivalent
*                       printable ASCII character preceded by an
*                       up-arrow or caret.
*
*       Entry:          Register X = Base address of buffer
*                       Register A = Length of buffer in bytes
```

```
*
*       Exit:           Register A = Number of characters in buffer
*
*       Registers Used: All
*
*       Time:           Not applicable.
*
*       Size:           Program 139 bytes
*

*EQUATES
BELL    EQU     $07     BELL CHARACTER
BSKEY   EQU     $08     BACKSPACE KEYBOARD CHARACTER
CR      EQU     $0D     CARRIAGE RETURN FOR CONSOLE
CRKEY   EQU     $0D     CARRIAGE RETURN KEYBOARD CHARACTER
CSRLFT  EQU     $08     MOVE CURSOR LEFT FOR CONSOLE
CTLOFF  EQU     $40     OFFSET FROM CONTROL CHARACTER TO PRINTED
*                         FORM (E.G., CONTROL-X TO X)
DLNKEY  EQU     $18     DELETE LINE KEYBOARD CHARACTER
DNARRW  EQU     $0A     DOWN-ARROW KEY (USED AS CONTROL INDICATOR
*                         ON KEYBOARD
LF      EQU     $0A     LINE FEED FOR CONSOLE
SPACE   EQU     $20     SPACE CHARACTER (ALSO MARKS END OF CONTROL
*                         CHARACTERS IN ASCII SEQUENCE)
STERM   EQU     $24     STRING TERMINATOR (DOLLAR SIGN)
UPARRW  EQU     $5E     UP-ARROW OR CARET USED AS CONTROL INDICATOR

RDLINE:
        *
        *INITIALIZE CHARACTER COUNT TO ZERO, SAVE BUFFER LENGTH
        *
INIT:
        CLRB                    CHARACTER COUNT = 0
        PSHS    A               SAVE BUFFER LENGTH IN STACK
        *
        *READ LOOP
        *READ CHARACTERS UNTIL A CARRIAGE RETURN IS TYPED
        *
RDLOOP:
        JSR     RDCHAR          READ CHARACTER FROM KEYBOARD
                                *DOES NOT ECHO CHARACTER
        *
        *CHECK FOR CARRIAGE RETURN, EXIT IF FOUND
        *
        CMPA    #CR             CHECK FOR CARRIAGE RETURN
        BEQ     EXITRD          END OF LINE IF CARRIAGE RETURN
        *
        *CHECK FOR BACKSPACE AND DELETE CHARACTER IF FOUND
        *
        CMPA    #BSKEY          CHECK FOR BACKSPACE KEY
        BNE     RDLP1           BRANCH IF NOT BACKSPACE
        JSR     BACKSP          IF BACKSPACE, DELETE ONE CHARACTER
        BRA     RDLOOP           THEN START READ LOOP AGAIN
        *
        *CHECK FOR DELETE LINE CHARACTER AND EMPTY BUFFER IF FOUND
        *
```

```
RDLP1:
        CMPA    #DLNKEY         CHECK FOR DELETE LINE KEY
        BNE     RDLP2           BRANCH IF NOT DELETE LINE
DEL1:
        JSR     BACKSP          DELETE A CHARACTER
        TSTB                    CHECK IF BUFFER EMPTY
        BNE     DEL1            CONTINUE UNTIL BUFFER EMPTY
                                *THIS ACTUALLY BACKS UP OVER EACH
                                * CHARACTER RATHER THAN JUST MOVING
                                * UP A LINE
        BRA     RDLOOP
        *
        *KEYBOARD ENTRY IS NOT A SPECIAL CHARACTER
        *CHECK IF BUFFER IS FULL
        *IF FULL, RING BELL AND CONTINUE
        *IF NOT FULL, STORE CHARACTER AND ECHO
        *
RDLP2:
        CMPA    ,S              COMPARE COUNT AND BUFFER LENGTH
        BCS     STRCH           JUMP IF BUFFER NOT FULL
        LDA     #BELL           BUFFER FULL, RING THE TERMINAL'S BELL
        JSR     WRCHAR
        BRA     RDLOOP          THEN CONTINUE THE READ LOOP
        *
        *BUFFER NOT FULL, STORE CHARACTER
        *
STRCH:
        STA     ,X+             STORE CHARACTER IN BUFFER
        INCB                    INCREMENT CHARACTER COUNT
        *
        *IF CHARACTER IS CONTROL, THEN OUTPUT
        * UP-ARROW FOLLOWED BY PRINTABLE EQUIVALENT
        *
        CMPA    #SPACE          CONTROL CHARACTER IF CODE IS
*                                 BELOW SPACE (20 HEX) IN ASCII
*                                 SEQUENCE
        BCC     PRCH            JUMP IF A PRINTABLE CHARACTER
        PSHS    A               SAVE NONPRINTABLE CHARACTER
        LDA     #UPARRW         WRITE UP-ARROW OR CARET
        JSR     WRCHAR
        PULS    A               RECOVER NONPRINTABLE CHARACTER
        ADDA    #CTLOFF                 CHANGE TO PRINTABLE FORM
PRCH:
        JSR     WRCHAR          ECHO CHARACTER TO TERMINAL
        BRA     RDLOOP          THEN CONTINUE READ LOOP
        *
        *EXIT
        *SEND NEW LINE SEQUENCE (USUALLY CR,LF) TO TERMINAL
        *LINE LENGTH = CHARACTER COUNT
        *
EXITRD:
        JSR     WRNEWL          ECHO NEW LINE SEQUENCE
        TFR     B,A             RETURN LINE LENGTH IN A
        LEAS    1,S             REMOVE BUFFER LENGTH FROM STACK
        RTS
```

```
*************************************************************
*
* THE FOLLOWING SUBROUTINES ARE TYPICAL EXAMPLES USING THE
* BASIC CALLS FOR THE RADIO SHACK TRS-80 COLOR COMPUTER
*
*************************************************************

*COLOR COMPUTER EQUATES
KBDPTR  EQU     $A000           POINTER TO KEYBOARD INPUT ROUTINE
*                                 CHARACTER ENDS UP IN REGISTER A
*                                 ZERO FLAG = 1 IF NO CHARACTER,
*                                   0 IF CHARACTER
OUTPTR  EQU     $A002           POINTER TO OUTPUT ROUTINE
*                                 UNIT NUMBER GOES IN LOCATION
*                                   $006F (0 = SCREEN)
*                                 CHARACTER GOES IN REGISTER A


*************************************************************
*ROUTINE: RDCHAR
*PURPOSE: READ A CHARACTER BUT DO NOT ECHO TO OUTPUT DEVICE
*ENTRY: NONE
*EXIT:  REGISTER A = CHARACTER
*REGISTERS USED: ALL
*************************************************************

RDCHAR:
        *
        *WAIT FOR CHARACTER FROM CONSOLE
        *EXIT UNLESS IT IS CONTROL INDICATOR
        *
        JSR     [KBDPTR]        POLL KEYBOARD
        BEQ     RDCHAR          LOOP UNTIL KEY PRESSED
        CMPA    #DNARRW         CHECK IF CONTROL CHARACTER
        BNE     RDCHXT          EXIT IF NOT CONTROL
        *
        *IF CONTROL CHARACTER, WAIT UNTIL NEXT KEY IS READ
        *THEN CONVERT NEXT KEY TO ASCII CONTROL CHARACTER
        *
CNTCHR:
        JSR     [KBDPTR]         POLL KEYBOARD
        BEQ     CNTCHR          LOOP UNTIL KEY PRESSED
        CMPA    #'A             COMPARE WITH ASCII A
        BLO     RDCHXT          EXIT IF LESS THAN A
        SUBA    #CTLOFF         ELSE CONVERT TO CONTROL
                                * CHARACTER EQUIVALENT
        *
        *EXIT WITH CHARACTER IN REGISTER A
        *
RDCHXT:
        RTS                     RETURN ASCII CHARACTER IN REGISTER A

*****************************************
*ROUTINE: WRCHAR
*PURPOSE: WRITE CHARACTER TO OUTPUT DEVICE
```

```
*ENTRY:          REGISTER A = CHARACTER
*EXIT:  NONE
*REGISTERS USED: ALL
****************************************

WRCHAR:
        *
        *WRITE A CHARACTER TO OUTPUT DEVICE
        *LOCATION $006F MUST CONTAIN UNIT NUMBER (0 = SCREEN)
        *
        JSR     [OUTPTR]        SEND CHARACTER
        RTS


*********************************************************
*ROUTINE: WRNEWL
*PURPOSE: ISSUE NEW LINE SEQUENCE TO TERMINAL
*         NORMALLY, THIS SEQUENCE IS A CARRIAGE RETURN AND
*         LINE FEED, BUT SOME COMPUTERS REQUIRE ONLY
*         A CARRIAGE RETURN.
*ENTRY: NONE
*EXIT:  NONE
*REGISTERS USED: ALL
*********************************************************

WRNEWL:
        *SEND NEW LINE STRING TO TERMINAL
        LDY     #NLSTRG         POINT TO NEW LINE STRING
        JSR     WRSTRG          SEND STRING TO TERMINAL
        RTS
NLSTRG:         FCB     CR,LF,STERM     NEW LINE STRING

****************************************
*ROUTINE: BACKSP
*PURPOSE: PERFORM A DESTRUCTIVE BACKSPACE
*ENTRY: A = NUMBER OF CHARACTERS IN BUFFER
*       X = NEXT AVAILABLE BUFFER ADDRESS
*EXIT:  IF NO CHARACTERS IN BUFFER
*         Z = 1
*       ELSE
*         Z = 0
*         CHARACTER REMOVED FROM BUFFER
*REGISTERS USED: ALL
****************************************

BACKSP:
        *
        *CHECK FOR EMPTY BUFFER
        *
        TSTB                    TEST NUMBER OF CHARACTERS
        BEQ     EXITBS          BRANCH (EXIT) IF BUFFER EMPTY
        *
        *OUTPUT BACKSPACE STRING
        * TO REMOVE CHARACTER FROM DISPLAY
        *
        LEAX    -1,X            DECREMENT BUFFER POINTER
```

```
        LDA     ,X              GET CHARACTER
        CMPA    #SPACE          IS IT A CONTROL CHARACTER?
        BNE     BS1             NO, BRANCH, DELETE ONLY ONE CHARACTER
        LDX     #BSSTRG         YES, DELETE 2 CHARACTERS
                                * (UP-ARROW AND PRINTABLE EQUIVALENT)
        JSR     WRSTRG          WRITE BACKSPACE STRING
BS1:    LDX     #BSSTRG
        JSR     WRSTRG          WRITE BACKSPACE STRING
        *DECREMENT CHARACTER COUNT BY 1
        DECB                    ONE LESS CHARACTER IN BUFFER
EXITBS:
        RTS
*
*DESTRUCTIVE BACKSPACE STRING FOR TERMINAL
*THE COLOR COMPUTER DOES NOT PROVIDE A FLASHING CURSOR WHEN
*  RUNNING THIS ROUTINE, SO ONLY A BACKSPACE CHARACTER IS
*  NECESSARY
*IF THE CURSOR WERE ENABLED, THE SEQUENCE BACKSPACE, SPACE,
*  BACKSPACE WOULD BE NECESSARY TO MOVE THE CURSOR LEFT,
*  PRINT A SPACE OVER THE CHARACTER, AND MOVE THE CURSOR LEFT
*
BSSTRG:         FCB     CSRLFT,STERM

**********************************
*ROUTINE: WRSTRG
*PURPOSE: OUTPUT STRING TO CONSOLE
*ENTRY: X = BASE ADDRESS OF STRING
*EXIT:  NONE
*REGISTERS USED: ALL
**********************************

WRSTRG:
        LDA     ,Y+             GET CHARACTER FROM STRING
        CMPA    #STERM          CHECK IF AT END
        BEQ     WREXIT          EXIT IF AT END
        JSR     [OUTPTR]        WRITE CHARACTER
        BRA     WRSTRG          CHECK NEXT CHARACTER
WREXIT:
        RTS

*
*       SAMPLE EXECUTION:
*
*EQUATES
PROMPT  EQU     '?              OPERATOR PROMPT = QUESTION MARK

SC8A:
        *
        *READ LINE FROM TERMINAL
        *
        LDA     #PROMPT         WRITE PROMPT (?)
        JSR     WRCHAR
        LDX     #INBUFF         GET INPUT BUFFER ADDRESS
        LDA     #LINBUF         GET LENGTH OF BUFFER
        JSR     RDLINE          READ LINE
        TSTA                    CHECK LINE LENGTH
```

```
        BEQ     SC8A            READ NEXT LINE IF LENGTH IS 0
        *
        *ECHO LINE TO CONSOLE
        *
        LDX     #INBUFF         POINT TO START OF BUFFER
WRBUFF:
        LDA     ,X              WRITE NEXT CHARACTER
        JSR     WRCHAR
        INX                     INCREMENT BUFFER POINTER
        DECB                    DECREMENT CHARACTER COUNT
        BNE     WRBUFF          CONTINUE UNTIL ALL CHARACTERS SENT
        JSR     WRNEWL          THEN END WITH CR,LF
        BRA     SC8A            READ NEXT LINE


*DATA SECTION
LINBUF  EQU     16              LENGTH OF INPUT BUFFER
INBUFF  RMB     LINBUF          INPUT BUFFER

        END
```

8B Write a line to an output device (WRLINE)

Writes characters until it empties a buffer with given length and base address. Assumes the system-dependent subroutine WRCHAR, which sends the character in register A to an output device.

WRLINE is an example of an output driver. The actual I/O subroutines will, of course, be computer-dependent. A specific example in the listing is for a Radio Shack Color Computer with TRS-80 BASIC in ROM.

Procedure The program exits immediately if the buffer is empty. Otherwise, it sends characters to the output device one at a time until it empties the buffer.

Entry conditions

Base address of buffer in register X
Number of characters in the buffer in register A

Exit conditions

None

Example

Data:	Number of characters = 5
	Buffer contains 'ENTER'
Result:	'ENTER' sent to the output device

Registers used All

Execution time 16 cycles overhead plus 19 cycles per byte. This does not, of course, include the execution time of WRCHAR.

Program size 14 bytes

Data memory required None

Special case

An empty buffer results in an immediate exit with nothing sent to the output device.

```
*       Title           Write Line
*       Name:           WRLINE
*
*       Purpose:        Write characters to output device
*
*       Entry:          Register X = Base address of buffer
*                       Register A = Number of characters in buffer
*
*       Exit:           None
*
*       Registers Used: All
*
*       Time:           Indeterminate, depends on the speed of the
*                       WRCHAR routine.
*
*       Size:           Program         14 bytes
*

WRLINE:
        *
        *EXIT IMMEDIATELY IF BUFFER IS EMPTY
        *
        TSTA                    TEST NUMBER OF CHARACTERS IN BUFFER
        BEQ     EXITWL          BRANCH (EXIT) IF BUFFER EMPTY
                                * X = BASE ADDRESS OF BUFFER
        *
        *LOOP SENDING CHARACTERS TO OUTPUT DEVICE
        *
        TFR     A,B             MOVE CHARACTER COUNT TO B
WRLLP:
        LDA     ,X+             GET NEXT CHARACTER
        JSR     WRCHAR          SEND CHARACTER
        DECB                    DECREMENT COUNTER
        BNE     WRLLP           CONTINUE UNTIL ALL CHARACTERS SENT
EXITWL:
        RTS                     EXIT

*********************************************************
*
* THE FOLLOWING SUBROUTINES ARE TYPICAL EXAMPLES USING THE
* RADIO SHACK TRS-80 COLOR COMPUTER WITH BASIC IN ROM
*
*********************************************************
```

```
****************************************
*ROUTINE: WRCHAR
*PURPOSE: WRITE CHARACTER TO OUTPUT DEVICE
*ENTRY: REGISTER A = CHARACTER
*EXIT:  NONE
*REGISTERS USED: ALL
****************************************
* COLOR COMPUTER EQUATES
CLRSCN  EQU     $A928           STARTING ADDRESS FOR ROUTINE
*                                 THAT CLEARS SCREEN
OUTPTR  EQU     $A002           POINTER TO OUTPUT ROUTINE
*                                 UNIT NUMBER GOES IN LOCATION
*                                   $006F (0 = SCREEN)
*                                 CHARACTER GOES IN REGISTER A

WRCHAR:
*
*       SEND CHARACTER TO OUTPUT DEVICE FROM REGISTER A
*       LOCATION $006F SHOULD CONTAIN A UNIT NUMBER
*       (DEFAULT IS SCREEN = 0)
*
        JSR     [OUTPTR]        SEND CHARACTER
        RTS


*
*       SAMPLE EXECUTION:
*
*CHARACTER EQUATES
CR      EQU     $0D             CARRIAGE RETURN FOR CONSOLE
LF      EQU     $0A             LINE FEED FOR CONSOLE
PROMPT  EQU     '?              OPERATOR PROMPT = QUESTION MARK

SC8B:
        *
        *CALL BASIC SUBROUTINE THAT CLEARS SCREEN
        *
        JSR     CLRSCN          CLEAR SCREEN
        *
        *READ LINE FROM CONSOLE
        *
        LDA     #PROMPT         OUTPUT PROMPT (?)
        JSR     WRCHAR
        LDX     #INBUFF         POINT TO INPUT BUFFER
        JSR     RDLINE          READ LINE FROM CONSOLE
        PSHS    A               SAVE LINE LENGTH IN STACK
        LDA     #2              OUTPUT LINE FEED, CARRIAGE RETURN
        LDX     #CRLF
        JSR     WRCHAR
        *
        *WRITE LINE TO CONSOLE
        *
        PULS    A               RESTORE LINE LENGTH FROM STACK
        LDX     #INBUFF         GET BASE ADDRESS OF BUFFER
        JSR     WRLINE          WRITE LINE
        LDX     #CRLF           OUTPUT CARRIAGE RETURN, LINE FEED
```

```
        LDA     #2              LENGTH OF CRLF STRING
        JSR     WRLINE          WRITE CRLF STRING
        BRA     SC8B            REPEAT CLEAR, READ, WRITE SEQUENCE

*DATA SECTION
CRLF    FCB     CR,LF           CARRIAGE RETURN, LINE FEED
LINBUF  EQU     $10             LENGTH OF INPUT BUFFER
INBUFF: RMB     LINBUF          DATA BUFFER

        END
```

8C Parity checking and generation (CKPRTY, GEPRTY)

Generates and checks parity. GEPRTY generates even parity for a 7-bit character and places it in bit 7. An even parity bit makes the total number of 1 bits in the byte even. CKPRTY sets the Carry flag to 0 if a data byte has even parity and to 1 otherwise. A byte's parity is even if it has an even number of 1 bits and odd otherwise.

Procedures

1. GEPRTY generates even parity by counting the number of 1s in the seven least significant bits of register A. The least significant bit of the count is an even parity bit. The program shifts that bit to the Carry and then to bit 7 of the data.

2. CKPRTY counts the number of 1 bits in the data by repeatedly shifting it left logically and testing the Carry. The program quits when the shifted data becomes zero. The least significant bit of the count is an even parity bit; the program concludes by shifting that bit to the Carry.

Entry conditions

1. GEPRTY
Data in register A

2. CKPRTY
Data in register A

Exit conditions

1. GEPRTY
Data with even parity in bit 7 in register A

2. CKPRTY
Carry = 0 if the data has even parity, 1 if it had odd parity

Examples

1. GEPRTY
   (a) Data: (A) = 42_{16} = 01000010_2 (ASCII B)

Result: (A) = 42_{16} = 01000010_2 (ASCII B with bit 7 cleared)
Even parity is 0, since 01000010_2 has an even number (2) of 1 bits

(b) Data: (A) = 43_{16} = 01000011_2 (ASCII C)
Result: (A) = $C3_{16}$ = 11000011_2 (ASCII C with bit 7 set)
Even parity is 1, since 01000011_2 has an odd number (3) of 1 bits

2. CKPRTY
   (a) Data: (A) = 42_{16} = 01000010_2 (ASCII B)
   Result: Carry = 0, since 01000010_2 has an even number (2) of 1 bits
   (b) Data: (A) = 43_{16} = 01000011_2 (ASCII C)
   Result: Carry = 1, since 01000011_2 has an odd number (3) of 1 bits

---

Registers used

1. GEPRTY: A, B, CC
2. CKPRTY: A, B, CC

Execution time

1. GEPRTY: 95 cycles maximum
2. CKPRTY: 91 cycles maximum

The execution times of both routines depend on how many 1 bits the data contains and how rapidly the logical shifting makes it zero. Both execute faster if many of the less significant bits are zeros.

Program size

1. GEPRTY: 15 bytes
2. CKPRTY: 10 bytes

Data memory required 1 stack byte for GEPRTY

```
*       Title           Generate and Check Parity
*       Name:           GEPRTY, CKPRTY
*
*       Purpose:        GEPRTY generates even parity in bit 7
*                       for a 7-bit character.
```

```
*                       CKPRTY checks the parity of a byte
*
*       Entry:          GEPRTY - data in register A
*                       CKPRTY - data in register A
*
*       Exit:           GEPRTY - data with even parity in bit 7
*                       in register A
*                       CKPRTY - Carry = 0 if parity is even,
*                       Carry = 1 if parity is odd
*
*       Registers Used: GEPRTY - A, B, CC
*                       CKPRTY - A, B, CC
*
*       Time:           GEPRTY - 95 cycles maximum
*                       CKPRTY - 91 cycles maximum
*
*       Size:           Program  25 bytes
*                       Data     1 stack byte
*
*
*

*
*       GENERATE EVEN PARITY
*
GEPRTY:
        CLRB            NUMBER OF 1 BITS = ZERO
        ASLA            DROP DATA BIT 7
        PSHS    A       SAVE SHIFTED DATA IN STACK
        *
        *COUNT 1 BITS UNTIL DATA BECOMES ZERO
        *
CNTBIT:
        BPL     SHIFT   BRANCH IF NEXT BIT (BIT 7) IS 0
        INCB            ELSE INCREMENT NUMBER OF 1 BITS
SHIFT:
        ASLA            SHIFT DATA LEFT
        BNE     CNTBIT  BRANCH IF THERE ARE MORE 1 BITS LEFT
        *
        *MOVE EVEN PARITY TO BIT 7 OF DATA
        *
        LSRB            MOVE EVEN PARITY TO CARRY
                        *NOTE EVEN PARITY IS BIT 0 OF COUNT
        PULS    A       RESTORE SHIFTED DATA FROM STACK
        RORA            ROTATE TO FORM BYTE WITH EVEN PARITY IN BIT 7
        RTS
*
*       CHECK PARITY
*
CKPRTY:
        CLRB            NUMBER OF 1 BITS = ZERO
        TSTA            TEST DATA BYTE
        *
        *COUNT 1 BITS UNTIL DATA BECOMES ZERO
        *
BITCNT:
```

```
        BPL     CSHIFT    BRANCH IF NEXT BIT (BIT 7) IS 0
        INCB              ELSE INCREMENT NUMBER OF 1 BITS
SHIFT:
        ASLA              SHIFT DATA LEFT
        BNE     BITCNT    BRANCH IF THERE ARE MORE 1 BITS LEFT
        *
        *MOVE PARITY TO CARRY
        *
        LSRB              MOVE PARITY TO CARRY
                          *NOTE PARITY IS BIT 0 OF COUNT
        RTS
*
*       SAMPLE EXECUTION:
*

        *
        *GENERATE PARITY FOR VALUES FROM 0..127 AND STORE THEM
        *  IN BUFFER 1
        *
SC8C:
        LDX     #BUFR1    GET BASE ADDRESS OF BUFFER
        CLRA              START DATA AT ZERO
GPARTS:
        PSHS    A         SAVE DATA IN STACK
        JSR     GEPRTY    GENERATE EVEN PARITY
        PULS    B
        STA     B,X       SAVE VALUE WITH EVEN PARITY
        TFR     B,A       RETURN DATA VALUE TO A
        INCA              ADD 1 TO DATA VALUE
        CMPA    #128      HAVE WE REACHED HIGHEST VALUE?
        BNE     GPARTS    BRANCH IF NOT DONE
        *
        *CHECK PARITY FOR ALL BYTES IN BUFFER 1
        *CARRY = 1 IF ROUTINE FINDS A PARITY ERROR AND REGISTER
        *  X POINTS TO THE BYTE WITH THE ERROR
        *CARRY = 0 IF ROUTINE FINDS NO PARITY ERRORS
        *
        LDX     #BUFR1    GET BASE ADDRESS OF BUFFER
        LDA     #129      CHECK 128 BYTES
        PSHS    A         SAVE COUNT ON STACK
CPARTS:
        DEC     ,S        DECREMENT COUNT
        BEQ     CPEXIT    EXIT IF ALL BYTES CHECKED
        LDA     ,X+       GET NEXT DATA BYTE
        JSR     CKPRTY    CHECK PARITY
        BCC     CPARTS    IF NO ERROR, CONTINUE THROUGH VALUES
        LEAX    -1,X      PARITY ERROR - MAKE X POINT TO IT
CPEXIT:
        LEAS    1,S       REMOVE COUNT BYTE FROM STACK
        BRA     SC8C      BRANCH FOR ANOTHER TEST

*
*DATA SECTION
*
BUFR1   RMB     128       BUFFER FOR DATA VALUES WITH EVEN PARITY

        END
```

8D CRC16 checking and generation (ICRC16,CRC16,GCRC16)

Generates a 16-bit cyclic redundancy check (CRC) based on the IBM Binary Synchronous Communications protocol (BSC or Bisync). Uses the polynomial $X^{16} + X^{15} + X^2 + 1$. Entry point ICRC16 initializes the CRC to 0 and the polynomial to its bit pattern. Entry point CRC16 combines the previous CRC with the one generated from the current data byte. Entry point GCRC16 returns the CRC.

Procedure Subroutine ICRC16 initializes the CRC to 0 and the polynomial to a 1 in each bit position corresponding to a power of X present in the formula. Subroutine CRC16 updates the CRC for a data byte. It shifts both the data and the CRC left eight times; after each shift, it exclusive-ORs the CRC with the polynomial if the exclusive-OR of the data bit and the CRC's most significant bit is 1. Subroutine CRC16 leaves the CRC in memory locations CRC (more significant byte) and CRC + 1 (less significant byte). Subroutine GCRC16 loads the CRC into register D.

Entry conditions

1. For ICRC16: none

2. For CRC16: data byte in register A, previous CRC in memory locations CRC (more significant byte) and CRC + 1 (less significant byte), CRC polynomial in memory locations PLY (more significant byte) and PLY + 1 (less significant byte).

3. For GCRC16: CRC in memory locations CRC (more significant byte) and CRC + 1 (less significant byte).

Exit conditions

1. For ICRC16
0 (initial CRC value) in memory locations CRC (more significant byte) and CRC+1 (less significant byte)
CRC polynomial in memory locations PLY (more significant byte) and PLY+1 (less significant byte)

2. For CRC16: CRC with current data byte included in memory loca-

tions CRC (more significant byte) and CRC + 1 (less significant byte)

3. For GCRC16: CRC in register D

Examples

1. Generating a CRC
Call ICRC16 to initialize the polynomial and start the CRC at 0
Call CRC16 repeatedly to update the CRC for each data byte
Call GCRC16 to obtain the final CRC

2. Checking a CRC
Call ICRC16 to initialize the polynomial and start the CRC at 0
Call CRC16 repeatedly to update the CRC for each data byte (including the stored CRC) for checking
Call GCRC16 to obtain the final CRC; it will be 0 if there were no errors

Note that only ICRC16 depends on the particular CRC polynomial used. To change the polynomial, simply change the data ICRC16 loads into memory locations PLY (more significant byte) and PLY + 1 (less significant byte).

Reference

J. E. McNamara, *Technical Aspects of Data Communications*, 3rd ed., Digital Press, Digital Equipment Corp., 12-A Esquire Road, Billerica, MA, 1989. This book contains explanations of CRC and communications protocols.

Registers used

1. By ICRC16: CC, X
2. By CRC16: none
3. By GCRC16: CC, D

Execution time

1. For ICRC16: 23 cycles
2. For CRC16: 490 cycles overhead plus an average of 34 cycles per

data byte, assuming that the previous CRC and the polynomial must be EXCLUSIVE-ORed in half of the iterations

3. For GCRC16: 11 cycles

Program size

1. For ICRC16: 13 bytes
2. For CRC16: 42 bytes
3. For GCRC16: 4 bytes

Data memory required 4 bytes anywhere in RAM for the CRC (2 bytes starting at address CRC) and the polynomial (2 bytes starting at address PLY). CRC16 also requires 7 stack bytes to save and restore the user registers.

```
*       Title           Generate CRC-16
*       Name:           ICRC16, CRC16, GCRC16
*
*
*       Purpose:        Generate a 16 bit CRC based on IBM's Binary
*                       Synchronous Communications protocol. The CRC is
*                       based on the following polynomial:
*                         (^ indicates "to the power")
*                               X^16 + X^15 + X^2 + 1
*
*                       To generate a CRC:
*                           1)  Call ICRC16 to initialize the CRC
*                               polynomial and clear the CRC.
*                           2)  Call CRC16 for each data byte.
*                           3)  Call GCRC16 to obtain the CRC.
*                               It should then be appended to the data,
*                               high byte first.
*
*                       To check a CRC:
*                           1)  Call ICRC16 to initialize the CRC.
*                           2)  Call CRC16 for each data byte and
*                               the 2 bytes of CRC previously generated.
*                           3)  Call GCRC16 to obtain the CRC. It will
*                               be zero if no errors occurred.
*
*       Entry:          ICRC16 - None
*                       CRC16  - Register A = Data byte
*                       GCRC16 - None
*
*       Exit:           ICRC16 - CRC, PLY initialized
*                       CRC16  - CRC updated
*                       GCRC16 - Register D = CRC
```

```
*
*       Registers Used: ICRC16 - CC,X
*                       CRC16  - None
*                       GCRC16 - CC,D
*
*       Time:           ICRC16 - 23 cycles
*                       CRC16 - 490 cycles overhead plus an average of
*                       34 cycles per data byte.  The loop timing
*                       assumes that half the iterations require
*                       EXCLUSIVE-ORing the CRC and the polynomial.
*                       GCRC16 - 11 cycles
*
*       Size:           Program  59 bytes
*                       Data     4 bytes plus 7 stack bytes for CRC16
*


CRC16:
        *
        *SAVE ALL REGISTERS
        *
        PSHS    CC,D,X,Y        SAVE ALL REGISTERS
        *
        *LOOP THROUGH EACH DATA BIT, GENERATING THE CRC
        *
        LDB     #8              8 BITS PER BYTE
        LDX     #PLY            POINT TO POLYNOMIAL
        LDY     #CRC            POINT TO CRC VALUE
CRCLP:
        PSHS    D               SAVE DATA, BIT COUNT
        ANDA    #%10000000      GET BIT 7 OF DATA
        EORA    ,Y              EXCLUSIVE-OR BIT 7 WITH BIT 15 OF CRC
        STA     ,Y
        ASL     1,Y             SHIFT 16-BIT CRC LEFT
        ROL     ,Y
        BCC     CRCLP1          BRANCH IF BIT 7 OF EXCLUSIVE-OR IS 0
        *
        *BIT 7 IS 1, SO EXCLUSIVE-OR CRC WITH POLYNOMIAL
        *
        LDD     ,X              GET POLYNOMIAL
        EORA    ,Y              EXCLUSIVE-OR WITH HIGH BYTE OF CRC
        EORB    1,Y             EXCLUSIVE-OR WITH LOW BYTE OF CRC
        STD     ,Y              SAVE NEW CRC VALUE
        *
        *SHIFT DATA LEFT AND COUNT BITS
        *
CRCLP1:
        PULS    D               GET DATA, BIT COUNT
        ASLA                    SHIFT DATA LEFT
        DECB                    DECREMENT BIT COUNT
        BNE     CRCLP           JUMP IF NOT THROUGH 8 BITS
        *
        *RESTORE REGISTERS AND EXIT
        *
        PULS    CC,D,X,Y        RESTORE ALL REGISTERS
        RTS
```

```
*****************************************
*ROUTINE: ICRC16
*PURPOSE: INITIALIZE CRC AND PLY
*ENTRY: NONE
*EXIT:  CRC AND POLYNOMIAL INITIALIZED
*REGISTERS USED: X
*****************************************

ICRC16:
        LDX     #0              CRC = 0
        STX     CRC
        LDX     #$8005          PLY = 8005H
        STX     PLY
                                *8005 HEX REPRESENTS X^16+X^15+X^2+1
                                *  THERE IS A 1 IS IN EACH BIT
                                *  POSITION FOR WHICH A POWER APPEARS
                                *  IN THE FORMULA (BITS 0, 2, AND 15)
        RTS

*****************************************
*ROUTINE: GCRC16
*PURPOSE: GET CRC VALUE
*ENTRY: NONE
*EXIT:  REGISTER D = CRC VALUE
*REGISTERS USED: D
*****************************************

GCRC16:
        LDD     CRC             D = CRC
        RTS

*DATA
CRC:    RMB     2               CRC VALUE
PLY:    RMB     2               POLYNOMIAL VALUE


*
*       SAMPLE EXECUTION:
*

        *
        *GENERATE CRC FOR THE NUMBER 1 AND CHECK IT
        *
SC8D:
        JSR     ICRC16          INITIALIZE CRC, POLYNOMIAL
        LDA     #1              GENERATE CRC FOR 1
        JSR     CRC16
        JSR     GCRC16
        JSR     ICRC16          INITIALIZE AGAIN
        LDA     #1
        JSR     CRC16           CHECK CRC BY GENERATING IT FOR DATA
        TFR     Y,D              AND STORED CRC ALSO
        JSR     CRC16           HIGH BYTE OF CRC FIRST
        TFR     B,A             THEN LOW BYTE OF CRC
        JSR     CRC16
        JSR     GCRC16          CRC SHOULD BE ZERO IN D
```

```
        *
        *GENERATE CRC FOR THE SEQUENCE 0,1,2,...,255 AND CHECK IT
        *
        JSR     ICRC16          INITIALIZE CRC, POLYNOMIAL
        CLRB                    START DATA BYTES AT 0
GENLP:
        JSR     CRC16           UPDATE CRC
        INCB                    ADD 1 TO PRODUCE NEXT DATA BYTE
        BNE     GENLP           BRANCH IF NOT DONE

        JSR     GCRC16          GET RESULTING CRC
        TFR     D,Y             SAVE CRC IN Y
        *
        *CHECK CRC BY GENERATING IT AGAIN
        *
        JSR     ICRC16          INITIALIZE CRC, POLYNOMIAL
        CLRB                    START DATA BYTES AT 0
CHKLP:
        JSR     CRC16           UPDATE CRC
        INCB                    ADD 1 TO PRODUCE NEXT DATA BYTE
        BNE     CHKLP           BRANCH IF NOT DONE
        *
        *INCLUDE STORED CRC IN CHECK
        *
        TFR     Y,D             GET OLD CRC VALUE
        JSR     CRC16           INCLUDE HIGH BYTE OF CRC
        TFR     B,A             INCLUDE LOW BYTE OF CRC
        JSR     CRC16

        JSR     GCRC16          GET RESULTING CRC
                                *IT SHOULD BE 0
        BRA     SC8D            REPEAT TEST

        END
```

8E I/O device table handler (IOHDLR)

Performs input and output in a device-independent manner using I/O control blocks and an I/O device table. The I/O device table is a linked list; each entry contains a link to the next entry, the device number, and starting addresses for routines that initialize the device, determine its input status, read data from it, determine its output status, and write data to it. An I/O control block is an array containing device number, operation number, device status, and the base address and length of the device's buffer. The user must provide IOHDLR with the base address of an I/O control block and the data if only one byte is to be written. IOHDLR returns the status byte and the data (if only one byte is read).

This subroutine is an example of handling input and output in a device-independent manner. The I/O device table must be constructed using subroutines INITDL, which initializes the device list to empty, and INSDL, which inserts a device into the list.

An applications program will perform input or output by obtaining or constructing an I/O control block and then calling IOHDLR. IOHDLR uses the I/O device table to determine how to transfer control to the I/O driver.

Procedure The program first initializes the status byte to 0, indicating no errors. It then searches the device table, trying to match the device number in the I/O control block. If it does not find a match, it exits with an error number in the status byte. If it finds a match, it checks for a valid operation and transfers control to the appropriate routine from the device table entry. That routine must then transfer control back to the original caller. If the operation is invalid (the operation number is too large or the starting address for the routine is 0), the program returns with an error number in the status byte.

Subroutine INITDL initializes the device list, setting the initial link to 0.

Subroutine INSDL inserts an entry into the device list, making its address the head of the list and setting its link field to the previous head of the list.

Entry conditions

1. For IOHDLR

Base address of input/output control block in register X

Data byte (if the operation is to write 1 byte) in register A

2. For INITL: none

3. For INSDL: base address of a device table entry in register X

Exit conditions

1. For IOHDLR
I/O control block status byte in register A if an error is found; otherwise, the routine exits to the appropriate I/O driver
Data byte in register A if the operation is to read 1 byte

2. For INITL: device list header (addresses DVLST and DVLST+1) cleared to indicate empty list

3. For INSDL: device table entry added to list

Example

The example in the listing uses the following structure:

Input/output operations

Operation number	Operation
0	Initialize device
1	Determine input status
2	Read 1 byte from input device
3	Read *N* bytes (usually 1 line) from input device
4	Determine output status
5	Write 1 byte to output device
6	Write *N* bytes (usually 1 line) to output device

Input/output control block

Index	Contents
0	Device number
1	Operation number
2	Status
3	More significant byte of base address of buffer
4	Less significant byte of base address of buffer

5	More significant byte of buffer length
6	Less significant byte of buffer length

Device table entry

Index	Contents
0	More significant byte of link field (base address of next element)
1	Less significant byte of link field (base address of next element)
2	Device number
3	More significant byte of starting address of device initialization routine
4	Less significant byte of starting address of device initialization routine
5	More significant byte of starting address of input status determination routine
6	Less significant byte of starting address of input status determination routine
7	More significant byte of starting address of input driver (read 1 byte only)
8	Less significant byte of starting address of input driver (read 1 byte only)
9	More significant byte of starting address of input driver (read *N* bytes or 1 line)
10	Less significant byte of starting address of input driver (read *N* bytes or 1 line)
11	More significant byte of starting address of output status determination routine
12	Less significant byte of starting address of output status determination routine
13	More significant byte of starting address of output driver (write 1 byte only)
14	Less significant byte of starting address of output driver (write 1 byte only)
15	More significant byte of starting address of output driver (write *N* bytes or 1 line)
16	Less significant byte of starting address of output driver (write *N* bytes or 1 line)

If an operation is irrelevant or undefined (such as output status determination for a keyboard or input driver for a printer), the corresponding starting address in the device table is 0.

Status values

Value	Description
0	No errors
1	Bad device number (no such device)
2	Bad operation number (no such operation or invalid operation)
3	Input data available or output device ready

Registers used

1. By IOHDLR: All
2. By INITDL: CC, X
3. By INSDL: CC, U, X

Execution time

1. For IOHDLR: 75 cycles overhead plus 23 cycles for each unsuccessful match of a device number
2. For INITDL: 14 cycles
3. For INSDL: 22 cycles

Program size

1. For IOHDLR: 62 bytes
2. For INITL: 7 bytes
3. For INSDL: 9 bytes

Data memory required 5 bytes anywhere in RAM for the base address of the I/O control block (2 bytes starting at address IOCBA), the device list header (2 bytes starting at address DVLST), and temporary storage for data to be written without a buffer (1 byte at address BDATA).

```
*       Title           I/O Device Table Handler
*       Name:           IOHDLR
*
*       Purpose:        Perform I/O in a device independent manner.
*                       This can be done by accessing all devices
*                       in the same way using an I/O Control Block
*                       (IOCB) and a device table. The routines here
*                       allow the following operations:
*
*                       Operation number  Description
*                           0             Initialize Device
*                           1             Determine input status
*                           2             Read 1 byte
*                           3             Read N bytes
*                           4             Determine output status
*                           5             Write 1 byte
*                           6             Write N bytes
*
*                       Adding operations such as Open, Close, Delete,
*                       Rename, and Append would allow for more complex
*                       devices such as floppy or hard disks.
*
*                       A IOCB is an array consisting of elements
*                       with the following form:
*
*                       IOCB + 0 = Device Number
*                       IOCB + 1 = Operation Number
*                       IOCB + 2 = Status
*                       IOCB + 3 = High byte of buffer address
*                       IOCB + 4 = Low byte of buffer address
*                       IOCB + 5 = High byte of buffer length
*                       IOCB + 6 = Low byte of buffer length
*
*                       The device table is implemented as a linked
*                       list. Two routines maintain the list: INITDL,
*                       which initializes it to empty, and INSDL,
*                       which inserts a device into it.
*                       A device table entry has the following form:
*
*                       DVTBL + 0 = High byte of link field
*                       DVTBL + 1 = Low byte of link field
*                       DVTBL + 2 = Device Number
*                       DVTBL + 3 = High byte of device initialization
*                       DVTBL + 4 = Low byte of device initialization
*                       DVTBL + 5 = High byte of input status routine
*                       DVTBL + 6 = Low byte of input status routine
*                       DVTBL + 7 = High byte of input 1 byte routine
*                       DVTBL + 8 = Low byte of input 1 byte routine
*                       DVTBL + 9 = High byte of input N bytes routine
*                       DVTBL + 10= Low byte of input N bytes routine
*                       DVTBL + 11= High byte of output status routine
*                       DVTBL + 12= Low byte of output status routine
*                       DVTBL + 13= High byte of output 1 byte routine
*                       DVTBL + 14= Low byte of output 1 byte routine
*                       DVTBL + 15= High byte of output N bytes routine
```

```
*                       DVTBL + 16= Low byte of output N bytes routine
*
*       Entry:          Register X = Base address of IOCB
*                       Register A = For write 1 byte, contains the
*                                    data (no buffer is used).
*
*       Exit:           Register A = Copy of the IOCB status byte
*                                    Except contains the data for
*                                    read 1 byte (no buffer is used).
*                       Status byte of IOCB is 0 if the operation was
*                       completed successfully; otherwise, it contains
*                       the error number.
*
*                       Status value    Description
*                           0           No errors
*                           1           Bad device number
*                           2           Bad operation number
*                           3           Input data available or output
*                                       device ready
*
*       Registers Used: All
*
*       Time:           75 cycles overhead plus 23 cycles for each
*                       device in the list which is not the one
*                       requested
*
*       Size:           Program  78 bytes
*                       Data     5 bytes
*

*IOCB AND DEVICE TABLE EQUATES
IOCBDN  EQU     0       IOCB DEVICE NUMBER
IOCBOP  EQU     1       IOCB OPERATION NUMBER
IOCBST  EQU     2       IOCB STATUS
IOCBBA  EQU     3       IOCB BUFFER BASE ADDRESS
IOCBBL  EQU     5       IOCB BUFFER LENGTH
DTLNK   EQU     0       DEVICE TABLE LINK FIELD
DTDN    EQU     2       DEVICE TABLE DEVICE NUMBER
DTSR    EQU     3       BEGINNING OF DEVICE TABLE SUBROUTINES
*OPERATION NUMBERS
NUMOP   EQU     7       NUMBER OF OPERATIONS
INIT    EQU     0       INITIALIZATION
ISTAT   EQU     1       INPUT STATUS
R1BYTE  EQU     2       READ 1 BYTE
RNBYTE  EQU     3       READ N BYTES
OSTAT   EQU     4       OUTPUT STATUS
W1BYTE  EQU     5       WRITE 1 BYTE
WNBYTE  EQU     6       WRITE N BYTES
*STATUS VALUES
NOERR   EQU     0       NO ERRORS
DEVERR  EQU     1       BAD DEVICE NUMBER
OPERR   EQU     2       BAD OPERATION NUMBER
DEVRDY  EQU     3       INPUT DATA AVAILABLE OR OUTPUT DEVICE READY
IOHDLR:
        *
        *SAVE IOCB ADDRESS AND DATA (IF ANY)
```

```
        *
        STX     IOCBA           SAVE IOCB ADDRESS
        STA     BDATA           SAVE DATA BYTE FOR WRITE 1 BYTE
        *
        *INITIALIZE STATUS BYTE TO INDICATE NO ERRORS
        *
        LDA     #NOERR          STATUS = NO ERRORS
        STA     IOCBST,X        SAVE STATUS IN IOCB
        *
        *CHECK FOR VALID OPERATION NUMBER (WITHIN LIMIT)
        *
        LDB     IOCBOP,X        GET OPERATION NUMBER FROM IOCB
        CMPB    #NUMOP          IS OPERATION NUMBER WITHIN LIMIT?
        BCC     BADOP           JUMP IF OPERATION NUMBER TOO LARGE
        *
        *SEARCH DEVICE LIST FOR THIS DEVICE
        *
        LDA     IOCBDN,X        GET IOCB DEVICE NUMBER
        LDX     DVLST           GET FIRST ENTRY IN DEVICE LIST
        *
        *X = POINTER TO DEVICE LIST
        *B = OPERATION NUMBER
        *A = REQUESTED DEVICE NUMBER
        *
SRCHLP:
        *CHECK IF AT END OF DEVICE LIST (LINK FIELD = 0000)
        CMPX    #0              TEST LINK FIELD
        BEQ     BADDN           BRANCH IF NO MORE DEVICE ENTRIES
        *
        *CHECK IF CURRENT ENTRY IS DEVICE IN IOCB
        *
        CMPA    DTDN,X          COMPARE DEVICE NUMBER, REQUESTED DEVICE
        BEQ     FOUND           BRANCH IF DEVICE FOUND
        *
        *DEVICE NOT FOUND, SO ADVANCE TO NEXT DEVICE
        * TABLE ENTRY THROUGH LINK FIELD
        * MAKE CURRENT DEVICE = LINK
        *
        LDX     ,X              CURRENT ENTRY = LINK
        BRA     SRCHLP          CHECK NEXT ENTRY IN DEVICE TABLE
        *
        *FOUND DEVICE,  SO VECTOR TO APPROPRIATE ROUTINE IF ANY
        *B = OPERATION NUMBER IN IOCB
        *
FOUND:
        *GET ROUTINE ADDRESS (ZERO INDICATES INVALID OPERATION)
        ASLB                    MULTIPLY OPERATION NUMBER TIMES 2 TO
                                * INDEX INTO TABLE OF 16-BIT ADDRESSES
        ADDB    #DTSR           ADD OFFSET TO START OF SUBROUTINE
                                * ADDRESSES
        LDX     B,X             GET SUBROUTINE ADDRESS
        BEQ     BADOP           JUMP IF OPERATION INVALID (ADDRESS = 0)
        PSHS    X               SAVE SUBROUTINE ADDRESS ON STACK
        LDA     BDATA           A = DATA BYTE FOR WRITE 1 BYTE
        LDX     IOCBA           GET BASE ADDRESS OF IOCB
        RTS                     GOTO SUBROUTINE
```

```
BADDN:
        LDA     #DEVERR                 ERROR CODE -- NO SUCH DEVICE
        BRA     EREXIT
BADOP:
        LDA     #OPERR          ERROR CODE -- NO SUCH OPERATION

EREXIT:
        LDX     IOCBA           POINT TO IOCB
        STA     IOCBST,X        SET STATUS BYTE IN IOCB
        RTS

****************************************
*ROUTINE: INITDL
*PURPOSE: INITIALIZE DEVICE LIST TO EMPTY
*ENTRY: NONE
*EXIT:  DEVICE LIST SET TO NO ITEMS
*REGISTERS USED: X
****************************************

INITDL:
        *INITIALIZE DEVICE LIST HEADER TO 0 TO INDICATE NO DEVICES
        LDX     #0              HEADER = 0 (EMPTY LIST)
        STX     DVLST
        RTS


*********************************************
*ROUTINE: INSDL
*PURPOSE: INSERT DEVICE INTO DEVICE LIST
*ENTRY: REGISTER X = ADDRESS OF DEVICE TABLE ENTRY
*EXIT:  DEVICE INSERTED INTO DEVICE LIST
*REGISTERS USED: U,X
*********************************************

INSDL:
        LDU     DVLST           GET CURRENT HEAD OF DEVICE LIST
        STU     ,X              STORE CURRENT HEAD OF DEVICE LIST
        STX     DVLST           MAKE DVLST POINT TO NEW DEVICE
        RTS

*
*DATA SECTION
IOCBA:  RMB     2               BASE ADDRESS OF IOCB
DVLST:  RMB     2               DEVICE LIST HEADER
BDATA:  RMB     1               DATA BYTE FOR WRITE 1 BYTE



*
*       SAMPLE EXECUTION:
*
*CHARACTER EQUATES
CR      EQU     $0D             CARRIAGE RETURN CHARACTER
LF      EQU     $0A             LINE FEED CHARACTER

SC8E:
```

```
        *INITIALIZE DEVICE LIST
        JSR     INITDL          CREATE EMPTY DEVICE LIST

        *SET UP CONSOLE AS DEVICE 1 AND INITIALIZE IT
        LDX     #CONDV          POINT TO CONSOLE DEVICE ENTRY
        JSR     INSDL           ADD CONSOLE TO DEVICE LIST
        LDA     #INIT           INITIALIZE OPERATION
        STA     IOCBOP,X
        LDA     #1              DEVICE NUMBER = 1
        STA     IOCBDN,X
        LDX     #IOCB           INITIALIZE CONSOLE
        JSR     IOHDLR

        *SET UP PRINTER AS DEVICE 2 AND INITIALIZE IT
        LDX     #PRTDV          POINT TO PRINTER DEVICE ENTRY
        JSR     INSDL           ADD PRINTER TO DEVICE LIST
        LDA     #INIT           INITIALIZE OPERATION
        STA     IOCBOP,X
        LDA     #2              DEVICE NUMBER = 2
        STA     IOCBDN,X
        LDX     #IOCB           INITIALIZE PRINTER
        JSR     IOHDLR

        *
        *LOOP READING LINES FROM CONSOLE, AND ECHOING THEM TO
        * THE CONSOLE AND PRINTER UNTIL A BLANK LINE IS ENTERED
        *
TSTLP:
        LDX     #IOCB           POINT TO IOCB
        LDY     #BUFFER         POINT TO BUFFER
        STY     IOCBBA,X        SAVE BUFFER ADDRESS IN IOCB
        LDA     #1              DEVICE NUMBER = 1 (CONSOLE)
        STA     IOCBDN,X
        LDA     #RNBYTE         OPERATION IS READ N BYTES
        STA     IOCBOP,X
        LDY     #LENBUF
        STY     IOCBBL,X        SET BUFFER LENGTH TO LENBUF
        JSR     IOHDLR          READ LINE FROM CONSOLE
        *
        *STOP IF LINE LENGTH IS 0
        *
        LDX     #IOCB           POINT TO IOCB
        LDY     IOCBBL,X        GET LINE LENGTH
        BEQ     SC8END          BRANCH (EXIT) IF LINE LENGTH IS 0
        *
        *SEND CARRIAGE RETURN TO CONSOLE
        *
        LDA     #W1BYTE         OPERATION IS WRITE 1 BYTE
        STA     IOCBOP,X        SAVE IN IOCB
        LDA     #CR             CHARACTER IS CARRIAGE RETURN
        JSR     IOHDLR          WRITE 1 BYTE (LINE FEED)

        *
        *ECHO LINE TO CONSOLE
        *
        LDX     #IOCB           POINT TO IOCB
```

```
        LDA     #WNBYTE         OPERATION = WRITE N BYTES
        STA     IOCBOP,X        SAVE OPERATION NUMBER IN IOCB
        LDA     #1              DEVICE NUMBER = CONSOLE
        STA     IOCBDN,X        SAVE DEVICE NUMBER IN IOCB
        JSR     IOHDLR          WRITE N BYTES ON CONSOLE
        *
        *ECHO LINE TO PRINTER
        *
        LDX     #IOCB           POINT TO IOCB
        LDA     #WNBYTE         OPERATION = WRITE N BYTES
        STA     IOCBOP,X        SAVE OPERATION NUMBER IN IOCB
        LDA     #1              DEVICE NUMBER = PRINTER
        STA     IOCBDN,X        SAVE DEVICE NUMBER IN IOCB
        JSR     IOHDLR          WRITE N BYTES ON PRINTER
        *
        *SEND LINE FEED TO PRINTER
        *
        LDX     #IOCB           POINT TO IOCB
        LDA     #W1BYTE         OPERATION = WRITE 1 BYTE
        STA     IOCBOP,X        SAVE OPERATION NUMBER IN IOCB
        LDA     #LF             CHARACTER IS LINE FEED
        JSR     IOHDLR          SEND LINE FEED TO PRINTER

        BRA     TSTLP           LOOP TO READ NEXT LINE
SC8END:
        BRA     SC8E            REPEAT TEST

*
*       DATA SECTION
*
LENBUF  EQU     127             I/O BUFFER LENGTH
BUFFER  RMB     LENBUF          I/O BUFFER

*IOCB FOR PERFORMING IO
IOCB:   RMB     1               DEVICE NUMBER
        RMB     1               OPERATION NUMBER
        RMB     1               STATUS
        FDB     BUFFER          BUFFER ADDRESS
        RMB     2               BUFFER LENGTH

*DEVICE TABLE ENTRIES
CONDV:  FDB     0               LINK FIELD
        FCB     1               DEVICE 1
        FDB     CINIT           CONSOLE INITIALIZE
        FDB     0               NO CONSOLE INPUT STATUS
        FDB     0               NO CONSOLE INPUT 1 BYTE
        FDB     CINN            CONSOLE INPUT N BYTES
        FDB     0               NO CONSOLE OUTPUT STATUS
        FDB     COUT            CONSOLE OUTPUT 1 BYTE
        FDB     COUTN           CONSOLE OUTPUT N BYTES

PRTDV:  FDB     0               LINK FIELD
        FCB     2               DEVICE 2
        FDB     PINIT           PRINTER INITIALIZE
        FDB     0               NO PRINTER INPUT STATUS
        FDB     0               NO PRINTER INPUT 1 BYTE
```

```
        FDB     0               NO PRINTER INPUT N BYTES
        FDB     0               NO PRINTER OUTPUT STATUS
        FDB     OUT             PRINTER OUTPUT 1 BYTE
        FDB     POUTN           PRINTER OUTPUT N BYTES

*
*       RADIO SHACK TRS-80 COLOR COMPUTER EQUATES
*
BDRATE  EQU     $0096           MEMORY LOCATION CONTAINING OUTPUT
*                                 BAUD RATE
B2400   EQU     18              VALUE CORRESPONDING TO 2400 BAUD
CLRSCN  EQU     $A928           STARTING ADDRESS FOR ROUTINE
*                                 THAT CLEARS SCREEN
KBDPTR  EQU     $A000           POINTER TO KEYBOARD INPUT ROUTINE
*                                 (CHARACTER ENDS UP IN REGIRSTER A)
*                                 ZERO FLAG = 1 IF NO CHARACTER,
*                                   0 IF CHARACTER
OUTPTR  EQU     $A002           POINTER TO OUTPUT ROUTINE
*                                 UNIT NUMBER GOES IN LOCATION
*                                   $006F (0 = SCREEN)
*                                 CHARACTER GOES IN REGISTER A
PRDVNO  EQU     $FE             PRINTER DEVICE NUMBER
UNITNO  EQU     $006F           MEMORY LOCATION CONTAINING UNIT
*                                 NUMBER FOR OUTPUT ROUTINE
*                                 (0 = SCREEN)

************************************
*CONSOLE I/O ROUTINES
************************************

*CONSOLE INITIALIZE
CINIT:
        JSR     CLRSCN          CLEAR SCREEN
        RTS                     RETURN

*CONSOLE READ 1 BYTE
CINN:
        LDU     IOCBBL,X        GET BUFFER LENGTH
        PSHS    U               SAVE BUFFER LENGTH IN STACK
        LDU     IOCBBA,X        POINT TO DATA BUFFER
        LDY     #0              INITIALIZE BYTE COUNTER TO 0
        *
        *LOOP READING BYTES UNTIL DATA BUFFER IS FULL
        *
CIN:
        JSR     [KBDPTR]        POLL KEYBOARD
        BEQ     CIN             LOOP UNTIL A KEY IS READ
        CMPA    #CR             CHECK FOR CARRIAGE RETURN
        BEQ     CREXIT          BRANCH (EXIT) IF CARRIAGE RETURN
        STA     ,U+             SAVE BYTE IN DATA BUFFER
        LEAY    1,Y             INCREMENT BYTE COUNT
        CMPY    ,S              CHECK IF BUFFER FULL
        BNE     CIN             BRANCH (LOOP) IF BUFFER NOT FULL
        *
        *CLEAN STACK AND EXIT
        *
```

```
CREXIT:
        STY     IOCBBL,X        SAVE NUMBER OF BYTES READ
        LEAS    2,S             CLEAN STACK
        RTS                     EXIT


*CONSOLE WRITE 1 BYTE
COUT:
        CLR     UNITNO          SET UNIT NUMBER FOR CONSOLE (0)
        JSR     [OUTPTR]        WRITE BYTE
        RTS


*CONSOLE WRITE N BYTES
COUTN:
        CLR     UNITNO          SET UNIT NUMBER FOR CONSOLE (0)
OUTPUT:
        LDY     IOCBBL,X        GET NUMBER OF BYTES TO WRITE
        LDX     IOCBBA,X        POINT TO DATA BUFFER
CWLOOP:
        LDA     ,X+             GET NEXT DATA BYTE
        JSR     [OUTPTR]        WRITE BYTE
        LEAY    -1,Y            DECREMENT BYTE COUNT
        BNE     CWLOOP          CONTINUE THROUGH N BYTES
        RTS                     RETURN

**************************************
*PRINTER ROUTINES
**************************************

*PRINTER INITIALIZE
PINIT:
        LDB     #B2400          SET PRINTER TO 2400 BAUD
        STB     BDRATE          SAVE BAUD RATE
        RTS

*PRINTER OUTPUT 1 BYTE
POUT:
        LDB     #PRDVNO         GET PRINTER DEVICE NUMBER
        STB     UNITNO          SAVE AS UNIT NUMBER
        JSR     [OUTPTR]        WRITE 1 BYTE
        CLR     UNITNO          RESTORE UNIT NUMBER TO CONSOLE (0)
        RTS

*PRINTER OUTPUT N BYTES
POUTN:
        LDB     #PRDVNO         GET PRINTER DEVICE NUMBER
        STB     UNITNO          SAVE AS UNIT NUMBER
        JSR     OUTPUT          WRITE LINE
        CLR     UNITNO          RESTORE UNIT NUMBER TO CONSOLE (0)
        RTS

        END
```

8F Initialize I/O ports (IPORTS)

Initializes a set of I/O ports from an array of port device addresses and data values. Examples are given of initializing the common 6809 programmable I/O devices: 6820 or 6821 Peripheral Interface Adapter (PIA), 6840 Programmable Timer Module (PTM), and 6850 Asynchronous Communications Interface Adapter (ACIA).

This subroutine provides a generalized method for initializing I/O sections. The initialization may involve data ports, data direction registers that determine whether bits are inputs or outputs, control or command registers that determine the operating modes of programmable devices, counters (in timers), priority registers, and other external registers or storage locations.

Tasks the user may perform with this routine include:

1. Assign bidirectional I/O lines as inputs or outputs.
2. Put initial values in output ports.
3. Enable or disable interrupts from peripheral chips.
4. Determine operating modes, such as whether inputs are latched, whether strobes are produced, how priorities are assigned, whether timers operate continuously or only on demand, etc.
5. Load starting values into timers and counters.
6. Select bit rates for communications.
7. Clear or reset devices that are not tied to the overall system reset line.
8. Initialize priority registers or assign initial priorities to interrupts or other operations.
9. Initialize vectors used in servicing interrupts, DMA requests, and other inputs.

Procedure The program loops through the specified number of ports, obtaining each port's memory address and initial value from the array and storing the value in the address. This approach does not depend on the number or type of devices in the I/O section. The user may add or delete devices or change the initialization by changing the array rather than the program.

Each array entry consists of the following:

1. More significant byte of port's memory address.
2. Less significant byte of port's memory address.
3. Initial value to be sent to port.

Entry conditions

Base address of array of port addresses and initial values in register X
Number of entries in array (number of ports to initialize) in register A

Exit conditions

All data values sent to port addresses

Example

Data: Number of ports to initialize = 3
Array elements are:
More significant byte of port 1's memory address
Less significant byte of port 1's memory address
Initial value for port 1
More significant byte of port 2's memory address
Less significant byte of port 2's memory address
Initial value for port 2
More significant byte of port 3's memory address
Less significant byte of port 3's memory address
Initial value for port 3

Result: Initial value for port 1 stored in port 1 address
Initial value for port 2 stored in port 2 address
Initial value for port 3 stored in port 3 address

Note that each element consists of 3 bytes containing:

More significant byte of port's memory address
Less significant byte of port's memory address
Initial value for port

Registers used A, B, CC, U, X

Execution time 10 cycles overhead plus 23 × *N* cycles for each port entry. If, for example, NUMBER OF PORT ENTRIES = 10, execution time is

$10 + 10 \times 23 = 10 + 230 = 240$ cycles

Program size 13 bytes plus the size of the table (3 bytes per port)

Data memory required None

```
*       Title           Initialize I/O Ports
*       Name:           IPORTS
*
*       Purpose:        Initialize I/O ports from an array of port
*                       addresses and values.
*
*       Entry:          Register X = Base address of array
*
*                       The array consists of 3 byte elements
*                          array+0 = High byte of port 1 address
*                          array+1 = Low byte of port 1 address
*                          array+2 = Value to store in port 1 address
*                          array+3 = High byte of port 2 address
*                          array+4 = Low byte of port 2 address
*                          array+5 = Value to store in port 2 address
*                                  .
*                                  .
*                                  .
*
*       Exit:           None
*
*       Registers Used: A,B,CC,U,X
*
*       Time:           10 cycles overhead plus 23 * N cycles for
*                       each port, where N is the number of bytes.
*

*       Size:           Program 13 bytes
*


IPORTS:
        *
        *EXIT IMMEDIATELY IF NUMBER OF PORTS IS ZERO
        *
        TSTA                    TEST NUMBER OF PORTS
        BEQ     EXITIP          BRANCH IF NO PORTS TO INITIALIZE
        *
        *LOOP INITIALIZING PORTS
        *
INITPT:
        LDU     ,X++            GET NEXT PORT ADDRESS
```

```
        LDB     ,X+             GET VALUE TO SEND THERE
        STB     ,U              SEND VALUE TO PORT ADDRESS
        DECA                    COUNT PORTS
        BNE     INITPT          CONTINUE UNTIL ALL PORTS INITIALIZED
        *
        *EXIT
        *
EXITIP:
        RTS

*
*       SAMPLE EXECUTION:
*

*
*INITIALIZE
*  6820/6821 PIA (PERIPHERAL INTERFACE ADAPTER)
*  6850 ACIA (ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER)
*  6840 PTM (PROGRAMMABLE TIMER MODULE)
*
*ARBITRARY DEVICE MEMORY ADDRESSES
*
*  6820/6821 PIA ADDRESSES
*
PIADRA  EQU     $A400           6821 PIA DATA REGISTER A
PIACRA  EQU     $A401           6821 PIA CONTROL REGISTER A
PIADRB  EQU     $A402           6821 PIA DATA REGISTER B
PIACRB  EQU     $A403           6821 PIA CONTROL REGISTER B
*
*  6840 PTM ADDRESSES
*
PTMC13  EQU     $A100           6840 PTM CONTROL REGISTERS 1,3
PTMCR2  EQU     $A101           6840 PTM CONTROL REGISTER 2
PTM1MS  EQU     $A102           6840 PTM TIMER 1 MSB
PTM1LS  EQU     $A103           6840 PTM TIMER 1 LSB
PTM2MS  EQU     $A104           6840 PTM TIMER 2 MSB
PTM2LS  EQU     $A105           6840 PTM TIMER 2 LSB
PTM3MS  EQU     $A106           6840 PTM TIMER 3 MSB
PTM3LS  EQU     $A107           6840 PTM TIMER 3 LSB
*
*  6850 ACIA ADDRESSES
*
ACIADR  EQU     $A200           6850 ACIA DATA REGISTER
ACIACR  EQU     $A201           6850 ACIA CONTROL REGISTER
ACIASR  EQU     $A201           6850 ACIA STATUS REGISTER

SC8F:
        LDX     BEGPIN          GET BASE ADDRESS OF INITIALIZATION
                                * ARRAY
        LDA     SZINIT          GET SIZE OF ARRAY IN BYTES
        JSR     IPORTS          INITIALIZE PORTS
        BRA     SC8F            REPEAT TEST

PINIT:
*
*INITIALIZE 6820 OR 6821 PERIPHERAL INTERFACE ADAPTER (PIA)
```

```
*
*
        *
        *  PORT A = INPUT
        *  CA1 = DATA AVAILABLE, SET ON LOW TO HIGH TRANSITION,
        *    NO INTERRUPTS
        *  CA2 = DATA ACKNOWLEDGE HANDSHAKE
        *
        FDB    PIACRA              PIA CONTROL REGISTER A ADDRESS
        FCB    %00000000           INDICATE NEXT ACCESS TO DATA
                                   * DIRECTION REGISTER (SAME ADDRESS
                                   * AS DATA REGISTER)
        FDB    PIADRA              PIA DATA DIRECTION REGISTER A ADDRESS
        FCB    %00000000           ALL BITS INPUT
        FDB    PIACRA              PIA CONTROL REGISTER A ADDRESS
        FCB    %00100110           * BITS 7,6 NOT USED
                                   * BIT 5 = 1 TO MAKE CA2 OUTPUT
                                   * BIT 4 = 0 TO MAKE CA2 A PULSE
                                   * BIT 3 = 0 TO MAKE CA2 INDICATE
                                   *   DATA REGISTER FULL
                                   * BIT 2 = 1 TO ADDRESS DATA REGISTER
                                   * BIT 1 = 1 TO MAKE CA1 ACTIVE
                                   *   LOW-TO-HIGH
                                   * BIT 0 = 0 TO DISABLE CA1 INTERRUPTS
        *
        *  PORT B = OUTPUT
        *  CB1 = DATA ACKNOWLEDGE, SET ON HIGH TO LOW TRANSITION,
        *    NO INTERRUPTS
        *  CB2 = DATA AVAILABLE, CLEARED BY WRITING TO DATA
        *    REGISTER B, SET TO 1 BY HIGH TO LOW TRANSITION ON CB1
        *
        FDB    PIACRB              PIA CONTROL REGISTER B ADDRESS
        FCB    %00000000           INDICATE NEXT ACCESS TO DATA
                                   * DIRECTION REGISTER (SAME ADDRESS
                                   * AS DATA REGISTER
        FDB    PIADRB              PIA DATA DIRECTION REGISTER B ADDRESS
        FCB    %11111111           ALL BITS OUTPUT
        FDB    PIACRB              PIA CONTROL REGISTER B ADDRESS
        FCB    %00100100           * BITS 7,6 NOT USED
                                   * BIT 5 = 1 TO MAKE CB2 OUTPUT
                                   * BIT 4 = 0 TO MAKE CB2 A PULSE
                                   * BIT 3 = 0 TO MAKE CB2 INDICATE
                                   *   DATA REGISTER FULL
                                   * BIT 2 = 1 TO ADDRESS DATA REGISTER
                                   * BIT 1 = 0 TO MAKE CB2 ACTIVE
                                   *   HIGH-TO-LOW
                                   * BIT 0 = 0 TO DISABLE CB1 INTERRUPTS
*
*INITIALIZE 6850 ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER
*   (ACIA OR UART)
*
        *  8 BIT DATA, NO PARITY
        *  1 STOP BIT
        *  DIVIDE MASTER CLOCK BY 16
        *  NO INTERRUPTS
        *
```

```
        FDB     ACIACR              ACIA CONTROL REGISTER ADDRESS
        FCB     %00000011           PERFORM MASTER RESET
                                    * 6850 HAS NO RESET INPUT
        FDB     ACIACR              ACIA CONTROL REGISTER ADDRESS
        FCB     %00010101           * BIT 7 = 0 TO DISABLE
                                    *   RECEIVE INTERRUPTS
                                    * BIT 6 = 0 TO MAKE RTS LOW
                                    * BIT 5 = 0 TO DISABLE
                                    *   TRANSMIT INTERRUPTS
                                    * BIT 4 = 1 TO SELECT 8-BIT DATA
                                    * BIT 3 = 0 FOR NO PARITY
                                    * BIT 2 = 1 FOR 1 STOP BIT
                                    * BIT 1 = 0, BIT 0 = 1 TO
                                    *   DIVIDE MASTER CLOCK BY 16
*
*INITIALIZE 6840 PROGRAMMABLE TIMER MODULE (PTM)
*
*         CLEAR ALL TIMER COUNTERS
*         RESET TIMERS
*         OPERATE TIMER 2 IN CONTINUOUS MODE, DECREMENTING COUNTER
*           AFTER EACH CLOCK CYCLE
*         SET TIME CONSTANT TO 12 CLOCK CYCLES
*         THIS GENERATES A SQUARE WAVE WITH PERIOD 2 * (12 + 1)
*           = 26 CYCLES
*
*         THIS INITIALIZATION PRODUCES A 2400 HZ CLOCK FOR USE
*           IN DIVIDE BY 16 DATA TRANSMISSION
*         IT ASSUMES A 1 MHZ SYSTEM CLOCK, SO A PERIOD OF
*           (1,000,000)/(16*2400) = 26 CYCLES WILL GENERATE
*           A 38,400 (16*2400) HZ SQUARE WAVE
*
        FDB     PTM1MS              PTM TIMER 1 MS BYTE
        FCB     0                   CLEAR TIMER 1 MS BYTE
        FDB     PTM1LS              PTM TIMER 1 LS BYTE
        FCB     0                   CLEAR TIMER 1 LS BYTE
        FDB     PTM2MS              PTM TIMER 2 MS BYTE
        FCB     0                   CLEAR TIMER 2 MS BYTE
        FDB     PTM2LS              PTM TIMER 2 LS BYTE
        FCB     0                   CLEAR TIMER 2 LS BYTE
        FDB     PTM3MS              PTM TIMER 3 MS BYTE
        FCB     0                   CLEAR TIMER 3 MS BYTE
        FDB     PTM3LS              PTM TIMER 3 LS BYTE
        FCB     0                   CLEAR TIMER 3 LS BYTE
        FDB     PTMCR2              PTM TIMER 2 CONTROL REGISTER
        FCB     %00000001           ADDRESS TIMER 1 CONTROL REGISTER
        FDB     PTMC13              PTM TIMER 1,3 CONTROL REGISTER
        FCB     %00000001           RESET TIMERS
        FDB     PTMC13              PTM TIMER 1,3 CONTROL REGISTER
        FCB     0                   REMOVE RESET
        FDB     PTMCR2              PTM TIMER 2 CONTROL REGISTER
        FCB     %10000010           * BIT 7 = 1 TO PUT SQUARE
                                    *   WAVE OUTPUT ON O2
                                    * BIT 6 = 0 TO DISABLE INTERRUPT
                                    * BIT 5 = 0 FOR PULSE MODE
                                    * BIT 4 = 0 TO INITIALIZE COUNTER
                                    *   ON WRITE TO LATCHES
```

```
                                    * BIT 3 = 0 FOR CONTINUOUS OPERATION
                                    * BIT 2 = 0 FOR 16-BIT OPERATION
                                    * BIT 1 = 1 TO USE CPU CLOCK
                                    * BIT 0 = 0 TO ADDRESS CONTROL
                                    *   REGISTER 3
        FCB     PTM2MS              PTM TIMER 2 MS BYTE
        FDB     0                   MS BYTE OF COUNT
        FCB     PTM2LS              PTM TIMER 2 LS BYTE
        FDB     12                  LS BYTE OF COUNT
ENDPIN:                             END OF ARRAY
BEGPIN: FDB     PINIT               BASE ADDRESS OF ARRAY
SZINIT: FCB     (ENDPIN-PINIT)/3    NUMBER OF PORTS TO INITIALIZE

        END
```

8G Delay milliseconds (DELAY)

Provides a delay of between 1 and 256 ms, depending on the parameter supplied. A parameter value of 0 is interpreted as 256. The user must calculate the value CPMS (cycles per millisecond) to fit a particular computer. Typical values are 1000 for a 1 MHz clock and 2000 for a 2 MHz clock.

Procedure The program simply counts down register X for the appropriate amount of time as determined by the user-supplied constant. Extra instructions account for the call (JSR) instruction, return instruction, and routine overhead without changing anything.

Entry conditions

Number of milliseconds to delay (1 – 256) in register A

Exit conditions

Returns after the specified number of milliseconds with all registers except the condition code register unchanged

Example

Data:	(A) = number of milliseconds = $2A_{16} = 42_{10}$
Result:	Software delay of $2A_{16}$ (42_{10}) milliseconds, assuming that user supplies the proper value of CPMS

Registers used CC

Execution time 1 ms × (A)

Program size 31 bytes

Data memory required None

Special case (A) = 0 causes a delay of 256 ms.

```
*       Title           Delay Milliseconds
*       Name:           DELAY
*

*       Purpose:        Delay from 1 to 256 milliseconds
*
*       Entry:          Register A = number of milliseconds to delay.
*                         A 0 equals 256 milliseconds
*
*       Exit:           Returns to calling routine after the
*                       specified delay.
*
*       Registers Used: CC
*
*       Time:           1 millisecond * Register A
*
*       Size:           Program 54 bytes


*
*EQUATES
*CYCLES PER MILLISECOND - USER-SUPPLIED
*
CPMS    EQU     1000    *1000 = 1 MHZ CLOCK
                        *2000 = 2 MHZ CLOCK
                        *     .
                        *     .
MFAC    EQU     CPMS/20         MULTIPLYING FACTOR FOR ALL
                                * EXCEPT LAST MILLISECOND
MFACM   EQU     MFAC-4          MULTIPLYING FACTOR FOR LAST
                                * MILLISECOND

*
*METHOD:
* THE ROUTINE IS DIVIDED INTO 2 PARTS. THE CALL TO
* THE "DLY" ROUTINE DELAYS EXACTLY 1 LESS THAN THE
* NUMBER OF REQUIRED MILLISECONDS. THE LAST ITERATION
* TAKES INTO ACCOUNT THE OVERHEAD TO CALL "DELAY" AND
* "DLY".  THIS OVERHEAD IS 78 CYCLES.
*
DELAY:  *
        *DO ALL BUT THE LAST MILLISECOND
        *
        PSHS    D,X             SAVE REGISTERS
        LDB     #MFAC           GET MULTIPLYING FACTOR
        DECA                    REDUCE NUMBER OF MS BY 1
        MUL                     MULTIPLY FACTOR TIMES (MS - 1)
        TFR     D,X             TRANSFER LOOP COUNT TO X
        JSR     DLY
```

```
        *
        *ACCOUNT FOR 80 MS OVERHEAD DELAY BY REDUCING
        * LAST MILLISECOND'S COUNT
        *
        LDX     #MFAC1          GET REDUCED COUNT
        JSR     DLY             DELAY LAST MILLISECOND
        PULS    D,X             RESTORE REGISTERS
        RTS


*****************************
*ROUTINE: DLY
*PURPOSE: DELAY ROUTINE
*ENTRY: REGISTER X = COUNT
*EXIT:  REGISTER X = 0
*REGISTERS USED: X
*****************************

DLY:    BRA     DLY1
DLY1:   BRA     DLY2
DLY2:   BRA     DLY3
DLY3:   BRA     DLY4
DLY4:   LEAX    -1,X
        BNE     DLY
        RTS

*
*       SAMPLE EXECUTION:
*


SC8G:
        *
        *DELAY 10 SECONDS
        *  CALL DELAY 40 TIMES AT 250 MILLISECONDS EACH
        *
        LDB     #40             40 TIMES (28 HEX)
QTRSCD:
        LDA     #250            250 MILLISECONDS (FA HEX)
        JSR     DELAY
        DECB
        BNE     QTRSCD          CONTINUE UNTIL DONE

        BRA     SC8G            REPEAT OPERATION

        END     PROGRAM
```

9 Interrupts

9A Unbuffered interrupt-driven input/output using a 6850 ACIA (SINTIO)

Performs interrupt-driven input and output using a 6850 ACIA (Asynchronous Communications Interface Adapter) and single-character input and output buffers. Consists of the following subroutines:

1. INCH reads a character from the input buffer.
2. INST determines whether the input buffer is empty.
3. OUTCH writes a character into the output buffer.
4. OUTST determines whether the output buffer is full.
5. INIT initializes the 6850 ACIA, the interrupt vectors, and the software flags. The flags indicate when data can be transferred between the main program and the interrupt service routines.
6. IOSRVC determines which interrupt occurred and provides the proper input or output service. In response to the input interrupt, it reads a character from the ACIA into the input buffer. In response to the output interrupt, it writes a character from the output buffer into the ACIA.

Procedures

1. INCH waits for a character to become available, clears the Data Ready flag (RECDF), and loads the character into register A.

2. INST sets the Carry flag from the Data Ready flag (RECDF).

3. OUTCH waits for the output buffer to empty, stores the character in the buffer, and sets the Character Available flag (TRNDF).

4. OUTST sets the Carry flag from the Character Available flag (TRNDF).

5. INIT clears the software flags, resets the ACIA (a master reset, since the device has no reset input), and determines the ACIA's operating mode by placing the appropriate value in its control register. INIT starts the ACIA with input interrupts enabled and output interrupts disabled. See Subroutine 8E for more details about 6850 ACIA initialization.

6. IOSRVC determines whether the interrupt was an input interrupt (bit 0 of the ACIA status register = 1), an output interrupt (bit 1 of the ACIA status register = 1), or the product of some other device. If the input interrupt occurred, the program reads the data, saves it in memory, and sets the Data Ready flag (RECDF). The lack of buffering results in the loss of any unread data at this point.

If the output interrupt occurred, the program determines whether data is available. If not, the program simply disables the output interrupt. If data is available, the program sends it to the ACIA, clears the Character Available flag (TRNDF), and enables both the input and the output interrupts.

The special problem with the output interrupt is that it may occur when no data is available. We cannot ignore it or it will assert itself indefinitely, creating an endless loop. Nor can we clear an ACIA output interrupt without sending data to the device. The solution is to disable output interrupts. But this creates a new problem when data is ready to be sent. That is, if output interrupts are disabled, the system cannot learn from an interrupt that the ACIA is ready to transmit. The solution to this is to create an additional, non-interrupt-driven entry to the routine that sends a character to the ACIA. Since this entry is not caused by an interrupt, it must check whether the ACIA's output register is empty before sending it a character. The special sequence of operations is the following:

1. Output interrupt occurs before new data is available (i.e. the ACIA

becomes ready for data). The response is to disable the output interrupt, since there is no data to be sent. Note that this sequence will not occur initially, since INIT disables the output interrupt. Otherwise, the output interrupt would occur immediately, since the ACIA surely starts out empty and therefore ready to transmit data.

2. Output data becomes available. That is, the system now has data to transmit. But there is no use waiting for the output interrupt, since it has been disabled.

3. The main program calls the routine (OUTDAT), which sends data to the ACIA. Checking the ACIA's status shows that it is, in fact, ready to transmit a character (it told us it was by causing the output interrupt). The routine then sends the character and re-enables the interrupts.

Unserviceable interrupts occur only with output devices, since input devices always have data ready to transfer when they request service. Thus output devices cause more initialization and sequencing problems in interrupt-driven systems than do input devices.

The solution shown here may, however, result in an odd situation. Assume that the system has output data but the ACIA is not ready for it. The system must then wait with interrupts disabled for the ACIA to become ready. That is, an interrupt-driven system must disable its interrupts and wait idly, polling the output device. We could solve this problem with an extra software flag (output interrupt expected). The service routine would change this flag if the output interrupt occurred when no data was available. The system could then check the flag and determine whether the output interrupt had already occurred (see Subroutine 9C).

Entry conditions

1. INCH: none
2. INST: none
3. OUTCH: character to transmit in register A
4. OUTST: none
5. INIT: none

Exit conditions

1. INCH: character in register A
2. INST: Carry = 0 if input buffer is empty, 1 if it is full
3. OUTCH: none
4. OUTST: Carry = 0 if output buffer is empty, 1 if it is full
5. INIT: none

Registers used

1. INCH: A, CC
2. INST: A, CC
3. OUTCH: A, CC
4. OUTST: A, CC
5. INIT: A

Execution time

1. INCH: 40 cycles if a character is available
2. INST: 12 cycles
3. OUTCH: 87 cycles if the output buffer is empty and the ACIA is ready to transmit
4. OUTST: 12 cycles
5. INIT: 76 cycles
6. IOSRVC: 63 cycles to service an input interrupt, 99 cycles to service an output interrupt, 42 cycles to determine interrupt is from another device. Note that it takes the processor 21 cycles to respond to an interrupt, since it must save all user registers. The execution times given include these cycles.

Program size 144 bytes

Data memory required 6 bytes anywhere in RAM for the received

data (address RECDAT), receive data flag (address RECDF), transmit data (address TRNDAT), transmit data flag (address TRNDF), and the address of the next interrupt service routine (2 bytes starting at address NEXTSR).

```
*       Title           Simple interrupt input and output using a 6850
*                       ACIA and single character buffers.
*       Name:           SINTIO
*

*       Purpose:        This program consists of 5 subroutines that
*                       perform interrupt driven input and output using
*                       a 6850 ACIA.
*
*                       INCH
*                         Read a character.
*                       INST
*                         Determine input status (whether input
*                         buffer is empty).
*                       OUTCH
*                         Write a character.
*                       OUTST
*                         Determine output status (whether output
*                         buffer is full).
*                       INIT
*                         Initialize.
*
*       Entry:          INCH
*                         No parameters.
*                       INST
*                         No parameters.
*                       OUTCH
*                         Register A = character to transmit
*                       OUTST
*                         No parameters.
*                       INIT
*                         No parameters.
*
*       Exit:           INCH
*                         Register A = character.
*                       INST
*                         Carry = 0 if input buffer is empty,
*                         1 if character is available.
*                       OUTCH
*                         No parameters
*                       OUTST
*                         Carry = 0 if output buffer is empty,
*                         1 if it is full.
*                       INIT
*                         No parameters.
*
*       Registers used: INCH
*                         A,CC
*                       INST
*                         A,CC
*                       OUTCH
```

```
*                         A,CC
*                       OUTST
*                         A,CC
*                       INIT
*                         A
*
*       Time:           INCH
*                         40 cycles if a character is available
*                       INST
*                         12 cycles
*                       OUTCH
*                         87 cycles if output buffer is empty and
*                         the ACIA is ready to transmit
*                       OUTST
*                         12 cycles
*                       INIT
*                         76 cycles
*                       IOSRVC
*                         42 cycles minimum if the interrupt is not ours
*                         63 cycles to service an input interrupt
*                         99 cycles to service an output interrupt
*                         These include the time required for the
*                         processor to respond to an interrupt
*                         (21 cycles).
*
*       Size:           Program   144 bytes
*                       Data      6 bytes
*

*ARBITRARY 6850 ACIA MEMORY ADDRESSES
ACIADR  EQU     $A000           ACIA DATA REGISTER
ACIACR  EQU     $A001           ACIA CONTROL REGISTER
ACIASR  EQU     $A001           ACIA STATUS REGISTER
*TRS-80 COLOR COMPUTER INTERRUPT VECTOR
INTVEC  EQU     $010D           VECTOR TO INTERRUPT SERVICE ROUTINE

*
*     READ A CHARACTER FROM INPUT BUFFER
*
INCH:
        JSR     INST            GET INPUT STATUS
        BCC     INCH            WAIT IF NO CHARACTER AVAILABLE
        CLR     RECDF           INDICATE INPUT BUFFER EMPTY
        LDA     RECDAT          GET CHARACTER FROM INPUT BUFFER
        RTS
*
*     DETERMINE INPUT STATUS (CARRY = 1 IF DATA AVAILABLE)
*
INST:
        LDA     RECDF           GET DATA READY FLAG
        LSRA                    SET CARRY FROM DATA READY FLAG
                                * CARRY = 1 IF CHARACTER AVAILABLE
        RTS

*
*    WRITE A CHARACTER INTO OUTPUT BUFFER AND THEN ON TO ACIA
```

```
*
OUTCH:
        PSHS    A               SAVE CHARACTER TO WRITE

        *WAIT FOR OUTPUT BUFFER TO EMPTY, STORE NEXT CHARACTER
WAITOC:
        JSR     OUTST           GET OUTPUT STATUS
        BCS     WAITOC          WAIT IF OUTPUT BUFFER FULL
        PULS    A               GET CHARACTER
        STA     TRNDAT          STORE CHARACTER IN BUFFER
        LDA     #$FF            INDICATE BUFFER FULL
        STA     TRNDF
        JSR     OUTDAT          SEND CHARACTER TO PORT
        RTS

*
*    DETERMINE OUTPUT STATUS (CARRY = 1 IF OUTPUT BUFFER FULL)
*
OUTST:
        LDA     TRNDF           GET TRANSMIT FLAG
        LSRA                    SET CARRY FROM TRANSMIT FLAG
        RTS                       CARRY = 1 IF BUFFER FULL

*
*INITIALIZE INTERRUPT SYSTEM AND 6850 ACIA
*
INIT:
        *
        *DISABLE INTERRUPTS DURING INITIALIZATION BUT SAVE
        *  PREVIOUS STATE OF INTERRUPT FLAG
        *
        PSHS    CC              SAVE CURRENT FLAGS (PARTICULARLY I FLAG)
        SEI                     DISABLE INTERRUPTS DURING
                                * INITIALIZATION
        *
        *INITIALIZE TRS-80 COLOR COMPUTER INTERRUPT VECTOR
        *
        LDX     INTVEC          GET CURRENT INTERRUPT VECTOR
        STX     NEXTSR          SAVE IT AS ADDRESS OF NEXT SERVICE
                                * ROUTINE
        LDX     #IOSRVC         GET ADDRESS OF OUR SERVICE ROUTINE
        STX     INTVEC          SAVE IT AS INTERRUPT VECTOR
        *
        *INITIALIZE SOFTWARE FLAGS
        *
        CLR     RECDF           NO INPUT DATA AVAILABLE
        CLR     TRNDF           OUTPUT BUFFER EMPTY
        CLR     OIE             INDICATE NO OUTPUT INTERRUPT NEEDED
                                * 6850 READY TO TRANSMIT INITIALLY
        *
        *INITIALIZE 6850 ACIA (UART)
        *
        LDA     #%00000011    MASTER RESET ACIA (IT HAS NO RESET INPUT).
        STA     ACIACR
        LDA     #%10010001    INITIALIZE ACIA MODE
                              *BIT 7 = 1 TO ENABLE INPUT INTERRUPTS
```

```
                                *BITS 6,5 = 0 TO DISABLE OUTPUT INTERRUPTS
                                *BITS 4,3,2 = 100 FOR 8 DATA BITS, 2 STOP
                                *  BITS
                                *BITS 1,0 = 01 FOR DIVIDE BY 16 CLOCK
        STA     ACIACR
        PULS    CC              RESTORE FLAGS (THIS REENABLES INTERRUPTS
                                * IF THEY WERE ENABLED WHEN INIT WAS
                                * CALLED)
        RTS


*
*GENERAL INTERRUPT HANDLER
*
IOSRVC:
        *
        *GET ACIA STATUS: BIT 0 = 1 IF AN INPUT INTERRUPT,
        *  BIT 1 = 1 IF AN OUTPUT INTERRUPT
        *
        LDA     ACIASR          GET ACIA STATUS
        LSRA                    EXAMINE BIT 0
        BCS     RDHDLR          BRANCH IF AN INPUT INTERRUPT
        LSRA                    EXAMINE BIT 1
        BCS     WRHDLR          BRANCH IF AN OUTPUT INTERRUPT
        JMP     [NEXTSR]        NOT THIS ACIA, EXAMINE NEXT INTERRUPT
*
*INPUT (READ) INTERRUPT HANDLER
*
RDHDLR:
        LDA     ACIADR          LOAD DATA FROM 6850 ACIA
        STA     RECDAT          SAVE DATA IN INPUT BUFFER
        LDA     #$FF
        STA     RECDF           INDICATE INPUT DATA AVAILABLE
        RTI

*
*OUTPUT (WRITE) INTERRUPT HANDLER
*
WRHDLR:
        LDA     TRNDF           TEST DATA AVAILABLE FLAG
        BEQ     NODATA          JUMP IF NO DATA TO TRANSMIT
        JSR     OUTDT1          ELSE SEND DATA TO 6850 ACIA
        BRA     WRDONE            (NO NEED TO TEST STATUS)
*
*IF AN OUTPUT INTERRUPT OCCURS WHEN NO DATA IS AVAILABLE,
* WE MUST DISABLE IT (IN THE 6850) TO AVOID AN ENDLESS LOOP.  LATER,
* WHEN A CHARACTER BECOMES AVAILABLE, WE CALL THE OUTPUT ROUTINE
* OUTDAT WHICH MUST TEST ACIA STATUS BEFORE SENDING THE DATA.
* THE OUTPUT ROUTINE MUST ALSO REENABLE THE OUTPUT INTERRUPT AFTER
* SENDING THE DATA.  THIS PROCEDURE OVERCOMES THE PROBLEM OF AN
* UNSERVICED OUTPUT INTERRUPT ASSERTING ITSELF REPEATEDLY, WHILE
* STILL ENSURING THAT OUTPUT INTERRUPTS ARE RECOGNIZED AND THAT
* DATA IS NEVER SENT TO AN ACIA THAT IS NOT READY FOR IT.
*THE PROBLEM IS THAT AN OUTPUT DEVICE MAY REQUEST SERVICE BEFORE
* THE COMPUTER HAS ANYTHING TO SEND (UNLIKE AN INPUT DEVICE THAT
* HAS DATA WHEN IT REQUESTS SERVICE).
```

```
*
NODATA:
        LDA     #%10010001      ESTABLISH ACIA OPERATING MODE
                                * WITH OUTPUT INTERRUPTS DISABLED
        STA     ACIACR
WRDONE:
        RTI

*************************************
*ROUTINE: OUTDAT, OUTDT1 (OUTDAT IS NON-INTERRUPT DRIVEN ENTRY POINT)
*PURPOSE: SEND A CHARACTER TO THE ACIA
*ENTRY: TRNDAT = CHARACTER TO SEND
*EXIT:  NONE
*REGISTERS USED: A,CC
*************************************

OUTDAT:
        LDA     ACIASR          CAME HERE WITH INTERRUPTS DISABLED
        AND     #%00000010      TEST WHETHER ACIA OUTPUT REGISTER EMPTY
        BEQ     OUTDAT          BRANCH (WAIT) IF IT IS NOT EMPTY
OUTDT1:
        LDA     TRNDAT          GET THE CHARACTER
        STA     ACIADR          SEND CHARACTER TO ACIA
        CLR     TRNDF           INDICATE OUTPUT BUFFER EMPTY
        LDA     #%10110001      ESTABLISH ACIA OPERATING MODE WITH
        STA     ACIACR            OUTPUT INTERRUPTS ENABLED
        RTS

*
*DATA SECTION
*
RECDAT  RMB     1               RECEIVE DATA
RECDF   RMB     1               RECEIVE DATA FLAG
                                *  (0 = NO DATA, FF = DATA AVAILABLE)
TRNDAT  RMB     1               TRANSMIT DATA
TRNDF   RMB     1               TRANSMIT DATA FLAG
                                *  (0 = BUFFER EMPTY, FF = BUFFER FULL)
NEXTSR  RMB     2               ADDRESS OF NEXT INTERRUPT SERVICE
                                *  ROUTINE

*
*       SAMPLE EXECUTION:
*
*CHARACTER EQUATES
ESCAPE  EQU     $1B             ASCII ESCAPE CHARACTER
TESTCH  EQU     'A              TEST CHARACTER = A

SC9A:
        JSR     INIT            INITIALIZE 6850 ACIA, INTERRUPT SYSTEM
        CLI                     ENABLE INTERRUPTS
        *
        *SIMPLE EXAMPLE - READ AND ECHO CHARACTERS
        * UNTIL AN ESC IS RECEIVED
        *
LOOP:
        JSR     INCH            READ CHARACTER
```

```
        PSHS    A
        JSR     OUTCH           ECHO CHARACTER
        PULS    A
        CMPA    #ESCAPE         IS CHARACTER AN ESCAPE?
        BNE     LOOP            STAY IN LOOP IF NOT
        *
        *AN ASYNCHRONOUS EXAMPLE
        * OUTPUT "A" TO CONSOLE CONTINUOUSLY BUT ALSO LOOK AT
        * INPUT SIDE, READING AND ECHOING ANY INPUT CHARACTERS.
        *
ASYNLP:
        *
        *OUTPUT AN "A" IF OUTPUT IS NOT BUSY
        *
        JSR     OUTST           IS OUTPUT BUSY?
        BCS     ASYNLP          JUMP IF IT IS
        LDA     #TESTCH
        JSR     OUTCH           OUTPUT TEST CHARACTER
        *
        *CHECK INPUT PORT
        *ECHO CHARACTER IF ONE IS AVAILABLE
        *EXIT ON ESCAPE CHARACTER
        *
        JSR     INST            IS INPUT DATA AVAILABLE?
        BCS     ASYNLP          JUMP IF NOT (SEND ANOTHER "A")
        JSR     INCH            GET CHARACTER
        CMPA    #ESCAPE         IS IT AN ESCAPE?
        BEQ     DONE            BRANCH IF IT IS
        JSR     OUTCH           ELSE ECHO CHARACTER
        BRA     ASYNLP          AND CONTINUE

DONE:
        BRA     SC9A            REPEAT TEST

        END
```

9B Unbuffered interrupt-driven input/output using a 6821 PIA (PINTIO)

Performs interrupt-driven input and output using a 6821 PIA and single-character input and output buffers. Consists of the following sub-routines:

1. INCH reads a character from the input buffer.
2. INST determines whether the input buffer is empty.
3. OUTCH writes a character into the output buffer.
4. OUTST determines whether the output buffer is full.
5. INIT initializes the 6820 PIA and the software flags. The flags indicate when data can be transferred between the main program and the interrupt service routines.
6. IOSRVC determines which interrupt occurred and provides the proper input or output service. That is, it reads a character from the PIA into the input buffer in response to the input interrupt, and it writes a character from the output buffer into the PIA in response to the output interrupt.

Procedure

1. INCH waits for a character to become available, clears the Data Ready flag (RECDF), and loads the character into register A.
2. INST sets the Carry flag from the Data Ready flag (RECDF).
3. OUTCH waits for the output buffer to empty, places the character from register A in the buffer, and sets the character available flag (TRNDF). If an unserviced output interrupt has occurred (i.e. the output device has requested service when no data was available), OUTCH actually sends the data to the PIA.
4. OUTST sets Carry from the Character Available flag (TRNDF).
5. INIT clears the software flags and initializes the 6821 PIA by loading its control and data direction registers. It makes port A an input port, port B an output port, control lines CA1 and CB1 active low-to-high, control line CA2 a brief output pulse indicating input acknowledge (active-low briefly after the CPU reads the data) and control line CB2 a write strobe (active-low after the CPU writes the data and lasting until the peripheral becomes ready again). INIT also enables the input inter-

rupt on CA1 and the output interrupt on CB1. See Appendix 2 and Subroutine 8E for more details about initializing 6821 PIAs.

6. IOSRVC determines whether the interrupt was an input interrupt (bit 7 of PIA control register A = 1), an output interrupt (bit 7 of PIA control register B = 1), or the product of some other device. If an input interrupt occurred, the program reads the data, saves it in the input buffer, and sets the Data Ready flag (RECDF). The lack of buffering results in the loss of any unread data at this point.

If an output interrupt occurred, the program determines whether any data is available. If not, the program simply clears the interrupt and clears the flag (OIE) that indicates the output device is actually ready (i.e. an output interrupt has occurred at a time when no data was available). If data is available, the program sends it from the output buffer to the PIA, clears the Character Available flag (TRNDF), sets the Output Interrupt Expected flag (OIE), and enables both the input and the output interrupts.

The special problem with the output interrupt is that it may occur when no data is available to send. We cannot ignore it or it will assert itself indefinitely, causing an endless loop. The solution is simply to clear the 6821 interrupt by reading the data register in port B.

But now we have a new problem when output data becomes available. That is, since the interrupt has been cleared, it obviously cannot inform the system that the output device is ready for data. The solution is to have a flag that indicates (with a 0 value) that the output interrupt has occurred without being serviced. We call this flag OIE (Output Interrupt Expected).

The initialization routine clears OIE (since the output device starts out ready for data). The output service routine clears it when an output interrupt occurs that cannot be serviced (no data is available) and sets it after sending data to the 6821 PIA (in case it might have been cleared). Now the output routine OUTCH can check OIE to determine whether an output interrupt is expected. If not, OUTCH simply sends the data immediately.

Note that we can clear a PIA interrupt without actually sending any data. We cannot do this with a 6850 ACIA (see Subroutines 9A and 9C), so the procedures there are somewhat different.

Unserviceable interrupts occur only with output devices, since input devices always have data ready to transfer when they request service. Thus output devices cause more initialization and sequencing problems in interrupt-driven systems than do input devices.

Entry conditions

1. INCH : none
2. INST: none
3. OUTCH: character to transmit in register A
4. OUTST: none
5. INIT: none

Exit conditions

1. INCH: character in register A
2. INST: Carry = 0 if input buffer is empty, 1 if it is full
3. OUTCH: none
4. OUTST: Carry = 0 if output buffer is empty, 1 if it is full
5. INIT: none

Registers used

1. INCH: A, CC
2. INST: A, CC
3. OUTCH: A, CC
4. OUTST: A, CC
5. INIT: A

Execution time

1. INCH: 40 cycles if a character is available
2. INST: 12 cycles
3. OUTCH: 98 cycles if the output buffer is not full and the PIA is ready for data; 37 additional cycles to send the data to the 6821 PIA if no output interrupt is expected.
4. OUTST: 12 cycles
5. INIT: 99 cycles

6. IOSRVC: 61 cycles to service an input interrupt, 97 cycles to service an output interrupt, 45 cycles to determine that an interrupt is from another device. These times all include the 21 cycles required by the CPU to respond to an interrupt.

Program size 158 bytes

Data memory required 7 bytes anywhere in RAM for the received data (address RECDAT), receive data flag (address RECDF), transmit data (address TRNDAT), transmit data flag (address TRNDF), output interrupt expected flag (address OIE), and the address of the next interrupt service routine (2 bytes starting at address NEXTSR).

```
*       Title           Simple interrupt input and output using a 6821
*                       Peripheral Interface Adapter and single
*                       character buffers.
*       Name:           PINTIO
*
*
*       Purpose:        This program consists of 5 subroutines that
*                       perform interrupt driven input and output using
*                       a 6821 PIA.
*
*                       INCH
*                         Read a character.
*                       INST
*                         Determine input status (whether input
*                         buffer is empty).
*                       OUTCH
*                         Write a character.
*                       OUTST
*                         Determine output status (whether output
*                         buffer is full).
*                       INIT
*                         Initialize 6821 PIA and interrupt system.
*
*       Entry:          INCH
*                         No parameters.
*                       INST
*                         No parameters.
*                       OUTCH
*                         Register A = character to transmit
*                       OUTST
*                         No parameters.
*                       INIT
*                         No parameters.
*
*       Exit:           INCH
*                         Register A = character.
```

```
*                       INST
*                         Carry = 0 if input buffer is empty,
*                         1 if character is available.
*                       OUTCH
*                         No parameters
*                       OUTST
*                         Carry = 0 if output buffer is
*                         empty, 1 if it is full.
*                       INIT
*                         No parameters.
*
*       Registers Used: INCH
*                         A,CC
*                       INST
*                         A,CC
*                       OUTCH
*                         A,CC
*                       OUTST
*                         A,CC
*                       INIT
*                         A
*
*       Time:           INCH
*                         40 cycles if a character is available
*                       INST
                          12 cycles
*                       OUTCH
*                         98 cycles if output buffer is not full and
*                         output interrupt is expected
*                       OUTST
*                         12 cycles
*                       INIT
*                         99 cycles
*                       IOSRVC
*                         45 cycles minimum if the interrupt is not ours
*                         61 cycles to service an input interrupt
*                         97 cycles to service an output interrupt
*                         These include the 21 cycles required for the
*                         processor to respond to an interrupt.
*
*       Size:           Program   158 bytes
*                       Data      7 bytes
*

*
*6821 PIA EQUATES
*ARBITRARY 6821 PIA MEMORY ADDRESSES
*
PIADRA  EQU     $A400           PIA DATA REGISTER A
PIADDA  EQU     $A400           PIA DATA DIRECTION REGISTER A
PIACRA  EQU     $A401           PIA CONTROL REGISTER A
PIADRB  EQU     $A402           PIA DATA REGISTER B
PIADDB  EQU     $A402           PIA DATA DIRECTION REGISTER B
PIACRB  EQU     $A403           PIA CONTROL REGISTER B
*
*TRS-80 COLOR COMPUTER INTERRUPT VECTOR
```

```
*
INTVEC  EQU     $010D           VECTOR TO INTERRUPT SERVICE ROUTINE
*
*READ A CHARACTER FROM INPUT BUFFER
*
INCH:
        JSR     INST            GET INPUT STATUS
        BCC     INCH            WAIT IF NO CHARACTER AVAILABLE
        CLR     RECDF           INDICATE INPUT BUFFER EMPTY
        LDA     RECDAT          GET CHARACTER FROM INPUT BUFFER
        RTS

*
*DETERMINE INPUT STATUS (CARRY = 1 IF DATA AVAILABLE)
*
INST:
        LDA     RECDF           GET DATA READY FLAG
        LSRA                    SET CARRY FROM DATA READY FLAG
                                * CARRY = 1 IF CHARACTER AVAILABLE
        RTS

*
*WRITE A CHARACTER INTO OUTPUT BUFFER
*
OUTCH:
        PSHS    A               SAVE CHARACTER TO WRITE

        *WAIT FOR OUTPUT BUFFER TO EMPTY, STORE NEXT CHARACTER
WAITOC:
        JSR     OUTST           GET OUTPUT STATUS
        BCS     WAITOC          WAIT IF OUTPUT BUFFER FULL
        PULS    A               GET CHARACTER
        STA     TRNDAT          STORE CHARACTER IN OUTPUT BUFFER
        LDA     #$FF            INDICATE OUTPUT BUFFER FULL
        STA     TRNDF
        TST     OIE             TEST OUTPUT INTERRUPT EXPECTED FLAG
        BNE     EXITOT          EXIT IF OUTPUT INTERRUPT EXPECTED
        JSR     OUTDAT          SEND CHARACTER IMMEDIATELY IF
                                * NO OUTPUT INTERRUPT EXPECTED
EXITOT:
        RTS


*
*DETERMINE OUTPUT STATUS (CARRY = 1 IF OUTPUT BUFFER FULL)
*
OUTST:
        LDA     TRNDF           GET TRANSMIT FLAG
        LSRA                    SET CARRY FROM TRANSMIT FLAG
        RTS                     CARRY = 1 IF BUFFER FULL


*
*INITIALIZE INTERRUPT SYSTEM AND 6821 PIA
*
INIT:
```

```
*
*DISABLE INTERRUPTS DURING INITIALIZATION BUT SAVE
*  PREVIOUS STATE OF INTERRUPT FLAG
*
PSHS    CC              SAVE CURRENT FLAGS (PARTICULARLY I FLAG)
SEI                     DISABLE INTERRUPTS DURING
                        * INITIALIZATION
*
*INITIALIZE TRS-80 COLOR COMPUTER INTERRUPT VECTOR
*
LDX     INTVEC          GET CURRENT INTERRUPT VECTOR
STX     NEXTSR          SAVE IT AS ADDRESS OF NEXT SERVICE
                        * ROUTINE
LDX     #IOSRVC         GET ADDRESS OF OUR SERVICE ROUTINE
STX     INTVEC          SAVE IT AS INTERRUPT VECTOR
*
*INITIALIZE SOFTWARE FLAGS
*
CLRA
STA     RECDF           NO INPUT DATA AVAILABLE
STA     TRNDF           OUTPUT BUFFER EMPTY
STA     OIE             INDICATE NO OUTPUT INTERRUPT NEEDED
                        * 6821 READY TO TRANSMIT INITIALLY

*
*INITIALIZE 6821 PIA (PARALLEL INTERFACE)
*
CLR     PIACRA          ADDRESS DATA DIRECTION REGISTERS
CLR     PIACRB
CLR     PIADDA          MAKE PORT A INPUT
LDA     #$FF            MAKE PORT B OUTPUT
STA     PIADDB
LDA     #%00101111
STA     PIACRA          SET PORT A AS FOLLOWS:
                        *BITS 7,6 NOT USED
                        *BIT 5 = 1 TO MAKE CA2 OUTPUT
                        *BIT 4 = 0 TO MAKE CA2 A PULSE
                        *BIT 3 = 1 TO MAKE CA2 A BRIEF INPUT
                        *  ACKNOWLEDGE
                        *BIT 2 = 1 TO ADDRESS DATA REGISTER
                        *BIT 1 = 1 TO MAKE CA1 ACTIVE LOW-TO-
                        *  HIGH
                        *BIT 0 = 1 TO ENABLE CA1 INTERRUPTS
LDA     #%00100111
STA     PIACRB          SET PORT B AS FOLLOWS:
                        *BITS 7, 6 NOT USED
                        *BIT 5 = 1 TO MAKE CB2 OUTPUT
                        *BIT 4 = 0 TO MAKE CB2 A PULSE
                        *BIT 3 = 0 TO MAKE CB2 A LONG OUTPUT
                        *  BUFFER FULL
                        *BIT 2 = 1 TO ADDRESS DATA REGISTER
                        *BIT 1 = 1 TO MAKE CB1 ACTIVE LOW-TO-
                        *  HIGH
                        *BIT 0 = 1 TO ENABLE CB1 INTERRUPTS
PULS    CC              RESTORE FLAGS (THIS REENABLES INTERRUPTS
                        * IF THEY WERE ENABLED WHEN INIT WAS
```

```
                                * CALLED)
        RTS

*
*INTERRUPT MANAGER
*DETERMINES WHETHER INPUT OR OUTPUT INTERRUPT OCCURRED
*
IOSRVC:
        *
        *INPUT INTERRUPT FLAG IS BIT 7 OF CONTROL REGISTER A
        *OUTPUT INTERRUPT FLAG IS BIT 7 OF CONTROL REGISTER B
        *
        LDA     PIACRA          CHECK FOR INPUT INTERRUPT
        BMI     RDHDLR          BRANCH IF INPUT INTERRUPT
        LDA     PIACRB          CHECK FOR OUTPUT INTERRUPT
        BMI     WRHDLR          BRANCH IF OUTPUT INTERRUPT
        JMP     [NEXTSR]        INTERRUPT IS FROM ANOTHER SOURCE

*
*INPUT (READ) INTERRUPT HANDLER
*
RDHDLR:
        LDA     PIADRA          READ DATA FROM 6821 PIA
        STA     RECDAT          SAVE DATA IN INPUT BUFFER
        LDA     #$FF
        STA     RECDF           INDICATE CHARACTER AVAILABLE
        RTI

*
*OUTPUT (WRITE) INTERRUPT HANDLER
*
WRHDLR:
        LDA     TRNDF           TEST DATA AVAILABLE FLAG
        BEQ     NODATA          JUMP IF NO DATA TO TRANSMIT
        JSR     OUTDAT          SEND DATA TO 6821 PIA
        RTI
*
*IF AN OUTPUT INTERRUPT OCCURS WHEN NO DATA IS AVAILABLE,
* WE MUST CLEAR IT (IN THE 6821) TO AVOID AN ENDLESS LOOP.  LATER,
* WHEN A CHARACTER BECOMES AVAILABLE, WE NEED TO KNOW THAT AN
* OUTPUT INTERRUPT HAS OCCURRED WITHOUT BEING SERVICED.  THE KEY
* TO DOING THIS IS THE OUTPUT INTERRUPT EXPECTED FLAG OIE.  THIS FLAG IS
* CLEARED WHEN AN OUTPUT INTERRUPT HAS OCCURRED BUT HAS NOT BEEN
* SERVICED.  IT IS ALSO CLEARED INITIALLY SINCE THE 6821 PIA STARTS
* OUT READY.  OIE IS SET WHENEVER DATA IS ACTUALLY SENT TO THE PIA.
* THUS THE OUTPUT ROUTINE OUTCH CAN CHECK OIE TO DETERMINE WHETHER
* TO SEND THE DATA IMMEDIATELY OR WAIT FOR AN OUTPUT INTERRUPT.
*THE PROBLEM IS THAT AN OUTPUT DEVICE MAY REQUEST SERVICE BEFORE
* THE COMPUTER HAS ANYTHING TO SEND (UNLIKE AN INPUT DEVICE THAT
* HAS DATA WHEN IT REQUESTS SERVICE).  THE OIE FLAG SOLVES THE
* PROBLEM OF AN UNSERVICED OUTPUT INTERRUPT ASSERTING ITSELF
* REPEATEDLY, WHILE STILL ENSURING THE RECOGNITION OF OUTPUT
* INTERRUPTS.
*
NODATA:
        LDA     PIADRB          READ PORT B DATA REGISTER TO CLEAR
```

```
                                * INTERRUPT
        CLR     OIE             DO NOT EXPECT AN INTERRUPT

WRDONE:
        RTI


*************************************
*ROUTINE: OUTDAT
*PURPOSE: SEND CHARACTER TO 6821 PIA
*ENTRY: TRNDAT = CHARACTER TO SEND
*EXIT:  NONE
*REGISTERS USED: A,CC
*************************************

OUTDAT:
        LDA     TRNDAT          GET DATA FROM OUTPUT BUFFER
        STA     PIADRB          SEND DATA TO 6821 PIA
        CLR     TRNDF           INDICATE OUTPUT BUFFER EMPTY
        LDA     #$FF            INDICATE OUTPUT INTERRUPT EXPECTED
        STA     OIE             OIE = FF HEX
        RTS


*DATA SECTION
RECDAT  RMB     1               RECEIVE DATA
RECDF   RMB     1               RECEIVE DATA FLAG (0 = NO DATA,
                                *  FF = DATA)
TRNDAT  RMB     1               TRANSMIT DATA
TRNDF   RMB     1               TRANSMIT DATA FLAG
                                *  (0 = BUFFER EMPTY, FF = BUFFER FULL)
OIE     RMB     1               OUTPUT INTERRUPT EXPECTED
                                * ( 0 = INTERRUPT OCCURRED WITHOUT
                                *  BEING SERVICED, FF = INTERRUPT
                                *  SERVICED)
NEXTSR  RMB     2               ADDRESS OF NEXT INTERRUPT SERVICE
                                *  ROUTINE
*
*       SAMPLE EXECUTION:
*
*CHARACTER EQUATES
*
ESCAPE  EQU     $1B             ASCII ESCAPE CHARACTER
TESTCH  EQU     'A              TEST CHARACTER = A

SC9B:
        JSR     INIT            INITIALIZE 6821 PIA, INTERRUPT SYSTEM
        CLI                     ENABLE INTERRUPTS
        *
        *SIMPLE EXAMPLE - READ AND ECHO CHARACTERS
        * UNTIL AN ESC IS RECEIVED
        *
LOOP:
        JSR     INCH            READ CHARACTER
        PSHS    A
```

```
        JSR     OUTCH           ECHO CHARACTER
        PULS    A
        CMPA    #ESCAPE                 IS CHARACTER AN ESCAPE?
        BNE     LOOP            STAY IN LOOP IF NOT
        *
        *AN ASYNCHRONOUS EXAMPLE
        * OUTPUT "A" TO CONSOLE CONTINUOUSLY BUT ALSO LOOK AT
        * INPUT SIDE, READING AND ECHOING INPUT CHARACTERS.
        *
ASYNLP:
        *OUTPUT AN "A" IF OUTPUT IS NOT BUSY
        JSR     OUTST           IS OUTPUT BUSY?
        BCS     ASYNLP          BRANCH (WAIT) IF IT IS
        LDA     #TESTCH
        JSR     OUTCH           OUTPUT TEST CHARACTER
        *
        *CHECK INPUT PORT
        *ECHO CHARACTER IF ONE IS AVAILABLE
        *EXIT ON ESCAPE CHARACTER
        *
        JSR     INST            IS INPUT DATA AVAILABLE?
        BCS     ASYNLP          BRANCH IF NOT (SEND ANOTHER "A")
        JSR     INCH            GET CHARACTER
        CMPA    #ESCAPE         IS IT AN ESCAPE?
        BEQ     DONE            BRANCH IF IT IS
        JSR     OUTCH           ELSE ECHO CHARACTER
        BRA     ASYNLP          AND CONTINUE

DONE:
        BRA     SC9B            REPEAT TEST

        END
```

9C Buffered interrupt-driven input/output using a 6850 ACIA (SINTB)

Performs interrupt-driven input and output using a 6850 ACIA and multiple-character buffers. Consists of the following subroutines:

1. INCH reads a character from the input buffer.
2. INST determines whether the input buffer is empty.
3. OUTCH writes a character into the output buffer.
4. OUTST determines whether the output buffer is full.
5. INIT initializes the buffers, the interrupt system, and the 6850 ACIA.
6. IOSRVC determines which interrupt occurred and services ACIA input or output interrupts.

Procedures

1. INCH waits for a character to become available, gets the character from the head of the input buffer, moves the head of the buffer up one position, and decreases the input buffer counter by 1.

2. INST clears Carry if the input buffer counter is 0 and sets it otherwise.

3. OUTCH waits until there is space in the output buffer (i.e. until the output buffer is not full), stores the character at the tail of the buffer, moves the tail up one position, and increases the output buffer counter by 1.

4. OUTST sets Carry if the output buffer counter is equal to the buffer's length (i.e. if the output buffer is full) and clears Carry otherwise.

5. INIT clears the buffer counters and sets all buffer pointers to the buffers' base addresses. It then resets the 6850 ACIA and sets its operating mode by storing the appropriate value in its control register. It initializes the ACIA with input interrupts enabled and output interrupts disabled. See Subroutine 8E for more details about initializing 6850 ACIAs. INIT also clears the OIE flag, indicating that the ACIA is ready to transmit data, although it cannot cause an output interrupt.

6. IOSRVC determines whether the interrupt was an input interrupt

(bit 0 of the ACIA status register = 1), an output interrupt (bit 1 of the ACIA status register = 1), or the product of some other device. If the input interrupt occurred, the program reads a character from the 6850 ACIA. If there is room in the input buffer, it stores the character at the tail of the buffer, moves the tail up one position, and increases the input buffer counter by 1. If the buffer is full, it simply discards the character.

If the output interrupt occurred, the program determines whether output data is available. If not, it simply disables the output interrupt (so it will not interrupt repeatedly) and clears the OIE flag that indicates the ACIA is actually ready. The flag tells the main program that the ACIA is ready even though it cannot force an interrupt. If there is data in the output buffer, the program obtains a character from the buffer's head, sends it to the ACIA, moves the head up one position, and decreases the output buffer counter by 1. It then enables both input and output interrupts and sets the OIE flag (in case the flag had been cleared earlier).

The new problem with multiple-character buffers is the management of queues. The main program must read the data in the order in which the input interrupt service routine receives it. Similarly, the output interrupt service routine must send the data in the order in which the main program stores it. Thus we have the following requirements for handling input:

1. The main program must know whether the input buffer is empty.

2. If the input buffer is not empty, the main program must know where the oldest character is (i.e. the one that was received first).

3. The input interrupt service routine must know whether the input buffer is full.

4. If the input buffer is not full, the input interrupt service routine must know where the next empty place is (i.e. where it should store the new character).

The output interrupt service routine and the main program have similar requirements for the output buffer, although the roles of sender and receiver are reversed.

We meet requirements 1 and 3 by maintaining a counter ICNT. INIT initializes ICNT to 0, the interrupt service routine adds 1 to it whenever it receives a character (assuming the buffer is not full), and the main program subtracts 1 from it whenever it removes a character from the buffer. Thus the main program can determine whether the input buffer is empty by checking if ICNT is 0. Similarly, the interrupt service

routine can determine whether the input buffer is full by checking if ICNT is equal to the size of the buffer.

We meet requirements 2 and 4 by maintaining two pointers, IHEAD and ITAIL, defined as follows:

1. ITAIL is the address of the next empty location in the input buffer.
2. IHEAD is the address of the oldest character in the input buffer.

INIT initializes IHEAD and ITAIL to the base address of the input buffer. Whenever the interrupt service routine receives a character, it places it in the buffer at ITAIL and moves ITAIL up one position (assuming that the buffer is not full). Whenever the main program reads a character, it removes it from the buffer at IHEAD and moves IHEAD up one position. Thus IHEAD 'chases' ITAIL across the buffer with the service routine entering characters at one end (the tail) while the main program removes them from the other end (the head).

The occupied part of the buffer thus could start and end anywhere. If either IHEAD or ITAIL reaches the physical end of the buffer, we simply set it back to the base address. Thus we allow wraparound on the buffer; i.e. the occupied part of the buffer could start near the end (say, at byte #195 of a 200-byte buffer) and continue back past the beginning (say, to byte #10). Then IHEAD would be BASE + 194, ITAIL would be BASE + 9, and the buffer would contain 15 characters occupying addresses BASE + 194 through BASE + 199 and BASE through BASE + 8.

Entry conditions

1. INCH: none
2. INST: none
3. OUTCH: character to transmit in register A
4. OUTST: none
5. INIT: none

Exit conditions

1. INCH: character in register A
2. INST: Carry = 0 if input buffer is empty, 1 if a character is available

3. OUTCH: none
4. OUTST: Carry = 0 if output buffer is not full, 1 if it is full
5. INIT: none

Registers used

1. INCH: A, CC, X
2. INST: A, CC
3. OUTCH: A, CC, X
4. OUTST: A, CC
5. INIT: A

Execution time

1. INCH: approximately 86 cycles if a character is available
2. INST: 21 cycles
3. OUTCH: approximately 115 cycles if the output buffer is not full and an output interrupt is expected. Approximately an additional 79 cycles if no output interrupt is expected.
4. OUTST: 26 cycles
5. INIT: 106 cycles
6. IOSRVC: 112 cycles to service an input interrupt, 148 cycles to service an output interrupt, 44 cycles to determine the interrupt is from another device. These times all include the 21 cycles required by the CPU to respond to an interrupt.

Note The approximations here are the result of the variable amount of time required to update the buffer pointers with wraparound.

Program size 235 bytes

Data memory required 11 bytes anywhere in RAM for the heads and

tails of the input and output buffers (2 bytes starting at addresses IHEAD, ITAIL, OHEAD, and OTAIL, respectively), the number of characters in the buffers (2 bytes at addresses ICNT and OCNT), and the OIE flag (address OIE). This does not include the actual input and output buffers. The input buffer starts at address IBUF and its size is IBSZ; the output buffer starts at address OBUF and its size is OBSZ.

```
*       Title           Interrupt input and output using a 6850 ACIA
*                       and multiple character buffers.
*       Name:           SINTB
*
*       Purpose:        This program consists of 5 subroutines which
*                       perform interrupt driven input and output using
*                       a 6850 ACIA.
*
*                       INCH
*                         Read a character.
*                       INST
*                         Determine input status (whether input
*                         buffer is empty).
*                       OUTCH
*                         Write a character.
*                       OUTST
*                         Determine output status (whether output
*                         buffer is full).
*                       INIT
*                         Initialize 6850 ACIA and interrupt system.
*
*       Entry:          INCH
*                         No parameters.
*                       INST
*                         No parameters.
*                       OUTCH
*                         Register A = character to transmit
*                       OUTST
*                         No parameters.
*                       INIT
*                         No parameters.
*
*       Exit:           INCH
*                         Register A = character.
*                       INST
*                         Carry = 0 if input buffer is empty,
*                         1 if character is available.
*                       OUTCH
*                         No parameters
*                       OUTST
*                         Carry = 0 if output buffer is not
*                         full, 1 if it is full.
*                       INIT
*                         No parameters.
*
*       Registers Used: INCH
*                         A,CC,X
```

```
*                       INST
*                         A,CC
*                       OUTCH
*                         A,CC,X
*                       OUTST
*                         A,CC
*                       INIT
*                         A,X
*
*       Time:           INCH
*                         Approximately 86 cycles if a character is
*                         available
*                       INST
*                         21 cycles
*                       OUTCH
*                         Approximately 115 cycles if output buffer is
*                         not full and output interrupt is expected.
*                       OUTST
*                         26 cycles
*                       INIT
*                         106 cycles
*                       IOSRVC
*                         44 cycles minimum if the interrupt is not ours
*                         112 cycles to service an input interrupt
*                         148 cycles to service an output interrupt
*                         These include the 21 cycles required for the
*                         processor to respond to an interrupt.
*
*       Size:           Program 235 bytes
*                       Data    11 bytes plus size of buffers
*
*       Buffers:        The routines assume two buffers starting at
*                       address IBUF and OBUF.  The length of the
*                       buffers in bytes are IBSZ and OBSZ.  For the
*                       input buffer, IHEAD is the address of the
*                       oldest character (the next one the main
*                       program should read), ITAIL is the address of
*                       the next empty element (the next one the service
*                       routine should fill), and ICNT is the number of
*                       bytes currently filled with characters.  For the
*                       output buffer, OHEAD is the address of the
*                       character (the next one the service routine
*                       should send), OTAIL is the address of the next
*                       empty element (the next one the main program
*                       should fill), and OCNT is the number of bytes
*                       currently filled with characters.
*
*       Note:           Wraparound is provided on both buffers, so that
*                       the currently filled area may start anywhere
*                       and extend through the end of the buffer and
*                       back to the beginning.  For example, if the
*                       output buffer is 40 hex bytes long, the section
*                       filled with characters could extend from OBUF
*                       +32H (OHEAD) through OBUF+10H (OTAIL-1).  That
*                       is, there are 19H filled bytes occupying
*                       addresses OBUF+32H through OBUF+10H.  The buffer
```

```
*                       thus looks like a television picture with the
*                       vertical hold skewed, so that the frame starts
*                       above the bottom of the screen, leaves off at
*                       the top, and continues at the bottom.

*
*6850 ACIA (UART) EQUATES
*ARBITRARY 6850 ACIA MEMORY ADDRESSES
*
ACIADR  EQU     $A400           ACIA DATA REGISTER
ACIASR  EQU     $A401           ACIA STATUS REGISTER
ACIACR  EQU     $A401           ACIA CONTROL REGISTER
*
*TRS-80 COLOR COMPUTER INTERRUPT VECTOR
*
INTVEC  EQU     $010D           VECTOR TO INTERRUPT SERVICE ROUTINE

*
*READ CHARACTER FROM INPUT BUFFER
*
INCH:
        JSR     INST            GET INPUT STATUS
        BCC     INCH            BRANCH (WAIT) IF NO CHARACTER AVAILABLE
        DEC     ICNT            REDUCE INPUT BUFFER COUNT BY 1
        LDX     IHEAD           GET CHARACTER FROM HEAD OF INPUT BUFFER
        LDA     ,X
        JSR     INCIPTR         MOVE HEAD POINTER UP 1 WITH WRAPAROUND
        STX     IHEAD
        RTS

*
*RETURN INPUT STATUS (CARRY = 1 IF INPUT DATA AVAILABLE)
*
INST:
        CLC                     CLEAR CARRY, INDICATING BUFFER EMPTY
        TST     ICNT            TEST INPUT BUFFER COUNT
        BEQ     EXINST          BRANCH (EXIT) IF BUFFER EMPTY
        SEC                     SET CARRY TO INDICATE DATA AVAILABLE
EXINST:
        RTS                     RETURN, CARRY INDICATES WHETHER DATA
                                * IS AVAILABLE
*
*WRITE A CHARACTER INTO OUTPUT BUFFER
*
OUTCH:
        PSHS    A               SAVE CHARACTER TO WRITE

        *WAIT UNTIL OUTPUT BUFFER NOT FULL, STORE NEXT CHARACTER
WAITOC:
        JSR     OUTST           GET OUTPUT STATUS
        BCS     WAITOC          BRANCH (WAIT) IF OUTPUT BUFFER FULL
        INC     OCNT            INCREASE OUTPUT BUFFER COUNT BY 1
        LDX     OTAIL           POINT AT NEXT EMPTY BYTE IN BUFFER
        PULS    A               GET CHARACTER
        STA     ,X              STORE CHARACTER AT TAIL OF BUFFER
        JSR     INCOPTR         MOVE TAIL POINTER UP 1
```

```
        STX     OTAIL
        TST     OIE             TEST OUTPUT INTERRUPT EXPECTED FLAG
        BNE     EXWAIT
        JSR     OUTDAT          OUTPUT CHARACTER IMMEDIATELY IF
                                * OUTPUT INTERRUPT NOT EXPECTED
EXWAIT:
        RTS


*
*OUTPUT STATUS (CARRY = 1 IF OUTPUT BUFFER FULL)
*
OUTST:
        LDA     OCNT            GET OUTPUT BUFFER COUNT
        CMPA    #SZOBUF         IS OUTPUT BUFFER FULL?
        SEC                     SET CARRY, INDICATING OUTPUT BUFFER
                                *  FULL
        BEQ     EXOUTS          BRANCH (EXIT) IF OUTPUT BUFFER FULL
        CLC                     INDICATE OUTPUT BUFFER NOT FULL
EXOUTS:
        RTS                     CARRY = 1 IF BUFFER FULL, 0 IF NOT

*
*INITIALIZE 6850 ACIA, INTERRUPT SYSTEM
*
INIT:
        *
        *DISABLE INTERRUPTS DURING INITIALIZATION BUT SAVE
        *  PREVIOUS STATE OF INTERRUPT FLAG
        *
        PSHS    CC              SAVE CURRENT FLAGS (PARTICULARLY I FLAG)
        SEI                     DISABLE INTERRUPTS DURING
                                * INITIALIZATION
        *
        *INITIALIZE TRS-80 COLOR COMPUTER INTERRUPT VECTOR
        *
        LDX     INTVEC          GET CURRENT INTERRUPT VECTOR
        STX     NEXTSR          SAVE IT AS ADDRESS OF NEXT SERVICE
                                * ROUTINE
        LDX     #IOSRVC         GET ADDRESS OF OUR SERVICE ROUTINE
        STX     INTVEC          SAVE IT AS INTERRUPT VECTOR
        *
        *INITIALIZE BUFFER COUNTERS AND POINTERS, INTERRUPT FLAG
        *
        CLR     ICNT            INPUT BUFFER EMPTY
        CLR     OCNT            OUTPUT BUFFER EMPTY
        CLR     OIE             INDICATE NO OUTPUT INTERRUPT EXPECTED
        LDX     #IBUF           INPUT HEAD/TAIL POINT TO BASE
        STX     IHEAD            ADDRESS OF INPUT BUFFER
        STX     ITAIL
        LDX     #OBUF           OUTPUT HEAD/TAIL POINT TO BASE
        STX     OHEAD            ADDRESS OF OUTPUT BUFFER
        STX     OTAIL
        *
        *INITIALIZE 6850 ACIA
        *
```

```
        LDA     #%00000011      MASTER RESET 6850 ACIA (NOTE IT
        STA     ACIACR            HAS NO RESET INPUT)
        LDA     #%10010001
        STA     ACIACR          SET ACIA OPERATING MODE
                                *BIT 7 = 1 TO ENABLE INPUT INTERRUPTS
                                *BITS 6,5 = 0 TO DISABLE OUTPUT
                                *  INTERRUPTS
                                *BITS 4,3,2 = 100 FOR 8 DATA BITS,
                                *  2 STOP BITS
                                *BITS 1,0 = 01 FOR DIVIDE BY 16 CLOCK
                                *  MODE
        PULS    CC              RESTORE FLAGS (THIS REENABLES INTERRUPTS
                                *  IF THEY WERE ENABLED WHEN INIT WAS
                                *  CALLED)
        RTS


*
*INPUT/OUTPUT INTERRUPT SERVICE ROUTINE
*
IOSRVC:
        *
        *GET ACIA STATUS: BIT 0 = 1 IF AN INPUT INTERRUPT,
        *  BIT 1 = 1 IF AN OUTPUT INTERRUPT
        *
        LDA     ACIASR
        LSRA                    MOVE BIT 0 TO CARRY
        BCS     RDHDLR          BRANCH IF AN INPUT INTERRUPT
        LSRA                    MOVE BIT 1 TO CARRY
        BCS     WRHDLR          BRANCH IF AN OUTPUT INTERRUPT
        *
        *INTERRUPT WAS NOT OURS, TRY NEXT SOURCE
        *
        JMP     [NEXTSR]        INTERRUPT IS FROM ANOTHER SOURCE
*
*SERVICE INPUT INTERRUPTS
*
RDHDLR:
        LDA     ACIADR          READ DATA FROM ACIA
        LDB     ICNT            ANY ROOM IN INPUT BUFFER?
        CMPB    #SZIBUF
        BEQ     EXITRH          BRANCH (EXIT) IF NO ROOM IN INPUT BUFFER
        INC     ICNT            INCREMENT INPUT BUFFER COUNT
        LDX     ITAIL           STORE CHARACTER AT TAIL OF INPUT BUFFER
        STA     ,X
        JSR     INCIPTR         INCREMENT TAIL POINTER WITH WRAPAROUND
        STX     ITAIL
EXITRH:
        RTI

*
*OUTPUT (WRITE) INTERRUPT HANDLER
*
WRHDLR:
        TST     OCNT            TEST OUTPUT BUFFER COUNT
        BEQ     NODATA          BRANCH IF NO DATA TO TRANSMIT
```

```
        JSR     OUTDAT          ELSE OUTPUT DATA TO 6850 ACIA
        RTI
*
*IF AN OUTPUT INTERRUPT OCCURS WHEN NO DATA IS AVAILABLE,
* WE MUST DISABLE IT TO AVOID AN ENDLESS LOOP. WHEN THE NEXT CHARACTER
* IS READY, IT MUST BE SENT IMMEDIATELY SINCE NO INTERRUPT WILL
* OCCUR.  THIS STATE IN WHICH AN OUTPUT INTERRUPT HAS OCCURRED
* BUT HAS NOT BEEN SERVICED IS INDICATED BY CLEARING OIE (OUTPUT
* INTERRUPT EXPECTED FLAG).
*
NODATA:
        CLR     OIE             DO NOT EXPECT AN INTERRUPT
        RTI


************************************
*ROUTINE: OUTDAT
*PURPOSE: SEND CHARACTER TO 6850 ACIA FROM THE OUTPUT BUFFER
*ENTRY: X CONTAINS THE ADDRESS OF THE CHARACTER TO SEND
*EXIT:  NONE
*REGISTERS USED: A,X,CC
************************************

OUTDAT:
        LDA     ACIASR
        AND     #%00000010      IS ACIA OUTPUT REGISTER EMPTY?
        BEQ     OUTDAT          BRANCH (WAIT) IF REGISTER NOT EMPTY
        LDX     OHEAD           GET HEAD OF OUTPUT BUFFER
        LDA     ,X              GET CHARACTER FROM HEAD OF BUFFER
        STA     ACIADR          SEND DATA TO ACIA
        JSR     INCOPTR         INCREMENT POINTER WITH WRAPAROUND
        DEC     OCNT            DECREMENT OUTPUT BUFFER COUNTER
        LDA     #%10110001
        STA     ACIACR          ENABLE 6850 INPUT AND OUTPUT INTERRUPTS
                                * 8 DATA BITS, 2 STOP BITS, DIVIDE BY
                                * 16 CLOCK
        LDA     #$FF
        STA     OIE             INDICATE OUTPUT INTERRUPTS ENABLED
        RTS

************************************
*ROUTINE: INCIPTR
*PURPOSE: INCREMENT POINTER INTO INPUT
*       BUFFER WITH WRAPAROUND
*ENTRY: X = POINTER
*EXIT:  X = POINTER INCREMENTED WITH WRAPAROUND
*REGISTERS USED: CC
************************************

INCIPTR:
        LEAX    1,X             INCREMENT POINTER BY 1
        CMPX    #EIBUF          COMPARE POINTER, END OF BUFFER
        BNE     RETINC          BRANCH IF NOT EQUAL
        LDX     #IBUF           IF EQUAL, SET POINTER BACK TO BASE OF
                                * BUFFER
RETINC:
```

```
        RTS


*************************************
*ROUTINE: INCOPTR
*PURPOSE: INCREMENT POINTER INTO OUTPUT
*       BUFFER WITH WRAPAROUND
*ENTRY: X = POINTER
*EXIT:  X = POINTER INCREMENTED WITH WRAPAROUND
*REGISTERS USED: CC
*************************************

INCOPTR:
        LEAX    1,X             INCREMENT POINTER BY 1
        CMPX    #EOBUF          COMPARE POINTER, END OF BUFFER
        BNE     RETONC          BRANCH IF NOT EQUAL
        LDX     #OBUF           IF EQUAL, SET POINTER BACK TO BASE OF
                                * BUFFER
RETONC:
        RTS


*DATA SECTION
IHEAD:  RMB     2               POINTER TO OLDEST CHARACTER IN INPUT
                                * BUFFER (NEXT CHARACTER TO READ)
ITAIL:  RMB     2               POINTER TO NEWEST CHARACTER IN INPUT
                                * BUFFER (LAST CHARACTER READ)
ICNT:   RMB     1               NUMBER OF CHARACTERS IN INPUT BUFFER
OHEAD:  RMB     2               POINTER TO OLDEST CHARACTER IN OUTPUT
                                * BUFFER (LAST CHARACTER WRITTEN)
OTAIL:  RMB     2               POINTER TO NEWEST CHARACTER IN OUTPUT
                                * BUFFER (NEXT CHARACTER TO SEND)
OCNT:   RMB     1               NUMBER OF CHARACTERS IN OUTPUT BUFFER
SZIBUF  EQU     10              SIZE OF INPUT BUFFER
IBUF:   RMB     SZIBUF          INPUT BUFFER
EIBUF   EQU     $               END OF INPUT BUFFER
SZOBUF  EQU     10              SIZE OF OUTPUT BUFFER
OBUF:   RMB     SZOBUF          OUTPUT BUFFER
EOBUF   EQU     $               END OF OUTPUT BUFFER
OIE:    RMB     1               OUTPUT INTERRUPT EXPECTED
                                * ( 0 = NO INTERRUPT EXPECTED,
                                *  FF = INTERRUPT EXPECTED)
NEXTSR: RMB     2               ADDRESS OF NEXT INTERRUPT SERVICE
                                *  ROUTINE
*
*       SAMPLE EXECUTION:
*
*CHARACTER EQUATES
ESCAPE  EQU     $1B             ASCII ESCAPE CHARACTER
TESTCH  EQU     'A              TEST CHARACTER = A

SC9C:
        JSR     INIT            INITIALIZE 6850 ACIA, INTERRUPT SYSTEM
        *
        *SIMPLE EXAMPLE - READ AND ECHO CHARACTERS
        * UNTIL AN ESC IS RECEIVED
```

```
        *
LOOP:
        JSR     INCH            READ CHARACTER
        PSHS    A
        JSR     OUTCH           ECHO CHARACTER
        PULS    A
        CMPA    #ESCAPE         IS CHARACTER AN ESCAPE?
        BNE     LOOP            STAY IN LOOP IF NOT
        *
        *AN ASYNCHRONOUS EXAMPLE
        * OUTPUT "A" TO CONSOLE CONTINUOUSLY BUT ALSO LOOK AT
        * INPUT SIDE, READING AND ECHOING ANY INPUT CHARACTERS.
ASYNLP:
        *OUTPUT AN "A" IF OUTPUT IS NOT BUSY
        JSR     OUTST           IS OUTPUT BUSY?
        BCC     ASYNLP          JUMP IF IT IS
        LDA     #TESTCH
        JSR     OUTCH           OUTPUT CHARACTER
        *
        *CHECK INPUT PORT
        *ECHO CHARACTER IF ONE IS AVAILABLE
        *EXIT ON ESCAPE CHARACTER
        *
        JSR     INST            IS INPUT DATA AVAILABLE?
        BCS     ASYNLP          JUMP IF NOT (SEND ANOTHER "A")
        JSR     INCH            GET CHARACTER
        CMPA    #ESCAPE         IS IT AN ESCAPE CHARACTER?
        BEQ     DONE            BRANCH IF IT IS
        JSR     OUTCH           ELSE ECHO CHARACTER
        BRA     ASYNLP          AND CONTINUE

DONE:
        BRA     SC9C            REPEAT TEST

        END
```

9D Real-time clock and calendar (CLOCK)

Maintains a time-of-day 24-hour clock and a calendar based on a real-time clock interrupt generated from a 6840 Programmable Timer Module (PTM). Consists of the following subroutines:

1. CLOCK returns the base address of the clock variables.

2. ICLK initializes the clock interrupt and the clock variables.

3. CLKINT updates the clock after each interrupt (assumed to be spaced one tick apart).

Procedure

1. CLOCK loads the base address of the clock variables into register X. The clock variables are stored in the following order (lowest address first): ticks, seconds, minutes, hours, days, months, less significant byte of year, more significant byte of year.

2. ICLK initializes the 6840 PIT, the interrupt system, and the clock variables. The arbitrary starting time is 00:00.00 (12 a.m.) 1 January 1980. A real application would clearly require outside intervention to load or change the clock.

3. CLKINT decrements the remaining tick count by 1 and updates the rest of the clock variables if necessary. Of course, the number of seconds and minutes must be less than 60 and the number of hours must be less than 24. The day of the month must be less than or equal to the last day for the current month; an array of the last days of each month begins at address LASTDY.

If the month is February (i.e. month 2), the program checks if the current year is a leap year. This involves determining whether the two least significant bits of memory location YEAR are both 0s. If the current year is a leap year, the last day of February is the 29th, not the 28th.

The month number may not exceed 12 (December) or a Carry to the year number is necessary. The program must reinitialize the variables properly when carries occur; i.e. to DTICK; seconds, minutes, and hours to 0; day and month to 1 (meaning the first day and January, respectively).

G. J. Lipovski has described an alternative approach using a 60 Hz clock input and all three 6840 timers. See pp. 340–341 of his book titled *Microcomputer Interfacing* (Lexington Books, Lexington, MA, 1980).

Entry conditions

1. CLOCK: none
2. ICLK: none
3. CLKINT: none

Exit conditions

1. CLOCK: base address of clock variables in register X
2. ICLK: none
3. CLKINT: none

Examples

These examples assume that the tick rate is DTICK Hz (less than 256 Hz—typical values would be 60 Hz or 100 Hz) and that the clock and calendar are saved in memory locations

TICK	number of ticks remaining before a carry occurs, counted down from DTICK
SEC	seconds (0 – 59)
MIN	minutes (0 – 59)
HOUR	hour of day (0 – 23)
DAY	day of month (1 – 28, 29, 30, or 31, depending on month)
MONTH	month of year (1 – 12 for January through December)
YEAR and YEAR + 1	current year

1. Starting values are 7 March 1986, 11:59.59 p.m. and 1 tick left. That is:

(TICK) = 1
(SEC) = 59
(MIN) = 59
(HOUR) = 23
(DAY) = 07
(MONTH) = 03
(YEAR and YEAR+1) = 1986

Result (after the tick): 8 March 1986, 12:00.00 a.m. and DTICK ticks. That is:

(TICK) = DTICK
(SEC) = 0
(MIN) = 0
(HOUR) = 0
(DAY) = 08
(MONTH) = 03
(YEAR and YEAR + 1) = 1986

2. Starting values are 31 December 1986, 11:59.59 p.m. and 1 tick left. That is:

(TICK) = 1
(SEC) = 59
(MIN) = 59
(HOUR) = 23
(DAY) = 31
(MONTH) = 12
(YEAR and YEAR+1) = 1986

Result (after the tick): 1 January 1987, 12:00.00 a.m. and DTICK ticks. That is:

(TICK) = DTICK
(SEC) = 0
(MIN) = 0
(HOUR) = 0
(DAY) = 1
(MONTH) = 1
(YEAR and YEAR + 1) = 1987

Registers used

1. CLOCK: CC, X
2. ICLK: A, B, CC, X, Y
3. CLKINT: none

Execution time

1. CLOCK: 8 cycles
2. ICLK: 115 cycles
3. CLKINT: 59 cycles if only TICK must be decremented, 244 cycles

maximum if changing to a new year. These times include the 21 cycles required by the CPU to respond to an interrupt.

Program size 190 bytes

Data memory required 8 bytes anywhere in RAM for the clock variables (starting at address CLKVAR)

```
*       Title           Real time clock and calendar
*       Name:           CLOCK
*
*       Purpose:        This program maintains a time of day 24 hour
*                       clock and a calendar based on a real time clock
*                       interrupt from a 6840 programmable timer.
*
*                       CLOCK
*                         Returns base address of clock variables
*                       ICLK
*                         Initializes 6840 timer and clock interrupt
*
*       Entry:          CLOCK
*                         None
*                       ICLK
*                         None
*
*       Exit:           CLOCK
*                         Register X = Base address of time variables
*                       ICLK
*                         None
*
*       Registers Used: A,B,CC,X,Y
*
*       Time:           CLOCK
*                         8 cycles
*                       ICLCK
*                         115 cycles
*                       CLKINT
*                         If decrementing tick only, 59 cycles
*                         Maximum if changing to a new year, 244
*                         cycles
*                         These include the 21 cycles required for the
*                         processor to respond to an interrupt.
*
*       Size:           Program   190 bytes
*                       Data      8 bytes
*
*  6840 PROGRAMMABLE TIMER MODULE (PTM)
*
*  INITIALIZE TIMER 2 OF 6840 PTM AS 50 HZ SQUARE WAVE
*    GENERATOR FOR USE IN TIME-OF-DAY CLOCK.
```

```
*  TIMER GENERATES INTERRUPT AT END OF EACH 10 MS
*    INTERVAL (EVERY HALF-CYCLE)
*  WE ASSUME A 1 MHZ CLOCK INTO THE 6840, SO THAT A COUNTER VALUE
*    OF 1,000,000/100-1 = 9,999 (270F HEX) IS NEEDED TO GENERATE
*    A 50 HZ SQUARE WAVE

*ARBITRARY MEMORY ADDRESSES FOR 6840 PTM
PTMC13    EQU   $A800          CONTROL REGISTERS 1 AND 3
PTMCR2    EQU   $A801          CONTROL REGISTER 2
PTMT1H    EQU   $A802          TIMER 1, MORE SIGNIFICANT BYTE
PTMT1L    EQU   $A803          TIMER 1, LESS SIGNIFICANT BYTE
PTMT2H    EQU   $A804          TIMER 2, MORE SIGNIFICANT BYTE
PTMT2L    EQU   $A805          TIMER 2, LESS SIGNIFICANT BYTE
PTMT3H    EQU   $A806          TIMER 3, MORE SIGNIFICANT BYTE
PTMT3L    EQU   $A807          TIMER 3, LESS SIGNIFICANT BYTE
PTMSR     EQU   $A801          STATUS REGISTER
PTMT2C    EQU   $A804          TIMER 2 COUNTER

*6840 PTM MODE BYTE, COUNTER VALUE
PTMMOD    EQU   %01000000      *BIT 0 = 0 TO ACCESS CR3
                               *BIT 1 = 0 TO USE ENABLE CLOCK
                               *BIT 2 = 0 FOR 16-BIT COUNT MODE
                               *BITS 3,5 = 00 FOR CONTINUOUS COUNTING
                               *BIT 4 = 0 FOR ACTIVATE WHEN LATCHES
                               *  WRITTEN
                               *BIT 6 = 1 TO ENABLE INTERRUPT
                               *BIT 7 = 0 TO DISABLE OUTPUT
PTMCNT    EQU   9999           COUNTER VALUE = 9999

*
*DEFAULT TICK VALUE (100 HZ REAL-TIME CLOCK)
*
DTICK     EQU   100            DEFAULT TICK VALUE


*RETURN BASE ADDRESS OF CLOCK VARIABLES
CLOCK:
          LDX   #CLKVAR        GET BASE ADDRESS OF CLOCK VARIABLES
          RTS

*
*INITIALIZE 6840 PTM TO PRODUCE REGULAR CLOCK INTERRUPTS
*OPERATE TIMER 2 CONTINUOUSLY, PRODUCING AN INTERRUPT EVERY
*  100 MS
*
ICLK:
          LDA   #%00000001
          STA   PTMCR2         ADDRESS CONTROL REGISTER 1
          STA   PTMC13         RESET TIMERS
          CLR   PTMC13         ALLOW TIMERS TO OPERATE
          LDD   #0             CLEAR COUNTERS 1,3
          STD   PTMT1H
          STD   PTMT3H
          LDA   #PTMMOD        SET TIMER 2'S OPERATING MODE
          STA   PTMCR2
          LDD   #PTMCNT        PUT COUNT IN TIMER 2
```

```
        STD   PTMT2H          START TIMER 2
        *
        *INITIALIZE CLOCK VARIABLES TO ARBITRARY VALUE
        *JANUARY 1, 1980 00:00.00 (12 A.M.)
        *A REAL CLOCK WOULD NEED OUTSIDE INTERVENTION
        * TO SET OR CHANGE VALUES
        *
        LDX   #TICK
        LDA   #DTICK
        STA   ,X              INITIALIZE TICKS
        CLRA
        STA   1,X             SECOND = 0
        STA   2,X             MINUTE = 0
        STA   3,X             HOUR = 0
        LDA   #1              A = 1
        STA   4,X             DAY = 1 (FIRST)
        STA   5,X             MONTH = 1 (JANUARY)
        LDY   #1980
        STY   6,X             YEAR = 1980
        CLI                   ENABLE INTERRUPTS
        RTS


*SERVICE CLOCK INTERRUPT
CLKINT:
        LDA     PTMSR         CLEAR INTERRUPT BY READING STATUS
        LDA     PTMT2C          AND THEN COUNTER
        LDX     #CLKVAR
        DEC     TICKIDX,X     SUBTRACT 1 FROM TICK COUNT
        BNE     EXITCLK       JUMP IF TICK COUNT NOT ZERO
        LDA     #DTICK        SET TICK COUNT BACK TO DEFAULT
        STA     TICKIDX,X

        *SAVE REMAINING REGISTERS
        CLRA                  0 = DEFAULT FOR SECONDS, MINUTES, HOURS

        *INCREMENT SECONDS
        INC     SECIDX,X      INCREMENT TO NEXT SECOND
        LDA     SECIDX,X
        CMPA    #60           SECONDS = 60?
        BCS     EXITCLK       EXIT IF LESS THAN 60 SECONDS
        CLR     SECIDX,X      ELSE SECONDS = 0

        *INCREMENT MINUTES
        INC     MINIDX,X      INCREMENT TO NEXT MINUTE
        LDA     MINIDX,X
        CMPA    #60           MINUTES = 60?
        BCS     EXITCLK       EXIT IF LESS THAN 60 MINUTES
        CLR     MINIDX,X      ELSE MINUTES = 0

        *INCREMENT HOUR
        INC     HRIDX,X       INCREMENT TO NEXT HOUR
        LDA     HRIDX,X
        CMPA    #24           HOURS = 24?
        BCS     EXITCLK       EXIT IF LESS THAN 24 HOURS
        CLR     HRIDX,X       ELSE HOUR = 0
```

```
        *INCREMENT DAY
        LDA     MTHIDX,X        GET CURRENT MONTH
        LDY     #LASTDY
        LDA     A,Y             GET LAST DAY OF CURRENT MONTH
        INC     DAYIDX,X        INCREMENT DAY
        CMPA    DAYIDX,X        IS IT LAST DAY?
        BCS     EXITCLK         EXIT IF NOT AT END OF MONTH
        *
        *DETERMINE IF THIS IS END OF FEBRUARY IN A LEAP
        * YEAR (YEAR DIVISIBLE BY 4)
        *
        LDA     MTHIDX,X        GET MONTH
        CMPA    #2              IS THIS FEBRUARY?
        BNE     INCMTH          JUMP IF NOT, INCREMENT MONTH
        LDA     YRIDX+1,X       IS IT A LEAP YEAR?
        ANDA    #%00000011
        BNE     INCMTH          JUMP IF NOT
        *
        *FEBRUARY OF A LEAP YEAR HAS 29 DAYS, NOT 28 DAYS
        *
        LDA     DAYIDX,X        GET DAY
        CMPA    #29
        BCS     EXITCLK         EXIT IF NOT 1ST OF MARCH
        *INCREMENT MONTH
INCMTH:
        LDA     #1              DEFAULT IS 1 FOR DAY AND MONTH
        STA     DAYIDX,X        DAY = 1

        LDA     MTHIDX,X
        INC     MTHIDX,X        INCREMENT MONTH
        CMPA    #12             WAS OLD MONTH DECEMBER?
        BCS     EXITCLK         EXIT IF NOT
        LDA     #1              ELSE
                                * CHANGE MONTH TO 1 (JANUARY)
        STA     MTHIDX,X

        *INCREMENT YEAR
        LDD     YRIDX,X         GET YEAR
        ADDD    #1              ADD 1 TO YEAR
        STD     YEAR            STORE NEW YEAR

EXITCLK:
        *RESTORE REGISTERS AND EXIT
        RTI                     RETURN

*ARRAY OF LAST DAYS OF EACH MONTH
LASTDY:
        FCB     31              JANUARY
        FCB     28              FEBRUARY (EXCEPT LEAP YEARS)
        FCB     31              MARCH
        FCB     30              APRIL
        FCB     31              MAY
        FCB     30              JUNE
        FCB     31              JULY
        FCB     31              AUGUST
        FCB     30              SEPTEMBER
```

```
        FCB     31              OCTOBER
        FCB     30              NOVEMBER
        FCB     31              DECEMBER

*CLOCK VARIABLES
CLKVAR:
TICK:   RMB     1               TICKS LEFT IN CURRENT SECOND
SEC:    RMB     1               SECONDS
MIN:    RMB     1               MINUTES
HOUR:   RMB     1               HOURS
DAY:    RMB     1               DAY (1 TO NUMBER OF DAYS IN A MONTH)
MONTH:  RMB     1               MONTH 1=JANUARY .. 12=DECEMBER
YEAR:   RMB     2               YEAR

*
*       SAMPLE EXECUTION
*

*CLOCK VARIABLE INDEXES
TCKIDX  EQU     0               INDEX TO TICK
SECIDX  EQU     1               INDEX TO SECOND
MINIDX  EQU     2               INDEX TO MINUTE
HRIDX   EQU     3               INDEX TO HOUR
DAYIDX  EQU     4               INDEX TO DAY
MTHIDX  EQU     5               INDEX TO MONTH
YRIDX   EQU     6               INDEX TO YEAR
SC9D:
        JSR     ICLK            INITIALIZE CLOCK

        *INITIALIZE CLOCK TO 2/7/86 14:00:00 (2 PM, FEB. 7, 1986)
        JSR     CLOCK           X = ADDRESS OF CLOCK VARIABLES
        CLR     SEC             SECONDS = 0
        CLR     MIN             MINUTES = 0
        LDA     #14             HOUR = 14 (2 PM)
        STA     HOUR
        LDA     #7              DAY = 7
        STA     DAY
        LDA     #2              MONTH = 2 (FEBRUARY)
        STA     MONTH
        LDX     #1986
        STX     YEAR
        *
        *WAIT FOR CLOCK TO BE 2/7/86 14:01:20 (2:01.20 PM, FEB. 7, 1986)
        *
        *NOTE: MUST BE CAREFUL TO EXIT IF CLOCK IS ACCIDENTALLY
        * SET AHEAD.  IF WE CHECK ONLY FOR EQUALITY, WE MIGHT NEVER
        * FIND IT.  THUS WE HAVE >= IN TESTS BELOW, NOT JUST =.
        *
        *WAIT FOR YEAR >= TARGET YEAR
        JSR     CLOCK           X = BASE ADDRESS OF CLOCK VARIABLES
        LDY     TYEAR           Y = YEAR TO WAIT FOR
WAITYR:
        *COMPARE CURRENT YEAR AND TARGET YEAR
        CMPY    YEAR
        BHI     WAITYR          BRANCH IF YEAR NOT >= TARGET YEAR
        *
```

```
        *WAIT FOR REST OF TIME UNITS TO BE GREATER THAN OR EQUAL
        *  TO TARGET VALUES
        *
        LDY     #TARGET         POINT TO TARGET VALUES
        LEAX    MTHIDX,X        POINT TO END OF TIME VALUES
        LDB     NTUNIT          NUMBER OF TIME UNITS IN COMPARISON
        *
        *GET NEXT TARGET VALUE
        *
WTTIM:
        LDA     ,Y+             GET NEXT TARGET VALUE
        *
        *WAIT FOR TIME TO BE GREATER THAN OR EQUAL TO TARGET
        *
WTUNIT:
        CMPA    ,X
        BHI     WTUNIT          BRANCH IF UNIT NOT >= TARGET VALUE
        LEAX    -1,X            PROCEED TO NEXT UNIT
        DECB                    DECREMENT NUMBER OF TIME UNITS
        BNE     WTTIM           CONTINUE UNTIL ALL UNITS CHECKED
        *
        *DONE
        *
HERE:
        BRA     HERE            IT IS NOW TIME OR LATER

*
*TARGET TIME - 2/7/87, 14:01:20 (2:01.20 PM, FEB. 7, 1987)
*
TYEAR:  FDB     1987            TARGET YEAR
NTUNIT: FCB     5               NUMBER OF TIME UNITS IN COMPARISON
TARGET: FCB     2,7,14,1,20     TARGET TIME (MONTH,DAY,HR,MIN,SEC)

        END
```

A 6809 Instruction set summary

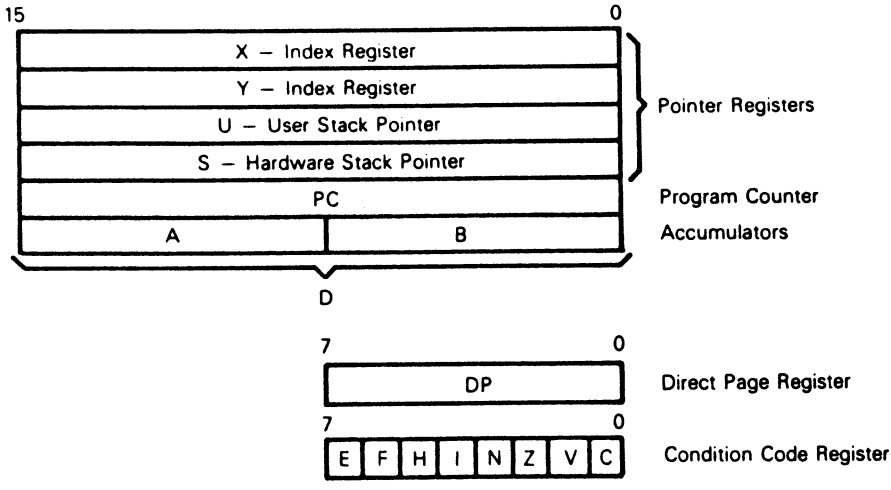


Figure A-1 6809 programming model.

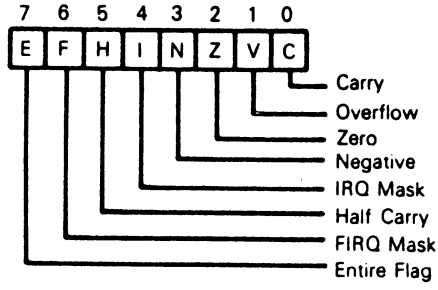


Figure A-2 6809 condition code register.

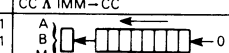
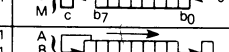
Table A-1 6809 instruction set.

Instruction	Forms	Addressing Modes: Immediate Op	Immediate ~	Immediate #	Direct Op	Direct ~	Direct #	Indexed[1] Op	Indexed[1] ~	Indexed[1] #	Extended Op	Extended ~	Extended #	Inherent Op	Inherent ~	Inherent #	Description	5 H	3 N	2 Z	1 V	0 C
LSL	LSLA													48	2	1	A } ← 0	•	↕	↕	↕	↕
	LSLB													58	2	1	B } ← 0	•	↕	↕	↕	↕
	LSL				08	6	2	68	6+	2+	78	7	3				M } ← 0 (b7 … b0, C)	•	↕	↕	↕	↕
LSR	LSRA													44	2	1	A } 0 →	•	0	↕	•	↕
	LSRB													54	2	1	B } 0 →	•	0	↕	•	↕
	LSR				04	6	2	64	6+	2+	74		3				M } 0 → (b7 … b0, C)	•	0	↕	•	↕
MUL														3D	11	1	A × B → D (Unsigned)	•	•	↕	•	9
NEG	NEGA													40	2	1	$\overline{A}$ + 1 → A	8	↕	↕	↕	↕
	NEGB													50	2	1	$\overline{B}$ + 1 → B	8	↕	↕	↕	↕
	NEG				00	6	2	60	6+	2+	70	7	3				$\overline{M}$ + 1 → M	8	↕	↕	↕	↕
NOP														12	2	1	No Operation	•	•	•	•	•
OR	ORA	8A	2	2	9A	4	2	AA	4+	2+	BA	5	3				A ∨ M → A	•	↕	↕	0	•
	ORB	CA	2	2	DA	4	2	EA	+	2+	FA	5	3				B ∨ M → B	•	↕	↕	0	•
	ORCC	1A	3	2													CC ∨ IMM → CC					7
PSH	PSHS	34	5+[4]	2													Push Registers on S Stack	•	•	•	•	•
	PSHU	36	5+[4]	2													Push Registers on U Stack	•	•	•	•	•
PUL	PULS	35	5+[4]	2													Pull Registers from S Stack	•	•	•	•	•
	PULU	37	5+[4]	2													Pull Registers from U Stack	•	•	•	•	•
ROL	ROLA													49	2	1	A } ← C ← (b7 … b0)	•	↕	↕	↕	↕
	ROLB													59	2	1	B } ← C ← (b7 … b0)	•	↕	↕	↕	↕
	ROL				09	6	2	69	6+	2+	79	7	3				M } ← C ← (b7 … b0)	•	↕	↕	↕	↕
ROR	RORA													46	2	1	A } → C → (b7 … b0)	•	↕	↕	•	↕
	RORB													56	2	1	B } → C → (b7 … b0)	•	↕	↕	•	↕
	ROR				06	6	2	66	6+	2+	76	7	3				M } → C → (b7 … b0)	•	↕	↕	•	↕
RTI														3B	6/15	1	Return From Interrupt					7
RTS														39	5	1	Return from Subroutine	•	•	•	•	•
SBC	SBCA	82	2	2	92	4	2	A2	4+	2+	B2	5	3				A − M − C → A	8	↕	↕	↕	↕
	SBCB	C2	2	2	D2	4	2	E2	4+	2+	F2	5	3				B − M − C → B	8	↕	↕	↕	↕
SEX														1D	2	1	Sign Extend B into A	•	↕	↕	0	•
ST	STA				97	4	2	A7	4+	2+	B7	5	3				A → M	•	↕	↕	0	•
	STB				D7	4	2	E7	4+	2+	F7	5	3				B → M	•	↕	↕	0	•
	STD				DD	5	2	ED	5+	2+	FD	6	3				D → M M + 1	•	↕	↕	0	•
	STS				10 DF	6	3	10 EF	6+	3+	10 FF	7	4				S → M M + 1	•	↕	↕	0	•
	STU				DF	5	2	EF	5+	2+	FF	6	3				U → M M + 1	•	↕	↕	0	•
	STX				9F	5	2	AF	5+	2+	BF	6	3				X → M M + 1	•	↕	↕	0	•
	STY				10 9F	6	3	10 AF	6+	3+	10 BF	7	4				Y → M M + 1	•	↕	↕	0	•
SUB	SUBA	80	2	2	90	4	2	A0	4+	2+	B0	5	3				A − M → A	8	↕	↕	↕	↕
	SUBB	C0	2	2	D0	4	2	E0	4+	2+	F0	5	3				B − M → B	8	↕	↕	↕	↕
	SUBD	83	4	3	93	6	2	A3	6+	2+	B3	7	3				D − M M + 1 → D	•	↕	↕	↕	↕
SWI	SWI[6]													3F	19	1	Software Interrupt 1	•	•	•	•	•
	SWI2[6]													10 3F	20	2	Software Interrupt 2	•	•	•	•	•
	SWI3[6]													11 3F	20	1	Software Interrupt 3	•	•	•	•	•
SYNC														13	≥4	1	Synchronize to Interrupt	•	•	•	•	•
TFR	R1, R2	1F	6	2													R1 → R2[2]	•	•	•	•	•
TST	TSTA													4D	2	1	Test A	•	↕	↕	0	•
	TSTB													5D	2	1	Test B	•	↕	↕	0	•
	TST				0D	6	2	6D	6+	2+	7D	7	3				Test M	•	↕	↕	0	•

Notes:

1. This column gives a base cycle and byte count. To obtain total count, add the values obtained from the INDEXED ADDRESSING MODE table, in Appendix F.
2. R1 and R2 may be any pair of 8 bit or any pair of 16 bit registers.
   The 8 bit registers are: A, B, CC, DP.
   The 16 bit registers are: X, Y, U, S, D, PC
3. EA is the effective address.
4. The PSH and PUL instructions require 5 cycles plus 1 cycle for each **byte** pushed or pulled.
5. 5(6) means: 5 cycles if branch not taken, 6 cycles if taken (Branch instructions).
6. SWI sets I and F bits. SWI2 and SWI3 do not affect I and F.
7. Conditions Codes set as a direct result of the instruction.
8. Value of half-carry flag is undefined.
9. Special Case − Carry set if b7 is SET.

Table A-1 6809 instruction set (continued).

Instruction	Forms	Addressing Modes: Immediate Op	Immediate ~	Immediate #	Direct Op	Direct ~	Direct #	Indexed Op	Indexed ~	Indexed #	Extended Op	Extended ~	Extended #	Inherent Op	Inherent ~	Inherent #	Description	5 H	3 N	2 Z	1 V	0 C
ABX														3A	3	1	B + X → X (Unsigned)	•	•	•	•	•
ADC	ADCA	89	2	2	99	4	2	A9	4+	2+	B9	5	3				A + M + C → A	↕	↕	↕	↕	↕
	ADCB	C9	2	2	D9	4	2	E9	4+	2+	F9	5	3				B + M + C → B	↕	↕	↕	↕	↕
ADD	ADDA	8B	2	2	9B	4	2	AB	4+	2+	BB	5	3				A + M → A	↕	↕	↕	↕	↕
	ADDB	CB	2	2	DB	4	2	EB	4+	2+	FB	5	3				B + M → B	↕	↕	↕	↕	↕
	ADDD	C3	4	3	D3	6	2	E3	6+	2+	F3	7	3				D + M:M + 1 → D	•	↕	↕	↕	↕
AND	ANDA	84	2	2	94	4	2	A4	4+	2+	B4	5	3				A ∧ M → A	•	↕	↕	0	•
	ANDB	C4	2	2	D4	4	2	E4	4+	2+	F4	5	3				B ∧ M → B	•	↕	↕	0	•
	ANDCC	1C	3	2													CC ∧ IMM → CC					7
ASL	ASLA													48	2	1		8	↕	↕	↕	↕
	ASLB													58	2	1		8	↕	↕	↕	↕
	ASL				08	6	2	68	6+	2+	78	7	3					8	↕	↕	↕	↕
ASR	ASRB													47	2	1		8	↕	↕	•	↕
	ASR													57	2	1		8	↕	↕	•	↕
	ASR				07	6	2	67	6+	2+	77	7	3					8	↕	↕	•	↕
BIT	BITA	85	2	2	95	4	2	A5	4+	2+	B5	5	3				Bit Test A (M ∧ A)	•	↕	↕	0	•
	BITB	C5	2	2	D5	4	2	E5	4+	2+	F5	5	3				Bit Test B (M ∧ B)	•	↕	↕	0	•
CLR	CLRA													4F	2	1	0 → A	•	0	1	0	0
	CLRB													5F	2	1	0 → B	•	0	1	0	0
	CLR				0F	6	2	6F	6+	2+	7F	7	3				0 → M	•	0	1	0	0
CMP	CMPA	81	2	2	91	4	2	A1	4+	2+	B1	5	3				Compare M from A	8	↕	↕	↕	↕
	CMPB	C1	2	2	D1	4	2	E1	4+	2+	F1	5	3				Compare M from B	8	↕	↕	↕	↕
	CMPD	10	5	4	10	7	3	10	7+	3+	10	8	4				Compare M:M + 1 from D	•	↕	↕	↕	↕
		83			93			A3			B3											
	CMPS	11	5	4	11	7	3	11	7+	3+	11	8	4				Compare M:M + 1 from S	•	↕	↕	↕	↕
		8C			9C			AC			BC											
	CMPU	11	5	4	11	7	3	11	7+	3+	11	8	4				Compare M:M + 1 from U	•	↕	↕	↕	↕
		83			93			A3			B3											
	CMPX	8C	4	3	9C	6	2	AC	6+	2+	BC	7	3				Compare M:M + 1 from X	•	↕	↕	↕	↕
	CMPY	10	5	4	10	7	3	10	7+	3+	10	8	4				Compare M:M + 1 from Y	•	↕	↕	↕	↕
		8C			9C			AC			BC											
COM	COMA													43	2	1	$\overline{A} \rightarrow A$	•	↕	↕	0	1
	COMB													53	2	1	$\overline{B} \rightarrow B$	•	↕	↕	0	1
	COM				03	6	2	63	6+	2+	73	7	3				$\overline{M} \rightarrow M$	•	↕	↕	0	1
CWAI		3C	≥20	2													CC ∧ IMM → CC Wait for Interrupt					7
DAA														19	2	1	Decimal Adjust A	•	↕	↕	0	↕
DEC	DECA													4A	2	1	A − 1 → A	•	↕	↕	↕	•
	DECB													5A	2	1	B − 1 → B	•	↕	↕	↕	•
	DEC				0A	6	2	6A	6+	2+	7A	7	3				M − 1 → M	•	↕	↕	↕	•
EOR	EORA	88	2	2	98	4	2	A8	4+	2+	B8	5	3				A ⊻ M → A	•	↕	↕	0	•
	EORB	C8	2	2	D8	4	2	E8	4+	2+	F8	5	3				B ⊻ M → B	•	↕	↕	0	•
EXG	R1, R2	1E	8	2													R1 → R2[2]	•	•	•	•	•
INC	INCA													4C	2	1	A + 1 → A	•	↕	↕	↕	•
	INCB													5C	2	1	B + 1 → B	•	↕	↕	↕	•
	INC				0C	6	2	6C	6+	2+	7C	7	3				M + 1 → M	•	↕	↕	↕	•
JMP					0E	3	2	6E	3+	2+	7E	4	3				EA[3] → PC	•	•	•	•	•
JSR					9D	7	2	AD	7+	2+	BD	8	3				Jump to Subroutine	•	•	•	•	•
LD	LDA	86	2	2	96	4	2	A6	4+	2+	B6	5	3				M → A	•	↕	↕	0	•
	LDB	C6	2	2	D6	4	2	E6	4+	2+	F6	5	3				M → B	•	↕	↕	0	•
	LDD	CC	3	3	DC	5	2	EC	5+	2+	FC	6	3				M:M + 1 → D	•	↕	↕	0	•
	LDS	10	4	4	10	6	3	10	6+	3+	10	7	4				M:M + 1 → S	•	↕	↕	0	•
		CE			DE			EE			FE											
	LDU	CE	3	3	DE	5	2	EE	5+	2+	FE	6	3				M:M + 1 → U	•	↕	↕	0	•
	LDX	8E	3	3	9E	5	2	AE	5+	2+	BE	6	3				M:M + 1 → X	•	↕	↕	0	•
	LDY	10	4	4	10	6	3	10	6+	3+	10	7	4				M:M + 1 → Y	•	↕	↕	0	•
		8E			9E			AE			BE											
LEA	LEAS							32	4+	2+							EA[3] → S	•	•	•	•	•
	LEAU							33	4+	2+							EA[3] → U	•	•	•	•	•
	LEAX							30	4+	2+							EA[3] → X	•	•	↕	•	•
	LEAY							31	4+	2+							EA[3] → Y	•	•	↕	•	•

Legend:

- OP Operation Code (Hexadecimal)
- ~ Number of MPU Cycles
- # Number of Program Bytes
- + Arithmetic Plus
- − Arithmetic Minus
- • Multiply
- $\overline{M}$ Complement of M
- → Transfer Into
- H Half-carry (from bit 3)
- N Negative (sign bit)
- Z Zero (Reset)
- V Overflow, 2's complement
- C Carry from ALU
- ↕ Test and set if true, cleared otherwise
- • Not Affected
- CC Condition Code Register
- : Concatenation
- V Logical or
- ∧ Logical and
- ⊻ Logical Exclusive or

Table A-2 6809 indexed addressing modes.

Type	Forms	Non Indirect				Indirect			
		Assembler Form	Postbyte OP Code	× ~	+ #	Assembler Form	Postbyte OP Code	+ ~	+ #
Constant Offset From R (twos complement offset)	No Offset	,R	1RR00100	0	0	[,R]	1RR10100	3	0
	5 Bit Offset	n, R	0RRnnnnn	1	0	defaults to 8-bit			
	8 Bit Offset	n, R	1RR01000	1	1	[n, R]	1RR11000	4	1
	16 Bit Offset	n, R	1RR01001	4	2	[n, R]	1RR11001	7	2
Accumulator Offset From R (twos complement offset)	A — Register Offset	A, R	1RR00110	1	0	[A, R]	1RR10110	4	0
	B — Register Offset	B, R	1RR00101	1	0	[B, R]	1RR10101	4	0
	D — Register Offset	D, R	1RR01011	4	0	[D, R]	1RR11011	7	0
Auto Increment/Decrement R	Increment By 1	,R+	1RR00000	2	0	not allowed			
	Increment By 2	,R++	1RR00001	3	0	[,R++]	1RR10001	6	0
	Decrement By 1	,-R	1RR00010	2	0	not allowed			
	Decrement By 2	,--R	1RR00011	3	0	[,--R]	1RR10011	6	0
Constant Offset From PC (twos complement offset)	8 Bit Offset	n, PCR	1XX01100	1	1	[n, PCR]	1XX11100	4	1
	16 Bit Offset	n, PCR	1XX01101	5	2	[n, PCR]	1XX11101	8	2
Extended Indirect	16 Bit Address	—	—	—	—	[n]	10011111	5	2

R = X, Y, U or S X = 00 Y = 01
X = Don't Care U = 10 S = 11

$\overset{+}{\sim}$ and $\overset{+}{\#}$ Indicate the number of additional cycles and bytes for the particular variation.

Table A-3 6809 interrupt vector locations.

Interrupt Description	Vector Location	
	MS Byte	LS Byte
Reset ($\overline{\text{RESET}}$)	FFFE	FFFF
Non-Maskable Interrupt ($\overline{\text{NMI}}$)	FFFC	FFFD
Software Interrupt (SWI)	FFFA	FFFB
Interrupt Request ($\overline{\text{IRQ}}$)	FFF8	FFF9
Fast Interrupt Request ($\overline{\text{FIRQ}}$)	FFF6	FFF7
Software Interrupt 2 (SWI2)	FFF4	FFF5
Software Interrupt 3 (SWI3)	FFF2	FFF3
Reserved	FFF0	FFF1

B Programming reference for the 6821 PIA device

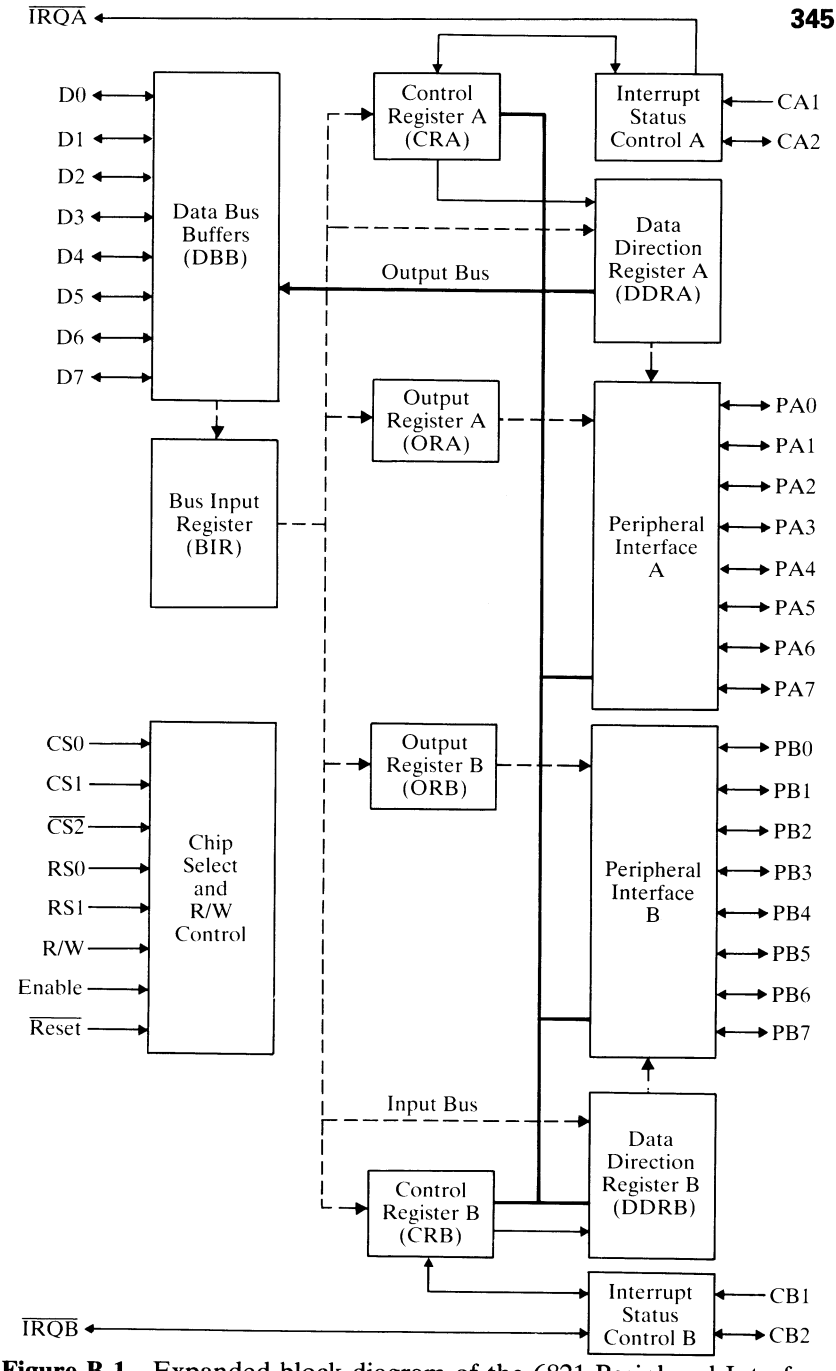


Figure B-1 Expanded block diagram of the 6821 Peripheral Interface Adapter (PIA).

Table B-1 Internal addressing for the 6821 PIA.

RS1	RS0	Control Register Bit CRA-2	Control Register Bit CRB-2	Location Selected
0	0	1	X	Peripheral Register A
0	0	0	X	Data Direction Register A
0	1	X	X	Control Register A
1	0	X	1	Peripheral Register B
1	0	X	0	Data Direction Register B
1	1	X	X	Control Register B

X = Don't Care

Table B-2 6821 control register formats.

	7	6	5	4	3	2	1	0
CRA	IRQA1	IRQA2	CA2 Control			DDRA Access	CA1 Control	
	7	6	5	4	3	2	1	0
CRB	IRQB1	IRQB2	CB2 Control			DDRB Access	CB1 Control	

Table B-3 Control of interrupt inputs CA1 and CB1.

CRA-1 (CRB-1)	CRA-0 (CRB-0)	Interrupt Input CA1 (CB1)	Interrupt Flag CRA-7 (CRB-7)	MPU Interrupt Request $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$)
0	0	↓ Active	Set high on ↓ of CA1 (CB1)	Disabled – $\overline{\text{IRQ}}$ remains high
0	1	↓ Active	Set high on ↓ of CA1 (CB1)	Goes low when the interrupt flag bit CRA-7 (CRB-7) goes high
1	0	↑ Active	Set high on ↑ of CA1 (CB1)	Disabled – $\overline{\text{IRQ}}$ remains high
1	1	↑ Active	Set high on ↑ of CA1 (CB1)	Goes low when the interrupt flag bit CRA-7 (CRB-7) goes high

Notes: 1. ↑ indicates positive transition (low to high)
2. ↓ indicates negative transition (high to low)
3. The interrupt flag bit CRA-7 is cleared by an MPU Read of the A Data Register, and CRB-7 is cleared by an MPU Read of the B Data Register.
4. If CRA-0 (CRB-0) is low when an interrupt occurs (interrupt disabled) and is later brought high, $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$) occurs after CRA-0 (CRB-0) is written to a "one"

Table B-4 Control of CA2 and CB2 as interrupt inputs. CRA-5 (CRB-5) is LOW.

CRA-5 (CRB-5)	CRA-4 (CRB-4)	CRA-3 (CRA-3)	Interrupt Input CA2 (CB2)	Interrupt Flag CRA-6 (CRB-6)	MPU Interrupt Request $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$)
0	0	0	↓ Active	Set high on ↓ of CA2 (CB1)	Disabled – $\overline{\text{IRQ}}$ remains high
0	0	1	↓ Active	Set high on ↓ of CA2 (CB2)	Goes low when the interrupt flag bit CRA-6 (CRB-6) goes high
0	1	0	↑ Active	Set high on ↑ of CA2 (CB2)	Disabled – $\overline{\text{IRQ}}$ remains high
0	1	1	↑ Active	Set high on ↑ of CA2 (CB2)	Goes low when the interrupt flag bit CRA-6 (CRB-6) goes high

Notes: 1. ↑ indicates positive transition (low to high)
2. ↓ indicates negative transition (high to low)
3. The interrupt flag bit CRA-6 is cleared by an MPU Read of the A Data Register, and CRB-6 is cleared by an MPU Read of the B Data Register.
4. If CRA-3 (CRB-3) is low when an interrupt occurs (interrupt disabled) and is later brought high, $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$) occurs after CRA-3 (CRB-3) is written to a "one"

Table B-5 Control of CA2 as an output. CRA-5 is HIGH.

CRA-5	CRA-4	CRA-3	CA2 Cleared	CA2 Set
1	0	0	Low on negative transition of E after an MPU Read "A" Data operation.	High when the interrupt flag bit CRA-7 is set by an active transition of the CA1 signal.
1	0	1	Low on negative transition of E after an MPU Read "A" Data operation.	High on the negative edge of the first "E" pulse which occurs during a deselect.
1	1	0	Low when CRA-3 goes low as a result of an MPU Write to Control Register "A".	Always low as long as CRA-3 is low. Will go high on an MPU Write to Control Register "A" that changes CRA-3 to "one".
1	1	1	Always high as long as CRA-3 is high. Will be cleared on an MPU Write to Control Register "A" that clears CRA-3 to a "zero".	High when CRA-3 goes high as a result of an MPU Write to Control Register "A".

Table B-6 Control of CB2 as an output. CRB-5 is HIGH.

CRB-5	CRB-4	CRB-3	CB2	
			Cleared	Set
1	0	0	Low on positive transition of the first E pulse following and MPU Write "B" Data Register operation.	High when the interrupt flag bit CRB-7 is set by an active transition of the CB1 signal.
1	0	1	Low on the positive transition of the first E pulse after an MPU Write "B" Data Register operation.	High on the positive edge of the first "E" pulse following an "E" pulse which occured while the part was deselected.
1	1	0	Low when CRB-3 goes low as a result of an MPU Write in Control Register "B".	Always low as long as CRB-3 is low. Will go high on an MPU Write in Control Register "B" that changes CRB-3 to "one".
1	1	1	Always high as long as CRB-3 is high. Will be cleared on an MPU Write Control Register "B" results in clearing CRB-3 to "zero".	High when CRB-3 goes high as a result of an MPU Write into Control Register "B".

C ASCII character set

LSD \ MSD		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	¦
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL